I/O aspects for parallel event processing frameworks

Workshop on Concurrency in the many-Cores Era

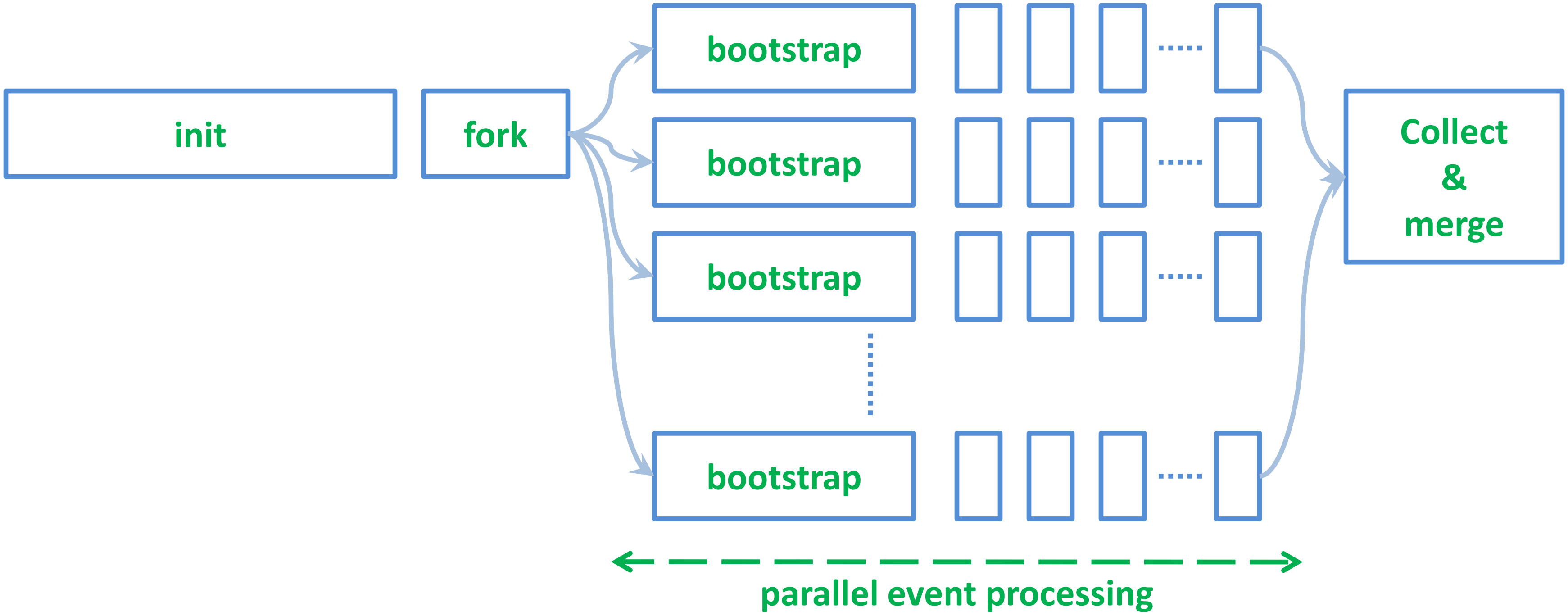
Peter van Gemmeren (Argonne/ATLAS)

Outline

- AthenMP
- Input / Output considerations:
  - What’s bad now, waste wall, CPU, memory and disk
- Developments:
  - Scatter
  - Input format

AthenaMP: The ATLAS multi-core Control framework

- In initial AthenaMP implementation, I/O is somewhat of an afterthought
  - Each worker node produces its own output file, which need to be merged after all worker are done.
  - Done in serial, can take significant amount of wall clock time.

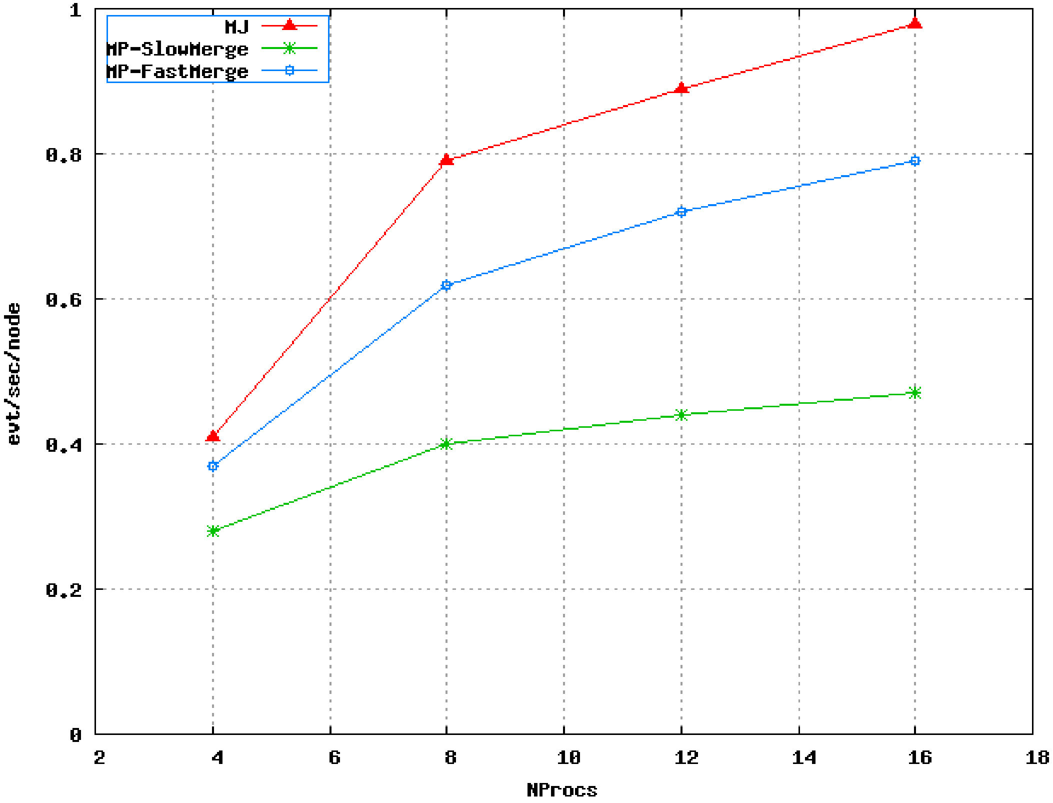


This is not a yet a multi-core I/O framework!

- **Read data**: A process (initialization, event execute,…) reads part of the input file (e.g., to retrieve one event).
  - All worker use the same input file.
    - Multiple access may mean **poor read performance**, especially if events are not consecutive.
- **Uncompress / Stream ROOT baskets**: Each worker will retrieve its own event data, which means reading multiple ROOT baskets, uncompressing them and streaming them into persistent objects.
  - ROOT baskets contain object member of several events, so multiple worker may use the same baskets and each of them will uncompress them independently:
    - **Wastes CPU time** (multiple uncompress of the same data)
    - **Wastes memory** (multiple copies of the same Basket, not shared)
- **Write data**: Each process writes its own output file, **which need to be merged**.
- **Compress / Stream to ROOT baskets**: Writers compress data separately.
  - Suboptimal compression factor (costs storage and CPU time at subsequent reads)
    - **Wastes memory** (each worker needs its own set of output buffer)

The Result: Control framework is only one side of the coin…

- Each process writes its own output file, **which need to be merged**.
- Which is done in serial and can kill you.
- Previous Tier0 tests of athenaMP found 50% event throughput reduction mainly caused by serial merge



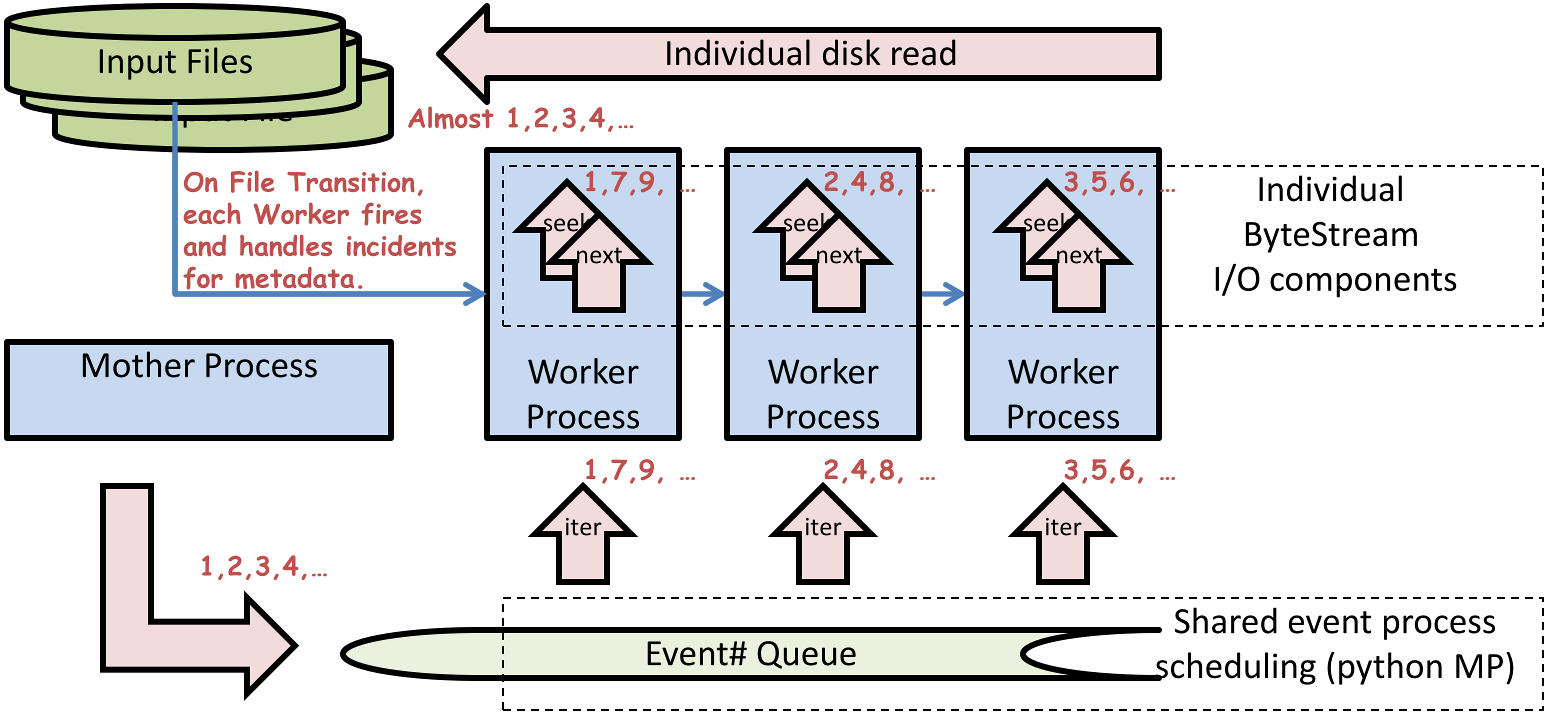
Plot stolen from V. Tsulaia

Need to develop multi-core I/O framework

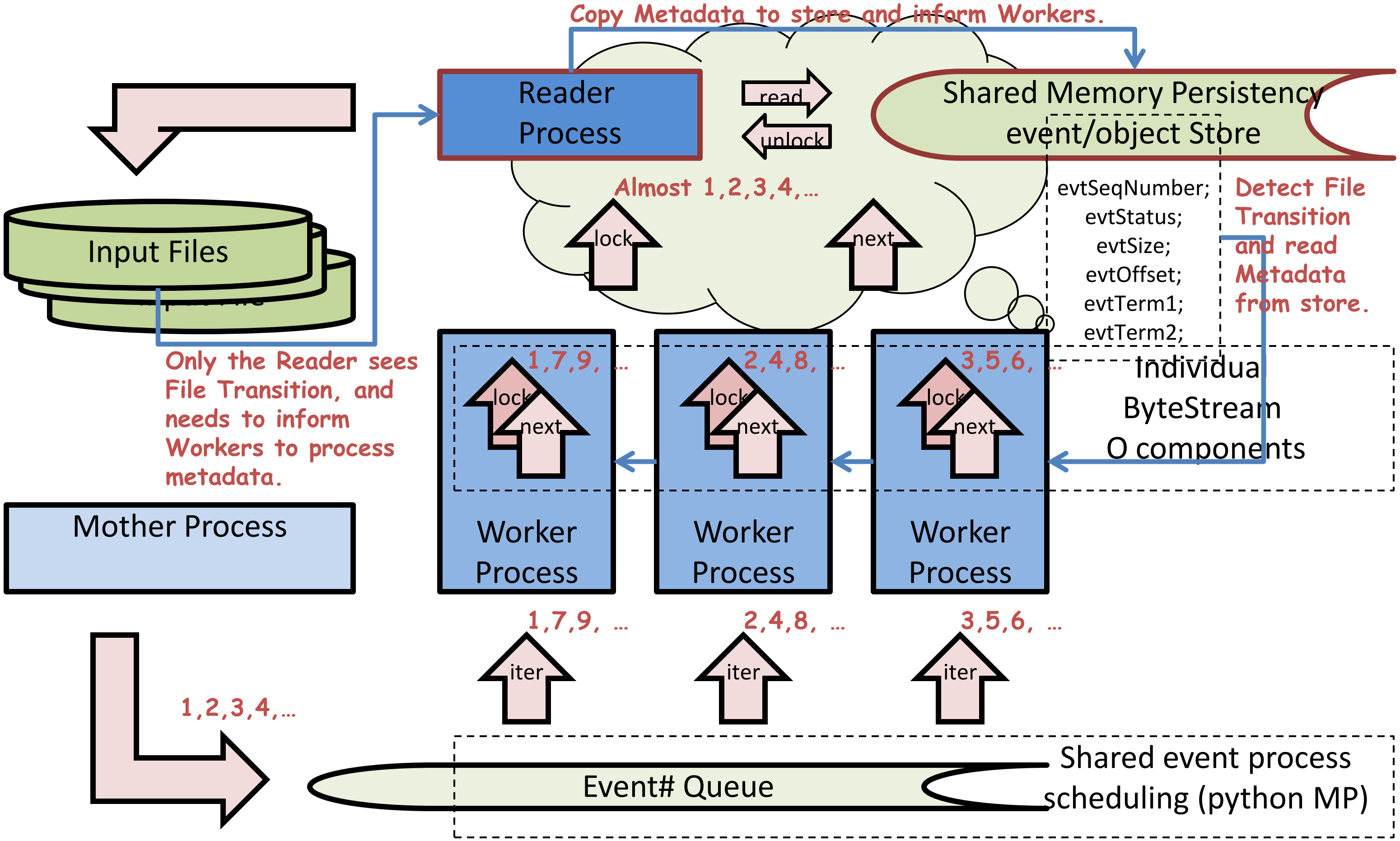
- Input:
  - Event/Object Scatter by dedicated Reader Process:
  - Possible approaches:
    - Event retrieve:
      - The single reader, reads all data for many events, and assembles ‘persistent events’.
      - The reader provides these events to the worker via shared memory.
    - Object level retrieve:
      - A single reader, reads all DataHeader and provides these to the worker via a queue.
      - Using AthenaPOOL/StoreGate object retrieval mechanism, a request is send to a reader.
      - The reader reads the persistent data object and sends it to the worker.
- The simplest implementation is for RAW data
  - No real gain expected, but good exercise

Current AthenaMP Architecture for reading ByteStream (RAW) in event queue mode

- *Event scheduling is managed by the mother process using a python MP queue with event numbers for workers to iterate over.*
- *I/O is done by worker, which first seek and than read the scheduled event*
- *File access is almost sorted*
  - *only when one worker passes another in the seek to next step, things get out of order. Can happen, but not often*



Alternative: Architecture for sharing ByteStream events in queue mode, no metadata yet



Copy Metadata to store and inform Workers.

Reader Process

read

unlock

Shared Memory Persistency event/object Store

Input Files

Almost 1,2,3,4,…

lock

next

evtSeqNumber;
evtStatus;
evtSize;
evtOffset;
evtTerm1;
evtTerm2;

Detect File Transition and read Metadata from store.

Only the Reader sees File Transition, and needs to inform Workers to process metadata.

1,7,9, …

2,4,8, …

3,5,6, …

lock

next

Individual ByteStream O components

Mother Process

Worker Process

Worker Process

Worker Process

1,7,9, …

2,4,8, …

3,5,6, …

iter

iter

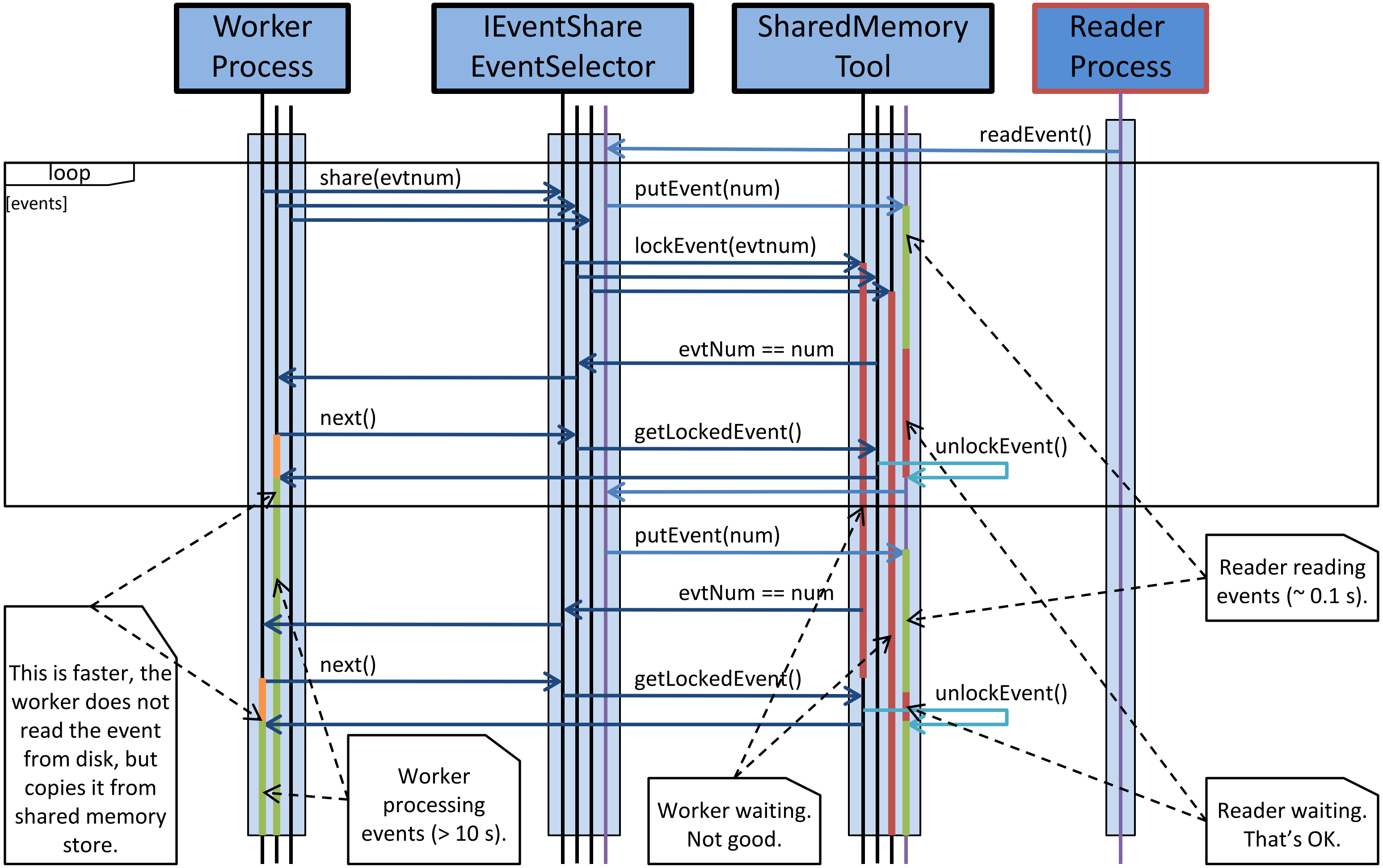
iter

1,2,3,4,…

Event# Queue

Shared event process scheduling (python MP)

Sequence Diagram for RAW sharing



Reader reading events (~ 0.1 s).

This is faster, the worker does not read the event from disk, but copies it from shared memory store.

Worker processing events (> 10 s).

Worker waiting. Not good.

Reader waiting. That's OK.

Some Testing

- Some initial performance testing:
  - No differences seen (or really expected) for up to 8 Workers.
    - But no penalty was observed either, which may be a surprise for a ‘proof of concept’ implementation that is not optimized
  - Small (~5%) slowdown with 16 Workers and single Reader.
    - Single reader does get congested as it is currently only pre-reading 1 event
- Could easily extend framework to allow multiple dedicated Reader Processes assigned to provide events to groups of Workers
- And of course, metadata handling requires special attention…

Second leg: Parallel ‘chunked’ access to ROOT input files.

- ROOT baskets contain object member of several events, so multiple worker may use the same baskets and each of them will uncompress them independently:
  - **Wastes CPU time** (multiple uncompress of the same data)
  - **Wastes memory** (multiple copies of the same Basket, not shared)
- Small test showed that reading (disk, decompress, streaming, P->T) ESD via athenaMP on 4 cores takes **40% longer** than in a serial job.
- In rel. 17, ATLAS changed ROOT data layout (see https://indico.fnal.gov/conferenceDisplay.py?confId=4862):
  - No splitting (fewer baskets)
  - AutoFlush every 5/10 events (depending on data product)
- Allows athenaMP to schedule small ‘chunks’ of events and avoid double decompression.
  - Test reading ESD on 4 cores showed penalty reduced to less than 20%.

Outlook

- Output of course is just as important
  - Bought some time by moving to fast/hybrid merging
- ROOT team is developing TMemFile which may help
  - But ATLAS needs externalized references to TTree entries.

- Shared Writer will have more difficult scheduling
  - Doesn't know when it is done
- Plan to exercise with ByteStream events first.

- Longer term, ATLAS may want to scatter objects (rather than events) to match retrieval granularity on derived data products

Summary

- Don’t just plan for the next Control Framework, consider the next I/O Framework to support it as well.