TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs

Guanyu Feng¹, Huanqi Cao¹, Xiaowei Zhu², Bowen Yu¹, Yuanwei Wang¹, Zixuan Ma¹, Shengqi Chen¹, and Wenguang Chen¹

¹ Tsinghua University, ² Ant Group
NVMe SSD Array with High Performance

- **Modern hardware**
  - More PCIe lanes (64 per CPU)
  - PCIe 4.0 / 5.0 Standard
  - U.2 Interface, 2.5" PCIe SSDs

- **NVMe SSD Array**
  - More PCIe attached SSDs
  - Higher bandwidth
  - Increasing storage density

- **High performance**

- **A dual-socket 2U commodity server**
  - Attaching 16 NVMe SSDs
  - 50 TB Capacity
  - 50 GB/s Bandwidth
  - 10M IOPS
Out-of-core Processing on NVMe SSD Array

- **NVMe SSD Array**
- **Pros:**
  - Cheap
  - Large
  - Fast
- **Cons:**
  - Block-wise I/O
  - High Latency
  - Limited Write Endurance

- **Cache in Memory**

- Extending memory capacity by NVMe SSD Array
- In-memory programs $\rightarrow$ Out-of-core programs
- Processing datasets larger than memory
Existing Cache: OS Page Cache

- Based on virtual memory
- Easy to use (mmap / swapping): **Not requiring code rewrites**
- Good **in-memory** performance on cache hits: Hardware TLB + MMU

- Poor **out-of-core** performance on cache misses
  - **Hardware**: TLB shootdown & Context switch
  - **Implementation**: Global locking (not scaling) & I/O Stack

- To efficiently use NVMe SSD Array: Non-trivial OS & CPU modifications

- **What about user-space solutions?**
Existing Cache: User-space Block Cache

- Databases & Data processing systems
- Better out-of-core performance
- Kernel-bypass I/O stack: efficiently using NVMe SSD Array
- Customizations: block-size, eviction policy ...

- Applying user-space cache: **High development cost**
  - Cache API calls
  - Block awareness

- Making the user-space cache **user-transparent** and **easy-to-use**
  - Not requiring code rewrites for in-memory programs

- Scalable in-memory programs ➔ Efficient out-of-core programs
Challenges and Techniques

- Not requiring code rewrites for in-memory programs
  - **Virtual Memory Interface**
  - **Compile-time Instrumentation**

- Virtual Memory Interface ➔ Per-access software address translations
  - **Multi-level Cache Design**
  - **Software Address Translation Cache (SATC)**

- Fully utilizing NVMe SSD Array
  - **Good Scalability**
  - **Message-passing Based Delegation + Lock-free Data Structure**
High-level Architecture of TriCache
Compile-time Instrumentation

User Application Code

load  store

TriCache LLVM Compiler Plugin

get_raw_ptr(load)  get_raw_ptr(store)

Instrumented Binary with TriCache Runtime
Compile-time Instrumentation

- Based on LLVM IR
- Adding address translation calls
  - Redirect memory operations to TriCache Runtime
  - Just like virtual memory
  - Support SIMD and Atomic OPs
- Modifying memory layout
  - Set the highest bit of addresses
  - Fast identifying memory types
- Libraries
  - Recompile by TriCache plugin
  - Library Hooking and Trap Handler
- TriCache memory allocations
  - Size threshold
  - Memory usage limitation
  - Manual specifies

```
%dst = load T* %ptr
%real_ptr =
  if high_bit(%ptr)
    get_raw_ptr(%ptr, LOAD)
  else
    %ptr
%dst = load T* %real_ptr

store T %src, T* %ptr
%real_ptr =
  if high_bit(%ptr)
    get_raw_ptr(%ptr, STORE)
  else
    %ptr
store T %src, T* %real_ptr
```

- User-space Virtual Memory
- Non-canonical Virtual Memory
- TriCache-space Virtual Memory
- Kernel-space Virtual Memory
An Example on TriCache

```c
size_t strlen(char* str) {
    len = 0;
    while (str[len] != 0)
        ++len;
    return len;
}
```

1. Compile to LLVM IR

```
%ptr = str + len
%data = load char* %ptr
```

2. Instrument

```
%ptr = str + len
%raw_ptr =
    get_raw_ptr(%ptr, LOAD)
%data = load char* %raw_ptr
```
Software Address Translation Cache

User Application Code

load \( \rightarrow \) store

TriCache LLVM Compiler Plugin

get\_raw\_ptr(load) \( \rightarrow \) get\_raw\_ptr(store)

Instrumented Binary with TriCache Runtime

User Thread

\( \uparrow \) get\_raw\_ptr

Direct SATC

\( \uparrow \) pin/unpin

Private SATC

\( \uparrow \) pin/unpin

Shared Cache

read/write

Backend Storage

Message Passing

User Application Code

TriCache LLVM Compiler Plugin

Instrumented Binary with TriCache Runtime

User Thread

\( \uparrow \) get\_raw\_ptr

Direct SATC

\( \uparrow \) pin/unpin

Private SATC

\( \uparrow \) pin/unpin

Shared Cache

read/write

Backend Storage

Message Passing
Software Address Translation Cache

- Multi-level Cache
  - Re-using translated pointers (Locality)
  - Human efforts ➔ Automatic mechanism
  - Higher levels: smaller, faster, hotter

- Three levels (Tri...Cache)

- Shared Cache: manage data
  - Shared by all user threads

- Private SATC
  - Thread-local: reducing Shared Cache operations
  - Thread-local segments and shared hot data

- Direct SATC
  - Direct mapping: fast lookups and maintenance
  - Thread-local (Fiber / Stackful Coroutine)
  - Consecutive operations: sequential reads / writes

- Inclusive Policy: avoiding broadcast
  - Reference counting: pin / unpin
**An Example on TriCache**

```c
size_t strlen(char* str) {
    len = 0;
    while (str[len] != 0) ++len;
    return len;
}
```

**Diagram:**

1. **Compile to LLVM IR**
   - `%ptr = str + len`
   - `%data = load char* %ptr`

2. **Instrument**
   - `%ptr = str + len`
   - `%raw_ptr = get_raw_ptr(%ptr, LOAD)`
   - `%data = load char* %raw_ptr`

3. **Pin**
   - `Return Raw Pointer`

4. **Read**
   - `Read into Memory`

5. **Return Raw Pointer**

6. **Pin**
   - `Return Raw Pointer`

7. **Pin**
   - `Return Raw Pointer`

8. **Return**
   - `Read into Memory`
Scalable Shared Cache

User Application Code

TriCache LLVM Compiler Plugin

Instrumented Binary with TriCache Runtime

User Thread

get_raw_ptr

Direct SATC

pin/unpin

Private SATC

User Thread

get_raw_ptr

Direct SATC

pin/unpin

Private SATC

User Thread

get_raw_ptr

Direct SATC

pin/unpin

Private SATC

MP Client

MP Server

Cache Partition 0

Cache Partition 1

Cache Partition ...

Cache Partition N

Shared Cache

read/write

Backend Storage

load

store

get_raw_ptr(load)

get_raw_ptr(store)
Scalable Shared Cache

- Message-passing based delegation
  - Single-threaded servers
    - Message Passing
    - Cache Management
    - Storage Operations
  - Easy to customize
  - More servers → Better performance
- Server: Metadata operations
- Client: Memory operations
- Cache-hit Fast Path
Evaluation

- Experimental Setup
  - CPU: 2x AMD EPYC 7742 CPUs
    - 128 cores, 256 hyper-threads
  - 512GB Memory
  - Testing Storage: 8x Intel P4618 DC SSDs
    - Capacity: 51 TB
    - Seq-read: 52 GB/s, Seq-write 42 GB/s
    - 4KB-read: 9.6M IOPS, 4KB-write: 3.9M IOPS
  - OS: Debian 11.1 with Linux kernel 5.10
  - Compiler: Clang 13.0.1
Evaluation: End-to-end Performance

- Graph processing
  - Small and random accesses on large datasets

- Key-value store
  - Hotness distribution and temporal patterns

- Big-Data Analytics
  - Sequential I/O throughput

- Graph Database
  - Atomic memory accesses and cache consistency
Evaluation: End-to-end Performance

- Outperform OS page cache by orders of magnitude on out-of-core processing
- Comparable to or even faster than specialized out-of-core systems
## Evaluation: Multi-level Cache Design

<table>
<thead>
<tr>
<th></th>
<th>Direct SATC</th>
<th></th>
<th>Private SATC</th>
<th></th>
<th>Shared Cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Miss Rate</td>
<td>Hit Cycles</td>
<td>Miss Rate</td>
<td>Hit Cycles</td>
<td>Miss Rate</td>
<td>Acc. Cycles</td>
</tr>
<tr>
<td>PageRank</td>
<td>0.003</td>
<td>52.6</td>
<td>0.063</td>
<td>321</td>
<td>0.626</td>
<td>2.36M</td>
</tr>
<tr>
<td>Shuffle Sort</td>
<td>0.001</td>
<td>63.0</td>
<td>0.001</td>
<td>162</td>
<td>0.969</td>
<td>1.68M</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>0.045</td>
<td>143</td>
<td>0.007</td>
<td>488</td>
<td>0.929</td>
<td>789K</td>
</tr>
</tbody>
</table>

- SATCs handle most of memory accesses
- Compare to Hardware Latency
  - 50 cycles ≈ a NUMA-local L3 cache hit / L2 false sharing
  - 150 cycles ≈ half of a NUMA-local memory access
  - 450 cycles ≈ a cross-NUMA memory access
More evaluation in the paper

- Micro-benchmarks
  - Compare TriCache with mmap and FastMap
  - Speedups under different access patterns

- Performance Breakdown
  - Different Storage Backend
  - Performance impact of Direct SATC and Private SATC
  - Performance under different numbers of threads
TriCache

- A new user-space cache mechanism
  - User-transparent and providing high out-of-core performance
- Virtual Memory Interface
- Multi-level Cache Design
- Good Scalability

Thank you!

Available at:  https://github.com/thu-pacman/TriCache

fgy18@mails.tsinghua.edu.cn
cwg@tsinghua.edu.cn