MultiVAC Programming Purplepaper

Flexible Sharding Supporting Turing-Complete Smart Contracts

The MultiVAC Foundation

May of 2019, Version 0.1

**This purplepaper is produced by the MultiVAC technical team under the direction of Dr. Xiang Ying (email: shawn@mtv.ac). We wish to thank Dr. Ge Li, Zhong Dong, Nan Lin, Liang He, Shuzhi Huang, Libo Shen and Zhengyuan Ma for their contributions.*

1 Introduction

Blockchain technology driven by sharding will soon redefine the traditional Von Neumann computer architecture. Blockchains are the next generation of decentralized world computers: a shared computing resource which unlike supercomputers do not belong to any individual. For this reason, blockchain's realization depends on innovating new architectures to break through throughput constraints, fashioning new transmission strategies to streamline data flow, and pioneering new types of decentralization to ensure equality. It is imperative for sharding programs at the forefront of blockchain innovation to transcend blockchain 2.0's narrow view of sharding, dividing the world only into network sharding, transaction sharding and state sharding. This categorization ignores the fact that blockchain sharding is entering a period not of incremental improvement but of continuous revolution.

As a next-generation blockchain platform, MultiVAC designs a parallelized all-dimensional sharding architecture, in effect upgrading the world computer from a single-core to a multi-core system. Our sharding module allows blockchains to scale processing power with size, making it an ideal platform for smart contract developers targeting widespread adoption. MultiVAC's low miner threshold ensures that ordinary users can freely join the network so that control of the system is always in the hands of the ordinary user, and our Turing-complete LLVM development platform gives developers the flexibility and freedom to realize all types of complicated business logic. MultiVAC was developed with a vision to rebuild the "One-CPU-One-Vote" concept that Satoshi Nakamoto pioneered in Bitcoin, undermined by professional ASIC miners in the latter half decade. Our team firmly believes that this original vision will be rebuilt, stronger than ever, on MultiVAC.

1.1 The Decentralization and Latency Tradeoff

Imagine two systems: On the one hand, imagine an absolutely centralized supercomputer with infinite vertical expansion, handling all of a user's computation and storage. This system realizes the most efficient processing possible but also requires every user to fully trust the supercomputer operator. On the other hand, imagine a completely decentralized computer system, such that each participating computer owns their own data and is responsible for processing part of a larger program. Nodes now no longer have to trust in a centralized authority,ev yet since all nodes now depend on each other they need to constantly transmit instructions and data though the anonymous network. Such a system is very inefficient.

In a similar fashion, de-centralization and low latency are two opposite extremes in the field of distributed computing which covers all blockchain systems. An effective blockchain system is thus dependent on wise technical tradeoffs: the pursuit of pure speed should not be considered an end goal in itself as it leads to centralization and the pursuit of decentralization for decentralization's sake should not be considered an end goal in itself because it leads to system inefficiencies.

The optimal tradeoff level is determined by the original purpose of blockchain decentralization: to provide democratic operating mechanisms that all participants contribute to, enabling community governance for all. Blockchain is the pursuit of an equal and transparent digital society, a society which allows complex and creative ideas to flourish atop a framework that is expressive, efficient, and enabling. The ideal system would thus enable the tradeoff to be determined to the highest extent possible by its developers, not inhibiting smart contract developers in in terms of programming language, development support or network design.

From the very inception, the MultiVAC has been designed to model this ideal network, effectively achieving high throughput and low latency while allowing smart contract developers maximum flexibility over network processing. MultiVAC is committed to creating a robust and supportive developer environment supporting a rich variety of community applications without ever sacrificing blockchain's core value of decentralization. MultiVAC resolutely makes every effort to minimize the participation threshold in the blockchain network for ordinary users in a secure and decentralized way, adhering to the original blockchain vision of producing a world computational community held in common by all, a global society of the people.

1.2 Present Challenges for Public Blockchains

MultiVAC's design addresses four challenges currently faced by public blockchains supporting smart contracts:

1. ***Throughput constraints*.** Most public blockchains involve consensus methods that mire them in low throughput, the most typical of which is Proof of Work. In Proof of Work, every transaction needs to have a network-wide validation, making the whole network's processing power constrained to that of a single node. Not only is Proof of Work inefficient, it is also dependent on unproductive Hash computations that result in energy waste.

2. ***Storage and transmission bottlenecks.*** An enduring open problem in blockchain design is the design of a blockchain's storage system. Current systems require that every node in the whole network store the entire network's transaction log. The longer such a system runs, the larger the logs become, requiring larger storage and transmission capabilities for system miners. For this reason, the whole network will inevitably move towards professional miner centralization, both limiting the blockchain's performance and depriving ordinary users of the right to participate.
3. ***Rigid and Constrained Development.*** Some public blockchain systems abandon Turing Completeness in order to simplify system design, resulting in serious limitations in compatible business logic. For example, some systems force developers to use the MapReduce framework or another subset of modern programming languages, sometimes missing common abstractions such as the object-oriented framework. Developers must not only learn a new language but also face severe limitations when they do, creating a detrimental environment for application development.
4. ***Violation of decentralization.*** As mentioned earlier, a blockchain system should be equal, transparent, and decentralized. In order to improve a blockchain network's throughput, some schemes choose to sacrifice decentralization and adopt the usage of super-nodes, casting aside blockchain's original core values of decentralization and equality. MultiVAC considers this unacceptable as it is fundamentally opposed to blockchain's ethos.

1.3 Summary of this Paper

In this presentation, we present MultiVAC's smart contract sharding solution which addresses the aforementioned challenges. MultiVAC's design has several main features:

- **MultiVAC is the first blockchain sharding technology that achieves linear expansion without significant performance reductions.** MultiVAC's asynchronous sharding architecture provides the basis for building large, high-volume blockchain networks at scale.
- **MultiVAC is a Turing-complete blockchain sharding scheme.** It provides a flexible programming framework for developers to build complex applications without giving up any of the functionality available in a modern programming language.
- **MultiVAC maintains a low equipment barrier for network participation**, ensuring decentralization and continuous control of the network by ordinary miners.
- MultiVAC provides base-layer infrastructure to address the problem of blockchain scalability at the fundamental level. **On top of this infrastructure, MultiVAC's modular design allows for the addition of myriad further extensions.**

We present our design below in seven parts. In Chapter 2, we summarize current problems facing blockchain systems and how MultiVAC addresses them. In Chapter 3, we discuss smart contracts in MultiVAC and how they can be crafted and deployed. In Chapter 4, we discuss MultiVAC's smart contract execution in technical detail, focusing on modes of data management. In Chapter 5, we discuss how MultiVAC handles cross-shard interactions as the central question in blockchain sharding. In Chapter 6, we review how MultiVAC's smart contract system interacts with MultiVAC's storage, execution, and consensus modules, and we bring the presentation to a conclusion in Chapter 7.

2 Overview of MultiVAC Technology

MultiVAC builds a novel sharding architecture to address today's fundamental blockchain needs. The most pressing problem in contemporary blockchain design is how to scale up low transaction speed and throughput. Many solutions have been proposed for the throughput issue without meeting with full success. A summary of such solutions is presented below.

- Supernodes: By handing over the power of consensus to a few nodes with super-high performance, the usage of supernodes dramatically improves transaction throughput. However, this kind of system is essentially a reversion back into pseudo-centralization, giving the supernode oligopoly full control over the system at the expense of ordinary users, subverting the essence of blockchain decentralization.
- Off-Chain Scalability: Including multi-chains, side chains, lightning networks and state channels, off-chain scalability solutions settle transactions off the main blockchain and can increase throughput by orders of magnitude. However, these solutions patch the problem but do not get at its heart, moving transaction settlement off of the secure main blockchain. Indeed, such solutions actually introduce new security risks, resulting in a less-than-ideal effect.
- On-Chain Scalability: On-chain scalability solutions include Directed Acyclic Graphs (DAG) and Sharding, involving the redesign of the blockchain structure itself. Directed acyclic graphs have seen rapid operation in laboratory environments but suffer due to transmission overhead from network storms in real-life operation. It is sharding solutions which show the most promise. Sharding has been the focus of scalability efforts of the most well-known blockchain projects such as Dfinity and Ethereum, and is inspired by a proven and effective scalability solution used in distributed systems. It involves splitting up the blockchain into multiple units in a decentralized and scalable fashion.

2.1 Definition of Blockchain Sharding

MultiVAC focuses on blockchain sharding as the center of our scalability efforts. We define blockchain sharding as the division of a blockchain network into a series of sub-networks, each of which maintains a separate blockchain that runs consensus. Because sharding allows for parallel execution of the consensus algorithm, it improves network throughput almost without limit. To increase the throughput of the network, the network may simply add more shards.

However, optimal sharding for smart-contract enabled blockchains is still an open problem. This is largely due to difficulties involving data reliability and data consistency.

2.2 Current Issues in Blockchain Sharding

Difficulty in blockchain sharding comes from two sources. The first is data reliability. When data is produced on any shard, it must go through a consensus process in order to be deemed reliable. Designing a robust consensus mechanism that works across many shards is the challenge of data reliability. The second is data consistency. Even if all shard data is reliably confirmed through consensus, other shards must receive this data and update it in the appropriate way. Turing-complete smart contracts require updates in a particular order, resulting in messy cross-shard synchronizations. Below we discuss MultiVAC's solutions to these challenges.

2.2.1 Data Reliability and Consensus

Blockchains are decentralized systems owned by the whole network and require data reliability guarantees to a much higher extent than traditional single-party distributed systems. In a decentralized blockchain system there are no trust relationships between nodes, so the network must always be wary about forgers and fraudsters. This is the issue addressed in data reliability. Data reliability is fundamentally solved by blockchain consensus and the design of a robust consensus mechanism across shards to select validating miners.

Sharding distributes blockchain consensus into sub-consensus processes, all of which are confirmed by miners in their respective shards. This creates several difficulties for consensus. For one thing, the overall consensus security in a sharded blockchain is the security level of its weakest shard. This must be kept at a high level to ensure secure operation. For another, as shards enable large-scale expansion of system performance each shard must process an ever-larger volume of state information, putting immense system pressure on confirming miners. This must be addressed to avoid unrealistic network requirements as the network grows in size.

Some sharding projects randomize the selection of block nodes, using fast and random switching of block production rights to improve security. However, this solution is difficult to achieve with smart contracts because of the increased difficulty of transferring all relevant smart contract data compared to a block production right. Another naive solution is for all miners to save all the data. This brings a huge storage pressure to each miner. Even assuming that all miners have a high storage endowment, each block needs to be broadcast and synchronized with all nodes in the network, creating a huge transmission demand. Blockchains that require such high demands in storage and transmission inevitably become more and more centralized, distorting the original intention of blockchain systems.

MultiVAC's unique sharding solution seeks to address the above problems with a sharding design that innovates over traditional sharding schemes. We describe its features below:

- **Separation of data control and storage responsibilities.** MultiVAC divides network nodes into miner nodes and storage nodes. Storage nodes store transaction state while miner nodes only need to save a summary of the current state in order to verify data and confirm transactions. Storage nodes are clients of miner nodes and are completely impotent to misbehave; if any data is modified or deleted without authorization the storage node will be immediately ignored by all miner nodes.
- **Data interaction based on trusted proof.** MultiVAC has designed a delicate distributed data structure that enables miners to retain only summary information but perform full verification, ensuring the accuracy of inter-node communication while keeping miner demands to a minimum. This enables for compact communication and safe incremental updates.
- **All-dimensional computation, storage and transmission sharding.** MultiVAC not only conducts computational sharding but also shard storage and shard transmission. Sharded storage nodes need only to store relevant transaction data that will be requested by nodes in that shard, ensuring that network transmission is also kept inside of the shard. This allows for a great reduction in a blockchain’s size and transmission cost. With this insight, it becomes possible for blockchain throughput to increase linearly with the number of shards while reducing the threshold of blockchain participation to ordinary miners.

2.2.2 Data Consistency

The second difficulty in designing a sharded blockchain is data consistency. Turing-complete smart contracts are blockchain programs that can execute any logic that may be executable on a normal computer. Such contracts require consistency in transaction order, which may be processed differently on different shards. Each shard must confirm their transactions asynchronously, so the system has to keep track of the original order in which the transactions were made, and enforcing this order remains an enduring problem. Overly strong enforcement creates impossible transactions demands while overly weak enforcement undermines the blockchain's ability to process smart contracts. The challenges around data consistency include:

- Ensuring consistency in cross-shard data modification.
- Avoiding performance degradations resulting from over-reliance on global data.

2.2.2.1 Cross-Shard Modification Consistency

In distributed systems, consistency requirements fall on a spectrum between strong and weak.

- Some systems exhibit stronger forms of consistency. In the strongest case, every time a transaction is sent across shards its transaction amount would be locked until a receipt message is obtained from the receiving shard. This ensures that the order of every transaction is consistent but is impractical even if the granularity of locking is extremely small. In all strongly consistent systems, system performance declines sharply due to locking and transaction congestion.
- Other systems choose weaker forms consistency. These may split cross-shard transactions into sub-transactions: for example, a transaction from shard A to B would be split into a deduction in shard A and a remittance to shard B, with the deduction executed by miners of shard A and the remittance executed by miners of shard B. Two drawbacks of such a weakly consistent system are that remittances may be lost and that execution order in shards A and B may not be the same. The first issue is be addressed by repeatedly sending remittances, but the more difficult second issue requires that the execution order of all transactions be changeable after execution in shard B.

Despite the impracticality of strongly consistent networks, without strong forms of consistency it is difficult to design a smart contract system in which operations may depend on each other. For this reason, some projects remove the problem of execution order completely by constraining the virtual machine instruction set, abandoning Turing completeness and only allowing the execution of instructions that do not depend on execution order. This creates severe limitations on the types of applications that can be developed.

MultiVAC chooses a compromise between strong and weak consistency and designs a system for *monotonic consistency*, which is strong enough to ensure transaction ordering without requiring the locking of any transaction data. This system thus guarantees both Turing-completeness and high performance, and consists of three parts:

1. **MultiVAC nodes host a virtual machine supporting an unconstrained Turing-complete instruction set,** allowing for all types of development logic.

2. **MultiVAC divides cross-shard merge operations into two types according to consistency requirements:** Cross-shard operations with lower consistency requirements are allowed to update quickly without execution ordering and cross-shard operations with higher consistency requirements are required to be ordered for execution. Developers have the option to choose one or the other according to their business requirements.
3. **MultiVAC supports asynchronous programming for developers,** in which data does not interact and is not shared among shards, as a solution for users to who wish to avoid synchronization latency when possible.

2.2.2.2 Dependence on Global Data

A second aspect of data consistency problem is the over-dependence on global data. Global data requires that all shards utilize one particular piece of data, which is a difficult demand in parallel processing. How to design a sharding system for the presence of global data is a difficult problem for all blockchain systems.

Take the example of an e-commerce smart contract, in which a purchaser buys a limited-stock good from a supplier. This smart contract must record the remaining amount of stock of every item to prevent out-of-stock cases, but because the remaining stock amount is global data it must be read and written every time the smart contract is invoked from any shard. Several naive solutions include locking the data so that reads and writes only occur in one shard at a time and limiting permissions to allow global data invocation only on some particular shards. Clearly, both schemes result in throughput constraints. Instead, MultiVAC chooses the following:

1. MultiVAC introduces shard-level data to be used in place of global data whenever possible: Instead of one universal variable to store the global system state, it is almost always sufficient to maintain one state variable per shard that comprise the global state in aggregate. In the above e-commerce example, the total stock amount can be divided into stock allocated to individual shards. Although the quantity will have to be rebalanced from time to time, for the vast majority of time the system will not need to check more than one shard when determining whether or not an order exceeds the in-stock quantity.
2. MultiVAC allows users to operate on all shards, enabling them to make best use of distributed shard data. In the e-commerce smart contract, as the in-stock quota of some shards will be exhausted faster than that of others, MultiVAC allows users to initiate purchases from all shards. When the total amount on a shard is exhausted, users can choose to run the smart contract in other shards without waiting for the smart contract manager to rebalance quotas across shards.
3. MultiVAC allows for the usage of universal global data when sharded global data does not suffice. This data is stored and maintained by a single shard but its usage is avoidable in most cases.

MultiVAC's **decentralized design for global data** aims to maximize the performance of blockchain sharding so as to allow each shard to be to run as independently as possible, thus reducing limitations to scalability.

2.3 General Architecture of MultiVAC Smart Contracts

Motivated by the above considerations, we present MultiVAC's general smart contract processing system. MultiVAC's design philosophy is functional modularization for iterability and scalability; We plan the system in modules so that each module can flexibly integrate new technologies, keeping pace with the times. MultiVAC divides network responsibilities into the **storage module, consensus module, and execution module,** with responsibility for each module handled by MultiVAC's two node types: **miner nodes** and **storage nodes**. The functional separation of execution and storage between MultiVAC's two node types greatly reduces the hardware burden on ordinary miners while ensuring security. Storage nodes by themselves have no voting rights and do not invoke consensus, so they pose no risk of centralization.

A high-level overview of MultiVAC's smart contract system is given below. The elaboration of this system will be the subject of the rest of this paper.

- ***Client module.*** The Client module is composed of user nodes. User nodes are the interfaces used by network clients to interact with the network. The Client module interacts with human users and sends transactions to the Shard module.
- ***Shard module.*** The Shard module manages the MultiVAC network's shard system. Each shard maintains a separate blockchain with independent storage and contains two types of network nodes: miner nodes and storage nodes. Miner nodes operate the execution module and consensus module of every shard as sub-modules and storage nodes operate the shard's storage module as a sub-module. In addition to these sub-modules, the shard module also asynchronously manages cross-shard data transfer.
    - ***Storage module.*** The Storage module is run by storage nodes that maintain a full-state database. Miner nodes each hold a very concise summary of the database and rely on the summary to verify the data's authenticity.
    - ***Consensus module.*** The Consensus module is run by miner nodes and seeks to achieve consensus about a new block's transactions. The Consensus module uses an efficient Byzantine fault-tolerant algorithm to determine whether or not miners have come to agreement.
    - ***Execution module.*** The Execution module is a Turing-complete virtual machine responsible for executing smart contract transactions, packaging their outputs into newly proposed blocks. When a smart contract is executed, a miner node obtains the smart contract's code from a storage node, runs the smart contract on its execution module and returns the results to the storage node.

The full system can be illustrated as follows:

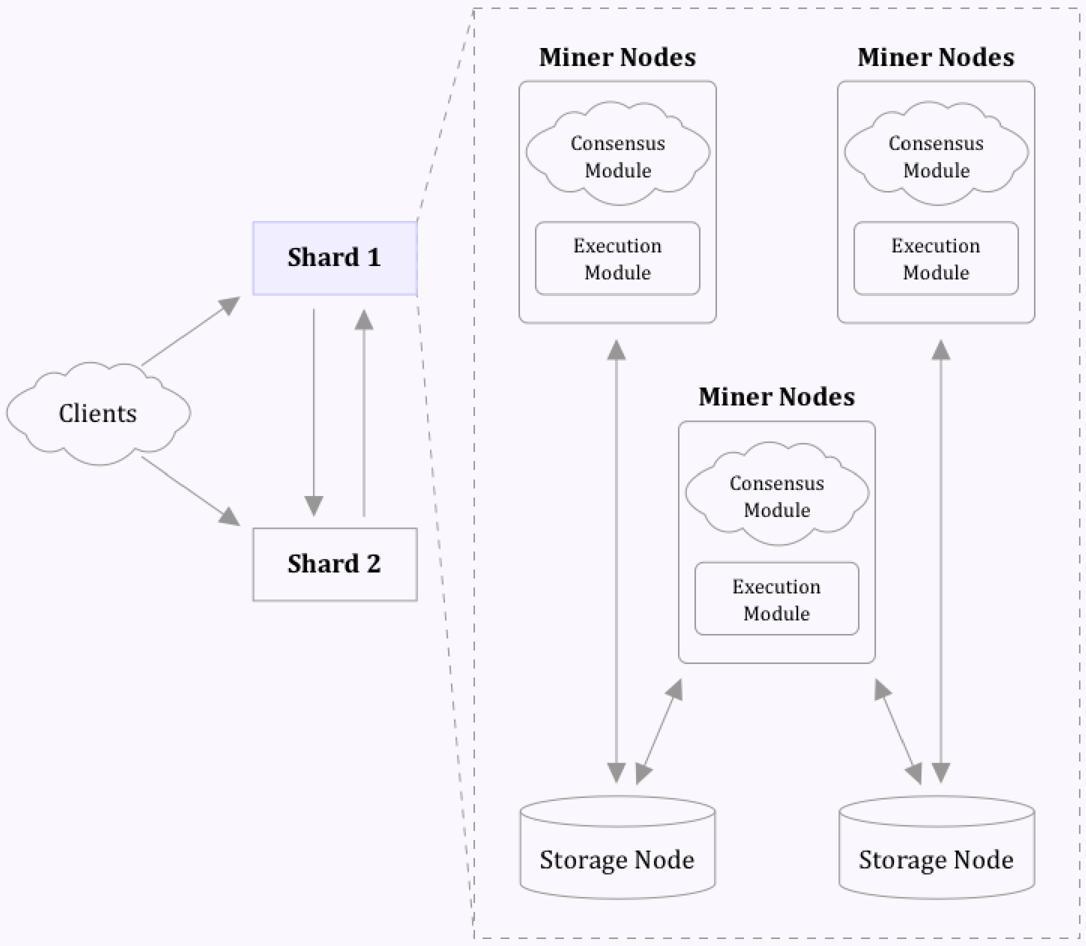

Figure 1: MultiVAC Overall System Design.

3 Deployment and Invocation of Smart Contracts

Unlike traditional smart contract systems, MultiVAC smart contracts must consider the existence of shards. In MultiVAC's shard design, code for all smart contracts are deployed on every shard. This design decision was made to allow the network's computational resources to be used wherever most available, avoiding network congestion and resource waste. Upon registration, MultiVAC users receive a public-private key pair. Users may call smart contracts on any shard, using their key pair to send a transaction signaling the smart contract's address with the corresponding shard number. This contract call is relayed to the corresponding shard's storage nodes, which supply necessary data for execution to the corresponding shard's miner nodes.

3.1 Sharding of User Transactions

As mentioned earlier, MultiVAC allows users to call smart contracts from any shard. This design choice has the following advantages:

- It diverts users from congregating and congesting any particular shard.

- It encourages users to conduct in-shard transactions.

3.1.1 Preventing User Congregation

MultiVAC chooses not to assign users to shards. If each user was constrained to calling smart contracts only in an assigned shard, there may be a smart contract with many users calls in one shard and too few calls in another. This imbalance of user demand creates backlog and latency in some shards and relative idleness in others.

3.1.2 Encouraging Users to Conduct In-Shard Transactions

Consider a random normal transaction from a sender shard to a recipient shard. If this transaction was fully random then cross-shard transactions should asymptotically approach 100% as the number of shards increases:

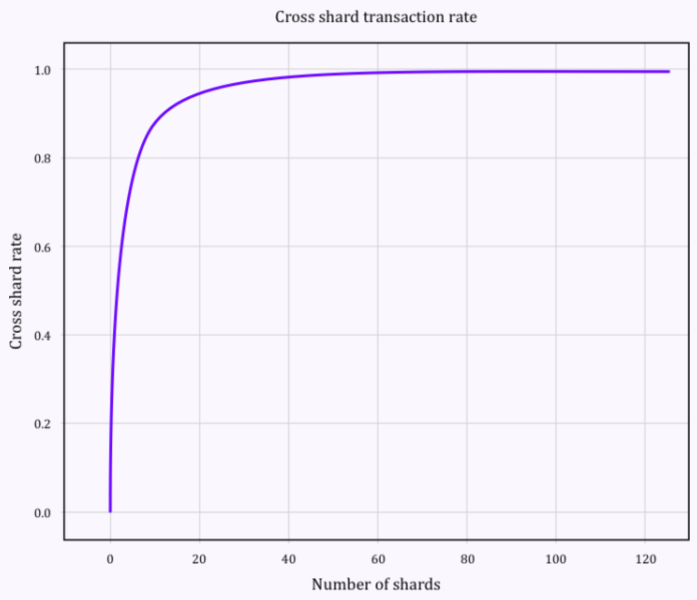

Figure 2: The relationship between shard count and cross-shard transaction rate.

Each cross-shard transaction requires a miner to send newly generated data outputs to the recipient's shard, which must then be merged into the recipient shard's Main Merkle Tree before they can be used. In contrast, in-shard transactions can be used by the payee as soon as it is confirmed by an in-shard miner. For this reason, cross-shard transactions increase transaction latency significantly over in-shard transactions.

From a user's point of view, not only do cross-shard transactions increase latency, they will also result in higher gas usage costs. Allowing users to have an account in all shards effectively encourages users to minimize their costs and network latency by choosing to conduct more in-shard transactions, thus organically reducing cross-shard interaction load.

3.2 Smart Contract Deployment

A smart contract in MultiVAC comprises both code and data, both of which are stored in MultiVAC's DataCell units. Code and data for all smart contracts are duplicated on all shards in order to make optimal use of parallelism and scalability except for the case of universal global data which is only stored on the *main shard* from which the smart contract was originally deployed.

When a user wishes to deploy a new smart contract, he initiates a transaction to a built-in *deploy* function on any main shard *i*. When this transaction is confirmed by *i*'s miners, they generate the new smart contract address and assign it to the smart contract. The contract is then initialized as follows:

- Miners of every shard create a shard DataCell to save the smart contract's code;
- Miners of every shard create a shard DataCell to initialize shard data according to the contract's logic;
- If applicable, Miners in the main shard *i* create a global DataCell in shard *i* to initialize global data according to the contract's logic.

Once the *deploy* transaction is confirmed by every shard's miners and the corresponding initialization data is stored in every shard's storage node, the smart contract is ready to be called. Since transactions that call global data must run on the main shard, we provide an operation to reset the main shard and migrate global data if the original main shard becomes too congested, evenly distributing traffic across shards.

3.3 Smart Contract Invocation

Users invoke smart contracts by sending transactions. We define a **Transaction** as a message from a user to a storage node to call a smart contract. Transactions contain the following information:

- The user (sender) address,
- The recipient (smart contract) address,
- The shard number of the transaction call,
- The function(s) from the smart contract to be invoked, and the Merkle Tree index of any parameters or data required by such a function.

When the corresponding shard's storage nodes receive the user's Transaction, they relay necessary data with their Merkle Path to the shard's miner nodes. The miner nodes use the Merkle Path to check the data's authenticity and subsequently executes the smart contract on the virtual machine in its execution module. The miner writes the results to a block and finally joins the new block to the shard's blockchain if consensus is reached through the miner's consensus module.

MultiVAC processes smart contract outputs differently depending on data type and consistency requirement. We turn to the different types of smart contract outputs processed in MultiVAC in the following section.

4 Smart Contract Data Models

This section introduces the models used in MultiVAC for data storage and transaction processing. This and further sections depend on knowledge of the Merkle Tree data structure introduced in MultiVAC's Yellow Paper, to which the reader is recommended for technical details.

In order to describe MultiVAC's data models, we introduce a running example. Suppose we design a smart contract for a token system on the MultiVAC platform, hereinafter referred to as XYZ. The XYZ smart contract provides the necessary operations for establishing the total number of XYZ tokens, for converting XYZ tokens to and from MTV tokens, and for transferring XYZ tokens between users. We use the XYZ system to illustrate design principles behind MultiVAC's data model.

4.1 Introduction

Nearly all smart contracts depend on state information saved to the blockchain. Without state information, a smart contract can either only perform very limited operations or must continuously obtain data from nodes who may not be trusted by the rest of the network.

State processing is simple in unsharded smart contract systems including the original Ethereum, because every transaction is executed by every miner. Therefore, so long as miners reach consensus on which blocks have been generated, the state reached after the transaction will be consistent without additional processing. In a sharded system, however, miners must deal with state changes from transactions in all shards. This requires a complex state synchronization mechanism.

MultiVAC stores state data in JSON format using data units called DataCells. Each DataCell belongs to a specific smart contract, and the set of all DataCells in the smart contract constitute the total state of the smart contract. Taking the XYZ example, the total state is constituted by the remaining XYZ quota and the number of XYZ tokens held by each user in the network. When the smart contract is executed, DataCells are added, deleted or modified in the state and internal consistency of all DataCells in the system are guaranteed through the state synchronization mechanism.

DataCells from all smart contracts are combined in the Merkle Tree data structure and are packaged into blocks. These blocks undergo consensus which if successful are stored in a shard's storage node. When the data is subsequently needed, miners query the data from the storage node according to the data's Merkle Path. Miners first check the authenticity of returned data from the storage node against their own Merkle Root before using it.

A DataCell contains the following components, with the specific content and purpose of each component discussed below:

- The DataCell's **smart contract address**
- The DataCell's **user address**
- The DataCell's **type**: Account Type or UTXO Type
- The DataCell's **scope**
- The DataCell's **data**
- The DataCell's **shard**

4.2 DataCell Addresses: Smart Contract and User

Each DataCell consists of two addresses: the smart contract address and the user address. These two determine which users and smart contracts may modify the DataCell.

In the XYZ example, a user acquires some XYZ tokens by converting from MTV and generates a DataCell to record the tokens acquired in the exchange. The user address of the DataCell is the user's account address and the smart contract address of the DataCell is the address of the overall XYZ smart contract. These addresses limit spending of the DataCell contents to the user and limit modification of the DataCell contents to calls from the XYZ smart contract.

A DataCell's user address can also be set to null, allowing any user to modify its data. For example, the data representing remaining XYZ quota in any given shard is used by all users, so its DataCell address would be set to null. The mechanisms through which MultiVAC guarantees the consistency of this data is discussed in Section 5.

4.3 DataCell Types

Data storage models previously used in public blockchains include the account model and the UTXO model. Both have their advantages and disadvantages in different use cases. MultiVAC is the first blockchain design allowing data to be stored in both of these forms. This allows developers the flexibility to pick a suitable data type for all of their variables.

- An **Account-Type DataCell** is unique and cannot be deleted. Its location is fixed in the Merkle Tree and can only be modified in-place.
- A **UTXO-Type DataCell** can have multiple copies, each of which is deleted after use. UTXO transactions can be used one time as the input to a transaction. Its location is not fixed and needs to be indexed by the user.

Suppose an example of cross-shard communication in the XYZ smart contract, where user A transfers money in shard 1 to user B in shard 2. If the XYZ token data is Account-Type, then the transfer transaction needs to be split into two steps: deducting the value from user A's account and remitting the money to user B's account. To perform the latter step, storage nodes in shard A must also provide user B’s account information to miners, doubling network traffic costs and increasing transaction delays. Using the UTXO-Type DataCell, miners can simply update their Main Merkle Tree without these additional steps, greatly reducing the cost of cross-shard transfers.

In other cases when global or shard-level data is required, the Account-Type DataCell is a more suitable data structure because it can be easily indexed. In the XYZ smart contract, the remaining available amount of XYZ token in any given shard is shard-level data. This should be stored in an Account-Type DataCell because a UTXO-Type DataCell would make the data impossible to index, inviting the possibility of the data being produced multiple times or deleted.

All data in MultiVAC is one of the above two types. By selecting the appropriate data type, a developer may tailor network data-processing to requirements for his particular use case.

4.4 DataCell Scope

As in traditional distributed databases, data consistency is a vitally important consideration in blockchain systems. Due to different consistency requirements for different types of data, MultiVAC divides data into three categories according to scope. Users are guaranteed data consistency for all reads and writes in the same scope:

- **Global data**: Global data is data that may be read and written by any user in the whole network and whose scope is any shard in the whole network;
- **Shard data**: Shard data is data that belongs to a shard whose scope is that shard. It will be consistent for any user accessing the data through that shard.
- **Personal data**: Personal data is data that can be modified only by a single user. It is always consistent for that user.

Correspondingly in our XYZ example:

- **Global data** include data used to represent the remaining allowance for token sales, which is a system-wide number that needs to be called only in rare instances.
- **Shard data** include data representing the number of distributed but unowned tokens allocated to each shard.
- **Personal data** include data representing the number of tokens owned by each user.

While personal data can be Account-Type or UTXO-Type according to user needs, global and shard data must be of Account-Type. Global data only exists in one shard and the cases in which they are required should be relatively rare. Though over-dependence on global data is an old problem in all distributed systems, the usage of global data can be avoided in most cases. As is common practice, we encourage developers to parallelize function calls for maximum scalability. In the XYZ smart contract, the total amount of tokens in the system is stored as global data but is only rarely invoked; Instead, users interact with the amount of tokens available on any shard when they make XYZ transactions.

5 Smart Contract Cross-Shard Interaction

When dealing with cross-shard interaction, a sharded blockchain must consider a balance between data consistency and performance. Designs that achieve strong consistency are often accompanied by data lock-in and complex processes that drag down performance. Lower consistency requirements improve performance but may create problems in execution ordering and inconvenient programming demands for developers.

MultiVAC provides a flexible, free, and efficient cross-sharding scheme which includes two cross-shard interaction mechanisms: **Data Addition** for low-cost UTXO cross-sharding and **Data Modification** for high-cost Account cross-sharding. Developers may select from these two types according to their business needs.

The MultiVAC network is divided into several shards, each of which contain a group of miners and storage nodes. Each shard has independent storage and do not share any data with any other; Rather, shards interact through asynchronous messaging. Confirmed blocks contain a list of data modification instructions which are the outcomes of executed smart contracts. When miner completes a block, instructions relevant for any particular shard is sent to that shard for processing.

After each shard receives their relevant instructions for a block, these instructions are ordered if required and are sequentially executed to enforce monotonic data consistency across shards. In MultiVAC, cross-shard interaction occurs asynchronously, such that after sending a current batch of instructions a miner can immediately start to work on the next without waiting for a receipt message from other shards.

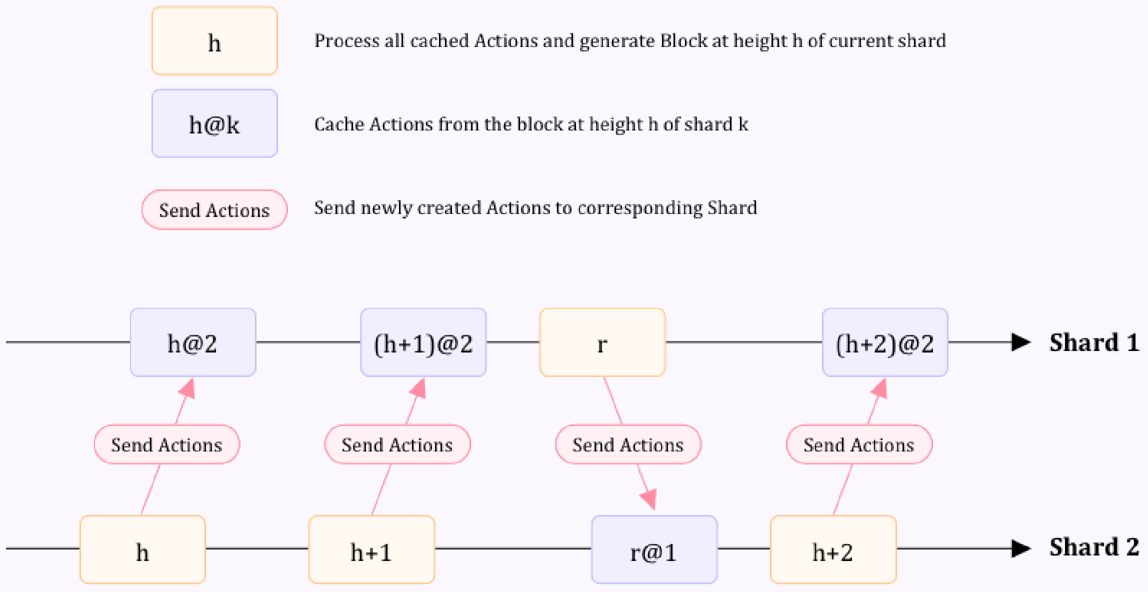
h
Process all cached Actions and generate Block at height h of current shard
h@k
Cache Actions from the block at height h of shard k
Send Actions
Send newly created Actions to corresponding Shard
h@2
(h+1)@2
r
(h+2)@2
Shard 1
Send Actions
Send Actions
Send Actions
Send Actions
h
h+1
r@1
h+2
Shard 2

Figure 3: MultiVAC's asynchronous block verification process.

5.1 Actions: Modifying Data

After miner nodes execute smart contracts in MultiVAC, they generate a list of execution outputs which must be added back to the global state. These outputs are called **actions** and are requests sent from miner nodes to storage nodes to add, delete, or update, or modify data based on computational results. Actions may be passed by a miner to storage nodes of its own shard or to storage nodes of other shards, and come in four main types:

- **Add**: Create a new DataCell. Add may be performed across shards.
- **Del**: Delete an existing DataCell. Del cannot be performed across shards and can only act on UTXO-type data.
- **Update**: Update existing data in an in-shard DataCell without changing access rights. Update cannot be performed across shards and can only act on Account-type data.
- **Merge**: Generate an Account-Type DataCell and send it to a recieving shard, which runs a developer-defined smart contract to change its state based on the DataCell contents. Since the logic of how to modify data across shards is differenct for every use case, developers are responsible for writing their own merge code to ensure computation according to their needs. Merge is performed across shards and is triggered to modify Account-type data in any other shard.

We distinguish an *action* from a *transaction*, which are requests from users to miners to trigger smart contract execution. Actions are generated from smart contract execution results and are relayed by miner nodes to storage nodes to update data in relevant shards. Because transactions are generated by the user, they may fail because of code errors or insufficient gas fees. In contrast, actions are generated by the system, and so long as the basic assumptions about MultiVAC's blockchain network are valid (e.g. the ratio of malicious nodes is less than 1/3), the generated actions of a valid smart contract will be executed successfully all the time.

5.2 Creation of Cross-Shard Actions

As mentioned above, MultiVAC supports four types of actions: add, del, update, and merge. Because del and update occur in-shard, their operation is straightforward: the miner node sends the action to its shard's storage node and the operation is completed. The operation of the two cross-shard actions, add and merge, are a little more complicated. **Add** is mainly used to add UTXO DataCells to other shards (although it can also be used to create a new Account-type DataCell in another shard) while **Merge** is mainly used to update Account-Type DataCells in another shard. Examples of the two operations in the XYZ token example are given below:

- When a user transfers XYZ tokens across shards, the system deletes some of his tokens in-shard and generates add actions, sending UTXO DataCells containing the payee's new tokens to the recipient shard. The transaction is completed when the new DataCells are added to the recipient shard's Main Merkle Tree.
- When the XYZ smart contract manager assigns each shard's token quotas, he or she triggers a transfer transaction which reduces the amount of unused tokens in the main shard and generates a merge action containing the transfer amount to the target shard. Because each shard's XYZ quota is Account-Type, miners in recipient shards must execute a manager-defined merge contract in order to incorporate the new quota into their existing data.

Every time a miner completes a new block, a set of add and merge actions are created. The miner sorts these actions by target shard and execution number and creates two Block Merkle Trees: The Add Merkle Tree and the Merge Merkle Tree. The miner then records the roots of the two Merkle Trees into the new block header and broadcasts the block header to the whole network. In addition, the miner generates a set of Partial Add Merkle Trees and Partial Merge Merkle Trees from content in the new block relevant to every shard, and delivers them to the corresponding shard's storage nodes. The storage node verifies the received Partial Merkle Trees according to their Block Headers, and uses them to reconstruct its in-shard Main Merkle Tree. For more details about this process, please refer to MultiVAC's Yellow Paper.

5.3 Execution of Cross-Shard Operations

When a miner receives a confirmed block header from another shard, it caches its corresponding Add and Merge Merkle Roots. When the miner generates a new block for its own shard, he appends the Add Merkle Tree to his shard's Main Merkle Tree and runs the actions in the generated Merge Merkle Tree. The process by which Merkle Trees are appended to each other are discussed previously in the Yellow Paper.

Before running Actions in the Merge Merkle Tree, in-shard miners who did not generate the action must request the Action details and corresponding updated data from the shard's storage node. Miners obtain the complete Partial Merge Merkle Tree from an adjacent miner or storage node and verify the action's authenticity against the Merge Merkle Root obtained from the block headers.

When the miner is selected as the shard's block producer, he will first sort all received blocks by height that have not yet been added to the shard's Main Merkle Tree. The miner adds the Add Merkle Tree corresponding to its shard height, and runs all actions in the Merge Merkle Tree corresponding to their shard height.

All cached blocks are first added to the shard's Main Merkle Tree before the miner adds its newly generated block. The block height of the newly generated block is updated to reflect the addition of the previously cached blocks.

We summarize the advantages of MultiVAC's cross-sharding mechanism below:

- **Monotonic consistency**: By sorting cached data according to (shard, block height), merge operations of lower shard id's must be carried out before that of higher shards id's. By sorting merge operations in a block, operations are guaranteed to be processed according to the block order it was originally generated from its shard. MultiVAC thus guarantees the monotonous consistency of cross-shard data processing between shards.
- **Separate processing for add and modify**: Add is a light operation requiring only the Add Merkle Root in a block's header, while merge operations are heavier, requiring communication with storage nodes. Depending on data needs, developers may select one or the other to achieve higher performance and lower cost. In the XYZ contract, a more frequent and less demanding token transfer operation is designed as an add operation while a less frequent but more demanding quota modification operation is designed as a merge operation.

6 Storage, Consensus, and Execution Modules

In this final section, we review how our smart contract operations interact with MultiVAC's modular design, which was covered in MultiVAC's Yellow Paper. MultiVAC shards operate the storage, execution, and consensus modules, such that storage nodes are responsible for the storage module and miner nodes are responsible for the execution and consensus modules.

6.1 The Storage module

MultiVAC's storage module is managed by each shard's storage nodes. This module manages all the shard's data and ensures that data is accurately updated. Individual storage nodes are incentivized in tokens for taking part in storage activities. In addition to storing and updating data, the storage module also collects users' transactions and relays them to miners alongside with the transaction's corresponding data.

It must be noted that storage nodes provide data storage services but do not have the right to modify data of their own volition. Any modifications they make are only deemed acceptable by miners if it matches miners' own internal states represented by the Miner's Merkle Root. When storage nodes serve data, they must send the data with its relevant Merkle Root. Miners first check if this Merkle Root matches their internal states before proceeding. If the storage node makes a false update or changes the data in any way between or during transactions, its Merkle Root will also change. Storage nodes that do not produce an accurate Merkle Root to miners will be ignored and their data will be rejected, and miners will switch to using other storage nodes. In this fashion, storage nodes can only provide services to miners. They are not able to conduct any operations of their own volition and still be allowed to operate within the MultiVAC system.

6.2 The Execution Module

MultiVAC's smart contract execution module is the first module managed by MultiVAC's miner nodes. The purpose of the execution module is to process smart contract function calls. The execution module contains a virtual machine which loads smart contract code and data to execute smart contract functions, delivering as output a series of data modification actions. Outside of the virtual machine, the execution module merges these outputs and records them in a block, giving them to the consensus module for consensus.

When smart contracts are processed, they produce additions, deletions, updates, or modifications to be sent to the storage nodes. Actions produced by smart contracts are permissioned, as MultiVAC limits data modification rights of smart contracts only to the specific DataCells which belong to it. However, smart contracts may define public interfaces that are callable by other smart contracts, whereby other smart contracts can modify its managed data. This framework ensures that if the modifying smart contract made an error, the error can be contained and will not affect other users.

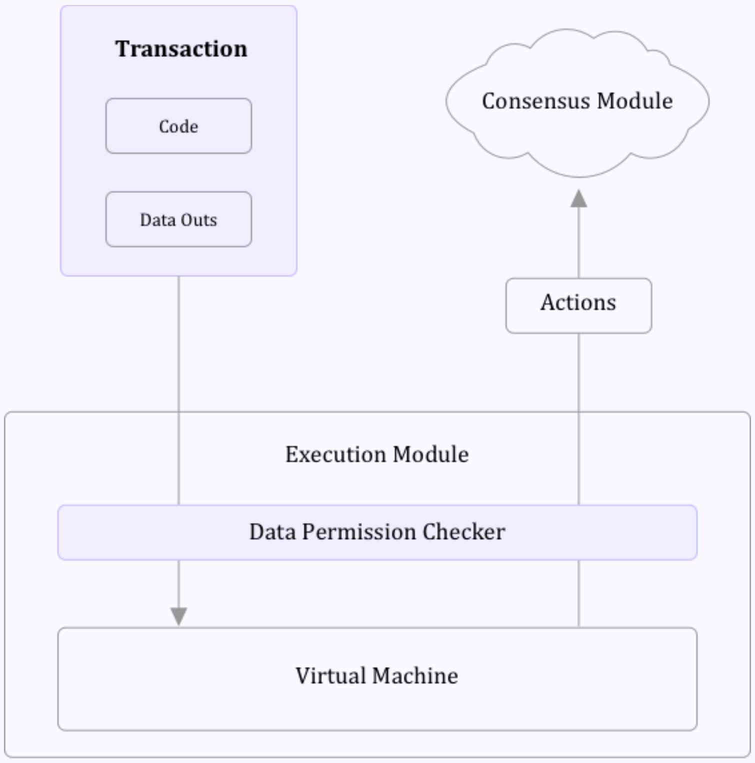

Figure 4: The interactions between MultiVAC's consensus and execution modules.

At the center of MultiVAC‘s execution module is a fast, secure and debuggable virtual machine. The virtual machine has built-in a Turing-complete instruction set which is a portable compilation target of LLVM, a general compiler framework. LLVM can translate many high-level languages such as C, C++, and Rust into Intermediate Representation (IR), optimizing the code using a mature optimization module and translating the optimized code into instructions for target hardware. Under this framework, developers on MultiVAC can flexibly select different high-level languages according to their specific conditions and maximize reuse of their existing codebases. LLVM provides multiple-language source-level debugging as well as performance profiling to facilitate the quick development of efficient code.

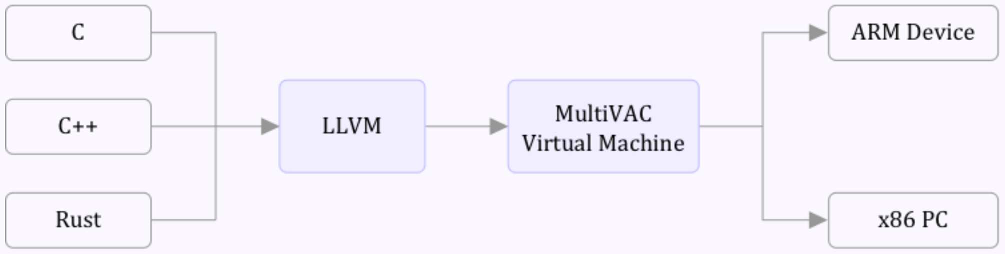

Figure 5: The LLVM front-end compiler and MultiVAC's back-end virtual machine.

MultiVAC currently chooses software-based virtual machines in order to best facilitate cross-platform support and participation. However, because LLVM allows for target architecture switching, MultiVAC wishes in the future to facilitate execution efficiency by also supporting light hardware execution modules running MultiVAC’s virtual machine. Such hardware machines will have no advantage in consensus selection because MultiVAC’s Proof-of-Stake mechanism ensures full randomness to all miners regardless of their hardware use. However, they would allow for more efficient smart contract execution, significantly furthering blockchain’s throughput volume and capability in handling real-economy problems.

Finally, in order to avoid infinite-loop attacks, a distributed system should have a catch-safe mechanism to solve the Turing halting problem. For such a mechanism, MultiVAC’s virtual machine is equipped with a gas charging mechanism similar to Ethereum. Gas is charged for every instruction that is executed on the virtual machine, which halts if the gas cost is prematurely depleted.

6.3 The Consensus Module

In addition to the execution module, MultiVAC miners are also responsible for a shard’s consensus module. MultiVAC’s consensus module consists of running MultiVAC’s consensus algorithm on new blocks generated by miners in the shard. The miner independently checks the block’s validity and votes whether or not to add the block into the blockchain.

MultiVAC’s consensus module runs a Proof-of-Stake based Byzantine consensus algorithm. In Proof-of-Stake, miners are randomly rotated using a random key that is included in each block generation. This random key is provably unpredictable *a priori*, preventing any miners from inserting themselves into the equal selection process.

To participate in the consensus, miners must first lock a certain amount of MultiVAC tokens as deposit. This prevents miners from setting up fake accounts to gain unfair advantage. A miner's selection probability is proportional to the amount of stake he puts up, and miner forfeits his stake if he misbehaves.

Finally, miners run the Byzantine consensus algorithm, collecting several rounds of voter signatures on the new blocks' validity. Given enough signatures, miners will achieve consensus on the block and will broadcast the consensus message to every shard along with its relevant block content. For these consensus services, miners are incentivized in tokens for participating in block production.

6.4 The Overall MultiVAC System

Having given an overview of MultiVAC smart contracts and their interactions with MultiVAC's storage, execution, and consensus modules, we draw here a full picture of the process whereby a smart contract call is processed in MultiVAC:

- To trigger smart contracts on any shard, user nodes must send a transaction request to storage nodes in the shard containing the smart contract address and the shard number.
- The storage node searches the database for the smart contract address contained in the transaction request, providing relevant smart contract code and data to miner nodes in the shard.
- Miners validate the legitimacy of the transaction call by checking it against the user's public key and validate the legitimacy of the smart contract data by checking them against the miner's stored Merkle Roots.
- If the code and data is legitimate, miners will run the computations in the smart contract and generate a list of smart contract outputs consisting of data actions to be packaged into blocks.
- Once block packaging is complete, Miners confirm the blocks among themselves and update their Merkle Roots. (At this stage they also process and update any previously received cached blocks from other shards, which we address below.) Miners divide up the current block content into Partial Merkle Trees as relevant for each shard and distribute the partial block content to each shard.
- Miners in all shards receive new block headers produced by other shards. They cache the blocks and, when it becomes time for them to output a block, request relevant data from storage nodes to execute add and merge actions from the received blocks, and compute the new Merkle Roots. This computation does not update the storage, it allows miners to update their summary states corresponding to what the storage should be.
- When a new block is produced in the new shard, the new shard's storage nodes receive the new block with the partial blocks from other shards. Storage nodes also execute the add and merge actions, updating their internal storage and Merkle Root to match the miners'. If the storage node's Merkle Root is not consistent with the miners', the storage node's data will be rejected by miners in the next operation. At this point, the transaction has been completed.

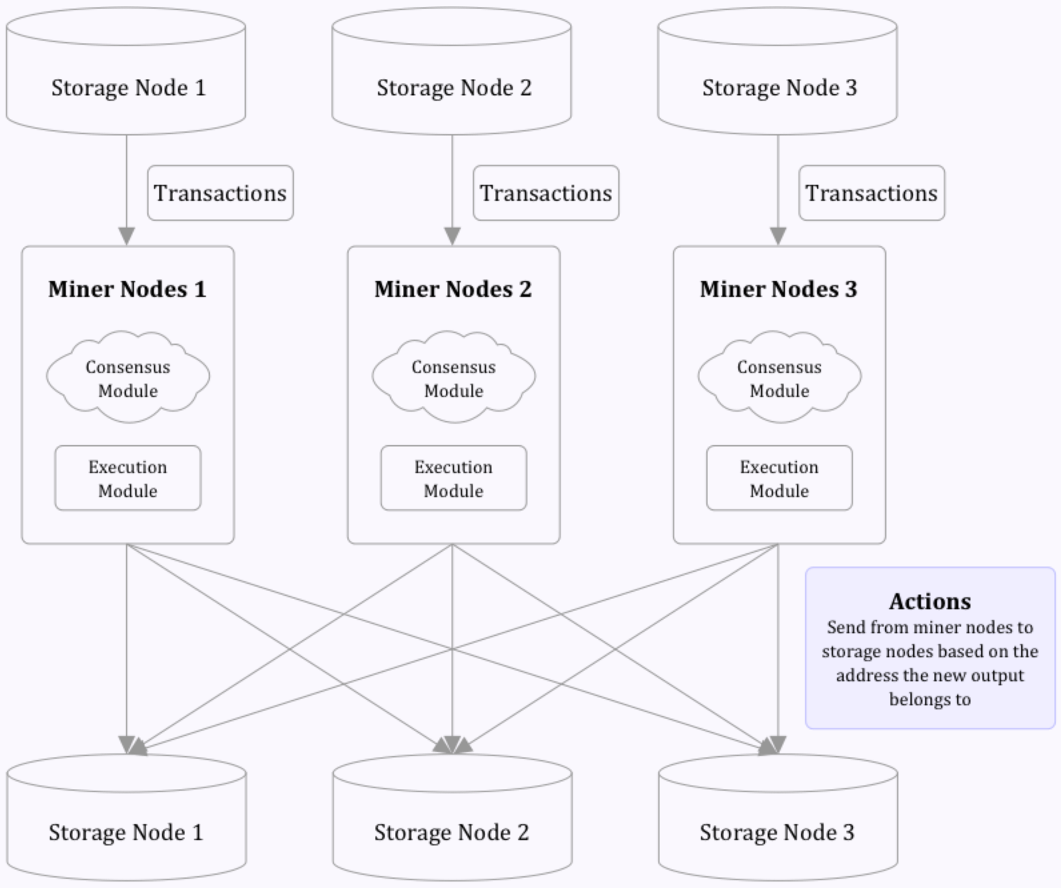

Figure 6: Cross-shard interactions between miner and storage nodes in MultiVAC.

7 Conclusion

In the above sections we present MultiVAC's novel blockchain sharding architecture supporting Turing-complete smart contracts.

The central challenge facing large-scale public blockchains today is that of low throughput. The most promising solution to the throughput issue, blockchain sharding, allows for high scalability with a number of difficult design challenges. Sharding is the division of a blockchain network into a series of sub-networks which separately process transactions. MultiVAC has made a variety of design choices to allow blockchains to maximize throughput while maintaining fairness and decentralization.

MultiVAC's design was based on a few core principles. The first is resolute commitment to the core idea of decentralization. MultiVAC gives final control over the blockchain network to ordinary miner nodes, the requirements for which are always securely kept to a low level through an elegant Proof-of-Stake mechanism. Processing in the MultiVAC network will never leave the control of ordinary miners, and anyone using a PC with modern capabilities will be able to participate in full capacity.

The second core principle is modular design. MultiVAC implements an intricate and empowering storage mechanism as separate from the execution mechanism, improving system efficiency without ever threatening decentralization. Every action of MultiVAC storage nodes is checked and validated by miners, and this secure separation of powers using Merkle Tree cryptography frees MultiVAC miners from storage and throughput constraints.

On top of these foundations, MultiVAC designs a vital and powerful smart contract execution system that maintains full Turing completeness. Its implications are that smart contract developers can achieve any kind of application logic that they would be able to achieve on a normal computer. In addition, MultiVAC designs for giving developers maximum development flexibility, enabling them to tailor blockchain processing according their specific needs.

It must be reiterated that MultiVAC is base-level architecture. MultiVAC fundamentally redesigns the blockchain process and tackles and addresses the problems of blockchain scalability at their root. On top of this base layer infrastructure, space for a myriad of design extensions are possible, including a lighting network or privacy computing.

For this reason, MultiVAC is the first fully general and fully decentralized public blockchain that can maintain linear scalability with any increase of user load. A practical, realizable next generation sharded blockchain platform, MultiVAC robustly supports high throughput commercial applications at any level. MultiVAC tirelessly works towards implementing the cutting-edge of blockchain, playing a central role in the implementation of blockchain technology throughout the real economy.