Design of a request/response buffering application for I/O intensive workloads

Florian Grötschla, Giovanna Lehmann Miotto, Roland Sipos
Overview

1. Detector readout challenges
2. Specification and design principles
3. Components and implementation details
4. Demonstration
5. Outlook
Readout challenges

1. **Wide range of frontend electronics:** TPC electronics, SiPM readout, pixel chips, etc.
2. **Variety of aggregator I/O devices:** COTS servers, PCIe FPGA carrier boards, NICs, etc.
3. **Payload characteristics:** different arrival rate and payload size combinations (fixed/variable)
4. **Quasi real-time performance:** Requirement for high throughput (∼100GBit/s), low latency, and scalability
Frontends

TPC readout

SiPM readout

raw data

different rates and payload sizes

COTS servers

FPGA boards

NICs
Readout specification

• Support all possible front-end types: be agnostic about data rate and payload size
• Buffer received data for a specified/maximum amount of time
• Respond to data requests (with time-windows)
  → Buffered data can be indexed to maintain search-ability
• In-flight data processing: Error and consistency checks with custom algorithms (e.g.: hit-finding) also supported
• Persist data on command for requested time
Readout components

- **Readout Type**: A wrapper for the raw data that provides functionalities needed by the buffer and other components
- **Latency Buffer**: Provide data structure for data buffering and lookup mechanism of indexed data
- **Frame Processor**: Pre- and post-processing of raw input data
- **Request Handler**: Respond to data requests, handle recording
- **Readout Model**: Contains all other components, calls interface implementations and hands off resources (e.g.: buffer)
Dataflow diagram

Frontend domain
- Frontend (TPC)
- Frontend (PDS)

Data processing domain
- Data reception
- Pre-processing pipeline
- Parallel post-processing

Buffering and storage domain
- Latency Buffer

Data request/response domain
- Data request/response mechanism
- Data recording (persistency)
- Local data store

Software framework

Raw data flow:
- Frontend (TPC) -> Data reception -> Pre-processing pipeline -> Latency Buffer
- Frontend (PDS) -> Data reception

Metadata:
- Data lookup
- Extracted data
- Issue request
- Data response
- Write data

Wednesday, 1st December 2021
ACAT 2021
Concepts and models

• Diverse set of frontends, having varied characteristics in payload arrival rate, size, and order

• **Main goal**: Avoid reimplementation and code duplication for different frontends
  → Keep the core functionalities fully generic

• **Concepts**: Well defined interfaces

• **Models**: Interchangeable implementations of the concepts
Latency buffers

• Stores data and provides lookup routine based on unique identifier (e.g.: timestamp) in frames

• **Fixed rate queue**: Offset calculation for lookup mechanism → Implementation based on *Folly* lock-free queue → Extension with binary search capability

• **Skiplist**: Uses *Folly* concurrent skiplist for non-ordered data

• Memory allocation strategies supported: NUMA aware, aligned

  **All implementations are interchangeable!**
Frame Processor

- **Preprocessing pipeline:** Tasks can be registered to a sequence of operations that are executed for every payload
- **Parallel postprocessing:** One thread for every registered function is created that is fed with the payloads
- Preprocessing can change frame, postprocessing works on a constant pointer
- Postprocessing works directly on a pointer to the frame in the latency buffer, no extra copies are involved
Request handling

- Request handler threads deep-copy data from latency buffer
- Read-only, so handle them in parallel via thread pool
- Periodic cleanup thread removes old data from latency buffer
- Synchronize cleanup and requests to avoid interference
- Waiting requests: Postpone requests that are for not yet present data
Data recording

• High bandwidth requirement imposes optimized implementation (exploit kernel features like O_DIRECT)
• Use of boost streams and support for compression
• 2 recording modes are supported
  • Deep-copy: From latency buffer to boost stream and writer
  • Zero-copy: From aligned latency buffers, directly to storage device
Demonstration

- The library is successfully integrated and used in different scenarios
  - With payload software emulators (fixed rate & size, variable rate & size)
  - With FPGA carrier boards (FELIX readout)
    - For the TPC readout of DUNE Vertical-Drift ColdBox (WIB frontend)
  - With NICs
    - For the photodector of ProtoDUNE-SP (SSP frontend)
- Successful tests of the recording implementations using different storage media
Server: 2S Intel Cascade Lake (Xeon(R) Gold 6242), 64 cores with 192GB RAM
Left: memory I/O: < 60% bandwidth utilization, Right: total CPU load per socket: 50%
Workload: Receiving at 166kHz x 5568 Bytes x 10 links = 8.6 GB/s and software hit finding
Recording performance

Left: same setup as before, Right: 3 consecutive recordings of 100 seconds each
Writing 10 links (8.6GB/s) to a software RAID0 of 4 NVMe SSDs (Samsung 970 Pro 1TB)
Outlook

• Main focus is on **scalability** and **resource locality** studies
  → Optimize CPU and memory bandwidth utilization via dynamic thread affinity balancers
  → Improve performance on multi-socket systems, exploiting NUMA awareness and socket interconnect capabilities

• Integration of the generic readout library into other DAQ software frameworks (e.g.: DAQling)
Modularity through templating

Readout for WIB

```cpp
auto readout_model = ReadoutModel<
    // Readout Type: WIB Struct
    WIBSS,
    // Request Handler
    DefaultRequestHandlerModel<WIBSS,
        FixedRateQueueModel<WIBSS>>,
    // Latency buffer
    FixedRateQueueModel<WIBSS>,
    // Frame Processor
    WIBFrameProcessor>(run_marker);
readout_model.init(args);
```

Readout for DAPHNE

```cpp
auto readout_model = ReadoutModel<
    // Readout Type: DAPHNE Struct
    DAPHNESS,
    // Request Handler
    DAPHNEListRequestHandler,
    // Latency buffer
    SkipListLatencyBufferModel<DAPHNESS>,
    // Frame Processor
    DAPHNEFrameProcessor>(run_marker);
readout_model.init(args);
```