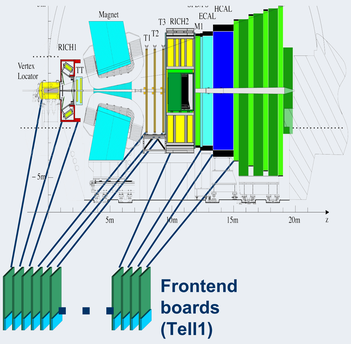
Optimization of the HLT Resources in the LHCb Experiment

M.Frank, C. Gaspar, E v. Herwijnen, B. Jost, N. Neufeld, R. Schwemmer
(CERN / LHCb, 1211 Geneva 23, Switzerland)

The software based high level trigger processing (HLT) of the LHCb experiment is based on nodes with multiple cores. Using process based parallelization to filter particle collisions from the LHCb experiment on such nodes leads to expensive consumption of memory and hence significant cost increase. We present an approach to minimize the resource consumption of the filter applications and to reduce the start-up time. Described is the duplication of threads and the handling of files open in read-write mode when duplicating filter processes and the possibility to bootstrap the event filter applications directly from preconfigured checkpoint files. This led to a reduced memory consumption of roughly 60 % in the nodes of the LHCb HLT farm and a reduction of the configuration time of more than 90 %.



Frontend boards (Tell1)

Readout Network

High Level Trigger Farm
- Selects particle collisions with interesting heavy flavor decays.
- Output rate 2-10 kHz

Storage System

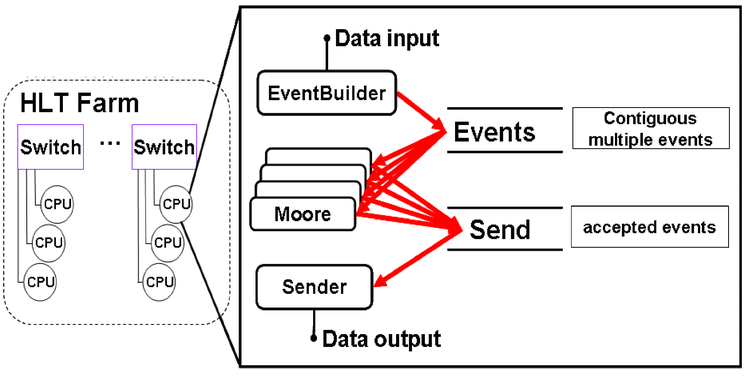
LHCb DAQ

LHCb is designed to exploit the finite lifetime and large mass of charmed and beauty hadrons to distinguish heavy flavor particles from the background in inelastic pp scattering. The second level or High Level Trigger (HLT) is purely software based.

The **front end readout boards (TELL1 boards)** send data at a rate of roughly 1 MHz through a **switching network** to the HLT farm nodes, where software algorithms decide whether the event is to be accepted and sent to the storage system for later offline analysis.
The **HLT hardware** consists of roughly 1500 dual processor units grouped to 57 subfarms, which host about **25000 trigger processes** (Moore). On each node execute up to 32 trigger processes as shown in the schematic below. These trigger processes occupy many resources and use considerable time to initialize. We were suffering from:
- **Disk access:** (shared nfs) during configuration (libraries and data)
- **Memory usage:** Resident size of one trigger process ~ 1.7 GB
- **Configuration time:** typically around 20 minutes for the entire farm

The problems are largely independent and were addressed independently. The physics software performing the event filtering based on the Gaudi framework was unchanged. All optimizations were done transparently.

Overall memory reduction 60 %
Configuration time reduced > 90 %



Minimizing Disk Access

All 25000 trigger processes start simultaneously and access the files at the sae time:
- Several hundred MB of configuration data (fieldmap, conditions, etc)
- Several hundred MB of library images.
- Python reads many precompiled files
- Significant: file lookup in PATH, PYTHONPATH, LD_LIBRARY_PATH

Limit:
NFS latency and load

Most of the files do not change between consecutive restarts

Solution:
local cache of read-only files on worker node (fuse fs)
=> On the first access the file is copied from the network to a ram disk.
=> On the next restart, subsequent file accesses or file lookups are local.
=> Immediate response, no latency

Improves startup time significantly.

Minimizing Memory Usage

- Memory footprint at startup roughly 1.3 GB.
- While processing stabilizes at 1.7 GB.
- For 24 processes (12 cores * hyper-threads) not affordable

But
- 60 % of the memory used is written once and accessed in read-only mode.
- Memory objects representing the detector geometry, the description of the magnetic field etc. are never modified.

Linux allows to share read-only memory between processes,
if these have a common ancestor.
⇒ Need to fork children after initialization phase will only allocate physical memory if page is modified

Forking: Requirements
⇒ All existing threads must be preserved. Fork does not replicate existing threads.
⇒ Open file handles must be preserved. After fork file handles are shared. Replicate temporary R/W files for child.
⇒ Reestablish communication layer to ECS.

Forking: Actions (1)
⇒ Send signal to all child threads (syscall tkill)
⇒ Suspend threads by waiting on a Futex cell
⇒ Save thread state (registers, TLS)
⇒ Save all open file descriptors (file state, position and the content of temporary files)
⇒ Install child specific environment (process name, etc) used to access event data and control the process by ECS
⇒ Fork child (syscall): Preserve thread library

After fork:
⇒ Each child restores the file access, recreates the threads and re-establishes the ECS communication layer.
⇒ The parent process acts as a watch-dog and in case of a premature end forks a replacement.

(1) See also: J.Ansel, G.Cooperman, M.Rieker, Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux, PDPTA'06

Minimizing Configuration Time

HLT trigger process:
=> many collaborating components with individual configuration
=> Detector description & Co
Time consuming initialization process O(several minutes)

Alternative: Checkpoint trigger processes
⇒ Initialize one HLT process
⇒ Dump process (checkpoint)
⇒ Restart process from checkpoint

Checkpoint Requirements:
As for forking

Checkpoint content:
⇒ Environment of the parent process
⇒ State of all open file descriptors.
⇒ All mapped memory sections including the heap and stack. If memory sections are readable, save the data content.
⇒ The binary code necessary to restore file descriptors and memory mappings.
⇒ For restore optimization all linked shared libraries are optionally cached.

The checkpoint is complete and does not require further images.

Process restart (static linked exe)

⇒ Avoid address clashes
⇒ Open checkpoint and restore libs
⇒ Restore environment
⇒ Maps the binary restore code at the location present at write.(1)
⇒ Unmap all other memory areas. Switch to temporary stack(1)
⇒ map images, restore files, switch to regular stack.(1)
⇒ Restore threads and continue like for forking

Distributing Checkpoint Files

Checkpoint file size ~1.8 GB
⇒1400 workers:: ~2.5 TB
⇒ gzip: ~1.8 GB → ~350 MB

Checkpoint file distribution or
access with nfs impossible

Solution:
Download files using BitTorrent protocol

⇒Similar problem as big file sharing networks have
⇒**Every client is a server** if it has a complete copy or fragments of the file
⇒ Integrated **download rate rises exponentially** and is **network limited**.
⇒ **Limit communication** between clients to not overload network

2-Tier Implementation
⇒ Trigger process issues request to the local loader
⇒ Request forwarded to the loader on the sub-farm controls node,
⇒ which forward to primary seeder with file access.
⇒ Primary file loader serves as a seed to all subfarm controls nodes.

⇒ Each of the subfarm controls node only serves limited number of clients.
⇒ Subfarm controls node has a local cache of checkpoint files