Chapter 16

Field Execution on Classical CPUs

Reinterpreting CPU Execution as Field Dynamics

proFQuansistor

Abstract

This document presents a concrete realization of Field Computing on classical CPU-based systems. Building on the architectural foundations established in Book III, it demonstrates how modern CPUs can be reinterpreted as field execution substrates without any modification to hardware, instruction sets, operating systems, or compilers.

Classical CPUs execute instruction streams that are traditionally modeled procedurally. This whitepaper reframes CPU execution as the evolution of computational fields, where instructions function as operators, execution contexts form admissible regions, and scheduling shapes field trajectories rather than instruction order. The reinterpretation preserves full compatibility with existing software stacks while enabling determinism guarantees, auditability, and governed execution.

The focus is on general-purpose CPUs running contemporary operating systems, with an emphasis on portability and immediate deployability. The document avoids hypothetical hardware extensions and instead identifies semantic and orchestration-layer mechanisms that lift classical execution into a field-native framework.

This whitepaper serves as the first concrete specialization of Field Computing on Classical Infrastructure. Subsequent documents extend the approach to GPUs, heterogeneous systems, and workload-specific optimizations, establishing classical CPUs as the primary entry point for practical field-native computation.

1 CPU as an Implicit Field Machine

Classical CPUs are commonly described as sequential or weakly concurrent machines that execute instruction streams under the control of a program counter. While this abstraction is useful for programming, it does not accurately reflect the operational structure of modern processors. In reality, contemporary CPUs implement complex, distributed execution environments whose behavior is best understood as field-like rather than sequential.

A field machine, in the sense of QFC, is a system in which multiple operators act concurrently on a shared and structured state space, subject to locality constraints, interaction rules, and admissibility conditions. Modern CPUs satisfy these criteria implicitly. Instructions are decoded, scheduled, executed, and retired in parallel across pipelines and execution units. State is distributed across registers, caches, buffers, and memory, with coherence protocols governing interaction.

From a field perspective, the instruction pipeline constitutes a local dynamical region of the

field. Instructions propagate through stages that enforce ordering, dependency resolution, and admissibility. Out-of-order execution and speculative execution introduce multiple potential trajectories, of which only admissible ones are committed. This selection process is a concrete instantiation of field admissibility in hardware.

Caches and memory hierarchies further reinforce the field interpretation. They define spatial locality and coupling between operations. Cache coherence protocols act as interaction constraints that mediate how operator effects propagate through the field. Latency, contention, and memory consistency are not anomalies; they are expressions of field geometry.

Concurrency mechanisms such as simultaneous multithreading, multi-core execution, and vector units expand the field spatially. Multiple instruction streams coexist and interact within shared subsystems. The resulting execution behavior emerges from collective dynamics rather than from any single instruction sequence.

Importantly, none of these mechanisms require reinterpretation or modification to function as field components. They already enforce constraints, manage interaction, and resolve admissibility. What is missing is a semantic framework that recognizes and governs these dynamics at the system level.

By identifying CPUs as implicit field machines, Field Computing does not challenge classical correctness. Programs still execute correctly according to their language and ISA semantics. The field interpretation operates orthogonally, providing a higher-level understanding and governance of execution dynamics.

This perspective establishes CPUs as natural substrates for field-native execution. The remainder of this document builds upon this insight, reinterpreting instructions as operators, lifting scheduling to the field level, and demonstrating how determinism and auditability can be achieved without altering the fundamental behavior of classical processors.

2 Instructions as Operators (Reinterpretation)

In classical programming models, instructions are treated as sequential steps in a control flow. Each instruction is assumed to advance execution from one well-defined state to another in a linear progression. While this abstraction is convenient for reasoning about programs, it obscures the true operational role of instructions in modern CPUs. From a field perspective, instructions are more accurately described as operators acting on a shared computational field.

An operator, in the sense used throughout QFC, is a transformation that acts locally on a structured state space under admissibility constraints. Classical CPU instructions already satisfy this definition. Each instruction specifies a localized transformation of registers, memory, or control state, subject to dependency rules, resource availability, and architectural constraints. Execution does not consist of applying instructions in a fixed order, but of composing admissible operator actions within the execution field.

Reinterpreting instructions as operators shifts the focus from sequence to interaction. Multiple instructions may be active concurrently, competing for execution units, interacting through shared caches, or constrained by dependency graphs. Their combined effect is not a simple sum of individual steps, but an emergent result of operator composition under hardware-imposed constraints.

Operator composition in CPUs is dynamic rather than static. The microarchitecture continuously forms and dissolves compositions through mechanisms such as instruction scheduling, reordering, speculation, and retirement. From the field viewpoint, these mechanisms define which operator compositions are admissible at a given moment. Inadmissible compositions are suppressed or deferred, while admissible ones propagate the field state forward.

Importantly, the operator interpretation does not alter instruction semantics. Each instruction still performs exactly the transformation defined by the ISA when it is executed. The reinterpretation concerns how instructions participate in a larger structure of interactions. An instruction's meaning remains local, while its role in computation becomes relational.

This operator-centric view clarifies several classical phenomena. Hazards, stalls, and dependencies are no longer anomalies to be managed procedurally; they are constraints on operator composition. Instruction-level parallelism is not an optimization layered on top of sequential execution, but an intrinsic property of the operator field.

By treating instructions as operators, Field Computing establishes a direct bridge between classical execution and field semantics. Programs become specifications of operator availability rather than prescriptions of execution order. This reinterpretation prepares the ground for lifting scheduling to the field level, where admissibility and governance shape execution trajectories without constraining instruction behavior.

The next chapter builds on this foundation by showing how scheduling can be redefined above the operating system scheduler as a field-level modulation of admissibility rather than as a mechanism for time-sharing or instruction prioritization.

3 Field Scheduling above the OS Scheduler

Classical operating systems schedule execution by allocating processor time to threads and processes. This form of scheduling is fundamentally procedural: it decides when an execution context may run, but it does not govern what kinds of execution are admissible or what guarantees must hold for the resulting computation. Field Scheduling operates at a different level. It does not replace the OS scheduler; it exists above it.

In Field Computing, scheduling is not concerned with time slices, priorities, or fairness at the instruction or thread level. Instead, it modulates admissibility of execution domains and operator compositions. The OS scheduler remains responsible for efficient utilization of CPU resources, while Field Scheduling determines which execution contexts are permitted to evolve as part of a governed computational field.

This separation is essential. The OS scheduler is optimized for responsiveness, throughput, and resource sharing. It has no semantic awareness of determinism requirements, auditability obligations, or contractual guarantees. Field Scheduling introduces this awareness by governing which domains may be active, suspended, or composed at any given moment.

Field Scheduling acts through discrete governance decisions rather than continuous intervention. Domains are admitted, activated, suspended, or terminated based on governance policies and contract satisfaction. Once a domain is active, execution proceeds freely under the OS scheduler. Field Scheduling does not preempt threads or reorder instructions; it constrains admissibility boundaries rather than execution order.

From the field perspective, OS scheduling decisions correspond to micro-level trajectory selection within an admissible region. Field Scheduling defines the region itself. Hardware and OS mechanisms choose specific execution paths, but only among those paths that are permitted by field-level governance.

This layered approach avoids conflicts with existing systems. No changes to kernel schedulers are required. Real-time policies, priority classes, and affinity mechanisms remain intact. Field Scheduling operates orthogonally, enabling strong guarantees without disrupting established scheduling behavior.

Field Scheduling also enables coordination across multiple execution contexts and processes. Workflows spanning multiple processes, containers, or even machines can be governed coherently,

something classical schedulers cannot express. Dependencies, phase transitions, and conditional activation are expressed declaratively as governance rules rather than procedurally as synchronization code.

By lifting scheduling above the operating system, Field Computing achieves governance without control. Execution remains opportunistic, performant, and hardware-driven, while system-level guarantees are enforced through admissibility modulation. This redefinition of scheduling is a cornerstone of deploying field-native computation on classical CPUs without invasive system changes.

The next chapter builds on this result to show how determinism and auditability emerge naturally from field scheduling and governance, even in the presence of classical nondeterministic execution.

4 Determinism and Audit on Classical CPUs

Classical CPUs are often regarded as hostile to determinism due to out-of-order execution, concurrency, cache effects, and operating system scheduling. From a field perspective, these characteristics do not prevent determinism; they merely relocate it. This chapter explains how determinism and auditability emerge at the field level without constraining low-level execution on classical CPUs.

Determinism in Field Computing is not defined as identity of instruction traces or timing behavior. Such a definition would indeed be incompatible with modern CPUs. Instead, determinism is defined as equivalence of terminal field states under a declared governance contract. Multiple execution trajectories may be admissible, provided they converge to outcomes that are equivalent with respect to the field's invariants.

Field Scheduling and governance contracts jointly enforce this form of determinism. By restricting admissible operator compositions and interactions, governance ensures that nondeterministic variations introduced by hardware or the OS cannot affect the semantic outcome. Execution remains free to explore admissible trajectories, but the space of admissible outcomes is constrained.

Auditability follows directly from this construction. Because unlawful execution trajectories are structurally excluded, audit does not require observing execution as it unfolds. There is no need for instruction-level tracing, deterministic replay, or checkpointing. Audit consists in verifying that the execution domain was admitted under the correct contracts and that its terminal field state satisfies the declared invariants.

On classical CPUs, this approach aligns naturally with existing execution behavior. Hardware mechanisms select specific execution orders opportunistically, optimizing for performance and resource availability. From the field perspective, these selections correspond to choosing one admissible trajectory among many. Determinism is preserved because all admissible trajectories are equivalent at the field level.

This separation has important practical consequences. Performance optimizations such as speculative execution, vectorization, and parallelism do not need to be disabled to achieve determinism. In fact, they become allies rather than obstacles. The faster and more aggressively the CPU explores execution space, the more efficiently it reaches an admissible terminal state.

Audit records are compact and meaningful. Rather than storing voluminous execution logs, systems record governance configurations, lifecycle events, and terminal invariants. These records are independent of execution duration, concurrency level, or hardware specifics, making them stable across platforms and deployments.

By defining determinism and auditability at the field level, Field Computing reconciles strong guarantees with the realities of classical CPU execution. This reconciliation is the key to deploying governed, accountable computation on existing infrastructure without sacrificing performance or

compatibility.

The final chapter of this document translates these theoretical guarantees into concrete operational benefits and deployment strategies, demonstrating how field execution on classical CPUs can be adopted immediately.

5 Immediate Benefits and Zero-Hardware-Change Deployment

Field Execution on Classical CPUs delivers concrete benefits precisely because it does not interfere with existing hardware, operating systems, or software stacks. By operating as a semantic and governance layer above classical execution, it enables new guarantees and operational capabilities without requiring any changes to instruction sets, microarchitecture, kernels, or compilers.

The most immediate benefit is outcome-level determinism on inherently nondeterministic systems. Classical CPUs may execute instructions in varying orders due to scheduling and microarchitectural effects, yet field governance ensures that all admissible execution trajectories converge to equivalent terminal field states. This enables reproducible results for numerical computation, scientific workloads, and verification-sensitive tasks without sacrificing performance.

A second benefit is auditability without execution overhead. Because admissibility constraints exclude unlawful execution by construction, there is no need for instruction-level tracing, logging, or replay. Audit is reduced to verification of governance configuration, lifecycle events, and declared invariants. This dramatically lowers operational complexity while increasing trustworthiness.

Isolation is achieved without virtualization overhead. Execution domains provide strong isolation guarantees at the governance level while sharing the same physical CPU resources. This allows mixed-trust workloads to coexist safely on the same system without resorting to heavyweight virtual machines or containers, improving efficiency and composability.

From a deployment standpoint, Field Execution can be introduced incrementally. Existing applications do not need to be rewritten. Selected workloads can be wrapped in governed execution domains, while the rest of the system continues to operate unchanged. Field-aware runtimes and orchestration layers coexist with traditional process management and scheduling.

Integration requires no privileged access to hardware or the kernel. Field governance may be implemented entirely in user space, with optional coordination through existing OS interfaces. This makes deployment feasible in environments ranging from personal workstations to high-performance clusters and cloud infrastructure.

The zero-hardware-change property also reduces adoption risk. Organizations can experiment with Field Execution using existing CPU infrastructure, evaluate benefits on real workloads, and scale deployment gradually. There is no dependency on vendor support, specialized hardware, or long procurement cycles.

By providing immediate benefits without infrastructure disruption, Field Execution on Classical CPUs establishes a practical entry point for field-native computation. It demonstrates that Field Computing is not a future replacement for classical systems, but a present-day upgrade that enhances reliability, auditability, and governance on today's CPUs. This foundation prepares the extension of field execution concepts to GPUs and heterogeneous systems in subsequent documents.