Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain

Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, and Alexandra Zapuc, *DFINITY, Zurich*

https://www.usenix.org/conference/atc23/presentation/arutyunyan

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain

Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, Alexandra Zapuc
DFINITY, Zurich

Abstract

The Internet Computer (IC) is a fast and efficient decentralized blockchain-based platform for the execution of general-purpose applications in the form of smart contracts. In other words, the IC service is the antithesis of current serverless computing. Instead of ephemeral, stateless functions operated by a single entity, the IC offers decentralized stateful serverless computation over untrusted, independent datacenters. Developers deploy stateful *canisters* that serve calls either to end-users or other canisters. The IC programming model is similar to serverless clouds, with applications written in modern languages such as Rust or Python, yet simpler: state is maintained automatically, without developer intervention.

In this paper, we identify and address significant systems challenges to enable efficient decentralized stateful serverless computation: scalability, stateful execution through orthogonal persistence, and deterministic scheduling. We describe the design of the IC and characterize its operational data gathered over the past 1.5 years, and its performance.

1 Introduction

Recently, the technological advances in blockchain [29], cryptography [27] and consensus protocols [5, 9, 24] have enabled more and more efficient execution of decentralized Web3 [56] applications and smart contracts. Platforms that service such applications are larger than ever [42], consisting of thousands of nodes, processing billions of requests, storing large quantities of data and connecting many users. Currently, the research community lacks a clear understanding of the operational data of such large-scale platforms, their challenges and performance, beyond testnet deployments with synthetic workloads and failure patterns. In this article, we introduce the Internet Computer (IC), its design, several of its systems challenges and real-world operational performance data.

The IC is a decentralized platform for the execution of general-purpose decentralized applications (dapps). Listing 1 shows an example for such a dapp. In current serverless offerings, this application would not work without an external service, as functions are stateless. Instead, the IC enables `decentralized and stateful serverless` computing. The IC protocol [53] runs on globally distributed servers in independent datacenters. It is highly scalable and efficient in executing applications. The main goals of the IC are decentralization, security and performance.

```
use ic_cdk_macros::{query, update};
use std::{cell::RefCell, collections::HashMap};

thread_local! {
    static STORE: RefCell<HashMap<String, u64>> = RefCell::default();
}
#[update]
fn insert(key: String, value: u64) {
    STORE.with(|store| store.borrow_mut().insert(key, value));
}
#[query]
fn lookup(key: String) -> u64 {
    STORE.with(|store| *store.borrow().get(&key).unwrap_or(&0))
}
```

Listing 1: Functional key-value store canister. The update call adds a key value pair; the query call gets values by keys. State is stored on the canister heap and persisted transparently.

In particular, the IC aims to enable governance and evolution to be controlled by different parties in a trustless and fault-tolerant manner instead of a central entity. The IC must also provide strong integrity and access control guarantees for the apps running on it as well as the users interacting with it in an efficient way. Overcoming these challenges requires novel blockchain technology, cryptography and consensus protocols [9, 27, 53]. Those advances need to be combined with a carefully crafted system design. In this paper, we focus on those systems-related challenges at the application execution layer and we present our solutions and operation data.

Application developers deploy dapps (equivalent to serverless function workflows) on the IC without the cumbersome process of resource management, just like in serverless environments. The dapps interact with each other and with

end-users. Each dapp is composed of *canisters*, the smallest units containing code and data, an immediate equivalent to serverless functions. Such canisters can be combined to build complex and powerful smart contracts. Figure 1 depicts our protocol stack. The IC operates as a large-scale replicated state machine. To achieve wide-range scalability, the IC nodes are partitioned (sharded) [13, 57] into *subnets*, each running its own replicated state machine.

Central to this article is the execution environment, which ensures that the actions implemented by developers are triggered efficiently and deterministically, and that consumed resources are accounted for. For simplicity and portability, applications are programmed in a high-level language such as Rust, but compiled down to *WebAssembly* [28]. Canisters run isolated from one another inside sandboxed processes that execute code running under a WebAssembly virtual machine.

We identified significant execution layer systems challenges that we addressed when designing and building the IC. First, as opposed to serverless environments [49], our applications are long-running and **(C1)** `stateful`. The IC enables this through an efficient mechanism to track modified memory during canister call execution [39]. Coupled with canister statefulness, the IC programming model is inspired by an event-driven, actor-based model, where application programmers implement functionality that responds to messages from users or other canisters. To simplify programmer experience the IC offers `orthogonal persistence` [14, 31] – the system running the canister automatically persists the canister memory state without users taking action toward this goal. Therefore, sending messages to a canister is just like invoking multiple times a serverless function, with the distinction that function state modified by earlier calls is persisted without the programmer explicitly saving data in external services.

Second, similarly to current scalability challenges [30, 50, 58] in serverless computing, our nodes need to run thousands of canisters (or functions) per node. This entails achieving intra-node **(C2)** `scalability`. This is a must to accommodate large subnets with tens of thousands of canisters.

Third, we emphasize **(C3)** `determinism`. Since the IC is a large decentralized replicated state machine, all nodes must transition to the same next state, despite potentially malicious user input, canister code and node behavior. In a first step, this requires strict message *ordering*. Moreover, determinism is necessary in *scheduling* [2, 36, 44] actions that alter the replicated state and, of course, the actual *state changes* must be performed deterministically as well.

Of utmost importance for the IC is ensuring **(C4)** `security` for application developers and end-users. More precisely, the IC design aims to minimize the trust application developers and end users must place in individual entities. Thus, the IC relies on strong integrity, availability and access control guarantees. In particular, only valid messages will be processed and the response can be verified as long as more than two thirds of the nodes are honest. Canisters cannot inspect or change the state of other canisters or other parts of the system. These guarantees are achieved by carefully crafted low-level operating systems and runtime mechanisms as well as through the use of virtualization and sandboxing techniques.

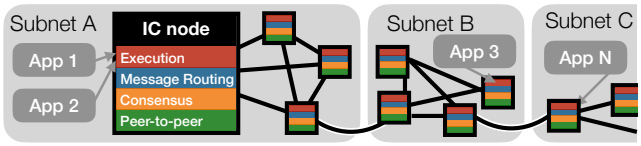


Figure 1: The protocol stack run by the nodes of the Internet Computer. The P2P layer disseminates protocol and user-generated messages. Message validation and ordering is established by the consensus layer. Messages are then routed to the execution environment and trigger efficient deterministic (replicated) computation of the apps deployed on the IC. Apps on different subnets can send messages to each other.

In this paper we show that the years-old mantra saying blockchains are slow and inefficient is coming to an end. Addressing these challenges efficiently means the IC is, to the best of our knowledge, the fastest and most efficient blockchain to date, outpacing the execution speed, transaction and execution costs [16] of other blockchains by orders of magnitude, while having significantly fewer carbon emissions [10]. More importantly, it enables decentralized stateful serverless computing. We therefore omit comparing the IC with other blockchains, but rather compare its performance with native and non-decentralized client-server architectures, whose performance we aim to achieve.

Having operated the IC for more than 1.5 years, we share our experience in designing and building the IC, with an emphasis on its systems challenges. The IC was launched in May 2021. As of January 2023, it hosts over 230,000 canisters for a total state of 2.5 TB (more than twice the size of the Ethereum blockchain) running services ranging from social media to decentralized finance. Our main contributions are:

1. We present the high-level design of the IC (Section 2).
2. We present systems challenges of the IC execution layer, with a focus on the memory subsystem, orthogonal persistence, deterministic scheduling and scalability (Section 3).
3. We showcase the performance of the IC. We introduce high-level operational data, which we open to the public. We present end-to-end application performance and study in-depth the IC performance compared to native applications. We discuss the (performance) implications of decentralization and statefulness (Section 4).

2 The Internet Computer Design

In this section we briefly introduce the IC. For a more comprehensive article explaining protocol aspects in more depth we refer the reader to the IC whitepaper [53].

Motivation. The IC aims to provide efficient multi-tenant, general-purpose, and secure computation in a decentralized

and geo-replicated manner tolerating Byzantine faults, offering developers a modern and easy to use programming model.

Overview. The nodes of the IC run a network of replicated state machines [48], which interact with each other via messages. State machine replication achieves the same output state for a service replicated on multiple machines. Each state machine generates new states by applying *deterministic* state transformations — based on the deterministic execution of the canisters' code provided by the app developers — to the previous state by processing ordered input messages from users and other canisters. The result is a new state and output messages to canisters and users.

Subnets and nodes. The nodes (term used interchangeably with replicas, machines, or servers) of the IC are partitioned into *subnets*, each subnet providing state machine replication for the set of canisters deployed on it.

Each node in a subnet runs all the canisters deployed in that subnet. Subnets can be smaller or larger: we have subnets with 80,000+ canisters and subnets with several hundreds. Most subnets have 13 nodes that are geo-replicated across the Americas, Europe and Asia. For applications in need of improved security, we have higher-replication subnets, spanning up to 40 nodes. A subnet should continue to function even if some replicas are faulty. The replicas running the IC protocol are hosted on servers in geographically distributed, independently operated data centers, bolstering security and decentralization.

Currently, the IC nodes are homogeneous. Homogeneity is important for system parts that are executing code running in the replicated state machine, as otherwise, speed may be reduced due to too many slow nodes. However, functionality outside of the replicated state machine, such as for executing non-replicated query calls can be scheduled proportionally to each machine's resource availability.

The IC supports heterogeneous subnets as long as all machines in a subnet are homogeneous. For example, certain subnets have 13 machines while others have 40, certain subnets have different disks and IO throughput and charging is based on the number of nodes in a subnet (i.e., replication factor). Overall horizontal scalability is achieved through the sharding mechanism. This effectively allows the IC to scale horizontally adding massive numbers of nodes without much additional overhead.

2.1 Failure Model

To maximize decentralization, the IC is designed for Byzantine fault-tolerance, in which faulty nodes may deviate in an arbitrary way from the IC protocol.

In any given subnet with $n \geq 3f+1$ nodes, at most f nodes may behave in a faulty manner. This is the highest number of failures which can be tolerated without additional assumptions on failures and message delivery [22, 48]. The failures account for software bugs, power outages as well as outright malicious behavior by colluding nodes. To limit the exposure and maximize decentralization, the nodes in a subnet are chosen in different geographical areas, jurisdictions and node provider organizations. In the future, trusted execution environments will further reduce the attack surface.

Traditional systems [8, 11, 38] often assume a weaker crash-stop failure model and aim to be available if a subset of nodes crash, but cannot cope with Byzantine behavior. The performance implications of Byzantine fault tolerance over crash-stop are acceptable for the applications deployed on the IC.

2.2 IC interface

The Internet Computer provides two distinct types of calls (i.e., requests sent to canisters): update and query calls. We refer to these operations interchangeably as either calls, requests, or messages. The IC also provides special calls for the canister life cycle: canister *creation*, canister *installation* and canister *upgrades*. Those are special forms of update calls.

Update calls can modify canister state. They are executed on all machines in a subnet participating in state machine replication. The calls are ordered and validated by consensus in a Byzantine fault-tolerant manner. This order, together with deterministic execution of canister code and relevant parts of the IC, provide state machine replication guarantees. Since consensus is computation and communication heavy, agreement and execution of update calls is done in batches to optimize throughput. Thus, update call latency is dependent on the time spent for consensus to reach agreement on blocks.

While update calls for different canisters may be executed in parallel, update calls for the same canister are processed sequentially. The response to an update call is threshold-signed by $2f+1$ nodes, i.e., a super-majority of the nodes created a signature collectively, hence users can verify correctness without having to communicate with multiple nodes.

The IC API guarantees atomicity for update calls as long as no further calls to other canisters are made. Updates that modify local state and run local computation are always atomic. For computation that calls into other canisters/smart contracts 2PC protocols could be implemented.

Query calls, on the other hand, do not change the canister's persisted state. As such, a query call may be processed directly by a single replica without passing through consensus. This reduces the query call latency significantly.

The correctness of query call responses from individual machines can be verified despite the Byzantine fault tolerance failure model with *certified variables*. Such variables carry threshold signatures which are generated collectively by a super-majority of the nodes in a subnet. Data elements that programmers want to verify via certified variables need to be arranged in a Merkle tree [35]. With certified variables, elements of a canister's state can be verified by clients even when talking to a single IC node.

Note that the Internet Computer does not guarantee any

order between query calls and other calls to the system (neither query nor update). If the order of calls matters, canister developers must use update calls and/or provide a versioning scheme as part of the canister code.

Applications running on different subnets can call each other by means of an asynchronous pull-based reliable communication primitive on top of consensus on both the sending as well as the receiving subnetwork.

IC Programming. Currently, developer support for applications programmed in *Rust*, *Motoko* [18], or Python [15] exists. The IC canister code is compiled down to WebAssembly, which is executed under a sandboxed virtual machine on the IC nodes. Any language that can be compiled down to WebAssembly could also be used. A detailed description of WebAssembly execution is provided in Section 3. Listing 1 shows an example of a functional 15-line key-value store canister implemented in Rust. It exports one update call to insert elements in the kv-store and one query call to retrieve them.

2.3 The IC Protocol Stack

As illustrated in Figure 1, the Internet Computer Protocol consists of four layers.

Peer-to-peer Layer. Within a subnet, nodes exchange information to achieve consensus on the replicated state and the messages to be processed next. To this end, the peer-to-peer layer offers a (prioritized) broadcast service to the layers above. To conserve bandwidth, peer-to-peer relies on an advert-based mechanism, where nodes first send a small advert to announce they have an artifact. Other nodes can then request the artifact if they need it, based on the details in the advert. Peer-to-peer relies on TLS over TCP streams between the nodes of a subnet. On top of that, it adds further reliability with notifications for unsent messages (in case of sender-side errors), and automatic connection re-establishment and requests for recent adverts.

Consensus Layer. Incoming messages must be validated and ordered so all replicas process them in the same order. The IC uses a novel consensus protocol [9] briefly described here.

The protocol proceeds in rounds. The replicas grow a tree of blocks referencing valid predecessor blocks. Their local trees form a consistent yet sometimes locally incomplete tree view. In each round, a pseudo-random process is used to assign each replica a unique rank. The replica of lowest rank is the *leader* of that round. When the leader is honest and the network is synchronous, the leader will propose a block, which the other honest nodes in the subnet will validate and add to their local tree. If the leader is not honest or the network is not synchronous, some other replicas of higher rank may also propose blocks, have them validated and added to the tree. Whenever $2f+1$ replicas report that they added exactly one block to the tree, this block and its predecessors on the path to the root are declared finalized and the non-finalized parts of the tree up to this height are pruned.

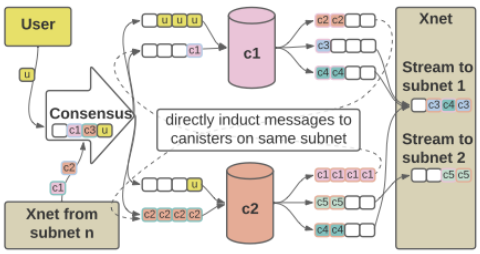


Figure 2: Routing messages through the IC protocol stack. Messages for canisters, issued by users or canisters on other subnets, are validated and ordered by consensus. Subsequently, messages are put into input queues for their destination canister. Messages created by canisters are put into output queues from where they are either transferred to their respective input queues on the same canister (bypassing consensus) or sent as part of streams to their target subnet.

One can prove that this protocol provides the consensus properties, namely *safety* (i.e., all replicas in fact agree on the same ordering of inputs) and *liveness* (i.e., all replicas should make steady progress). The IC consensus protocol guarantees safety despite asynchrony. This means that there is no assumption of an upper bound on the time to send information from one node to another. For liveness short intervals with fast message delivery are sufficient. The IC consensus protocol degrades gracefully when some replicas are malicious.

Message Routing. Once the consensus layer orders input messages, they are delivered to the message routing layer. The destination canister for each message is selected and the messages are enqueued for processing by the execution environment. During execution, the destination canister updates its state as part of the replicated state machine and generates outputs handed back to the message routing layer.

The message routing layer enqueues messages in one of multiple input queues. For each canister C running on a subnet, there are several input queues — there is one queue specifically for user-generated messages to C. Furthermore, each other canister C', from which C receives messages, gets its own queue. In each round, the execution layer will consume some of the inputs in these queues, update the replicated state of the relevant canisters, and place outputs in various output queues. For each canister C running on a subnet, there are several output queues — each other canister C', with whom C communicates, gets its own queue. The message routing layer will take the messages in these output queues and place them into subnet-to-subnet streams to be processed by a crossnet transfer protocol, whose job it is to actually transport these messages to other subnets. This is visualized in Figure 2.

Thus, the replicated state comprises the state of the canisters, as well as "system state", including the above-mentioned queues and streams. Thus, both the message routing and execution layers are involved in updating and maintaining the replicated state of a subnet. It is essential that all of this state

is updated in a completely deterministic fashion, so that all replicas maintain exactly the same state.

The consensus layer is decoupled from the message routing and execution layers, in the sense that only messages from finalized blocks of the chain reach routing and execution. Temporary block tree branches are pruned before their payloads are passed to message routing. This is in contrast to other blockchains which execute blocks speculatively, before ordering and validating them [46].

Execution Layer. The Execution Environment operates in rounds, during which it takes messages from canister input queues and executes the corresponding Wasm function with the message as payload. Based on the input and canister state, the execution environment updates the canister state, and could additionally add messages to output queues. One of the main challenges is that computation must be deterministic for state machine replication to work.

A scheduler determines in which order messages are executed in each round. The main goals of the scheduler are (see Section 3.3 for a more detailed description): (1) it must be *deterministic*; (2) it should distribute workloads *fairly* among canisters (3) optimizing for *throughput* over *latency*.

The IC offers orthogonal persistence, an illusion given to programs to run forever: the heap of each canister is automatically preserved and restored the next time it is called. Listing 1 shows an example key-value store that illustrates how easy it is to use orthogonal persistence. The key-value store in this case is backed by a simple Rust HashMap stored on the Wasm heap by means of a thread-local variable. We use a RefCell to provide interior mutability. The example would also be possible without it, but mutating the thread-local variable would be unsafe in that case, as the Rust compiler cannot guarantee exclusive access to it.

3 Systems Challenges of the IC

We focus on the execution layer of the IC and discuss the challenges C1-C4, as well as their solutions.

The IC can execute arbitrary programs. The basic computational unit in the IC is called a canister. Canister programs are encoded in WebAssembly (Wasm) [28], a binary instruction format for a stack-based virtual machine.

The main goal is to execute `deterministically`, `securely` and `efficiently` the functions triggered by messages sent to canisters. Each canister is executing under a long-running Wasm virtual machine whose state is persisted over long periods of time. In terms of efficiency IC nodes are able to sustain running tens of thousands of canisters [17]. The *memory subsystem* of the nodes addresses challenge **C1**.

The IC needs to scale up, by achieving high resource utilization on individual nodes. This is important to achieve performance comparable to native systems and to amortize the cost of state machine replication. Essential for achieving

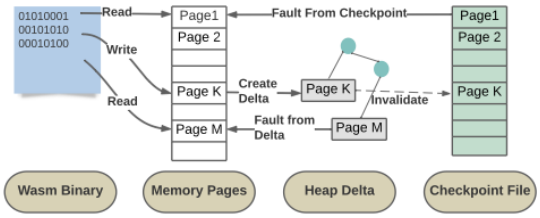


Figure 3: Memory faulting architecture, including heap delta and checkpoint file. When Wasm instructions trigger page faults, memory contents can be faulted in from the memory checkpoint. When pages are dirtied by writes, heap deltas are created, which invalidate page content in the checkpoint file. Subsequent faults are served directly from heap deltas. Periodically, heap deltas are merged into a new checkpoint.

these goals is to enable efficient execution of developer code through Wasm code execution, solving challenge **C2**.

To ensure correct and deterministic state machine replication, we designed and implemented a deterministic scheduler for the IC nodes. Our scheduler implements a deterministic time slicing mechanism, effectively solving challenge **C3**.

Security is achieved at multiple layers of the IC through trust, consensus, byzantine fault-tolerance and so forth. Details about these can be found in our whitepaper [53]. At this layer, we ensure security through operating systems and virtualization mechanisms effectively solving challenge **C4**.

3.1 C1 - Statefulness - The Memory Subsystem

Currently, canisters can use up to 52 GiB of memory to be accessed by users. Any implementation of orthogonal persistence has to solve two problems: **(1)** How to map the persisted memory into the Wasm memory; and **(2)** How to keep track of all modifications in the Wasm memory so that they can be persisted later. We use page protection to solve both problems. We divide the entire address space of the Wasm memory into 4 KiB pages. All pages are initially marked as inaccessible using the page protection flags of the OS.

The first memory access triggers a page fault, pauses the execution, and invokes a signal handler. The signal handler then fetches the corresponding page from persisted memory and marks the page as read-only. Subsequent read accesses to that page will succeed without any help from the signal handler. The first write access will trigger another page fault, however, and allow the signal handler to remember the page as modified and mark the page as readable and writable. All subsequent accesses to that page (both r/w) will succeed without invoking the signal handler.

Invoking a signal handler and changing page protection flags are expensive operations. Messages that read or write large chunks of memory cause a storm of such operations, degrading performance of the whole system. This can cause severe slowdowns under heavy load.

Versioning: Heap Delta and Checkpoint Files. A canister

Figure 4: The performance improvement given by memory faulting optimizations (lower is better). Note the logarithmic vertical axis. Speedups range from 1.25X to 3.5X.

executes update messages sequentially, one by one. Queries, in contrast, can run concurrently to each other and to update messages. The support for concurrent execution makes the memory implementation much more challenging. Consider that a canister is executing an update message at (blockchain) block height H. At the same time, there could still be a previous long-running query that started earlier, at block height $H-K$. This means the same canister can have multiple versions of its memory active at the same time; this is used for the parallel execution of queries and update calls.

A naive solution to this problem would be to copy the entire memory after each update message. That would be slow and use too much storage. Thus, our implementation takes a different route. It keeps track of the modified memory pages in a persistent tree data-structure [41] called Heap Delta that is based on Fast Mergeable Integer Maps [37]. At a regular interval (i.e., every N rounds), there is a checkpoint event that commits the modified pages into the checkpoint file after cloning the file to preserve its previous version. Figure 3 shows how the Wasm memory is constructed from Heap Delta and the checkpoint file.

Memory Faulting Optimizations. We describe below three optimizations we designed to improve memory faulting.

◊ **Optimization 1:** *Memory mapping the checkpoint file pages.* This reduces the memory usage by sharing the pages between multiple calls being executed concurrently. This optimization also improves performance by avoiding page copying on read accesses. The number of signal handler invocations remains the same as before, so the issue of signal storms is still open after this optimization.

◊ **Optimization 2:** *Page tracking in Queries.* All pages dirtied by a query are discarded after execution. This means that the signal handler does not have to keep track of modified pages for query calls. As opposed to update calls, for queries we introduced a fast path that marks pages as readable and writable on the first access. This low-hanging fruit optimization made queries 1.5x-2x faster on average.

◊ **Optimization 3:** *Amortized prefetching of pages.* The idea behind the most impactful optimization is simple: to reduce the number of page faults, we need to do more work per signal handler invocation. Instead of fetching a single page at a time, the signal handler tries to speculatively prefetch pages. The right balance is required here because prefetching too many pages may degrade performance of small messages that access only a few pages. The optimization computes the largest contiguous range of accessed pages immediately preceding the current page. It uses the size of the range as a hint for prefetching more pages. This way the cost of prefetching is amortized by previously accessed pages. As a result, the optimization reduces the number of page faults in memory intensive messages by an order of magnitude.

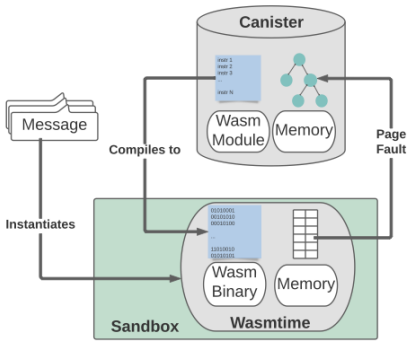


Figure 5: The execution of messages instantiates a Wasmtime instance in a sandboxed environment. Each sandbox can run multiple wasmtime instances. The Wasm module is compiled to a binary running inside the VM while memory accesses are faulted in from the memory heap deltas and checkpoint file.

These optimizations bring substantial benefits for the performance of the memory faulting component of the execution environment. Figure 4 plots the performance optimizations we achieved when enabling all three optimizations in comparison with turning them off. We measured this for a memory intensive benchmark which allocates 8 bytes, 1 MiB, or 512 MiB. The optimizations allow the IC to improve its throughput for memory-intensive workloads as depicted in the Figure and no performance degradation was observed for other workloads.

3.2 C2 - Scalability: Wasm Execution

To process a message, the canister executes the corresponding function in the Wasm module. Figure 5 depicts this process. Function execution requires a Wasm instance which is a combination of Wasm code and memory. One of the challenges is that we cannot afford to keep one Wasm instance alive for each of the canisters running in a subnet because we would run out of memory. Instead, we construct Wasm instances on demand for each message and dispose of them after the message execution. Thus, the latency of message execution depends on the instantiation time and the actual time to execute the function. Included in this instantiation time is also the time to compile the Wasm code. One optimization we deployed is to cache compilations of Wasm code.

For one of the most used canisters in production, the compilation cache optimization reduces the P99 for running non-replicated queries by 2 orders of magnitude. This is depicted in Figure 6. Therefore, for our real-world example, cold-start times are now below 10 ms for previously compiled user-code

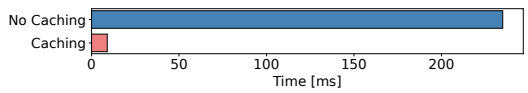


Figure 6: Compilation caching effects. P99 for running a short (1 ms), non-replicated query on a cold started canister.

and around 230 ms when compiled for the first time. This is an important achievement compared to major serverless providers where cold-start times are in the order of seconds to tens of seconds [1, 49, 51, 55]. However, compilation time varies proportionally with the complexity of the Wasm code being compiled. Therefore, for certain applications longer first-time compilation times are to be expected—subsequent calls are optimized by caching the compilations.

Section 3.1 describes how the Wasm instance manages a canister's memory in checkpoint files or heap deltas.

Instrumentation: Time-slicing and Accounting. Since Wasm is Turing-complete we need a mechanism to ensure termination of message execution. Otherwise, a faulty or malicious smart contract would be able to stall the progress of the blockchain by potentially infinitely long running messages. As everything else in the execution layer, the point at which we do that needs to be deterministic. However, execution duration is not, as execution might be slightly different on different nodes due to nondeterministic events in the system (e.g. one machine having a page fault, but not the other one).

Instead, we instrument Wasm code to count the number of instructions that have been executed for each message that is being processed by the system. To reduce the performance overhead, our compiler extension performs counting at the basic block level instead of individual instructions. Concretely, at the start of an execution round we initialize the global instruction counter of the Wasm module for each canister to the instruction limit. In each basic block of the Wasm module we insert a snippet of code to decrement the counter by the number of instructions in the basic block. In re-entrant blocks, such as function and loop headers, we insert code that aborts message execution when the counter is negative.

Another benefit of quantifying computation done via instruction counting is that we can deterministically charge canisters for performed work. The canisters are charged for the resources they are consuming, including computation, communication and storage. For that reason, the IC needs to account for resource usage of all canisters in the system. Two examples for resources being accounted are memory accesses (estimated by the number of pages read and written) as well as CPU instructions used. Memory accesses are tracked with the memory protection mechanisms as described in Section 3.1.

Resources also need to be accounted for when serving users query calls. Queries are especially complex since their execution is non-replicated due to their execution on just a single node. However, the canister still needs to be charged deterministically by all nodes as the balance of a canister is part of the canister's state which is managed by state machine replication. This is currently an open problem we are investigating.

3.3 C3 - Deterministic Scheduling

Since we are operating a replicated state machine, it is essential that each replica processes the same inputs in the same order. To achieve this, the replicas in a subnet run a consensus protocol [21], which ensures that they process inputs in the same order. If the IC code executing those messages as well as the canister code itself is deterministic, the internal state of each replica will evolve over time in exactly the same way, and each replica will produce exactly the same sequence of outputs in the absence of hardware-related problems.

Granularity. For simplicity, the scheduler works at a coarse level, scheduling canisters instead of individual messages and executing each canister until there are no more messages in its queues or the system-defined instruction limit for a round is reached. IC nodes are modern dual-socket multi-core servers. Deterministic behavior on such a machine can be achieved when canisters are pre-allocated to specific CPU cores at the beginning of each round. Our current scheduler takes this approach because it is a simple and effective design choice which is then easily proven correct (see Appendix A).

Allocation and fairness. To ensure responsiveness under heavy load, canisters have the option of paying upfront for a *compute allocation*. Since canisters are single threaded, a *compute allocation* is a fraction of one CPU core, expressed in percentage points. Only part of a subnet's compute capacity can be allocated, ensuring progress for canisters with zero compute allocation, i.e. best effort canisters. Fairness is defined as guaranteeing canister compute allocations (i.e., a backlogged canister with compute allocation A executing at least A full rounds out of every 100) and evenly distributing the remaining capacity ("free compute") across all canisters.

Given a deterministic state machine with N CPU cores (and $N \times 100$ compute capacity), we schedule (at least) N canisters to execute a *full round*: a round, in which a canister either exhaust the instruction limit or completes the execution of all their enqueued messages. The scheduling algorithm uses credits accumulated across rounds as priority: an amount of credits equal to the canister's compute allocation plus a uniform share of the *free compute* is credited to every canister at the beginning of every round; canisters in the priority queue are assigned round-robin to CPU cores (each of the first N canisters are scheduled first on a CPU core), and 100 credits are debited from each canister that executes a full round.

Our algorithm's time complexity to compute the schedule algorithm is linear in the number of canisters, which is acceptable because scheduling only happens once per round (alongside other operations that require linear time). The scheduling algorithm has the following properties:

- `Correctness wrt. compute allocation`: every backlogged canister gets a "full execution round" at least A out of every 100 rounds, where A is the canister compute allocation.

- `High throughput`: in the absence of information regarding the number of instructions required to execute each message ahead of time (which might allow for better bin packing); and given the constraint that canisters must be allocated to specific cores ahead of time; the algorithm provides optimal throughput by executing canisters allocated to each core until the round instruction limit is reached.
- `Fairness`: the credits system ensures that (backlogged) canisters with equal compute allocations get the same number of "full execution rounds" over a long enough time period.

Appendix A defines and anlyses the scheduler formally.

Deterministic Time Slicing. The scheduler, as explained above, although effective, is suboptimal for messages that trigger executions of varying length. Each subnet of the IC operates in epochs of many rounds. Messages run either to completion or to a predefined upper limit on the number of instructions per round. In the second case, the execution is aborted and returns an error. In this case, the user would have to rewrite the algorithm to make the execution of the long-running message span multiple execution rounds. This is less than ideal because of two reasons. First, it can artificially increase round duration due to stragglers, which leads to overall throughput loss and slowdown. Second, it can lead to significant CPU waste because a long-running execution that is aborted due to reaching instruction limits will be re-executed.

To solve this problem, we designed a *deterministic time slicing* mechanism on top of our scheduler, where each message execution longer than a round is `sliced` in a number of intervals with roughly equal numbers of instructions. This is akin to time slicing in modern schedulers [7], although the biggest challenge here is to enforce determinism. Time slicing also increases the amount of intermediate state, that needs to be kept while a message is preempted. The slices achieved here are then scheduled as described before.

3.4 C4 - Ensuring Security

The security model of the IC aims to provide *access control* and *integrity* of canister and system data in the presence of malicious canisters and users. Canisters can specify a method that accepts or rejects requests, e.g., based on caller ID, resource consumption etc. Only correctly signed messages are then processed. Responses to update calls and queries for certified variables are threshold-signed by the subnet nodes, so clients can verify authenticity. Canisters cannot inspect or change state of other canisters or parts of the system. This is guaranteed by eliminating the main Wasm attack vectors.

An adversary may craft Wasm code to: (1) exploit bugs in the Wasm engine to escape its protection mechanism; and (2) perform side-channel attacks to obtain data from the system or other canisters [33]. The IC protects against these attacks using OS isolation and sandboxing. Each canister is compiled and executed in its own sandboxed process that communicates only with the main replica process via security-audited IPC.

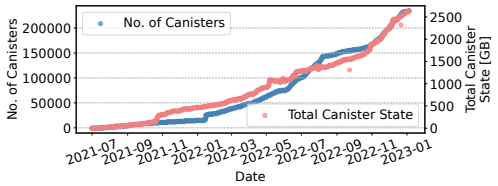


Figure 7: Total number of canisters running on the IC.

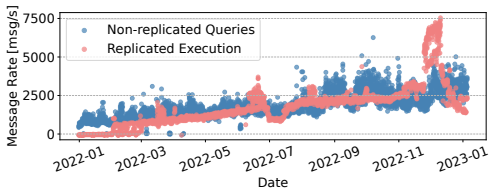


Figure 8: The rate of non-replicated and replicated messages.

Sandboxes are given minimal permissions needed to execute using object-based access control (SELinux).

In the future, hardware-based security, offering fully encrypted VMs with the possibility to attest remotely if the expected VMs are running, will increase obstacles curious and malicious node providers face.

4 The Internet Computer In Data

We present data related to the operation and performance of the IC. We show the growth and usage patterns the IC is experiencing, its overall performance (in comparison with native code) and identify sources of overhead with regard to the systems challenges presented in Section 3.

The Internet Computer Hardware. The IC currently runs on homogeneous hardware that is hosted by independent node providers. The chosen configuration makes use of dual-socket AMD EPYC 7302 processors with a total of 32 physical cores running at 3 GHz, each core having 2 hardware threads. The IC servers make use of 503 GiB of memory. AMD chips were chosen due to their secure encrypted virtualization feature to enable VM encryption across VM upgrades in combination with remote attestation.

The data presented in Sections 4.1-4.2 is gathered from production. Experiments discussed in the rest of this section are executed on an internal testnet that mimics subnets on the IC. The difference to IC machines is that two IC VMs are deployed to each host, instead of one. Testnet machines are hence expected to be slower for concurrent workloads.

4.1 A High-level View of the IC

IC Growth (C1 + C2). We focus on the high-level operational data gathered from the IC. The usage has been steadily increasing since launch, showing an acceleration of the numbers of deployed applications since the beginning of 2022. Figure 7 depicts the number of deployed canisters over time, as well as their overall allocated state, which reaches 2.5TB.

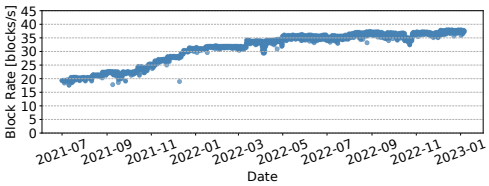


Figure 9: The block rate of the IC.

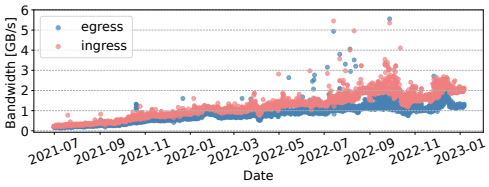
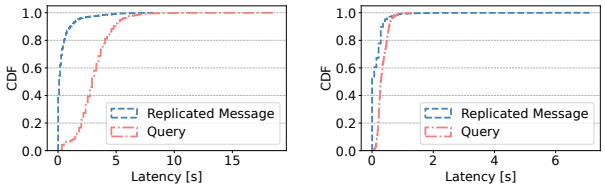


Figure 10: Aggregated IC Nodes network traffic.

Workload Growth (C1 + C2). Together with the increase in the deployed applications we also observe an increase in the workload deployed on the IC. Figure 8 plots the arrival rate of messages over time. As expected, non-replicated messages (i.e., queries), which do not pass through consensus and do not alter canister state are being triggered significantly more often than replicated messages by our users. Replicated execution, which incurs the consensus overhead is used less — we assume only for operations that need to alter application state. An interesting observation to make here is that in February 2022, we have changed the way in which we quantify the number of replicated messages. Whilst before this date, the number only sums up *update* calls, after this date the date adds also replicated execution that the canisters use for their operation (e.g., periodic heartbeats). The graph shows a significant increase in the number of replicated messages after February 2022. We note here that it is common practice that for *operational systems*, metrics sometimes change meaning over time as the systems itself is refined and continuously evolving. The significant increase in replicated execution in Dec 2022 is due to the launch of a popular application, while the later drop corresponds to a change in the call pattern of the same application.

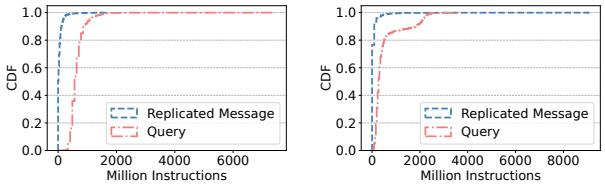
IC Scaling Out (C2). Over time, the IC has increased its capacity to sustain increased workloads and achieve decentralization. Evidence to sustain the scaling out of the IC is given by examining the *block rate* – the number of blocks generated by the consensus layer. Figure 9 plots these data, showing an increase of 57% since launch.

Similar evidence to sustain growth of the IC in terms of higher workload demands is represented by the increased network bandwidth used to exchange messages between nodes. Figure 10 depicts the aggregated bandwidth of traffic generated: since launch, the traffic generated between the nodes has increased from around 250 MB/s to about 3 GB/s, over an order of magnitude increase over a period of a year. The amount of ingress and egress traffic is very similar, as expected.



(a) OpenChat Subnet. *(b) DSCVR Subnet.*

Figure 11: The time queries or replicated messages take in the execution layer, without consensus overhead.



(a) OpenChat Subnet. *(b) DSCVR Subnet.*

Figure 12: The number of instructions spent for executing queries or replicated messages.

4.2 The IC Performance

The IC has attracted the developers of several large applications, for example: *OpenChat* (a decentralized chat application), *DSCVR* (decentralized social news aggregator), and *distrikt* (a decentralized professional social media platform). These applications have added a significant increase in workload complexity for the IC. We present an in-depth analysis of the metrics of the subnet that runs the OpenChat canisters. We focus mostly on data related to the systems challenges described in Section 3.

The subnet hosting the OpenChat application is composed of 13 replica nodes distributed geographically (in North America, Europe and East Asia). The subnet hosts 80,000 canisters.

Replicated vs. Non-Replicated Execution. (C1-C3) First, we assess the duration of non-replicated queries in comparison with replicated update calls in Figure 11. We compare the data in the OpenChat subnet with the data coming from the subnet running DSCVR (and other applications). Note that this comparison does not include the overhead of the P2P, consensus and messaging layer, but only measures the time spent in the Execution layer. By analyzing the data in Figure 11 we conclude that queries run longer for the OpenChat subnet, while updates dominate on the DSCVR subnet. Figure 12 plots the number of Wasm instructions for queries compared to replicated messages. The average number of instructions executed for queries is significantly higher than for replicated messages for OpenChat, leading to longer executions. On the contrary, on the DSCVR subnet the behavior is the exact opposite. This shows that the IC runs a diverse set of applications, with varied needs and characteristics.

Memory Overhead (C1 + C2). The heap delta is used in-between checkpoints to keep track of modified memory pages. This data structure can affect the subnets' ability to scale up

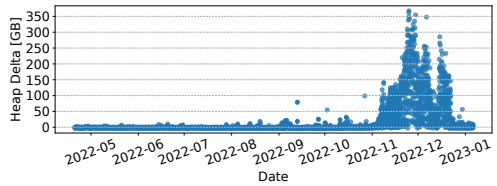


Figure 13: Heap delta for an IC subnet over time.

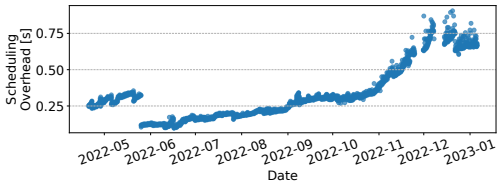


Figure 14: Scheduling overhead for an IC subnet over time.

to many concurrent canisters as tracking modifications incurs overhead. Figure 13 shows how the heap delta for all canisters on OpenChat subnet evolves over time. The data shows that on average the heap delta is below 5 GB, but more recently, since November 2022 it has significantly increased, up to 350 GB. This shows a large increase in OpenChat usage, aggregated for more than 80,000 canisters. On subnets with fewer canisters heap deltas are typically smaller.

Scheduling Overhead (C3). When scheduling replicated messages the same ordering has to be ensured on all replicas so that determinism is guaranteed. Deciding which message gets scheduled for execution and at what time is a costly operation that can affect scalability. We plot the scheduling overhead, i.e., the time to compute the schedule for one round, in the OpenChat subnet in Figure 14. The immediate conclusion is that this overhead is in the order of hundreds of milliseconds per round for this subnet. An interesting point is the overhead reduction which can be observed around the end of May 2022. This is attributed to an optimization related to the scheduling process, namely checking if a canister has messages to be run. We checked the scheduling overhead for subnets with smaller numbers of canisters as well. Our conclusion is that this overhead is proportionally smaller (as the overhead complexity is $O(N \log N)$ for N canisters). Similarly to the heap delta observation, the scheduling overhead increased significantly since November 2022. This is also because the number of users (and canisters) for OpenChat has significantly increased, leading to many more messages being scheduled for execution in this subnet. We investigate ways to reduce this overhead to support efficient scheduling with larger workloads.

4.3 The IC Virtualization Stack

We quantify the impact of the IC virtualization. In Section 3 we described how user code is executed. In short, user code is compiled to Wasm, which gets instrumented and compiled to a binary that gets executed inside a sandbox. To achieve orthogonal persistence and stateful execution we keep track of memory writes in a persistent data structure from which the Wasm VM is faulting in its pages. This indirection introduces a non-trivial overhead. We quantify this overhead for two types of workloads: compute- and memory-intensive. Each of these workloads stress different resources of the stack.

Compute Intensive Workload (C2). We implemented a workload that calculates prime numbers up to a given integer number. This is a single threaded workload, which we wrote in Rust and deployed on the IC. We ran the same workload (identical Rust code) on an IC machine natively (compiled to an x86 binary), without the entire virtualization stack. Furthermore, we ran the same Rust code on one of the top-3 serverless providers. Experiments in the IC have been measured from within the Execution Environment and hence do not contain network latency or the cost of other parts of the IC stack. We provide similar data for the serverless provider and take the latency from the provider's dashboard. The experimental data is presented in Table 1. The overhead is computed against the native execution (not against the serverless provider).

First, the Internet Computer performance compared to native execution is good for longer-running workloads considering the extra features that the IC execution environment provides: sandboxing, accounting and tracking changes. Second, we emphasize that the IC performance is in the same order of magnitude with one of the top-3 serverless providers. Considering the extra operations that the IC does to offer its users decentralization, security etc., we deem these performance data encouraging, especially taking into account the fact that the serverless execution is faster than native execution in our case. This directly implies that the hardware running the serverless platform is very likely faster than the hardware we described in Section 4.

Memory Intensive Workload (C1). We performed a similar experiment with a memory intensive workload. Here, memory is accessed sequentially in strides of 8 bytes. The totally allocated memory is 1 GB. In this experiment we only compare against a native execution because all current serverless platforms are not stateful. Therefore, the serverless functions access memory directly (i.e., without any faulting architecture, persistence, versioning) through either microVMs [1] or containers [43]. A more direct comparison would involve a serverless function that stores its state in a storage environ-

n	**IC** [ms]	**Native** [ms]	**Serverless** [ms]	**Slowdown IC / native**
0	1.40	0.02	3.53	70 X
100	1.43	0.03	1.93	47 X
1,000	2.35	0.94	2.94	2.5 X
10,000	41.54	33.85	19.65	1.22 X
50,000	718.26	610.73	347.54	1.17 X

Table 1: Median computation time for a compute intensive workload running on the IC, native execution and running on a serverless provider (average for serverless due to lack of raw data) over 30 executions. The workload identifies primes in the first n integers.

Operation	Data Size [Bytes]	IC [ms]	Native [ms]	Slowdown IC / native
Read	50,000	2.15	0.02	107 X
Read	5,000,000	26.36	1.83	14.2 X
Read	50,000,000	195.52	18.27	10.7 X
Write	50,000	2.28	0.02	114 X
Write	5,000,000	33.79	2.14	15.8 X
Write	50,000,000	277.36	19.13	14.5 X

Table 2: Median computation time over 30 executions of a memory intensive benchmark performing strided reads/writes of different sizes on the IC and native execution.

ment [32], but this is outside the scope of this article.

We therefore compare the execution time of executing the benchmark compiled to a native x86 binary on one of the IC node to the time of executing the same benchmark as canister code. Table 2 summarizes our findings. For workloads that touch up to 50 MiB of data, the overhead of running this benchmark on the IC is approximately 10 X to 15 X. Lower amounts of data touched incur larger slowdowns, with smaller data giving the largest slowdown.

Even though the slowdown compared to native execution seems large, we remind the reader that a rather deep virtualization stack is involved in memory operations. Further, the IC needs to account for resource consumption and track memory writes, which the native version does not do. Finally, update calls pass through consensus and the entire IC stack, therefore offering the users all the Internet Computer benefits: decentralization, security, and tamper-proof execution. Moreover, we remind the reader that the IC is orders of magnitude faster and more efficient than other blockchains. We are further working on improving the memory faulting layer using write barriers [6] or userfaultfd [39] so that in the future we can reach our goal of (close-to-)native performance.

The Cost of Decentralization and Statefulness (C1-C4). One of the overarching goals of the IC is to offer levels of performance as close as possible to *native* and traditional client-server architecture performance. With regard to user-perceived overhead, the factors contributing most are the consensus protocol, the networking, and the crypto primitives, as well as memory faulting. At a macro level, this overhead can be observed by re-interpreting Figures 9 and 10. All the replication protocol-related mechanisms leads to network traffic (e.g., a few MB per machine per second, see Figure 10, considering that at the moment of writing the IC runs on over 500 machines) and computational overhead for the creation and validation of blocks and the messages contained therein. This overhead is not crippling the operation of the IC and its benefits significantly outweigh its downsides.

We ran many memory-intensive *update* calls in one of our testing and benchmarking subnets. Memory-intensive updates especially stress the entire system stack because they: (i) modify significant amounts of the canister state; (ii) need replicated execution; (iii) require consensus for ordering. Therefore, we quantify the overhead of system components by running `Linux perf`. Figure 15 is an instance of quantifying

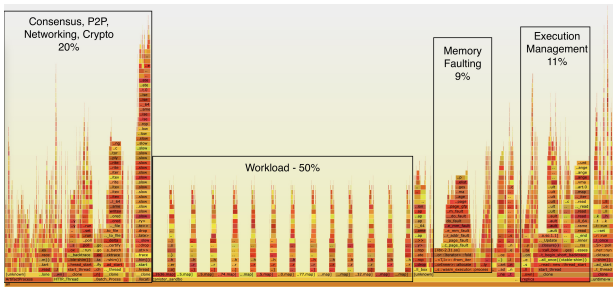


Figure 15: Decentralization and statefulness overheads when several memory-intensive `update calls` are made. Data presented as flame graphs [26].

such overhead. We observe that the actual workload takes approximately 50% of the used CPU time (not all the CPU capacity is used). A large fraction of the overhead can be attributed to the consensus and P2P protocol, networking or crypto primitives stack, together they account for roughly 20% of the CPU time. Another sizeable CPU time consumption is due to the memory faulting subsystem (9%) and the execution management stack (11%). The latter involves all processing related to canister administration, Wasm instrumentation, communication with sandboxes.

4.4 End-to-end Performance (C1-C4)

We quantify the end-to-end performance of our subnets using the default subnet size of 13 nodes. Note that for enhanced security guarantees (e.g., tolerating more malicious nodes) the IC hosts even larger subnets (i.e., 40 machines). All subnets are geo-replicated, i.e., are composed of machines running on multiple continents – the Americas, Europe, Asia. Our results are depicted in Table 3.

In contrast to other experiments we execute requests with insignificant execution overhead, so we can safely attribute the latency overhead to other layers. A geo-replicated subnet is able to run ∼78K queries per second with a latency of 50ms-200ms, given by differences in geographic location of client and targeted IC node. Since no coordination among nodes is needed for query execution, we can safely attribute this latency to the networking layers. In terms of updates (stateful and replicated execution), a geo-replicated subnet is able to serve 950 updates per second for a latency of 1-4s (which

Op	Throughput (ops / s)	Latency (s)	Overheads
Query	78,000	0.05-0.2	Networking
Update	950	1-4	Networking, Consensus, Replicated Execution, Statefulness

Table 3: End-to-End performance for the two operations supported by the IC.

includes networking, consensus and replication overheads). Note that this latency is comparable to the top-3 serverless platforms cold starts [54].

5 Related Work

The IC builds on fruitful years of research at many levels: from consensus, to peer-to-peer networking, cryptographic protocols, blockchain, and operating systems. We limit our discussion to blockchain-related technology and large-scale systems making use of it, and the value of opening up and discussing data from large-scale operational systems.

Blockchain Execution Environments. It has become a mantra that blockchains are slow. Just like for the IC, others have investigated ways in which computations and transactions running atop blockchains can be sped up. These are related to either speeding up consensus [12, 52], using software transactional memory [25,46,47], enforcing determinism and ordering [34, 52], or optimizing execution layers [2]. The performance evaluation in these works relies on synthetic workloads in test environments. To the best of our knowledge, this paper is the first to report on the performance of the execution environment of a real-world blockchain deployment.

Operational Systems and Data. Next to the more traditional workloads, such as high-performance scientific computing [20], analytics [4], cluster workloads [45], recently data centers started providing Blockchain-as-a-service offerings [23]. As Amvrosiadis et al. point out [3] data set diversity is key to understand the characteristics of workloads and to tailor new resource management schemes. The community recognizes and emphasizes the need of analyzing operational systems, such as the Microsoft Serverless platform [49], CloudLab [19], or the evolution of the Google data center network [40]. We believe our article adds significant data and insight on the design, operation and growth of systems that offer general-purpose computation capabilities on top of blockchain platforms.

6 Conclusion

The Internet Computer overcomes traditional blockchain limitations with respect to speed, storage costs, and computational capacity. We demonstrated how the novel design of the IC, coupled with solving low-level systems challenges enables decentralized stateful serverless. In particular, we presented an in-depth description of the execution layer of the IC followed by an evaluation of operational and performance data over real-world workloads as well as compute and memory-intensive benchmarks.

The IC code and data can be found here:
- IC code: https://github.com/dfinity/ic
- Dashboard: https://dashboard.internetcomputer.org/
- Dataset API: https://ic-api.internetcomputer.org/api

References

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)* (2020), pp. 419–434.

[2] Amiri, M. J., Agrawal, D., and El Abbadi, A. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 1337–1347.

[3] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 533–546.

[4] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), pp. 1383–1394.

[5] Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., and Danezis, G. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (2019), pp. 183–198.

[6] Blackburn, S. M., and Hosking, A. L. Barriers: Friend or foe? In *Proceedings of the 4th international symposium on Memory management* (2004), pp. 143–151.

[7] Bouron, J., Chevalley, S., Lepers, B., Zwaenepoel, W., Gouicem, R., Lawall, J., Muller, G., and Sopena, J. The battle of the schedulers:{FreeBSD}{ULE} vs. linux {CFS}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 85–96.

[8] Brooker, M., Chen, T., and Ping, F. Millions of tiny databases. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 463–478.

[9] Camenisch, J., Drijvers, M., Hanke, T., Pignolet, Y.-A., Shoup, V., and Williams, D. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (2022), pp. 81–91.

[10] Carbon Crowd. CO2 emissions assessment of the internet computer. https://wiki.internetcomputer.org/wiki/L1_comparison, 2022.

[11] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (aug 2013).

[12] Danezis, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 34–50.

[13] Dang, H., Dinh, T. T. A., Loghin, D., Chang, E.-C., Lin, Q., and Ooi, B. C. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data* (2019), pp. 123–140.

[14] DEARLE, A., KIRBY, G. N., AND MORRISON, R. Orthogonal persistence revisited. In *International Conference on Object Databases* (2009), Springer, pp. 1–22.

[15] DEMERGENT LABS. Python cdk for the internet computer. https://github.com/demergent-labs/kybra, 2023.

[16] DFINITY FOUNDATION. Comparison between the internet computer and other l1 blockchains. https://wiki.internetcomputer.org/wiki/L1_comparison, 2022.

[17] DFINITY FOUNDATION. The internet computer subnets. https://dashboard.internetcomputer.org/subnets, 2023.

[18] DFINITY FOUNDATION. The motoko programming language. https://internetcomputer.org/docs/current/developer-docs/build/cdks/motoko-dfinity/motoko/, 2023.

[19] DUPLYAKIN, D., RICCI, R., MARICQ, A., WONG, G., DUERIG, J., EIDE, E., STOLLER, L., HIBLER, M., JOHNSON, D., WEBB, K., ET AL. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)* (2019), pp. 1–14.

[20] FEITELSON, D. G., TSAFRIR, D., AND KRAKOV, D. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing 74*, 10 (2014), 2967–2982.

[21] FISCHER, M. J. The consensus problem in unreliable distributed systems (A brief survey). In *Fundamentals of Computation Theory, Proceedings of the 1983 International FCT-Conference, Borgholm, Sweden, August 21-27, 1983* (1983), vol. 158 of *Lecture Notes in Computer Science*, Springer, pp. 127–140.

[22] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[23] GAI, K., GUO, J., ZHU, L., AND YU, S. Blockchain meets cloud computing: A survey. *IEEE Communications Surveys & Tutorials 22*, 3 (2020), 2009–2030.

[24] GARAY, J., AND KIAYIAS, A. Sok: A consensus taxonomy in the blockchain era. In *Cryptographers' track at the RSA conference* (2020), Springer, pp. 284–318.

[25] GELASHVILI, R., SPIEGELMAN, A., XIANG, Z., DANEZIS, G., LI, Z., XIA, Y., ZHOU, R., AND MALKHI, D. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. *arXiv preprint arXiv:2203.06871* (2022).

[26] GREGG, B. The flame graph. *Communications of the ACM 59*, 6 (2016), 48–57.

[27] GROTH, J., AND SHOUP, V. On the security of ecdsa with additive key derivation and presignatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2022), Springer, pp. 365–396.

[28] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 185–200.

[29] HUANG, H., KONG, W., ZHOU, S., ZHENG, Z., AND GUO, S. A survey of state-of-the-art on blockchains: Theories, modelings, and tools. *ACM Computing Surveys (CSUR) 54*, 2 (2021), 1–42.

[30] JIA, Z., AND WITCHEL, E. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 691–707.

[31] JORDAN, M. J., AND ATKINSON, M. P. Orthogonal persistence for java—a mid-term report. *Morrison et al.[161]* (1999), 335–352.

[32] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 427–444.

[33] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., ET AL. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[34] MEHRARA, M., HAO, J., HSU, P.-C., AND MAHLKE, S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices 44*, 6 (2009), 166–176.

[35] MERKLE, R. C. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology* (Berlin, Heidelberg, 1987), CRYPTO '87, Springer-Verlag, p. 369–378.

[36] NGUYEN, D., LENHARTH, A., AND PINGALI, K. Deterministic galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices 49*, 4 (2014), 499–512.

[37] OKASAKI, C., AND GILL, A. Fast mergeable integer maps. In *Workshop on ML* (1998), pp. 77–86.

[38] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[39] PENG, I., MCFADDEN, M., GREEN, E., IWABUCHI, K., WU, K., LI, D., PEARCE, R., AND GOKHALE, M. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2019), IEEE, pp. 71–78.

[40] POUTIEVSKI, L., MASHAYEKHI, O., ONG, J., SINGH, A., TARIQ, M., WANG, R., ZHANG, J., BEAUREGARD, V., CONNER, P., GRIBBLE, S., ET AL. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 66–85.

[41] PROKOPEC, A., BRONSON, N. G., BAGWELL, P., AND ODERSKY, M. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), pp. 151–160.

[42] PSARAS, Y., AND DIAS, D. The interplanetary file system and the filecoin network. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)* (2020), IEEE, pp. 80–80.

[43] RANDAZZO, A., AND TINNIRELLO, I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), IEEE, pp. 209–214.

[44] RAVICHANDRAN, K., GAVRILOVSKA, A., AND PANDE, S. Destm: harnessing determinism in stms for application development. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), pp. 213–224.

[45] REISS, C., WILKES, J., AND HELLERSTEIN, J. L. Google cluster-usage traces: format+ schema. *Google Inc., White Paper 1* (2011).

[46] RUAN, P., LOGHIN, D., TA, Q.-T., ZHANG, M., CHEN, G., AND OOI, B. C. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 543–557.

[47] SAAD, M. M., KISHI, M. J., JING, S., HANS, S., AND PALMIERI, R. Processing transactions in a predefined order. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019), pp. 120–132.

[48] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (1990), 299–319.

[49] SHAHRAD, M., FONSECA, R., GOIRI, Í., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 205–218.

[50] SHILLAKER, S., AND PIETZUCH, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 419–433.

[51] SILVA, P., FIREMAN, D., AND PEREIRA, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[52] SURI-PAYER, F., BURKE, M., WANG, Z., ZHANG, Y., ALVISI, L., AND CROOKS, N. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 1–17.

[53] THE DFINITY TEAM. The Internet Computer for Geeks. Cryptology ePrint Archive, Paper 2022/087. https://eprint.iacr.org/2022/087.

[54] USTIUGOV, D., AMARIUCAI, T., AND GROT, B. Analyzing tail latency in serverless clouds with stellar. In *2021 IEEE International Symposium on Workload Characterization (IISWC)* (2021), IEEE, pp. 51–62.

[55] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 133–146.

[56] WEYL, E. G., OHLHAVER, P., AND BUTERIN, V. Decentralized society: Finding web3's soul. *Available at SSRN 4105763* (2022).

[57] ZAMANI, M., MOVAHEDI, M., AND RAYKOVA, M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 931–948.

[58] ZHANG, T., XIE, D., LI, F., AND STUTSMAN, R. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 1–12.

A Appendix – Scheduler Analysis

Notation For a vector $\mathbf{v}$ we write $\mathbf{v}_j$ for the j'th entry in $\mathbf{v}$. By overloading notation, we write $\mathbf{e}_j$ for the unit vector with 1 on the j'th position and 0 everywhere else. We write $|\mathbf{v}|$ for $\sum_j \mathbf{v}_j$ – for vectors with positive entries $|\cdot|$ corresponds to $|\cdot|_1$.

For a set S we write $|S|$ for its size. For a real number x we write $|x|$ for its absolute value. We write $x \leftarrow a$ for assigning to variable x value a. If S is a set, we write $x \leftarrow S$ for a deterministic way of assigning a value $s \in S$ to variable x.

Problem statement An allocation for t canisters is represented by a vector $\mathbf{a} = (a_1, a_2, \ldots, a_t)$, a vector in $\{1,2,\ldots,v\}^t$ for some v. Given an allocation vector $\mathbf{a} = (a_1, a_2, \ldots, a_t)$, we define a deterministic (stateful) scheduling algorithm which, for each round $k \in \mathbb{N}$ outputs some index $sch(k) \in \{1,2,\ldots,t\}$, or equivalently, some unit vector $\mathbf{e}_{j^*}$ (with $j^* \in \{1,2,\ldots,t\}$)[1].

Intuitively, a good scheduler approximates the allocation vector well, i.e. for large enough k it outputs j a number of times proportional to its allocation. We formalize this intuition as follows.

[1] To simplify notation, we ignore that the scheduler is stateful

For each $k \in \mathbb{N}$ and $i \in \{1,2,\ldots,t\}$ let $idxs(k,i) = \{j \mid j \leq k, sch(j) = i\}$ the set of indexes j such that i was scheduled in round j (i.e. $s(j) = i$). For a good scheduler, the quantity

$$\left| \frac{|idxs(k,i)|}{k} - \frac{a_i}{|a|} \right|$$

is "small" for all large enough k. Informally, the above relation states that the scheduler has allocated k rounds proportional to the desired allocation.

Scheduler description Given an allocation vector $\mathbf{a}$, we define the following scheduler. The state of the scheduler at step k is given by vectors $\mathbf{d}(k), \mathbf{p}(k), \mathbf{s}(k)$ defined as follows[2].

The initial state is $\mathbf{d}(0) = \mathbf{s}(0) = (0,0,\ldots,0)$. For $k \geq 1$ define:

$$\begin{cases} \mathbf{p}(k) = \mathbf{d}(k-1) + \mathbf{a} \\ j^* \leftarrow \{j \mid \mathbf{p}_j(k) \geq \mathbf{p}_l(k),\ \forall l \in \{1,2,\ldots,t\}\} \\ \mathbf{s}(k) = \mathbf{s}(k-1) + \mathbf{e}_{j^*}, \\ \mathbf{d}(k) = \mathbf{p}(k) - \mathbf{e}_{j^*} \cdot |\mathbf{a}| \\ sch(k) = \mathbf{e}_{j^*} \end{cases}$$

Analysis The following lemma states some invariants that hold throughout the execution of the scheduler.

Lemma A.1. *For any $k \in \mathbb{N}$ it holds that:*

$$|\mathbf{d}(k)| = 0$$

For any $k \in \mathbb{N}^*$ it holds that:

$$|\mathbf{p}| = |\mathbf{a}|$$

Proof. We prove the invariant holds true for $\mathbf{d}$ by induction on k. The invariant for $\mathbf{p}$ follows immediately.

Base case For $k = 0$ we have that $\mathbf{d}(0) = (0,0,\ldots,0)$ so the invariant holds trivially.

Induction step By definition,

$$\mathbf{d}(k) = \mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}|$$

for some $j^* \in \{1,2,\ldots,t\}$. We then get that:

$$|\mathbf{d}(k)| = \left|\mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*}\right| = 0 + |\mathbf{a}| - |\mathbf{a}| = 0$$

□

The following lemma establishes a relation between $\mathbf{d}$ and $\mathbf{s}$. Informally, the relation says that $\mathbf{s}(k)$ is an approximation of $k \cdot \frac{\mathbf{a}}{|\mathbf{a}|}$ – the quality of the approximation is given by the entries in $\mathbf{d}$.

Lemma A.2. *For any $k \in \mathbb{N}$ it holds that:*

$$\mathbf{d}(k) = k \cdot \mathbf{a} - \mathbf{s}(k) \cdot |\mathbf{a}|$$

Proof. Proof by induction.

[2] In fact $\mathbf{p}$ is explicitly maintained only for convenience of analysis; it can be reconstructed from $\mathbf{d}, \mathbf{s}$ and $\mathbf{a}$

Base case For $k = 0$ we have that $\mathbf{d}(k) = \mathbf{s}(k) = (0, 0, \ldots, 0)$ so the equality holds trivially.

Induction step Assume it holds for $k-1$, i.e. $\mathbf{d}(k-1) = (k-1) \cdot \mathbf{a} - \mathbf{s}(k-1) \cdot |\mathbf{a}|$. By definition, $\mathbf{d}(k) = \mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}|$ for some j^*. From the induction step, this can be rewritten as

$$\begin{aligned} \mathbf{d}(k) &= (k-1) \cdot \mathbf{a} - \mathbf{s}(k-1) \cdot |\mathbf{a}| + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}| \\ &= k \cdot \mathbf{a} - \mathbf{s}(k) \cdot |\mathbf{a}| \end{aligned}$$

□

The following lemma establishes a lower bound on the the debt which processes can accumulate throughout their lifetime.

Lemma A.3. *For any $k \in \mathbb{N}$ and $j \in \{1, 2, \ldots, t\}$ it holds that:*

$$1 - |\mathbf{a}| \leq \mathbf{d}_j(k)$$

Proof. Proof by induction.

Inductive step The inequality trivially holds for $j \neq j^*$ since by definition (and the induction hypothesis):

$$\mathbf{d}_j(k) = \mathbf{d}_j(k-1) + \mathbf{a}_j \geq \mathbf{d}_j(k-1) \geq 1 - |\mathbf{a}|$$

To prove the bound for j^*, notice that by Lemma A.1, for any $k \in \mathbb{N}$:

$$\sum_{j=1}^{t} \mathbf{d}_j(k) + \mathbf{a}_j(k) = |\mathbf{a}|$$

Since j^* is such that $\mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} \geq \mathbf{d}_j(k-1) + \mathbf{a}_j$ for all j and $t \leq |\mathbf{a}|$, it holds that

$$\mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} \geq \frac{|\mathbf{a}|}{t} \geq 1.$$

So, we have that

$$\begin{aligned} \mathbf{d}_{j^*}(k) &= \mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} - (\mathbf{e}_{j^*} \cdot |\mathbf{a}|)_{j^*} \\ &\geq 1 - |\mathbf{a}| \end{aligned}$$

□

Finally, the following theorem establishes that in $|\mathbf{a}|$ rounds, job i is scheduled $\mathbf{a}_i$ times.

Theorem A.4.

$$\mathbf{s}(|\mathbf{a}|) = \mathbf{a}$$

Proof. By Lemma A.2, it holds that:

$$\mathbf{d}(|\mathbf{a}|) = |\mathbf{a}| \cdot \mathbf{a} - \mathbf{s}(|\mathbf{a}|) \cdot |\mathbf{a}| = (\mathbf{a} - \mathbf{s}(|\mathbf{a}|)) \cdot |\mathbf{a}|$$

Since $\mathbf{d}_{j^*} \geq 1 - |\mathbf{a}|$ (by Lemma A.3) and $\mathbf{d}_j(|\mathbf{a}|)$ is an integer divisible by $|\mathbf{a}|$ (by the above equality) then, it holds that $\mathbf{d}_j(k) \geq 0$. Since $|\mathbf{d}(|\mathbf{a}|)| = 0$ then $\mathbf{d}(|\mathbf{a}|) = (0, 0, \ldots, 0)$ and the desired equality follows.

□