Bullet Cache

Balancing speed and usability
in a cache server

Ivan Voras <ivoras@freebsd.org>

What is it?

- People know what **memcached** is... mostly
- Example use case:
  - So you have a web page which is just dynamic enough so you can't cache it completely as an HTML dump
  - You have a SQL query on this page which is 99.99% always the same (same query, same answer)
  - ...so you cache it

Why a cache server?

- Sharing between processes
  - ... on different servers
- In environments which do not implement application persistency
  - CGI, FastCGI
  - PHP
- Or you're simply lazy and want something which works...

A little bit of history...

- This started as my “pet project”...
  - It's so ancient, when I first started working on it, Memcached was still single-threaded
  - It's gone through at least one rewrite and a whole change of concept
- I made it because of the frustration I felt while working with Memcached
  - Key-value databases are so very basic
  - “I could do better than that” :)

Now...

- Used in production in my university's project
- Probably the fastest memory cache engine available (in specific circumstances)
- Available in FreeBSD ports (databases/mdcached)
- Has a 20-page User Manual :)

What's wrong with memcached?

- Nothing much – it's solid work
- The classic problem:
  cache expiry / invalidation
  - memcached accepts a list of records to expire (inefficient, need to maintain this list)
- It's fast – but is it fast enough?
  - Does it really make use of multiple CPUs as efficiently as possible?

Introducing the Bullet Cache

1. **Offers a smarter data structure to the user side than a simple key-value pair**
2. **Implements "interesting" internal data structures**
3. **Some interesting bells & whistles**

User-visible structure

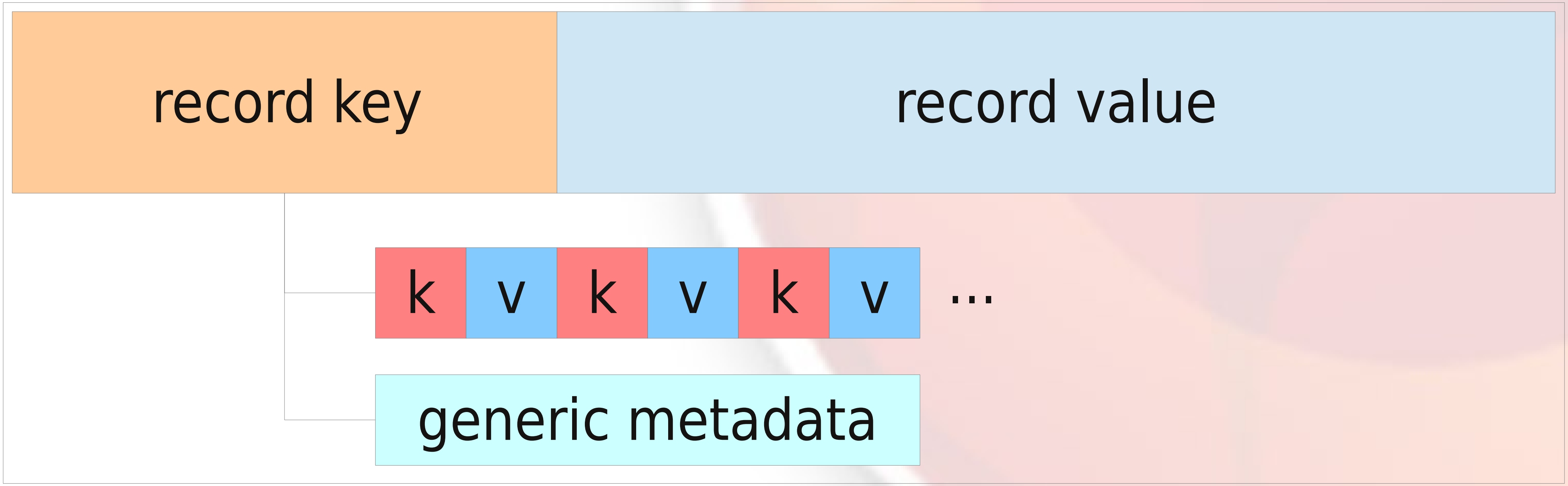
- Traditional (memcached) style:
  - Key-value pairs
  - Relatively short keys (255 bytes)
  - ASCII-only keys (?)
  - ~~(ASCII-only protocol)~~
  - Multi-record operations only with a list of records
  - Simple atomic operations (relatively inefficient - atoi())

Introducing record tags

- They are metadata
- Constraints:
  - Must be fast (they are NOT db indexes)
  - Must allow certain types of bulk operations
- The implementation:
  - Both key and value are signed integers
  - No limit on the number of tags per record
  - Bulk queries: (tkey X) && (tval1, [tval2...])

Record tags

- *I heard you like key-value records so I've put key-value records into your key-value records...*



Metadata queries (1)

- Use case example: a web application has a page “/contacts” which contains data from several SQL queries as well as other sources (LDAP)
  - Tag all cached records with `(tkey,tval) = (42, hash(“/contacts”))`
  - When constructing page, issue query: `get_by_tag_values(42, hash(“/contacts”))`
  - When expiring all data, issue query: `del_by_tag_values(42, hash(“/contacts”))`

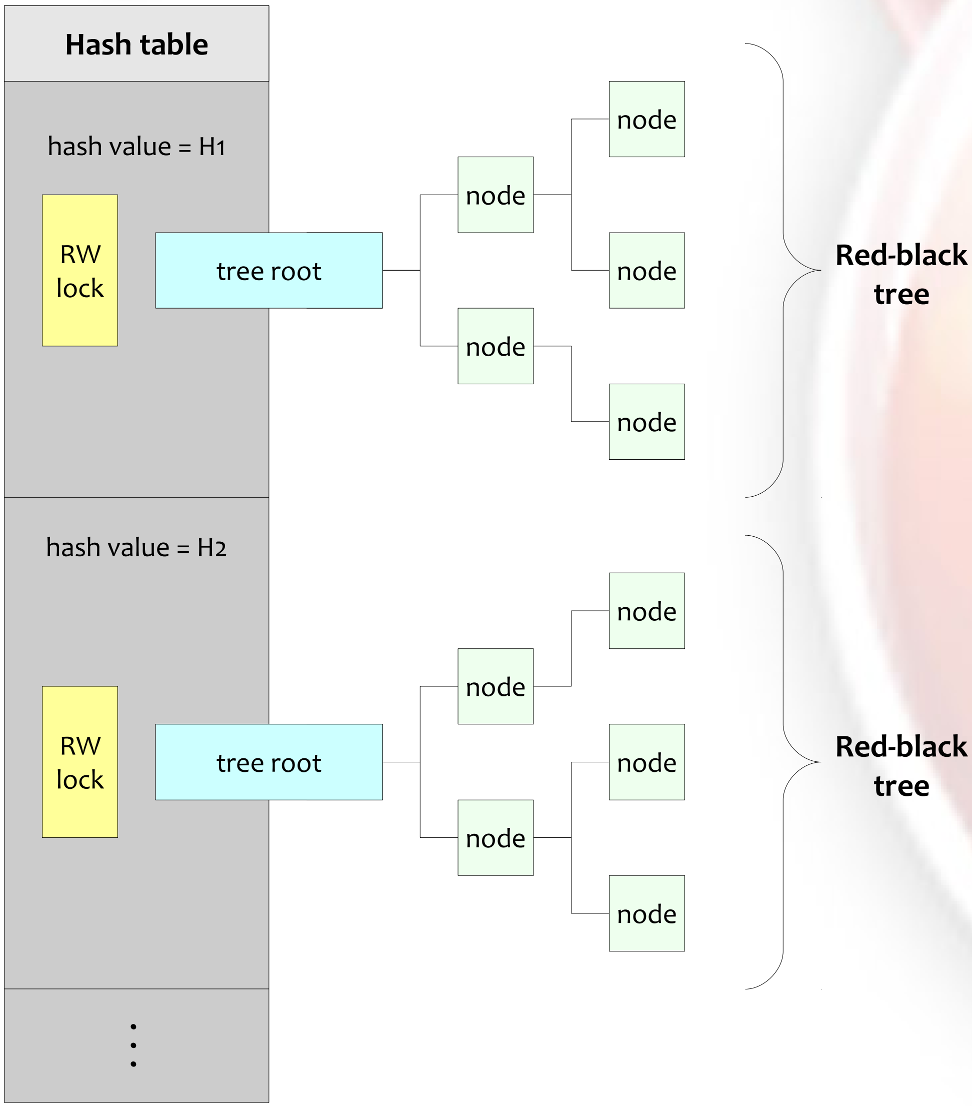
Metadata queries (2)

- Use case example: Application objects are stored (serialized, marshalled) into the cache, and there's a need to invalidate (expire) all objects of a certain type
  - Tag records with (tkey, tval) = (object_type, instance_id)
  - Expire with `del_by_tag_values(object_type, instance_id)`
  - Also possible: tagging object interdependance

Under the hood

- It's "interesting"...
- Started as a C project, now mostly converted to C++ for easier modularization
  - Still uses C-style structures and algorithms for the core parts – i.e. not std::containers
- Contains tests and benchmarks within the main code base
  - C and PHP client libraries

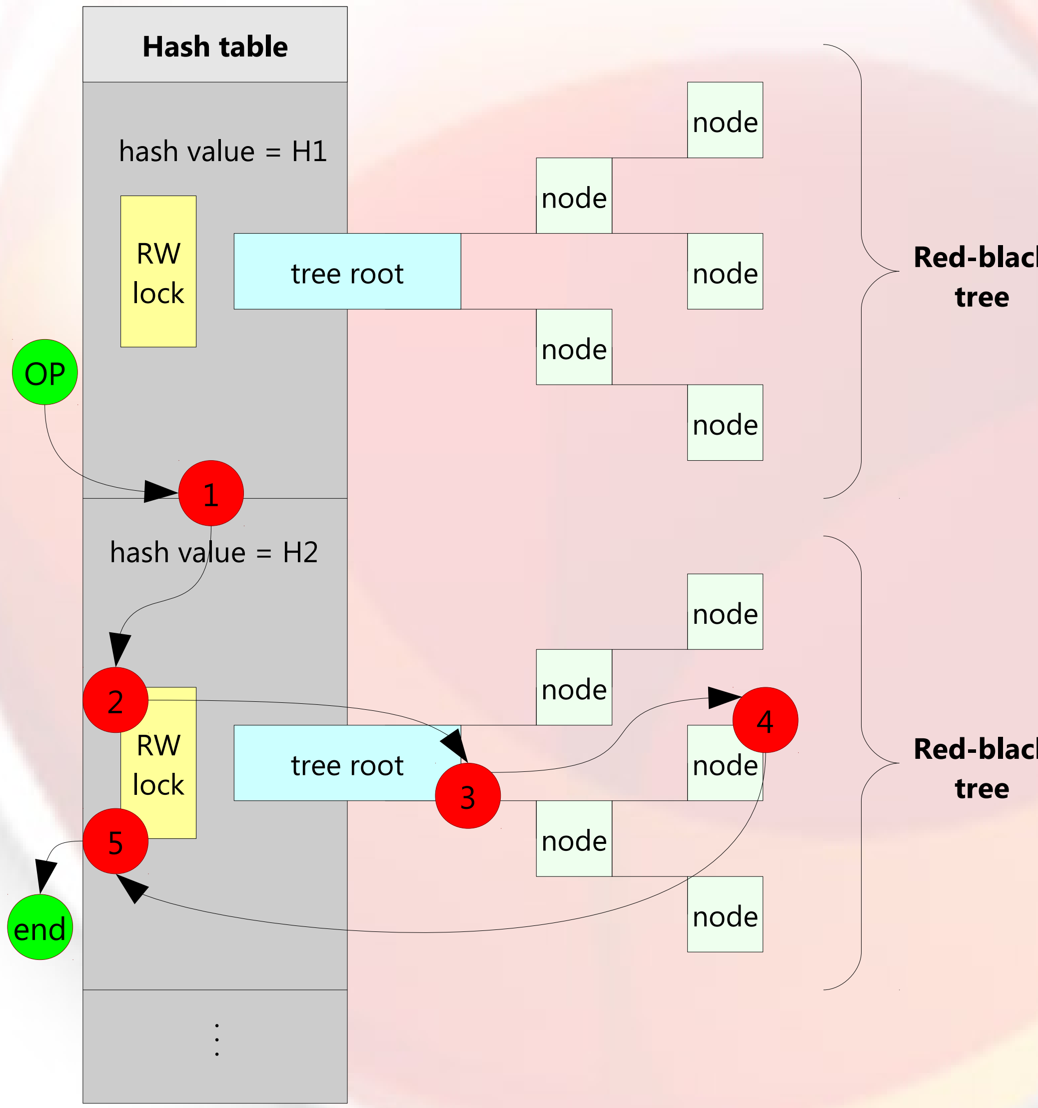
The main data structure



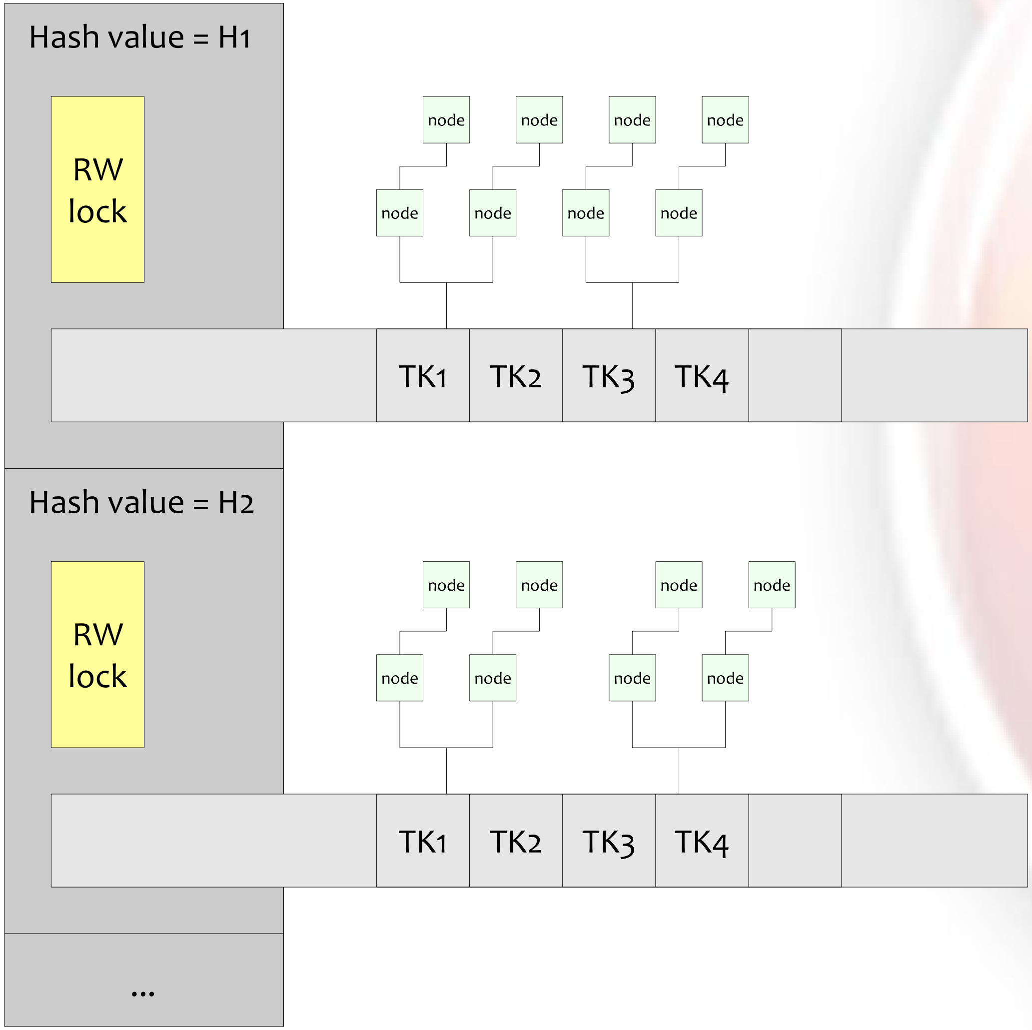
- A "forest of trees", anchored in hash table buckets
- Buckets are directly addressed by hashing record keys
- Buckets are protected by rwlocks

Basic operation

1. Find h = Hash(key)
2. Acquire lock #h
3. Find record in RB tree indexed by key
4. Perform operation
5. Release lock #h



Record tags follow a similar pattern

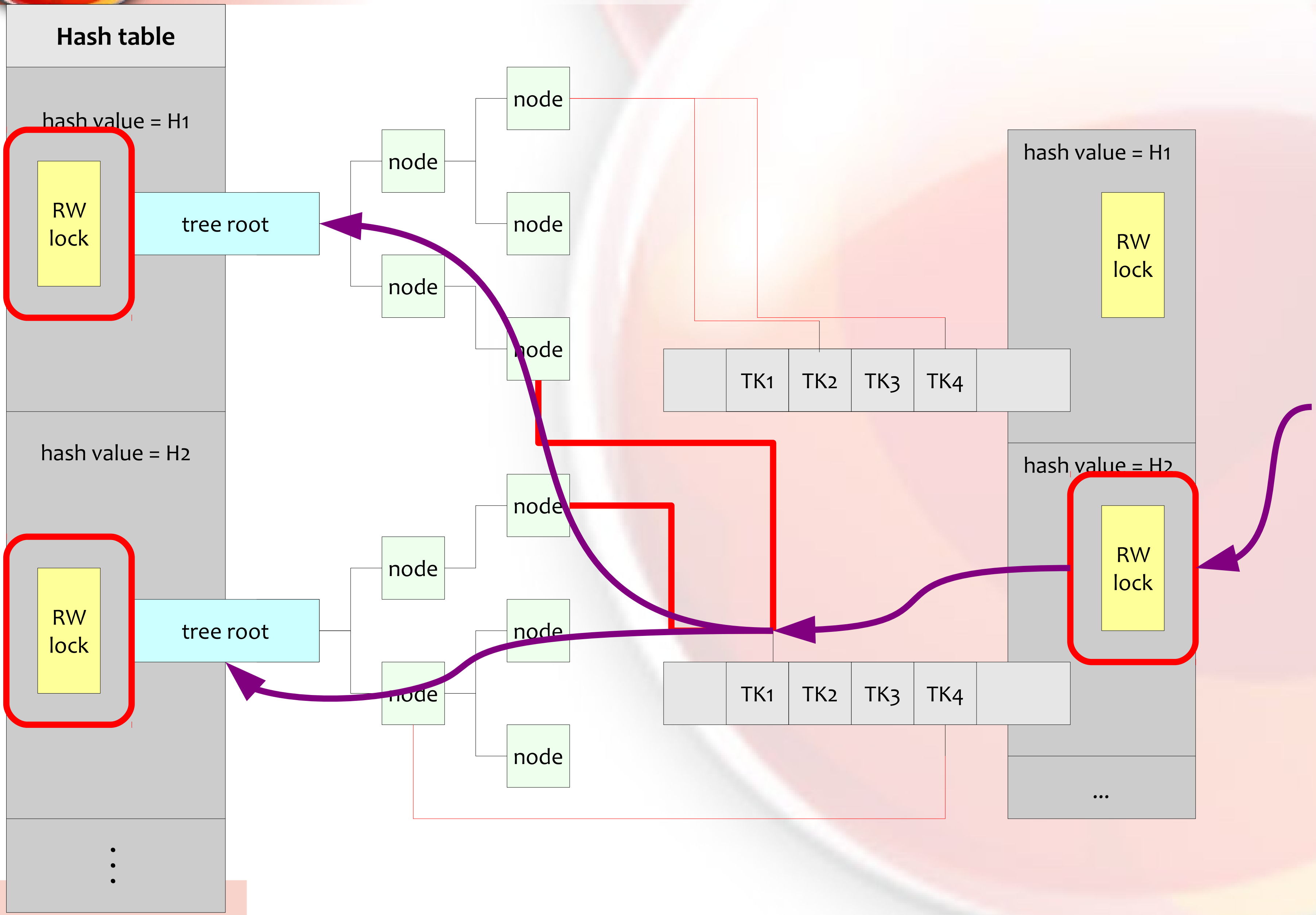


- The tags index the main structure and are maintained (almost) independently

Concurrency and locking

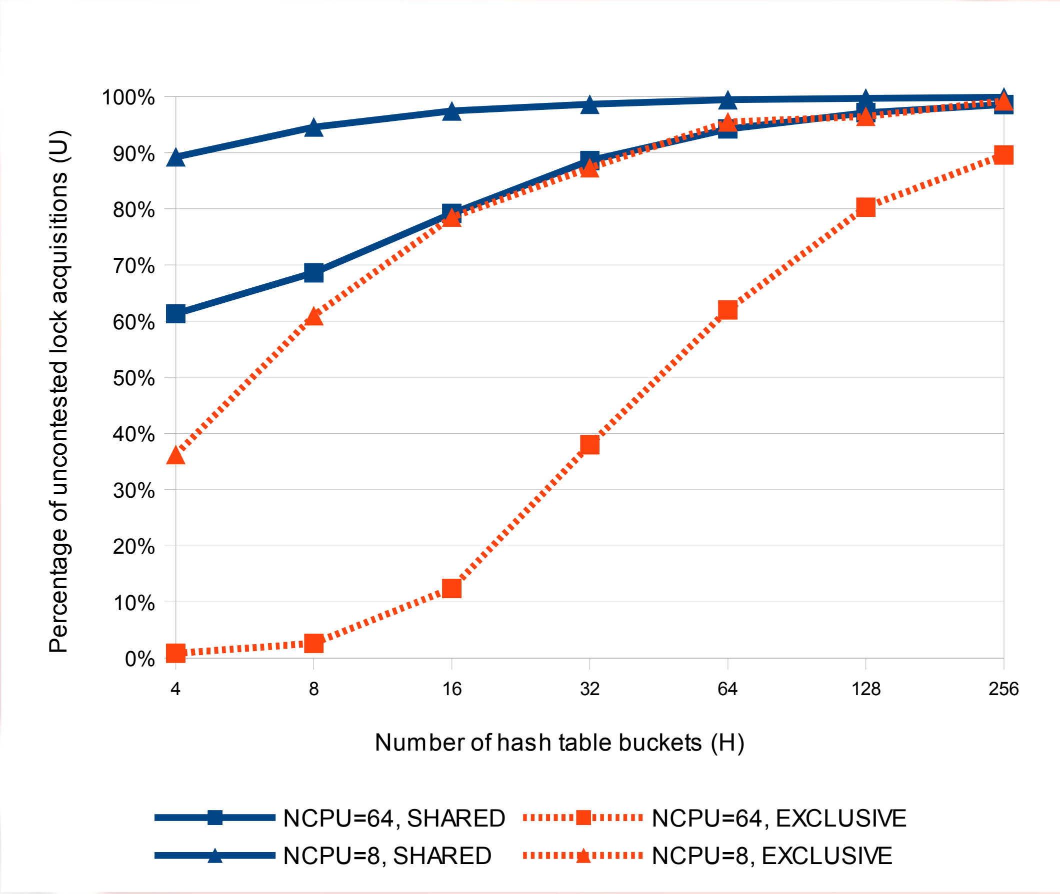
- Concurrency is great – the default configuration starts 256 record buckets and 64 tag buckets
- Locking is without ordering assumptions
  - *_trylock() for everything
  - rollback-and-retry
  - No deadlocks
    - Livelocks on the other hand need to be investigated

Two-way linking between records and tags



Concurrency

- Scenario 1:
  - A record is referenced → need to hold N tag bucket locks
- Scenario 2:
  - A tag is referenced → need to hold M record bucket locks



Percentage of uncontested lock acquisitions

Multithreading models

- Aka “which thread does what”
- Three basic tasks:
  - T1: Connection acceptance
  - T2: Network IO
  - T3: Payload work
- The big question: how to distribute these into threads?

Multithreading models

- SPED : Single process, event driven
- SEDA : Staged, event-driven architecture
- AMPED : Asymmetric, multi-process, event-driven
- SYMPED : Symmetric, multi-process, event driven

Model	New connection handler	Network IO handler	Payload work
SPED	1 thread	In connection thread	In connection thread
SEDA	1 thread	N1 threads	N2 threads
SEDA-S	1 thread	N threads	N threads
AMPED	1 thread	1 thread	N threads
SYMPED	1 thread	N threads	In network thread

All the models are event-driven

- The “dumb” model: thread-per-connection
- Not really efficient
    - (FreeBSD has experimented with KSE and M:N threading but that didn't work out)
- IO events: via kqueue(2)
- Inter-thread synchronization: queues signalled with CVs

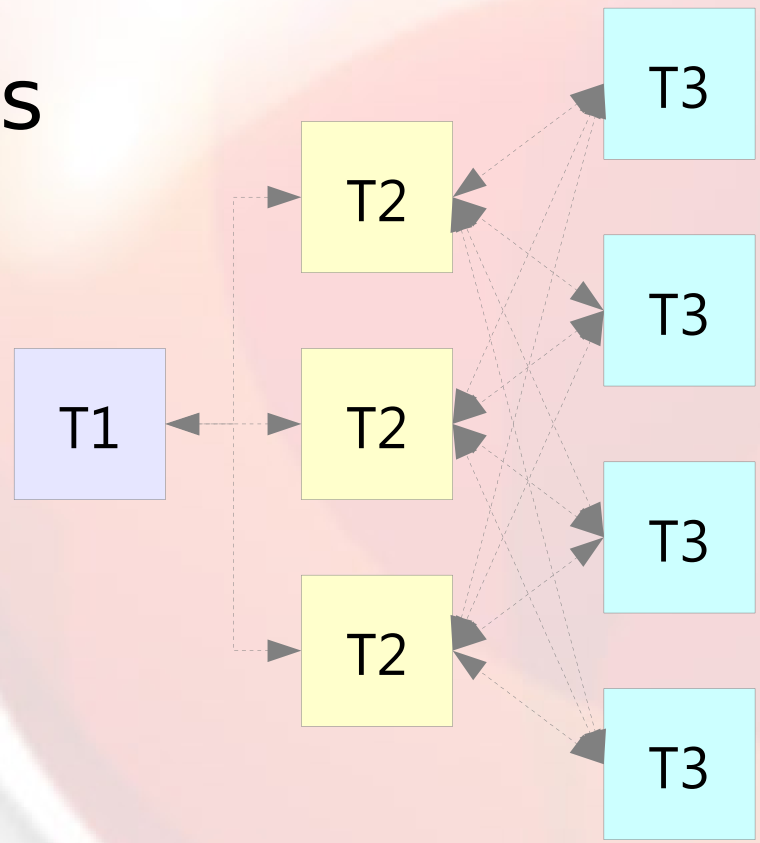
SPED

- Single-threaded, event-driven
- Very efficient on single-CPU systems
- Most efficient if the operation is very fast (compared to network IO and event handling)
- Used in efficient Unix network servers

```
Get a list of events from the OS
    Loop through the list
        Parse event
        Perform operation
        Return data
    Prepare for the new list
```

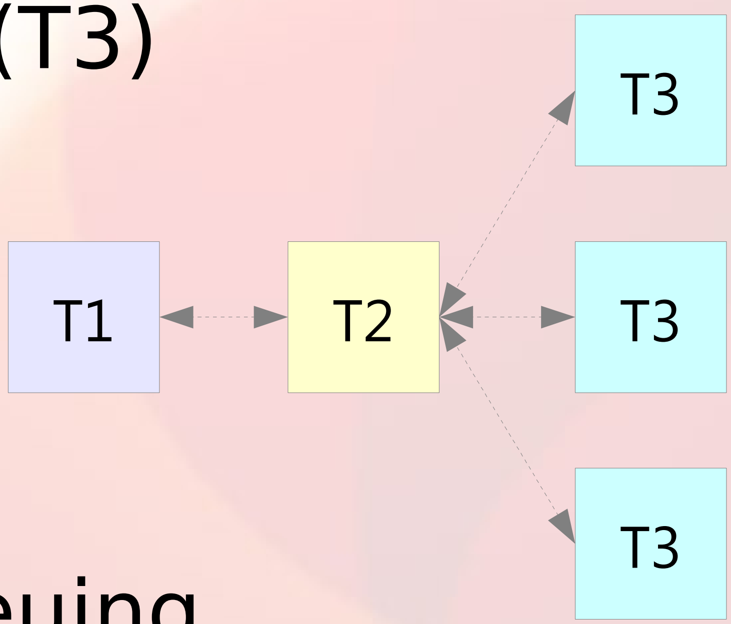
SEDA

- Staged, event-driven
- Different task threads instantiated in different numbers
- Generally, N1 != N2 != N3
- The most queueing
- The most separation of tasks – most CPUs used



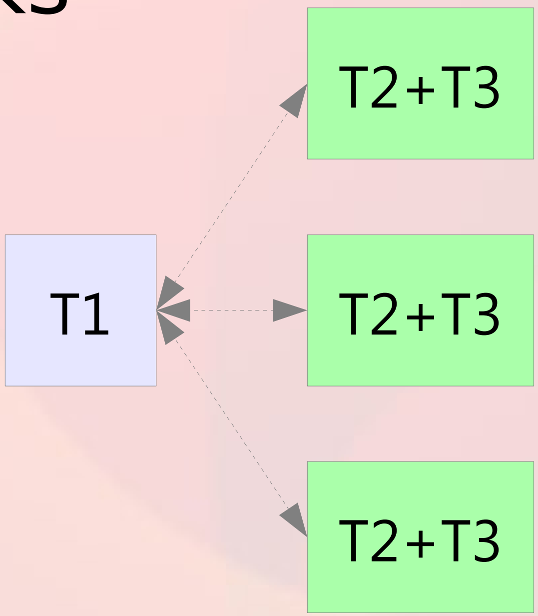
AMPED

- Asymmetric multi-process event-driven
- Asymmetric: N(T2) != N(T3)
- Assumes network IO processing is cheap compared to operation processing
- Moderate amount of queuing
- Can use arbitrary number of CPUs



SYMPED

- Symmetric multi-process event-driven
- Symmetric: grouping of tasks
- Assumes network IO and operation processing are similarly expensive but uniform
- Sequential processing inside threads
- Similar to multiple instances of SPED



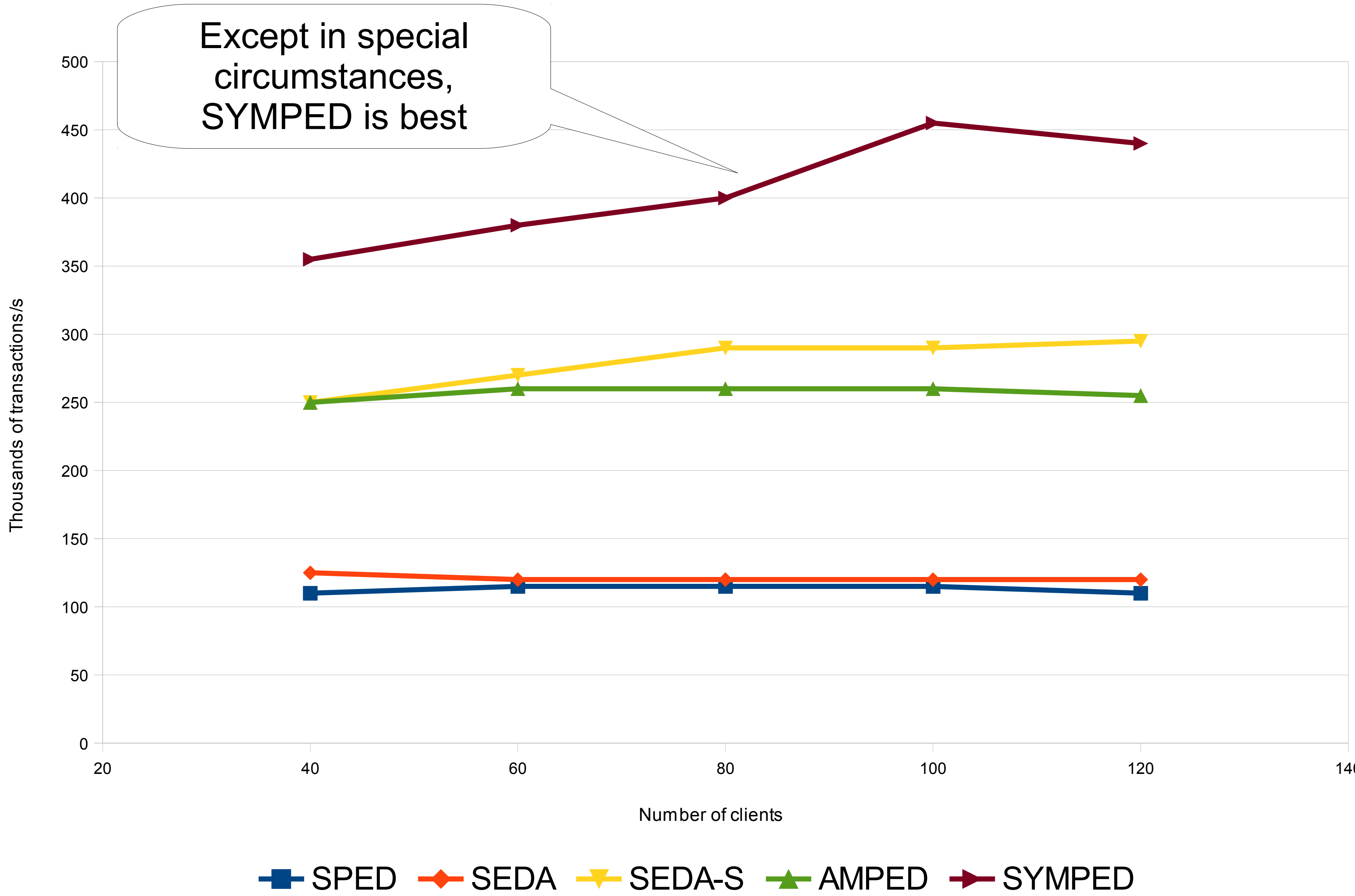
Multithreading models in Bullet Cache

- Command-line configuration:
  - n : number of network threads
  - t : number of payload threads
- n=0, t=0 : SPED
- n=1, t>0 : AMPED
- n>0, t=0 : SYMPED
- n>1, t>0 : SEDA
- n>1, t>1, n=t : SEDA-S (symmetrical)

How does that work?

- SEDA: the same network loop accepts connections and network IO
- Others: The network IO threads accept messages, then either:
  - process them in-thread or
  - queue them on worker thread queues
- Response messages are either sent in-thread from whichever thread generates them or finished with the IO event code

Performance of various models



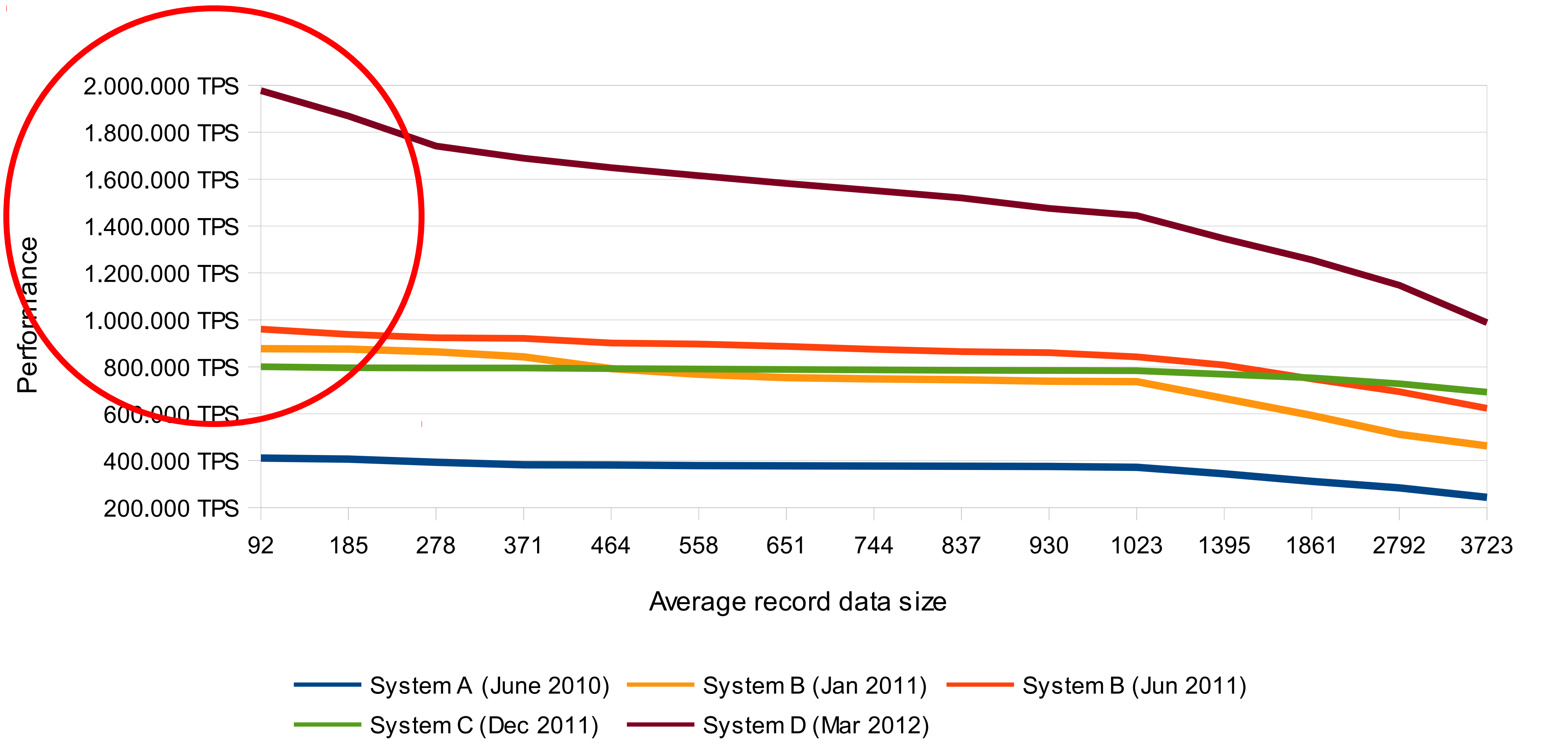
Why is SYMPED efficient?

- The same thread receives the message and processes it
- No queueing
  - No context switching
  - In the optimal case: no any kind of (b)locking delays
- Downsides:
  - Serializes network IO and processing within the thread (which is ok if per-CPU)

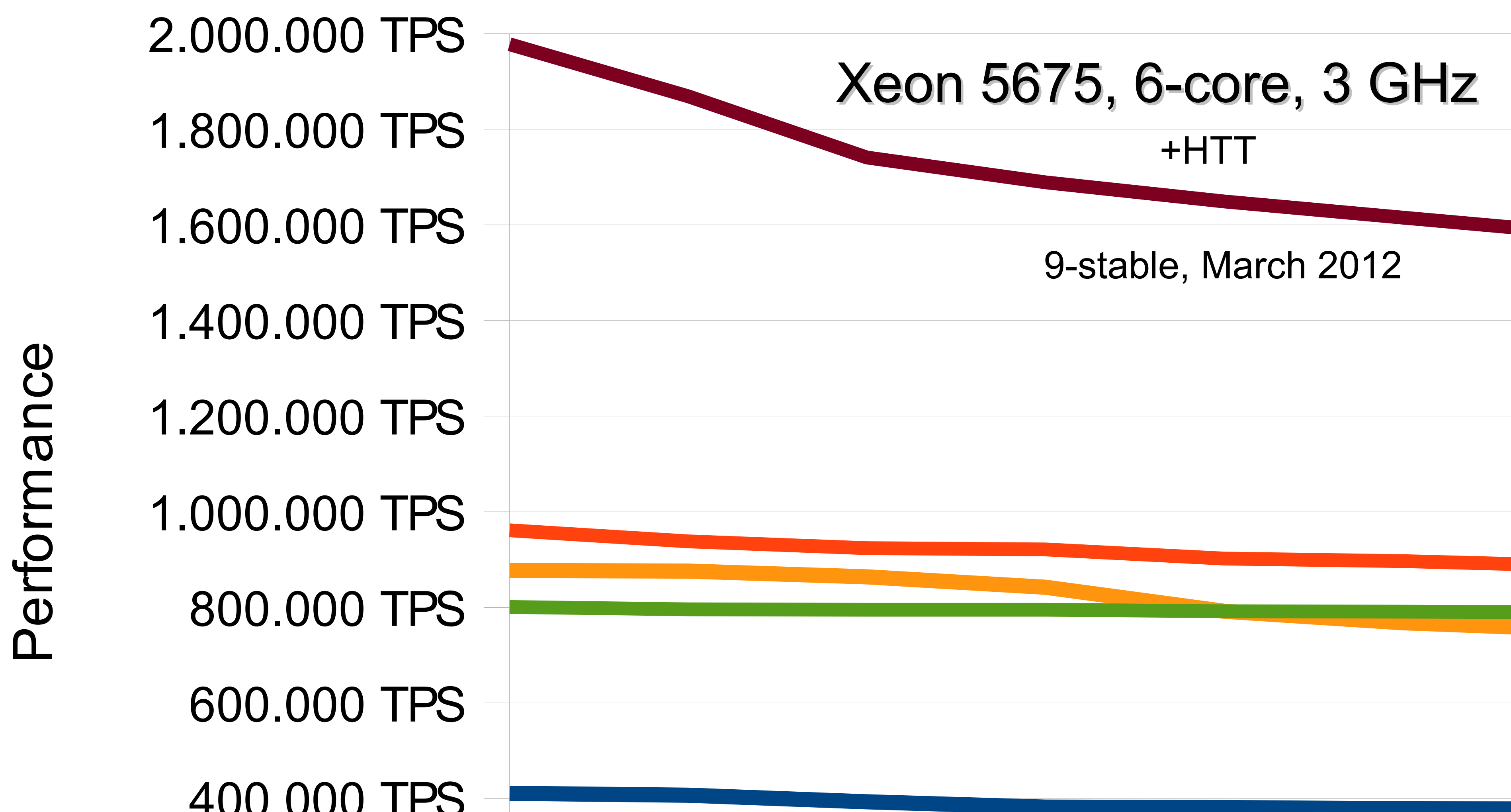
Notable performance optimizations

- “zero-copy” operation
  - Queries which do not involve complex processing or record aggregation are are satisfied directly from data structures
- “zero-malloc” operation
  - The code re-uses memory buffers as much as possible; the fast path is completely malloc()- and memcpy()-free
- Adaptive dynamic buffer sizes
  - malloc() usage is tracked to avoid realloc()

State of the art



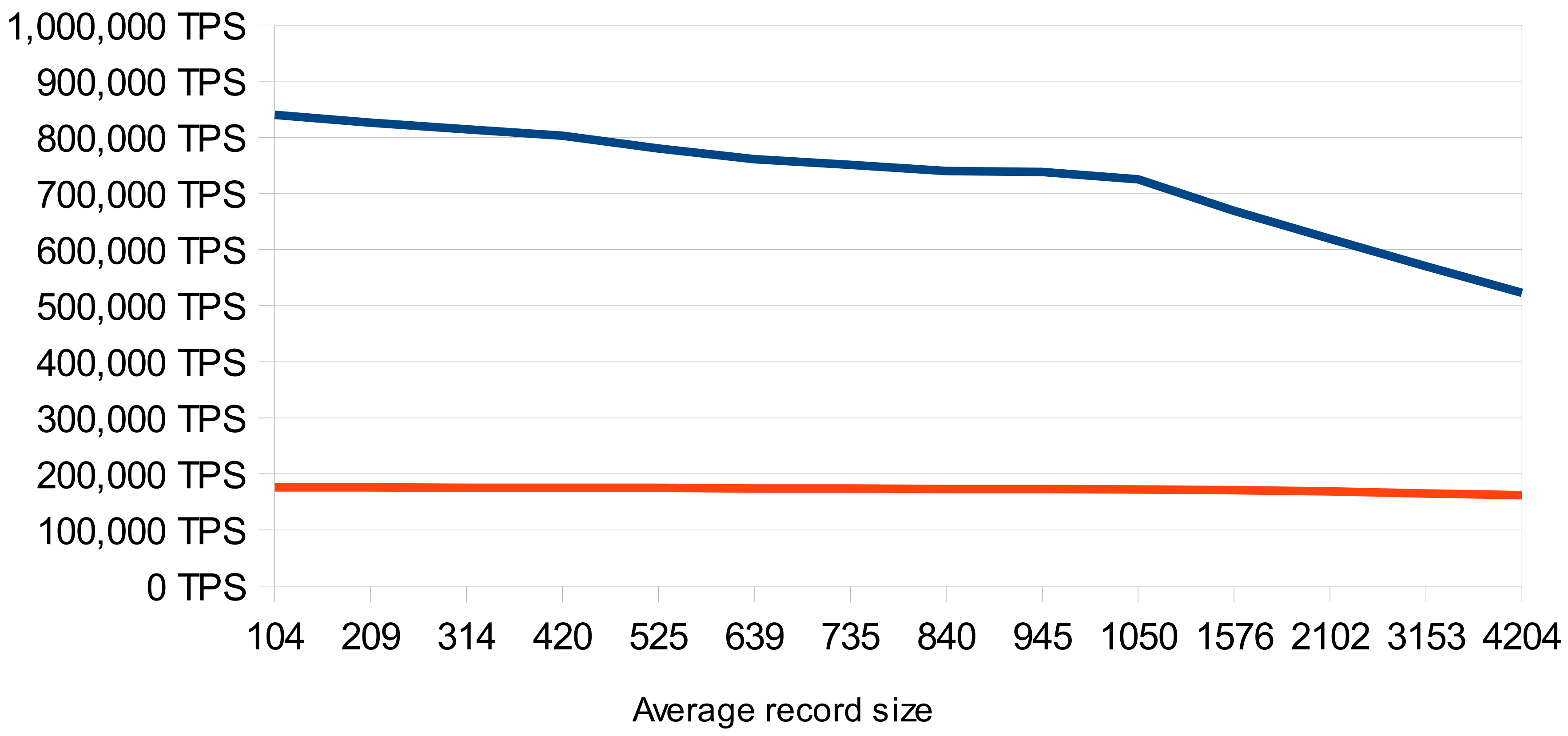
State of the art



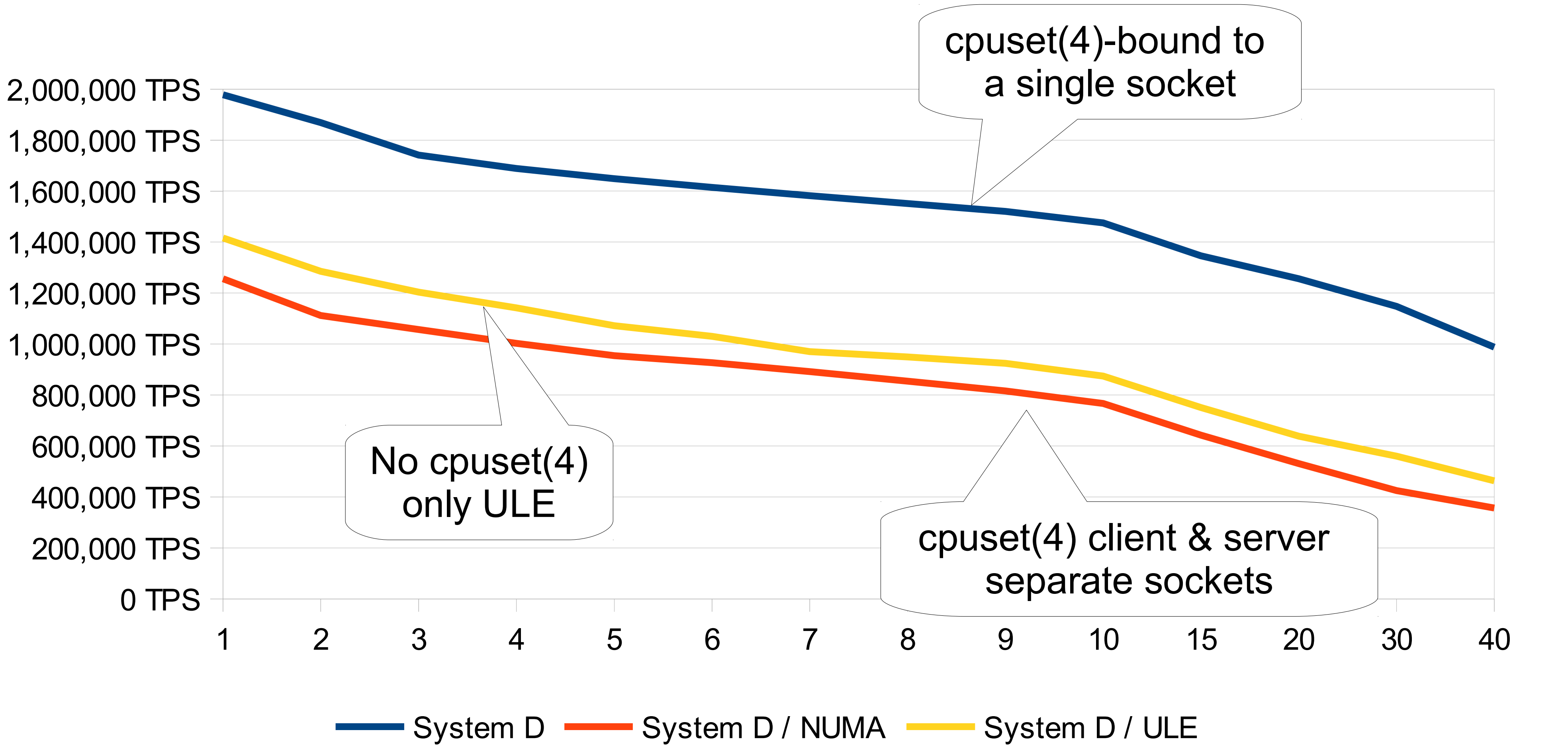
… under certain conditions

- The optimal, fast path (zero-memcpy, zero-malloc, optimal buffers)
  - Which is actually less important, we know that these algorithms are fast...
- Using Unix domain sockets
  - Much more important
  - FreeBSD's network stack (the TCP path) is currently basically nonscalable to SMP?
  - UDP path is more scalable … WIP

TCP vs Unix sockets

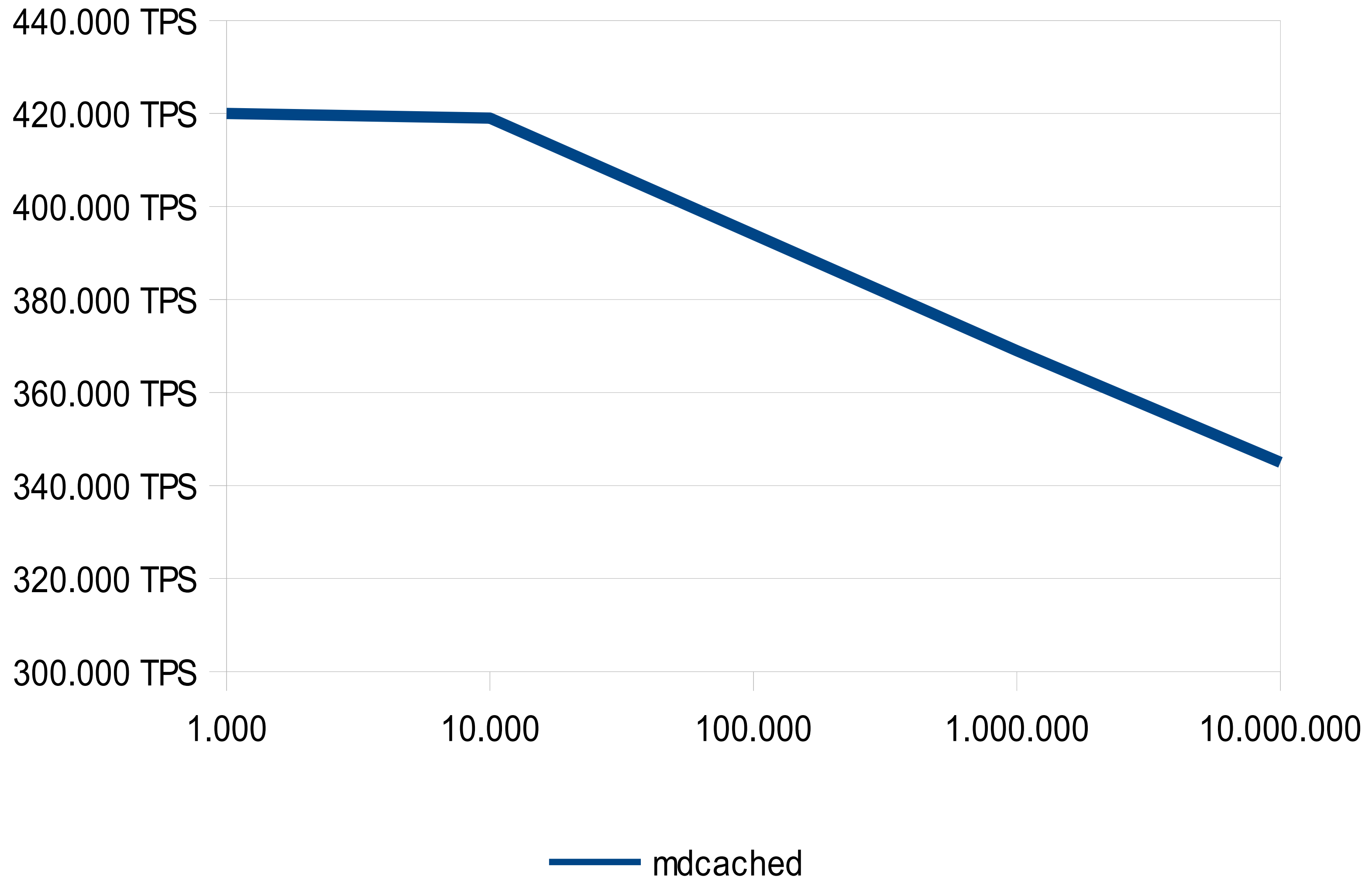


NUMA Effects



It's unlikely that better NUMA support would help at all...

Scalability wrt number of records



Bells & whistles

- Binary protocol (endian-dependant)
- Extensive atomic operation set
  - cmpset, add, fetchadd, readandclear
- “tstack” operations
  - Every tag (tk,tv) identifies a stack
  - Push and pop operations on records
- Periodic data dumps / chekpoints
  - Cache pre-warm (load from file)

Usage ideas

- Application data cache, database cache
  - Semantically tag cached records
  - Efficient retrieval and expiry (deletion)
- Primary data storage
  - High-performance ephemeral storage
  - Optional periodic checkpoints
- Data sharing between app server nodes
- Esoteric: distributed lock manager, stack

Bullet Cache

Balancing speed and usability
in a cache server

http://www.sf.net/projects/mdcached

Ivan Voras <ivoras@freebsd.org>