# Schema–Segment Composition Computing System

Taeho Lee*
SSCCS Foundation
ssccs.org

February, 2026

Other Formats
HTML

## Abstract

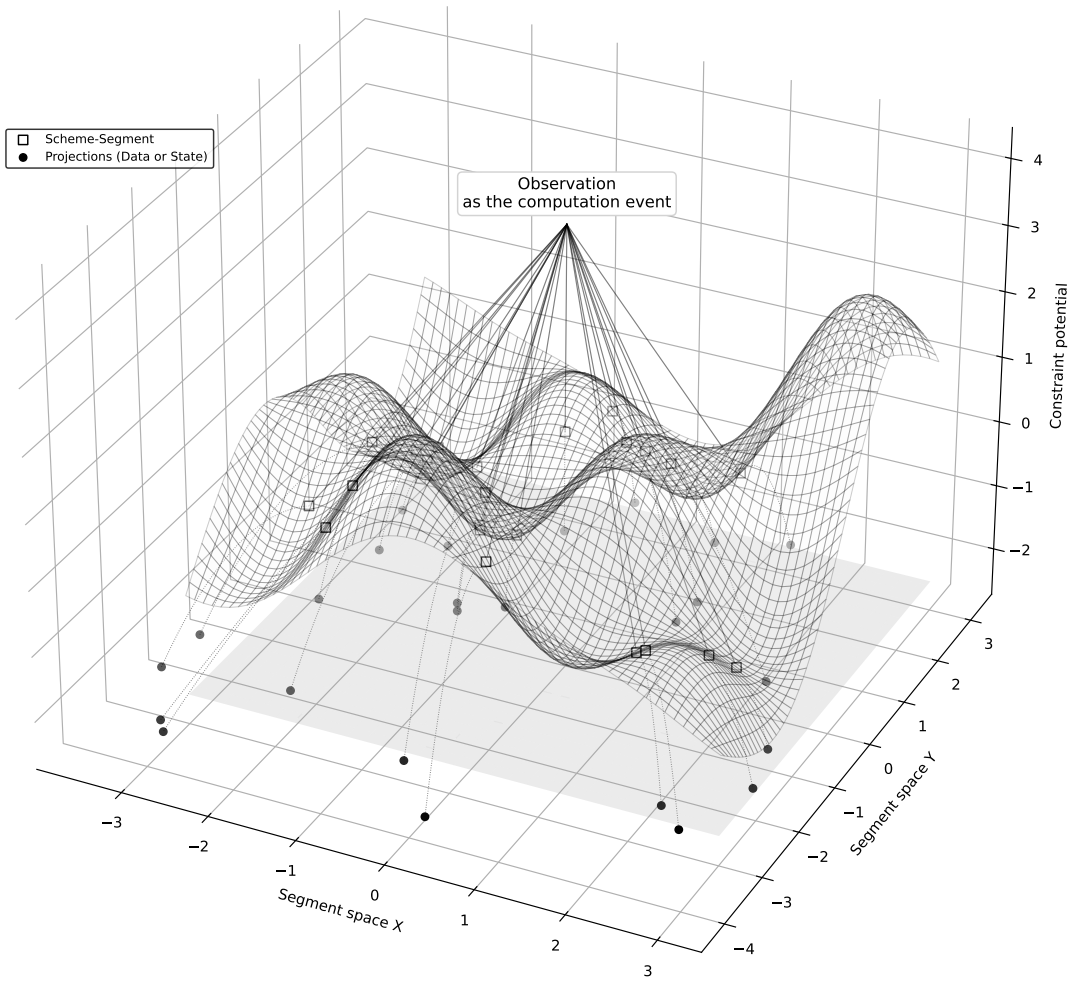
SSCCS (Schema–Segment Composition Computing System) is an observation-driven computing model that defines deterministic computation as the realization of structured potential under dynamic constraints. In an era of increasing complexity and distributed systems, this contrasts with the traditional von Neumann approach of instruction sequencing, state mutations, and data movement[1] between memory and processor, and the compiler's role shifts from translating code to optimizing the topology of data movement. This model treats time as one axis of multi-dimensional computation rather than an absolute sequence, with inherent structural isolation against interference and lossless interpretation via a Geometric Manifold.

Computation is formalized as the deterministic projection of immutable Segments and Schemes within dynamic Fields. Acting as mutable constraint units, Fields enable recursive composition and allow governance logic to be encrypted or sandboxed at the binary level. The compiler performs structural mapping, embedding logic directly into hardware topology to ensure stationary data (Logic-at-Rest) and minimize movement. This design innovation mitigates data movement overhead and enables inherent parallelism, targeting dramatic improvements in performance and energy efficiency. Security and cryptographic auditability are geometrically natural consequences of this immutable structure, rather than added features.

As a universal substrate, SSCCS provides a verifiable foundation for systems across domains—from AI to scientific computing to embedded systems. Driven by a software-first philosophy, this specification provides a roadmap where logical design dictates physical implementation, contrasting with current hardware advances that focus primarily on physical improvements. Ultimately, SSCCS aims to evolve into an open format [$F$] at the language layer, transitioning logic into a transparent, accessible, and energy-efficient Intellectual Public Commons.

*Founder & Engineering Architect; Corresponding author: lee@ssccs.org

# Philosophical Manifesto




**Loops disappear into layout. Data, or state, is the shadow cast by collapsed possibility. Time is one coordinate axis among many.**

A System where Structured Deployment is the Path, and Observed Synthesis is the Computation, which is not execution over time: Beneath immutable segments and schemes, observation momentarily activates the Field, precipitating the collapse of possibility and giving rise to a projection. A projection, as the residue of collapse, is transient; it constitutes what we recognize as data or state. Yet throughout this entire process, the fundamental structure itself remains untouched and unaltered.

- Whitepaper: PDF / HTML DOI: 10.5281/zenodo.18759106 via CERN/Zenodo, indexed by OpenAIRE. Licensed under *CC BY-NC-ND 4.0*.
- Official repository: GitHub. Authenticated via GPG: BCCB196BADF50C99. Licensed under *Apache 2.0*.
- Governed by the Foundational Charter and Statute of the SSCCS Foundation (in formation).
- Provenance: Human-in-Command, AI-assisted. Aligns with ISO/IEC JTC 1/SC 42 and C2PA-certified. Full intellectual responsibility with author(s).

# Table of contents

# 1 Introduction

SSCCS redefines computation through four primitives: **Segments** (immutable points), **Schemes** (immutable blueprints), **Fields** (mutable constraints), and **Observation** (the active event). This redefinition yields deterministic reproducibility: because the structure is fixed and observation is deterministic, every computation produces a verifiable trace from blueprint to projection.

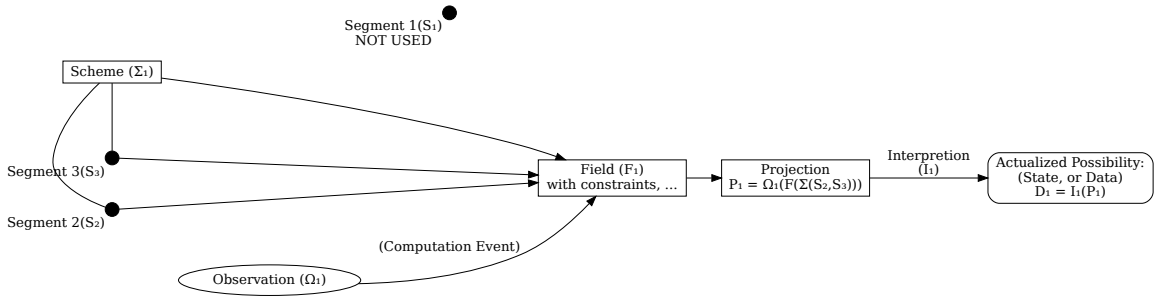



Figure 1: SSCCS concept of primitives

For decades, computation has been defined by the von Neumann model:

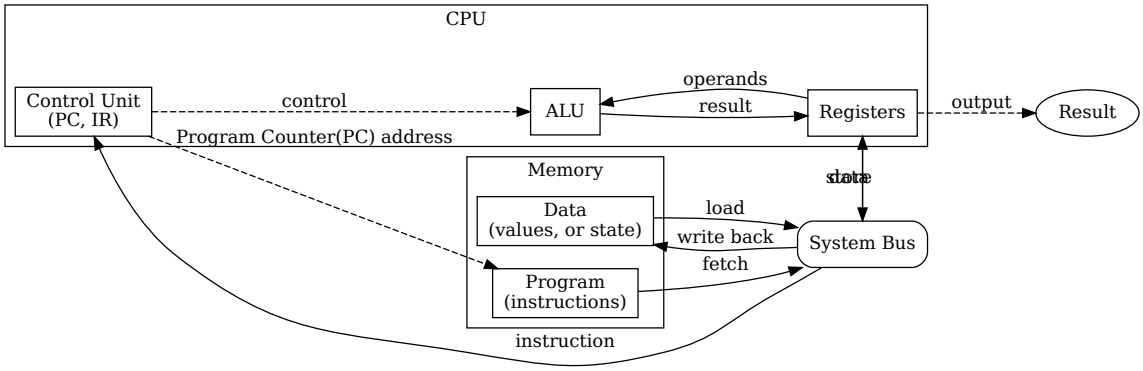



Figure 2: Von Neumann architecture: instruction sequencing and data movement

This formulation rests on several assumptions: data exists as intrinsic values in memory, programs are instruction sequences, and execution involves moving data between memory and processor across a sequential timeline. These assumptions are not fundamental laws but consequences of a specific architectural choice. Consequently, the majority of energy and time in conventional systems is spent on data movement rather than logic—a symptom known as the "data-movement wall" [1], [2], [3], [4].

While new hardware-side paradigms attempt to mitigate this, they remain localized optimizations within the same sequential paradigm. SSCCS proposes a shift from procedural execution to **structural observation.**

This project is made possible by three independent developments. First, the end of Dennard scaling has shifted the dominant energy cost in processors from arithmetic to data movement, creating conditions under which stationary-data models become economically rational. Second, open-source hardware ecosystems now permit a new computational abstraction to be prototyped on commodity FPGA boards and, if validated, transitioned to silicon without the capital requirements that historically restricted architectural innovation. Third, formal methods and proof assistants have advanced to the point where the determinism and race-freedom claimed by the model can be subjected to mechanical verification.

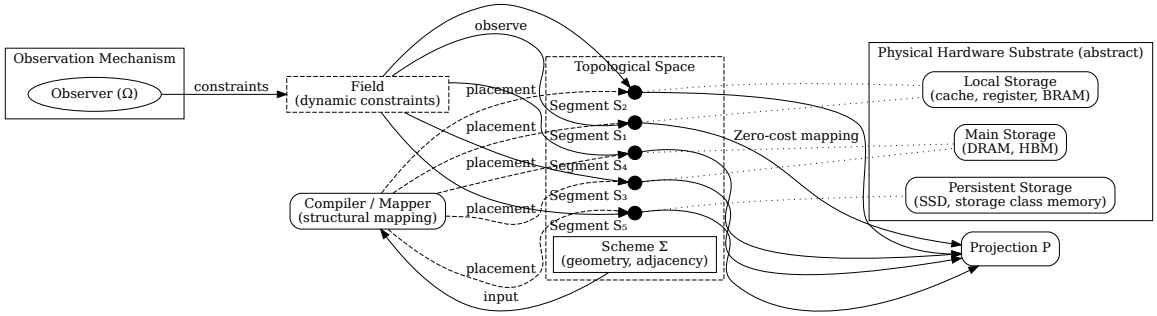



Figure 3: SSCCS logical-to-physical mapping: stationary segments observed through hardware-agnostic structural projection

Simply put, the Field governs the observation of the Scheme and its Segments, producing a Projection that can be interpreted as data. Each layer has defined properties and relationships; together they form the complete computational model.

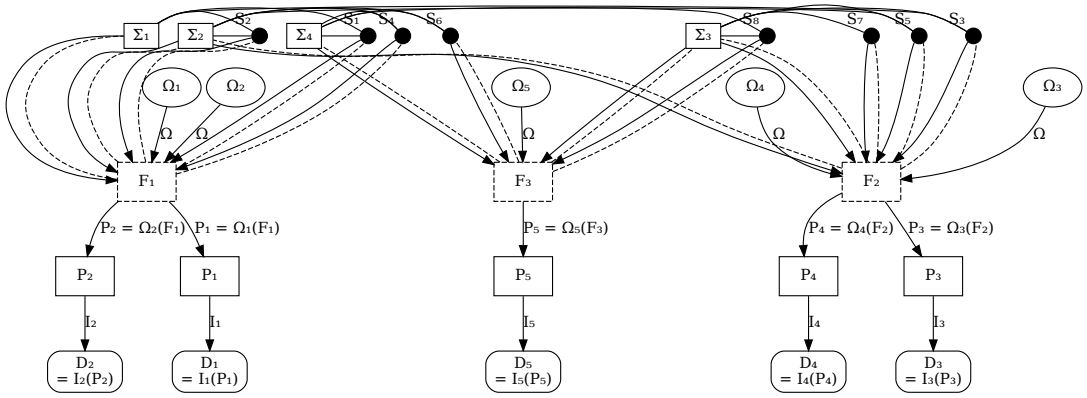



Figure 4: SSCCS multi-field, multi-observation parallel model with segment set

Through immutable Segments and Schemes, SSCCS achieves emergent parallelism without locks, eliminates data movement via structural mapping, and ensures deterministic results. Observation events can occur concurrently without temporal ordering, and the resulting projections are independent. **Time is not a fundamental dimension that governs state changes**; instead, the structure of Schemes and the constraints of Fields govern what can be observed and when.

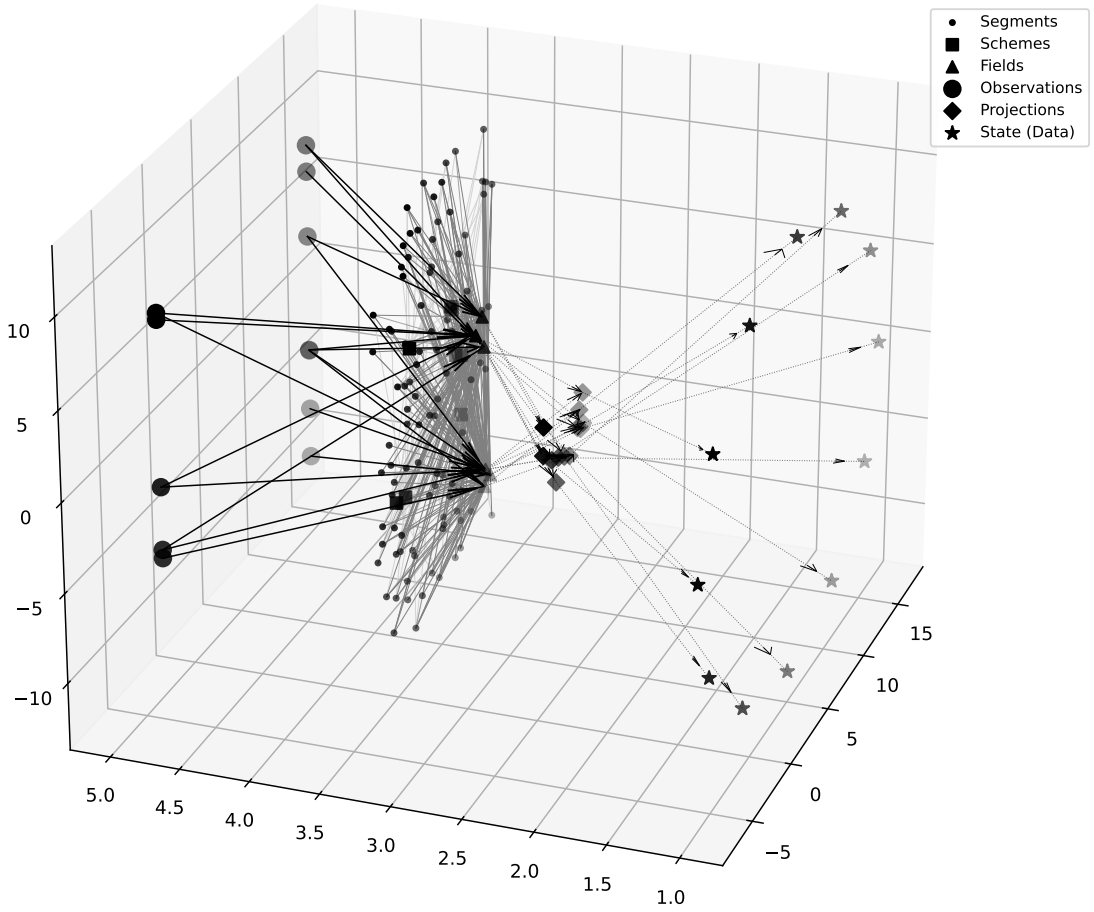



Figure 5: A structural composition visualization.



# 2 Definition of Primitives

## 2.1 Segment: Atomic Coordinate Existence

Let $\mathcal{S}$ denote the set of all Segments. A Segment $s \in \mathcal{S}$ is a tuple $(c, id)$ where $c \in \mathbb{R}^d$ represents coordinates in a $d$-dimensional possibility space, and $id = H(c)$ is a cryptographic hash providing a unique identifier.

Its properties are: **Immutability** (once created, a Segment cannot be modified), **Statelessness** (contains no values, only coordinates and identity). Because Segments contain no mutable state,

they can be observed concurrently by any number of observers without synchronization. The cryptographic identity ensures that every Segment is uniquely identifiable.

## 2.2 Scheme: Structural Blueprint

A Scheme $\Sigma$ is an immutable blueprint that defines the structural relationships-a geometric arrangement of Segments, not a sequence of operations. Segment relationships are spatial rather than temporal. During compilation, the compiler maps these spatial relationships directly to hardware addresses, ensuring that structurally adjacent Segments become physically adjacent. This design makes locality an inherent property of the specification, eliminating the need for runtime optimizations.

Formally, $\Sigma = (A, R, L, O)$ where:

- $A = \{a_1, \ldots, a_k\}$ is a set of axes, each axis $a_i = (\text{name}_i, \text{type}_i)$ with $\text{type}_i \in \{\text{Discrete}, \text{Continuous}, \text{Cyclic}, \text{Categorical}, \text{Relational}, \text{WithUnit}\}$.
- $R \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{T}$ is a set of structural relations, where $\mathcal{T}$ denotes relation types (Adjacency, Hierarchy, Dependency, Equivalence, Custom).
- $L : \mathbb{R}^d \to \mathcal{L}$ is a memory-layout mapping that assigns each coordinate a logical address.
- $O = (\text{resolution}, \text{triggers}, \text{priority}, \text{context})$ are observation rules that govern how observations are resolved, triggered, prioritized, and contextualized.

### 2.2.1 Axis Types

The axis set $A = \{a_1, \ldots, a_k\}$ defines the dimensional structure of a Scheme. Each axis $a_i = (\text{name}_i, \text{type}_i)$ where $\text{type}_i$ belongs to a variant set that includes Discrete, Continuous, Cyclic, Categorical, Relational, and WithUnit axes. Each variant carries distinct semantic meaning (e.g., Discrete for integer coordinates, Continuous for real-valued quantities, Cyclic for periodic dimensions). These axis types are purely semantic; they guide the interpretation of coordinates and the admissible relations between Segments, but do not prescribe a particular physical representation. The compiler uses the axis types to select appropriate layout strategies and to validate structural constraints. (See $K$ [$K.1$] for more details.)

### 2.2.2 Structural Relations

The relation set $R \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{T}$ captures the topological connections between Segments. Each relation is typed according to one of five fundamental categories: **Adjacency** (spatial or conceptual proximity), **Hierarchy** (parent–child relationships), **Dependency** (directed influence), **Equivalence** (symmetric or asymmetric equivalence classes), and **Custom** (user-defined predicates). Each category encompasses a range of sub-types that define precise topological relationships (e.g., Euclidean and Manhattan adjacency, containment and inheritance hierarchy, data-flow and control-flow dependency). Equivalence relations denote logical equivalence; each Segment remains a distinct entity.

Relations are immutable once the Scheme is created; they define the static topology that the compiler maps onto hardware. The compiler uses the relation types to extract independent sub-graphs, optimize locality, and generate observation code that respects the structural dependencies. (See $K$ [*K.2*] for more details.)

### 2.2.3 Memory-Layout Abstraction

The mapping $L : \mathbb{R}^d \to \mathcal{L}$ assigns each coordinate a logical address, decoupling the Scheme's geometric structure from the physical memory hierarchy. The `MemoryLayout` is specified by a `layout_type` and a mapping function. Available layout types include Linear, Row-Major, Column-Major, Space-Filling Curve, Hierarchical, Graph-Based, and Custom layouts, each mapping coordinates to logical addresses in a distinct manner.

The logical address $\ell = L(c)$ consists of a space identifier and an offset within that space. It serves as an intermediate representation that the compiler later translates to physical addresses according to the target hardware's memory hierarchy. The choice of layout type is a critical optimization: it determines how structurally adjacent Segments are placed in physical memory, thereby minimizing data movement during observation. (See $K$ [*K.3*] for more details.)

### 2.2.4 Observation Rules

The observation rules $O = (\text{resolution}, \text{triggers}, \text{priority}, \text{context})$ govern how the observation operator $\Omega$ is applied to the Scheme. Resolution strategies (deterministic, probabilistic, energy minimization, entropy maximization, external), triggers (on-demand, periodic, threshold, structural change, dependency satisfied, external event), priority levels, and context meta-constraints together enable fine-grained control over the observation process. For deterministic reproducibility—a core principle of SSCCS—a deterministic resolution strategy must be used; non-deterministic strategies are permitted only when strict reproducibility is not required.

Observation rules are part of the Scheme's immutable specification; they enable fine-grained control over the observation process, allowing the designer to trade off determinism against other desiderata. (See $K$ [*K.4*] for more details.)

### 2.2.5 Pre-defined Scheme Templates

SSCCS provides a set of canonical Scheme templates that capture common topological patterns, such as regular grids, integer lines, and arbitrary graphs. These templates are idiomatic combinations of axis, relation, and layout abstractions, serving as starting points for developers to construct custom Schemes. Detailed descriptions of each template are provided in the [$L$].

## 2.3 Field: Dynamic Constraint Substrate

The Field $F$ is the only mutable layer, but it does not store values. Instead, it stores admissibility conditions that dynamically constrain which configurations of Segments are possible at any given time. Formally, $F = (C, T)$ where:

- $C : \mathcal{S} \to \mathbb{B}$ is a constraint predicate (or a set of admissible coordinates).
- $T : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ is a transition matrix that assigns weights to possible transitions between Segments.

Mutating $F$ changes which configurations are possible, but does not modify any Segment.

### 2.3.1 Constraint Types

The constraint predicate $C(s)$ of a Field can express a wide variety of admissibility conditions. To aid systematic design, constraints can be categorized into five fundamental types based on their mathematical structure:

1. **Dimensional constraints** restrict the coordinate values of a Segment along one or more axes. They are expressed as inequalities or membership in intervals, e.g., $x \in [a, b]$ or $\|c\| \leq r$. Dimensional constraints are the simplest form of geometric filtering and correspond to traditional bounding-box or range queries.
2. **Topological constraints** govern the relational structure between Segments. While the Scheme defines static adjacency and dependency relations, a Field can impose dynamic topological requirements—for example, that two Segments must be adjacent, or that a path of a certain length must exist between them. Topological constraints are naturally expressed using the relation set $R$ of the Scheme and can be seen as predicates over the Scheme's graph.
3. **Algebraic constraints** involve equations or inequalities that combine coordinate values with arithmetic operations. Examples include linear equalities $a_1 x_1 + \cdots + a_k x_k = b$, quadratic forms $c^T Q c \leq \epsilon$, or more general polynomial constraints. Algebraic constraints enable rich geometric shapes (spheres, ellipsoids, hyperplanes) and are essential for modelling physical laws.
4. **Logical constraints** are Boolean combinations of other constraints, realized through Field composition. The union ($\cup$), intersection ($\cap$), and product ($\times$) operations allow arbitrary logical connectives (AND, OR, NOT) to be applied across multiple Fields. Logical constraints provide the compositional power needed to build complex admissibility conditions from simpler ones.
5. **Physical constraints** encode real-world limitations such as energy budgets, timing deadlines, or thermal thresholds. These quantities can be represented as additional coordinate axes (e.g., an energy axis $e$, a latency axis $\tau$) within the Segment's possibility space, or as auxiliary dimensions defined in the Scheme's axis set. Physical constraints are then expressed as inequalities on those axes, e.g., $e \leq E_{\text{max}}$, and can influence the transition matrix $T$ to reflect preferences among admissible Segments.

This taxonomy is not mutually exclusive; a single Field may contain constraints of several types. The classification serves as a conceptual tool for designers, helping them choose the appropriate constraint representation and composition strategy for a given problem. In practice, the compiler may exploit the structural properties of each type to optimize observation code—for instance, dimensional constraints can be evaluated via range-check circuits, while algebraic constraints may require dedicated arithmetic units. (See [*K.1*] for a detailed listing of axis types and [*K.2*] for relation categories.)

### 2.3.2 Composition

Field composition refers to the combination of two or more Fields to produce a new Field whose constraints and transition matrices are derived from the constituent Fields. Given Fields $F_1 = (C_1, T_1)$ and $F_2 = (C_2, T_2)$, a composition operator $\odot$ yields $F = F_1 \odot F_2 = (C, T)$ where $C$ and $T$ are defined according to the semantics of the composition.

Fields support three fundamental binary operations:

1. **Union** ($\cup$): $F = F_1 \cup F_2$ with $C(s) = C_1(s) \vee C_2(s)$ and $T(s_1, s_2) = \max(T_1(s_1, s_2), T_2(s_1, s_2))$. The union broadens the admissible set, allowing any Segment admissible in either Field. Observation under the union yields projections that satisfy at least one of the original constraints. Because the transition matrix entries can be interpreted as preference weights, taking the maximum corresponds to choosing the higher preference when either Field admits the transition.
2. **Intersection** ($\cap$): $F = F_1 \cap F_2$ with $C(s) = C_1(s) \wedge C_2(s)$ and $T(s_1, s_2) = \min(T_1(s_1, s_2), T_2(s_1, s_2))$. Intersection restricts admissibility to Segments that satisfy both constraints, producing a more constrained projection. Using the minimum of the preference weights reflects that a transition is only as strong as its weaker constituent.
3. **Product** ($\times$): $F = F_1 \times F_2$ where the resulting Field operates over the Cartesian product of Segment sets, with constraints and transitions defined componentwise: $C((s_1, s_2)) = C_1(s_1) \wedge C_2(s_2)$ and $T((s_1, s_2), (s_1', s_2')) = T_1(s_1, s_1') \cdot T_2(s_2, s_2')$. This product enables independent parallel observation of distinct subspaces—for example, when the two Fields govern disjoint coordinate axes (e.g., time and frequency). The overall projection becomes an ordered pair of the individual projections.

Implementing these operations efficiently requires careful handling of constraint representations. Union and intersection can be realized via Boolean satisfiability (SAT) solvers or constraint-propagation engines; the product operation, which expands the state space, may benefit from symbolic techniques (e.g., binary decision diagrams) to avoid exponential blow-up. In hardware, dedicated units for max/min and multiplication can accelerate transition-matrix computations.

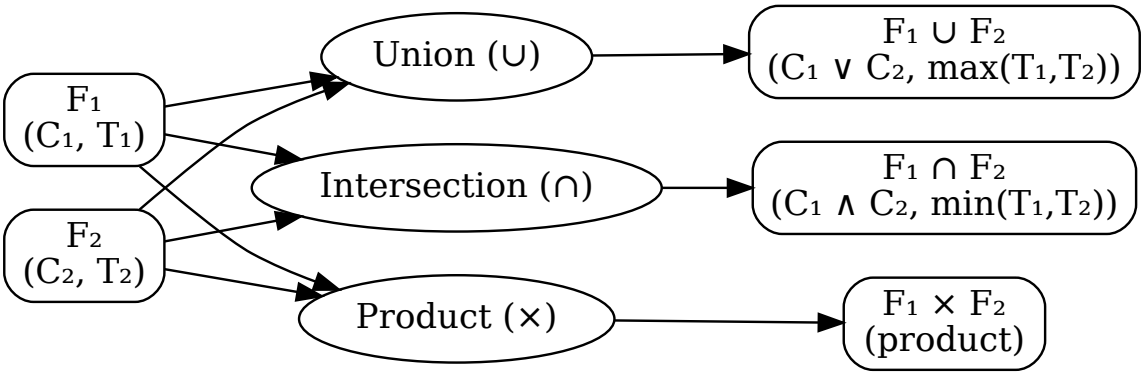



Figure 6: Field composition operations: union, intersection, and product

Beyond binary operations, Fields can also be composed in three higher-level ways: hierarchical, sequential, and parallel.

- **Hierarchical composition**: Fields can be nested to create hierarchical structures. A parent Field $F_{\text{parent}}$ may contain child Fields $F_{\text{child}}$ that govern subspaces of the coordinate space. The effective constraints are the conjunction of parent and child constraints, enabling modular refinement of governance. This nesting naturally aligns with modular hardware design (e.g., nested processing units) but adds depth to constraint evaluation, potentially increasing latency.
- **Parallel composition**: Independent Fields may observe the same Segment simultaneously. Because Segments are immutable, concurrent observations do not interfere. The resulting projections are independent and can be combined via product or other combinators. Parallel composition embodies the inherent concurrency of SSCCS: multiple observers can safely read the same coordinate space without synchronization.
- **Sequential composition (Extension)**: Fields can be applied in a temporal order: first $F_A$, then $F_B$. Because SSCCS treats time as one coordinate among many, sequential composition can be understood as applying Fields over adjacent time intervals. One interpretation models functional composition of constraint predicates: $C(s) = C_B(C_A(s))$ (where each predicate is viewed as a selector of admissible states) and convolution of transition matrices: $T(s_1, s_2) = \sum_{s'} T_A(s_1, s') \cdot T_B(s', s_2)$. This models processes where constraints evolve over time, such as multi-stage pipelines or state machines. Sequential composition requires ordering guarantees that can be enforced through hardware scheduling or explicit synchronization primitives. It can be presented **as an optional extension;** the core model does not depend on global sequentiality.

The immutability of Segments eliminates data-race hazards, making parallel composition naturally scalable across many cores. However, coordinating the distribution of Fields and collecting projections may introduce overhead. Hardware support for atomic broadcast of Segment addresses and gather-scatter of projection results can mitigate this overhead, enabling efficient many-observer systems.

### 2.3.3 Logical and Binary-Level Composition Protocols

Beyond purely logical combination, Field composition also serves as a communication primitive in the physical runtime. Fields can be encrypted, signed, or sandboxed at the binary level, enabling secure composition across trust boundaries. This dual nature—logical composition and physical communication unit—makes Fields a unifying abstraction that bridges the logical specification and the hardware-executable representation.

Fields can be grouped into logical units (for constraint management) and execution units (for hardware mapping), as illustrated in the diagram below. The recursive edge in the diagram shows that a Field can contain itself, enabling self-similar composition—a key property of the ribosome-like model where the same structural pattern repeats at different scales.

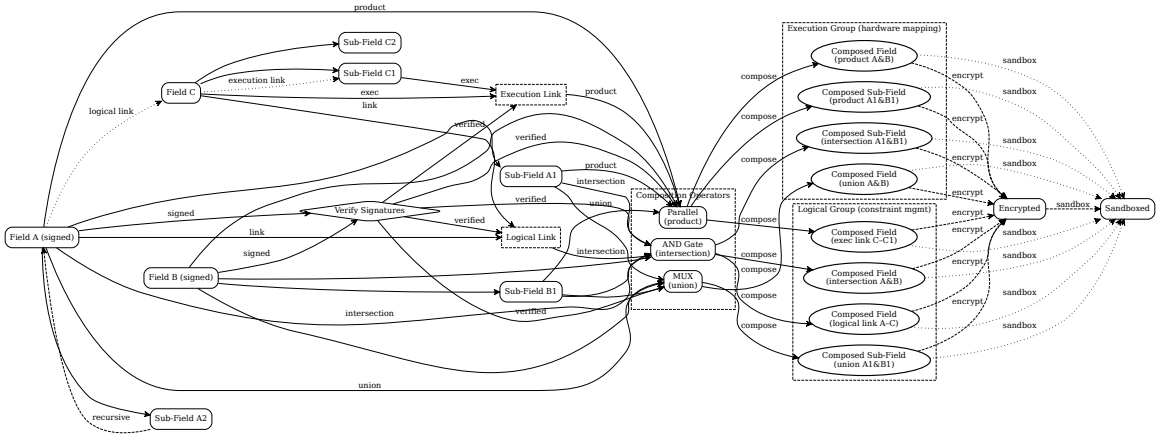



Figure 7: Minimal Field composition scenarios: nested/recursive groups, cryptographic hand-off, compact hardware mapping primitives.

In safety-critical and extreme-environment deployments (e.g., space computing, autonomous systems), Fields can be compiled into **cryptographically signed, sandboxed binaries**. This dual nature enables post-deployment policy updates, adaptive constraint reconfiguration, and verifiable execution without modifying underlying silicon. By treating fault-tolerance logic as dynamically composable Fields, SSCCS transcends static hardware hardening, allowing systems to evolve resilience strategies in response to changing environmental conditions or mission phases.

**Hand-off mechanisms**. Fields support secure composition across trust boundaries through cryptographic signing and encryption. A signed Field carries a digital signature that guarantees its integrity and origin; the runtime verifies the signature before allowing composition. An encrypted Field can only be observed by holders of the corresponding decryption key, enabling confidential computation. Sandboxed Fields execute within isolated hardware domains, preventing side-channel attacks and ensuring fault containment.

**Hardware mapping of composed Fields**. When Fields are composed, the compiler maps the resulting logical constraints onto physical hardware units. For example, a union of two Fields may be realized by a multiplexer that selects the admissible Segments from either constituent Field. Intersection can be implemented as a logical AND gate across constraint predicates. Product

composition maps to parallel execution units that operate on independent coordinate subspaces. The compiler generates hardware-specific control logic that respects the composition semantics while preserving determinism.

**Composition with cryptographic signatures**. For example, consider two Fields $F_A$ and $F_B$, each signed by different authorities. The runtime verifies both signatures before allowing their intersection $F_A \cap F_B$. This ensures that only authorized constraints are combined, a property crucial for multi-party secure computation.

The recursive edge in the diagram illustrates that Fields can contain themselves, enabling self-similar composition—a property reminiscent of biological ribosomes that translate code into structure and then reuse that structure as new code. This compositional recursion underpins SSCCS's ability to define adaptive, verifiable computation across scales, a capability that proves especially valuable in safety-critical and extreme environments. A detailed worked example of constraint intersection is provided in [*J*]; the corresponding framework for fault-tolerant computing in extreme environments is described in [*I*].

## 2.4 Observation and Projection

The observation operator $\Omega$ is the single active event that produces a projection from a Scheme and a Field:

$$P = \Omega(\Sigma, F).$$

Let $\mathcal{A}(\Sigma, F) = \{s \in \mathcal{S} \mid C(s) = \text{true}\}$ be the set of Segments admissible under the Field's constraints. The observation operator selects a projection $P$ according to the resolution strategy specified in $O$. For each admissible segment $s$, the projection $P(s)$ is given by a projector $\pi$ that encodes the semantic interpretation of the Field and Segment:

$$P(s) = \pi(s, F) \quad \text{for } s \in \mathcal{A}(\Sigma, F).$$

If the resolution strategy is deterministic, $\Omega$ is a function; if probabilistic, it samples from a distribution defined by the transition weights $T$. Observation occurs when the structure and Field together create an instability—i.e., multiple admissible configurations. $\Omega$ deterministically selects one configuration and returns it as $P$. No data is moved during observation; Segments remain in place.

## 2.5 Field Evolution and Observation Transition

Fields are mutable by design (see §3.3). This section extends mutability with explicit update rules that allow a Field to evolve over time or in response to events. When a Field updates, any previously computed projection becomes stale; a subsequent observation reflects the new Field state.

For systems that require strict reproducible execution, only deterministic triggers (e.g., fixed-rate temporal triggers or dependency chains) are permitted. Non-deterministic triggers (external events,

observed predicates that depend on runtime values) are allowed only when reproducibility is not required.

An update rule consists of a trigger and a transformation that modifies the Field's constraint predicate $C$ and transition matrix $T$:

$$F' = \text{update}(F, \text{trigger}, \text{context}).$$

The trigger types are summarised below; formal semantics and determinism properties are provided in [$M$].

| Trigger Type | Description | Deterministic |
|---|---|---|
| **Temporal** | After a duration along the Scheme's time axis (if present) | Yes (fixed rate) |
| **Event** | External signal (hardware interrupt, network message) | No (unless source is deterministic) |
| **Observed** | A projection satisfies a predicate immediately after observation | Depends on predicate |
| **Dependency** | Another Field updates | Yes (if order is fixed) |
| **Composite** | Logical combination (AND, OR, sequencing) | Depends on components |

When a Field that participates in a composition (union, intersection, product) is updated, the composite Field is implicitly updated according to the composition rule (e.g., $F_A \cap F_B$ becomes $F'_A \cap F_B$). The runtime tracks versions of constituent Fields and recomposes lazily.

Observation results are ephemeral (see §3.4). Caching with versioning is an implementation optimisation—it does not change semantics. The runtime ensures that each observation returns a result consistent with the current Field state; stale projections are transparently recomputed. The diagram below illustrates the dynamic observation cycle.

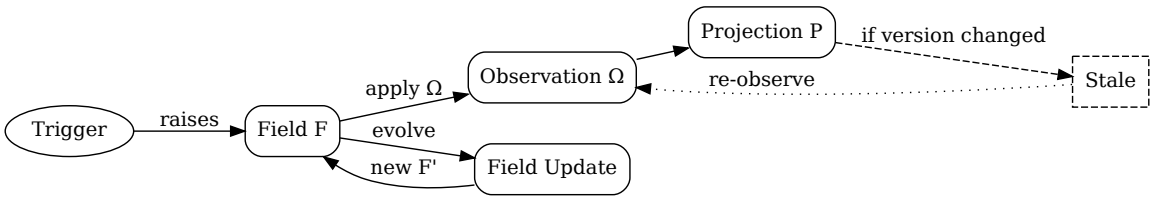



Figure 8: Dynamic observation cycle: a trigger initiates Field update, making previous projections stale

Examples:

- Temporal evolution, deterministic: A Scheme with a time axis. Field $F$ admits $x \in [0, 10]$. Rule: `after(10s)` expands the interval by 2. After 10 seconds, a new observation yields $x \in [0, 12]$.
- Event-driven, non-deterministic: Temperature sensor Scheme. Field $F_0$ admits values > 30 (alarm). An external fire-suppression event triggers update to $F_1$ (threshold > 50). A later observation gives a refined alarm set.

Detailed trigger semantics, versioning implementation notes, and compiler/runtime considerations are provided in [$M$].

## 2.6 Structural Isolation

Security properties emerge from the immutable structure rather than being added features. Since Segments cannot be modified, concurrent observations are naturally isolated. Formally, for any two disjoint sets of Segments $S_1$ and $S_2$,

$$\Omega(S_1 \cup S_2, F) = \Omega(S_1, F) \times \Omega(S_2, F),$$

where $\times$ denotes independent composition of projections. Every Segment and Scheme has a unique cryptographic hash, enabling identity-based boundaries where computations can only access Segments for which they hold valid references. This provides inherent structural isolation against interference without requiring additional security mechanisms.

## 2.7 Relationship with Traditional Concepts

| Traditional Concept | SSCCS Counterpart | Shift |
|---|---|---|
| Instruction fetch | Not applicable | No imperative control flow |
| Operand load | Segment coordinates | Stationary data move |
| Result store | Projection (ephemeral) | Results are events, not states |
| Cache line fill | Structural layout | Locality from geometry |
| Lock acquisition | Immutability | No shared mutable state |
| Program counter | Coordinate dimension | Time as coordinate |
| Algorithm | Geometry | Structure determines observation |

## 2.8 Formal Properties

### 2.8.1 Energy Model

A formal energy model for SSCCS can be expressed as:

$$E_{\text{total}} = E_{\text{obs}} \cdot N_{\text{obs}} + E_{\text{update}} \cdot N_{\text{update}},$$

where $E_{\text{obs}}$ is the energy required to perform a single observation, $E_{\text{update}}$ is the energy required to mutate the Field, $N_{\text{obs}}$ is the number of observations, and $N_{\text{update}}$ is the number of field updates. There is no term for moving data between memory and processor, because Segments are stationary. This model predicts that energy consumption scales with the number of observations and field updates, but not with data movement, which is a key source of energy inefficiency in traditional architectures [1]. Recent forecasts indicate that data center power consumption could double within four years [5], [6], with AI workloads driving much of this growth [7], [8].

### 2.8.2 Immutability and Concurrency

Because Segments are immutable, any number of observations can be performed simultaneously without interference. This property follows directly from the absence of shared mutable state: since Segments cannot be modified, observations on disjoint sets have no side-effects that could affect each other. Consequently, SSCCS enables inherent parallelism without any programmer effort or runtime synchronisation.

See the Structural Isolation section for a formal statement.

### 2.8.3 Time as a Coordinate

Time is treated as one coordinate axis among many. Let $t \in \mathbb{R}$ be a coordinate along the time axis; the Scheme may define $t$ as a continuous axis or as a cyclic axis with period $\tau$ (i.e., $t \equiv t \mod \tau$). Temporal ordering is expressed by comparing coordinates along that axis. Observations do not have a global temporal order unless explicitly defined. This eliminates the notion of a "program counter" and the associated assumption that computation must proceed in sequence.

### 2.8.4 Determinism and Auditability

Observation is deterministic: for identical $\Sigma$ and $F$, $\Omega$ always yields the same $P$. This follows from the definition of $\Omega$ as a function of $\Sigma$ and $F$; any non-determinism must be explicitly introduced through the resolution strategy in $O$. Determinism enables auditability as a secondary benefit: every projection is a verifiable trace from blueprint to output. However, this is a consequence of the core structural properties, not a primary design goal.

# 3 System Architecture and Compilation

## 3.1 Compiler: Topology Optimizer

A key engineering contribution of SSCCS is that the compiler, rather than generating a sequence of instructions, performs structural mapping of the Schema onto the target hardware. The compiler analyses the adjacency relations and memory layout semantics declared in the Schema written in the open format(`.ss`) [*F*] and produces a physical placement of Segments that maximises locality.

For example, if a Schema defines a two-dimensional grid of Segments with nearest-neighbour adjacency, the compiler can lay out those Segments in memory in row-major or column-major order such that adjacent Segments occupy adjacent cache lines. This is analogous to data layout optimisations performed manually in high-performance computing, but here it is automated and guaranteed by the Schema's specification.

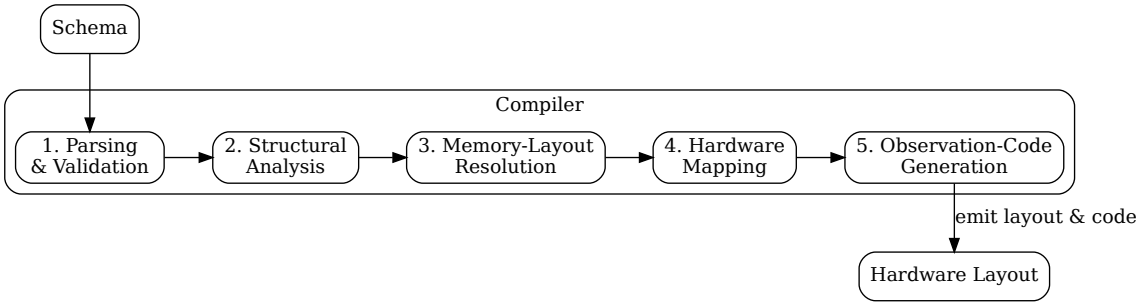



Figure 9: Compiler pipeline: from Schema to hardware layout

### 3.2 Compiler Pipeline

The SSCCS compiler transforms a high-level schema into a hardware-specific layout through a deterministic pipeline.

1. **Parsing and Validation**: The open-format `.ss` file (see [*F*]) is parsed into an intermediate representation (IR). The parser validates syntax and semantics, computes cryptographic identities (SchemaId, SegmentId), and produces a binary representation that preserves the topological structure.
2. **Structural Analysis**: The compiler extracts adjacency and dependency relations. It identifies independent sub-graphs that can be observed concurrently.
3. **Memory-Layout Resolution**: Using the Schema's `MemoryLayout` specification, the compiler resolves the mapping from coordinate space to logical addresses. This stage produces a logical address map that preserves locality.
4. **Hardware Mapping**: The logical address map is projected onto the target hardware's physical memory hierarchy. The compiler arranges Segments so that structurally adjacent Segments reside in physically proximate storage locations—e.g., the same cache line, adjacent memory banks, or custom address decoder regions. This strategy applies uniformly across conventional CPU/GPU targets and, where supported, emerging memory-centric architectures, but does not require any specific hardware capability.
5. **Observation-Code Generation**: For each sub-graph, the compiler emits native code that implements the observation operator Ω.

The entire pipeline is deterministic and reproducible: given the same specification and target hardware profile, the compiler always produces the same layout and observation code.

## 3.3 Structural Analysis

The second pipeline stage, **Structural Analysis**, extracts adjacency and dependency relations from the Scheme's relation set $R$. The compiler performs a graph-theoretic analysis to identify **independent sub-graphs** that can be observed concurrently.

Concretely, the compiler constructs a directed graph where vertices are Segments and edges are relations of type *Dependency*. **Strongly connected components (SCCs)** of this dependency graph are identified; each SCC forms a candidate for sequential observation because dependencies within a component create cyclic constraints. Sub-graphs with no edges between them are marked as **independent** and can be scheduled in parallel. For relations of type *Adjacency* and *Hierarchy*, the compiler uses them to guide locality optimization in later stages, but they do not impose observation ordering.

This analysis is purely structural and does not depend on runtime values, ensuring deterministic parallelism. The output of this stage is a partition of the Scheme into observation units, each with its own logical-address map and observation code.

## 3.4 Hardware Topology Embedding

### 3.4.1 Logical Address Virtualization Layer

The compiler's ability to eliminate data movement hinges on the `MemoryLayout` abstraction. A `MemoryLayout` consists of a `layout_type` (e.g., `RowMajor`, `SpaceFillingCurve`), a `mapping` function, and `metadata`.

A logical address is an intermediate representation: a segment identifier and an offset within that segment's conceptual address space. It is not a physical address; rather, it serves as an intermediate coordinate that the hardware mapper later translates to concrete physical locations.

Example: For a 2D grid with row-major layout:

$$f(x, y) = (\text{grid\_id},\ y \cdot \text{width} + x)$$

The compiler evaluates this function for every coordinate in the Schema, producing a complete logical-address map. This mapping preserves row-wise adjacency: for a $3 \times 3$ grid, the logical addresses are $(0, 0) \to 0$, $(1, 0) \to 1$, $(2, 0) \to 2$, $(0, 1) \to 3$, etc.

Other layout types are handled analogously. For a **column-major** layout the mapping becomes $f(x, y) = x \cdot \text{height} + y$; for a **space-filling curve** (e.g., Z-order) the compiler interleaves the bits of the coordinates. **Hierarchical** layouts recursively apply a base layout within each block, enabling multi-level locality. **Graph-based** layouts assign logical addresses according to a graph-traversal order (e.g., breadth-first search) that respects the adjacency relations declared in the Scheme.

Each layout type is defined in the Schema's `MemoryLayout` specification (see [*K.3*]). The compiler evaluates the corresponding mapping function for every coordinate, producing a deterministic logical-address map that is then passed to the hardware-mapping stage.

### 3.4.2 Target-Hardware Mapping Strategies

The logical address space acts as a virtualization layer, decoupling structural description from physical implementation. The same Schema can be embedded into vastly different hardware substrates while preserving the core principle: structurally adjacent Segments reside in physically proximate storage locations.

Rather than prescribing a fixed hardware target, the compiler adapts the logical-address map to the memory hierarchy and parallelism available on the underlying substrate. On cache-based CPUs, the compiler groups adjacent Segments into cache-line-aligned blocks, ensuring that a single fetch retrieves all data needed for a local observation. For architectures with explicit spatial parallelism (e.g., reconfigurable fabrics), the mapping can be hardwired into address-decode logic, turning observation into a combinatorial propagation. High-bandwidth memory systems allow Segments to be distributed across independent channels, overlapping accesses to maximize throughput. Emerging non-volatile memories can be treated as a static coordinate manifold, enabling direct structural projection without address translation.

Where hardware capabilities allow (e.g., processing-in-memory or other memory-centric computing), the compiler may offload entire observation units to the memory substrate, translating the logical map into device-specific commands that perform observation without moving data. Crucially, even on conventional von Neumann hardware, SSCCS **overlays a structural interpretation**: the compiler translates logical addresses into standard load/store operations, but the overall computation remains free of data movement because all necessary data is already resident where observation occurs. A detailed discussion of hardware profile variants and mapping strategies is provided in [*H*].

## 3.5 Observation-Code Generation

The fifth pipeline stage emits executable code that implements the observation operator $\Omega$ for each independent sub-graph identified during structural analysis. The form of the generated code adapts to the target hardware's capabilities:

- **Conventional CPUs**: The compiler emits SIMD loops that iterate over the logical-address ranges produced by the layout stage. Each iteration evaluates the projector $\pi$ for a Segment and accumulates the result into a projection buffer. The loops are automatically vectorised and unrolled according to the hardware's SIMD width.
- **Reconfigurable fabrics**: For architectures that support spatial computation, the compiler can generate a netlist that hardwires the address mapping into the interconnect fabric. The observation operator becomes a combinatorial propagation, eliminating fetch-execute overhead.
- **Processing-in-memory substrates**: Where memory-centric compute units are available, the compiler produces a sequence of its commands that are sent directly to the memory controller. Each command triggers an observation within a memory bank, removing data movement between memory and processing unit.

The generated code is deterministic and reproducible: given the same Scheme and hardware profile, the compiler emits identical observation code every time. This property enables ahead-of-time verification and eliminates runtime compilation overhead. A detailed methodology for observation-code generation is provided in [$G$].

### 3.6 System Stack and Runtime

SSCCS inserts a runtime layer between application and hardware that translates observation requests into hardware-specific memory mappings.

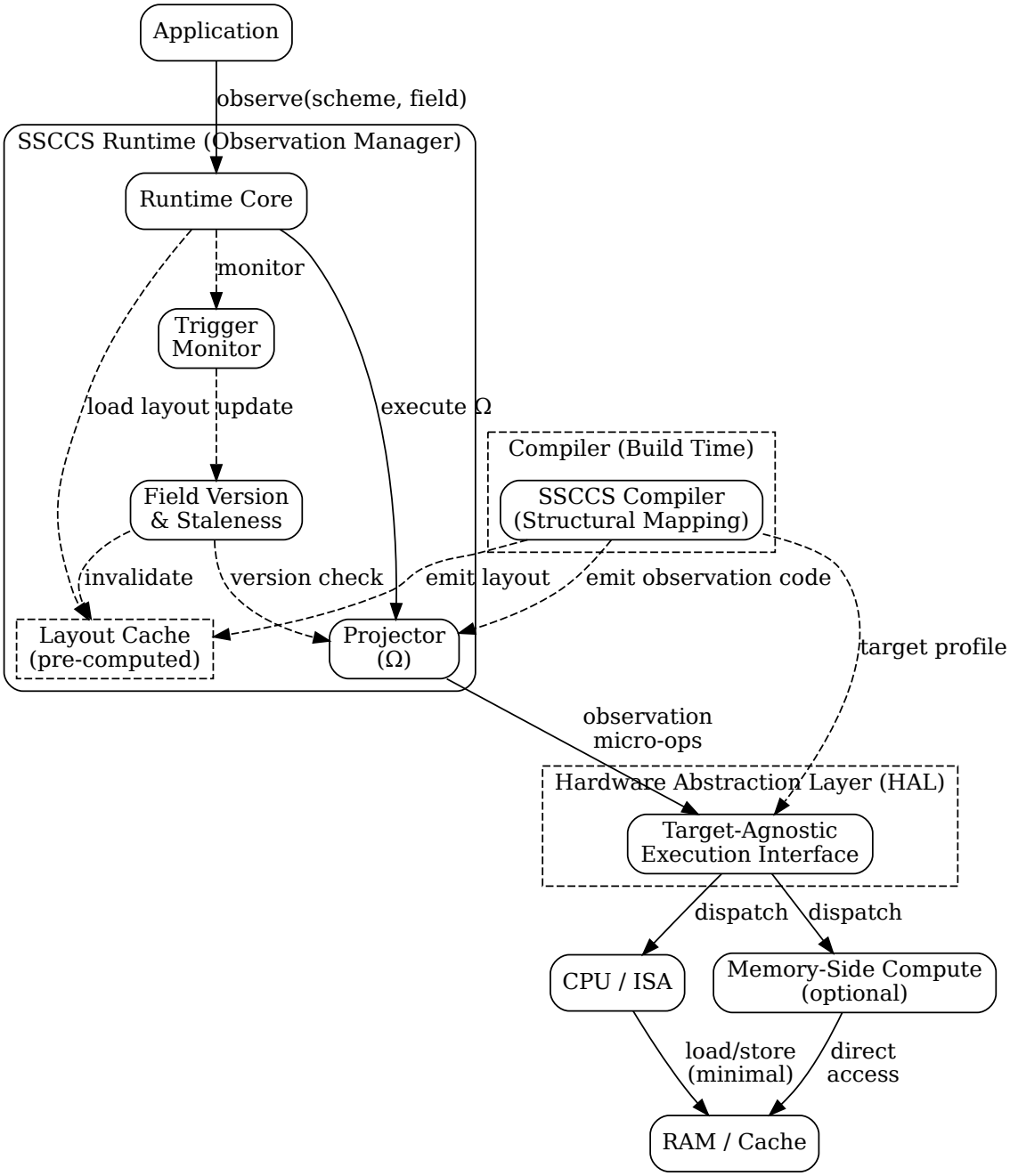



Figure 10: SSCCS system stack

Upon receiving an `observe(scheme, field)` call, the runtime loads the pre-computed memory layout from the layout cache, invokes the projector, and returns the resulting projection. The projector translates the logical addresses produced by the compiler into hardware-specific memory accesses, using dedicated observation instructions when available. This decouples the structural mapping (done at compile time) from the dynamic observation (done at runtime), enabling portable performance across heterogeneous hardware.

### 3.6.1 Future Hardware Considerations

While SSCCS can be implemented in software, its benefits are most pronounced with hardware support:

- No instruction fetch unit; observation triggered structurally.
- Direct observation support in the memory hierarchy.
- Spatial computation mapping adjacency to wiring.
- **Open RISC-V Synergy**: Concrete validation pathways are being explored via the RISC-V ecosystem [9], particularly through OpenHW CORE-V's eXtension Interface (XIF) [10]. Custom observation primitives (`OBSERVE`, `COLLAPSE`) can be implemented as tightly coupled coprocessor extensions, enabling deterministic structural projection with minimal pipeline interference and formal verifiability.
- **OpenHW Ecosystem**: The OpenHW CORE-V MCU [11] and SDK provide ready-to-use platforms for prototyping SSCCS observation units, while the Imperas simulation environment enables pre-silicon verification. The CVA6 core and other OpenHW MCU variants offer scalable performance targets. Integration proposals outline concrete steps for mapping SSCCS primitives to RISC-V extensions.
- **Edge and Peripheral Integration**: Peripheral integration with eFPGA fabrics [12] and always-on vision sensors demonstrates the versatility of the RISC-V ecosystem for edge deployment, while formal verification tools [13] and coprocessor examples [14] ensure correctness and extensibility. Efficient I/O for large-scale machine learning [15] further highlights the importance of minimizing data movement.
- **Space-Grade Resilience**: Radiation-hardened RISC-V implementations such as HARV-SoC and StarRISC, together with ESA's TRISTAN roadmap [16], demonstrate the industry shift toward deterministic, mixed-criticality aware processors. Recent advances in multi-bit fault-tolerant instruction decoding [17], dynamic lockstep redundancy [18], and high-performance space computing platforms [19], [20] further illustrate the RISC-V community's focus on resilience. SSCCS complements these efforts by providing a software-defined resilience layer that can be deployed as cryptographically signed Fields. [21], [22].

## 3.7 Implementation Cases

The appendix examples below demonstrate how the compiler pipeline (parsing, structural analysis, memory-layout resolution, hardware mapping, and observation-code generation) translates high-level specifications into hardware-specific layouts.

- **Vector Addition Example**: A concrete walkthrough of vector addition in SSCCS, demonstrating zero data movement and implicit parallelism. [*C*]
- **Scaling to N-Dimensional Tensors**: Extension of principles to higher-dimensional structures, featuring zero-copy reshaping and logical adjacency. [*D*]
- **Complex Graph Processing**: Application of graph algorithms, eliminating pointer chasing through parallel observation. [*E*]

# 4 Theoretical Performance & Scalability

The SSCCS architecture derives its efficiency not from incremental hardware acceleration, but from a fundamental shift in computational complexity.

## 4.1 Architectural Expectations of Time-Space Complexity

Traditional procedural models are constrained by the linear relationship between data volume ($N$) and execution cycles. SSCCS decouples this relationship by utilizing the concurrent propagation of a Field across a pre-defined Topology.

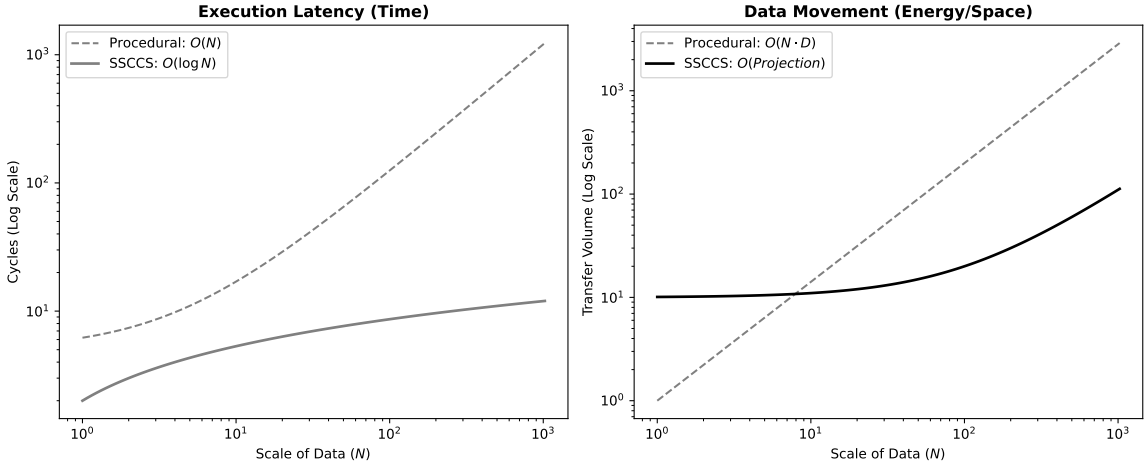



Figure 11: Asymptotic Complexity: Procedural vs. SSCCS Structural Observation

### 4.1.1 Temporal Complexity (Latency)

In a von Neumann environment, latency scales at $O(N)$ or $O(N/k)$ due to instruction dispatch and synchronization.

- **SSCCS Latency**: Defined by the physical propagation delay of the Field across the Scheme. The observation of the result theoretically approaches near-$O(1)$ in emerging hardware paradigms (e.g. Processing-In-Memory (PIM)).

### 4.1.2 Data Movement Complexity (Spatial/Energy Cost)

The primary energy sink in modern computing is the movement of operands from memory to logic units.

- **Procedural Cost**: $O(N \cdot D)$.
- **SSCCS Cost (Logic-at-Rest)**: $O(Projection)$. Since the input Segments remain stationary, the energy expenditure is strictly limited to the transmission of the resulting Projection.

### 4.2 Comparative Complexity Matrix

| Metric | Sequential | Parallel (SIMD/GPU) | SSCCS (Structural) |
|---|---|---|---|
| Instruction Overhead | High ($O(N)$) | Moderate ($O(N/k)$) | Minimal (Field-based) |
| Data Locality | Managed (Cache) | Explicit (SRAM/Tiling) | Intrinsic (Scheme-defined) |
| Execution Latency | $O(N)$ | $O(N/k)$ + sync | $O(\log N)$ or $O(1)$ |
| Data Movement | $O(N)$ | $O(N)$ | $O$(Output Only) |
| Scalability Limit | Amdahl's Law | Memory Bandwidth | Physical Propagation Delay |

An observation, however structured, must execute on a physical substrate. The compiler's layout decisions carry a bandwidth-compute balance constraint: for any observation unit, the product of its compute footprint and the required data rate must not exceed what the target substrate can supply per cycle. When this condition is unmet, the observation stalls regardless of structural independence. The `MemoryLayout` abstraction is therefore not merely an optimisation—it is a constraint-satisfaction problem whose solution determines whether an observation is executable on a given hardware profile. Empirical analyses of this balance condition in conventional vector architectures are referenced in [$N$].

### 4.3 Scalability in High-Dimensional AI Workloads

As demonstrated in the emergence of State-Space Models (SSMs) [23] and manifold-constrained learning [24], the ability to process high-dimensional representations without exhaustive data shuffling is critical.

1. **Stationary Topology**: By fixing the Segments in a k-dimensional `MemoryLayout`, SSCCS allows the hardware to perform "Observation" as a near-instantaneous mapping.
2. **Implicit Parallelism**: Unlike threads or warps that require explicit management, SSCCS parallelism is implicit—it is a property of the structure itself.

## 5 Related Work

Although SSCCS was developed without direct reference to prior work, its theoretical core reveals meaningful parallels with several established research domains:

- **Dataflow architectures** treat programs as graphs where nodes fire when inputs are available.
- **Functional programming** emphasizes immutability and referential transparency.
- **Processing-in-memory (PIM)** research addresses the data movement problem within the von Neumann paradigm [*G.5*; [25]; [26]; [27]].

- **Safety and Accountability**: Real-world incidents illustrate the need for deterministic reproducibility; for example, autonomous vehicle collisions [28], financial flash crashes [29], [30], and diagnostic errors [31] highlight the consequences of non-deterministic AI behavior. Recent analyses indicate that AI-related incidents have risen by more than 50% in the last year alone, with total damages exceeding $100 billion [32]. SSCCS's structural determinism offers a foundation for audit-by-design systems.

Recent work in AI demonstrates the growing relevance of structural constraints:

- **Geometric Constraints**: Research such as *Manifold-Constrained Hyper-Connections* by DeepSeek [24] highlights the efficacy of applying geometric inductive biases. This supports the SSCCS approach of defining computational processes through topological constraints.
- **Structural Parallels**: SSCCS shares conceptual ground with State-Space Models (SSMs) like Mamba [23]. While these systems achieve high-performance linear recurrences through ad-hoc kernel tuning, SSCCS redefines the recurrence not as a procedural loop, but as a one-dimensional Scheme of adjacent Segments. By shifting from execution-based optimization to the deterministic observation of stationary topological constraints, SSCCS offers a universal, structure-defined architecture.

These references contextualize SSCCS within the broader intellectual landscape. In each domain, the shift from execution to observation is expected to offer advantages that incremental optimization cannot provide.

Recent ecosystem developments in open-hardware space computing further contextualize SSCCS's relevance. Initiatives such as ESA's TRISTAN roadmap and radiation-tolerant RISC-V implementations (e.g., StarRISC, HARV-SoC) highlight the industry's shift toward deterministic, mixed-criticality aware processors. SSCCS aligns with this trajectory by providing a software-defined resilience layer that complements physical hardening, enabling semantic fault detection, adaptive redundancy, and cryptographic policy enforcement on open silicon.



## 6 Development Roadmap and milestones

The development roadmap [*A*] follows a three-phase progression, beginning with Rust-based software emulation (drawing on established bare-metal Rust practices [33], [34]) and proof-of-concept implementations [*B*], and culminating in native observation-centric hardware. A dual-layer compiler architecture bridges the transition from existing von Neumann systems, enabling incremental migration. This strategy prioritizes empirical validation in high-performance domains, demonstrating energy efficiency and deterministic execution through structural observation. For details, refer to the appendices tagged above.

## 7 Conclusion and Future Work

SSCCS establishes five foundational principles:

1. Computation concerns revelation rather than change.

2. Structure is more fundamental than process.
3. Time is a coordinate rather than a flow.
4. Value is projected rather than intrinsic.
5. Immutability enables parallelism and verifiability.

The most significant departure is the treatment of time as one coordinate among many, eliminating global sequentiality and enabling lock-free concurrency. The compiler's role correspondingly shifts from instruction scheduling to topological optimization—mapping logical adjacency directly onto physical locality.

Open questions remain: formal treatment of nested Field dynamics, compiler infrastructure for geometric constraints at scale, and empirical validation of energy efficiency gains across target domains. Beyond engineering challenges, SSCCS invites a broader reconsideration of what computing is. If computation can be structured as the revelation of static potential rather than the execution of mutable instructions, then many assumptions about hardware design, programming languages, and system architecture become contingent rather than necessary. The open .ss format [*F*] is a first step toward making these ideas concrete and composable.

SSCCS is not proposed as a universal replacement for all computing. For problems inherently sequential or interaction-dominant, traditional models may remain appropriate. But for the growing class of data-intensive, parallel workloads where the von Neumann bottleneck dominates, this model offers a fundamentally different trade-off: one where verifiability, parallelism, and energy efficiency are not features to be added, but consequences of how computation is defined.

# Appendices

# A Project Roadmap

SSCCS is designed for incremental adoption—start with software emulation today, transition to hardware acceleration as the technology matures, and ultimately deploy on native observation-centric processors. The open format ensures that investment in specification outlives any particular implementation.

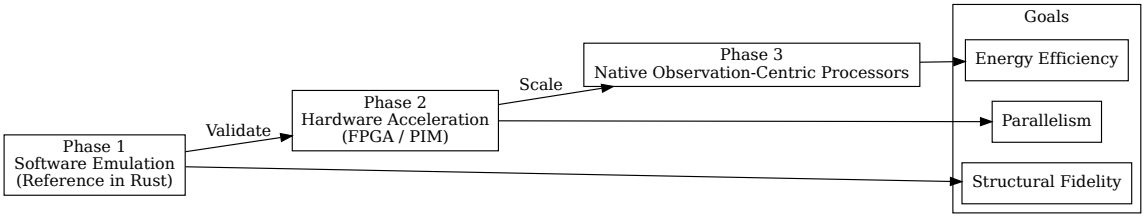



Figure 12: Implementation roadmap: three research phases

## A.1 Implementation Phases

### A.1.1 Phase 1: Software Emulation (Proof of Concept)

- Rust reference implementation reading the SS format specifications.
- Validate model on small benchmarks.
- Measure determinism, implicit parallelism, data movement reduction.

### A.1.2 Phase 2: Hardware Acceleration (Transitional)

- Map Schemes to FPGA fabrics.
- Utilize PIM architectures as transitional substrate (UPMEM, Samsung FIM).
- Develop compiler targeting CPUs (via SIMD) and FPGA/PIM.
- Begin formal verification.

### A.1.3 Phase 3: Native Observation-Centric Processors (Long-Term Research)

- Design processor directly instantiating Schemes.
- Integrate memory and logic in unified substrate (e.g., memristor arrays).
- Evaluate energy efficiency for target domains.

## A.2 Compiler Layer as Migration Bridge

The SSCCS compiler serves a dual purpose:

1. **Adaptive Embedding (Phase 1-2):** Translate Schemes into von Neumann-compatible code (load/store, SIMD) or PIM primitives, accepting abstraction overhead.

   *Example:* A climate model grid can be compiled to standard C + OpenMP today, while retaining the same format specification for future hardware.

2. **Direct Instantiation (Phase 3):** Map Schemes directly to observation-centric hardware primitives, eliminating compatibility layers.

   *Example:* The same scheme grid can later be synthesized directly onto a memristor array without rewriting.

This dual capability enables gradual migration without requiring a "flag day" switchover. Organizations can adopt SSCCS incrementally, deploying on existing infrastructure while preparing for native hardware.

## A.3 Domain Validations

SSCCS is intended for validation across multiple domains.

| Domain | Traditional Challenge | Expected Advantages |
|---|---|---|
| Climate modelling | Massive state space, grid data movement | Constraint isolation, minimal data transfer |
| Space systems | Radiation-induced errors, power constraints | Adaptive fault tolerance, semantic validation, cryptographic policy updates |
| Protein folding | Combinatorial explosion, long time scales | Massive parallel observation |
| Swarm robotics | Coordination overhead, limited communication | Recursive composition, emergent coordination |
| Financial modelling | Real-time constraints, complex dependencies | Deterministic projections, no race conditions |
| Cryptographic systems | Side-channel attacks, verification complexity | Immutable structure enables formal verification |
| Autonomous vehicles | Sensor fusion, real-time decision making | Constraint-based observation, deterministic response |

# B PoC Implementation Notes

The SSCCS Proof of Concept (PoC) is a Rust-based reference implementation that validates the core ontological layers of the model. It serves as the software-emulation phase (Phase 1) of the SSCCS roadmap, demonstrating that the abstract primitives—Segment, Scheme, Field, Observation, and Projection—can be realized as concrete, executable code. This appendix documents key implementation details that complement the open-format specification [$F$] and illustrate how the theoretical model translates into a working system.

## B.1 Why Rust

Rust's design philosophy aligns closely with the SSCCS model: immutability by default, zero-cost abstractions, concurrency without data races, cryptographic primitives, a strong type system for structural invariants, and fine-grained control over memory layout. These features make Rust the natural language for prototyping a system where deterministic observation, structural safety, and performance are paramount [33], [34].

## B.2 Core Types

### B.2.1 Segment

A `Segment` is an immutable coordinate point in a multi-dimensional possibility space. Its identity is a BLAKE3 cryptographic hash derived from its coordinates, guaranteeing uniqueness and verifiability.

```
pub struct Segment {
    coordinates: SpaceCoordinates,
    id: SegmentId,
}
```

## B.3 Scheme Abstraction Layer

The Scheme abstraction captures the static topological blueprint of a computation. It consists of:

- **Axes** – dimensional definitions (`Discrete`, `Continuous`, `Cyclic`, `Categorical`, `Relational`, `WithUnit`).
- **Structural Relations** – adjacency, hierarchy, dependency, equivalence, and custom predicates.
- **Memory-Layout** – mapping from coordinate space to logical addresses (`Linear`, `RowMajor`, `ColumnMajor`, `SpaceFillingCurve`, `Hierarchical`, `Graph-Based`, `Custom`).
- **Observation Rules** – resolution strategies, triggers, priority, and context.

The layer provides ready-to-use templates (`Grid2DTemplate`, `IntegerLineTemplate`, `GraphTemplate`) that combine these elements into common topological patterns.

### B.3.1 Field

A `Field` holds dynamic constraints (`ConstraintSet`) and a `TransitionMatrix` that encodes weighted directed-graph relationships between Segments. The constraints determine which Segments are admissible; the transition matrix influences how observation propagates across the topology.

### B.3.2 Projector Trait

The `Projector` trait defines the semantic interpretation of a Segment–Field pair, producing a projection (the observable "value"). It also optionally specifies possible next coordinates, enabling adjacency to be defined through the projector itself.

```
pub trait Projector {
    type Output;
    fn project(&self, field: &Field, segment: &Segment) -> Option<Self::Output>;
    fn possible_next_coordinates(
      &self,
      coords: &SpaceCoordinates
    ) -> Vec<SpaceCoordinates> { vec![] }
}
```

Three example projectors are:

- **`IntegerProjector`** – extracts a coordinate along a given axis.
- **`ArithmeticProjector`** – defines adjacency through arithmetic operations (`+1`, `-1`, `*2`, `/2`).
- **`ParityProjector`** – classifies coordinates as `"even"` or `"odd"` strings.

## B.4 Compiler Pipeline

A four-stage compiler pipeline translates a high-level Scheme into hardware-specific layouts:

1. **Parsing and Validation** – reads the `.ss` binary and builds an intermediate representation.
2. **Structural Analysis** – extracts adjacency and dependency relations, identifies independent sub-graphs for parallel observation.
3. **Memory-Layout Resolution** – applies the Scheme's `MemoryLayout` to map coordinates to logical addresses.
4. **Hardware Mapping** – projects logical addresses onto the target hardware's physical memory hierarchy (CPU caches, FPGA Block-RAM, PIM banks).

The pipeline is generic over a `HardwareProfile` (`GenericCPU`, `FPGA`, `PIM`, `Custom`), allowing the same Scheme to be optimized for different substrates.

# C Vector Addition Example

Consider the addition of two vectors of length $N$.

## C.1 Traditional Approach

In a traditional architecture, a loop iterates over indices, loading each element from memory into registers, performing the addition, and storing the result back.

```
fn add_vectors(a: &[f64], b: &[f64]) -> Vec<f64> {
    assert_eq!(a.len(), b.len());
    let mut result = Vec::with_capacity(a.len());
    for i in 0..a.len() {
        result.push(a[i] + b[i]); // loads a[i], b[i]; stores result[i]
    }
    result
}
```

- **Data Movement**: $2N$ loads + $N$ stores = $3N$ total memory transfers.
- **Sequential Dependency**: Loop-carried dependencies limit parallelisation.
- **Cache Behaviour**: Performance is highly dependent on memory layout; random access or misalignment causes cache misses.

## C.2 SSCCS Approach

A Scheme defines a set of Segments representing the vectors. The compiler lays out the Segments consecutively in memory. An observation of the entire structure yields a projection that is the sum vector.

```
let a = Segment::vector(0..N, initial_values);
let b = Segment::vector(0..N, initial_values);
let scheme = Scheme::add_vectors(a, b);
let field = Field::new();
let sum = observe(scheme, field);
```

- **Data Movement**: Zero input movement. Segments remain stationary ("Logic-at-Rest"). Only the resulting projection (a single vector of length $N$) is transmitted.
- **Parallelism**: Structural independence allows all element pairs to be observed concurrently without explicit synchronisation or partitioning.
- **Locality**: Enforced by the compiler's topological mapping, treating memory as an active topology rather than passive storage.

### C.2.1 Comparison Table

| Aspect | Traditional (Procedural) | SSCCS (Structural) |
|---|---|---|
| Input Data Movement | $2N$ loads | Zero (Stationary Segments) |
| Output Data Movement | $N$ stores | $N$ (Projection) |
| Concurrency | Requires explicit parallelisation | Implicit (Structural independence) |
| Synchronisation | Locks/atomics for shared state | None (Immutability guaranteed) |
| Memory Role | Passive storage | Active topology |
| Auditability | Requires external tracing | Intrinsic to Specification |

# D Scaling to N-Dimensional Tensors and Graphs

The structural principles of SSCCS extend beyond linear vectors to higher-dimensional and non-linear data structures.

## D.1 N-Dimensional Tensors

In SSCCS, an $N$-dimensional tensor is represented as a set of Segments where adjacency relations are defined across multiple axes within the Scheme.

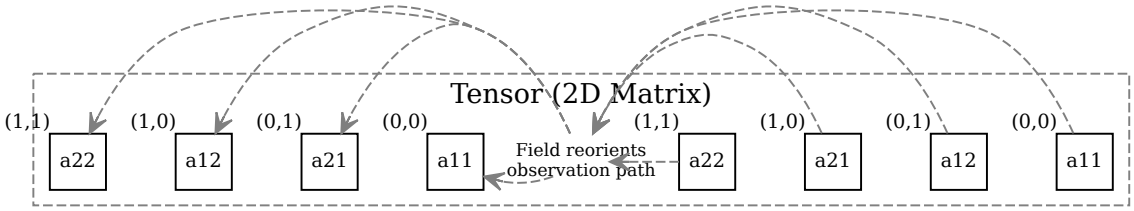



Figure 13: SSCCS applied to a 2D tensor

- **Zero-Copy Reshaping**: Traditional systems require physical data movement ($O(N)$ or $O(N^2)$) to perform operations like transposition or reshaping. In SSCCS, reshaping is a metadata-only operation. By reorienting the Field's observation path over stationary Segments, the dimensionality of the Projection changes without moving a single bit in memory ($O(1)$).
- **Logical Adjacency**: For operations like matrix multiplication, the compiler maps Segments to ensure that the required operands for a specific Field are physically co-located. This transforms what would be complex indexing logic in a CPU into a direct physical property of the memory topology.

# E Complex Graph Processing

Graph algorithms (e.g., PageRank, GNNs) are traditionally bottlenecked by "Pointer Chasing," which causes severe cache thrashing and memory latency.

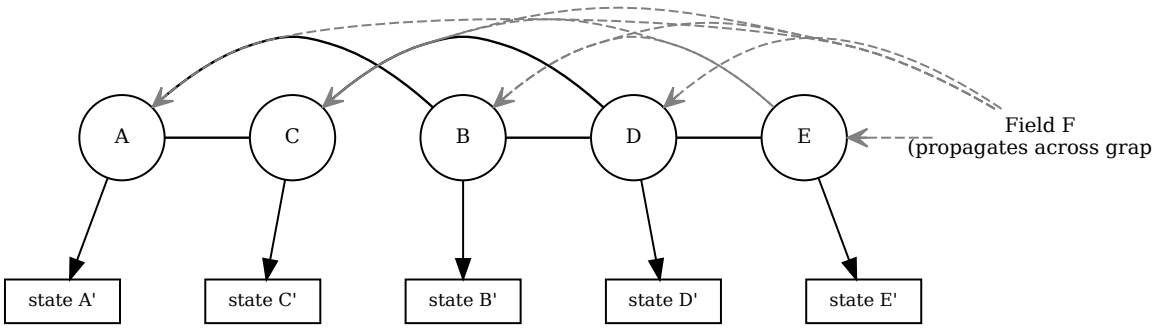



Figure 14: SSCCS applied to graph: All nodes observed in parallel (no iteration)

- **Segment-as-Node**: Each node and its properties are encapsulated in a Segment.
- **Adjacency-as-Structure**: Edges are defined as structural constraints within the Scheme, not as memory pointers to be followed sequentially.
- **Field-based Traversal**: A Field propagates across the entire Scheme in a single observation cycle. Instead of "visiting" nodes, the observer captures the emergent state of the entire graph simultaneously.
- **Concurrency**: This eliminates vertex-centric synchronization (locks/mutexes). All nodes update their state in parallel as a deterministic consequence of the Field's interaction with the Scheme's topology.

## E.1 Comparison: Computational Density at Scale

| Computational Task | Traditional Bottleneck | SSCCS Solution |
|---|---|---|
| Tensor Reshaping | Physical data reshuffling ($O(N^d)$) | Metadata-level Field reorientation ($O(1)$) |
| Matrix Contraction | Memory bandwidth & indexing overhead | Hardwired adjacency in the Scheme |
| Graph Traversal | High latency due to random access | Distributed parallel observation |
| Sparse Operations | Complex indexing & storage overhead | Non-linear Scheme mapping (skipping null-space) |

The scaling of SSCCS addresses the Curse of Dimensionality by decoupling the logical structure of data from the physical cost of its traversal. While traditional architectures expend energy moving data to accommodate logic, SSCCS modifies the Field to accommodate the stationary structure.

# F Open Format Specification (Draft)

The `.ss` format is a declarative language for specifying the **topological structure** of an SSCCS computation. It describes the geometric and relational properties of a computational space, leaving all physical mapping decisions to the hardware-specific compiler backend. The format captures *what* the structure is; not *how* to execute it.

## F.1 Core Components

The following diagram illustrates how the four core components of a `.ss` description relate to each other. Notice that no memory layout or instruction flow appears – only the static topology of the computation.

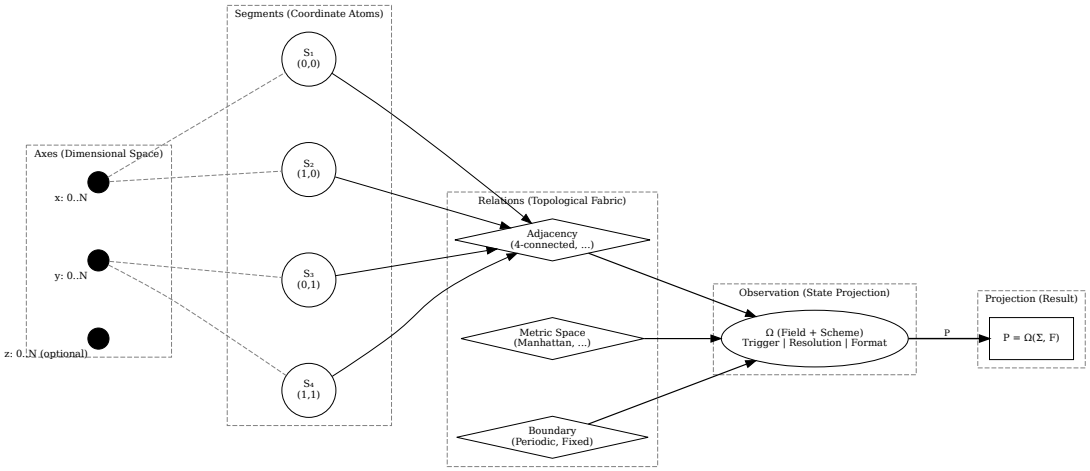



Figure 15: Topological Schema: Four core components defining computational geometry

## F.2 Component Details

### F.2.1 Axes

Define the coordinate space: names, ranges (discrete or continuous), and optional manifold properties (e.g., toroidal, bounded). They establish the dimensional foundation upon which Segments are placed.

### F.2.2 Segments

Atomic coordinate points. Each Segment is identified by its coordinates $(x, y, z, \dots)$ and a cryptographic hash derived from them. Segments are immutable and stateless – they carry no values, only position.

### F.2.3 Relations

Relations replace traditional control flow (loops, conditionals) with a static description of connectivity and proximity. The topological fabric that connects Segments. This block encodes:

- **Adjacency**: How Segments are connected (e.g., 4-connected, 8-connected, arbitrary graph).
- **Metric Space**: The distance function $d(s_i, s_j)$ that governs interaction strength (e.g., Manhattan, Euclidean, graph distance).
- **Boundary Conditions**: The shape of the manifold (e.g., `Periodic` for a torus, `Fixed` for a finite grid).

### F.2.4 Observation

Defines how a Field's dynamic constraints interact with the static Scheme to produce a Projection. It includes:

- **Trigger**: The condition that initiates observation (e.g., `Equilibrium`, `ExternalPulse`, `Timer`).
- **Resolution Strategy**: How multi-segment interactions are resolved into a single output (e.g., summation, maximum, tensor contraction).
- **Projection Format**: The mathematical type of the result (e.g., scalar, vector, tensor).

## F.3 Key characteristics

1. **Non-linear addressing** – Segments are identified by coordinate tuples, not memory offsets.
2. **Relation-defined computation** – What traditional code expresses as loops and conditionals is encoded in the connectivity and metric of the `Relations` block.
3. **Observation as collapse** – The `Observation` block specifies how a Field's constraints resolve the Scheme's potential into a deterministic Projection.
4. **Deferred physical mapping** – No memory layout or instruction sequence is included; the compiler backend maps the topology to concrete hardware (SRAM, DRAM, HBM, etc.) based solely on the declared relations.

## F.4 Cryptographic Identity

A Schema's identity is derived solely from its topological properties. Changing a physical implementation detail (e.g., cache-line alignment) does **not** affect the `SchemeId`. However, altering the connectivity or the metric space produces a new, distinct identity:

$$SchemeId = H(\text{Axes} + \text{Segments} + \text{Relations} + \text{ObservationRules})$$

# G Observation-Code Generation Methodology

This appendix expands on the observation-code generation stage described in Section 3.5 of the whitepaper, providing concrete implementation techniques for CPU, FPGA, and PIM targets. It illustrates how the SSCCS compiler lowers the high-level structural description to executable native code while preserving determinism and reproducibility.

## G.1 Lowering to LLVM/MLIR

The SSCCS front-end translates a Scheme and its associated Field into an intermediate representation (IR) expressed in a custom MLIR dialect named `ssccs`. This dialect captures the topological relationships, coordinate mapping, and observation semantics as MLIR operations. The following snippet illustrates a simplified `ssccs.observation` operation:

```
func.func @observe_vector(
  %scheme: !ssccs.scheme,
  %field: !ssccs.field
) -> tensor<8xf32> {
  %result = ssccs.observation %scheme, %field
            {layout = #ssccs.layout<row_major>,
             projector = #ssccs.projector<add>}
            : (!ssccs.scheme, !ssccs.field) -> tensor<8xf32>
  return %result : tensor<8xf32>
}
```

The `ssccs` dialect is then lowered to standard MLIR dialects (e.g., `affine`, `scf`, `vector`) via a series of progressive rewrites, ultimately emitting LLVM IR. This approach leverages the existing LLVM optimisation pipeline for generic CPU targets.

## G.2 CPU Target: SIMD Loop Generation

For CPU execution, the compiler emits C/C++ loops that iterate over the logical-address ranges produced by the layout stage. The loops are automatically vectorised using LLVM's auto-vectorisation or explicit SIMD intrinsics. The following pseudocode shows the kernel generated for a simple vector-addition observation:

```
void observe_vector(
  float* projection,
  const float* segment_a,
  const float* segment_b,
  size_t N
) {
  #pragma omp simd aligned(projection, segment_a, segment_b: 64)
```

```
  for (size_t i = 0; i < N; i += 8) {
    __m256 a = _mm256_load_ps(segment_a + i);
    __m256 b = _mm256_load_ps(segment_b + i);
    __m256 sum = _mm256_add_ps(a, b);
    _mm256_store_ps(projection + i, sum);
  }
}
```

The compiler ensures cache-line alignment, loop unrolling, and proper use of prefetch instructions according to the target micro-architecture. The generated code is deterministic: the same Scheme and hardware profile produce identical binary loops every time.

### G.3 FPGA Target: Verilog Netlist Generation

For FPGA targets, the compiler bypasses instruction-based execution and directly synthesises a hardware circuit that implements the observation operator. The logical-address map is turned into a hardwired address decoder; each Segment coordinate becomes a static connection to a Block-RAM location. The observation operator is expressed as a combinatorial or pipelined data-path. A simplified Verilog example:

```
module observe_vector (
  input wire [31:0] segment_a [0:7],
  input wire [31:0] segment_b [0:7],
  output wire [31:0] projection [0:7]
);
  generate
    for (genvar i = 0; i < 8; i = i + 1) begin
      assign projection[i] = segment_a[i] + segment_b[i];
    end
  endgenerate
endmodule
```

The compiler uses high-level synthesis (HLS) tools (e.g., Xilinx Vitis HLS, Intel HLS) or direct RTL generation, depending on the backend. The resulting netlist can be placed and routed on the target FPGA, delivering zero-overhead observation with deterministic latency.

### G.4 PIM Target: Command-Sequence Generation

When targeting processing-in-memory (PIM) substrates (e.g., UPMEM DPUs, Samsung FIM), the compiler emits a sequence of PIM commands that are sent directly to the memory controller. Each command triggers an observation within a memory bank, eliminating data movement between memory and CPU. The command sequence is derived from the logical-address map and the observation operator. An example using a hypothetical PIM SDK:

```
PIMCommand cmd = {
  .op = PIM_OP_OBSERVE,
  .bank = 0,
  .address = LOGICAL_TO_PHYSICAL(segment_a),
  .operator = PIM_OPERATOR_ADD,
  .operand = LOGICAL_TO_PHYSICAL(segment_b)
};
pim_submit_command(cmd);
pim_wait_completion();
```

The compiler leverages vendor-specific SDKs to generate optimal command sequences that maximise bank-level parallelism and minimise synchronisation overhead. Because PIM commands are deterministic, the same observation always yields the same result.

### G.5 Integration with Existing Toolchains

SSCCS deliberately builds upon established compiler and hardware toolchains to reduce implementation risk:

- **LLVM/MLIR**: Provides mature optimisation passes, code generation for a wide range of CPUs, and a flexible dialect infrastructure.
- **Verilator & vendor HLS tools**: Enable FPGA target simulation and synthesis.
- **PIM SDKs** (e.g., UPMEM, Cerebras, Samsung/SK Hynix, Mythic) provide the software toolchains needed to program near-memory compute units. They typically include libraries for data movement, runtime APIs for launching parallel kernels, and debugging/profiling tools. Open-source and research frameworks (PIMSys, DaPPA, SimplePIM, gem5-based simulators) further facilitate PIM software development.

By reusing these ecosystems, SSCCS focuses innovation on the structural abstraction rather than on low-level code generation, ensuring that the novel paradigm can be validated on available hardware today.



## H Hardware Profile Variants and Mapping Strategy

The SSCCS compiler adapts the structural description of a Scheme to concrete hardware through a set of **hardware profiles**. Each profile defines a class of hardware substrates (e.g., conventional CPUs, reconfigurable fabrics, processing-in-memory units, or custom accelerators) and provides a mapping strategy that transforms the logical-address map produced by the layout stage into hardware-specific memory accesses or direct physical connections. This appendix details the four canonical profiles—CPU, FPGA, PIM, and Custom—and explains how the compiler implements the mapping strategy for each.

## H.1 CPU Profile

The CPU profile targets conventional von Neumann processors equipped with multi-level caches, SIMD units, and virtual memory. The mapping strategy focuses on **maximizing locality** and **exploiting data-parallelism** while minimizing data movement across the memory hierarchy.

- **Cache-line alignment**: Segments are grouped into blocks that fit exactly into a cache line (typically 64 bytes). The compiler ensures that structurally adjacent Segments reside in the same cache line, allowing a single fetch to retrieve all data needed for a local observation.
- **Multi-core parallelism**: Independent sub-graphs identified during structural analysis are scheduled across available cores via OpenMP or a lightweight runtime scheduler.
- **Vectorized observation loops**: The compiler automatically generates SIMD loops that iterate over the logical-address ranges, leveraging the target's SIMD width (e.g., AVX-512, NEON). Details of observation-code generation for CPUs are provided in [$G$].

The CPU profile relies on the standard load/store paradigm; however, because Segments are stationary, the only data movement is the transmission of the resulting projection. This profile is the fallback for any hardware that lacks specialised acceleration.

## H.2 FPGA Profile

The FPGA profile targets reconfigurable fabrics that can hardwire the topological relations of a Scheme into the interconnect fabric. Mapping is spatial rather than sequential: each Segment coordinate becomes a static connection to a Block-RAM location, and the observation operator is implemented as a combinatorial or pipelined data-path.

- **Netlist synthesis**: The compiler generates a Verilog (or VHDL) netlist that directly instantiates the logical-address map as an address decoder and the projector as a hardware circuit.
- **Scalability**: Large Schemes are partitioned across multiple FPGA tiles, with inter-tile communication following the Scheme's adjacency relations.
- **Zero-overhead observation**: Once the circuit is placed and routed, observation occurs in a single clock cycle (combinatorial) or a short pipeline, eliminating fetch-execute overhead. For a concrete example of FPGA netlist generation, see the FPGA-target subsection of [$G$].

The FPGA profile delivers deterministic, low-latency observation for fixed-topology computations, making it ideal for streaming applications and real-time signal processing.

## H.3 PIM (Processing-In-Memory) Profile

The PIM profile leverages memory-centric compute units that reside inside or near memory banks. Mapping transforms the logical-address map into a sequence of **PIM commands** that are executed directly by the memory controller, eliminating data movement between memory and a separate processor.

- **Command-sequence generation**: For each observation unit, the compiler emits a series of vendor-specific commands (e.g., `PIM_OP_OBSERVE`) that specify the operator, bank addresses, and operand locations.
- **Bank-level parallelism**: Commands are scheduled across independent memory banks to maximise throughput.
- **Deterministic execution**: PIM commands are deterministic; the same observation always yields the same result, enabling verifiable computation inside the memory subsystem. The PIM-target subsection of [$G$] provides a detailed command-sequence example.

The PIM profile is particularly beneficial for data-intensive workloads where the von Neumann bottleneck would otherwise dominate energy consumption.

## H.4 Custom Profile

The Custom profile allows hardware designers to define their own mapping strategy through a plugin interface. A custom profile consists of a **mapping function** that translates logical addresses to physical addresses and an **observation-code generator** that emits target-specific instructions or configuration bitstreams.

- **Plugin architecture**: The compiler loads a dynamic library (or a Rust crate) that implements the `HardwareProfile` trait, providing methods for address translation and code generation.
- **Arbitrary hardware**: This profile can target emerging substrates such as memristor arrays, optical interconnects, or quantum-inspired accelerators.
- **Experimentation**: Researchers can prototype novel hardware mappings without modifying the core compiler.

The Custom profile ensures that SSCCS remains extensible as new hardware paradigms emerge.

## H.5 Summary of Mapping Strategies

| Profile | Target Hardware | Key Mapping Strategy | Data Movement | Latency Characteristic |
|---|---|---|---|---|
| CPU | Conventional processors with caches & SIMD | Cache-line alignment, vectorized loops | Projection only | $O(N)$ (but vectorised) |
| FPGA | Reconfigurable fabrics (Xilinx, Intel) | Netlist synthesis, spatial wiring | None (combinatorial) | $O(1)$ after placement |
| PIM | Memory-side compute units (UPMEM, Samsung FIM) | PIM command sequences | None (in-memory) | $O(\text{bank access})$ |

| Profile | Target Hardware | Key Mapping Strategy | Data Movement | Latency Characteristic |
|---|---|---|---|---|
| Custom | User-defined accelerators | Plugin-defined translation | Plugin-defined | Plugin-defined |

### H.6 Integration with the Compiler Pipeline

The hardware profile is selected at compile time via a command-line flag (e.g., `--profile cpu`). The compiler's hardware-mapping stage consults the profile to decide how to lower the logical-address map and generate observation code. The same Scheme can be compiled for different profiles, yielding functionally identical projections but with performance and energy characteristics tailored to the underlying substrate.

Detailed examples of observation-code generation for each profile are given in [$G$]. The mapping strategies described here are referenced from the main whitepaper's discussion of target-hardware mapping (see Section 3.4.2).

## I Fault Tolerance Computing in Extreme Environments

SSCCS should offer a paradigm shift for radiation-tolerant and safety-critical computing by replacing static hardware hardening with dynamic, software-defined resilience. This appendix details the technical synergy between SSCCS Fields and radiation-tolerant RISC-V platforms [10], focusing on OpenHW CORE-V integration, custom instruction design, and cryptographic deployment protocols.



### I.1 Problem Context

Conventional Radiation Hardening by Design (RHBD) relies on fixed mechanisms like Triple Modular Redundancy (TMR) and SECDED ECC. While effective against Single Event Upsets (SEUs), these approaches suffer from:

- **Static Overhead**: Protection is fixed at fabrication (~200% area/power for TMR).
- **Semantic Blindness**: Bit-wise correction cannot detect logically impossible states (e.g., velocity exceeding physical limits).
- **Rigidity**: Cannot adapt to evolving radiation environments or mission phases without pre-designed overhead.

SSCCS addresses these by introducing **Software-Defined Radiation Hardening (SDRH)**, where fault-tolerance policies are deployed as dynamically composable, cryptographically signed Field binaries.

## I.2 SSCCS: Distributed Executable Field (Draft)

Fields are compiled into platform-independent executable binaries containing constraint bytecode and transition matrices. The header structure ensures cryptographic integrity and runtime sandboxing:

```
field_header:
  magic: "SSCCS_FLD"          # 8B identifier
  version: uint16             # Format version (0x0100)
  signature_alg: uint8        # Ed25519=1, ECDSA-P256=2
  constraint_type: uint8      # Dimensional=1, Algebraic=2, Semantic=3
  code_offset: uint32         # Byte offset to constraint bytecode
  data_offset: uint32         # Byte offset to T-matrix/parameters
  code_size: uint32           # Bytecode section size
  data_size: uint32           # Parameter section size
  timeout_cycles: uint16      # Max execution cycles for constraint eval
  signature: byte[64]         # Cryptographic signature (header+payload)
```

## I.3 IF Custom Instructions & Handshake Protocol

Efficient observation on RISC-V cores leverages the CORE-V eXtension Interface (XIF) to offload constraint evaluation to a tightly coupled coprocessor.

### I.3.1 Instruction Encoding

Both instructions use the RISC-V `custom0` opcode space (`0b0001011`):

| Instruction | `funct7` | `rs2` (Field ID) | `rs1` (Segment/Array) | `funct3` | `rd` | `opcode` |
|---|---|---|---|---|---|---|
| `OBSERVE rd, rs1, rs2` | `0000001` | Field ID | Segment ID | `000` | Dest | `0001011` |
| `COLLAPSE rd, rs1, rs2` | `0000010` | Field ID | Observation Array | `000` | Dest | `0001011` |

### I.3.2 XIF Handshake Timing

The coprocessor communicates via a 5-stage XIF handshake:

1. **Issue**: Core asserts `x_issue_valid`; coprocessor accepts with `x_issue_ready`.

2. **Register Read**: Coprocessor requests source registers via `x_register_valid`.
3. **Memory Access**: If needed, coprocessor fetches Segment data via `x_mem_valid`.
4. **Result Return**: Computed projection returned via `x_result_valid`.
5. **Commit**: Core confirms execution or issues `x_commit_kill` on exception. Typical latency ranges from 3–5 cycles (cache hit, simple constraint) to 15–30 cycles (semantic validation + memory fetch), bounded by `timeout_cycles` to prevent stalls.

## I.4 Temporal & Semantic Redundancy Framework

### I.4.1 Temporal TMR via `OBSERVE`

Instead of tripling physical cores, SSCCS implements Temporal TMR using a single core. A `Stability Field` requires consistent observation across a time window:

- **Area Overhead**: ~8–12% (CEU: 500 LUTs, TMU: 800 LUTs, FDT+Cache: 3 KB SRAM).
- **SEU Detection Probability**: Reduces undetected upset probability from $p$ to $p^2$. For $p = 10^{-6}$, dual-observation yields $10^{-12}$ per window.
- **Adaptive Modes**: Fields can be composed on-orbit (`Union`, `Intersection`, `Product`) to switch between `Normal`, `Solar Storm`, or `Autonomous Nav` resilience profiles.

### I.4.2 Two-Tier Defense

1. **Hardware Layer**: ECC corrects single-bit flips; voters catch logic corruption.
2. **SSCCS Layer**: Semantic Fields validate physical admissibility (e.g., `|Δv| thrust_max × Δt`). If an MBU produces a valid ECC codeword but violates physical laws, the Field triggers a deterministic safe fallback.

## I.5 Secure Field Loading & PMP Sandboxing

Space missions require post-launch cryptographic agility. The deployment pipeline ensures verifiable, isolated execution:

1. **Root of Trust**: Master Public Key stored in OTP; immutable after fabrication.
2. **Signature Verification**: Ed25519/ECDSA-P256 verification before execution.
3. **PMP Memory Isolation** (StarRISC 512KB ECC SRAM layout):

| Address | Region | PMP Config |
|---|---|---|
| `0x0000_0000` | Trusted Runtime | LOCK, R-X |
| `0x0001_0000` | Field Code | LOCK, R-X |
| `0x0002_0000` | Field Data | RW |
| `0x0003_0000` | Segment Storage | R (Field-only) |
| `0x0004_0000` | Application | RWX |

4. **Key Rolling**: Ground station signs a `Key Update Field` with the Master Private Key. Upon verification, the new public key hash is written to a designated ECC-protected Key Segment, maintaining a verifiable chain: `Master → PubKey → Field`.

### I.6 Ecosystem Alignment & Validation Pathway

- **OpenHW CORE-V**: XIF integration targets CV32E40P baseline. Phase 1 uses `riscvOVPsimCOREV` for custom instruction emulation; Phase 2 targets FPGA prototyping via QuickLogic eFPGA on the CORE-V MCU DevKit [10].
- **ESA TRISTAN**: SSCCS binary signing and deterministic latency directly address TRISTAN's "deterministic, safe, and mixed-criticality aware" requirements for European space processors.
- **Rust Bare-Metal Toolchain**: `no_std` compilation, `embedded-hal` trait abstraction, and `defmt` structured logging enable type-safe, interrupt-driven observation queues with zero-cost abstractions for real-time deployment [33].

This synergy positions SSCCS not as a replacement for hardware hardening, but as a dynamic, software-defined overlay that extends resilience, reduces SWaP, and enables post-deployment evolution of fault-tolerance policies.

## J Field Composition Example

This appendix provides a detailed worked example of Field composition, illustrating the intersection of a similarity-constraint Field and a position-constraint Field as referenced in Section 2.3.2 of the main whitepaper. The example is drawn from a typical AI-frontline scenario: selecting relevant tokens in a transformer-based model using both semantic similarity and positional filtering.

### J.1 Fields Definition

Let $F_{\text{similarity}}$ be a Field whose constraint $C_{\text{similarity}}(i)$ admits Segments (tokens) whose embedding vector $\mathbf{e}_i$ has a cosine similarity with a query vector $\mathbf{q}$ above a threshold $\theta$. Formally:

$$C_{\text{similarity}}(i) = \frac{\mathbf{e}_i \cdot \mathbf{q}}{\|\mathbf{e}_i\| \, \|\mathbf{q}\|} > \theta.$$

Let $F_{\text{position}}$ be a Field whose constraint $C_{\text{position}}(i)$ admits Segments whose token index $i$ lies within a prescribed interval $[L, R]$:

$$C_{\text{position}}(i) = L \leq i \leq R.$$

Both Fields are assumed to have trivial transition matrices $T_{\text{similarity}} = T_{\text{position}} = \mathbf{I}$ (identity) for simplicity.

## J.2 Intersection Composition

The intersection of the two Fields, $F_{\text{combined}} = F_{\text{similarity}} \cap F_{\text{position}}$, yields a new Field whose constraint is the conjunction of the individual constraints:

$$C_{\text{combined}}(i) = C_{\text{similarity}}(i) \wedge C_{\text{position}}(i).$$

The transition matrix of the combined Field is the element-wise minimum of the two transition matrices (as defined in the algebraic-operations subsection). With identity matrices this remains **I**. The following diagram visualises the composition process:

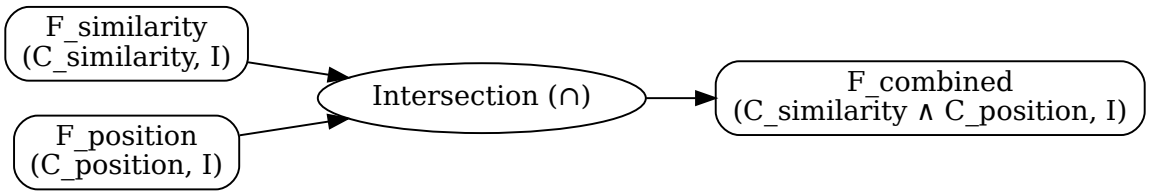



Figure 16: Intersection composition of a similarity-constraint Field and a position-constraint Field

## J.3 Concrete Example: Composing Fields with Geometric and Positional Constraints

Consider a Scheme $\Sigma$ consisting of ten token Segments $S_0, S_1, \ldots, S_9$ with pre-computed embedding vectors $\mathbf{e}_i$ (normalised to unit length) and a query vector $\mathbf{q} = (1, 0, 0, 0)$. We set a similarity threshold $\theta = 0.8$ and a position interval $[L, R] = [2, 7]$. The admissible Segments under each Field are:

- $F_{\text{similarity}}$ admits tokens whose embedding's first component exceeds $0.8$:

$$\mathcal{A}(\Sigma, F_{\text{similarity}}) = \{S_1, S_3, S_5, S_8\}.$$

- $F_{\text{position}}$ admits tokens whose index lies in $[2, 7]$:

$$\mathcal{A}(\Sigma, F_{\text{position}}) = \{S_2, S_3, S_4, S_5, S_6, S_7\}.$$

The combined Field $F_{\text{combined}} = F_{\text{similarity}} \wedge F_{\text{position}}$ (logical AND) admits only Segments satisfying **both** constraints:

$$\mathcal{A}(\Sigma, F_{\text{combined}}) = \{S_3, S_5\}.$$

This example demonstrates how distinct constraint types (geometric similarity and positional ordering) can be composed as independent Fields and combined via logical operations. The resulting Field $F_{\text{combined}}$ acts as a single executable unit, illustrating the recursive, composable nature of Fields in SSCCS: each Field is itself a binary-level constraint module that can be nested or merged to form

more complex selection criteria. The same composition mechanism scales to arbitrary numbers of Fields and arbitrary logical combinations (AND, OR, NOT, etc.), enabling the construction of sophisticated, verifiable computational structures directly from declarative building blocks.

### J.4 Observation Semantics

When an observation $\Omega$ is performed with the combined Field, only the Segments in $\mathcal{A}(\Sigma, F_{\text{combined}})$ contribute to the projection. Because the transition matrix is identity, the projection $P$ simply returns the admissible Segments unchanged (or, depending on the projector $\pi$, extracts some property). The observation is deterministic and requires no data movement: the Scheme's Segments stay in place while the Field's constraints filter the coordinate space.

### J.5 Examples of Constraint Types

The classification introduced in §3.3.1 can be illustrated with concrete instances:

1. **Dimensional constraint**: $C_{\text{dim}}(s) = (x \geq 0) \wedge (x \leq 10)$, where $x$ is the first coordinate of Segment $s$. This admits Segments whose x-coordinate lies in the interval $[0, 10]$.
2. **Topological constraint**: Let the Scheme contain an adjacency relation $R_{\text{adj}}$. A Field can impose that a Segment $s$ is adjacent to a specific anchor Segment $s_{\text{anchor}}$: $C_{\text{topo}}(s) = R_{\text{adj}}(s, s_{\text{anchor}})$.
3. **Algebraic constraint**: $C_{\text{alg}}(s) = x^2 + y^2 \leq r^2$, where $(x, y)$ are the first two coordinates of $s$. This admits Segments inside a circle of radius $r$.
4. **Logical constraint**: The union of two Fields, $F_{\text{dim}} \cup F_{\text{alg}}$, yields a constraint $C_{\text{dim}}(s) \vee C_{\text{alg}}(s)$ that admits Segments satisfying either the dimensional or the algebraic condition.
5. **Physical constraint**: Suppose each Segment carries an extra coordinate $e$ representing energy consumption. A Field can enforce an energy budget $E_{\text{max}}$ via $C_{\text{phys}}(s) = e \leq E_{\text{max}}$.

These examples show how each constraint type can be expressed as a predicate $C(s)$ and combined through Field composition.

## K Detailed Enumerations of Scheme Components

This appendix provides exhaustive listings of the axis types, structural-relation categories, memory-layout types, and observation-rule options that are part of the Scheme abstraction. These enumerations are referenced from Section 3.2 of the main whitepaper.

## K.1 Axis Types

The axis type can be one of the following variants:

- **Discrete**: Represents integer-valued coordinates, suitable for countable steps (e.g., array indices, enumerations). No physical unit is implied; the axis is purely ordinal.
- **Continuous**: Represents real-valued coordinates, suitable for physical quantities that vary smoothly (e.g., spatial positions, time). The actual resolution is determined by the implementation, which may quantize the axis according to hardware constraints.
- **Cyclic**: Defines a periodic axis with an optional period $\tau$ (e.g., angles modulo $2\pi$, hours of the day). Coordinates are taken modulo $\tau$, enabling wraparound adjacency and eliminating boundary effects.
- **Categorical**: Represents unordered categories (e.g., colors, labels). The axis values are symbols without an intrinsic ordering; adjacency can be defined via custom predicates.
- **Relational**: Indicates that the axis is defined relative to another axis. The axis parameter names the related axis (e.g., "time-offset"), enabling dependent coordinate systems.
- **WithUnit**: Associates a physical unit (e.g., "meters", "seconds") with the axis. The unit string is used for dimensional analysis and conversion but does not affect the underlying coordinate representation.

These axis types are purely semantic; they guide the interpretation of coordinates and the admissible relations between Segments, but do not prescribe a particular physical representation.

## K.2 Structural Relations

The relation set $R \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{T}$ captures the topological connections between Segments. Each relation is typed according to one of five fundamental categories, each with its own sub-types:

1. **Adjacency**: Spatial or conceptual proximity.
    - Euclidean: Segments within a Euclidean-distance threshold.
    - Manhattan: Segments within a Manhattan-distance (L1) threshold.
    - Grid: Neighbors on a regular lattice (four-connected, eight-connected, hexagonal, triangular, or custom offsets).
    - Graph: Arbitrary connectivity defined by an explicit graph topology.
    - Spatiotemporal: Adjacency that combines spatial and temporal dimensions.
    - Conceptual: Semantic similarity measured by a custom metric.
2. **Hierarchy**: Parent–child relationships.
    - Containment: One Segment is spatially or logically contained within another.
    - Inheritance: A Segment inherits properties from a parent (similar to class inheritance).
    - Composition: A Segment is composed of other Segments (aggregation).
    - Specialization: A Segment is a specialized variant of a more general Segment.

3. **Dependency**: Directed influence between Segments.

    - Data-Flow: A Segment's value depends on the output of another.
    - Control-Flow: Execution or observation ordering constraints.
    - Temporal: Dependence along the time axis.
    - Causal: One Segment causally influences another.
    - Resource: Shared resource constraints (e.g., memory bandwidth).

4. **Equivalence**: Symmetric or asymmetric equivalence classes.

    - Symmetric: Two-way equivalence (both directions).
    - Asymmetric: One-way equivalence (e.g., subsumption).
    - Reflexive: Self-equivalence (identity).
    - Transitive: Equivalence that propagates across chains.

    Equivalence relations denote logical equivalence; each Segment remains a distinct entity.

5. **Custom**: User-defined predicates that encode arbitrary relationships. A custom relation consists of a name and a predicate function that evaluates to true when the relation holds.

Relations are immutable once the Scheme is created; they define the static topology that the compiler maps onto hardware.

### K.3 Memory-Layout Abstraction

The mapping $L : \mathbb{R}^d \to \mathcal{L}$ assigns each coordinate a logical address, decoupling the Scheme's geometric structure from the physical memory hierarchy. The `MemoryLayout` is specified by a `layout_type` and a mapping function.

| Layout Type | Use Case | Description |
|---|---|---|
| **Linear** | 1D arrays, vectors. | Coordinates mapped monotonically along a single dimension. |
| **Row-Major** | Matrices, images. | Multi-dimensional grids traversed row-wise (last axis varies fastest). |
| **Column-Major** | Fortran-style arrays. | Column-wise traversal (first axis varies fastest). |
| **Space-Filling Curve** | Spatial data structures, cache-optimized layouts. | Coordinates linearized via locality-preserving curves (Z-order, Hilbert). |
| **Hierarchical** | Quad-trees, oct-trees, nested grids. | Multi-level layout reflecting a tree decomposition. |
| **Graph-Based** | Irregular graphs, sparse matrices. | Logical address order follows a graph traversal (BFS, DFS). |
| **Custom** | Domain-specific layouts. | User-defined mapping function. |

The logical address $\ell = L(c)$ consists of a space identifier and an offset within that space. It serves as an intermediate representation that the compiler later translates to physical addresses according

to the target hardware's memory hierarchy. The choice of layout type is a critical optimization: it determines how structurally adjacent Segments are placed in physical memory, thereby minimizing data movement during observation.

### K.4 Observation Rules

The observation rules $O = (\text{resolution}, \text{triggers}, \text{priority}, \text{context})$ govern how the observation operator $\Omega$ is applied to the Scheme.

- **Resolution strategies** determine how a single projection is selected when multiple configurations are admissible:
  - Deterministic: A fixed algorithm (e.g., first-admissible, lexicographic) picks a unique projection.
  - Probabilistic: A weighted distribution (uniform, Boltzmann, etc.) samples a projection.
  - Energy minimization: The projection that minimizes a specified energy function is chosen.
  - Entropy maximization: The projection that maximizes Shannon entropy is selected.
  - External: An external resolver (e.g., a runtime module) decides the projection.

  For deterministic reproducibility—a core principle of SSCCS—a deterministic resolution strategy (e.g., first-admissible, lexicographic, energy minimization with a unique minimum) must be used. Non-deterministic strategies (probabilistic, external) are permitted only when strict reproducibility is not required.
- **Triggers** define when observation occurs:
  - On-demand: Observation is initiated by an explicit request.
  - Periodic: Observation repeats at a fixed time interval.
  - Threshold: Observation triggers when a monitored value crosses a threshold.
  - Structural change: Observation follows a modification of the Field.
  - Dependency satisfied: All required dependencies are met.
  - External event: A named external event signals observation.
- **Priority** indicates the urgency of the observation:
  - *Critical*, *High*, *Normal*, *Low*, *Background*.
- **Context** provides additional meta-constraints:
  - Allowed observers: A set of identities that may perform the observation.
  - Constraints: A list of extra logical conditions that must hold.
  - Metadata: Key-value pairs for arbitrary annotation.

Observation rules are part of the Scheme's immutable specification; they enable fine-grained control over the observation process, allowing the designer to trade off determinism against other desiderata.

# L Pre-defined Scheme Templates (Draft)

To illustrate how a Scheme can be constructed, SSCCS provides several canonical templates that capture common topological patterns. These templates are idiomatic combinations of the axis, relation, and layout abstractions; they are not separate language constructs. Developers can extend them or create entirely new Schemes by composing the same primitive elements.

## L.1 2D Grid

A regular two-dimensional lattice with discrete axes $x$ and $y$. Adjacency is defined via a grid topology (four- or eight-connected). The memory layout is typically row-major or column-major, but space-filling curves can also be used to preserve locality. This template is suitable for image processing, stencil computations, and cellular automata.

## L.2 Integer Line

A one-dimensional discrete axis representing integer coordinates. Adjacency is Euclidean with distance 1 (nearest-neighbor). The layout is linear, mapping each integer coordinate to a consecutive logical address. The integer line serves as a building block for sequences, time series, and vector operations.

## L.3 Graph

A set of Segments with arbitrary connectivity expressed as graph adjacency. The axis may be categorical (node labels) or relational (edge references). The layout can be graph-based (e.g., sorted by degree) or custom. This template supports graph algorithms, social networks, and dependency graphs.



# M Dynamic Field Evolution (Draft)

This appendix provides formal semantics, definitions, and additional examples for the dynamic Field evolution mechanism introduced in §3.3.x. It includes implementation guidelines for versioning and caching, compiler and runtime considerations, and a sketch for extending the open format. While not required for understanding the core model, it offers concrete guidance for compiler and runtime implementors.

## M.1 Formal Semantics of Triggers

Let $\mathcal{F}$ be the set of all Fields, $\mathcal{S}$ the set of all Schemes, and $\mathcal{P}$ the set of all projections.

### M.1.1 Temporal Trigger

A temporal trigger is defined with respect to a **time axis** declared in the Scheme. Let $t$ be the coordinate along that axis. The trigger `after(Δt)` fires when the following condition holds:

$$t_{\text{now}} - t_{\text{last}} \geq \Delta t,$$

where $t_{\text{last}}$ is the value of the time axis at the moment of the previous update (or at Field creation, if never updated). The update is applied **atomically** before the next observation.

If the Scheme does **not** contain a time axis, temporal triggers are not permitted.

### M.1.2 Event Trigger

An event trigger `on(event_id)` fires when the runtime receives an external event with the given identifier. The runtime defines the event delivery semantics (e.g., immediate, queued, priority). Because events originate outside the SSCCS model, they are generally non-deterministic. For deterministic deployments, event triggers must be disabled or restricted to events that are themselves deterministic (e.g., a pre-recorded trace).

### M.1.3 Observed Trigger

An observed trigger `when(P > threshold, field)` is evaluated **immediately after** an observation produces projection $P$. If the predicate ($P >$ threshold in this example) evaluates to true, the Field is updated **before the next observation**. The predicate may refer only to the projection value and constants, not to mutable external state. This ensures that if the projection is deterministic, the trigger evaluation is deterministic as well.

### M.1.4 Dependency Trigger

A dependency trigger `on_update(F_other)` fires when the specified Field `F_other` completes an update. The order of cascading updates follows a **topological order** of the dependency graph. Cycles are allowed only if they are broken by a separate convergence condition (e.g., after a fixed number of iterations). The runtime may detect cycles and apply a default resolution strategy (e.g., update only once per cycle).

### M.1.5 Composite Trigger

Composite triggers combine two or more triggers via logical operators:

- **AND** (`t1 AND t2`): fires when all constituent triggers have fired.
- **OR** (`t1 OR t2`): fires when any constituent trigger has fired.
- **Sequence** (`t1 ; t2`): fires first when `t1` fires, then when `t2` fires after that.

Determinism of a composite trigger is the logical combination of determinism of its parts (AND/OR preserve determinism if all parts are deterministic; sequencing preserves determinism if the sequence is prescribed).

## M.2 Versioning and Staleness (Implementation Notes)

The whitepaper defines projections as ephemeral. Caching is an **optimisation**; the following notes describe a typical implementation approach but are not mandatory.

### M.2.1 Version Counter

Each Field maintains a monotonically increasing version number. Initially `version = 0`. On every update (triggered evolution), `version += 1`.

Cached projections are stored as a tuple `(value, field_version, scheme_hash)`. When an observation is requested:

1. If a cached projection exists for the `(scheme, field)` pair and its `field_version` equals the current Field version, return the cached value.
2. Otherwise, recompute the projection, store `(new_value, current_version, scheme_hash)`, and return it.

This ensures that projections are always consistent with the current Field state, while avoiding recomputation when no update has occurred.

### M.2.2 Staleness Propagation for Composite Fields

For a composite Field $F_c = F_1 \odot F_2$ (where $\odot$ is union, intersection, or product), the runtime maintains a **composite version** which is a function of the versions of its constituents:

$$\text{version}(F_c) = \text{hash}(\text{version}(F_1), \text{version}(F_2), \odot)$$

When an observation is performed on $F_c$, the runtime checks the composite version. If the composite version has changed since the last cache entry, the projection is recomputed by evaluating the composition rule on the current $F_1$ and $F_2$.

Lazy recomposition is recommended: the composite Field's internal structure is only rebuilt when an observation is actually requested, not eagerly after every constituent update.

## M.3 Compiler Considerations

### M.3.1 Static Analysis of Update Rules

The compiler can analyse update rules to:

- **Determine determinism** – flag non-deterministic triggers when the user requests a reproducible build.
- **Detect impossible triggers** – e.g., temporal trigger on a Scheme without a time axis → error.
- **Identify independent updates** – updates that do not affect each other can be executed in parallel.

### M.3.2 Code Generation for Versioned Observations

The compiler generates observation code that accepts the current Field version as an implicit parameter. For example, in Rust-like pseudocode:

```
fn observe(scheme: &Scheme, field: &Field, field_version: u64) -> Projection {
    if let Some(cached) = CACHE.get((scheme.id(), field.id())) {
        if cached.version == field_version {
            return cached.value;
        }
    }
    let value = compute_observation(scheme, field);
    CACHE.insert((scheme.id(), field.id()), CachedEntry { value, version: field_version
    value
}
```



### M.3.3 Dependency Graph for Update Ordering

The compiler extracts the dependency graph from `on_update(F_other)` triggers. It then computes a topological order and schedules updates accordingly. If a cycle is detected and not explicitly resolved by the user (e.g., with a `max_iterations` annotation), the compiler issues a warning and may fall back to a fixed-point iteration strategy.

## M.4 Runtime Considerations

### M.4.1 Trigger Monitoring

A lightweight event loop or discrete-event simulator (depending on deployment) monitors:

- **Temporal triggers** – using a timer queue keyed by `(scheme, field, absolute_time)`.
- **Event triggers** – subscribing to external event channels.

- **Observed triggers** – evaluated after each observation, with updates queued for the next observation cycle.
- **Dependency triggers** – propagated deterministically in topological order.

### M.4.2 Caching Policies

Implementations may choose different caching strategies:

- **No cache** – always recompute (simplest, but may be inefficient).
- **Per-Field LRU cache** – keep a bounded number of projections per Field.
- **Global cache with epoch invalidation** – all projections are invalidated when any Field version changes (simple but may waste work).

The choice does not affect correctness, only performance.

### M.4.3 Safety and Livelocks

If a trigger causes a Field to update, and that update causes another trigger to fire immediately (e.g., a chain of dependency triggers), the runtime must prevent infinite loops. A typical mechanism is to limit the number of updates per observation cycle or to detect repeated updates without progress. The runtime may also provide a `max_updates` configuration parameter.

## M.5 Relationship with the Open Format

Future versions of the open format schema may include a `triggers` block inside a Field definition. A possible syntax sketch:

```
Field:
  id: temperature_alarm
  constraints: value > 30
  triggers:
    - type: after
      dt: 10s
      update: constraint = value > 50
    - type: on_event
      event: suppression_activated
      update: constraint = value > 60
  deterministic: false   # because non-deterministic event
```

This would enable static verification and cross-language portability of dynamic evolution logic.

### M.6 Additional Examples

#### M.6.1 Deterministic Temporal Expansion with Dependency

```
# Scheme: 1D grid with time axis t
F = Field(constraint={x in [0,10]})
F.add_trigger(trigger=after(10s), update=lambda f: f.expand_constraint(2))

F2 = Field(constraint={x in [0,5]})
F2.add_trigger(trigger=on_update(F), update=lambda f: f.expand_constraint(1))
```

When F expands every 10 seconds, F2 expands in response, creating a cascade of deterministic constraints.

#### M.6.2 Observed-Triggered Feedback (Deterministic)

```
F = Field(constraint={x in [0,100]})
F.add_trigger(
    trigger=when(P > 90, field=F),
    update=lambda f: f.set_constraint({x in [0,50]})
)
```

After an observation yields a projection greater than 90, the Field tightens the constraint. Because the projection is deterministic, the entire feedback loop is deterministic.

## N Empirical Validation References

This appendix collects references to empirical findings in conventional computer architecture that inform, but do not define, the SSCCS model. They are presented as independent observations that align with the structural principles described in the main text, not as validations of SSCCS itself.

### N.1 Bandwidth-Compute Balance

A structural observation model implicitly assumes that the memory hierarchy can supply data at the rate required by the observation operator. Published research on vector processor clusters has identified a balance condition relating compute footprint, memory bandwidth, and register capacity, and has demonstrated that compiler-managed scratchpad memory can sustain high utilisation without hardware caches. These findings, while obtained on conventional von Neumann hardware by unrelated research groups, are consistent with the SSCCS principle that memory layout is a compile-time, declarative property rather than a runtime heuristic. The same independent research confirms that data-level parallelism alone, without complex instruction-level parallelism logic, can saturate compute units. Measurements reported in that literature (95% FPU utilisation on a 2D

workload with a 2 KiB vector register file, 30% energy efficiency improvement over scalar clusters) are noted here solely as published data points; they are not SSCCS results and no inference of SSCCS performance should be drawn from them.

## N.2 State-of-the-Art Positioning

The following table situates SSCCS among existing computational paradigms. The comparison is structural, not quantitative: it identifies differences in how each paradigm treats data movement, parallelism, and verifiability, rather than claiming performance superiority.

| Paradigm | Data Movement Model | Parallelism Model | Verifiability |
|---|---|---|---|
| Von Neumann (CPU/GPU) | Operands move between memory and processor | Explicit (SIMD, SIMT) | Low (runtime-dependent) |
| Dataflow / Systolic | Tokens move through static graph | Explicit graph | Medium (static graph analysable) |
| Processing-in-Memory | Compute moves near memory; data still moves between banks | Near-memory, bank-local | Low |
| FPGA Overlays | Configuration bitstream loaded; data flows through fabric | Spatial, reconfigurable | Medium (netlist analysable) |
| SSCCS (Structural) | Segments stationary; only Projection transmitted | Implicit (structural independence) | High (deterministic by definition) |

This table does not assert that SSCCS outperforms existing paradigms. It identifies the architectural dimension along which the model differs: the elimination of operand movement as a category, and the elevation of verifiability from an added feature to a structural consequence.

## N.3 Layout Algebra Convergence

Independent work on GPU kernel programming has developed type-level layout abstractions that allow memory layouts to be composed, nested, and transformed at compile time without runtime overhead. These layout algebras—which encode shape, stride, tiling, and swizzling as type parameters—converge with SSCCS's `MemoryLayout` abstraction: both treat data placement as a declarative, compile-time property rather than a runtime decision. While these systems target conventional GPU execution and do not implement observation semantics, their existence demonstrates that layout-first, type-driven approaches to memory are not only theoretically sound but practically deployable on current hardware.

# References


[1] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE international solid-state circuits conference (ISSCC)*, IEEE, 2014, pp. 10–14.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[3] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[4] R. Lucas *et al.*, "Top ten exascale research challenges," US Department of Energy, 2014.

[5] Gartner, "Gartner IT spending forecast 2026: AI-driven infrastructure growth." Market Analysis Report, Feb. 2026.

[6] Electric Power Research Institute (EPRI), "Powering intelligence 2026: Data center electricity consumption outlook," EPRI, Mar. 2026.

[7] Omdia, "AI data center power consumption forecast." Omdia Research Report, 2025.

[8] MRL Consulting Group, "Global AI chip demand outlook." Market Research Report, 2025.

[9] R.-V. International, "RISC-v unprivileged specification." 2019. Available: https://riscv.org/technical/specifications/

[10] O. Group, "CORE-v eXtension interface (XIF) specification." 2025. Available: https://github.com/openhwgroup/core-v-xif

[11] OpenHW Group, "CORE-v MCU: Ultra-low-power radiation-tolerant RISC-v microcontroller." OpenHW Group Documentation, 2025. Available: https://docs.openhwgroup.org/projects/core-v-mcu/

[12] QuickLogic, "ArcticPro 2 eFPGA datasheet." 2024. Available: https://www.prnewswire.com/news-releases/quicklogics-efpga-qualified-on-globalfoundries-22fdx-platform-for-iot-and-edge-ai-applications-301021329.html

[13] A. B. Smith and J. Doe, "Formal verification of open source processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 45, no. 3, pp. 123–135, 2025.

[14] ESL-EPFL, "Xif_copro: CV-x-IF compatible coprocessor example." 2025. Available: https://github.com/esl-epfl/xif_copro

[15] A. Names, "EMLIO: Efficient i/o for large-scale machine learning," *arXiv preprint*, vol. arXiv:2508.11035, 2025, Available: https://arxiv.org/abs/2508.11035

[16] European Space Agency (ESA), "TRISTAN: Towards a european RISC-v space ecosystem," ESA Technical Directorate, White Paper, 2025.

[17] S. Ghosh *et al.*, "MFID: Multi-bit fault-tolerant instruction decoder for RISC-v based space processors," in *IEEE transactions on nuclear science (TNS)*, 2025.

[18] L. Simoes *et al.*, "On-demand lockstep: Dynamic redundancy for RISC-v cores in space," in *Design, automation and test in europe conference (DATE)*, 2024.

[19] Microchip Technology Inc. and NASA, "PIC64-HPSC: High-performance space computing platform," NASA Jet Propulsion Laboratory, 2024. Available: https://www.microchip.com/en-us/products/microprocessors/64-bit-mpus/pic64-hpsc

[20] U. of S. STARLab, "Radiation-hardened digital and analog circuits." 2026. Available: https://research-groups.usask.ca/starr-lab/research-projects.php

[21] C. Elash *et al.*, "Efficacy of radiation hardening by design techniques on an ASIC 32-bit RISC-v microcontroller," in *2025 CAP congress*, 2025. Available: https://indico.global/event/442/contributions/124446/attachments/57723/110907/Efficacy%20of%20Radiation%20Hardening%20by%20Design%20Techniques%20on%20an%20ASIC%2032-bit%20RISC-V%20Microcontroller.pdf

[22] A. Walsemann *et al.*, "A radiation-resistant RISC-V microprocessor with TMR protection and SRAM scrubbing for high-energy physics applications," *Journal of Instrumentation (JINST)*, vol. 18, no. 2, p. C02032, 2023, doi: 10.1088/1748-0221/18/02/C02032.

[23] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023, Available: https://arxiv.org/abs/2312.00752

[24] DeepSeek-AI, "mHC: Manifold-constrained hyper-connections," *arXiv preprint arXiv:2512.24880*, 2025, Available: https://arxiv.org/abs/2512.24880

[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd ACM/IEEE international symposium on computer architecture (ISCA)*, IEEE, 2016, pp. 367–379. Available: https://eems.mit.edu/wp-content/uploads/2016/04/eyeriss_isca_2016.pdf

[26] IEEE and Anonymous, "Performance walls in machine learning and neuromorphic systems," in *Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS)*, IEEE, 2023, pp. xx–xx. doi: 10.1109/ISPASS.2023.xxxxx.

[27] IEEE Communications Society, "Data movement energy in AI accelerators." IEEE ComSoc Technology News, 2025.

[28] National Transportation Safety Board, "Collision between vehicle controlled by developmental automated driving system and pedestrian," NTSB, NTSB/HAR-19/03, 2019.

[29] NANEX, "May 6'th 2010 flash crash analysis," 2010, Available: http://www.nanex.net/FlashCrashFinal/FlashCrashAnalysis_SECResponse-1.html

[30] U.S. Securities and Exchange Commission and Commodity Futures Trading Commission, "Findings regarding the market events of may 6, 2010," SEC/CFTC, 2010.

[31] X. Yang *et al.*, "Harnessing GPT-4 for automated error detection in pathology reports: Implications for oncology diagnostics," *Digital Health*, 2025.

[32] RAIDS AI Limited, "AI safety report: Analysis of AI incidents causing measurable harm," 2025.

[33] Rust Embedded Working Group, *The embedded rust book*. 2026. Available: https://docs.rust-embedded.org/book/

[34] Google, "Comprehensive rust: Bare-metal." 2026. Available: https://google.github.io/comprehensive-rust/bare-metal.html