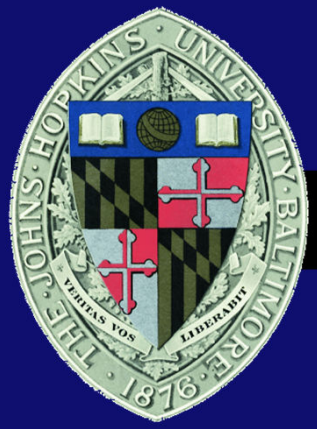
A Parallel Page Cache: IOPS and Caching for Multicore Systems

Da Zheng

Randal Burns

Alex Szalay

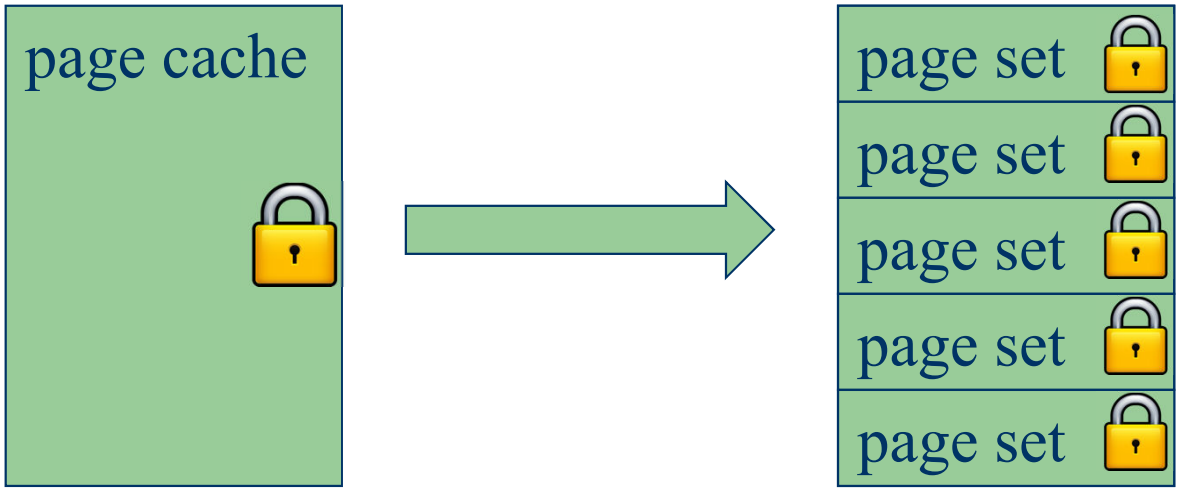
Johns Hopkins University



HotStorage'12, *13 June 2012*

Parallel page cache

- Scalable for 1M IOPS workload
- Partition the global cache into many small page sets
  - Eliminate lock contentions
- For SMP and NUMA
- Designed for cloud data service



Outline

- Motivation
- Linux page cache
- Goal
- Design
  - Set-Associative cache
  - NUMA-SA cache
- Experiments
- Future work

Motivation

- Hardware trend: high IOPS + many cores
- Cloud I/O: randomness + lower cache hits
  - Key/value stores: index lookup
- OS page cache
  - Designed for magnetic disks (thousands of IOPS)
  - Designed for high cache hits

The current OS page cache doesn't work with cloud workload

Can we use direct I/O?

- The performance of SSD and memory

	Random IOPS	Latency	Granularity
ioDrive Octal	1,300,000	45,000ns	512B
OCZ Vertex 4	120,000	20,000ns	512B
DDR3-1333	7,300,000	15ns	128B

- Memory cache provides higher throughput and hides latency

Memory cache is necessary

Linux page cache

- Pages are managed in a global page pool.
- Linux use a radix tree as index for pages in a file.
- Searching the index: protected by Read-Copy-Update (RCU).
  - Good for reading cache
- Page replacement requires to grab spin locks.

Locks destroy the performance of Linux page cache

Our goal

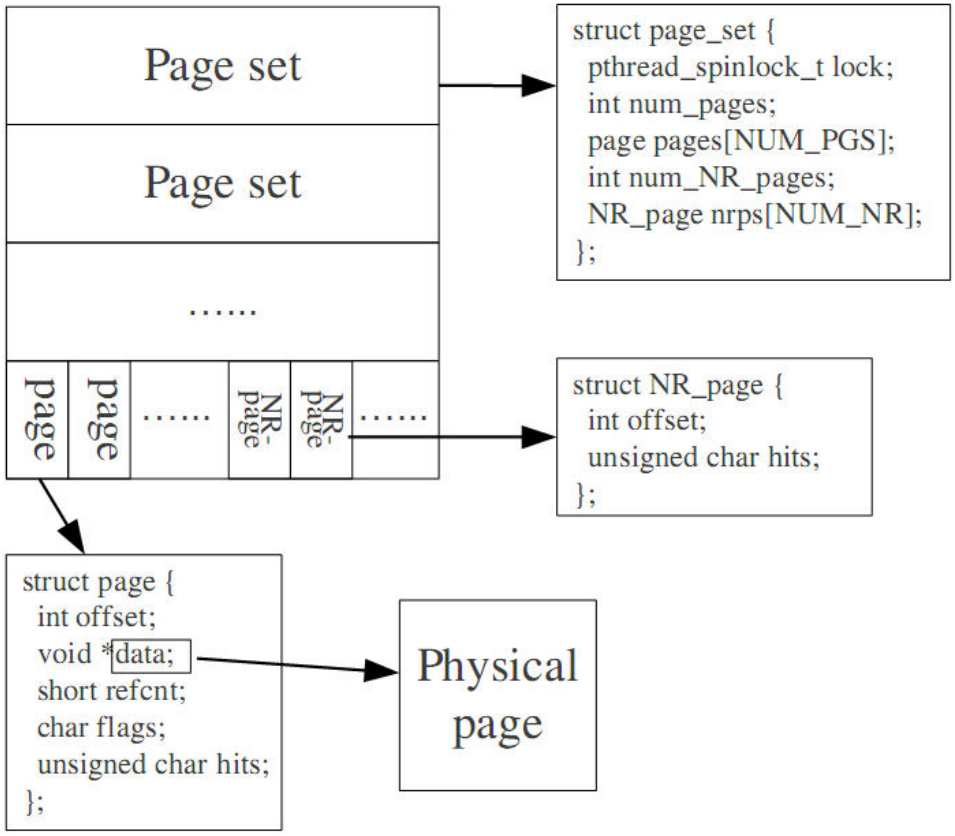
- A memory cache with extremely low overhead for massive parallel access
- Focus on cloud workload
  - Most accesses are reads
  - Power law distribution: few pages are accessed many times; most pages are accessed few times

Set-associative cache

- **Problem**: reduce lock contention
- **Solution**: replace a global lock with fine-grain locks

Set-associative cache

- Each set
  - Small number of pages: 8
  - Non-resident pages
  - Lock: protects the page metadata in the set
- All metadata of a page set in 4 cache lines
  - Few cache misses for searching and updating



Set-associative cache

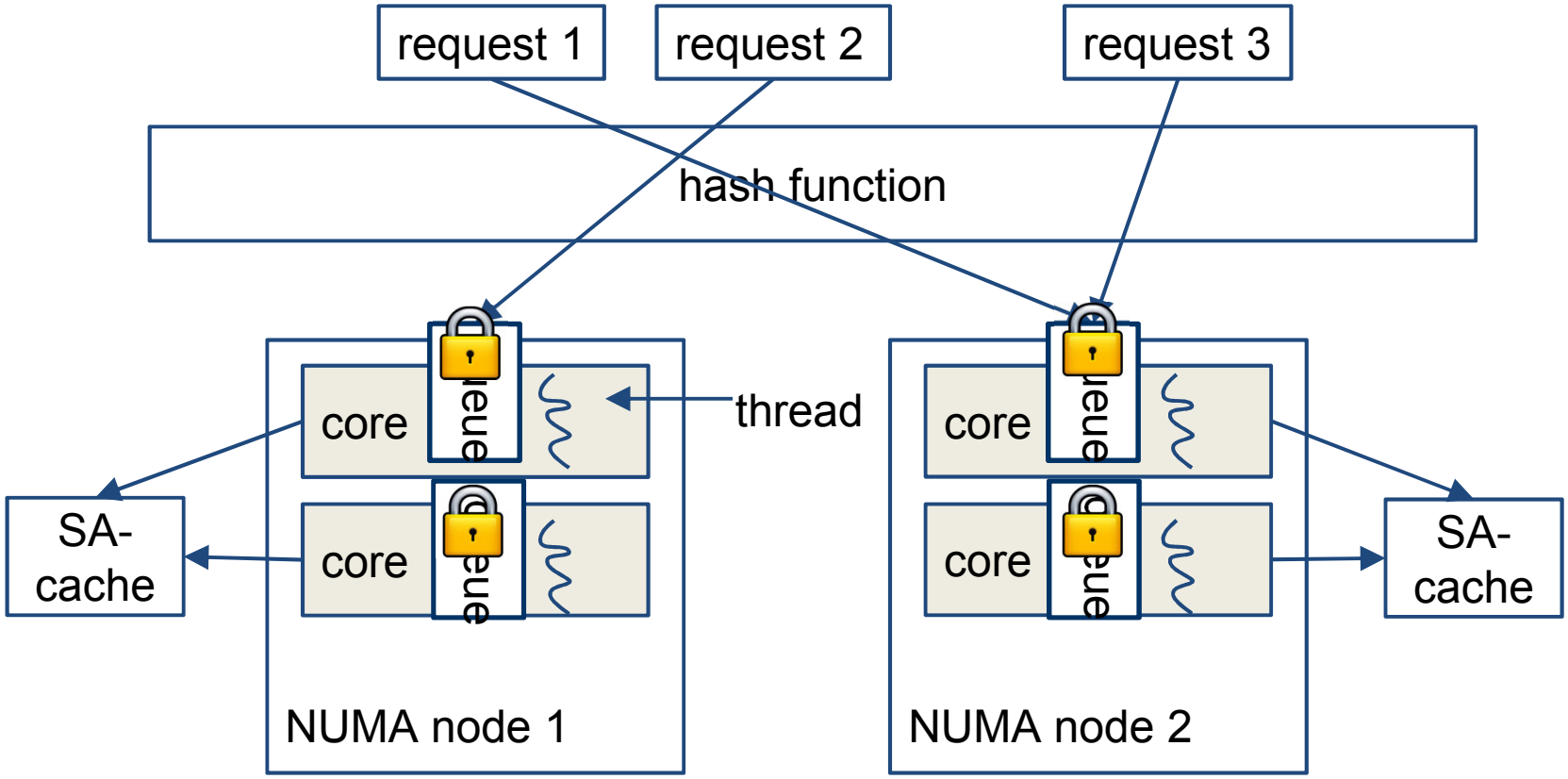
- Page eviction policy works inside a set
  - Pages: LFU
  - Non-resident pages: LRU
- Designed for SMP

NUMA-SA cache

- **Problem**: remote memory access has long latency
  - Avoid remote memory access
- **Solution**: partition the cache by NUMA nodes
  - All cores in a NUMA node share a cache partition
  - A NUMA node is treated as a node in the distributed system

NUMA-SA cache

- Requests are hashed and redistributed
- A work thread is attached to a core to serve requests
- Inter-core communication via message passing



NUMA-SA cache

- Requests are bundled for efficiency
  - Tradeoff: throughput vs. latency
- Asynchronous IO-like programming interface
  - `ssize_t access(io_request *requests, int num)`
  - `void wait_replies(int max_replies, reply_callback_t func)`

Experiments

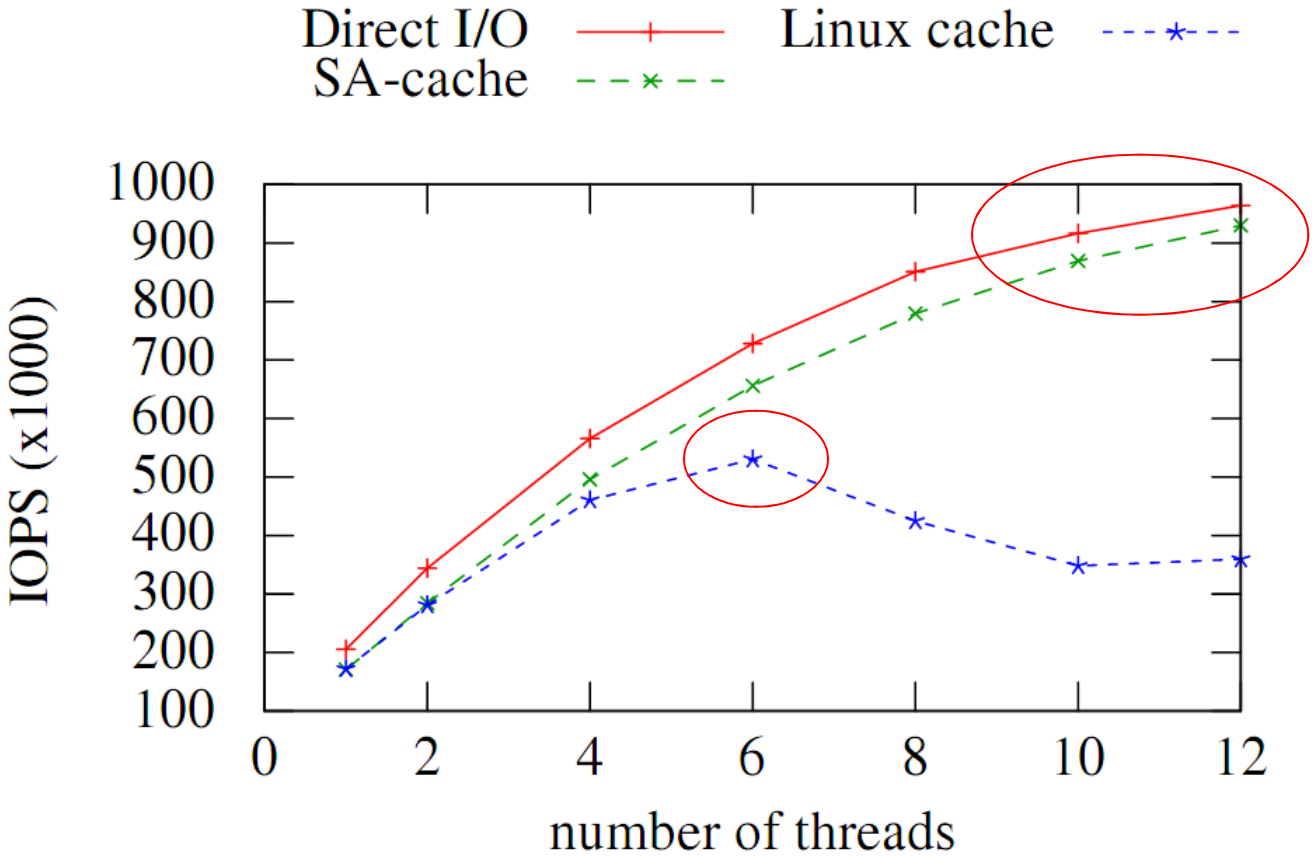
- Experiment 1: scalability of our cache with a high page turnover rate
  - Under random workload without cache hits
- Experiment 2: cache hit rate of our cache
  - Under zipfian workload
- Experiment 3: overall run-time performance of our cache
  - Under zipfian workload

Experiment setup

- Random read-only workload: YCSB
- On Ramdisk
- 48-core NUMA machine

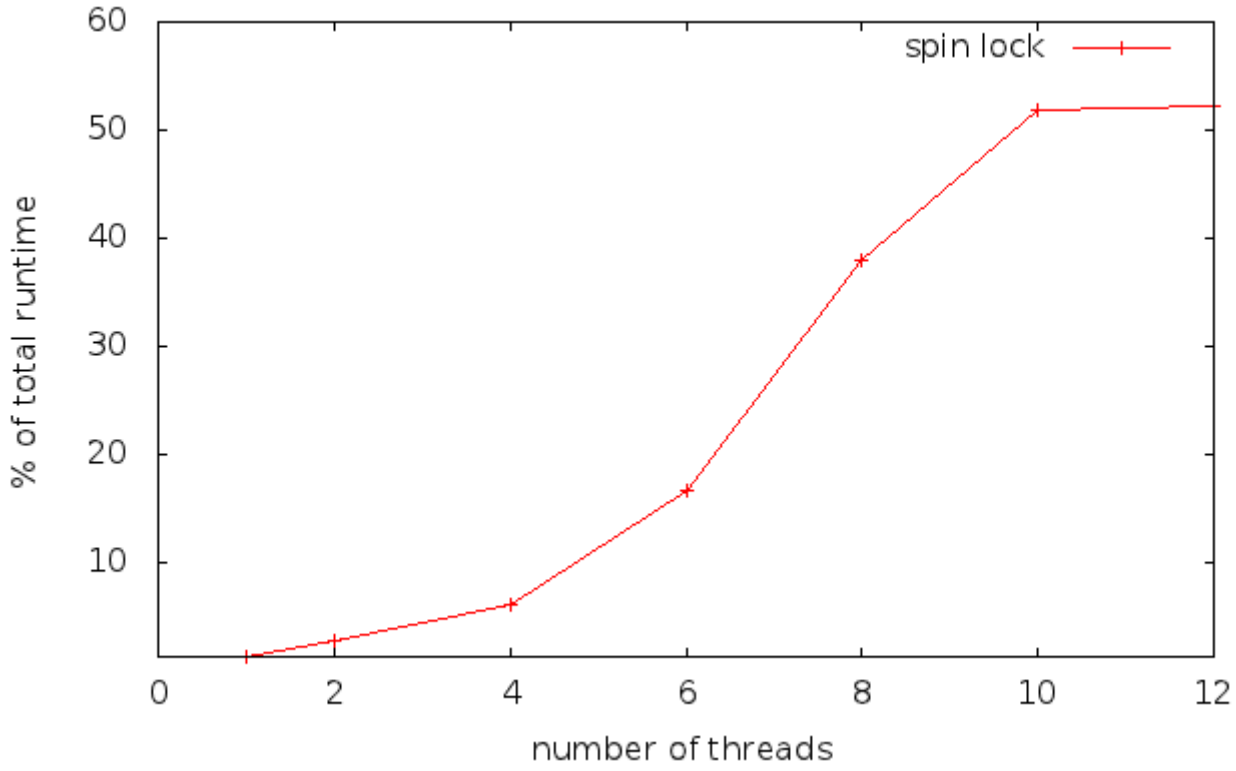
SA-cache vs. direct IO

- SA-cache scale as well as direct IO in SMP



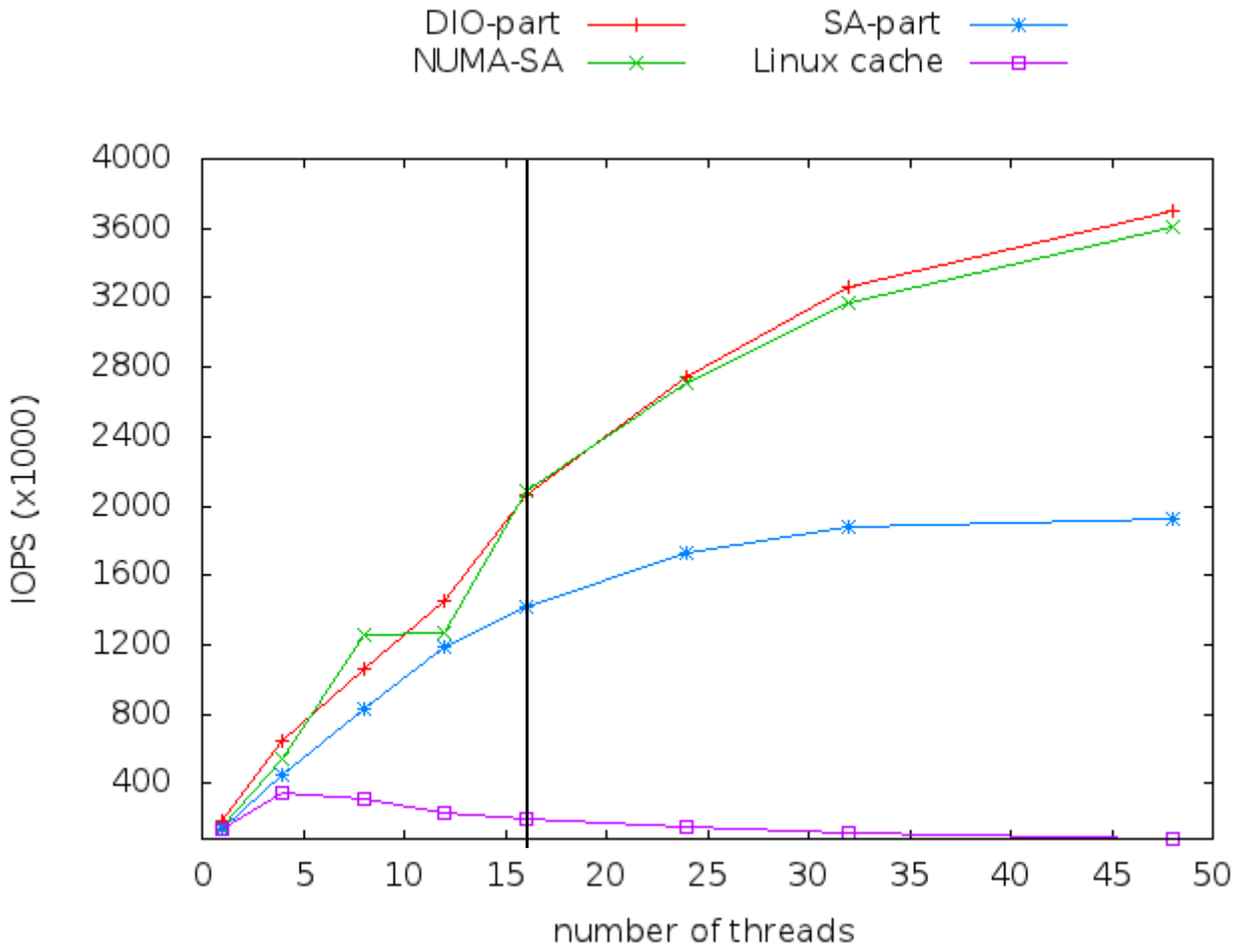
Why does Linux cache scale poorly?

- Lock overhead leads to poor performance in the Linux page cache



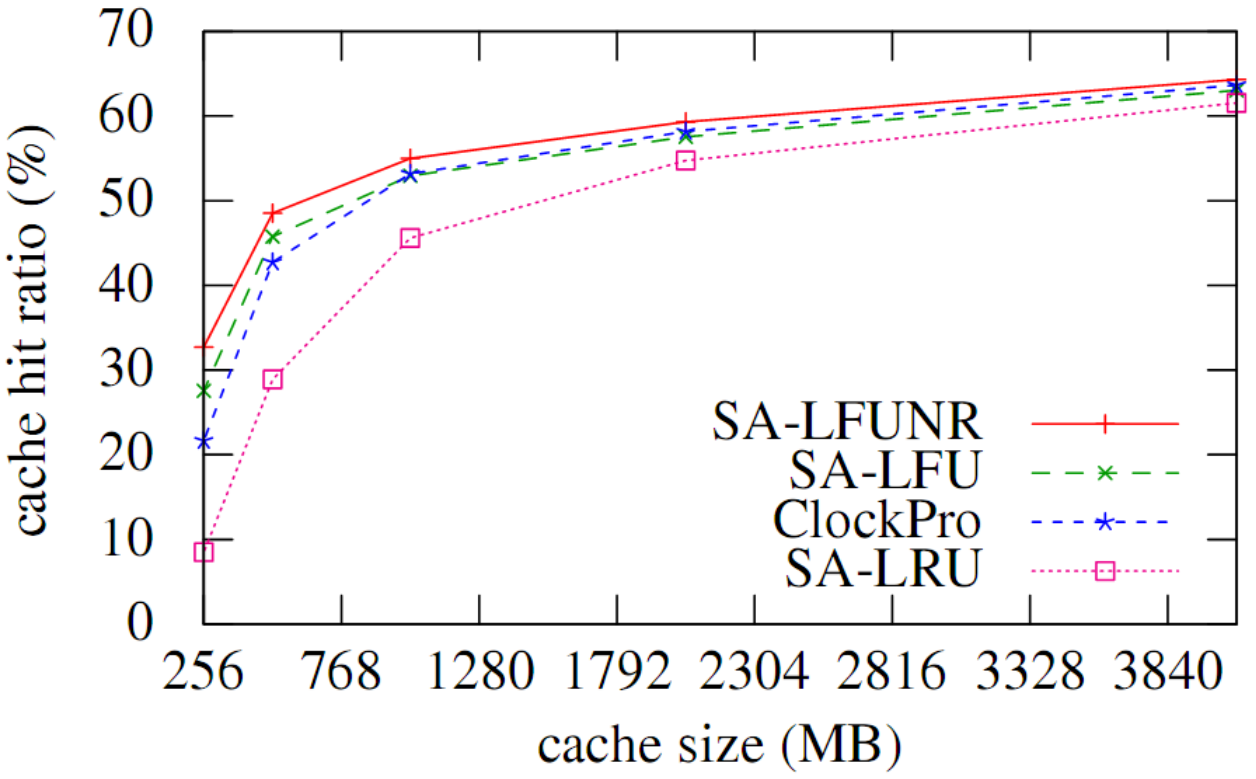
NUMA-SA cache vs. direct IO

- NUMA-SA cache scales well

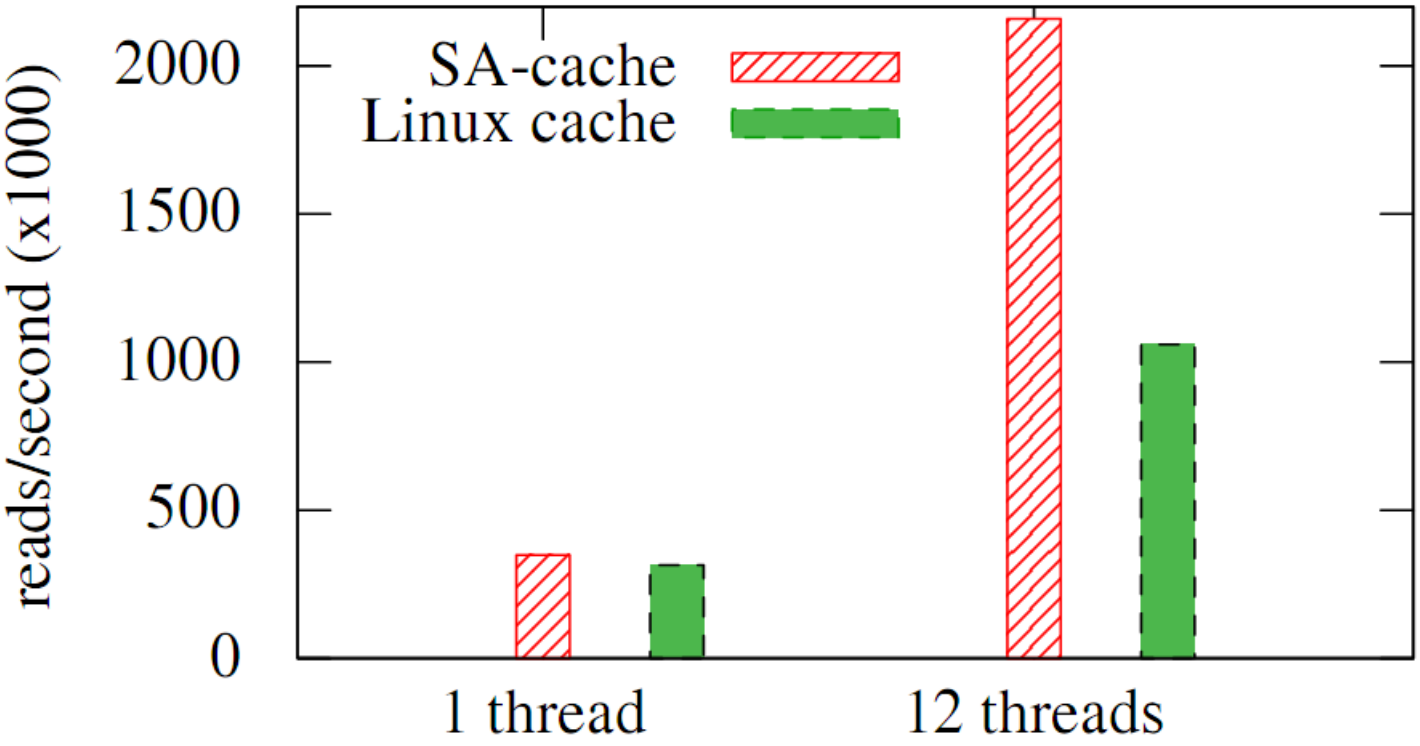


Cache hits – zipfian workload

- Frenquency works better than recency
- SA-LFU(NR) realizes a cache hit rate similar to ClockPro.



Runtime –zipfian workload



Future work

- Performance evaluation on SSD array
  - Performance is currently measured on ramdisk
  - Overhead in the block layer and below may impact our design.
- In-kernel implementation
  - Our cache is currently implemented in the user space
- Dynamic cache sizing
  - Our cache size is static.

Email: dzheng5@jhu.edu