What Are You Waiting For?

Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!

13th Workshop on Accelerating Analytics and Data Management (ADMS22), September 2022, Sydney, Australia

What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!

Leonard von Merzljak
Technical University of Munich
leonard.von-merzljak@tum.de

Philipp Fent
Technical University of Munich
fent@in.tum.de

Thomas Neumann
Technical University of Munich
thomas.neumann@in.tum.de

Jana Giceva
Technical University of Munich
jana.giceva@in.tum.de

ABSTRACT

In the last ten years, SSDs achieved astonishing improvements in capacity per dollar and performance. Today, they are 30× cheaper than DRAM, and the difference is growing. Additionally, they are more than ten times faster than a few years ago, with a single SSD providing a throughput of 7 GB/s. Modern servers have enough PCIe lanes to directly attach multiple NVMe SSDs. That allows us to linearly scale the storage throughput and diminish the bandwidth gap between DRAM and SSDs. However, it requires a lot of parallel I/O requests to exploit multiple directly-attached SSDs, and the read latency is also very high.

In this paper, we propose to use asynchronous I/O and coroutines to continuously generate a lot of parallel I/O requests and hide the I/O latency. As a result, we get optimal throughput with up to 16× less compute resources than synchronous I/O, and we substantially flatten the performance cliff when exceeding main memory. We also show how to integrate coroutines into the code-generating, analytical DBMS Umbra and describe how we can call pre-compiled C++-Coroutines from the generated code. Finally, we present our new asynchronous index-nested-loop join algorithm that improves Umbra's end-to-end performance for analytical queries by up to 60%.

Table 1: Price and performance metrics of DRAM and SSDs.

	DRAM	SSD
config	8 × 64 GB	8 × 1.92 TB
cost-benefit	0.19 GB/$	5.8 GB/$
seq. read	152 GB/s (≥ 25 threads)	50 GB/s (≥ 4 threads)
rand. read	74 GB/s (≥ 72 threads)	48 GB/s (≥ 4 threads)
read. latency	181 ns (for 64 bytes)	73 µs (for 4 KiB)

even if they cache the entire database in-memory, the buffer manager is still the most expensive component of traditional systems. Since in-memory systems assume that the entire database fits into memory, they can substantially improve the performance by removing the buffer manager entirely.

1.1 In-Memory DBMSs Are Uneconomical

We currently observe two hardware trends that make us question the viability of pure in-memory systems and reconsider caching systems [26, 29]. First, the trend of rapidly dropping DRAM prices slowed down significantly in the last ten years [16]. Considering

Why Are We Interested in SSDs?

DRAM Is Expensive

- The trend of dropping DRAM prices slowed down significantly
- **The amount of data we want to analyze is ever-growing**
- ⇒ The cost of buying enough DRAM capacity increases disproportionally
- ⇒ **In-memory systems are increasingly becoming uneconomical**

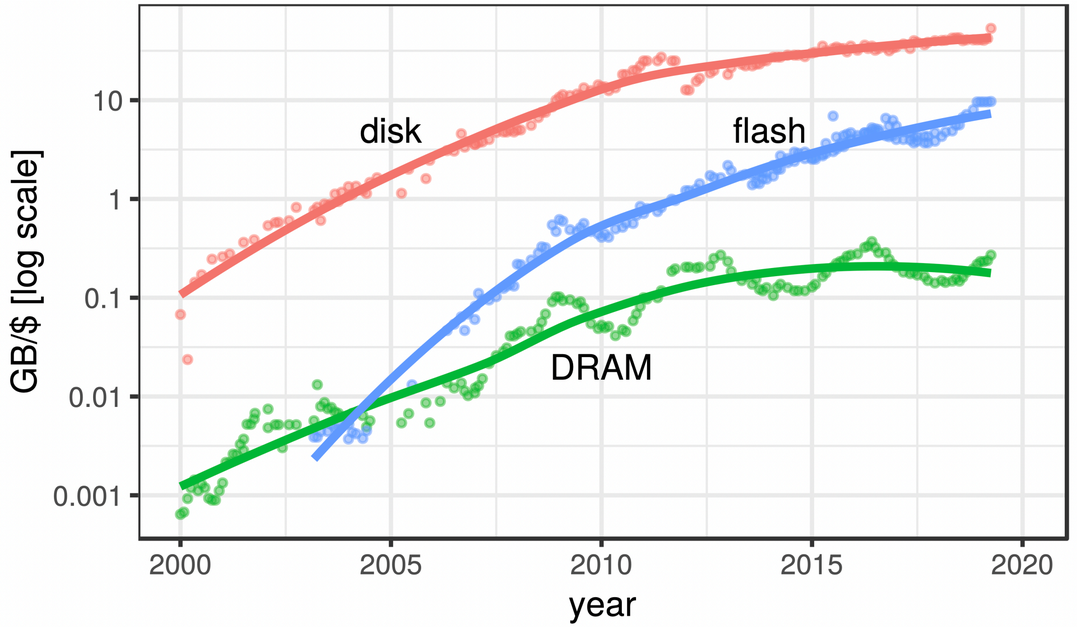

Figure 1: Historical disk, flash, and DRAM capacity per dollar.
data source: https://jcmit.net/memoryprice.htm
Figure copied from "Exploiting Directly-Attached NVMe Arrays in DBMS" (Haas et al., CIDR '19)

Why Are We Interested in SSDs?

SSDs Keep Improving

- SSDs are **30 times cheaper than DRAM**
- **7 GB/s read bandwidth** over 4 PCIe 4.0 lanes using the NVMe interface
- Modern CPUs have enough PCIe 4.0 lanes for 16 directly-attached SSDs (e.g., using RAID 0)
- **Theoretical read bandwidth of 112 GB/s**

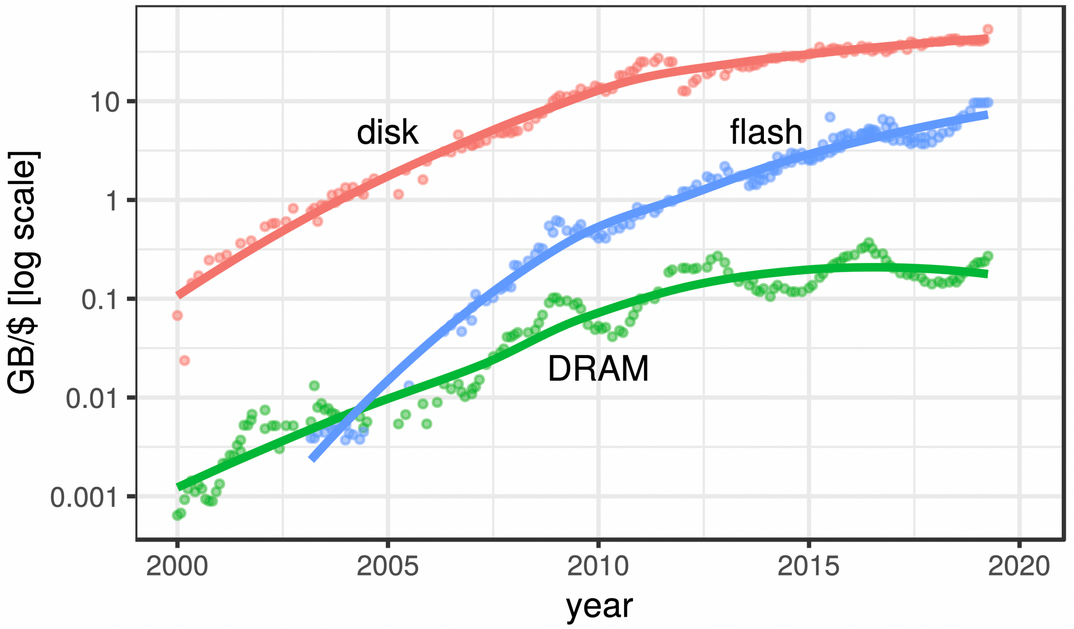

Figure 1: Historical disk, flash, and DRAM capacity per dollar.

data source: https://jcmit.net/memoryprice.htm

Figure copied from "Exploiting Directly-Attached NVMe Arrays in DBMS" (Haas et al., CIDR '19)

Exploiting SSDs Is Challenging

Keep All Flash Chips Busy

- SSDs consist of **dozens of flash chips**:
  - manage a subset of the storage cells
  - **can be accessed in parallel**
- How to achieve high bandwidth?
  - **Read hundreds of pages in parallel to provide work for all flash chips**

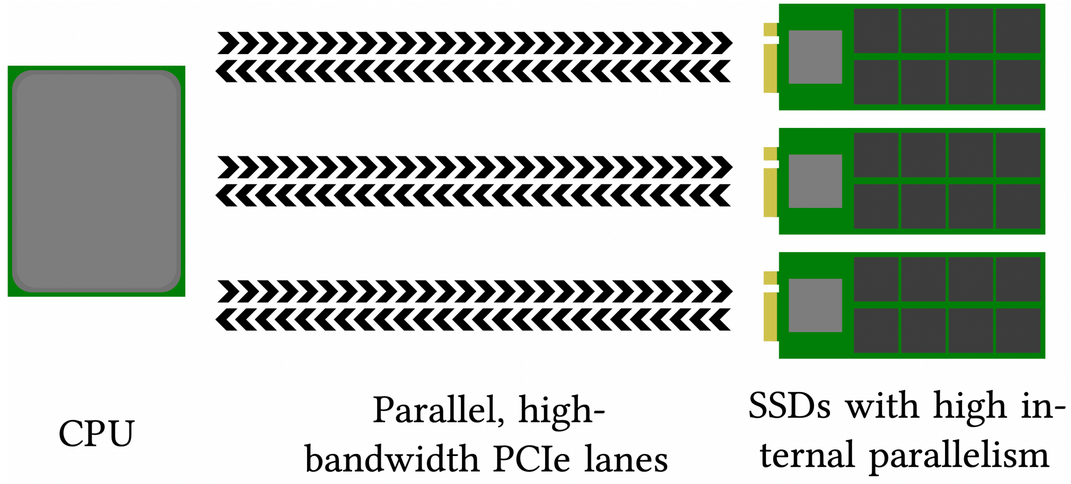

Figure 2: The storage architecture of modern SSDs.

Exploiting SSDs Is Challenging

High Read Latency

- **The latency of reading data from SSDs is much higher than from DRAM**
- With a synchronous (blocking) I/O interface, **threads spend a lot of time waiting**
- The CPU and the SSDs are underutilized

```
#include <unistd.h>

bool doPread(int fd, void *buf, size_t count, off_t offset) {
  while (count) {
    auto res = pread(fd, buf, count, offset);
    if (res < 1) {
      return false;
    }
    count -= res;
    buf = static_cast<char *>(buf) + res;
    offset += res;
  }
  return true;
}
```

What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!

Building Blocks
Asynchronous I/O

- `io_uring` (new Linux I/O interface) for asynchronous I/O
- Provides two operations:
  1. Submitting an I/O request (**non-blocking**)
  2. Waiting or polling for the completion of submitted requests
- **<u>Use asynchronous I/O to schedule hundreds of parallel I/O requests to provide work for all flash chips!</u>**

```
io_uring ring;

// Submit an I/O request
io_uring_sqe *sqe = io_uring_get_sqe(&ring);
io_uring_prep_read(sqe, fd, buf, count, offset);
io_uring_sqe_set_data(sqe, reinterpret_cast<void *>(42));
io_uring_submit(&ring); // non-blocking

// Do something useful in the meantime ...

// Wait for the completion of the I/O request
io_uring_cqe *cqe;
io_uring_wait_cqe(&ring, &cqe);
void *data = io_uring_cqe_get_data(cqe);
io_uring_cqe_seen(&ring, cqe);
```

Building Blocks

C++20-Coroutines

- For asynchronous I/O, we need to **suspend a function on an I/O request**
- A coroutine is a function that can **suspend execution to be resumed later**
- Sequential code that executes asynchronously
- **Use coroutines to hide the I/O latency by suspending a function on an I/O request and resuming another!**

```
task<bool> doAsyncRead(IOUring &ring, int fd, void *buf,
                       size_t count, off_t offset) {
  while (count) {
    auto res = co_await IOUringReadAwaiter{ring, fd, buf,
                                           count, offset};
    if (res < 1) {
      co_return false;
    }
    count -= res;
    buf = static_cast<char *>(buf) + res;
    offset += res;
  }
  co_return true;
}
```

Micro-Benchmarks

Asynchronous I/O for Query Processing

Hardware Overview

- AMD EPYC CPU with 64 cores (128 hardware threads)
- 512 GiB of DDR4-3200 RAM
- **8 Samsung PM9A3 PCIe 4.0 NVMe SSDs**
- Linux software RAID 0

Table 1: Price and performance metrics of DRAM and SSDs.

	DRAM	SSD
config	8 × 64 GB	8 × 1.92 TB
cost-benefit	0.19 GB/$	5.8 GB/$
seq. read	152 GB/s (≥ 25 threads)	50 GB/s (≥ 4 threads)
rand. read	74 GB/s (≥ 72 threads)	48 GB/s (≥ 4 threads)
read. latency	181 ns (for 64 bytes)	73 µs (for 4 KiB)

Asynchronous I/O for Query Processing

Experimental Setup

- Asynchronous I/O for **table scans**:
  - TPC-H Q1 (low-cardinality aggregation)
- Asynchronous I/O for **index lookups**:
  - TPC-H Q14 using an **asynchronous index-nested-loop join**
- LeanStore-based buffer manager using **direct I/O**
- Morsel-driven parallelism with a **coroutine-per-morsel approach**

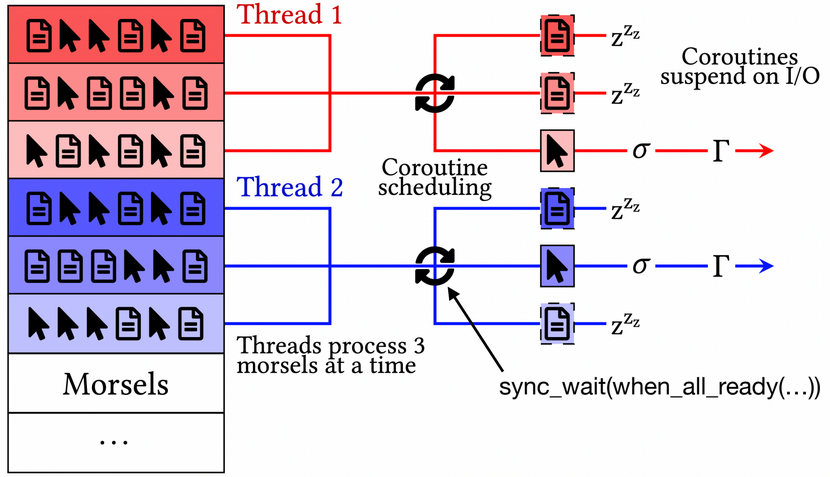

Figure 4: Threads fetch multiple morsels for table scans and start one coroutine per morsel.

Asynchronous I/O for Query Processing

Higher Throughput with Less Compute

- For sequential I/O, asynchronous I/O allows us to **reach higher throughput than synchronous I/O with 4 times fewer threads**
- For random I/O, **asynchronous I/O achieves better performance with 16 times fewer threads**
- Frees up resources for in-memory workloads, or allows downsizing the compute resources for more economical operation

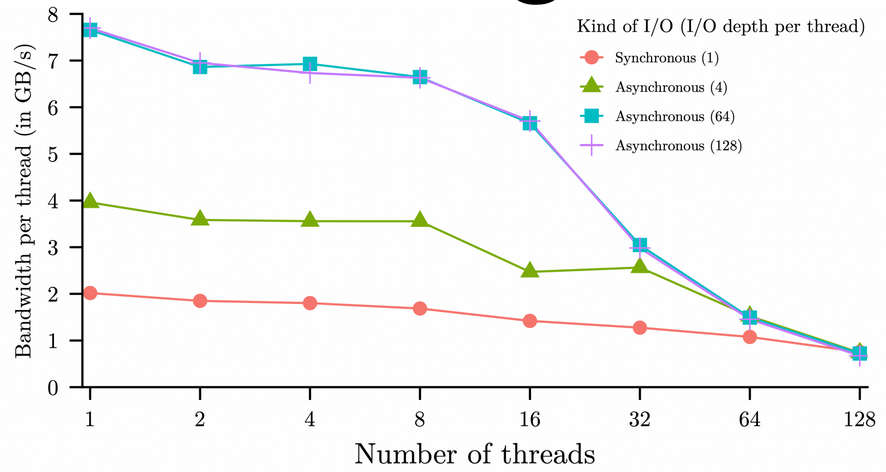

Figure 5: Throughput per thread of processing TPC-H Q1. Page size of 64 KiB, 60% cached.

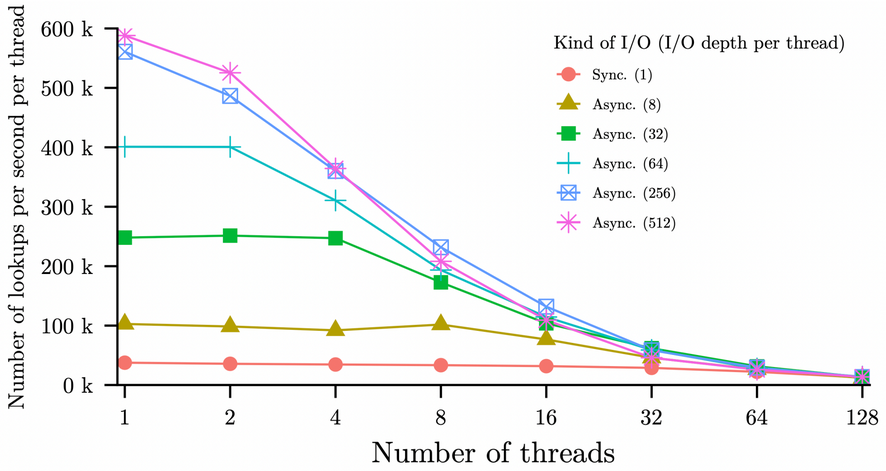

Figure 8: Lookups per second per thread of processing TPC-H Q14. Page size of 4 KiB, 60% cached.

Asynchronous I/O for Query Processing

Graceful Degradation for Out-Of-Memory

- What happens if the working set's size exceeds the memory capacity?
- **Even with 90% cached**, the throughput of synchronous I/O is still 15 GB/s below the throughput of asynchronous I/O **when nothing is cached**
- **Asynchronous I/O gets very close to the optimal throughput**

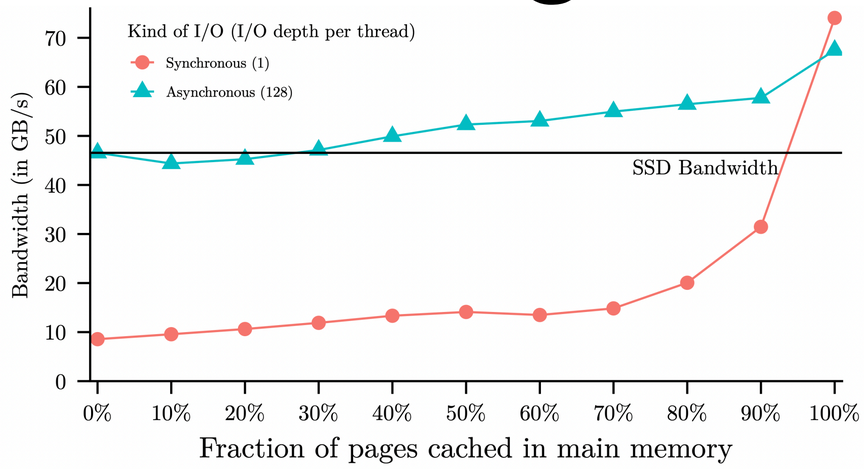

Figure 6: Throughput of processing TPC-H Q1. Page size of 64 KiB, 8 threads.

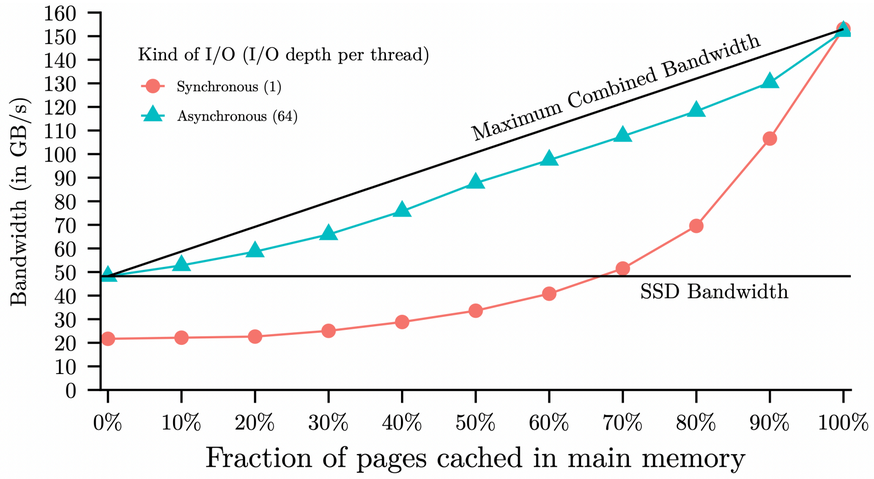

Figure 7: Throughput of processing TPC-H Q1. Page size of 64 KiB, 32 threads.

Asynchronous I/O in a Code-Generating System

Asynchronous I/O in Umbra

Umbra Does Not Generate C++ Code

- Umbra: Code-generating system written in C++ **supporting out-of-memory execution**
- Umbra allows us to **call pre-compiled C code**
  - We can **wrap a C++-Coroutine into C code** (see paper)
- We need to be able to **await C++-Coroutines from the generated code**
- Therefore, we need **Codegen-Coroutines:**
  - Compilation backends **translate a Codegen-Coroutine into a state machine** (see paper)

Asynchronous I/O in Umbra

Add Support for Asynchronous Index-Nested-Loop Joins

- Evaluate TPC-H Q4, Q5, and Q10 on SF100
- 16 threads, I/O depth per thread of 256, 128 tuples per coroutine
- Direct I/O
- Varied the size of the buffer manager to **simulate out-of-memory**
- Increasing the number of threads to 64 or 128 makes performance difference disappear

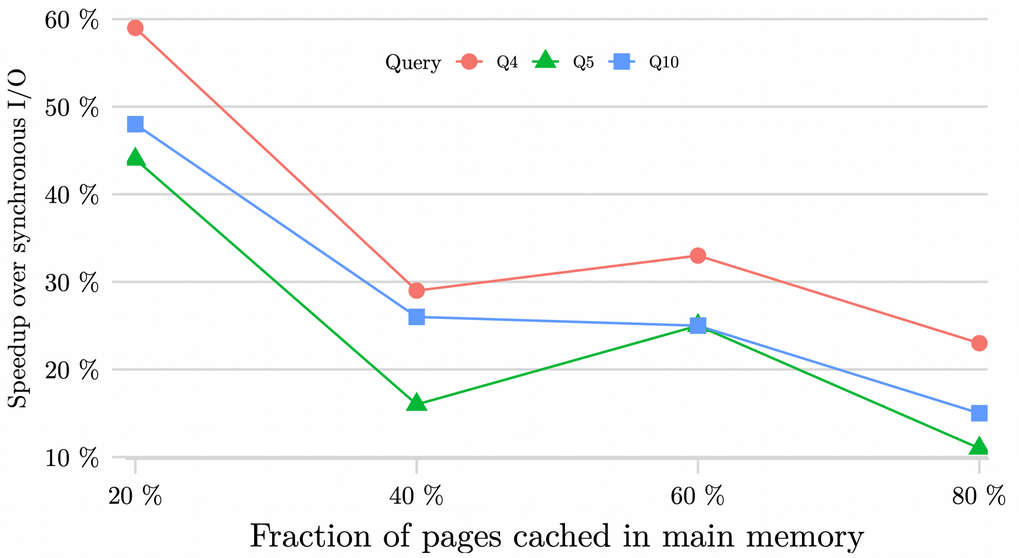

Figure 10: Asynchronous index-nested-loop joins on the TPC-H benchmark.

Conclusion

- Reach high throughput with fewer threads than synchronous I/O
- Flatten the performance cliff when going out-of-memory
- Low end: more economical, high-performance data analysis
- High end: process terabytes of data with near in-memory performance on a single node