DYNAMIC RESOURCE SCHEDULING OF JUPYTER NOTEBOOKS AT CELL-GRANULARITY

By: Louis Jenkins

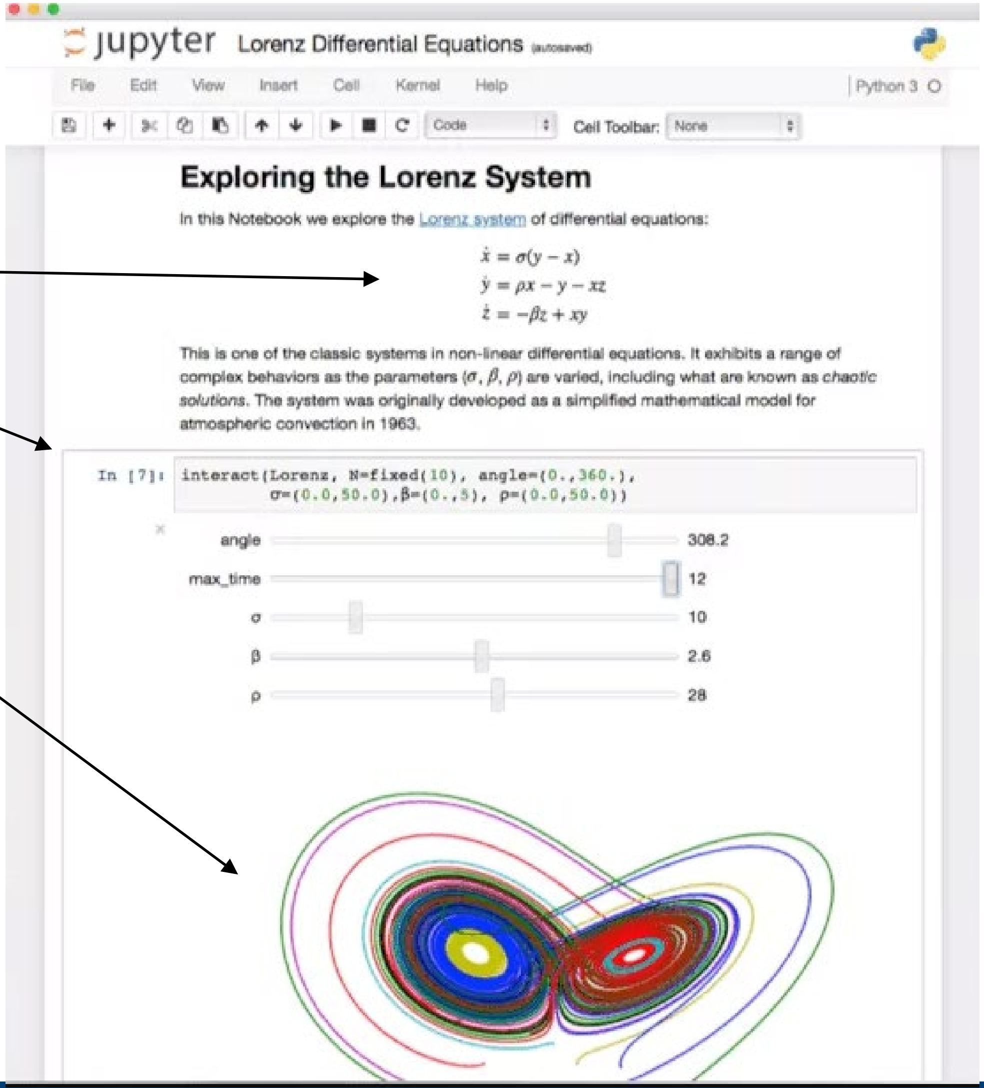
UNIVERSITY of ROCHESTER

Disclaimer

- Research Topic drastically changed in early 2022…
  - Prior work was in Persistent Memory
    - (Coincidence) Intel's Optane PMM was 'killed off' in Q2 2022 (July 28th)

What are Jupyter Notebooks?

- Web App that enables creation and sharing of:
  - Code (Python, Julia, R, etc.)
  - Equations (LaTeX)
  - Visualizations (Graphs and Figures)
- Used in many domains
  - Data Science, Computational Science, ML, etc.
- Run on many types of hardware
  - Laptops and Desktops
  - Cloud Computing (Google Collab)
  - Supercomputers (NERSC)



* Source: juptyer.org

Jupyter Notebook Cells

- Blocks of code in a Jupyter Notebook are called *cells*
  - Cells can be created, modified, and deleted at any time
  - Cells can be run any number of times in any arbitrary order
  - Cells must be run serially (no two cells can run in parallel)
  - Cells can be followed by arbitrary periods of idleness, referred to as *think time*
  - Cells can vary wildly in terms of system resource requirements

Jupyter Notebook on Supercomputers

- Jupyter Notebooks on Supercomputers pose interesting questions
  - Job allocations on supercomputers typically are exclusive by nature
    - "What happens if a Notebook has a lot of 'think time' or uses small bursts of computation?"
  - Supercomputers use static and coarse-grained batch allocation schemes
    - "What happens to unused resources when a long-running cell is not using it? What happens to unused cores when running a single-threaded cell? What about GPUs?"
  - Supercomputers often have long queue times for allocations
    - "What happens if a data scientist wants to just quickly generate results for analysis? What is an acceptable level of responsiveness for an 'interactive' application?"

Proposed Solution

- Co-Allocation of Jupyter Notebooks (Shared Allocation)
  - Jupyter Notebooks share allocations of resources to offset under-provisioning
- Allocate Resources at the beginning of Notebook Cells
  - Only provide a notebook cell what it needs, no more and no less
- Design a specialized scheduler (“Dynamic Scheduler”) for Notebooks
  - Optimized scheduling solution for Notebooks that factors in ‘think time’

Evaluation Criteria for a Scheduler

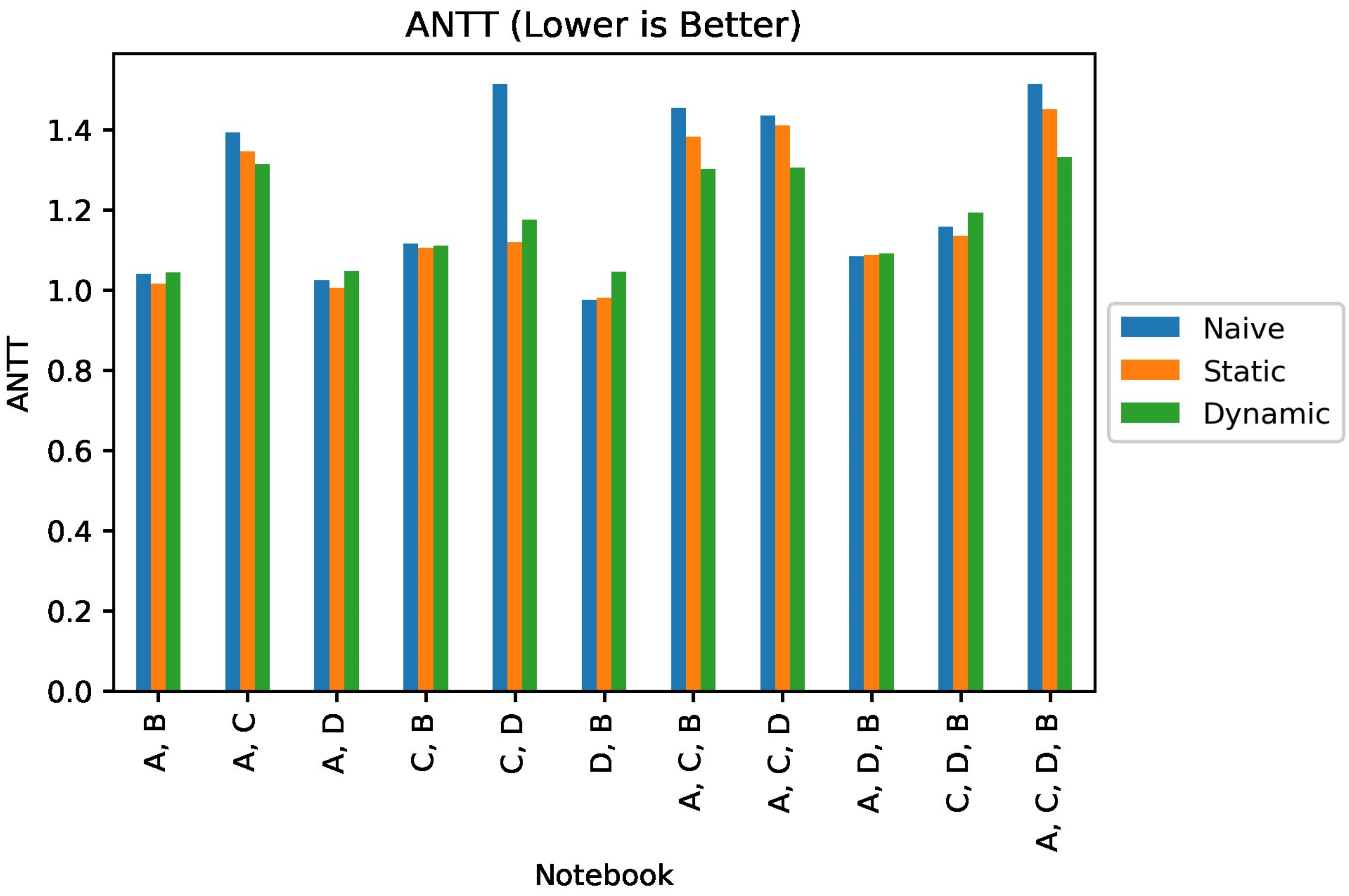
- Utilize two metrics to assess capability of scheduler
  - Averaged Normalized Turn-Around Time (ANTT) [Minimize]
    - Used for evaluating performance of individual Notebooks
  - System Throughput (STP) [Maximize]
    - Used for evaluating performance of entire system.
- Dynamic Scheduler compared to two other schedulers
  - “Naïve” Scheduler – Defer to OS Scheduler
  - “Partitioned” Scheduler – Evenly Partition and Defer to OS Scheduler
- Dynamic Scheduler makes decisions at the boundaries of Notebook cells
  - Determine # of CPUs, # of GPUs, amount of memory, etc. via offline traces
  - Allocate Resources at beginning of cell, De-Allocate at end of cell

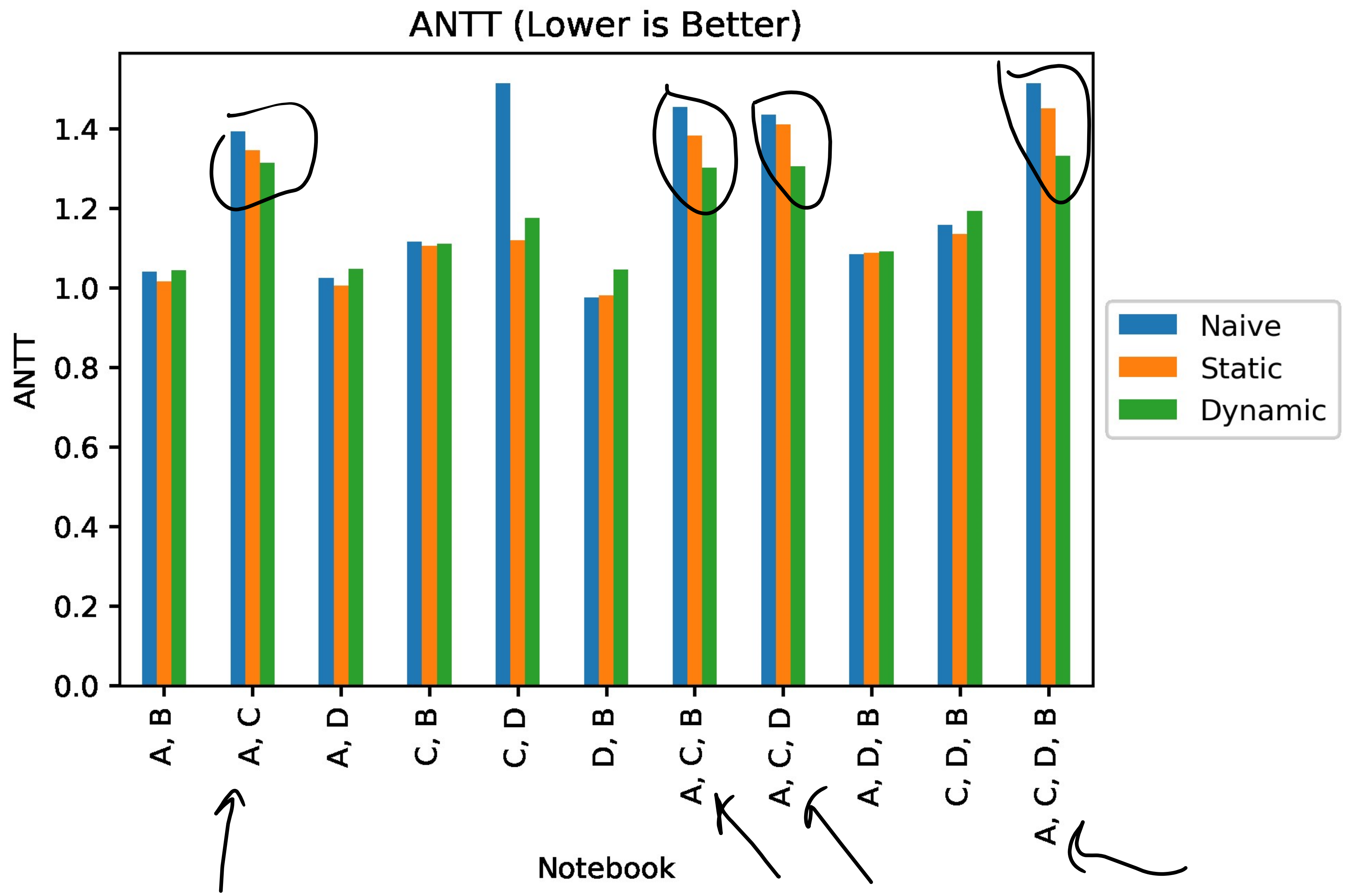
Best, Average, and Worst Case

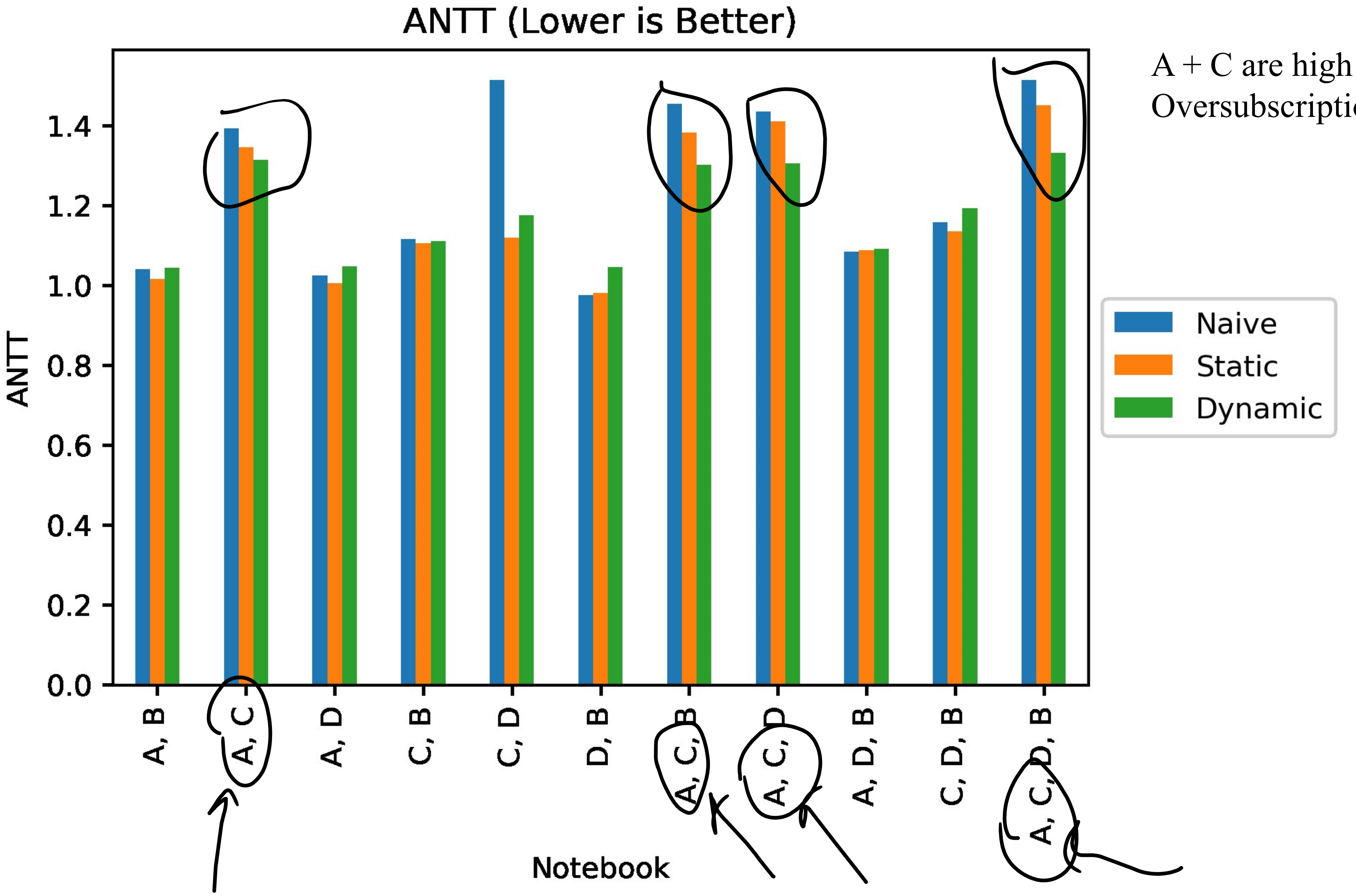
- Custom Scheduler for a Jupyter Notebook would need to handle:
  - Worst-Case: No Idle Time, Non-Interactive
    - Running pre-written Notebook from top-to-bottom
  - Average-Case: Sporadic but ‘Realistic’ (injected) Idle Time, Semi-Interactive
    - Notebook that has an attentive active user
  - Best-Case: Exaggerated Idle Time
    - “Oops, I left my Notebook running overnight”
- Goal: Need Target Application for each of these cases

Worst Case – Target Application #1

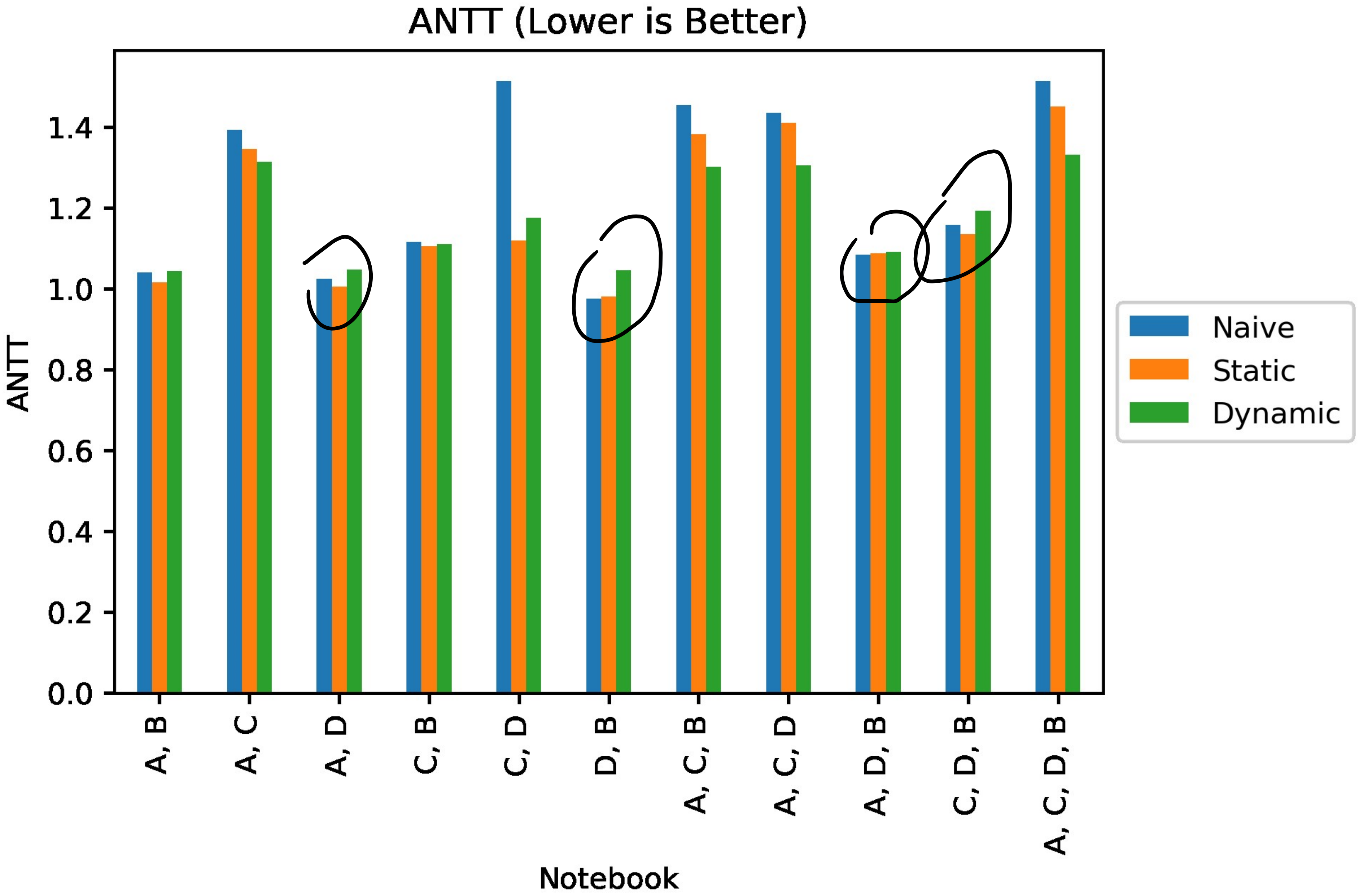
- Experiment includes a set of Machine Learning Notebooks (handson-ml2)
  - Notebook A – Cycles of High-to-Low Compute (single or all cores)
  - Notebook B – Low Compute (primarily single core)
  - Notebook C – High Compute (primarily all cores)
  - Notebook D – Low Compute (primarily single core)
- Combinations of Notebooks are run at least N times
  - Combinations: A,B,C,D,AB,AC,AD,BC,BD,CD,ABC,…
    - Example (N=3, ABCD): A ran 12 times, B ran 8 times, C ran 4 times, D ran 3 times
  - Goal: Explore how the schedulers handled these various workloads
  - Side-Note: Notebooks were ordered by "How long they took to run"

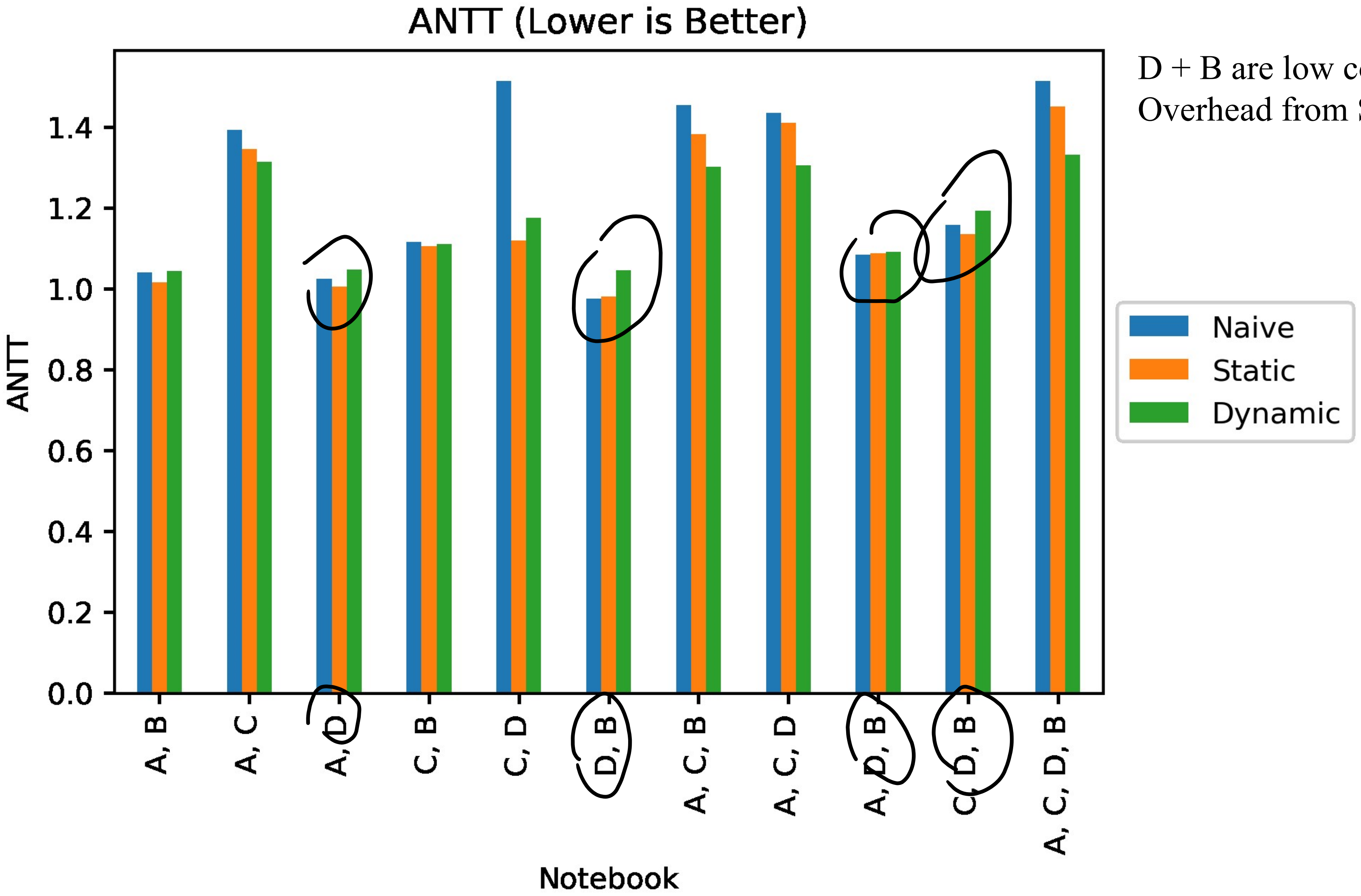

ANTT (Lower is Better)
ANTT
Notebook
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Naive
Static
Dynamic


ANTT (Lower is Better)
ANTT
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Notebook
Naive
Static
Dynamic

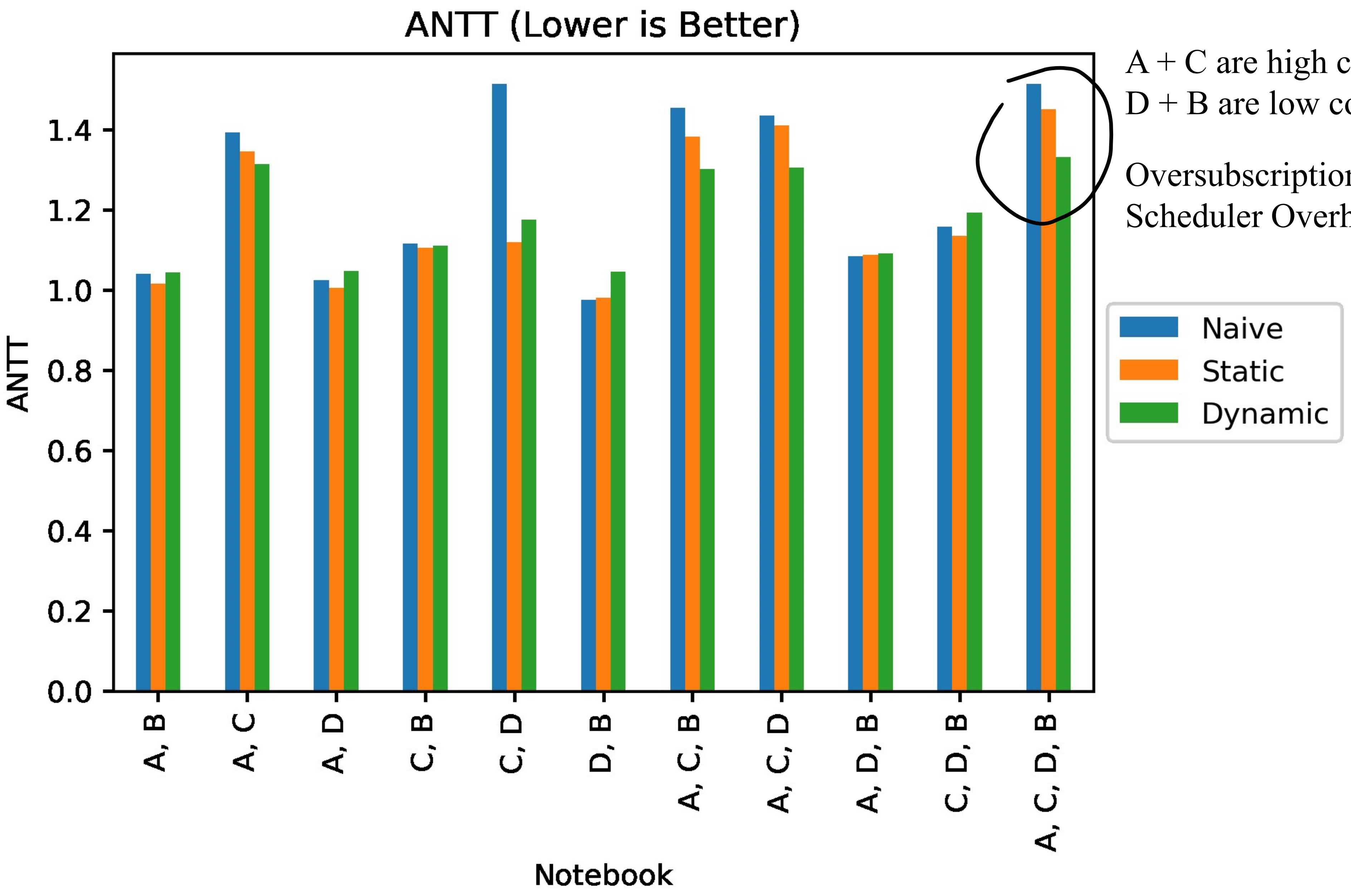


A + C are high compute!
Oversubscription Avoidance!


ANTT (Lower is Better)
ANTT
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Notebook
Naive
Static
Dynamic

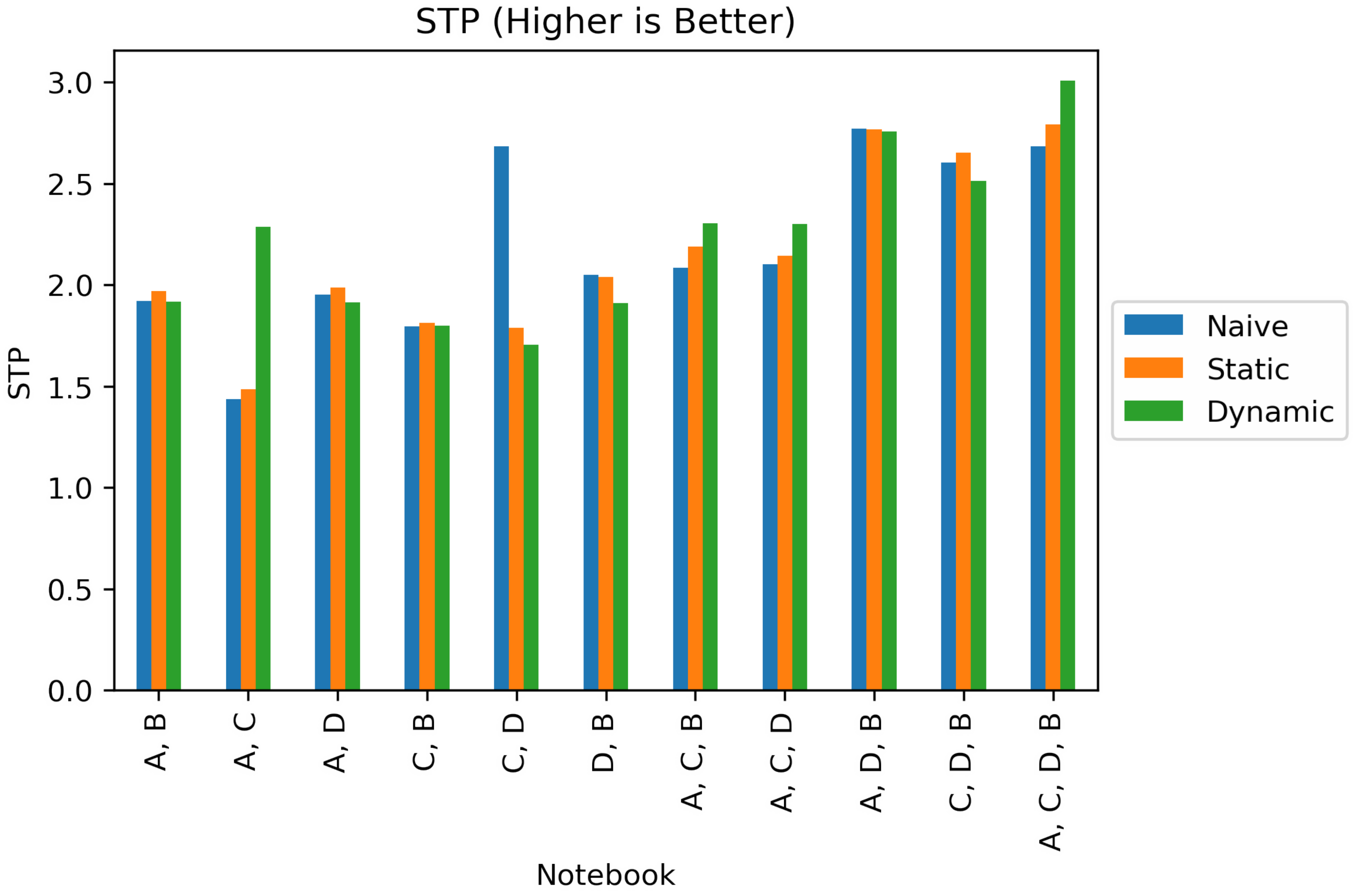


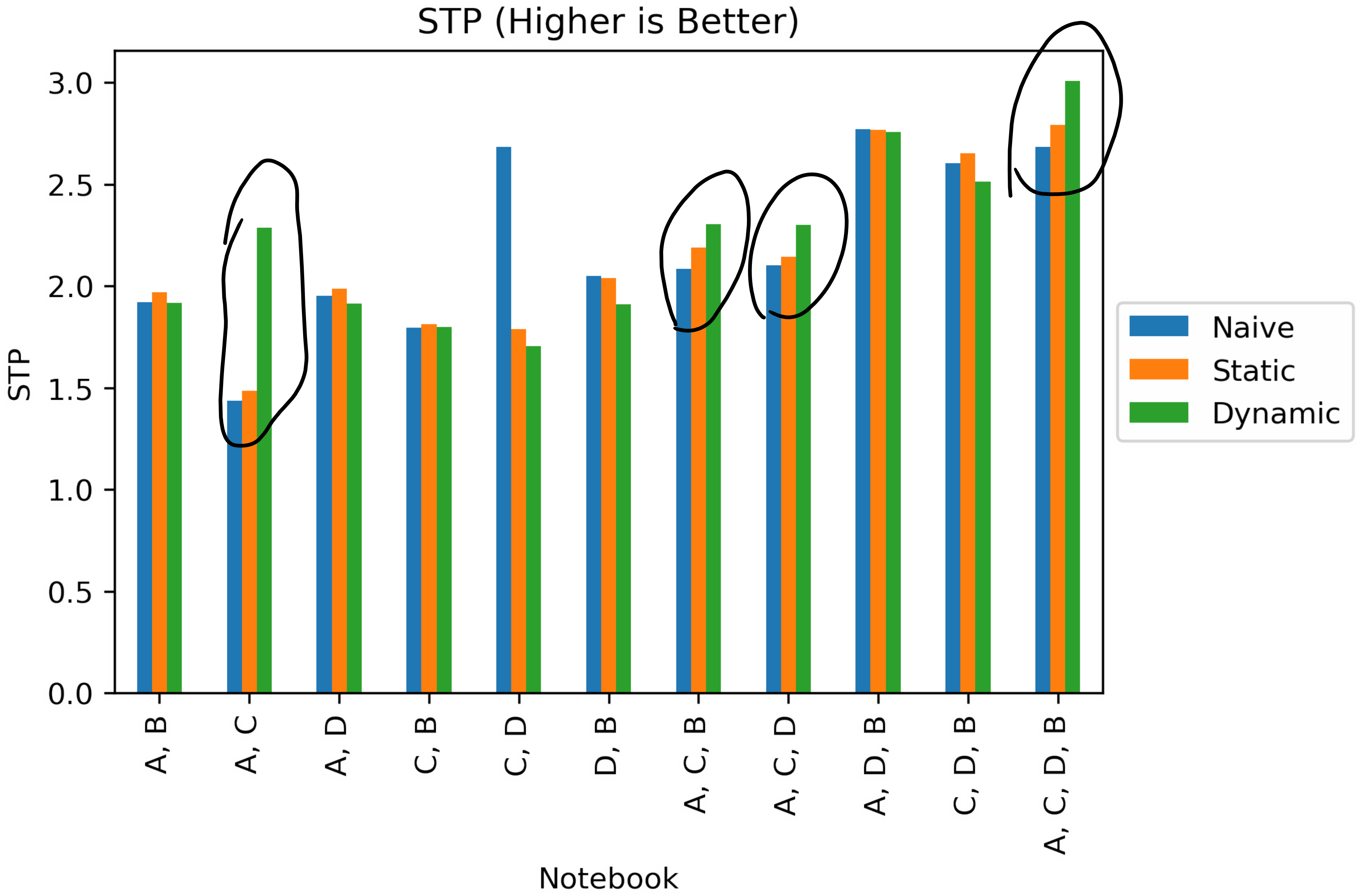
D + B are low compute!
Overhead from Scheduler!

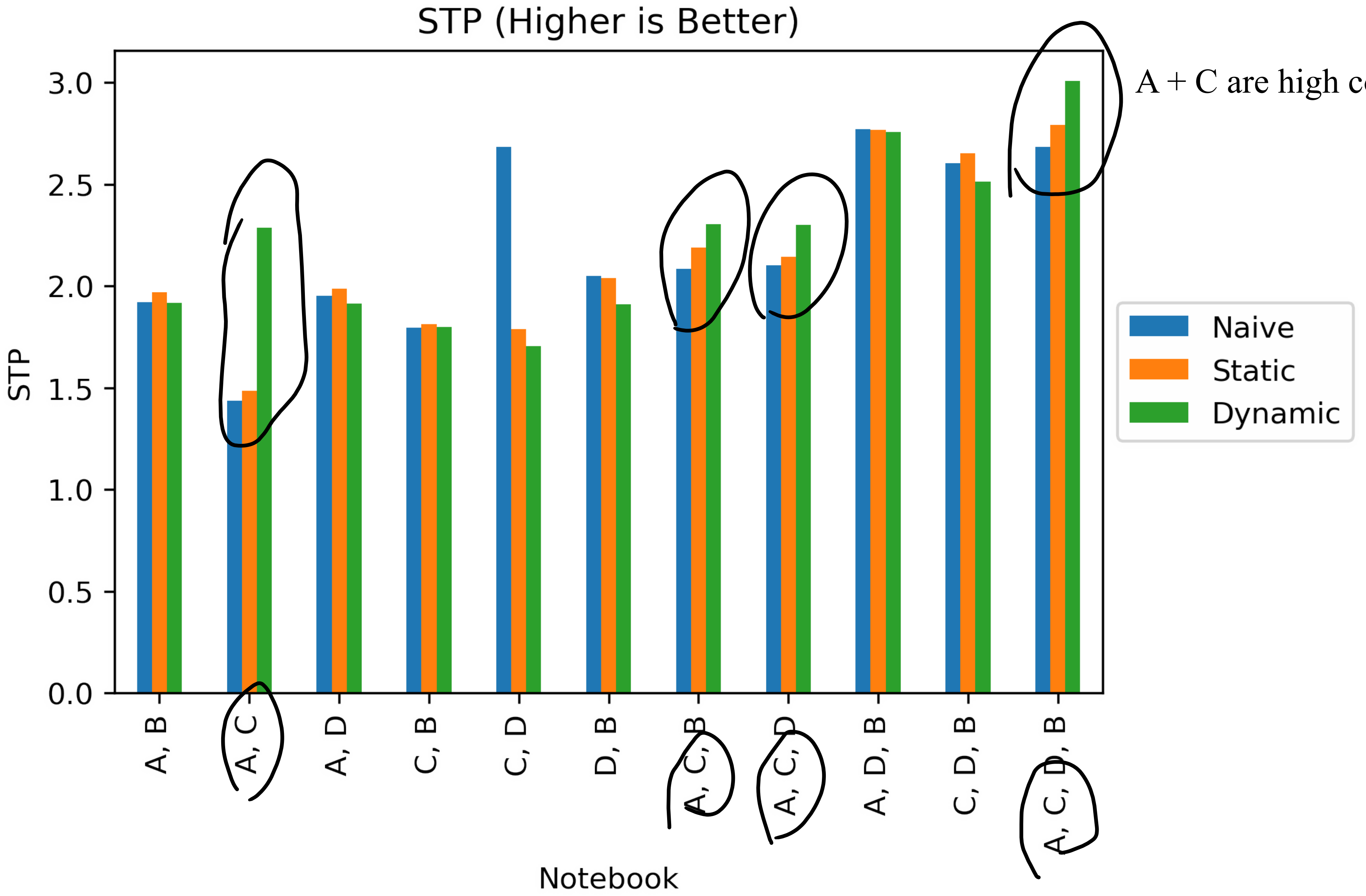


A + C are high compute
D + B are low compute

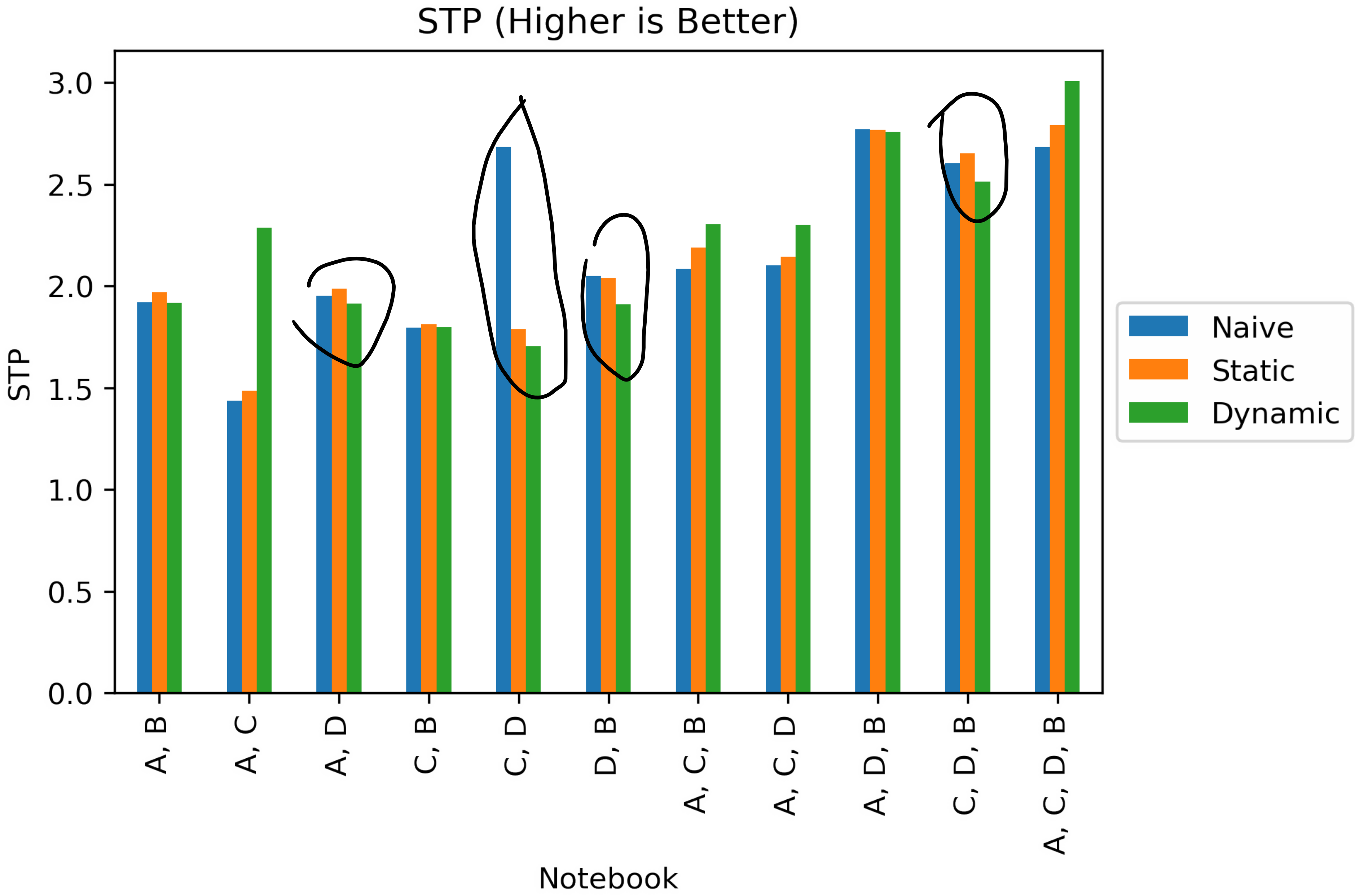
Oversubscription Avoidance > Scheduler Overhead

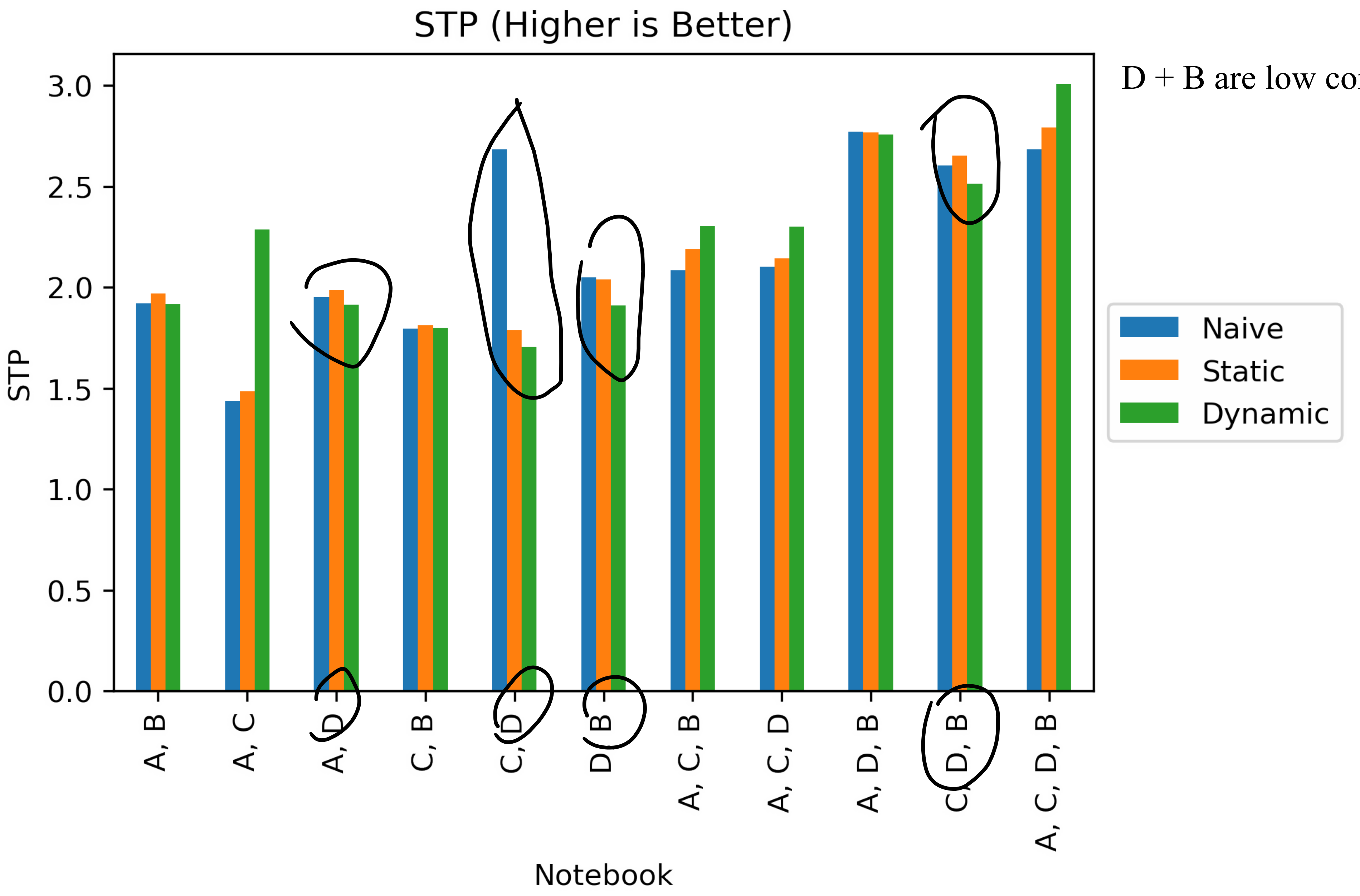

STP (Higher is Better)
STP
3.0
2.5
2.0
1.5
1.0
0.5
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Notebook
Naive
Static
Dynamic


STP (Higher is Better)
STP
3.0
2.5
2.0
1.5
1.0
0.5
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Notebook
Naive
Static
Dynamic



A + C are high compute!


STP (Higher is Better)
STP
3.0
2.5
2.0
1.5
1.0
0.5
0.0
A, B
A, C
A, D
C, B
C, D
D, B
A, C, B
A, C, D
A, D, B
C, D, B
A, C, D, B
Notebook
Naive
Static
Dynamic



D + B are low compute!

Take-Away

- Dynamic Scheduler has benefit of oversubscription avoidance
  - High-compute notebooks seem to show some improvement
  - Has high degree of overhead (visible in undersubscribed systems)
- Native Scheduler has benefit of handling undersubscription
  - Has no overhead
- Static Scheduler is mostly in-between both
  - No dynamic oversubscription avoidance, poor handling of undersubscription

Average Case: Target Application #2

- Arkouda is an HPC-class pseudo-replacement for NumPy/Pandas
  - Python-Frontend Client which communicates with Chapel-Backend Server
  - Replaces imports to NumPy/Pandas with Arkouda's wrappers
- "Poster-Child" for Interactive HPC
  - Python Front-End usable from a Jupyter Notebook
  - Server is allocated across multiple nodes on the backend server
    - Idleness of Server is based on idleness of Client
  - Real active users make use of the application on real supercomputers
    - Provides opportunity to collect data about real applications, including **idle time**

Arkouda Trace Logs

- Obtained Trace Logs from real users
  - Order of operations, operands, time taken, memory consumed, **idle time**
- Provides trace logs for 4 compute clusters
  - SL (Legacy) [7/13/2021 – 6/27/2022] {<320 Nodes, 64 Cores} 100M+ Rows
  - BB [12/15/2021 – 3/8/2022] {1 Node, 64 Cores} <1M Rows
  - NC [2/25/2022 – 9/14/2022] {<40 Nodes, 64 Cores} <20M Rows
  - SE [4/1/2022 – 7/8/2022] {<60 Nodes, 64 Cores} <4M Rows
- Aggregate Statistics
  - Out of 130M cells, 1.9M had ‘think time’ (~1.5%)
  - Mean think time was ~15s (max of 40,000s, std of 696s)

Future Goal: Average + Best Case Analysis

- Worst-Case: Non-Interactive Notebooks (no idle time)
  - Semi-Explored with Machine Learning Notebooks
- Average-Case: Semi-Interactive Notebooks (‘realistic’ idle time)
  - Injection of idle time based on trace records
- Best-Case: Fully-Interactive Notebooks (exaggerated idle time)
  - Injection of arbitrarily long idle time