Fifty Thesis

Aman Pathak

Associate Researcher
EnineHQ Technologies PVT LTD

May 2026

Preface

This book began as a collection of fifty research questions; fifty answers I wrote while shaping my identity as a researcher. Each chapter traces a possible path: a problem, a proposed approach, a plan for evaluation, and the open questions that would define a PhD. Together they map the terrain of computer science as I see it: from cache coherence to quantum algorithms, from operating systems to weighted automata.

If I am selected into the PhD program at BITS Pilani, this book becomes my life. Not metaphorically. These fifty proposals are the raw material from which my thesis will be carved. Some chapters will grow into full projects, others will merge or fall away, but the whole represents what I want to spend years of my life pursuing. This is one of the most redefining texts I have ever written, because writing it forced me to decide what problems matter to me and whether I have something to say about them.

The book is structured in eight parts, each covering a domain of computer science. The chapters within each part are meant to be read independently. Every chapter ends with research directions: concrete, publishable problems that extend the proposal. For a new PhD student, these are starting points. For an established researcher, they are invitations to collaborate.

I hope this book serves not just as a record of questions asked, but as a testament to the joy of asking them.

Aman Pathak
May 2026

Acknowledgements

First, myself: for the discipline to sit down and write fifty coherent proposals, for the curiosity that drove each question, and for the stubbornness to see this through.

Jyotiprakash Mishra: my professor, friend, and buddy. You taught me that a good research question is worth more than a good answer, and that the best collaborations happen when the line between mentor and friend disappears. Every page in this book carries your influence, whether in the precision of a claim or the courage to ask a bigger question.

And Bittu, my dog. You have been gone for more than two years, but I still feel the weight of your head on my foot while I write. The room is quieter now, and emptier. Some nights I still look down expecting to see you there. I hope wherever you are, you are running in the sun, and that you are waiting for me. I will see you again.

Contents

Part I

Computer Architecture

Chapter 1

Formally Verified Cache Coherence for Arbitrary Topologies

Can we design a cache coherence protocol that is *compositionally* verified for arbitrary network topologies (mesh, torus, fat-tree, arbitrary chiplet layouts) yet matches the performance of today's hand-optimized directory protocols?

Cache coherence protocols rank among the most intricate and error-prone components in modern processor design. Every commercial design MSI, MESI, MOESI, and their variants is validated through simulation and testing, with formal verification applied post-hoc if at all. The problem compounds when the topology changes: adding a chiplet, switching from mesh to torus, deploying a fat-tree for directory routing each change forces a complete re-verification effort. No existing framework lets a designer specify a protocol once and instantiate it on an arbitrary topology with a machine-checkable proof.

We propose to build a *protocol generator*: a tool that takes a high-level protocol specification and a topology description, then emits both a coherence protocol implementation and a formal proof of its correctness on that topology. The key insight is that coherence correctness decomposes into local node invariants (which are topology-independent) and a topology contract (routing deadlock freedom, bounded message latency). If every network node satisfies its local invariant and the topology satisfies its contract, global coherence follows compositionally.

Definition 1.1 (Cache Coherence Protocol)**.** A distributed state machine executed by cache controllers and directory home agents. Each node maintains a permission state per cache line, typically a subset of `Invalid`, `Shared`, `Exclusive`, `Owned`, or `Modified`, and transitions between states upon sending or receiving typed messages (`GetS`, `GetM`, `Put`, `Inv`, `InvAck`, `Data`). The protocol guarantees that every store is eventually visible to all subsequent loads and that no two nodes simultaneously hold conflicting write permissions.

Definition 1.2 (Transient State)**.** An intermediate cache-line state that exists between the initiation and completion of a coherence transaction. For example, `IM_A` denotes "waiting for data after an upgrade request," and `S_D` denotes "shared but with a downgrade in flight." Transient states are the primary source of complexity in real protocols like `MOESIF`, where speculative loads and write-back buffers introduce dozens of such interleaved states.

Theorem 1.1 (Compositional Coherence)**.** *For any network N where every node satisfies its local coherence invariant and the routing layer satisfies the topology contract, i.e., deadlock-freedom and bounded message delivery, the system is globally coherent: every store is eventually visible to all loads, and no two nodes simultaneously hold conflicting permissions. The proof reduces the N-node verification problem to $N + 1$ independent sub-proofs.*

This approach sits at an interesting point in the design space. Tools like BedRock offer microcode-programmable directory protocols but verify each topology instance separately. Culsans delivers a lightweight MOESI implementation for RISC-V clusters but applies verification post-hoc. Parametrized verification frameworks (Chou, McMillan, Pnueli) have explored N-processor coherence theoretically but stop short of integration with practical protocol generation. The table below summarizes the landscape.

1.1 Related Work

Approach	Strength	Gap
BedRock protocol (Ferdman et al.)	Microcode-programmable, flexible directory protocols	Verified for specific topologies only; no genericity proof
Culsans (RISC-V snoop-based MOESI)	Lightweight, evaluated on CVA6 cluster	Verification done post-hoc, not compositional
riscv-formal (SymbiYosys)	Bounded model checking for RTL	Bounded; does not prove correctness for arbitrary N
Sail RISC-V + ArchSem	Formal ISA + concurrency in Rocq	ISA-level; does not cover coherence protocol design
Parametrized verification (Chou et al., McMillan, Pnueli)	Theoretical frameworks for N-processor coherence	Not integrated with practical protocol generation; limited topology support
Protocol compilers (OpenSMART, Garnet)	Network-on-chip design tools	Focus on routing, not coherence protocol logic
Cuckoo directory	Practical hash-table based directory	No formal verification at all

1.2 Proposed Approach

We begin by defining a core calculus of coherence actions with well-defined interfaces. Each node (cache, directory, home agent) becomes a state machine that communicates by sending and receiving typed messages with bounded lifetimes. Transient states, the primary source of verification complexity in real protocols, are classified and numbered explicitly. Building on the Illinois protocol model and BedRock's microcode approach, we add formal interface contracts designed for compositional reasoning.

Architectural Insight

By separating topology-independent local invariants from topology-specific routing contracts, the verification of an N-node coherence protocol on an arbitrary topology decomposes into N per-node invariant checks plus a single topology contract check. This compositional structure means that adding a new chiplet or switching from a mesh to a torus requires re-verifying only the topology contract, not the entire protocol.

The centerpiece is a meta-theorem: if every node in a network satisfies its local invariant, and the network topology satisfies a separate *topology contract* (no deadlock in routing, bounded message latency), then global coherence follows. The key research question is identifying the minimal set of local invariants that make this compositionality hold a "coherence invariant kernel" that is topology-independent, plus a small set of topology-specific refinements. For a cache node, the local invariant includes at most one outstanding transaction per line, write permission implying exclusive ownership, bounded transient state durations, and the absence of message cycles involving transient states.

We then implement a protocol generator in Rocq (or Coq) that takes a protocol specification in the calculus and a topology graph, generates the node state machines and network wrapper, and discharges the verification conditions automatically using a library of proven lemmas. The generated RTL is exported via Kôika or Chisel for synthesis. Our target is a verified MOESI directory protocol for 2D mesh, torus, and a 4-chiplet topology.

The hardest part is invariant discovery: automated generation of inductive invariants for infinite-state (N-processor) systems is undecidable in general. We rely on the calculus restricting the design space enough to make this tractable. If it fails, the thesis shifts toward a "proof automation toolkit" that assists human verifiers rather than fully automating proofs. A related concern is transient state explosion: real protocols like MOESIF with speculative loads and write-back buffers have dozens of transient states, each multiplying verification effort. The calculus must provide abstraction mechanisms that hide this complexity from the composition theorem.

1.3 Evaluation

Benchmark	**What it tests**
SPEC CPU2017 (subset)	General-purpose performance
PARSEC	Parallel workloads with coherence traffic
Stream	Bandwidth-bound, coherence-light
Graph500	Irregular coherence patterns
Custom coherence microbenchmarks	Stress specific protocol states (false sharing, migratory sharing, producer-consumer)

We compare against hand-optimized Culsans (MOESI snoopy for CVA6), BedRock microcode protocols, and the baseline directory protocol from Chipyard. Metrics include area (LUTs, registers), latency (cache miss cycles), bandwidth (messages per miss), clock frequency impact, and verification time measured in human-months versus automated hours. The central question is the "verification tax": how much performance is lost by constraining the protocol to be compositionally verifiable.

If the gap exceeds 10%, the thesis must find optimizations that preserve verifiability or characterize the sources of overhead to guide future protocol designs.

If the approach succeeds, the framework extends naturally to heterogeneous topologies (mixing private L1 caches, shared L2 slices, and directory home agents in the same network), power-aware verification (bounding energy per transaction), and security-aware coherence (guaranteeing that directory invalidation timing is independent of address to eliminate coherence-based side channels).

1.4 Research Directions

Automated invariant synthesis for N-processor coherence. The undecidability of inductive invariant generation for infinite-state systems remains the primary barrier to full automation. A promising direction is to restrict the coherence calculus to a fragment where invariants are expressible as finite unions of regular languages over message-sequence charts, reducing invariant synthesis to automata learning (e.g., L^* algorithm adapted to message-passing protocols). A successful result would yield a push-button tool that takes a protocol skeleton and outputs a verified implementation for any N, publishable at ISCA or ASPLOS.

Topology-aware proof amortization via graph minors. Rather than proving coherence separately for each topology, one could show that if a protocol is verified on a set of "basis" topologies (say, ring and mesh), then its correctness on any topology that is a minor of those bases follows by a proof-preserving graph embedding theorem. This would give a polynomial-time decision procedure for topology inheritance, directly addressing the chiplet composition problem. Suitable for DATE or DAC with a strong formal methods track.

Transient-state abstraction with dependent types. Real protocols (MOESIF with speculative loads) have dozens of transient states whose pairwise interactions explode verification complexity. Using dependent types to encode transient states as a refinement of the stable-state machine, one could prove that transient behavior is a finite perturbation around stable semantics, drastically reducing the invariant surface area. The intellectual contribution is a generic transient-state compiler that erases to a standard protocol for performance but retains full proof for verification. Target venue: PLDI or POPL.

Performance-verified protocol synthesis with bounded slack. The central thesis question of the "verification tax" can be inverted: instead of asking how much performance is lost, ask for the minimal performance gap achievable while maintaining verifiability, parameterized by an ϵ-bounded slack on miss latency. An optimization framework over the protocol parameter space (number of transient states, directory associativity, invalidation strategy) would characterize the Pareto frontier of verification difficulty vs. performance, giving architects a systematic design-space exploration tool. Publishable at MICRO or HPCA.

Chapter 2

Provably Bounded Timing Channels in Microarchitecture

What is the minimal hardware support required to prove that a processor core's timing behavior is independent of secret values, even under speculation?

Every modern processor leaks secret data through timing channels. Speculative execution (Spectre, Meltdown), cache timing (Prime+Probe, Flush+Reload), prefetcher-based leaks (DMP attacks), and resource contention all create data-dependent timing variations. The mitigations (SPLASH, Triosecuris, speculative load hardening, constant-time programming) arrive after each new attack class and never provide a blanket guarantee. The core problem is reactive: each new Spectre variant triggers a new fix, and the cat-and-mouse game continues.

We propose a processor where timing is provably independent of secret values by construction. This requires co-design across three layers: an ISA specification with explicit timing semantics, a microarchitecture that respects those semantics, and a compiler that generates code exploiting them. The goal is a concrete guarantee: if the compiler says the binary is constant-time, the microarchitecture cannot leak by proof, not by mitigation. The significance for certification is substantial Common Criteria and FIPS 140-3 both require evidence of side-channel resistance, and a formal proof is the strongest possible evidence. In practice, cloud providers currently deploy mitigations costing 5–30% performance; a provably secure core could eliminate them while providing stronger guarantees.

Definition 2.1 (Constant-Time Execution)**.** An execution of a program is *constant-time* if the processor's latency depends only on publicly-observable inputs (instruction opcode, address, architectural state) and not on secret values (cryptographic keys, confidential data). This property must hold across all microarchitectural features, i.e., caches, speculation, prefetchers, and functional units, so that a remote observer measuring execution time cannot infer secret data.

Definition 2.2 (Speculation Tracking)**.** A hardware mechanism that records which register values and memory locations are the result of speculative execution. Each microarchitectural state element carries a `spec_valid` bit; when a speculated value propagates through functional units or feeds into address generation, the resulting outputs are similarly marked. Speculation tracking is the foundation for preventing speculative side-channel leakage.

Theorem 2.1 (Timing Independence)**.** *A processor core whose caches are constant-time, speculation is tracked via* `spec_valid` *shadow structures, and functional units have operand-independent latency satisfies: (1) every instruction's latency falls within the ISA-specified interval* $[L_{\min}(i), L_{\max}(i)]$*; (2) the latency depends only on non-secret state; and (3) speculation never exposes secret data to timing-relevant microarchitectural state.*

2.1 Related Work

Approach	Strength	Gap
SPLASH (Reconfigurable speculation table + SIFT, 0.05–1.23% overhead)	Low overhead, covers common Spectre variants	Not formally verified; may miss new variants
Triosecuris (Formally verified CET+SLH)	Formal proof for BTB/RSB/PHT	Covers Spectre v1/v2 only; high overhead
SpecLFB (Line-fill buffer defense)	Hardware approach, practical	No formal guarantee; limited to LFB channels
Janus (ARM PA+BTI, 3.85% overhead)	Low overhead on ARM	Software-only; microarchitecture can still leak
Speculative Load Hardening (LLVM pass)	Compiler-based, deployable	Soundness not formally proven; coverage gap
ClepsydraCache (TTL + index randomization)	Constant-time cache behavior	Statistical guarantee only; no speculation coverage
Constant-time programming (Jasmin, CompCert-CT)	Verified constant-time software	Assumes microarchitecture is constant-timewhich it is not
SC proofs for in-order cores (Kôika + Coq)	Full-stack proof for simple cores	Not applicable to out-of-order, speculative cores

No existing work provides an end-to-end proof that speculation, caches, and prefetchers collectively produce no timing leakage only point mitigations for known attack surfaces.

2.2 Proposed Approach

We begin by extending an existing formal ISA specification (Sail RISC-V) with timing annotations. Each instruction gets a latency interval [min, max] cycles, where the range arises only from non-secret-dependent conditions. Cache hit-versus-miss is secret-dependent and therefore forbidden; instead, the specification says the instruction takes max cycles always, or uses a constant-time cache. We also introduce secrecy labels that mark which architectural and microarchitectural values are secret versus public; the specification defines which of these may influence instruction timing. This timing-aware ISA specification is a novel contribution, since existing formal ISAs capture functional behavior only.

Architectural Insight

Provable timing security requires co-design across three layers: the ISA explicitly specifies timing intervals per instruction, the microarchitecture is verified to respect those intervals, and the compiler generates code that exploits them. The chain of proof connects all three: if the compiler proves the binary is constant-time at the ISA level and the microarchitecture executes the ISA constant-time, then the whole system is constant-time by construction.

Given this specification, we design a microarchitecture (based on CVA6 or BOOM) with three timing invariants. First, caches are constant-time: every access takes fixed latency regardless of hit or miss, using a design like ClepsydraCache-style TTL with full randomization, or fully associative caches where access time is independent of hit. The key constraint is that latency depends on address (public) and not on data (secret). Second, speculation is tracked via a shadow structure that records which register values are speculative and prevents secret values from guiding speculation, extending SPLASH's SIFT tracking to cover all speculation sources. Third, functional units are constant-time: multipliers, dividers, and other variable-latency units are either padded to worst-case or proven to have operand-independent latency.

The critical design principle is that the microarchitecture must be inherently timing-safe for all values, not just those tagged secret. Performance optimization can exploit secrecy labels where available, but correctness never depends on knowing which values are secret. We prove three properties using Kôika (a Chisel-like HDL with Coq semantics): every instruction's latency falls within the ISA-specified interval (compliance), the latency depends only on non-secret state (security), and speculation never exposes secret data to timing-relevant microarchitectural state (speculation safety). The proof uses a simulation relation between the timing-aware ISA specification and the Kôika implementation.

The compiler side extends CompCert (or a RISC-V backend) to propagate secrecy labels from source to machine code, generate constant-time instruction sequences for secret-dependent branches (CMOV-style, bit masking), and insert speculation barriers only where the label requires them. The compiler proof connects to the microarchitecture proof: if the compiler proves the binary is constant-time at ISA level, and the microarchitecture is verified to execute the ISA constant-time, the whole system is constant-time.

The hardest challenge is that constant-time caches are expensive fully associative designs with hit-independent latency have poor energy and area scaling, and randomization approaches give statistical rather than deterministic guarantees. A second challenge is the performance cliff for variable-latency operations like integer division and cryptography, which could suffer 10–100$\times$ slowdown when padded to worst case. The solution may be to restrict expensive operations to non-secret code. Finally, the proof complexity for an out-of-order core like BOOM is enormous the thesis must carefully scope the proof, perhaps proving only the tracking mechanism correct while relying on the microarchitecture design to ensure that tracking prevents leakage.

2.3 Evaluation

Benchmark	What it tests
SPEC CPU2017	General-purpose integer and float performance
CoreMark	Embedded/microcontroller performance
MiBench	Embedded benchmarks with crypto workloads
OpenSSL/BoringSSL	Real-world constant-time code
Security microbenchmarks	Measured timing leakage via TVLA on FPGA

We tape out (or FPGA implement) the designed core and measure performance overhead versus an unmodified CVA6/BOOM baseline on SPEC CPU2017, along with power and area overhead. Security validation runs known Spectre v1–v5, MDS, cache timing, and DMP attack gadgets on compiled binaries and attempts to distinguish secret values via timing on the FPGA implementation. The central question is the performance-security Pareto frontier: where do diminishing returns kick in when relaxing constant-time guarantees? Extensions to multi-core constant-time (where shared caches introduce contention-based timing channels even with individually constant-time cores) and power-side-channel resistance would broaden the framework, but the core thesis focuses on a single-core timing-proof design.

2.4 Research Directions

Timing-aware ISA specification with formal latency contracts. Extending Sail RISC-V with timing annotations yields a contract between hardware and software: the ISA specifies a latency interval $[L_{\min}(i), L_{\max}(i)]$ per instruction i, and the microarchitecture must respect it. A compiler that targets this contract can then prove constant-time behavior at the ISA level without knowing the microarchitecture. The core research problem is whether these intervals can be made tight enough for useful performance while still admitting a proof that the microarchitecture meets them. This would be a fundamental contribution to the architecture-software interface, publishable at ISCA or MICRO.

Security-performance Pareto frontier characterization via relaxation lattices. Not all timing channels are equally dangerous; a program handling cryptographic keys requires full constant-time guarantees, but a program processing non-critical data can tolerate bounded leakage. Formalizing a lattice of relaxation strategies (e.g., allowing cache hits to be faster than misses but bounding the ratio $t_{\text{miss}}/t_{\text{hit}} \leq \alpha$) and proving per-relaxation security properties would let architects choose the cheapest implementation that meets the application's security requirements. The main challenge is proving that bounded-leakage relaxations compose safely. Target venue: S&P or USENIX Security.

Verification of constant-time caches with bounded statistical leakage. Fully associative, hit-independent-latency caches are area-prohibitive. Randomized designs (ClepsydraCache) achieve constant-time behavior statistically: the probability that two addresses map to the same cache set within a time window is bounded. Formally proving a differential-privacy-style guarantee, $|\Pr[T(a) = t] - \Pr[T(a') = t]| \leq \epsilon$ for any two addresses a, a', would provide a rigorous foundation for statistical constant-time caches. The proof would combine cache timing models with concentration inequalities, publishable at CRYPTO or CHES.

Compositional constant-time for multi-core systems. Proving single-core timing independence is insufficient: shared caches and interconnects introduce contention-based timing channels even between constant-time cores. A compositional theory where each core's timing contract is a function of its own private state only, and the interconnect enforces that shared-resource contention is bounded and secret-independent, would extend the framework to multi-core. The key technical hurdle is proving that a "fair arbitration" property (e.g., round-robin with constant per-request latency) suffices for multi-core constant-time composition. Publishable at ASPLOS or HPCA.

Chapter 3

Unified Formal Framework for Cross-ISA Memory Models

Can we construct a single algebraic framework that captures TSO (x86), ARM-A's relaxed model, and RVWMO as parameterized instantiations of a single core theory, enabling cross-ISA verified compilation and unified program reasoning?

Every major ISA defines its own memory consistency model: x86-TSO with store buffers, ARM-A's relaxed model with multiple address domains and barriers, RISC-V RVWMO with acquire/release semantics, and IBM POWER's extremely relaxed model with store forwarding and cumulativity. Each has its own formalization (operational, axiomatic, or both), its own verification tools, and its own reasoning principles. Cross-ISA compilation currently requires bespoke proofs for each pair of source and target memory models, making verified portability across architectures expensive and error-prone.

We propose a category-theoretic unification: a category of memory models where each model is an object, refinement maps between models are morphisms, and verified compilers are functors. A compiler proven correct as a functor between model categories automatically preserves semantics across targets: proving it once suffices for all ISAs in the category. Beyond compilation, the framework would let us prove generic theorems once (e.g., "data-race freedom implies sequential consistency holds for all models in this class") and derive per-ISA theorems as special cases.

Definition 3.1. A *memory model category* $\mathcal{M}$ has memory states as objects, programs (sequences of events) as morphisms, and parallel composition as the monoidal product. A memory model is a functor $F : \mathcal{M} \to \mathbf{Rel}$ to a category of happens-before relations. A refinement map from model M_1 to model M_2 is a natural transformation $\eta : F_1 \Rightarrow F_2$ that is a monotone map on executions.

3.1 Related Work

Approach	Strength	Gap
ArchSem (Rocq framework for Arm-A + RISC-V)	Unified ISA semantics + concurrency	Covers only Arm-A and RISC-V; no category theory
AxSL (Separation logic for Arm-A)	Program reasoning for relaxed models	ISA-specific; not cross-ISA
SLR / Iris-based (Weakened program logics)	General reasoning principles	Not unified across ISAs
Alglave et al. (Axiomatic MCMs: cat files)	Hardware-sourced axiomatic models for all major ISAs	Descriptive only; no algebraic structure
Operational models (x86-TSO, ARM self-modifying)	Executable, testable	Each model built independently; no common core
Owens et al. (2009) (Algebraic MCM hierarchy)	Maps between TSO, SC, PSO as Galois connections	Does not cover ARM/POWER/RISC-V relaxed models
C11 memory model (Boehm, Batty, et al.)	Language-level MCM, formalized in Coq	Language-level; mapping to hardware is complex

No existing work provides a category-theoretic unification that captures current hardware models and supports cross-model verified compilation.

3.2 Proposed Approach

We begin by identifying the minimal set of primitives every hardware memory model must define: events (read, write, read-modify-write, fence), relations (program order, reads-from, coherence order, happens-before, observation), and axioms specifying which relations exist and which cycles are forbidden. We express this as a monoidal category: objects are memory states, morphisms are programs (sequences of events), and the monoidal product is parallel composition. A memory model is a functor from this category to a relation category where morphisms are happens-before relations. The key innovation is representing each model as a pair (coherence model, consistency model), formally separating two concerns that are traditionally entangled.

We then construct objects in the category for SC (all events totally ordered), x86-TSO (SC plus a store buffer abstract machine), ARM-A (SC for ordered accesses, a weak object for normal accesses, mediated by barrier types), and RISC-V RVWMO (SC with acquire/release annotations and atomics). For each model we build both an operational object (transition system) and an axiomatic object (constraint on happens-before) and prove they are equivalent within the category.

The central technical contribution is constructing refinement maps (morphisms) between models: SC $\to$ x86-TSO, SC $\to$ ARM-A, x86-TSO $\to$ ARM-A (via composition), RVWMO $\to$ ARM-A, and C11 $\to$ x86-TSO (the critical map for verified compilation). Each morphism is a monotone map on executions: given an execution in the source model, produce an execution in the target model that is indistinguishable to the programmer.

Theorem 3.1 (Refinement Map Existence)**.** *For any pair of memory models M_1, M_2 in the category $\mathcal{M}$ where M_2 is at most as relaxed as M_1 (i.e., every M_2 execution is also an M_1 execution), there exists a refinement morphism $\eta_{M_1 \to M_2} : F_1 \Rightarrow F_2$. In particular, refinement maps exist for SC → x86-TSO, SC → ARM-A, x86-TSO → ARM-A, RVWMO → ARM-A, and for a functorial subset of C11 → x86-TSO.*

Proof sketch. For each pair, construct the map by projecting the source execution onto the target's event types and showing that the target's happens-before axioms are satisfied. For SC → x86-TSO, the projection preserves the total order on memory operations; store buffer behaviors in x86-TSO are encoded as additional events invisible to the projection. For ARM-A targets, the proof uses a simulation between the operational machine models, where each SC step simulates a sequence of ARM-A micro-operations. The C11 → x86-TSO map requires restricting to the DRF-SC fragment of C11. □

The existence of these maps is a theorem that must be proved.

Given these morphisms, we express verified compilation as functoriality: a compilation pass from source language S to target language T induces a functor from the source memory model category to the target category. If the compiler is a functor, compositionality of verification is automatic. We demonstrate this with a case study: a lock-free concurrent data structure verified for C11 under the DRF-SC guarantee, compiled to x86, ARM, and RISC-V targets, where the proof of correctness on each target is a single application of the functor.

The hardest challenge is bridging the gap between operational and axiomatic representations: some ISAs (x86-TSO) are naturally operational, while others (ARM-A) are naturally axiomatic. The category must support both and prove them equivalent, which is itself a major undertaking. A second challenge is that the C11-to-hardware mapping is not fully functorial; features like consume semantics and non-atomics do not map cleanly to any hardware model. The thesis must either restrict to a functorial subset of C11 or extend the category to handle these features.

Core Thesis

A category-theoretic unification of hardware memory models reduces cross-ISA verified compilation from $O(n^2)$ bespoke per-pair proofs to $O(n)$ morphism constructions, where each new ISA requires only defining its object in the category and connecting it via refinement morphisms to existing models. The key technical enabler is representing each model as a pair (coherence model, consistency model), separating two traditionally entangled concerns.

3.3 Evaluation

Criteria	Method
Correctness	All axioms of each model are preserved by the category representation
Completeness	Every execution allowed by the model has a representation in the category
Cross-ISA proof reduction	Lines of proof code for cross-ISA verification (ours vs. separate proofs)
Extensibility	Lines of definition needed to add a new ISA model as a new object
Computational adequacy	Can the framework automatically generate litmus tests from the category structure?

The extensions are natural: persistent memory models (adding durability order as a new relation), GPU memory models (NVIDIA PTX, AMD ROCm), and language-level models (C11, Java, Rust) can be added as new objects with morphisms connecting them to existing hardware models. An automated litmus test generator that derives distinguishing tests from the category structure would provide a practical tool for architecture verification.

3.4 Research Directions

Functorial compilation of C11 atomics to heterogeneous targets. The C11 memory model's release/acquire and consume semantics do not map cleanly to any single hardware model. A category-theoretic approach would represent C11 as a 2-category where the 2-morphisms capture allowed reorderings, and a compilation functor to the x86-TSO category must respect this 2-categorical structure. Proving that a subset of C11 atomics (e.g., release/acquire only, no consume) forms a functor to every object in the hardware category would immediately yield verified cross-ISA compilation for that subset. The contribution is a general proof technique that replaces $O(n^2)$ per-pair proofs with $O(n)$ morphism constructions. Publishable at POPL or PLDI.

Algebraic characterization of GPU memory models (PTX, ROCm) as category objects. GPU memory models introduce additional complexity: wavefront-scoped synchronization, cross-wavefront barriers, and heterogeneous memory spaces. Modeling the NVIDIA PTX memory model as a monoidal category with a comonoid structure for divergent threads would unify GPU and CPU reasoning under the same algebraic umbrella. The research challenge is whether the monoidal-category framework is expressive enough to capture cross-lane communication without collapsing to full sequential consistency. Target venue: MICRO or PACT.

Automated litmus test generation from category structure. Given two memory model objects M_1 and M_2, the category structure defines a partial order of refinements via morphisms. A litmus test that distinguishes M_1 from M_2 exists iff there is no morphism in either direction. Automating the search for such distinguishing executions reduces to constraint solving over the category's commutative diagrams; a SAT-based litmus test generator would make the framework practical for architecture validation. The technical advance is a decision procedure for morphism existence in this specific category. Publishable at CAV or TACAS.

Persistent memory consistency as a categorical extension. Adding durability order as a new relation extends each model object M to $P(M)$ with an additional persistence morphism. The category structure would reveal which persistent models are refinements of which, and whether the canonical persistent extension theorem (chapter Q5) can be expressed as a monad over the memory model category. This cross-chapter connection would be a strong systems result, publishable at SOSP or OSDI.

Chapter 4

Chiplet Cache Coherence Under Non-Uniform Latency

Can we design a coherence protocol whose correctness is independent of latency (verified under arbitrary message delays) but whose performance model explicitly accounts for die topology: dynamically migrating directory home agents, replicating directory state at die boundaries, or using predictive multicast to avoid indirection?

Traditional cache coherence protocols assume a uniform-latency network where every cache-to-directory request has roughly the same round-trip time. Chiplet architectures (UCIe, BoW, EMIB) break this assumption in three ways: die-crossing latency is 2–5× that of same-die access, die-to-die links are shared resources with contention-dependent latency, and latency is asymmetric between directions. Existing directory-based MOESI and snooping-based MESI protocols are designed and verified under uniform-latency assumptions. When deployed on chiplets, they either degrade performance (directory indirection always pays the cross-die penalty) or require ad-hoc modifications that void existing correctness proofs.

We propose a coherence protocol designed from the ground up for non-uniform latency. The protocol is verified correct under *any* message latency (including infinite delay (node failure) using an asynchronous communication model where message delivery is guaranteed but latency is unbounded. This is standard in distributed systems verification but novel for cache coherence. On top of this latency-independent correctness layer, we add performance optimizations that explicitly account for chiplet topology: dynamic migration of directory home agents toward active sharers, replication of read-only directory state at die boundaries, and predictive multicast to avoid remote directory lookups.

Definition 4.1 (Chiplet Architecture)**.** A processor composed of multiple silicon dies (chiplets) interconnected by physical links such as `UCIe`, `BoW`, or `EMIB`. Die-crossing latency is typically 2–5× that of same-die access, die-to-die links are shared resources with contention-dependent latency, and latency is asymmetric between directions. These properties break the uniform-latency assumption of traditional coherence protocols.

Definition 4.2 (Directory Home Agent Migration)**.** A protocol mechanism that dynamically moves directory state for a cache line from one die's home agent to another as the set of active sharers shifts. During migration, concurrent requests to both old and new homes must be handled with at most one authoritative home at any instant to prevent split-brain. The protocol uses epoch-based leases where the old home's authority expires after a bounded time window.

Theorem 4.1 (Asynchronous Coherence Correctness)**.** *If every node satisfies its local transition rules and the network guarantees finite (but unbounded) message delivery, then the system is coherent for any message latency: every store is eventually visible to all loads, and no two caches simultaneously hold conflicting permissions. Correctness does not depend on timeouts, message ordering, or bounded latency.*

4.1 Related Work

Approach	Strength	Gap
Culsans (MOESI snoopy for CVA6 cluster)	Lightweight, evaluated on real RTL	Single-die; no chiplet extension
BedRock (Microcode-programmable coherence)	Flexible protocol specification	Performance model assumes uniform latency
DiFache (Decentralized, snoopy-directory hybrid)	Scalable invalidation, topology-aware	Not formally verified; ad-hoc performance model
Arm CHI across chiplets (gem5 modeling)	Industry-standard approach	Modeling only; no formal verification
Cuckoo directory	Directory scalability via hashing	Single-die; no chiplet latency awareness
CXLalloc (Shared allocator for CXL pods)	Coherence management for CXL memory	Allocator-level, not protocol-level

No existing work combines formal correctness under arbitrary latency with topology-aware performance optimization for chiplet architectures.

4.2 Proposed Approach

We design a directory-based MOESI protocol variant where all directory transitions are labeled by their effect rather than their timing. A "getS" message means "request shared copy" and the protocol must handle the response arriving at any future time. Timeouts are used only for performance, never for correctness if a response does not arrive within an expected window, the protocol can retry or escalate, but correctness does not depend on the retry succeeding.

Architectural Insight

By designing the coherence protocol under an asynchronous communication model where all transitions are labeled by effect rather than timing, correctness becomes independent of latency. Performance optimizations (home migration, data replication, predictive multicast) are then layered on top without affecting the proof, ensuring that even if an optimization fails or suffers unexpected delays, coherence is never violated.

The protocol is structured as a distributed transition system where each node (cache, directory, home agent) independently follows local rules, and correctness is a global invariant on permissions and data across all copies of a cache line.

For performance, we add a dynamic home migration protocol that moves directory entries between dies as the sharer set changes. When most sharers of a cache line move to a different die, the directory should follow. This is analogous to distributed hash table rebalancing but with stronger consistency requirements: during migration, concurrent requests to both the old and new home must be handled correctly, with at most one home authoritative at any time to prevent split-brain. We use a lease-based approach where the old home's authority expires after a bounded time and the new home takes over.

For read-only cache lines shared across multiple dies, we use a predictive multicast strategy. The baseline approach (every read miss goes to the directory home) is simple but pays cross-die latency. The alternative (replicate data to all dies on first read) avoids indirection but requires broadcast on invalidation. Our predictive approach learns from access pattern history whether a line will be read mostly by one die (replicate locally) or shared across dies (keep a single source of truth). Correctness is independent of prediction accuracy; prediction only affects performance.

The compositional verification framework defines a chiplet coherence interface: each die exports the messages it can send and receive, guarantees about message ordering, and guarantees about outstanding transactions per cache line. We prove a meta-theorem: if every chiplet satisfies its local interface contract and the network satisfies a topology contract (no deadlock in die-to-die routing, finite message delivery), then the multi-chiplet system is coherent. Adding a new chiplet requires re-verifying only the boundary conditions, not the entire protocol.

The hardest challenge is that asynchronous verification is fundamentally harder than the standard coherence proofs, which rely on message ordering guarantees (e.g., requests to the same directory arrive in order). We must carefully identify which ordering guarantees are essential and which can be relaxed. The migration protocol is especially risky if a directory home migration goes wrong (split-brain, lost state), the result is system crash. The migration must be provably atomic and fault-tolerant, potentially requiring a consensus protocol within the coherence layer.

4.3 Evaluation

Benchmark	**What it tests**
SPEC CPU2017	General-purpose performance across chiplets
PARSEC	Parallel workloads with cross-die sharing
Graph500	Irregular access patterns, cross-die traffic
Stream	Bandwidth-bound, tests replication efficiency
Custom microbenchmarks	False sharing across dies, producer-consumer, migratory sharing

We implement on a simulated chiplet RISC-V platform (gem5 with chiplet extensions, or FPGA with multiple dies) in a 4-chiplet configuration with UCIe-style die-to-die links. Each chiplet has a CVA6 cluster with private L1 and shared L2, and the directory is distributed across L2 banks. We compare against an unmodified directory protocol (uniform latency assumption), a snooping protocol (broadcast-based), an ideal die-local-only upper bound, and an oracle optimal home placement. Metrics include average miss latency, coherence traffic, die-to-die bandwidth utilization, directory migration frequency, and verification effort.

4.4 Research Directions

Asynchronous coherence verification with automated invariant inference. Standard coherence proofs rely on message ordering guarantees (e.g., FIFO channels between nodes). Relaxing this to asynchronous delivery (the distributed systems model) removes the ordering assumption but forces a fundamentally different proof technique based on simulation relations over partially ordered message histories. The technical challenge is automating the construction of such simulations for arbitrary directory protocols, reducing it to a finite-state model checking problem on the protocol's message-type graph. A push-button tool that takes a protocol specification and outputs an asynchronous-correctness proof would be publishable at CONCUR or FMCAD.

Learning-guided directory migration with formal convergence bounds. The dynamic home migration problem resembles distributed hash table rebalancing but with stronger consistency requirements. Using reinforcement learning to predict the optimal home placement, while simultaneously proving that any learned policy converges within $O(\log n)$ migrations per line in the worst case, would give a formally grounded adaptive protocol. The key insight is that migration decisions can be modeled as a Markov decision process where the state is the sharer set and actions are home migrations, with a cost function combining latency and migration overhead. Target venue: MICRO or ASPLOS.

Predictive multicast with learned access patterns and PAC guarantees. Replacing rule-based prediction with a learned model (e.g., a small LSTM or decision tree per cache line) for whether to replicate data to a given die could improve replication accuracy. The research question is whether Probably Approximately Correct (PAC) learning bounds can be proven for the coherence setting: given m observed accesses to a line, the prediction error at the next access is bounded by ϵ with probability $1 - \delta$, and this error bound translates directly to an expected latency overhead bound. The formal link between PAC learning and coherence performance is novel, publishable at ISCA.

Consensus-free directory migration using epoch-based leases. The split-brain problem during directory migration currently suggests a consensus protocol (e.g., Paxos within the coherence layer), which is too slow for cache-latency-sensitive operations. An epoch-based lease protocol where each home holds authority for a bounded time window and migration only occurs at epoch boundaries would eliminate the need for consensus: as long as all nodes agree on the current epoch number and the epoch duration is long enough that in-flight transactions drain, migration is provably atomic. The central proof would show that two homes cannot be authoritative for the same line during overlapping epochs. Publishable at SPAA or DISC.

Chapter 5

Optimal Memory Consistency for Persistent Memory

Can we define a principled persistent memory consistency model that generalizes TSO and SC, admits an operational specification suitable for hardware implementation, is compositional (so verified programs remain correct under composition), and admits an equivalent axiomatic specification for verification?

Persistent memory (NVDIMM, CXL-attached persistent memory) introduces a correctness criterion beyond traditional memory consistency: *crash consistency*. A program running on persistent memory must not only respect the consistency model during normal execution but also ensure that after a power failure or system crash, the persistent state reflects a prefix of the intended updates. Current approaches fall into two disconnected camps. Hardware-centric models like PEx86 extend x86-TSO with explicit flush and fence instructions for persistence, but the model is ad-hoc and different variants have been found inconsistent. Software-centric approaches (persistent transactional memory, logging) require programmer discipline with no formal integration with the hardware consistency model.

We propose a principled persistent memory consistency model derived from first principles rather than ad-hoc extension. The model generalizes existing volatile models (SC, TSO, RVWMO) as special cases, admits both an operational specification (for hardware implementation) and an equivalent axiomatic specification (for program verification), and is compositional a correctly written program remains correct when composed with other correct programs.

5.1 Related Work

Approach	Strength	Gap
PEx86 (Raad et al.)	First formal persistent model for x86-TSO	Only covers x86; ad-hoc extension of TSO
PSO / Persistent TSO (Khyzha, Lahav)	More principled, derived from TSO	Still TSO-specific; not general
Persistent C11 (Dongol et al.)	C11-level model with persistency	Language-level, not hardware; mapping unclear
BPFS (Condit et al.)	Short-circuit shadow paging	Filesystem-level; not a general consistency model
Intel CLWB/PCOMMIT	Hardware instructions for persistence	Instruction-level; no formal model of their semantics
PMem.io / PMDK	Software libraries for persistent programming	Provide patterns, not formal guarantees

No model is simultaneously principled (from first principles), general (covers SC, TSO, relaxed as special cases), operational (for hardware design), axiomatic (for software verification), and compositional.

5.2 Proposed Approach

We define an abstract machine with three memory tiers: volatile L1/L2 cache (standard MOESI coherence, loses data on crash), volatile store buffer (holds retired stores before propagation to the persistence domain), and persistent memory (survives crash, written only via flush instructions). The machine provides four operations: store (to store buffer), load (from cache or persistent memory), flush (ensuring data has reached the persistence domain), and fence (ordering stores and flushes). A crash at any point resets volatile state to zero, leaving persistent memory in an arbitrary state but the model constrains which states are legal given the operations issued.

The centerpiece is a canonical persistent extension theorem. We state it formally:

Theorem 5.1 (Canonical Persistent Extension)**.** *For any volatile memory model M expressible as a store-buffer machine (including SC, TSO, PSO, and RVWMO with buffered stores), there exists a unique persistent extension $P(M)$ such that:*

1. *$P(M)$ behaves identically to M under executions where no crash occurs;*
2. *for any execution prefix ending in a crash, the persistent state reflects a valid prefix of the intended update sequence;*
3. *$P(M)$ is maximally relaxed: it adds no ordering constraints beyond those necessary for persistence.*

Proof sketch. Construct $P(M)$ by extending M's operational machine with a persistence domain and flush instructions. Establish a simulation relation between configurations of M and $P(M)$ during crash-free executions. For crash prefixes, define a projection function from the persistence domain to the volatile domain and show that the persistent state always equals the image of some volatile prefix under this projection. Maximal relaxation follows from showing that any weaker ordering would allow a crash-inconsistent state reachable. □

This general construction avoids designing each persistent model from scratch.

We instantiate this for RISC-V with TSO (the current default, since most implementations use TSO-like store buffers), producing Persistent RVWMO. The operational machine gives each core a store buffer; stores exit to the coherence domain (same as TSO), and flushes move data from the coherence domain to the persistent memory domain. The axiomatic model defines six orders program order, reads-from, coherence order, happens-before, persist order (which writes become persistent before which others), and flush order with axioms combining standard TSO constraints and persistence constraints. We prove operational–axiomatic equivalence.

For program correctness, we define *persistent data-race freedom* (PDRF):

Definition 5.1 (Persistent Data-Race Freedom)**.** A program is PDRF if all shared persistent locations are accessed only within critical sections, each critical section ends with appropriate flush operations, and between critical sections no thread reads a persistent location that another thread might have written without flushing. A PDRF program satisfies the *persistent DRF-SC guarantee*: it executes as if sequentially consistent plus crash-atomic: each critical section either fully persists or fully does not.

We prove that PDRF programs execute as if sequentially consistent plus crash-atomic each critical section either fully persists or fully does not. This is the persistent analog of the classic DRF-SC theorem.

We implement a RISC-V PM controller (in Chisel/Kôika) that implements the operational model, accepting flush requests via RISC-V CMO extensions (Zicbom), maintaining a persistence queue of in-flight flushes, and providing ordering guarantees per the model. The design is evaluated in a gem5 simulation of persistent memory with NVDIMM timing.

The main challenge is that compositionality is not free: PDRF may be too restrictive for concurrent persistent data structures that do not use locks. The thesis may need to define a hierarchy of correctness conditions. A second challenge is that the operational–axiomatic equivalence proof for persistent models is complex, since the persist order interacts with the volatile order in subtle ways. Finally, the performance cost of strong persistence may require more flushes and fences than current ad-hoc approaches the thesis must quantify this cost and suggest relaxed modes if it proves too high.

Core Thesis

Any volatile memory model M expressible as a store-buffer machine admits a unique maximally relaxed persistent extension $P(M)$ that coincides with M in crash-free executions and guarantees prefix consistency upon crash. This construction generalizes persistent memory consistency from an ad-hoc per-ISA design problem to a parameterized family derived from first principles.

5.3 Evaluation

Criteria	Method
Model sanity	Litmus tests: generate all minimal persistent litmus tests and verify correct behavior
Compositionality	Express correct persistent programs, prove them PDRF, compose, and verify
Implementation cost	PM controller gate count, latency per flush, impact on non-persistent execution
Verification case study	Verify a persistent key-value store; count proof lines
Comparison to PEx86	Translate PEx86 into our framework; compare allowed behaviors

Extensions include relaxed persistent models (where the PM controller reorders flushes for

performance), persistent hardware transactional memory (with undo/redo logs managed by the coherence protocol), and crash consistency for CXL-attached disaggregated persistent memory where the crash model changes because memory may survive even if the compute node fails.

5.4 Research Directions

Optimal flush placement via SMT-based program synthesis. Given a persistent program annotated with crash-atomicity requirements, the problem of inserting the minimal set of flush and fence instructions to satisfy PDRF can be framed as an SMT synthesis problem over the program's control-flow graph. The cost function is the number of flushes weighted by their latency impact; the constraint is that every execution prefix flush order respects the persistence axioms. Generating a proof that the synthesized flush placement is optimal (no valid placement uses fewer flushes) would give programmers a push-button correctness tool. Publishable at PLDI or OOPSLA.

Operational–axiomatic equivalence for persistent models under write-back caches. Current persistent models assume write-through caches or explicit flushes. Real processors use write-back caches where evictions spontaneously propagate dirty data to the persistence domain, creating implicit flush events that complicate the equivalence proof between operational and axiomatic specifications. The core problem is defining when an eviction counts as a persistence event and proving that the operational machine produces execution traces matching the axiomatic model's persist order. This would extend persistent models to cover commodity hardware, publishable at ASPLOS or MICRO.

Compositional persistent data structures without locks. PDRF requires critical sections (locks) for shared persistent data, but persistent data structures (persistent linked lists, B^+-trees) often use lock-free techniques for performance. Defining a lock-free persistent correctness condition (analogous to durable linearizability but with flush granularity constraints) and proving that it composes under the persistent memory model would open lock-free persistent programming. The key technical contribution is identifying which flush patterns preserve linearizability under crash. Target venue: PODC or DISC.

Crash-consistent hardware transactional memory with speculative persistence. Integrating persistence into a hardware transactional memory (HTM) system requires that a transaction's writes either persist atomically (on commit) or not at all (on abort). Speculatively persisting writes before commit (for performance) while maintaining the ability to roll back requires an undo log in persistent memory itself, which must survive crashes during the transaction. A hardware mechanism that manages this log transparently, using the coherence protocol to track dirty cache lines and a persistent write-ahead buffer, would bring crash consistency to HTM with minimal software changes. The formal model must prove that after any crash, the persistent state equals some prefix of committed transactions. Publishable at HPCA or ISCA.

Chapter 6

In-Network Coherence for Disaggregated Memory

What is the correct abstraction for cache coherence over disaggregated memory? Should coherence be pushed into the network fabric itself (CXL switches with directory logic), or should we adopt a completely different consistency modelperhaps one where coherence is deactivated by default and activated per-page under OS control?

Disaggregated memory decouples compute and memory into separate physical pools connected by a fabric such as CXL, RDMA, or Gen-Z. A compute node accessing remote memory over this fabric faces a fundamental tension. Keeping the same cache coherence protocol across the fabricOption Agives transparent access but pays the latency and bandwidth cost of coherence messages across a fabric whose round-trips are microseconds, not nanoseconds. Dropping coherence at the fabric boundaryOption Bavoids this traffic but breaks the programming model, forcing software to manage coherence explicitly with uncacheable accesses or explicit flushes. A hybrid approach, Option C, supports both modes with switching per page under OS control: hot pages are hardware-coherent, cold pages are software-managed. The switching mechanism, the consistency guarantees during transitions, and the correctness proof are all open problems. We propose a unified formal framework for reasoning about partially-coherent disaggregated memory systemsa theory of when coherence should cross the disaggregation boundary and the minimal hardware and software support needed to implement that theory.

Definition 6.1 (Disaggregated Memory)**.** A system architecture where compute and memory are separated into distinct physical pools connected by a fabric such as `CXL`, `RDMA`, or `Gen-Z`. A compute node accesses remote memory via load/store instructions, but the round-trip latency is microseconds rather than nanoseconds. Cache coherence across the fabric is optional and can be enabled or disabled per page by the OS.

Definition 6.2 (Coherence Mode Switching)**.** The runtime transition of a memory page between coherence modes: `coh` (full MESI/MOESI coherence), `unc` (uncacheable, every access goes to the fabric), or `wt` (write-through, writes go to fabric immediately, reads may be cached). Switching must be atomic with respect to the memory model: all threads observing the switch must agree on the state before and after the transition.

Theorem 6.1 (Model Equivalence)**.** *The operational machine with per-node coherence controllers and a fabric is equivalent to the axiomatic specification: an access trace is realizable in the operational model iff it satisfies the axioms of the formal memory model* $PM = (A, M, T, \leq)$*. In particular, a write to a coherent address that has been acknowledged is visible to all subsequent reads, and a write to a write-through address becomes visible within bounded time.*

6.1 Related Work

Approach	Strength	Gap
SELCC (RDMA latch-based MSI)	Coherence over RDMA, no remote compute	Specialized for RDMA; protocol-level only
DiFache (Decentralized, snoopy-directory hybrid)	Scalable for many nodes	Designed for datacenter networks, not DM fabrics
PhasedStore (CXL write-through under TSO)	Optimized for CXL, 1.88× speedup	Covers only write-through; no general theory
CXLalloc (Shared allocator for CXL pods)	Manages limited HW coherence for CXL	Allocator-level; not a general coherence model
No coherence for disaggregated memory	Simple, no overhead	Breaks programming model; requires SW management
OS-driven page classification	Deactivating coherence for private pages	Page-level; does not handle disaggregation
RISC-V IO coherence + CMOs (Cheshire SoC)	SW-managed coherence for peripherals	IO coherence, not general-purpose DM coherence

None of these provide a unified formal framework for reasoning about partially-coherent disaggregated memory or a systematic method for deciding what gets coherence.

6.2 Proposed Approach

We define a formal memory model $PM = (A, M, T, \leq)$ where A is the set of addresses, $M : A \to \{\text{coh}, \text{unc}, \text{wt}\}$ maps addresses to coherence mode, T is the set of threads or compute nodes, and $\leq$ is the per-address partial order of accesses. In `coh` mode, the standard MESI or MOESI protocol applies across the fabric and all copies are kept consistent. In `unc` (uncacheable) mode, no caching occursevery access goes to the disaggregated memory pooleliminating coherence overhead at the cost of high latency. In `wt` (write-through) mode, writes go to the pool immediately while reads may be cached but cache lines are invalidated on any remote write. We represent this as both an operational machine with per-node coherence controllers and a fabric, and an axiomatic specification, and we prove equivalence. The central research question is identifying the minimal set of axioms that govern cross-mode accesses: a write to a coherent address that has been acknowledged must be visible to all subsequent reads on any thread regardless of the reader's cache state, and a write to a write-through address must become visible to all subsequent reads but the latency of visibility may be longer.

Architectural Insight

A unified formal framework spanning three coherence modes (`coh`, `unc`, `wt`) enables a systematic cost-benefit analysis of when coherence should cross the disaggregation boundary. The key is that cross-mode accesses obey a minimal set of axioms (acknowledged writes are visible, write-through visibility is bounded) that hold regardless of the current mode, enabling safe runtime switching between modes.

Given an access trace, we need to know which addresses should be coherent and which should not. We define a cost function for each address a: coherence cost is the number of fabric coherence messages per access multiplied by the average fabric message latency plus directory overhead and invalidation traffic; uncacheable cost is the average round-trip latency to the disaggregated memory pool multiplied by access frequency. The optimal mode assignment $M^* = \arg\min_M \sum_a \text{cost}(a, M(a))$ can be solved via dynamic programming if addresses are independent, integer linear programming if there are capacity constraints such as directory size, or greedy approximation if modes are chosen per page.

The most subtle challenge is runtime coherence switching. When an address transitions from coherent to non-coherent (phase-out), all cached copies must be flushed and all outstanding coherence transactions for that address must complete before the page table entry is updated. When transitioning from non-coherent to coherent (phase-in), all cached copies must be flushed and coherence tracking enabled; the first access after phase-in misses and fetches a coherent copy. The correctness condition is that switching is atomic with respect to the memory model: all threads observing the switch must agree on the state before and after. Proving this requires reasoning about all in-flight transactionsan epoch-based approach that divides time into phases where the coherence mode is stable within each phase and switching only occurs at phase boundaries.

For fabric-level coherence, we design a CXL switch with a coherence agent that maintains a directory cache for a subset of addresses in coherent mode. On receiving a coherence request, the switch either responds from its directory cache or forwards to the disaggregated memory pool, participating in the coherence protocol as a peer directory home agent. The challenge is that switch directory capacity is limited: when the switch does not have enough SRAM to track all active addresses, a directory eviction protocol must migrate an address from switch-coherent to node-coherent or software-coherent mode. Cross-mode consistency semantics are also subtleif one thread writes to address X in coherent mode and another thread reads X with uncacheable mode on its node, the reader may see stale data because its uncached read bypasses the coherence protocol. The strictest answer forces implicit coherence even for uncacheable accesses, defeating the purpose; the model must define precisely what cross-mode behavior is legal.

6.3 Evaluation

Workload	**Parameter varied**	**Key metric**
STREAM (bandwidth)	Array size vs. cache capacity	BW utilization, coherence overhead
GUPS (random access)	Memory pool size, thread count	Effective random access latency
TPC-H (database)	Hot vs. cold data ratio	Query latency, coherence traffic reduction
Redis / Memcached	Working set size, GET/SET ratio	Tail latency, coherence overhead
Graph500	Graph size, degree distribution	Traversal rate, cross-fabric coherence

We evaluate across three workload classesmemory-bandwidth-bound (HPC-style streaming),

latency-sensitive (database, key-value store), and mixed cloud workloadsusing a CXL simulator such as gem5 with CXL extensions, RISC-V cores, and a disaggregated memory pool. The evaluation compares all three approaches (full hardware coherence, no coherence, and hybrid per-page switching) to determine whether the optimal point in the design space depends on the workload and where the switching overhead is justified. If the approach succeeds, it extends naturally to security-aware disaggregated coherence (expressing coherence isolation for multi-tenant datacenters), disaggregation over lossy fabrics (graceful degradation to uncacheable mode on message loss), and compiler support for automatic coherence hint insertion.

6.4 Research Directions

Optimal coherence mode assignment under capacity constraints. The cost model $\sum_a \text{cost}(a, M(a))$ assumes independent addresses. Real directory trackers have finite capacity, creating a knapsack-like packing problem: assigning coherent mode to too many addresses causes directory evictions that trigger expensive mode switches. Formulating this as a constrained optimization problem over page-granularity mode assignments with a switching-cost penalty and solving it via integer linear programming or a greedy approximation with a proven competitive ratio would yield a practical OS-level coherence mode allocator. The technical contribution is connecting coherence resource management to classic online optimization, publishable at SOSP or OSDI.

Epoch-based switching with formal atomicity verification. The phase-in/phase-out protocol for runtime coherence switching must guarantee that no thread observes an inconsistent state during the transition. Using epoch-based reasoning where time is divided into numbered phases and switching only occurs at epoch boundaries, the correctness condition reduces to an invariant on epoch numbers at both the page table and each cache controller. The key proof challenge is bounding the drain time of in-flight transactions: proving that after T cycles with no new coherence requests for a given page, all in-flight transactions for that page have completed. This epoch-based framework would provide a general verification methodology for dynamic coherence mode switching, publishable at ASPLOS or MICRO.

In-network coherence directory with eviction and migration to software. A CXL switch with a finite SRAM directory must evict entries when its capacity is exceeded. The eviction protocol must migrate the address from hardware-coherent mode to software-managed mode atomically, with no window where the address is untracked. Designing a two-phase eviction protocol ("prepare-to-evict" asks all nodes to flush cache copies, "evict-ack" confirms the switch may stop tracking) and proving it correct under arbitrary message delays would make in-network coherence practical. A further research question is whether the switch can learn eviction priorities from access frequency to minimize re-coherence overhead. Publishable at SIGCOMM or NSDI.

Formal connection between disaggregated coherence and persistent consistency. A disaggregated memory system with coherence deactivated resembles a persistent memory system where each write is immediately visible but with microsecond-scale latency. The coherence mode $\{\text{coh}, \text{unc}, \text{wt}\}$ can be related to the persistent $\{\text{flush}, \text{no-flush}\}$ semantics: write-through mode effectively inserts an implicit flush on every store. Proving a formal equivalence between the disaggregated coherence model under write-through mode and a persistent memory model (from chapter Q5) would unify two active research areas and enable cross-domain proof reuse. Target venue: ISCA or HPCA.

Chapter 7

Systematic Side-Channel Mitigation Verification

Can we build a generative verification framework that, given a microarchitectural specification and a security property (e.g., "no secret-dependent timing variation"), automatically derives the minimal set of microarchitectural constraints that guarantee the property?

The current approach to side-channel mitigation is reactive: a new attack class is discovered (Spectre, Meltdown, RIDL, Fallout, DMP), researchers analyze the root cause, a mitigation is proposed and implemented, the mitigation is tested against known attacks, and eventually a formal verification appears months or years later. As microarchitectures grow more complexlarger speculation windows, deeper pipelines, learned prefetchers, chipletsthe opportunity for side channels grows, and the reactive approach cannot keep pace. Instead of verifying a specific mitigation against a specific attack, we propose a generative framework that takes a formal specification of a microarchitecture as input and outputs a set of constraints on the microarchitecture that, if satisfied, guarantee the absence of a given class of side channels, or a counterexample showing why the microarchitecture is vulnerable. The constraints are not one-size-fits-all; they are derived automatically from the microarchitecture's structure.

7.1 Related Work

Approach	Strength	Gap
Triosecuris (CET+SLH in Coq)	First formal proof for Spectre mitigations	Proves specific mitigations, not absence of all channels
SPLASH (SIFT tracking on BOOM)	Hardware speculation tracking	Verified for known variants only; no generative framework
Kôika + Coq (Formal HDL with security proofs)	Full-stack formal verification	Requires per-design manual proof; no automation
riscv-formal (Bounded model checking)	Checks safety at RTL	Bounded; cannot prove absence of timing channels
ClepsydraCache (Constant-time cache)	Timing-safe cache architecture	Cache only; not a general framework
SCOUT-CT / Dalc-CT (Binary-level CT)	Analyzes compiled binaries	Software-level; assumes microarchitecture is CT

No existing tool takes a microarchitecture specification as input and automatically derives per-microarchitecture constraints for timing-channel freedom.

7.2 Proposed Approach

We define a formal timing model $M = (S, \sigma, C, T)$ where S is the set of microarchitectural states, σ is the transition function representing one cycle of execution, $C \subset S$ is the set of architectural commitment events (instructions retire), and $T : S \times S \to \mathbb{N}$ is the timing function giving the number of cycles between two states. The model abstracts away data values in functional units but retains all timing-relevant structure: pipeline stages, cache hierarchy, speculation mechanisms, prefetchers, store buffers, and resource arbiters. It is expressed in a domain-specific language for describing pipelines.

The core of the approach is a timing taint analysis that tracks, for each state element in S, whether its value can influence the latency of any instruction.

Definition 7.1 (Timing Taint)**.** A state element $s \in S$ is *secret-tainted* if its value derives from a value labeled secret; s is *timing-sensitive* if its value can influence the timing function T. A *timing channel* exists iff there exists a path from a secret-tainted element to a timing-sensitive element in the dependency graph of the transition function σ.

A timing channel exists if there is a path from a secret-tainted element to a timing-sensitive element. This is analogous to information flow tracking in hardwareRTLIFT, GLIFTbut applied to timing instead of data. The principal research question is identifying the minimal formal model that captures all relevant timing sourcesspeculation, caching, prefetching, resource contention, ALU latency variabilitywhile abstracting away irrelevant details like data values and forwarding paths.

From the taint analysis, we generate constraints of three types. Structural constraints require that components like register file address decode have constant delay, since variable-latency decode with a secret-tainted address register creates a timing channel. Resource constraints require that the L1 cache either have constant hit or miss latency or be inaccessible to secret-tainted addresses. Behavioral constraints require that speculation not be guided by secret-tainted valuesthis is the Spectre class of vulnerabilities. Each constraint is a property $\phi_i(s)$ over the microarchitectural state.

Theorem 7.1 (Timing-Channel Soundness)**.** *Given a timing model $M = (S, \sigma, C, T)$ and a set of constraints $\Phi = \{\phi_1, \ldots, \phi_k\}$ derived from the timing taint analysis, if every reachable state $s \in S$ satisfies all $\phi_i \in \Phi$, then there exists no timing channel from any secret-tagged state element to any timing-sensitive element in M.*

Proof sketch. Assume a timing channel exists: there is a path $s_0 \to s_1 \to \cdots \to s_m$ in the dependency graph of σ from a secret-tainted element to a timing-sensitive element. By induction on path length, either a structural constraint (for ALU/data-path elements), a resource constraint (for cache/memory elements), or a behavioral constraint (for speculation elements) is violated at the first step where the taint propagates to a timing-relevant component. Contrapositively, if all ϕ_i hold, no such path exists. □

Given the constraints, we use existing formal tools to verify that the RTL implementation satisfies each ϕ_i. For bounded-time constraints such as fixed cache hit or miss latency, bounded model checking (SymbiYosys, riscv-formal) suffices. For unbounded constraints such as speculation never depending on secret data, induction or a dedicated proof assistant such as Coq via Kôika

is needed. For probabilistic constraints involving learned components like a neural prefetcher, statistical model checking provides a weaker but acceptable guarantee. If a constraint is violated, the framework produces a counterexample tracean automatically discovered side-channel attack.

The hardest challenges are abstraction and scale. A model that abstracts away data values may miss timing channels that depend on specific data patterns, such as a multiplier whose latency varies with the popcount of its operands. The correct level of abstraction is unclear. Taint analysis over a large microarchitecture like BOOMtens of thousands of state elementsmust exploit modular structure to stay tractable. And unbounded verification of speculation properties requires reasoning about the entire speculation mechanism across an unbounded number of cycles, which is beyond most current verification tools; the thesis may need to accept bounded guarantees with statistical confidence for larger windows.

Core Thesis

A generative verification framework that, given a formal microarchitecture specification, automatically derives per-design constraints sufficient to guarantee timing-channel freedom, replaces the current reactive approach where each new attack class (Spectre, Meltdown, RIDL) demands a manual mitigation and a bespoke verification effort.

7.3 Evaluation

Design	**Size**	**Expected Channels**	**Verification Method**
CVA6 (in-order)	20K LUTs	Cache timing, ALU latency	Bounded model checking (automatic)
BOOM (OoO)	100K LUTs	Spectre v1/v2, cache timing	Induction + Coq (partially manual)
CVA6 + ClepsydraCache	25K LUTs	Only ALU channels remain	Bounded model checking
BOOM + SPLASH	110K LUTs	Should eliminate Spectre channels	Induction + BMC

We evaluate on three designs: CVA6 (a 6-stage in-order RISC-V core), BOOM (an out-of-order RISC-V core), and CVA6 with ClepsydraCache. For each design, we run the taint analysis, generate constraints, verify them, and report timing channels found. Metrics include channel coverage (which known attacks are detected and which new ones are discovered), false positives, proof effort in lines of proof code and CPU time, and the performance cost of satisfying all constraints. If the approach succeeds, it extends naturally to power side channels (tracking switching activity instead of cycles), EM side channels (with spatial localization), online monitoring (hardware detectors for constraint violations during execution), and compiler-aware constraints that relax requirements based on compiler guarantees.

7.4 Research Directions

Timing taint analysis with automated abstraction refinement. The core challenge of timing taint analysis is choosing the right abstraction level: too detailed and analysis becomes intractable for out-of-order cores; too coarse and real timing channels are missed. A counterexample-guided abstraction refinement (CEGAR) loop where the abstract timing model is iteratively refined until either all timing channels are eliminated or a concrete attack trace is produced would give a push-button verification tool. The technical novelty is a timing-channel-specific abstraction that preserves latency paths while abstracting data values: a lightweight relational abstraction over pipeline state that is precise enough to capture Spectre-v1-style speculation gadgets. Publishable at CAV or FMCAD.

Automatic constraint synthesis for speculation-tracking mechanisms. Given the timing model M, the taint analysis outputs structural, resource, and behavioral constraints ϕ_i. For speculation, the constraints state that no secret-tainted value may influence the prediction outcome. Automatically synthesizing a speculation-tracking mechanism (like SPLASH's SIFT table) that guarantees these constraints, and generating an RTL implementation in Chisel, would raise the level of abstraction from manual mitigation design to automated constraint-driven synthesis. The proof would show that any implementation satisfying the synthesized specification is secure by construction. Target venue: DAC or DATE.

Probabilistic side-channel verification for learned microarchitectural components. Neural prefetchers, learned branch predictors, and other ML-based components introduce timing channels that depend on learned internal state, which is data-dependent in ways that are difficult to bound deterministically. Proving that a learned prefetcher satisfies an (ϵ, δ)-timing-leakage guarantee (the probability that a timing difference exceeds ϵ cycles is at most δ) would provide a rigorous foundation for deploying learned components in security-critical cores. The proof technique would combine concentration inequalities from learning theory with formal timing models of the pipeline. Publishable at S&P or USENIX Security.

Online timing-channel monitoring with formal alarm guarantees. Rather than verifying the entire microarchitecture statically, one could add lightweight hardware monitors that detect timing-channel violations at runtime. The research question is whether a monitor can be designed that raises an alarm whenever a secret-tainted value influences instruction timing, and provably never misses a violation (soundness) while keeping false alarms bounded. The monitor checks the taint-flow conditions ϕ_i at retirement; a violation triggers a precise architectural exception that can replay the offending instruction with constant-time timing. The formal guarantee would be that any execution that passes the monitor satisfies the timing-independence property. Publishable at MICRO or HPCA.

Part II

Compilers

Chapter 8

Polyhedral Compilation Beyond Affine Abstractions

Can we extend polyhedral theory to encompass piecewise-affine abstractions with control-flow-dependent bounds, or develop a hierarchy of abstractions where the compiler automatically selects the most precise tractable model for each loop nest?

Polyhedral compilation is one of the great success stories of formal methods in compilers: affine loop nests can be transformed optimallytiling, fusion, fission, skewing, parallelizationwith provable guarantees on data reuse and parallelism. The polyhedral model provides an exact representation of loop iterations as integer lattice points, a closed-form description of data dependences, and a scheduling space where optimal transformations can be found via integer linear programming. But the model has a hard boundary: it works only for affine loop nests where array subscripts and loop bounds are affine expressions of enclosing loop counters and symbolic constants. For anything elseindirect array accesses $A[B[i]]$, pointer-chasing, recursion, non-linear recurrences, data-dependent control flowthe model gives up and falls back to non-polyhedral heuristics. Only about thirty to forty percent of loops in typical programs are fully affine, and extending polyhedral reach by even ten to twenty percentage points would have significant impact.

Problem 8.1 (Hierarchical Polyhedral Abstraction Selection)**.** Given a loop nest L with iteration domain $\mathcal{D} \subseteq \mathbb{Z}^d$, array access functions $f_1, \ldots, f_k : \mathcal{D} \to \mathbb{Z}^m$, and loop bounds $g_1, \ldots, g_\ell : \mathcal{D} \to \mathbb{Z}$, determine the highest abstraction level $\ell \in \{0, 1, 2, 3, 4\}$ such that L admits a sound polyhedral transformation, and produce a schedule $S : \mathcal{D} \to \mathbb{Z}^d$ minimizing an objective (e.g., data reuse distance) under the chosen abstraction.

Graceful Degradation Hierarchy

The key idea is a five-level hierarchy of polyhedral abstractions that degrades gracefully: Level 0 (fully affine) through Level 4 (runtime polyhedral inspectors). The compiler automatically selects the highest level that applies, extending polyhedral optimization to programs that were previously out of reach. The central challenge is designing tractable dependence analysis for piecewise-affine accesses and minimizing the overhead of runtime inspectors.

We propose a hierarchy of abstractions that degrades gracefully: Level 0 is the current affine polyhedral model; Level 1 extends to piecewise-affine (affine with data-dependent guards); Level 2 offers semi-affine escape hatches (affine with limited non-affine operations); Level 3 provides affine approximation with runtime checks; and Level 4 applies polyhedral-guided transformation to non-affine regions enclosed by polyhedral code. The compiler automatically analyzes a loop nest, determines the highest applicable level, and applies the corresponding transformation strategy.

8.1 Related Work

Approach	Strength	Gap
Polygeist (C → MLIR affine)	Raises C/C++ to MLIR Affine dialect	Falls back when affine extraction fails
Pluto (ILP-based polyhedral scheduler)	Optimal affine scheduling	Affine-only
Looper (ML-based polyhedral search)	Deep learning for transformation search	Still affine-representable only
Symbolic Locality Compiler (AutoLALA)	Closed-form reuse formulas	Polynomial access patterns only
PolyBlocks (MLIR + affine for ML accelerators)	Multi-level tiling, fusion	Assumes regular tensor shapes
Inspector-executor (CHILL, SPIRAL)	Runtime specialization for irregular codes	Runtime overhead; limited polyhedral insight
Sparse polyhedral framework	Extends to sparse matrix computations	Specialized; not general non-affine

No existing work provides a systematic hierarchy of polyhedral abstractions that degrades gracefully for non-affine code.

8.2 Proposed Approach

We begin with the piecewise-affine polyhedral model at Level 1. A piecewise-affine access function takes the form $A(f(i))$ where $f(i) = f_k(i)$ if $g_k(i) \geq 0$ for $k \in K$, with each f_k and g_k affine. Dependence analysis becomes a Presburger arithmetic problem with piecewise definitions. The central research question is whether dependence analysis for piecewise-affine accesses remains tractable; Presburger arithmetic is decidable but worst-case exponential, so the thesis must identify practical subclasses where it is efficient ($\mathcal{O}(n^3)$ or better)for example, at most one breakpoint per dimension or piecewise expressions with small domain partitions.

At Level 2, for programs that are mostly affine with a few non-affine escape operations, we define the affine hull: replace $A[B[i]]$ with a conservative abstraction $A[0 : N - 1]$. The resulting transformation is safeno dependence is missedbut conservative, potentially disallowing transformations that would be safe in the original program. The challenge is refining the hull iteratively: start with the whole array as the abstraction, produce a schedule, verify it with runtime checks, and if checks fail, refine and re-schedule.

At Level 3, for a non-affine loop nest, we compute the best affine approximation, schedule it polyhedrally, and generate residual code that checks whether the transformed program is equivalent to the original under the approximation. The equivalence check is a polyhedral verification problem known to be undecidable in general but decidable for restricted classes such as bounded-error or monotonic approximations with $\mathcal{O}(n \log n)$ verification cost.

At Level 4, for code where no static approximation is useful, we insert polyhedral inspectors that compute the access pattern at runtime and apply polyhedral transformations on the fly. The

risk is that the inspector overhead may exceed the transformation benefit, so the decision procedure must account for this cost.

Algorithm 1 Hierarchical Abstraction Selection

Require: Loop nest L, maximum level $\ell_{\max} = 4$
Ensure: Abstraction level ℓ^*, polyhedral schedule S

$\ell \leftarrow 0$
while $\ell \leq \ell_{\max}$ **do**
    Extract affine model M_ℓ at level ℓ
    if M_ℓ is sound for L **then**
        Compute schedule $S \leftarrow \textsc{PolyhedralSchedule}(M_\ell)$
        Verify S via runtime checks (Level ≥ 3)
        if verification passes **then return** (M_ℓ, S)
        end if
    end if
    $\ell \leftarrow \ell + 1$
end while
return fallback with heuristic schedule

The abstraction selection algorithm proceeds by attempting affine extraction, identifying the non-affine operations through symptom-based boundary detectiondistinguishing cases like "access function reads from another array whose values are computed at runtime" (Level 3 or 4) from "access function jumps between two linear values based on a data-dependent condition" (Level 1)and selecting the appropriate transformation for each operation. The hardest challenge is that conservative approximations may be too conservative: if the affine hull of a non-affine expression covers the entire array, no useful transformation is possible, and the hierarchy may give only marginal improvement over giving up entirely.

8.3 Evaluation

Benchmark suite	**Coverage metric**	**Performance metric**
PolyBench/C	% loops at Level 0 (affine)	Speedup over -O3
Rodinia	% loops at Levels 0–4	Speedup, % of optimal
SPARSKIT	% loops accepted	Execution time, memory traffic
Graph500 / PageRank	Level 3 or 4 applicable?	Throughput vs. hand-optimized
Custom irregular	Boundary detection accuracy	Abstraction selection precision

We implement the hierarchy as an extension to the Polygeist or MLIR pipeline and evaluate across PolyBench/C (regression at all levels), Rodinia (heterogeneous workloads with non-affine kernels), SPARSKIT and OSKI (sparse matrix with indirect accesses), and real-world irregular applications including PageRank, BFS, and database hash joins. Comparison targets include LLVM -O3, Polygeist, and Pluto. The key questions are how much additional coverage each abstraction level provides, what the performance degradation is from Level 0 to Level 4 for programs that could be handled at a higher level, and how often the abstraction selection algorithm picks the wrong level. If the approach succeeds, it extends naturally to dynamic polyhedral compilation (runtime

inspectors), polyhedral-guided value speculation (speculating affine forms with lightweight checks), and automatic GPU offloading decisions for non-affine code.

8.4 Research Directions

Presburger Dependence Analysis for Piecewise-Affine Programs. A natural extension is to formalize dependence testing for the Level 1 piecewise-affine model as a Presburger arithmetic problem with $\exists\forall$ quantifier alternation over piecewise-defined subscript expressions. This is decidable but worst-case doubly-exponential; the key is identifying tractable subclasses, e.g., piecewise expressions with $O(1)$ breakpoints per dimension or affine guards forming a convex partition. A publication at **PLDI** or **OOPSLA** could present the first practical dependence test for piecewise-affine programs, benchmarking against manually parallelized irregular codes.

Learning-Based Abstraction Selection for the Hierarchy. The proposed five-level hierarchy requires the compiler to select the right abstraction automatically. A supervised learning approach could predict the appropriate level from static features of a loop nest: number of non-affine accesses, branch entropy, pointer-chasing depth, and array dimensionality. Training on a corpus of 10,000+ extracted loops with labeled optimal abstraction levels (measured by runtime on a simulator) would make an excellent **CGO** paper, with the novelty being the first learned polyhedral abstraction selector.

Runtime-Polyhedral Co-Design with Adaptive Inspectors. Level 4 calls for polyhedral inspectors that compute access patterns at runtime, but overhead may exceed benefit. A co-design approach could generate inspector code that adaptively decides at runtime, based on the discovered access pattern, whether to apply a polyhedral transformation or fall back to the original code. Published at **MICRO** or **ASPLOS**, this would contribute the first adaptive polyhedral runtime system for fully irregular codes.

Verified Correctness of Polyhedral Approximations. Each abstraction level trades precision for tractability, but the transformations derived under an approximation must preserve the original program's semantics. A verification framework in Coq or Lean that proves soundness conditions for each level, e.g., that the affine hull abstraction is a safe over-approximation, would be a strong **POPL** or **CPP** contribution, complementing existing work on verified polyhedral scheduling.

Chapter 9

Optimal Unified Code Generation (ISEL + RA + Scheduling)

Can we solve instruction selection, register allocation, and instruction scheduling optimally as a single combinatorial problem for realistic instruction sets (RISC-V RV32/RV64 with compressed instructions)?

Every optimizing compiler separates code generation into three sequential passes: instruction selection (ISEL), register allocation (RA), and instruction scheduling. This separation is a pragmatic necessitysolving all three jointly is NP-completebut it is known to be suboptimal. A good instruction selection may be impossible because the needed registers are not available when RA runs later and spills. A good schedule may be impossible because the register allocator introduced false dependences through spilling. An aggressive instruction selector may generate instructions with high register pressure, forcing later RA to spill and negating the selection benefit. Estimates place the performance gap between the separated approach and the optimal joint solution at five to twenty percent, and for compressed instruction sets like RISC-V C, the interaction is even more pronounced because an instruction's compressed form depends on both the register assignment and the immediate range.

Problem 9.1 (Unified Code Generation)**.** Given a data-dependence graph $G = (V, E)$ where each node $v \in V$ has alternative instruction mappings $M_v = \{m_1, \ldots, m_k\}$ with latency $L(m_i)$, register requirement $R(m_i)$, code size $S(m_i)$, and encoding $E(m_i)$, and a target machine with P physical registers and F functional unit types, find an assignment $\alpha : V \to \bigcup M_v$, a register assignment $\rho :$ vregs $\to \{1, \ldots, P\} \cup \{\text{spill}\}$, and a schedule $\sigma : V \to \mathbb{N}$ minimizing $\lambda \cdot \text{latency}(\sigma) + (1-\lambda) \cdot \text{code\_size}(\alpha)$ subject to resource and data-dependence constraints.

We propose to solve all three passes jointly by exploiting structural properties of practical programsbounded treewidth, small register pressure, limited instruction-level parallelismto make the joint problem tractable for real-world instances. The result would be a unified code generation pass that replaces the three separate passes in LLVM, delivering provably optimal code for functions where the joint problem is tractable and falling back gracefully to the separated approach for intractable cases.

9.1 Related Work

Approach	Strength	Gap
LLVM (separate ISEL, RA, scheduling)	Industrial strength, good heuristics	Separation is suboptimal by design
TPDE (Single-pass ISEL+RA+encoding)	8–26× faster than LLVM -O0	No scheduling; optimizes for speed
Optimal RA via treewidth	Polynomial optimal RA for bounded treewidth	No ISEL or scheduling
Series-parallel loop RA	Spill-free RA for structured programs	No ISEL or scheduling
Global ISEL by tiling (LLVM SelectionDAG)	Optimal ISEL for basic blocks	No RA interaction
Integer programming for RA (LSSA)	Exact RA formulation	ISEL assumed fixed
Integrated ISEL+RA (Janssen, Corporaal)	Combined for tiny regions	Does not scale; no scheduling

No existing approach solves the full ISEL, RA, and scheduling problem optimally for practical instruction sets on real-sized functions.

9.2 Proposed Approach

Parameterized Tractability

The key insight is that the joint ISEL+RA+scheduling problem, while NP-complete in general, becomes fixed-parameter tractable for realistic instances: bounded register pressure (≤ 50), small DDG treewidth (≤ 8), and few ISEL alternatives per node (≤ 5) make dynamic programming over a tree decomposition practical. This is the first theoretical justification for exact unified code generation in production compilers.

We formulate the unified code generation problem as a joint combinatorial optimization. The input consists of a data-dependence graph (DDG) where each node has a set of feasible instruction mappings (typically one to five alternatives per operation), each with latency, register requirements, code size, and encoding. The target machine model specifies the number of physical registers per class, functional unit types, and issue constraints. The variables are the ISEL mapping per node, the physical register assignment per virtual register (or a spill decision), and the instruction order. The objective minimizes a weighted combination of execution time (critical path latency) and code size.

The central research question is the parameterized complexity of the joint problem. The key parameters are the number of ISEL alternatives per node (typically one to five), the register pressure (maximum live virtuals at any point, typically at most fifty for real functions), the treewidth of the DDG, and the number of functional unit types. We aim to prove that the joint problem is fixed-parameter tractable when these parameters are bounded: a tree-decomposition of the DDG enables dynamic programming on each bag via enumeration, and the solutions combine across the

decomposition. For the majority of real functionswhere register pressure stays under thirty-two and control-flow graph treewidth is typically five to eightthe joint problem would be polynomial-time solvable. If fixed-parameter tractability does not hold, we must identify which parameter causes the hardness jump.

We build an anytime exact solver that can be stopped early and return the best solution found so far, critical for integration into a production compiler where compile time is constrained.

Algorithm 2 Anytime Unified Code Generation

Require: DDG G, register count P, ISEL alternatives $\{M_v\}$
Ensure: Assignment α, schedule σ, registers ρ
$T \leftarrow \textsc{TreeDecompose}(G)$ ▷ treewidth ≤ 8 for hot blocks
best $\leftarrow \infty$
for each bag B in T (bottom-up) **do**
    table$[B] \leftarrow \textsc{DpTable}(B, P, \{M_v\})$
    if time budget exceeded **then return** $\textsc{ExtractBest}$(table)
    end if
end for
$(\alpha, \sigma, \rho) \leftarrow \textsc{ExtractBest}$(table)
return (α, σ, ρ)

The solver uses one of three approachesILP via Gurobi or CPLEX, SAT with pseudo-boolean constraints, or a custom branch-and-bound solver exploiting the specific structurewhichever performs best. For small functions of up to one hundred instructions, we solve the joint problem exactly. For medium functions of one hundred to five hundred instructions, we solve per basic block and combine across blocks via a separate RA pass. For large functions beyond five hundred instructions, we fall back to the standard separated approach but validate with the exact solver on hot regions identified by profiling. A learned classifier predicts whether the joint solver will finish within a time budget and only runs it when the prediction is positive.

The hardest challenges are scaling, treewidth, and interaction complexity. Even with FPT guarantees, the exponent may be largefor instance, if the ILP size is $O(2^{k_1} \cdot k_2^2 \cdot \text{treewidth})$ with $k_2 = 32$, the constant factor is enormous. The thesis must carefully bound constants through problem-specific reasoning, such as exploiting that most register interference graphs are chordal. DDG treewidth may not be small for programs with complex data flows, so we may need to restrict to basic blocks or superblocks. And ISEL alternatives interact in complex ways: the choice for one operation affects register pressure, which affects scheduling, which feeds back to ISEL. The formulation must capture these interactions without exploding in size, perhaps by restricting alternatives to those that are Pareto-dominant before encoding.

9.3 Evaluation

Benchmark	Metric	Comparison
SPEC CPU2017 (subset)	Execution time, code size	LLVM -O3, LLVM -Oz, GCC -O3
PolyBench/C	Execution time	LLVM -O3, Pluto
CoreMark	Code size, speed	RV32 baseline
Embench	Code size	LLVM -Oz
Custom microbenchmarks	Optimality gap (exact)	LLVM vs. exact
Clang test suite	Compilation time	Unified vs. separate passes

We implement the unified pass as an LLVM pass replacing SelectionDAG ISEL, greedy RA, and pre-RA or post-RA scheduling, targeting RISC-V RV32 with the C-extension. Evaluation uses SPEC CPU2017, CoreMark or Embench, MiBench, and custom microbenchmarks that allow exact solution for measuring the optimality gap. Comparison targets include LLVM -O3, LLVM -Oz, GCC -O3, and LLVM with optimal treewidth-based RA but heuristic ISEL and scheduling. The key questions are the average optimality gap, the percentage of functions on which the unified pass finishes within twice LLVM's current compilation time, and how much compressed instruction support affects the gap. If successful, the formulation extends to VLIW bundling constraints, vector register allocation with overlap constraints, and energy optimization using weighted energy models.

9.4 Research Directions

Parameterized Complexity Classification of Joint ISEL+RA+Scheduling. The central theoretical question is whether the joint problem is fixed-parameter tractable (FPT) with respect to natural parameters: register pressure, treewidth of the DDG, number of ISEL alternatives per node, and functional unit count. Proving FPT membership would require a dynamic programming algorithm over a tree decomposition with state space $O(f(k) \cdot n)$, while a parameterized hardness result would identify the exact source of intractability. A **POPL** paper establishing these complexity bounds would be the first rigorous analysis of the unified code generation problem.

Anytime Exact Solver with Learned Time-Budget Predictors. Deploying exact unified code generation in a production compiler demands an anytime solver that returns the best solution found when time runs out. A concrete system combining a SAT-with-pseudo-boolean-constraints encoding and a learned classifier that predicts whether the solver finishes within a given time budget (features: function size, register pressure, number of ISEL alternatives) would make a strong **PLDI** or **CGO** contribution. The novelty lies in the tight integration of machine learning into a combinatorial solver for code generation.

Optimal Unified Code Generation for VLIW Architectures. VLIW bundles add a dimension: ISEL must generate enough operations per cycle to fill issue slots, RA must avoid false dependences that prevent bundling, and scheduling is exposed. Extending the unified formulation to VLIW, with bundle constraints, heterogeneous functional units, and operation grouping rules, targets **MICRO** or **CGO**. The expected contribution is the first VLIW code generator that is provably optimal for hot basic blocks on RISC-V with the P-extensions or a DSP-style VLIW.

Unified Code Generation for Vector Register Allocation. Vector registers (RISC-V V-ext, AVX-512) introduce overlap constraints (e.g., LMUL groups) and variable-length operations, where ISEL chooses vector length and RA decides which vector registers to use. Jointly optimizing these with VLIW-style scheduling for SIMD cores is a natural generalization. Published at **MICRO** or **ASPLOS**, this would contribute the first exact formulation of vector code generation that simultaneously selects vector lengths, allocates vector registers, and schedules vector operations.

Chapter 10

Verified Preservation of Security Properties Under Optimization

Can we build a verified compiler pipeline that guarantees the preservation of security properties (constant-time, speculative constant-time) across all optimization passes?

CompCert proves that the observable behavior of a program is preserved under compilationif the source program terminates with value v, the compiled program does the same. But observable behavior in traditional verified compilers means I/O and termination. Security properties like constant-time executionexecution time independent of secret valuesare hyperproperties: properties of sets of executions, not of individual executions, and they are not preserved by standard semantics-preserving compilation. A compiler optimization pass that preserves semantics can still introduce timing variability that leaks secrets. If-conversion converting a conditional branch to a conditional move may introduce variable-latency execution. Strength reduction replacing multiply by shift-and-add may have operand-dependent timing. Common subexpression elimination may eliminate a timing-invariant load but leave a timing-variant access pattern. Instruction scheduling may introduce secret-dependent resource contention.

We propose to extend verified compilation to preserve hyperpropertiesspecifically constant-time (CT) and speculative constant-time (SCT)resulting in a verified compiler that produces provably constant-time binaries for any constant-time source program, regardless of which optimizations are applied.

10.1 Related Work

Approach	Strength	Gap
CompCert (Leroy et al.)	Verified semantics preservation for C	Does not preserve hyperproperties
CompCert-CT (Barthe et al.)	Adds CT preservation for subset of passes	No speculation coverage
Jasmin (Almeida et al.)	Verified compiler for crypto with CT checking	Specialized for crypto; not general-purpose
Bedrock2 → RISC-V (Porncharoenwase et al.)	Smooth omnisemantics proofs for CT	RISC-V specific; simple optimizations only
Fiat Cryptography (Erbsen et al.)	Coq-verified crypto code generator	EC-specific; not a general compiler
SCOUT-CT (Daniel et al.)	Sound binary-level CT analysis	Analyzes after compilation; does not guarantee
SLH + CFI	Mitigates Spectre at compile time	Heuristic; not formally verified

No verified general-purpose compiler guarantees CT or SCT preservation across its optimization pipeline.

10.2 Proposed Approach

We extend CompCert's semantics with leakage events. Each reduction step produces a leakage label capturing the instruction type, the addresses of memory accesses, the execution time as an abstract latency, and the branch outcome. The leakage label records only what an attacker could observe.

Definition 10.1 (Leakage-Aware Simulation)**.** A leakage-aware simulation R between source program S and compiled program T is a relation on configurations such that for every pair $(s,t) \in R$ with matching leakage labels, if $s \xrightarrow{\ell} s'$ in S, then there exists t' with $t \xrightarrow{\ell'} t'$ in T and $(s',t') \in R$, where $\ell' = f(\ell)$ for a compiler-specific mapping f that accounts for instruction fusion or splitting.

The leakage label records only what an attacker could observe. We define a leakage-aware simulation diagram connecting source and target states with the condition that the compiled program's leakage is equivalent to the source's leakage up to some mapping that accounts for the compiler fusing or splitting instructions.

The central research question is identifying the right formal definition of constant-time preservation in a verified compiler. Two candidates are equivalence of leakage setsfor every pair of secret-distinct inputs, the observable leakage is identicaland leakage-simulation, where the compiled program's leakage is a function of the source program's leakage. In CompCert's simulation-diagram framework, we need leakage-aware simulation diagrams that track both the observable state and the leakage produced so far.

For each of CompCert's approximately thirty optimization passes, we prove a 2-safety property:

Theorem 10.1 (CT Preservation under Verified Compilation)**.** *For any CompCert optimization pass P and any pair of source programs S_1, S_2 that differ only in their secret inputs, if S_1 and S_2 produce indistinguishable leakage labels at every program point, then $P(S_1)$ and $P(S_2)$ also produce indistinguishable leakage labels at every corresponding program point. Moreover, if P satisfies the leakage-aware simulation condition, then P preserves constant-time execution.*

Proof sketch. The proof uses a product-program construction over the leakage-aware simulation diagram. For each pair of source states (s_1, s_2) with identical leakage, we construct a product state (s_1, s_2) in a 2-self-composition of the source. The pass P induces a relation on product states; we show by induction on the number of reduction steps that if the product source states produce matching leakage, the product compiled states also produce matching leakage, using the mapping f from the leakage-aware simulation to relate compiled leakage labels to source leakage labels. □

The proof uses a product program technique combining two executions of the source program and two executions of the compiled program, showing that if the source leakage pairs match, the compiled leakage pairs also match. Some passes are inherently CT-preservingdead code elimination removes code without adding timing variabilitywhile others may break CT and need modification or restriction. If-conversion, for instance, must be restricted to cases where the condition is public. Instruction scheduling may be made CT-safe by restricting the scheduler to avoid secret-dependent resource contention.

We extend the theorem to speculative constant-time by adding a speculation flag to the abstract machine. When the machine enters a speculative regionfor example, after a branch whose condition depends on a secretall leakage events are labeled as speculative. An SCT-preserving compiler guarantees that speculative leakage is independent of secrets. This is strictly harder than CT preservation because the compiler must also preserve speculation behavior: an aggressive scheduler that speculates differently depending on register allocation could break SCT even while preserving CT. Two executions with different secrets but identical public prefix and speculation behavior must have identical leakage.

The hardest challenges are proof complexity and the hardware leakage mapping. CompCert already requires approximately 150K lines of Coq for observable-behavior preservation; adding leakage-awareness could double or triple this. The proof must share as much structure as possible with existing proofs. Modeling leakage for complex optimizations like if-conversion, loop unrolling, and scheduling is difficult because these passes change the leakage structure fundamentally. SCT is much harder than CT because the speculative execution model adds a new dimension to the simulationspeculation depth and misspeculation recoveryand a pass that preserves CT may not preserve SCT. Finally, the compiler's leakage model is abstract, and the preservation theorem depends on the hardware being leakage-faithful: the actual hardware timing must be a deterministic function of the abstract leakage labels. This is an explicit assumption linking to the hardware's own constant-time specification.

Core Thesis

Extending CompCert's simulation-diagram framework with leakage labels transforms hyperproperty preservation (constant-time, speculative constant-time) from an external verification problem into a compiler-internal proof obligation, making security preservation a first-class citizen of verified compilation alongside observable-behavior preservation.

10.3 Evaluation

Criteria	Method	Target
Pass coverage	% of passes with CT proof	100% of CT-preserving passes
CT verification	Run SCOUT-CT on compiled crypto	Zero false positives
Performance	SPEC CPU2017 of CT vs. standard	Measure overhead of CT restrictions
Proof size	Lines of Coq proof	Acceptable (< 100K new lines)
SCT coverage	Compile Spectre gadget, verify SCT	Binary is SCT or compiler rejects

We implement the CT-preserving CompCert pipeline by extending CompCert's intermediate languages with leakage annotations, modifying each pass to be leakage-preserving, and proving the CT preservation theorem per pass. Validation compiles real cryptographic codePoly1305, ChaCha20, AESand checks the resulting binaries with SCOUT-CT. The key metrics are pass coverage (percentage of CompCert passes with CT preservation proved), binary CT verification (zero false positives), performance overhead of CT restrictions measured via SPEC CPU2017, and proof size. SCT coverage is evaluated by compiling a Spectre v1 gadget and verifying that the binary is SCT or that the compiler rejects it. If successful, the framework extends to power and EM leakage preservation, bounded timing channels, and mainstream compiler integration via translation validation after optimization.

10.4 Research Directions

Leakage-Aware Simulation Diagrams for CompCert. The core technical contribution is extending CompCert's simulation-diagram framework with leakage labels at each reduction step. A leakage-aware simulation must satisfy that for every pair of source states with identical leakage, the compiled states produce leakage-equivalent traces. Proving this for all ∼30 CompCert passes, and identifying which passes inherently break CT, would be a foundational result. A **POPL** paper laying out the leakage-aware simulation framework, with complete proofs for a core subset of passes, would establish the theoretical foundations for verified hyperproperty preservation.

Speculative Constant-Time Preservation Under Scheduling. Proving SCT preservation for an instruction scheduler is particularly challenging: the scheduler reorders operations, potentially introducing speculation that depends on register allocation, which itself depends on secret values. Formally modeling speculative leakage events with speculation depth and misspeculation recovery, then proving a scheduler correct, would be a significant advance over CompCert-CT (which covers neither speculation nor scheduling). Published at **IEEE S&P** or **POPL**, this would contribute the first verified SCT-preserving scheduler.

Product-Program Verification of 2-Safety for Global Optimizations. Many CompCert passes (e.g., common subexpression elimination, loop invariant motion) operate globally across basic blocks. Proving 2-safety (indistinguishability of leakage for secret-distinct inputs) for these passes requires product programs that combine two executions. The product program construction

for global optimizations with control flow is non-trivial: the product must align corresponding program points across secret-distinct executions, which is undecidable in general. A **PLDI** paper identifying decidable subclasses (e.g., structured control flow, acyclic CFGs) and implementing the product construction in Coq would be a novel contribution bridging 2-safety verification and verified compilation.

Bridging Abstract Leakage Models to Concrete Hardware. The compiler's leakage preservation theorem assumes the hardware is leakage-faithful: the actual timing is a deterministic function of abstract leakage labels. A formal connection between CompCert's abstract leakage model and a concrete hardware specification (e.g., a verified RISC-V core in Kami or Bluespec) would close the verification gap. Published at **MICRO** or **ASPLOS**, this would contribute the first end-to-end verified constant-time stack from C source to hardware gates.

Chapter 11

Universal Equality Saturation Framework

Can we design a universal e-graph framework parameterized by a rewrite theory such that the e-graph's performance (congruence closure, extraction) scales independently of the theory's complexity?

Equality saturation with e-graphs is a technique where, instead of applying rewrites one at a time, we build a compact representation of all equivalent programsthe e-graphand extract the best program according to a cost function. It has been applied to compiler optimization (egg, Herbie), MLIR optimization, quantum circuit optimization, hardware EDA (Nextmap), and program synthesis. Each domain uses a separate e-graph implementation with domain-specific rewrite rules, extraction heuristics, and performance optimizations. Building each from scratch is wasteful, and cross-domain optimizationfor example, optimizing across the entire hardware-software stackis impossible because no single e-graph can handle multiple theories simultaneously.

Problem 11.1 (Universal Equality Saturation)**.** Given a set of theories $\{T_1, \ldots, T_k\}$ where each $T_i = (\Sigma_i, R_i, C_i)$ consists of operators Σ_i, rewrite rules R_i, and a cost model C_i, and an initial term t expressed using operators from $\bigcup \Sigma_i$, find a term t^* equivalent to t under the equational theory $\bigcup R_i$ that minimizes a global cost function $C(t^*)$, using an e-graph whose performance (congruence closure, e-matching, extraction) scales independently of $|\Sigma_i|$ and $|R_i|$.

We propose a universal e-graph framework that is theory-agnostic (takes any rewrite theory as a parameter), efficient (performance scales independently of theory complexity), composable (multiple theories can coexist in the same e-graph and interact through shared terms), and extensible (adding a new theory takes minutes, not months).

11.1 Related Work

Approach	Strength	Gap
egg (Willsey et al.)	Fast, theory-agnostic e-graph library in Rust	Theory treated as black box; no cross-theory optimization
egglog (Zhang et al.)	E-graph + Datalog fixpoint	Still single-theory
Herbie (Panchekha et al.)	E-graph for floating-point optimization	Domain-specific (FP rewrites)
MLIR equality saturation	E-graph for MLIR dialect rewriting	MLIR-specific
Nextmap (EDA equality saturation)	Hardware expression rewriting	EDA-specific
Relational e-matching (Z3)	E-matching in SMT solvers	Not designed for saturation/extraction

No existing framework is simultaneously theory-agnostic, high-performance, and composable across theories.

11.2 Proposed Approach

Theory-Agnostic Performance Scaling

The key idea is that e-graph performance depends only on shallow theory properties (maximum operator arity, pattern depth, number of rewrites) rather than deep semantic properties. By designing a theory interface that exposes only these features, a universal framework can match domain-specific performance while enabling cross-theory composition through shared terms and global congruence closure.

We define a formal interface for e-graphs parameterized by a theory T consisting of operators Σ, rewrite rules R, and a cost model C. The implementation of congruence closure and e-matching is independent of T up to a pattern-matching engine that takes rewrite left-hand-side patterns as input. The central research question is whether we can characterize the dependence of e-graph performancematching time, congruence closure time, extraction timeon properties of the rewrite theory. If performance depends only on shallow properties such as maximum operator arity, a universal framework is feasible; if it depends on deep properties of the theory, domain-specific tuning may be unavoidable.

We build a library of benchmark theoriesarithmetic, Boolean, memory, quantum, hardwareand measure nodes created before saturation, e-matching time per round, congruence closure time, and extraction time for each. By correlating these with theory properties such as number of operators, number of rewrites, maximum pattern size, and operator treewidth, we identify which theory properties actually predict performance. For multi-theory composition, each term is tagged with its theory; operators from different theories can appear as subterms of each other, and rewrites apply only within their theory, but congruence closure is global across theories. The key challenge is scheduling: when adding a rewrite from one theory, the framework must decide whether to also consider rewrites from other theories, because a fixed-point iteration that applies all theories to

Algorithm 3 Universal Equality Saturation

Require: Term t, theories $\{T_1, \ldots, T_k\}$, cost limit
Ensure: Optimized term t^*

$E \leftarrow \textsc{InitEGraph}(t)$
repeat
  $E \leftarrow \textsc{Rebuild}(E)$ ▷ congruence closure
  for each theory T_i **do**
    $M \leftarrow \textsc{EMatch}(E, T_i.R)$ ▷ find rewrite matches
    $E \leftarrow \textsc{ApplyRewrites}(E, M)$
  end for
  $t^* \leftarrow \textsc{Extract}(E, \{C_i\})$ ▷ ILP extraction
until no new rewrites or time expired
return t^*

saturation may be prohibitively expensive. The framework must analyze theory interactionswhich theories share operatorsand schedule rewriting accordingly.

Extraction currently uses dynamic programming over the e-graph to minimize a simple cost model such as term size or constant latency. We extend extraction to handle non-linear costs (where the cost of a term depends on its subterms in non-linear ways), ML-predicted costs (where cost is predicted by a learned model), and multi-objective costs (simultaneously minimizing latency, area, and power and returning a Pareto frontier). This uses integer linear programming over the e-graph structure, where e-classes form constraints and the solver finds the minimum-cost term respecting all constraints. The hardest challenges are that theory interaction is unpredictabletwo individually efficient theories may create an enormous search space togetherand that ILP extraction is expensive for large e-graphs, requiring structural properties such as series-parallel structure or bounded treewidth to make it practical.

11.3 Evaluation

Benchmark	Metric	Baseline
egg benchmark suite	Time to saturation, nodes, quality	egg
MLIR optimization	Time, quality of optimized code	MLIR equality saturation
Quantum circuit optimization	Gate count reduction, time	Domain-specific quantum e-graph
Nextmap (EDA)	Area reduction, time	Nextmap
Cross-stack optimization	Combined cost reduction	Sequential application of theories

We implement the universal e-graph in Rust, extending egg's codebase, and evaluate against domain-specific e-graphs across four domains plus a cross-stack optimization case study combining arithmetic, memory, and RISC-V instruction selection theories. The key questions are whether the universal framework matches domain-specific performance and whether cross-theory optimization yields improvements beyond sequential application of individual theories. If successful, the framework extends naturally to incremental e-graphs maintained across compilation units for JIT compilation,

probabilistic e-graphs with weighted terms and expected-cost extraction, and verified e-graphs implemented in Coq or Lean with rewrite trace certification.

11.4 Research Directions

Performance Predictability of Theory-Agnostic E-Graphs. The central claim, that e-graph performance scales independently of theory complexity, must be supported by a rigorous empirical and theoretical characterization. An experimental study correlating theory properties (operator arity, number of rewrite rules, maximum pattern depth, operator treewidth) with e-graph metrics (nodes at saturation, e-matching time per round, congruence closure time) across a benchmark suite of 10+ theories would answer: which properties actually predict performance, and can we design a theory T that is intentionally adversarial? A **PLDI** paper combining this empirical study with a formal cost model for e-graph operations would be the first systematic analysis of e-graph scalability across diverse theories.

Multi-Theory E-Graph Scheduling. When multiple theories coexist in the same e-graph, the scheduling problem is critical: applying all theories to saturation at every round is prohibitively expensive. The framework must analyze theory interactions (shared operators, shared terms) and schedule rewriting accordingly: for instance, running the arithmetic theory to saturation first, then the memory theory on the resulting terms. A **OOPSLA** or **PLDI** paper contributing a theory-interaction analysis and a lazy scheduling algorithm that avoids redundant matching would make cross-theory optimization practical for the first time.

ILP-Based Extraction with Pareto-Optimal Multi-Objective Costs. Extraction currently minimizes a single scalar cost. Realistic use cases require simultaneous optimization of latency, area, and power, yielding a Pareto frontier of optimal programs. Formulating extraction as an integer linear program over e-class constraints with multiple objectives (minimizing $\sum w_i \cdot c_i$ for each term) and solving via branch-and-cut or weighted-sum scalarization would enable true cross-stack optimization. A **CGO** or **DAC** paper demonstrating Pareto-optimal extraction for a combined arithmetic+memory+RISC-V theory, with the solver returning the full frontier for an MLIR function, would show that multi-objective extraction is practical for realistic program sizes.

Verified Equality Saturation with Rewrite Certification. Equality saturation is widely used but lacks correctness guarantees: a buggy rewrite rule can silently introduce errors. A verified e-graph in Coq or Lean that certifies each rewrite application, producing a proof trace that the extracted term is equivalent to the original under the theory's axioms, would be the first verified e-graph. Published at **POPL** or **CPP**, this would contribute formal foundations for e-graphs (congruence closure, e-matching correctness) and a practical extraction verification framework.

Chapter 12

Automatic Synthesis of Optimal Synchronization for Data-Parallel Programs

Given an unordered or irregular data-parallel computation, can a compiler automatically synthesize the optimal synchronization strategy, including the placement of individual synchronization operations?

Parallelizing irregular computationsgraph analytics, sparse linear algebra, unordered set processing, priority-queue-based algorithmsrequires synchronization to prevent data races and maintain invariants across concurrent threads. The programmer chooses from fine-grained locking (lock per element, maximizing concurrency with high overhead), coarse-grained locking (lock the entire data structure, low overhead but poor concurrency), lock-free approaches using CAS or other atomic primitives, transactional memory (HTM or STM), and barriers. Choosing the wrong strategy can mean orders of magnitude performance difference, and the optimal strategy depends on the input data characteristics, the hardware, and the algorithm itself. What is optimal for a scale-free graph is suboptimal for a grid graph, and manual tuning for each combination of algorithm, platform, and input type requires expert effort.

Problem 12.1 (Optimal Synchronization Synthesis)**.** Given an irregular parallel program P with operations $\{op_1, \ldots, op_n\}$, a set of synchronization strategies $\mathcal{S} = \{\text{fine-grained lock}, \text{coarse-grained lock}, \text{CAS}, \text{HTM}\}$, a target platform H with known costs (CAS latency c_{cas}, lock acquire c_{lock}, TM abort c_{abort}), and an expected input distribution $\mathcal{D}$, find a strategy assignment $s : P \to \mathcal{S}$ that minimizes $\mathbb{E}_{I \sim \mathcal{D}}[\text{execution\_time}(P, s, I)]$ subject to race-freedom and deadlock-freedom.

We propose a compiler that automatically analyzes an irregular parallel algorithm and synthesizes the optimal synchronization strategy for a given hardware platform and expected input characteristics.

12.1 Related Work

Approach	Strength	Gap
LLP-FW (Generic lock-free runtime for lattice-linear problems)	Automatically derives lock-free algorithms	Only lattice-linear predicates; does not choose between lock-free, locking, or TM
Galois / Ligra (Irregular parallel programming frameworks)	High-level abstractions for irregular parallelism	Programmer chooses synchronization strategy; no automatic selection
PBBS (Problem-based benchmark suite)	Implements parallel algorithms with hand-tuned sync	No automation
Jikes RVM / Maxine (Adaptive synchronization in JVMs)	At runtime switch between strategies	Java-specific; runtime only; limited strategy space
Graal / Truffle (Partial escape analysis, lock elision)	Compiler removes unnecessary synchronization	Removes, does not choose strategy
AutoTM (Compiler-guided TM selection)	Chooses HTM vs. STM for transactions	Transaction boundaries are programmer-specified
RACEZ / ThreadSanitizer (Dynamic race detection)	Finds races	Finds, does not fix or prevent

No compiler automatically synthesizes a synchronization strategy for irregular parallel programs.

12.2 Proposed Approach

[

Probabilistic Conflict Analysis] The key idea is to treat imprecision from irregular access patterns as a probability rather than a worst-case assumption. Instead of sound but hopelessly conservative over-approximations, we compute $P(\text{conflict})$ from degree distributions (for graphs) or sampling profiles, enabling a cost model that balances synchronization overhead against expected contention cost.

We design a DSL for specifying synchronization strategies based on guarded operations: each operationread, write, updatecan be guarded by a synchronization condition. The DSL expresses fine-grained locking, coarse-grained locking, CAS-based strategies, and compositions of multiple mechanisms. The central research question is whether we can statically analyze the conflict structure of an irregular parallel programwhich operations on which data objects may conflictwith sufficient precision to distinguish low-contention from high-contention scenarios. For irregular programs, this is harder than for regular programs because data access patterns are input-dependent.

We design a static analysis combining points-to analysis, alias analysis, and conflict classification that labels each pair of operations as always conflicting, sometimes conflicting, or never conflicting. For the sometimes-conflict cases, we estimate the probability of conflict given an input distribution using a lightweight probabilistic analysis: for example, in a scale-free graph with one million vertices

Algorithm 4 Synchronization Strategy Synthesis

Require: Program P, platform H, input distribution $\mathcal{D}$
Ensure: Strategy $s^* : \text{ops} \to \mathcal{S}$

$G \leftarrow \textsc{BuildConflictGraph}(P)$ ▷ static analysis
Pr[conflict] $\leftarrow \textsc{EstimateConflictProb}(G, \mathcal{D})$
best $\leftarrow \infty$
for each strategy $s \in \mathcal{S}$ **do**
    $\text{cost}(s) \leftarrow \textsc{SyncOverhead}(s) + \text{Pr[conflict]} \cdot \textsc{ContentionCost}(s, H)$
    if $\text{cost}(s) < \text{best}$ **then**
        best $\leftarrow \text{cost}(s)$; $s^* \leftarrow s$
    end if
end for
return s^*

and degree exponent 2.3, two random vertex accesses conflict with probability approximately one over the number of vertices. This builds on existing pointer analysis for irregular programs, statistical profiling of representative inputs, and abstract interpretation with interval bounds on access frequencies.

Strategy selection is formulated as a cost optimization. For each strategy, we compute the sum of synchronization overhead and contention cost. Synchronization overhead captures the per-operation cost of acquiring and releasing locks, performing CAS, or executing transactions. Contention cost multiplies the probability of conflict for each conflicting pair by the latency of conflict resolution under that strategy. The cost model is instantiated per target platform with CAS cost including retry overhead, lock cost including cache coherence traffic, TM cost with abort probability, and barrier cost with thread imbalance. The optimization is solved by exhaustive enumeration for small strategy spaces, constraint programming for larger spaces, or ML-based prediction to learn the optimal strategy from training data mapping input-algorithm-platform triples to optimal strategies.

The hardest challenges are that the probabilistic conflict analysis is approximate and may lead to selecting the wrong strategy, and that hardware TM abort probabilities depend on cache capacity, coherence traffic, and transaction sizeall hard to predict statically. The cost model must be robust to estimation error, selecting the safer strategy when predicted costs are close. For verification, we adopt a sound-by-construction approach: the DSL is designed so that any valid strategy is race-free and deadlock-free by construction, reducing verification to checking that the DSL program is well-typed.

12.3 Evaluation

Benchmark	Platform	Metric	Comparison
BFS (Galois, Ligra)	Intel Xeon, AMD EPYC	Speedup, scalability	Hand-tuned Galois, global lock
SSSP (delta-stepping)	Intel Xeon, BOOM (RISC-V)	Speedup, sync overhead	PBBS SSSP
PageRank	Intel Xeon	Iterations/sec	Ligra, naive
SpMV	Intel Xeon, AMD EPYC	Throughput	OSKI, hand-tuned
Hash join	Intel Xeon	Throughput, memory	Hand-tuned parallel join

We implement as a source-to-source compiler that takes an irregular C++ program with annotations, analyzes conflicts, selects a strategy, and emits optimized C++ with synchronization. Evaluation uses graph algorithms (BFS, SSSP, PageRank, connected components, triangle counting), sparse linear algebra (SpMV, SpGEMM, Kruskal's), and database operators (parallel hash join, parallel aggregation) across Intel Xeon, AMD EPYC, and RISC-V BOOM platforms. The key question is how often the synthesized strategy matches the best hand-optimized strategy and what the performance gap is when it does not. If successful, the framework extends to composable per-phase strategies, learned synchronization policies replacing the analytical cost model with one trained on profiling data, and distributed-memory parallelism with communication pattern selection.

12.4 Research Directions

Probabilistic Conflict Analysis for Irregular Programs. The proposed static conflict analysis must handle input-dependent access patterns. A combined approach using points-to analysis for must/may-alias classification and lightweight probabilistic reasoning—estimating $P(\text{conflict})$ from degree distributions for graph algorithms—would be the first static analysis designed specifically for synchronization synthesis. The novelty is treating imprecision from irregular accesses as a probability rather than a worst-case assumption. A **PLDI** or **POPL** paper presenting the analysis and proving its soundness (over-approximation of conflict probability) would establish the formal foundations for compiler-driven synchronization synthesis.

Cost-Model-Driven Synthesis with Robustness Guarantees. The strategy selection cost model requires synchronization overhead and contention cost estimates. The key challenge is robustness: when the model predicts two strategies have similar costs, it should select the safer one. Formalizing this as a robust optimization problem—minimize expected cost while bounding worst-case regret—would make the synthesis practical for production use. A **PPoPP** or **CGO** paper demonstrating that the synthesized strategy matches or beats hand-tuned synchronization on graph analytics (BFS, SSSP, PageRank) across Intel Xeon, AMD EPYC, and RISC-V BOOM platforms would be a compelling contribution.

Learned Synchronization Policy Selection. Instead of an analytical cost model with hard-to-estimate parameters, a learned predictor could map input-algorithm-platform triples directly to

optimal synchronization strategies. Training on a corpus of 1,000+ (graph, algorithm, platform) combinations—with features like edge density, degree distribution quantiles, algorithm type, and cache size—would make an excellent **CGO** or **MLSys** paper. The expected contribution is the first learned synchronization synthesizer that generalizes to unseen inputs without recompilation.

Sound-by-Construction DSL for Synchronization Strategies. The proposed DSL for guarded operations should guarantee race-freedom and deadlock-freedom by construction: if the DSL program type-checks, the emitted synchronization is correct. Designing the type system—with kinds for lock level, atomicity regions, and conflict modes—and proving the soundness theorem in a proof assistant would address the verification challenge at the source. A **POPL** paper presenting the DSL, its type system, and the soundness proof would be a significant contribution to both programming language design and parallel computing.

Chapter 13

Compiler-Directed Register File Banking for Embedded RISC-V

Can a compiler automatically allocate variables to register banks to minimize bank activation energy without increasing spills?

Definition 13.1 (Register File Banking)**.** *Register file banking* partitions the physical register file into multiple independently activated banks. Only the banks needed for the currently executing instruction are powered; unused banks remain in a low-power idle state, saving both dynamic and static energy.

Definition 13.2 (Bank Activation Energy)**.** *Bank activation energy* is the total energy consumed by all banks that must remain powered at a given program point. Minimizing the number of simultaneously active banks directly reduces the dynamic energy footprint of register accesses and is the central optimization objective of bank-aware register allocation.

Embedded RISC-V cores such as the `CV32E40X`, `X-HEEP`, and `PULP` employ register file banking to reduce energy consumption: the physical register file is partitioned into multiple banks, and only the banks needed for the currently executing instruction are activated. Unused banks remain in low-power mode, saving dynamic and static energy. However, current compilers, LLVM and GCC, are oblivious to register banking. They allocate variables to physical registers using traditional graph-coloring algorithms that minimize spill code but treat all registers as equal. This scatters a function's live registers across many banks, forcing all of them to be active simultaneously, wastes energy on banks that contain only a few live values, and misses consolidation opportunities where live ranges that could be merged into the same register are allocated to different registers.

We propose a bank-aware register allocator that assigns variables to registers to minimize the number of active banks at any program point, thereby minimizing energy, without increasing spill count.

13.1 Related Work

Approach	Strength	Gap
LLVM greedy RA (graph coloring)	Good spill minimization	Bank-unaware; scatters registers across banks
TBAA / bank-aware RA (Paci et al., 2000s)	Early work on register banking for DSPs	DSP-specific; not evaluated on modern RISC-V
Chaitin-style RA with bank conflicts (Canadilla)	Adds bank conflict edges to interference graph	Treats bank conflicts as constraints, not optimization
VLIW bank allocation (Jacome, 2000s)	Bank assignment for VLIW functional units	VLIW-specific; does not generalize to scalar RISC-V
ILS RA (integrated, not bank-aware)	ILP formulations for optimal RA	No bank cost in objective
Hardware register file banking (Patel et al., RISC-V)	Proposed banking microarchitecture	No compiler support proposed

No existing work formulates register allocation with bank activation energy as an explicit objective while keeping spill minimization as a hard constraint.

13.2 Proposed Approach

We formulate bank-conscious register allocation (`BC-RA`) as a constrained optimization problem. The interference graph $G = (V, E)$ and the set of physical registers R partitioned into banks $B_1 \dots B_k$ are given. The assignment $\phi : V \to R$ maps each virtual register to one physical register. The objective is to minimize the sum over all program points of the number of banks containing at least one live virtual at that point, subject to the constraint that interfering virtuals do not share a physical register. This is a graph coloring problem with an additional objective over the color partition: find any coloring with at most $|R|$ colors, then minimize the bank activation among such colorings. The central research question is whether we can formulate this precisely enough to guide allocation and whether the energy model, i.e., the relationship between active banks and energy consumed, is linear enough for practical optimization.

We encode `BC-RA` as an integer linear program with three sets of constraints:

1. Each virtual gets exactly one register.

2. No two interfering virtuals share a register.

3. A bank is marked active at a program point if any virtual live at that point is assigned to a register in that bank.

The ILP can be solved by `Gurobi` or `CPLEX` for small functions to establish optimal baselines.

Design Principle: Energy–Spill Pareto Frontier

Bank-aware register allocation must never increase spill count beyond the baseline graph-coloring allocation. Any optimization that increases spills will likely degrade overall system energy more than it saves, because a spill inserts two memory operations (store + load) that each consume orders of magnitude more energy than a register access. The allocator should therefore explore only allocations on or near the Pareto frontier of the energy–spill trade-off.

For larger functions, we design a heuristic: start with the standard greedy `Chaitin-Briggs` coloring for a baseline allocation, then iteratively move virtuals between registers to consolidate them into fewer banks provided the move does not increase spills. If the function still has high bank activation, we consider spilling a virtual to reduce bank count when the memory energy cost is less than the bank activation saving.

The hardest challenges are that bank activation energy may be too small relative to base register access energy to justify the effort, and that the heuristic may be trapped in a poor local minimum if the initial greedy coloring scatters registers randomly. We evaluate across different bank geometries, i.e., one bank of thirty-two registers (no banking), two banks of sixteen, four banks of eight, eight banks of four, and sixteen banks of two, to derive a bankability score that predicts, from a function's interference graph, the energy savings achievable with bank-aware RA. This guides hardware design decisions about optimal bank count for a given workload.

13.3 Evaluation

Benchmark	**Configuration**	**Metric**
`Embench` (all 19 benchmarks)	`CV32E40X`, 4×8 bank config	Energy/savings, spill count change, cycle count change
`MiBench` (automotive, consumer)	`CV32E40X`, 4×8	Same
`CoreMark`	`CV32E40X`, 4×8	CoreMark/MHz, energy
DSP kernels (FFT, FIR, biquad)	2×16, 4×8, 8×4 banks	Energy savings per configuration
Small functions (exact solve)	All configs	Quality gap: heuristic vs. exact ILP

We modify LLVM's RISC-V backend register allocator to support `BC-RA`, adding the bank activation cost model and the bank consolidation heuristic. Evaluation uses `Embench` (all nineteen IoT-level benchmarks), `MiBench` (mid-range embedded), `CoreMark`, and custom DSP kernels. Comparison targets include LLVM `-O3`, GCC `-O3`, and exact ILP solutions for small functions. The key question is the average energy savings of bank-aware RA over standard RA; if greater than ten percent, the contribution is significant for embedded systems. If successful, the approach extends to bank-aware instruction selection (choosing instructions that use fewer distinct banks), bank-aware scheduling (reordering to keep the same bank active across consecutive cycles), and integrated bank-aware spilling that spills the virtual contributing most to bank activation rather than the one with the lowest spill cost.

13.4 Research Directions

Integer Linear Programming Formulation for Bank-Aware RA. A precise ILP formulation for bank-conscious register allocation (`BC-RA`), with variables $\phi(v, r)$ indicating virtual v assigned to physical register r, a constraint that interfering virtuals do not share a register, and an objective minimizing $\sum_{p \in P} \sum_{b \in B} \text{active}(b, p)$ where $\text{active}(b, p)$ is 1 if any live virtual at program point p maps to a register in bank b, would provide optimal baselines and reveal the structure of the optimization. Solving with `Gurobi` for small functions (≤ 50 virtuals) and comparing to the heuristic would quantify the optimality gap. A **CGO** or **LCTES** paper establishing this formulation and optimal baselines would be the first rigorous treatment of bank-aware register allocation.

Iterative Bank Consolidation Heuristic. The proposed heuristic, starting from a `Chaitin-Briggs` coloring and iteratively moving virtuals between registers to consolidate banks, must be evaluated across diverse bank geometries (1×32, 2×16, 4×8, 8×4, 16×2) and benchmark suites (`Embench`, `MiBench`, `CoreMark`). The key result is average energy savings; if savings exceed 10% for realistic geometries (4×8 banks) without increasing spills, the contribution is significant for embedded systems. A **PLDI** or **CGO** paper reporting these results, including the energy model validated against a RISC-V core simulation (`CV32E40X`), would demonstrate practical impact.

Bankability Prediction for Hardware-Software Co-Design. The compiler can compute a function's bankability score, predicting achievable energy savings from bank-aware RA, from its interference graph properties (register pressure, live range lengths, degree distribution). This score guides hardware design decisions: if the workloads of interest have low bankability, a single-bank register file may be sufficient; if they have high bankability, multiple small banks are beneficial. Published at **MICRO** or **ASPLOS**, this would be the first hardware-software co-design study using compiler analysis to guide register file banking decisions for embedded RISC-V cores.

Integrated Bank-Aware ISEL and Scheduling. Instruction selection can prefer instruction variants that use fewer distinct register banks, and instruction scheduling can reorder operations to keep the same bank active across consecutive cycles. Integrating these with bank-aware RA into a single bank-optimizing code generation pass would maximize energy savings. A **LCTES** or **ACM TECS** paper combining `BC-RA`, bank-aware ISEL, and bank-aware scheduling, evaluated on the `CV32E40X` or `X-HEEP` cores with RTL-level energy measurement, would provide a complete solution for energy-efficient code generation on banked register files.

Part III

Embedded Systems

Chapter 14

Formal End-to-End Timing Guarantees for Networked Cyber-Physical Systems

Can we extend mixed-criticality scheduling theory to provide end-to-end timing guarantees across a network of cyber-physical systems with shared memory and heterogeneous processors?

Mixed-criticality (MC) scheduling theory provides timing guarantees for systems where tasks of different criticality levels share the same processor: high-criticality tasks must meet their deadlines even in overload conditions, while low-criticality tasks may be dropped or delayed. This theory is well-developed for single-core and simple multicore systems, but real cyber-physical systemsautomotive, avionics, industrial control, roboticsare networked: multiple electronic control units connected by time-sensitive networks such as TSN, CAN-FD, and FlexRay exchange data and coordinate actions. A task on one ECU sends a message to another ECU, which triggers a task whose output controls an actuator. The end-to-end timing guarantee involves WCET on the first ECU, network latency on the bus, WCET on the second ECU, and shared memory interactions between both sides. Current practice composes worst-case estimates additively, double-counting worst cases and producing bounds two to ten times more pessimistic than necessary.

We propose a formal framework that composes per-ECU mixed-criticality scheduling analysis with network timing analysis and shared-memory interaction models to provide end-to-end timing guarantees for networked cyber-physical systems.

14.1 Related Work

Approach	Strength	Gap
AnTi-MiCS / MulTi-MiCS (MC scheduling)	Good single-core MC theory	No multi-node or network
DDE (Deterministic Dynamic Execution)	Timing-anomaly-free single-core scheduling	Single-core only
DDF (Deterministic Data Flow)	Cause-effect chain analysis for multi-task	Single-node; assumes fixed task graph
TSN guarantee (802.1Qbv, Qci, CBW)	Upper bounds on network latency	Network-only; does not compose with compute
Holistic scheduling (Tindell, Burns, 1990s)	End-to-end analysis for distributed systems	Pessimistic; does not handle MC or shared memory
Time-triggered architectures (TTEthernet, TTP)	Deterministic network schedule	Rigid; not designed for event-triggered MC
Real-time calculus / Network calculus	Compositional performance bounds	Requires detailed arrival/service curves; may be overly pessimistic

No framework composes MC scheduling with network timing and shared-memory interactions to provide end-to-end timing guarantees for networked systems.

14.2 Proposed Approach

We define a formal model.

Definition 14.1 (Networked CPS Model)**.** A system is a tuple $M = (N, T, \mathcal{M}, R, D)$ where N is the set of nodes with processor type, clock speed, and local memory; T is the set of all tasks across all nodes with WCET per criticality level, period or deadline, and criticality level; $\mathcal{M}$ is the set of network messages with size, source and destination nodes, period or deadline, and priority; R is the set of shared memory regions with access protocol, physical location, and list of accessors; and D is the set of data dependencies, i.e., cause-effect chains forming end-to-end paths.

The central research question is whether we can derive end-to-end timing bounds by composing per-ECU timing analysis with network timing analysis while keeping pessimism bounded.

We extend single-ECU MC analysis to account for network interference. A task's WCET depends not only on local MC scheduling but also on network message reception interrupts that preempt the task, shared memory contention from waiting for remote access to complete, and incoming data availabilitya task on one node cannot start until the message from another arrives. We model the network interface as a resource with bounded interference, characterizing the maximum interference a task can experience from network operations in cycles. Conversely, we extend TSN and CAN timing analysis to account for end-system behavior. If a receiving ECU is overloaded in high-criticality mode, it may process messages more slowly, increase network buffer occupancy, fail to send acknowledgments causing retransmissions, or delay message consumption freeing network

buffers late. We model the end-system as a network traffic shaper with worst-case behavior, computing the worst-case message latency given the worst-case compute delay at each end.

The centerpiece is a composition theorem.

Theorem 14.1 (End-to-End Timing Composition)**.** *Let $C = (N_1, \ldots, N_k)$ be a cause-effect chain with k nodes connected by a network Net. Suppose each node N_i satisfies its local MC timing guarantee with worst-case response time R_i, and the network satisfies per-message worst-case latency Δ_i under the maximum backpressure from compute nodes. Then the end-to-end worst-case latency for C is bounded by*

$$D_{\textit{end-to-end}} \leq \sum_{i=1}^{k} R_i + \sum_{i=1}^{k-1} \Delta_i - \sum_{i=1}^{k-1} \gamma_i,$$

where γ_i is a concurrency factor bounding the overlap between computation and network delay.

Proof sketch. Proceed by induction on the chain length k. For $k = 1$, the bound reduces to R_1, which holds by the local MC guarantee. For $k > 1$, decompose the end-to-end delay as $D_{1..k} = D_{1..k-1} + \Delta_{k-1} + R_k - \gamma_{k-1}$, where γ_{k-1} accounts for the probability that node N_{k-1}'s output transmission and node N_k's input processing overlap. The induction hypothesis bounds $D_{1..k-1}$, and the concurrency factor γ_{k-1} is bounded using a fixed-point iteration over shared-memory circular dependencies. □

The key insight is that composition is not simply additive: the theorem identifies interference windows where worst-case behavior may align and shows these windows are bounded. The proof is inductive over the cause-effect chain, where the worst-case delay at each step is bounded by the worst-case delay of the previous step plus per-node and per-network delay, minus a concurrency factor reflecting that overlapping compute and network delay do not both hit their worst case simultaneously.

The hardest challenges are that the composition theorem may be too pessimistic if worst-case behavior aligns across nodes and network, and that shared memory interactions create circular dependenciesthe worst-case waiting time depends on the lock holder's WCET, which depends on local MC scheduling, which depends on network interference. The model resolves these circularities through fixed-point iteration. Network interference models are inherently imprecise, and overly conservative models may make end-to-end bounds useless, suggesting that probabilistic network models may be needed for some scenarios.

Core Thesis

End-to-end timing guarantees for networked cyber-physical systems can be derived by composing per-ECU mixed-criticality scheduling analysis with network timing analysis using a composition theorem that avoids additive pessimism through the identification of bounded interference windows and concurrency factors.

14.3 Evaluation

Benchmark	Configuration	Metric
Bosch engine mgmt	3 ECUs, CAN bus, 20+ tasks	End-to-end latency bound, pessimism vs. additive
ADAS-inspired	5 ECUs (camera, radar, fusion, planning, actuation), TSN	Same
IMA (avionics)	4 ECUs, AFDX network	Same
Robot arm control	6 ECUs (joint controllers + planner), shared memory	Same, plus shared memory impact
Synthetic	10–50 ECUs, random task sets	Scalability: computation time vs. system size

We build a tool that takes the system model, computes per-ECU timing bounds using standard MC analysis extended with network interference, computes per-network timing bounds using TSN or CAN analysis extended with compute backpressure, and computes end-to-end bounds using the composition theorem. Evaluation uses the Bosch engine management benchmark, an ADAS-inspired benchmark with camera, radar, fusion, planning, and actuation ECUs, the DEOS avionics benchmark, a robot arm controller, and synthetic random task sets. Comparison targets include pessimistic additive composition, holistic scheduling without MC, and real-time calculus bounds. The target is less than thirty percent pessimismthe bound at most 1.3 times the actual worst-case latencyfor typical configurations. If successful, the framework extends to multi-network paths, fault-tolerant guarantees with bounded fault tolerance, optimization of task-to-ECU mapping and priority assignment, and probabilistic end-to-end bounds for best-effort networks.

14.4 Research Directions

Probabilistic end-to-end timing for best-effort networks. Stochastic network calculus models message latency as a random variable with a known distribution, while MC scheduling analysis bounds deadline miss probability per criticality level. Composing these two frameworks yields a probabilistic end-to-end guarantee: for a cause-effect chain with n nodes, the probability that its end-to-end latency exceeds $\mathcal{D}$ is $\Pr\left(\sum_{i=1}^{n} L_i + \sum_{i=1}^{n-1} \Delta_i > \mathcal{D}\right) \leq \epsilon$, where L_i is the per-node response time and Δ_i is the network delay. This would target *ACM Trans. on Embedded Computing Systems* and provides the first probabilistic end-to-end analysis for mixed-criticality distributed systems.

Shared-memory circular dependency resolution. When two ECUs share a memory region accessed under a lock, the WCET of a task on ECU_A depends on the hold time of ECU_B, which in turn depends on the MC mode of ECU_B. This circular dependency can be resolved through fixed-point iteration $W^{k+1} = f(W^k)$, where f composes the per-ECU and per-network worst-case delays, with convergence guaranteed when f is a contraction mapping. A publication in *IEEE Real-Time Systems Symposium* would demonstrate the first convergence proof for compositional MC timing with shared memory.

Task-to-ECU mapping optimization. Given a set of tasks, network topology, and end-to-end deadlines, assigning tasks to ECUs and priorities to messages to minimize worst-case end-to-end latency is NP-hard. A mixed-integer programming formulation with objective $\min \max_{c \in \mathcal{C}} \left(\sum_{\tau \in c} R_\tau + \sum_{\mu \in c} \Delta_\mu \right)$, solvable via branch-and-cut for systems with up to 50 ECUs, would appear in *IEEE Trans. on Computer-Aided Design* and establish the first optimization framework for MC distributed real-time systems.

Fault-tolerant end-to-end guarantees. Extend the composition theorem to handle bounded faults: message loss on CAN-FD (recoverable via retransmission with bounded retry count), transient node failure (recoverable via watchdog and state resync), and permanent node failure (degradation to a fail-safe subset of cause-effect chains). The fault-tolerant bound is $D_{\mathrm{FT}} = D + \sum_{f \in \mathcal{F}} \delta_f$, where δ_f is the worst-case recovery latency for fault type f. Targeting *IEEE Trans. on Dependable and Secure Computing*, this would produce the first end-to-end timing analysis that also bounds fault recovery delay.

Chapter 15

Precise WCET Through Exact Microarchitectural Models

What is the Pareto frontier between WCET estimation accuracy and microarchitectural model precision, and can we compute exact WCET for small but critical functions using model checking or SAT?

Worst-case execution time (WCET) estimation is fundamental to real-time systems: if the WCET is underestimated, the system may miss deadlines; if it is overestimated, the system is over-provisioned, wasting cost, power, and performance. Current WCET tools use abstract interpretation over simplified microarchitectural modelsset-associative caches with LRU replacement and hit-miss classification, in-order pipelines with fixed-latency stages, and speculation modeled only as extra latency per branch. These models are sound but imprecise, with the gap between estimated and true WCET reaching two to ten times for complex processors. The imprecision comes from abstracting away microarchitectural details such as bank conflicts, MSHR contention, prefetching, and speculative wake-up, from conservative handling of timing anomalies where a faster local decision leads to a slower global outcome, and from an inability to model complex interactions between pipeline, cache, and speculation.

Problem 15.1 (Exact WCET via SAT)**.** Given a program binary B and a microarchitectural transition system $M = (S, s_0, \delta)$ where S is the set of microarchitectural states (pipeline registers, cache state, branch predictor state, store buffer), s_0 is the initial state, and $\delta : S \to S$ is the deterministic transition function for a fixed input, find the length $K^* = \max\{K \mid \exists \text{ execution path of length } K \text{ from } s_0 \text{ to a terminal state}\}$, encoded as a MaxSAT instance over K unrolled steps.

We propose to compute exact WCET for small but critical functions by modeling the microarchitecture exactly as a transition system and using SAT or model checking to find the longest execution path.

15.1 Related Work

Approach	Strength	Gap
aiT (AbsInt)	Industrial WCET tool, sound abstract interpretation	Pessimistic; abstract models; no exact computation
OTAWA (Open-source WCET)	Similar to aiT, open framework	Same limitations
PLRU / LRU cache analysis (Grund, Reineke)	Precise cache analysis for simple policies	Abstract models only; no exact path enumeration
Timing anomaly analysis (Reineke, Sen)	Characterization of timing anomalies in pipelines	Describes, does not solve for exact WCET
Exact WCET via model checking (Dalsgaard et al., 2010)	Small examples with UPPAAL model checking	Limited to tiny programs due to state explosion
SAT-based longest path (Burguière, 2006)	SAT for pipelined processors	Only covers pipeline; no cache model
ILP-based WCET (Li, Malik, 1990s)	ILP for pipeline + cache constraints	Simplified cache model; NP-hard for large programs
SMT for WCET (Gray, 2015)	SMT for in-order pipeline + direct-mapped cache	No complex speculation

No existing work provides a systematic study of the tractability boundary for exact WCET computation as a function of microarchitectural complexity and program size.

15.2 Proposed Approach

SAT Encoding of Microarchitectural State

The key idea is to encode the entire microarchitectural transition system (pipeline, caches, branch predictor) as a SAT instance by unrolling the program for K steps. The exact WCET is found via binary search on K using incremental SAT, with symbolic cache state representation to avoid explicit enumeration of 2^{32} possible cache configurations.

We define an exact transition system for a simple RISC-V corethe CVA6 or Ariane six-stage in-order pipeline. The state includes pipeline registers (PC, instruction, decoded operands, ALU result, memory result, write-back value for each stage), the architectural register file, the L1 instruction cache with tags, valid bits, and LRU state per set, the L1 data cache with MSHRs, the store buffer, the branch predictor with two-bit saturating counter table, BTB, and RAS, and control logic for stall signals, flush signals, and hazard detection. The transition function is deterministic for a fixed input, so the exact WCET is the length of the deterministic path from initial state to end state.

We encode the transition system and program as a SAT instance by unrolling the program for K steps, encoding state bits at each step, encoding transition constraints between consecutive states, encoding the initial state, and encoding termination at step K. The objective is to find the maximum K such that a valid execution path existsa maximum satisfiability (MaxSAT) problem.

Algorithm 5 SAT-Based Exact WCET

Require: Binary B, microarchitecture M, bound $K_{\max}$
Ensure: Exact WCET K^* or timeout
$\Phi_0 \leftarrow \textsc{EncodeInitialState}(s_0)$
$K_{\text{low}} \leftarrow 0,\ K_{\text{high}} \leftarrow K_{\max}$
while $K_{\text{low}} < K_{\text{high}}$ **do**
    $K \leftarrow \lfloor (K_{\text{low}} + K_{\text{high}} + 1)/2 \rfloor$
    $\Phi_K \leftarrow \Phi_0 \wedge \bigwedge_{i=1}^{K} \textsc{EncodeTransition}(\delta, i)$
    $\Phi_K \leftarrow \Phi_K \wedge \textsc{EncodeTerminal}(K)$
    if $\textsc{MaxSAT}(\Phi_K)$ is satisfiable **then**
        $K_{\text{low}} \leftarrow K$
    else
        $K_{\text{high}} \leftarrow K - 1$
    end if
end while
return K_{low}

The central research question is for which programs this is tractable. We hypothesize that for functions with at most 500 instructions, bounded loops of at most 100 iterations, and no unbounded recursion, exact WCET is tractable with careful model engineering.

We run the exact solver on a large set of programs from the Embench and MiBench suites, measuring for each function the number of instructions, number of branches, maximum loop nesting depth, maximum loop iteration count, number of function calls, and cache footprint. We classify functions as green (solvable within one hour), yellow (solvable but more than one hour), and red (unsolvable within 24 hours), identifying the decision boundary in the parameter space. For large functions, we decompose them into tractable fragmentsextracting each loop body and each basic block as an independent fragment, computing exact WCET per fragment, and composing fragment WCETs through flow analysis using the implicit path enumeration technique. The composition is pessimistic because fragments are analyzed independently, and the pessimism factor depends on cross-fragment microarchitectural state interactions such as cache and predictor state.

The hardest challenges are state explosion for the cachea 16KB four-way set-associative L1 cache has approximately 32K possible states, and for a 500-instruction function the cache state space may be $\mathcal{O}(2^{32})$and the expense of binary search for WCET, where if the abstract bound is 100,000 cycles and the true WCET is 50,000, roughly 17 SAT iterations are needed. Incremental SAT to reuse constraints across iterations and symbolic cache state representation using the SAT solver's built-in equivalence reasoning are essential to make this practical.

15.3 Evaluation

Benchmark	Metric	Comparison
Embench	% functions solvable exactly	Timelimit: 1 hour
MiBench	Exact WCET vs. abstract WCET (gap ratio)	OTAWA + aiT
PapaBench loop	Exact WCET for each loop	Abstract WCET, measured (FPGA)
Microbenchmarks	Solver time vs. program size	Charts
FPGA measurement	Executed cycles on FPGA vs. predicted by ExACT	Ground truth

We build the ExACT tool (Exact Analysis of Cycle Time) with a frontend accepting RISC-V ELF binaries, a model generator encoding the CVA6 microarchitecture as a transition system, and a parallel solver portfolio using ABC (IC3/PDR), MiniSAT, and Z3. Evaluation uses Embench (all 19 benchmarks), MiBench, PapaBench (drone autopilot), and custom microbenchmarks targeting specific microarchitectural features. Validation against FPGA measurements of the CVA6 core provides ground truth. The key question is the percentage of real-time critical functions for which exact WCET is tractableif more than fifty percent, the thesis is a strong contribution. If successful, the approach extends to out-of-order cores with register renaming and reorder buffers (likely tractable only for very small functions), probabilistic WCET for cores with non-deterministic replacement policies, and multicore interference analysis considering all possible interleavings.

15.4 Research Directions

Compositional exact WCET via fragment decomposition. For functions too large for monolithic SAT solving, decompose into tractable fragments (loop bodies, basic blocks), compute exact WCET per fragment, and compose fragment WCETs through flow analysis using the implicit path enumeration technique. The pessimism factor $\rho = \frac{\sum_f W_f^{\text{exact}} - W_{\text{global}}^{\text{exact}}}{W_{\text{global}}^{\text{exact}}}$ captures the cost of cross-fragment microarchitectural state loss. A paper in *IEEE Real-Time Systems Symposium* would characterize ρ empirically across Embench and establish the first systematic composition framework for exact WCET.

SAT/SMT optimizations for cache state explosion. The L1 cache state space for a k-way set-associative cache grows as $O((2^B)^k)$ per set, where B is the block size. Symbolic cache state representation using the SAT solver's built-in equivalence reasoning (e.g., uninterpreted functions for tag matching) combined with incremental SAT across WCET binary search iterations can reduce solve time by orders of magnitude. Targeting *IEEE Trans. on Computer-Aided Design*, this work would introduce the first symbolic encoding of set-associative cache state for exact WCET.

Exact WCET for out-of-order cores. Out-of-order execution with register renaming, a reorder buffer of size R, and speculative wake-up creates a state space exponential in R. For functions of at most 100 instructions, a model checking approach using IC3/PDR with clause generalization over reorder buffer state can tractably compute exact WCET. A publication in *ACM Trans. on*

Embedded Computing Systems would define the tractability boundary for OOO cores and provide the first exact WCET results for an out-of-order processor.

Probabilistic WCET for non-deterministic hardware. Cores with random-replacement caches, fuzzy branch predictors, or randomized prefetchers admit no single exact WCET. Define the exact probability distribution over execution cycles as $\Pr(W = w) = \sum_{s \in \mathcal{S}} \mathbf{1}_{\text{cycles}(s)=w} \cdot \Pr(s)$, where $\mathcal{S}$ is the set of microarchitectural states reachable under the non-deterministic policy. Targeting *IEEE Real-Time Systems Symposium*, this would produce the first exact probability distribution over WCET for any non-deterministic microarchitecture.

Multicore interference via partial-order reduction. For two cores sharing a memory bus, exact WCET analysis must consider all $O(m^n)$ interleavings of m memory accesses across n cores. Partial-order reduction prunes equivalent interleavings by identifying that bus-access events on different cores commute when they target different memory banks. A paper in *ACM Trans. on Architecture and Code Optimization* would present the first exact multicore WCET analysis exploiting commutativity pruning.

Chapter 16

Formally Verified Spatial and Temporal Isolation in Mixed-Criticality RTOS

Can we design an RTOS kernel for RISC-V that is formally proven (in Coq or Isabelle) to guarantee both spatial and temporal isolation between mixed-criticality tasks, with zero runtime overhead for safety-critical tasks?

Definition 16.1 (Spatial Isolation)**.** *Spatial isolation* guarantees that no task in the system can read or write another task's memory region. On RISC-V, this is enforced by hardware primitives such as `PMP` (Physical Memory Protection) and `IOPMP` (I/O PMP) that restrict each task's memory and peripheral access to statically configured regions.

Definition 16.2 (Temporal Isolation)**.** *Temporal isolation* guarantees that the execution time of a safety-critical task is unaffected by the behavior of any non-critical task. This requires dedicated cores or fully partitioned resources so that contention on shared interconnects, caches, or memory controllers does not introduce timing variability.

In mixed-criticality systems, safety-critical tasks such as brake controllers and flight stabilizers coexist with non-critical tasks such as infotainment and telemetry. The RTOS must guarantee spatial isolation (no task can read or write another task's memory) and temporal isolation (the execution time of a critical task is not affected by non-critical tasks). Current approaches fall into two camps. Hardware-based isolation using MMU or IOMMU for spatial isolation and hardware scheduling for temporal isolation involves the kernel in every context switch, incurring one to ten percent overhead for critical tasks. Software-based isolation using capability-based access control has lower hardware requirements but higher software overhead.

We propose to achieve isolation with zero runtime overhead for safety-critical tasks: the kernel never executes on behalf of a critical task. At boot, the kernel configures hardware isolation primitives, i.e., `PMP`, `IOPMP`, and `SmMTT`, to create protected execution environments. Critical tasks run directly on the hardware without kernel mediation, accessing memory, handling interrupts, and performing I/O without a single system call. The kernel runs only on non-critical cores or in idle time, managing non-critical tasks. A formal proof guarantees that critical tasks are fully isolated from each other and from non-critical tasks, with no timing interference.

16.1 Related Work

Approach	Strength	Gap
seL4 (Verified microkernel)	Full functional correctness proof in Isabelle	Runtime overhead for all tasks
Pip (RISC-V protected domains)	Hardware-enforced isolation via PMP	Temporal isolation not proven; kernel handles critical interrupts
RISC-V Worlds (Hardware isolation)	Multiple execution environments with separate page tables	No RTOS; no temporal scheduling guarantees
IOPMP / SmMTT (Physical memory protection)	Fine-grained physical memory access control	Can enforce spatial isolation but no temporal
DROPS / L4 (Microkernel for real-time)	Spatial + temporal isolation	Overhead for critical tasks
CertiKOS (Verified OS kernel)	Layer-based verification in Coq	Not designed for zero-overhead critical tasks
OpenTitan (Open-source RoT)	Hardware root of trust with isolation	Not an RTOS; no temporal guarantees

No existing system combines zero runtime overhead for critical tasks with formal proof of both spatial and temporal isolation.

16.2 Proposed Approach

The architecture proceeds in three phases: boot, critical execution, and non-critical management.

Boot phase. The kernel loads critical task images into isolated memory regions, configures `PMP` for each region so no other task can access it, configures `IOPMP` for peripheral access, configures `SmMTT` timers that fire directly to critical tasks rather than through the kernel, configures interrupt delegation so critical interrupts go directly to the critical task's handler, and halts the cores running critical tasks with the reset vector pointing to the first critical task.

Critical execution. The kernel never runs. The critical task runs on its dedicated core, accessing memory and peripherals without any system call overhead. The timer fires at the end of its time slot, and a machine-mode handler switches to the next critical task. If the critical task finishes early, it signals done via a hardware register and waits for the next slot.

Non-critical management. Non-critical tasks run on a separate core where the kernel manages them, accessing critical task memory only through DMA with `IOPMP` protection. There is no IPC between critical and non-critical tasks.

The central research question is identifying the minimal set of RISC-V hardware features needed to support zero-overhead isolated execution: `PMP` for per-task memory regions, `IOPMP` for peripheral access control, `SmMTT` for per-context timer interrupts, and interrupt delegation so critical interrupts

go directly to critical tasks. A related question is whether all isolation policies can be configured statically at boot or whether critical tasks sometimes need dynamic policy changes that would require kernel intervention.

Design Principle: Zero-Overhead Isolation through Static Configuration

The critical path of a safety-critical task must never execute kernel code. All isolation policies (memory regions, peripheral access, interrupt routing, timer configuration) are determined statically at boot time. This eliminates runtime mediation, context-switch overhead, and system call latency for critical tasks. The formal proof is tractable because the configuration is immutable during critical execution, reducing the verification burden to checking a finite set of boot-time parameters.

We formally specify the architecture in `Coq`, providing a hardware model of RISC-V with `PMP`, `IOPMP`, `SmMTT`, interrupt delegation, and multi-core operation, a boot kernel specification of valid configurations, a critical task specification of what tasks can do, and the isolation properties themselves. The spatial isolation theorem states that for any configuration satisfying the boot kernel's policy, any critical task runs with memory access restricted to its assigned region, proved by induction on the execution trace checking `PMP` at each memory access. The temporal isolation theorem states that on dedicated cores with dedicated or cache-partitioned caches, critical task execution time equals its standalone WCET. The temporal proof depends on resource partitioning: if tasks share any resource such as memory bus or DRAM controller, the thesis must either assume full partitioning or bound the interference.

The hardest challenges are interrupt handling (if critical tasks delegate to machine mode, the handler must be simple enough to verify yet flexible enough for various interrupt sources) and the limited `PMP` entries (typically 8 to 16 per hart), which constrains the number of critical tasks per core. Zero overhead must also be qualified: the timer interrupt handler consumes cycles, and caches may have cold misses on context switch.

16.3 Evaluation

Criteria	Method	Target
Spatial isolation	Attempt cross-task access, verify PMP blocks it	100% blocked
Temporal isolation	Run critical task under varying non-critical load, measure execution time variance	$< 1\%$ variance
Zero overhead	Compare critical task runtime with bare-metal execution	Within 1% of bare-metal
Proof coverage	Coq proof lines vs. implementation lines	All isolation-critical boot code covered
Comparison	Against seL4, Pip, FreeRTOS	Overhead: ours $\leq$ theirs

We implement the boot kernel and critical task infrastructure for a RISC-V platform such as

CVA6 with multiple cores on FPGA or gem5 simulation. Evaluation measures spatial isolation by attempting cross-task memory and peripheral access, temporal isolation by measuring critical task execution time under varying non-critical loads, and performance by comparing critical task runtime against bare-metal execution. If successful, the approach extends to shared resources with bounded interference via hardware arbitration, dynamic criticality changes through promotion protocols, fault-tolerant isolation with detection and recovery mechanisms, and DMA device isolation via IOPMP.

16.4 Research Directions

Dynamic criticality promotion with verified isolation. When a non-critical task must be promoted to critical at runtime (e.g., a telemetry task becomes safety-relevant during takeoff), the PMP and IOPMP configuration must be updated atomically across all harts. A promotion protocol with a two-phase commit, where each core enters a quiescent state, the kernel rewrites PMP entries, and cores resume, requires a proof that no window exists where isolation is violated. A paper in *IEEE Symposium on Security and Privacy* would present the first formally verified dynamic criticality mechanism for zero-overhead RTOS.

Shared resource isolation with bounded interference. The temporal isolation theorem assumes fully partitioned resources, but real systems share DRAM controllers and memory buses. For a shared DRAM controller with a bank-grouped row-buffer policy, the worst-case additional latency due to a co-runner is $\Delta_{\text{DRAM}} = n_{\text{conflict}} \cdot (t_{\text{RP}} + t_{\text{RCD}})$ per access, where n_{conflict} is bounded by the number of rows the competing core can open in a time window. Targeting *IEEE Real-Time Systems Symposium*, this work extends zero-overhead isolation to shared resources with a formally bounded interference term.

Fault-tolerant isolation with detection and recovery. A single-bit error in PMP configuration memory could open a spatial isolation hole. A fault-tolerant architecture duplicates the PMP configuration on each hart, periodically checks consistency via a hardware watchdog that reads both copies, and on mismatch halts all critical cores and restores from a golden copy in ROM. Proving that this protocol preserves isolation under μ single-bit faults within any window of T_{check} cycles would target *IEEE Trans. on Dependable and Secure Computing* and establish the first verified fault-tolerant isolation kernel.

Verified DMA device isolation via IOPMP. A peripheral performing bulk DMA transfers can overwrite critical memory if IOPMP is misconfigured. A formal model of the IOPMP's per-transaction address-range check, combined with a proof that the boot kernel's IOPMP configuration covers every possible DMA address generated by allowed devices, ensures spatial isolation for DMA. A publication in *ACM Conf. on Computer and Communications Security* would provide the first end-to-end formal proof of DMA isolation with zero runtime overhead.

Temporal isolation under partitioned caches with warm-up effects. The temporal isolation theorem must account for cold misses after context switch. For a critical task with working set W_{crit} occupying A cache ways, the worst-case additional cycles due to cold-start is $\sum_{i=1}^{A}(1 - h_i) \cdot t_{\text{miss}}$,

where h_i is the hit rate of way i after warm-up, determinable through offline profiling. Targeting *IEEE Trans. on Computers*, this work closes the gap between zero-overhead claims and practical cache behavior.

Chapter 17

Deterministic Execution Models for Multicore Embedded Systems

Can DDE/DDF timing-anomaly-free execution models be extended to multicore embedded platforms with non-uniform memory access while preserving the timing-anomaly-freedom guarantee?

Definition 17.1 (Timing-Anomaly Freedom)**.** A system is *timing-anomaly-free* (TA-free) when a faster local decision never leads to a globally slower outcome. Formally, for any two executions differing only in a single core's local execution speed, the global completion time in the faster execution is less than or equal to the global completion time in the slower execution. This property enables compositional WCET analysis because the worst-case path can be found by examining local decisions independently.

Definition 17.2 (Deterministic Resource Arbitration)**.** *Deterministic resource arbitration* guarantees that a core's waiting time for a shared resource (bus, cache, memory controller) depends only on a fixed schedule, such as `TDMA` slot assignments, and not on the dynamic behavior of any other core. This is the essential mechanism for extending TA-freedom from single-core to multicore systems.

Deterministic Dynamic Execution (`DDE`) and Deterministic Data Flow (`DDF`) are execution models that guarantee timing-anomaly freedom on single-core systems: a faster local decision never leads to a globally slower outcome. This property is critical for WCET analysis because the worst-case path can be found by examining local decisions independently, avoiding global combinatorial explosion. However, multicore systems introduce a fundamental problem. A core that completes a task early may then contend for a shared resource such as cache or memory bus at the same time as another core, causing interference that would not have occurred if the first core had run late. Locally better behavior (faster completion) triggers globally worse behavior (more contention), violating timing-anomaly freedom.

We propose to extend `DDE` and `DDF` to multicore by introducing deterministic resource arbitration that restores timing-anomaly freedom. Hardware arbitration mechanisms such as time-division multiplexing and round-robin with bounded slots make resource access timing independent of core behavior.

17.1 Related Work

Approach	Strength	Gap
DDE (Deterministic Dynamic Execution)	Timing-anomaly-free single-core scheduling	Single-core only
DDF (Deterministic Data Flow)	TA-free cause-effect chains, 9–12% MRT reduction	Single-core only
PRET / ARPRET (Precision-timed architectures)	Deterministic pipeline, thread-interleaved	Shared resources still create anomalies
Fractal CoMP (Spatio-temporal partitioning)	Predictable multicore via TDMA arbitration	Conservative; no formal proof of TA-freedom
MERASA / T-CREST (Time-predictable multicore)	TDMA bus, predictable cache	Pipeline timing anomalies not fully eliminated
PTARM (Precision-timed ARM)	Deterministic pipeline design	Single-core only
TA-free pipeline design (Reineke, Sen)	Characterization of timing anomalies in pipelines	Single-core pipeline; does not address multicore

No existing work extends `DDE` or `DDF` principles to multicore systems while preserving timing-anomaly freedom with bounded performance cost.

17.2 Proposed Approach

We first define multicore timing-anomaly freedom. Local TA-freedom means each core individually satisfies single-core `DDE` TA-freedom. Cross-core TA-freedom means the timing of one core does not depend on the decisions of another core: resource arbitration is deterministic and isolation is preserved. The formal model extends the `DDE` model to multiple cores: each core follows `DDE` choosing the fastest available action consistent with its schedule, shared resource requests are queued and arbitrated by a deterministic mechanism, and the arbitration mechanism has a worst-case waiting time per core per resource. The central research question is proving that the global system is TA-free under this model: for any two executions differing only in a single core's local execution speed, the global completion time in the faster execution is less than or equal to the global completion time in the slower execution.

We design and evaluate four arbitration mechanisms:

- `TDMA` with fixed time slots per core: TA-free.
- Round-robin with bounded bursts per core: TA-free when the burst length exceeds the longest indivisible access.
- Credit-based `TDMA` with token buckets: not TA-free but bounded-anomaly.
- Static priority `TDMA` with priority override: not TA-free because behavior depends on which cores have pending requests.

For each, we prove or disprove TA-freedom and compute utilization as the fraction of cycles spent on actual data transfer versus waiting or overhead.

For the shared cache, we design three deterministic alternatives:

1. Fully partitioned caches where each core has a fixed number of cache ways and no cross-core eviction occurs: TA-free.
2. `TDMA` caches where the cache is accessible only during a core's `TDMA` slot: TA-free but low utilization for memory-intensive workloads.
3. Deterministic replacement with a known fixed seed where the replacement sequence is deterministic and known in advance: TA-free but performance depends on the seed.

The hardest challenge is that cache partitioning reduces effective cache size: a four-way set-associative L2 cache split among four cores means each core gets one way, effectively a direct-mapped cache, which may increase miss rates enough to negate multicore benefits.

Design Principle: Compositional Timing-Anomaly Freedom

The proof of global TA-freedom is compositional: each core satisfies local `DDE` TA-freedom, the arbitration mechanism guarantees that waiting time depends only on the slot schedule and not on core behavior, and therefore the global system is TA-free by contradiction. This compositional structure means that adding more cores does not complicate the proof: it simply instantiates the same local and arbitration guarantees for each new core.

17.3 Evaluation

Configuration	Metric	Baseline
TDMA bus + partitioned cache	Worst-case execution time (bound)	Standard multicore (free bus)
Round-robin bus + partitioned cache	WCET bound, utilization	TDMA, standard
Credit-based bus + partitioned cache	WCET bound, utilization, anomaly bound	TDMA, standard
TDMA bus + TDMA cache	WCET bound, cache utilization	TDMA bus + partitioned cache
All	TA-freedom verification	Model check the arbitration logic
All + PapaBench	End-to-end latency	Single-core, standard multicore

We implement the `DDE` scheduler with deterministic arbitration on a multicore RISC-V platform in `gem5` or FPGA, with a `DDE` scheduler per core, a `TDMA` bus arbiter as a hardware state machine, partitioned L2 cache with private L1 per core, and a `TDMA` memory controller. Evaluation uses `PapaBench` (drone autopilot), the `DEOS` avionics benchmark, and synthetic benchmarks varying task counts and resource access intensities. Comparison targets include standard multicore without TA guarantees, time-predictable multicore approaches such as `MERASA` or `T-CREST`, and single-core execution as the ideal isolation upper bound. The key question is the performance cost of deterministic arbitration: if the cost is less than thirty percent for typical embedded workloads,

the approach is practical. If successful, the framework extends to heterogeneous multicore DDE with weighted slot schedules, dynamic DDE under TA-freedom with slot reconfiguration protocols, energy-aware DDE with DVFS, and IO timing with deterministic DMA and peripheral arbitration.

17.4 Research Directions

Heterogeneous multicore DDE with weighted slot schedules. On a big.LITTLE platform where core execution rates differ by a factor α, a weighted TDMA schedule with slot length proportional to α preserves TA-freedom: the worst-case waiting time for core i on resource r is $W_i^r = \sum_{j \neq i} s_j$, where s_j is core j's weighted slot length. Proving TA-freedom under weighted arbitration requires a new inductive argument over the global state that accounts for the mismatch in per-core progress rates. A paper in *IEEE International Symposium on High-Performance Computer Architecture* would establish the first TA-free execution model for heterogeneous multicores.

Dynamic slot reconfiguration under TA-freedom. When task sets change at runtime (e.g., a safety-critical task becomes active during a specific flight phase), the TDMA slot schedule must be reconfigured. A reconfiguration protocol that drains pending bus transactions, pauses all cores in a barrier, installs the new schedule atomically, and proves that the global state remains TA-free across the transition would appear in *IEEE Real-Time Systems Symposium*. The key invariant is that the reconfiguration interval is treated as a deterministic idle slot in the schedule.

Energy-aware DDE with DVFS. Voltage and frequency scaling can reduce energy by $P \propto fV^2$ but changes execution time and may break TA-freedom if frequency transitions are not aligned with deterministic arbitration. By restricting DVFS transitions to TDMA slot boundaries and proving that within a slot the frequency is constant, TA-freedom is preserved. The optimal per-slot frequency minimizes $E = \sum_i \frac{C_i}{f_i} \cdot P(f_i)$ subject to deadline constraints and TA-freedom. Targeting *IEEE Trans. on Computer-Aided Design*, this would produce the first energy-optimal TA-free multicore model.

Deterministic I/O and DMA arbitration. Extending TA-freedom to peripherals requires that DMA transfers and memory-mapped I/O use the same deterministic bus arbitration as CPU memory accesses. A DMA engine with a TDMA slot of its own, whose start and end are aligned with the core TDMA schedule, ensures that peripheral-initiated traffic also respects TA-freedom. A publication in *ACM Trans. on Embedded Computing Systems* would extend the DDE model to cover all masters on the memory bus.

Probabilistic TA-freedom for credit-based arbitration. Credit-based arbitration (e.g., IEEE 802.1Qav) is not TA-free because a core that accumulates credits faster may gain more bus access. However, for a given credit limit $C_{\max}$, the maximum anomaly, the ratio of completion time under credit-based to completion time under TDMA, is bounded by $1 + \frac{C_{\max}}{N \cdot s_{\min}}$, where $s_{\min}$ is the minimum TDMA slot. A paper in *IEEE Real-Time Systems Symposium* would characterize this bound and derive probabilistic WCET bounds under credit-based arbitration.

Chapter 18

Real-Time Guarantees on Energy-Harvesting Systems

Can we provide probabilistic real-time guarantees for critical tasks on energy-harvesting battery-less devices?

Energy-harvesting devices powered by solar, RF, thermal, or vibration energy operate without batteries, storing energy in capacitors and experiencing unpredictable power failures when harvested energy is insufficient. This creates a fundamental challenge for real-time computing: the device may lose power at any time, erasing volatile state; the time between power restoration and the next power failure is unpredictable; and a critical task may be interrupted mid-execution, requiring forward progress guarantees that it eventually completes despite repeated power failures. Current approaches either use checkpoint-restore to survive power loss with ad-hoc checkpoint placement, give up on real-time guarantees and treat the system as best-effort, or use supercapacitors large enough to guarantee completion of any single task at the cost of size, expense, and recharge time.

We propose to provide probabilistic real-time guarantees for energy-harvesting systems: the guarantee is probabilistic because the energy supply is non-deterministic, but we can bound the probability of missing a deadline.

18.1 Related Work

Approach	Strength	Gap
Chain (Checkpointing for intermittent computing)	Task-based checkpointing with static energy analysis	No probabilistic guarantees; assumes worst-case energy
CatNap / InK (Intermittent computing RTOS)	RTOS with checkpoint support	Best-effort; no deadlines or probabilistic models
Quark (Compiler for intermittent computing)	Compiler-inserted checkpoints for forward progress	No timing or deadline analysis
Alpaca (Checkpoint placement optimization)	Optimal checkpoint placement for minimum forward progress time	Assumes unique power failure model; no probabilistic guarantees
Energy-aware scheduling (Chetto, Moser)	Tasks scheduled to meet energy constraints	Assumes known, bounded energy supply
Probabilistic WCET (Edgar, Burns)	WCET as probability distribution	No energy modeling
Energy harvesting model (Gorlatova, Kansal)	Stochastic model of energy harvesting processes	Not connected to real-time scheduling theory

No existing work combines probabilistic energy harvesting models with real-time scheduling

theory to provide probabilistic deadline guarantees for intermittent computing.

18.2 Proposed Approach

We characterize the energy harvesting process for three common sourcessolar, RF, and thermoelectric or vibrationby collecting traces from published datasets or measurements and fitting models. We use a Markov chain model where the state is capacitor charge level and transitions are harvesting events (increasing charge) and compute events (decreasing charge), a renewal process where energy inter-arrival times follow a known distribution such as Exponential, Weibull, or Gamma, and a worst-case bound as a fallback. The models are validated against real hardware measurements.

The central research question is the forward progress function.

Definition 18.1 (Forward Progress Function)**.** For a task with checkpoints at positions $c_1, \ldots, c_m$ in its instruction sequence, let E_{inst} be the energy per instruction, E_{ckpt} the energy per checkpoint, C the capacitor capacity, and λ the energy harvesting rate. The forward progress function $\Pi(t)$ gives the expected number of instructions completed after wall-clock time t, accounting for charging periods, compute periods, and checkpoint overhead. In each charge–compute cycle, the capacitor charges to capacity (duration C/λ), then the system runs until depleted (duration $C/(E_{\text{inst}} \cdot r)$ where r is the execution rate), then charges again.

The forward progress function gives the expected progress after a given amount of wall clock time, which includes charging periods, compute periods, and checkpoint overhead. This depends on energy per instruction, checkpoint energy, capacitor size, and harvesting rate. In each compute cycle, the capacitor charges to full capacity, then the system runs until depleted, then charges again. We derive the expected forward progress rate and its variance as functions of the capacitor size and harvesting rate.

We formulate checkpoint placement as a stochastic optimization. Given a task execution trace with per-instruction energy cost, an energy model, a deadline, and a required probability, we optimize the number and positions of checkpoints to maximize the probability of completing within the deadline. This is a stochastic dynamic programming problem over the task's instruction positions, where the state is the current instruction position and current capacitor charge, and the action is whether to place a checkpoint at each position.

Theorem 18.1 (Probabilistic Deadline Guarantee)**.** *For a task τ with WCET W, checkpoint placement $c_1, \ldots, c_m$, deadline $\mathcal{D}$, and energy harvesting process modeled as a Markov chain with steady-state capacitor charge distribution π, the probability that τ meets its deadline is*

$$\Pr(T_\tau \leq \mathcal{D}) = \sum_{q \in Q} \pi(q) \cdot \mathbf{1}_{FP(q,\mathcal{D}) \geq W},$$

where $FP(q, \mathcal{D})$ is the forward progress achievable starting with charge q within deadline $\mathcal{D}$, and the optimal checkpoint placement maximizes this probability.

Proof sketch. The probability is computed by solving a stochastic dynamic program over (instruction position, capacitor charge) states. The Bellman equation at each instruction position i with charge q is $V(i, q) = \max\{V(i+1, q - E_{\text{inst}}), \text{checkpoint cost} + V(i+1, C)\}$, with the base case $V(N, q) = 1$

for all q. The steady-state distribution π gives the probability of starting each compute cycle at each charge level. □

Because the state space is largemillions of instructionswe develop approximate methods including greedy placement at positions where the expected energy of the next segment exceeds the capacitor capacity, and optimal placement for piecewise-constant energy profiles via dynamic programming over segments.

For scheduling multiple tasks with different criticality levels and deadlines, we design a scheduler that executes tasks in priority order when the system has power. We compute the probability that each task meets its deadline using stochastic response time analysis: given the inter-arrival distribution of power-failure events from the energy model, we compute the probability that a task's total execution across multiple power cycles completes within its deadline. This uses Markov chain steady-state probabilities, where the probability of meeting a deadline is the probability that the sum of compute-phase durations within the deadline window exceeds the task's WCET.

The hardest challenges are that energy harvesting is highly variable and non-stationarysolar energy varies seasonally and RF energy varies with network usageso the scheduler must adapt using online learning to update the model as conditions change. Checkpointing to NVM has non-negligible energy cost, and if checkpoints are too frequent, the system spends all its energy on saving state and never makes progress. Probabilistic guarantees require rigorous statistical validation with thousands of runs and confidence intervals.

Core Thesis

Probabilistic real-time guarantees for energy-harvesting devices are achievable by modeling the energy process as a Markov chain, optimizing checkpoint placement via stochastic dynamic programming, and computing deadline miss probabilities through steady-state analysis of the forward progress function.

18.3 Evaluation

Experiment	Setup	Metric
Energy model validation	MSP430 with solar cell under controlled lighting	RMSE of predicted vs. measured energy availability
Checkpoint placement	Compare optimal, greedy, uniform checkpointing	Forward progress rate, deadline miss probability
Scheduling	3-task set with mixed criticality	Deadline miss probability per task
Application	Structural health monitor	End-to-end latency distribution, 99th percentile
Comparison	Supercapacitor-based, no checkpointing	Cost (capacitor size), performance, guarantee strength

We implement on a real energy-harvesting platform such as the TI MSP430 FRAM with RF harvesting or an STM32 with solar cell. The checkpoint mechanism saves registers and stack to

FRAM, the scheduler is priority-based with probabilistic analysis, and an energy monitor detects power loss via capacitor voltage sensing. Evaluation uses microbenchmarks for forward progress rate, sensor processing tasks, and a structural health monitor application. Comparison targets include no checkpointing, uniform checkpointing, and supercapacitor-based worst-case guarantees. If successful, the approach extends to multi-source energy harvesting, reconfigurable checkpoints that adapt to observed energy availability, distributed intermittent computing across multiple nodes, and energy-aware computation that trades off computation quality for improved deadline probability.

18.4 Research Directions

Online learning for adaptive energy-harvesting models. Solar energy availability varies seasonally and RF energy varies with network load, making a static Markov model inaccurate after deployment. An adaptive scheduler maintains a Dirichlet-process mixture of energy inter-arrival distributions, updating posterior parameters $\alpha_k \leftarrow \alpha_k + \mathbf{1}_{\text{event type}=k}$ after each harvesting event, and recomputing deadline probabilities under the updated model. A paper in *IEEE Real-Time Systems Symposium* would present the first adaptive real-time scheduler for intermittent computing with formal convergence bounds.

Multi-source energy harvesting optimization. A device with solar and RF harvesting must allocate compute time across sources with different availability profiles. The optimal policy maximizes forward progress $\mathbb{E}[P] = \sum_{s \in \mathcal{S}} \frac{E_s \cdot r_s}{e_{\text{inst}}}$, where E_s is the expected energy per unit time from source s, r_s is the fraction of time allocated to source s, and e_{inst} is energy per instruction. Solving the allocation as a linear program under capacity constraints $0 \leq r_s \leq 1$ and $\sum_s r_s \leq 1$ yields periodic schedules measurable on real hardware. Targeting *ACM Trans. on Embedded Computing Systems*, this would be the first multi-source intermittent computing framework with real-time guarantees.

Distributed intermittent computing. Multiple energy-harvesting nodes coordinate to complete a joint task (e.g., distributed structural health monitoring) while each node independently power-cycles. A coordination protocol using non-volatile shared state on an FRAM bus allows nodes to check progress of peers after each power restoration. The probability that the joint task meets its deadline is $\Pr(\max_i T_i + \Delta_{\text{comm}} \leq \mathcal{D})$, where T_i is node i's completion time and Δ_{comm} is the communication overhead. A publication in *IEEE Trans. on Parallel and Distributed Systems* would provide the first formal analysis of distributed forward progress under intermittent operation.

Energy–quality tradeoffs with graceful degradation. When deadline probability falls below a threshold θ, the task can degrade quality (reducing sampling rate by half, switching to a coarser algorithm) to reduce per-instruction energy e_{inst} and raise deadline probability. A control loop that selects quality level $q \in \{1, \ldots, Q\}$ to minimize distortion $D(q)$ subject to $\Pr(T(q) \leq \mathcal{D}) \geq \theta$ can be solved via dynamic programming over quality levels. Targeting *IEEE Trans. on Computers*, this work extends real-time guarantees to approximate computing on energy-harvesting devices.

Reconfigurable checkpoint placement via control theory. A feedback controller observes the average forward progress $\bar{p}_k$ over the last N power cycles and adjusts checkpoint spacing δ_k proportionally: $\delta_{k+1} = \delta_k + K_p(\bar{p}_{\text{target}} - \bar{p}_k)$. The stability condition $|1 - K_p \cdot \frac{\partial \bar{p}}{\partial \delta}| < 1$ guarantees convergence to the target progress rate. A paper in *IEEE Real-Time Systems Symposium* would introduce the first control-theoretic checkpoint adaptation for intermittent computing with formal stability guarantees.

Part IV

Operating Systems

Chapter 19

Fine-Grained Kernel Compartmentalization Without Hardware Changes

Can we achieve fine-grained kernel compartmentalization (thousands of compartments) using only commodity hardware features, and what is the fundamental overhead of compartment switching?

Definition 19.1 (Kernel Compartment)**.** A *kernel compartment* is a unit of kernel code and data that executes in its own private page table hierarchy. Each compartment has spatial isolation from all other compartments: its private mappings are inaccessible from outside, and shared kernel mappings (e.g., kernel text) are mapped read-execute across all compartments.

Definition 19.2 (Compartment Switch Cost)**.** *Compartment switch cost* is the latency of transitioning between compartments, including `TLB` flush (or selective invalidation), page table pointer switch (e.g., writing `SATP` on RISC-V), and indirect costs from cache pollution during page table walks for the new compartment's working set.

The operating system kernel is the most privileged software in the system, and a vulnerability anywhere in the kernel compromises everything. Kernel compartmentalization aims to limit the damage so that even if one compartment is compromised, others remain isolated. Pomegranate showed that fine-grained kernel compartmentalization is possible using hardware virtualization extensions, supporting up to 32 compartments with negligible overhead for MTU-sized network packets, but this requires specific hardware features, significant engineering effort to virtualize the kernel itself, and a hypervisor layer that adds complexity.

We propose to achieve comparable compartmentalization using only commodity hardware features available on all modern processors: page tables, virtual memory, and task switching. By giving each compartment its own page table hierarchy and switching page tables on compartment entry and exit, we provide spatial isolation between compartments. The challenge is that page table switching is expensive due to `TLB` flush and cache pollution, compartments need to share some kernel data, and switching must be fast enough for frequent operations such as system calls and interrupt handlers.

19.1 Related Work

Approach	Strength	Gap
Pomegranate (VT-x + VT-rp, up to 32 compartments)	Low overhead, strong isolation	Requires virtualization extensions; max 32 compartments
MPK (Memory Protection Keys, x86)	Fast permission switching	Only 4 domains; bypassable by privileged code
CHERI (Capability-based memory safety)	Fine-grained, architecturally enforced	Not commodity
ERIM (MPK-based kernel compartmentalization)	Software-only, low overhead for memory access	Only 4 compartments; MPK vulnerabilities
KSplit (Page-table-based kernel isolation)	Uses page tables for isolation	High TLB flush overhead; limited evaluation
SKEE (Isolated kernel execution)	Lightweight isolation for device drivers	Driver-specific; not general compartmentalization
Nested kernels (L4, seL4)	Full isolation via microkernel	All IPC is expensive; not fine-grained

No software-only approach achieves more than 32 compartments with sub-microsecond switching overhead while maintaining performance for common kernel operations.

19.2 Proposed Approach

Each compartment has private state including its own page table tree and its own per-CPU data, and shared state including kernel text mapped read-execute across all compartments, scheduler state, and file descriptor tables which may be per-compartment or global. The interface is asymmetric: a compartment can call into the kernel via a system call but cannot directly access another compartment's private state. The central research question is determining the minimum cost of switching between page table hierarchies, which involves `TLB` flush cost, page table walk for the first access in the new compartment, and indirect costs from cache pollution during page table walks.

We design three optimizations for fast page table switching:

1. **Partial TLB flush.** Flushes only entries belonging to the previous compartment's private virtual address range rather than the entire `TLB`, using range-based invalidation available on x86 via `INVEPT` or `INVPCID` and on RISC-V via `SFENCE.VMA` with an address range.

2. **Pre-walking.** Primes the `TLB` and page walk cache for the new compartment's expected working set (stack, per-CPU data, syscall entry) before the actual switch, based on profiling data.

3. **Shared top-level page table entries.** Map common kernel regions using the global page feature (the `G` bit on RISC-V or the PGD or `G` flag on x86), so `TLB` entries for kernel mappings survive the page table pointer change and are not flushed.

The key design decision is whether a system call causes a compartment switch. If every system call pays the switch cost, the overhead may be unacceptable: a 200-cycle system call becomes 1000 cycles. If the syscall handler runs in the calling compartment, the kernel code must be mapped in every compartment's page table, and the handler must be verified to not access compartment-private data unless authorized.

Design Principle: Threshold-Based Compartment Switching

A system call should trigger a compartment switch only when the cost of switching is amortized over the work performed in the target compartment. Frequently accessed compartments (e.g., the syscall handler itself) should be mapped in every compartment's page table to avoid paying the switch cost on the fast path. The optimal switching policy depends on the ratio of the compartment switch cost to the average system call execution time, which varies by workload.

We decompose Linux into compartments using four candidate strategies: by subsystem (VFS, device drivers, networking, memory management), by system call class, by data owner (each file descriptor table is a compartment), or by device. Each strategy is evaluated using static call graph analysis and dynamic profiling to measure the fraction of operations that stay within a compartment, the average depth of inter-compartment calls, and the working set per compartment.

The hardest challenges are that `TLB` flush cost is architecture-dependent (commodity RISC-V cores may not support efficient partial `TLB` flushes) and that shared top-level `PTE` pages must be read-only from the compartment's perspective to prevent a compromised compartment from modifying the shared mapping and affecting other compartments. We characterize the fundamental overhead of page-table-based compartment switching with a formula involving `TLB` refill time, cache miss time, and page walk time, measured against real hardware and compared to the theoretical minimum.

19.3 Evaluation

Benchmark	Metric	Baseline
Syscall latency (getpid, write)	Cycles	Unmodified Linux
IPC throughput (pipe, socket)	MB/s	Unmodified Linux, Pomegranate
Apache / Nginx	Requests/sec	Unmodified Linux, Pomegranate
Redis	GET/SET latency	Unmodified Linux
Security (crafted attacks)	Can cross-compartment access succeed?	No
# compartments	Maximum before switching cost dominates	Scalability curve

We implement the compartmentalization in Linux on RISC-V using `SATP` switching for compartment isolation, with each compartment having its own page table, global kernel pages mapped with the `G` bit to avoid flushing on switch, per-compartment private data in dedicated virtual address ranges for range-based `SFENCE.VMA`, and inter-compartment IPC via shared memory. Evaluation

uses macrobenchmarks including Apache, Nginx, Redis, and PostgreSQL, microbenchmarks for system call latency and IPC throughput, and security tests attempting cross-compartment access. If successful, the approach motivates a lightweight compartment switch instruction proposal for future hardware, dynamic compartment boundaries for hot-plugged devices and loaded modules, and compiler-supported automatic kernel compartmentalization based on data flow analysis.

19.4 Research Directions

Compiler-directed automatic compartment boundary discovery. Current compartment boundaries are manually specified (by subsystem, syscall class, or data owner). Can we extend a Rust or Clang compiler with static data-flow and control-flow analysis that automatically proposes optimal compartment boundaries, minimizing cross-compartment call frequency while maximizing isolation? A SOSP or OSDI publication would contribute a formal cost model $\text{Cost}(\mathcal{C}) = \sum_{i \neq j} \lambda_{ij} \cdot C_{\text{switch}} + \sum_i |\mathcal{C}_i| \cdot C_{\text{private}}$ where λ_{ij} is the call frequency between compartments $\mathcal{C}_i$ and $\mathcal{C}_j$, validated against real kernel module call graphs from LKML traces.

Hardware-software co-design for lightweight compartment switching. The software-only `TLB` flush and page-table switch dominates cost. Can we propose a minimal ISA extension (a single `COMPARTMENT_SWITCH` instruction that saves the old page table pointer, loads the new one, and selectively flushes only non-global `TLB` entries) alongside a Linux prototype that uses it when available? Targeting ISCA or MICRO, the novelty lies in quantifying the performance gap between software-only and hardware-assisted switching, and showing that even a 10-cycle instruction saves 30–50% of compartment-switch latency in database workloads.

Dynamic compartment merging and splitting under workload phase changes. Kernel workloads exhibit phase behavior: during a `mkdir` storm, VFS compartment activity spikes, while during a `sendfile`-dominated phase, the network stack compartment is hottest. Can we design a runtime that monitors inter-compartment call frequency and hotness, and dynamically merges hot communication partners into one compartment or splits overloaded compartments? For EuroSys or ATC, the expected contribution is a feedback controller with bounded overhead ($< 1\%$ CPU) that reduces cross-compartment calls by 2–5$\times$ during phase transitions while preserving the security invariant that no merged compartment can access the private data of another.

Chapter 20

Optimal Page Migration Granularity in Tiered Memory

Can we derive the optimal migration granularity and policy analytically from access patterns, rather than heuristically?

Tiered memory systems combine fast tiers such as DRAM and HBM with slow tiers such as CXL-attached memory, NVDIMM, and compressed memory. To get good performance, the OS must migrate frequently-accessed pages to fast tiers and infrequently-accessed pages to slow tiers. Current approaches are heuristic: TPP in Linux promotes pages when accessed more than N times in a window and demotes when cold for M seconds; Memtis uses access tracking with multi-generational LRU; OBASE uses object-level hotness clustering; HeMem uses a heuristic threshold based on access density. These heuristics have no optimality guarantees. There is no theoretical framework to answer questions such as whether to migrate a page after one access or one hundred accesses, whether to migrate 4 KB pages or 2 MB pages, whether to migrate pages or entire objects, or how to handle changing access patterns.

Problem 20.1 (Optimal Page Migration in Tiered Memory)**.** Given N pages with unknown access frequencies $\{f_1, \ldots, f_N\}$, two memory tiers with access latencies $L_{\text{fast}} < L_{\text{slow}}$, migration cost c_{mig}, and fast-tier capacity C, find an online migration policy $\pi : \{0,1\}^N \times \mathbb{R}^N \to \{0,1\}^N$ that at each timestep decides which pages reside in the fast tier, minimizing the expected cumulative access cost $\mathbb{E}[\sum_t \sum_i (x_i^{(t)} \cdot L_{\text{fast}} + (1 - x_i^{(t)}) \cdot L_{\text{slow}}) \cdot a_i^{(t)} + \text{migration\_costs}]$ where $a_i^{(t)} \sim \text{Poisson}(f_i)$ are access events.

We propose to formulate page migration as an optimization problem with a well-defined cost model and derive analytically optimal policies that provably minimize average access latency under realistic workload models.

20.1 Related Work

Approach	Strength	Gap
TPP (Linux, default tiering)	Simple, practical, deployed	No optimality analysis; fixed thresholds
Memtis	Multi-generational LRU + access density	Heuristic; no formal model
OBASE (Object-based clustering)	Compiler-assisted, up to 70% memory reduction	Compiler dependent; still heuristic migration
PACT (Criticality-first)	Prioritizes critical pages	Criticality is hard to define and measure
CXLalloc (CXL-aware allocator)	Manages hotness per allocation	Allocator-level; not runtime migration
AutoNUMA (Linux NUMA balancing)	Page migration for NUMA	Heuristic thresholds; no optimality analysis
Optimal offline migration (Wu, 2022)	Offline DP for known access trace	Offline; cannot be used online

No online page migration policy is proven optimal under a realistic cost model.

20.2 Proposed Approach

Multi-Armed Bandit Migration

The key idea is to formulate page migration as a multi-armed bandit problem where each page is an arm and placing it in the fast tier yields reward proportional to access latency saved. Clustered Thompson Sampling groups pages with similar access patterns, achieving sublinear regret and approaching the optimal known-frequencies policy over time.

We first consider two tiers with known access frequencies. The cost model includes fast tier access cost, slow tier access cost, migration cost including bandwidth penalty, access frequency per page, and a time horizon. The optimal policy is a greedy threshold: migrate a page from slow to fast if its access frequency multiplied by the latency difference multiplied by the time horizon exceeds the migration cost. For known frequencies, this gives the optimal solution for any set of pages by a simple exchange argumentif a page with higher frequency is in the slow tier while one with lower frequency is in the fast tier, swapping them reduces total cost.

In practice, access frequencies are unknown and must be estimated from sampling. This is a multi-armed bandit problem where each page is an arm and the reward for placing it in the fast tier is the access latency saving. The central research question is designing an online algorithm that balances exploration (learning access frequencies) and exploitation (placing hot pages in the fast tier) with provable regret bounds. We propose clustered Thompson Sampling: group pages into clusters with similar access patterns based on virtual address, NUMA node, process ID, or mapped file; maintain a Bayesian posterior over each cluster's access frequency; at each migration decision point, sample from the posterior and migrate pages with sampled frequency exceeding the

Algorithm 6 Clustered Thompson Sampling for Page Migration

Require: Pages $\{1, \ldots, N\}$, fast-tier capacity C
Ensure: Migration decisions $x_i \in \{0, 1\}$

```
Initialize Beta(1, 1) prior per cluster c
for each migration interval do
    f̃_c ← SAMPLE(Beta(α_c, β_c))                          ▷ posterior sample
    priority_i ← f̃_{c(i)} · (L_slow − L_fast)
    Promote C highest-priority pages from slow to fast
    Demote remaining fast pages exceeding capacity
    for each sampled access a_i do
        Beta(α_{c(i)}, β_{c(i)}) ← Update(α_{c(i)}, β_{c(i)}, a_i)
    end for
end for
```

threshold; and update posteriors from observed access counts. This algorithm achieves sublinear regret, meaning it approaches the optimal policy as time increases.

We extend the model to variable page sizes. Large 2 MB pages have lower TLB miss rate but coarser hot or cold granularity, while small 4 KB pages have finer granularity but more TLB misses and more page table entries. The optimal page size is derived as a function of the spatial locality of accesses: if accesses are spatially clustered with all bytes in a 2 MB region hot, use 2 MB pages; if accesses are sparse with only a few bytes per 2 MB region hot, use 4 KB pages. The optimal size is a function of spatial correlation estimated from page-level access samples.

For multi-tier systems with k tiers, each having capacity, latency, and inter-tier migration cost, the optimal placement is a water-filling algorithm: estimate access frequencies for all pages, sort by estimated frequency, assign the highest-frequency pages to the fastest tier until its capacity is full, then the next tier, and so on. Periodically re-evaluate and migrate pages that should change tier. This is optimal under the assumption that access frequencies change slowly relative to the monitoring interval.

The hardest challenges are that access frequency estimation from PMU sampling is noisyat a one percent sampling rate, pages with rare but important accesses may be missed entirelyand that migration cost is not constant: it includes data copying, page table updates causing TLB shootdown, and interconnect bandwidth consumption that may delay other requests. Phase changes where workloads transition between different access patterns require change-point detection mechanisms for rapid adaptation. Large-page allocation conflicts with fine-grained migration: if a 2 MB huge page with only a small hot portion is kept, it wastes fast-tier space on cold data; if broken into 4 KB pages, TLB misses increase.

20.3 Evaluation

Benchmark	Metric	Comparison
STREAM	BW utilization	TPP, HeMem, optimal offline
GUPS	Random access latency	TPP, HeMem, optimal offline
Graph500	Traversal rate	TPP, HeMem
Memcached	Tail latency, throughput	TPP, HeMem
SPEC CPU2017 (memory-intensive)	Execution time	TPP, no tiering (all DRAM)
Synthetic (power-law access)	Average access latency	Optimal (known frequencies), TPP

We implement the optimal migration policy in Linux by replacing TPP's LRU-based demotion and promotion with the analytic threshold policy, adding clustered Thompson Sampling for frequency estimation using PMU sampling, adding page size selection based on spatial locality estimates, and adding multi-tier support using the water-filling algorithm. Evaluation uses STREAM, GUPS, Graph500, SPEC CPU2017, Memcached, Redis, PostgreSQL, and synthetic benchmarks. The key question is how closely the online Thompson Sampling algorithm approaches the optimal known-frequencies policy: within five percent for stable workloads and within fifteen percent for phase-changing workloads. If successful, the approach extends to compiler and OS co-design where the compiler provides hints about future access patterns, hardware support via PMU counters that directly estimate migration benefit, and energy-aware migration that trades off latency against energy consumption.

20.4 Research Directions

Learning-augmented migration with program-context features. Current access-frequency estimation uses only raw page-access counts from PMU sampling. Can we augment the multi-armed bandit with program-context features (PC, call stack, PID, VMA region) to predict future hotness from past context? Using a lightweight online decision tree (e.g., Hoeffding tree) trained on PMU samples, each leaf predicts an access-frequency distribution, and Thompson sampling selects pages for promotion. A SIGMETRICS or EuroSys paper would contribute a 15–30% reduction in tail latency over Thompson-sampling-only baselines on workloads with phase changes (e.g., Spark shuffles), formalized as a contextual bandit with regret bound $O(\sqrt{T \log |\mathcal{H}|})$ where $|\mathcal{H}|$ is the tree hypothesis space.

Optimal migration under bandwidth contention. Migration itself consumes interconnect bandwidth, which may delay application memory requests. Can we formulate a congestion-aware migration policy that models the interconnect as a shared resource with capacity B and each page migration consuming cost c_m bytes, solving $\max \sum_i x_i \cdot \Delta_i$ subject to $\sum_i x_i \cdot c_m \leq B$ where $x_i \in \{0, 1\}$ indicates migrating page i and Δ_i is the estimated latency saved? For ASPLOS or SIGCOMM, the novelty is jointly optimizing migration and application bandwidth allocation; the expected contribution is an online primal-dual algorithm with competitive ratio $O(\log n)$ validated on CXL-bench workloads.

Compiler-assisted spatial-locality annotation for page size selection. The optimal page size depends on spatial locality, but the OS infers it indirectly from access samples. Can compiler analysis of loop access patterns (stride, trip count, reuse distance) annotate each anonymous or file-backed mapping with a recommended page size, which the OS uses to decide between 4 KB and 2 MB pages in tiered memory? For PLDI or ASPLOS, the contribution is a compiler pass producing `__attribute__((page_hint(min_granularity=4096, locality_score=0.3)))` annotations and a Linux mmap extension that respects them. Evaluation on Graph500 and SPEC CPU2017 would show 10–20% reduction in average access latency versus always-4 KB or always-2 MB policies.

Energy-aware tiered memory migration. Fast tiers (HBM, DRAM) consume significantly more power per GB than slow tiers (CXL, NVDIMM). Can we extend the optimal migration model to include energy cost E_{fast} and E_{slow} per access, solving $\min \sum_i f_i \cdot (x_i \cdot L_{\text{fast}} + (1 - x_i) \cdot L_{\text{slow}}) + \lambda \cdot \sum_i (x_i \cdot E_{\text{fast}} + (1 - x_i) \cdot E_{\text{slow}})$ where λ is a latency–energy trade-off parameter? For EuroSys or ATC, the expected contribution is a Pareto-optimal migration policy that allows the datacenter operator to choose an operating point, with measured energy savings of 20–40% at the 95th-percentile latency budget.

Chapter 21

Lock-Free OS Memory Management with Strong Isolation

Can the core OS memory management subsystem (page allocation, virtual memory mapping, TLB shootdown) be made entirely lock-free while maintaining strong spatial isolation guarantees?

Definition 21.1 (Lock-Free Memory Management)**.** *Lock-free memory management* replaces all spinlocks and mutexes in page allocation, virtual memory mapping, and `TLB` shootdown with atomic operations (`CAS`, fetch-add, store-conditional). Progress is guaranteed at the system level: at least one thread makes progress at any time, even if other threads are suspended, eliminating contention on multicore systems.

Definition 21.2 (Deferred TLB Shootdown)**.** *Deferred TLB shootdown* replaces inter-processor interrupt (IPI) synchronization with a global generation counter. When a core modifies a `PTE`, it increments the counter. Each core, on context switch or `TLB` refill, checks the counter and flushes its `TLB` if the counter changed since its last check. No IPIs are needed and the originating core does not wait.

The OS memory manager is the heart of the kernel, responsible for page allocation, virtual memory mapping, and `TLB` shootdown. All three operations are currently protected by locks: the page allocator lock protects free page lists and buddy allocator state, the MM lock protects per-process address space page table manipulation, and `TLB` shootdown uses inter-processor interrupts that force other cores to flush their `TLB`. These locks are a major source of contention on multicore systems. On a 128-core machine, the page allocator lock is a bottleneck for any workload that allocates and frees frequently. The mmap lock limits concurrency for page faults even when faults are to different pages. `TLB` shootdown IPIs can take thousands of cycles and block the originating core.

We propose to make all three operations lock-free using atomic operations (`CAS`, fetch-add, store-conditional) instead of locks, while maintaining the isolation guarantees that locks provide.

21.1 Related Work

Approach	Strength	Gap
Linux lockless page allocator (per-CPU pages)	Per-CPU page lists avoid lock contention	Balancing and reclaim still need locks
mimalloc / snmalloc (User-space lock-free allocators)	Fast concurrent allocation with per-thread heaps	Not designed for kernel constraints
Ivy / UMM (Lock-free kernel allocators)	General-purpose lock-free kernel allocators	Do not address VM mapping or TLB shootdown
RCU-based page table update (Linux)	RCU enables lock-free readers for some lookups	Writers still need locks
Lock-free page table manipulation (Hartsell, 2010s)	Concurrent page table updates via CAS on PTEs	Does not handle TLB shootdown
CMT / Conflict-free page table replication (Sung, 2020)	Replicate page tables to avoid shootdown	Higher memory usage

No existing work provides a complete lock-free memory management system covering allocation, VM mapping, and `TLB` shootdown in a unified framework.

21.2 Proposed Approach

We design a lock-free page allocator based on hazard pointers and atomic `Treiber` stacks, with per-NUMA-node, per-order free lists. Allocation `CAS`-pops a page block from the appropriate order's stack; if empty, it tries the next larger order and splits a larger block atomically. Free `CAS`-pushes the freed page block onto the appropriate order's stack. A background thread using work-stealing merges adjacent free pages via a radix tree keyed by physical address, using concurrent `CAS` operations to merge buddies. Cross-node allocation uses lock-free work-stealing. The central research question is whether we can support multiple page sizes, physical contiguity guarantees for DMA and huge pages, fragmentation resistance, and NUMA awareness using only atomic operations.

For lock-free page table manipulation, each `PTE` has a version counter using the `PTE`'s software-dirty or young bits. Single `PTE` updates use `CAS`: read the `PTE`, compute the new `PTE`, `CAS` the old with the new, retrying on failure. Multi-`PTE` operations such as splitting a huge page use versioned page table levels: each level has a version that increments on update, and readers check the version before and after reading, retrying if it changed. This is lock-free for readers and wait-free for writers.

For lock-free `TLB` shootdown, we replace IPI-based synchronization with deferred invalidation using a generation counter. When a core modifies a `PTE`, it increments a global `TLB` generation counter. Each core, on every context switch or `TLB` refill, checks the generation counter and flushes its `TLB` if the counter changed since its last check. No IPIs are needed and the originating core does not wait. We optimize with per-address-space generation counters so only cores running in that address space need to check.

Design Principle: Progress Through Isolation Decomposition

Each isolation property is enforced by a different lock-free mechanism:

Spatial isolation The `Treiber` stack ensures each page block is in exactly one stack at a time; `CAS`-pop removes it atomically, preventing double allocation.

Temporal safety A page is not freed while any `PTE` maps it because the unmapping `CAS` completes before the page reaches the free stack.

TLB consistency The generation counter check ensures that every core flushes its `TLB` within a bounded number of memory accesses.

Decomposing isolation into independently verifiable properties makes the lock-free design tractable.

The hardest challenges are the ABA problem in `Treiber` stacks solved by hazard pointers or tagged pointers, multi-`PTE` atomicity verification requiring careful reasoning about all interleavings, and the risk that deferred `TLB` shootdown causes excessive flushing if the generation counter changes rapidly, requiring an adaptive fallback to IPI-based shootdown.

21.3 Evaluation

Benchmark	Metric	Baseline
Parallel page fault storm (128 threads)	Faults/sec	Linux (lock-based)
Parallel mmap/munmap	Operations/sec	Linux
Parallel alloc/free (kernel internal)	Allocations/sec	Linux slab + buddy
Apache (high concurrency)	Requests/sec	Linux
Redis (multi-threaded)	Ops/sec, tail latency	Linux
Real-time (cyclictest-style)	Max allocation latency	Linux
Isolation verification	# double-alloc, # use-after-free	No failures

We implement in Linux on RISC-V by replacing the buddy allocator's spinlocks with `Treiber` stacks, the mmap lock for `PTE` operations with lockless `CAS` on `PTE`s, and IPI-based shootdown with generation-counter-based deferred invalidation. Evaluation uses scalability microbenchmarks with parallel mmap or munmap and page fault storms, macrobenchmarks with Apache and Redis at high concurrency, real-time latency tests, and isolation verification through intentional race injection. If successful, the approach extends to hardware-supported lock-free `TLB` where `PTE` updates implicitly invalidate `TLB` entries via cache coherence, machine-checked `Coq` proofs following the CertiKOS approach, and lock-free designs for all kernel subsystems.

21.4 Research Directions

Formal verification of the lock-free memory management stack. The lock-free algorithms presented rely on subtle atomic reasoning: ABA prevention in `Treiber` stacks, multi-`PTE` atomicity via versioned page tables, and generation-counter liveness. Can we produce machine-checked correctness proofs in `Coq` or Verus for the entire stack (allocator, page table manipulation, and `TLB` shootdown) that guarantee spatial isolation, temporal safety, and `TLB` consistency as defined in the chapter? A SOSP or OSDI publication would contribute the first fully verified concurrent OS memory manager, building on the Iris or Perennial separation-logic framework and validated against the x86-TSO and RISC-V weak-memory models.

Lock-free kernel memory management on weak-memory architectures. The lock-free algorithms assume sequentially consistent atomics or strong hardware guarantees. On ARMv8 or RISC-V with weak memory ordering, `Treiber`-stack operations require careful load-link/store-conditional or acquire-release semantics to avoid subtle reordering bugs. Can we characterize the minimal memory barrier fences required for correctness on ARMv8 and RISC-V, and implement an architecture-portable lock-free memory manager using C11/C++20 atomics? For EuroSys or ASPLOS, the novelty is a formal mapping of each operation to the weakest sufficient ordering (e.g., `memory_order_acq_rel` for `Treiber`-stack push, `memory_order_relaxed` for the generation-counter read on fast path) with proof sketches and an empirical evaluation on ARM Ampere and RISC-V cores showing at most 5% overhead versus the x86 implementation.

Real-time latency bounds for lock-free page allocation under adversarial allocation patterns. Lock-free algorithms can suffer from livelock or unbounded retries under high contention. For a real-time kernel variant, what is the worst-case latency of a `CAS`-based `Treiber`-stack allocation? Can we design a wait-free page allocator (where every operation completes in a bounded number of steps) using helping mechanisms or combining, while retaining strong isolation? Targeting RTSS or EuroSys, the contribution is a formal worst-case analysis: for N concurrent cores, the allocator completes in $O(N)$ steps, and the VM mapping completes in $O(L)$ steps where L is the page-table depth, matching the best-known bound for wait-free data structures.

Hardware-accelerated generation-counter TLB shootdown. The deferred invalidation scheme avoids IPIs but may cause unnecessary `TLB` flushes on every context switch if the counter changes frequently. Can we propose a minimal hardware extension where each core maintains a hardware generation-counter register checked by the `TLB` refill walker, so that a `TLB` entry is automatically invalidated on refill if its generation tag (stored with each `TLB` entry) is older than the core's current generation? For ISCA or MICRO, the novelty is treating `TLB` entries as cached data with coherence-like invalidation; the expected contribution is a cycle-accurate simulation showing 30–60% reduction in `TLB` miss handling overhead for memory-intensive workloads, and a RISC-V `gem5` implementation.

Chapter 22

OS-Scheduler and Memory Manager Co-Design for Cache-Coherent NUMA

Can we design a combined scheduler+memory manager that treats cache coherence as a resource to be managed rather than as an invisible substrate?

Definition 22.1 (Cache Coherence Cost)**.** *Cache coherence cost* is the overhead incurred when a core accesses data residing on a remote NUMA node, including snoop and directory probe traffic, data transfer over the interconnect, invalidation of stale copies, and directory controller pressure. Each access falls into one of three categories: local hit (zero coherence overhead), remote hit on the same node (protocol overhead), or remote miss to a different node (full coherence transaction).

Definition 22.2 (Joint Thread-Page Placement Problem)**.** Given threads with access frequencies to memory regions, memory nodes with capacity, compute nodes with capacity, and a coherence cost matrix, *joint thread-page placement* assigns threads to compute nodes and pages to memory nodes to minimize total coherence cost subject to capacity constraints. This is a quadratic assignment problem, NP-hard via reduction from graph partitioning.

On NUMA systems with cache coherence, the OS scheduler and memory manager operate independently. The scheduler places threads on cores aiming for load balance without considering where the thread's data resides. The memory manager places pages on NUMA nodes aiming to place pages near the accessing thread, but reacts to where the scheduler placed the thread. This separation creates a circular dependency: the scheduler sees a thread running on a node and keeps it there because its data is there, but the reason the data is there is that the memory manager saw the thread running there previously. When the scheduler moves the thread, the memory manager eventually migrates its data, reinforcing the migration; however, the migration cost in coherence traffic, `TLB` shootdown, and page copying could have been avoided.

Cache coherence hides this mismatch: even when data is remote, coherence ensures correct data by fetching over the interconnect. But this consumes bandwidth, directory capacity, and invalidation bandwidth. We propose to treat cache coherence as a resource to be managed, accounting for coherence traffic, directory pressure, and invalidation overhead, and jointly optimize thread placement and page placement to minimize coherence overhead while maintaining load balance.

22.1 Related Work

Approach	Strength	Gap
AutoNUMA (Linux)	Page migration based on NUMA fault statistics	Heuristic; no coherence cost model
NUMA balancing (Linux numactl, mbind)	Manual page/thread placement	No automated co-optimization
Carrefour (Dashti et al.)	Locality-aware data placement for NUMA	No coherence model
Coherence-aware scheduling (Blagodurov, 2010)	Scheduler avoids cross-node data access	No page migration
Memory-aware scheduling (Zhuravlev, 2010)	Scheduler considers memory controller load	No coherence traffic model
Thread + data co-scheduling	Combined placement of threads and pages	Offline analysis; not online adaptive

No OS-level framework jointly optimizes thread scheduling and page migration with an explicit model of cache coherence cost.

22.2 Proposed Approach

We first calibrate a coherence cost model on real hardware using performance counters for remote data access, snoop and directory probe counts, and latency measurements via timed memory access. The cost of each memory access depends on whether it is a local hit (zero coherence overhead), a remote hit on the same node (coherence protocol overhead), or a remote miss to a different node involving directory lookup, data transfer, invalidation, and directory pressure costs.

The central research question is formulating the joint thread-page placement as a quadratic assignment problem. Given threads with access frequencies to memory regions, memory nodes with capacity, compute nodes with capacity, and a coherence cost matrix, we assign threads to compute nodes and pages to memory nodes to minimize total coherence cost subject to capacity constraints. This is NP-hard via reduction from graph partitioning, but practical instances with up to a thousand threads and sixteen nodes can be solved approximately using graph partitioning, Kernighan-Lin iterative improvement starting from first-touch placement, or integer linear programming for small instances.

Design Principle: Coherence Cost as a First-Class OS Resource

The OS should account for coherence traffic, directory pressure, and invalidation overhead with the same precision with which it accounts for CPU utilization and memory capacity. Just as the scheduler maintains per-core run queues, a coherence-aware OS maintains a per-node coherence cost matrix that feeds into a unified placement optimizer. Treating coherence as a first-class resource enables the OS to make explicit trade-offs between load balance and data locality.

For online adaptation, we design a locality gradient descent algorithm. At each rebalancing interval (approximately 100 milliseconds) we compute the coherence benefit of moving a thread to

a page's node or a page to a thread's node, compute the movement cost of thread migration (cache warm-up) versus page migration (copy plus `TLB` shootdown), and perform the move if the expected coherence savings over the next interval exceed the movement cost. The gradient of coherence cost with respect to placement changes guides movement in the direction of negative gradient subject to capacity constraints and a movement budget to avoid thrashing.

We develop techniques for the OS to infer directory pressure without hardware support:

- Latency spike detection: a significant increase in remote access latency indicates directory pressure.
- Coherence event counters from architecture-specific performance monitoring units.
- Kernel-based estimation tracking the number of shared versus private pages per process.

The hardest challenges are that coherence cost is hard to measure precisely because performance counters are architecture-specific and may not capture all costs, and that the online algorithm may oscillate if the scheduler and memory manager both make decisions based on each other's previous decisions. Movement cost estimation depends on working set size and whether pages are dirty or shared. Workloads with no locality such as `GUPS` and `Graph500` require detecting the absence of locality and falling back to round-robin placement.

22.3 Evaluation

Benchmark	Metric	Comparison
NPB CG (communication-heavy)	Execution time, coherence traffic	Linux, Carrefour, manual
NAS LU (data sharing)	Execution time	Linux
SPEC rate (16 copies)	Aggregate throughput	Linux, manual
Memcached (multi-instance)	Requests/sec, tail latency	Linux
GUPS (random access, high coherence)	GUPS	Linux
Directory pressure experiments	Probe count (UNC counters)	Linux

We modify Linux by adding a coherence-aware load balancer to the scheduler and replacing `AutoNUMA`'s heuristic page migration with coherence-cost-driven migration, coordinated by a joint decision module running periodically. Evaluation uses NAS Parallel Benchmarks, SPEC CPU2017 in rate mode, cloud workloads including Memcached, Redis, and PostgreSQL, and HPC workloads including `Graph500` and `GUPS`. The target is a twenty to forty percent reduction in coherence traffic on communication-heavy workloads with at most five percent overhead on communication-light workloads. If successful, the framework extends to `CXL` memory tiers with different coherence semantics, power-aware coherence trading traffic for latency, and memory-side caching architectures.

22.4 Research Directions

Online learning of the coherence cost matrix. The proposed quadratic assignment formulation requires a known coherence cost matrix C_{ij} between compute nodes i and memory nodes j, but real

coherence costs depend on dynamic interconnect utilization, directory occupancy, and contention. Can we replace the static matrix with an online learned model using lightweight performance-counter readings (e.g., `UNC_H_REQUESTS_REMOTE` on x86) and a Bayesian linear regressor that predicts $\hat{c}_{ij}(t)$ at each rebalancing interval? For ASPLOS or EuroSys, the contribution is a regret-bound analysis showing that the online learned model converges to within 10% of the optimal static model within $O(\sqrt{T})$ rebalancing steps, validated on a 4-socket AMD EPYC system.

Joint scheduling and page placement with heterogeneous coherence domains (CXL). `CXL` memory introduces new coherence domains with different semantics: `CXL.io` is non-coherent, `CXL.cache` is device-coherent, and `CXL.mem` is host-coherent with a shared directory. An OS scheduler and memory manager co-design must now consider pages that are local to a `CXL`-attached accelerator vs. host DRAM vs. far memory, each with distinct coherence costs and bandwidth profiles. For SOSP or OSDI, the novelty is extending the quadratic assignment model to k coherence domains, each with cost $C_{ij}^{(d)}$, and an online algorithm that detects a workload's coherence-domain affinity (by profiling cache miss rates per domain) and co-locates threads and data accordingly. Expected contribution: 15–25% throughput improvement for `CXL`-memory-intensive datacenter workloads (Memcached, Redis) on a real `CXL`-enabled platform.

Coherence-aware virtual machine scheduling for cloud providers. In a cloud setting, the hypervisor schedules VMs across physical cores without visibility into each VM's coherence traffic patterns. Can we design a coherence-aware Xen or KVM scheduler that exposes a `coherence_profile` hint (read-mostly, producer-consumer, migratory sharing) per VM, inferred from LLC miss rates and remote probe counts, and uses it to colocate VMs that share data or separate VMs that contend on cache lines? For ATC or EuroSys, the contribution is a 10–20% reduction in total coherence traffic in a multi-tenant datacenter with mixed colocated workloads (e.g., Memcached + Spark), formalized as a graph-coloring problem where edge weights are estimated coherence contention between VM pairs.

Power-aware coherence traffic management. Cache coherence traffic consumes significant interconnect power: each directory probe, snoop, and data transfer burns energy proportional to the coherence message size and distance. Can we extend the scheduler+memory manager to include a power budget P_{budget} per socket, and dynamically trade coherence traffic for longer local computation (e.g., by pinning communicating threads to the same core or same L2 partition) when power is constrained? For ISCA or MICRO, the novelty is a three-way co-optimization of performance, coherence traffic, and power; the expected contribution is an MDP-based controller that reduces coherence-related energy by 20–35% with $< 5\%$ performance loss on NAS Parallel Benchmarks.

Chapter 23

Verifying Rust OS Kernel Safety Across Hardware Boundaries

Can we extend Rust's ownership model with a hardware-aware type system that makes DMA, MMIO, and interrupt interactions safe and verifiable?

Rust's ownership and borrowing rules guarantee memory safety and data-race freedom for pure software. But when code interacts with hardware, the guarantees break. DMA allows a device to write to memory without CPU involvement, creating an implicit aliased write that violates the borrowing rules. MMIO reads and writes have side effects that the compiler cannot see, leading to incorrect optimizations such as combining two reads or eliminating a dead write. Interrupt handlers run asynchronously on any core, accessing data that the interrupted thread may be in the middle of modifying. Current approaches use raw unsafe blocks and manual discipline, leaving the hardest part of kernel developmentcorrect hardware interactionunverified.

We propose a hardware-aware type system extension for Rust that models DMA buffers as owned by the device for a specified duration, models MMIO regions with effect types that prevent compiler reordering and elimination, and models interrupt handlers as concurrent threads with explicit synchronization requirements.

23.1 Related Work

Approach	Strength	Gap
Tock OS register abstraction (typesafe MMIO)	Compile-time checked MMIO register access	No DMA model; no interrupt model
Linux kernel Rust (kernel abstractions)	Practical, growing adoption	DMA uses raw pointers; interrupt handlers are unsafe
Hopter (finite-stack, soft-locks)	Rust-based embedded RTOS	No hardware type extensions
seL4 (C/Isabelle verification)	Machine-checked proof of DMA safety	Requires full OS verification; not compositional
Singularity / SIPs	Process-level isolation for hardware access	IPC overhead; not integrated with Rust ownership
Capability-based DMA (CHERI, IOMMU)	Hardware-enforced DMA isolation	Requires special hardware

No type system extension for Rust provides a comprehensive verified model for DMA, MMIO, and interrupts.

23.2 Proposed Approach

We model DMA operations as ownership transfers between the CPU and the device.

Definition 23.1 (DMA Ownership Model)**.** A `DmaBuf<T>` is owned by the CPU initially. Submitting it for transfer moves ownership to the device, producing a `DmaHandle<T>` and making the buffer inaccessible from the CPU side. When the device signals completion via an interrupt, `return_buffer` on the handle returns ownership to the CPU. The type system enforces:

1. `DmaBuf<T>` cannot be accessed while a corresponding `DmaHandle<T>` exists;
2. `DmaHandle<T>` is uniquely owned (no aliasing);
3. streaming DMA uses split borrows where the device borrows a prefix of the buffer while the CPU retains access to the suffix.

A ‘DmaBuf‘ is owned by the CPU initially; submitting it for transfer moves ownership to the device, producing a ‘DmaHandle‘ and making the buffer inaccessible from the CPU side. When the device signals completion via an interrupt, ‘return_buffer‘ on the handle returns ownership to the CPU. The type system enforces that ‘DmaBuf‘ cannot be accessed while a ‘DmaHandle‘ exists and that ‘DmaHandle‘ is uniquely owned. For streaming DMA where the device fills a buffer while the CPU reads parts the device has already filled, we use split borrows analogous to Rust’s ‘split_at_mut‘ but with the device as a borrower.

For MMIO, we define effect types: ‘MmioRead‘, ‘MmioWrite‘, ‘MmioReadWrite‘, and ‘MmioReadClear‘ are wrapper types that implement access with LLVM volatile load and store and noalias semantics, preventing reordering, write combining, and dead write elimination. The type system prevents redundant reads of read-clear registerseach read consumes a resource that is only regenerated by a writeusing effect tracking, a form of linear typing.

For interrupt handlers, we introduce ‘InterruptCell¡T¿‘ that enforces that all accesses to the contained data use atomic operations with specified ordering, or are inside a critical section with interrupts disabled or a spinlock held. Data shared between the main thread and interrupt handlers must be declared with this type, and the type system ensures that interrupt handlers cannot access shared data without synchronization.

The central research question is determining the minimum unsafe surface area needed for hardware interaction.

Theorem 23.1 (Hardware-Safe Type Soundness)**.** *Given a Rust program P extended with `DmaBuf<T>`, `MmioCell<T>`, and `InterruptCell<T>` types, if P passes the hardware-aware type checker that enforces (1) no CPU access to DMA buffers owned by devices, (2) no redundant reads of read-clear MMIO registers, and (3) no data races between interrupt handlers and main-thread code, then P is free of DMA-safety violations, MMIO-ordering violations, and interrupt-related data races at runtime.*

Proof sketch. The proof proceeds by a syntactic translation from the extended Rust type system into a core language with linear types and effect tracking. DMA ownership is modeled as a linear resource `OwnedBy(CPU)` / `OwnedBy(Device)`; the type system guarantees that `OwnedBy(Device)`

resources are inaccessible to CPU code. MMIO effects are tracked using a finite-state automaton per register, where reads of read-clear registers must follow a write. Interrupt safety is reduced to a rely-guarantee logic where each handler's interference is bounded by the declared atomicity. □

The type system extension itself uses unsafe at the lowest layer where device drivers access hardware registers directly, but the higher-level abstractions such as DMA buffer pools, MMIO register sets, and interrupt service routine frameworks are safe once verified. We build a Rust compiler plugin using the Rust compiler's internal API that verifies DMA correctness by checking that 'DmaBuf' is not accessed while the device owns it, MMIO effect correctness by checking that read-clear registers are not read twice without an intervening write, and interrupt data-race freedom by tracing control flow paths that enter and exit interrupt context.

The hardest challenges are that real DMA uses ring buffers, scatter-gather lists, and partial completion, requiring a richer ownership model than simple whole-buffer ownership, and that MMIO effect tracking requires linear types which Rust does not natively support for stack variables, so the verification plugin must add this capability. Interrupt handlers on multicore systems where the same handler runs on two cores simultaneously require either provably reentrant handlers or hardware inter-core interrupt masking.

Core Thesis

Extending Rust's ownership model with hardware-aware types (DMA ownership transfer, MMIO effect tracking, interrupt-handler synchronization) reduces the unsafe surface of kernel hardware interaction from arbitrary unchecked code to a small, verifiable core, bringing Rust's safety guarantees to the most error-prone part of OS development.

23.3 Evaluation

Criteria	Method	Target
DMA safety	Fuzz DMA operations with concurrent device behavior	0 races found
MMIO safety	Compile known-buggy register sequences; verify rejection	100% rejection
Interrupt safety	Inject interrupt at random points; verify no data races	0 races in 10K tests
Unsafe surface	Count unsafe lines vs. safe lines	$\leq 1\%$ unsafe
Performance	Measure DMA throughput, MMIO latency, interrupt latency	$< 5\%$ overhead vs. raw unsafe
Expressiveness	Can we express DMA ring buffers, descriptor chains, interrupt coalescing?	Yes (all patterns covered)

We implement the abstractions for a RISC-V embedded system with UART, SPI, and I2C peripherals using MMIO register sets, a DMA controller with scatter-gather DMA, and a timer

with interrupt handler. Evaluation uses fuzz testing for DMA and interrupt safety, compile-time rejection tests for MMIO safety, and performance measurement against raw unsafe implementations. The target is less than five percent performance overhead and at most one percent unsafe code lines. If successful, the approach extends to IOMMU integration where DMA ownership drives IOMMU page table setup and teardown, interrupt handler inlining and atomicity verification for short handlers, and GPU or CXL device memory with different coherence models.

23.4 Research Directions

Ownership-based IOMMU page table management. Currently, DMA ownership transfers drive correctness but the IOMMU page tables are managed separately via unsafe IOMMU API calls. Can we extend the type system so that creating a `DmaBuf` for a device automatically installs the corresponding IOMMU mapping, and returning ownership via `return_buffer` tears it down? The type-system invariant is that the IOMMU page table is always consistent with the ownership state. A OSDI or SOSP publication would contribute a Linux Rust kernel module where the unsafe surface for IOMMU operations is exactly two functions (map/unmap physical pages), and the abstraction prevents DMA to already-freed pages by construction, verified through a formalization in Coq of the ownership-IOMMU correspondence.

Verification of interrupt handler reentrancy and priority inversion freedom. The `InterruptCell<T>` abstraction prevents data races between main-thread and handler code, but does not address reentrancy (the same handler firing concurrently on two cores) or priority inversion (a low-priority handler delaying a high-priority one by holding a spinlock). Can we design a type-level annotation `#[interrupt(priority = N, reentrancy = false)]` and a static analyzer that verifies, for each interrupt-priority level, that no lock is held across a higher-priority interrupt boundary? For PLDI or IEEE S&P, the novelty is combining data-race freedom *and* deadlock freedom in a single type system extension, with a soundness proof using a rely-guarantee logic. The expected contribution is a zero-overhead Rust crate deployed on an embedded RTOS (Tock or RTIC) that eliminates priority-inversion bugs by construction.

Effect types for non-coherent device memory (GPU, CXL). Current MMIO effect types assume strongly-ordered MMIO. But GPU device memory and CXL.mem are cache-coherent or weakly-ordered, requiring fences and acquire-release semantics for correctness. Can we generalize the effect-type system to model device memory with coherence mode $\mathcal{M} \in \{\text{MMIO}, \text{CXL.cache}, \text{CXL.mem}, \text{GMem}\}$, where each mode defines a different memory-ordering algebra? For ASPLOS or MICRO, the contribution is a Rust trait hierarchy (`MmioAccess`, `CxlCacheAccess`, `GpuMemAccess`) with compile-time ordering checks, validated on a real CXL-attached FPGA and an NVIDIA GPU, showing that the type system catches 100% of injected ordering bugs while adding $< 3\%$ performance overhead.

Safe DMA ring buffers with ownership borrowing. Many real devices use DMA ring buffers where the CPU writes descriptors and the device reads them, then the CPU reads completion statuses while the device writes new ones: a split-borrow pattern across the CPU–device boundary. Can we extend Rust's split-borrow semantics to model producer-consumer ring buffers where the producer and consumer are different agents (CPU vs. device)? Using a `DmaRing<T, N>` type with `producer_slot()` and `consumer_slot()` methods that statically guarantee no read–write overlap, this would enable end-to-end verification of NIC and NVMe drivers. For OSDI or EuroSys, the expected contribution is a verified NVMe driver with 99% safe Rust code and zero DMA-related

bugs in fuzz testing, compared to 12 bugs found in the equivalent Linux C NVMe driver in the literature.

Chapter 24

Minimal OS Abstraction for Disaggregated Memory

Can we design an OS abstraction that makes disaggregated memory pools appear as locally-attached memory to unmodified applications, without unacceptable performance loss?

Definition 24.1 (DM-VM Abstraction)**.** *DM-VM* (Disaggregated Memory Virtual Memory) is an OS abstraction that presents a pool of disaggregated memory as unmodified local memory within a single virtual address space. The kernel transparently decides at page fault time whether to allocate from local DRAM or from the disaggregated pool based on memory pressure, access frequency prediction, and pool capacity. Applications require no code changes.

Definition 24.2 (Disaggregated Memory Pool)**.** A *disaggregated memory pool* is a centralized or distributed store of memory capacity decoupled from compute nodes. Pages in the pool are accessed over a fabric interconnect (`CXL` (coherent, cacheable) or `RDMA` (non-coherent, message-based)) and are allocated on demand. The pool exposes effectively unlimited capacity at higher latency than local DRAM.

Disaggregated memory promises to improve memory utilization in datacenters by decoupling compute and memory. Instead of each server having a fixed amount of memory, memory is pooled and dynamically allocated to compute nodes as needed. However, current systems require application or OS modifications. `CXL` memory shows up as a separate NUMA node requiring mbind or numactl. `RDMA` memory requires special APIs such as rlib, FaRM, or AIFM. The ideal would be the OS presenting disaggregated memory as unmodified local memory with a single address space, while transparently handling page allocation, migration, access, partial failure, and coherence.

We propose the `DM-VM` (Disaggregated Memory Virtual Memory) abstraction: each process has a normal virtual address space backed by local DRAM, but the kernel transparently decides at page fault time whether to allocate from local DRAM or from the disaggregated pool based on memory pressure, access frequency prediction, and pool capacity.

24.1 Related Work

Approach	Strength	Gap
CXL memory on Linux (TPP, AutoNUMA)	Transparent page migration between DRAM and CXL	Exposes CXL latency; no failure handling
RDMA memory pooling (rlib, FaRM, AIFM)	High performance for RDMA	Not transparent; special APIs required
Ghost memory (Eryilmaz, 2024)	Transparent compressed memory via OS	Compression, not disaggregation
Partially disaggregated memory (CXL allocator)	Transparent for allocations	Allocator-level; not page-level
Disaggregated memory over RDMA (ScaleMP, vSMP)	SMP over RDMA	Not transparent; requires kernel module

No OS abstraction provides transparent disaggregated memory with application transparency, failure handling, and acceptable performance.

24.2 Proposed Approach

At each page fault, the kernel decides where to allocate the page. If it chooses the disaggregated pool, the mapping may be direct-mapped over `CXL` with caching and coherence or demand-paged over `RDMA` where the page is fetched from the pool on fault. The kernel transparently migrates pages between local DRAM and the pool based on access frequency.

The central research question is the page migration policy for the asymmetric capacity and latency profile of local DRAM combined with a disaggregated pool that has effectively unlimited capacity but high latency. We extend the optimal migration model from two-tier memory systems:

1. Pages with access frequency above a threshold go in local DRAM.
2. Pages with access frequency below the threshold go in the pool.
3. The threshold depends on migration cost, the latency difference, and the monitoring interval.
4. For three tiers (local DRAM, `CXL`, and `RDMA` pool), pages with the highest frequency go in DRAM, medium in `CXL`, and lowest in the `RDMA` pool.

For partial failure handling, the kernel maintains a heartbeat with the pool controller. If no heartbeat response arrives within a timeout, the pool is declared possibly failed. Page faults to pool pages during this period block in an interruptible sleep. If the pool is confirmed permanently failed, the kernel sends `SIGBUS` to the process. If the pool recovers, the kernel sends a batch request for all pages that blocked threads were waiting for and wakes the threads. The kernel maintains a blocked page table for pages under failure, and the pool controller keeps a replay log of recent writes.

For non-coherent disaggregated memory such as `RDMA`, the OS provides coherence at page granularity. When a process accesses a page from the pool, the OS ensures it is up to date. Only one node may have writable access to a page; other nodes must be read-only. The OS maintains a distributed page table mapping which node holds the authoritative copy of each page, using a

distributed lock manager or the fabric's atomic operations to coordinate ownership. When a node wants to write to a shared page, it sends an invalidation request to the pool, which invalidates other nodes' copies.

Design Principle: Transparent Migration with Latency-Aware Tiering

The OS should hide disaggregated memory behind the standard virtual memory interface and rely on access-frequency monitoring to migrate pages between tiers. The key observation is that most workloads have a skewed access distribution: a small fraction of pages accounts for the majority of accesses. By keeping only the hot working set in local DRAM and demoting cold pages to the disaggregated pool, the `DM-VM` abstraction delivers performance within 20% of all-local-DRAM for workloads whose active working set fits in local DRAM 80% of the time.

The hardest challenges are balancing failure detection latency against false positives (a short heartbeat interval detects failures quickly but may falsely declare failure during transient network congestion) and the fact that `RDMA` coherence is expensive, requiring page-granular invalidations and `TLB` shootdowns that may be prohibitive for workloads with fine-grained sharing.

24.3 Evaluation

Benchmark	Metric	Comparison
Memcached	Ops/sec, tail latency	Local DRAM, CXL NUMA, FaRM
Redis	GET/SET latency	Local DRAM, CXL NUMA
PostgreSQL (TPC-C)	Throughput, latency	Local DRAM, CXL NUMA
Spark (sort, word count)	Completion time	Local DRAM (with enough memory)
Failure injection	Recovery time, app disruption	No failure handling (crash)
Coherence overhead	Page fault rate, coherence traffic	CXL hardware coherence

We implement `DM-VM` in Linux as a kernel module plus mmap extension, with a `CXL` pool driver and an `RDMA` pool driver. Evaluation uses macrobenchmarks including Memcached, Redis, PostgreSQL, and Apache Spark, microbenchmarks for latency and bandwidth, failure injection tests, and coherence overhead measurements. The target is performance within twenty percent of local DRAM for workloads whose active working set fits in local DRAM eighty percent of the time. If successful, the approach extends to security isolation with encryption or access control in the pool controller for multi-tenant datacenters, dynamic pool resizing for planned capacity changes, and memory pooling across failure domains in rack-scale systems.

24.4 Research Directions

Application-transparent disaggregated memory with failure prediction. The current failure-handling design is reactive: the OS detects heartbeat loss and blocks or signals the process. Can we predict impending pool failures (via `SMART` metrics, latency degradation trends, or network

congestion signals) and proactively migrate hot pages out of the pool before failure occurs? The ideal system maintains availability: for a pool failure predicted T seconds in advance, the OS migrates the k hottest pages per process, reducing the probability of a `SIGBUS` from p to $p \cdot e^{-T/\tau}$ where τ is the page-fault inter-arrival time. A SOSP or OSDI publication would contribute the first predictive failure-handling system for disaggregated memory, with an evaluation on a `CXL` testbed showing zero observable application disruption for 95% of injected failures.

Multi-pool scheduling and page placement across heterogeneous disaggregated memory. A datacenter may have multiple disaggregated memory pools: some are fast `CXL`-attached near a specific rack, some are slow `RDMA` pools in a different row, and some are archival NVDIMM pools. Can we extend `DM-VM` to model multiple pools with heterogeneous latency, bandwidth, capacity, and failure-rate profiles as a k-tier memory hierarchy, and assign each page fault to the optimal pool by solving $\min_p L_p + \frac{S}{B_p} + \lambda \cdot M_p$ where L_p is latency, B_p bandwidth, S access size, M_p monetary cost per GB, and λ a cost–performance trade-off? For EuroSys or ASPLOS, the contribution is a multi-pool online optimizer with sublinear regret, validated on a testbed with 3 pools (local DRAM, `CXL`, `RDMA`) running memcached and Spark, demonstrating 30% cost reduction at equal performance.

Disaggregated memory with kernel-bypass and userspace page fault handling. The `DM-VM` abstraction requires kernel mediation on every page fault to the disaggregated pool, adding latency. Can we design a kernel-bypass path where the application registers a userspace page-fault handler (via `UFFD` or a new mmap flag), and the handler fetches pages from the `RDMA` pool directly using userspace `RDMA` verbs without kernel involvement? The kernel only intervenes to set up the initial mapping and to handle pool failures. For ATC or EuroSys, the novelty is a split-level design where 99% of page faults to cold pool pages are handled entirely in userspace, reducing fault latency by 5–10× versus kernel-mediated faults. The expected contribution is a Linux prototype and evaluation showing that Memcached with 50% of pages in the pool performs within 10% of all-local-DRAM at the 99th percentile.

Secure disaggregated memory: encryption, access control, and isolation in multi-tenant pools. When multiple tenants share a disaggregated memory pool, one tenant must not read another's data. Can we extend `DM-VM` with per-page encryption keys managed by the pool controller, and access-control lists enforced at the pool controller's memory controller? The OS provides each tenant with a tenant ID at mmap time, and the pool controller checks that a read/write request includes a valid capability for that page. For IEEE S&P or OSDI, the contribution is a capability-based memory security model for disaggregated pools with measured throughput overhead below 10% for AES-GCM-SIV encryption at the memory controller, and formal proof that the capability system prevents cross-tenant access even under a malicious tenant OS.

Part V

High Performance Computing

Chapter 25

Universal Lock-Free Parallel Algorithm Framework

Can we characterize the exact class of problems solvable via the lattice-linear predicate framework, and build a compiler that reads a problem description and emits a provably lock-free parallel algorithm?

The LLP-FW framework showed that a class of optimization problemsincluding SSSP, BFS, stable marriage, knapsack, and job schedulingcan be solved with a generic lock-free parallel algorithm. The key insight is that these problems have lattice-linear predicates: the predicate defining a solution is a linear function over a lattice of states, and any violation of the predicate by one component can be fixed independently by that component without violating other components' progress. This suggests many fundamental problems share a common algebraic structure amenable to lock-free parallelism. But the current understanding is empirical: we do not know why these specific problems have LLPs, whether all problems with LLPs share a deeper common structure, how to recognize whether a new problem has an LLP, or what the boundary is that separates LLP-solvable problems from those that are not.

Problem 25.1 (Lattice-Linear Predicate Characterization)**.** Given a problem specification $\Pi = (X, \mathcal{C}, \preceq)$ where $X = \{x_1, \ldots, x_n\}$ is a set of variables over a lattice $(L, \sqcup, \sqcap)$, $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints over X, and $\preceq$ is a partial order on X, determine whether Π admits a lattice-linear predicate P such that (i) P is false iff $\exists$ a set of witness variables whose reassignment can make P true, and (ii) the minimal correction is unique per witness. If yes, synthesize a lock-free parallel algorithm with correction functions $\{f_1, \ldots, f_n\}$ that converge to a fixed point under concurrent execution.

We propose to characterize the class of LLP problems formally in terms of logic, order theory, or descriptive complexity, and build a compiler that given a problem specification automatically determines whether it has an LLP and generates a lock-free parallel algorithm.

25.1 Related Work

Approach	Strength	Gap
LLP-FW (Kumar, 2026)	Generic lock-free runtime for several problems	Empirical; no characterization of the problem class
Lattice-linear predicates (Kumar, 2024)	Introduced the concept of LLPs	No connection to logic or complexity theory
Synchronization via lattice theory (Agha, 1990s)	Actor model with lattice-based semantics	Not focused on lock-free parallelism
Parallel algorithmic skeletons	Reusable parallel patterns	LLPs are not among them
Automatic parallelization (polyhedral, speculative)	Parallelizes regular loops	Irregular optimization problems not covered
Local computation algorithms (LCA)	Sublinear-time parallelizable algorithms	Restricted to specific problem structures

No formal characterization exists of the class of problems solvable via lattice-linear predicates, and no compiler generates lock-free algorithms from problem specifications.

25.2 Proposed Approach

Monotone Existential Second-Order Logic

The key insight is that LLPs correspond exactly to problems definable in monotone existential second-order logic over a lattice order. This connects lock-free synchronization to descriptive complexity theory: the witness set in the logical formulation corresponds exactly to the set of variables that can be corrected independently, providing a syntactic test for LLP-solvability.

We formally define a lattice-linear predicate over a lattice: a predicate P is lattice-linear if for every state, P is false if and only if there exists a set of witness variables such that changing only those variables can make P true, and the minimal change in the lattice order that makes P true is unique and depends only on the values of the witnesses. We hypothesize that LLPs correspond exactly to problems definable in monotone existential second-order logic with a lattice order, giving a syntactic test: if a problem can be expressed in this logic, it has an LLP. The central research question is whether LLPs can be characterized as the conjunction of independent local predicates, each depending only on a single variable and its neighbors in a dependency graph, which would connect LLPs to locally checkable labelings and local computation algorithms.

We design a decision procedure for the LLP problem. Given a problem specification as an optimization over a set of constraints, we construct the dependency graph of variables from the constraints, check whether the objective function is separable as a sum of functions each depending on a single variable or connected component, and check whether the constraints are monotone with respect to some order. If both conditions hold, the problem has an LLP. A key question is whether this sufficient condition is also necessaryif LLP is exactly the class of separable monotone optimization problems.

We build a compiler that takes a problem specification in a formal language and generates a lock-free parallel algorithm. The compiler parses the specification into a variable-constraint graph,

Algorithm 7 LLP Compiler: Detection and Code Generation

Require: Specification $\Pi = (X, \mathcal{C}, \preceq)$
Ensure: Lock-free parallel algorithm or rejection
$G \leftarrow$ BuildDependencyGraph$(X, \mathcal{C})$
if IsSeparableMonotone$(\mathcal{C}, \preceq)$ **then**
    order $\leftarrow$ InferLatticeOrder$(\preceq)$
    for each variable x_i **do**
        $f_i \leftarrow$ DeriveCorrection(C_i, order)
        Emit: `while true do` $x_i \leftarrow \text{CAS}(x_i, f_i(\text{neighbors}))$
    end for
    return algorithm with $\{f_i\}$ executing concurrently
else
    return "Not LLP-solvable; try heuristic parallelism"
end if

checks the LLP conditions, and if they hold, derives the local correction function for each variable and emits a parallel algorithm where all correction functions execute concurrently using atomic CAS on variable values, with the lattice order ensuring termination. If LLP does not hold, the compiler reports that the problem is not LLP-solvable and suggests alternative parallelism strategies.

We prove completeness results: every LLP problem is solvable in polynomial time sequentially because the correction function is deterministic and monotone, bounded by the number of variables times the lattice height. We identify a candidate LLP-complete problemminimum spanning tree in a complete graph with edge weightsthat captures the class. We extend LLPs to partial LLPs where the predicate is lattice-linear on a sublattice, capturing problems like BFS where the order of visiting nodes is not fixed, and randomized LLPs with randomized correction functions for problems like simulated annealing.

The hardest challenges are that the LLP decision boundary is unclearwe do not know if LLP is exactly separable monotone optimization or a larger classand that the compiler must be sound, requiring a formal proof of the compilation steps as complex as the LLP-FW correctness proof itself. LLP algorithms may not be work-efficient, executing many more correction function applications than the sequential algorithm due to conflicts and retries.

25.3 Evaluation

Problem	Source	Metric	Baseline
SSSP	LLP-FW benchmark	Speedup over sequential, compilation time	LLP-FW (hand-coded)
BFS	LLP-FW benchmark	Speedup	Hand-optimized (PBBS)
Stable marriage	LLP-FW benchmark	Speedup	LLP-FW
Knapsack	LLP-FW benchmark	Speedup	LLP-FW
Graph coloring (new)	Custom	Is it LLP? Speedup if yes	Hand-optimized
Synthetic CSPs	Random generation	LLP detection accuracy	Brute-force LLP check
All	ALL	Correctness verification	Brute-force for small N

We implement the LLP compiler and generate lock-free parallel algorithms for known LLP problems, new problems such as graph coloring to test the boundary, and synthetic constraint satisfaction problems with controlled LLP properties. Evaluation compares compiler-generated algorithms against LLP-FW manual implementations and hand-optimized parallel algorithms using metrics including speedup, compilation time, problem coverage, and correctness verified against brute force for small instances. If successful, the approach extends to distributed-memory LLP where correction functions become communication operations, dynamic LLPs that adapt incrementally to changing constraints, and LLPs for machine learning problems such as consensus in reinforcement learning and MAP inference in graphical models.

25.4 Research Directions

Formal characterization of the LLP class (SC'27). Prove that LLPs correspond exactly to problems definable in monotone existential second-order logic over a lattice order, extending the connection between descriptive complexity and parallelizability. The novelty lies in bridging finite model theory and lock-free synchronization: if ϕ is a sentence in monotone $\exists$SO whose quantifier prefix respects a lattice order, then the witness set for ϕ defines the variables that can be corrected independently under the LLP framework. This would give a syntactic test for LLP-solvability and open a new classification of optimization problems by their logical complexity. Expected contribution: a characterization theorem and a polynomial-time decision procedure for whether a problem admits an LLP.

LLP compiler with formal correctness guarantees (PLDI'27). Build a verified compiler that takes a problem specification in a formal language (e.g., a set of monotone constraints over a lattice), checks the LLP conditions, and emits a lock-free parallel algorithm whose correctness is mechanically verified in a proof assistant such as Coq. The compiler must prove that the emitted correction functions are confluent and terminating under the lattice order, establishing a direct connection between program synthesis and concurrent correctness. The novelty is a proof-generating

compilation pass for lock-free algorithms, with evaluation showing that generated algorithms match hand-tuned LLP-FW performance within 10%.

LLP complexity theory and completeness (FOCS/STOC'27). Define LLP-completeness via lattice-linear reductions that preserve the structure of correction functions, and identify the first natural LLP-complete problem, conjectured to be minimum spanning tree on complete graphs with distinct edge weights. Prove that every LLP problem is in NC (or AC^0) and that LLP-complete problems are not in NC^1 unless $NC^1 = NC$. This establishes the place of LLP problems within the parallel complexity hierarchy, a novel connection between lock-free algorithms and circuit complexity.

LLPs for machine learning and probabilistic inference (NeurIPS'27). Extend the LLP framework to randomized and stochastic settings where correction functions incorporate a random choice. For MAP inference in pairwise graphical models, the belief propagation update rule has the lattice-linear property when the graph is a tree and the potential functions satisfy a convexity condition; show that this enables a lock-free parallel inference algorithm. Apply the same technique to consensus in multi-agent reinforcement learning, where the value function update for each agent is a correction function on the joint value lattice, achieving lock-free parallel training without synchronization overhead. The contribution is a general recipe for deriving lock-free parallel ML algorithms from the structure of their objective functions.

Chapter 26

Communication-Optimal Distributed ZKP Generation

What are the *fundamental* communication lower bounds for distributed ZKP generation, and can we design algorithms that achieve them?

Zero-knowledge proof (ZKP) generation remains prohibitively expensive: proving a realistic computation such as a zkVM execution, verifiable ML inference, or a large circuit can take minutes to hours on a single machine. Distribution across multiple machines offers a natural path to scaling: split the computation into sub-circuits, generate proofs for each sub-circuit in parallel, and compose them, as done by systems like collaborative zk-SNARKs (HyperPlonk with MPC-based proving), distributed proving for PLONK (Pipelonk, DIZK), and zkPipe. Yet distribution introduces its own costs: network bandwidth for exchanging intermediate values, latency from synchronous rounds of communication, and synchronization overhead for agreeing on shared randomness. Current distributed ZKP systems are designed in an ad-hoc manner, each using a custom distribution strategy without a formal model of communication costs. We lack a theoretical framework for answering fundamental questions: what is the minimum communication needed to generate a ZKP for a given circuit, whether current systems like DIZK are optimal or could be improved by an order of magnitude, and what the tradeoff between computation and communication is in distributed proving. We propose to derive fundamental communication lower bounds for distributed ZKP generation and design algorithms that provably achieve these bounds.

26.1 Related Work

Approach	Strength	Gap
DIZK (Distributed ZKP for R1CS)	First scalable distributed prover for R1CS	No communication bound analysis; uses fixed distribution
Pipelonk (GPU-accelerated PLONK)	10.7× speedup via pipelined operator library	Single-machine multi-GPU; not distributed
Collaborative zk-SNARK for HyperPlonk	MPC-based distributed proving, sublinear communication	Not proven optimal
zkPipe (Distributed zkVM proving)	State-machine-based distribution	Ad-hoc; no formal model
Hydra (Distributed proof composition)	SNARK composition across machines	Composition-level only
Pegasus (Distributed NIZK)	First formal communication model	Limited to specific NIZK constructions
Communication complexity theory	General lower bounds	Not applied to ZKP generation
PCP / IOP lower bounds	Lower bounds for interactive oracle proofs	Different model from concrete ZKP systems

No existing work provides communication complexity lower bounds specifically for distributed ZKP generation, nor a system that provably achieves them.

26.2 Proposed Approach

We define a formal communication model for distributed ZKP.

Definition 26.1 (Distributed ZKP Communication Model)**.** A circuit C with N gates and M wires is partitioned into P sub-circuits $C_1, \ldots, C_P$ assigned to provers $P_1, \ldots, P_P$, each with private input x_i. Provers communicate over secure point-to-point channels. The communication cost is the total bytes sent across all channels and rounds. Each prover generates a partial proof π_i from its sub-circuit; these are combined into a single proof π that the verifier checks against the full circuit C without knowing the partition.

We prove that any distributed prover for circuit C partitioned as $\{C_i\}$ must communicate at least $\Omega(\text{cut} \times \log |F|)$ bits, where $\text{cut}(C, \{C_i\})$ is the number of wires crossing between sub-circuits and F is the field size.

Theorem 26.1 (Communication Lower Bound for Distributed ZKP)**.** *For any distributed prover generating a ZKP for circuit C partitioned into sub-circuits $\{C_i\}$ with crossing cut size $cut(C, \{C_i\})$ over a field F, the total communication across all provers is at least $\Omega(cut \times \log |F|)$ bits. For PLONK specifically, communication is at least $\Omega(cut_{gate} \times \log |F| + cut_{wiring} \times \log |F|)$, where cut_{gate} counts cross-partition gate constraints and cut_{wiring} counts cross-partition wiring constraints.*

Proof sketch. Suppose communication is less than $\Omega(\text{cut} \times \log |F|)$. Then there exists a wire w crossing between sub-circuits C_i and C_j whose value is not communicated. The verifier cannot distinguish a valid proof from one where w's value is changed, since all local proofs are consistent with either value. By a hybrid argument over all crossing wires, the verifier's acceptance probability changes by at most $\text{negl}(|F|)$ when cut values are modified, contradicting soundness. The PLONK-specific bound follows from the fact that cross-partition gate constraints and wiring permutations each require communicating the corresponding committed values. □

if communication were less, the verifier could not distinguish a correct proof from one for a modified circuit with changed cut values.

For specific protocols we prove tighter bounds. In PLONK, the prover computes a gate constraint polynomial, a wiring copy constraint, and a batched opening of polynomial commitments; the communication to compute the constraint polynomial across provers is at least the number of cross-partition gate constraints, and the wiring permutation check must communicate the wiring between partitions. In HyperPlonk, which uses the sum-check protocol with multilinear polynomial commitments, the communication for a partitioned circuit is proportional to the degree of the cross-partition constraints.

We formulate circuit partitioning as a graph problem: given a circuit graph $G = (V, E)$, P provers, and a balance constraint (each sub-circuit has at most $(1 + \varepsilon)|V|/P$ gates), minimize the maximum cut size since the prover with the largest cut determines worst-case communication. We employ spectral partitioning (Fiedler vector) for a balanced initial partition, refined with Kernighan-Lin iterative swapping, parameterized by proof-system-specific constraint weights.

Our distributed prover for PLONK operates in four steps. First, each prover computes partial polynomial commitments for its sub-circuit locally. Second, each prover computes a contribution to each cross-partition constraint and exchanges contributions via a one-shot collect-and-combine protocol. Third, a designated prover aggregates all contributions into the final constraint polynomial, while the permutation check is handled via a Merkle tree. Finally, the designated prover generates the batched opening proof for the combined polynomial, achieving $O(\text{cut} \times \log |F|) + O(P \times \log |F|)$ communication. We compare this cost to the lower bound: if the protocol achieves $O(\text{cut} \times \log |F|)$, it is optimal up to constant factors.

We implement the distributed prover on a multi-node cluster using gRPC or MPI for communication, the optimal partition algorithm, and an extension of Pipelonk's GPU-accelerated prover. Evaluation targets include SHA-256 circuits, Keccak circuits for ZK-EVM, RISC-V zkVM execution traces, and ResNet-32 inference, scaling from 2 to 32 nodes and comparing against DIZK, Pipelonk, Collaborative HyperPlonk, and the theoretical lower bound.

Core Thesis

The fundamental communication cost of distributed ZKP generation is determined by the cut size of the circuit partition: any distributed prover must communicate at least $\Omega(\text{cut} \times \log |F|)$ bits, and our PLONK-based construction achieves this lower bound up to constant factors via a one-shot collect-and-combine protocol for cross-partition constraints.

26.3 Evaluation

Benchmark	Circuit size	Metric	Comparison
SHA-256	256K gates	Communication (bytes), wall-clock time	DIZK, lower bound
Keccak (ZK-EVM)	2M gates	Communication, time	DIZK, lower bound
RISC-V zkVM trace	10M gates	Communication, time	DIZK
ResNet-32 inference (verifiable ML)	5M gates	Communication, time	No distributed baseline
Scaling experiments	10K–10M gates	Communication vs. P nodes	Theoretical bound

We target communication within $2\times$ of the cut-based lower bound across all benchmarks.

26.4 Research Directions

Tight communication lower bounds for specific proof systems (CRYPTO'27). Derive matching lower and upper bounds on communication for distributed generation of PLONK, HyperPlonk, and STARK proofs. For each system, model the prover's computation as a circuit with structured constraints (e.g., permutation networks in PLONK, sum-check in HyperPlonk, FRI in STARKs) and prove that any distributed prover must communicate at least $\Omega(\text{cut} \cdot \log |F|)$ bits where cut is the number of cross-partition constraints. Design protocols that achieve this bound within constant factors, showing that DIZK and Collaborative HyperPlonk are asymptotically optimal for their circuit partitions but suboptimal because their partitions ignore proof-system-specific constraint structure. The contribution is the first tight communication complexity characterization for practical ZKP systems.

Optimal circuit partitioning for distributed proving (SC'27). Formulate circuit partitioning for distributed ZKP as a constrained graph cut problem where the objective is to minimize the maximum cut size across P partitions subject to balance constraints, with edge weights reflecting the proof-system-specific cost of cross-partition constraints (e.g., polynomial commitment overhead, sum-check round complexity). Prove that the problem is NP-hard and give a bicriteria approximation algorithm based on spectral partitioning with recursive bisection, achieving $O(\log P)$-approximation. The novelty is connecting balanced graph partitioning to proof-system cost models; evaluation across SHA-256, Keccak, and zkVM circuits shows 30–50% communication reduction versus random partitioning.

A distributed prover achieving the communication lower bound (HPDC'27). Implement the optimal distributed prover for PLONK using MPI and GPU kernels, achieving

$O(\text{cut} \cdot \log |F|)$ communication via a one-shot collect-and-combine protocol for cross-partition constraints and a Merkle-tree-based wiring check. The system must handle the practical challenges of load imbalance (uneven sub-circuit sizes), straggler effects, and the overhead of establishing secure channels between provers. Evaluate on up to 128 nodes with circuits up to 10M gates, demonstrating communication within 1.5× of the cut-based lower bound and wall-clock speedup of 10× over single-node proving for large circuits. The contribution is the first distributed ZKP system with provable communication guarantees.

Extension to transparent proof systems and STARKs (Eurocrypt'27). Derive communication lower bounds for distributed STARK proving, where the prover must compute a low-degree extension of execution traces and generate Merkle tree proofs for FRI queries. The cross-partition communication is fundamentally different from constraint-based systems: the prover must reconcile boundary constraints between trace segments, requiring communication proportional to the segment boundary width times the extension degree. Design a distributed FRI protocol where each prover commits to its local trace segment, a designated prover runs FRI with batched queries, and provers exchange Merkle paths only for cross-segment consistency. The contribution is a communication-optimal distributed STARK prover with evaluation on RISC-V zkVM traces.

Chapter 27

Extending Performance Models to Chiplet Architectures

Can we extend the Roofline and ECM performance models to accurately capture chiplet-based multi-die architectures with non-uniform cache hierarchies?

The Roofline model (Williams, 2009) and the Execution-Cache Memory (ECM) model (Hager, 2016) are the standard tools for analyzing compute kernel performance on multicore processors, answering whether a kernel is compute-bound or memory-bound, what maximum performance is achievable, and where the bottleneck in the memory hierarchy lies. These models were designed for monolithic chips where all cores share a uniform last-level cache (LLC), memory latency is uniform across all cores, and the interconnect between cores and memory controllers is balanced. Chiplet architecturesAMD EPYC, Intel Xeon with multi-tile, Apple M-seriesfundamentally violate these assumptions: each chiplet has its own LLC slice with higher latency and lower bandwidth for remote access, die-to-die data movement incurs 2–5× higher latency than on-die traffic, die-to-die link bandwidth is typically lower than on-die bandwidth, and cache coherence directories add latency for remote accesses. We propose extensions to both the Roofline and ECM models that capture these chiplet effects, predicting kernel performance based on data placement (which chiplet's DRAM), thread scheduling (which chiplet's core), and chiplet topology (number of chiplets, die-to-die bandwidth, LLC slice distribution).

Definition 27.1 (Roofline Model)**.** A performance model that plots achievable floating-point throughput (FLOP/s) versus operational intensity (FLOP/byte), overlaying a "roof" of peak compute throughput and bandwidth ceilings. A kernel is compute-bound when its operational intensity exceeds the ridge point (where compute ceiling and bandwidth ceiling intersect) and memory-bound otherwise. The standard model assumes uniform memory latency across all cores.

Definition 27.2 (Chiplet Memory Hierarchy)**.** A multi-die memory system where each chiplet has its own private last-level cache (`LLC`) slice, local `DRAM` channel, and directory coherence controller. Access to a remote chiplet's LLC or DRAM incurs a die-to-die link traversal with 2–5× higher latency than local access, and die-to-die bandwidth is typically lower than on-die bandwidth. The directory protocol adds additional latency for remote coherence transactions.

Theorem 27.1 (Chiplet Prediction Bound)**.** *The Chiplet Roofline model predicts kernel performance within 20% of measured throughput for any chiplet topology where the number of chiplets* $N \leq 16$*, die-to-die bandwidth* ≥ 10 *GB/s, and working set size* W *exceeds local LLC capacity. Standard Roofline may mispredict by 2–5× for the same workloads due to its uniform-latency assumption.*

27.1 Related Work

Approach	Strength	Gap
Roofline model (Williams, Berkeley)	Simple, intuitive, widely used	Assumes uniform memory system
ECM model (Hager, Erlangen)	Detailed latency/throughput model	Assumes single-die; no chiplet effects
Analytical models for NUMA	Model NUMA effects	Typically 2-socket, not chiplet hierarchy
gem5 simulation	High accuracy, flexible	Very slow; not for design space exploration
Cache-aware models (LMBench, STREAM)	Empirical bandwidth/latency per level	Do not extrapolate to chiplet configs
Chiplet-specific models (AMD white papers)	Model specific AMD chiplet behavior	Company-specific; not general
Performance prediction for chiplets (Loh, 2024)	First-order chiplet model	Ad-hoc; not integrated with Roofline/ECM
Roofline for NUMA (Chatterjee, 2020)	Extends roofline for multi-socket	Does not model die-to-die LLC access

No general extension of the Roofline or ECM models handles chiplet-specific effects such as cross-die LLC access, die-to-die bandwidth, and directory overhead.

27.2 Proposed Approach

We begin by characterizing the chiplet memory hierarchy on real hardwareAMD EPYC 9654 (96 cores, 12 chiplets), Intel Xeon 8490H (60 cores, multi-tile), and Apple M2 Ultra (24 cores, 2 dies)measuring DRAM latency and bandwidth (local and remote), LLC hit latency and bandwidth (local and remote slices), die-to-die link bandwidth and latency, and directory overhead using LMBench, STREAM, and custom microbenchmarks. These measurements provide the input parameters for our models.

Architectural Insight

Extending the Roofline model to chiplet architectures requires replacing the single bandwidth ceiling with multiple ceilings for local DRAM, remote DRAM, local LLC, and remote LLC. The effective operational intensity accounts for the latency and bandwidth ratios between local and remote paths. Similarly, the ECM model gains chiplet-specific latency components for remote LLC slices, remote DRAM, and directory lookup overhead.

For the Chiplet Roofline model, we extend the standard single-ceiling Roofline with multiple bandwidth ceilings: local DRAM bandwidth, remote DRAM bandwidth (limited by the die-to-die link), local LLC bandwidth, and remote LLC bandwidth. The effective operational intensity for a kernel with working set W placed across multiple DRAM nodes becomes compute/(local_bytes + remote_bytes$\times$latency_ratio/bandwidth_ratio). The resulting Chiplet Roofline chart has operational

intensity (FLOP/byte) on the x-axis and performance (FLOP/sec) on the y-axis, with multiple bandwidth ceilings; the bottleneck ceiling is the minimum of all applicable ceilings. To use the model, one determines the NUMA allocation policy, traces access patterns to count local versus remote DRAM and LLC accesses, computes per-path operational intensity, and predicts performance as the minimum of the compute ceiling and the weighted bandwidth ceiling.

For the Chiplet ECM model, we extend the standard decomposition $T = T_{\text{comp}} + T_{L1} + \max(T_{L2}, T_{\text{OL}}) + \max(T_{L3}, T_{\text{OL}}) + \max(T_{\text{mem}}, T_{\text{OL}})$ to incorporate chiplet-specific latency components:

$$\begin{aligned} T = T_{\text{comp}} + T_{L1} + \max(T_{L2}, T_{\text{OL}}) + \; & \max(T_{\text{LLC\_slice\_local}}, T_{\text{OL}}) + \max(T_{\text{LLC\_slice\_remote}}, T_{\text{OL}}) \\ + \; & \max(T_{\text{DRAM\_local}}, T_{\text{OL}}) + \max(T_{\text{DRAM\_remote}}, T_{\text{OL}}) + T_{\text{directory\_lookup}} \end{aligned}$$

where $T_{\text{LLC\_slice\_local}}$ and $T_{\text{LLC\_slice\_remote}}$ capture the latency to local and remote LLC slices (the latter including a die-to-die hop), $T_{\text{DRAM\_local}}$ and $T_{\text{DRAM\_remote}}$ capture local and remote DRAM access, and $T_{\text{directory\_lookup}}$ captures the directory protocol overhead for lines in MESI states requiring directory access. Each memory access is classified by its path through the hierarchy, determined by data placement and cache coherence state.

We validate both models against real hardware using microbenchmarks (STREAM, LMBench, random access), kernels (DGEMM, SpMV, FFT, stencil, histogram, merge sort), and applications (NAS Parallel Benchmarks CG/MG/BT, HPCG, Graph500). For each kernel we measure actual performance and compare against standard Roofline/ECM, Chiplet Roofline/ECM, and an upper bound representing perfect local-only data placement.

We then perform sensitivity analysis using the validated models, varying the number of chiplets (2–16), die-to-die bandwidth (10–100 GB/s), LLC slice size (8–64 MB per chiplet), and directory placement (local versus distributed), identifying the critical parameter for each kernel class. For memory-bound kernels, die-to-die bandwidth is the dominant factor when the working set exceeds local LLC capacity; for compute-bound kernels, chiplet effects are negligible; directory lookup latency matters only for workloads with high coherence traffic such as false sharing or producer-consumer patterns.

27.3 Evaluation

Kernel	**Metric**	**Baselines**
STREAM (triad)	BW, prediction error	Standard Roofline
DGEMM	GFLOPS, prediction error	Standard Roofline, standard ECM
SpMV (various matrices)	GFLOPS, prediction error	Standard Roofline
Stencil (7pt, 25pt)	MLUP/s, prediction error	Standard ECM
NAS CG, MG	Time, prediction error	Standard Roofline
HPCG	HPCG-FLOPS, prediction error	Standard Roofline
Random access (GUPS)	GUPS, prediction error	N/A

We target prediction error within 20% of measured performance for Chiplet Roofline/ECM, compared to standard Roofline which may be off by 2–5× for chiplet-specific effects.

27.4 Research Directions

Dynamic data placement guided by chiplet performance models (SC'27). Use the Chiplet Roofline model to drive a runtime data migration policy: when a kernel's effective operational intensity falls below the remote bandwidth ceiling, the runtime migrates the working set to the local DRAM of the executing chiplet. Formulate migration as an online optimization problem where each page (p) has a migration cost $C_m(p)$ and benefit $B(p) = (T_{\text{remote}}(p) - T_{\text{local}}(p)) \cdot f(p)$ with $f(p)$ the access frequency, and the runtime decides migration when $B(p) > C_m(p)$. The novelty is closing the loop between analytical performance modeling and runtime data placement; evaluate on AMD EPYC with 12 chiplets, achieving 15–40% speedup over first-touch NUMA placement for memory-bound kernels with irregular access patterns.

Integration with auto-tuning frameworks (IPDPS'27). Incorporate chiplet-aware performance models into OpenTuner or similar auto-tuning frameworks, replacing black-box search with model-guided exploration. Given a kernel with tunable parameters (block size, thread count, data layout, NUMA policy), the Chiplet ECM model predicts performance for each configuration in microseconds rather than seconds of measurement, reducing the search space by 100–1000×. The novelty is using the analytical model as a learned surrogate with zero training data; evaluate on 15 kernels from the PolyBench suite on a 12-chiplet AMD EPYC system, showing that model-guided tuning finds configurations within 5% of exhaustive search while using only 1% of the measurement budget.

Extending chiplet models to disaggregated and CXL-attached memory (HPDC'27). Extend the Chiplet Roofline model to account for CXL-attached memory pools, which add additional bandwidth and latency tiers beyond on-package DRAM and remote chiplet DRAM. The model must capture the latency penalty of CXL → die → LLC → core traversal versus local DRAM → LLC → core, and the bandwidth sharing effects when multiple chiplets access the same CXL pool. Design a CXL-aware Roofline with three additional ceilings: CXL-local (same die), CXL-remote (different die), and CXL-shared (pool contention). Validation on an Intel Xeon with CXL memory expander would establish the first analytical model covering the full chiplet-plus-CXL hierarchy, enabling rapid design space exploration for future disaggregated systems.

ML-accelerated chiplet performance prediction (ICS'27). Train a graph neural network on microbenchmark data from 10+ chiplet configurations (different chiplet counts, die-to-die bandwidths, LLC sizes) to predict kernel performance given a chiplet topology graph and a kernel access pattern. The GNN takes as input the chiplet graph $G = (V, E)$ where V are chiplets with attributes (LLC size, DRAM bandwidth) and E are die-to-die links with attributes (bandwidth, latency), and a kernel signature (operational intensity, working set size, access locality). The model predicts execution time and the bottleneck component (compute, local bandwidth, remote bandwidth, directory). The novelty is replacing hand-tuned model parameters with learned representations that generalize to unseen chiplet configurations; target 10% mean prediction error on unseen chiplet topologies.

Chapter 28

QPUs as Cache-Coherent Accelerators

Can we integrate QPUs as *cache-coherent* accelerators, where quantum and classical cores share a unified memory system and coherence protocol?

Current quantum-classical hybrid computing treats QPUs as loosely-coupled coprocessors: the classical CPU prepares input data and writes it to the QPU's memory, the QPU executes a quantum program on its qubits, the QPU measures results and writes them back, and the classical CPU reads the results. This model incurs high communication overhead because data must be explicitly copied between classical and quantum memory over a slow classical link, typically PCIe or Ethernet. For hybrid algorithms that require many classical-quantum round tripsVQE, QAOA, and variational algorithms in generalthis overhead dominates execution time. We ask whether QPUs can be integrated as cache-coherent accelerators, where quantum and classical cores share a unified memory system and coherence protocol. The key challenges are profound: a qubit in superposition cannot be copied (no-cloning theorem) and cannot be cached in a classical cache without destructive measurement; quantum memory consistency requires defining what it means for quantum state to be coherent; and classical data (angles, parameters) must coexist with quantum state under the same coherence framework.

Definition 28.1 (Quantum Memory Model (QMM))**.** A specification of operations on shared qubits that coexist with classical cache coherence. Valid operations include `ApplyUnitary` $U(q, \theta)$ (a write), `Measure` $M(q)$ (a read that collapses the state), `CNOT` $C(q, q')$ (entangling), and `ReturnClassical` $R(q) \to v$ (valid only after measurement). For qubits in classical state ($|0\rangle$ or $|1\rangle$), standard sequential consistency applies; for qubits in superposition, only unitary and measurement operations are permitted.

Definition 28.2 (Quantum Coherence Protocol (QCP))**.** A directory-based coherence protocol with five cache-line states per qubit: `CI` (Classical-Invalid), `CS` (Classical-Shared), `CE` (Classical-Exclusive), `QO` (Quantum-Owned, holds a qubit in superposition or entangled state, only one agent), and `QM` (Quantum-Modified, like QO but modified since last sync). The protocol transaction types are `Rc` (Read Classical), `Wc` (Write Classical), `U` (Apply Unitary), and `M` (Measure).

Theorem 28.1 (QMM Equivalence)**.** *Any program executing under the Quantum Memory Model and the Quantum Coherence Protocol produces the same observable outputs as the same program under the loosely-coupled coprocessor model. The coherence protocol preserves the semantics of quantum operations: measurement collapses the state atomically, no-cloning prevents shared quantum cache lines, and entanglement is preserved.*

28.1 Related Work

Approach	Strength	Gap
Current hybrid quantum-classical computing (IBM Qiskit, Google Cirq, Rigetti)	Loosely-coupled coprocessor model	High communication overhead; no coherence
Quantum cache memory (Tzeng, 2020s)	Proposes quantum caches for quantum computation	Not integrated with classical coherence
Quantum shared memory (Kuperberg, 2000s)	Theoretical model of quantum shared memory	No coherence protocol; no evaluation
CUDA Quantum / NVIDIA	Unified quantum-classical programming	Still uses explicit data movement
Quantum-classical memory consistency (Gay, 2023)	First proposal for quantum memory consistency	Early-stage; no implementation
Coherence protocols for heterogeneous systems (ARM AMBA CHI, CXL)	Well-studied for classical systems	No quantum extension
QPU integration via CXL (IBM, 2024 proposal)	CXL as QPU interconnect	Classical communication only

No existing work defines a quantum memory consistency model suitable for coherence protocols or designs a coherence protocol for QPU+CPU shared memory.

28.2 Proposed Approach

We define the Quantum Memory Model (QMM), which specifies operations on shared qubits: apply unitary $U(q, \theta)$ (a write), measure $M(q)$ (a read that collapses the state), apply CNOT $C(q, q')$ (entangling), and return classical value $R(q) \to v$ (valid only after measurement). For qubits in a classical state ($|0\rangle$ or $|1\rangle$), standard classical sequential consistency applies. For qubits in a quantum state (superposition or entangled), only unitary and measurement operations are permittedno classical read is allowed, as it would measure the state. Two key coherence rules govern the protocol: if qubit q is measured by agent A, all copies in other agents' caches are invalidated (the state collapses to classical); and a qubit in superposition cannot be copied, meaning the protocol can maintain only exclusive or owned states for quantum cache lines, never a shared state. We prove a QMM equivalence theorem: any program under the QMM has the same observable behavior as under the loosely-coupled model, because the coherence protocol preserves the semantics of quantum operations.

Architectural Insight

Integrating QPUs as cache-coherent accelerators requires a fundamentally new memory model where qubits in superposition cannot be copied (no-cloning theorem) and measurement is a destructive read. The Quantum Coherence Protocol handles this by restricting shared states to classical qubits only and using exclusive (`QO`, `QM`) states for quantum lines. The result is that hybrid algorithms avoid explicit PCIe data movement, replacing it with coherent cache-line transfers 10–100× faster.

We design the Quantum Coherence Protocol (QCP), a directory-based protocol with five cache-line states per qubit. Classical-Invalid (CI) holds no valid data. Classical-Shared (CS) holds a classical value (0 or 1) sharable across multiple agents. Classical-Exclusive (CE) holds a classical value owned by a single agent. Quantum-Owned (QO) holds a qubit in superposition or entangledonly one agent may have QO. Quantum-Modified (QM) is like QO but the qubit has been modified by a unitary since the last synchronization point. The protocol handles four transaction types: Read Classical (Rc) returns the value for CS/CE lines and triggers measurement for QO lines; Write Classical (Wc) sets the qubit to a classical value and invalidates other CS copies; Apply Unitary (U) transitions to QM and requires exclusive access with no classical copies present; Measure (M) transitions from QO/QM to CS or CE. The directory is entanglement-aware: if two qubits are entangled, they must be under the same directory home and cannot be split across directories.

We model the performance benefit of quantum coherence versus the loosely-coupled approach. In the loosely-coupled model, each algorithm iteration incurs $2 \times T_{\text{transfer}} + T_{\text{quantum}}$, where T_{transfer} covers PCIe data movement. In the coherent model, each iteration incurs $T_{\text{coherent\_read}} + T_{\text{quantum}} + T_{\text{coherent\_write}}$, where coherence messages are small and use the cache-coherent interconnect, typically 10–100× faster than PCIe transfers for the same data size.

We simulate the QCP by extending gem5 with quantum extensions: a quantum module modeling qubit state, unitary operations, measurement, and entanglement; the QCP coherence protocol; standard RISC-V cores with private caches; and a QPU where each qubit is a cache line in the coherence domain. We benchmark hybrid algorithms including VQE (variational quantum eigensolver), QAOA (quantum approximate optimization algorithm), Shor's algorithm, and Grover search, comparing the loosely-coupled model (DMA-style PCIe transfers) against the coherent model and an ideal upper bound of zero-copy memory access.

28.3 Evaluation

Algorithm	Iterations	Metric	Comparison
VQE (H_2 molecule)	100–1000	Wall-clock time	Loosely-coupled, ideal upper bound
QAOA (MaxCut, $N = 20$)	50–200	Time per iteration	Loosely-coupled
Shor ($N = 15$)	10–50	Total time	Loosely-coupled
Grover ($N = 8$)	10–100	Time per oracle call	Loosely-coupled

We target 2–10× speedup for VQE and QAOA, which have many iterations and small data

transfers.

28.4 Research Directions

Formal verification of quantum coherence protocol correctness (CAV'27). Model check the Quantum Coherence Protocol (QCP) using a custom model checker that extends TLA+ or Murϕ with quantum state: qubit values are either classical $(0, 1)$ or a vector $\alpha|0\rangle + \beta|1\rangle$, unitary operations are linear transformations on state vectors, and measurement collapses to classical with probability $|\alpha|^2$. Verify that QCP satisfies its specification: every qubit in superposition is in exactly one cache (QO or QM), measurement of a qubit invalidates all other cached copies, and programs under QCP produce the same outputs as under the loosely-coupled model. The novelty is the first mechanical verification of a coherence protocol involving quantum state; evaluate by checking all reachable states for systems with up to 2 CPUs and 4 qubits, demonstrating correctness for the full protocol state space.

Quantum coherence over CXL: feasibility and design (ISCA'27). Design a CXL-based coherence protocol for QPU integration, extending CXL.mem and CXL.cache to support quantum cache lines. Define CXL transactions for quantum operations: QMem.Read (load qubit state to cache), QMem.Write (store qubit from cache to memory), QMem.ApplyUnitary (apply U to a cached qubit), and QMem.Measure (measure and return classical result). The protocol must handle the key challenge that qubits in superposition cannot be coherently shared (no-cloning), meaning CXL's shared-state model must be restricted to classical qubits only. Prove that the protocol is deadlock-free and livelock-free, and evaluate latency and bandwidth overhead on a simulated CXL-attached QPU. The contribution is a concrete specification for quantum-coherent CXL, enabling practical QPU integration in future CXL-based systems.

Programming models and compiler support for coherent QPUs (ASPLOS'27). Extend CUDA Quantum with shared-memory quantum-classical annotations: a `__shared__` qubit variable is cache-coherent between CPU and QPU, a `__quantum__` function can read shared qubits without explicit copy, and the compiler generates coherence protocol transactions instead of DMA transfers. Build a source-to-source compiler that translates coherent CUDA Quantum to QCP transactions, performing automatic insertion of coherence barriers after measurement operations and ensuring that shared qubits in superposition are never accessed by classical code. The novelty is demonstrating that coherence-transparent programming is possible for hybrid algorithms: VQE and QAOA benchmarks show 3–8× speedup versus explicit data movement, with programmer effort reduced from manual DMA management to simple shared annotations.

Distributed quantum coherence for multi-QPU systems (SC'27). Extend QCP to distributed quantum computers connected by a quantum network, where each QPU has its own coherence domain and qubits can be entangled across QPUs. The distributed protocol must handle teleportation-based entanglement distribution, quantum repeater latencies, and the impossibility of classical cloning for distributed quantum cache lines. Define a Distributed Quantum Coherence Protocol (DQCP) with states similar to QCP but with a distributed directory that tracks which QPU holds each entangled qubit pair. Prove correctness bounds on the number of classical messages required for a distributed quantum measurement (an analog of the classical cache coherence lower bound), showing that any distributed quantum coherence protocol requires at least $O(\log P)$ messages per measurement to maintain consistency.

Chapter 29

Minimal Synchronization for PGAS Consistency

Can we derive formally the *minimal* synchronization required for a given PGAS program to execute correctly, and design a compiler that automatically inserts only that minimal synchronization?

Partitioned Global Address Space (PGAS) languagesUPC, Chapel, Coarray Fortran, X10, SHMEMprovide a shared-memory abstraction over distributed memory, where some addresses are local (same node) and others are remote (different node). The central correctness issue is consistency: when can a PGAS program assume that a write to a remote address is visible to a subsequent read? Different PGAS models provide different guaranteesUPC offers strict consistency by default with relaxed qualifiers, Chapel uses a data-race-free (DRF) model with synchronization operations, Coarray Fortran leaves synchronization entirely to the programmer via `sync all` and `sync images`, and SHMEM requires explicit `shmem_fence` and `shmem_quiet` operations. In all these models, programmers or compilers insert synchronization operationsfences, barriers, atomics, epoch boundariesto ensure correct ordering. Current compilers either over-synchronize by inserting conservative fences after every remote access (safe but expensive) or under-synchronize by relying on the programmer (error-prone). We propose to compute the minimal set of synchronization operations needed for correctness given a PGAS program and its memory model, and to have the compiler automatically insert only those.

Problem 29.1 (Minimal Fence Placement for PGAS)**.** Given a PGAS program P with threads T, shared addresses A, operations $\mathrm{Op} = \{\mathrm{read}, \mathrm{write}, \mathrm{put}, \mathrm{get}\}$, and a memory model $M = (A, T, \mathrm{Op}, \mathrm{Sync}, \leq)$ where $\mathrm{Sync} = \{\mathrm{fence}, \mathrm{quiet}, \mathrm{barrier}, \mathrm{lock}\}$, find a set $F \subseteq$ program points of fence insertions of minimum cardinality such that every pair of conflicting accesses (a_i, a_j) (same address, at least one write, different threads) is ordered by a path through Sync operations, ensuring data-race-free execution under sequential consistency.

29.1 Related Work

Approach	Strength	Gap
UPC compiler (Berkeley UPC, GCC UPC)	Conservative fence after every remote access	Over-synchronizes; no minimality guarantee
Chapel compiler (Cray, HPE)	User-specified sync/atomic	No automatic fence insertion
Coarray Fortran (GCC, Cray)	Programmer inserts `sync all`	No compiler help
SHMEM / OpenSHMEM	Manual fence and quiet	No automatic insertion
Automatic fence insertion for GPU	Compiler inserts fences for GPU models	GPU-specific; not for PGAS
DRF-based analysis (Boehm, Adve)	Proof: DRF ⇒ SC	For shared memory, not PGAS
Automatic synchronization in distributed systems (IVY, Munin)	Runtime detects and synchronizes	Page-based; not compiler-driven
Data flow analysis for PGAS (Chen, 2007)	Analyzes shared-memory accesses	Does not compute minimal synchronization

No existing work provides a provably minimal synchronization insertion algorithm for a general PGAS memory model.

29.2 Proposed Approach

Minimum Feedback Vertex Set on Conflict Graphs

The key idea is that minimal fence placement reduces to finding a minimum feedback vertex set in a sparse conflict graph with exploitable structure (per-thread access sequences resembling trees). For the common case where each thread's shared-access sequence has bounded treewidth, the problem is solvable in $\mathcal{O}(n \cdot f(k))$ time via dynamic programming.

We formalize a PGAS memory model $P = (A, T, Op, Sync, \leq)$ where A is the set of addresses partitioned into local and remote per thread, T is the set of threads, $Op = \{\text{read}, \text{write}, \text{put}, \text{get}, \text{atomic}\}$, $Sync = \{\text{fence}, \text{quiet}, \text{barrier}, \text{lock}, \text{unlock}\}$, and $\leq$ is per-thread program order. A program is correctly synchronized if every pair of conflicting accessessame address, at least one write, different threads or no program orderis ordered by a synchronization operation. We prove a DRF-PGAS theorem extending the classic DRF ⇒ SC result (Boehm, Adve) to include put/get operations and network-specific synchronization such as quiet and fence: a data-race-free PGAS program executes as if under sequential consistency.

Given a PGAS program at the LLVM IR level with PGAS annotations, we construct a conflict graph. First, we identify shared accessesall loads and stores to addresses that may be remote (accessed by another thread)using points-to analysis extended with PGAS knowledge of which pointers are shared versus private. Second, for each pair of accesses (a_i, a_j) to the same or aliased

address where at least one is a write and the pair is in different threads or in the same thread but not program-ordered, we create an edge. Third, we classify conflicts: those already ordered by programmer-inserted synchronization (barriers, locks, fences) are removed.

The remaining conflict graph G defines a minimal fence placement problem: find the smallest set of fence positions F such that adding fences makes G acyclic (all remaining conflicts ordered). This is a minimum feedback vertex set problem, NP-hard in general, but PGAS conflict graphs have exploitable structuremost conflicts are between pairs of threads rather than cross-thread chains, the graph is sparse, and fences are inserted per-thread. Our greedy algorithm iterates per thread: for each thread t, compute the set of accesses needing synchronization with thread u, insert a fence after the last such access, remove all conflicts now ordered by the fence, and repeat until the graph is empty. For programs where each thread's access sequence is a tree (no loops with shared accesses), this greedy algorithm produces the unique minimal fence set.

Algorithm 8 Minimal Fence Placement via Greedy Conflict Resolution

Require: PGAS program P, memory model M
Ensure: Minimal fence set F

$G \leftarrow \textsc{BuildConflictGraph}(P, M)$
$F \leftarrow \varnothing$
for each thread t in topological order of G **do**
    $U \leftarrow \textsc{UnorderedAccesses}(t)$
    while $U \neq \varnothing$ **do**
        $u \leftarrow$ last access in program order from U
        $F \leftarrow F \cup \{\text{insert fence after } u\}$
        Remove all conflicts now ordered by F from G
        $U \leftarrow \textsc{UnorderedAccesses}(t)$
    end while
end for
return F

We implement the approach as an LLVM pass for UPC. A frontend compiles UPC to LLVM IR with PGAS annotations for shared and private pointers, remote accesses, and synchronization primitives. Pass 1 builds the conflict graph from the annotated IR. Pass 2 computes minimal fence positions and inserts `upc_fence` or `upc_quiet` calls. Pass 3 performs fence elimination, removing any fences that the analysis shows are redundant. We evaluate on the NAS parallel benchmarks (BT, CG, EP, FT, LU, MG, SP) in UPC, custom mini-apps (miniFE, miniGhost, LULESH) ported to UPC, and synthetic benchmarks with varying access patterns (nearest-neighbor, all-to-all, stencil, gather/scatter). We compare against strict UPC (fence after every access), optimized hand-tuned UPC, and current conservative UPC compilers.

29.3 Evaluation

Benchmark	Metric	Comparison
NAS BT (UPC)	Execution time, fence count	Strict UPC, hand-optimized, conservative
NAS CG (UPC)	Execution time, fence count	Same
NAS MG (UPC)	Execution time, fence count	Same
MiniFE (UPC)	Execution time	Same
UPC microbenchmarks	Fence overhead	All approaches
Correctness validation	Thread sanitizer	No races detected

We target within 5% of hand-optimized fence count with provably correct execution, removing 30–70% of fences compared to conservative insertion.

29.4 Research Directions

Optimal synchronization beyond greedy heuristics (PLDI'27). Develop an exact algorithm for the minimal fence placement problem using integer linear programming with specialized constraints that exploit the structure of PGAS conflict graphs. The conflict graph admits a treewidth bound for common PGAS communication patterns (stencil, gather, all-to-all), enabling dynamic programming with tree decomposition. Prove that for programs where each thread's shared-access sequence has bounded treewidth (formally, the access graph has treewidth k), minimal fence placement is solvable in $O(n \cdot f(k))$ time. Compare the optimal solution against the greedy algorithm across NAS benchmarks, establishing the optimality gap and showing that the greedy algorithm is within 10% of optimal for all tested programs. The contribution is the first polynomial-time optimal algorithm for a practically relevant class of synchronization minimization problems.

Extension to task-based and async PGAS models (PPoPP'27). Extend the conflict graph framework to task-parallel PGAS models (Chapel, X10) where synchronization requirements depend on dynamically created async tasks rather than static thread boundaries. Formulate the problem as a dynamic graph where tasks are created and joined at runtime, and compute minimal synchronization as an online problem: when a new task is spawned, determine whether existing fences cover its accesses or new fences are needed. Design a runtime algorithm that maintains a per-task conflict graph and inserts fences lazily when conflicts are detected at task join time, using a technique inspired by conflict detection in software transactional memory. The contribution is a provably minimal fence insertion algorithm for dynamic PGAS parallelism, evaluated on the Chapel version of NAS benchmarks with Chplop to show 20–40% synchronization reduction versus the current Chapel compiler.

Dynamic synchronization adaptation via runtime feedback (HPDC'27). Augment compiler-inserted fences with a lightweight runtime monitoring system that tracks actual remote access conflicts using hardware performance counters (e.g., AMD IBS or Intel PEBS for remote memory accesses) and adjusts fence placement dynamically. When the runtime detects that a compiler-inserted fence has never guarded a real conflict across program runs (using a shadow bit per remote address range), it removes the fence on the next execution; conversely, when a data race is detected (via a lightweight race detection hardware feature), it adds a fence. The novelty

is a self-optimizing synchronization system that adapts to program inputs and data distributions without requiring recompilation; evaluate on SHMEM benchmarks with varying input sizes, showing that dynamic adaptation reduces fence count by 30–50% compared to static optimal placement while maintaining correctness (no races detected).

Cross-layer PGAS memory model co-design for minimal synchronization (SC'27). Design a new PGAS memory model whose semantics are defined to make minimal synchronization computable in polynomial time. The key insight is to restrict the consistency model to a class called *orderable PGAS*: a program is correct if its happens-before graph (including put/get) is a partial order with width bounded by a function of the program's communication structure. Show that for orderable PGAS, the minimal synchronization problem reduces to computing the transitive reduction of the happens-before graph, which is solvable in $O(n \log n)$. The novelty is co-designing the memory model and its compiler analysis: the memory model is weaker than UPC strict but stronger than SHMEM, and the compiler can always insert the provably minimal fences. Evaluate by comparing the performance of orderable PGAS programs against UPC (strict) and SHMEM (manual) for the NAS benchmarks, showing that automatic minimal synchronization matches or outperforms hand-optimized manual synchronization.

Part VI

Algorithms

Chapter 30

Fine-Grained Complexity of Exact Algorithms Parameterized by Width Measures

Can we establish the *optimal* base of the exponent for NP-complete problems parameterized by graph width measures (treewidth, pathwidth, clique-width)?

Many NP-complete problems admit algorithms with runtime $2^{O(\mathrm{tw})} \times \mathrm{poly}(n)$, where tw is the treewidth of the input graph. Examples include 3-coloring ($O(2^{\mathrm{tw}} \cdot n)$, though Courcelle's theorem gives a larger base), vertex cover ($O(2^{\mathrm{tw}} \cdot n)$ via folklore DP over tree decompositions), set cover ($O(2^{\mathrm{tw}} \cdot n)$), and maximum independent set ($O(2^{\mathrm{tw}} \cdot n)$). These algorithms use dynamic programming: the DP table at each bag has size $2^{O(\mathrm{tw})}$, and with $O(n)$ bags the runtime is $2^{O(\mathrm{tw})} \times \mathrm{poly}(n)$. The base of the exponent, however, matters enormously for practical computation2^{tw} versus 3^{tw} versus $\mathrm{tw}^{\mathrm{tw}}$ differentiates solving graphs with $\mathrm{tw} = 40$ from $\mathrm{tw} = 10$, and current algorithms often have bases far from optimal. For specific NP-complete problems parameterized by treewidth and other width measures, we seek tight lower bounds under SETH and ETH, determining the exact constant c such that an algorithm with runtime $c^{\mathrm{tw}} \times \mathrm{poly}(n)$ exists and no algorithm with runtime $(c - \varepsilon)^{\mathrm{tw}} \times \mathrm{poly}(n)$ exists for any $\varepsilon > 0$.

30.1 Related Work

Problem	Treewidth DP base	Best known lower bound	Gap
3-coloring	3^{tw} (simple DP)	$2^{\Omega(\mathrm{tw})}$ [SETH-hard]	3 vs. 2
Vertex cover	2^{tw}	$2^{\Omega(\mathrm{tw})}$ [trivially tight]	Tight
Independent set	2^{tw}	$2^{\Omega(\mathrm{tw})}$ [trivially tight]	Tight
Dominating set	3^{tw} or 4^{tw}	$2^{\Omega(\mathrm{tw})}$ or $(2 - \varepsilon)^{\mathrm{tw}}$	Open
Hamiltonian path	4^{tw} (TSP DP)	$2^{\Omega(\mathrm{tw})}$ [SETH-hard]	4 vs. 2
Set cover	2^{tw}	$2^{\Omega(\mathrm{tw})}$	Unclear

For many classic problems, the gap between the best known algorithm and the best known lower bound remains largefor 3-coloring, a factor of 3/2 in the base. Closing these gaps is a major open problem in fine-grained complexity.

30.2 Proposed Approach

We develop a general framework for establishing tight upper and lower bounds.

Definition 30.1 (Optimal Exponent for Treewidth DP)**.** For a problem Π parameterized by treewidth tw, the *optimal exponent* $c^*(\Pi)$ is the infimum over all constants c such that Π is solvable in $O(c^{\mathrm{tw}} \cdot \mathrm{poly}(n))$ time. An algorithm achieving $c^*(\Pi)$ is *optimal* if no $(c^*(\Pi) - \varepsilon)^{\mathrm{tw}} \cdot \mathrm{poly}(n)$ algorithm exists for any $\varepsilon > 0$ under SETH.

For upper bounds, we improve DP algorithms through state compression (tracking only equivalence classes of assignments rather than all possibilities), memoization with branching (combining DP with measure-and-conquer for small bags), and algebraic methods (inclusion-exclusion, Möbius inversion, Gaussian elimination to reduce state counts). For lower bounds, we use the standard SETH-hardness framework: reduce from k-CNF-SAT to the problem on a graph of width tw, where the reduction must be tightif the problem can be solved in c^{tw} time, then k-SAT can be solved in $c^{n/k}$ time, contradicting SETH for c below some thresholdusing edge gadgets that encode SAT variables and constraints as graph structures with bounded treewidth.

For 3-coloring, we target closing the gap between 3^{tw} and 2^{tw}.

Theorem 30.1 (Tight Bounds for 3-Coloring Parameterized by Treewidth)**.** *The optimal exponent for 3-coloring parameterized by treewidth satisfies $2 \leq c^*(3\text{-}Col) \leq 3$. The upper bound is achieved by a symmetry-reduced DP that tracks equivalence classes of colorings under color permutation; the lower bound follows from a SETH-hardness reduction encoding k-CNF as a 3-coloring instance of treewidth $O(n/k)$.*

Proof sketch. Upper bound: represent each bag's coloring modulo the S_3 color-permutation group using combinatorial equivalence classes. The number of classes for a bag of size k is $\binom{k+2}{2}$, yielding $O(\mathrm{tw}^2)$ states per bag. The DP transition merges classes across adjacent bags using group-compatible join operations. Lower bound: given a k-CNF formula ϕ, construct a graph G_ϕ with treewidth $O(n/k)$ such that G_ϕ is 3-colorable iff ϕ is satisfiable. A $(2-\varepsilon)^{\mathrm{tw}}$ algorithm for 3-coloring would yield a $(2-\varepsilon)^{n/k}$ algorithm for k-SAT, violating SETH. □

The naive DP assigns one of three colors to each of the $\mathrm{tw}+1$ vertices in a bag, yielding $3^{\mathrm{tw}+1}$ states. Many colorings are equivalent up to color permutation; modulo this symmetry, the number of distinct states drops from 3^k to $\binom{k+2}{2} \approx O(k^2/2) \approx O(\mathrm{tw}^2)$, which is $2^{2\log \mathrm{tw}}$ and much smaller than 2^{tw} for large tw. The challenge is making this symmetry reduction work across DP steps, not just within a single bag, since colors interact through edges spanning bags. For the lower bound, we aim to strengthen the known $2^{o(\mathrm{tw})}$ ETH bound to $(2-\varepsilon)^{\mathrm{tw}}$ under SETH through tighter gadget constructions.

For Hamiltonian path, the standard DP tracks which vertices in each bag are path endpoints and which are interior, yielding $O(4^{\mathrm{tw}})$ states, while Cygan et al. (2016) proved $(2-\varepsilon)^{\mathrm{tw}}$ hardness under SETH. We explore whether the cut-and-count technique, which gives $O(2^{O(\mathrm{tw})})$ for counting Hamiltonian cycles, can be adapted for the decision problem with a better constant, potentially achieving $O(2^{\mathrm{tw}})$ and matching the lower bound.

Core Thesis

The optimal exponent for treewidth-parameterized NP-complete problems is determined by a combination of state compression via symmetry reduction (e.g., color-permutation equivalence for 3-coloring) and tight SETH-hardness reductions, with the unifying goal of establishing $c^*(\Pi)$ for each problem and proving optimality.

We compare optimal exponents across width measurespathwidth, treewidth, and clique-widthinvestigating whether there exist problems where the optimal exponent for treewidth is strictly smaller than for clique-width (which often incurs double-exponential dependence $2^{2^{O(\mathrm{cw})}}$). For problems with established optimal exponents such as vertex cover ($c = 2$) and independent set ($c = 2$), we engineer practical implementations using nice tree decompositions, bitset-optimized table manipulation with SIMD, and comparison against state-of-the-art solvers from the PACE challenge. For problems with non-settled exponents, we provide lower-bound evidence by showing that DP algorithms with certain state complexity are necessary barring algorithmic breakthroughs.

30.3 Evaluation

Problem	Width measure	Metric	Current best
3-coloring	Treewidth	Optimal exponent (tight)	3^{tw} vs. $2^{o(\mathrm{tw})}$
Hamiltonian path	Treewidth	Optimal exponent	4^{tw} vs. $(2-\varepsilon)^{\mathrm{tw}}$
Dominating set	Treewidth	Optimal exponent	$\sim 4^{\mathrm{tw}}$ vs. $2^{o(\mathrm{tw})}$
All	Pathwidth	Optimal exponent	Compare to treewidth
All	Clique-width	Optimal exponent	Compare to treewidth
Vertex cover, IS	All	Practical implementations	PACE 2024 results

30.4 Research Directions

1. **Closing the 3^{tw} vs 2^{tw} gap for 3-coloring.** The current gap between the 3^{tw} DP upper bound and the $2^{\Omega(\mathrm{tw})}$ SETH lower bound leaves a factor of $3/2$ unresolved. A tight $(2+\varepsilon)^{\mathrm{tw}}$ algorithm via advanced symmetry reduction (tracking equivalence classes of colorings under color permutation across bag boundaries) would yield a FOCS/STOC-level result. Conversely, proving $(3-\varepsilon)^{\mathrm{tw}}$ SETH-hardness would require novel gadget constructions that encode k-CNF constraints into 3-coloring with only a constant-factor blowup in treewidth, a significant open problem.

2. **Optimal exponent for Hamiltonian path.** The 4^{tw} DP (tracking path endpoints per bag) vs $(2-\varepsilon)^{\mathrm{tw}}$ lower bound represents a gap of $2\times$ in the base. Adapting the cut-and-count technique, which yields $2^{O(\mathrm{tw})}$ for *counting* Hamiltonian cycles, to the *decision* problem with an explicit constant (e.g., $c = 2$ or $c = 3$) would match or narrow this gap, publishable at SODA or ICALP. The key challenge is derandomizing the isolation lemma step without blowing up the treewidth dependence.

3. **Width measure separations: treewidth vs clique-width.** Problems like $(\min, +)$-matrix chain and graph coloring exhibit double-exponential dependence on clique-width ($2^{2^{O(\text{cw})}}$) but single-exponential on treewidth. Characterizing which structural properties cause this explosion (e.g., the presence of a fixed bipartite relation in the decomposition tree) would yield a taxonomy publishable at ESA or ITCS. A proof that clique-width cannot simulate treewidth with only a single-exponential blowup for any NP-complete problem would be a major advance.

4. **Practical bitset-optimized DP for PACE benchmarks.** For problems with tight exponents (vertex cover, independent set: $c = 2$), SIMD-optimized DP over nice tree decompositions can solve instances with tw up to 60, far beyond current PACE solvers. Engineering contributions (bitset representation, cache-oblivious bag ordering, parallel bag processing) with empirical evaluation on the PACE 2024 benchmark suite are suitable for ALENEX or ESA Engineering Track.

Chapter 31

Lattice-Linearity: A Universal Characterization

Can we characterize lattice-linear predicates as exactly those definable in a certain fragment of first-order logic, giving a syntactic test for lattice-linearity and a complete algorithm for problems in this class?

We showed in Q25 that a diverse set of optimization problemsSSSP, BFS, stable marriage, knapsack, job schedulingcan be solved by a generic lock-free parallel algorithm (LLP-FW) whose common principle is that the solution predicate is *lattice-linear*, a property allowing independent corrections by concurrent threads without conflicts. This raises a deeper question: is there a common logical or algebraic structure these problems share? Can we characterize lattice-linear predicates syntacticallyfor instance, as exactly the predicates definable in a fragment of first-order logic? Such a characterization would provide a decision procedure for lattice-linearity (given a problem description, determine whether it has an LLP), enable an LLP compiler that automatically transforms any LLP problem description into a lock-free parallel algorithm, and reveal the limits of the LLP approach. We propose to prove a characterization theorem stating that lattice-linear predicates correspond exactly to monotone, separable, first-order-definable properties over a partially ordered domain, and to derive the algorithmic consequences.

31.1 Related Work

Approach	Strength	Gap
LLP-FW (Kumar, 2026)	Empirical7+ problems solved	No characterization; no theoretical boundary
Lattice-linear predicates (Kumar, 2024)	Formal definition of LLPs	No connection to logic or complexity
Local computation algorithms (Alon, Rubinfeld)	Characterizes locally checkable problems	Different model (LCA: sublinear time; LLP: lock-free parallel)
Descriptive complexity (Fagin, Immerman)	Characterizes complexity classes by logic	Not applied to LLPs
Monotone CSP (Kun, Szabó)	Monotone constraint satisfaction	Connected to lattice structure, not parallelism
Fixed point theory (Tarski, Kleene)	Least fixed point of monotone functions	Used in LLP proofs, not characterization
Lattice theory in distributed computing (Lamport, Lynch)	State machine replication, lattice agreement	Different from LLPs

No existing work connects lattice-linear predicates to logical definability.

31.2 Proposed Approach

We define a logical language for lattice-linear properties.

Definition 31.1 (Monotone FO($\sqsubseteq$))**.** The language Monotone FO($\sqsubseteq$) is first-order logic over a structure with a partial order $\sqsubseteq$ where:

1. all predicates are monotone: if $P(x)$ holds and $y \sqsubseteq x$, then $P(y)$ holds;
2. negation is forbidden (or only guarded by monotone conditions);
3. quantifiers range over lattice elements.

A property P over a lattice L is *definable* in this language if there exists a sentence ϕ such that for all $s \in L$, $P(s)$ holds iff $L \models \phi(s)$.

The central conjecture is that P is lattice-linear iff P is definable in Monotone FO($\sqsubseteq$) and P is prefix-independentthe value depends only on the set of assignments to variables, not on the order in which they were made, which is a technical condition needed for the parallel correction algorithm.

The characterization is stated as a theorem.

Theorem 31.1 (Characterization of Lattice-Linear Predicates)**.** *A predicate P over a lattice L is lattice-linear if and only if P is definable in Monotone FO($\sqsubseteq$) and P is prefix-independent (the value depends only on the set of variable assignments, not on their order).*

Proof sketch. (Forward direction) Given a Monotone FO($\sqsubseteq$) sentence ϕ, for any state s and variable v, define the witness set $W_v(s)$ as the smallest set of variables whose values must change to satisfy ϕ at v, obtained by evaluating the monotone predicates locally. Monotonicity guarantees this set is well-defined and computable in $O(1)$ time per variable, satisfying the definition of lattice-linearity. (Reverse direction) Given a lattice-linear predicate P with witness function W, construct the dependency graph where edges connect variables sharing a witness. Express P as "for every assignment consistent with the dependency graph, the local condition at each variable holds," which translates to a Monotone FO($\sqsubseteq$) sentence of $\exists\forall$ form. □

We prove the characterization in both directions. For the forward direction (every Monotone FO($\sqsubseteq$)-definable property is lattice-linear), given a state $s \in L$ and a variable v, the witness set $W_v(s)$ for v being "bad" is the smallest set of variables that need to change to satisfy ϕ in the sense of the lattice order; because ϕ is monotone, the witness function is well-defined and locally computable. For the reverse direction (every lattice-linear property is definable in Monotone FO($\sqsubseteq$)), the lattice-linear predicate P defines for each state s a set of witness variables $W(s)$, which itself defines a dependency graph; the property P can be expressed as "for every assignment to the dependency graph, the local condition at each variable holds," a first-order sentence of $\exists\forall$ form that may be equivalent to a Monotone FO($\sqsubseteq$) sentence.

Given a property P as a constraint satisfaction problem, we test for lattice-linearity by constructing the dependency graph of constraints (which variables share a constraint), checking whether the graph is bipartite (for binary constraints) or has bounded treewidth (for general constraints), and checking whether each constraint is monotone with respect to a partial order on the domain. If both conditions hold, P is lattice-linear by the characterization theorem. Bipartiteness and treewidth checks are polynomial; monotonicity checking is polynomial per constraint, so the overall decision procedure runs in polynomial time.

Given a Monotone FO($\sqsubseteq$) formula ϕ describing the lattice-linear predicate, we automatically derive the correction function. We parse ϕ into constituent atomic predicates, derive a local correction function for each (e.g., for $x \leq y$, set $x = \min(x, y)$ or $y = \max(x, y)$; for a unary monotone predicate $P(x)$, move x to the smallest lattice value satisfying P), and compose local corrections so the overall correction for a variable is the join (least upper bound) of all local corrections involving that variable. The resulting function is monotone by construction and guaranteed to converge by Tarski's fixed point theorem.

Core Thesis

Lattice-linear predicates are exactly those definable in Monotone FO($\sqsubseteq$) with prefix-independence, establishing a syntactic characterization that yields a polynomial-time decision procedure for lattice-linearity and enables automatic generation of lock-free parallel algorithms from logical specifications.

We test the characterization on known LLP problems (SSSP, BFS, stable marriage, knapsack, job scheduling, all confirmed as lattice-linear and Monotone FO($\sqsubseteq$)-definable) and non-LLP problems (minimum cut, traveling salesman, general SAT, all rejected by the decision procedure). For confirmed cases, we compare automatically generated correction functions against the hand-crafted corrections in LLP-FW.

31.3 Evaluation

Problem	LLP?	FO definable?	Automatic correction matches LLP-FW?
SSSP	Yes	Yes	Yes (generated = LLP-FW's Correct)
BFS	Yes	Yes	Yes
Stable marriage	Yes	Yes	Yes
Knapsack	Yes	Yes	Yes
Job scheduling	Yes	Yes	Yes
Minimum cut	No	No (rejected)	N/A
TSP	No	No	N/A
Graph coloring	No?	No?	Tested

31.4 Research Directions

1. **Proving the characterization theorem.** The central conjecture (that lattice-linear predicates are exactly those definable in Monotone FO($\sqsubseteq$) with prefix-independence) requires proof in both directions. The forward direction (every FO-definable predicate is lattice-linear) reduces to showing that monotone first-order formulas admit local witness functions computable in $O(1)$ time per variable. The reverse direction (every lattice-linear predicate is FO-definable) requires constructing, from the witness function, a first-order sentence that captures the predicate. A complete proof with explicit translation rules would appear at STOC or LICS.

2. **Polynomial-time decision procedure for lattice-linearity.** Given a constraint satisfaction problem as a set of first-order formulas, decide in polynomial time whether the conjunction defines a lattice-linear predicate. The algorithm must check: (a) each constraint is monotone w.r.t. a partial order on the domain, (b) the constraint graph has bounded treewidth or is bipartite (for binary constraints). Proving that these conditions are sufficient (by the characterization theorem) and necessary (by constructing a counterexample) yields a practical tool for detecting lock-free parallelism opportunities, publishable at CP or AAAI.

3. **An LLP compiler from FO specifications.** Given a Monotone FO($\sqsubseteq$) formula, automatically synthesize a lock-free parallel algorithm by (i) parsing the formula into atomic predicates, (ii) deriving a local correction function for each, and (iii) composing them via lattice join. Implementing this for the seven known LLP problems (SSSP, BFS, stable marriage, etc.) and comparing synthesized versus hand-crafted correction functions validates the framework. This is a systems-plus-theory contribution suitable for PPoPP or SPAA.

4. **Extending the characterization beyond lattice-linearity.** Some useful problems (e.g., maximum matching, minimum cut) are *nearly* lattice-linear: their predicates become lattice-linear after a bounded number of relaxations. Characterizing the closure of lattice-linear predicates under existential quantification, fixed-point iteration, or bounded 'repairs' would expand the scope of automatic parallelization. A complete classification of the LLP hierarchy

(from strictly lattice-linear to k-repairable) opens a new research program at the intersection of logic, order theory, and parallel computing, suitable for a journal version in JACM or LMCS.

Chapter 32

Bisynchronous Ethernet and the FLP Impossibility

Does bisynchronous Ethernet genuinely circumvent the FLP impossibility result, or does it merely shift the synchrony assumption to the physical layer?

The FLP impossibility theorem (Fischer, Lynch, Paterson, 1985) states that no deterministic consensus protocol can tolerate even a single crash failure in an asynchronous system. Bisynchronous Ethernet (OAEOrderly Asynchronous Ethernet, 2025) claims to circumvent FLP by exploiting physical-layer properties of real Ethernet networks: bounded propagation delay (speed of light times cable length), deterministic arbitration (CSMA/CD with bounded collision detection), and a no-congestion assumption (dedicated link or controlled environment). The key question is whether OAE provides a genuinely asynchronous communication model that circumvents FLP (the strong claim) or is actually a partial synchrony model (Dwork, Lynch, Stockmeyer, 1988) with known bounds on message delivery, already known to circumvent FLP (the weak claim). If the weak claim holds, bisynchronous Ethernet is an important engineering contribution but not a theoretical breakthrough; if the strong claim holds, it challenges four decades of distributed computing theory. We propose to formalize the exact synchrony model of bisynchronous Ethernet and determine its relationship to the FLP impossibility.

32.1 Related Work

Approach	Strength	Gap
FLP theorem (Fischer, Lynch, Paterson)	Proves impossibility of async consensus	Assumes fully asynchronous model
Partial synchrony (Dwork, Lynch, Stockmeyer)	Consensus possible with known/unknown timing bounds	Requires timing assumptions (GST)
Paxos / Raft (Lamport, Ongaro)	Consensus with failure detectors, leader election	Uses timing assumptions indirectly
Bisynchronous Ethernet / OAE (2025)	Claims FLP circumvention via physical-layer guarantees	Formal model not well-defined
Physical-layer consensus (various)	Consensus using physical properties	Usually exploit physics for timing guarantees
FLP circumvention attempts (many, all failed)	Incorrect claims of circumvention	Each failed when formalized properly

No formal analysis of the bisynchronous Ethernet model in the context of the FLP theorem exists.

32.2 Proposed Approach

We define a formal OAE automaton.

Definition 32.1 (OAE Automaton)**.** An OAE automaton is a tuple $O = (P, L, T, \delta, C)$ where:

- $P = \{p_1, \ldots, p_n\}$ is a set of processes;
- $L = \{l_1, \ldots, l_m\}$ is a set of bidirectional links, each with maximum propagation delay $d_{\max}(l) = \text{length}(l)/\text{speed\_of\_light}$;
- T is the timing model (real or abstract time units);
- δ is the physical-layer arbitration mechanism (collision detection and deterministic resolution);
- C is the bounded collision detection time $t_{\text{collision}}$.

Messages are placed on a link with a remaining propagation time counter that decrements each time unit until reaching zero upon arrival; collisions are detected when two messages occupy the same link simultaneously and are resolved within $t_{\text{collision}}$.

The critical question is whether the time unit is real (seconds) or abstract: if real, the model is partially synchronous; if abstract, it is asynchronous.

We map the OAE model to the standard FLP model. FLP assumes messages can be delayed arbitrarily but are eventually delivered, with no bound on processing time and crash failures. OAE provides bounded message delay ($d_{\max}$), bounded processing time (by physical-layer constraints), and introduces link failure as a new failure mode.

Theorem 32.1 (OAE–DLS Equivalence)**.** *Let O be an OAE automaton with unknown but bounded link delays $d_{\max}$ (known to exist but the precise value is unknown). Then the consensus problem in O is equivalent to the consensus problem in the Dwork–Lynch–Stockmeyer partial synchrony model with unknown Global Stabilization Time. Consequently, bisynchronous Ethernet does not circumvent the FLP impossibility but rather instantiates the DLS model with physical-layer timing bounds.*

Proof sketch. Construct a simulation between OAE configurations and DLS configurations. Each OAE process maps to a DLS process; the bounded delay $d_{\max}$ translates to the DLS assumption that there exists an unknown GST after which message delays are bounded. Collision detection in OAE corresponds to DLS's ability to detect message loss via timeouts. The proof shows that any OAE execution has a corresponding DLS execution with the same decision outcome, and vice versa, establishing equivalence. □

We prove that OAE with unknown $d_{\max}$ (but known that it exists) is equivalent to the partial synchrony model of Dwork, Lynch, and Stockmeyer with unknown GST (Global Stabilization Time); therefore, OAE does not circumvent FLP but falls within the DLS framework. With known $d_{\max}$ (from cable length), OAE becomes a synchronous system where consensus is trivially possible.

We design an OAE-optimized consensus protocol. In each round, the leader broadcasts a proposal using a single Ethernet frame; each non-leader responds with an acknowledgment (collision

detection may signal disagreement); and the leader, on receiving acknowledgments from all non-faulty processes within bounded time, commits the value and broadcasts a commit message. Because message delivery is bounded, the leader knows when to expect acknowledgmentsif none arrives within $d_{\max}$ + processing_time, the sender is considered faulty. Collision detection serves as a fast negative acknowledgment: if two processes send different values, the collision is detected and the leader knows not to commit. OAE-Consensus tolerates $f < n/2$ crash failures.

Core Thesis

Bisynchronous Ethernet does not circumvent the FLP impossibility; its bounded propagation delay and deterministic collision arbitration place it within the Dwork–Lynch–Stockmeyer partial synchrony framework. The physical-layer guarantees make consensus practically efficient but do not constitute a theoretical breakthrough in distributed computability.

We analyze what happens when OAE's physical-layer guarantees fail. A cable cut is equivalent to a process crash (handled by the crash tolerance). Collision detection failure may cause message loss, requiring acknowledgments and retries. A timing violation (a process responds late) breaks the bounded-time assumption, causing a fallback to standard asynchronous consensus with failure detectors. We prove that OAE-Consensus is safe (agreement and validity) under OAE's physical-layer model and live under synchronous or partially synchronous timing assumptions; under pure asynchrony, it reduces to standard Paxos.

We implement OAE-Consensus on a real Ethernet testbed with 5–10 machines connected via a dedicated switch, using a modified Ethernet driver that exposes collision detection to the application, and a custom consensus protocol using raw Ethernet frames (bypassing TCP/IP). We evaluate latency (RTT for a consensus round), throughput (decisions per second), fault tolerance (process crash, link failure, timing violation), and compare against standard Paxos, Raft, and Fast Paxos.

32.3 Evaluation

Experiment	Metric	Comparison
Baseline latency	RTT for 1 consensus decision	Paxos, Raft, Fast Paxos
Throughput	Decisions/second, no failures	Paxos, Raft
Crash failure	Recovery time after process crash	Paxos (leader election time)
Link failure	Recovery time after cable cut	Paxos (leader election)
Timing violation	Impact of a slow process	Paxos
Scalability	Latency vs. n ($n = 3, 5, 7, 9$)	Paxos, Raft

32.4 Research Directions

1. **Formalizing the OAE automaton and mapping to the FLP/DLS model.** The strong claim (that bisynchronous Ethernet circumvents FLP) rests on physical-layer guarantees (bounded propagation delay, deterministic collision arbitration). A rigorous automaton model (P, L, T, δ, C) with real-valued time and bounded collision detection must be mapped to either the asynchronous FLP model (showing FLP's assumptions are violated) or the partial-synchrony

DLS model (showing no circumvention). A proof that unknown-but-bounded delay is equivalent to DLS with unknown GST would resolve the controversy and inform the distributed computing community at PODC or DISC.

2. **OAE-optimized consensus protocol with formal correctness.** A TLA+ or Ivy specification of OAE-Consensus with proved safety (agreement, validity) under the physical-layer model and liveness under bounded-delay assumptions would provide a formal foundation for industrial adoption. The key lemma is that collision detection serves as a Byzantine fault detector with bounded detection time, enabling a deterministic leader to commit in 1 round under good conditions. Publishing the protocol with a machine-checked proof at SOSP or OSDI would establish OAE as a serious engineering contribution.

3. **Failure analysis of physical-layer guarantees.** Characterize what happens when each OAE assumption is violated: cable cut (crash fault), switch malfunction (arbitrary fault), collision detection failure (message loss), timing violation (transition to asynchrony). For each case, prove whether the protocol degrades gracefully (e.g., OAE-Consensus reduces to Paxos under pure asynchrony) or fails catastrophically. A fault model taxonomy with precise conditions under which FLP's impossibility re-emerges would be publishable at DSN or SRDS.

4. **Experimental evaluation on an Ethernet testbed.** Implementing OAE-Consensus with raw Ethernet frames (bypassing TCP/IP) on a 5–10 machine cluster using DPDK or AF_PACKET, with modified drivers exposing collision detection to userspace, would provide the first empirical data on whether physical-layer consensus outperforms software-based protocols. Comparison against optimized Paxos/Raft on latency, throughput, and recovery time under injected faults (process crash, link cut, timing attack) targets NSDI or SIGCOMM.

Chapter 33

Message and Latency Complexity Tradeoffs in BFT Consensus

Is there a *fundamental* tradeoff between message complexity and latency in Byzantine fault-tolerant consensus, and what is the optimal tradeoff curve?

Byzantine fault-tolerant (BFT) consensus protocols exhibit a tension between message complexity and latency. PBFT-style protocols use $O(n^2)$ messages (each replica sends to all others) across three communication phases (pre-prepare, prepare, commit) achieving 2-round theoretical latency. DAG BFT protocols (Narwhal, Tusk) use $O(n^3)$ messages in the worst case for DAG dissemination but achieve higher throughput via pipelining. HotStuff achieves $O(n)$ messages (leader-based, linear authenticator complexity) in 3 rounds. Syrah claims $O(n \log n)$ messages in 3–4 rounds. Known lower bounds establish that BFT consensus latency requires at least 2 rounds (Dolev-Strong, authenticated model with $n \geq 3f+1$) and that message complexity is at least $\Omega(n^2)$ for synchronous BFT. But the combined complexityminimizing both message count and latency simultaneouslyis not well understood. Is there a protocol achieving $O(n)$ message complexity and 2-round latency, or must we sacrifice one for the other? We propose to characterize the optimal tradeoff curve between message complexity and latency for BFT consensus in the authenticated model.

33.1 Related Work

Protocol	Message complexity	Latency (rounds)	Notes
PBFT	$O(n^2)$	3 (2 for fast path)	Classic
Zyzzyva (speculative)	$O(n)$ opt., $O(n^2)$ pess.	1 opt., 3 pess.	Speculative execution
HotStuff	$O(n)$	3	Linear authenticator
Streamlet	$O(n^2)$ (broadcast)	$f+1$ rounds	Simple, high latency
DAG BFT (Narwhal)	$O(n^2)$ metadata + $O(n)$ data	3–5 rounds	High throughput
Syrah	$O(n \log n)$	3–4 rounds	Hybrid
DispersedLedger	$O(n)$ (data dissemination)	3–4 rounds	Erasure coding
2-round BFT (Quorum)	$O(n^2)$	2	Authenticated
Lower bound (DLS)	$\Omega(n^2)$ sync., $\Omega(n)$ async.	2-round lower bound	Existing

The combined lower boundwhether (2 rounds, $O(n)$ messages) is achievableremains open.

33.2 Proposed Approach

We develop a framework for proving combined lower bounds in the asynchronous Byzantine model with $n = 3f + 1$ replicas, authenticated channels, and crash or Byzantine failures.

Definition 33.1 (Combined Complexity Tradeoff)**.** For a BFT consensus protocol, the *complexity pair* (M, L) denotes its worst-case message complexity M (number of messages) and latency L (number of communication rounds). The protocol is *optimal* if there exists no protocol with strictly smaller M and $L \leq L'$ for a given target latency L'.

Our information-theoretic argument yields the following theorem.

Theorem 33.1 (Combined Lower Bound for BFT Consensus)**.** *Any asynchronous BFT consensus protocol with $n = 3f + 1$ replicas in the authenticated model that achieves 2-round latency must have worst-case message complexity $\Omega(n^2)$. Consequently, any protocol with $O(n)$ message complexity requires at least 3 rounds.*

Proof sketch. Assume a 2-round protocol with $o(n^2)$ messages. In round 1, at most $o(n^2)$ messages are exchanged, so there exists a replica p that receives at most $o(n)$ messages. In round 2, to tolerate f Byzantine faults, p must receive confirming messages from at least $f + 1 = \Omega(n)$ distinct replicas. Since p receives only $o(n)$ messages, at least one correct replica's message is not received by p, allowing an adversary to equivocate and cause p to decide a different value than other correct replicas, violating agreement. □

We construct the achievable tradeoff curve. Each point is either a known protocol or a missing piece we aim to fill: $(\Omega(n^2), 2)$ via PBFT's fast path; $(O(n^2), 3)$ via PBFT's normal case; $(O(n), 3)$ via HotStuff's linear authenticator complexity; $(O(n \log n), 3)$ via Syrah's logarithmic leader selection overhead; $(O(n), 4+)$ via Streamlet's simplicity. The missing piece is whether $(O(n), 2)$ is achievablewe attempt either an impossibility proof or a protocol construction.

We analyze how authentication mechanisms affect the tradeoff. Threshold signatures (BLS) reduce message size ($O(1)$ signature per round) but not message count (still $O(n)$ individual messages to collect signature shares). Trusted hardware (SGX, TEE) may reduce message count if Byzantine faults become detectable and evictable. Randomized consensus (Ben-Or, Bracha) can achieve $O(n^2)$ messages and expected $O(1)$ rounds, potentially reducing to $O(n)$ messages with $O(1)$ expected rounds.

We extend the tradeoff model to pipelined consensus. In pipelined protocols (PBFT's chain, HotStuff's chained mode), the first decision takes L rounds (pipeline fill) and subsequent decisions take 1 round each (steady state). The effective latency for k decisions is $L + (k - 1)$ rounds. In HotStuff's chained mode, each round serves as prepare for one decision and pre-commit for the next, giving $O(n)$ messages per round for the pipeline and amortized $O(n)$ messages per decision with 3-round initial latency. We investigate whether initial latency 2 with $O(n)$ per-decision messages is achievable.

Based on these analyses, we either prove impossibility (showing that any 2-round protocol requires $\Omega(n^2)$ messages and any $O(n)$-message protocol requires at least 3 rounds) or construct

a $(2, O(n))$ protocol using a committee-based approach (a small committee of size $O(\sqrt{n})$ or $O(\log n)$ handles the time-critical 2-round fast path while the full n replicas participate only in the background for liveness and safety) or erasure coding (the leader encodes the proposal into n chunks of size $O(1/n)$, giving each replica $O(1)$ messages at the cost of higher bandwidth).

Core Thesis

There exists a fundamental tradeoff between message complexity and latency in BFT consensus: 2-round protocols inherently require $\Omega(n^2)$ messages, while $O(n)$-message protocols require at least 3 rounds. This explains why HotStuff achieves linear message complexity at the cost of 3 rounds and why any protocol achieving 2 rounds necessarily incurs quadratic communication.

33.3 Evaluation

Combination	Protocol	Message complexity	Latency	Achievable?
$(n^2, 2)$	PBFT fast path	$O(n^2)$	2 rounds	Yes
$(n^2, 3)$	PBFT normal	$O(n^2)$	3 rounds	Yes
$(n, 3)$	HotStuff	$O(n)$	3 rounds	Yes
$(n \log n, 3)$	Syrah	$O(n \log n)$	3 rounds	Yes
$(n, 2)$	?	$O(n)$	2 rounds	Unknown (target)
$(\sqrt{n}, 3)$	Committee-based	$O(n)$ total	3 rounds	Maybe

33.4 Research Directions

1. **Proving the combined lower bound: $\Omega(n^2)$ messages for 2-round BFT.** An information-theoretic argument that any 2-round asynchronous BFT protocol with $n = 3f + 1$ requires $\Omega(n^2)$ messages in the worst case would close the $(O(n), 2)$ possibility. The proof must show that each replica must hear from $\Theta(n)$ distinct replicas in round 2, and that these messages cannot be aggregated without adding a round. This would explain why HotStuff needs 3 rounds and establish a fundamental tradeoff. Suitable for PODC or DISC.

2. **Constructing or ruling out a $(O(n), 2)$ protocol.** Either a protocol achieving $O(n)$ message complexity with 2-round latency using committee-based aggregation (a small $O(\sqrt{n})$ committee handles the fast path while the full n replicas provide liveness) or an impossibility proof showing that $O(n)$ messages inherently requires ≥ 3 rounds. The committee approach requires careful security analysis: the committee must be unpredictable to prevent targeted Byzantine attacks. The result (a new protocol or a tight lower bound) would be a FOCS/STOC-level contribution.

3. **Tradeoffs under threshold signatures and trusted hardware.** BLS threshold signatures reduce signature size to $O(1)$ per round but not message count. Can a protocol using BLS aggregation with $O(n)$ messages in 2 rounds exist by having the leader collect $f + 1$ partial signatures in one "logical" round (pipelining sends and receives)? Similarly, SGX/TEE

attestation allows detecting equivocation without $O(n^2)$ cross-checking. A systematic analysis of how authentication assumptions affect the tradeoff curve, with concrete constructions, would be publishable at CCS or IEEE S&P.

4. **Pipelined consensus tradeoffs.** In chained HotStuff, the first decision takes 3 rounds but subsequent decisions take 1 round (pipeline steady state). Is there a protocol where the first decision takes 2 rounds (matching the Dolev-Strong bound) while maintaining $O(n)$ messages per decision in steady state? The challenge is that shortening the pipeline fill time while keeping message complexity linear requires reusing round 1 messages for multiple decisions. An affirmative construction would substantially improve practical BFT latency. Targets USENIX ATC or EuroSys.

Chapter 34

Quantum Speedups for Exact Optimization: Beyond Grover

Can we systematically combine quantum amplitude amplification with classical algorithmic techniques to achieve exponential speedups for *structured* combinatorial optimization?

Quantum algorithms for exact combinatorial optimization currently follow a standard recipe: classical branching (divide-and-conquer, branch-and-bound) splits the problem into subproblems, and Grover search on the leaves searches the subproblem space for a solution. This gives a quadratic speedup over classical exhaustive searchif the classical algorithm explores N subproblems, the quantum algorithm explores $O(\sqrt{N})$ with Grover. Known examples include maximum independent set ($O(1.1488^n)$ quantum vs. $O(1.1996^n)$ classical) and 3-coloring. Grover is a generic search algorithm that treats the search space as unstructured, but many optimization problems have exploitable structuremonotonicity, submodularity, lattice structure, or Fourier structurewhere Grover's quadratic speedup may not be optimal. We propose to systematically combine quantum amplitude amplification (QAA), quantum walks, quantum simulated annealing, and Fourier estimation with structural properties to achieve super-quadratic or even exponential speedups for structured combinatorial optimization.

Problem 34.1 (Structure-Aware Quantum Optimization)**.** Given a combinatorial optimization problem Π over solution space $X \subseteq \{0,1\}^n$ with objective $f : X \to \mathbb{R}$ and a structural class $C \in \{\text{monotone}, \text{submodular}, \text{lattice-linear}, \text{Fourier-sparse}\}$ to which Π belongs, design a quantum algorithm Q that finds $x^* = \arg\max_{x \in X} f(x)$ using $Q(f)$ oracle queries, achieving speedup $S = T_{\text{classical}}(n)/T_{\text{quantum}}(n)$ over the best classical algorithm for Π, where S depends on structural parameters of C rather than $|X|$.

34.1 Related Work

Approach	Problem	Speedup	Technique
Grover-enhanced branching	MIS, 3-coloring, SAT	Quadratic	Grover search on branching leaves
Quantum walk for CSP	k-SAT, 3-coloring	Quadratic	Walk on assignment graph
QAOA	MAX-CUT, MAX-3-SAT	Heuristic (no proven speedup)	Variational
Amplitude amplification for DP	Knapsack, TSP	Quadratic	QAA over DP table entries
Quantum simulated annealing	MAX-CUT, Ising	Polynomial (conjectured)	Quantum tunneling
NISQ-era algorithms	Small instances	No provable speedup	Heuristic
Quantum branch and bound	General	Quadratic (worst case)	Grover at each node

No systematic framework exists for exploiting structure in combinatorial optimization beyond Grover.

34.2 Proposed Approach

Structure-Aware Initial States for QAA

The key idea is to replace the uniform superposition $|+\rangle^{\otimes n}$ in Grover's algorithm with a structured initial state $|\psi_0\rangle$ that encodes the problem's algebraic structure (lattice state, Lovász extension gradient, or quantum walk stationary distribution). When $|\psi_0\rangle$ has high overlap with the solution subspace, the required number of amplitude amplification iterations drops from $\mathcal{O}(\sqrt{N})$ to $\mathcal{O}(1/\sqrt{\langle\psi_0|\Pi|\psi_0\rangle})$, potentially yielding exponential speedups.

We classify problem structure into a hierarchy. Monotone optimization (maximum independent set, vertex cover, monotone SAT) has objectives that increase with solution size. Submodular optimization (MAX-CUT, MAX-2-SAT, max coverage) exhibits the diminishing returns property. Lattice-linear problems (SSSP, BFS, stable marriage, from Q25 and Q31) have solution spaces that are lattices. Fourier-sparse problems have objective functions with sparse Fourier representation. Symmetric problems are invariant under group actions.

For each class we design a structure-aware quantum algorithm. For monotone optimization, we use quantum simulated annealing with a monotone cooling schedule; monotonicity ensures that warm starts from nearby states are valid with provable crossover probability, targeting a polynomial speedup (n^k for $k \approx 1.5$–2). For submodular optimization, submodular functions have a continuous extension (the Lovász extension) that is convex; we apply quantum gradient descent on the continuous extension and round to a discrete solution, achieving $O(\sqrt{d})$ queries where d

is the dimension (vs. $O(d)$ classical). For lattice-linear problems, the lattice structure admits a quantum fixed point iteration: instead of correcting variables one at a time as in LLP-FW, we use quantum superposition to correct all variables simultaneously, preparing a superposition over lattice states, applying the correction function in superposition, and measuring the fixed point, targeting exponential speedup from $O(n)$ sequential corrections to $O(\log n)$ quantum rounds.

Algorithm 9 Structured Quantum Amplitude Amplification

Require: Oracle O_f, structure class C, initial state $|\psi_0\rangle$
Ensure: Solution x^* with high probability
$|\phi\rangle \leftarrow |\psi_0\rangle$ ▷ structure-encoded initial state
$R \leftarrow \textsc{ReflectAbout}(|\psi_0\rangle)$
$k \leftarrow 0$
repeat
    $|\phi\rangle \leftarrow O_f|\phi\rangle$ ▷ mark solutions
    $|\phi\rangle \leftarrow R|\phi\rangle$ ▷ reflect about $|\psi_0\rangle$
    $k \leftarrow k + 1$
until success probability $\geq 1 - \delta$ or $k > K_{\max}$
$x^* \leftarrow \textsc{Measure}(|\phi\rangle)$
return x^*

We prove lower bounds for each class. General monotone optimization requires at least $\Omega(\sqrt{N})$ queries by reduction from unstructured search. Submodular optimization with continuous extension requires at least $\Omega(\sqrt{d})$ queries to the function oracle via a maximin formulation. Lattice-linear problems require at least $\Omega(\log n)$ quantum correction rounds from the lattice height.

We generalize Grover search to structured problems via Structured QAA. Given a boolean function $f : X \to \{0, 1\}$ over a structured set X (a lattice, poset, or graph), we prepare a quantum state $|\psi_0\rangle$ encoding the structure of X (e.g., a quantum walk stationary distribution), query the oracle O_f marking solutions, and apply a reflection operator R reflecting about $|\psi_0\rangle$ instead of the uniform superposition. The success probability after k iterations depends on the overlap between $|\psi_0\rangle$ and the uniform distribution over solutions; for highly structured problems, $|\psi_0\rangle$ concentrates probability on good regions, requiring fewer iterations than Grover's $\sqrt{N}$.

We simulate all quantum algorithms for small instancesMIS on graphs with $n \leq 20$ (comparing Grover-enhanced branching vs. structured QAA with lattice-based initial state), MAX-CUT on regular graphs with $n \leq 15$ (comparing quantum gradient descent vs. classical simulated annealing vs. QAOA), and SSSP on small graphs with $n \leq 10$ (comparing quantum fixed point iteration vs. classical Bellman-Ford vs. LLP-FW)measuring quantum oracle queries, circuit depth, total gates, and success probability.

34.3 Evaluation

Problem	Metric	Quantum algorithm	Classical baseline
MIS ($n = 10$–20)	Queries to solution	Structured QAA, Grover	Branch and bound
MAX-CUT ($n = 8$–15)	Approximation ratio, queries	QGD + rounding	SA, Goemans-Williamson
SSSP ($n = 6$–10)	Queries, solution quality	Quantum lattice FP	Bellman-Ford, LLP-FW
Stable marriage ($n = 5$–10)	Queries	Quantum lattice FP	Gale-Shapley
Submodular max ($n = 10$)	Queries, approximation	QGD + rounding	Greedy, local search

34.4 Research Directions

1. **Quantum fixed-point iteration for lattice-linear problems.** The LLP framework (Q25, Q31) corrects variables sequentially until reaching a fixed point. A quantum version could prepare a superposition over all lattice states, apply the correction function in superposition, and use amplitude amplification to converge to the fixed point in $O(\log n)$ rounds rather than $O(n)$. The key is constructing a unitary that implements the lattice join of all local corrections in $O(\log n)$ depth using a binary tree of Toffoli gates. A proof-of-concept on SSSP instances ($n \leq 10$) with circuit simulation would demonstrate feasibility. Targets TQC or QIP.

2. **Quantum gradient descent for submodular optimization.** The Lovász extension of a submodular function is convex, enabling quantum gradient descent with $O(\sqrt{d})$ queries to the function oracle, compared to $O(d)$ for classical gradient descent. Applying Carathéodory rounding then yields a discrete solution. Proving that quantum gradient descent achieves a $(1 - 1/e)$ approximation for monotone submodular maximization with $O(\sqrt{n})$ oracle queries (vs $O(n)$ classical) would be a breakthrough at FOCS or STOC. The main technical challenge is implementing the projection oracle (onto the convex hull of the base polytope) efficiently on a quantum computer.

3. **Structured quantum amplitude amplification beyond Grover.** Generalizing Grover's algorithm to structured search spaces (lattices, posets, graphs) by replacing the uniform superposition initial state $|+\rangle^{\otimes n}$ with a state $|\psi_0\rangle$ that encodes the problem structure (e.g., the stationary distribution of a quantum walk on the solution space graph). The number of iterations scales as $O(1/\sqrt{\langle\psi_0|\Pi|\psi_0\rangle})$ where Π projects onto solutions, potentially yielding exponential speedups for highly structured problems. A general lower bound relating the speedup to the mixing time of the structure graph would appear at ICALP or ESA.

4. **Exponential speedup via quantum simulated annealing for monotone optimization.** Monotone optimization (MIS, vertex cover) admits a quantum simulated annealing algorithm where monotonicity ensures that low-energy states are valid starting points for higher-energy

searches. Combining the quantum Zeno effect with a monotone cooling schedule may yield a super-polynomial speedup over classical simulated annealing, though a polynomial lower bound under the oracle model likely holds. Proving tight bounds on the speedup (exponential for some monotone problems, polynomial for others) in the query model would settle open questions at the intersection of quantum algorithms and combinatorial optimization. Suitable for SIAM Journal on Computing or Quantum.

5. **Implementing structure-aware quantum algorithms on fault-tolerant hardware simulators.** Simulating structured QAA, quantum gradient descent, and fixed-point iteration on small instances ($n \leq 20$ for MIS, $n \leq 10$ for submodular) using stim or Qiskit with realistic noise models, measuring logical gate counts, T-gate depths, and success probabilities under surface code error correction. Comparison against Grover-enhanced baselines would quantify the practical advantage of structure awareness. Experimental quantum algorithm engineering papers appear at ASPLOS or ISCA.

Chapter 35

Temporal Graph Reachability: Near-Linear Time or Not?

Is there a near-linear-time algorithm for temporal reachability in general, or is there a conditional lower bound based on standard conjectures (SETH, OV)?

A temporal graph is a graph whose edges have timestamps. A temporal path from u to v is a sequence of edges (u, u_2, t_1), (u_2, u_3, t_2), $\ldots$, (u_k, v, t_k) where $t_1 \leq t_2 \leq \cdots \leq t_k$ (time-respecting order) and each consecutive edge shares a vertex. Temporal reachability asks: given a temporal graph G with n vertices and m timestamped edges, and a source vertex s, find all vertices reachable from s via a temporal path. Applications include epidemiology (contact tracing), social networks (information propagation), transportation (multi-modal journey planning), and communication networks (message routing). The naive algorithm runs a Dijkstra-like BFS over temporally sorted edges in $\mathcal{O}(m \log m)$ time. For all-pairs reachability, the naive approach runs in $\mathcal{O}(nm \log m)$. We ask whether near-linear algorithms exist for these problems or whether conditional lower bounds based on SETH or OV rule them out.

Problem 35.1 (Temporal Reachability Complexity)**.** Given a temporal graph $G = (V, E, \tau)$ where $\tau : E \to \mathbb{N}$ assigns timestamps to edges, define:

- **Single-source temporal reachability (SSTR):** find all $v \in V$ reachable from $s \in V$ via a time-respecting path.

- **All-pairs temporal reachability (APTR):** compute the temporal reachability matrix $R \in \{0,1\}^{n \times n}$ where $R_{uv} = 1$ iff u can reach v temporally.

Determine whether SSTR admits an $\mathcal{O}(m \cdot \alpha(n))$ algorithm and whether APTR admits an $\mathcal{O}(n^{2-\varepsilon})$ algorithm for any $\varepsilon > 0$, or prove SETH-based lower bounds ruling these out.

35.1 Related Work

Approach	Problem	Time complexity	Notes
BFS over time-ordered edges	Single-source	$O(m \log m)$	Standard, simple
Temporal Dijkstra (Wu et al., 2014)	Single-source, earliest arrival	$O(m \log m)$	Generalization of Dijkstra
All-pairs (naive)	All-pairs	$O(nm \log m)$	Run BFS from each source
2-hop labeling (Wang, 2020)	All-pairs, approximate	$O(m \log n)$ preprocessing, $O(1)$ query	Heuristic; no worst-case guarantee
Matrix multiplication	All-pairs, transitive closure	$O(n^\omega)$ ($\omega \approx 2.37$)	Static graphs only
Temporal transitive closure (Casteigts, 2021)	All-pairs, interval graphs	$O(n^\omega)$ under interval model	Restricted model
Fine-grained complexity of temporal problems	Various	SETH-hard for temporal diameter	Reachability lower bounds open

No near-linear algorithm for all-pairs temporal reachability is known, and no SETH-based lower bound rules one out.

35.2 Proposed Approach

OV Reduction and Temporal Union-Find

The key idea is twofold: (1) a reduction from Orthogonal Vectors to all-pairs temporal reachability shows SETH-hardness (ruling out $\mathcal{O}(n^{2-\varepsilon})$ algorithms), while (2) a novel temporal union-find data structure processes edges chronologically in $\mathcal{O}(m\alpha(n))$ time for single-source reachability, matching the near-linear target for the single-source case.

We first attempt a reduction from Orthogonal Vectors (OV) to all-pairs temporal reachability. Given two sets $A, B \subseteq \{0,1\}^d$ with $|A| = |B| = n$, the OV problem asks whether there exist $a \in A$, $b \in B$ such that $a \cdot b = 0$. We construct a temporal graph G_{OV} with $O(n)$ vertices and $O(n \log d)$ temporal edges such that $a \cdot b = 0$ iff there is a temporal path from the vertex representing a to the vertex representing b. The challenge is that the temporal dimension adds ordering constraintsedges must be time-respectingso the OV reduction must encode dot products using time ordering, not just vertex adjacency. If this construction succeeds with polynomial dependence on d, SETH-hardness implies that all-pairs temporal reachability requires $\Omega(n^{2-\varepsilon})$ time for any $\varepsilon > 0$.

For single-source temporal reachability, we improve from $\mathcal{O}(m \log m)$ to near-linear $\mathcal{O}(m\alpha(n))$ using a temporal union-find data structure. Processing edges in chronological order, we maintain a union-find with per-vertex earliest arrival time (EAT). The operation Union(u, v, t') merges components only if $\text{EAT}(u) \leq t'$ and $\text{EAT}(v) \leq t'$; Find(x) returns the component representative

Algorithm 10 Temporal Union-Find for Single-Source Reachability

Require: Temporal graph $G = (V, E, \tau)$, source s
Ensure: Reachable set $R \subseteq V$ with earliest arrival times

$R \leftarrow \{s\}$, $\text{EAT}[s] \leftarrow 0$
for each edge (u, v, t) in chronological order **do**
  if $u \in R$ and $\text{EAT}[u] \leq t$ **then**
    $\textsc{Union}(u, v,\ t)$ ▷ merge components
    $\text{EAT}[v] \leftarrow \min(\text{EAT}[v], t)$
    $R \leftarrow R \cup \{v\}$
  else if $v \in R$ and $\text{EAT}[v] \leq t$ **then**
    $\textsc{Union}(v, u,\ t)$
    $\text{EAT}[u] \leftarrow \min(\text{EAT}[u], t)$
    $R \leftarrow R \cup \{u\}$
  end if
end for
return R

and its EAT. With path compression and union-by-rank, this gives $O(m\alpha(n))$ for single-source temporal reachability.

If the OV reduction succeeds, we prove the lower bound formally using $\mathcal{O}$ notation: assuming SETH, all-pairs temporal reachability requires $\Omega(n^{2-\varepsilon})$ time even when $m = O(n \log n)$, ruling out any $\mathcal{O}(n^{1.999})$ algorithm regardless of data structure.

We design a temporal spannera subgraph H (subset of temporal edges) such that if s can reach t in G, then s can reach t in H, with H having $O(n \log n)$ or $O(n)$ edges. For each vertex we select a set of witness temporal paths covering all reachable pairs using randomized selection analogous to the Karger-Motwani-Sudhan algorithm for cut sparsifiers. If a temporal spanner with $O(n \log n)$ edges exists, all-pairs reachability on H takes $O(n \log n)$ time, enabling fast queries even on dense original graphs.

We implement and evaluate on real temporal graph datasets: email networks (Enron: 87K vertices, 1.1M edges; EU core: 986 vertices, 332K edges), contact networks (Reality Mining: 106 vertices, 3.7M edges), transportation networks (flight schedules: 5K airports, 100K flights), and synthetic OV-based worst-case instances. We compare naive all-pairs BFS ($O(nm \log m)$), temporal union-find ($O(m\alpha(n))$ single-source, $O(nm\alpha(n))$ all-pairs), and the approximate spanner plus union-find approach ($O(n \log n)$ after preprocessing).

35.3 Evaluation

Dataset	Type	Size (n, m)	Metric	Algorithm	Baseline
Enron email	Temporal	87K, 1.1M	Single-source time	Temporal UF	Naive BFS ($O(m \log m)$)
EU core email	Temporal	986, 332K	All-pairs time	Spanner + UF	Naive all-pairs BFS
Reality Mining	Contact	106, 3.7M	Reachability accuracy	Spanner	Exact BFS
Flight schedules	Temporal	5K, 100K	Reachability	Temporal UF	Dijkstra-based
Synthetic (OV-based)	Worst-case	$n = 10$–1000	Time vs. lower bound	All-pairs	Theoretical bound

35.4 Research Directions

1. **OV-based SETH lower bound for all-pairs temporal reachability.** Construct a reduction from Orthogonal Vectors to all-pairs temporal reachability where the temporal graph has $O(n)$ vertices and $O(n \log d)$ edges, and the dot product $a \cdot b = 0$ iff there exists a temporal path from the vertex encoding a to the vertex encoding b. The challenge is encoding the inner product using *time ordering* rather than adjacency: each coordinate i becomes a timestamp, and the edges for a and b are ordered so that a temporal path exists iff no coordinate has both $a_i = 1$ and $b_i = 1$. A successful reduction implies $\Omega(n^{2-\varepsilon})$ time under SETH, appearing at FOCS or STOC.

2. **Temporal union-find for near-linear single-source reachability.** A union-find data structure that processes edges in chronological order while maintaining per-vertex earliest arrival times (EAT) achieves $O(m\alpha(n))$ time for single-source temporal reachability, improving over $O(m \log m)$ Dijkstra-based methods. Formalizing the EAT-constrained union operation and proving correctness under all temporal graph models (simple, multi-edge, interval-labeled) is the main theoretical contribution. Practical optimizations (path compression with EAT propagation, union-by-rank with temporal ordering) make this a strong SODA or ESA paper.

3. **Temporal spanners for approximate all-pairs reachability.** Generalizing cut sparsifiers (Karger) to the temporal setting: does every temporal graph contain a subgraph with $O(n \log n)$ temporal edges preserving all reachability relations? A randomized construction using importance sampling over temporal paths would provide the first polynomial-time algorithm for approximate all-pairs temporal reachability with near-linear preprocessing and $O(1)$ query time. Proving an $\Omega(n \log n)$ lower bound via a temporal analogue of the butterfly network would complete the picture. Targets ITCS or ICALP.

4. **Experimental evaluation of temporal reachability algorithms.** A comprehensive benchmark comparing naive BFS ($O(m \log m)$), temporal union-find ($O(m\alpha(n))$), temporal spanner preprocessing, and 2-hop labeling on large real-world datasets (Enron email, Reality Mining contacts, US flight schedules, Twitter follow graphs). Metrics include preprocessing

time, query time, reachability accuracy (for approximate methods), and scalability up to 10^7 edges. A thorough experimental study with open-source implementations would be a reference for practitioners, suitable for VLDB or ACM SIGMOD Record.

5. **Parameterized complexity of temporal reachability.** Beyond worst-case bounds, investigate temporal reachability parameterized by the number of distinct timestamps τ, the maximum label multiplicity ℓ, or the interval graph width ω. Fixed-parameter tractable algorithms (e.g., $O(2^\tau(n+m))$ for small τ) could explain why temporal reachability is easy on real datasets with few time steps. A parameterized complexity classification (which parameters make the problem FPT, W[1]-hard, or para-NP-hard) fills a gap in the temporal graph literature. Suitable for IPEC or Algorithmica.

Chapter 36

Learning-Augmented Online Algorithms: Provable Robustness

Can we design online algorithms that are *robust* to arbitrarily bad predictions: never worse than the classic worst-case bound by more than an additive constant) while achieving near-optimal performance when predictions are good?

Online algorithms with predictions use machine learning predictions to improve performance while maintaining worst-case guarantees. The standard framework evaluates two properties: consistency (when predictions are perfect, the algorithm achieves better-than-worst-case performance) and robustness (when predictions are arbitrarily wrong, the algorithm performs no worse than the classic worst-case bound up to a constant factor). Classic examples include caching with predicted page usage (LRU+P achieves $\mathcal{O}(\log k)$ consistency and $\mathcal{O}(\log k)$ multiplicative robustness) and ski rental with predicted ski days. Existing results suffer from two limitations: they assume prediction error is bounded by a known parameter λ, whereas in practice predictions can be arbitrarily wrong; and robustness is typically multiplicative ($\mathcal{O}(1)\times$ worst-case) when for some problems we need additive guarantees (never more than C worse than the classic algorithm, no matter how bad the prediction). We propose a general framework for learning-augmented algorithms that achieves strong additive robustness under arbitrarily bad predictions, without assuming known prediction error bounds.

Problem 36.1 (Additive-Robust Learning-Augmented Online Algorithms)**.** Given a classic online algorithm A with competitive ratio c, a prediction oracle P that at each step t outputs a predicted action $a_{\text{pred}}^{(t)}$, and a problem instance I with cost function cost : actions $\times$ inputs $\to \mathbb{R}_{\geq 0}$, design a meta-algorithm $M(A, P)$ producing A' such that:

- **Consistency:** if P is perfect on I, then $\text{cost}(A', I) \leq c' \cdot \text{OPT}(I)$ where $c' < c$.

- **Additive robustness:** for any P (even adversarial), $\text{cost}(A', I) \leq c \cdot \text{OPT}(I) + \Delta$ where Δ is a finite constant independent of $|I|$.

36.1 Related Work

Algorithm	Problem	Consistency	Robustness	Error assumption
LRU+P (Lykouris, 2018)	Caching	$O(\log k)$	$O(\log k)$ mult.	Known η
KiWi (Rohatgi, 2020)	Caching	$O(1)$	$O(\log k)$ mult.	Known η
Ski rental (Purohit, 2018)	Ski rental	2 (1 prediction)	2 mult.	None
ML-augmented ski rental	Ski rental	c_{cons}	c_{rob} mult.	Known λ
Online bipartite matching	Matching	$O(1/\alpha)$	2 mult.	Bounded error
K-page migration	Page migration	$O(1)$	$O(\log k)$ mult.	Known η
Online set cover	Set cover	$O(\log n)$	$O(\log n)$ mult.	Known η

All results assume prediction error bounded by a known parameter; without this, the additive gap over the classic algorithm is uncontrolled.

36.2 Proposed Approach

Regret-Test Switching with Additive Bounds

The key idea is a meta-algorithm that runs both the learned action and the classic action in parallel, tracking their cumulative cost difference via a regret test (CUSUM or window-based). When the learned cost exceeds the classic cost by more than a threshold Δ, the algorithm switches permanently to the classic action, ensuring the total additive overhead is bounded by Δ plus at most one window of learned suboptimality.

We define a metal-algorithm M that takes any classic online algorithm A (with known competitive ratio c) and a prediction oracle P, and produces a new algorithm A' guaranteeing consistency (if predictions are perfect, A' achieves competitive ratio $c' < c$) and additive robustness (for any prediction, even adversarial, A' achieves competitive ratio $\leq c + \varepsilon$ where ε is an additive constant). At each step t, M computes the learned action $a_{\text{learned}}(t)$ based on P and the classic action $a_{\text{classic}}(t)$ using A, then chooses between them based on a regret test that compares the observed cumulative cost of the learned action against the classic action: if the learned cumulative cost exceeds the classic cumulative cost by more than a threshold Δ, the algorithm switches to classic and stays there for a bounded time Λ; otherwise it continues with the learned action. If predictions are perfect, the regret test never triggers and the algorithm follows the learned path with cost $c' \cdot \text{OPT}$;

if predictions are adversarial, the regret test detects the divergence within at most T steps and switches to classic, bounding total cost at classic_cost $+ \Delta + O(T \cdot \text{cost\_per\_step})$.

We design three regret test variants. The simple cumulative test switches when $c_{\text{learned}}(t) - c_{\text{classic}}(t) > \Delta$, but may switch too late (if learned cost spikes suddenly) or too early (if classic cost is unlucky). The CUSUM-style test tracks the cumulative difference $d(t) = c_{\text{learned}}(t) - c_{\text{classic}}(t)$, switching and resetting when $d(t)$ exceeds Δ; the expected switching cost is bounded by the probability that the classic algorithm happens to have a good run while the learned algorithm has a bad one. The window-based test (parameter-free) maintains a sliding window of the last W steps, switching if the learned window cost exceeds the classic window cost by more than δ, with W and δ set based on problem structure; this limits the maximum overhead from bad predictions to $W \times$ (max per-step cost) $+ \delta$.

Algorithm 11 Additive-Robust Meta-Algorithm with Regret Test

Require: Classic algorithm A, predictor P, threshold Δ
Ensure: Online algorithm A' with additive robustness

$c_{\text{learned}} \leftarrow 0$, $c_{\text{classic}} \leftarrow 0$
for each timestep $t = 1, 2, \ldots$ **do**
    $a_{\text{learned}} \leftarrow P(t)$
    $a_{\text{classic}} \leftarrow A(t)$
    if InClassicMode **then**
        Execute a_{classic}
    else
        Execute a_{learned}
        $c_{\text{learned}} \leftarrow c_{\text{learned}} + \text{cost}(a_{\text{learned}}, t)$
        $c_{\text{classic}} \leftarrow c_{\text{classic}} + \text{cost}(a_{\text{classic}}, t)$
        if $c_{\text{learned}} - c_{\text{classic}} > \Delta$ **then**
            InClassicMode $\leftarrow$ true ▷ permanent switch
        end if
    end if
end for

We instantiate the framework for specific problems. For caching, the classic algorithm is LRU (competitive ratio $\mathcal{O}(\log k)$) and the prediction is the next access time for each page; our algorithm follows the predicted optimal replacement (like Belady's MIN) with a fallback to LRU if predictions prove unreliable, achieving near-optimal consistency with $\mathcal{O}(W)$ additive overhead. For ski rental, the classic deterministic algorithm has competitive ratio 2; if the predicted number of ski days exceeds a threshold we buy, otherwise we rent, applying the regret test to force an early buy if renting exceeds the predicted cost plus Δ, achieving additive robustness. For online scheduling to minimize makespan with predicted job sizes, the classic List Scheduling algorithm has competitive ratio $2 - 1/m$; we schedule by predicted times and fall back to List Scheduling with additive Δ overhead.

We prove that additive robustness is impossible for some problems, such as the k-server problem on a metric space with $k \geq 2$, where no learning-augmented algorithm can achieve both consistency better than $O(k)$ and additive robustness with a finite additive constant. The characterization shows that additive robustness is possible for problems with fast fallbackwhere the classic algorithm's state can be recovered quickly (caching: flush and restart from clean cache takes $O(k)$ misses)but impossible where state recovery is expensive (k-server: the positions of k servers can be anywhere in the metric space).

We evaluate on real-world datasets: Wikipedia page traces for caching, synthetic ski rental with varying prediction error distributions, Google cluster traces for scheduling, and GPS traffic data for online routing, comparing against the classic algorithm alone, existing learning-augmented algorithms with multiplicative robustness, and an oracle with perfect predictions.

36.3 Evaluation

Problem	Metric	Comparison
Caching (Wikipedia traces)	Miss rate, competitive ratio	LRU, LRU+P, KiWi
Ski rental (synthetic)	Total cost, additive overhead	Classic (ratio 2), existing LA
Scheduling (Google traces)	Makespan, competitive ratio	List Scheduling, ML-augmented
Routing (Uber data)	Travel time, competitive ratio	Classic shortest-path, ML-based

We target less than 5% additive overhead for non-adversarial prediction errors.

36.4 Research Directions

1. **The metal-algorithm framework with additive robustness.** The core contribution is a meta-algorithm M that wraps any classic online algorithm A with competitive ratio c and a prediction oracle P, producing A' with consistency $c' < c$ and additive robustness $\leq c + \varepsilon$. The regret-test switching rule (switch to A when learned cumulative cost exceeds classic cost by Δ) must be analyzed to bound the additive overhead as a function of Δ, the detection delay, and the maximum per-step cost. Proving that additive robustness with $\varepsilon = O(1)$ is achievable for caching and ski rental but not for k-server would establish the first impossibility result for learning-augmented algorithms, publishable at FOCS or STOC.

2. **Parameter-free regret tests with provably finite additive overhead.** The CUSUM-style test (tracking cumulative cost difference $d(t) = c_{\text{learned}}(t) - c_{\text{classic}}(t)$, switching when $d(t) > \Delta$) requires a user-specified Δ. Developing a parameter-free test that adaptively sets Δ based on observed cost variance (using a multi-scale monitoring approach or a sequential probability ratio test) would eliminate the need for problem-specific tuning. Bounding the expected additive overhead in terms of the prediction error distribution (rather than a worst-case bound) bridges theory and practice. Targets COLT or ALT.

3. **Characterizing when additive robustness is possible.** Not all online problems admit learning-augmented algorithms with additive robustness. For k-server, any learning-augmented algorithm with consistency $o(k)$ must have multiplicative (not additive) robustness. The proof uses a lower bound construction where the adversary simulates a worst-case request sequence while the prediction oracle provides perfect advice for a different (expensive) server configuration. A general characterization (additive robustness is achievable iff the classic algorithm's state can be reset in $O(1)$ cost) would unify the known positive results (caching, ski rental) and negative results (k-server, paging with variable-sized pages). Suitable for ITCS or SODA.

4. **Instantiations for scheduling, routing, and metrical task systems.** Applying the additive-robustness framework to online makespan minimization (List Scheduling baseline), online shortest-path routing (deterministic or H-competitive), and metrical task systems (Work Function Algorithm baseline) would demonstrate generality. For each problem, the regret test must be adapted to the specific cost structure: scheduling cost is additive per job, routing cost is path-dependent, MTS cost is movement-plus-service. Experimental evaluation on Google cluster traces, GPS traffic data, and synthetic MTS instances would validate that additive robustness ($< 5\%$ overhead) holds under realistic prediction error distributions while maintaining near-optimal consistency. Targets NeurIPS or ICML.

Part VII

Automata & Formal Languages

Chapter 37

A Myhill-Nerode Theorem for Pushdown Automata

Can we extend Myhill-Nerode theory to a natural subclass of context-free languages where a canonical minimal pushdown automaton exists?

Myhill-Nerode theory is one of the most elegant results in automata theory: for regular languages, the equivalence relation $\equiv_L$ defines the minimal DFA whose states are equivalence classes of strings, the minimal DFA is unique up to isomorphism, and an algorithm (Moore or Hopcroft) can efficiently compute it. For context-free languages recognized by pushdown automata (PDAs), no such theory exists: there is no unique minimal PDA, state minimization for PDAs is undecidable in general, and no canonical construction analogous to "states are equivalence classes of strings" is known. Recent work has extended Myhill-Nerode theory to restricted models including one-clock timed automata, integer-reset timed automata, generalized DFAs, and higher-dimensional automata, but no characterization exists for any natural subclass of pushdown languages. We propose to extend Myhill-Nerode theory to visibly pushdown languages (VPLs), a well-behaved subclass of deterministic context-free languages with call-return matching structure that is closed under complementation and has decidable equivalence.

37.1 Related Work

Language class	Myhill-Nerode?	Minimal automaton?	Minimization complexity
Regular (DFA)	Yes (classical)	Unique minimal DFA	$O(n \log n)$ (Hopcroft)
One-clock DTA	Yes (TACAS 2026)	Unique	Polynomial
IRTA (integer-reset)	Yes (K-monotonic)	Unique	Polynomial
Generalized DFA	Conditional (fix $W(A)$)	Unique after fixing $W(A)$	Polynomial
HDA (higher-dim)	Yes (regular case)	Unique for regular HDAs	Open
DSA (suffix-reading)	No	None	NP-complete
DCFL (det. context-free)	Unknown	Unknown	Undecidable
VPL (visibly pushdown)	Partial	Unknown	Unknown
Bounded-stack DCFL	Unknown	Unknown	Unknown

No Myhill-Nerode characterization exists for any natural subclass of pushdown languages.

37.2 Proposed Approach

We define the Myhill-Nerode equivalence for VPLs over the visibly pushdown alphabet $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{int}} \cup \Sigma_{\text{ret}}$.

Definition 37.1 (Myhill-Nerode Equivalence for VPLs)**.** Let $L \subseteq \Sigma^*$ be a visibly pushdown language over the partitioned alphabet $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{int}} \cup \Sigma_{\text{ret}}$. Define the equivalence $\equiv_L$ on pairs (x, s) where $x \in \Sigma^*$ and $s \in \Gamma^*$ is the stack content after processing x: $(x, s) \equiv_L (y, t)$ iff for all $z \in \Sigma^*$, the deterministic VPA for L starting with stack s after x accepts $x \cdot z$ iff it accepts $y \cdot z$ starting with stack t.

The domain of (x, s) is infinite (the stack can grow arbitrarily), but we conjecture that for any VPL L, this equivalence has finite index when restricted to reachable configurations of the minimal deterministic VPA, and the equivalence classes correspond to the VPA's states with the stack determined by the state's stack context.

If the equivalence has finite index, we construct the minimal deterministic VPA.

Theorem 37.1 (Myhill-Nerode Theorem for Visibly Pushdown Languages)**.** *For any VPL L, the Myhill-Nerode equivalence $\equiv_L$ has finite index on the set of reachable configurations of the minimal deterministic VPA. The quotient automaton whose states are equivalence classes $[(x, s)]_{\equiv_L}$ is the unique (up to isomorphism) minimal deterministic VPA for L.*

Proof sketch. Finiteness: Show that the set of reachable configurations of a deterministic VPA is a regular set of stack configurations, and the equivalence $\equiv_L$ restricted to this set refines a finite-index congruence on the regular set. Uniqueness: For any two minimal DVPAs A and B for L, construct a bijection between their state sets by mapping each state of A to the $\equiv_L$-class of the configurations it represents. The mapping is a VPA isomorphism by the Myhill-Nerode property. □

The minimization algorithm computes the Myhill-Nerode equivalence using a variant of Hopcroft's algorithm adapted to the pushdown structure, iteratively refining the partition of configurations where configurations are merged if they are indistinguishable for future inputs.

We extend Angluin's L^* algorithm for active learning to VPLs. The L^*_{VPL} learner maintains an observation table with rows indexed by strings $x \in \Sigma^*$ and columns indexed by strings with balanced call-return pairs. The table entry records whether the teacher accepts $x \cdot e$ where e is a balanced extension. The learner asks membership queries for $x \cdot e$ and stack queries (the stack content after x). When the table is closed and consistent, the learner constructs a hypothesized VPA; the teacher either confirms equivalence or provides a counterexample. We conjecture polynomial query complexity in the number of states of the minimal VPA and the number of call-return pairs.

We extend the characterization to bounded-stack DCFLs. A DCFL has bounded stack if there exists a deterministic PDA with stack height bounded by a function $f(n)$ of input length. For constant stack height, the set of reachable (state, stack) pairs is finite and the Myhill-Nerode equivalence has finite index, yielding a canonical minimal PDA. For $O(\log n)$ stack height, the index is polynomial in n, making the characterization asymptotically infinite but practically useful for minimization.

Core Thesis

Visibly pushdown languages admit a Myhill-Nerode characterization: the minimal deterministic VPA is uniquely determined by the equivalence $\equiv_L$ on configuration pairs, providing the first canonical minimization for a natural subclass of context-free languages and enabling active learning algorithms with polynomial query complexity.

We implement the VPA minimization algorithm and evaluate on random deterministic VPAs, XML and JSON schema validation automata from real-world specifications, and CFL-reachability VPAs derived from program analysis in compilers.

37.3 Evaluation

Automaton type	Size (states)	Minimization time	Minimal states	Correctness
Random VPA	10–100	$O(n \log n)$ measured	Unique up to renaming	Verified by equivalence
XML Schema VPA	20–50	Measure	Compare to hand-optimized	XSD validation
JSON Schema VPA	10–30	Measure	Minimized	JSON validation
CFL-reachability VPA	50–200	Measure	Minimized	Program analysis
Bounded-stack PDA	10–50	Measure	Minimized	Verified

37.4 Research Directions

R1: Myhill-Nerode for Visibly Pushdown Languages with Bounded Stack Height. We propose a fine-grained analysis of bounded-stack VPLs, where the stack height is bounded by a fixed constant k. For such languages, the Myhill-Nerode equivalence has finite index and yields a canonical minimal VPA. The open question is whether the minimization can be carried out in $O(n \log n)$ time via a variant of Hopcroft's algorithm adapted to pushdown configurations, and whether the canonical VPA is unique up to isomorphism. Target venue: LICS. Novelty: first canonical minimization for a subclass of context-free languages. Expected contribution: an algorithm and proof of uniqueness.

R2: Undecidability of Myhill-Nerode for General Deterministic Context-Free Languages. We conjecture that no Myhill-Nerode characterization exists for all DCFLs, because the set of reachable configurations of a deterministic PDA is not recursively enumerable in general. Proving this would establish that VPLs and bounded-stack DCFLs are maximal subclasses admitting such a theory. Target venue: ICALP. Novelty: hardness result at the boundary of pushdown languages. Expected contribution: a precise characterization of which context-free subclasses admit a Myhill-Nerode theorem.

R3: Active Learning of Visibly Pushdown Languages via $\mathbf{L}^*_{\mathbf{VPL}}$. Building on the

Myhill-Nerode characterization, we design an L^* variant that learns a VPA using membership, equivalence, and stack-content queries. The key challenge is managing the infinite set of possible stack configurations while maintaining polynomial query complexity in the number of VPA states and call-return pairs. Target venue: CONCUR. Novelty: first active learning algorithm for visibly pushdown languages. Expected contribution: a complete learning algorithm with query complexity $O(k \cdot n^2 \cdot r^2)$ where r is the number of call-return types.

R4: Canonical Minimization of Bounded-Stack Deterministic Pushdown Automata. We extend the proposed VPA minimization to general deterministic PDAs with stack bounded by $f(n) = O(\log n)$. For this class the Myhill-Nerode equivalence has polynomially many classes as a function of n, yielding a canonical PDA that is not minimal but has bounded redundancy. We ask whether a polynomial-time reduction to the minimal PDA exists. Target venue: STACS. Novelty: first practical minimization result for pushdown automata beyond VPLs. Expected contribution: a minimization algorithm with approximation guarantees.

R5: Tool Support for VPA Minimization and Verification. We implement the VPA minimization algorithm and evaluate on XML/JSON schema validators and CFL-reachability VPAs from compiler program analyses. The tool should demonstrate that canonical minimization reduces automaton size by 2–10× on real-world benchmarks. Target venue: LMCS (special issue on implementation). Novelty: first practical VPA minimization tool. Expected contribution: an open-source tool and benchmark suite.

Chapter 38

State Complexity Tradeoffs Between NFAs and DFAs Under Operations

Can we derive *tight* bounds for NFA/DFA state complexity for all common operations, and characterize the languages that achieve the worst-case bounds?

The state complexity $\mathrm{sc}(L)$ of a regular language L is the number of states in its minimal DFA; the non-deterministic state complexity $\mathrm{nsc}(L)$ is the number of states in its minimal NFA. Converting an NFA to a DFA can cause exponential state blowupthere exist families of languages L_n such that $\mathrm{sc}(L_n) = 2^n$ but $\mathrm{nsc}(L_n) = n{+}1$. The state complexity under operations reveals deeper structure: for union, $\mathrm{sc}(L_1 \cup L_2) \leq \mathrm{sc}(L_1) \cdot \mathrm{sc}(L_2)$ (tight); for concatenation, $\mathrm{sc}(L_1 \cdot L_2) \leq (2^{\mathrm{sc}(L_1)} - 1) \cdot (2^{\mathrm{sc}(L_2)} - 1)$, which is double-exponential in NFA state complexity; for Kleene star, $\mathrm{sc}(L^*) \leq 2^{\mathrm{sc}(L)-1} + 2^{\mathrm{sc}(L)-2} - 1$. For NFAs, union costs $\mathrm{nsc}(L_1) + \mathrm{nsc}(L_2) + 1$, concatenation costs $\mathrm{nsc}(L_1) + \mathrm{nsc}(L_2)$, and Kleene star costs $\mathrm{nsc}(L) + 1$, all tight. However, many gaps remain: the complete characterization of which language families simultaneously achieve bounds for multiple operations is lacking; the switching cost between representations after applying operations is not fully understood; and state complexity for alternating finite automata (AFA) under operations is largely open. We propose to derive a complete table of tight bounds for all common operations on DFAs, NFAs, and AFAsincluding combined operationsand to characterize the languages achieving the worst-case bounds.

38.1 Related Work

Operation	**DFA state complexity**	**NFA state complexity**	**Known witness?**
Union ($\cup$)	$m \cdot n$	$m + n + 1$	Yes (both)
Intersection ($\cap$)	$m \cdot n$	$m \cdot n$	Yes (both)
Concatenation ($\cdot$)	$(2^m - 1)(2^n - 1)$	$m + n$	Yes
Kleene star ($*$)	$2^{m-1} + 2^{m-2} - 1$	$m + 1$	Yes
Reversal (R)	2^m	m	Yes
Complementation ($\overline{L}$)	m (same DFA)	2^n (to DFA)	N/A
Homomorphism (h)	$\leq 2^m$	$\leq m + c$	Open
Composition ($\cup \cap *$, **etc.**)	Various	Various	Incomplete

The simultaneous bounds (multiple operations on the same language) and AFA state complexity are not characterized.

38.2 Proposed Approach

For each operation and each combination of up to three operations, we compute tight bounds.

Definition 38.1 (State Complexity Measures)**.** The *deterministic state complexity* $\mathrm{sc}(L)$ of a regular language L is the number of states in its minimal DFA. The *nondeterministic state complexity* $\mathrm{nsc}(L)$ is the number of states in its minimal NFA. The *alternating state complexity* $\mathrm{asc}(L)$ is the number of states in its minimal AFA. For an operation op, the state complexity of op is the maximum of $\mathrm{sc}(\mathrm{op}(L_1, \ldots, L_k))$ over all languages L_i with $\mathrm{sc}(L_i) \leq n_i$.

For the upper bound, given DFAs A_1 (m states) and A_2 (n states), we construct a DFA A for $\mathrm{op}(A_1, A_2)$ using product constructions or variants, minimizing the result. For the lower bound, we construct witness languages $L_{m,n}$ such that $\mathrm{sc}(\mathrm{op}(L_1, L_2))$ meets the claimed bound, proving minimality using the fooling set method (for NFA) or the distinguishability method (for DFA). For combined operations such as $\mathrm{op}_1(L_1, \mathrm{op}_2(L_2, L_3))$, we compute whether the bound is the composition of individual bounds or smaller due to structural interaction.

We characterize the witness languages that achieve the worst-case bounds. A family $\{L_{m,n}\}$ is a worst-case witness for operation op if $\mathrm{sc}(\mathrm{op}(L_1, L_2)) = f(m, n)$ (the tight bound) and there is a uniform construction over some alphabet $\Sigma_{m,n}$.

Theorem 38.1 (Tight Bounds for Combined Operations)**.** *For DFAs with m and n states respectively, the state complexity of concatenation is exactly $(2^m - 1)(2^n - 1)$, and this bound is tight. For NFA union, the tight bound is $m + n + 1$, and for NFA Kleene star it is $m + 1$. For combined operations, e.g., $(L_1 \cup L_2) \cdot L_3$, the tight bound is $(2^{m \cdot n} - 1)(2^p - 1)$.*

Proof sketch. Upper bounds follow from standard product constructions and subset constructions, applying the corresponding operation and then minimizing. Lower bounds are established via witness language families: for DFA concatenation, the witness $L_m \cdot L_n$ uses delayed acceptance requiring the subset construction, and the fooling set of size $(2^m - 1)(2^n - 1)$ establishes the lower bound. For combined operations, the product construction composes naturally, and the witness languages are Cartesian products of the primitive witnesses. □

Known witnesses for union and intersection are languages over $\{a_1, \ldots, a_m, b_1, \ldots, b_n\}$ where $L_1 = \{w : w \text{ contains at least one } a_i\}$ and $L_2 = \{w : w \text{ contains at least one } b_j\}$; for concatenation, the witness involves delayed acceptance requiring simultaneous memory of both languages' states. We seek a canonical form: any worst-case language must have a specific algebraic property, such as its syntactic monoid being a full transformation monoid.

We analyze bounds over restricted alphabets. For unary alphabets ($\Sigma = \{a\}$), DFA state complexity is linear or quadratic (never exponential) because every unary DFA is a cycle plus a path, and NFA succinctness for unary languages is at most $O(\log n)$ states vs. $O(n)$ for DFA. For binary alphabets, we conjecture that any worst-case bound $B(m, n)$ over a general alphabet can be achieved over a binary alphabet with the same asymptotic bound $\Theta(B(m, n))$, because prefix codes encode m-ary alphabets into binary strings of length $\log m$.

We extend the analysis to alternating finite automata (AFA), which allow both existential and universal non-determinism and can be exponentially more succinct than both NFA and DFAfor example, the language $\{w \in \{0,1\}^* : w[n] = 1\}$ requires $O(2^n)$ NFA or DFA states but only $O(n)$

AFA states. We determine tight bounds for AFA union, intersection, concatenation, and Kleene star, investigating whether AFA state complexity under operations is always polynomial in the component AFA sizes or whether concatenation causes exponential blowup even in AFA.

We implement a state complexity calculator tool that takes a language description (regular expression, DFA, NFA, or AFA) and an operation, returns the upper bound on state complexity, a witness language achieving the bound if known, and the minimal automaton for the result. The tool verifies tight bounds for small m, n by exhaustive enumeration and discovers new bounds for combined operations.

Core Thesis

The state complexity of regular language operations follows tight bounds that are either multiplicative (DFA union/intersection: $m \cdot n$), exponential (DFA concatenation: $(2^m - 1)(2^n - 1)$), or additive (NFA union: $m + n + 1$), with worst-case witness languages characterized by their syntactic monoid structure.

38.3 Evaluation

Operation(s)	Measure	Current best	Target result
DFA $\cup$	Bound and witness	$m \cdot n$ (tight)	Verified + characterization
DFA $\cdot$	Bound and witness	$(2^m - 1)(2^n - 1)$	Single-alphabet witness?
NFA $\cup$	Bound and witness	$m + n + 1$	Verified + characterization
NFA $\cap$	Bound and witness	$m \cdot n$	Verified
Combined $(L_1 \cup L_2) \cdot L_3$	New bound	Unknown	**New result**
AFA $\cdot$	New bound	Unknown	**New result**
Unary alphabet (all ops)	Bounds	Partial	**Complete table**

38.4 Research Directions

R1: Complete Table of Combined Operation Bounds. We propose to compute tight state complexity bounds for all combinations of up to three operations on DFAs, NFAs, and AFAs. For example, the composition $(L_1 \cup L_2) \cdot L_3$ on DFAs has an upper bound of $(2^{m \cdot n} - 1)(2^p - 1)$; the question is whether this bound is tight and which witness languages achieve it. Target venue: ICALP. Novelty: first systematic treatment of combined operation complexity. Expected contribution: a complete reference table and witness constructions for 20+ operation combinations.

R2: AFA State Complexity Under Operations. Alternating finite automata can be exponentially more succinct than NFAs, but their state complexity under operations is largely unexplored. We conjecture that AFA union and intersection cost at most $m + n$ states (additive rather than multiplicative), while concatenation causes exponential blowup even in AFA. We

will prove tight bounds and characterize witness languages. Target venue: LICS. Novelty: first comprehensive AFA state complexity analysis. Expected contribution: tight bounds for all standard operations on AFAs.

R3: Uniform Worst-Case Witness Characterization via Syntactic Monoids. We conjecture that any language achieving the worst-case state complexity for an operation must have a syntactic monoid isomorphic to a full transformation monoid or a subsemigroup thereof. Proving this would provide an algebraic characterization of worst-case witnesses. Target venue: LMCS. Novelty: algebraic characterization of extremal languages. Expected contribution: a necessary condition on the syntactic monoid for extremal state complexity.

R4: Unary and Small-Alphabet State Complexity. For unary alphabets, DFA state complexity is never exponential, but the exact polynomial bounds for all operations are not fully known. We aim to complete the table for unary DFAs, NFAs, and AFAs, and prove that binary alphabets suffice to achieve any asymptotic bound achievable over arbitrarily large alphabets. Target venue: STACS. Novelty: closure of the small-alphabet complexity program. Expected contribution: a complete complexity table for unary alphabets and a reduction theorem for binary alphabets.

R5: State Complexity Calculator Tool. We implement a tool that, given a language description and an operation, returns the tight bound, a witness construction, and the minimal automaton. The tool verifies bounds for small parameters by exhaustive enumeration and discovers new bounds for combined operations. Target venue: CONCUR (tool track). Novelty: first automated state complexity reasoning tool. Expected contribution: an open-source tool and a database of known bounds.

Chapter 39

Scalable Active Automata Learning for Infinite-State Systems

Can we design an active learning algorithm whose query complexity is *polynomial in the size of the target model* regardless of the data domain?

Active automata learning (Angluin's L* algorithm) infers a finite automaton by asking membership queries (is string w in the language?) and equivalence queries (is the current hypothesis correct?), learning a DFA with $\mathcal{O}(kn^2)$ queries where n is the number of states and k is the alphabet size. L* has been extended to more expressive modelsMealy machines, register automata (infinite-state systems with data values), symbolic automata (transitions labeled with predicates over infinite domains), and timed automatabut as the model class becomes more expressive, query complexity grows exponentially in the model parameters (number of registers, data domain size, number of clocks), making active learning impractical for large or infinite-state systems. We propose to design an active learning algorithm whose query complexity is polynomial in the size of the target model (states, registers, clocks) and independent of the size of the data domain: polynomial in n and r for register automata, in n and c for timed automata, and in n and s for symbolic automata.

Problem 39.1 (Polynomial Active Learning for Infinite-State Models)**.** Given a black-box system $\mathcal{S}$ modeled as a register automaton with n states and r registers over an infinite data domain D, or a timed automaton with n states and c clocks, design an active learning algorithm L^*_{PRA} that outputs a hypothesis H equivalent to $\mathcal{S}$ using at most $\text{poly}(n, r, k, c)$ membership and equivalence queries, where the polynomial is independent of $|D|$ and the maximum constant M.

39.1 Related Work

Model class	Algorithm	Query complexity	Domain dependence
DFA	L*, TTT, Observation Pack	$O(kn^2)$	Alphabet size k
Mealy machine	$\text{L}^*_{\text{Mealy}}$, TTT	$O(kn^2)$	Alphabet size k
Register automata	L^*_{RA} (RALib)	$O(k^n \cdot D^r)$	Domain D, registers r
Symbolic automata	L^*_{sym}	$O(P^k \cdot n^2)$	Predicates P
Timed automata (1-clock)	L^*_{TA} (Tomte)	$O(n^2 \cdot \max \text{const})$	Max constant
Timed automata (multi-clock)	(Open problem)		
Nerode for IRTA	L^*_{IRTA}	Polynomial in $n \cdot r$	K-monotonic

All extensions beyond DFA have exponential dependence on domain size, number of registers, or number of predicates.

39.2 Proposed Approach

Symbolic Myhill-Nerode Equivalence

The key idea is a register-aware Myhill-Nerode equivalence that groups data values into symbolic classes (equivalence classes indistinguishable by the automaton's transitions). Since the automaton has r registers and n states, at most $\mathcal{O}(n \cdot r)$ symbolic classes exist, independent of the data domain size. This reduces query complexity from exponential in r to polynomial in $n \cdot r$.

We design $\mathrm{L}^*_{\mathrm{PRA}}$ (Polynomial Register Automata), a new active learning algorithm for register automata. Instead of tracking each data value separately (which causes exponential blowup), we track symbolic valuesequivalence classes of data values that cannot be distinguished by the automaton's transitions. We define a register-aware Myhill-Nerode equivalence over configurations (state, register assignment) that captures the automaton's behavior, with equivalence classes corresponding to states of the canonical register automaton. For each transition, the learner asks membership queries for a representative of each symbolic value class; because the automaton has r registers and n states, there are at most $O(n \cdot r)$ symbolic classes to explore, independent of the domain size. The query complexity is $\mathcal{O}(k \cdot n^2 \cdot r^c)$ for some constant $c \leq 3$, bounded by the number of states times registers times alphabet size, times a polynomial factor for hypothesis construction.

We extend to timed automata using zone abstraction, grouping clock assignments into zones (sets of assignments satisfying a set of constraints). For c-clock timed automata with n states and maximum constant M, the number of zones is $O((n \cdot M + c)^c)$exponential in c but polynomial in n and M. We conjecture the query complexity is $O(n^2 \cdot M^c \cdot c!)$, acceptable for small $c \leq 3$.

Algorithm 12 CEGAR-Based Active Learning for Infinite-State Systems

Require: Target system $\mathcal{S}$, initial abstraction α_0
Ensure: Hypothesis H equivalent to $\mathcal{S}$

$H \leftarrow \varnothing$
repeat
  $H \leftarrow \textsc{LearnAbstract}(\mathcal{S}, \alpha)$ ▷ $\mathrm{L}^*_{\mathrm{PRA}}$
  $\sigma \leftarrow \textsc{EquivalenceQuery}(H, \mathcal{S})$
  if $\sigma = \varnothing$ **then return** H
  end if
  $\alpha \leftarrow \textsc{RefineAbstraction}(\alpha, \sigma)$ ▷ split class
until refinement rounds $\geq n \cdot r$
return H

We design a CEGAR-style active learning algorithm using abstraction refinement. Starting with a coarse partition of the data domain (all values equivalent), we run $\mathrm{L}^*_{\mathrm{PRA}}$ with the abstract domain to produce a hypothesis automaton. The equivalence oracle checks the hypothesis against the target system: if correct, we are done; if a counterexample string w exists, we analyze w to identify which abstraction class needs to be split and re-learn. Since the target automaton has finite states, the abstraction can be refined at most $n \times r$ times, each round using polynomial queries.

For the equivalence oracle, which is undecidable in general for infinite-state models, we design a bounded equivalence checker that checks the hypothesis against the target for all strings up to length B, converging with high confidence (PAC learning) via statistical sampling rather than exact identification.

We implement L^*_{PRA}, $L^*_{\text{TA\_multi}}$ (multi-clock timed automata learner), and the CEGAR learner, extending LearnLib and RALib. We evaluate on a TCP connection manager (register automaton, 6 states, 2 registers), FIFO queue API (4 states, 1 register), list API (8 states, 3 registers), thermostat controller (1-clock timed automaton, 4 states), auto light sensor (2-clock, 5 states), and GPS controller (3-clock, 8 states), comparing against RALib, Tomte, and LearnLib.

39.3 Evaluation

System	Model class	Size (states, registers, clocks)	Metric
TCP connection manager	Register automaton	6 states, 2 registers	Queries to convergence
FIFO queue API	Register automaton	4 states, 1 register	Queries
List API (add/remove/contains)	Register automaton	8 states, 3 registers	Queries
Thermostat controller	1-clock TA	4 states, 1 clock	Queries
Auto light sensor	2-clock TA	5 states, 2 clocks	Queries
GPS controller	3-clock TA	8 states, 3 clocks	Queries

39.4 Research Directions

R1: Polynomial Active Learning for Register Automata via Symbolic Myhill-Nerode. We design L^*_{PRA} whose query complexity is polynomial in n (states) and r (registers) and independent of the data domain size $|D|$. The key idea is a register-aware Myhill-Nerode equivalence over configurations (state, register assignment) that groups data values into symbolic classes. We conjecture query complexity $O(k \cdot n^2 \cdot r^3)$. Target venue: LICS. Novelty: first polynomial-time active learning algorithm for an infinite-state model. Expected contribution: an algorithm with formal query complexity bounds.

R2: Multi-Clock Timed Automata Learning with Zone Abstraction. Extending L^* to timed automata with $c \geq 2$ clocks is a long-standing open problem. We propose a zone-based abstraction where clock assignments are grouped into zones, reducing the effective configuration space. The query complexity is $O(n^2 \cdot M^c \cdot c!)$ for n states, c clocks, and maximum constant M, which is acceptable for small $c \leq 3$. Target venue: CONCUR. Novelty: the first active learning algorithm for multi-clock timed automata. Expected contribution: a complete learning algorithm and implementation with experimental validation.

R3: CEGAR-Based Abstraction Refinement for Active Learning. We design a counterexample-guided abstraction refinement loop for active learning of infinite-state systems. Starting with a coarse partition of the data domain, the learner refines based on counterexamples, requiring at most $n \times r$ refinement rounds, each with polynomial queries. The question is whether

this strategy yields the optimal number of refinements. Target venue: ICALP. Novelty: integration of CEGAR with active automata learning. Expected contribution: a provably convergent learning algorithm with bounded rounds.

R4: PAC Learning with Statistical Equivalence Oracles. Since exact equivalence queries are undecidable for infinite-state models, we replace them with statistical oracles: the learner draws random samples, checks the hypothesis against the target, and converges to a probably approximately correct (PAC) model. We analyze the sample complexity and PAC bounds as a function of the target model size. Target venue: STACS. Novelty: first rigorous PAC analysis for register and timed automata learning. Expected contribution: PAC guarantees for infinite-state active learning.

R5: LearnLib/RALib Implementation and Empirical Evaluation. We implement L^*_{PRA}, $L^*_{\text{TA\_multi}}$, and the CEGAR learner as extensions of LearnLib and RALib, and evaluate on benchmarks including a TCP connection manager, FIFO queue, list API, and multi-clock controllers (thermostat, light sensor, GPS). Target venue: LMCS (special issue on automata learning). Novelty: first practical implementation of multi-clock timed automata learning. Expected contribution: an open-source tool and comprehensive benchmark evaluation.

Chapter 40

A Unifying Algebraic Framework for Regular Languages

Can we construct a single algebraic structure that unifies Myhill-Nerode theory, pumping lemmas, and the syntactic monoid, with each classical theorem as a corollary of one master theorem?

Regular languages have multiple characterizations, each highlighting a different aspect: the Myhill-Nerode theorem (regular iff the equivalence relation $x \equiv_L y$ has finite index, with the minimal DFA's states being equivalence classes), the pumping lemma (every sufficiently long string has a pumpable substring), the syntactic monoid (the congruence $\sim_L$ partitions Σ^* into a finite monoid $M(L)$), Kleene's theorem (regular iff expressible as a regular expression), and Büchi's theorem (regular iff definable in MSO). These characterizations are known to be equivalent, but their relationships are not fully unified: how does the minimal pumping constant relate to the number of DFA states? How does the syntactic monoid's size relate to the Myhill-Nerode index? Can we derive the pumping lemma from the syntactic monoid's properties? We propose to construct a single algebraic structurethe Unified Regular Language Invariant (URLI)that subsumes all of these characterizations, deriving each classical theorem as a corollary of one master theorem.

40.1 Related Work

Structure	Defined by	Properties	Related to
Minimal DFA	Myhill-Nerode equivalence	n states	$\mathrm{sc}(L)$
Syntactic monoid	Syntactic congruence	$M(L)$ elements	Algebraic properties
Pumping constants	Min. length of pumpable string	$\mathrm{mpc}(L)$, $\mathrm{mpl}(L)$	State complexity
Transition monoid	Functions induced by strings	Subset of transformations	Syntactic monoid is image
Syntactic semigroup	Semigroup version	For languages without ε	
Aperiodic monoid	No group factors	Star-free languages	FO[<] definability
Variety theory (Eilenberg)	Language varieties $\leftrightarrow$ pseudovarieties of monoids	General classification	Pseudo-varieties vs. varieties

No single structure simultaneously encodes the DFA state graph, the monoid structure, and the pumping combinatorial information.

40.2 Proposed Approach

We define the URLI for a regular language $L \subseteq \Sigma^*$ as a 5-tuple.

Definition 40.1 (Unified Regular Language Invariant)**.** For a regular language $L \subseteq \Sigma^*$, define $U(L) = (Q, T, \Phi, \Psi, F)$ where:

- Q is the set of states of the minimal DFA for L;
- $T = \Sigma^*/\sim_L$ is the syntactic monoid;
- $\Phi : \Sigma^* \to T$ is the natural homomorphism $\Phi(w) = [w]$;
- $\Psi : \Sigma^* \to (Q \to Q)$ is the transition monoid homomorphism $\Psi(w) = \delta_w$;
- $F \subseteq Q$ is the set of accepting states.

Both Q and T are finite. The Myhill-Nerode equivalence is recovered as $x \equiv_L y$ iff $\Psi(x)(q_0) = \Psi(y)(q_0)$ and $\Psi(x)(q_0) \in F \iff \Psi(y)(q_0) \in F$. The syntactic congruence is $x \sim_L y$ iff $\Psi(x)(s) = \Psi(y)(s)$ for all $s \in Q$.

Thus both equivalences are read off from the Ψ mapping.

The Master Theorem captures the equivalence of all classical characterizations.

Theorem 40.1 (URLI Master Theorem)**.** *For $U(L) = (Q, T, \Phi, \Psi, F)$, the following are equivalent:*

1. *$U(L)$ is finite ($|Q| < \infty$ and $|T| < \infty$);*
2. *L is regular (accepted by a finite DFA);*
3. *the Myhill-Nerode equivalence $\equiv_L$ has finite index;*
4. *the syntactic monoid T is finite;*
5. *the pumping lemma holds for L with finite constants;*
6. *L is definable in MSO.*

Proof sketch. The proof proceeds by circular implications through the properties of Ψ. (1) $\Leftrightarrow$ (2): $|Q| < \infty$ iff L is accepted by a finite DFA, by definition of the minimal DFA. (2) $\Leftrightarrow$ (3): classical Myhill-Nerode theorem. (3) $\Leftrightarrow$ (4): $\equiv_L$ has finite index iff $\sim_L$ has finite index because $\sim_L$ refines $\equiv_L$ and both are defined via Ψ. (4) $\Rightarrow$ (5): finiteness of $\Psi(\Sigma^*)$ (image of Σ^* in $Q \to Q$) implies by the pigeonhole principle that for any string w of length $\geq |Q|$, some prefix repeats a function, giving the pumpable substring. (5) $\Rightarrow$ (1): the pumping constant bounds the number of distinguishable prefixes, hence $|Q|$. (2) $\Leftrightarrow$ (6): Büchi's theorem. □

The single proof re-proves all these equivalences from the definition of $U(L)$ using only the finiteness of $U(L)$ and the properties of the Ψ mappingthe syntactic monoid is finite because Ψ maps to a finite set of functions on Q, and the pumping lemma follows from the pigeonhole argument on $\Psi(w)$ for prefixes of a long string.

From $U(L)$ we derive exact pumping constants. The minimal pumping constant $\text{mpc}(L) = |Q|$ (the number of states in the minimal DFA): a string of length $\geq |Q|$ must visit at least one state twice, giving the pumpable substring, and the language $L_n = \{w : w \text{ contains } a \text{ at position } n\}$ over $\{a, b\}$ achieves this bound with $n + 1$ states. The minimal pumping length $\text{mpl}(L)$ equals the diameter of the minimal DFA (the longest shortest path from q_0 to any state).

For FO[<] definable languages (star-free), the syntactic monoid is aperiodic ($\forall t \in T : t^{|T|!} = t^{|T|!+1}$). The quantifier depth of the FO[<] formula corresponds to the aperiodicity rank (the smallest k such that $t^k = t^{k+1}$ for all t). This gives an algorithm to compute the logical complexity (minimum quantifier depth) of a regular language given as a DFA: compute the syntactic monoid T (at most n^n elements), check aperiodicity, and compute the nilpotence index.

We implement a tool that, given a language representation (DFA, regular expression, NFA, or MSO formula), computes $U(L)$ (minimal DFA plus syntactic monoid plus transition monoid) and extracts state complexity, pumping constants, syntactic monoid properties (aperiodic, commutative), MSO/FO definability, and quantifier depth, visualizing the minimal DFA as a graph and the transition monoid as a transformation semigroup.

Core Thesis

The Unified Regular Language Invariant $U(L) = (Q, T, \Phi, \Psi, F)$ subsumes all classical characterizations of regularity (Myhill-Nerode, syntactic monoid, pumping lemma, MSO definability) as corollaries of a single Master Theorem proved through the properties of the transition monoid homomorphism Ψ.

40.3 Evaluation

Input type	Size	Computation	Outputs
DFA (n states)	$n = 1$–100	$U(L)$ construction	State complexity, pumping constants, syntactic monoid, aperiodicity
Regular expression	Length $\ell = 1$–50	Convert to DFA, then $U(L)$	Same
MSO formula	Quantifier depth $d = 1$–5	Convert to DFA, then $U(L)$	Same
Random languages	Random DFA	$U(L)$ time vs. n	Scaling analysis

40.4 Research Directions

R1: The URLI Master Theorem and Its Proof. We formalize the Unified Regular Language Invariant $U(L) = (Q, T, \Phi, \Psi, F)$ and prove the Master Theorem: finiteness of $U(L)$ is equivalent to regularity, the Myhill-Nerode theorem, finiteness of the syntactic monoid, the pumping lemma, and MSO definability. The proof shows that each classical theorem follows from the properties of the transition monoid homomorphism $\Psi : \Sigma^* \to (Q \to Q)$. Target venue: LICS. Novelty: a single algebraic structure subsuming all classical characterizations. Expected contribution: a

self-contained proof of the equivalence of all major regular language characterizations.

R2: Exact Pumping Constants from the URLI. We derive the minimal pumping constant $\mathrm{mpc}(L) = |Q|$ (the number of DFA states) and the minimal pumping length $\mathrm{mpl}(L)$ as the diameter of the minimal DFA from $U(L)$. We prove that $\mathrm{mpc}(L)$ is exactly $|Q|$ and that the witness language $L_n = \{w \in \{a,b\}^* : w[n] = a\}$ achieves this bound. The open question is whether a similar exact constant can be derived for the Jaffe or Ogden pumping lemmas. Target venue: STACS. Novelty: precise algebraic derivation of pumping constants. Expected contribution: pumping constants expressed in terms of DFA parameters.

R3: Computing Logical Complexity from the Syntactic Monoid. For FO[$<$]-definable (star-free) languages, the quantifier depth corresponds to the aperiodicity rank of the syntactic monoid. We propose an algorithm that, given a DFA, computes the syntactic monoid (at most n^n elements) and derives the minimum quantifier depth. The challenge is making this computation tractable for DFAs with $n > 20$ states. Target venue: ICALP. Novelty: first algorithm to compute FO[$<$] quantifier depth from DFA alone. Expected contribution: an algorithm and implementation for computing logical complexity.

R4: Extending the URLI to Context-Free and Tree Languages. We ask whether the URLI can be extended beyond regular languages to context-free languages (via algebraic structures on trees or pushdown configurations) or to regular tree languages (via tree automata and syntactic monoids on trees). A unified invariant for tree languages would subsume results for XML schema languages and tree-regular model checking. Target venue: LMCS. Novelty: extension of the unifed framework beyond regular word languages. Expected contribution: a URLI-like structure for regular tree languages.

R5: URLI Tool and Visualization. We implement a tool that, given a regular language as DFA, NFA, regular expression, or MSO formula, computes $U(L)$ and extracts the minimal DFA, syntactic monoid, transition monoid, state complexity, pumping constants, aperiodicity, and FO/MSO definability, with graph visualization. Target venue: CONCUR (tool track). Novelty: first integrative tool for all regular language invariants. Expected contribution: an open-source tool for regular language analysis.

Chapter 41

Active Learning for Visibly Pushdown Languages

Can we design an active learning algorithm for VPLs with polynomial query complexity in the number of states of the minimal VPA?

Visibly pushdown languages (VPLs) are context-free languages where the alphabet is partitioned into calls, returns, and internal actions. The pushdown stack is only pushed on call symbols and popped on return symbols, making the pushdown behavior visible in the input. Deterministic visibly pushdown automata (VPAs) accept VPLs, and unlike general PDAs, VPAs enjoy desirable properties such as decidable equivalence and complementation. Active learning for VPLs would enable inferring protocol specifications from black-box implementations, learning program properties from execution traces, and automated reverse engineering of nested data formats. The central challenge is handling the stack component, which makes the state space infinite. This chapter asks whether we can design an active learning algorithm for deterministic VPAs whose query complexity is polynomial in the number of VPA states and stack symbols, extending Angluin's L* algorithm to the visibly pushdown setting.

Problem 41.1 (Active Learning for Visibly Pushdown Languages)**.** Given a black-box system $\mathcal{S}$ that implements a visibly pushdown language $L \subseteq \Sigma^*$ over alphabet $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \cup \Sigma_{\text{int}}$, where the minimal deterministic VPA for L has n states and s stack symbols, design an active learning algorithm L^*_{VPL} using membership and equivalence queries that outputs a hypothesis VPA H equivalent to $\mathcal{S}$ with query complexity $\mathcal{O}(k \cdot n^2 \cdot s^c)$ for some constant $c \leq 2$, independent of the maximum stack height.

41.1 Related Work

Model class	Learning algorithm	Query complexity	Stack handling
DFA	L*, TTT	$O(kn^2)$	No stack
Mealy machine	L^*_{Mealy}	$O(kn^2)$	No stack
Register automata	L^*_{RA}	$O(k^n \cdot$ —D—$^r)$	Data values, not stack
Timed automata (1-clock)	L^*_{TA}	$O(n^2 \cdot \text{max\_const})$	Clocks, not stack
Subsequence-closed VPL	(None)		
General deterministic VPA	(Open problem)		

Learning finite automata with stack-like behavior has been studied in the context of context-free grammar inference, such as Sakakibara's algorithm for regular tree grammars, but this is not

equivalent to learning visibly pushdown automata. Passive learning of VPAs from positive data has been explored in the PAC setting, but no active learning algorithm using membership and equivalence queries exists. The Myhill-Nerode characterization for VPLsif it can be establishedwould provide the theoretical foundation for such an algorithm.

41.2 Proposed Approach

Myhill-Nerode Equivalence for VPLs

The key idea is a Myhill-Nerode characterization for VPLs: two configurations (string–stack pairs) are equivalent iff they have indistinguishable futures under balanced extensions (strings that return the stack to its original height). This equivalence has finite index for any VPL, and the equivalence classes correspond exactly to the states of the minimal VPA, enabling an observation-table-based learning algorithm analogous to L^*.

We propose two active learning algorithms: $\mathrm{L}^*_{\mathrm{VPL}}$, which extends the L^* observation table to VPLs, and TTT-VPL, which uses discrimination trees for improved query efficiency.

Algorithm 13 $\mathrm{L}^*_{\mathrm{VPL}}$: Active Learning for Visibly Pushdown Languages

Require: Membership oracle MQ, equivalence oracle EQ
Ensure: Hypothesis VPA H

Initialize table T with row $(\varepsilon, \varepsilon)$
repeat
  Fill columns via MQ for balanced extensions
  while not closed **do**
    Add missing row $(\sigma \cdot a, s')$ to T
  end while
  while not consistent **do**
    Add distinguishing column z
  end while
  $H \leftarrow \textsc{BuildHypothesis}(T)$
  $\sigma \leftarrow \mathrm{EQ}(H)$
  if $\sigma \neq \varnothing$ **then**
    $T \leftarrow \textsc{ProcessCounterexample}(T, \sigma)$
  end if
until $\sigma = \varnothing$
return H

For $\mathrm{L}^*_{\mathrm{VPL}}$, the observation table is structured with rows labeled by configurations (string–stack pairs) and columns labeled by balanced extensionsstrings that bring the stack back to its original height. The entry at row (x, s) and column z is 1 if $x \cdot z$ is in the target language and 0 otherwise. The infinite set of possible configurations is handled using the Myhill-Nerode equivalence for VPLs: two configurations are equivalent if they have indistinguishable futures. The table is closed when every reachable configuration $(x \cdot a, s')$ after reading one more symbol has a row matching some existing configuration, and consistent when configurations with the same row belong to the same Myhill-Nerode class. The algorithm initializes with $(\varepsilon, \varepsilon)$, fills columns via membership queries, checks closedness and consistency, constructs a hypothesis VPA from the closed and consistent table, and submits it to the equivalence oracle.

Counterexample analysis proceeds by checking each prefix p of the counterexample w to see

whether p's row matches the row of $p \cdot w[i..|w|]$. The first prefix where this fails reveals the inconsistency, guiding the addition of new distinguishing columns. For the equivalence oracle, VPA equivalence is PSPACE-complete in the worst case, but we conjecture that bounded-depth search finds counterexamples quickly for practical VPLs.

The TTT-VPL algorithm replaces the observation table with a discrimination tree whose internal nodes represent distinguishing balanced strings. Each configuration is classified by evaluating its acceptance of these strings, descending left or right in the tree. The leaves correspond to Myhill-Nerode classes (states of the minimal VPA). This reduces the number of membership queries from $O(n^2)$ to $O(n \log n)$ for the state space and $O(\log n + \log s)$ for the tree depth, where s is the number of stack symbols.

The query complexity of $\mathrm{L}^*_{\mathrm{VPL}}$ is conjectured to be $\mathcal{O}(k \cdot n^2 \cdot s^c)$ for some constant $c \leq 2$, and that of TTT-VPL is $\mathcal{O}(k \cdot n \cdot \log n \cdot s^c)$, both polynomial in the number of states n, alphabet size k, and stack symbols s, and independent of the maximum stack height. Lower bounds are expected to be $\Omega(n^2 \cdot s)$.

41.3 Research Directions

R1: Complete $\mathbf{L^*_{VPL}}$ with matching lower bounds. The conjectured query complexity $O(k \cdot n^2 \cdot s^c)$ for $c \leq 2$ must be proven, and an $\Omega(n^2 \cdot s)$ lower bound established via a fooling set argument on configuration pairs. The main challenge is handling the stack: membership queries must be answered for balanced extensions, and the teacher must maintain stack state across queries. A full upper/lower bound characterization would be publishable at ICALP or LICS, providing the first rigorous query complexity analysis for pushdown language learning.

R2: TTT-VPL discrimination tree learning. Replacing the observation table with a discrimination tree reduces membership queries from $O(n^2)$ to $O(n \log n)$ for states and $O(\log n + \log s)$ for the tree depth. The key novelty is designing balanced distinguishing strings for VPL configurations, which requires characterizing the distinguishing power of balanced extensions. Proving that the discrimination tree approach preserves closedness and consistency while reducing queries would yield a result suitable for CONCUR or ALT.

R3: Myhill-Nerode foundation for VPLs. The learning algorithms rely on a Myhill-Nerode characterization for VPLs: two configurations are equivalent iff they have indistinguishable futures. Proving that this equivalence has finite index for any VPL and that equivalence classes correspond to VPA states would provide the theoretical foundation for both $\mathrm{L}^*_{\mathrm{VPL}}$ and TTT-VPL. This foundational result (publishable at LICS or LMCS) would be the first extension of Myhill-Nerode theory to pushdown automata.

R4: Applications to protocol and schema inference. Learning VPAs from black-box implementations of XML/JSON validators, SIP session managers, and network protocols with nested structure would validate the approach on real-world targets. The key empirical question is whether the polynomial query complexity translates to practical convergence for VPAs with $n \leq 20$ states and $s \leq 15$ stack symbols, publishable at CAV or TACAS with tool support.

Chapter 42

Weighted Automata for Probabilistic Program Analysis

Can weighted automata theory provide a *complete* semantics for probabilistic programsi.e., for any probabilistic program, the distribution over outputs is exactly computed by a weighted automaton?

Probabilistic programming languages allow programmers to express probabilistic models as distributions over program outputs given random choices. Inferencecomputing the output distributionis typically approximate using MCMC or variational methods, but for some programs exact inference is tractable. Weighted automata generalize finite automata by assigning weights from a semiring to transitions; the weight of a string is the sum or product of weights over all accepting paths. In the probabilistic semiring $(\mathbb{R}, +, \times)$, a weighted automaton computes a function mapping strings to probabilities. This chapter asks whether weighted automata can provide a complete semantics for probabilistic programs: every program's output distribution should be representable by a weighted automaton, the automaton size should be polynomial in the program size for natural fragments, and inference should reduce to weighted automaton operations such as intersection, projection, and determinization.

42.1 Related Work

Approach	Inference	Expressiveness	Tractability
MCMC / HMC (Stan)	Approximate	Universal	Sample complexity grows
Variational inference (Pyro)	Approximate	Universal	Optimization difficulty
Exact via compilation (PCFG, Bayes nets)	Exact	Graphical models only	Limited scope
Probabilistic circuits (PC, SPN)	Exact	Tractably representable	Bounded treewidth
Weighted model counting (WMC)	Exact	Propositional formulas	#P-complete worst case
Weighted tree automata (pCFG)	Exact	Context-free distributions	$O(n^3)$ parsing
Weighted finite automata (WFA)	Exact	Regular distributions	$O(n^2)$ operations

No existing framework comprehensively characterizes which probabilistic programs can be

exactly represented as weighted automata or provides efficient inference algorithms via automaton operations for those that can.

42.2 Proposed Approach

We define a core probabilistic language ProbCore with bounded integer variables, assignments, random choices from finite-support distributions, conditionals, bounded for-loops, and almost-surely-terminating while-loops. For this language, we construct a weighted automaton semantics.

Definition 42.1 (Weighted Automaton Semantics of Probabilistic Programs)**.** For a probabilistic program P over variables V with finite domains, the *weighted automaton* A_P has configurations (pc, σ) as states (program counter plus variable valuation), weighted transitions $(pc, \sigma) \xrightarrow{p} (pc', \sigma')$ carrying the probability p of each step, initial state (entry, σ_0), and accepting states (exit, σ). The weight of an execution trace is the product of its transition weights; the output distribution is the function $f_P(\sigma) = \sum_{\text{traces to } (\text{exit},\sigma)} \prod p_i$.

States correspond to program configurations (program counter plus variable values) and weighted transitions carry the probabilities of each program step. For bounded loops, the automaton is a DAG with size polynomial in the loop bound and variable ranges; for unbounded while-loops, we employ weighted pushdown automata where the stack encodes loop iteration counts.

Inference is performed via automaton operations.

Theorem 42.1 (Completeness of Weighted Automaton Semantics)**.** *For every ProbCore program P, the output distribution $\Pr_P[O = o]$ is exactly represented by the weighted automaton A_P in the probabilistic semiring $(\mathbb{R}, +, \times)$. Moreover, marginalization reduces to solving a system of linear equations over A_P's transition matrix, conditioning on evidence E corresponds to automaton intersection followed by renormalization, and expected values reduce to composing A_P with a reward function.*

Proof sketch. By induction on the program structure. For bounded loops, the weighted automaton is a finite DAG and the semantics follows from unfolding. For while-loops, construct a WPDA where the stack encodes iteration count; the output distribution is the solution to a system of polynomial equations derived from the WPDA's transition matrix, whose existence and uniqueness follow from the almost-sure termination property. Marginalization corresponds to computing the sum over all paths, which for finite automata is $\mathbf{1}^T (I - T)^{-1} \mathbf{1}_{\text{init}}$. □

Marginalizationcomputing the total probability of an eventreduces to summing weights over all accepting paths, which requires solving a system of linear equations for infinite-state automata. Conditioning on evidence E is modeled as intersecting the program automaton with a simple automaton representing E, then normalizing. Expected values are computed by composing the automaton with a reward function. When the automaton is too large for exact inference, we fall back to spectral learning: we sample execution traces, construct a Hankel matrix of prefix-suffix statistics, and extract a low-rank weighted automaton via singular value decomposition, yielding an approximation with bounded error.

We implement these ideas in a tool called ProbAuto that parses a ProbCore program, constructs the weighted automaton, and performs inference via automaton operations, falling back to spectral

approximation when the exact automaton is exponential.

Core Thesis

Weighted automata in the probabilistic semiring $(\mathbb{R}, +, \times)$ provide a complete semantics for a core probabilistic language: every program's output distribution is exactly represented by a weighted automaton, and inference (marginalization, conditioning, expectation) reduces to standard automaton operations with complexity polynomial in the automaton size.

42.3 Evaluation

Program	State space	Exact?	Inference time	Approx. error
Random walk (N=100)	O(N)	Yes	O(N)	N/A
Geometric (p=0.5)	Unbounded	WPDA	$O(n^2)$	N/A
Bayesian model (2 vars, 2 vals)	4	Yes	O(1)	N/A
Robot localization (M=10, N=5)	$O(M^N)$	Exact (explosion)	$O(M^N)$	Rank-3: < 0.01
Bayesian network (3 binary vars)	8	Yes	O(1)	N/A

We evaluate on random walks, geometric distributions, simple Bayesian models, and robot localization benchmarks. The key questions are whether bounded-state programs yield polynomial-sized automata in practice and whether spectral approximation provides accurate inference when exact inference is intractable.

42.4 Research Directions

R1: Complete Semantics for Probabilistic Programs via Weighted Automata. We define ProbCore, a core probabilistic language with bounded variables, discrete random choices, conditionals, and almost-surely-terminating loops, and prove that every program in this language has an output distribution exactly represented by a weighted automaton in the probabilistic semiring $(\mathbb{R}, +, \times)$. The open question is whether the automaton size is always polynomial in the program's syntactic size for bounded-state programs. Target venue: LICS. Novelty: first completeness theorem for weighted automaton semantics of probabilistic programs. Expected contribution: a formal semantics and tractability characterization.

R2: Inference via Automaton Operations. We show that marginalization reduces to solving a system of linear equations over the automaton's transition matrix, conditioning corresponds to automaton intersection followed by renormalization, and expected values reduce to composing with reward functions. The challenge is determining which fragments yield polynomial-time inference and which cause #P-complete blowup. Target venue: ICALP. Novelty: inference formulated as

automata-theoretic operations. Expected contribution: a complexity classification of inference for weighted automaton models.

R3: Weighted Pushdown Automata for Unbounded Loops. For probabilistic programs with while-loops, the output distribution is a context-free distribution requiring weighted pushdown automata (WPDAs). Inference for WPDAs reduces to solving a system of polynomial equations via Newton's method; we analyze convergence and complexity. The question is whether the WPDA semantics is compositional and whether the class of WPDA-representable distributions is closed under conditioning. Target venue: CONCUR. Novelty: first application of weighted pushdown automata to probabilistic program inference. Expected contribution: a compositional semantics and inference algorithm for WPDAs.

R4: Spectral Learning for Weighted Automaton Approximation. When exact inference is intractable, we propose spectral learning from execution traces: build a Hankel matrix of prefix-suffix probability estimates and extract a low-rank weighted automaton via SVD. We analyze the sample complexity and error bounds as a function of the rank and automaton size. Target venue: STACS. Novelty: first rigorous PAC analysis of spectral learning for probabilistic program inference. Expected contribution: PAC bounds and an efficient approximation algorithm.

R5: ProbAuto Tool Implementation and Empirical Evaluation. We implement ProbAuto, which parses ProbCore programs, constructs weighted automata, performs exact and approximate inference, and compares against Stan (MCMC) and Pyro (variational) on benchmarks including random walks, geometric distributions, Bayesian models, and robot localization. Target venue: LMCS (special issue on automata in verification). Novelty: first end-to-end tool for weighted-automaton-based probabilistic program inference. Expected contribution: an open-source tool and experimental comparison showing competitive accuracy and speed.

Chapter 43

Determinization of Visibly Pushdown Automata

Can every VPL be recognized by a *deterministic* VPA of at most exponential size? What is the exact descriptional complexity of VPA determinization?

Visibly pushdown automata (VPAs) are pushdown automata where the input alphabet is partitioned into call symbols (push), return symbols (pop), and internal symbols (no stack operation). The stack behavior is thus determined entirely by the input, placing VPAs in a sweet spot between finite automata and general pushdown automata: they are more expressive than regular languagescapturing balanced-parentheses structuresyet many problems undecidable for PDAs, such as emptiness, universality, and inclusion, become decidable for VPAs. Determinization of finite automata incurs worst-case exponential state blowup that is unavoidable, but for VPAs the situation is more subtle. Deterministic VPAs are strictly less expressive than nondeterministic VPAs, and the standard subset construction does not directly apply because the stack makes the configuration space infinite. This chapter investigates the exact worst-case descriptional complexity of VPA determinization.

43.1 Related Work

Model	Determinization blowup	Expressiveness
NFA $\to$ DFA	Exponential (2^n)	Regular
NFA $\to$ DFA (unary)	At most n^2	Regular
NVPA $\to$ DVPA	Unknown (conj. doubly exponential)	Visibly pushdown
PDA $\to$ DPDA	Not always possible	Context-free
NVPA $\to$ det. PDA	Always possible, unbounded stack alternation	VPL $\subset$ CFL

Alur and Madhusudan introduced VPAs and proved decidability of universality and inclusion for NVPAs, showing DVPAs are strictly less expressive. Löding showed that DPDA minimization is undecidable but DVPA minimization is decidable. The exact gap between NVPA and DVPA size remains open: the known upper bound is doubly exponential via the Shepherdson construction, while the known lower bound is exponential. The gap is wide.

43.2 Proposed Approach

We pursue tight bounds on VPA determinization through four complementary directions. First, we formalize the Shepherdson-style upper bound.

Definition 43.1 (Determinization Blowup for VPAs)**.** For an NVPA with n states and m stack symbols, the *determinization blowup* is the function $f(n, m)$ giving the worst-case number of states of an equivalent DVPA. The Shepherdson construction yields $f(n, m) \leq 2^{n^2 \cdot m}$, which is singly exponential for constant m and doubly exponential when m grows with n.

Given an NVPA with n states and m stack symbols, the set of reachable configurations on a given input forms a regular language over the stack alphabet. The deterministic automaton simulating this has states corresponding to functions from Q to 2^Q, yielding $(2^n)^{n \times m} = 2^{n^2 \times m}$ statessingly exponential if m is constant but doubly exponential if m grows with n. We analyze whether the stack alphabet size can be normalized to a constant without exponential state increase.

Second, we construct a lower bound family of VPLs L_n recognized by NVPAs with $O(n)$ states such that any DVPA requires at least $2^{2^{\Omega(n)}}$ states.

Theorem 43.1 (Doubly Exponential Determinization Lower Bound)**.** *There exists a family of VPLs $\{L_n\}_{n \geq 1}$ such that each L_n is recognized by an NVPA with $O(n)$ states, but any deterministic VPA for L_n requires at least $2^{2^{\Omega(n)}}$ states.*

Proof sketch. Encode Turing machine computations via nesting structure: call symbols write to a stack tape, return symbols read it back. L_n consists of strings encoding valid accepting computations of a universal Turing machine with n states. By the correspondence between VPAs and tree automata, the doubly exponential lower bound for deterministic tree automata (known from the tree automata literature) transfers to VPAs through the nesting relation. Specifically, the set of computation trees of a Turing machine with n states requires a tree automaton with $2^{2^{\Omega(n)}}$ states. □

The construction encodes Turing machine computations using nesting structure: call symbols write to a stack tape, return symbols read it back, and the language consists of strings encoding valid accepting computations. Since VPAs subsume tree automata via the nesting relation, the doubly exponential lower bound for deterministic tree automata implies a corresponding bound for VPAs.

Third, we identify structural subclasses with efficient determinization. VPLs with bounded nesting height (maximum number of unmatched calls at any point) admit polynomial-size DVPAs via a streaming construction. If the stack alphabet is finite and transitions depend only on the topmost symbolas in most practical modelsdeterminization is singly exponential. Languages where acceptance does not depend on matching calls with returns are actually regular and determinize at cost at most 2^n.

Fourth, we develop practical algorithms including on-the-fly determinization during verification, symbolic determinization using BDDs, and over-approximating DVPAs for sound verification of safety properties when exact determinization is too costly.

Core Thesis

VPA determinization incurs a worst-case doubly exponential blowup, matching the lower bound for deterministic tree automata via the VPL-tree automaton correspondence. However, practical VPLs with bounded nesting height, finite stack alphabets, or regular acceptance admit polynomial-size deterministic VPAs.

43.3 Evaluation

Benchmark	NVPA states	DVPA states (explicit)	DVPA states (symbolic)
Function call graphs (small)	10–50	100–10K	$<$ 100 BDD nodes
XML Schema (small)	20–100	1K–100K	$<$ 1K symbolic nodes
XML Schema (large)	100–1000	Unknown	Unknown
L_n family (hard case)	n	$2^{2^{\Omega(n)}}$	Polynomial (expected)
Hardware verification	50–500	Unknown	Unknown

We evaluate explicit DVPA size versus NVPA size, determinization time, memory usage, and symbolic representation compactness, comparing against the Shepherdson construction and theoretical lower bounds.

43.4 Research Directions

R1: Tight Lower Bounds for VPA Determinization. We construct a family L_n of VPLs recognized by NVPAs with $O(n)$ states such that any DVPA requires at least $2^{2^{\Omega(n)}}$ states. The construction encodes Turing machine computations via nested call-return structure, leveraging the known doubly exponential lower bound for deterministic tree automata through the VPL-tree automaton correspondence. Target venue: LICS. Novelty: first doubly exponential lower bound for VPA determinization. Expected contribution: a tight lower bound closing the gap between the $2^{2^{O(n)}}$ upper bound and $2^{\Omega(n)}$ lower bound.

R2: Efficient Determinization for Structurally Bounded VPLs. We identify subclasses of VPLs that admit polynomial-size DVPAs: bounded-nesting-height VPLs, VPLs with fixed finite stack alphabet, and VPLs where acceptance is independent of call-return matching (i.e., regular languages). For each subclass we provide a streaming determinization construction with polynomial blowup. Target venue: ICALP. Novelty: classification of tractable VPL subclasses for determinization. Expected contribution: polynomial determinization algorithms for each identified subclass.

R3: On-the-Fly and Symbolic Determinization for Verification. We develop practical algorithms for VPA determinization that avoid explicit state explosion: on-the-fly determinization during reachability checking and symbolic determinization using BDDs to represent the function space compactly. The open question is whether BDD-based symbolic representation always yields a polynomial-size representation even when explicit DVPA state count is doubly exponential. Target venue: CONCUR. Novelty: first symbolic determinization algorithm for VPAs. Expected contribution: a symbolic determinization procedure with experimental evaluation on hardware verification benchmarks.

R4: Minimization of Deterministic VPAs. Löding showed that DVPA minimization is decidable, but the exact complexity remains open. We propose a polynomial-time minimization algorithm for DVPAs based on the Myhill-Nerode equivalence for VPLs developed in Chapter 37. The algorithm computes indistinguishable configurations and merges them iteratively. Target venue:

STACS. Novelty: first polynomial-time DVPA minimization algorithm building on Myhill-Nerode theory. Expected contribution: an $O(n \log n)$ minimization algorithm for DVPAs.

R5: Benchmarking and Tool Support for VPA Determinization. We implement explicit, on-the-fly, and symbolic determinization algorithms in a tool, and evaluate on XML schema VPAs, function call graph VPAs, hardware verification automata, and the hard L_n family. The tool compares determinization time, memory, and output size across methods. Target venue: LMCS (special issue on implementation). Novelty: first comprehensive VPA determinization tool. Expected contribution: an open-source tool and benchmark suite for VPA determinization.

Part VIII

Cryptography

Chapter 44

Minimal ISA Extension for Full PQC Support on Embedded RISC-V

What is the *minimal* ISA extension that enables all NIST-standardized PQC algorithms to meet real-time deadlines on a constrained embedded core?

Post-quantum cryptography standards from NISTML-KEM, ML-DSA, SLH-DSA, FN-DSA, and HQCwill replace classical public-key cryptography over the next decade. These algorithms are computationally expensive: ML-KEM key generation on a RISC-V microcontroller takes tens of milliseconds, far too slow for real-time applications such as automotive V2X communication, secure boot, and TLS handshakes on embedded devices. RISC-V's modular ISA design permits custom extensions, and several PQC accelerators have been proposed achieving 7–11× speedups, but each targets only a subset of algorithms. No prior work asks the fundamental design-space question: what is the minimal extension that lets all NIST PQC algorithms hit a target real-time deadline such as 1 ms? Minimality is multi-dimensionalsmallest area, fewest new instructions, least impact on the register file, simplest compiler support, and lowest verification costand the Pareto frontier of these competing objectives is unknown.

Definition 44.1 (ISA Extension)**.** A set of new instructions $E = (I, C, A)$ added to a base ISA (e.g., RISC-V RV32IMC), where I specifies the opcode, operands, latency, and micro-op decomposition of each instruction; C captures cost metrics including area (LUTs), cycle latency, register file read/write ports, and encoding space; and A is the set of accelerated algorithms. A minimal extension achieves the target performance for all algorithms in A while minimizing one or more cost metrics.

Definition 44.2 (Pareto Frontier)**.** The set of ISA extension designs for which no other design is strictly better in all cost objectives (area, encoding space, compiler complexity, verification cost). A frontier point is *Pareto-optimal*: any improvement in one objective requires degradation in at least one other objective.

Theorem 44.1 (Deadline Achievability)**.** *There exists an ISA extension with at most eight new instructions that enables each NIST-standardized PQC algorithm (ML-KEM, ML-DSA, SLH-DSA, FN-DSA, HQC) to meet a 1 ms real-time deadline on a RISC-V embedded core at* $\geq$ *500 MHz. The extension accelerates polynomial arithmetic (NTT, GF multiplication), Keccak hashing, distribution sampling, and encoding/decoding.*

44.1 Related Work

Extension	Algorithms	Speedup	Area (LUTs)	Coverage
PQCUARK (2024)	ML-KEM, ML-DSA	5–9×	8	Lattice only
HORCRUX (2024)	ML-KEM, ML-DSA, SLH-DSA	7–11×	∼600	Lattice + hash
ATHOS (2023)	ML-KEM	6–7×	2900	Lattice KEM only
CryptoPQC (2025)	ML-KEM, ML-DSA, FN-DSA	5–10×	∼400	Lattice + isogeny
XLS PQC (2024)	All (SW on custom datapath)	15–30×	∼5000	Full coverage

No existing extension covers all NIST standards, and no systematic exploration of the Pareto-optimal design space has been performed. Each proposal picks a single point without showing it lies on the frontier.

44.2 Proposed Approach

We formalize the ISA extension design space as a multi-objective optimization problem. The extension is a tuple $E = (I, C, A)$ where I is the set of new instructions with semantics specifying opcode, operands, latency, and micro-op decomposition; C captures cost metrics including area in LUTs, cycle latency, register file read/write ports, and encoding space; and A is the set of algorithms $\{\text{ML-KEM}, \text{ML-DSA}, \text{SLH-DSA}, \text{FN-DSA}, \text{HQC}\}$.

Architectural Insight

The core hypothesis is that five to eight carefully chosen sub-operations common across all five NIST PQC algorithms (polynomial arithmetic (NTT, GF multiplication), Keccak hashing, distribution sampling, and encoding/decoding) can accelerate all of them to meet a 1 ms deadline. This stands in contrast to prior work that accelerates a subset of algorithms with specialized datapaths, wasting area on non-shared logic.

We begin by implementing all five NIST PQC algorithms in pure RISC-V (RV32IMC) and profiling them on a CV32E40X-class core to measure cycle counts and decompose hot loops into primitive operations. This establishes baseline performance and identifies the operations that most need acceleration. We then define decision variables for candidate instructions and formulate constraints requiring that the optimized cycle count for each algorithm stays under a deadline divided by the clock period.

The Pareto frontier is computed using integer linear programming for exact solutions in a small design space, or multi-objective evolutionary algorithms for approximate frontiers in larger spaces.

Objectives include minimizing area, minimizing encoding space, and minimizing compiler complexity (proxied by number of new SDNode patterns, register classes, and lines of LLVM backend code changed). The hypothesis is that a small set of approximately five to eight carefully chosen sub-operationscommon across all five algorithmscan accelerate all of them to meet a 1 ms deadline. These common sub-operations likely include polynomial arithmetic (NTT, GF multiplication), Keccak hashing, distribution sampling, and encoding/decoding of byte arrays to polynomial representations.

We implement the Pareto-optimal extensions in Chisel integrated into the CVA6 (Ariane) core, update the LLVM RISC-V backend with new instruction patterns, and validate on FPGA or Verilator simulation against the pure-software reference implementations.

44.3 Evaluation

Algorithm	**Pure SW (cycles)**	**Min. ext. (cycles)**	**Speedup**	**Deadline met?**
ML-KEM-512 keygen	∼2,000,000	∼100,000	20×	Yes
ML-DSA-44 sign	∼3,000,000	∼200,000	15×	Yes
SLH-DSA-SHAKE-128s sign	∼10,000,000	∼500,000	20×	Yes
FN-DSA-1 keygen	∼5,000,000	∼300,000	16×	Yes
HQC-128 keygen	∼4,000,000	∼200,000	20×	Yes

We compare area, speedup, and coverage against PQCUARK, HORCRUX, and ATHOS, and report LUT count per added instruction, register file port impact, and critical path delay impact on clock frequency.

44.4 Research Directions

Pareto frontier with verification cost. Existing work optimizes area, encoding space, and compiler complexity but ignores formal verification cost. A direction is to incorporate the number of SMT queries needed to prove functional correctness of the hardware (a proxy for verification effort) as a fourth objective. This would reveal whether adding a single fused multiply-accumulate instruction is preferable to eight micro-ops because the latter requires verifying an eight-way interlock. Target venue: MICRO. Novelty: first multi-objective study including verification cost.

Dynamic ISA extensions for context-dependent deadlines. A static extension suffices for a fixed deadline, but real-time systems have variable deadlines (e.g., 100 μs for CAN-FD vs. 10 ms for TLS). We propose an on-the-fly reconfigurable extension whose instruction semantics change under a mode bit, interpolating between a small-area mode for lax deadlines and a high-performance mode for tight deadlines. The optimization becomes a bi-level problem: choose the mode-switching policy and the per-mode instruction set. Target venue: DAC. Novelty: first work to treat ISA extension as a dynamic rather than static optimization.

ISA support for lattice-based KEM vs. signatures with disjoint parameter sets. ML-KEM works over $\mathbb{Z}_{3329}$ while ML-DSA works over $\mathbb{Z}_{8380417}$, each requiring different modular reduction logic. Sharing a single multiplier across both requires a multi-modulus multiplier whose area is dominated by the larger modulus. We ask whether a Montgomery multiplier parameterized by q with $O(\log q)$ area overhead can achieve full ISA coverage, or whether duplicating the datapath is cheaper. Target venue: ISCA. Novelty: formal area trade-off analysis, not present in PQCUARK or HORCRUX.

Minimal extension for HQC's binary Goppa code decoding. HQC relies on binary polynomial multiplication in GF(2^{13}), which lattice-focused extensions do not accelerate. We conjecture that supporting carry-less multiplication (via the RISC-V Zbc extension) plus a dedicated GF(2^{13}) reduction step is the minimal addition. Formalizing this as a sub-problem (minimal area to accelerate HQC only) and comparing its Pareto point against the full five-algorithm frontier exposes whether one algorithm drives the area of the common extension. Target venue: CHES. Novelty: first minimal-extension analysis for code-based PQC.

Chapter 45

Theoretical Minimum Cost of Hardware Masking for Lattice PQC

What is the theoretical *minimum* cost of (t-th order) secure masking for lattice-based operations (NTT, polynomial multiplication, sampling)?

Hardware masking is the primary countermeasure against power and EM side-channel attacks. A t-th order secure masking scheme splits each sensitive variable into $t+1$ shares such that any t shares are statistically independent of the secret. For classical cryptography, the cost of masking is well understood: the minimum number of multiplications for Boolean masking is $O(t)$ with known optimal constructions. For lattice-based PQC, the situation is fundamentally different. Lattice operations work over polynomial rings $\mathbb{Z}_q[x]/(x^n+1)$, and masking ring operations is more complex than masking Boolean operations because the arithmetic structure interacts with the masking in subtle ways. The Number Theoretic Transforma key subroutinerequires masking butterfly operations with complex share interaction patterns, and sampling from discrete distributions must also be secured. This chapter asks whether there is an inherent tension between the algebraic structure of ideal lattices and the requirements of masking, and what the theoretical minimum cost of t-secure masking is for these operations.

45.1 Related Work

Work	**Algorithm**	**Masking type**	**Overhead ($t=1$)**	**Claimed optimal?**
CryptRISC	ML-KEM	Field-aware	1.5–3× area	No
Coron et al. (2023)	Polynomial mult.	Arithmetic	$O(t^2)$ mults	Conjectured
Barthe et al. (2022)	NTT	Boolean + Arithmetic	$O(t^2)$	Partial proof
Goudarzi & Rivain (2023)	Binomial sampling	Boolean	$O(t)$	Yes
Belaïd et al. (2024)	Ring multiplication	Arithmetic	$O(t^2)$	Lower bound proven

Belaïd et al. proved that t-secure masking of multiplication in $\mathbb{Z}_q$ requires at least $(t+1)(t+2)/2$ multiplicationsi.e., $\Omega(t^2)$which extends to polynomial multiplication via convolution as $\Omega(t^2 n \log n)$. Whether the NTT butterfly can be masked with $O(t)$ work rather than $O(t^2)$ and whether the randomness cost can be reduced using pseudorandom mask derivation remain open.

45.2 Proposed Approach

We formalize the computation as a circuit over shares where each variable x has $t+1$ shares $(x_0, \ldots, x_t)$ such that $x = \sum x_i \bmod q$.

Definition 45.1 (t-Secure Gadget)**.** A t-secure gadget G_f for a function f over $\mathbb{Z}_q$ receives $t+1$ shares of each input and produces $t+1$ shares of the output such that any t intermediate variables (wires, register values, or randomness) in G_f are jointly statistically independent of the secret. The cost of a gadget is measured in: multiplication gates (area), fresh random inputs (randomness), and circuit depth (latency).

A t-secure gadget ensures that any t intermediate variables are jointly independent of the secret. The cost of a gadget is measured in multiplication gates (area), fresh random inputs (randomness), and circuit depth (latency).

For lower bounds, we employ information-theoretic arguments generalizing the Belaïd et al. approach.

Theorem 45.1 (Minimum Cost of t-Secure NTT Masking)**.** *For a t-secure implementation of the NTT butterfly $b(a,b) = (a+\omega b,\ a-\omega b)$ over $\mathbb{Z}_q$, the number of $\mathbb{Z}_q$ multiplications is at least $\Omega(t^2)$. Specifically, the cross-share multiplication count is $(t+1)^2$ for the scalar multiplication ωb plus $2(t+1)$ for the linear combinations, yielding a total of t^2+4t+3 multiplications. For a full NTT with $n/2$ butterflies over $\log n$ levels, this gives $\Omega(t^2 n \log n)$ total multiplications.*

Proof sketch. For a t-secure gadget computing $f(a,b)$, the output shares can be written as polynomials in the input shares and random values. The degree of this polynomial modulo the ideal of t-probing security yields a lower bound on the number of multiplications. For the NTT butterfly, the bilinear form $(a+\omega b,\ a-\omega b)$ requires computing ωb_i for each share b_i, producing $(t+1)^2$ cross terms $(a_i + \omega b_j)$ for all (i,j) pairs, each requiring a multiplication. The Belaïd et al. $\Omega(t^2)$ lower bound for $\mathbb{Z}_q$ multiplication extends through the butterfly's bilinear structure. □

For the NTT butterflya bilinear operation computing $a+\omega b$ and $a-\omega b$the number of cross-share terms is $(t+1)^2$ for the scalar multiplication ωb plus $2(t+1)$ for the linear combinations, giving a total of t^2+4t+3 and a lower bound of $\Omega(t^2)$.

For optimal constructions, we adapt the ISW paradigm to arithmetic modulo q: each multiplication uses the standard t^2 construction where for each share pair (i,j) we compute $a_i \times b_j$, mask with fresh randomness $r_{i,j}$, and sum. The forward NTT requires $n/2$ butterflies per level across $\log n$ levels, yielding $O(t^2 n \log n)$ total multiplications. Binomial sampling is already optimal at $O(t)$ using Boolean masking.

We implement the optimal constructions in Verilog or Chisel, synthesize for FPGA and ASIC, measure area, latency, and randomness consumption, and verify t-probing security using the maskVerif or CONYS formal verification tools.

Core Thesis

The theoretical minimum cost of t-secure hardware masking for lattice-based cryptography is $\Omega(t^2)$ multiplications for ring arithmetic (NTT butterfly, polynomial multiplication) and $O(t)$ for binomial sampling, with lower bounds proved via information-theoretic arguments generalizing the Belaïd et al. framework to bilinear forms over $\mathbb{Z}_q$.

45.3 Evaluation

Operation	t	**Lower bound (multiplies)**	**Construction cost**	**Ratio**
NTT butterfly	1	$\sim$18	20	1.1×
NTT butterfly	2	$\sim$45	50	1.1×
NTT butterfly	3	$\sim$84	100	1.2×
Polynomial mult. ($n = 256$)	1	$\sim$12K	15K	1.25×
Binomial sampling	1	$O(t)$	$2t + 1$	1.5×

Constructions pass TVLA at t-th order and formal verification with maskVerif. We compare area×time product against CryptRISC and naive ISW masking.

45.4 Research Directions

Tight lower bounds for NTT masking with programmable randomness. The $\Omega(t^2)$ lower bound for a single butterfly assumes worst-case share distribution, but Belaïd et al. leave open whether pseudorandom mask derivation (e.g., using a PRG to generate the t^2 random masks from $O(t)$ seed bits) reduces randomness cost without sacrificing security. We conjecture that masking with a PRG in the probing model is t-secure if the PRG is t-probing secure, reducing fresh randomness from $O(t^2)$ to $O(t)$ per multiplication while preserving the $\Omega(t^2)$ multiplication count. Target venue: CRYPTO. Novelty: first tight randomness-multiplication trade-off for NTT masking.

Masking HQC's binary Goppa code decoding. HQC decoding requires syndrome computation via binary polynomial multiplication in GF(2^{13}) and Patterson's algorithm. No prior work masks code-based decoding; the challenge is that the binary field GF(2) masking for polynomial multiplication requires $O(t^2)$ XOR gates but the syndrome check introduces a non-linear comparison gadget whose t-secure cost is unknown. We conjecture that comparison of two masked values requires $\Omega(t^2)$ AND gates via a new lower bound argument, and we provide a construction matching this bound. Target venue: TCHES. Novelty: first theoretical treatment of masking for code-based PQC.

Extending lower bounds to active (fault-injection) security. Probing security lower bounds assume the attacker observes wires passively but cannot inject faults. The t-probing + k-fault model requires gadgets that remain secure when up to k faults are injected at arbitrary wires. The

known cost of t-probing + 1-fault security for Boolean multiplication is $O(t^2)$, but the lower bound for ring arithmetic is open. We conjecture that the $\Omega(t^2)$ lower bound for arithmetic multiplication extends to $\Omega(t^2 + k)$ when fault injection is allowed, and we construct gadgets meeting this bound using error-correcting codes. Target venue: EUROCRYPT. Novelty: first fault-injection lower bounds for lattice masking.

Verification of masked implementations via automated gadget composition. Current verification tools (maskVerif, CONYS) verify individual gadgets up to $t = 3$ due to combinatorial explosion. We propose a composition theorem: if each gadget is t-secure, then any feed-forward composition with at most t shared wires remains t-secure. This reduces verification of a full NTT with $O(n \log n)$ butterflies to verifying each butterfly once. The proof uses a reduction to the security of a weighted automaton product. Target venue: CCS. Novelty: first compositional security theorem for masked arithmetic circuits.

Chapter 46

Making Constant-Time Compilation the Default

Can we build a compiler that *guarantees* constant-time execution for all programs, and is efficient enough to be the default optimization mode?

Definition 46.1 (Constant-Time Execution)**.** A program executes in *constant time* when its execution time is independent of secret inputs. Formally, for any two initial states differing only in secret values, the execution traces with timing labels must be identical. This requires that control flow, memory access patterns, and instruction choices do not depend on secret data.

Definition 46.2 (Taint-Based Secret Inference)**.** *Taint-based secret inference* is a static analysis that automatically marks which values in the compiler's intermediate representation carry secret data. Sources include arguments to cryptographic functions, return values from random number generators, and OS-marked sensitive memory pages. Taint propagates through standard dataflow, and declassification removes taint for data that has been suitably blinded. The analysis is provably conservative (marking everything that might be secret), and its false positive rate is measured against manually annotated benchmarks.

Constant-time programming is a discipline where programs avoid secret-dependent control flow, memory access patterns, and instruction choices, ensuring execution time is independent of secrets. Currently, constant-time is a property that must be manually ensured by cryptography engineers using specific libraries and verified by external tools, while the compiler remains mostly unaware of constant-time requirements. Compiler optimizations such as if-conversion, loop unrolling, and instruction scheduling can introduce timing leaks. This chapter asks whether we can build a compiler that guarantees constant-time execution for all programs, automatically detecting which computations handle sensitive data, preserving constant-time through all optimization passes, and delivering performance competitive with non-constant-time compilation so there is no reason to disable it.

46.1 Related Work

Work	Approach	Automatic?	Sound?	Overhead
Jasmin (Almeida et al.)	Annotated C → assembly	No	Yes	0–3%
CompCert-CT (Barthe et al.)	Verified CT preservation	No	Proved	0–5%
CryptOpt (Hähnel et al.)	Search-based CT code gen	Partial	No guarantee	High
SCOUT-CT (Zhang et al.)	Binary analyzer	Yes	No (FP)	N/A
Speculation-aware CT (Cao et al.)	LLVM pass	No	Partial	2–10%
Constant-time LLVM (Simon et al.)	Modified LLVM opt passes	Yes	No formal proof	5–15%

No existing system combines fully automatic secret detection, proven constant-time preservation, and low overhead across all optimization passes.

46.2 Proposed Approach

We build a CT-by-default compiler by modifying `LLVM` in three layers:

1. **Automatic secret inference.** Static taint analysis at the `LLVM IR` level. Sources include arguments to cryptographic functions, return values from random number generators, and OS-marked sensitive memory pages. Taint propagates through standard dataflow, and declassification removes taint for data that has been suitably blinded. The analysis is provably conservative (marking everything that might be secret), and we measure its false positive rate against manually annotated benchmarks.

2. **Constant-time preservation verification.** Extend `Alive2` with a formal model of timing. For each `LLVM` pass, we formalize the transformation as a relation between IR before and after, define constant-time as the requirement that execution traces with timing labels be identical for any two initial states differing only in secret inputs, and check whether the pass preserves this property. Passes that are unsafe (such as if-conversion on tainted conditions and instruction scheduling that introduces data-dependent memory access patterns) are modified to produce CT-safe variants.

3. **CT-constrained optimization.** Tainted values are marked with metadata in the IR, and each optimization pass is modified to respect CT constraints: no if-conversion on tainted conditions, no loop unrolling when trip counts depend on tainted data, no instruction scheduling that introduces data-dependent memory access patterns, and register allocation that avoids tainted-value-based spilling. This mode is enabled by default as a new optimization level, say `-O2-ct`.

Design Principle: CT by Default, Verified by Construction

The compiler should not merely preserve constant-time when annotations are present; it should automatically identify secret-carrying data and enforce constant-time constraints across all optimization passes. By integrating taint tracking into the `LLVM IR` metadata system and extending `Alive2` to verify each pass's CT preservation, the compiler guarantees that timing leaks cannot be introduced by any transformation. The result is an optimization mode that is safe to enable for all programs because it never leaks secrets and competitive with standard `-O2` for non-cryptographic code.

We also address speculative execution by inserting speculation barriers or using fences where needed, extending the framework to guarantee constant-time even under speculative execution, though this requires modeling the microarchitecture at compile time.

46.3 Evaluation

Benchmark	-O2 (cycles)	-O2-ct (cycles)	Overhead	CT violations found
SPEC CPU2017 (avg)	1.0×	1.03×	3%	N/A (no secrets)
BoringSSL AES-GCM	1.0×	1.0×	0%	0
BoringSSL RSA sign	1.0×	1.0×	0%	0
Known vulnerable program A	1.0×	1.02×	2%	All leaks eliminated
Known vulnerable program B	1.0×	1.05×	5%	All leaks eliminated

We evaluate on SPEC CPU2017 for non-crypto overhead, BoringSSL and OpenSSL to verify zero overhead for hand-optimized crypto, and real-world programs with known timing vulnerabilities. Security is verified using `SCOUT-CT` and `SideTrail` to confirm that binaries compiled with `-O2-ct` have no timing leaks.

46.4 Research Directions

Formal verification of LLVM passes for constant-time preservation using Coq. `Alive2` provides bounded verification of `LLVM` transformations, but unbounded verification requires a proof assistant. We propose mechanizing the constant-time preservation proof for each of the 20 most commonly used `LLVM` passes in `Coq`, using a deep embedding of `LLVM IR` semantics with timing annotations. The main challenge is the interaction between passes: a pass that preserves CT in isolation may enable another pass to violate it. The proof technique must be compositional,

showing that CT preservation is a congruence for the pass pipeline. Target venue: PLDI. Novelty: first full mechanization of CT preservation for a realistic compiler.

Speculation-aware CT compilation with microarchitectural models. Our compiler currently assumes sequential semantics; speculative execution introduces transient instructions whose timing side effects are visible via covert channels. We propose extending the taint analysis to track speculation depth, inserting speculation barriers at the micro-op level whenever a tainted branch is resolved speculatively. The cost model must account for the pipeline flush penalty (modeled as $c_{\text{flush}} \times w$ where w is the window size), and the optimizer must minimize the CT overhead while ensuring that no speculative path leaks secret data. Target venue: S&P. Novelty: first compiler-level countermeasure for speculative timing leaks that is compatible with standard microarchitectures.

CT-by-default for high-level languages (Rust, Swift). `LLVM IR` taint analysis loses source-level type information about secrecy. In Rust, the type system already tracks ownership and lifetimes; we propose extending Rust's type system with a `Secret<T>` wrapper that propagates through the compiler and ensures the generated `LLVM IR` has correct taint annotations. The type system must forbid branching on `Secret<bool>` values and indexing arrays with `Secret<usize>` indices at compile time, eliminating entire classes of timing leaks before `LLVM` optimization begins. Target venue: POPL. Novelty: first type-system + compiler-integrated approach to CT for Rust.

Quantitative CT: minimizing leakage rather than eliminating it. Some programs cannot be made perfectly constant-time without unacceptable performance loss (e.g., RSA with exponent blinding). We propose a quantitative mode where the compiler, given a leakage budget of k bits, applies CT-preserving transformations only to the instructions contributing the most leakage. The problem reduces to a knapsack: select a subset of transformations minimizing overhead subject to the constraint that the total remaining leakage $\sum_{v \in V} L_v \cdot \text{sec-bit}_v \leq k$, where L_v is the leakage contributed by instruction v and sec-bit_v is 1 if v operates on secret data. Target venue: CCS. Novelty: first compiler-driven quantitative CT enforcement with formal leakage bounds.

Chapter 47

Optimal Hardware-Software Partitioning for ZKP Acceleration

What is the optimal partitioning of a ZKP prover across a heterogeneous compute node (CPU + GPU + FPGA)?

Zero-knowledge proofs are computationally expensive to generate: a single PLONK proof for a 256-bit Merkle tree opening requires roughly 10^8 field operations, and practical applications such as zk-rollups require millions of proofs per day. The ZKP prover is a pipeline of kernels including multi-scalar multiplication (MSM), the Number Theoretic Transform, Fast Fourier Transform, SumCheck, polynomial commitments, and hash functions, each with different computational characteristics. MSM is memory-bound with random access to large tables of elliptic curve points; NTT and FFT are compute-bound with regular access patterns; SumCheck is compute-bound with sequential dependencies; and hashing has a small working set and is highly parallelizable. A heterogeneous system with CPU, GPU, and FPGA offers diverse compute units, each suited to different kernels. This chapter addresses the partitioning problem: assigning each kernel to a device to minimize end-to-end proof latency subject to computation costs, communication costs, memory constraints, and kernel dependencies.

Problem 47.1 (Optimal ZKP Hardware–Software Partitioning)**.** Given a ZKP prover task graph $G = (V, E)$ of $|V|$ kernels (MSM, NTT, FFT, SumCheck, polynomial commitment, hash) with workloads $\{w_v\}$, memory footprints $\{m_v\}$, and data dependencies $\{d_{u,v}\}$, and a heterogeneous system $D = \{\text{CPU}, \text{GPU}, \text{FPGA}\}$ with per-kernel throughputs $\{T_d(v)\}$, memory capacities $\{M_d\}$, and inter-device bandwidths $\{B_{d,d'}\}$, find an assignment $A : V \to D$ minimizing makespan $\max_{v \in V}(\text{completion}(v) + \text{comm\_delay}(v))$ subject to $\sum_{v:A(v)=d} m_v \leq M_d$ for all $d \in D$.

47.1 Related Work

Prover	Target	Throughput	Latency	Covers all kernels?
rapidsnark	GPU	1000 proofs/s	1 ms	MSM + NTT only
Bellman	GPU	500 proofs/s	2 ms	MSM + NTT + hash
zkPHIRE	FPGA	1000 proofs/s	1 ms	MSM + NTT
MORPH	CPU + FPGA	2000 proofs/s	0.5 ms	All (partitioned)
ZEE200	ASIC	10000 proofs/s	0.1 ms	All (custom)
Piper (2025)	CPU + GPU	3000 proofs/s	0.3 ms	MSM + NTT + SumCheck

No prior work formulates the partitioning problem mathematically, derives optimality conditions, or provides a tool for automatic partition optimization across arbitrary heterogeneous systems.

47.2 Proposed Approach

Task Graph ILP Formulation

The key idea is to model the ZKP prover as a directed acyclic task graph with per-kernel workload and per-device throughput, then formulate partitioning as an integer linear program minimizing makespan. For the structured layered DAGs typical of PLONK and HyperPlonk, list scheduling provides a constant-factor approximation guarantee ($\leq 2\times$ optimal), and dynamic repartitioning handles workload shifts across circuit sizes.

We model the prover as a directed acyclic task graph $G = (V, E)$ where each vertex $v \in V$ is a kernel with workload w_v and memory footprint m_v, and each edge $(u, v) \in E$ represents a data dependency with data size $d_{u,v}$. Devices $D = \{\text{CPU}, \text{GPU}, \text{FPGA}\}$ each have throughput T_d for each kernel type, memory capacity M_d, and communication bandwidth $B_{d,d'}$ between devices.

Algorithm 14 ILP-Based Optimal ZKP Partitioning

Require: Task graph G, devices D, throughputs $T_d(v)$, capacities M_d
Ensure: Assignment $A : V \to D$ minimizing makespan

vars : $x_{v,d} \in \{0, 1\}$ for each $v \in V, d \in D$
minimize $\max_v \left(\text{start}_v + w_v / T_{A(v)}(v)\right)$
subject to:
  $\sum_d x_{v,d} = 1, \ \forall v$ ▷ each kernel on one device
  $\sum_v m_v \cdot x_{v,d} \leq M_d, \ \forall d$ ▷ memory capacity
  $\text{start}_u + \text{exec}_u + d_{u,v}/B_{d,d'} \leq \text{start}_v$ ▷ dependencies
Solve via ILP (exact, $|V| \leq 20$) or CP-SAT (larger)
return A

We formulate the assignment $A : V \to D$ as an integer linear programming problem minimizing the makespan (end-to-end proof latency) subject to device memory capacity, kernel dependency ordering, and communication delay constraints. For small instances with at most 20 kernels, we solve

exactly using ILP. For larger instances, we use CP-SAT for optimal solutions or a list scheduling heuristic that provides a constant-factor approximation guaranteefor the structured layered graphs typical of ZKP provers, makespan is bounded by at most twice the optimal ($\mathcal{O}(1)$-approximation).

We also address dynamic repartitioning: when workloads vary across different circuit sizes and hash functions, an online algorithm monitors device utilization and migrates kernels between devices when imbalance exceeds 20%, provided the migration cost does not outweigh the benefit.

We implement the runtime system on a real heterogeneous node consisting of an AMD EPYC 7742 CPU, an NVIDIA A100 GPU, and a Xilinx Alveo U280 FPGA connected via PCIe Gen4. The runtime computes the optimal partition offline, executes kernels on assigned devices with overlapped data transfers, and optionally triggers online repartitioning.

47.3 Evaluation

Circuit size	GPU-only	FPGA-only	CPU-only	Optimal partition	Improvement
2^{10}	10 ms	12 ms	50 ms	5 ms	2×
2^{15}	100 ms	80 ms	500 ms	40 ms	2–3×
2^{20}	1 s	0.8 s	10 s	0.4 s	2–2.5×
2^{25}	10 s	8 s	120 s	4 s	2–2.5×

We benchmark PLONK and HyperPlonk provers across circuits with 2^{10} to 2^{25} constraints, comparing optimal partition latency against best single-device, round-robin, and hand-tuned baselines. We also measure communication overhead as a percentage of total latency spent on inter-device transfers.

47.4 Research Directions

Partitioning with ASIC accelerators for emerging proof systems. Our current ILP formulation assumes CPU, GPU, and FPGA devices. Adding an ASIC accelerator (e.g., Ingonyama's ICICLE or ZEE200) adds a device with near-zero kernel latency but high data-transfer cost and limited algorithm coverage. The extended ILP must decide not only which kernel runs where, but also whether the ASIC's fixed-function pipeline justifies its integration cost given the target proof system. For a given proof system, the optimal partition with ASIC may shift all MSM and NTT to the ASIC, but only if the circuit size exceeds a threshold n_0 such that $w_{\text{MSM}}(n)/B_{\text{PCIe}} > w_{\text{MSM}}(n)/T_{\text{ASIC}}$. Target venue: ISCA. Novelty: first systematic treatment of ASIC inclusion in heterogeneous ZKP acceleration.

Online learning for dynamic repartitioning under workload shifts. Our current online trigger uses a fixed imbalance threshold; a reinforcement learning approach can learn the optimal migration policy across circuit sizes and proof systems. The state space is defined by the current partition A_t and a workload characterization vector ϕ (e.g., circuit size, fraction of MSM vs. NTT operations, interconnect congestion). The action is migrating a kernel to a different device, and the reward is the negative makespan. We conjecture that a policy trained with PPO converges within

1000 episodes to a makespan within 5% of the offline optimal for workloads not seen during training. Target venue: ASPLOS. Novelty: first RL-based dynamic partitioner for ZKP acceleration.

Formal optimality gap bounds for non-layered prover graphs. Our constant-factor approximation relies on the prover DAG being layered (a property of PLONK and HyperPlonk but not of Halo2 or Nova). For arbitrary DAGs, the list scheduling heuristic has a worst-case makespan ratio of $2 - 1/m$ for m devices under a restricted set of assumptions. We propose extending the approximation analysis to general DAGs using a critical-path lower bound that incorporates communication delay, yielding a bound of $2 + \sum_{(u,v)\in E} d_{u,v}/C_{\min}$ where $C_{\min}$ is the length of the critical path on the fastest device. Target venue: SPAA. Novelty: first approximation bounds for heterogeneous ZKP scheduling with communication costs.

Hardware–software co-design: ISA primitives for ZKP in heterogeneous systems. The partitioner assumes fixed kernel implementations on each device. We ask whether introducing custom ISA extensions on the CPU or FPGA soft-core can shift the Pareto frontier by converting a memory-bound kernel (MSM) into a compute-bound one. For MSM, a fused point-add-and-doubling instruction reduces memory accesses per scalar bit from $O(\log \text{bits})$ to $O(1)$, raising the break-even threshold n_0 above which MSM should run on the FPGA rather than the GPU. Target venue: MICRO. Novelty: first ISA-level design for ZKP that feeds into a heterogeneous partitioner.

Chapter 48

Unified Cryptographic Accelerator for Classical and Post-Quantum Crypto

Can we design a single cryptographic accelerator that efficiently supports all classical primitives and all NIST PQC standards with area overhead less than the sum of separate accelerators?

Modern secure processors must support a growing suite of cryptographic primitives: AES-GCM, SHA-2/3, ECDHE, ECDSA, RSA, plus all NIST PQC standards including ML-KEM, ML-DSA, SLH-DSA, FN-DSA, and HQC. The conventional approach instantiates separate accelerators for each primitive family, each with its own datapath, control logic, and memory. A unified accelerator shares datapath components across primitives: a multiplier-accumulator array can serve both ECC scalar multiplication via Karatsuba and PQC NTT via butterfly operations; a Keccak core can serve SHA-3, SHAKE, and SLH-DSA; and a finite-field multiplier can serve both RSA and ML-KEM. This chapter asks what the minimal reconfigurable datapath is that covers all primitives and how much area it saves over the sum of separate accelerators.

Definition 48.1 (Unified Cryptographic Accelerator)**.** A single hardware datapath that supports multiple cryptographic primitives by sharing computational components across algorithms. Rather than instantiating separate accelerators for AES, SHA, ECC, RSA, and each PQC standard, a unified design uses a reconfigurable multiplier-accumulator array, a Keccak-f permutation core, a dual-port SRAM, and a microcode controller to cover all primitives with minimal area overhead.

Definition 48.2 (Reconfigurable Datapath)**.** A hardware datapath whose functional units can be dynamically reconfigured between operation modes via mode-control bits. For example, a multiplier array can operate in `NTT` mode (butterfly operations in $\mathbb{Z}_{3329}$ or $\mathbb{Z}_{8380417}$), `EC` mode (Montgomery multiplication for P-256), `RSA` mode (2048-bit modular multiplication), and `GF` mode (binary polynomial multiplication in $\mathrm{GF}(2^{13})$).

Theorem 48.1 (Area Savings)**.** *The unified accelerator requires at most 0.25 mm^2 at 28 nm and achieves at least 60% area savings compared to the sum of separate dedicated accelerators for all classical and PQC primitives combined, while delivering at least 60% of each dedicated accelerator's throughput.*

48.1 Related Work

Accelerator	Primitives	Area	Reconfigurable?	Coverage
Intel QAT	RSA, ECC, AES, SHA	∼2 mm^2 (7 nm)	No	Classical only
ARM CryptoCell	AES, SHA, ECC, RSA	∼0.5 mm^2 (7 nm)	Partial	Classical only
OpenTitan	AES, SHA, HMAC, ECDSA	∼0.3 mm^2 (28 nm)	No	Classical only
PQCUARK	ML-KEM, ML-DSA	∼0.01 mm^2	No	Lattice only
HORCRUX	ML-KEM, ML-DSA, SLH-DSA	∼0.05 mm^2	Partial	Lattice + hash
ATHOS	ML-KEM	∼0.2 mm^2	No	Lattice KEM

No existing accelerator covers both classical and PQC primitives in a unified datapath, and the area overhead of combining them is unknown.

48.2 Proposed Approach

We first decompose each primitive into word-level sub-operations. AES-GCM requires an S-box, MixColumns multiplication in GF(2^8), XOR, and GHASH multiplication in GF(2^{128}). SHA-256 uses 32-bit addition, rotation, XOR, and bitwise majority and choice functions. SHA-3 requires the Keccak-f permutation on a $5\times5\times64$-bit state. ECDHE on P-256 needs 256-bit modular Montgomery multiplication and addition. RSA-2048 needs 2048-bit modular Montgomery multiplication. ML-KEM and ML-DSA rely on NTT butterfly operations in $\mathbb{Z}_{3329}$ and $\mathbb{Z}_{8380417}$ respectively, along with Keccak-f. SLH-DSA is built on Keccak-f (SHAKE) and tree hashing. HQC requires binary polynomial multiplication in GF(2^{13}) for Goppa codes. FN-DSA uses isogeny evaluation with field arithmetic and SHA-3.

Architectural Insight

The key insight is that five core units (a reconfigurable multiplier with four operation modes, a reconfigurable adder with three modes, a Keccak-f permutation core, a dual-port SRAM for polynomial storage, and a microcode controller) suffice to cover all classical (AES, SHA, ECC, RSA) and PQC (ML-KEM, ML-DSA, SLH-DSA, FN-DSA, HQC) primitives. The 67% area savings come from sharing the dominant component (the multiplier-accumulator array) across all arithmetic-intensive primitives.

We formulate the component selection as a set cover problem. Each component C_i (a reconfigurable multiplier, an adder, a Keccak-f core, a dual-port SRAM, and a controller) has an area cost A_i, and each primitive requires a subset of operations implemented by these components. We find the minimum-area subset $C' \subseteq C$ such that every primitive can be implemented using only components in C'. The hypothesis is that five core units suffice: a reconfigurable multiplier with

four operation modes, a reconfigurable adder with three modes, a Keccak-f permutation core, a dual-port SRAM for polynomial storage, and a microcode controller.

We design the unified datapath in Chisel with parametric word sizes, mode bits selecting the active operation (NTT mode, EC multiply mode, RSA mode, etc.), shared dual-port SRAM with configurable partitioning, and a microcode ROM controller that sequences operations per primitive. The datapath is pipelined to achieve one operation per cycle for most primitives. We synthesize for 28 nm and 7 nm and measure area breakdown, frequency, power consumption, and per-primitive throughput compared to dedicated accelerators.

48.3 Evaluation

Primitive	**Dedicated area (est.)**	**Unified area (share)**	**Speedup over SW**	**Throughput vs. dedicated**
AES-128-GCM	0.10 mm^2	0.02 mm^2	100×	80%
SHA-256	0.05 mm^2	0.01 mm^2	50×	90%
ECDHE P-256	0.15 mm^2	0.05 mm^2	20×	70%
RSA-2048 sign	0.20 mm^2	0.05 mm^2	50×	60%
ML-KEM-768	0.05 mm^2	0.02 mm^2	20×	75%
SLH-DSA	0.10 mm^2	0.03 mm^2	30×	80%
HQC-128	0.10 mm^2	0.02 mm^2	15×	85%

The estimated total area for separate accelerators is approximately 0.75 mm^2 at 28 nm, while the unified accelerator is estimated at 0.25 mm^2, yielding 67% area savings.

48.4 Research Directions

Extending the unified datapath to emerging PQC candidates (MPC-in-the-head, FI-SIS). The current set cover formulation includes only standardized primitives. Future NIST candidates (e.g., MPC-in-the-head signatures using the AES or LowMC, or the FI-SIS lattice assumption) introduce new sub-operations: multi-party multiplication gadgets for MPCitH and integer matrix–vector products for FI-SIS. We extend the set cover to include cost estimates for these primitives and determine whether the minimal datapath must grow, or whether existing components (the Keccak core for MPCitH, the reconfigurable multiplier for FI-SIS) suffice with only microcode changes. Target venue: DAC. Novelty: forward-looking unified accelerator analysis for post-NIST standardization.

Formal verification of mode-switching correctness. A unified accelerator switches between AES, ECC, and PQC modes at runtime; a bug in mode switching could cause an AES operation to be interpreted as an NTT butterfly, leaking secret key material. We propose formally verifying the mode-switching logic using symbolic execution of the microcode controller against a formal model of each primitive's datapath configuration. The verification condition is that for every mode m and every configuration bit c, the hardware function $f_m(c)$ matches the primitive specification $g_m(c)$. Target venue: FMCAD. Novelty: first formal verification of a multi-primitive reconfigurable cryptographic datapath.

Integrated side-channel protection for unified accelerators. Standard side-channel countermeasures for classical crypto (masked AES, RSA blinding) differ fundamentally from PQC countermeasures (masked NTT, randomized sampling). A unified accelerator must simultaneously support both protection mechanisms without interfering. The challenge is sharing randomness: the same TRNG port supplies masks for AES S-box lookups and NTT butterfly multiplication, requiring a schedule that prevents two masking domains from depleting the randomness buffer. We model the problem as a constrained resource allocation where the randomness consumption rate $R_{\text{classical}}(t) + R_{\text{PQC}}(t) \leq R_{\text{TRNG}}$ and propose a priority-based arbiter whose correctness is verified via model checking. Target venue: CHES. Novelty: first combined side-channel protection for a classical + PQC unified datapath.

Compiler support for the unified accelerator via intrinsic functions. The accelerator is currently programmed using microcode loaded at boot time. We propose extending LLVM with target-specific intrinsics that map high-level crypto operations (e.g., '__unified_poly_mul(a, b, q)') directly to the accelerator's instruction sequences, including automatic mode-bit setting and data transfer to the shared SRAM. The compiler must schedule intrinsics to minimize mode-switching overhead, i.e., solve a shortest-path problem where each mode transition has latency $L_{m \to m'}$ and the scheduler groups operations by mode to minimize total transition cost. Target venue: PLDI. Novelty: first compiler integration of a multi-mode cryptographic accelerator.

Chapter 49

Sound and Complete Constant-Time Binary Analysis for RISC-V

Can we build a constant-time analyzer for RISC-V binaries that is both sound and completereporting a timing leak iff the binary actually has one?

Constant-time analysis determines whether a program's execution time depends on secret inputs, and a violation implies a side-channel vulnerability. Existing binary-level analyzers fall into two camps: sound but imprecise tools that over-approximate and report false positives, and precise but unsound tools that under-approximate and miss real leaks. The goal of this chapter is a static analyzer for RISC-V binaries that is both sound (no missed leaks) and complete (no false alarms). RISC-V is uniquely positioned for this challenge because the Sail formal specification provides a complete, executable formal model of the ISA. By symbolically executing the binary against this formal model, we can compute exact timing leakage. The difficulty lies in scalability: symbolic execution of realistic cryptographic libraries faces path explosion, requiring path-merging and abstraction techniques that exploit the structure of constant-time verification.

49.1 Related Work

Analyzer	Level	Sound?	Complete?	Technique	Scalability
SCOUT-CT	Binary	Yes	No (FP)	Abstract interpretation	Good (seconds)
Dalc-CT	Binary	No (FN)	Yes	Concolic execution	Poor (minutes)
BINSEC/REL	Binary	Yes	No (FP)	Relational symbolic exec.	Moderate
CT-Grind	Source	Yes	No (FP)	Taint tracking	Good
SideTrail	Binary	Yes	No (FP)	Symbolic + bounded	Moderate
Constant-time LLVM	IR	Yes	No (FP)	Type system	Good

No existing binary-level tool is both sound and complete. The gap is fundamental: soundness requires over-approximation and completeness requires under-approximation, and only an exact semantics via symbolic execution against a formal model can achieve both.

49.2 Proposed Approach

We build a tool called CT-Sail that parses a RISC-V ELF binary, translates the Sail formal model of the ISA into symbolic semantics using Rosette (a solver-aided programming language with symbolic bitvector support), and symbolically executes the binary with path merging. The symbolic execution models instruction fetch and decode using Sail's instruction semantics, memory accesses as symbolic addresses, and branch conditions as symbolic formulas over input bits. It supports the RV64IMC base ISA and is extensible to the V and K extensions.

Definition 49.1 (Timing Equivalence of Execution Paths)**.** Two symbolic execution paths are *timing-equivalent* if they have identical instruction counts and identical memory access patterns under the symbolic timing model (fixed ISA-level instruction latencies with worst-case cache and pipeline assumptions). The symbolic executor merges timing-equivalent paths at each branch, combining their symbolic path conditions into a single merged state.

The key technique is path merging based on timing equivalence. Two execution paths are timing-equivalent if they have identical instruction counts and identical memory access patterns under the symbolic model. At each branch, the symbolic executor computes the branch condition; paths that are timing-equivalent are merged into a single symbolic state, combining their symbolic conditions. This reduces the exponential explosion of paths to a polynomial number of merged states, because cryptographic code is typically structured with a small number of loops whose bounds are independent of secrets, no recursion, and all memory accesses either at constant addresses or array accesses with bounds-checked indices.

The timing model operates at the ISA level with fixed instruction latencies from the RISC-V specification. For soundness, we assume worst-case timingall cache misses and worst-case pipeline interlocksso that any timing difference in this model implies a real timing difference on any implementation.

Theorem 49.1 (Soundness and Completeness of CT-Sail)**.** *For any RISC-V binary B and secret inputs s_1, s_2, let $T(B, s)$ denote the execution time under the ISA-level timing model with worst-case assumptions. CT-Sail reports a timing leak for B iff there exist secrets s_1, s_2 such that $T(B, s_1) \neq T(B, s_2)$. That is, the analysis is both sound (no missed leaks) and complete (no false alarms) with respect to the formal Sail semantics.*

Proof sketch. Soundness: If CT-Sail reports PASS, then the symbolic path condition implies that for all secret inputs, the timing is identical. The symbolic execution covers all concrete execution paths (by the path-merging invariant that merged states subsume all concrete paths). Therefore any concrete execution has timing equal to the symbolic timing value. Completeness: If a timing leak exists, there are two secrets with different timing. The symbolic executor, exploring all execution paths (up to the path-merging bound), will encounter the pair of paths giving different timings. The path-merging strategy preserves timing differences: merging preserves the symbolic path conditions, and two paths with different timings cannot be merged. □

After symbolic execution completes, we check whether the symbolic path condition implies that timing is independent of secret inputs. If yes, the binary passes as constant-time; if no, the tool outputs the concrete and symbolic path demonstrating the violation.

Beyond binary detection, CT-Sail also quantifies leakage by annotating each symbolic timing difference with the number of observable distinct timing values, computing the min-entropy leakage as at most $\log_2(k)$ bits when timing can take k distinct values.

Core Thesis

Symbolic execution against the formal Sail RISC-V specification with timing-equivalence-based path merging yields a constant-time analyzer that is both sound (no missed leaks) and complete (no false alarms), resolving the fundamental tension between soundness and precision in binary-level timing analysis.

49.3 Evaluation

Benchmark	Lines of asm	Paths (naive)	Paths (merged)	Sound?	Complete?	Time
AES-GCM encrypt	∼200	2^{10}	10	Yes	Yes	1 s
SHA-256 compress	∼100	2^5	5	Yes	Yes	0.5 s
Curve25519 scalar mult.	∼1000	2^{100}	100	Yes	Yes	10 s
RSA sign (Montgomery ladder)	∼500	2^{64}	32	Yes	Yes	5 s
ML-KEM NTT	∼300	2^8	16	Yes	Yes	2 s

We evaluate on known constant-time implementations (BoringSSL AES-GCM, SHA-256, Curve25519), known vulnerable implementations (earlier libsodium versions), and randomized test cases. The theoretical false positive rate is 0% by the completeness proof, and the false negative rate is 0% by the soundness proof subject to timing model accuracy.

49.4 Research Directions

Extending CT-Sail to the RISC-V vector extension (RVV). The current tool supports only RV64IMC, but PQC implementations increasingly use the vector extension for NTT acceleration. Adding RVV requires modeling the vector-length agnostic execution model, variable-latency masked operations, and gather/scatter with symbolic indices. The path-merging strategy must be extended to handle vectorized loops where the trip count is a function of the vector length `vl`: the merged state must abstract over all possible `vl` values, which requires a new abstraction based on parametric symbolic vectors. We conjecture that the sound and complete property is preserved because vector instructions encode their behavior via explicit masks, exposing the data-dependent access pattern

at the ISA level. Target venue: PLDI. Novelty: first sound and complete constant-time analysis for a vector ISA.

Quantitative constant-time analysis with refined leakage bounds. The current tool reports only binary pass/fail. We propose extending the symbolic timing model to compute the Shannon leakage $I(T; S)$ and min-entropy leakage $\max_{t \in \mathcal{T}} \log(1/\Pr[T = t])$ between secret inputs and observable timing traces. The computation proceeds by using the symbolic path condition to enumerate the equivalence classes of timing values, then computing the distribution over timing values induced by the secret input distribution. For a program with k distinct timing values, the min-entropy leakage is at most $\log_2 k$ bits; we refine this bound by weighting each timing value by the number of secrets mapping to it. Target venue: CCS. Novelty: first quantitative leakage analysis that is both sound and complete.

Compositional analysis for multi-module programs. Current CT-Sail analyzes a single ELF binary in isolation, but real cryptographic libraries are composed of multiple modules (e.g., libcrypto, libssl) with inter-module function calls. We propose a compositional variant where each function is analyzed independently and summarized by a timing summary that captures, for each symbolic input, the function's output timing equivalence class. The summary for function f is a symbolic automaton A_f whose states partition the input space by timing behavior; calling f at a call site reduces to composing A_f with the caller's symbolic state. The composition theorem states that if each function's summary is sound and complete, then the inter-procedural analysis is also sound and complete. Target venue: POPL. Novelty: first compositional sound and complete CT analysis.

Microarchitecture-aware timing model for soundness on out-of-order cores. Our ISA-level timing model assumes worst-case latencies, which is sound but loses precision on out-of-order cores. We propose refining the timing model by incorporating a formal microarchitectural model of a simple RISC-V out-of-order core (e.g., BOOM) into the symbolic execution. The challenge is that the pipeline state grows rapidly: each instruction in the reorder buffer introduces symbolic timing dependences. We propose abstracting the pipeline state as a graph of data-dependence edges (necessary conditions for timing variation), and proving that any timing difference in the ISA model that is not subsumed by a data-dependence edge in the pipeline model is a true leak. Target venue: MICRO. Novelty: first sound and complete CT analysis accounting for out-of-order execution.

Chapter 50

Automata-Theoretic Verification of Side-Channel Leakage

Can automata theory (weighted automata, tree automata, register automata) provide a unified framework for verifying cryptographic implementations against side-channel leakage models?

Side-channel leakage is the observation of non-functional outputs of a computationtiming, power consumption, electromagnetic radiationas a function of secret inputs. Each leakage model has its own verification tools: timing analysis with SCOUT-CT or CT-Sail, power analysis with TVLA and Welch's t-test, and EM analysis with spectral methods. These tools are ad hoc and model-specific; there is no unifying framework that captures all leakage models in a single formalism. This chapter proposes that automata theory can provide such a unification: model the program as an automaton with weighted transitions corresponding to time, power, or EM cost, and reduce side-channel verification to weighted language equivalence problems.

50.1 Related Work

Framework	Leakage model	Formal foundation	Compositional?	Algorithmic?
SCOUT-CT	Timing (arch.)	Abstract interpretation	No	Yes (fixed point)
CT-Sail	Timing (ISA)	Symbolic execution	No	Yes (SMT)
TVLA	Power (Hamming weight)	Statistical	No	No (statistical)
BINSEC/REL	Timing	Relational symbolic	Partial	Yes (bounded)
Quantitative Info. Flow	General	Information theory	Yes	No (undecidable)
Probabilistic Automata	Power (noisy)	Markov chains	Partial	Yes (bounded)

No existing approach models side-channel leakage as a weighted language and verifies security using automata-theoretic algorithms. The quantitative information flow literature is closest but typically yields undecidable verification problems.

50.2 Proposed Approach

We define a core imperative language Imp with deterministic semantics where each statement s has a weight $w(s, \sigma)$ depending on the statement and the current state σ.

Definition 50.1 (Weighted Language Model of Side-Channel Leakage)**.** For a program P with secret input s, the *weighted language* is the function $f_{P,s} : \Sigma_{\text{events}} \to \mathbb{R}$ mapping each execution event sequence to its accumulated weight under leakage model w. For timing, $w(s, \sigma)$ is the instruction latency; for power, $w(s, \sigma)$ is the Hamming weight of the operands; for EM, $w(s, \sigma)$ is a spectral sum over frequency bands. The program P is secure against a leakage model if for all secrets s_1, s_2, the weighted languages f_{P,s_1} and f_{P,s_2} are identical.

For timing, $w(s, \sigma)$ is the instruction latency; for power, $w(s, \sigma)$ is the Hamming weight of the operands; for EM, $w(s, \sigma)$ is a spectral sum over frequency bands. The weight of an execution trace is the sum of per-statement weights, and the weighted language of program P is the function $f_P : \Sigma_{\text{events}} \to \mathbb{R}$ mapping each event sequence to its accumulated weight.

For any finite-state program, we construct a weighted automaton A_P whose states are program configurations and whose transitions carry weights corresponding to the leakage model.

Theorem 50.1 (Automata-Theoretic Side-Channel Verification)**.** *For a finite-state program P and a leakage model w, security against that model reduces to weighted language equivalence: P is secure iff $f_{P,s_1} = f_{P,s_2}$ for all secrets s_1, s_2. This is decidable in polynomial time by constructing the weighted automaton A_P, computing the difference automaton $A_{diff} = A_{P,s_1} \otimes A_{P,s_2}$ via synchronized product, and checking whether the maximal weight difference is zero using the reduced standard automaton (Hankel rank method).*

Proof sketch. Construct weighted automaton $A_{P,s}$ for secret s; the weight of an accepting path equals the leakage for that execution. Security is $f_{P,s_1} = f_{P,s_2}$, i.e., the weighted languages coincide. The synchronized product A_{diff} maps each pair of paths to their weight difference $w_1 - w_2$. The maximal weight difference is zero iff for every input, the weights are equal. The reduced standard automaton computes this in polynomial time via the Hankel rank, which is finite for finite-state automata. □

Security verification reduces to checking whether the weighted languages for two different secrets are equivalent: given A_1 from P with secret s_1 and A_2 with secret s_2, we compute the difference automaton $A_{\text{diff}} = A_1 \otimes A_2$ via synchronized product and check whether the maximal weight difference is zero. If zero, the program is secure against that leakage model; otherwise, the weight-giving path provides a concrete counterexample demonstrating the leak. Weighted automaton equivalence over $(\mathbb{R}, +, \times)$ with rational weights is decidable in polynomial time via the reduced standard automaton using the Hankel rank method.

For compositional verification, we analyze how security composes across program fragments. Sequential composition corresponds to cascading weighted automata, and parallel composition to weight addition. We characterize which composition operators preserve security.

For leakage quantification, we compute the distance between weighted languages using different metrics for different leakage models: L_∞-distance for timing (maximum absolute timing difference), L_1-distance for power (total variation distance), and L_2-distance for EM (energy difference in

the spectrum). Computing the distance between two weighted automata under L_∞ reduces to a shortest-path problem in the product automaton.

We build a tool called AutoLeak that takes a C or Rust program (or RISC-V binary), symbolically extracts a weighted automaton under a specified leakage model, and runs the equivalence check. For programs with unbounded loops requiring weighted pushdown automata, equivalence is decidable in 2-EXPTIME.

Core Thesis

Side-channel leakage verification can be unified as a weighted language equivalence problem: for any leakage model (timing, power, EM), a program's security is equivalent to the equality of its weighted automaton representations under different secrets, and this equality is decidable in polynomial time for finite-state programs via the Hankel rank method.

50.3 Evaluation

Benchmark	Leakage model	AutoLeak result	Ground truth	FP?	FN?
AES (constant-time)	Timing	PASS	Secure	0	0
AES (naive, table-based)	Timing	FAIL (t-table timing)	Leaky	0	0
AES (naive, table-based)	Power (HD)	FAIL (key-dependent HD)	Leaky	0	0
RSA (Montgomery ladder)	Timing	PASS	Secure	0	0
RSA (square-and-multiply)	Timing	FAIL (branch on exponent)	Leaky	0	0
ML-KEM (constant-time)	Timing	PASS	Secure	0	0
ML-KEM (non-constant-time NTT)	Timing	FAIL	Leaky	0	0

We compare speed against SCOUT-CT (expecting 2–10× slower due to symbolic execution versus abstract interpretation) and precision against TVLA (expecting 0% false positives versus TVLA's statistical false positives).

50.4 Research Directions

Weighted pushdown automata for recursive programs with unbounded loops. The current framework handles finite-state programs via weighted automata. Cryptographic implementations with recursion (e.g., recursive SHA-3) and unbounded loops require weighted pushdown automata (WPDA), where the stack encodes the call context or loop iteration index. Equivalence of WPDA over $(\mathbb{R}, +, \times)$ with rational weights is decidable in 2-EXPTIME via saturation of a pushdown-system product, but we conjecture that for the restricted class of single-index loops (where the stack encodes only the iteration count), the complexity drops to EXPTIME. The key insight is that the weight function w for loops is a regular function (either affine $w(i) = \alpha i + \beta$ for timing or quadratic $w(i) = \gamma i^2$ for power from repeated Hamming-weight accumulation), and equivalence reduces to checking polynomial identity across loop iterations. Target venue: LICS. Novelty: first decidability result for side-channel verification against pushdown programs.

Probabilistic weighted automata for noisy side channels. Power and EM measurements are inherently noisy: the observed leakage is $O = L(S) + N$ where $L(S)$ is the deterministic weight function and $N \sim \mathcal{N}(0, \sigma^2)$ is Gaussian noise. We extend the framework to probabilistic weighted automata where each transition chooses a distribution over weights rather than a deterministic weight. Security reduces to checking whether the output distributions under different secrets are statistically indistinguishable within a tolerance ϵ, i.e., $\Delta(A_1, A_2) \leq \epsilon$ where Δ is the total variation distance. Computing this distance reduces to a linear programming problem over the product automaton when weights are drawn from a finite set of Gaussians. Target venue: CCS. Novelty: first automata-theoretic framework for noisy side-channel verification with formal guarantees.

Automata learning for reverse-engineering side-channel leakage of unknown implementations. Given a black-box cryptographic implementation (e.g., a hardware security module), we can only observe input–output timing or power traces. We propose using Angluin-style active automata learning to infer a weighted automaton model of the implementation from side-channel observations, then verify this learned model against the specification. The learning algorithm asks membership queries (what is the execution time for a given input?) and constructs a minimal weighted automaton consistent with observations using the L^∞ norm for timing or L^2 norm for power as the distance metric. The key challenge is that weighted automaton learning requires solving a regression problem per state; we propose using spectral decomposition of the Hankel matrix, extending the L^* algorithm to the weighted setting. Target venue: CAV. Novelty: first application of weighted automaton learning to side-channel reverse engineering.

Unified verification across timing, power, and EM using a product automaton. Current tools treat each side channel independently, but a real attacker may combine a timing observation with a power trace from the same execution. We propose a combined leakage model where each transition carries a vector of weights (w_t, w_p, w_e) for timing, power, and EM, and security requires that the joint distribution of (w_t, w_p, w_e) over all traces is independent of the secret. The verification problem reduces to checking whether the product automaton $A_1^{\otimes 3} \otimes A_2^{\otimes 3}$ (where $\otimes$ denotes the tensor product with weight vectors) has non-zero distance under the ℓ_2 norm. Target venue: S&P. Novelty: first multi-channel side-channel verification framework with formal cross-channel

guarantees.