The Eclipse Performance Thesis

Benjamin Livshits, Alexander Petrosyan, Olivier Desenfans,
and the Eclipse Labs Team

March 17, 2025

Abstract

We have witnessed a tremendous amount of computational power becoming widely available, spurred to a large extent by the needs of LLM training and inference. Blockchain as a field has thus far not benefited from the availability of hardware such as new powerful GPUs and various other hardware acceleration techniques, such as FPGAs, SmartNICs, etc. At Eclipse, we are aiming to correct this oversight: these high-performance computing devices utilized effectively could increase the amount of computation the blockchain makes available by orders of magnitude, something we refer to as *GigaCompute*.

The Eclipse Performance Thesis focuses on building the most performant SVM Layer-2 blockchain with a new SVM client we call GSVM through a combination of software-hardware co-design, cross-layer optimizations, workload non-interference, and dynamic scaling, leveraging the unique freedoms of an L2 to depart from the typical L1 design constraints. Moreover, as an optimistic rollup, Eclipse can take advantage of the significant additional compute capacity without being unduly limited by the speed and cost of zero-knowledge proof generation.

Eclipse argues that co-designing software and hardware is crucial for modern blockchains, taking advantage of the unique position of Eclipse as an L2 to overcome the limitations of off-the-shelf hardware. Cross-layer optimizations across networking, runtime, and storage parts of the transaction execution stack, such as pre-fetching account data in storage at the time of transaction sequencing, are essential for latency reduction. The design of GSVM aims to deliver application workload non-interference and dynamic scaling to meet growing application demands.

We highlight that Eclipse's position as a Layer-2 is fundamentally different in terms of performance constraints from Layer-1 blockchains, which tend to rely on generic hardware and prioritize optimizing consensus over transaction execution efficiency. Eclipse addresses these limitations by separating security from performance through the fraud proof mechanism, and then shifting its focus for GSVM onto validator performance.

To realize the vision of what we call GigaCompute in GSVM, Eclipse points out that while TPS is widely used it is ultimately a flawed metric, advocating for the extension of Compute Units (CUs) to account for concurrency, computational co-processors such as GPUs, and to discourage locking patterns that hinder GSVM's design principles. Component-specific design principles include:

- **Network**: near line-speed processing, probabilistic execution pre-confirmations, performance-based sequencing, application-specific sequencing, and latency-optimized transaction routing.
- **Runtime**: self-improving runtime using reinforcement learning, computational abstraction, hybrid concurrency for resource non-interference, and hotspot-aware hardware affine scheduler.
- **Database**: disk access-minimizing sequencer-driven caching, hotspot-aware parallel NVMe array, hardware-accelerated SSD-aligned accounts database, and fast state commitments for light clients.

An analysis of state contention in Solana highlights some of the shortcomings of the current concurrency model. Longer-term, better concurrency for modern hardware requires moving beyond Solana's currently used pessimistic approach, exploring more parallel-friendly scheduling and data formats.

Lastly, this thesis outlines how three application domains can benefit from the extra processing capacity of the GSVM: AI, Gaming, and DePIN. In the long run, all three domains require large-scale compute and efficient concurrency, precisely what GSVM's GigaCompute ambitions aim to deliver.

1 Introduction

The goal of Eclipse is to deliver the most performant SVM Layer 2 (L2) network. Although the SVM stack is already performance-oriented, thus providing a desirable starting point, we take further advantage of the unique opportunities that Eclipse has as a Layer-2 chain that are not available in the L1 blockchain context. Indeed, an *optimistic rollup* effectively separates integrity concerns from performance considerations. The former are satisfied with the help of fraud proofs [1] that allow outside parties to mount challenges if the rollup operator is suspected of violating computational integrity; similarly, concerns around *censorship resistance* can be addressed with the help of forced transactions and escape hatches [2], [3]. So long as the underlying data availability (DA) layer is able to keep up with the throughput and the fees the rollup collects cover operational DA and L1 costs, and re-execution for fraud proofs is not overly expensive, we can directly focus on improving the performance of the validator.

This separation of security from performance is at the heart of Eclipse's design. With the security foundations in place, Eclipse can build a *supercomputer*. Fortunately, we have seen a great deal of progress in unlocking a significant amount of hardware performance. Some of the most recent advances in GPU-based processing and FPGA-based low-latency computation have become significantly more popular as a result of recent AI and LLM breakthroughs.

We at Eclipse have a real opportunity to bring much of the same hardware progress and computational advances to the field of blockchain. Bringing this computational capacity — orders of magnitude more than current chains support — to the blockchain context is a path to what we call *GigaCompute*. At Eclipse, we follow these guiding principles for the new Eclipse GSVM client:

1. **Software-hardware co-design.** We argue that careful software-hardware co-design truly unlocks the performance potential of modern blockchains. In the rest of this document, we will illustrate this with numerous examples, ranging from SmartNIC devices with support for near line speed execution very close to the network to custom support for faster key-value store databases with the help of zoned NVMe devices and custom firmware. We claim that relying on out-of-the box hardware will not achieve breakthrough performance, and that customizing both the software and hardware to the workloads we can profile and understand well will provide the best outcomes. Admittedly, we at Eclipse are significantly inspired by the tremendous process that we have witnessed in applying hardware-software co-design in the domains of deep learning and LLMs. We want to bring a similar level of performance enhancement to on-chain computing.

2. **Cross-layer optimizations.** Our goal is to co-design the major three components of the GSVM *client*, *networking*, *runtime*, and *storage* to fully take advantage of cross-layer optimization opportunities. For example, performance-driven sequencing can reorder transactions and signal the storage back-end to pre-fetch account data, drastically reducing I/O stalls. Coupled with concurrency control awareness, this synergy leads to near-zero cache misses.

3. **Workload non-interference.** The design principle of Eclipse is to deliver the ability to run applications without *interference*, i.e. if a DEX is experiencing a significant load, it should not adversely undermine the ability of other users to play an on-chain game. The principle of non-interference aligns with the approach of *local fee markets* in Solana [4], [5]. This principle implies that isolated spikes for some hotspots and some applications do not degrade the performance for all users; local fee pricing is similarly *hotspot*-specific. This also means that many advantages of L3s can be delivered within the context of the shared address space of an L2. We aim to expand these principles to other aspects of GSVM design beyond the fees; specifically, we want to make sure that we are able to ensure core affinity for the scheduler so that execution can happen on multiple cores without much interference. Similarly, replying on parallel SSD arrays means that we can ensure parallel access to multiple hotspots, avoiding interference as well.

4. **Dynamic scaling and chain elasticity.** Our goal is to provide enough capacity to scale as application demand grows. Capacity may also scale together with demand; this can be accomplished by dynamically adding more execution cores or provisioning additional storage NVMe when a

Feature	**Solana**	**Polkadot**	**NEAR**
CPU	Powerful multi-core CPU (e.g., 16+ cores, 3.5+ GHz)	Quad-core CPU (e.g., 2.4+ GHz)	Quad-core CPU (e.g., 3.0+ GHz)
RAM	128 GB+ DDR4 RAM	16 GB+ DDR4 RAM	32 GB+ DDR4 RAM
Storage	High-speed NVMe SSD (1 TB+) with high write endurance	NVMe SSD (500 GB+)	NVMe SSD (1 TB+)
Network	Gigabit Ethernet with stable connectivity	Gigabit Ethernet with stable connectivity	Gigabit Ethernet with stable connectivity

Figure 1: Hardware requirements for nodes on Solana, Polkadot, and NEAR.

new demanding application goes live. This is a form of horizontal scaling that can be satisfied with more capacity allocated across the layers of the system, including nodes, execution cores, and storage devices.

Our goal is to deliver a new GSVM client for the Eclipse rollup based on these principles. Taken together, these four principles guide every layer of the Eclipse GSVM client, from the networking pipeline to committing the final state. A more detailed discussion can be found in Section 1.4.

1.1 Limitations of Layer-1 Blockchains (L1s)

Similarly to those who think of operating systems in terms of hardware that was available in the 1980s [6], blockchain designs have been largely blocked by what is possible in the Layer 1 blockchain space, where the validator nodes run client software on a variety of operating systems and hardware platforms and perhaps even different client implementations, using a wide-area BFT consensus. In particular, we observe two major limitations in typical Layer-1 designs:

- **Consensus focus.** Wide-area consensus is usually the focus of performance-oriented work, in part because the time to achieve consensus in a wide-area setting is typically much larger than transaction execution time. This is despite major strides in the area of speeding up consensus, such as the HotStuff family of algorithms [7]–[11] used, for example, by Espresso shared sequencing or DAG-based work, such as Narwhal and Bullshark [12]–[14]. Some of these ideas have migrated to rollups that are applied mainly to sequencer design [10], [15]. Unlike these projects, we believe that decentralization can be achieved through a combination of other means.

- **Over-reliance on generic hardware.** Although they require a fairly significant amount of computational capacity, as in Figure 1, it is debatable whether the actual hardware *utilization* is particularly high. In Section 2 we show that "out-of-the-box" Solana does not use the available CPU cores particularly efficiently, not to mention GPU cores. In fact, blockchain developers are not given even what a higher-end phone would be able to enjoy in terms of compute capacity.

We point out that in most cases, Layer-2 blockchains have not gone beyond the assumptions that are used by Layer-1s, an omission that we aim to rectify in Eclipse.

1.2 Metrics: Why Do We Need GigaCompute?

The choice of the performance metric for which we will optimize is important. Some optimizations will impact some metrics more than others; therefore, understanding why we have chosen one particular trade-off over another requires knowing the rationale behind our initial choice of metric.

Latency. The first and most obvious metric is response latency. It is the one aspect of performance that users experience most directly. Latency is situational. It is helpful when trying to understand the feasibility of certain applications, such as on-chain gaming (Section 7.2), and high-frequency trading. Unfortunately, latency also comes with many ill-defined corner cases. For instance, latency for a transaction that received a pre-confirmation, was subsequently rejected, and then network soft-forked, and finally the consensus resolved to accept the transaction can be one of three numbers. Furthermore, in the context of an *optimistic* Layer-2 the end-to-end latency: from initiating the transaction to settlement, is not determined by the performance of the underlying components. Therefore, we will use latency sparingly and only in situations in which it is helpful to think of the problem in terms of a specific kind of latency.

Action	**CU Cost**
`Vec⟨u32⟩`: 10 items	628
`Vec⟨u64⟩`: 10 items	682
`Vec⟨u8⟩`: 10 items	462
`if` expression	100
Transfer SOL	150
Create Account	3,000
Create `struct`	7,000
Initialize counter	5,000
Create token	3,000
Mint	4,500
Burn	4,000
Transfer	4,500
Initialize Anchor[1]	10,526
Account creation	20,544
Token transfer	50,387

Figure 2: A breakdown of a typical program's execution Compute Unit consumption and cost.

TPS. Transaction throughput, typically measured in transactions per second (TPS), is the most widely discussed metric. This metric is universal and composable, where each blockchain component has a specific volume of transaction processing capacity. This unlocks the ability to optimize each component in isolation and to focus optimization efforts on the components with the lowest throughput capacity.

[1]Anchor is the most popular and officially endorsed framework for writing Solana programs. It is safe to assume that the vast majority of Solana programs will utilize the Anchor framework.

TPS is certainly helpful when thinking about consensus performance. Specifically, recent research, such as Mysticeti by Babel *et al.* [12] and the HotStuff algorithm family [9], reveals TPS nearing 100,000, with Mysticeti highlighting reduced latency. Eclipse is able to achieve 100,000 TPS in a lab setting using Solana's Proof-of-History/Tower BFT mechanisms.

One problem with TPS is that not all transactions are alike. There are many academic papers that cite throughput numbers in excess of 50–100 K TPS, but this is often measured on transfers only, as opposed to computation-heavy transactions. Consider both *slow* and *fast* transaction types. Slow transactions are small messages that cause a large amount of computation, for example, a zero-knowledge proof, while fast transactions are simple native token transfers. Generating zero-knowledge proofs can consume orders of magnitude more in terms of processing cycles than a simple token transfer. However, each transaction counts equally in terms of statistics, while clearly requiring a vastly different amount of computation to execute. When reporting performance numbers, citing a higher TPS may be mainly due to optimizations in the fast transaction pipeline. As such, real-world performance is nowhere near the reported numbers, because slow transactions are inevitable. This is neither the fault of the benchmarks nor the optimization; TPS is not a sufficient metric if the workload is unrepresentative. We are not alone in thinking that TPS is not a completely suitable metric [16], [17]. For example, Monad, another performance-focused chain, takes the approach that TPS is not the right metric to track. Although we share some of the concerns, we believe that we should bring more nuance to this discussion.

1.3 Beyond Gas

Smart contract chains use metering to both compensate infrastructure operators and ensure that on-chain programs terminate. Although the term *gas* is popular in the Ethereum ecosystem, we will borrow the term from Solana and refer to *compute units (CU)*.

Various methods are employed in systems like Polkadot, which utilizes benchmarking. In the Polkadot network, weights play a crucial role in regulating and managing the execution of transactions and other processes. They act as an abstract metric for the computational resources used, ensuring equitable, efficient, and secure operations. Weights measure the computational effort needed to execute specific operations, such as token transfers, smart contract deployments, or block validations. Although they aren't directly linked to particular units like CPU time or memory, they symbolize the overall computational effort. To avoid exceeding block limits, Polkadot implements a weight estimation method to forecast the computational expense of transactions before they occur. Transactions are pre-assigned estimated weights, and if the actual weight surpasses the estimate, the transaction might be declined or incur extra fees. This weight-based fee model allows for extending computation to encompass aspects like custom pricing for database access operations [18].

Computational units (CUs) in Solana serve a similar purpose as gas in Ethereum, but with key differences in their calculation and detailed fee structure; for example, we omit many important details related to things like *rent*, referring the interested reader to Zhao [19]. Similarly to what is found in other metering approaches, the high-level aim is to quantify the computational effort required to process a transaction, including the execution of instructions, data storage, and bandwidth usage. Each operation performed within a transaction consumes a specific number of CUs [20]–[23], and the total CU consumption determines the computational resources utilized. To prevent runaway processes and ensure network stability, Solana imposes limits on CU consumption. The maximum compute units allowed at the moment are shown in Figure 3. These limits are crucial in preventing runaway programs that loop or run indefinitely or consume excessive resources. By enforcing the CU limits, Solana ensures that transactions are processed efficiently and the network remains stable.

CU	
per instruction[2]	200,000
per transaction	1,400,000
per block	48,000,000
per user per block	12,000,000
Storage	
serialized size of transaction	1,232 B
storage size per account	10 MB
Call Depth	
cross-program invocation	4
call stack depth per program	64
Stack Size	
stack frame size	4 KB
program heap size	32 KB

Figure 3: Resource limits in Solana.

Our approach at Eclipse is to start with the CU approach used at Solana, primarily thinking of our performance aim as increasing CUs-per-second. We plan to extend the CU approach in several ways:

- Gas- or CU-based metering is fundamentally better suited to single-threaded execution blockchains. There should be no reason why there cannot be multiple applications running and consuming a significant amount of compute on different machine cores with no interference. As such, we believe that CU thresholds such as those in Figure 3 are limiting for a system like Eclipse that targets dynamic scaling.

- There is room to discourage operations that make it difficult to follow the Eclipse design principles by pricing them higher; an example of this is charging for excessive locking.

- Furthermore, we plan to update the basic notion of CU to include the reliance on *computational co-processors*. These provide specific performance metrics that are highly specialized based on the underlying computation. We give some motivation for this in Section 7. For example, when running LLMs on top of Eclipse, it is appropriate to use *tokens* as a unit of measure.

[2]Each transaction consists of multiple instructions.

1.4 GSVM Design Principles: Towards GigaCompute

Beyond the three overarching principles that contribute to the performance of Eclipse: ① software-hardware co-design, ② cross layer-optimizations, ③ workload non-interference, and ④ dynamic scaling, we propose more design principles that are component-specific, as shown below.

1.4.1 Network

The reader is referred to Section 3 for more details.

- **Near line-speed processing.** Many transactions can be handled much faster close to the network card, reducing the latency; this is often described as near-data processing. Opportunities include operations such as signature verification on SmartNICs, deduplication, and aggressively identifying transactions likely to fail.

- **Probabilistic execution pre-confirmations.** Pre-confirmations [24] are a popular way to notify the user about their transaction inclusion or execution with low latency. We are considering some advanced approximated computation techniques that would probabilistically give execution results for some portion of transactions without involving the runtime. We believe that probabilistic execution pre-confirmations provide the right UX for the end-user, effectively side-stepping debates around block times, block sizes, etc.

- **Performance-based sequencing.** While sequencing has been the focus of a great deal of effort in the rollup domain [25], we propose performance-based sequencing as a key approach to increasing the actual parallelism and reducing cache misses for the storage layer.

- **Application-specific sequencing.** Application-Specific Sequencing (ASS) [26] emerged as a pivotal innovation, granting apps greater control over the inclusion and ordering of transactions affecting their state. At Eclipse, by customizing the sequencing mechanisms to their specific needs, we want to give applications control of MEV capture and subsequent redistribution.

- **Latency-optimized transaction routing.** At Eclipse, we want to minimize the last-hop latency for accessing the rollup by deliberately locating the nodes in a way that is aligned with user and application geo-location. This meshes with the idea of squeezing the latency via pre-confirmations as well.

1.4.2 Runtime

The reader is referred to Section 5 for more details.

- **A self-improving runtime that relies on reinforcement learning** Profile-guided optimization is a time-honored principle in modern runtimes and in the optimization of complex systems in general. At Eclipse, we plan to amplify this principle by training on prior transaction histories and on ongoing traffic patterns, using reinforcement learning.

- **Computational abstraction**: Given that some apps or programs are relatively simple, computational abstraction can be used to compress or short-circuit the computation in a way where a portion of transactions can be computed near the wire, ideally on the SmartNIC, avoiding unnecessary data copying and reducing the end-user latency [27].

- **Hybrid concurrency to support transaction resource non-interference.** While a number of chains such as Aptos, Sui, Monad, and Solana focus on parallel transaction processing, they often suffer from a low *observed level* of parallelism. We aim to improve on that at Eclipse by using a finer-grained level of parallelism and shifting to a hybrid model of optimistic and pessimistic concurrency.

- **Hotspot-aware hardware affine scheduler utilizing hundreds of cores resulting in computational non-interference.** Cores at this stage of hardware development are a commodity; machines that have hundreds of both CPU and GPU cores are quite accessible. At Eclipse we plan to do aggressive real-time priority fee-based scheduling that liberally uses available cores. Specifically, different usage hotspots can be run on different cores, thus avoiding computational interference and effectively providing dynamic scaling.

1.4.3 Database

The reader is referred to Section 6 for more details.

- **Disk access-minimizing sequencer-driven caching.** As the cost of cold fetches from disk is quite high even for modern SSDs, our aim at Eclipse is to make sure that we minimize key/value cache misses and blocking disk access by pre-fetching relevant account data at sequencing time.

- **Hotspot-aware parallel NVME array back-end supporting data non-interference.** Given that different computational hotspots can be located on different NVMe devices, their access can be done in parallel with virtually no interference, thus helping with dynamic scaling. These hotspots can be dynamically migrated across the storage device array as workloads change.

- **Hardware-accelerated SSD-aligned accounts database.** At Eclipse, our goal is to move away from the RocksDB database [28] used by Solana to one that results in much more alignment with the underlying hardware

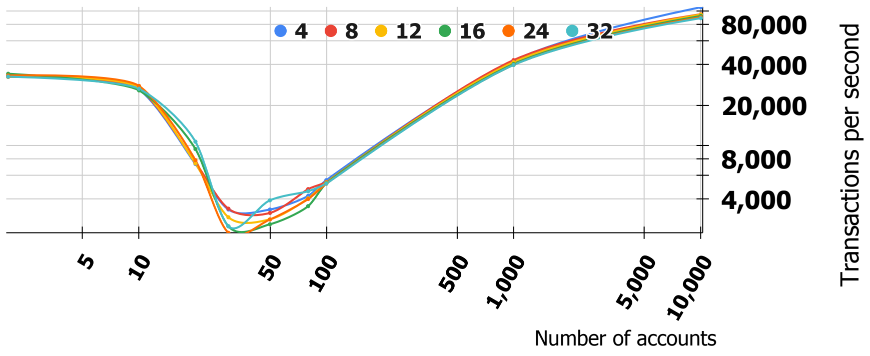


Figure 4: Contention analysis on a synthetic benchmark. The number of accounts between ten and 100 results in a significant degradation in TPS, and does not depend on the number of threads.

and the available amount of RAM, taking full advantage of the available performance.

- **Fast state commitments for light clients.**While Solana does not do state merklization by default, we aim to support light clients and as such plan to produce frequent streaming state commitments to users such as those running light clients. These commitments can be computed on extra CPU or GPU cores, without significantly compromising performance.

1.5 Paper Organization

The structure of the remainder of this document is as follows. Section 2 discusses our transaction contention and opportunities for greater parallelism. Section 3 illustrates opportunities for processing close to the network that help GSVM achieve near line-rate throughput. Section 4 talks about our approach to transaction sequencing and dynamic scheduling, which we refer to as performance-based sequencing. Section 5 is a collection of smaller optimizations that nevertheless give an idea of the specific possibilities pertaining to Solana. Section 6 discusses the storage layer in detail, specifically focusing on acceleration options for key-value stores, especially those that rely on specialized hardware. Section 7 contains a discussion of three classes of applications that will benefit from the performance characteristics of GSVM: AI, gaming, and DePIN. Finally, Section 8 concludes.

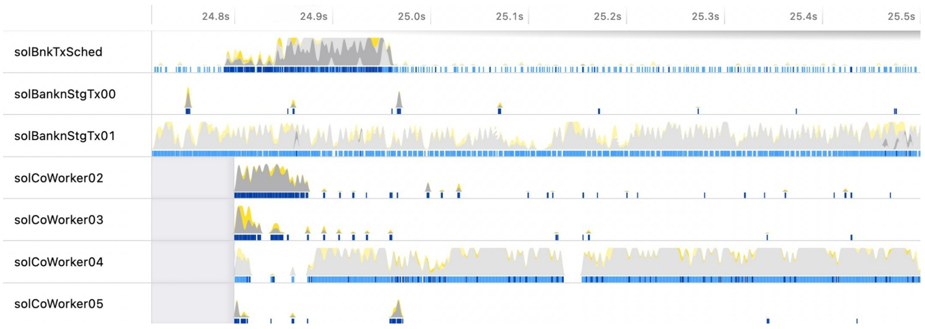


Figure 5: Load balancing across the threads in `agave` under a representative workload; see https://x.com/alessandrod/status/1872533928172769574 for details.

2 Towards Greater Levels of Parallelism

In this section, we expand on what we believe to be one of the key areas of opportunity to increase performance for GSVM. Solana offers a significant degree of parallelism compared to most of the blockchains; however, more effort is needed to bring us to the GigaCompute scale. For example, Figure 4 shows that lock contention can result in severely degraded throughput numbers in the context of a synthetic benchmark [29].

2.1 The Inaugural Memecoin Incident

Performance issues are not just a theoretical consideration: they have a real-world cost to the health of the Solana ecosystem. The network contention that resulted from the `$TRUMP` memecoin, as well as the resulting spike in demand on the Solana chain in January 2025 was discussed in great detail by Helius [30]; we provide only a brief summary below.

- The `$TRUMP` and `$MELANIA` tokens are issued on the network; memecoin launch leveraged Meteora's Dynamic Liquidity Market Maker (DLMM) protocol as its primary liquidity venue;
- the `$TRUMP` memecoin experienced a dramatic surge in value, climbing from under \$10 on the morning of January 18th to a peak of \$74.59 the following day. `$TRUMP` sucked liquidity from most other Solana memecoins, causing tokens to fall in value, as traders sold their existing holdings to buy `$TRUMP` — a wave of arbitrage transactions was triggered programmatically and manually;
- the price of `$SOL` briefly plummeted to \$200, approximately \$50 below its price on centralized exchanges; this price disparity activated arbitrage bots, flooding an already busy network with transactions and driving up the fees;

- the resulting congestion results in degraded Jito validator performance; during the window of degradation, traders using Jito tips quickly adapted to using priority fees. Median priority fees jumped $5,000\times$, non-vote transaction count fell 66%, and compute units per block fell 50%.

The incident can be described as a mixed blessing at best: while about half a million users were on-boarded onto Solana as a result of the memecoin launch, the end-user experience was probably mixed given the effects enumerated above. The Helius article [30] highlights two culprits explaining the events above:

Resource utilization and scheduling issues. Consider Figure 5 which shows thread utilization over time in `agave`. We are interested in the activity of the `solBnkTxSched` thread, which is responsible for building a directed acyclic graph of data dependency and passing off execution to the four `solCoWorker02` to `solCoWorker05` threads, which handle the execution of the BPF code. The `solQuicTpuRt` threads are responsible for ingestion of transactions sent to the leader in the current consensus round and not for transaction execution. The `solBanknStgTx01` and `solBanknStgTx00` threads are responsible for the so-called banking stage: they process proof-of-history and related activities that are not directly connected to transaction execution. As one can see, the work distribution is largely uneven, effectively becoming single-threaded. Specifically, the scheduling thread stops all activity at around 25.0 second mark. Over a period of approximately 0.1 seconds, the worker threads go from having work evenly distributed across the threads to only one thread `solCoWorker04` doing all the work.

According to an analysis by Anza, the Agave validator's Transaction Processing Unit has up to 93% free capacity (https://x.com/alessandrod/status/1881472335804506504). For each transaction included in a block, the validator is ingesting two- to four hundred more that are not included: current block limits are relatively low, meaning fewer transactions are included before the validator stops and switches to the next block. Some of the Solana proposals explore raising block limit CUs; see Section 3 for details. Effectively saturating the worker threads with execution would enable further performance in the validator client, which Anza estimated to be approximately one million TPS.

2.2 Unlocking More Parallelism

Solana offers a significant degree of parallelism as a consequence of its architecture. The programs are effectively stateless and their interaction with external accounts is explicit; in the *pessimistic concurrency model* that Solana relies on, accessed account lists must be declared ahead of time.

In practice, this declarative approach can impose significant limitations on the program developer, as Heimbach *et al.* highlight [31]. Transactions that access data that are not in the access list fail and revert, so programmers often over-declare.

Transaction commutativity. We propose implementing an interface to inform the scheduler of transaction's properties, which we can use to perform these reductions and reduce the overall system load. In the SVM programming model, race conditions are avoided by preventing read/write and write/write conflicts within program account access lists. However, not all such conflicts violate the SVM invariants. For example, token transfers commute under most circumstances. Since the SVM does not guarantee any form of ordering on transactions, we can exploit this commutativity. Reordering transactions enables us to execute transactions that the SVM would otherwise serialize in parallel, balancing the load on multiple execution threads.

Optimistic concurrency. An alternative to the pessimistic concurrency model is to execute transactions optimistically: transactions would run concurrently, and their inputs and outputs would be checked for potential conflicts. If a transaction is found to have invalid inputs (for instance, it uses outdated data), it is re-executed with the correct information to ensure soundness. As the experience of Block-STM suggests [29], this re-execution is often relatively infrequent in practice.

Hybrid concurrency. We can reduce the number of transactions that need to be re-executed using access lists. Solana programs are stateless, and access lists provide an exhaustive list of inputs. This would allow us to checkpoint the program reads and writes, suspending execution where needed to ensure correct ordering. We can also attempt to schedule operations pessimistically, and to switch to optimistic concurrency control under tunable saturation thresholds. There are more fine-grained approaches that we can adopt as well, such as having pessimistically concurrent batches message-passing to an optimistically executing thread pool. For example, we can section off transactions for which re-execution would be expensive to be free of data races, while executing simple transactions optimistically.

3 Near Line-Rate Transaction Processing

In designing GSVM, one of our goals is to expose as much of the underlying hardware capabilities as we can, in order to improve the experience of the end-user. The primary motivation for achieving near line-speed processing is to lower end-to-end latency, a critical metric from the end-user's perspective, as well as to release more resources to the main processing pipeline, increasing the effective throughput.

3.1 Towards Line-Speed Processing for GSVM

By performing partial or complete transaction handling at the network boundary, Eclipse's GSVM client can speed up or simplify the validator pipeline. Specifically, we envision the following capabilities at near line-rate speeds.

- **Filtering failing transactions and transaction de-duplication.** One could do quick sanity checks, signature validation, or replay detection to

Reference	Year	Hardware	Complexity	Frequency	Latency	Throughput
Turan *et al.*	2018	Xilinx Zynq 7030	11.1KLuts +16DSPs	82Mhz	1.6ms	160Kbps
Yang *et al.*	2021	0.18um	21,737Gates	285Mhz	3.3ms	76.7Kbps
Salarifard *et al.*	2019	Xilinx Zynq 7030	12.9KLuts +182DSPs	87Mhz	0.044ms	5,818Kbps
Mehrabi *et al.*	2019	Xilinx Zynq 7030	8.77KLuts	173Mhz	4.6ms	55.8Kbps
Tao *et al.* [32]	2022	28nm	40,166 Gates	450Mhz	1.37ms	186.9Kbps

Figure 6: Comparing different implementations of `ed25519` signature computation on FPGAs from recent papers; this table comes from Tao *et al.* [32].

filter out obviously invalid or duplicate transactions or to be more resistant to DOS attacks [33]–[35].

- **Efficient signature verification.** There is a long history of performing signature verification in hardware. For example, Tao *et al.* [32] present a comparison of low-latency FPGA-based approaches for `ed25519` signature computation over the last several years, as shown in Figure 6; note that signature verification is usually faster; we have picked this signature scheme due to its relevance to the SVM stack.

- **State prefetching.** When transactions arrive on the NIC, it presents a natural opportunity to prefetch the required state by identifying the accounts that need loading. We propose a prefetching strategy that proactively initializes the RAM cache by transferring accounts from the SSD during the transaction queuing phase. This approach guarantees that once transactions are ready for execution, all necessary accounts are already in memory. This technique is a good example of cross-layer optimizations within GSVM.

- **Pre-confirmations.** Pre-confirmations can be served much faster, after ensuring transaction validity. Although not final, these early signals allow users to quickly gauge potential outcomes of their transactions, improving UX for latency-critical applications. We propose two types of pre-confirmations that the NIC can provide at near line-rate speed:

  – The first one is *inclusion* pre-confirmations, which is mostly an indication that a) the transaction is valid, and b) that the current network conditions strongly indicate that it will be included in the chain.

  – We also introduce a notion of *probabilistic execution pre-confirmations*: with partial access to the blockchain state, the NIC can quickly simulate specific types of transactions [36] or to apply a pre-trained classifier to determine its chance of success. Upon receiving this confidence interval, the caller can then determine the right strategy, such as waiting for execution, retrying with different parameters, etc.

The clear advantage here is that the NIC is able to serve both pre-confirmation types extremely fast.

3.1.1 SmartNICs and FPGAs

For a detailed examination of this area, please see surveys by Nickel *et al.* [37] and Du *et al.* [38]. The continuous enhancement of programmable hardware platforms, particularly FPGAs, supports the continuous advancement of low-latency packet processing capabilities. With the advent of new networking standards and protocols, the adaptability of FPGAs facilitates swift adjustments, enabling organizations to sustain competitive performance in a rapidly changing environment. Numerous recent projects illustrate that SmartNICs and FPGAs lead the efforts to increase packet processing and transactions in contemporary network frameworks. Their capacity to transfer processing duties from the CPU, augmented by improvements in programming models and hardware cohesion, makes them indispensable components in the progression of high-performance networking. In particular, much of this innovation has been motivated by the progress around AI training and inference, with these hardware devices directly contributing to both. We at Eclipse want to take full advantage of this recent progress in the design of GSVM.

The rapid advancement of networking technologies has necessitated the exploration of innovative solutions for packet processing and transaction acceleration. Among these solutions, SmartNICs (Smart Network Interface Cards) and FPGAs (Field-Programmable Gate Arrays) have emerged as critical components that can significantly enhance performance by offloading processing tasks from the CPU. Moreover, the integration of machine learning techniques with SmartNICs and FPGAs presents a promising avenue to enhance performance modeling and prediction. Although specific studies on this integration were not referenced in the provided literature, the potential for machine learning to improve the accuracy of performance models for SmartNIC programs is an area ripe for exploration. The dynamic nature of network traffic and the complexity of stateful applications can benefit from adaptive algorithms that optimize resource allocation and processing strategies in real-time.

SmartNICs are designed to integrate processing capabilities directly into the network interface, allowing for the offloading of various network functions from the host CPU. This integration is particularly beneficial in high-speed networking environments, where traditional kernel-based processing can introduce significant latency and overhead. For instance, Wang *et al.* [39] highlight the limitations of conventional network stacks, which often struggle to keep pace with the increasing speed of network interfaces. They propose the eXpress Data Path (XDP) as a viable solution for pre-stack packet processing acceleration, enabling more efficient handling of incoming packets directly on the NIC. This is a common optimization used, among others, by the Firedancer Solana validator [40]. This kind of approach not only reduces the load on the CPU, but also enhances overall system performance by allowing lightweight and compatible acceleration methods, potentially allowing for greater throughput in GSVM.

The use of AF_XDP, a variant of XDP, further optimizes packet processing by enabling the decomposition of network functions in both kernel and user spaces. Mostafa *et al.* [41] discuss the implementation of AF XDP for better packet processing performance, although the focus is primarily on IPv6 Segment Routing rather than a direct comparison with traditional DPDK implementations. This dual-layer approach allows for greater flexibility in managing network traffic while maintaining compatibility with existing kernel-based applications. AF_XDP, introduced in Linux kernel 4.18, provides zero-copy networking by bypassing the kernel stack while using vendor-provided kernel drivers, offering lower operational overhead than user-space networking frameworks like DPDK. The paper pinpoints design inefficiencies in AF_XDP kernel drivers, particularly in the mlx5 driver from NVIDIA-Mellanox.

The authors highlight performance bottlenecks, such as suboptimal MP-WQE (Multi-Packet Work Queue Entry) inlining, which causes excessive CPU overhead and latency spikes at specific packet sizes. The results in the paper indicate that AF_XDP surpasses conventional Linux socket APIs, yet DPDK still achieves superior throughput, with 36 million frames per second compared to 10 million, and lower latency of $9\mu s$ versus $22\mu s$ for 64-byte frames. The absence of configurable settings in the AF_XDP mlx5 driver hampers the ability to fine-tune performance, resulting in reduced efficiency during periods of high traffic. The paper provides an overview of 39 kernel drivers compatible with AF_XDP, describing both their features and drawbacks, thereby aiding developers in evaluating portability issues when transitioning AF_XDP-driven applications between various NIC drivers. In conclusion, the research asserts that AF_XDP holds potential but needs further driver refinements to achieve DPDK-like performance while retaining deployment simplicity. Although AF_XDP lags behind DPDK in throughput, further refinements could bring it closer to DPDK's performance. For Eclipse, enabling simpler zero-copy paths without replying on heavy-weight user-space frameworks is appealing for near line-rate transaction filtering.

FPGAs provide another layer of acceleration for packet processing tasks due to their reconfigurable architecture, which allows for the implementation of custom processing pipelines tailored to specific applications. Sakakibara *et al.* [36] demonstrate the potential of FPGAs in accelerating blockchain transfer systems, demonstrating how integrating network, memory, and compute interfaces in an FPGA can lead to improved response times and reduced server workloads. The ability to adapt the hardware configuration to meet specific processing requirements is particularly advantageous in environments characterized by variable workloads and high-throughput demands. The authors implement a prototype system on an FPGA-based NIC (NetFPGA-SUME board), integrating a key-value data store and supporting key blockchain operations such as

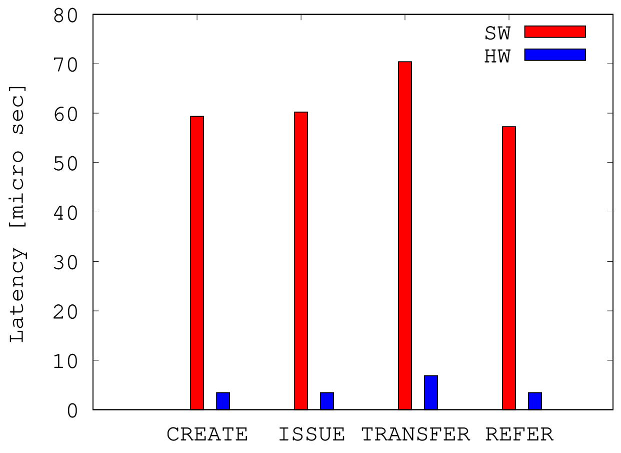


Figure 7: Latency for `CREATE`, `ISSUE`, `TRANSFER` and `REFER` commands from Sakakibara *et al.* [36]. Comparing FPGA-based execution to a CPU baseline we see that the FPGA approach yields significantly lower latency.

`CREATE`, `ISSUE`, `TRANSFER`, and `REFER`. By executing these operations directly on the FPGA, the system reduces network stack delays and improves transaction processing efficiency. Experiments show that the FPGA-based system achieves 6.04× higher throughput and 15.4× lower latency compared to traditional software-based blockchain implementations. The offloaded processing enables 100,000 transactions per second with sub-$7\mu s$ latency, meeting the performance requirements of modern off-chain transaction processing. The results demonstrate that FPGA-based NICs can significantly accelerate blockchain operations, making them a viable alternative to CPU-bound blockchain execution models, *in addition* to tasks such as signature checks or transaction filtering at the NIC level that we discussed in the context of GSVM. Figure 7 shows a dramatic reduction in latency for different commands.

The integration of SmartNICs and FPGAs presents interesting possibilities for improving packet processing efficiency. Rivitti *et al.* [42] investigate the potential of employing eBPF/XDP programs to create customized hardware designs for NICs, effectively merging software and hardware tuning. This method allows network developers to take advantage of the unique features of SmartNICs and FPGAs, facilitating highly efficient packet processing systems that can handle growing network speeds. This research introduces eHDL, a high-level synthesis tool that automatically converts eBPF/XDP programs into custom hardware pipelines designed for FPGA-based SmartNICs. Traditional software-driven packet processing struggles with the demands of faster network speeds, and while hardware acceleration offers promise, it often requires niche expertise. eHDL addresses this challenge by enabling software developers to design hardware-accelerated NIC functions using familiar eBPF/XDP programs, which are then automatically synthesized into FPGA architectures. The tool ensures efficient concurrent execution, stateful processing, and data integrity, allowing complex networking tasks to be executed directly in hardware. Tests on a Xilinx Alveo U50 FPGA NIC indicate that eHDL achieves 100 Gbps line-rate packet forwarding with merely 6.5%–13.3% FPGA resource utilization and 1-μs forwarding latency. Compared to CPU-based approaches, eHDL delivers $10-100\times$ higher throughput, proving its efficacy for use cases such as firewalls, load balancing, and deep packet inspection. By facilitating software-defined hardware acceleration without the need for hardware expertise, eHDL offers a scalable and efficient alternative to conventional software and programmable NIC solutions, making high-performance packet processing more accessible.

Despite the advantages offered by SmartNICs and FPGAs, several challenges remain in their implementation and integration into existing systems. Enberg *et al.* [43] emphasize the complexities associated with OS abstractions and the need for efficient resource partitioning to optimize I/O operations. The increasing speed of I/O devices necessitates a re-evaluation of traditional operating system designs, which may not adequately support the performance requirements of high-speed networking. Furthermore, traditional in-kernel network stacks often perform excessive work per packet, leading to inefficiencies that can hinder the overall performance of the system. The authors propose parakernel, a new OS structure that eliminates most traditional OS abstractions and securely partitions hardware resources to enable high-performance application-level parallelism.

By minimizing kernel involvement in data plane operations, the parakernel allows applications to directly control their allocated resources, improving efficiency and eliminating the overhead of kernel-based multiplexing. The parakernel partitions CPU cores, memory, and I/O devices, allowing applications to interact directly with hardware, while only multiplexing shared resources when necessary. This approach reduces OS overhead, improves system isolation, and eliminates outdated POSIX abstractions that hinder performance. The authors highlight that modern SmartNICs and hardware packet filters (such as eBPF/XDP) already provide kernel bypass mechanisms, reinforcing the need for OS architectures that prioritize direct resource management. By removing legacy OS constraints, the parakernel model enables low-latency, high-throughput computing, particularly in environments such as datacenters, cloud computing, and edge computing. Such an approach could further reduce overhead for near line-rate transaction handling in GSVM, by letting user-space processes directly manage specialized NICs. The reader is also referred to a discussion of unikernels in Section 5.4.

The application of SmartNICs and FPGAs extends beyond traditional networking scenarios. For example, Xi *et al.* [44] discuss the use of SmartNICs to accelerate stateful network applications, emphasizing the importance of efficient state management and resource allocation. Their work highlights the potential for SmartNICs to handle complex stateful operations that are typically challenging to implement in software alone. By offloading these tasks to dedicated hardware, organizations can achieve higher throughput and lower latency in their network applications. Traditional CPU-based packet processing suffers from high latency and resource contention, which limits scalability. Cora introduces a hybrid execution model, where SmartNICs handle stateful packet processing tasks such as connection tracking and packet classification, while the CPU focuses on complex computations. This approach significantly reduces CPU overhead and improves overall system efficiency. Experimental evaluations demonstrate that Cora achieves up to 5× higher throughput and reduces latency by 60% compared to CPU-based processing. The framework supports dynamic state management, ensuring consistency between the SmartNIC and CPU. By efficiently offloading critical packet processing tasks to programmable hardware, Cora enables high-performance, low-latency networking solutions, making it a promising approach for cloud data centers and high-speed network environments. Cora illustrates how stateful tasks such as partial transaction validation or execution to generate execution pre-confirmations be moved to NICs in GSVM, thus also lowering CPU contention.

4 Performance-Based Sequencing

Building on our discussion of near line-rate processing in the previous section, we next explore a specific approach we call *performance-based sequencing.* Eclipse, as a Layer-2 blockchain, avoids many constraints and limitations of L1s such as Ethereum or Solana. This gives us the freedom to explore multiple design choices. It also allows us to address some of the gaps in the protocol, such as adding native support for bundles, which is addressed with separate solutions such as Jito (`https://www.jito.network/`) on Solana. Approaches to transaction sequencing thus far have been largely driven by MEV-related concerns [25] or fairness [45]. At Eclipse, we aim to capture and redistribute the MEV, benefiting applications, while still delivering top-notch performance for the benefit of the end-user. GSVM has extensive freedom to revise sequencing rules and incorporate performance-oriented features.

4.1 Priority-Based Sequencing

Most L2s today use First-Come First-Served (FCFS) approaches to transaction sequencing. A frequent consequence of this approach is that traders invest in reducing latency to the sequencer through standard High-Frequency Trading techniques to probabilistically front-run other users. This practice can be highly profitable for traders, but none of the resulting MEV revenue is recirculated to the network. Capturing MEV revenue is important to reduce the negative impact of these activities on regular users. At Eclipse, we also believe that MEV should be redistributed at least in part to participating on-chain applications.

A more effective and direct way to capture MEV revenue is to make MEV-extracting actors compete with each other for priority access. Several solutions have been proposed to address this problem, including TimeBoost on Arbitrum and Jito on Solana.

- TimeBoost introduces the concept of *fast* and *slow lanes*, with users gaining control of the fast lane for a given period through an auction mechanism. Every other non-winning user effectively incurs a 200 ms latency penalty.
- Jito holds an auction during the first 200 ms of each block, during which units of one or more transactions (bundles) compete based on a priority fee called a "Jito tip." This mechanism is only supported on validator nodes that use Jito validator patches; otherwise, the validator defaults to the Solana priority fee mechanism. Jito offers two major benefits: bundles that allow grouping transactions beyond Solana's size limits and auctions that may offer more efficiency under contention compared to the default Solana scheduler.

Both solutions highlight the importance of structured auctions or priority fees.

The primary rationale for targeting approximately 200 ms block times stems from its approximation of the global average Round Trip Time (RTT) to the sequencer. This is a key design consideration, as it balances the economic incentives for sequencer co-location against the alternative of priority fee utilization. Reducing block times below this threshold risks amplifying the advantages derived from geographic proximity, potentially leading to a disproportionate concentration of latency-driven MEV. To counteract this, Eclipse is engineered to take advantage of a priority fee system from the outset, ensuring that GSVM nodes can incorporate robust scheduling mechanisms. This empowers traders to compete for faster inclusion based on fee bids rather than solely on geographic advantage, thus mitigating the necessity for extreme low-latency optimizations beyond basic co-location. Consequently, priority fees provide a more efficient and equitable mechanism for transaction inclusion compared to exclusive reliance on latency.

4.2 Transaction Scheduling

Performance is the main concern in our sequencer design. But, as we just saw, we cannot completely ignore the concern of MEV capture in this design. With these constraints in mind, let us set the requirements for our scheduler.

1. **Reduce latency**: latency should be minimized for all transactions, especially for transactions with priority fees. Paying more in fees should grant you reduced latency and faster inclusion. However, latency should be minimized for all transactions, independently of the contention on the network.
2. **Maximize resource utilization**: under congestion, the system should exploit CPU, memory, and I/O capacity as fully as possible without causing stalls.
3. **Relaxed ordering**: while paying higher priority fees ensures faster inclusion, the scheduler need not strictly rank transactions in perfect fee order.

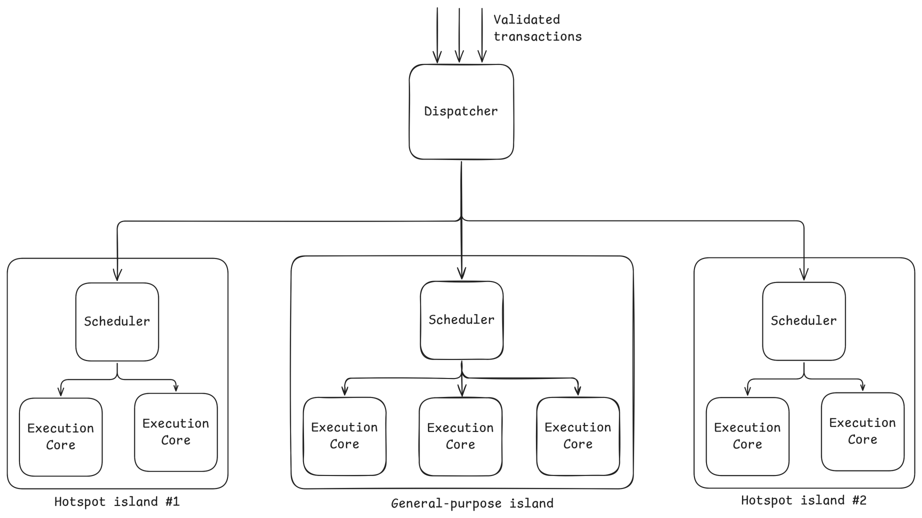


Figure 8: Dispatching transactions across multiple hotspot islands. This figure illustrates how multiple hotspot islands can operate in parallel, each handling distinct sets of accounts to minimize contention.

> This enables us to prioritize the two first requirements in situations where we need to arbitrage between performance and fee ordering. For reference, Solana does specify transaction ordering; it leaves the ordering up to the different client implementations.

4.3 Hotspot Islands

The first important observation when it comes to GSVM is that CPU cores have become somewhat of a commodity. Recent server-grade CPUs, such as the AMD EPYC 9965 have 192 CPU cores, more than enough for most blockchain applications.

The second observation is that designing a scheduler that handles shared resources and dispatches transactions to this many cores is a challenge in itself, especially under heavy contention. It is much easier to divide the problem and only schedule transactions that touch on the same accounts together. We can then process them on the same CPU core(s), sharing data in cache. This design extends Solana's local fee market approach: under enough usage, an app effectively receives dedicated execution resources, akin to having its own shard or app-chain.

Our proposal is to dedicate cores to the processing of transactions acting on highly contentious state. This practice, called pinning or clustering, dedicates one or more cores to specific application threads. We call this concept *hotspot islands*, effectively treating each heavily used state slice as its own concurrency domain, pinned to dedicated cores for local caching efficiency. We illustrate such a design in Figure 8, where we have two islands for two hotspots that execute concurrently, and a generic island for all the other transactions. In this example, the general-purpose island is scaled to three cores, whereas the two hotspot islands use two cores each.

The number of cores dedicated to each hostspot island depends on access patterns: if you have many reads and writes, writes may be serialized, while the reads may proceed in parallel on multiple cores. Given that we can train on prior blocks, these access patterns are fairly predictable for each application, making it possible to do accurate island sizing. In an ideal scenario, the data for each island would remain in L2 or L3 caches, minimizing the overhead of cross-core coherence.

4.4 Priority-based Real-time Scheduling

Each island has its own scheduler, whose job is to guarantee that all transactions are executed as fast as possible while prioritizing transactions whose priority fees are high. We introduce the idea of translating priority fees to *deadlines*, and formulate the problem as that of scheduling sporadic non-preemptive soft real-time tasks. It is proven that there is no optimal scheduling algorithm for the generic case in such systems and that computing an optimal schedule is NP-Hard [46] because of the non-preemptiveness constraint. However, it is entirely possible to derive non-optimal, but efficient scheduling algorithms based on *Earliest Deadline First* [47] (EDF) or one of its multi-core variants [48], adjusted for specific criteria, such as the number of deadline misses, misses of high-priority tasks, etc. A common approach is to reuse EDF, accept missed deadlines, and drop transactions eagerly if the load exceeds the capacity of the system. For example, if an application tries to submit 10 k transactions in a short burst, the scheduler could drop or defer some transactions to avoid a significant backlog from being formed, ensuring that high-priority transactions would still complete quickly. Although optimal non-preemptive scheduling is NP-hard, EDF-based heuristics suffice for blockchain workloads. We treat higher fees as shorter deadlines, letting them effectively jump ahead in the queue. Although there are algorithms to reduce cache misses through scheduling techniques [49], we prefer to bypass the problem entirely with the concept of hotspot islands. Each core will only work on a reduced amount of data, possibly even fitting most of the accounts required to process transactions inside the CPU caches.

4.5 A Motivating Example

Consider a scenario inspired by real-world usage: a popular AMM incurring heavy volume, and a viral on-chain game generating numerous small transactions. As shown in Figure 9, the system spawns separate islands for an AMM (with high fees, sequential transactions) and a clicker game (massively parallel but lower fees); meanwhile, general traffic is handled on a shared island.

- Activity on the AMM consists of many transactions with high priority fees,

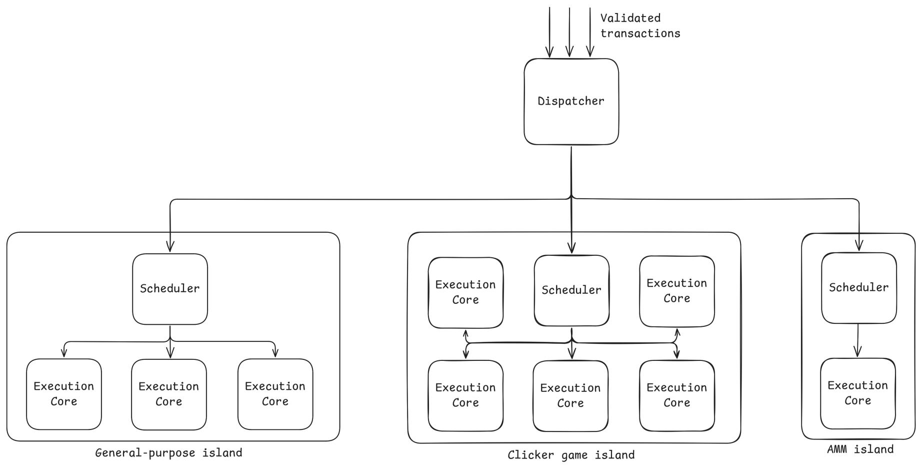


Figure 9: An example of allocation of resources, with a peak in trading and gaming activity.

as traders fight for inclusion. Because all trades in the AMM touch the same pool states, concurrency is limited. The AMM island typically runs on a single core, maximizing cache locality for that sequential workload;

- Activity resulting from the game does not have any associated priority fees, as users only care about transactions landing eventually. There is no contention between users, they all write to their own accounts. These transactions are inherently parallelizable. Dependencies between these interactions originating from the same user are likely to be infrequent.

We see that activity on the AMM gets its own island, but opportunities for parallelism are limited by the nature of the workload, and therefore this island only has one execution core. Priority-based scheduling guarantees a faster execution to transactions with high priority fees, on average. The game also has its own island. The workload is highly parallelizable and therefore the system spins up five execution cores. The execution threads will likely be I/O bound due to the nature of the workload (increase some counter in an account). This number can be adjusted at runtime, depending on traffic. Although we could handle such a workload on the general-purpose island, the goal here is to preserve bandwidth for all applications, while dedicating resources to the game for better UX. Although the AMM's single-thread usage is forced by account contention, the clicker game could scale to five threads. GSVM could reallocate threads dynamically as traffic patterns change.

4.6 Dynamic Scaling

The general-purpose island is required at all times to process the rest of transactions. The scheduler's job on this island will be much more involved as it has to schedule transactions invoking multiple applications. We do not foresee this to be a problem as a) if a hotspot occurs, it will be moved to its own hotspot and b) we can detect hotspots ahead of execution, as soon as transactions enter the NIC. Hotspots can therefore be created just in time for incoming bursts of activity; islands can be deallocated once activity slows down.

For the foreseeable future, we estimate that we will never need more cores than are available on modern server-grade CPUs for typical blockchain workloads. As of 2025, server-grade CPUs like AMD EPYC already feature up to 192 cores, well beyond the concurrency typical blockchains require. Even if usage outgrows a single node, horizontal scaling across multiple machines remains viable, with moderate overhead. This may cease to be true with more applications and users coming on-chain. Our claim is that this does not matter: this design even scales horizontally across multiple machines at the cost of a reasonable network overhead; in a multi-machine setup, each machine could handle certain hotspot islands, communicating partial states or block merges as needed. These observations also highlight that our design is likely to provide the non-interference property. The limiting factor in such a design is database access, as discussed in Section 6.

4.7 Conclusions

By combining clustering and real-time scheduling, we solve multiple problems that would be a great deal harder to solve with a single scheduler.

- Within an island, real-time scheduling can guarantee QoS for all transactions, while implementing logical *priority lanes* for transactions with high fees. Furthermore, it guarantees an efficient usage of each core in the island, reducing the total number of cores required for a given island.
- Dynamic scaling allows us to adapt to the current load on the network and reserve execution resources for the applications that need them the most at any given moment.
- Hotspot islands provide non-interference guarantees: memory accesses pertaining to specific applications are limited to specific cores, and the number of accounts to be accessed by each core is decreased. This drastically increases cache efficiency and potential throughput on each core. This property is conserved independently of the sequencer load. The scheduler on each island is not affected by the load on other applications.

In conclusion, clustering of accounts into hotspot islands, coupled with real-time priority scheduling, meets our concurrency, latency, and MEV capture goals. Each island independently optimizes throughput for its assigned workload, extending local fee markets and maximizing parallelization. The next sections illustrate how this strategy integrates with our runtime and database layers, ensuring end-to-end performance.

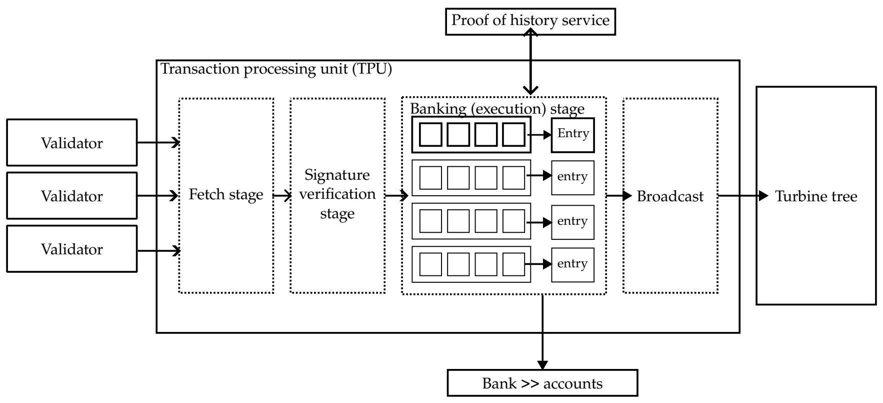


Figure 10: Schematic representation of the Solana transaction life-cycle. Credit: Anza

5 Optimizing the Runtime for GigaCompute

Having addressed transaction ingestion and near line-rate processing in Sections 3 and 4, we now turn to runtime optimizations beyond that initial pipeline stage. This section concentrates on optimizations applicable to the components of the transaction processing pipeline beyond the transaction ingestion phase and the pre-processing mentioned in the preceding sections. This involves several broadly focused areas of runtime optimization that deviate from the design decisions in the Solana's implementation of the SVM, to deliver performance we associate with GigaCompute. Section 7 talks about some computationally heavy workloads, such as AI. Our aim is to present the various techniques at our disposal for addressing the problem, pointing out the landscape of opportunities, rather than dedicating ourselves to a single approach.

5.1 Runtime Considerations

The execution runtime of Solana is an interesting piece of engineering and one that gives Eclipse as strong performance baseline; being initially developed with the purpose of being fast. Eclipse inherits much of this continuous block-building approach, but we are free to exploit both high-end hardware (with hundreds of CPU cores, parallel SSD arrays, etc.) and to engage in aggressive software-hardware co-design in ways that Solana may not fully address.

As another example, consider the leader rotation process in Solana, a *slot* is a unit of time which lasts 400 milliseconds, with leaders switching after every four slots (*i.e.* 1.6 seconds). Solana orchestrates leader rotation every 400 ms slot, ensuring the upcoming leader halts transaction forwarding to manage up to a gigabyte per second of inbound traffic, also reducing unnecessary traffic; two slots before assuming leadership in the topology, a validator node proactively halts transaction forwarding to prepare for its upcoming workload. The amount of inbound traffic reaches more than a gigabyte per second.

As illustrated in Figure 10, incoming transactions enter the Transaction processing unit (TPU), starting at the Fetch stage with transactions received via QUIC, followed by the Signature verification stage (sigverify); this is where signatures are validated, the number of signatures is checked, and where duplicate transactions are eliminated. Subsequently, the Banking stage is where the transactions are executed; it builds blocks and flushes finalized block account updates permanently.

The execution pipeline employs six CPU threads, significantly fewer than the 64–192 cores often found in modern high-end servers. The current Solana setup does not take advantage of GPU resources, either, leaving substantial headroom for parallel execution in Eclipse. It is important to note that this is significantly lower than what is available on a contemporary CPU, and the process does not make use of any GPU resources. Of these, four are dedicated to the execution of user-submitted transactions, while two handle voting transactions central to the proof-of-history (PoH) mechanism. For GSVM, we plan to scale the transaction execution pipeline across more cores, as described in Section 4 with hotspot islands. Transactions are grouped into entries and executed within the Solana Virtual Machine (SVM) with locked accounts. Updates to transactions are confirmed and executed only after verifying that they are recent, and have not been processed previously. An entry's hash is sent to the Proof of History service, and if successful, changes are committed to the bank, releasing the account locks.

5.2 Execution Model

In Solana, on-chain data is stored in accounts. The accounts containing executable code are called *programs*, where the executable code is stored as a specialized version of eBPF known as the Solana Bytecode Format (SBF). The supermajority of programs are written in Rust, the best supported language, although some are written in C, C++, Move, and Zig. Within Rust, a majority of programs is written using the Anchor macro library, which provides a domain-specific language to abstract account management, data (de)serialization, security checks, *etc.* and allows programmers to focus on the business logic of their application.

Solana programs are *stateless*, meaning that only data in accounts is persisted. Furthermore, the exhaustive list of accounts written to by any given program must be declared ahead of time, including special accounts known as `SysVars`. Programs can invoke other programs with a limited call depth of 4, with a mechanism known as cross-program invocation (CPI). This all amounts to a rudimentary form of referential transparency; for instance, we can replace any BPF program execution with any other executable format that results in the same observable effects on the accounts. This could be specialized hardware, such as a GPU, an FPGA, even an ASIC, or a different bytecode representation format, such as WASM; we further explore these possibilities in Section 5.3.

In Solana, transactions are executed concurrently and grouped into ledger "entries", each consisting of at most 64 transactions that do not conflict. This parallel transaction processing is simplified because every transaction must specify all accounts it will interact with, both for reading and writing. Although this design requires developers to manage account inclusion, it enables validators to circumvent race conditions by ensuring that only non-conflicting transactions are executed together in each entry. Conflicts arise if two transactions try to write to the same account or if one writes to and another reads from the same account. Consequently, conflicting transactions are assigned to separate entries, where they are processed one after the other, whereas non-conflicting transactions are executed simultaneously. As explained in Section 4, at Eclipse, we plan to expand scheduling and memory pinning, prioritizing minimal lock contention.

5.2.1 Bytecode and VM Considerations

Optimizing the execution environment is much more prevalent in blockchains that define their own execution environments and virtual machines. The majority of the work out there has been dedicated to the analysis of Ethereum [3], [50]–[56], most of which focuses on introducing parallel execution. Some discuss opportunities related to improving the performance of Merkle trees [57]–[59], with particular attention given to the use case of Ethereum, wherein any state modification must be propagated into the Merkle-Patricia trie, a data structure that acts as a prefix tree (a.k.a. *trie*), and is specialized for use in the Ethereum blockchain. We further discuss these data structures in Section 6.

Detecting and working with conflicting transactions is the main challenge in this setting. An approach to this is studied in the Software Transaction Memory (STM) libraries [60], [61], where memory accesses are tracked to detect conflicts. There are also some STM libraries that implement optimistic concurrency control [62] (OCC) where memory accesses are recorded, transactions are validated post execution, and conflicting transactions are aborted and re-executed. In general, the blockchain setting requires determinism, which many STM papers do not achieve. Other works use specific parts of the blockchain in conjunction with STM by pre-computing dependencies and calculating a directed acyclic graph of transactions which can be executed using a fork-join schedule [63].

There have also been works related to accelerating Ethereum execution, such as Forerunner [64], which gains performance by pre-executing transactions to give hints for final execution. By pre-executing multiple transaction orderings within the mempool, Forerunner achieves an effective average speedup of 8.39× on the transactions that it observes during the dissemination phase, which accounts for 95.71% of all transactions; the end-to-end speedup over all transactions is 6.06×. This approach resonates with our ideas related to near line-rate processing in Section 3.

A significant portion of this research is not directly motivated by the Solana context, but is of significant importance for Eclipse and has direct implications for GSVM, as we further discuss in Section 6.3. In particular, Merkle trees can enable light clients and considerably streamline fraud proofs; see Section 6.3.2

for more. This approach notably reduces the volume of data required to be submitted to a data availability layer compared to our existing design. Therefore, any advancements in Merkle trees positively impact the applications running on Eclipse.

5.2.2 eBPF and SBF

Historical overview. eBPF bytecode comprises a 64-bit instruction set for general computing purposes. It is an evolution of an earlier standard known as BPF[3]. As part of this advancement, eBPF comes with a new JIT compiler, which translates eBPF programs into native machine code compatible with common CPU architectures including x86_64, armv8, POWER, and RISC-V. Currently, eBPF enjoys widespread use; it protects network infrastructure at terabit throughput levels through technologies such as XDP and socket filters. XDP can even be implemented on commercially available SmartNIC hardware. Within the Cilium project, eBPF enables routing and load balancing across clusters containing thousands of servers.

eBPF vs. SBF v1. By combining the core capabilities of eBPF with Solana's tailored blockchain enhancements, SBF v1 furnishes Solana's bytecode with the key components required for Turing completeness. SBF v1 implements more rigorous runtime validations for memory access, a crucial aspect since Solana programs engage with accounts and mutable data structures located in on-chain storage. Notably, making the bytecode verifier less strict contributed to the Turing completeness of the SBF VM. Shifting safety verifications to runtime facilitates unrestricted memory access, indirect jumps, loops, and various other dynamic behaviors. SBF v1 ensures that programs can only reach authorized memory areas, protecting against security flaws and unintended modifications. Moreover, unlike eBPF, SBF allows programs to run computations for any period (within the constraints of compute units), thereby offering more robust runtime checks. SBF's opcodes are defined by Solana here: `https://github.com/trudever/solana-rbpf/blob/master/src/ebpf.rs`.

SBF v1 vs. v2. Solana uses its own open source version of BPF technology in Rust called SBF using Rust's `rbpf` crate, which is maintained almost exclusively by the Solana foundation. Below we summarize some of the changes that were introduced in v2 of SBF. Solana's SBF v2 was officially introduced on October 18, 2022, as part of the ongoing migration from the Berkeley Packet Filter (BPF) to the Solana Binary Format (SBF). SBF v2 was designed to increase performance, improve memory safety, enhance security, and optimize CPIs, all while maintaining determinism. Programs using v2 should benefit from faster execution, lower costs, and greater flexibility, making it a critical upgrade for Solana's ecosystem, as summarized below.

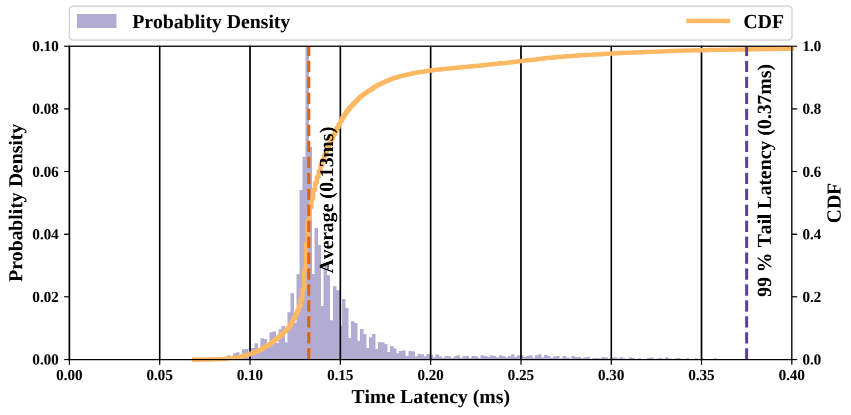
- *Instruction set changes.* SBF v2 introduced *far calls* and a hashed function address scheme, allowing programs to call external functions dynam-

[3]The acronym used to stand for Berkeley Packet Filter, but has lost its meaning.

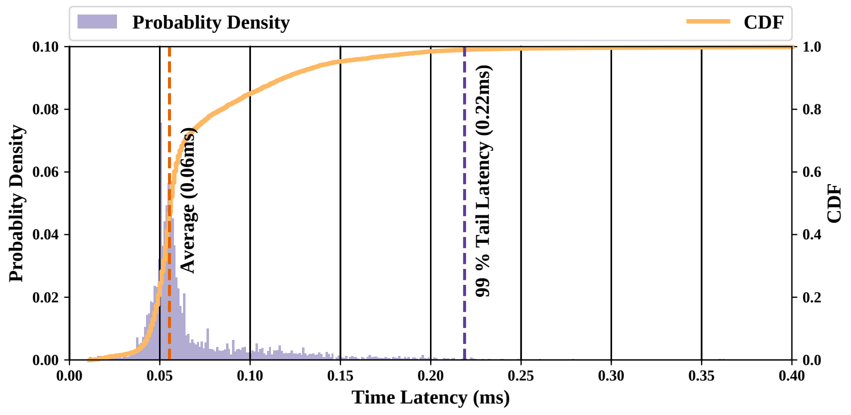
ically; indirect Calls (`callx` can only jump to verified function addresses, eliminating arbitrary jumps; byte swap (`le16`, `le32`, `le64`) and implicit sign extension on 32-bit operations were removed for performance consistency; new explicit unsigned/signed division/modulo instructions reduce JIT overhead; the new instruction allows single-instruction 64-bit constant loads, improving JIT efficiency.

- *Memory management improvements.* Code and read-only data are mapped directly, eliminating unnecessary copying, (zero copy); Cross-Program Invocations (CPIs) now share memory instead of cloning account data, drastically reducing execution cost; the verifier checks memory access at load time, eliminating runtime bounds checks where proven safe.
- *Execution model enhancements.* CPIs no longer spawn new VM instances; they behave like direct function calls, reducing overhead; programs can define multiple callable functions instead of a single entry point; reduced overhead on system calls, CPIs, and memory operations allow programs to use more compute power for logic rather than execution overhead.
- *Security and verification enhancements.* Stricter control flow rules were introduced: jumps cannot exit function boundaries, preventing unintended execution paths; prevents jumping into the middle of multi-word instructions like `LD_DW`; static bound analysis is used to ensure that all memory accesses are proven safe before execution, reducing runtime traps; lastly, programs must declare external function dependencies at deployment using `DT_NEEDED`, blocking unexpected behavior.
- *Performance optimizations.* The verifier's guarantees allow the JIT compiler to omit unnecessary checks, resulting in faster execution; system calls are mapped directly, avoiding VM-to-runtime transitions for built-in functions; lower CPI overhead is achieved through removing account cloning and redundant context switches, significantly speeding up program composability.
- *Backward compatibility and migration.* Loader v4 supports SBF v2, while older programs remain in previous loaders; developers must compile with `cargobuild − sbf` instead of `cargobuild − bpf`. Most high-level Rust/Anchor programs work without modification. SBF v2 programs are deployed on Loader v4, ensuring compatibility, while phasing out the old runtime.

Collectively, these enhancements (especially code/data mapping and reduced overhead on CPIs) allow high-end servers to push more transactions through in parallel, taking advantage of large CPU core counts, something that is relevant to GSVM. However, to date, we have not seen comprehensive measurements of the effects of the v2 upgrade, compared to v1 or eBPF [53], [65], [66]. We have similarly not seen any analysis of the security properties achieved by SBF v2, given some concerns about eBPF security [67]–[69]. Furthermore, we can



(a) Userspace echo server



(b) Unikernel echo server

Figure 11: Comparative performance uplift of unikernel versus userspace designs, illustrated on a simple echo server, from Raza *et al.* [71].

consider further eliminating the need for runtime checking with the help of more static analysis of bytecode [70].

Arena allocators. There is also value in optimizing BPF execution for the particular use cases related to blockchains. We can expect that many of the assumptions underlying the original design intent of BPF must be adjusted for blockchain use. For instance, mainline BPF is designed to operate on small programs, while blockchains are likely to have to execute larger programs as well. Previous work, such as Huang *et al.* [65] shows that some workloads can benefit from a different memory model than the classical memory model of eBPF. Providing an efficient ephemeral arena along with a suitably simple allocator can be beneficial. The de-allocation strategy is similarly simple: pre-allocating blocks for each program, and setting the memory to null bytes before the next execution provides both predictable memory footprint, a reduced pressure on the host operating system heap allocator among other things.

BPF everywhere. One key aspect of the Solana execution engine is that there is ongoing work to integrate as much of the Solana runtime into the BPF programs themselves [72]. If much of the execution environment is itself a BPF program, then it can be loaded into the Linux kernel, SmartNIC devices, and given some modifications to the validator code, it can itself pass through the same consensus and reproducible build pipeline. Furthermore, said validators have a much smaller API and ABI interface to implement, reducing the friction to validator client diversity. Finally, having no boundary between runtime and program allows for aggressive whole-program analysis, link-time optimization, inlining, and dead code elimination.

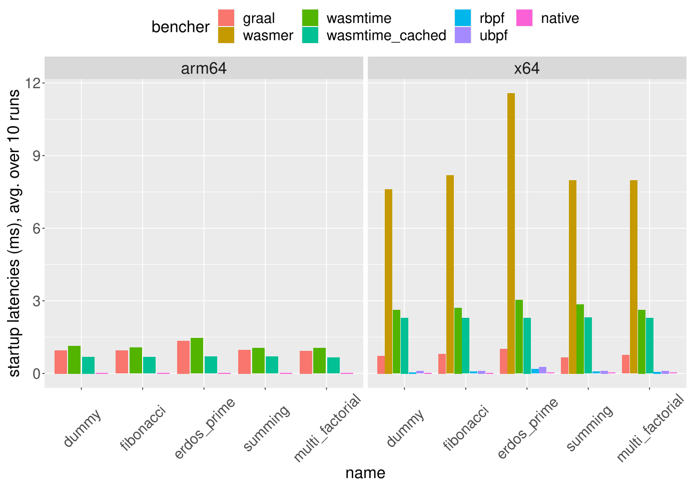
5.2.3 WASM vs. BPF for Faster Execution

There are good reasons to consider replacing the built-in Solana eBPF execution environment `rbpf` with an existing execution environment such as `wasmtime`. Some have observed that the WASM bytecode executes more slowly than eBPF. For example, Figure 12a from Raza *et al.* [71], demonstrates that startup time for virtual machines is lower in case of `rbpf` and `ubpf` and negligible in the case of native code. This seems to agree with the observation that WASM is slower than BPF.

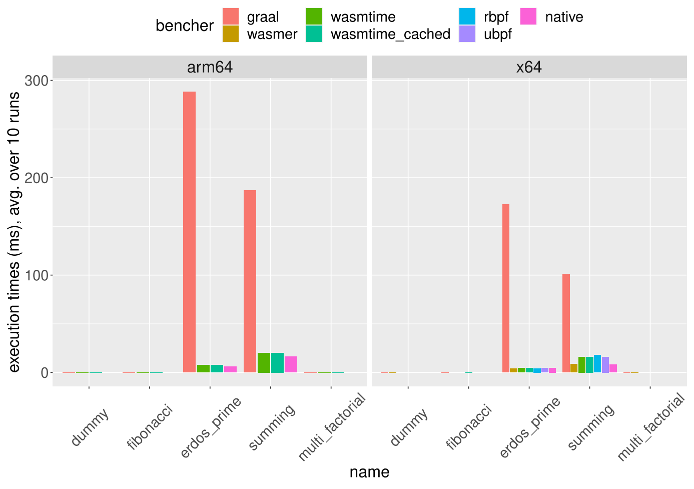
The situation is a little more complex; Figure 12b from Huang *et al.* [65], demonstrate that all WASM runtimes, but one were able to outperform `rbpf` in the `erdos` − `primes` workload. It is unclear if there are more situations in which a WASM implementation could be faster than the eBPF, and these should be explored. Moreover, WASM routinely generates more compact binaries [65], showing a $2--3\times$ difference is size. While this might seem trivial, it represents more than simply decreasing the RAM needs on full nodes. By allowing binaries to occupy less RAM, the runtime could cache a broader array of programs and pre-execute a greater number of them.

WASM has received a great deal of attention outside of performance. It is considered the go-to execution environment in the Hyperledger ecosystem and in the Polkadot ecosystem. There are works on bridging the gap between the virtual machine on Ethereum and the WASM bytecode, such as [53]. WASM has broader coverage, better tooling, and superior documentation [25], [65]. There is indication that WASM can cope with large data sets [73], and unlike *e.g.* the Java bytecode, or even eBPF, is polyglot, in the sense that many more programming languages can produce WASM bytecode that can be at least as efficiently executed as eBPF [65], [74]. There is much ongoing work to verify security guarantees in WASM [67].

Although it is not strictly necessary to move away from BPF, supplementing it with WASM can prove advantageous in specific scenarios. For instance, when



(a) Startup latencies.



(b) Total execution times.

Figure 12: Benchmarks comparing various execution engines and virtual machine runtimes. Credit: Huang *et al.* [65].

smaller binaries are preferred, these cases can be distinguished from those requiring optimal average performance. Furthermore, in some situations, WASM outperforms eBPF. Due to significant differences in the memory and multi-register execution models of eBPF, WASM facilitates application-specific enhancements. WASM, operating as a single-threaded Lisp machine with linear memory, offers substantial optimization potential, appreciated by those acquainted with Lisp. Given our emphasis on optimizations guided by profiling, there are numerous possibilities to explore within this domain. For instance, Eclipse could adopt a

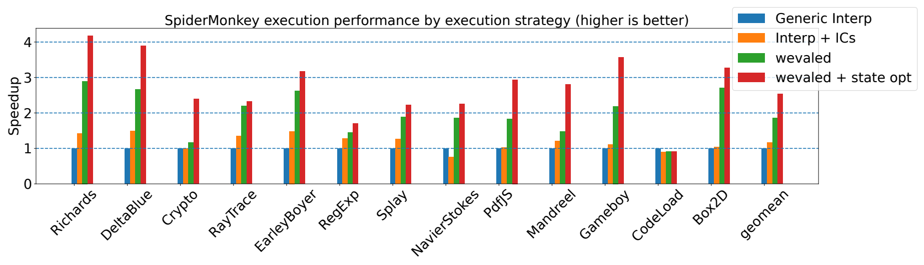


Figure 13: An example of performance optimizations of resulting from the method [75] of partial evaluation. The benchmarks were done using the Octane benchmark suite (SpiderMonkey), with interpreter in a WASM module (without and with IC), and *weval*-compiled code (without and with interpreter-state optimizations). Credit Fallin *et al.* [75].

hybrid strategy: defaulting to *rbpf* for known performance gains, while optionally supporting WASM for apps that benefit from smaller binaries or certain advanced optimizations.

Faster hardware support. Although the choice of eBPF is motivated by performance and is largely corroborated by studies such as Huang *et al.* [65], we believe that there are still many optimizations to be done. We point out that there is a WebAssembly Working Group instituted by the W3C, and in effect since 2017 [76]. They work to bring support for technology such as SIMD instructions to WebAssembly [77]; the equivalent working group for eBPF is a much smaller organization with much reduced scope and less activity [78]. Although there has been no need for any such organization so far, we believe that it is important to give eBPF and its JIT compiler the attention it needs. Although the Solana validator program makes heavy use of vector extensions, and the fact that AVX-512 is well supported [79] on the CPUs that are recommended for running the Solana validators, there is no support for leveraging any vector extensions within `rbpf`, limiting the execution performance of programs iterating over massive amounts of data.

5.2.4 Better JITs

Although the main aim of WASM was to rival native code by leveraging optimizing compilers for execution, JIT scenarios often struggle due to the lengthy compilation process and high memory usage during the initial launch of applications. Consequently, several WASM engines have adopted faster single-pass (baseline) compilers. These baseline compilers operate more swiftly, but still require time and extra code space for rarely executed sections. Given that traditional interpretation is generally ineffective, some engines have opted to compile WASM into a format that is more compact and easier to interpret, rather than converting it straight into machine code. However, this approach also results in time and memory overhead.

A recent paper by Titzer [80] explores the development of a fast in-place interpreter for WebAssembly. The findings reveal that by interpreting WASM code directly, both memory efficiency and speed are achieved, parallel to the performance seen with custom internal formats. This advancement significantly improves the WASM execution layer, leading to faster startup times and minimized memory consumption relative to previous engine designs. Further enhancement of the Just-in-Time (JIT) compiler cache's performance is worthwhile. One method involves adopting partial evaluation techniques [75], [81], which eliminate the need for a JIT. This method effectively doubled the execution speed in SpiderMonkey and achieved slightly less than a two-fold increase in a `lua` variant, as illustrated in Figure 13. Considering that most eBPF programs originate from compiled languages such as Rust and C++, this approach may bridge performance gaps.

Certain JIT implementations, including the V8 compiler, have adopted tracing JITs. These JITs boost execution speed by dynamically tracking and recording the execution routes, or *traces*, of frequently executed segments known as *hot paths*. Unlike standard JIT compilers that compile entire methods or functions, a tracing JIT specifically targets execution paths that are repeatedly run during runtime. By focusing on optimizing only these hot paths, tracing JITs can achieve better performance while avoiding unnecessary work on rarely executed code. These JITs are flexible, tailoring their optimizations according to the program's runtime behavior to more effectively align with specific inputs or usage patterns, something that we expect blockchain-based workloads to be able to greatly benefit from. Given Eclipse's setting, partial-evaluation-based optimizations can eliminate repeated overhead for hot paths in on-chain program execution, further boosting the throughput.

The meta-hybrid JIT compilation approach [82] merges trace-based and method-based compilation techniques to harness their benefits. This is implemented as a meta-JIT compiler framework, enabling the creation of a VM with a hybrid JIT capable of applying various program components simply by crafting an interpreter within their framework.

5.3 Lower-level Internal Representation

RISC-V, an open source instruction set architecture (ISA), has generated significant interest within the blockchain community. This architecture facilitates the development of cost-effective and specialized coprocessors. For instance, these cores might be optimized for tasks like signature verification, as modern cryptographic signature schemes often rely heavily on control flow, and feature dedicated registers for efficient hardware execution of group operations. Currently, several competitive Zero-Knowledge Virtual Machines (ZKVMs), including Risc Zero (`https://risczero.com/`), Nexus (`https://nexus.xyz/`), and Valida (`https://www.lita.foundation/`), mainly use RISC-V as their intermediate representation (IR).

As we elaborate in Section 7.1, consider that AI and ML workloads benefit greatly from specialized tensor cores. For the moment, blockchain systems do not expose APIs that would allow access to the GPU from the program. If one did, however, the on-chain code would be able to rely on efficient decentralized local inference, using Open Source LLMs, such as Deepseek. An example of such a specialized chain that provides access to GPU resources of participating miners is the DeepBrain Chain `https://www.deepbrainchain.org`.

This paves the way for the development of co-processors using GPUs or FPGAs, akin to BPU [83], a dedicated hardware accelerator crafted to boost the efficiency of smart contract execution on blockchain platforms. In conventional blockchain environments, nodes depend on general-purpose CPUs, which often struggle to meet the intensive computational requirements of handling transactions and updating states. BPU overcomes these challenges with a modularized, pipeline-optimized architecture that enhances EVM and smart contract operations, thereby notably lowering latency. This design includes an Application Engine tailored for frequently used smart contracts (like ERC20 token transactions) and a General Smart Contract (GSC) Engine for more comprehensive EVM executions. Experiments reveal that BPU can deliver up to a 191$\times$ speed increase over an Intel i7-7700K CPU when executing Ethereum smart contracts. Hardware acceleration of recurrent processes such as SHA3 hashing, arithmetic, and memory operations facilitates low-latency, high-throughput computations, thus advancing blockchain scalability. The multi-core nature of BPU further increases parallel processing capabilities, supporting simultaneous transaction executions with effective dependency management. Realized on a Xilinx Zynq-7000 FPGA, BPU demonstrates substantial performance enhancements, offering a compelling solution for next-generation blockchain systems, particularly relevant to high-demand dapps and financial dealings.

To illustrate even more aggressive possibilities for computational co-processors, Boutros *et al.* [84] explore the transition from traditional FPGAs to a new class of Reconfigurable Acceleration Devices (RADs) that integrate FPGA fabrics with general-purpose processors, specialized accelerator blocks, and high-performance Networks-on-Chip (NoCs). The authors introduce RAD-Sim, a cycle-level architecture simulator designed to evaluate RAD architectures and optimize application mapping. The tool enables rapid design space exploration, helping architects balance trade-offs between programmability, performance, and resource utilization when migrating applications from conventional FPGAs to RADs.

A practical investigation involving deep learning inference tasks illustrates that RAD-Sim aids in the co-design process of architecture and applications, enabling developers to redesign workloads for network-on-chip-based communication while incorporating specialized hardware accelerators. The findings reveal that three-dimensional stacked RADs can enhance performance by approximately 2.6$\times$ compared to conventional FPGAs, achieving up to 145 TOPS (trillion operations per second) when applied to deep learning tasks. By facilitating high-level modeling of diverse compute architectures, RAD-Sim contributes to the optimization of future reconfigurable devices, enhancing their applicability for data-intensive tasks in cloud and edge computing environments. These deep learning results provide a significant amount of inspiration to us at Eclipse.

As we further argue in Section 7.1, by supporting specialized co-processors (e.g., GPU, FPGA, or RISC-V enclaves) to enable the off-loading of heavy tasks like cryptographic verification or AI inference, Eclipse could boost the use cases similar to those handled by projects like DeepBrain Chain.

The interested reader is referred to a large number of surveys that cover the use of specialized hardware, such as GPUs and FPGAs, in the LLM context [85]–[89].

5.4 Unikernel Designs

One approach that leads to faster nodes and removes the overhead of the underlying operating system is unikernels. For example, Figure 11 shows a 2× latency improvements and a 41% throughput improvement due to a unikernel-based implementation. This approach has been explored in relation to database systems [90], as well as more recent work to blockchains [91]. It is debatable whether a unikernel is necessarily the optimal design for performance [92], given that much of the overhead can be eliminated by alternative means. However, it is generally accepted that unikernel systems offer better security [91], [93], and that is a combination that is much more favorable for the use-case of blockchains than just performance.

By minimizing OS overhead, unikernels allocate more CPU cycles to on-chain program execution, concurrency scheduling, and database I/O—heavy tasks. For instance, a specialized Unikernel-based Eclipse node could combine line-rate NIC offloads with a minimal OS design. One can imagine specialized node types running on top of a unikernel to reduce OS overheads while also reducing the Trusted Computing Base (TCB). This approach can be applied to sequencers, light clients, full nodes, indexers, etc. We see concrete opportunities in centering GSVM design around unikernel ideas, given the fact that we do not need to run nodes on top of the off-the-shelf stack. A unikernel-based node might also dedicate entire cores (as hotspot islands), while relying on minimal OS layers to reduce context-switch overhead, thus preserving cache locality.

5.5 Conclusions

In this section, we have presented a range of attractive options that have to do with improving the execution environment for running programs on Eclipse. The BPF-based JIT approach alone falls short of the true execution potential of GSVM. There are ample opportunities, ranging from using more performant JIT engines to varying our choice of the IR, to using operating system-based approaches such as unikernels to further cut down on the overhead.

6 Optimizing Storage for GigaCompute

RocksDB — the database of choice for the Solana validators — and LevelDB are among the most popular database systems used in blockchain implementations [94]. Despite this, projects such as Monad have identified multiple throughput and latency drawbacks inherent to these traditional databases regarding blockchain storage [28]. In the rest of this section, we outline several strategies that could substantially boost performance beyond what RocksDB offers. Using Eclipse's high-throughput approach to concurrency (Section 4) and the reliance on powerful hardware (Sections 3, 5), our goal with GSVM is to eliminate I/O as a bottleneck.

6.1 Key-Value Stores and Flash Memory

In recent decades, a major topic in database research has been improving key value stores. For comprehensive overviews of key-value stores using flash memory, address translation in flash memory, hybrid SSDs, computational storage, and GPU-accelerated databases, readers can consult a number of relevant papers [95]–[99]. Since Eclipse aspires to GigaCompute-scale throughput, these advanced KVS techniques, particularly those optimized for SSD and flash memory, are critical to sustaining high TPS on powerful hardware. This section focuses on key-value stores that are structured around a flat model consisting of key-value pairs. These stores are typically lightweight and support a limited set of basic operations such as `put`, `get`, and `delete`, with each key being distinct and directly referencing its associated data.

In key value databases, the primary data structure used for indexing is essential. Indexing essentially refers to a technique for retrieving a value by referring to a specific key. Usually, these indexing structures allocate a portion of the memory. A common practice is to maintain the references in memory while saving the actual data on storage devices. However, complications arise when the memory requirement per entry expands as this reduces the total number of stored key-value pairs. Consequently, it is common to shift at least some portion of the index to storage. However, this approach has drawbacks, as it necessitates additional read and write operations to storage for every key-value operation.

This is due to the fact that the index must now both be read and written, increasing the I/O operations required; this is sometimes referred to as I/O amplification. Index structures must balance between memory usage and I/O amplification, with balancing strongly influenced by the type of storage medium. For example, with slow storage, it is less beneficial to store more of the index. Consequently, flash-optimized key-value stores must tailor their approach to the characteristics of flash storage for maximum efficiency, as we discuss below. Typically, additional data structures, such as buffers, are utilized on top of the index to reduce I/O demands. Some of the most common data structures used for indexing in key-value stores include LSM-trees, B-trees, and hash tables.

6.1.1 Flash Support

Flash is a nonvolatile storage medium in which data are stored as an electronic charge on a floating gate between a control gate and the channel of a CMOS transistor. The number of bits that the aforementioned gate can store depends on the internal cell technology. For example, there are single-level cells (SLC) and multi-level cell (MLC) technologies. When the number of levels in a cell increases, so does the number of bits it can store. These technologies, as of now in order of their density, are: SLC, MLC, TLC, QLC, and PLC [100].

Multilevel cell technologies allow us to store more data in a smaller area but also have certain downsides: they result in lower throughput and increased *wear leveling* [96]. Wear leveling means that, after each write, the cell slowly degrades. This results in a decreased performance and eventually leads to a dead cell; this can lead to potential downtime in the worst case. It is therefore considered harmful to do excessive writes to flash. Cell degradation is also not just limited to writes; reads can also degrade the performance of cells, albeit to a lesser degree. This phenomenon is known as *read disturbance* [101]. In the blockchain context, wear leveling and read disturbance lead to availability concerns and can also heavily impact long-term operational costs, reinforcing the need for flash-specific data structures.

NAND. NAND flash technology enables for a compact arrangement of flash cells. This dense configuration is ideal for commercial mass storage; however, it restricts access to cells at the block level, with potential block sizes around 4 KiB [102]. In NAND flash, rewriting a block necessitates its prior erasure. Consequently, even updating a single bit requires rewriting an entire block and, eventually, erasing the previous one. A NAND package is structured into pages that are grouped into blocks, organized into planes, and contained within dies. Furthermore, NAND packages and planes can execute operations concurrently; NAND cells can be stacked vertically, resulting in 3D NAND. The vertical stack feature is significant, as the heat generated can propagate upward, potentially accelerating wear in nearby cells. NAND flash can then be used within a storage device, such as Solid State Drives (SSD) [102]. Note that SSDs can also use different types of storage in addition to flash. SSDs that use flash internally allow for parallelism through a number of independent NAND flash chips, allowing it to process multiple `get`, `put`, and `delete` requests in parallel [103]. This parallelism can be exposed to the host in the form of flash channels.

SSDs. SSDs do not just include non-volatile storage, they come with a controller that allows managing the device and usually come with a small bit of DRAM. SSDs are known as solid-state disks because they contain no moving parts. This is in contrast to more traditional storage, such as HDDs, which require moving a platter and an arm. This allows them to deliver both adequate sequential and random I/O, which would not be possible if parts had to move. There also exist various ways to connect SSDs; such as AHCI through SATA and NVMe through PCIe [104]. Depending on the connection, different protocols can be used, leading to different levels of throughput, latency, and concurrent access. Lastly, SSDs frequently contain firmware responsible for a File Translation Layer (FTL) and a Garbage Collector (GC). An FTL maps virtual addresses to physical addresses [98], [105]. This allows the SSD to determine what it considers an optimal location to store a block. It also eliminates the need for the host to issue erasure commands. For the host, overwrites are still allowed. The FTL determines how to manage this I/O internally. The Garbage collector (GC) is then used to remove dead blocks and move the blocks to more beneficial locations [106]. An optimized program should account for the internal logic of this firmware. There also exist a few alternatives that allow the host to remove the need for an internal FTL and GC (these must then be defined on the host), such as open-channel SSDs [107] and ZNS [108]. Open-channel SSDs are already used in a few key-value designs; The Unwritten Contract of Solid State Drives by He *et al.* offers a glimpse of some of the additional challenges [109]. Eclipse can directly benefit from parallel write throughput aligned with the inherent hotspot structure, leading to effective non-interference.

6.1.2 Performance Concerns

SSDs are known to be significantly faster than earlier storage devices, such as HDDs and tape. However, SSDs themselves do not have homogeneous performance. There can be a significant difference in the latency and concurrency capabilities of SSDs. Some devices such as Z-NAND and NVMe SSDs are known to be able to achieve ultralow latency and are therefore often referred to as ULL devices [110]. To give an example of how low low-latency is: Z-NAND can achieve a memory-read latency of $3\mu s$, which is reported to be $8\times$ faster than the fastest page access latency of modern multi-level cell flash storage [110]. Kourtis *et al.* come with similar numbers, reporting that fetching a 4KiB block on NVMe SSDs took $80\mu s$ and only $12\mu s$ on Z-NAND [111]. They even state that storage starts to challenge the performance of network I/O, as a common round-trip latency of a TCP packet over 10 Gigabit Ethernet is 25–50 μs. These fast devices are generally directly connected to PCIe and the NVMe protocol instead of a more traditional approach such as SAS/SATA and AHCI to achieve optimal performance. SAS and SATA do not match the performance that can be achieved with SSDs, limiting the bandwidth that can be achieved with SSDs. PCIe is better able to match the performance of the device. In the following, we discuss some of the common performance challenges.

Write amplification. Write amplification (WA) means that the physical amount of data that is written on the device is *in excess of* the logical data that are written. For example writing 64 KiB of data to the device, when the application issued only 2 KiB. This can be due to block size, garbage collection, and internals of various data structures. Reducing write amplification seems to be an especially prevalent problem in studies. That is because write amplification can significantly reduce both throughput and increase latency. Another problem is wear leveling, which can degrade flash devices and eventually make them unusable. Write amplification causes wear leveling and therefore inevitably also increases the monetary cost of using these devices. The effect of write amplification is also bigger in multi-cell technologies, such as MLC and TLC.

Read amplification. Read amplification means that the amount of device-internal data that is read is more than the user-visible data that is read. For example, issuing a read of 2 KiB and in the end reading a total of 64 KiB. This can, for example, be because the application needs to find the data, requiring more reads, or because of the block size of flash. This can be problematic in read-heavy stores that require low latency for reads. However, write amplification is generally *more problematic* and more expensive in the long run than read amplification because writes cause wear leveling, which is more harmful than read disturbance, and because of the asymmetric nature of I/O, which makes write operations more expensive than read operations.

Space amplification. Space amplification and space efficiency can also be a problem. Space amplification is an application problem and means that the key-value store stores more data on storage than there are key-value pairs. This is inevitable as some metadata are always needed for persistence. In addition, some key-value stores also reserve space for background operations, further limiting the amount that can be used for key-value data. The main concern is to keep the amount of extra data needed to a minimum. It is also a possibility to resort to compression, to reduce the amount of data that needs to be stored, sometimes creating negative space amplification. Space amplification can be problematic because it increases flash size requirements.

Garbage collection. Garbage collection is also often described as a problem. Garbage collection is caused by data erasure and moves data around in the background. Since the data are moved around, this can cause further write amplification. The background operations also introduce significant latency fluctuations. Works that focus on this problem aim to either reduce the latency fluctuations, use hints to help the garbage collection, or properly separate hot and cold data, which helps the garbage collection by only moving hot data.

Concerns about interactions with flash storage. When memory resources are constrained, it is impractical to keep full index data structures or large caches in RAM. Therefore, efforts are directed towards reducing memory consumption by utilizing more efficient index structures and shifting extra data to flash storage. In scenarios with lower latency and higher throughput, such as NVMe SSDs, the host CPU might act as a bottleneck. Consequently, optimizing the software stack provided by GSVM to fully utilize the available hardware performance is essential.

6.1.3 Indexing Data Structures

Three of the most common data structures used for indexing are Hash Tables, B-trees, and LSM-trees. Many ideas have been proposed to either optimize these data structures for flash or use them as a part of their key-value store. Therefore, we will take a closer look at these structures and go into detail about how they can be optimized for flash. For GSVM, concurrency and flash optimization must go hand in hand: data structures that minimize stalls under thousands of parallel read/write requests are essential.

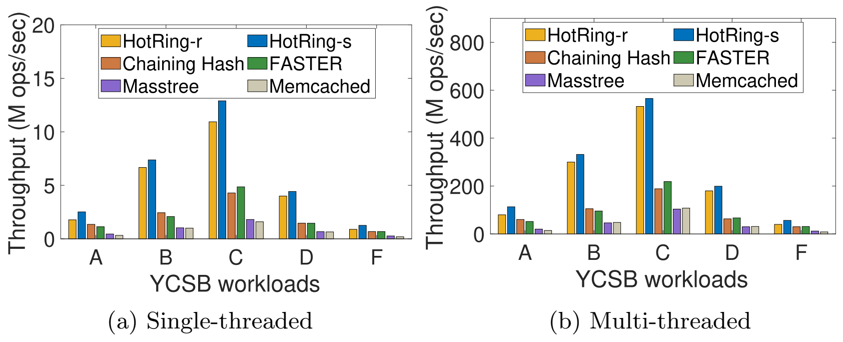


Figure 14: Throughput of HotRing (Chen *et al.* [112])) compared to other systems.

Hash tables. Hash tables as an approach encompass a diverse range of solutions, all anchored on a shared concept: they hold a table of keys associated with the addresses of their respective values, which is why they are sometimes known as mapping tables. This concept is closely aligned with that of key value stores, which are similarly structured collections of keys linked to values. Each mapping within this table occupies a specific position, referred to as a slot. In every solution we explore, the values are stored in flash memory, meaning that only the offsets to these values are kept in the mapping tables. Keys are typically stored alongside values, but frequently form part of the mapping table. When keys are not included, a hash or segment of the key is used instead in the mapping table. Part of the mapping table is kept in memory and can be reconstructed using the saved data. As the number of key-value pairs increases, hash-table structures pose a challenge because their size increases linearly with these pairs. As a result, the entire hash table might not fit in memory, requiring frequent storage and retrieval from permanent storage. This process can lead to read amplification either when the memory is scarce or in a heavily parallel environment.

Recognizing the workload *hotspots* is crucial in deciding which data should be cached. This approach is aligned with the Solana model of local fee markets, where access hotspots can be assigned different prices independently of the broader fee environment. The absence of hotspot awareness in current key value stores results in suboptimal performance for highly skewed workloads, such as those found in the blockchain context. Chen *et al.* [112] investigate hotspot-aware methodologies for in-memory index structures in key-value stores. They conduct an initial analysis of the potential advantages of ideal hotspot-aware indexes and discuss the challenges, such as hotspot shifts and issues with simultaneous access, in effectively utilizing hotspot-awareness. This parallels our concept of hotspot islands (Section 4), where high-skew workloads can be localized to dedicated CPU cores, complementing the hotspot-aware indexing at the database layer.

In light of these observations, they introduce a key-value store called HotRing, designed specifically to handle extensive simultaneous access to a limited subset of records. HotRing utilizes an ordering hash index framework that boosts the speed of accessing frequently requested items by adjusting head pointers closer to them. Additionally, it implements a lightweight mechanism to identify changes in hotspots in real time. The design of HotRing extensively incorporates lock-free structures for both standard operations (such as read and update) and specific operations related to HotRing (including hotspot shift detection, head pointer adjustments, and ordered-ring rehashing), thereby enhancing the efficiency of multicore architecture under high concurrency. Comprehensive experiments demonstrate that our method can achieve a 2.58× performance boost over other in-memory key value stores when dealing with highly skewed workloads.

B-trees. B-trees are a widely adopted tree data structure extensively utilized in various databases, particularly conventional relational databases [113], [114] and databases designed for intensive read operations, such as key value stores [115]. These structures are renowned for their efficient read capabilities and were often used in the early development of flash-based key-value stores, serving critical functions in systems such as FlashDB [116]. B-trees are not only of historic significance; they continue to be vital components in some modern systems such as WiredTiger, ForestDB, and Tucana [115], [117], [118]. In addition, they are sometimes utilized as elements within more complex structures, such as LSM-trees [119]. Their exceptional read performance is due to the constraint of read operations by the height of the tree, which requires just a single read per level. This feature renders B-trees especially beneficial for key-value stores with a predominant focus on reads.

Both the B-tree and the B+-tree are M-ary trees and both are designed to be self-balancing. This is accomplished by automatically merging and splitting the nodes on insertions and deletions. All of the leaves are on the same height. This ensures that the upper bounds on the required number of reads are equal to the height of the tree. What differs between the two is what is stored in nodes and in leaves. In the B-tree design, nodes and leaves contain key-value pairs with the key used as the index. In B+-trees on the other hand, nodes can only contain keys in the form of pivots. The keys can be duplicated all the way from the root to the leaf; leaves store the actual value pointers. In addition, the leaves can be linked. This linking has a major advantage because sequential access can be improved. Scan operations, for example, can go directly to the next linked leaf and do not require traversing the entire tree again for each subsequent read.

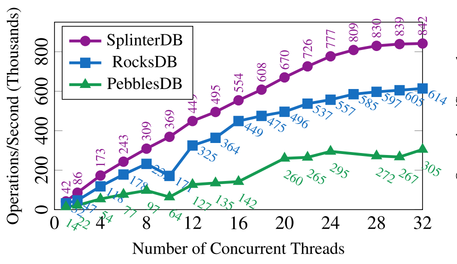
Another advantage of the B+-tree is that the values are stored only in leaves and not on a random level, which adds to the search efficiency. The key-value pairs are also sorted in the data structure, allowing the client to read only one node on each level of the tree. The earlier stated read performance is due to this sorting. B+-trees store each node and leaf separately on different blocks on the flash device. There is no guarantee that nodes and leaves will fill entire blocks, which can waste space and cause write amplification as the entire block still needs to be written [120]; amount of written data can be reduced with compression. Data is also not stored sequentially because inserts and updates change parts of the B-tree rather than appending their changes to the end. Reading, inserting, and updating thus incurs random I/O, as data need to be written to random locations. Random I/O causes performance overhead, but not just because of the storage medium.

We will demonstrate the reasons behind the occurrence of write amplification through examining insert operations. Generally, parts of index data structures are memory-cached. During an insert, data is written to the leaf blocks. If the required block resides in the cache, the write is performed directly in the cache. Otherwise, the block must first be fetched from disk for modification. Due to limited memory availability, the old cached block is typically written back to I/O. Kuszmaul *et al.* note that, in the worst-case scenario, each insertion can require rewriting an entire block [120]. When data is stored sequentially, multiple inserts can be consolidated and written simultaneously. Moreover, B+-trees inherently lack substantial buffering, which complicates the amortization of such operations. For GSVM, the ability to handle multiple simultaneous inserts without incurring excessive lock contention or random I/O is critical, especially under bursts of transaction load.

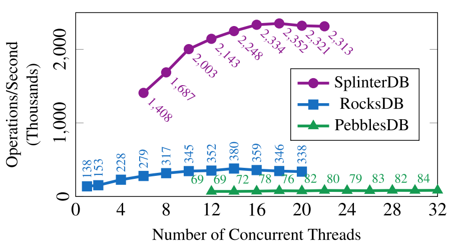
B$^\epsilon$ trees. The B$^\epsilon$-tree enhances the traditional B+tree by incorporating buffers into all its intermediate nodes. The proposal is to modify the original B+-tree design to include buffers; these buffers serve to queue operations, like writing new data or updating existing data, thereby distributing the expensive I/O tasks over time. Once a buffer reaches full capacity, it offloads its contents downwards through the tree until it reaches the leaf nodes, which lack buffers. B$^\epsilon$-trees excel in write-intensive environments without the drawbacks of compaction operations seen in LSM-trees. However, they incur increased random I/O demands, making them effective primarily on devices with rapid random I/O capabilities, such as NVMe SSDs.

When the entire index can fit into memory, this design proves to be especially useful, as it often demands a considerable memory space. An improved and optimized version of this design was implemented by Papagiannis *et al.* [118] in Tucana, which demonstrated reduced software overhead and enhanced throughput. Additionally, akin to WiscKey's approach [121], separating the index structure from the value data is feasible. This separation helps keep the index more compact and minimizes write amplification.

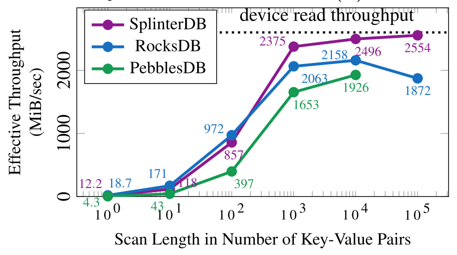
The continuing issue with the B$^\epsilon$-tree involves considerable write amplification, even with buffering. This occurs because any modification to a buffer necessitates rewriting the entire buffer. Conway *et al.* have introduced another data structure, the STBe-tree [122], which integrates concepts from both the LSM-tree and the B$^\epsilon$-tree. Instead of node-level direct buffers, it employs multiple sub-indexes formatted as B-trees, replacing the B$^\epsilon$-tree's log buffers. These B-trees can be updated separately, which reduces write amplification. Similar to the main B-tree, each sub B-tree should be accompanied by an AMQ, like a



(a) Read concurrency



(b) Insert concurrency



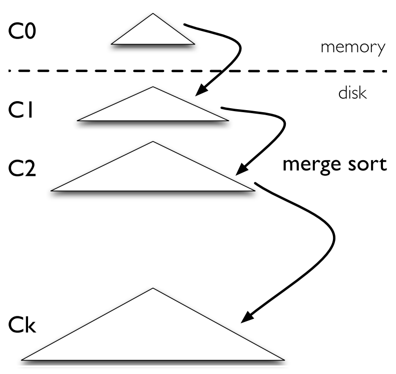
(c) Scan throuhgput

Figure 15: SplinterDB performance improvements from Conway *et al.* [122].

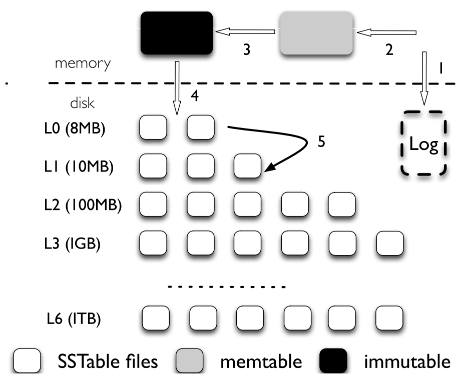
bloom filter, to minimize the number of necessary reads for these trees.

SplinterDB introduces a new concept for the STBe-tree known as flush-then-compact [122], which enhances I/O and CPU parallelism. Typically, data transfer from a parent node to a child node occurs through a process called flushing. The movement of data between nodes, requiring reorganization at the target node, is called compaction. This process necessitates holding locks while moving all values, leading to costly, immediate I/O operations. In the flush-then-compact approach, however, a pointer-swing is utilized instead. References to the active branches in the child node are copied from the parent, ensuring that locks are held for only a brief period, thereby improving concurrency. Subsequent value copying operations can be deferred and conducted asynchronously, without locks, allowing them to be easily scheduled in parallel. This approach could be used in other designs using compaction operations. As shown in Figure 15, SplinterDB outperforms RocksDB by a factor of 6–10$\times$ for insertions and $2 - 2.6\times$ for point queries, while offering performance comparable to RocksDB in small range queries (generally not so relevant for blockchain). Additionally, SplinterDB reduces write amplification by 2$\times$ compared to RocksDB. Such flush-then-compact strategies could mesh nicely with Eclipse's approach of greater parallelism and lock-free processing.

LSM-trees. On flash, storage writes are more expensive than reads, which makes it attractive to look for a data structure that is optimized for writes instead of reads. These types of data structures are commonly referred to as write-optimized indexes. An example of such a data structure is the log-structured



(a) Typical structure of an LSM-tree



(b) Structure proposed by Lu *et al.*

Figure 16: This figure from Lu *et al.* [121] shows the standard LSM-tree and LevelDB architecture. For LevelDB, inserting a key-value pair goes through many steps: (1) the log file; (2) the memtable; (3) the immutable memtable; (4) a SSTable in L0; (5) compacted to further levels.

merge tree (LSM) [123]. Most modern flash-optimized key value stores make use of LSM-trees [121], [124], [125]. Nevertheless, just putting a plain LSM-tree on flash storage does not immediately make it optimized. There are a few optimizations that can be done to increase its performance, most with trade-offs. Some of the more novel solutions will be discussed in more detail below. We will first discuss the general LSM-tree structure, followed by various optimizations and trade-offs.

The initial purpose of the LSM-tree, dating back to 1996, was its utilization with HDDs, a period when HDDs were prevalent and DRAM had expanded enough to support buffering. As a result, several LSM-tree features were specifically tailored to suit these system traits. The basic setup is initialized with a minimal section of data stored in memory, followed by several tiers on disk. Each successive tier increases the size, and when one fills up, it is merged in a sorted manner into the next tier. This straightforward structure has undergone significant optimization and modification, leading to the modern adaptation frequently referenced in the current literature on flash key-value stores; a notable implementation of this is found in LevelDB, used by Ethereum.

There are four substructures, chained together in a multi-stage (MS) structure. A sorted in-memory table, a sorted immutable in-memory table, a write-ahead log (WAL) on storage, and multiple sorted string tables (SSTables) that are written to storage. We consider how the data propagate through the structure by describing all the steps on the right-hand side of Figure 16 below.

1. On an insert or update, every key-value pair is first inserted into the WAL, which is an append-only log. This is similar to *journaling* in traditional databases and guarantees persistence and consistency [126]. In recovery, the WAL can be replayed as it contains all changes in order.

2. After writing to the WAL is complete, the same data are written again, but then to the in-memory table, which is typically small, a few MBs [127]. The in-memory table functions as a buffer that can be used to avoid small writes to disk.

3. When the in-memory table reaches a certain size, it is converted to an immutable in-memory table. These immutable tables should eventually be written to storage. At the same time, a new mutable memtable is created, or an old memtable is reused. The immutable in-memory tables are used to allow writes to continue to memory without a prolonged wait. Otherwise, the key-value store has to wait for the entire I/O operation to complete. New data is written to the mutable memtable, while the old immutable table is written to storage.

4. The immutable in-memory table is written to storage as is. It is written as a sorted file, known as a Sorted String Table (SSTable). Such an operation is known as a flush. An SSTable is sorted on keys and contains a subset of the total range of keys. The SSTable itself is thus essentially an immutable ordered collection of key-value pairs, sorted on keys. The SSTables are stored on storage in a series of levels. Reaching from level 0, commonly referred to as L0, up to level N, referred to as LN. The number of levels depends on the design and can be tweaked. Each level can hold only a certain amount of SSTables. This amount increases with each level, so higher levels can hold more data. LevelDB, for example, maintains a growth factor between levels of $10\times$ by default. For correctness, only two levels would be enough, but adding more levels helps with amortizing step (5).

5. Data can be moved to a higher level with a process known as compaction. Compaction moves a number of SStables from one level to the next level. This is done with an n-way merge on the SSTables in the next level. In this context n-way merge approximately refers to merge sorting on multiple SSTables in the next level. Since each SSTable is limited to a certain size, this can also introduce new SSTables and, in the worst case, create yet another compaction for the next level. This process thus leads to *cascading* write amplification, as writes have to be done for each level [128], instead of writing them just once in total. This write amplification can reach numbers as least as high as $50\times$ in LevelDB [127]. More than two levels help with amortizing compactions because updates move down multiple levels, not forcing an n-way merge with all the data on every compaction.

Because of compaction, we can be sure that each level contains non-overlapping SSTables, except for L0, the very first level. On L0, tables are simply appended. So whenever a read operation is performed, this can lead to one read in the mutable memtable, one in the immutable memtable, one for each SSTable in L0 and one for each additional level. Thus, read operations have significant read amplification. This is often mitigated by the usage of AMQs, such as bloom filters.

LSM-trees are favored for use with flash storage because of their reliance on appends and effective buffering. By utilizing LSM-trees, the costly nature of erase and write operations on flash can be alleviated, as these trees promote sequential rather than random I/O operations, which in turn minimizes the issues associated with expensive erasures and garbage collection. Moreover, LSM-trees mitigate the impact of frequent small writes by employing DRAM buffering, which increases throughput and reduces both latency and write amplification. However, LSM-trees also have drawbacks. Their leveled architecture leads to significant read and write amplification. Additionally, the compaction process, crucial for reducing the latency, diminishes available throughput and comes with a high software overhead, making it costly.

To highlight a project that uses software-hardware co-design, Zhou *et al.* [129] explore an alternative route to reduce CPU workload by utilizing GPUs to boost the efficiency of LSM storage engine compaction on high-performance SSDs, particularly for small to medium KV sizes. Their work addresses the computational limitations of compaction by developing streamlined GPU compaction units for every compaction stage and employing a layered acceleration technique across various compaction levels. Furthermore, they introduce two SSD-GPU data transfer methods, the pipeline mechanism and the P2P mechanism, to reduce the overhead of data transfers during compaction. Their GPU-driven compaction strategy is implemented on LevelDB and is evaluated against both basic CPU compaction and the leading-edge GPU-accelerated compaction approach, LUDA. This approach yields some notable enhancements: the compaction speed increases by as much as $3.61\times/2.24\times$; the write throughput sees an increase of up to $2.34\times/1.51\times$; and the combined read/write throughput improves by up to $2.02\times/1.30\times$, respectively.

6.2 Custom Hardware Support for KV Stores

Yang *et al.* [130] introduce an innovative key-value store tailored for scalable high-performance data processing, specifically in environments with intensive read, insert and scan operations. While traditional KVS systems often face challenges with workloads heavy in insertions and range queries, HeteroKV uses a diverse data structure that integrates a B+ tree with cache-efficient partitioned hash tables. The solution is implemented on a CPU-FPGA hybrid platform, where an FPGA-based dispatcher effectively organizes and allocates key-value requests, thus minimizing memory bottlenecks and reducing synchronization overhead on the CPU. This architecture supports high throughput and minimizes the latency across various workloads by harnessing software-hardware co-design. Experiments have shown that HeteroKV surpasses current state-of-the-art systems, reaching 430 million read operations per second (a $1.5\times$ improvement), 315 million insert operations per second (a $1.4\times$ improvement), and 15 million scan operations per second (a $5\times$ improvement) over existing systems. The FPGA-enhanced indexing and dispatching feature optimizes resource use, supporting *line rate performance* with batch processing. By merging a hardware-optimized dynamic B+ tree with cache-aware hash tables, HeteroKV delivers a fast, balanced key-value store.

To revisit a theme of near-wire processing we highlighted in Section 3, Li *et al.* [131] introduce an innovative method to enhance the performance of the key value stores by utilizing programmable NICs equipped with FPGA to shift KV tasks away from the CPU. Conventional in-memory KVS frameworks encounter CPU limitations, particularly as network bandwidth grows. Although RDMA-based approaches provide some relief, they are hindered by restricted primitives and synchronization burdens. KV-Direct enhances RDMA capabilities, permitting direct key-value access on host memory and completely bypassing the CPU. It uses a hash table and a memory allocator tailored for optimal PCIe bandwidth, and employs out-of-order execution to handle dependencies effectively. Experiments indicate that a single KV-Direct NIC can perform up to 180 million key-value operations per second, matching the throughput of multiple CPU cores, maintaining low tail latency (under 10 μs), and tripling power efficiency. Furthermore, KV-Direct exhibits near-linear scalability with added NICs, achieving up to 1.22 billion operations per second with 10 programmable NICs within one server—an improvement of nearly 10× over current systems. By removing the CPU dependency and optimizing PCIe and memory interactions, KV-Direct establishes a promising new milestone for high-performance, scalable in-memory key-value stores. These methods align perfectly with Eclipse's design principles, where offloading key-value operations to specialized hardware could help sustain near line-rate transaction throughput (Section 3).

Multi-stream SSDs. Multi-stream SSDs are SSDs that allow the host to mark write commands with a stream hint. Normal SSDs have a single append point, also known as a stream, where all new data is stored. Multi-stream SSDs allow for writing to multiple different streams [132], [133]. These streams can then be handled differently by the underlying FTL. This allows data to be differentiated and can thus help separate hot and cold data, which in turn helps to bridge the aforementioned semantic gap. It focuses on improving the communication between layers but does not remove any layers of itself. The main advantages of this improved communication will be more efficient garbage collection and reduced AWA.

This technology shows significant potential; for instance, Cassandra [125], which employs an LSM-tree, can utilize multi-stream SSDs by assigning different stream ids to separate segments of the store, effectively distinguishing hot and cold streams [133], [134]. The Write-Ahead Log (WAL) might be assigned to one stream, while bloom filters, cache, and metadata go to another. Additionally, different streams can be allocated for various LSM-tree levels, which considerably diminishes write amplification and enhances the efficiency of garbage collection. These concepts were tested similarly on RocksDB, employing strategies similar to those used with Cassandra [94], [115].

Multistream SSDs find application in B-tree architectures, such as WiredTiger in MongoDB [115]. In MongoDB, data is stored using files. The authors point out that simply assigning separate files to different streams is insufficient for their needs because it fails to solve the problem of internal data fragmentation. Instead, they suggest a boundary-based stream mapping approach, where sections of individual files can be distributed across multiple streams, thereby enabling differentiation between file sections. For example, some parts of a file can be classified as hot or cold. This aligns with the usage characteristics that we see in the blockchain context.

Native FTL support. An FTL maps virtual addresses to physical addresses [98]. This allows the SSD to determine what it considers an optimal location to store a block. It also eliminates the need for the host to issue erasure commands. There exist various SSDs with FTLs that already provide various transactional operations and persistency guarantees. Instead of adding layers on top of these FTLs with similar functionality, it can be advantageous to make direct use of the FTL.

The NVMKV architecture [135] works exclusively on devices equipped with specific features, such as atomic multi-block writes, persistent trim operations, and capabilities for existing and iterative operations. NVMKV translates all key-value tasks, such as get, put, and delete, into these FTL operations. Existing FTL operations already encompass guarantees like persistence and recoverability. Within such frameworks, keys are converted into hashes that devices interpret as logical addresses, which the FTL then maps to their actual physical locations. Consequently, the device fully manages the storage location of a hash. Hash collisions may be addressed using methods like linear probing, though generally these methods depend on the store's design and are outside the scope of this survey. Atomic multi-block writes facilitate the atomic writing of files, persistent trim operations ensure deletes are permanent, and iterate operations are applicable for scans, among others. This approach simplifies key-value store design, rendering it as a thin layer atop the FTL; in other words, the FTL serves as a key-value store itself.

An influential key-value store that contributed to the adoption of running these stores on Open-Channel SSDs is the design known as the LSM-tree based Key-Value Store on Open-Channel SSD (LOCS) [103]. This particular store has been realized using Software Defined Flash (SDF) [136], a variant of SSD developed by Baidu that possesses open-channel characteristics. Unlike typical SSDs, SDF effectively presents each channel as a separate device to the host.

FlashKV [137] presents a novel LSM-tree-based key-value store optimized for open-channel SSDs to improve storage performance and reduce write amplification. Traditional key-value stores running on SSDs suffer from redundant management layers in the LSM-tree, file system, and Flash Translation Layer (FTL), leading to inefficiencies and high overhead. FlashKV eliminates these redundancies by bypassing the file system and FTL, directly managing the raw flash device at the application level. FlashKV further leverages parallel data layout to exploit the SSD's internal parallelism, adaptive parallelism compaction to reduce write interference with reads, and compaction-aware caching to optimize memory usage. Additionally, a priority-based I/O scheduler ensures that client queries are prioritized over background compaction tasks. The evaluation demonstrate that FlashKV achieves 1.5× to 4.5× better performance compared to LevelDB and reduces write traffic by up to 50% under heavy workloads. By directly integrating hardware-aware flash management with LSM-tree optimizations, FlashKV effectively reduces write amplification and garbage collection overhead while improving throughput and latency. The results indicate that open-channel SSDs, when efficiently managed, can provide significant advantages for high-performance key-value storage systems. This approach aligns with the scheduling methodologies used in other LSM-tree frameworks such as SILK [138], where L0 compactions are deemed the most crucial and given precedence.

FlashKV [137], LOCS [103], and SILK [138] all come with different scheduling ideas, and yet all achieve significant performance gains. This raises the concern that scheduling operations and compactions are important for getting adequate throughput and latency. The key lessons that could be learned from these designs revolve around the principles of scheduling each operation differently, differentiating between client and background operations, and how operations can be scheduled across channels to achieve better throughput.

LeaFTL is a flash translation layer (FTL) [105] that uses a learning-based approach to manage address mapping through linear regression, adapting to dynamic data access patterns during runtime. By aggregating a substantial number of mapping entries into a cohesive segment, it significantly minimizes the memory usage of the address mapping table, enhancing data caching efficiency in SSD controllers. LeaFTL integrates several optimization strategies, such as verifying out-of-band metadata to handle prediction errors, refined flash allocation, and dynamic compacting of learned index segments. LeaFTL uses a validated SSD simulator in conjunction with a real-world open-channel SSD board. Performance evaluations across diverse storage workloads reveal that LeaFTL reduces mapping table memory usage by 2.9× and enhances overall storage performance by 1.4×, compared to the baseline FTL techniques. Multistream, open-channel, or custom FTL support can isolate high-write streams from read-critical data, allowing to balance the effects of new transactions with background compaction needs.

Zoned SSDs. The Zoned Namespace (ZNS) SSD [139] introduces an alternative device interface by expanding upon the NVMe specification with its own guidelines for ZNS. It serves as the successor to open-channel SSDs. The purpose of these specifications is to standardize techniques and enable optimization sharing between different devices. The central idea of the ZNS specification is the division of the device's storage capacity into distinct zones. Although these zones can be read in any order, writing to them must follow a sequential pattern, and any zone modification requires prior erasure. This aligns well with the properties of flash memory, potentially enhancing internal data management, boosting write speeds, reducing quality of service (QoS) deviations, and increasing storage efficiency. In a manner similar to open-channel SSDs but distinct from traditional block interfaces, a significant portion of the logic, such

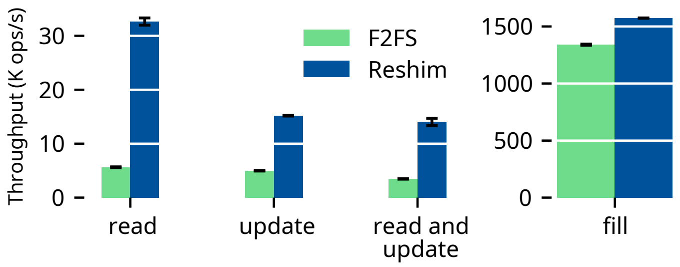


Figure 17: Reshim [140] allows dramatically faster writes and updates in WiredTiger. The isolation across logs eliminates contention and improves both read and update latency.

as a host-managed Flash Translation Layer (HFTL), can be delegated to the host.

Bjørling *et al.* [108] introduced a key-value store designed to operate on a ZNS SSD by enabling RocksDB to leverage ZenFS, a file system optimized for ZNS. This approach enhanced throughput and reduced write amplification relative to traditional block interface file systems like F2FS and XFS, primarily due to significantly diminished garbage collection effects

Bjørling *et al.* [108] introduce Zoned Namespace (ZNS) as a new interface standard for flash-based SSDs, aiming to eliminate inefficiencies associated with the traditional block interface. Traditional SSDs rely on a Flash Translation Layer (FTL) to manage logical-to-physical address mapping, garbage collection, and wear leveling, which results in high write amplification, unpredictable performance, and excessive DRAM and storage over-provisioning. ZNS SSDs, on the other hand, expose flash erase block boundaries and sequential write constraints to the host, shifting data placement responsibility to software. This allows ZNS SSDs to reduce garbage collection overhead, minimize DRAM usage, and maximize usable storage capacity while still maintaining media reliability. The authors evaluate ZNS-based optimizations on Linux, including modifications to f2fs (a flash-optimized file system) and RocksDB using ZenFS, a specialized ZNS-aware backend. Experiments show that ZNS SSDs achieve up to 2× higher throughput, 64% lower average read latency, and significantly reduced write amplification compared to traditional SSDs. By offloading data management to software, ZNS provides greater efficiency and performance predictability, making it an attractive alternative to conventional block-interface SSDs, especially for workloads with sequential writes, such as databases and log-structured storage systems. Such ZNS-backed designs are ideal for GSVM, ensuring that compaction or heavy writes do not stall critical read paths; zoning could be mapped to hotspot islands for further concurrency.

Purandare *et al.* presented Reshim [140], which acts as a user-space intermediate layer to improve data placement strategies based on affinity and lifespan, without needing changes to the operating system or applications. We demonstrate Reshim's easy integration with host-device cooperation in three well-known data-intensive applications: RocksDB, MongoDB, and CacheLib. These applications benefit from 2–6× higher write throughput, up to 6× reduced latency, and less write amplification when using Reshim, compared to filesystems like f2fs. Reshim rivals the performance of specialized application backends such as ZenFS, while offering broader applicability, lower latency, and better data placement benefits. Reshim also underscores the benefit of separating the complexity of placement logic, enabling the deployment of flexible placement strategies across varied applications and storage systems. Figure 17 illustrates a substantial improvement in the WiredTiger backend for MongoDB, particularly highlighting notable enhancements in read and update performance when using the LSM mode. As depicted in the figure, Reshim with multi-threaded readers achieves speeds over 6× faster, with workloads favoring writes reaching up to 3× faster. This efficiency is facilitated by the capability to segregate logs that are directed to the random write area from streams of LSM files residing on zones. The distinct separation of mmap-writes in a dedicated random write area with its exclusive buffers, combined with log-structured writes directed to a specific zone, contributes significantly to performance enhancements. In the context of Eclipse, each island from Section 4 could be given its own dedicated zone.

Near data processing. Near data processing is the practice of moving the computation close to the data. SSDs could for example come with their own computation units such as ARM chips or FPGAs. Such devices are also known as computational storage and various other names [97]. It allows the processing logic to be almost completely moved to the device with the benefits of having the computation close to the data and a lower lookup time. A beneficial side effect is not having to rely too much on the computing power of the host. Therefore, it takes the complete opposite direction of LOCS and ZNS. We recommend a survey by Lukken *et al.* [97] for more information on these devices.

Sun *et al.* [141] introduced Collaborative-KV (Co-KV), a novel approach that employs near-data processing specifically for key-value stores. Co-KV leverages LSM-trees by shifting the compaction tasks to the device itself, eliminating the necessity for data transfer between the host and device that is primarily intended to bypass compaction. Comprehensive experiments indicate that Co-KV generally increases overall throughput by approximately 2× and reduces write amplification by as much as 36.0% compared to LevelDB. When subjected to YCSB workloads, Co-KV enhances throughput by a factor of $1.7-2.4\times$, while simultaneously reducing write amplification and average latency by up to 30.0% and 43.0%, respectively.

Increasing concurrency of data access. A design that is frequently used to increase concurrency capabilities is the *shared nothing* approach. Each thread has its own data structures, and sharing of data between threads is kept to a minimum. Such an approach ensures that threads do not have to content for the same resource. This essentially removes locking issues and allows each thread to use its full capabilities. Some of such ideas used for key-value stores include: dividing the key-value stores index structures across threads [142], [143], letting each thread handle a subset of the key space [142] and per thread WALs [122].

Kvell uses a data structure that is optimized around the idea of *shared nothing* [142]. They propose giving each thread a subset of the key space and its own B-tree index structure. The B-trees are supposed to be completely stored in memory, negating ordinary B-tree drawbacks. Such a design can require some extra DRAM, which can be reduced by only storing the prefixes of keys in the B-tree. This is similar to what we have seen for hash designs. The design of Kvell also does not force storing data in sequence in any way, reducing the need for communication among threads but instead allowing more random I/O to occur.

Another fruitful idea, which is adjacent but not the same as shared nothing, is to make use of multiple memtables in the LSM-tree design. For example, using two mutable memtables for each individual channel [103] instead of only one mutable and one immutable table in total. This makes better use of both the internal flash parallelism and of the CPUs parallelism and does not rely on a single point of operations. With the increasing performance of storage, we can only expect to see more of such work in the future.

6.3 Authenticated Databases

Authenticated databases play an essential role in the blockchain ecosystem, where data integrity, security, and transparency are of utmost importance. These databases utilize various authenticated data structures (ADS) to ensure that data can be verified and trusted without reliance on a central authority. There is a long history of using authenticated data structures such as Merkle trees in the blockchain context.

Data structures in Ethereum. Ethereum has been at the forefront of some of the innovation. Specifically, Merkle Patricia Trees (MPTs) used in Ethereum offer key advantages over Merkle Trees, especially for systems like Ethereum:

- *Efficient key-value storage*: MPTs are designed for key-value storage (e.g., mapping addresses to balances), making them ideal for blockchain state management, unlike Merkle Trees, which are better for lists of items.
- *Prefix compression*: MPTs compress common key prefixes (e.g., shared address characters), reducing storage overhead, while Merkle Trees store keys in full.
- *Efficient updates*: MPTs only recompute nodes along the update path, making them faster for frequent state changes, whereas Merkle Trees require recomputing the entire path.
- *Sparse data support*: MPTs efficiently handle sparse datasets by storing only used keys, unlike Merkle Trees, which waste space on empty keys.
- *Fast lookups*: MPTs use a trie structure for quick key-based lookups, while Merkle Trees require traversing the entire tree.

- *State proofs*: MPTs generate compact proofs for key-value pairs, essential for verifying state transitions in blockchains, whereas Merkle Trees are less optimized for this.

In summary, MPTs are better suited for blockchain systems due to their efficient key-value storage, prefix compression, fast updates, sparse data handling, quick lookups, and support for state proofs.

Ethereum is in the process of eventually transitioning from Merkle Patricia Tries (MPTs) to Verkle Trees as part of its ongoing efforts to improve scalability, reduce storage requirements, and enhance the efficiency of the Ethereum network. This transition is particularly important for Ethereum's shift to a stateless client model, which is a key component of Ethereum's long-term scalability roadmap. Ethereum's move to Verkle trees is a critical step in its evolution toward greater scalability, efficiency, and decentralization. By reducing proof sizes, storage requirements, and bandwidth usage, Verkle trees enable stateless clients and pave the way for future upgrades like Danksharding. This transition is essential for Ethereum to support a growing user base and maintain its position as a leading blockchain platform. Figure 18 summarizes the expected benefits.

Polkadot. Recent studies in the Polkadot framework are advancing the development of merkleized data structures aimed at optimizing SSD performance [144], [145]. By applying the methods detailed in these posts, it is feasible to decrease the latency caused by disk operations in Merkle trees by more than ten times, reduce computational workload by more than half, and significantly lower total disk IOPS by a factor of 25 or more. Eclipse can likewise apply certain methods mentioned earlier for SVM state merklization, achieving significant concurrency without increasing the I/O overhead.

Avalanche. Firewood, a key-value database by Avalanche [146], is custom-made for managing large blockchain states using Merkle trees with minimal overhead. Built from scratch, it writes trie nodes directly to disk. Unlike conventional state management tools that use generic databases like LevelDB or RocksDB, Firewood utilizes trie architecture for on-disk indexing akin to a B+-tree. This eliminates the necessity of "emulating" a trie in a database not designed for its unique structure, enabling fast iteration crucial for state-sync queries without index compaction. Initially created to provide a highly efficient storage layer for the EVM, Firewood is adaptable for any blockchain that requires authenticated state management. Firewood focuses on maintaining only the latest revisions on disk, efficiently removing obsolete data once the revisions expire. Configured to retain a specific number of prior states, each revision generates a new root that can point to new or existing nodes from previous revisions. During the creation of the revision, a list of redundant nodes is identified and recorded both on disk in a future-delete log (FDL) and in memory. Once a revision expires, the nodes marked for deletion during its creation are freed. Firewood ensures system recoverability by avoiding references to new nodes in a revision until they are safely written

Feature	MPTs	Verkle Trees
Proof size	Large (logarithmic growth)	Small (constant size)
Storage requirements	High (many intermediate nodes)	Low (compact proofs)
Scalability	Limited by proof size and storage	Highly scalable
Stateless clients	Impractical due to large proofs	Ideal for stateless clients
Bandwidth requirements	High	Low
Cryptographic primitives	Traditional Merkle proofs	Polynomial commitments

Figure 18: MPTs vs. Verkle trees in the Ethereum context.

to disk and by meticulously handling the free list during both the creation and expiration of revisions. Firewood follows Avalanche's MerkeDB `https://github.com/ava-labs/avalanchego/blob/master/x/merkledb`. However, we are not aware of any public benchmarks related to Firewood.

6.3.1 The Role of Authenticated Data Structures

Authenticated data structures, such as Merkle trees and their variants, serve as the backbone of many blockchain systems. They allow users to verify the integrity of stored data in a decentralized manner. Below we summarize some recent advances.

Zhang *et al.* [147] discuss the Quick Merkle Database (QMDB) introduces a novel SSD-optimized Authenticated Data Structure (ADS) that significantly improves blockchain state management performance. Traditional Merkle Patricia Trie (MPT)-based databases suffer from high write amplification and excessive SSD I/O operations, making them inefficient for large-scale blockchain applications. QMDB overcomes these issues by unifying world state and Merkle tree storage, leveraging in-memory Merkleization, and reducing storage overhead via append-only twigs, which group updates into small, immutable subtrees. These innovations allow QMDB to reduce SSD reads and writes for Merkleization to zero, significantly improving blockchain execution efficiency while maintaining a small DRAM footprint, making it practical for both consumer-grade and enterprise hardware. The experimental results show that QMDB achieves a 6$\times$ higher throughput than RocksDB and a 8$\times$ better performance than NOMT, a leading verifiable database. It supports scaling up to 15 billion state entries, demonstrating its capability to handle workloads 10$\times$ larger than Ethereum's current state size. Furthermore, QMDB's efficient design allows it to run efficiently on both high-performance servers and low-cost consumer hardware, lowering barriers to blockchain participation. With its superior storage efficiency, faster state updates, and support for historical state proofs, QMDB presents a transformative improvement for blockchain state databases, enabling more scalable and decentralized blockchain applications. In addition, Zhang highlights the potential of using trusted execution environments (TEEs) to protect the integrity of data stored in authenticated databases. Although current TEEs can protect the integrity of the CPU and DRAM, they do not isolate persistent storage resources. QMDB addresses this gap by employing AES-GCM encryption for persistently stored data, thus enhancing security against potential copy attacks.

One of the major challenges in authenticated databases is the performance bottleneck associated with storage access. Li *et al.* propose the multi-Layer Versioned Multipoint Trie (LVMT), which significantly reduces IO amplifications during data access, thus improving the efficiency of authenticated storage in blockchain systems [148]. Traditional blockchain storage structures, such as Merkle Patricia Trie, suffer from high I/O amplification, where each update requires modifying multiple nodes, leading to $O(log(n))$ disk I/O operations. LVMT (Layered Versioned Multipoint Trie) addresses this inefficiency by integrating an Authenticated Multipoint Evaluation Tree (AMT) and append-only Merkle trees, reducing the computational overhead of updating commitments. By storing version numbers instead of value hashes and employing proof sharding, LVMT minimizes expensive elliptic-curve operations and distributes proof generation across multiple nodes, improving scalability and efficiency. The experimental results show that LVMT outperforms MPT, achieving up to 6× faster reads and writes up to 6× faster and improving the end-to-end blockchain throughput by up to 2.7×. The system significantly reduces read and write amplification, making storage operations more efficient while maintaining security and verifiability. By eliminating excessive disk I/O and leveraging cryptographic optimizations, LVMT enables high-performance blockchain execution, making it a promising solution for future blockchain architectures requiring scalable, authenticated storage.

Choi *et al.* introduce the LMPT, a groundbreaking authenticated data structure aimed at addressing storage limitations in high-performance blockchains [149]. Through optimizing storage management, LMPT boosts data integrity and security while enhancing overall system efficiency. By maintaining smaller intermediary tries in memory, LMPT reduces the read and write amplification issues associated with high-latency disk storage. Furthermore, this design facilitates the parallel and independent scheduling of I/O and transaction verifier threads. LMPTs ultimately diminish substantial I/O traffic on the critical transaction processing path. Experimental findings demonstrate that LMPTs can handle up to 6× more transactions per second with real-world workloads compared to current Ethereum clients. These advancements are crucial for the growth of authenticated databases as they directly tackle the shortcomings of current frameworks.

Verifiable ledger databases advance the idea of authenticated databases by guaranteeing that logs are maintained in a manner that they can only be appended to and any alterations are detectable. In the work of Yue *et al.*, GlassDB is presented, which uses a POS-tree—a variation of Merkle trees—to deliver effective proofs of data integrity [150]. This framework enables users to confirm that the logs on untrusted servers are indeed append-only, thereby improving confidence in the data management process. Additionally, commercial offerings such as Amazon's QLDB and Azure's SQLLedger employ similar methodologies, demonstrating the real-world applications of these technologies. Experiment results show that ChainKV surpasses previous Ethereum systems, exceeding them by up to 1.99× for synchronization and 4.20× for query operations.

The management of historical data is another critical aspect of authenticated databases in blockchain contexts. Liang *et al.* present MoltDB, which focuses on accelerating blockchain performance through the segregation of ancient states [151]. By optimizing how historical data are stored and accessed, this approach enhances the scalability and responsiveness of authenticated databases, ensuring that data remain secure and verifiable over time. MoltDB delivers a transaction throughput that is 1.3× greater than that of LevelDB, highlighting its efficiency in executing blockchain transactions. This improvement in throughput is achieved by separating ancient states from current states, thus minimizing unnecessary disk I/O. MoltDB decreases disk I/O latency by 30%, primarily by isolating ancient states from frequently accessed data. The read latency is reduced by $40 - 60\%$ in comparison to LevelDB and RocksDB, due to this effective state separation. Although the write latency is slightly highter ($5 - 46\%$) compared to LevelDB due to the additional extraction step, it remains lower than that of RocksDB.

The reliance on specialized hardware has become a promising strategy to boost the performance of authenticated databases. Deng *et al.* investigate how GPU parallelism can speed up Merkle Patricia Tries, which greatly increases the throughput of these data structures [57]. This technique not only enhances the efficiency of lookups and insertions but also tackles the scalability challenges of conventional blockchain systems. The authors pinpoint key obstacles in GPU acceleration, such as node-splitting conflicts and hash computation conflicts arising from simultaneous insertions. To overcome these, they propose PhaseNU, a lock-free node update algorithm, and LockNU, a GPU-based optimistic lock coupling solution, enabling users to select the most suitable option based on workload characteristics. In addition, PhaseHC optimizes the parallel hash computation to tackle dependency issues, enhancing performance. The authors incorporate their GPU-accelerated MPT into real-world applications, including the Ethereum client Geth and LedgerDB, demonstrating considerable performance improvements. Testing on various data sets reveals that their method can achieve up to a 29.69× speedup for insert operations and a 3.4× increase in the processing speed of blockchain transactions in Geth.

6.3.2 Authenticated Databases: Eclipse Security and Light Clients

Traditional blockchains use an authenticated database, i.e. a DB that is able to compute a cryptographic commitment to its entire state for every block. This so-called *state commitment* is frequently used to facilitate consensus, as it makes comparing the current state of two nodes computationally trivial. As described in Section 6.3, different blockchains resort to different, frequently ad hoc authenticated DB implementations, often based on a Sparse Merkle Tree. Merklization is estimated to account for approximately 50% of the total execution time to produce a block [144]. The complexity of insertions/updates in Merkle trees is typically $O(\log N_{updates})$. The main overhead is actually that the constant factor is $\log N_{accounts}$, which means that the cost of insertion/update increases logarithmically with the number of accounts in the chain.

To avoid this performance hurdle, Solana only computes a Merkle tree of all *updated accounts* instead. The root of this tree is called the *accounts delta hash*. Effectively, the account delta hash facilitates consensus, playing a role similar to that of full-state commitments. This is similar to *state diffs* used in *validity rollups*, such as ZKSync and Starknet. In addition to the delta hash, all Solana validators compute a full-state commitment every few hours, at epoch boundaries. However, this approach does not quite work in a Layer-2 context, as full-state commitments are required for fraud-proof systems to protect against malicious challengers [1], [152]. For hybridized ZK fraud proofs [153], full state commitments are required to keep the amount of blocks it needs to re-execute at a reasonable level. Without full-state commitments, challengers would have to re-execute thousands of blocks to prove the validity of their claims.

Light clients serve as a crucial infrastructure element within blockchain systems. They enable users to verify transactions on their own without the need to download and process entire blocks, avoiding complete reliance on their data providers, while using just a small portion of the hardware needed for full nodes. Essentially, they are the middle ground between the ease of using remote clients and the trustlessness and security offered by full clients, without significantly sacrificing either aspect. Although light clients have various applications, such as creating trustless cross-chain bridges, they are most commonly discussed in relation to wallets. On a fundamental level, wallets are required to (i) track the chain's latest state, (ii) verify account balances and nonces, (iii) access contract data like token balances, (iv) calculate transaction fees, (v) initiate transactions, and (vi) oversee pending transactions. For the majority of users, operating a full node solely for using a wallet is obviously impractical. Acquiring the specified data from an external RPC provider through a remote client offers great convenience but requires significant trust, thereby weakening security. As a result, light clients provide a trustless, secure, and lightweight option compatible with mobile devices, hardware wallets, or browsers.

Chatzigiannis *et al.* [154] conduct a survey focusing on blockchain light clients, crucial to enable devices with limited resources, such as mobile phones or web browsers, to connect to blockchain networks without the need to download and authenticate the entire blockchain history. Their paper explores a variety of current light client implementations and strategies, organizing them based on their functionality, efficiency, and security attributes. Addresses challenges such as long-range attacks, validator reconfigurations, and trade-offs found between decentralization, security, and efficiency. The review offers a detailed analysis and comparison of different light client architectures, including SPV (Simplified Payment Verification), FlyClient, NiPoPoW (Non-Interactive Proofs of Proof-of-Work), as well as SNARK-based models like Mina and Plumo. Furthermore, it covers cryptographic methods used in light clients, such as Merkle Trees, Accumulators, and Zero-Knowledge Proofs (ZKPs). The authors also identify several open research issues, such as enhancing light client security in Proof-of-Stake systems, developing privacy-focused light clients, and creating smart contract-based cross-chain light clients. This paper is a valuable resource for researchers and practitioners interested in advancing light clients.

Nonetheless, the absence of state commitments currently precludes the use of light clients on Solana. Some parties have developed what we refer to as "lighter" clients such as TinyDancer [155], which are fundamentally full clients with some features stripped to reduce resource usage. These are much less efficient than true light nodes. Full-state commitments enable capabilities that Solana's existing framework cannot provide. A key functionality is the operation of light clients: blockchain clients that differ from full nodes by not storing the entire blockchain state. Rather, they access necessary state data on an as-needed basis by communicating with other clients and verify this data using state proofs, akin to inclusion proofs for the fetched state within the full-state commitment. Light clients only need to periodically obtain block headers, thereby being resource-efficient. This allows light clients to run seamlessly on platforms such as web browsers or mobile devices, improving accessibility. Therefore, our long-term goal is to enable trustless operation of light clients for the SVM stack provided by Eclipse. Ultimately, providing full-state merklization or regular partial commits is important for Eclipse and aligns with broader decentralization objectives.

7 Application Domains

In this section, we explore some *unlocks* in different application domains that deliberately look beyond transfers and DeFi. Some application classes are currently limited by the possibilities offered by the blockchain capacity —in terms of throughput, high fees, and latency — that is available today. We believe that the applications in this section — AI, gaming, and DePIN, will drive the continuous demand for performance improvements, much beyond what is possible today. Expanding beyond current performance constraints will allow us to engage in fruitful application-platform co-design as well, where specialized compute capacity would be provided for prominent classes of applications with the help of special computational *co-processors*, along with its own approach to metering, as we suggest in Section 1.2.

7.1 AI and ML

Large Language Models (LLMs) are the latest wave of applications that have been used to generate text, images, code, and other forms of content. While they are taking the computing industry by storm, LLMs are computationally expensive. Despite significant progress in making training more affordable, training LLMs still often requires hundreds or thousands of high-end GPUs running for extended periods of time. Inference is relatively affordable on a per-invocation basis, but still requires a significant number of high-end GPUs or FPGA units.

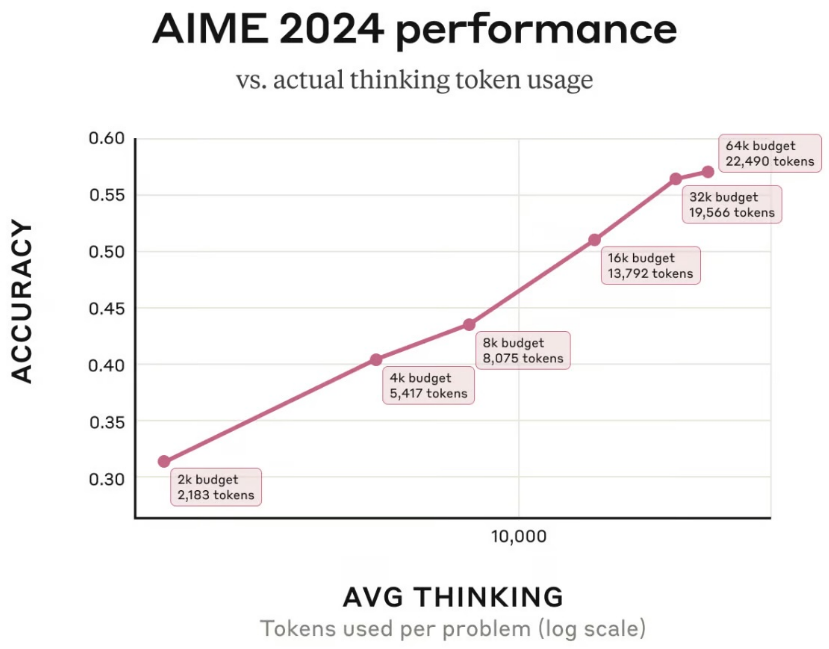


Figure 19: AIME 2024 (High school math competition) accuracy as a function of the number of tokens, provided by Anthropic.

Currently, such massive resource demands are typically beyond the reach of current blockchains; Eclipse's GigaCompute vision aims to change that.

Much progress has been made in terms of hardware-software co-design, something that is inspiring for us at Eclipse. There are some smaller models that require less resources. Some lower-scale models can run purely on CPUs, providing acceptable performance for some domains without requiring GPUs. Blockchain startups such as GensynAI, Prime Intellect, and Nous Psyche aim to democratize the training and inference of these models by using blockchain technology. Other startups like Grass produce datasets for more ethical and data-conscious training. Over time, we feel there is value in democratizing the processes of training, post-training, and inference by allowing all of them to be executed on-chain.

7.1.1 Performance metrics for LLMs

LLMs heavily rely on *tokens*; it is a key metric for gauging their performance, significantly influencing the determination of pricing structures. In English, a word typically equates to roughly 1.5 tokens, meaning 100 words convert to about 150 tokens. Although various tokenizers may break text into tokens differently, this fundamental idea remains. The size of the context windows

LLM Model	Number of Tokens	Input Token Price (per 1K tokens)	Output Token Price (per 1K tokens)
GPT-3	175 billion	\$0.002 - \$0.06	\$0.06 - \$0.12
Grok 3	700 billion	\$0.006 - \$0.15	\$0.15 - \$0.30 (est.)
DeepSeek Coder	1.5 trillion	\$0.008 - \$0.20	\$0.20 - \$0.40 (est.)
Gemini	1.5 trillion	\$0.010 - \$0.25	\$0.25 - \$0.50 (est.)
Claude 3 Sonnet	1.5 – 2 trillion (est.)	\$0.010 - \$0.25	\$0.25 - \$0.50 (est.)
ChatGPT 4	1.8 trillion	\$0.012 - \$0.30	\$0.30 - \$0.60 (est.)
WuDao 2.0	1.75 trillion	N/A	N/A
Claude 3.5 Haiku	2 trillion (estimated)	\$0.012 - \$0.30	\$0.30 - \$0.60 (est.)

Figure 20: Comparison of token usage and pricing length across different recent LLM models, sorted in ascending order of the number of tokens. The data for WuDao 2.0 is not publicly available, and there are no reliable estimates.

has expanded over time. Initially, LLMs could process context windows of around 2,000 tokens, but as shown in Figure 20, some modern models can manage trillions of tokens. This may seem excessive initially, as lengthy texts or conversations are rare. However, context windows include more than just dialogues; they also cover system prompts, supplementary documents, source code, and additional materials. Large documents, such as PDFs or code snippets, can quickly fill a context window. Although extended context windows offer advantages, they also have drawbacks. The primary challenge is computational demand: processing requirements grow quadratically with input length. Doubling the token count requires four times the computing power due to the need to assess the relationship between every token in the sequence. Figure 19 shows the performance of various models on AIME high school benchmarks[4]; these tokens effectively form a budget that is used during the inference stage.

7.1.2 Agentic AI

One of the most exciting applications of LLMs is *agentic* AI [156]–[159]: artificial intelligence systems that can independently make decisions and take actions to achieve specified goals, adapting as circumstances change. These systems often operate with minimal human oversight and can iteratively improve their own processes or strategies as they continuously learn from new data. Open-source implementations such as Open Hands [160] offer significant improvements to workflows.

The scope of potential applications of agentic models is vast and diverse. In the domain of software development, they can significantly streamline the process by automating tasks such as code review and debugging. For example, an agent could be instructed to write a program based on a given problem

[4] https://www.datacamp.com/blog/claude-3-7-sonnet, https://www.anthropic.com/news/claude-3-7-sonnet

description, then execute the program to check its output against expected results. If errors are detected, the agent would modify the code accordingly and repeat this process until the desired outcome is achieved.

These systems also present opportunities in data analysis and management. Agents can be configured to interpret data sets, perform statistical analysis, and draw insights that humans may miss due to their subjectivity or lack of experience with specific tools and techniques. In addition, agentic models offer potential advantages for personalized user experiences by adapting their behavior based on context and individual preferences. However, while the benefits of agentic AI are evident, these systems are not without challenges. One significant obstacle is ensuring reliability and accuracy in multi-step operations. The possibility of errors or mistakes compounding over time creates a need for robust error handling and correction mechanisms.

Blockchain provides inherent advantages that create reliable high-integrity frameworks capable of executing on-chain transactions. This is particularly beneficial in the realm of DeFi trading bots, which leverage a combination of on-chain data and sentiment analysis from platforms like Twitter/X. Many contend that integrating AI with the capability for crypto transactions will pave the way for a realm of autonomous agents grounded in financial frameworks. The capacity of AI agents to decode and react to intricate market indicators is actualized, allowing the rapid execution of decisions. These agents can calculate the most advantageous course by weighing factors such as anticipated profits, associated risks, and execution expenses. They also employ self-adaptive algorithms to adjust strategies as market dynamics changes. During decision-making, certain challenges may be too complex for a single agent to address effectively. As a result, numerous agents function within multi-agent systems (MAS), coordinating efforts across various DeFi protocols to enhance resource distribution, such as balancing liquidity across different pools. The capacity to parse large streams of data in real time fits perfectly with Eclipse's HPC-based design, enabling immediate on-chain trade execution by agentic AIs.

One of the most prominent recent agentic systems is ai16z `https://github.com/ai16z`. The protocol acts as a "Virtual Marketplace of Trust", leveraging AI agents to collect market data, assess community opinions, and conduct token trades both on-chain and off-chain. By incorporating feedback from members' investment expertise and rewarding valuable contributions, ai16z has developed a highly efficient investment fund (currently targeting memecoins) with robust decentralization features. Developers can construct agents using ai16z's Eliza Framework, which offers comprehensive tools and libraries for agent creation, testing, and deployment. Agents may be hosted either locally on a server or in Agentverse, ai16z's central hub for agents. For agents to communicate, they must register via Almanac and can utilize Mailbox to interact, even when hosted locally. On Eclipse, ai16z-style marketplaces can handle larger volumes of data and more frequent agent interactions, scaling the ecosystem to larger markets and more concurrent users.

7.1.3 GSVM for Trustless AI

The computational demands to run even basic AI models render the current blockchain framework ineffective. Systems such as agents depend on off-chain resources, utilizing either DePIN or centralized platforms like OpenAI to operate LLMs. Verifiable methods, such as Zero-Knowledge Machine Learning (ZKML), do exist, yet they cannot accommodate the complexity of modern LLMs at scale. Training models directly on a blockchain, such as Ethereum's mainnet, is evidently not feasible at present. Yet Eclipse's GigaCompute approach seeks to combine faster node hardware (GPUs, FPGAs) with on-chain proofs, bridging the gap between off-chain and trustless inference.

A high-throughput blockchain with robust compute enables AI models to run directly on-chain. Users would no longer need to rely solely on black-box output from off-chain servers. Instead, the operations of the neural net and its results are fully auditable. That level of trustless inference fosters applications like decentralized recommendation systems, automated scoring, or credit checks where participants can verify AI's fairness and correctness.

Eclipse proposes not only increases in compute that make this possible, but also the addition of dedicated hardware resources to further accelerate on-chain compute. One can imagine GPU acceleration for on-chain agents for training, post-training, and inference, ultimately enabling inference and fine-tuning of models on-chain.

7.2 Gaming

There are multiple uses for blockchains in gaming. The most widespread application is the creation of irrefutable non-fungible records on-chain for tracking in-game purchases and achievements. However, not much computational power nor space efficiency is ultimately required for this purpose. The only metric that has any bearing is transaction processing latency and time to finality; as such, these applications are more abundant in L2s such as Arbitrum and Eclipse, and L1 blockchains such as Solana, which produce more than one block per second. ombined with an RPC latency of as low as 50 milliseconds [161], the latency is sufficiently low for operations such as purchasing in-game items, recording global rankings, as well as achievements. Consequently, reducing the time needed to produce a block can benefit these applications.

However, there are a few instances [162]–[165] of entire games running as executable code on the chain, requiring no additional resources other than the blockchain itself. For the moment, these attempts are merely proofs of concept: the determinism of the game engine, coupled with the game's overall simplicity, make it suitable for on-chain execution. Still, we expect that it should be possible to run more sophisticated game engines on-chain, provided there is enough computational headroom, to push beyond turn-based low-compute scenarios. By scaling transaction throughput, more intricate game logic can reside fully on-chain.

There are in-between approaches as well, and entire SDKs dedicated to leveraging blockchains for gaming, but not using them for execution of game logic. For instance, Multisynq [166] adopts a model with bit-identical replicated state updates for the majority of the game logic. The problem of non-determinism in user input (as well as the security considerations about censorship) is handled by a Proof-of-History like synchronization (consensus) procedure. The entirety of this protocol can be ran as on-chain logic, the limiting factor being the limited computation per-block. As such, even a modest improvement in computational productivity per block (and per account) can unlock simpler, fully-on-chain implementations that do not need specialized clients, such as the Multisynq synchonizer.

7.2.1 MMORPGs and Blockchains

Hosting game-play on-chain for Massively Multiplayer Online Games (MMORPGs) offers several key advantages, particularly in terms of transparency, ownership, interoperability, and trustlessness. Here are the main benefits:

- Blockchain-based MMORPGs allow players to truly own their in-game assets (e.g., characters, items, skins) as non-fungible tokens (NFTs). These assets are stored on the blockchain, meaning that players have full control over them and can trade, sell, or use them across different games or platforms without relying on centralized game developers. This contrasts with traditional MMORPGs, where in-game assets are controlled by the game developer and can be lost if the game shuts down or the player is banned.

- On-chain game-play ensures that all game mechanics, rules, and transactions are transparent and verifiable. Players can audit the game's logic and ensure that no unfair advantages are given to certain players or manipulated by the developers. Smart contracts enforce the rules of the game, eliminating the possibility of cheating or manipulation by centralized entities.

- Blockchain-based MMORPGs enable interoperability, allowing players to use their assets across multiple games or platforms. For example, a sword earned in one game could be used in another game, provided both games support the same blockchain standards (e.g., ERC-721 or similar Solana standards [167] for NFTs). This creates a more connected gaming ecosystem and increases the utility and value of in-game assets.

- By hosting game-play on-chain, MMORPGs eliminate the need for centralized servers, reducing the risk of downtime, censorship, or data breaches. The state of the game is maintained by the blockchain, ensuring that it is always accessible and secure. Players do not need to trust the game developer to maintain the game or protect their assets, as the blockchain ensures trustless and immutable operations. On-chain games may leverage

Network Performance Metrics	
Latency (Ping Time)	Round-trip time (RTT) for data packets. Lower latency ensures smooth game-play.
Packet Loss	Percentage of lost data packets. High packet loss causes lag and desync.
Jitter	Variation in latency over time. High jitter causes inconsistent response times.
Bandwidth Usage	Amount of data transmitted between players and the server.
Synchronization Rate	Frequency of game world updates between clients and servers.
Server Performance Metrics	
Concurrent Player Count	Maximum number of players online simultaneously.
Server Tick Rate	Rate (in Hz) at which the server updates the game state.
Database Query Time	Time taken for retrieving/updating player data.
Server Uptime & Downtime	Percentage of time servers remain online and frequency of maintenance downtime.
Load Balancing Efficiency	How effectively server load is distributed across clusters or regions.
Resource Utilization	CPU, GPU, and RAM usage of game servers.
Number of Servers	Affected by expected number of concurrent playersa as well as game world size and complexity
Game-play & World Simulation Metrics	
World State Consistency	Ensures all players experience the same persistent game world state.
AI & NPC Response Time	Delay between AI/NPC receiving input and executing an action.
Dynamic Event Latency	Time taken for in-game events to be reflected across clients.
Physics Engine Performance	Measures lag or desync in physics-based interactions.
Content Generation Speed	Speed of dynamically generating the game world.

Figure 21: Gaming performance metrics.

cryptographic proofs (e.g., zero-knowledge proofs) to ensure that game outcomes are fair and verifiable. For example, the randomness of loot drops or battle outcomes can be proven to be fair and not manipulated by the developer or other players.

- Traditional MMORPGs are vulnerable to shutdowns if the developer decides to discontinue the game. In contrast, on-chain games are more resilient because the game's logic and assets are stored on the blockchain, which is decentralized and immutable. Even if the original developers abandon the game, the community can continue to maintain and evolve it. This effectively provides a specific form of censorship resistance.

7.2.2 Attacks

Cheating has been prevalent in multiplayer online games for a long time [168]. Various strategies fall into distinct categories or dimensions. One such category is *interrupting information dissemination*, which involves altering the update frequency or providing erroneous information to confuse other players about the current state of the game. This can give a player's character an unwarranted advantage during attacks. Peer-to-peer (P2P) nodes are particularly susceptible to this kind of cheating due to their role in sharing both their updates and those of others. Essentially, these nodes within the forwarding pool can disrupt the information flow with ease. Several forms of cheating fall into this category, including *escaping*, *time cheating*, *network flooding*, *fast rate cheating*, *suppressing correct chat*, *replay cheating*, and *blind opponent*. *Time cheating* involves using outdated timestamps, where a player, after receiving updates, submits their update with a backdated timestamp to avoid being caught. In contrast, *replay cheating* occurs when a cheater retransmits previously received signed and encrypted updates from another player. Furthermore, a cheater can obscure their actions from opponents by withholding some updates, a tactic known as *blind opponent*.

The second type of attacks includes illegal in-game actions that involve cheating by circumventing the game's inherent rules and altering its state via code modification. This category encompasses various cheating methods like *client-side code tampering*, *aimbots*, *spoofing*, and *consistency* cheats. An aimbot, in particular, is a program used in multiplayer first-person shooter (FPS) games to assist players in aiming at opponents. This tool enables automatic weapon targeting through smart software. On the other hand, a cheater may transmit inconsistent updates to different players, and when this strategy is used by either an individual or a team, it is termed a *consistency* cheat.

Although various detection and prevention methods are available, we emphasize the importance of commitments. A commitment scheme guarantees that participants will remain consistent in their behavior (actions). For example, time-related cheating can be mitigated by employing the lockstep protocol. This protocol requires all participants to initially submit a hash code of their upcoming actions; only once all hash codes are received do players exchange their actions. By verifying the hash code against the actual action, players can ensure that no actions were altered after receiving others' input. Although not all security threats in gaming, like botting and automation, can be resolved with this approach, numerous issues could be mitigated through a globally shared game state maintained on-chain.

7.2.3 Metrics

Game servers for MMORPGs are configured based on the specific requirements of the game, the budget, and the architectural choices made by the developers. Despite these variations, there is a noticeable shift towards employing more robust and specialized hardware. This includes using advanced technologies such as CPUs with numerous cores, substantial memory, rapid storage solutions, high-bandwidth networking components, and hardware acceleration to provide a rich, interactive gaming experience to a worldwide audience. When it is cost-effective, these servers often utilize high-performing hardware like DDR5 RAM, which offers a substantial boost in bandwidth and capacity over prior models, allowing for rapid storage and retrieval of extensive game data. Similarly, NVMe drives afford ultra-fast storage access, which minimizes loading times and enhances game interactivity. Even though GPUs are typically linked with client-side graphics rendering, they also have server-side applications, such as aiding in physics simulations, AI processing, or rendering game worlds in cloud gaming environments. FPGAs are another option, offering hardware acceleration for specific operations, which can enhance the performance of game mechanics or network functions. They may also be custom-configured to speed up particular game mechanics or AI processes.

The performance of MMORPGs is significantly influenced by network reliability, server capacity, the effectiveness of world simulation, and various metrics of player engagement. Enhancing these elements leads to more seamless gameplay, improved player retention, and lower operational expenses. Figure 21 offers a comprehensive overview of the associated MMORPG metrics.

7.2.4 GSVM for Gaming: From Turn-Based to Real-Time

Today, most games largely limit themselves to NFTs for in-game assets and achievements. Some games use on-chain statistics, while only some experimental games put their entire state on-chain [162]–[165]. One could say that the best you can do today for on-chain gaming is turn-based games, i.e. games that can afford relatively high interaction times.

Low latency means that players can now interact in real-time with the same virtual world. This paves the way for truly massive, persistent games powered by decentralized technology. With an increase in the compute capacity of GSVM, developers will be able to put most of the game logic on the chain, without compromising the important metrics listed in Figure 21. On-chain implementation enables transparency in game-play and provides new ways to combat cheating, a common problem in online competitive gaming [168], as outlined in Section 7.2.2. Finally, putting the entire logic on-chain benefits both developers and users: developers will need to maintain less back-end infrastructure, as they can get rid of all/most of their off-chain components. They can focus their full attention on the user experience and user interface of their games. Users no longer depend on the game developer to keep the game running and maintain ownership over both the game and their assets in it. We can also foresee convergence between the gaming tokens and DeFi activities, with the possibility of using gaming tokens as collateral, etc.

7.3 DePIN

Decentralized physical infrastructure networks (DePIN) are a popular application of blockchains. Typically, DePIN systems follow the template of connecting providers of a service to users, allowing one to exchange those in a completely automated fashion; we mention some of the most prominent DePIN examples.

- Filecoin serves as a decentralized storage network, linking individuals with unused hard drive space to those seeking cloud-like storage solutions.

Filecoin potentially offers more affordable rates than conventional storage options. For instance, a standard S3 storage bucket, usually priced around 10,000 USD, would cost roughly 230 USD monthly on Filecoin.

- Livepeer is a network that enables people with disposable GPU compute and demand for a specific workflow to provide service and benefit each other. The network is operated as a side-chain, leveraging the Arbitrum blockchain for financial operations. The client is written in Go, and requires access to an nVidia GPU.
- DIMO, built on the Polygon L2, for vehicle owners to gather data from their vehicles and monetize it through various applications and services.

The list of DePINs goes on; we have merely listed the most popular and most widely known projects.

7.3.1 Case Study: Helium

Helium [169] is a network that utilizes the blockchain to distribute rewards and incentivize participants. It allows participants with unused wireless network capacity to provide their wireless connection to customers in exchange for token compensation. Helium originally operated their token on Ethereum. However, following HIP-0070 [170], the core developers have chosen to move to Solana, citing the following advantages:

> There are a variety of advantages of integrating with the Solana ecosystem including fast and cheap transactions and native governance primitives, but it also brings a whole host of new developers into our ecosystem. The ecosystem's development of the new Solana Mobile Stack (and the Solana Saga phone) brings a closer connection to the Helium ecosystem which can enable connectivity and access for these devices.

Although one can fully expect Solana to have satisfied Helium's needs, improving the performance of the layer-1 validator software can confer further benefits to Helium, as well as similar DePIN projects. Helium uses Solana exclusively for its tokenomic operations. This means that there is a direct correlation between the costs of operating Helium and the efficiency of the network. Additionally, having a bigger computation budget would allow Helium to perform more and more of its operations on-chain. For example, proof-of-coverage for WiFi coverage is currently verified by specialized oracles. A bigger computational budget would allow this work to be done more straightforwardly on-chain, removing the need for oracles.

7.3.2 Case Study: Render

Render is another notable DePIN project that moved from Ethereum to Solana. Although the sentiment is broadly similar, the RNP-002 [171] highlighted other considerations that were important to Render:

- *Language support.* Most of Render's code requires high-throughput, high-performance GPU compute. This can only be done in languages with support for such APIs, and for Render support of C++ and Rust was particularly important.
- *Transaction costs.* They point out that transaction costs for a 200 frame rendering contract would require at least 200 transactions and with the gas costs at the time of writing, would amount to 5 USD, which they deem too high.
- *NFT compression.* Software optimization particularly regarding the NFT compression schemes developed by Metaplex [172], allows better than linear reduction in the total cost of operating the network.
- *Throughput.* They specifically mention that the number of transactions that can be handled per second in the case of Solana is much higher than the other blockchains on the list.

As one can see, taking the Solana validator code and improving its performance characteristics directly addresses many of the concerns of the Render network. Higher CU limits and more parallelism would directly drive up throughput, reduce the scarcity of blockspace, and thus drive down the costs of operating an NFT-driven system.

In this context, we should note some second-order effects. As we already mentioned, more performance would eventually permit verifying zero-knowledge proofs on-chain. This can significantly reduce the number of NFTs that need to be created to verify that the rendering has been performed. Over time, advanced HPC could allow more on-chain ZK proofs for rendered images, avoiding large NFT sets while preserving trust in the outcome.

It is worth noting a few commonalities with the previously discussed projects. Firstly, they all require a high performance network to support them. It is either a purpose-built blockchain network, like in Filecoin's case, an EVM L2, like Polygon, or a high-performance, high-throughput L1 like Solana in Helium and Render's case.

An essential point to note is that DePIN projects are not entirely tied to a blockchain. The majority of DePINs incorporate off-chain elements necessary for tasks that would be prohibitively costly to execute on-chain, or demand a particular native API unavailable to the blockchain's virtual machine. Nevertheless, by exposing the necessary APIs, ensuring verifiable computation, and increasing on-chain execution independent of oracles, the development process becomes more streamlined.

7.3.3 GSVM for DePIN: Scaling Physical Infrastructure with Real-Time Settlement

The decreased complexity is expected to have a beneficial impact on DePIN ecosystems in general. By delegating a significant portion of the workload to the validator, initial expenses decrease, reducing the starting costs, which in turn encourages greater competition across the projects.

The three main unlocks of GigaCompute for DePIN are the increase in compute capacity, extremely low transaction costs, and a reduction in latency. The combination of these features enables us to scale DePIN to millions of devices. One can imagine DePIN networks that enable anyone to sell or rent services using their phones, computers, cars, houses, etc. Fast on-chain smart contracts guarantee both traceability and payments between end-users and service providers.

Extremely low transaction costs enable pay-per-use payment models and payment streams, avoiding off-chain, trusted accounting methods used in many projects today. By performing every action on-chain, interactions with DePIN networks can be truly trustless.

The combination of compute capacity and low transaction fees makes it possible to check the status of each connected device more frequently. Although the nature of these checks may vary between DePIN types, more frequent challenges disincentivize cheaters and improve the quality of service of all networks.

Lastly, we believe that the reduction in latency that we can achieve through the mechanism of pre-confirmations will in many cases allow DePIN applications to provide near real-time settlement.

8 Conclusions

Our measurements show that the Eclipse blockchain can exhibit about 100,000 TPS in lab conditions. However, taking the performance of Eclipse to the next level, a roughly 10× improvement in computational capacity, which we call *GigaCompute*, from where it is today requires a significant rethinking of the entire design of the Eclipse validator and, over time, key infrastructure components such as light nodes, RPC nodes, and indexers.

In this thesis, we describe some of the core design principles behind Eclipse, most notably, *Software-hardware co-design* as well as *Cross-layer optimizations* such as being hotspot-aware, the principles of *Workload non-interference*, *Dynamic scaling*, and *chain elasticity*. In Section 1.4 we outline a number of exciting options for the GSVM client that help us to bring Eclipse closer to GigaCompute. We further argue that the pessimistic concurrency approach that Solana uses can lead to performance bottlenecks that we should try to avoid (Section 2). We present a wide range of exciting opportunities in terms of using custom hardware such as SmartNICs and FPGAs that give us near line-rate processing for signature verification or deduplication of transactions, offloading these activities off the CPU (Section 3). We demonstrate how performance-based sequencing, combining advanced scheduling and performance optimizations, enables dynamic horizontal scaling (Section 4).

We show that we can optimize the runtime through aggressive, profiling-guided optimizations and training on historical workloads; we propose to augment this with aggressive pre-computation, a concept we call *computation abstraction*, leading to a self-improving runtime (Section 5). We illustrate how to do cross-layer optimization with effective state pre-fetching, resulting in virtually no misses for the storage back-end. We further demonstrate how, in order to avoid bottlenecks around state storage, hardware-accelerate key-value database solutions can be used; we also highlight some options to rapidly generate state commitments, creating significant opportunities for light clients (Section 6).

Finally, in Section 7 we discuss the following domains that have significant potential to co-evolve with Eclipse, benefiting from its GigaCompute performance:

- **AI/ML**: On-chain inference or agentic AI that uses GPU nodes, trusting results without oracles.
- **Gaming**: Real-time on-chain worlds that support thousands of concurrent players with minimal latency.
- **DePIN**: Large-scale device networks that verify coverage or resource usage, and allow payments.

We also show how the notion of computational co-processors and a custom metering approach can create a significant unlock for high-performance applications running on GSVM.

References

[1] M. Al-Bassam, A. Sonnino, V. Buterin, and I. Khoffi. "Fraud and data availability proofs: Detecting invalid blocks in light clients." (2018), [Online]. Available: https://arxiv.org/abs/1809.09044.

[2] J. Gorzny, L. Po-An, and M. Derka, "Ideal properties of rollup escape hatches," in *Proceedings of the International Workshop on Distributed Infrastructure for the Common Good*, 7, 2022.

[3] S. Chaliasos, D. Firsov, and B. Livshits. "Towards a formal foundation for blockchain rollups." (2024), [Online]. Available: http://arxiv.org/abs/2406.16219 11/01/2024.

[4] Helius, *The truth about Solana local fee markets*, 2025.

[5] B. Acilan, A. Constantinescu, L. Heimbach, and R. Wattenhofer. "Transaction fee market design for parallel execution." (17, 2025), [Online]. Available: http://arxiv.org/abs/2502.11964 03/04/2025.

[6] T. Roscoe, "It's time for operating systems to rediscover hardware," in *Symposium on Operating Systems Design and Implementation*, 2021.

[7] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai. "Fast-HotStuff: A fast and resilient HotStuff protocol." (3, 2022), [Online]. Available: http://arxiv.org/abs/2010.11454 04/07/2023.

[8] D. Malkhi and K. Nayak. "HotStuff-2: Optimal two-phase responsive BFT." (2023), [Online]. Available: https://eprint.iacr.org/2023/397.

[9] D. Malkhi and M. Yin. "Lessons from HotStuff." (2023), [Online]. Available: https://arxiv.org/abs/2305.13556.

[10] Espresso Team. "The Espresso sequencer." (2024), [Online]. Available: https://hackmd.io/@EspressoSystems/EspressoSequencer.

[11] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. "HotStuff: BFT consensus in the lens of blockchain." (23, 2019), [Online]. Available: http://arxiv.org/abs/1803.05069 04/07/2023.

[12] K. Babel, A. Chursin, G. Danezis, *et al.* "Mysticeti: Reaching the limits of latency with uncertified DAGs." (13, 2024), [Online]. Available: http://arxiv.org/abs/2310.14821 03/03/2025.

[13] G. Danezis, E. K. Kogias, A. Sonnino, and A. Spiegelman. "Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus." (2022), [Online]. Available: http://arxiv.org/abs/2105.11827 05/22/2023.

[14] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. "Bullshark: DAG BFT protocols made practical." (7, 2022), [Online]. Available: http://arxiv.org/abs/2201.05677 02/12/2024.

[15] S. Motepalli, L. Freitas, and B. Livshits. "SoK: Decentralized sequencers for rollups." (5, 2023), [Online]. Available: http://arxiv.org/abs/2310.03616 11/01/2024.

[16] K. Fitcher. "TPS is a terrible metric, stop using it to compare systems." (2022), [Online]. Available: https://kelvinfichter.com/pages/thoughts/tps-is-dumb/.

[17] Monad. “What is TPS?” (2024), [Online]. Available: https://www.monad.xyz/wtf-is-tps.

[18] Polkadot. “Transactions weights and fees.” (2024), [Online]. Available: https://docs.polkadot.com/polkadot-protocol/basics/blocks-transactions-fees/fees/.

[19] J. Zhao. “Deep dive into resource limitations in Solana development – CU edition.” (2025), [Online]. Available: https://57blocks.io/blog/deep-dive-into-resource-limitations-in-solana-development-cu-edition#cu-limitations.

[20] Solana, *How to request optimal compute budget*, https://solana.com/developers/guides/advanced/how-to-request-optimal-compute, 2024.

[21] H. Dagli, *Optimizing Solana programs. actionable insights*, https://medium.com/\spacefactor\@m{}het2341999/optimizing-solana-programs-26c7ddd0299c, 2024.

[22] *Introduction to Solana Compute Units and transaction fees*, https://www.rareskills.io/post/solana-compute-unit-price, 2024.

[23] Anza, *Comprehensive compute fees - agave validator documentation*, https://docs.anza.xyz/proposals/comprehensive-compute-fees, 2025.

[24] Luganodes. “Preconfirmations: Explained.” (2024), [Online]. Available: https://www.luganodes.com/blog/preconfirmations-explained/.

[25] S. Motepalli, L. Freitas, and B. Livshits. “SoK: Decentralized sequencers for rollups.” en. (2023), [Online]. Available: http://arxiv.org/abs/2310.03616 01/31/2025.

[26] Eclispe Labs. “A deep dive into application-specific sequencing.” (2024), [Online]. Available: https://www.eclipse.xyz/articles/a-deep-dive-into-application-specific-sequencing.

[27] X. Wei, R. Cheng, and R. Chen, “Characterizing off-path SmartNIC for accelerating distributed systems,” 2023.

[28] B. Podgorelec, M. Turkanović, and M. Šestak, “A brief review of database solutions used within blockchain platforms,” in *Blockchain and Applications*, J. Prieto, A. Pinto, A. K. Das, and S. Ferretti, Eds., vol. 1238, 2020.

[29] F. Ajayi-Peters, *Block-STM vs. Sealevel: A comparison of parallel execution engines*, https://www.eclipse.xyz/articles/block-stm-vs-sealevel-a-comparison-of-parallel-execution-engines, 2024.

[30] Helius. “$Trump’s historic weekend on Solana: Records, trends, and insights.” (2025), [Online]. Available: https://www.helius.dev/blog/trump-solana-memecoin-records-trends-insights.

[31] L. Heimbach, Q. Kniep, Y. Vonlanthen, R. Wattenhofer, and P. Züst. “Dissecting the EIP-2930 optional access lists.” (2024), [Online]. Available: http://arxiv.org/abs/2312.06574.

[32] Y. Tao, H. Li, C. Zihe, and Z. Xiaohua, “Design and implementation of an Ed25519 coprocessor,” in *International Conference on Electronics, Communications and Information Technology (CECIT)*, 2022.

[33] W.-K. Lee, R. K. Zhao, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High throughput lattice-based signatures on GPUs: Comparing Falcon and Mitaka," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, 2024.

[34] Y. Ning, J. Dong, J. Lin, *et al.*, "Grasp: Accelerating hash-based PQC performance on GPU parallel architecture,"

[35] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao. "High-throughput GPU implementation of dilithium post-quantum digital signature." (22, 2023), [Online]. Available: http://arxiv.org/abs/2211.12265 11/17/2024.

[36] Y. Sakakibara, Y. Tokusashi, S. Morishima, and H. Matsutani, "Accelerating blockchain transfer system using FPGA-based NIC," in *IEEE Intlernational Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, 2018.

[37] M. Nickel and D. Göhringer, "A survey on architectures, hardware acceleration and challenges for in-network computing," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 1, 31, 2025.

[38] Y. Du, K. Chang, J. Shi, Y. Zhou, and M. Liu, "A survey on mechanisms for fast network packet processing," in *Proceedings of the International Conference on Computing, Networks and Internet of Things*, 26, 2023.

[39] F. Wang, G. Zhao, Q. Zhang, H. Xu, W. Yue, and L. Xie, "OXDP: Offloading XDP to SmartNIC for accelerating packet processing," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2023.

[40] Syndica. "A solana validator client explained: Everything you need to know." (2024), [Online]. Available: https://blog.syndica.io/a-solana-validator-client-explained-everything-you-need-to-know/.

[41] J. Mostafa, S. Chilingaryan, and A. Kopmann, "Are kernel drivers ready for accelerated packet processing using AF_XDP?" In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 7, 2023.

[42] A. Rivitti, R. Bifulco, A. Tulumello, M. Bonola, and S. Pontarelli, "eHDL: Turning eBPF/XDP programs into hardware designs for the NIC," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 25, 2023.

[43] P. Enberg, A. Rao, and S. Tarkoma, "I/O is faster than the CPU: Let's partition resources and eliminate (most) os abstractions," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 13, 2019.

[44] S. Xi, J. Gao, M. Liu, *et al.* "Cora: Accelerating stateful network applications with SmartNICs." (29, 2024), [Online]. Available: http://arxiv.org/abs/2410.22229 01/15/2025.

[45] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-Fairness for Byzantine Consensus," in *Advances in Cryptology*, D. Micciancio and T. Ristenpart, Eds., vol. 12172, Cham: Springer International Publishing, 2020.

[46] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, 1991.

[47] K. Jeffay, D. Stanat, and C. Martel, “On non-preemptive scheduling of period and sporadic tasks,” in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, 1991.

[48] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, “U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks,” in *Euromicro Conference on Real-Time Systems*, 2012.

[49] J. M. Calandrino and J. H. Anderson, “Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study,” in *Euromicro Conference on Real-Time Systems*, 2008.

[50] Z. Li, W. Xia, M. Cui, P. Fu, G. Gou, and G. Xiong, “Mining the characteristics of the ethereum p2p network,” in *Proceedings of the ACM International Symposium on Blockchain and Secure Critical Infrastructure*, 6, 2020.

[51] K. Wang, Q. Wang, Y. Li, Z. Guan, and Z. Chen. “Pioplat: A scalable, low-cost framework for latency reduction in ethereum blockchain.” (11, 2024), [Online]. Available: http://arxiv.org/abs/2412.08367 12/15/2024.

[52] C. Zhao, Y. Zhou, S. Zhang, T. Wang, Q. Z. Sheng, and S. Guo. “Dethna: Accurate Ethereum network topology discovery with marked transactions.” (17, 2024), [Online]. Available: http://arxiv.org/abs/2402.03881 11/30/2024.

[53] S. Zheng, H. Wang, L. Wu, G. Huang, and X. Liu. “VM matters: A comparison of WASM VMs and EVMs in the performance of blockchain smart contracts.” (11, 2021), [Online]. Available: http://arxiv.org/abs/2012.01032 11/05/2024.

[54] A. Kiayias and P. Lazos, “SoK: Blockchain governance,” in *Proceedings of the ACM Conference on Advances in Financial Technologies*, 19, 2022.

[55] P. P. VMs, *A survey of parallel EVM — devteam_48518*, https://medium.com/\spacefactor\@m{}devteam_48518/a-survey-of-parallel-evm-9aed9d50aee0, 2024.

[56] H. Lin, Y. Zhou, and L. Wu, *Operation-level concurrent transaction execution for blockchains*, 2022.

[57] Y. Deng, M. Yan, and B. Tang, “Accelerating Merkle Patricia trie with GPU,” *Proceedings of the VLDB Endowment*, vol. 17, no. 8, 2024.

[58] V. Capocasale, F. Pedone, and G. Perboli, “Parallel transaction execution in blockchain and the ambiguous state representation problem,” in *European Dependable Computing Conference (EDCC)*, 8, 2024.

[59] Z. Chen, X. Qi, X. Du, Z. Zhang, and C. Jin, “Peep: A parallel execution engine for permissioned blockchain systems,” in *Database Systems for Advanced Applications*, C. S. Jensen, E.-P. Lim, D.-N. Yang, *et al.*, Eds., vol. 12683, 2021.

[60] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proceedings of the annual international symposium on Computer architecture - ISCA ’93*, 1993.

[61] N. Shavit and D. Touitou, “Software transactional memory,” in *Proceedings of the Symposium on Principles of Distributed Computing*, 1995.

[62] D. Dice, O. Shalev, and N. Shavit, “Transactional locking ii,” in *Distributed Computing*. 2006.

[63] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "OptSmart: A space efficient optimistic concurrent execution of smart contracts," *Distributed and Parallel Databases*, vol. 42, no. 2, 2022.

[64] Y. Chen, Z. Guo, R. Li, *et al.*, "Forerunner: Constraint-based speculative transaction execution for ethereum," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 2021.

[65] W. Huang and M. Paradies, "An evaluation of WebAssembly and eBPF as offloading mechanisms in the context of computational storage," 2021.

[66] W. Wang, "How far we've come – a characterization study of standalone WebAssembly runtimes," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2022.

[67] J. Dejaeghere, B. Gbadamosi, T. Pulls, and F. Rochet, "Comparing security in eBPF and WebAssembly," in *Proceedings of the Workshop on eBPF and Kernel Extensions*, 10, 2023.

[68] S. Y. Lim, X. Han, and T. Pasquier. "Unleashing unprivileged ebpf potential with dynamic sandboxing." (15, 2023), [Online]. Available: http://arxiv.org/abs/2308.01983 11/21/2024.

[69] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang. "Moat: Towards safe BPF kernel extension." (7, 2024), [Online]. Available: http://arxiv.org/abs/2301.13421 11/21/2024.

[70] S. Yuan, B. Lion, F. Besson, and J.-P. Talpin, "Making an eBPF virtual machine faster on microcontrollers: Verified optimization and proof simplification," in *Dependable Software Engineering. Theories, Tools, and Applications*, H. Hermanns, J. Sun, and L. Bu, Eds., vol. 14464, 2024.

[71] A. Raza, P. Sohal, J. Cadden, *et al.*, "Unikernels: The next stage of Linux's dominance," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 13, 2019.

[72] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn. "Bpftime: Userspace eBPF runtime for uprobe, syscall and kernel-user interactions." (8, 2023), [Online]. Available: http://arxiv.org/abs/2311.07923 11/06/2024.

[73] C. Mao, Y. Su, S. Shan, and D. Li, *eWAPA: An eBPF-based WASI performance analysis framework for WebAssembly runtimes*, en, 2024.

[74] Z. Wang, J. Wang, Z. Wang, and Y. Hu, "Characterization and implication of edge WebAssembly runtimes," en, in *IEEE International Conf on High Performance Computing & Communications; Int Conf on Data Science & Systems; Int Conf on Smart City; Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2021.

[75] C. Fallin and M. Bernstein, *Partial evaluation, whole-program compilation*, 2024.

[76] The World Wide Web Consortium, *Proposed w3c charter: WebAssembly working group (until 2017-07-12) from xueyuan jia on 2017-06-13 (public-new-workw3.org from june 2017)*, https://lists.w3.org/Archives/Public/public-new-work/2017Jun/0005.html, 2017.

[77] "WebAssembly core specification," W3C, 5, 2019.

[78] D. Thaler, *BPF instruction set architecture (ISA)*, RFC 9669, 2024.

[79] Intel, *Intel® software development emulator — intel.com*, https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html, 2024.

[80] B. L. Titzer, "A fast in-place interpreter for WebAssembly," *Proceedings of the ACM on Programming Languages*, vol. 6, OOPSLA2 31, 2022.

[81] Y. Futamura, *High.-order Symb. Comput.*, vol. 12, no. 4, 1999.

[82] Y. Izawa and H. Masuhara, "Amalgamating different JIT compilations in a meta-tracing JIT compiler framework," in *Proceedings of the ACM SIGPLAN International Symposium on Dynamic Languages*, 17, 2020.

[83] T. Lu and L. Peng, "BPU: A blockchain processing unit for accelerated smart contract execution," en, in *Design Automation Conference (DAC)*, 2020.

[84] A. Boutros, E. Nurvitadhi, and V. Betz, "Architecture and application co-design for beyond-FPGA reconfigurable acceleration devices," *IEEE Access*, vol. 10, 2022.

[85] C. Kachris, "A survey on hardware accelerators for large language models," *Applied Sciences*, vol. 15, no. 2, 9, 2025.

[86] N. Koilia and C. Kachris. "Hardware acceleration of LLMs: A comprehensive survey and comparison." (5, 2024), [Online]. Available: http://arxiv.org/abs/2409.03384 02/27/2025.

[87] Z. R. K. Rostam, S. Szénási, and G. Kertész, "Achieving peak performance for large language models: A systematic review," *IEEE Access*, vol. 12, 2024.

[88] Z.-X. Zhang, Y.-B. Wen, H.-Q. Lyu, *et al.*, "AI computing systems for large language models training," 2025.

[89] R. Li, D. Fu, C. Shi, Z. Huang, and G. Lu, "Efficient llms training and inference: An introduction," *IEEE Access*, vol. 13, 2025.

[90] V. Leis and C. Dietrich, "Cloud-native database systems and unikernels: Reimagining os abstractions for modern hardware," en, *Proceedings of the VLDB Endowment*, vol. 17, no. 8, 2024.

[91] S.-F. Ion, C. Carabas, N. Ţăpuş, and D. Ciorbă, "Enhancing blockchain performance via unikraft: A case study of implementation and scalability analysis on MultiversX," en, in *2024 23rd RoEduNet Conference: Networking in Education and Research (RoEduNet)*, Bucharest, Romania, 2024.

[92] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A Linux in unikernel clothing," en, in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion Greece, 2020.

[93] F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, and F. Risso, "SURE: Secure unikernels make serverless computing rapid and efficient," en, in *Proceedings of the ACM Symposium on Cloud Computing*, 2024.

[94] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience," 2021.

[95] K. Doekemeijer and A. Trivedi. "Key-value stores on flash storage devices: A survey." (11, 2022), [Online]. Available: http://arxiv.org/abs/2205.07975 01/16/2025.

[96] A. I. Alsalibi, S. Mittal, M. A. Al-Betar, and P. B. Sumari, "A survey of techniques for architecting SLC/MLC/TLC hybrid flash memory–based SSDs," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 13, 10, 2018.

[97] C. Lukken and A. Trivedi. "Past, present and future of computational storage: A survey." (13, 2021), [Online]. Available: http://arxiv.org/abs/2112.09691 02/24/2025.

[98] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Computing Surveys*, vol. 46, no. 3, 2014.

[99] H. Sharma and A. Sharma. "A comprehensive overview of GPU accelerated databases." (19, 2024), [Online]. Available: http://arxiv.org/abs/2406.13831 01/07/2025.

[100] K. Ma, M. Liu, T. Li, Y. Yin, and H. Chen, "A low-cost improved method of raw bit error rate estimation for nand flash memory of high storage density," *Electronics*, vol. 9, no. 11, 12, 2020.

[101] C.-Y. Liu, Y.-M. Chang, and Y.-H. Chang, "Read leveling for flash storage systems," in *Proceedings of the ACM International Systems and Storage Conference*, 26, 2015.

[102] M. Cornwell, "Anatomy of a solid-state drive," *Communications of the ACM*, vol. 55, no. 12, 2012.

[103] P. Wang, G. Sun, S. Jiang, *et al.*, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems*, 14, 2014.

[104] D. Landsman and D. Walker, "AHCI and NVMe as interfaces for SATA express devices," 2013.

[105] J. Sun, S. Li, Y. Sun, C. Sun, D. Vucinic, and J. Huang, "LeaFTL: A learning-based flash translation layer for solid-state drives," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 27, 2023.

[106] Z. Jiao, J. Bhimani, and B. S. Kim, "Wear leveling in SSDs considered harmful," in *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems*, 27, 2022.

[107] M. Bjørling, "LightNVM: The Linux open-channel SSD subsystem,"

[108] M. Bjørling, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis, "ZNS: Avoiding the block interface tax for flash-based SSDs,"

[109] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The unwritten contract of solid state drives," in *Proceedings of the European Conference on Computer Systems*, 23, 2017.

[110] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," 2018.

[111] K. Kourtis, N. Ioannou, and I. Koltsidas, "Reaping the performance of fast NVM storage with uDepot," 2019.

[112] J. Chen, L. Chen, S. Wang, *et al.*, "HotRing: A hotspot-aware in-memory key-value store," 2020.

[113] D. Comer, “Ubiquitous b-tree,” *ACM Computing Surveys*, vol. 11, no. 2, 1979.

[114] G. Graefe, “Modern b-tree techniques,” *Foundations and Trends in Databases*, vol. 3, no. 4, 2010.

[115] T.-D. Nguyen and S.-W. Lee, “Optimizing mongodb using multi-streamed SSD,” in *Proceedings of the International Conference on Emerging Databases*, W. Lee, W. Choi, S. Jung, and M. Song, Eds., vol. 461, 2018.

[116] S. Nath and A. Kansal, “Flashdb: Dynamic self-tuning database for NAND flash,” 2007.

[117] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng, “ForestDB: A fast key-value storage system for variable-length string keys,” *IEEE Transactions on Computers*, vol. 65, no. 3, 1, 2016.

[118] A. Papagiannis, “Tucana: Design and implementation of a fast and efficient scale-up key-value store,” 2016.

[119] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, “An efficient memory-mapped key-value store for flash storage,” in *Proceedings of the ACM Symposium on Cloud Computing*, 11, 2018.

[120] B. C. Kuszmaul, “A comparison of fractal trees to log-structured merge (LSM) trees,” 2014.

[121] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “WiscKey: Separating keys from values in SSD-conscious storage,” *ACM Transactions on Storage*, vol. 13, no. 1, 28, 2017.

[122] A. Conway, V. Chidambaram, M. Farach-Colton, and R. Johnson, “SplinterDB: Closing the bandwidth gap for NVMe key-value stores,”

[123] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” *Acta Informatica*, vol. 33, no. 4, 1996.

[124] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and Michael Stumm, “Optimizing space amplification in RocksDB,” 2017.

[125] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 14, 2010.

[126] K. Rothermel, “ARIES/NT: A recovery method based on write-ahead logging for nested transactions,” 1989.

[127] O. Balmau, D. Didona, R. Guerraoui, *et al.*, “Triad: Creating synergies between memory, disk and log in log structured key-value stores,”

[128] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, “Building efficient key-value stores via a lightweight compaction tree,” *ACM Transactions on Storage*, vol. 13, no. 4, 30, 2017.

[129] H. Zhou, Y. Chen, L. Cui, G. Wang, and X. Liu, “A GPU-accelerated compaction strategy for LSM-based key-value store system,” 2024.

[130] H. Yang, Z. Li, J. Wang, S. Yin, S. Wei, and L. Liu, “HeteroKV: A scalable line-rate key-value store on heterogeneous CPU-FPGA platforms,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1, 2021.

[131] B. Li, Z. Ruan, W. Xiao, *et al.*, “KV-Direct: High-performance in-memory key-value store with programmable NIC,” in *Proceedings of the Symposium on Operating Systems Principles*, 14, 2017.

[132] J. Bhimani, J. Yang, Z. Yang, *et al.*, "Enhancing SSDs with multi-stream: What? why? how?" In *IEEE International Performance Computing and Communications Conference (IPCCC)*, 2017.

[133] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," 2014.

[134] Y. Chun, K. Han, and Y. Hong, "High-performance multi-stream management for SSDs," *Electronics*, vol. 10, no. 4, 18, 2021.

[135] L. Marmol, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, FTL-aware key-value store," 2015.

[136] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for web-scale internet storage systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 24, 2014.

[137] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV performance with open-channel SSDs," *ACM Transactions on Embedded Computing Systems*, vol. 16, 5s 31, 2017.

[138] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "Silk: Preventing latency spikes in log-structured merge key-value stores,"

[139] M. Bjørling, "From open-channel SSDs to zoned namespaces,"

[140] D. R. Purandare, P. Alvaro, A. Wildani, D. D. E. Long, and E. L. Miller. "Host-guided data placement: Whose job is it anyway?" (1, 2025), [Online]. Available: http://arxiv.org/abs/2501.00977 02/24/2025.

[141] H. Sun, W. Liu, J. Huang, and W. Shi. "Co-KV: A collaborative key-value store using near-data processing to improve compaction for the LSM-tree." (11, 2018), [Online]. Available: http://arxiv.org/abs/1807.04151 01/16/2025.

[142] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: The design and implementation of a fast persistent key-value store," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 27, 2019.

[143] A. Raina, J. Lu, A. Cidon, and M. J. Freedman. "Efficient compactions between storage tiers with prismdb." (25, 2022), [Online]. Available: http://arxiv.org/abs/2008.02352 02/24/2025.

[144] "Nearly optimal state merklization." (2023), [Online]. Available: https://www.prestonevans.me/nearly-optimal-state-merklization/.

[145] "Nearly optimal state merklization (in Polkadot-SDK)." (2024), [Online]. Available: https://www.rob.tech/blog/nearly-optimal-polkadot-merklization/.

[146] P. O'Grady and R. Kuris. "Introducing firewood: A next-generation database built for high-throughput blockchains." (2023), [Online]. Available: https://www.avax.network/blog/introducing-firewood-a-next-generation-database-built-for-high-throughput-blockchains.

[147] I. Zhang, R. Zarick, D. Wong, *et al.* "QMDB: Quick Merkle database." (9, 2025), [Online]. Available: http://arxiv.org/abs/2501.05262 01/14/2025.

[148] C. Li, S. M. Beillahi, G. Yang, M. Wu, W. Xu, and F. Long, "LVMT: An efficient authenticated storage for blockchain," *ACM Transactions on Storage*, vol. 20, no. 3, 31, 2024.

[149] J. A. Choi, S. M. Beillahi, S. F. Singh, *et al.*, "LMPT: A novel authenticated data structure to eliminate storage bottlenecks for high performance blockchains," *IEEE Transactions on Network and Service Management*, vol. 21, no. 2, 2024.

[150] C. Yue, T. T. A. Dinh, Z. Xie, *et al.*, "GlassDB: An efficient verifiable ledger database system through transparency," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, 2023.

[151] J. Liang, W. Chen, Z. Hong, H. Zhu, W. Qiu, and Z. Zheng, "MoltDB: Accelerating blockchain via ancient state segregation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 12, 2024.

[152] D. Nehab and A. Teixeira. "Dave: A decentralized, secure, and lively fraud-proof algorithm." (8, 2024), [Online]. Available: http://arxiv.org/abs/2411.05463 11/20/2024.

[153] R. Khalil. "Kailua: How it works." (2024), [Online]. Available: https://risczero.com/blog/kailua-how-it-works.

[154] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "SoK: Blockchain light clients," in *Financial Cryptography and Data Security*, I. Eyal and J. Garay, Eds., vol. 13411, 2022.

[155] TinyDancer. "TinyDancer: The first light client on Solana." (2024), [Online]. Available: https://www.tinydancer.io/.

[156] F. Bousetouane, "Agentic systems: A guide to transforming industries with vertical AI agents," 2025.

[157] D. Ogbu, "Agentic AI in computer vision domain - recent advances and prospects," *International Journal of Research Publication and Reviews*, vol. 5, no. 11, 2024.

[158] D. Gosmar and D. Dahl, "Hallucination mitigation with agentic AI NLP-based frameworks," 2025.

[159] V. Tupe and S. Thube, "AI agentic workflows and enterprise apis: Adapting api architectures for the age of AI agents," 2025.

[160] X. Wang, B. Li, Y. Song, *et al.*, *OpenHands: An open platform for AI software developers as generalist agents*, 2024.

[161] S. Jenkins, *Latency testing solana RPC APIs providers*, https://jshelbyj.medium.com/latency-testing-solana-rpc-apis-providers-67efbc5d2c9f, 2022.

[162] *Shibescription 61832420 — wonky-ord.dogeord.io*, https://wonky-ord.dogeord.io/shibescription/02d0c9820dc020acd8128d80955ff5b7fbeb8c80ee229a408dce5fe6573f404bi0, 2024.

[163] *Someone just put legendary '90s video game doom on Dogecoin*, https://www.coindesk.com/markets/2024/01/24/someone-just-put-90s-darling-game-doom-on-dogecoin, 2024.

[164] S. Nagel, *Running doom on blockchain: A landmark moment for cardano and hydra*, https://iohk.io/en/blog/posts/2024/08/16/running-doom-on-blockchain-a-landmark-moment-for-cardano-and-hydra/, 2024.

[165] C. Wagner and D. Canellis, *You can now play DOOM directly on Bitcoin*, https://blockworks.co/news/play-doom-bitcoin, 2021.

[166] V. Fraudenberg, *Multisynq: The missing protocol of the internet*, https://multisynq.io/downloads/Multisynq%20Litepaper.pdf, 2024.

[167] "Current and upcoming NFT standards on Solana." (2024), [Online]. Available: https://medium.com/@kaylaychi77/current-and-upcoming-nft-standards-on-solana-7746920cc0d0.

[168] M. L. Han, B. I. Kwak, and H. K. Kim, "Cheating and detection method in massively multiplayer online role-playing game: Systematic literature review," *IEEE Access*, vol. 10, 2022.

[169] A. Haleem, A. Allen, A. Thompson, M. Nijdam, and R. Garg, *Helium: A decentralized wireless network*, http://whitepaper.helium.com, 2018.

[170] H. C. Developers, *Hip/0070-scaling-helium.md*, https://github.com/helium/HIP/blob/main/0070-scaling-helium.md, 2022.

[171] R. Shea, *Rnps/rnp-002.md*, https://github.com/rendernetwork/RNPs/blob/main/RNP-002.md, 2023.

[172] Metaplex, *Expanding digital assets with compression for NFTs*, https://www.metaplex.com/blog/articles/expanding-digital-assets-with-compression-for-nfts, 2022.