What is Firedancer? A Deep Dive into Solana 2.0

A huge thank you to the Firedancer team for reviewing this article.

What's this Article About?

Solana is the fastest blockchain. But it can be faster. The current Solana Labs validator client is good but optimized for speed-to-market. With the benefit of hindsight, a clean slate, and decades of high-performance computing experience, Jump is dedicated to making Solana even faster and more reliable. Applying their experience in high-frequency trading, Jump is developing Firedancer. This is the most performant validator client on any blockchain. It proposes a complete rewrite of Solana's current validator client in the C programming language.

This article explores validators and the importance of validator client diversity. From there, we transition into why Jump is building a new validator client and how their experience in high-frequency trading positions them as the perfect team to develop Firedancer. Then, we'll explain what Firedancer is, how it works, why it's fast, how it's secured, and its current status.

By the end of this article, you will have a thorough understanding of Jump's new validator client. You will be familiar with their groundbreaking optimizations, making it the most performant validator client on any

blockchain. You will also understand why Firedancer is essential to the performance and reliability of the Solana network. This is the only article you need to learn about Firedancer.

The Appendix section contains an entirely optional primer on computer hardware and networking. It was added to provide the necessary context for the average reader so they can understand the more advanced hardware and networking concepts covered within this article. Nevertheless, context is provided where necessary. This article aims to be accessible so anyone using Solana can understand Firedancer and its significance.

What are Validators and What is Validator Client Diversity?

A validator is a computer that participates in a Proof of Stake blockchain. Validators form the backbone of Solana's network. They are responsible for processing transactions and participating in consensus. A validator helps secure the network by locking up a certain amount of Solana's native token as its stake. Think of it as a security deposit, financially exposing the validator to the network. This exposure incentivizes validators to perform their tasks accurately and efficiently as they receive rewards for their contributions. Validators are also penalized for malicious or faulty activity. A validator's stake is reduced for improper behavior in a process known as slashing. Thus, it is in a validator's best interest to perform their duties properly to grow their stake.

Validator clients are applications that validators use to perform their duties. The client is a base for validators, using their cryptographically unique identity to participate in consensus.

Having multiple clients distinct from one another improves fault tolerance in the event that one implementation fails. For instance, if no client controls greater than 33% of stake, a crash or a bug impacting liveness won't bring down the network. Similarly, if there's a bug in a client that leads to an invalid state transition, the network can avoid a safety failure if less than 33% of stake is using that client. This is because most of the network will remain on a valid state, preventing a split or fork in the blockchain. Thus, validator client diversity leads to greater network resilience because a bug or vulnerability in one client will not debilitate the entire network.

Client diversity is measured by the percentage of stake running through each client and the total number of clients available. At the time of writing this article, there are 1979 validators on the Solana network The two clients that these validators use on mainnet are provided by Solana Labs and Jito Labs. Solana launched with one validator client in March of 2020, developed by Solana Labs. In August 2022, Jito Labs released a second validator client. This client is a fork of the Solana Labs code maintained and deployed by Jito. The client optimizes the extraction of MEV (maximal extractable value) within blocks. Jito's client creates a pseudo-mempool since Solana streams blocks without one. As an aside, a mempool, or memory pool, is a backlog of pending and unconfirmed transactions. A pseudo-mempool allows validators to search through these transactions, bundle them together optimally, and submit them to Jito's Block Engine.

As of October 2023, the Solana Labs client holds 68.55% of the active stake, whereas Jito holds 31.45%. The number of validators using Jito's client is up 16% from the Solana Foundation's previous Health Report. The growing use of Jito's client indicates a positive trend toward client diversity.

While this news of growth is exciting, it isn't perfect. It is essential to emphasize that Jito's client is a fork of the Solana Labs client

. This means that Jito shares many components with the original validator codebase and is potentially vulnerable to bugs or exploits that would affect the Labs client. In an ideal future, Solana would have at least four independent validator clients. Different teams would build these clients in different programming languages. No single implementation would have more than 33% of the stake, as each client would hold ~25%. This idealized setup would lack a single point of failure in the entire

validator stack.

Developing a second independent validator client is vital to realizing this future, and Jump is dedicated to making it happen.

Why is Jump Building a New Validator Client?

Solana's mainnet has faced four block production halts in the past. Each required manual fixes by hundreds of validators. These outages have spotlighted concerns about Solana's network reliability. Jump contends the protocol is sound. Instead, they attribute the downtime to software module issues affecting consensus. So, Jump is developing a new validator client to address these issues. The overall goal of this client is to enhance Solana's network stability and efficiency.

The development of an independent validator client is a difficult task. However, this isn't the first time Jump has built a reliable global network. In the past, security transactions (i.e., buying and selling stock) were executed by market specialists manually. With the onset of electronic trading platforms, security exchanges became more open. This openness increased competition, automation, and reduced the time and cost for investors to trade. A technological arms race ensued among market specialists.

Traders live to trade. The optimal trading experience requires no-compromise in software, hardware, and networking solutions. These systems must have high machine intelligence, low real-time latency, high throughput, high adaptability, high scalability, high reliability, and high accountability.

Commodity solutions (i.e., software a firm can buy outright) are not a competitive advantage. Sending the correct order to an exchange ten times in second place is an expensive way to lose money. Staunch competition in high-frequency trading leads to a perpetual development cycle, building top-tier global trading infrastructure.

This scenario might sound familiar. The demands of a successful trading system resemble those of a successful blockchain. Blockchains need to be performant, fault tolerant, low latency networks. A slow blockchain is a piece of failed technology that cannot meet the demands of modern enterprise applications - it does nothing more than hamper innovation, scalability, and real-world utility. With over two decades of experience scaling global networks and developing high-performance systems, Jump is the perfect team to create an independent validator client. Kevin Bowers, Jump Trading's Chief Science Officer, is overseeing this process.

Why is the Speed of Light too Slow?

Kevin Bowers has spoken at length about the speed of light being too slow. The speed of light is a finite constant that provides a natural limitation to the number of computations a single transistor can process. Bits are currently modeled by electrons traveling through transistors. The Shannon Capacity Theorem (i.e., a maximum amount of error-free data that can be sent across a channel) restricts the number of bits transmitted through a transistor. Due to fundamental physics and information theory, computation speed is limited by how fast an electron can move through matter and how much data can be sent. These constraints become evident when pushing supercomputers to their limits. As a result, there is a "dramatic mismatch between the ability of computers to crunch numbers and their ability to move numbers around."

Take an Intel Core i9 13900K CPU as an example. It has 24 x86 cores with a base clock of 2.2 GHz and a peak turbo clock of 5.8 GHz. The worst-case scenario is that light would have to travel a total distance of ~52.0 mm across this CPU. The CPU's Manhattan Distance (i.e., the distance between two points measured along axes at right angles) is ~73.6 mm. At the CPU's peak turbo clock speed of 5.8 GHz, light can travel ~51.7 mm in air. This means that a signal can almost

make a single round trip between any two points on the CPU during a single clock cycle.

The reality of the situation is much worse. These measurements use the speed of light traveling through air, while these signals travel through silicon dioxide (SiO2

). Light can travel ~26.2 mm in silicon dioxide in a 5.8 GHz clock cycle. Light in silicon (Si) can only travel ~15.0 mm in a 5.8 GHz clock cycle - a little over half the long edge of the CPU.

The Firedancer team argues recent technological advances in computing have been about fitting more cores into a CPU rather than making them faster. When people needed more performance, they were encouraged to buy more hardware. This works for the time being when throughput is the bottleneck. The actual bottleneck is the speed of light. This natural restriction leads to a paralysis in decision-making. There isn't an immediate payoff to any one optimization as a system has many components, and none are well-optimized. The parts that do not get optimized will worsen over time because they will have less compute available. So, what now?

In the world of high-performance computing, everything must be optimized eventually

. The result is building systems for production trading and quantitative research that run at the limits of physics and information theory on a planetary scale. This includes creating custom network switching technology to lock-free algorithms designed with these physical limits in mind. Jump is as much a technology firm as a trading firm. At the cutting edge of science fiction and reality, with the striking similarities between the problems that both Jump and Solana currently face, Jump is developing Firedancer.

What is Firedancer?

Firedancer is a new, fully independent validator client developed by the Firedancer team in the C programming language. Firedancer is built with reliability in mind due to its modular architecture, minimal dependencies, and extensive testing process. It proposes a major rewrite of the three functional components of the Solana Labs client: Networking, Runtime, and Consensus. Each level is optimized for maximum performance so the client will operate at a capacity only limited by the validator's hardware. This differs from the performance limits validators currently face due to software inefficiencies. With Firedancer, Solana will scale with bandwidth and hardware.

The goals of Firedancer are to:

- Document and standardize the Solana protocol (in the end, a person should be able to create a Solana validator just by looking at the documentation and not the Rust validator code)
- Increase validator client diversity
- Improve ecosystem performance

How Does Firedancer Work?

Modular Architecture

Firedancer sets itself apart from current Solana validator clients with its distinctive modular architecture. Unlike the Solana Labs Rust validator client, which operates as a single process, Firedancer is composed of numerous individual Linux C processes known as tiles. A tile is a process and some memory. This tile architecture is fundamental to Firedancer's operational philosophy and approach to robustness and efficiency.

A process is an instance of a running program. It is a fundamental component of modern operating systems and represents the execution of a set of instructions. Each process has its own memory space and resources the operating system allocates and operates on independently of other processes. A process is like an independent worker in a large factory, handling a specific task with their own tools and workspace.

In Firedancer, each tile is an individual process with a defined role. For example, the QUIC tile is responsible for processing incoming QUIC traffic and forwarding the encapsulated transactions into the verify tile. The verify tile is responsible for signature verification, and so on for each tile. These tiles operate independently and concurrently, contributing to the system's overall functionality.Individual Linux processes allow for small, independent failure domains. This means that issues in one tile have a minimal impact - or a small "blast radius" - on the overall system. This approach differs from the Solana Labs Rust client as a single point of failure does not have the potential to compromise the entire validator instantly.

A key advantage to Firedancer's architecture is its ability to replace and upgrade each tile within seconds without any downtime. This capability starkly contrasts the Solana Labs Rust client's requirement for a complete shutdown ahead of upgrades. The difference stems from Rust's lack of ABI (Application Binary Interface) stability. This prevents on-the-fly upgrades in a pure Rust environment. The use of C processes, benefiting from the binary stability in the C runtime model, significantly reduces upgrade-related downtime. It is able to do this since the tiles manage the validator state in different workspaces. These shared memory objects persist as long as the validator is powered on. Each tile can seamlessly pick up processing from where it left off during a restart or upgrade.

Overall, Firedancer is built according to a NUMA-aware, tile-based architecture. We'll cover what this means in the next section. For now, this means that it provides dedicated hardware resources per thread. In this architecture, 1 CPU core is used per tile. It boasts high-performance message passing between the tiles, optimized for memory locality, resource layout, and component latencies.

Network Processing

Firedancer's network processing is designed to handle the intensive demands of the Solana network as it scales up to gigabit-per-second speeds. This process is split into incoming and outgoing activities.

Inbound activities primarily revolve around receiving transactions from users. Firedancer's performance is essential because consensus messages can get lost if a validator falls behind on packet processing. The current operational bandwidth for a Solana node is ~0.2 Gbps while the largest recorded spike on a Jump node was ~40 GBps. This spike in bandwidth showcases the need for a robust and scalable ingress processing solution.

Outbound activities include block packing, block creation, and sending shreds. Each of these steps is crucial for the secure and efficient operation of the Solana network. The performance of these tasks not only affects throughput but also the overall reliability of the network.

Firedancer aims to address historical weaknesses in Solana's peer-to-peer interface for processing transactions. A major shortcoming of Solana's peer-to-peer interface in the past was its lack of congestion control for incoming transactions. This shortcoming led to significant network outages on September 14th, 2021 (17 hours) and April 30th, 2022 (7 hours).

In response, Solana has made several network upgrades to properly handle high transaction loads. Firedancer follows suit by adopting QUIC for its flow control. QUIC is a multiplexed transport network protocol that forms the basis ofHTTP/3. It is instrumental in DDoS protection and managing network traffic. However, it is crucial to note that in some cases, the costs outweigh the benefits. QUIC, coupled with data centers' specialized hardware for mitigating DDoS attacks, removes the incentive behind transaction flooding.

QUIC's 151-page specification brought considerable complexity to development. Unable to find an existing C library that meets their licensing, performance, and reliability needs, the Firedancer team has built their own implementation Firedancer's QUIC implementation, nicknamed fd_quic, introduces optimized data structures and algorithms to ensure minimal memory allocation and prevent memory exhaustion.

Firedancer's custom networking stack is at the core of its processing capabilities. The stack was designed from scratch to leverage receive-side scaling (RSS). RSS is a form of hardware-accelerated network load balancing that distributes network traffic across various CPU cores to increase network processing parallelism. Each CPU core handles a portion of incoming traffic with minimal overhead. This approach outperforms traditional software-based load balancing by eliminating the need for complex schedulers, locks, and atomics.

Firedancer introduces a new message-passing framework for composing an application of highly-performant tiles. These tiles can bypass kernel networking, which is limited due to its socket-based nature, by utilizing AF_XDP. AF_XDP is an address family that is optimized for high-performance packet processing. Using AF_XDP enables Firedancer to read directly from network interface buffers.

This tile system facilitates various high-performance computing concepts in the Firedancer stack. This includes:

- [NUMA Awareness

](https://en.wikipedia.org/wiki/Non-uniform_memory_access?ref=jumpcrypto.com) - NUMA (Non-Uniform Memory Access) is a computer memory design where a processor can access its own memory faster than memory associated with another processor. For Firedancer, being NUMA-aware means the client can handle memory in multiprocessor configurations efficiently. This is important for high-volume transaction processing as it optimizes the use of available hardware resources.

- [Cache Locality

](https://en.wikipedia.org/wiki/Memory_hierarchy?ref=jumpcrypto.com) - Cache locality refers to using data already in a cache close to the processor. This is typically a convoluted form of temporal locality (i.e., recently accessed data). In Firedancer, a focus on cache locality means it is designed to process network data while minimizing latency and maximizing speed.

- [Lockless Concurrency

](https://en.wikipedia.org/wiki/Non-blocking_algorithm?ref=jumpcrypto.com#Lock-freedom) - Lockless concurrency refers to designing algorithms that do not require locking mechanisms (such as mutexes) to manage simultaneous operations. For Firedancer, lockless concurrency allows multiple network operations to occur in parallel without causing delays due to locks. Lockless concurrency enhances Firedancer's ability to process a high number of transactions concurrently.

- [Large Page Sizes

](https://en.wikipedia.org/wiki/Page_(computer_memory)?ref=jumpcrypto.com#Multiple_page_sizes) - Using large page sizes in memory management helps handle datasets by reducing page table lookups and potential memory fragmentation. For Firedancer, this means improved memory handling efficiency. This is beneficial for processing large volumes of network data.

Build System

Firedancer's build system is designed with a set of guiding principles in mind to guarantee reliability and consistency. It emphasizes minimizing external dependencies and treating all tools involved in the build process as dependencies themselves. This includes pinning every dependency, including compilers, to exact versions. A critical aspect of this system is its environmental isolation during build steps. Environment isolation enhances portability as the build process remains unaffected by the system environment.

How is Firedancer so Fast?

Advanced Data Parallelism

Firedancer's approach for cryptographic tasks such as ED25519 signature verification uses advanced data parallelism available inside modern processors. Modern CPUs have Single Instruction, Multiple Data (SIMD) instructions for processing multiple data elements simultaneously and optimizations for running multiple instructions per CPU cycle. It is generally more efficient in area, time, and power to have a single instruction operate on an array or vector of data elements in parallel. In this respect, parallel data processing improvements can dominate throughput compared to improvements in sheer processing speed.

One area where Firedancer uses data parallelism is to optimize computing signature verifications. This approach allows for handling arrays or vectors of data elements simultaneously to maximize throughput and minimize latency. At the heart of this ED25519 implementation is Galois Fields arithmetic. This form of arithmetic is well-suited for cryptographic algorithms and binary computations. In Galois Fields, operations like addition, subtraction, multiplication, and division are defined in a way that aligns with the binary nature of computer systems. Here is an example of a Galois Field defined by 23

:

The only issue is that ED25519 uses a Galois Field defined by 2255-19

. Think of the field elements as numbers from 0 to 2255-19

. This is what basic operations look like:

- x + y → grade school addition mod 2255-19
- x - y → grade school subtraction mod 2255-19
- x * y → grade school multiplication mod 2255-19
- 1/x → x to the 2255-21

mod 2255-19

Addition, subtraction, and multiplication are almost uint256_t

math (i.e., math with unsigned integers, where the largest value is 2256-1

). Division is challenging to compute. Commodity CPUs and GPUs don't do uint256_t math, much less "almost uint256_t math", and much less exceptionally hard weird division. Implementing this kind of math and making it highly performant is a question of how well we can emulate this kind of math.

Firedancer's implementation breaks down arithmetic by thinking of numbers more flexibly. If we apply the principles of grade school long division and multiplication, where we carry numbers from one column to the next, we can process these columns in parallel. The fastest way to emulate this kind of math is to represent a uint256_t as six 43-bit digits with a 9-bit "carry." This allows for existing 64-bit operations on CPUs while providing enough space for carry bits. This arrangement of numbers reduces the need for frequent carry propagation and allows Firedancer to handle large numbers more effectively.

This implementation leverages data parallelism by reorganizing the arithmetic computations into parallelized column sums. Processing columns in parallel accelerates computation overall as it turns what would have been a sequential bottleneck into a parallelizable task. Firedancer also uses vectorized instruction sets such as AVX512 and its IFMA extension (AVX512-IFMA). These sets allow for the processing of the Galois Fields arithmetic explained above, thereby enhancing speed and efficiency.

Firedancer's AVX512 accelerated implementation is fast

. On a single 2.3 GHz Icelake-server core, it is over double the performance per core clock than their 2022 Breakpoint demo. The implementation boasts 100% vector lane utilization and massive data parallelization. This is another masterful demonstration by the Firedancer team that, due to speed-of-light delays, it is much easier to do independent things in parallel than it is to do one thing at a time, even with custom hardware thrown at it.

Harnessing FPGAs for High-Speed Network Communications

CPUs can handle ~30,000 signature verifications per second per core. While they are an energy-efficient option, they fall short in large-scale operations. This limitation stems from their sequential processing approach. GPUs escalate this processing capability to ~1 million verifications per second per core. However, they are hampered by their substantial power consumption of around ~300W per unit and inherent latency due to batch processing.

FPGAs emerge as the superior alternative. They match the throughput of GPUs but with significantly reduced power consumption, around 50W per FPGA. Its latency also undercuts the ten milliseconds latency of GPUs. FPGAs offer a much more responsive solution for real-time processing at a latency of ~200 microseconds. Unlike the batch processing of GPUs, FPGAs in Firedancer process each transaction individually in a streaming fashion. Firedancer's use of FPGAs translates to an impressive throughput of 8 million signatures per second on a power budget of less than 400 W for 8 FPGAs.

The team showcased Firedancer's ED25519 signature verification process at Breakpoint 2022. This process involved several stages, including SHA-512 computation in a pure RTL pipeline with various checks and computations in a custom ECC-CPU processor pipeline. Basically, the Firedancer team wrote a compiler and an assembler for their custom processor, took the Python code from the RFC (Request for Comments), ran it with operator-overloaded objects to generate machine code, and then put the machine code on top of the ECC-CPU.

It is important to note that Firedancer uses an AWS accelerator form factor style to balance robustness with network connectivity. This selection addresses the challenges associated with direct network connectivity - a feature often limited by cloud providers. With this selection, Firedancer ensures seamless integration of its advanced capabilities within the constraints of cloud-based infrastructure.

It is crucial to recognize that different operations require tangible physical space, not just conceptual data space. Firedancer applies this understanding by strategically arranging physical components to be close and reusable. This configuration allows Firedancer to maximize the efficiency of its FPGA, achieving 8m TPS with a 7-year-old FPGA on an 8-year-old machine.

Optimizing Reed-Solomon Coding for Network Communications

The fundamental challenge in network communications is broadcasting new transactions around the world. The point-to-point nature of the Internet, finite amounts of bandwidth, and latency issues limit implementing traditional ideas such as using the network for direct broadcasting. Distributing data in a ring or tree structure partly addresses these issues but falls short as data packets can get lost during transmission.

Reed-Solomon coding is an elegant solution to these problems. It introduces data transmission redundancy (i.e., parity information) to recover lost packets. The concept is based on the principle that two points define a line, and any two points on this line can regenerate the original data points. Building a polynomial based on the data points and distributing different points of this function in separate packets can reconstruct the original data as long as the recipient receives at least two packets.

We build a polynomial because using the traditional formula for points on a line ($y = mx + b$) is slow computationally. Firedancer uses Lagrange Polynomials, a specialized method for polynomial construction, to speed things up. They simplify the process of creating the polynomial needed for Reed-Solomon coding. It also turns the process into a more efficient matrix-vector product that works for higher-order polynomials. This matrix is highly structured, with its patterns repeating recursively, such that the first row of this pattern entirely determines it. This structure means there is a faster way to multiply everything. Firedancer uses an O(n log n) approach presented in a 2016 article for multiplying by this matrix, the fastest known theoretical approach for Reed-Solomon coding. The result is the efficient computation of parity information compared to traditional methods:

- Over ~120 Gbps/core RS encoding
- Up to ~50 Gbps/core RS decoding
- These metrics are all compared to the current ~8 Gbps/core RS encoding (rust-rse)

Firedancer can compute parity 14x faster than traditional methods using this optimized approach to Reed-Solomon coding. This results in a fast, reliable data encoding and decoding process, which is essential for maintaining high throughput and low latency on a global scale.

How is Firedancer Secured?

Opportunities

All validators currently use software based on the original validator client. Firedancer can improve Solana's client and supply chain diversity if they are distinct from the Solana Labs client. This includes using similar dependencies and using Rust to develop their client.

The Solana Labs and Jito validator clients run as a single process. Adding security to a monolithic application is difficult once it runs in production. Validators running these clients would have to shut down for on-the-fly security upgrades in pure Rust. The Firedancer team can build secure architecture with their new client from the start.

Firedancer also has the bonus of learning from experience. Solana Labs developed the validator client in a startup environment. This fast-paced environment meant Labs needed to move fast to get to market quickly. This set them up poorly for future development. The Firedancer team can look at what Labs did and what teams on other chains have done and ask what they would do differently if they could develop a validator client from scratch.

Challenges

Despite being distinct from the Solana Labs client, Firedancer has to replicate its behavior closely. Failure to do so is a security concern as it could introduce consensus bugs due to incompatibility. This can be mitigated by incentivizing a percentage of stake to run on both clients, keeping Firedancer under 33% of the total stake for an extended period. Regardless, the Firedancer team needs to implement the complete feature set of the protocol, irrespective of how hard it is to implement correctly or securely. Everything must be aligned with Firedancer. Thus, the team cannot develop the code in isolation and must review it against the functionality of the Labs client. This is exacerbated by a lack of specification and documentation, meaning Firedancer has to introduce inefficient constructions from the protocol.

The Firedaner team must also be mindful that they are developing their new client in C. C does not have the memory safety guarantees that languages such as Rust provide natively. A primary goal of the Firedancer codebase is to reduce the occurrences and impact of memory safety vulnerabilities. Special attention needs to be paid to this goal as Firedancer is a project that moves fast. Firedancer must find a way to keep up development speed without introducing such bugs. OS sandboxing is the practice of isolating the tiles from the OS. Tiles are only allowed to access resources and perform system calls required for their job. Since tiles have a clearly defined purpose and the Firedancer team developed most of the client code, a tile's permissions are stripped down according to the Principle of Least Privilege.

Implementing Design for Defense in Depth

All software will have a security vulnerability at some point. Working backward from the premise that software will have bugs, Firedancer opts to limit any one vulnerability's potential impact. This approach is called defense in depth. Defense in Depth is a strategy that uses various security measures to protect an asset. If an attacker compromises one part of the system, additional measures exist to stop the threat from affecting the entire stack.Firedancer is designed to mitigate the stack between the vulnerability and exploit stage. For example, it would be hard for an attacker to exploit a memory safety vulnerability.

This is because preventing these kinds of attacks is a well-studied problem. The well-researched nature of memory safety in C has led to an arsenal of hardening techniques and compiler features the team uses for Firedancer. Even if an attacker could bypass industry best practices, it would be hard for the exploit to compromise the system. This is due to tile isolation and OS sandboxing.

Tile isolation is the outcome of Firedancer's parallel architecture. Each tile has a clear, single purpose since they run their own Linux process. For example, a QUIC tile is responsible for processing incoming QUIC traffic and forwarding the encapsulated transactions into the verify tile. Then, the verify tile is responsible for signature verification. Communication between the QUIC and verify tiles is done through a shared memory interface (i.e., Linux processes can pass data between one another). This shared memory interface between two tiles acts as an isolation boundary. If the QUIC tile had a bug allowing an attacker to execute arbitrary code when it processed a malicious QUIC packet, it wouldn't affect any other tile. In a monolithic process, this would be an instant compromise. An attacker could cause harm to the entire network if they exploit this vulnerability on multiple validators. An attacker may be able to degrade the QUIC tile's performance, but Firedancer's design limits them to the QUIC tile only.

OS sandboxing is the practice of isolating the tiles from the OS. Tiles are only allowed to access resources and perform system calls required for their job. Since tiles have a clearly defined purpose and almost all of the code was developed by the Firedancer team, a tile's permissions are stripped down according to the Principle of Least Privilege. Tiles are placed within their own Linux namespaces, providing a limited view of the system. This narrow view prevents a tile from accessing most of the file system, the network, and any other process running on the same system. Namespaces provide a security-first boundary. This, however, can still be bypassed if an attacker has a kernel exploit for privilege escalation. The system call interface is the last attack vector in the kernel that is reachable from tiles. To protect against this, Firedancer uses seccomp-BPF to filter system calls before the kernel processes them. The client can restrict tiles to a select group of system calls. In some cases, the parameters of syscalls can be filtered. This is important because Firedancer can ensure that read and write syscalls only operate on specific file descriptors.

Implementing an Embedded Security Program

Firedancer is designed with a comprehensive security program ingrained at every stage of its development. The client's security program is an ongoing collaboration between the development and security teams, setting a new standard for secure blockchain technology.

The process begins with self-service fuzzing infrastructure. As an aside, fuzzing is a technique that automatically detects crashes or error conditions that indicate vulnerabilities. This is done by stress-testing every component that accepts untrusted user inputs, including the P2P interface (parsers) and the SBPF virtual machine. OSS-Fuzz maintains continual fuzz coverage throughout code changes. The security team has also set up a dedicated ClusterFuzzer instance for continuous coverage-guided fuzzing. Developers and security engineers also contribute fuzzing harnesses (i.e., special versions of unit tests for security-critical components). Developers can also contribute new fuzz tests, which are picked up and tested automatically. The goal is that all parts are fuzzed heavily before they move into the next stage.

Internal code reviews help identify bugs that the tooling has missed. The focus at this stage is on high-risk, high-impact components. This stage is a feedback mechanism that feeds into the rest of the security program. The team applies all their lessons and uses these reviews to improve fuzzing coverage, introduce new static analysis checks for certain bug classes, and even implement large code refactoring to design their way out of complex attack vectors. External security reviews will supplement these internal reviews by leading industry experts and an active bug bounty program, both pre- and post-launch.

Firedancer also underwent extensive stress testing on various test networks. These test networks will be subject to attacks and failures such as duplicating nodes, failing network links, packet floods, and consensus violations. These networks face significantly higher loads than any realistic scenario on mainnet.

So, this leads us to the question: what's the current status of Firedancer?

What is the Current Status of Firedancer and What's Frankendancer?

The Firedancer team is developing Firedancer incrementally to modularize the validator client. This aligns with their goals for documentation and standardization. This approach ensures Firedancer stays up to date with Solana's latest developments. This led to the creation of Frankendancer. Frankendancer is a hybrid client model where the Firedancer team integrates its developed components into the existing validator client infrastructure. This development process allows for the gradual improvement and testing of new features.

Frankendancer is like putting a sports car in the middle of traffic. Performance will grow as more components are developed, and bottlenecks are removed. This modular development process fosters a customizable and flexible validator environment. Here, developers can modify or replace specific components in their validator client to suit their needs.

What's Actually Running?

Frankendancer implements all of a Solana validator's networking features:

- Inbound: QUIC, TPU, Sigverify, Dedup
- Outbound: Block packing, create/sign/send shreds (Turbine)

Frankendancer is using Firedancer's high-performance C networking code on top of the Solana Labs Rust runtime and consensus code.

The architecture of Frankendancer is designed with an eye toward high-end hardware optimization. While it supports low-end vanilla cloud hosts running standard Linux operating systems, the Firedancer team is optimizing Frankendancer for high-core-count servers. The long-term goal is to leverage hardware already available in the cloud to enhance efficiency and performance. The client supports multiple connections simultaneously, hardware acceleration, randomized flow steering for load distribution (i.e., ensure evenly distributed network traffic), and numerous process boundaries for extra security between components.

Technical efficiency is a cornerstone of Frankendancer. The system avoids memory allocations and atomic operations in the critical path, with all allocations being NUMA-optimized at initialization. This design ensures maximum efficiency and performance. Furthermore, the ability to inspect system components asynchronously and remotely, coupled with the flexibility in managing tiles (starting, stopping, and restarting asynchronously), adds a layer of robustness and adaptability to the system.

How Well Does it Perform?

Frankendancer can process 1,000,000 transactions per second (TPS) per tile on the network inbound side. This performance scales linearly with the number of cores used since 1 CPU core is used per tile. Frankendancer achieved this feat by utilizing four cores only, maxing out a 25 Gbps network interface card (NIC).

Frankendancer has made significant improvements in network outbound operations with its Turbine optimizations. Today's standard node hardware achieves a speed of 6 Gbps per tile. This includes substantial speed improvements in shredding (i.e., how block data is split up and sent across the network to validators). Compared to standard Solana nodes today, Frankendancer shows a shredding speed increase of ~22% without Merkle trees and almost double with Merkle trees. This is a massive improvement on the current block propagation and transaction ingestion performance by today's validators.

Firedancer's network performance demonstrates it has reached the hardware limit. It has achieved the maximum performance possible with today's standard validator hardware. This marks a significant technical milestone, displaying the client's ability to handle extreme workloads efficiently and effectively.

Frankendancer is Live on Testnet

Frankendancer is currently staked, voting, and producing blocks on testnet. It is co-existing compatibly with ~2900 other Solana Labs and Jito validators. This live deployment demonstrates Firedancer's robust performance on commodity hardware. It is currently on an Equinix Metal m3.large.x86 server is powered by an AMD EPYC 7513 CPU. Many other validators use the same server type. It offers a cost-effective solution with on-demand pricing that varies by location. Rates range from $3.10 to $4.65 per hour.

Firedancer's progress toward its mainnet launch opens up several possibilities for node hardware:

- Current validator hardware can lead to much higher performance capacity per node
- Firedancer's efficiency allows validators to use more affordable, lower-specification hardware while maintaining similar performance levels
- Firedancer's design positions it to leverage advancements in hardware and bandwidth

These developments, along with other initiatives such as Wiredancer (i.e., the Firedancer team's experiments with hardware

acceleration) and a modular Rust-based Runtime/SVM, position Firedancer as a forward-looking solution.

Firedancer's progress also opens up discussions regarding the potential for validators to run the Solana Labs client alongside Firedancer in a process known as side-caring. This approach could maximize network liveliness by leveraging the strengths of both clients and mitigating the potential impact of issues in either client to the overall network. Furthermore, this also opens up speculation about whether projects like Jito would consider forking Firedancer. This could lead to further optimizations in MEV extraction and transaction processing efficiency. Only time will tell.

Conclusion

Developers typically conceptualize operations as taking up data space rather than physical space. With the speed of light as a natural constraint, this assumption leads to slow systems that fail to optimize their hardware correctly. In a highly adversarial and competitive landscape, we must refrain from throwing more hardware at Solana and expecting it to perform better. We need to optimize. Firedancer revolutionizes how validator clients are structured and expected to operate. By building a reliable, highly modular, and performant validator client, the Firedancer team is preparing Solana for mass adoption.

Whether you are an entry-level developer or the average Solana user, it is vital to understand Firedancer and its significance. This technological feat makes the fastest, most performant blockchain currently on the market even better. Solana is designed to be a high throughput, low latency global state machine. Firedancer is a giant leap forward towards perfecting these goals.

If you've read this far, thank you, anon! Be sure to enter your email address below so you'll never miss an update about what's new on Solana. Ready to dive deeper? Join our Discord to start building the future on the most performant blockchain, today.

Additional Resources / Further Reading

- Jump's Website
- Firedancer GitHub Repository
- Breakpoint 2023: Firedancer Update
- Breakpoint 2023: Securing Firedancer
- Breakpoint 2023: Fast Reed-Solomon Coding For Network Communications
- Breakpoint 2023: FPGA Working at 8m TPS
- Firedancer's fd_quic Technical Milestone
- Solana Network Upgrades

Appendix

An Introduction to Computer Hardware and Networking

A computer is a machine that can be programmed to carry out sequences of arithmetic or logical operations automatically. These operations range from automating basic calculations to complex data processing. At its core, a computer blends hardware and software components to execute instructions and manage data. The hardware comprises the physical components, and the software includes the programs and operating systems that instruct the hardware on how to operate.

Useful computation generally involves four resources: compute, memory, disk storage, and networking. The compute aspect, primarily handled by CPUs, GPUs, and potentially FPGAs, is incredibly fast, capable of performing billions of operations per second. Accessing RAM is generally slower than performing a computation in the CPU. Disk storage provides a long-term storage solution useful for computation. However, accessing data from Solid State Drives (SSDs) and Hard Disk Drives (HDDs) is much slower than CPU operations. SSDs are often thousands of times slower, and HDDs are tens of thousands of times slower. Moreover, the network, including the Internet and local networks, is the slowest. It can be upwards of a million times slower than the CPU.

Understanding these differences in speed is crucial for grasping the design principles and efficiency considerations in high-performance computing applications such as Firedancer. The CPU's calculation speed sets the baseline, with memory, disk storage, and network access each adding a layer of delay in decreasing order of speed.

Central Processing Unit (CPU)

The CPU, or Central Processing Unit, is the cornerstone of a computer's functionality - it is the machine's brain. The CPU executes software instructions, performs calculations, and makes decisions based on the input it receives. It works by

processing binary signals, which are sequences of zeros and ones. Each unique series of binary codes corresponds to a specific instruction that the CPU interprets and acts upon. These instructions are handled sequentially, with the CPU completing one operation before moving on to the next. This sequential processing of instructions is fundamental to the CPU's role, determining how it performs complex calculations and makes decisions based on the input it receives.

Modern CPUs are often multi-core. This means they contain multiple processing units within a single chip, known as cores. Each core can execute instructions independently, allowing for the parallel processing of tasks. Multi-core architecture enhances the CPU's ability to handle operations simultaneously, significantly improving overall performance. However, it is important to note that operations are still processed sequentially within each core. This makes CPUs well-suited for complex, sequential tasks but inefficient for larger volumes of more straightforward, parallel tasks.

CPUs also have caches - small, high-speed memory units within the CPU. These caches store frequently accessed data and instructions, allowing quicker retrieval times than fetching data from the RAM. There are typically multiple levels of caches (L1, L2, L3, and sometimes L4), each with varying sizes and speeds. The L1 cache, being the smallest and fastest, is accessed first. The CPU checks the larger, slightly slower L2 cache if the required data is not found there, and so on. This hierarchical caching system reduces the CPU's wait time for data from the RAM and enhances the overall processing speed. Efficient cache usage has profound implications for an application such as Firedancer, where the distance from the CPU to RAM can hinder performance. This is particularly relevant for our discussions on the speed of light and using FPGAs.

As an aside, the term "x86" refers to a family of CPUs that follow a specific architecture initially developed byIntel. This architecture is known for its compatibility with a wide range of software, as most desktop and laptop computers sold are based on the x86 architecture family.

Graphics Processing Unit (GPU)

The GPU is a specialized processor initially developed to accelerate image and video rendering for computer graphics. Its primary function is to manage and enhance graphic performance, particularly in tasks requiring high-resolution visuals and complex graphic calculations such as video games or 3D modeling.

Over time, the GPU has evolved beyond its original purpose. Due to its architecture, GPUs have become invaluable in a broader range of data processing tasks. Its capability for efficient parallel processing suits it for applications that require handling large data sets simultaneously. In blockchain technology, GPUs are widely used for cryptocurrency mining. They excel in this role because they can process parallel workloads of cryptographic calculations more efficiently than a CPU.

Random Access Memory (RAM)

RAM is a computer's short-term memory that stores data actively being used or processed. This is where the CPU "remembers" what it is currently working on and keeps relevant information for quick access and processing.

Error-Correcting Code (ECC) RAM has the ability to detect and correct common types of internal data corruption. This feature is vital in environments where data accuracy is important, such as scientific computing, financial transactions, or blockchain nodes. ECC RAM can automatically identify and fix minor errors in the data it stores. This error correction process is performed through additional hardware in the RAM modules that checks the stored data.

Non-ECC RAM is the more commonly used type of memory in standard computers and consumer devices. While it lacks the error-correcting capabilities of ECC RAM, it is generally faster and less expensive. Non-ECC RAM is preferred for general use due to its cost-effectiveness and the relatively low risk of data errors in standard desktop computing.

Disk Storage

Disk storage is responsible for long-term data retention, even when a computer is turned off. There are two primary types of disk storage: Hard Disk Drives (HDDs) and Solid State Drives (SSDs).

HDDs are older disk drives that store data on magnetic disks known asplatters. The platters are paired with magnetic heads, usually attached to a moving actuator arm. A read/write head on this arm accesses the data while the disk spins. The mechanical nature of HDDs - spinning disks and moving heads - makes them relatively slower than SSDs. However, they offer more storage capacity for a lower price. This makes them cost-effective for bulk storage.

On the other hand, SSDs use flash memory to store data. Flash memory is an electronic, non-volatile storage solution that can be erased and reprogrammed. The use of flash memory means that, unlike HDDs, there are no moving parts involved. Instead, interconnected flash memory chips store the data. SSDs are faster than HDDs because they can access data instantly without waiting for a disk to spin or a read/write head to find the data. This speed makes SSDs an excellent choice for applications where quick data retrieval is crucial. However, this speed comes at a higher cost per gigabyte than HDDs.

There are also NVMe (Non-Volatile Memory Express) SSDs. These SSDs are designed to exploit their high-speed potential through a computer's Peripheral Component Interconnect Express (PCIe) bus. NVMe drives offer significantly higher speeds and lower latency than traditional SSDs. They are ideal for intensive workloads, such as high-frequency trading and blockchain applications, where rapid data processing and retrieval are critical. While NVMes come at a premium price, they

are starting to become the standard for high-performance computing.

Motherboard

A computer's motherboard is a large circuit board that interconnects and facilitates communication between a computer's parts. These parts include the CPU, GPU, RAM, storage devices, and peripheral devices such as keyboards and mice.

The motherboard is the central hub of activity. It ensures that each component can communicate effectively with one another. It is also vital in power distribution, channeling electricity from the power supply to each major part. The motherboard guarantees that each component receives the appropriate energy they need to operate.

The motherboard is responsible for managing data flow within the system. It oversees the routing of data from the CPU to RAM for processing, from the RAM to storage devices for saving, and from the GPU to the display outputs for visual renderings. In addition to this, the motherboard also houses the system's BIOS (Basic Input/Output System). The BIOS is integral to system control and monitoring. The BIOS initializes and tests a computer's hardware during startup. It also keeps tabs on system health, monitoring factors such as temperature, voltage, and fan speeds.

In the context of Firedancer's design, the motherboard's architecture plays a pivotal role. The proximity of the CPU to RAM impacts performance significantly, as shorter distances between them enhance data transfer speeds. This distance is essential for applications like Firedancer that demand high throughput and low latency. This is why NUMA-aware architecture and the potential use of FPGAs align with Firedancer's need for optimized memory allocation and processing efficiency. We'll cover what it means to be NUMA-aware and what FPGAs are later in this article.

Operating Systems, Virtual Machines, and Kernel-Level Optimizations

An operating system (OS) is the core software that manages the hardware and software in a computer. It is an intermediary, facilitating interactions between the user and the computer's hardware. It provides an interface for users to interact with the system and allocates and manages resources for various applications. Popular examples include Windows, macOS, and Linux.

The kernel is at the heart of every operating system. It is a crucial component that directly interacts with the system's hardware. The kernel has complete control over everything in the system. It manages memory allocation, process scheduling, and input/output requests. The kernel plays a vital role in the system's performance and stability by operating at this low level.

System Calls (Syscalls) interface user applications and the kernel. When an application needs to perform an operation that requires access to a system resource (e.g., reading a file or sending network data), it makes a syscall. Then the kernel performs the requested operation on behalf of the application. This mechanism ensures the controlled access of system resources to maintain system security and stability.

Virtual Machines (VMs) are software emulations of physical computers. They replicate the full functionality of a physical computer in an isolated environment while operating on top of a hypervisor (i.e., a type of software that creates and manages virtual environments on a host machine). VMs offer the advantage of security through isolation, resource efficiency, and flexibility for testing and development.

Understanding these concepts is crucial within the context of Firedancer. Firedancer employs several kernel-level optimizations to enhance performance:

- Large Static Allocation:

Firedancer minimizes dynamic memory allocations using large static allocations (i.e., shared memory allocations that other processes can share). Here, memory is allocated once and reused, reducing the overhead associated with frequent allocations and deallocations.

- Bypassing Standard Libraries:

Standard library functions often add extra layers of abstraction and can be less efficient for certain operations. Firedancer bypasses these standard libraries and makes syscalls directly. We'll cover this more in-depth when talking about Firedancer's tile architecture and how it optimizes its networking performance.

Firedancer tries to avoid syscalls and interacting with the OS as much as possible, as these operations slow tasks down significantly.

Ingress and Egress

Ingress and egress are terms used to describe data flow in and out of a computer system or network.

Ingress refers to the entry of data into a system. It is a general term that can include various activities, such as receiving data from the Internet, accepting user inputs, or collecting information from sensors in IoT (Internet of Things) devices. For network systems such as servers or blockchains, ingress involves critical tasks like receiving transaction requests, user

queries, or incoming data streams that need to be processed or stored. The efficient handling of ingress data is vital for the responsiveness and functionality of the system.

Egress refers to a system's outgoing data. This includes sending information online, generating responses to user requests, or transmitting processed data to other systems. For networked systems, egress involves sending validated transactions, broadcasting blockchain updates, or sending data to external storage locations. Properly managing egress data is essential for correctly disseminating information and ensuring the system communicates with other parts of the network effectively.

Pipelines and Data Parallelism

Imagine a pipeline as an assembly line in a factory. It breaks down a complex process into smaller, sequential stages. Each stage in the pipeline performs a particular operation. This approach is highly efficient for repetitive or continuous processing tasks, much like how a blockchain continuously processes transactions.

Data parallelism takes a different approach by processing multiple elements simultaneously but independently. It's as if the factory had more than one assembly line to process data. This is effective for tasks broken down into smaller sub-tasks. For blockchains, specifically in systems such as Firedancer, data parallelism is crucial for handling transactions or data processing tasks concurrently. Firedancer's parallel processing capabilities maximize computational throughput and reduce processing times.

Imagine each stage of the pipeline as an individual workstation in a factory. Each workstation is capable of operating independently and concurrently. This means that while one stage processes a part of the data, the next stage can work on another part simultaneously. This approach allows multiple pipeline stages to be active at the same time, significantly increasing throughput. Firedancer combines the sequential efficiency of pipelining with the simultaneous processing power of parallelism to process large transaction volumes.

Field-Programmable Gate Array (FPGA)

Field-Programmable Gate Arrays (FPGAs) are versatile integrated circuits that can be programmed or reconfigured after manufacturing. These circuits offer a level of adaptability not found in traditional hardware components. They consist of a vast grid of interconnected, programmable logic blocks, each capable of performing various digital functions. This design allows FPGAs to be highly flexible and efficient in applications where speed, parallel processing, and adaptability are paramount.

A typical FPGA consists of tiny programmable elements, all connected by programmable wires. This intricate network of parts and connections enables users to program different chip regions to perform specific tasks. This creates an entirely customized processing environment.

FPGAs excel in parallel processing by utilizing these tiny programmable elements simultaneously. Their structure facilitates efficient pipelining, where data is continuously fed through the processing stages. A well-designed pipeline decouples the system's throughput from latency. Although this pipeline may take longer to produce output than a conventional CPU, the system can handle multiple input data streams concurrently. The latency of each operation has little impact on the overall flow of data, much like water moving through a hose.

In the context of Firedancer, FPGAs offer a significant advantage. They can connect data pipelines to networks directly. FPGAs are more efficient for handling network traffic than traditional setups where GPUs rely on CPUs for data movement. Direct connectivity and the ability to create custom processing solutions, such as building out soft processors on the FPGA fabric, make them invaluable in developing a high-performance validator client.