IDAS Feasibility Study Results

Mr. Jacob Torrey
Program Manager, I2O

July 9, 2019

Overview

- Results from I2O-funded feasibility study on software invariance showed feasibility for synthesizing more adaptable software at a DoD-relevant scale

- As part of the IDAS BAA process, included herein are the results and findings from that study
  - This work is **not exhaustive**, and while it does show the feasibility of *a* path, it <u>should not preclude responses exploring other approaches</u>
  - Diversity of approaches sought in BAA for TA1 (multiple awards)

- Software changes on a much quicker timescale than physical systems rooted in physical constants, thus what a developer treats as invariants vary quite frequently—resulting in expensive modernization cycles that simply rebind to a newer foundation
  - Concretization is the term used throughout for collapsing a set of *mostly* equivalent options into a single point (e.g., choosing a `int_16t` to store a certain value over an `int32_t`)

- This study was performed by a broad team of universities, non-profits, and research industry organizations, primed by Apogee Research

Software development today

- Option 1: Manual analysis and coding: issue is early “concretization”
  - Analyze requirements, pick an architecture, APIs, protocols – there are many choices that all seem “equally good” at a certain time
  - Hand code – hope the choices made above don’t conflict with the implementation – make more choices that are the “best” under current assumptions
  - Pray things don’t change – rework when they do! Use containers and/or virtualization to try to isolate software from changes, but that creates its own set of compounding complexity

- Option 2: Formal methods for synthesis: issue is state space explosion
  - Define state space (this is large, because there are lots of free variables)
  - Define the actions and the impact each has on the state space (even larger)
  - Assert the requirements, feed into a solver (e.g., SMT) -> Blows up on realistic problems

- The crux of the issue (and its solution!) is that there are many “free variables” that expand the space into sets of equivalent options
  - Early concretization removes the complexity, but cuts off solutions that might matter
  - If we can reason over sets of equivalent options and defer the choice of the instance within the set, we can avoid the state space explosion without early concretization

BLUF: Three key findings

1. Separate representations of the problem to be addressed and the specific solution
    - If the representations are equivalent, it is too easy for a developer to pollute the problem with their vision of how it should be solved in a single instance
2. Representing the program as a set of higher-level intentions (viable solution space) instead of specific source code (point in solution space)
    - By fixing (concretizing) even small “don’t cares”, you can implicitly fix others, causing the constraint satisfaction problem to explode combinatorically
    - Most “don’t cares” are really “don’t care as long as you act sanely” and require semantic descriptions of what is means to act sanely
    - Using controllers to stabilize towards sane behavior allows for easy composition as the details of stabilization can be ignored, and only compositional stability analysis is needed
3. APIs are concretizations
    - APIs act as abstractions, separating implementation details from their interface
    - These “abstractions” are static, and typically come with human-documented descriptions and conventions for use
    - Once an API is defined, it is difficult to adjust to other options that may be equally good for a particular problem, but have different syntax or semantics
    - Dynamically-generated APIs allow for the automated negotiation of what each component needs and the generation of a custom API for that composition

Can we sidestep the trade between early concretization and state space explosion?

- Solution is to refactor the state space into a new basis
  - Fixed variables that are constrained to meet the requirements
  - Free variables that can take on any value without impacting the requirements
    - Free variables are “don’t cares”
    - Strong free variables are true “don’t cares”
    - Weaker free variables are “don’t care to a point”

- Most free variables are not truly free, need human common sense/intuition
  - E.g., sorting a list – even if you do not care how the sort is implemented (at a higher-level of abstraction), you have expectations about the correctness and performance of any particular implementation
  - Key is to be able to reason about the state space without caring about its full size, then combine with a controller to ensure abstractions are upheld (e.g., that any valid sort implementation in the set will be performant and correct)
  - Make weak free variables appear strong to reduce state space

Study called these “don’t cares” or invariants Symmetries
(If you see symmetry, think “don’t care”)

Control theory / abstracted invariance approach to software

- Analyze the problem for its symmetries (strong and weak)
- Strengthen the weak symmetries by generating matched controllers that provide run time stabilization of the implementation
- Compose the strong and dynamically-generated symmetries
  - Limit state space explosion to that defined by the problem
  - Synthesize in only the relevant degrees of freedom of the problem
  - Defer concretization until after compositions are complete
- Result: Significantly more scalable synthesis that evolves with the problem
- Run-time controllers can convert weak symmetries to act like strong symmetries
  - Lower levels of the synthesis can leave residual errors that will get “cleaned-up” at higher levels of the composition – makes for easier lower level synthesis problems
  - Allows for loose bounding on each element (with associated residual errors) that opens up the solution space to new options

Concrete examples of resilient compensation of residual errors

- Want a mobile robot (e.g., BB-8 from Star Wars) that can move quickly through a crowd without tipping over
  - If you force instantaneous stability (strong symmetry), complexity is significantly reduced but the robot would barely move
  - If you allow for eventual stability, the robot could lean into a run knowing that it could return to upright when needed
  - Lower level controller provides a weak symmetry with a residual error that is compensated for at a higher level of composition

- NoSQL relaxed the constraint on instantaneous consistency (to eventual consistency) that has changed the way we handle big data
  - Atomicity, consistency, isolation, durability (strong symmetry) makes database reasoning easy, but scalability is extremely limited as everything needs to lock to re-establish consistency (strong symmetry)
  - Relax consistency and use a controller to establish the symmetry at a higher level of the composition allows for efficient distributed solutions

Example: Concretization for solving at scale

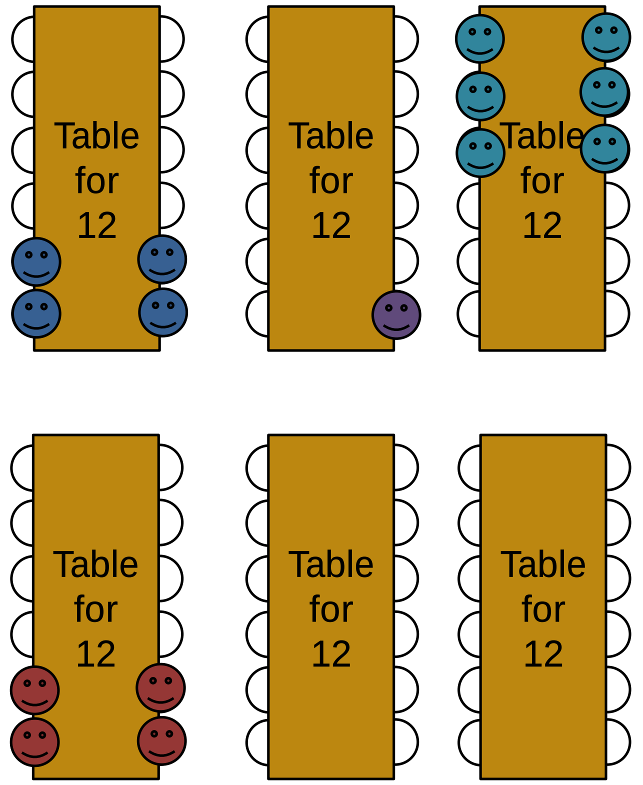
Today: manual early-concretization

- Requires over-provisioning due to inflexibility
    - Must accommodate expected worst case
- Ties solution to specific approach: costs to change once concretized are high

- This is the most common and natural solution to manage complexity

Example problem:

- Restaurant needs to seat parties of uncertain size (assumed between 1 & 12) when they arrive
- Goal: Maximize efficiency in seat utilization, while ensuring parties sit at tables together

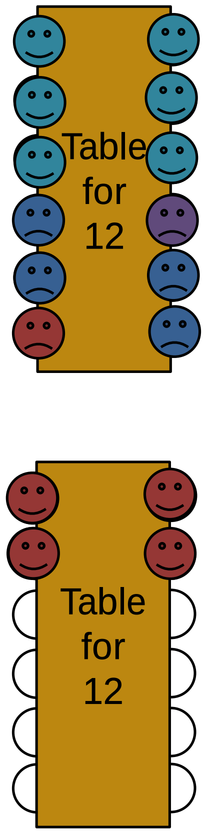


Uncertainty in party size managed by overprovisioned concretization

Difficult when assumptions change (e.g., party of size 15)

Past failures in automating complexity management

Under-constrained

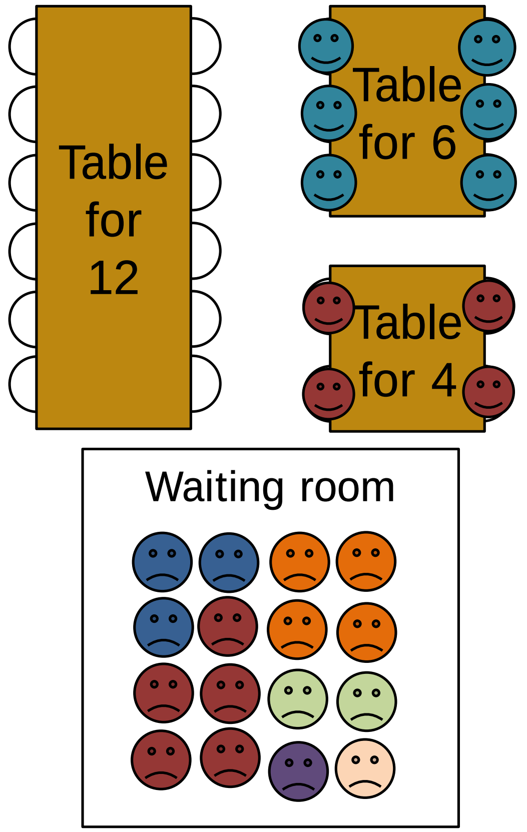


Maximum efficiency, but unhappy patrons

Naïve synthesis approach

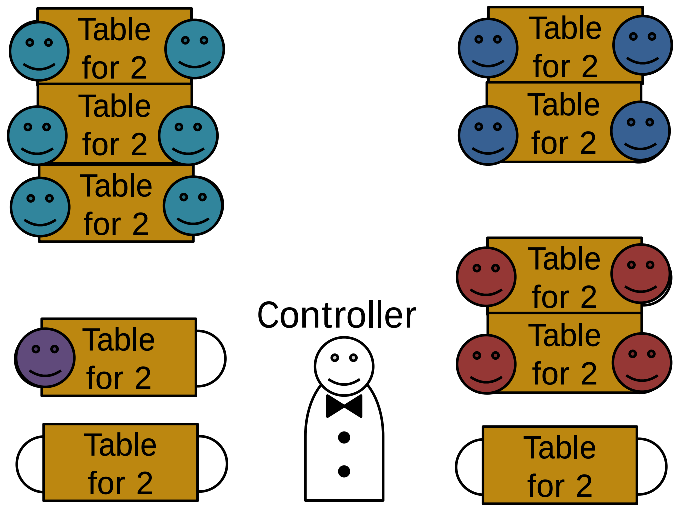
- Use problem constraints to attempt to search for satisfying solution
  - Without constraints of common sense, results don't solve problem
- Concretization on fixed table size requires taking number of tables into account
  - Scales in exponential-time with regard to number of tables
  - Reduces degrees-of-freedom in solving core problem

Over-constrained



Unable to seat parties at scale: unhappy patrons

Control-theoretic approach to managing scale



Acceptable efficiency and happy patrons

Solve now independent of solution size

Control-theoretic synthesis approach

- Realize unable to control party size, but add a controller to sense and respond
  - May accept some errors
  - Splits up problem at the controller level
- Control theory has been used to regulate errors in physical systems – insight for software systems

Encoding data to protect against noise

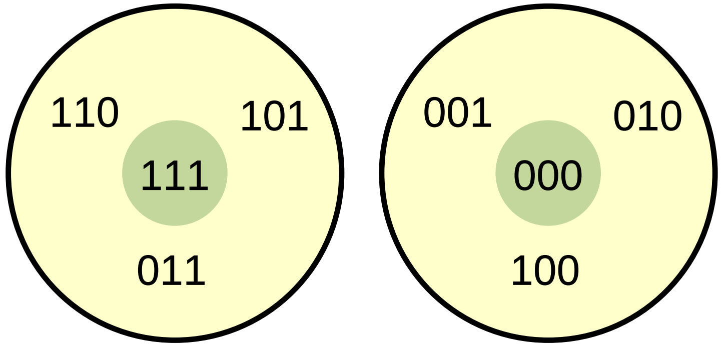
The next few slides describe how, if composition is solved as a sequential series of concretizations, the end result is less optimal (and more computationally intense) than if the concretizations are deferred.

This approach partitions the noise models from a root point, and explores the symmetries inherent in the problem, separate from the scale of solution (i.e., the number of data bits to encode vs. the number of line bits).

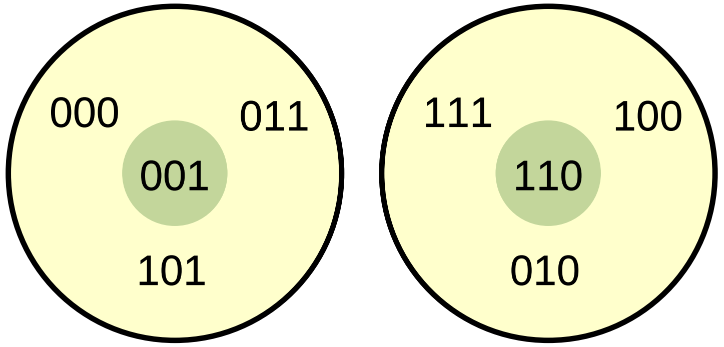
This again begins to hint at the core innovation: when you separate solving the problem, from scaling the problem to its concrete solution, solve time becomes independent of solution size

Weak near field noise: Partitions associated with Hamming 3 codes

- Abstraction: Consider N3 (random flip of up to 1 Bit) on 3 physical bits
  - For any starting point, the result of the action is one of four ending points
  - Example: 000 -> {000, 001, 010, 100}
  - Alternate: 001 -> {001, 000, 011, 101}
- Solve: Choose starting points to ensure no overlap of the resulting ending points
  - 000 -> {000, 001, 010, 100} &
    111 -> {111, 110, 101, 011}
  - Each set of ending points has a different invariant property (e.g., value of majority vote)
- Center points (starting points) are special
  - Invariant is held under 1 more action of the noise
  - Outer points break invariant on next action of the noise (weak symmetry)

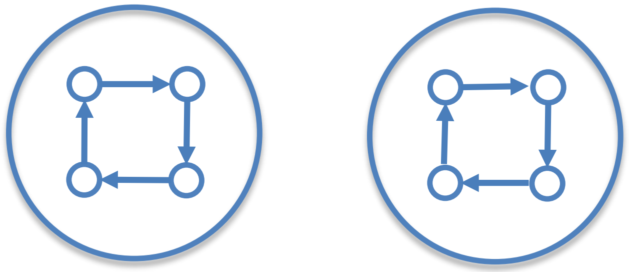


Example partitioning:
Centers 111 and 000

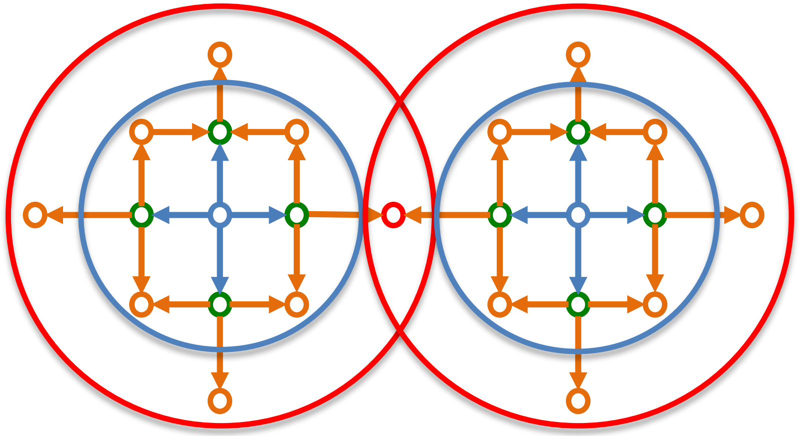


Example partitioning:
Centers 001 and 110

Geometric view of partitions and actions

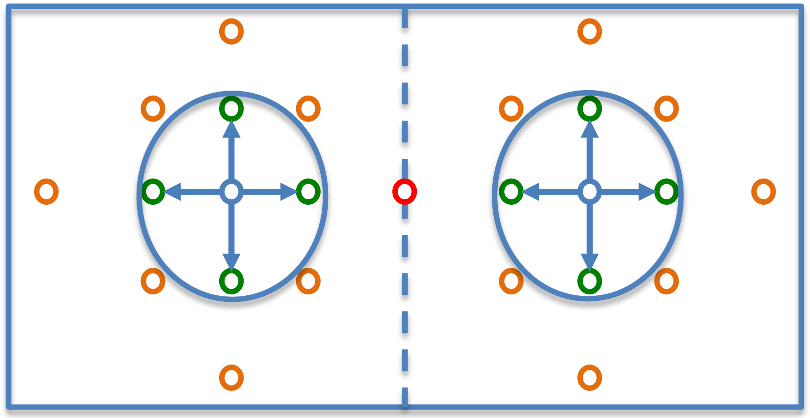


Action with
strong symmetry

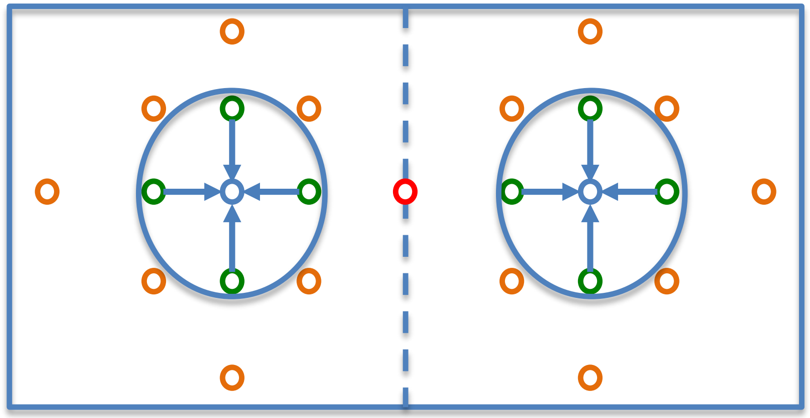


Actions with
weak symmetry

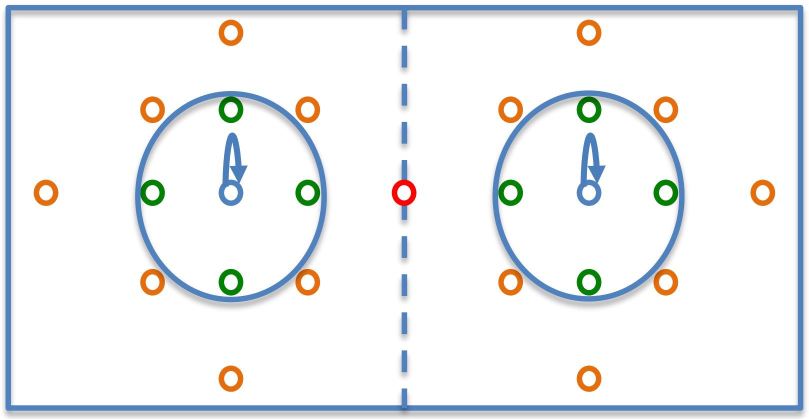
What is the role of a controller?
Strengthen the symmetry abstraction



Limited application with weak symmetry



Application of the controller

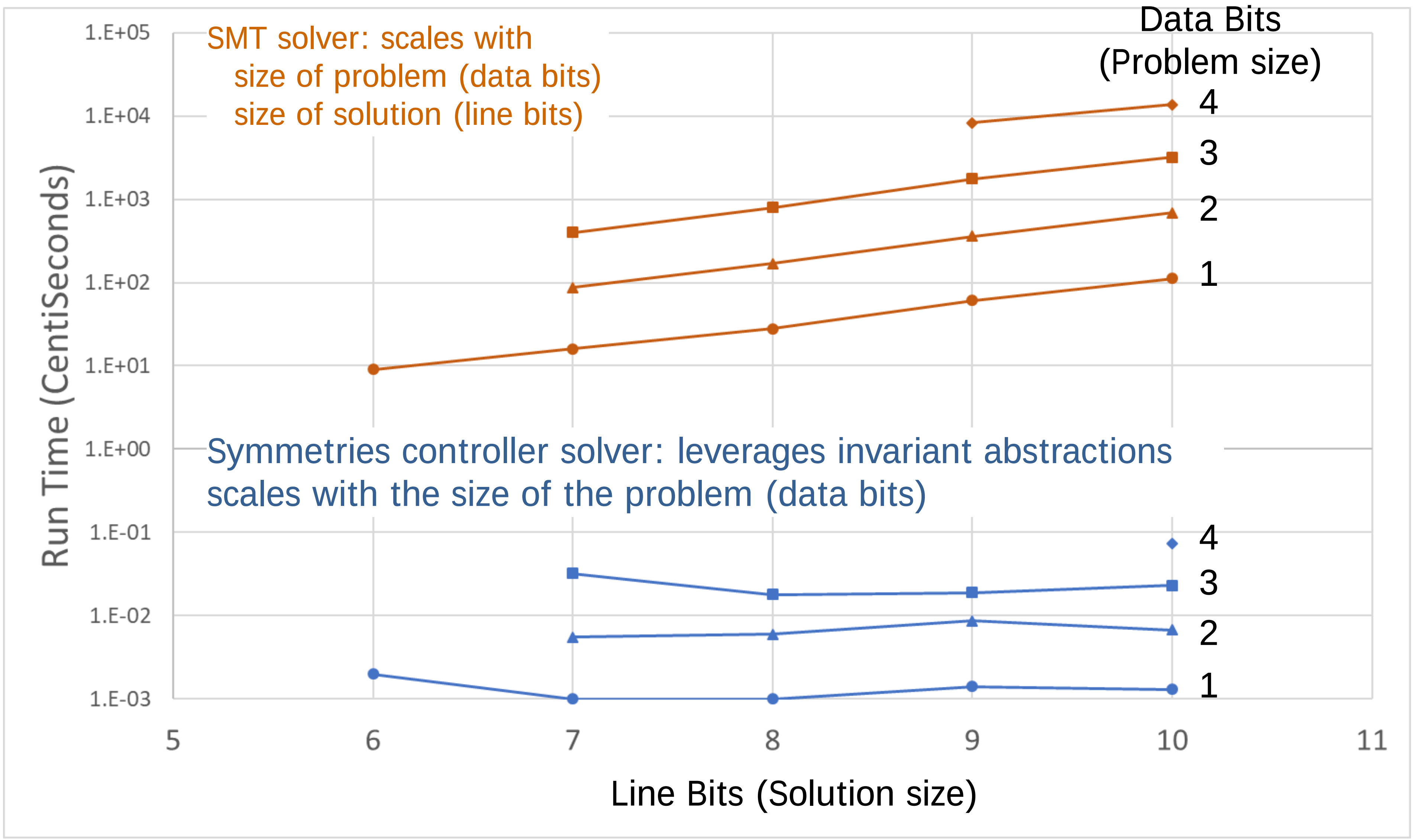


Composition of 1 step application with weak symmetry, then the controller

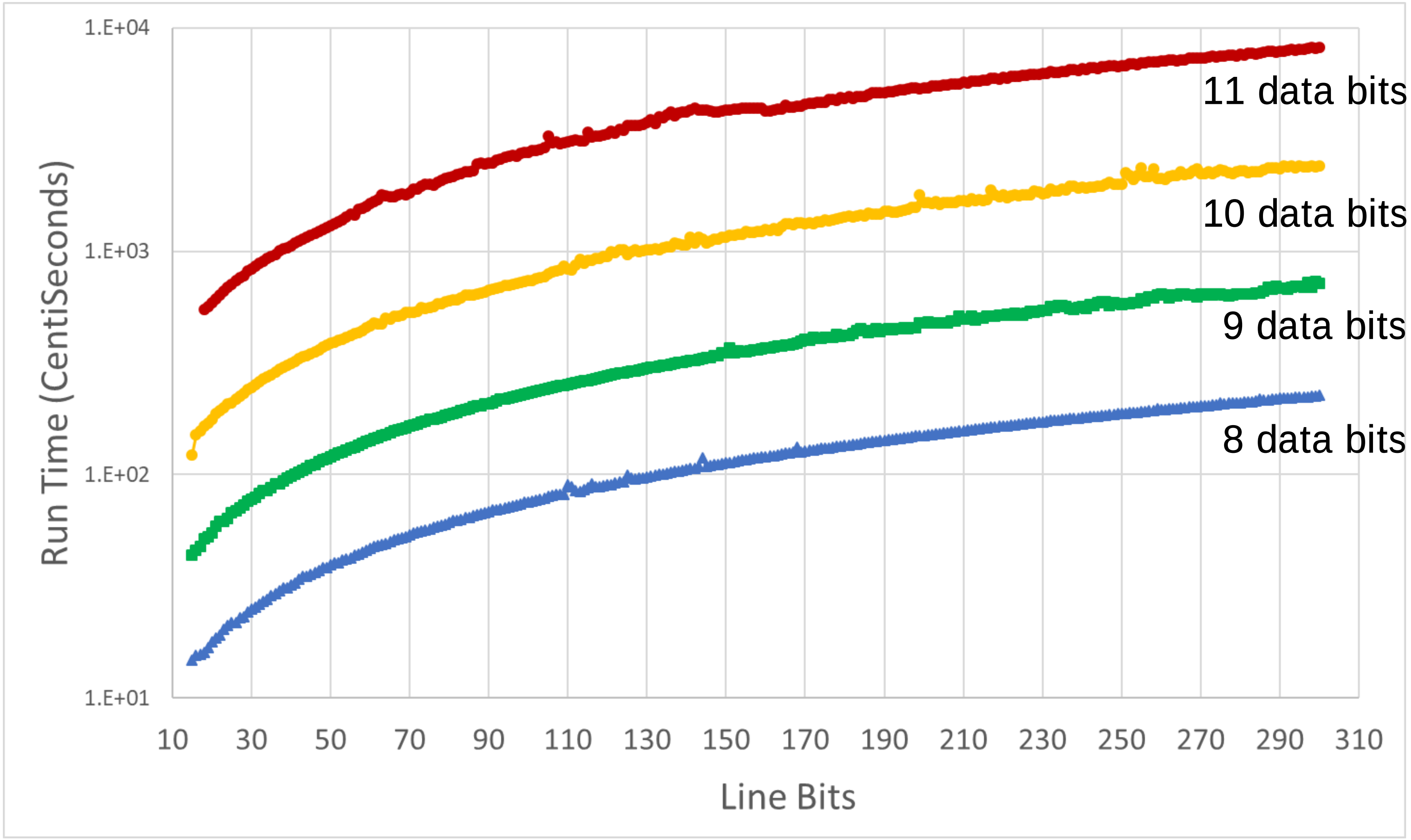
Composing N1 (strong far field) and N3 (weak near field)

- Let S be the space of all possible representations on m wires
- N1: $\{P_1, \dots, P_k \mid \text{each } P_i \text{ is a set of points with } P_i, P_j \text{ disjoint}; \quad \forall\, x \in P_i \to \neg x \in P_i\}$
  - N1 induces a degeneracy that reduces the extent of the space by a factor of 2
- N3: $\{\{e_1, \dots, e_k\} \mid \text{each } e_i \in S \text{ is an encoding point}; \quad \forall\, i \neq j, hamming(e_i, e_j) \geq 3\}$
  - N3 induces an approximate degeneracy (encoding points are special), so no strict reduction of the space is possible
- Solving over enumerated partitions is a packing problem and much more efficient than the SMT formulations that we had before
  - N1N3: $\left\{\{e_1, e_1'\} \dots, \{e_k, e_k'\} \mid e_i \in S_1; \forall i\ e_i' = \neg e_i; \forall i \neq j, H(e_i^{(\prime)}, e_j^{(\prime)}) \geq 3\right\}$
    where $S = S_1 \cup S_2\ and\ S_1 = \neg\, S_2$
- Propagating the N1 symmetry through N3 enumeration gives a more efficient constraint:
  - $H(\neg e_i, e_j) = \mathrm{m} - H\left(e_i, e_j\right)$, so can rewrite $H(e_i^{(\prime)}, e_j^{(\prime)}) \geq 3$ as $3 \leq H\left(e_i, e_j\right) \leq m - 3$
- Which yields, N1N3: $\{\{e_1, \dots, e_k \mid e_i \in S_1;\ \forall i \neq j, 3 \leq H\left(e_i, e_j\right) \leq m - 3\}\}$
  - A single constraint on the half space, S_1, whose solutions always yield a solution (via refinement of states in the partitions) on the full space S

Data shows the controller based solver's benefit of working in the problem size (log scale)

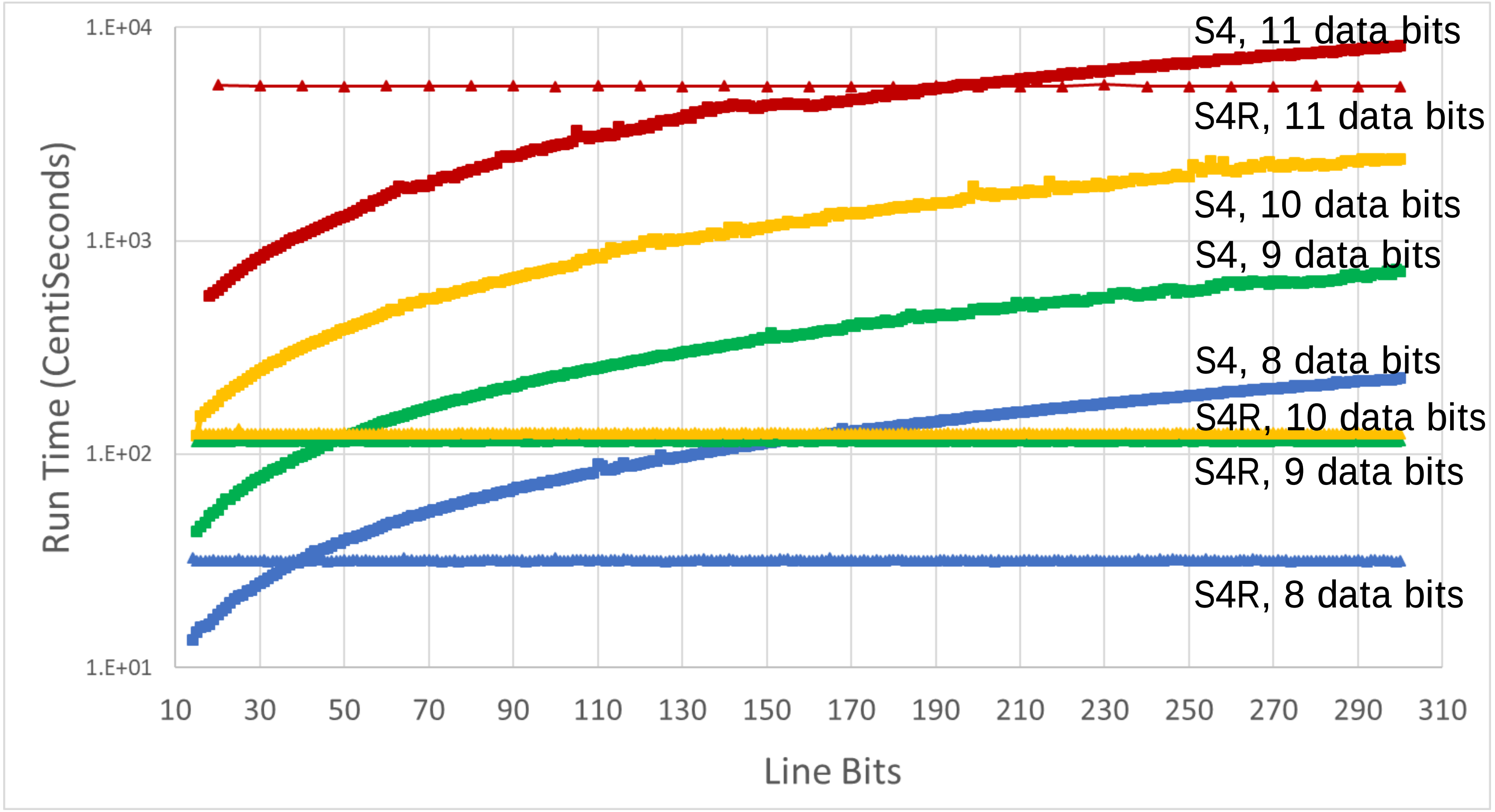


Symmetries solver is in the measurement noise - is it really scaling well with problem size?



Currently, scaling linearly with line bits, but we should be able to do better!

Initial scaling results of S4 Refined (S4R) with additional symmetry statements



Weak symmetries + control theory = strong(er) symmetries?

- Line encoding examples has strong, weak symmetries
  - Articulated strong (N1) and weak (N3) symmetry partitions
  - Showed efficient composition of partitions of strong and weak symmetries
- 3 core claims for data containers example
  - Data containers are weak symmetries
  - Can use control theory to strengthen symmetries
  - Strengthened symmetries will allow efficient composition (provide the benefits of strong symmetries)
- Key question: Can we recover the properties of strong symmetries by layering control theory on top of weak symmetries?
  - Can we project a (strengthened weak) symmetry through another to form a more compact representation?
  - Is there a generator form for a symmetrized weak symmetry?

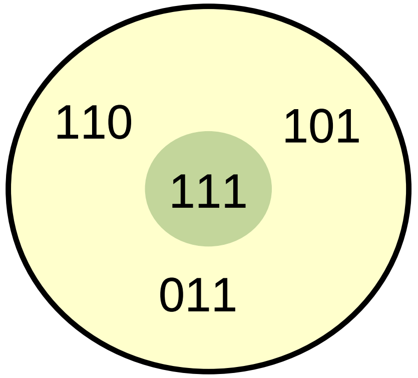
Data container partitions look a lot like N3 partitions

N3: Random flip of up to 1 bit

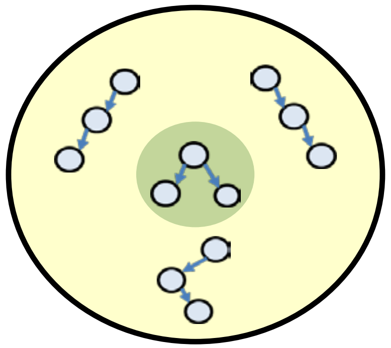
Partitions of asymmetric states

- All points decode to the same value
- Center point is special
- Arbitrary levels of “goodness” for larger Hamming Codes

- Data containers also create Partitions
  - For given data, multiple possible representations
  - Some are “better” than others (performance, etc.)
- Consider a binary tree with 3 data elements (not all points in the partition depicted)



Example partition:
Center is encoding point 111



Example partition for data containers:
Center is a balanced tree

Example problem: Application to store data

Problem statement

- Reason over an unbounded stream of data elements (e.g., integers)
- Insert at maximum rate: MDR
- Provide lookup functionality
  - Return true iff the requested value has been seen in the stream
  - Must return within a bounded period of time
- Can use unlimited number of computational nodes, each with limited resources, but the goal is to minimize the computational nodes used
  - Takes time to spin up new node (N ticks)
  - Basic storage (e.g., binary tree) supports insert and lookup, but grows slower as depth increases (T = N/MDR: max number of insertions to spin up new store)

Is there an efficient way to solve this problem without early concretization nor considering the full state space without abstraction at once ?

- Can we solve via hierarchical structure with a small fixed variables at each level?
- Transform into a state space where you can reason over a smaller space
- Will the resultant solution still be efficient?

Fixed and free variables of the atomic store

Assume an atomic binary tree store implementation is available for reuse

- Provides Add, Lookup and Rebalance functionality
- Lookup time corresponds with the depth of the tree
- Rebalance requires Lock, during which the tree doesn't support Add or Lookup
- Note: This early concretization isn't critical but makes the explanation more simple

Define the Symmetry Distance (SD) for the atomic store

- If the $SD > 0$ the atomic store meets the functional & performance requirements
- If the $SD = 0$ it still meets the requirements, but next addition might move it over the line
- If the $SD < 0$ the atomic store doesn't meet the requirements – i.e., you can't add and lookup within performance bound. This can happen if you lock the store

Simple solutions

- Define Symmetry Distance, SD, to be the worst case number of inserts before lookup will no longer return in specified interval
  - Each basic store reports its SD
  - Collections of stores will use a Composed SD (CSD) to manage state internally

- Case 0: A collection of stores with no locking
  - Define $CSD = \sum SD_j$, if $CSD \leq T$, spin up a new store
  - Delegate each insert to an atomic store; don't insert into stores with $SD \leq 0$
  - This solution works, but it is inefficient as you never balance the atomic stores

- Case 1: A collection of stores with at most one locked
  - Define $CSD = Max_{3rd} \{SD_j\}$, if $CSD \leq T$, spin up a new store
  - Delegate each insert to 2 atomic stores; don't insert into stores with $SD \leq 0$
  - Lock any one store, so at least one store containing the data is always available
  - This solution allows some rebalancing and requires storing 2 copies of the data
  - Solution is efficient if the single store rebalance time is fast enough

Simple case (Case 0): Flock of stores

To handle an unbounded stream of data, we use a variable size flock (array, container, bag, etc.) of stores

- Stream of data has a max data rate: MDR
- Spinning up a New Store takes N ticks
- Define $T = MDR * N$: Maximum number of insertions in the time to spin up a new store

Each store reports its SD_j

Define the Composed SD: $CSD = \sum SD_j$

- If $CSD \leq T$, spin up a new store
- Each composite store add requires an add to an atomic store, don't add to stores with $SD \leq 0$
- No locking, because if you lock a store, its contents are unavailable for lookup

This solution works, but it is inefficient as you never balance the atomic stores. In the next slide, we introduce a version that lets you rebalance

More interesting (Case 1): $k - 1$ locking store, a valid but bad approach

Each composite store add requires an add to k different atomic stores.

- This redundant add ensures that if we lock any $k - 1$ stores, at least one unlocked store will still contain any past value that has been added.
- Add to atomic stores such that you minimize the change to CSD. Again, this is a hard synthesis problem because you have to evaluate the impact of each possible add on the CSD. Don't add to atomic stores with $SDj \leq 0$.

$CSD =$ the number of worst case k redundant adds that keeps $\forall SDj \geq 0$.

- This is a valid constraint but forces an analysis over the full state space
- Must solve a "packing" problem to determine the best use of the SDjs such that you keep k stores with available SD by the last add.

Don't lock more than $k - 1$ stores at a time

- Can lock any store that doesn't move you from a stable point $(CSD \geq T)$ to an unstable point $(CSD < T)$
- Again, this is a valid constraint, but it makes the synthesis hard because you need to evaluate each locking action and determine if the resulting state is okay.

This definition is valid, but not practical as it makes the synthesis problem extremely difficult; See next slide for an alternate approach

$k-1$ locking store continued: A better approach

There is a “better” solution: $CSD = \mathrm{k}^{th}\ Max\ \{SD_j\}$;

- This is not a tight bound, for instance if $k = 2$, then the array of $SDj\ =$ $[6, 1, 1, 1, 1, 1]$ has a CSD of 5 for the original definition, but a CSD of 1 for this “better solution”.
- This CSD is conservative, so it will spin up new stores early because it isn’t properly accounting for the long tail of small SDjs
- This is okay, if we recognize the residual error and compensate for it in the top level controller.

Better CSD partitions the SDj values into three partitions:

- (1) Above kth Max index, (2) the kth Max index and (3) below the kth Max index.
- Note, if you insert into atomic stores in Partitions 3 then you don’t change the CSD. If you insert into partition 3 then partition 1, then you don’t change the CSD unless it is necessary.
- Remember: partition 3 contains the residual errors

Better CSD makes for a simpler add, but must balance the error

- Add to atomic stores such that you minimize the change to CSD: pick elements from partitions (3), then (1) then (2).
- Don’t add to atomic stores with $SDj\ <=\ 0$.
- Locking still has a hard synthesis problem (need to evaluate if the result of locking will result in a stable state)

$k - 1$ locking store continued: An even better approach

There is an even better solution: $CSD = (2\mathrm{k} - 1)^{th}\ Max\ \{SD_j\}$;

- This shifts the role of the CSD such that you add $k - 1$ margin
- Therefore, you can remove any $k - 1$ stores from play and you still are sure that the "better" CSD is still stable.
- Note: Another solution is to only lock in stores in Partition 3 which, by definition can't change the CSD

Now locking synthesis becomes easy

- Before, some combinations of locks would shift you from stable to unstable while others wouldn't (breaks the abstraction and forces reasoning over the full state)
- By adding margin (or partition 3 preference) to the CSD, all combinations of locks are stable, so you don't need to reason over that degree of freedom
- Pick any $k - 1$ stores to lock and rebalance
    - Note, this constraint holds the symmetry of the store, but it doesn't ensure that the locking is helpful! (if you always pick the same store, you won't get any benefit)
    - Need to add a soft constraint on the "expected benefit of rebalance" to break the symmetry of the stores for locking. We use the metric of $inefficiency = depth\_of\_tree - \log_2(\#elements + 1)$

These three examples of the $k - 1$ store hopefully demonstrate how by shifting the definition of the SD, we can create stronger abstractions at the cost of a residual error.

Not all residual errors are the same!

The first error ("better" CSD) doesn't properly account for the long tail of low SDj values.

- This is easily corrected for by creating an asymmetry between Partitions (1) and (3)
- The result is efficient (you use up the extra store capacity) even in the bounded short time scale
- Let's call this form of error an efficient abstraction

The second error ("even better" CSD) adds margin by adding $k-1$ extra atomic stores

- This is a constant cost, so in the bounded short time scale, it can be meaningful
- But as the number of data elements $\rightarrow \infty$, the number of atomic stores $\rightarrow \infty$
- The result is that the fraction of used resources that are inefficient goes to zero.
- Let's call this form of error "asymptotically efficient" abstraction

Some errors add a margin that grows with the optimal resource usage

- Let's call this form of error "inefficient abstraction": this is what we want to avoid

Two better solutions

- Problem: Single locking case may still be inefficient if data is unbalanced
  - Inefficiency in the tree increases with sortedness of the data and data insertion rate, since store can only rebalance 1 Base Store at a time
  - Even in best case, cannot achieve more than 25% efficiency: locking requires 2 copies of the data; also not able to use last row of Binary Tree (half the capacity!)
- Case 2: Collection of Case 1 stores to improve efficiency
  - Collection of Case 1 stores, with add delegated to any 1 store. Note: better to prefer inserting into stores with high efficiency
  - If rate of unbalance is too large, Case 1 stores will become inefficient. If this happens, spin up a new Case 1 store to give stores more time to improve efficiency
  - Efficient composition: *improves* performance without breaking existing abstraction
- Case 3: Pack trees to fill last row of binary tree
  - Collection of heterogenous stores: paired stores (tuple of stores that both contain the same data) and fully packed stores
  - Define $CSD = \sum SD_j$ over the pair stores, if $CSD \leq T$, spin up a new pair store
  - On insert, insert data into any pair store. When pair stores reach $SD = 0$, merge one store into a fully packed store and delete when the fully packed store comes online

Moving beyond the flock of $k-1$ locking Stores for more efficiency (Case 2)

$k-1$ store has a true symmetry (it always acts as a valid store) but it isn't always efficient

- Unbalance (inefficiency) in the tree grows with the MDR and sorted-ness of the data
- Can only rebalance $k-1$ atomic stores at a time. For most cases, $k=2$, so rebalance 1 store at a time
- What if the rate of unbalance exceeds the rate of rebalance provided by 1 lock at a time?

True symmetry of $k-1$ store allows for decoupled solutions that are still efficient!

- Create an flock of $k-1$ stores
- Store data in any of the current stores: they have a true symmetry so can always accept more data and will just spin up more atomic stores (note, better to prefer stores with high efficiency)
- The result is that, if the rate of unbalance is too large, the current $\mathrm{k}-1$ stores will become inefficient
- If this happens, spin up more $\mathrm{k}-1$ stores and insert into them giving the older $\mathrm{k}-1$ stores time to improve their efficiency
- Note, there is no need for predictions or analytic models of efficiency. This is due to the dynamically generated true symmetry of the $\mathrm{k}-1$ store and allows for the decoupled solution

New Algorithm:

- Define inefficiency of the $k-1$ Store = $\sum$[Inefficiencies of the Trees]
- Inefficiency of the tree = tree depth – log2 (1 + # Elements in the tree)
- Add to $k-1$ store that has the lowest inefficiency
- If the inefficiency of that store is high (above a threshold), add another $k-1$ store to the flock

Challenge: Implementation to beat 25% efficiency (Case 3)

25% driven by two factors:

- Not fully packed trees – needed because can't add with $SD = 0$
- Redundantly storing the data – need to support lookup while locked

To fix these, we need to address these two issues

- Form fully packed trees that don't need to be locked so don't need redundancy of the input data
- Do this by using short term redundancy and merging multiple trees to form a fully packed tree

Tree merging requires a new atomic tree function "return tree contents" (RTC) to support merging of trees

- Assume that the tree is tied up while returning its content – so need a redundant storage of data in another tree that isn't locked
- Simple pre-concretization: Use a double tree structure – one locks for RTC function, the other remains available for lookup
- Need to stop adding to the double tree structure during this cycle

Case 3: Structure of the solution

Five partitions:

- `Unlocked` Stores: These are double tree elements that redundantly store each addition
- `AddLocked` Stores: These are stores that are locked for add, but still support lookup
- `ToBeDeleted` Stores: These are stores that have already been merged into a new `FullyLoadedTree`. Need to keep 1 tree for lookup until the `FullyLoadedTree` is complete
- `NewFullyLoadedStore`: This is a partition that contains 1 new tree that is being loaded from other trees. As we are adding to this tree, it isn't yet available for lookup
- `FullyLoadedStore`: These are trees that have $SD = 0$ and are 100% full (zero inefficiency) so never need to be locked. As they don't get locked, no redundant stores are needed

	{ULS}	{ALS}	{TBDS}	{NFLS}	{FLS}
Lookup:	✓	✓	✓	✗	✓
Add:	✓	✗	✗	✗	✗

Case 3: On tick description

Define CSD of {ULS} = $Max\ (SDj)$

If CSD of {ULS} <= $MDR * N$; Spin up more Unlocked Stores

If {ULS} has elements w/ Low SDj, move each to {ALS} Partition

- Don't move Element with $Max(SDj)$, so CSD doesn't change!

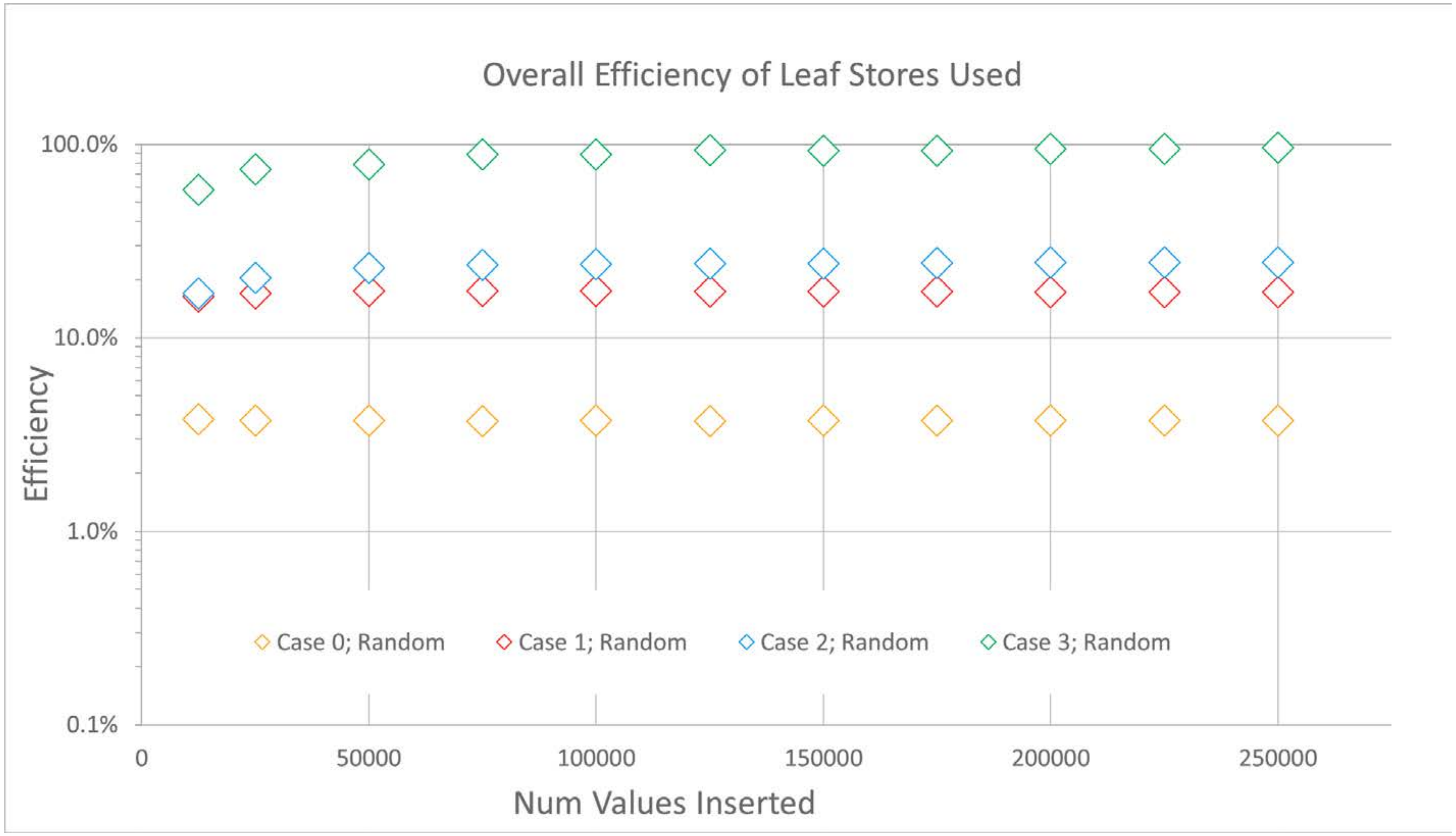
For each element in {ALS} (E_ALS: Double tree store)

- If low efficiency, rebalance subtree 1 then 2, then if efficiency is better, return to {ULS}, otherwise, proceed.
- Read out contents, process each Element (E: data value)
  - Note, double structure of E_ALS allows "read out" while supporting lookup
  - Is E in {NFLS} or {FLS} ? Discard : Add E to {NFLS}
  - If {NFLS} is full ? Rebalance; Move to {FLS}; Delete Elements of {TBDS} : Nothing
- Move E_ALS from {ALS} to {TBDS}; TBDS doesn't need a double tree, so can convert to single tree representation

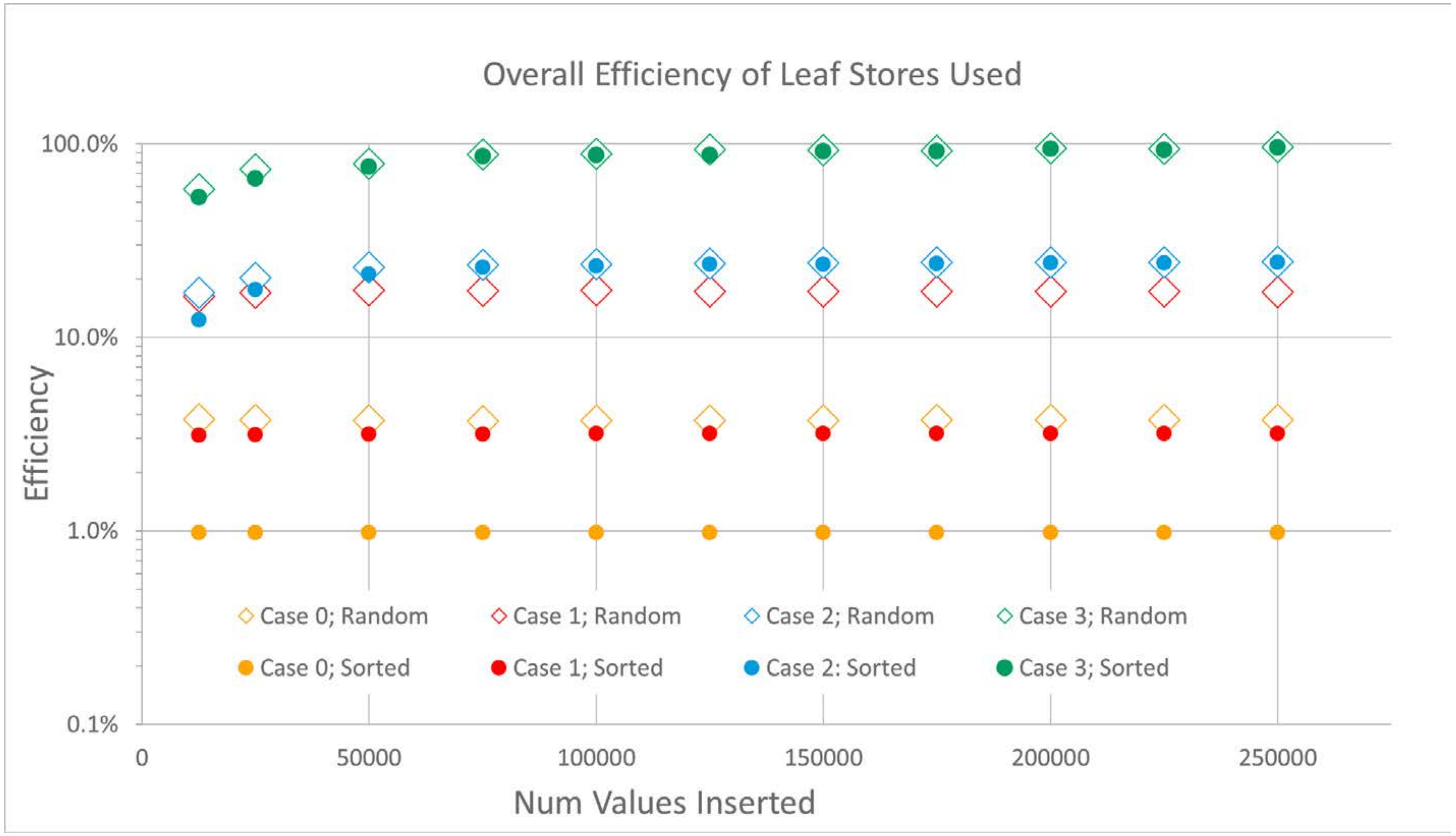
Results is that data ends up in FLS with 100% efficiency

- Transient partitions {ULS}, {ALS}, {TBDS} define asymptotic inefficiency

Performance results: 4 cases on random input data



Cases 2 and 3 are naturally resilient against worst case (sorted) input data



Time for some curveballs

- Changed key requirements on team to measure ability to adapt

- Case 4: Add delete functionality
  - Before repeats didn’t matter
  - Once a tree was fully-loaded and maximally efficient, it would persist as such

- Case 5: Account for transmission times
  - In reality, lookup times in distributed systems depend on physical location
  - Thought this curveball would be significant to adjust to location of nodes from a central application server

Timing a new re-synthesis for Cases 3, 4, and 5

New hire, physics/CS background, 2 weeks “training” on symmetries concepts (novice)

- Asked to implement Cases 3, 4, and 5 in Java and track time on each task

Case 3:

- ~1 day to implement and confirm performance was comparable

Case 4: allow deletions, which breaks a key symmetry (repeats don’t matter)

- Much harder, need to implement “holes” and move $SD = 0$ trees back into add rotation when they become too sparse. Need to be double trees, so reduces efficiency of solution
- ~6 hours (+ 2 hours of PI time) to adapt the symmetry statements
- ~12 hours to implement in Java
- ~5 hours testing

Case 5: data center with asymmetry due to distance to central aggregation

- Key change was to sub-partition the nodes based on their transit time to aggregator (expanding rings, each with a shorter tree to compensate for the transit time)
- ~30 minutes to adapt the symmetry statements
- ~2 hours to implement changes in Java
- ~30 minutes to confirm expected performance

New challenge (Case 4): Add `delete`

In addition to `adds` and `lookups`, the data stream can add `delete` of a value

- Lookup should now return true if the value has been added with no subsequent deletes
- A subsequent `add` should change the lookup value to true until another `delete`

`Delete` remove the invariant that once a value is added its lookup is true for all future time – which was the key symmetry to reach ~100% efficiency

- Fully Loaded Stores (FLS) could be singletons as they never needed to change
- Keeping the assumption that reading out the contents of a store is expensive (so can't do it without affecting lookup times), we now need at least two copies at all times

Multiple options for implementations

- Store `delete` in their own tree of `deletes`
- Store a `delete` bit in the main trees to denote that a prior value in that tree has been `deleted`
- Efficient solutions require that we bound the number of `deletes` that are stored at any given time

Adapting Case 3 partitions for delete to create a Case 4 solution

One solution is to store deletes as “holes” in a tree.

- Each delete request is broadcast to all stores.
- Every unlocked store containing the deleted element marks it deleted.
- Marking a tree node deleted is cheap but increases the store’s inefficiency.

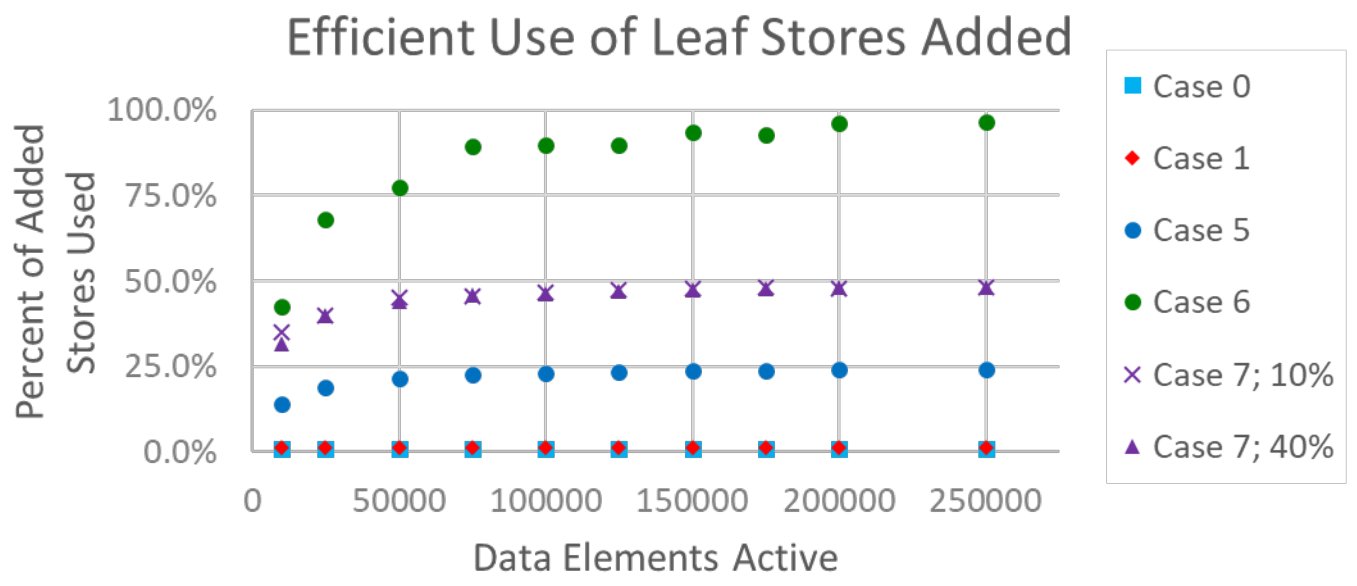
Holes get removed when filling an NFLS.

- No need to change the rebalance function to remove holes, but this is an optional optimization.
- When an FLS gets too hole-ridden, move it back to an ALS for either rebalance or incorporation into a new NFLS

A new delete stores partition (DS) buffers delete requests for locked stores. Locked stores incorporate deletes as holes before unlocking.

- Delete operations cannot be applied to substores being rebalanced, merged, etc. so they must be held elsewhere. Maintain one store per locked substore to buffer its deletes.
- Incorporating holes ensures that an unlocked store never contains inactive elements without knowing they are inactive.

Performance is near the 50% limit



- As two copies of every active element must be stored, Case 4 performance cannot exceed 50%.
  - This implementation achieves >48% efficiency. 50% asymptotic efficiency is achievable by constructing multiple NFLS's in parallel.

- Performance is largely independent to the pattern of deletes. Shown here are deletions of a random active element in 10% and 40% of requests.
  - Also tested long streams of deletes and deleting elements in FIFO and LIFO order.

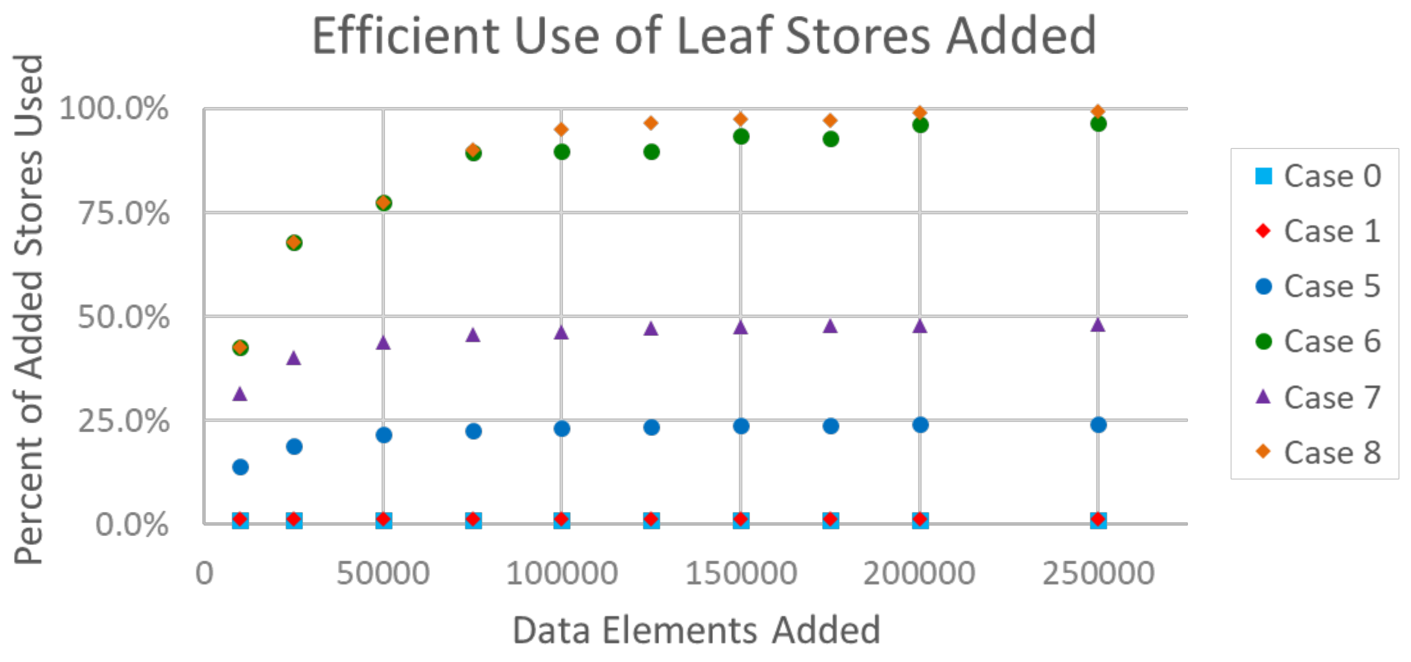
Adapting Case 3 partitions for statically varied transmission times (Case 5)

- When transmission times statically vary by substore, shorten a substore's allowed computation time to compensate for a longer transmission time.
  - The required response time is bounded, so an increase in the worst-case round-trip transmission time between aggregator and substore must be countered by some corresponding decrease
  - Assume per-substore transmission time bounds are known
  - Examples: delays due to physical distance or network topology distance (RTT)

- This generates a partitioning on substores' maximum supported SDs
  - Assumes substores use more computation time given larger max-SDs, e.g., any data store where symmetry distance maps to worst-case time to fulfill a request

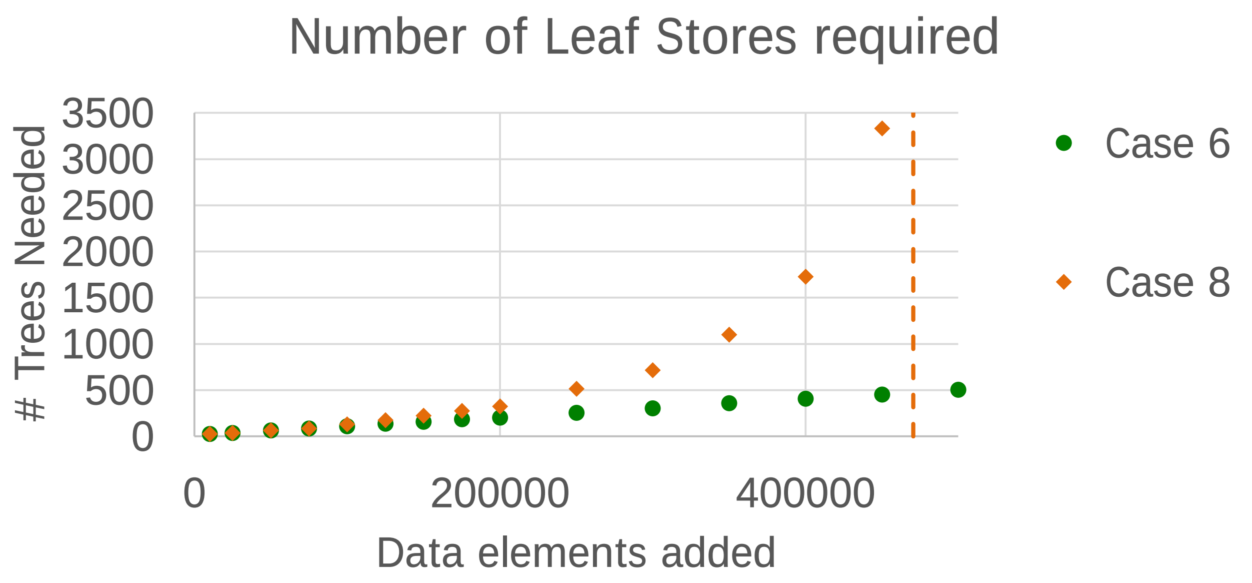
- Compose with Case 3 partitions
  - A good controller optimizes performance by provisioning substores with higher max-SDs first for NFLSes / FLSes and then for other stores.
  - If congestion-induced latency is a concern, organize ALS-to-NFLS merging to avoid increasing request latency.
  - Skip lookups during merge whenever they would increase request latency.

Case 5 achieves optimal asymptotic performance in the accessible storage



- Inputs were in sorted order (worst-case).
- Assumed partition n had π (C n)2 - π [C (n - 1)] 2 = π C2 (2 n - 1) substores, e.g. 2D physical distance forming rings from the aggregator
  - Chose C = 5 and initial tree capacity 210.
- Plotted vertically is “# elements / # possible to store in stores used”.

Case 5 rapidly uses more stores when capacity shrinks



- Inputs were in sorted order (worst-case).
- Assumed partition n had $\pi\,(C\,n)2 - \pi\,[C\,(n - 1)]\,2 = \pi\,C2\,(2\,n - 1)$ substores, e.g. 2D physical distance.
  - Chose C = 5 and initial tree capacity 210.
  - Total capacity is 470K data elements.

Analyzing the fully concretized solution isn’t feasible, but symmetries can help

Model checkers are great for confirming that a system is stable

- But the concretized state space is brutally large, so practically this is totally infeasible
- But the only degrees of freedom (DoFs) that should matter are the symmetry DoFs

By breaking problem up into partitions with nested stability criteria, analysis can be compartmentalized

- Only analyze transitions and constraints on system as it moves between states/partitions
- Marking states that are unsafe allows for rapid analysis to verify that system is stable and that errors are handled at correct level of abstraction

Caveat emptor:

- This type of analysis only verifies stability, not code correctness, nor functional correctness

Analyzing the fully concretized solution isn’t feasible, but symmetries can help

Model checkers are great for confirming that a system is stable

- But the concretized state space is brutally large, so practically this is totally infeasible
- But the only degrees of freedom (DoFs) that should matter are the symmetry DoFs

Consider Case 0 data store example:

- A binary tree of a fixed height has many possible configurations, approximated by

$$Total\ States \sim 1.5^{2^{height-1}}\ (height > 5)^*$$

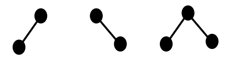
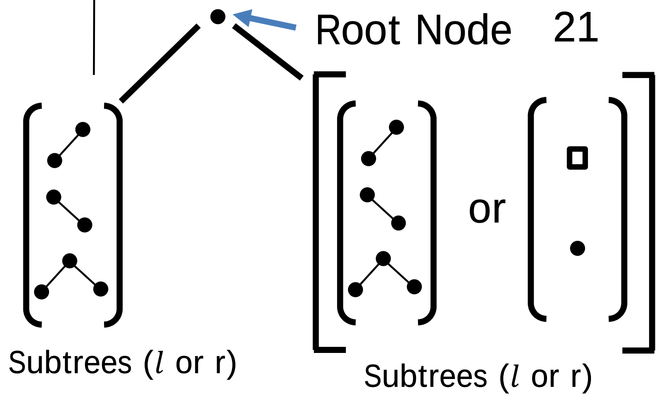
- Each of the S substores are a binary tree with height [0, SD_{max}] or the “bad state” (height > SD_{max})
- In each configuration, a substore may be spinning up with [0, N] ticks remaining (where 0 represents no store spinning up)

$$Total\ States \sim \left(\sum_{h=0}^{SD_{max}+1} 1.5^{2^{h-1}} \right)^S * (N+1)$$

Symmetry statement	Complexity in SD_{max}	Complexity in S	Complexity in N
None	Doubly exponential	Exponential	Polynomial

Full state space of binary search trees

- Any binary tree can be decomposed into 3 parts:
  - A root node
  - A left subtree
  - A right subtree
- Any binary tree of height h must fall into 1 of the following 3 categories:
  - Only left sub-tree has height h – 1
  - Only right sub-tree has height h – 1
  - Both left and right sub-trees both have height h – 1
- This leads to a recursive count of binary trees of height h

Tree Height	Structures	Count
0	□	1
1	●	1
2		3
3	 Root Node or Subtrees (l or r) Subtrees (l or r)	21

$$count(h) \stackrel{\text{def}}{=} c(h) = 2 * \left[c(h-1)\left(\sum_{i=0}^{h-2} c(i)\right)\right] + c(h-1)^2$$

Which approaches $(1.5028368\ldots)^{2^h} \rightarrow \sim 10^{181}$ for height = 10

See http://oeis.org/A001699 for more detail

Full configurations of substores don't matter, only heights of binary trees

Goal: reduce the complexity using symmetry statements

- Symmetry statements capture only the relevant degrees of freedom
- Removing irrelevant degrees of freedom will reduce the complexity of the model-checking problem without decreasing the fidelity of the model

First (and obvious!) symmetry statement

- Each substore is either good or bad
- A stable substore has SD = SDmax - height ≥ 0
- An unstable substore is represented by a single state, SD = SDmax - height < 0
- Tracking substores by SD reduces state space to (SDmax + 1) + 1 = SDmax + 2 states per substore

$$Total\ States = (SD_{Max} + 2)^S * (N + 1)$$

Symmetry statement	**Complexity in SD_{max}**	**Complexity in S**	**Complexity in N**
None	Doubly exponential	Exponential	Polynomial
+ Tree Height Symmetry	Polynomial	Exponential	Polynomial

Heights of individual trees don't matter, only the sum of excess heights (*Composed SD*)

- For top-level data container, only aggregate properties of substores matter
  - E.g. intuitively changing the ordering of stores doesn't matter…
  - On insertion, a substore must be store the data (but which substore doesn't really matter…)
- Define composed SD to be the sum of SD of substores or the "bad state"
  - Data container is stable if composed SD is ≥ 0 (i.e. not in the bad state)
  - Each substore contributes SD in [0, SDmax] or moves composed SD to the "bad state"
  - E.g., for 8 trees of $SD_{max} = 10$, composed SD would be [0, 80] or the "bad state"

$$Total\ States\ = (S * SD_{Max} + 2) * (N + 1)$$

- Composed SD has $S * SD_{max} + 1$ states plus the "bad state" = $S\ *\ SD_{max} + 2$ states

Symmetry statement	**Complexity in SD_{max}**	**Complexity in S**	**Complexity in N**
None	Doubly exponential	Exponential	Polynomial
+ Tree Height Symmetry	Polynomial	Exponential	Polynomial
+ Composed SD Symmetry	Polynomial	Polynomial	Polynomial

Exact value of CSD doesn't matter if $CSD > T$

T is the max CSD margin that may be consumed while new store spins up

- N is the number of ticks required to spin up a new substore
- $T = MDR * N$, is the max number of insertions that may occur
- On insertion, CSD decrements by 1 or 0, so T is the max possible decrease in CSD

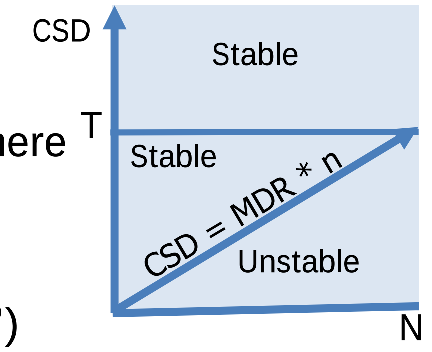
Collapse all states with $CSD > T$ to a single state

- If $CSD > T$, data store will remain valid while new store comes online
- Possible states for CSD are [0, T] or ($CSD > T$) or "bad state"

$$\begin{aligned} Total\ States &= ((T + 1) + 1 + 1) * (N + 1) \\ &= (MDR * N + 3) * (N + 1) \end{aligned}$$

Symmetry statement	**Complexity in SD_{max}**	**Complexity in S**	**Complexity in N**
None	Doubly exponential	Exponential	Polynomial
+ Tree Height Symmetry	Polynomial	Exponential	Polynomial
+ Sum of Excess Heights Symmetry	Polynomial	Polynomial	Polynomial
+ Collapse CSD > T Symmetry	Constant	Constant	Polynomial

Exact values of N, CSD don't matter, only relative values at new substore spin-up



Data container is stable if the CSD cannot go below 0

- For $CSD \leq T$, data container will be stable if $CSD \geq MDR * n$, where n is the time until the new substore comes online

Re-divide the space for $CSD \leq T$ along the $CSD = n * MDR$ axis

- Datastore is unstable if $CSD < 0$ regardless of spin-up state ("Bad")
- Datastore is stable if $CSD > T$ regardless of spin-up state ("Stable")
- Data Store is stable if $CSD \geq n * MDR$ if spinning-up ("also Stable")
- Data Store is unstable if $CSD \leq n * MDR$ if spinning-up ("possibly unstable")

$$Total\ States: \{CSD > T\}, \{CSD \geq n * MDR, spin\}, \{CSD \leq n * MDR, spin\}, \{CSD \leq 0\}$$

Symmetry statement	**Complexity in SD_{max}**	**Complexity in S**	**Complexity in N**
None	Doubly exponential	Exponential	Polynomial
+ Tree Height Symmetry	Polynomial	Exponential	Polynomial
+ Sum of Excess Heights Symmetry	Polynomial	Polynomial	Polynomial
+ Collapse CSD > T Symmetry	Constant	Constant	Polynomial
+ Store Spin-Up Symmetry	Constant	Constant	Constant

Symmetries collapse complexity to constant time

For Case 0, symmetries reduced complexity to a constant state space

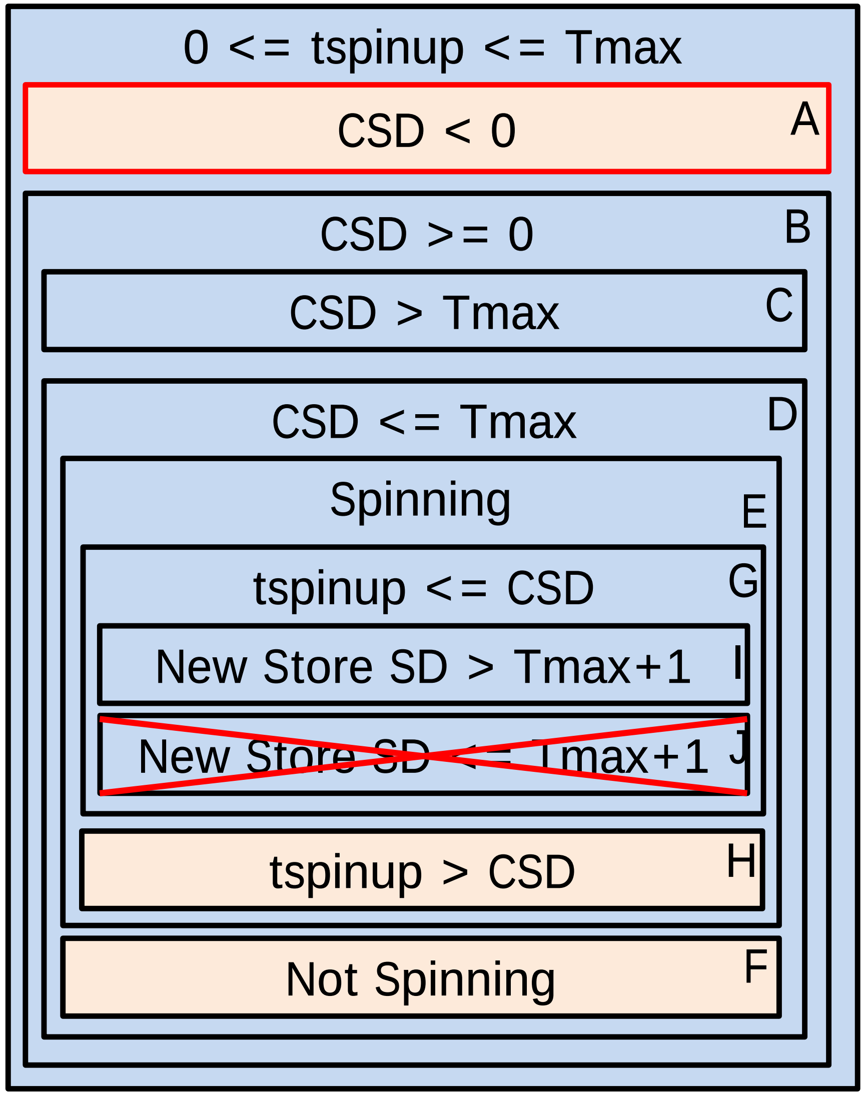
- Initial state space representation was doubly exponential
- Final state space representation had 4 states:
  $\{CSD > T\}, \{CSD \geq n * MDR, spin\}, \{CSD \leq n * MDR, spin\}, \{CSD < 0\}$

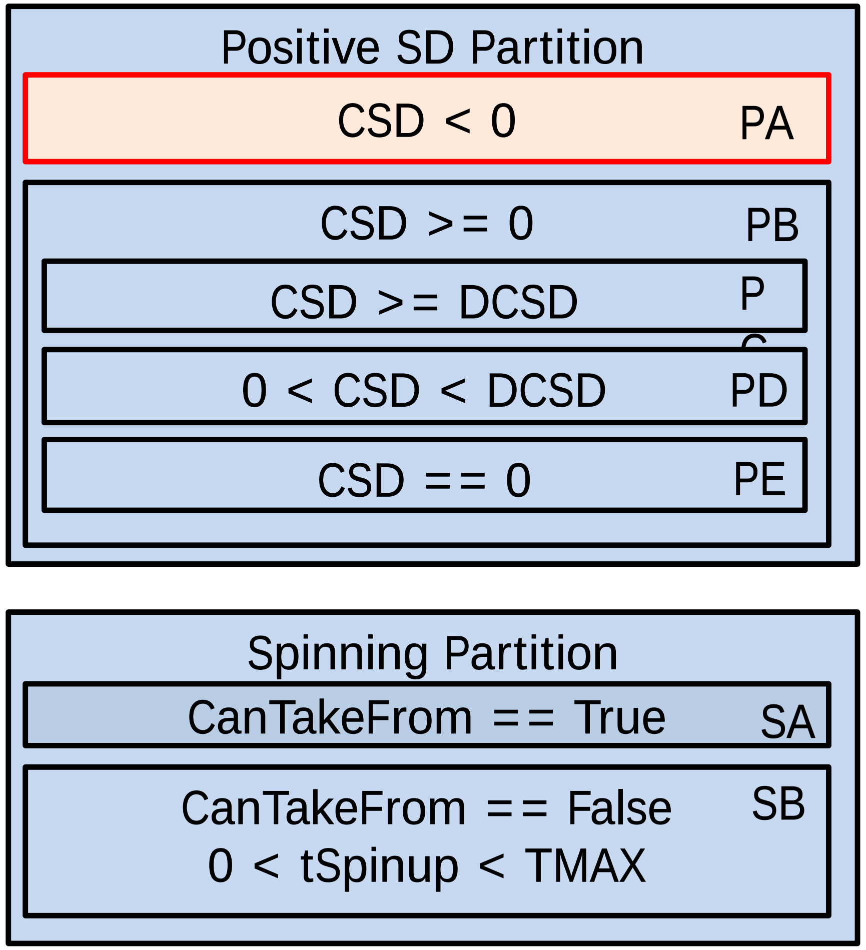
Is looking at the full complexity even necessary?

- Example demonstrated value of symmetries approach, but modeling full state space of binary trees seems unnecessary
- Can we use symmetries to start with, i.e. start with the symmetries and concretize state space only as required? (Hint: yes!)

Symmetry statement	**Complexity in SD_{max}**	**Complexity in S**	**Complexity in N**
None	Doubly exponential	Exponential	Polynomial
+ Tree Height Symmetry	Polynomial	Exponential	Polynomial
+ Sum of Excess Heights Symmetry	Polynomial	Polynomial	Polynomial
+ Collapse CSD > T Symmetry	Constant	Constant	Polynomial
+ Store Spin-Up Symmetry	Constant	Constant	Constant

Analysis can be broken up along partition boundaries





DCSD: default CSD (how much initial storage)

DARPA

www.darpa.mil