On the Power of In-Network Caching in the Hadoop Distributed File System

ERIC NEWBERRY, BEICHUAN ZHANG

Motivation

- In-network caching
  - Lots of research has been done about ICN/NDN caching, but mostly using synthetic traffic.
  - How much benefit is there for real applications?

- HDFS is a distributed file system for large-scale data processing
  - Used in many big data systems, e.g., Apache Spark, Apache
  - Deployed in many large clusters in production
- A promising application for ICN/NDN
  - In-network caching
  - Multipath, multi-source data transfer
  - Resiliency
  - Security

Research questions and approach

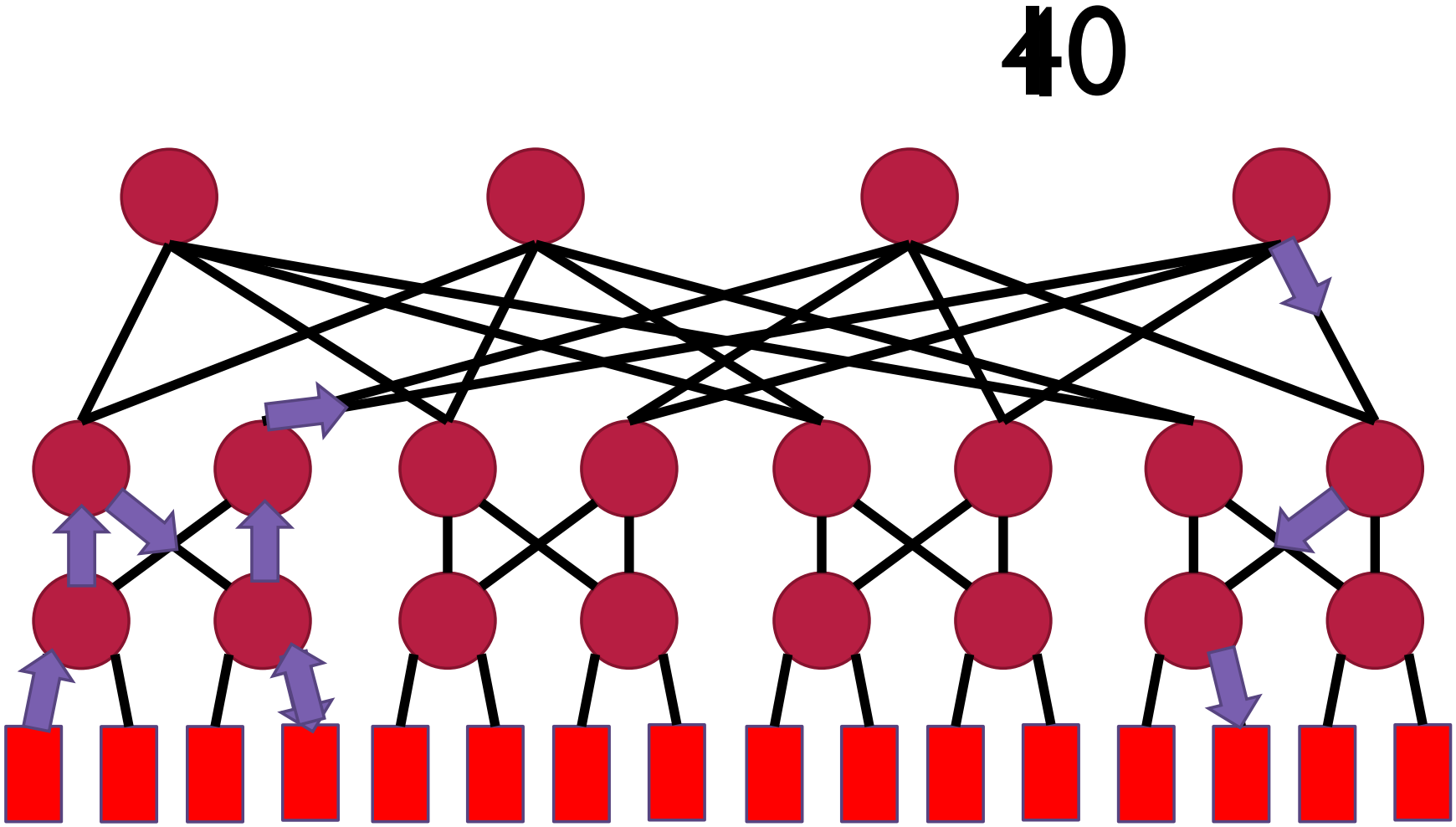
- Does in-network caching benefit HDFS applications? If so, how much?
  - Write and Read operations
  - Different applications have different I/O patterns.
- What's the impact of different cache replacement policies?
  - Also have the choice of using different policies at different network nodes.

- Approach: on AWS, run a number of Hadoop Spark apps, collect data traces, reply the traces in simulations to evaluate the effectiveness of in-network caching and the impact of different replacement policies.

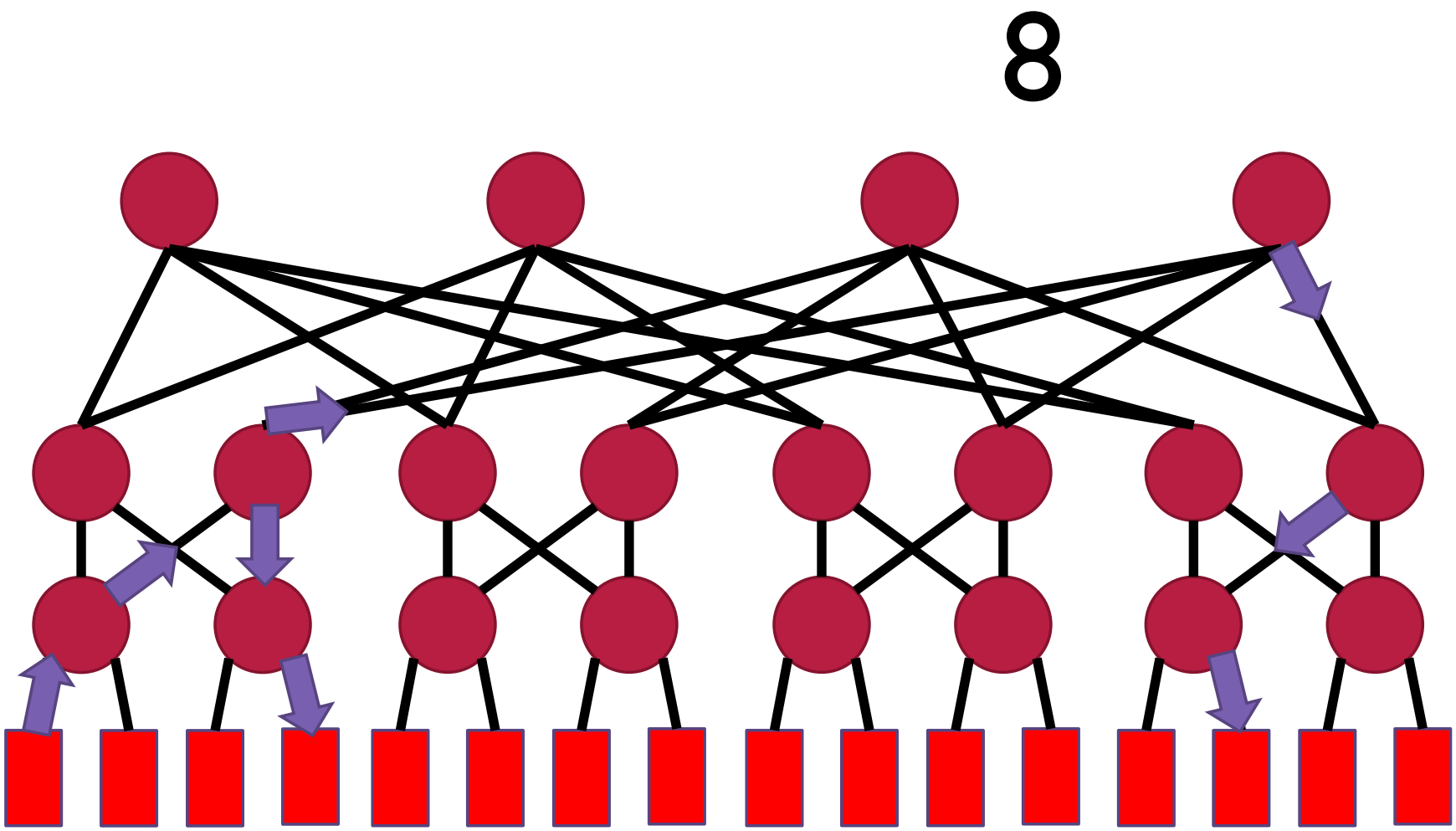
Write Operation

- HDFS writes data to multiple replica
  - Default 3, but configurable.
- Pipelining
  - Write to replica sequentially
- Can be converted to multicast in NDN
  - Notify replica about the write request, and replica retrieve data from the data source around the same time.

Traditional Pipelined Writes



Multicast



Write Traffic

Trace	Written Data (MB)	Network Transfer (MB)
aggregation	3519	7038
als	24	48
gbt	99	198
join	101	202
kmeans	6	12
linear	9	19
lr	19	38
rf	11	22
scan	19169	38338
sort	27784	55568
wordcount	5	11

Scenario	Pipelining (MB)	Multicast (MB)	Reduction
Third replica on same edge switch	1024	896	12.5%
Third replica on different edge switch	1280	1024	20.0%

Read Operation

- Cache read data in the network (in the form of Data packets)
- If multiple compute nodes request same data, later requests may hit a cache.
  - Reduce delay
  - Reduce overall network traffic
  - Reduce load on storing at DataNodes

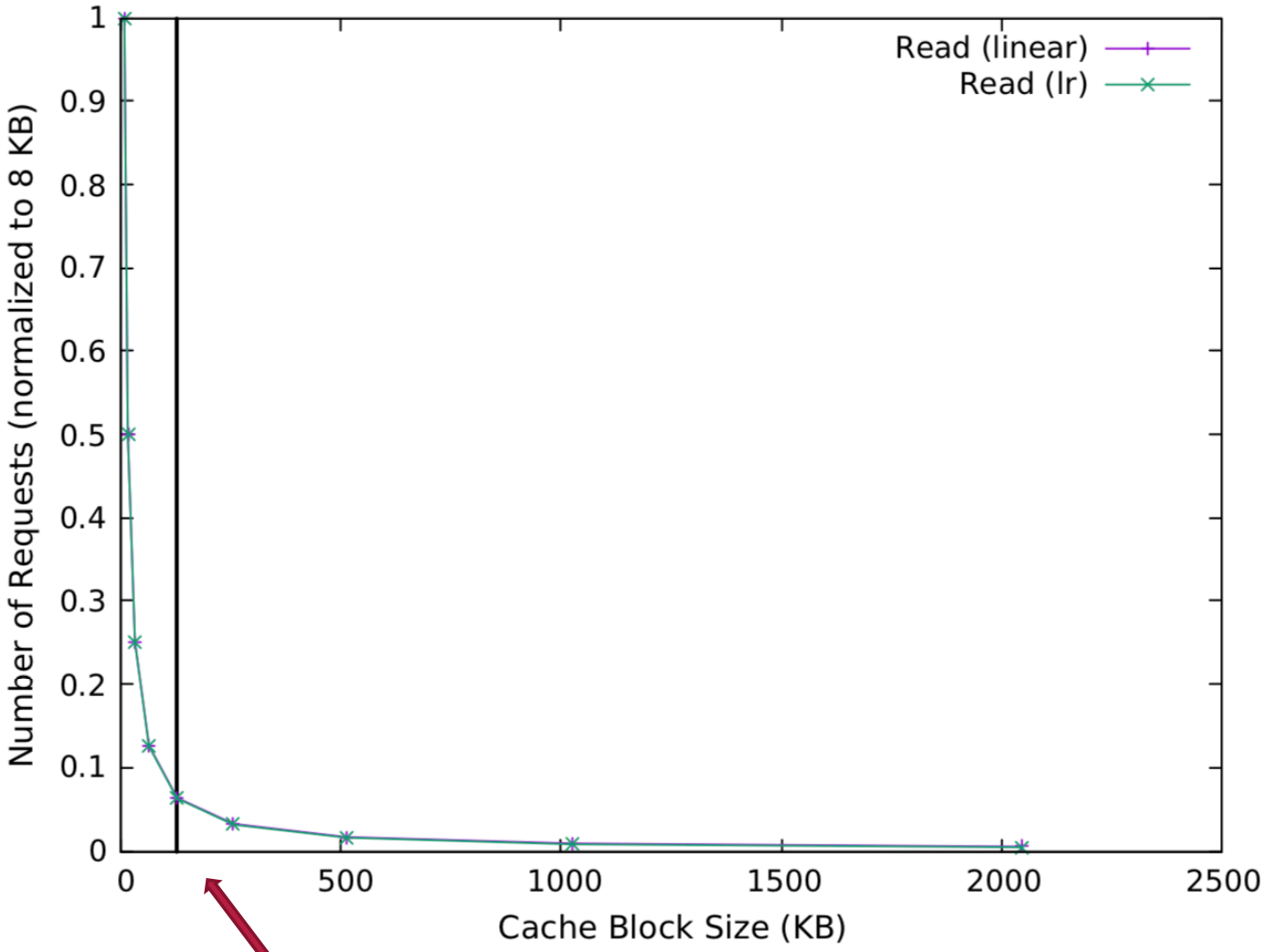
Read Traffic

Trace	Unique (MB)	Total (MB)
aggregation	221	230
als	1147	1156
gbt	33	43
join	4	13
kmeans	395	573
linear	20952	111195
lr	8893	19966
rf	1084	1094
scan	~0	9
sort	740	749
wordcount	847	1499

Caching Granularity

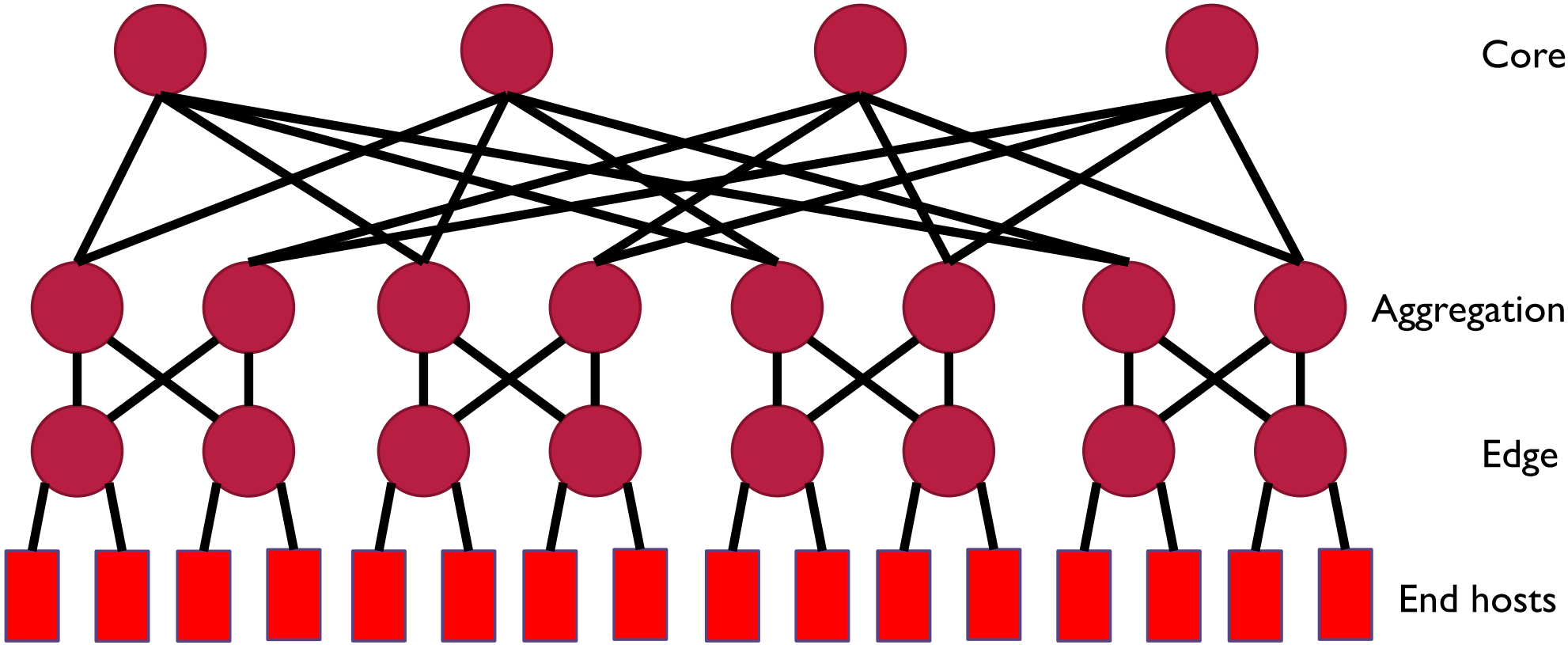
- What should be the size of the "cache block"?
  - NDN packets are the unit of caching. Need to segment and sign the data beforehand.
- Larger Data packets
  - Lower PIT, FIB, and Content Store overhead
  - But, coarser caching granularity and less efficient.
- Smaller Data packets
  - Higher PIT, FIB, and Content Store overhead
  - Finer caching granularity
- Need to balances data processing overhead vs. caching granularity

Block Size



Use 128KB packet size in simulations.

Network Topology is Fat Tree



Can use different cache replacement policies at different layers of switches.

Methodology

- Traces from Intel HiBench benchmark suite run on Apache Spark
- 128 compute/DataNodes, one coordinator/NameNode
- Replayed traces on simulated 128 end-host fat tree network
  - NDN-like caches located on every switch in network
- Evaluated effects of using different replacement policies
  - With same policy on all switches and with different policies at each layer
- Performance metric: total network traffic over all links
  - Count traffic once for every link it traverses
- Conducted 10 trials of each scenario with different host positions, then average.

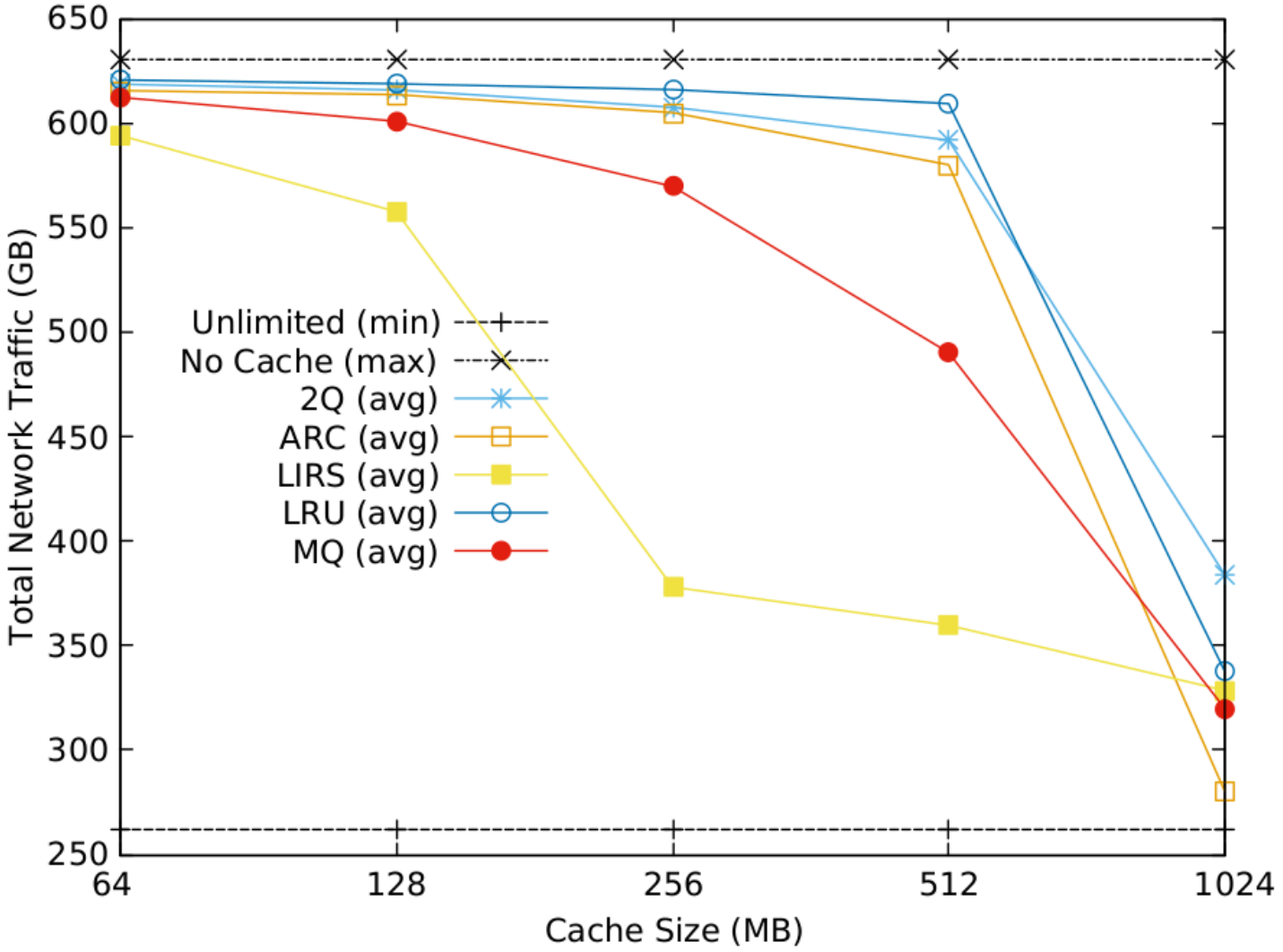
Replacement Policies

- Least Recently Used (LRU)
  - One of the simplest policies – discards blocks based upon last use time
- Two Queue (2Q)
  - Queue for hot blocks, queue for cold blocks, queue for recently evicted blocks
- Adaptive Replacement Policy (ARC)
  - Like 2Q, but uses dynamically-sized queues instead of fixed-sized queues
- Multi-Queue (MQ)
  - Separates blocks into multiple queues based upon access frequency
- Low Interreference Recency Set (LIRS)
  - Discards blocks based upon re-use distance (how soon they are used again)

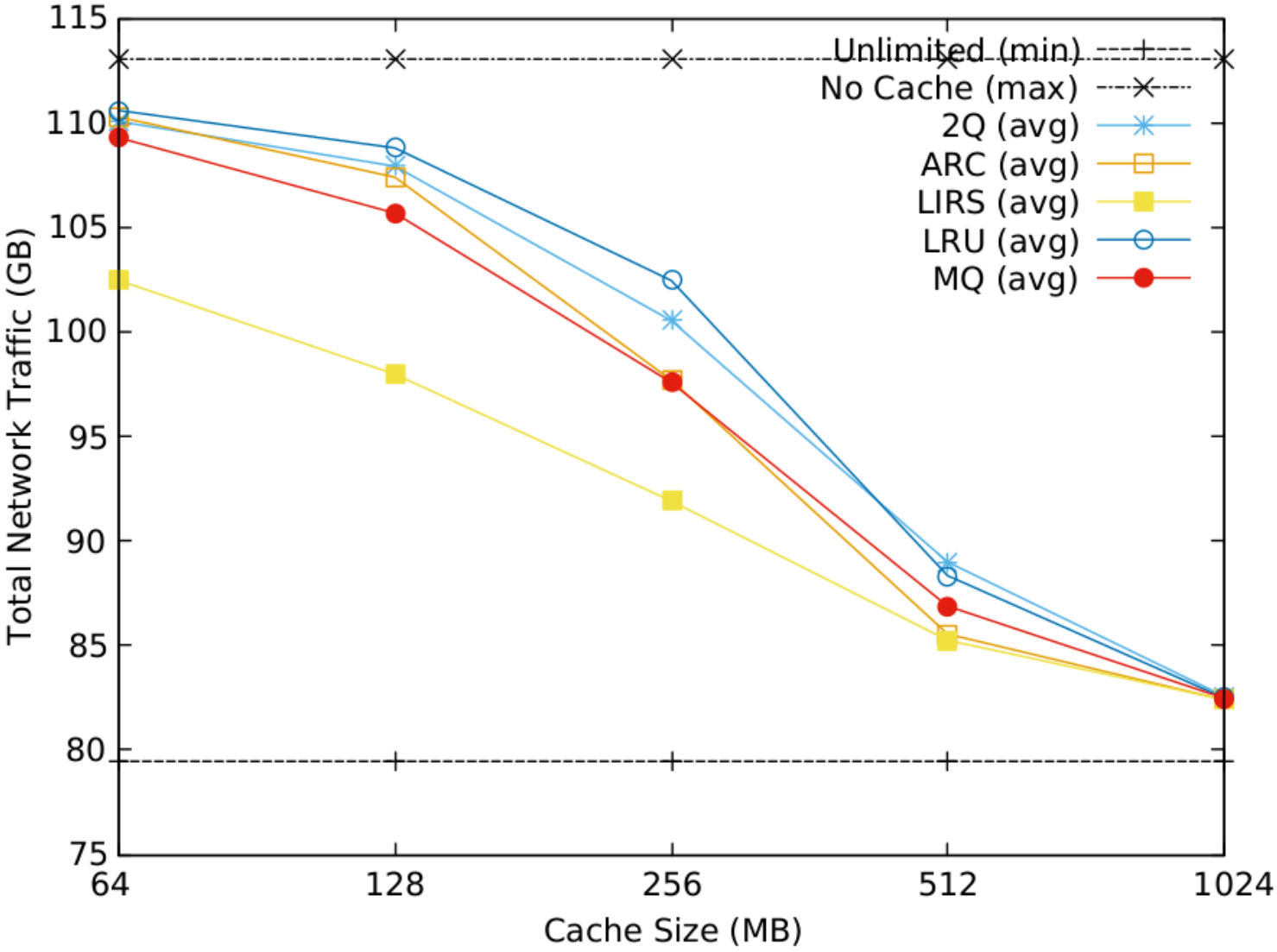
Benchmark Applications

- Replacement policies are largely irrelevant for multicast writes, so we focus on reads
- Two machine learning applications showed significant caching benefits for reads:
  - Linear regression (linear)
  - Logistic regression (lr)
- These applications have large amounts of intermediate data shared between partitions running on different DataNodes
  - Therefore, large number of reads

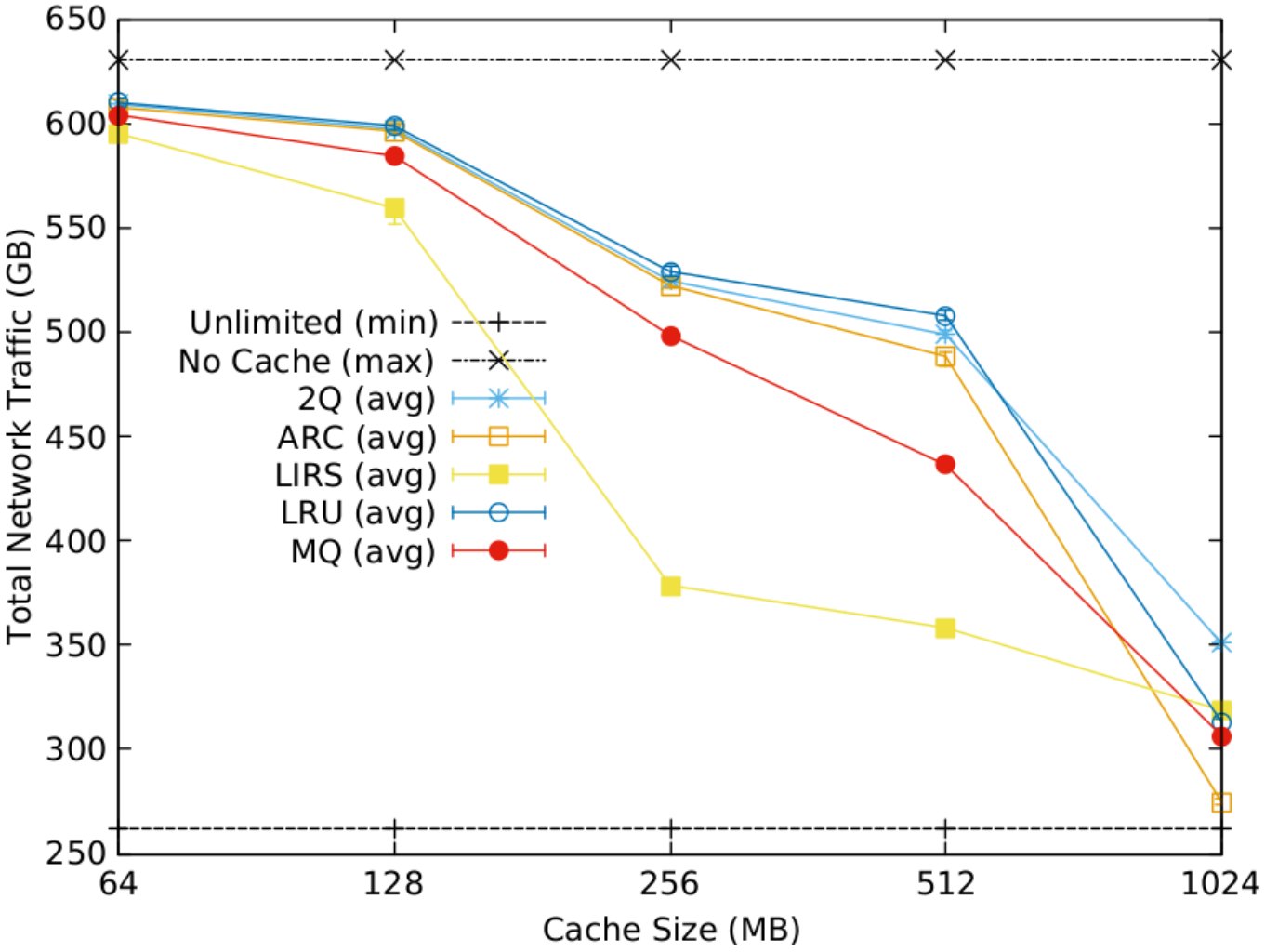
Same Replacement Policy Everywhere (Linear)



Same Replacement Policy Everywhere (LR)

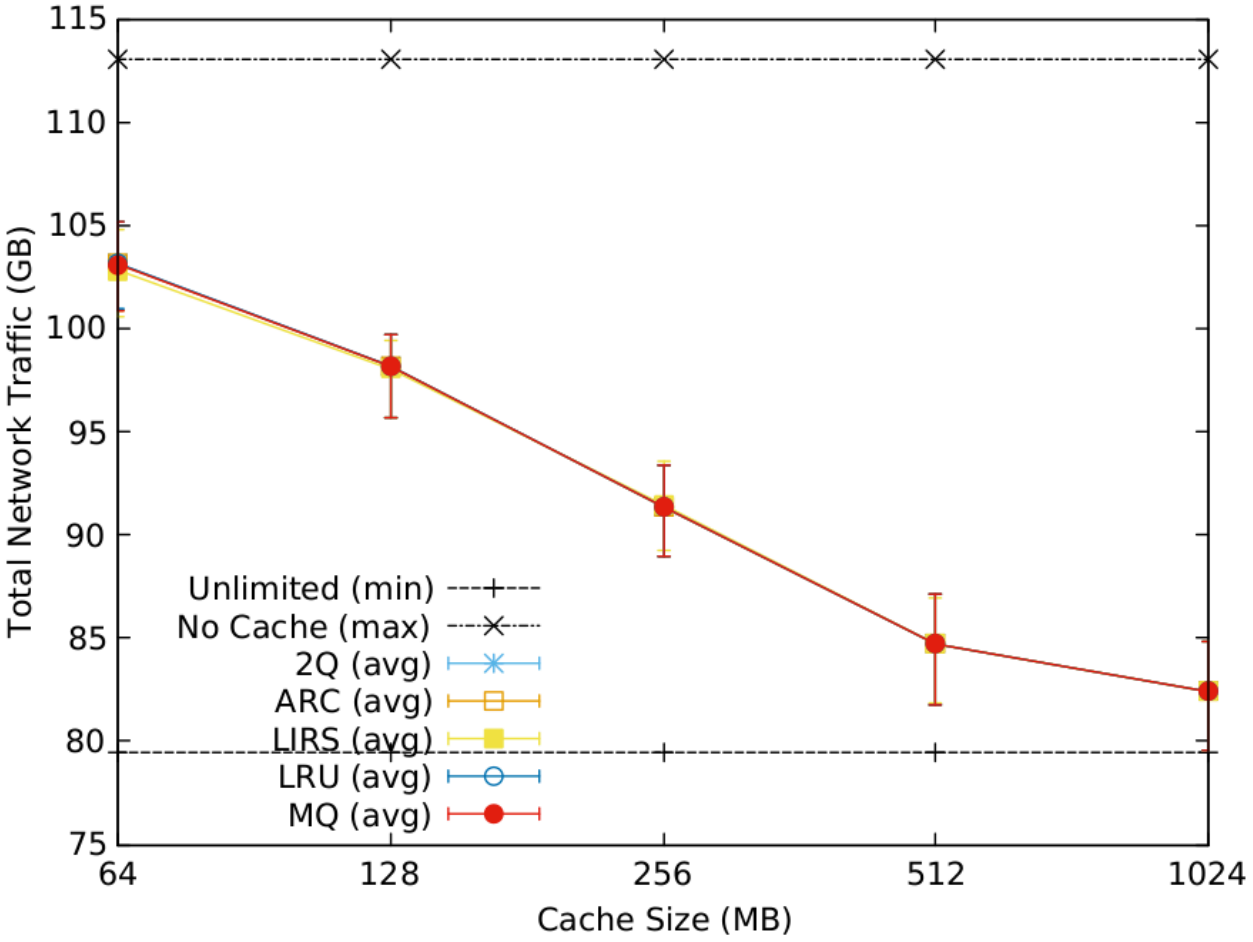


Layered Replacement Policies



Linear, (edge=2Q, aggr=LIRS)

Layered Replacement Policies



LR, edge=LIRS, aggr=MQ

Conclusions

- Multicast-like mechanism can reduce HDFS write traffic
- Applications that demonstrated greatest benefit from caching of read traffic were both learning applications that need to share lots of data among compute nodes.
- Overall, LIRS provided the best performance for these applications in our evaluations on fat trees
  - Generally, LIRS works better with smaller, and ARC slightly better with larger cache sizes
  - When both combined, even better performance for the largest application