The Impact of Thread-Per-Core Architecture on Application Tail Latency

Pekka Enberg, Ashwin Rao, and Sasu Tarkoma
University of Helsinki
ANCS 2019
Introduction

- Thread-per-core architecture has emerged to eliminate overheads in traditional multi-threaded architectures in server applications.

- Partitioning of hardware resources can improve parallelism, but there are various trade-offs applications need to consider.

- Takeaway: Request steering and OS interfaces are holding back the thread-per-core architecture.
Outline

• Overview of thread-per-core
• A key-value store
• Impact on tail latency
• Problems in the approach
• Future directions
Outline

• Overview of thread-per-core
• A key-value store
• Impact on tail latency
• Problems in the approach
• Future directions
What is thread-per-core?

- Thread-per-core = no multiplexing of a CPU core at OS level
- Eliminates thread context switching overhead [Qin 2019; Seastar]
- Enables elimination of thread synchronization by partitioning [Seastar]
- Eliminates thread scheduling delays [Ousterhout, 2019]
Interrupt isolation for thread-per-core

• The in-kernel network stack runs in kernel threads, which interfere with application threads.

• Network stack processing must be isolated to CPU cores not running application thread.

• Interrupt isolation can be done with IRQ affinity and IRQ balancing configuration changes.

• NIC receive side-steering (RSS) configuration needs to align with IRQ affinity configuration.
Partitioning in thread-per-core

- Partitioning of hardware resources (such as NIC and DRAM) can improve parallelism, by eliminating thread synchronization.

- Different ways of partitioning resources:
  - Shared-everything, shared-nothing, and shared-something.
Shared-everything

CPU0  CPU1  CPU2  CPU3

DRAM

Data
Shared-everything

Hardware resources are shared between all CPU cores.
Every request can be processed on any CPU core.
Shared-everything

CPU0  CPU1  CPU2  CPU3

DRAM

Data

Data access must be synchronized.
Shared-everything

• Advantages:
  • Every request can be processed on any CPU core.
  • No request steering needed.

• Disadvantages:
  • Shared-memory scales badly on multicore [Holland, 2011]

• Examples:
  • Memcached (when thread pool size equals CPU core count)

Shared-nothing
Hardware resources are partitioned between CPU cores.
Shared-nothing

Request can be processed on one specific CPU core.
Shared-nothing

Data access does not require synchronization.
Shared-nothing

Requests need to be steered.
Shared-nothing

• Advantages:
  • Data access does not require synchronization.

• Disadvantages:
  • Request steering is needed [Lim, 2014; Didona, 2019]
  • CPU utilisation imbalance if data is not distributed well (“hot partition”)
  • Sensitive to skewed workloads

• Examples:
  • Seastar framework and MICA key-value store

Didona et al. 2019. Sharding for Improving Tail Latencies in In-memory Key-value Stores. NSDI ’19
Lim et al. 2014. MICA: A Holistic Approach to Fast In-memory Key-value. NSDI ’14
Shared-something
Hardware resources are partitioned between CPU core clusters.
No synchronization needed for data access on different CPU clusters.
Data access needs to be synchronised within the *same* CPU core cluster.
Shared-something

• Advantages:
  
  • Request can be processed on many cores
  
  • Shared-memory scales on small core counts [Holland, 2011].
  
  • Improved hardware-level parallelism?
    
    • For example, partitioning around sub-NUMA clustering could improve memory controller utilization.
  
• Disadvantages:
  
  • Request steering becomes more complex.

Takeaways

• Partitioning improves parallelism, but there are trade-offs applications need to consider.

• Isolation of the in-kernel network stack is needed to avoid interference with application threads.
Outline

• Overview of thread-per-core
• A key-value store
• Impact on tail latency
• Problems in the approach
• Future directions
A shared-nothing, key-value store

- To measure the impact of thread-per-core on tail latency, we designed a shared-nothing key-value store.

- Memcached wire-protocol compatible for easier evaluation.

- Software-based request steering with message passing between threads.
  - Lockless, single-producer, single-consumer (SPSC) queue per thread.
Shared-nothing

Taking the shared-nothing model...
KV store design

...and implementing it on Linux.
KV store design

In-kernel network stack isolated on its own CPU cores.
KV store design

Application threads are running on their own CPU cores.
KV store design

Message passing between the application threads.
Outline

• Overview of thread-per-core

• A key-value store

• Impact on tail latency

• Problems in the approach

• Future directions
Impact on tail latency

• Comparison of Memcached (shared-everything) and Sphinx (shared-nothing)

• Measured read and update latency with the Mutilate tool

• Testbed servers (Intel Xeon):
  • 24 CPU cores, Intel 82599ES NIC (modern)
  • 8 CPU cores, Broadcom NetXtreme II (legacy)

• Varied IRQ isolation configurations.
Impact on tail latency

(a) IRQ affinity not configured, and IRQ balance enabled.

(b) IRQ affinity not configured, and IRQ balance disabled.

(c) IRQ affinity configured, and IRQ balance disabled.
Impact on tail latency

(a) IRQ affinity not configured, and IRQ balance enabled.

(b) IRQ affinity not configured, and IRQ balance disabled.

(c) IRQ affinity configured, and IRQ balance disabled.
99th percentile latency over concurrency for updates

![Graph showing latency over concurrency for updates]

- Memcached (legacy)
- Sphinxd (legacy)
- Memcached (modern)
- Sphinxd (modern)
99th percentile latency over concurrency for updates

![Graph showing 99th percentile update latency over number of concurrent connections for Memcached (legacy), Sphinxd (legacy), Memcached (modern), and Sphinxd (modern). The graph compares latency as the number of concurrent connections increases, with Memcached and Sphinxd depicted in different colors and markers. The chart highlights the performance differences between legacy and modern versions of these technologies.](image-url)
99th percentile latency over concurrency for updates

No locking, better CPU cache utilization.
Latency percentiles for updates

![Graph showing latency percentiles for updates comparing Memcached (legacy) and Memcached (modern) as well as Sphinx and Sphinxd (legacy) and Sphinxd (modern).]
Takeaways

- **Shared-nothing** model **reduces tail latency** for update requests, because partitioning **eliminates locking**.

- More results in the paper:
  - **Interrupt isolation reduces latency** for both shared-everything and shared-nothing.
  - No difference for read requests between shared-nothing and shared-something (no locking in either case).
Outline

• Overview of thread-per-core

• A key-value store

• Impact on tail latency

• Problems in the approach

• Future directions
Packet movement between CPU cores

CPU0

Userspace
Kernel
Hardware

SoftIRQ Thread
NIC RX Queue
IRQ
Poll

Application Thread
Socket
DRAM

CPU1

Message Passing

CPU2

SoftIRQ Thread
Socket
NIC RX Queue
IRQ
Poll

CPU3

Application Thread
Socket
DRAM

Network
A packet arrives on NIC RX queue and is processed by in-kernel network stack on CPU0.
Packet movement between CPU cores

Application thread receives the request on CPU1.
Packet movement between CPU cores

Request is steered to an application thread on CPU3.
Request steering inefficiency

- Inter-thread communication efficiency matters for software steering:
  - Message passing by copying is a bottleneck. Avoiding copies makes the implementation more complex.
  - Thread wakeup are expensive, batching is needed, but it increases latency.
  - Busy-polling is a solution, but it wastes CPU resources in some scenarios.
Partitioning scheme and skewed workloads

- Partitioning scheme is critical, but the design decision is application specific. Not always easy to partition.

- Skewed workloads are difficult to address with shared-nothing model.
Outline

- Overview of thread-per-core
- A key-value store
- Impact on tail latency
- Problems in the approach
- **Future directions**
Request steering with a programmable NIC?

- Program running on the NIC parses request headers, and steers request to correct application thread [Floem, 2018].

- Eliminates request software steering overheads and packet movement cost.

- On Linux, the Express Data Path (XDP) and eBPF interface could be used for this.
OS support for inter-core communication?

• On Linux, wakeup needed for inter-thread messaging are performed using eventfd interface or signals, but both have overheads.

• Adding better support for *inter-core communication* in the OS would help.
Non-blocking OS interfaces

• Thread-per-core requires non-blocking OS interfaces.

• New asynchronous I/O interfaces, such as io_uring on Linux, will help.

• Paging and memory-mapped I/O are effectively blocking operations (when you take a page fault), and must be avoided.
Network stack scheduling control

- In-kernel network stack runs in kernel threads, which interfere with application threads.

- Configuring IRQ isolation is possible, but hard and error-prone. Better interfaces are needed.

- Moving the network stack to user space helps.
Summary

• Thread-per-core architecture addresses kernel thread overheads.

• Partitioning of hardware resources has advantages and disadvantages, applications need to consider different trade-offs.

• Request steering is critical: CPU and NIC co-design and better OS interfaces are needed to unlock full potential of thread-per-core.
Thank you!

Email: penberg@iki.fi

Home page: penberg.org
Backup slides
Read latency (99th)

![Graph showing 99th percentile read latency vs. number of concurrent connections for different systems: Memcached (legacy), Sphinxd (legacy), Memcached (modern), Sphinxd (modern).]
Read latency

![Graph showing read latency for different systems]

- Memcached (legacy)
- Sphinxd (legacy)
- Memcached (modern)
- Sphinxd (modern)