An Exploration of Storage Pricing and Gas Accounting Designs for EVM Blockchains

Kushal Babel, Yacine Dolivet, Michael Setrin
Category Labs

February 9, 2026

Contents

1 Introduction

Like all blockchain resources, storage has to be priced appropriately for the sustainability of the blockchain. In this document, we are mainly concerned about the price of `SSTORE` and related opcodes when it creates a new storage slot, or frees up an existing storage slot.

The storage component of `SSTORE` is priced at 17,100 gas on Ethereum. At a gas price of 100 MONgwei (`min_base_fee`), this pricing is not sustainable without additional price or quantity controls. In this document, we explore four different approaches to controlling storage allocations and their tradeoffs. Table 2 provides a summary.

For preliminaries, recall that gas_bid, the bid for gas by transaction t in block k is given by:

$$\text{EIP-1559 tx type} : \text{gas\_bid}_t = \min(\text{priority\_fee\_per\_gas}_t + \text{base\_fee}_k, \text{max\_fee\_per\_gas}_t)$$
$$\text{Legacy tx type} : \text{gas\_bid}_t = \text{gas\_price}_t$$

2 Four Approaches to Controlling Storage

2.1 Real-time Gas Accounting (RGA) with revised relative gas price

This is the approach in the v1.2.0 of Monad Initial Specification Proposal.

2.1.1 Method

The costs of a few opcodes differ from Ethereum. The relative gas for storage ("s" component) and refund ("r" component) was increased, with similar accounting as Ethereum, except that gas_bid is paid on the gas_limit, and refund is done at the `min_base_fee` level.

Storage Gas Cost and Refunds: In the Ethereum Yellow paper, as a deterrent to state growth, creating a new storage slot costs a large amount of gas. However, refunds for clearing storage slots only give back a small fraction of the cost of creating that storage slot. As a result, the incentive to reduce state growth is limited. In Monad, `SSTORE` gas cost and refunds are changed to address this problem. To account for significantly higher execution capacity, the cost of creating a storage slot is relatively higher; however, the refund upon freeing the storage slot is proportionally much higher.
The `SSTORE` gas cost can be expressed in terms of component costs:

- `m`: baseline cost for the operation
- `w`: cost of writing a value to storage
- `s`: cost of creating a non-zero storage slot
- `r`: refund for clearing a storage slot

In order to achieve the desired effect, we propose changing the values of `s` and `r` while leaving the values of `m` and `w` unchanged:

Component Cost	Description	Ethereum	Monad
`m`	baseline	100	100
`w`	write to storage	2800	2800
`s`	create non-zero slot	17100	125000
`r`	refund for clearing slot	4800	120000

Now, we specify the gas cost of `SSTORE` in terms of these component costs depending on the following inputs.
Inputs:

- value: new value to be stored.
- current_value: current value of the storage slot.
- original_value: value of the storage slot before the current transaction.

Gas cost for `SSTORE`:

```
if value == current_value:
    gas_cost = 100 (m)
elif current_value == original_value:
    if original_value == 0:
        gas_cost = 127900 (m + w + s)
    else:
        gas_cost = 2900 (m + w)
else:
    gas_cost = 100 (m)
```

Gas refund for `SSTORE`:

```
gas_refund = 0
if value != current_value:
    if current_value == original_value:
        if original_value != 0 and value == 0:
            gas_refund += 120000 (r)
    else:
        if original_value != 0:
            if current_value == 0:
                gas_refund -= 120000 (−r)
            elif value == 0:
                gas_refund += 120000 (r)
        if value == original_value:
            if original_value == 0:
                gas_refund += 127800 (s + w)
            else:
                gas_refund += 2800 (w)
```

The storage refund storage_refund is then calculated as:

$$\text{storage\_refund} = \text{gas\_refund} \cdot \texttt{min\_base\_fee} \tag{1}$$

Storage other than SSTORE: Similar to `SSTORE`, the `CREATE` and `CREATE2` opcodes also use storage for smart contract accounts and bytecodes deployed through them. The relative storage cost of `CREATE` and `CREATE2` is revised as follows:

Component Cost	Ethereum	Monad
"code_deposit_cost"	200/byte	1200/byte
static base cost	32000	160000

2.1.2 Tradeoffs

- Pros:
  1. Simplicity (Transaction fee is still one dimensional).
  2. Compatible with ERC-4337 entrypoint contract, with a small caveat that it requires charging pessimistic costs to the users and refunds do not flow back.
- Cons:
  1. Backward incompatible: opcode cost for SSTORE is different from Ethereum mainnet and most L2 EVMs.
  2. Higher gas-used variance: for transactions with dynamic branching, the effective gas-used variance is relatively higher than Ethereum mainnet if different branches have different SSTORE operations.
  3. Dead-weight source 1: gas-limit is charged at consensus time however storage refund is delayed to execution time. This source is unavoidable even with perfect estimation of the gas-limit. *In the limit, this presents a DoS vector where a transaction could consume the entire gas limit at the time of inclusion but not really pay for it.*
  4. Dead-weight source 2: each individual user needs to estimate their transaction gas-limit conservatively due to the possible presence of other transactions earlier in the same block impacting their gas-used. This source is also unavoidable at equilibrium.

Remark: Note that the Monad Initial Specification Proposal v1.2.0 had other pricing changes related to cold access cost and precompiles.

2.2 Deferred Gas Accounting (DGA)

This approach also uses revised relative gas prices, but defers the gas accounting of storage allocations until the end of the transaction so as to net out allocations and deallocations before comparing against the gas limit[1]. This allows to lower the gas limit of the transaction due to any refunds. Lower gas limit and deferred accounting has a few benefits:

- Unlike the storage related refunds priced at `min_base_fee` in RGA, refunds here can be made in part at the higher price of gas_bid_t . More precisely, any storage allocation can be fully matched with a storage deallocation at gas_bid_t. Any excess refunds happen at `min_base_fee`. This is a consequence of the fact that a lower gas_limit_t is sufficient to do lots of storage if lots of refund are also being generated due to freeing up of slots.
- The blockspace, which is packed based on gas limit, can be packed more efficiently (lowers the dead-weight loss created by the fact that gas, a unidimensional measure, is being used to price storage).

The deferred accounting has pros and cons:

- Pros:
  1. During transaction execution, the smart contracts do not see the higher relative pricing, which may preserve compatibility of some internal calls made with a gas limit budget according to the original gas costs.
  2. Dead-weight source 1 is considerably reduced since storage refunds can be netted against storage allocated for the purpose of setting transaction gas-limit. /mikeThe degree of this reduction is highly case-specific
  3. Dead-weight source 2 is also mitigated since now the contribution of storage to the gas-used variance is also reduced.
  4. TFM is still one-dimensional.
- Cons:
  1. Not compatible with the current ERC-4337 entrypoint contract, which does require the worst case charge to show up realtime during the execution of the transaction (through the 0x5A (GAS()) opcode). This is so that the bundler can charge each user operation appropriately. Under the deferred accounting approach, a modified entrypoint contract that utilizes a new opcode for "deferred gas" would be needed to calculate the appropriate charge for each user operation.

2.2.1 Method

We introduce a new quantity called *deferred gas* in order to perform this acocunting.
`SSTORE` Calculations are a lot simpler:
Inputs:

- value: new value to be stored.
- current_value: current value of the storage slot.
- original_value: value of the storage slot before the current transaction.

Gas cost for `SSTORE`:

```
gas_cost = 100 (m)
```

[1]The deferred accounting approaches reflect the underlying implementation, which also defers the storage writes rather than doing them in realtime while executing the transaction.

This low cost allows for reducing gas_limit$_t$ dramatically if there are refunds as well in the transaction, thereby reducing the charge to the user.
Note that the opcode 0x5A (GAS()) continues to return gas_limit$_t$ − gas_cost.

Deferred Gas for `SSTORE`:
A new quantity called deferred_gas is computed.

```
if (value != current_value) and (current_value == original_value):
    if original_value == 0:
        deferred_gas = 127800 (w + s)
    else:
        deferred_gas = 2800 (w)
```

Gas refund for `SSTORE`:

The computation of gas_refund remains unchanged (same as RGA). Payment rule for user and Burn rule also remains unchanged. We now calculate storage_refund as:

$$\text{storage\_refund} = \max(0, \text{gas\_refund} - \text{deferred\_gas}) \cdot \texttt{min\_base\_fee} \tag{2}$$

$$\text{Payment}_t = \text{gas\_bid}_t \cdot \text{gas\_limit}_t - \text{storage\_refund} \tag{3}$$

$$\tag{4}$$

Note that contrary to RGA, only the *net refund* (if there is one) is priced at `min_base_fee` rate which is potentially lower than the price at which the storage was originally allocated.
Execution Logic for transaction t:

```
Steps:
1. balance[t.sender] -= gas_bid(t) x gas_limit(t)
2. (gas_cost, gas_refund, deferred_gas) = Execute(t) // Do not write to disk yet
3. Revert if gas_limit < deferred_gas + gas_cost -  gas_refund
4. Can flush the writes to the disk at or after this point.


Execute(t) in step 2 enforces the following invariant throughout execution, else reverts:
(1) gas_limit >= gas_cost.
(2) gas_limit >= (gas_cost + deferred_gas) - Δ.
Invariant (2) is optional because Δ is large (think millions). No DOS attack vector as I/O in step
     4 is performed after verifying step 3.
```

2.3 Separate Storage Charge - SSC

2.3.1 Overview

Gas metering of storage related opcodes (`SSTORE`, `CREATE`, `CREATE2`) is modified to only account for computation and I/O costs and leave out the storage related component. Consequently, the gas consumption is reduced especially significantly for creation of new storage slots through `SSTORE`. This allows transaction gas limits to be set lower and be more predictable, allowing for more transactions to be packed for a given block gas limit. At the end of transaction execution (and before reserve balance checks), a separate storage charge (denominated in a fixed MON price per byte) is levied based on the net growth of storage due to account creations, storage (de)allocations, and contract code deployed. If the net growth is negative, the storage charge is negative, resulting in a refund to the transaction sender. Almost all of the cost is refunded upon deallocations. New precompiles are introduced for smart contracts to be able to introspect the accrued storage charge throughout the execution of transaction. Storage charge is upper bounded as a function of the gas limit, so as to prevent unbounded charges. If the storage charge assessed at the end of the transaction

execution is higher than this implicit limit, the transaction is reverted and no storage charge is levied (only the gas related charges would apply). Slots created and freed, or accounts created and destructed, within the same transaction count for exactly zero storage charge.

2.3.2 TFM

Payment Rule for User: The payment for transaction t is given by the sum of the gas fees and the storage charge:

$$\begin{aligned} \text{Payment}_t &= \text{gas\_fees}_t + \text{storage\_charge}_t \\ \text{gas\_fees}_t &= \text{gas\_bid}_t \cdot \text{gas\_limit}_t \\ \text{storage\_charge}_t &= (\text{bytes\_created}_t - \texttt{STORAGE\_REFUND\_PCT} \cdot \text{bytes\_freed}_t) \cdot \texttt{STORAGE\_COST\_PER\_BYTE} \end{aligned}$$

Maximum Storage Charge Bound: To prevent unbounded storage charges, a maximum is enforced:

$$\begin{aligned} \text{max\_storage\_charge}_t = \max \big(&\texttt{STORAGE\_MAX\_SPEND\_FLOOR\_BYTES} \cdot \texttt{STORAGE\_COST\_PER\_BYTE}, \\ &\texttt{STORAGE\_MAX\_SPEND\_FACTOR} \cdot \texttt{MIN\_BASE\_FEE} \cdot \text{gas\_limit}_t\big) \end{aligned}$$

If $\text{storage\_charge}_t > \text{max\_storage\_charge}_t$, the transaction reverts and storage charge is not levied.

Burn Rule: The burn for each transaction t in block k is the gas burn and the storage escrow's burn/mint:

$$\begin{aligned} \text{Burn}_t &= \text{gas\_burn}_t + \text{storage\_charge}_t \\ \text{gas\_burn}_t &= \text{base\_fee}_k \cdot \text{gas\_limit}_t \end{aligned}$$

Note that the storage charge is burnt (or minted if negative).

Reward Rule for the Validator: The reward for the validator for block k is given by the sum of the priority gas fees and a constant block reward (staking and consensus-related rewards/penalties are not included here):

$$\text{Reward}_k = \sum_{t \in \text{block}} (\text{gas\_bid}_t - \text{base\_fee}_k) \cdot \text{gas\_limit}_t$$

The storage charge does not affect validator rewards.

2.3.3 Opcode Gas Costs

SSTORE Opcode Cost: The `SSTORE` opcode gas cost is computed using the following components:

- $\texttt{m} = 100$ (baseline opcode cost)
- $\texttt{w} = 2800$ (write cost)
- $\texttt{s} = 0$ (storage allocation cost, now handled separately)
- $\texttt{r} = 0$ (refund, now handled separately)

Gas cost for `SSTORE`:

```
# Inputs:
#   value: new value to be stored
#   current_value: current value of the storage slot
#   original_value: value before the current transaction

if value == current_value:
    gas_cost = 100 (m)
```

```
elif current_value == original_value:
    gas_cost = 2900 (m + w)
else:
    gas_cost = 100 (m)
```

Note: The storage allocation component (s = 125000 gas in Initial Proposal) is now handled through the separate storage charge mechanism. Similarly, refunds are handled separately rather than through gas refunds.

CREATE and CREATE2 Opcode Costs: The CREATE and CREATE2 opcodes deploy contract bytecode to persistent storage. The gas cost structure is:

$$\begin{aligned}\text{CREATE/CREATE2 gas cost} = \text{static\_cost} + \text{init\_code\_cost} + \text{memory\_expansion\_cost} \\ + \text{deployment\_execution\_cost} + \text{code\_deposit\_cost}\end{aligned}$$

Under SSC, the code deposit cost is slashed by half and revised to the following:

- $\text{code\_deposit\_cost} = 100 \cdot \text{deployed\_code\_size}$ (per byte)

This reflects only the write I/O cost. The persistent storage cost for the deployed bytecode is accounted for separately in the storage charge based on deployed_code_size.

2.3.4 Storage Charge Parameters

The following protocol parameters govern storage charge computation:

Storage Size Constants:

- STORAGE_ACCOUNT_BYTES = 128 (bytes per account)
- STORAGE_SLOT_BYTES = 64 (bytes per storage slot, includes key and value)
- STORAGE_CODE_BYTES = 1 (bytes per byte of contract code)

Pricing Parameters:

- STORAGE_COST_PER_BYTE = 0.0002 MON (cost per byte of storage)
- STORAGE_REFUND_PCT = 0.99 (refund percentage for deallocations)

Bounds Parameters:

- STORAGE_MAX_SPEND_FACTOR = 10 (multiplier for maximum storage spend relative to gas payment)
- STORAGE_MAX_SPEND_FLOOR_BYTES = 128 (minimum bytes guaranteed allocatable in any transaction)

Cost Examples: Using STORAGE_COST_PER_BYTE = 0.0002 MON:

- Cost per storage slot: $64 \cdot 0.0002 = 0.0128$ MON
- Cost per account: $128 \cdot 0.0002 = 0.0256$ MON
- Cost for 100 GB of state: $100 \cdot 10^9 \cdot 0.0002 = 20{,}000{,}000$ MON

2.3.5 Storage Charge Computation

State Change Tracking: During transaction execution, the system tracks:

- $\text{accounts\_created}_t$: number of new accounts created
- $\text{accounts\_cleared}_t$: number of accounts selfdestructed
- slots_created_t: number of new storage slots allocated (transitions from zero to non-zero)
- slots_cleared_t: number of storage slots deallocated (transitions from non-zero to zero)
- $\text{code\_bytes\_created}_t$: total bytes of contract code deployed

Note that if the freed storage was created in the same transaction, then the creation count is decremented accordingly, rather than counting both creation and deletion. This ensures that storage created and freed within the same transaction results in zero net storage charge.

Storage Bytes Calculation:

$$\begin{aligned}\text{bytes\_created}_t = \text{accounts\_created}_t &\cdot \texttt{STORAGE\_ACCOUNT\_BYTES} \\ &+ \text{slots\_created}_t \cdot \texttt{STORAGE\_SLOT\_BYTES} \\ &+ \text{code\_bytes\_created}_t \cdot \texttt{STORAGE\_CODE\_BYTES}\end{aligned}$$

$$\begin{aligned}\text{bytes\_freed}_t = \text{accounts\_cleared}_t &\cdot \texttt{STORAGE\_ACCOUNT\_BYTES} \\ &+ \text{slots\_cleared}_t \cdot \texttt{STORAGE\_SLOT\_BYTES}\end{aligned}$$

Storage Charge Formula:

$$\begin{aligned}\text{storage\_charge}_t = \texttt{STORAGE\_COST\_PER\_BYTE} &\cdot \text{bytes\_created}_t \\ - \texttt{STORAGE\_COST\_PER\_BYTE} &\cdot \texttt{STORAGE\_REFUND\_PCT} \cdot \text{bytes\_freed}_t\end{aligned}$$

The storage charge can be negative if the transaction results in net deallocation. In this case, the user receives a refund.

End-of-Transaction Logic:

```
# After transaction execution completes:

# 1. Compute storage changes
storage_bytes_created = (accounts_created * STORAGE_ACCOUNT_BYTES +
                         slots_created * STORAGE_SLOT_BYTES +
                         code_bytes_created * STORAGE_CODE_BYTES)

storage_bytes_cleared = (accounts_cleared * STORAGE_ACCOUNT_BYTES +
                         slots_cleared * STORAGE_SLOT_BYTES)

# 2. Compute storage charge
storage_charge = (STORAGE_COST_PER_BYTE * storage_bytes_created -
                  STORAGE_COST_PER_BYTE * STORAGE_REFUND_PCT * storage_bytes_cleared)

# 3. Enforce maximum storage charge bound
max_storage_charge = max(
    STORAGE_MAX_SPEND_FLOOR_BYTES * STORAGE_COST_PER_BYTE,
    STORAGE_MAX_SPEND_FACTOR * MIN_BASE_FEE * gas_limit
)
```

```
if storage_charge > max_storage_charge:
    revert_transaction()

# 4. Deduct storage charge from user balance
user_balance -= storage_charge

# 5. Check final balance
if user_balance < 0:
    revert_transaction()

# 6. Perform reserve balance checks (if applicable)
if not reserve_balance_check():
    revert_transaction()

# 7. Finalize transaction and commit state changes
```

2.3.6 Reserve Balance Modifications

In the Monad TFM specification, reserve balance logic ensures that accounts have sufficient funds to pay for pending transactions despite asynchronous execution. Under SSC, the reserve balance check must account for both gas payments and potential storage charges.

Consensus Check for Emptying Transaction: When performing block validity checks for block n, consensus must verify that each transaction sender has sufficient balance to cover the maximum possible spend:

$$\begin{aligned}\text{max\_storage\_charge}_t = \max\big(&\texttt{STORAGE\_MAX\_SPEND\_FLOOR\_BYTES} \cdot \texttt{STORAGE\_COST\_PER\_BYTE},\\ &\texttt{STORAGE\_MAX\_SPEND\_FACTOR} \cdot \texttt{MIN\_BASE\_FEE} \cdot \text{gas\_limit}_t\big)\end{aligned}$$

$$\text{total\_charge\_for\_emptying}_t = \text{gas\_bid}_t \cdot \text{gas\_limit}_t + \text{value}_t + \text{max\_storage\_charge}_t$$

The reserve balance logic uses $\text{total\_charge\_for\_emptying}_t$ when computing worst-case balance scenarios for pending emptying transactions. This ensures that even if a transaction allocates the maximum allowed storage, the sender will have sufficient balance to pay for it.

2.3.7 Discussion

Who gets the refund from storage escrow? The user that creates the storage slot could be different from the user that frees the storage slot. The user that frees the storage slot gets the refund from the storage escrow, while the user that created the storage slot pays the storage escrow charge. However, smart contract developers can control who effectively pays the storage escrow charge and who effectively gets the refund, by charging different fees for operations that create and free up storage slots. For instance, in the context of a limit order book, the maker fees can be low to offset the storage escrow charge, while the taker fees can be higher to offset the refund from the storage escrow.

2.3.8 Tradeoffs

- Pros:
    1. The storage charge is not a part of the effective fees if the slot is freed up; it really is a temporary escrow[2]. This allows the effective transaction cost to be quite low. All of the refund is available irrespective of which transaction the deallocation happens in.

[2]Technically speaking the economic cost for the user allocating storage for a period of time is the time-value of the capital locked over that period.

2. The gas-limit of the transaction can be kept low and predictable, and the block space, which is packed based on transaction gas-limit, can be packed more efficiently.

- Cons:

1. The total payment from the user can be higher than $\text{gas\_limit}_t \cdot \text{gas\_bid}_t$ if there are net storage allocations. The payment is upper bounded by $\text{gas\_limit}_t \cdot \text{gas\_bid}_t \cdot (1 + \frac{\texttt{charge\_per\_slot}}{\text{gas\_bid}_t \cdot \texttt{SSTOREgas}})$.
2. The realtime gas accounting through the 0x5A (GAS()) opcode becomes less meaningful since gas is not a good proxy for the total payment anymore. This can break compatibility with EIP-4337 account abstraction where the entrypoint contract needs to know the maximum possible charge for the user operation. Modifications to the ERC-4337 entrypoint contract based on a new opcode that exposes net storage allocations in realtime should suffice to appropriately charger user operations.
3. TFM is now 2-dimensional which may imply integration work by dapps developpers.

2.4 Storage Control by Quantity - SCQ

When designing mechanisms to regulate resource usage, one faces the classic choice between control by price—adjusting economic incentives through fees—and control by quantity—directly capping the resource itself; in the case of blockchain storage, this translates into deciding whether to control storage via estimating the *fair price* or by imposing a limit on storage created per block.

When the marginal damage of overuse rises steeply and is hard to measure in advance[3], control by quantity can be more efficient than control by price, because a cap guarantees the worst-case outcome is avoided, even if the fair price of storage is uncertain.

2.4.1 Method:

Unlike the three approaches above, in the SCQ approach, the relative gas prices do NOT need to be revised; only a constraint on the amount of net storage created per block (`storage_limit`) is enforced during execution to avoid the worst case of storage creation[4]. The parameter `max_slots_per_block` can be set by the validator committee (either hard-coded as part of execution client itself, or better yet, through an onchain precompile which allows to efficiently respond to a worst-case situation without requiring client updates).

2.4.2 Guidance on setting the parameter `max_slots_per_block`

One possible way to set `max_slots_per_block` is to calculate the upper bound on the storage created in the prior approaches. In a control-by-price setting the maximum storage increase per day is *implied* by `SSTORE` opcode cost. In the context of the RGA proposal (Section 2.1), table 1 shows that this proposal corresponds to a theoretical upper bound of roughly 11GB per day or equivalently a maximum number of 1600 `SSTORE` per block. This viewpoint can be leveraged to help choose a control-by-quantity threshold (`max_slots_per_block` =1600). The status quo before any `SSTORE` changes is equivalent to `max_slots_per_block` being unbounded.

Metric	**Value**
blocks per sec	2.5
max gas per sec	500M
SSTORE gas in RGA	127900k
max slots per sec	$\sim$ 3909
max slots per block	$\sim$ 1563

Table 1: Upper bound on storage allocation under the RGA approach

[3] as with blockchain storage, where uncontrolled state growth (due to mispricing) can threaten node viability

[4] SCQ approach can complement any control by price approach, including the other three approaches described in this document.

2.4.3 Tradeoffs of SCQ

- Pros:
  1. The relative gas price of `SSTORE` does NOT need to be revised, avoiding the downstream consequences as seen in other approaches
- Cons:
  1. It becomes possible for a transaction to revert at execution time because of the storage bound despite having sufficiently high transaction gas limit. This should be a tail event, occurring only when the net storage creations in a block has exceed the quantity control. The `max_slots_per_block` parameter can be raised dynamically by the validator committee if needed.
  2. While the worst-case is controlled explicitly, it is *cheaper* than RGA to attain it so it is more likely to occur on a relative basis.
  3. This approach in isolation does not incentivize better smart contract state hygiene (it would need to be combined with one of the other approaches to get that benefit).

Essentially, a control by quantity approach aims to defend against the worst-case while avoiding the inefficiencies inherent in guessing the right price.

3 Tradeoffs Summary

Approach	Needs Revised Gas Pricing	Refund	Deadweight Loss to Block Gas Limit	Compatible with ERC-4337	Payment Cap is Gas Limit × Gas Bid
RGA (§2.1)	Yes	At `min_base_fee` only (even in the same tx)	High	Yes	Yes
DGA (§2.2)	Yes	At gas_bid upto net zero, excess refunds at `min_base_fee`	Low	No	Yes
SSC (§2.3)	Separate charge	Full refund (across transactions)	None	No	No
SCQ (§2.4) + Ethereum pricing	No	At `min_base_fee` only (even in the same tx)	None	Yes	Yes

Table 2: Comparison of storage control approaches across key tradeoff dimensions