# I/O Strategies for Multicore Processing in ATLAS

**P van Gemmeren[1], S Binet[2], P Calafiura[3], W Lavrijsen[3], D Malon[1] and V Tsulaia[3] on behalf of the ATLAS collaboration**

[1]Argonne National Laboratory, Argonne, Illinois 60439, USA

[2]Laboratoire de l'Accélérateur Linéaire/IN2P3, 91898 Orsay Cédex, France

[3]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

E-mail: gemmeren@anl.gov

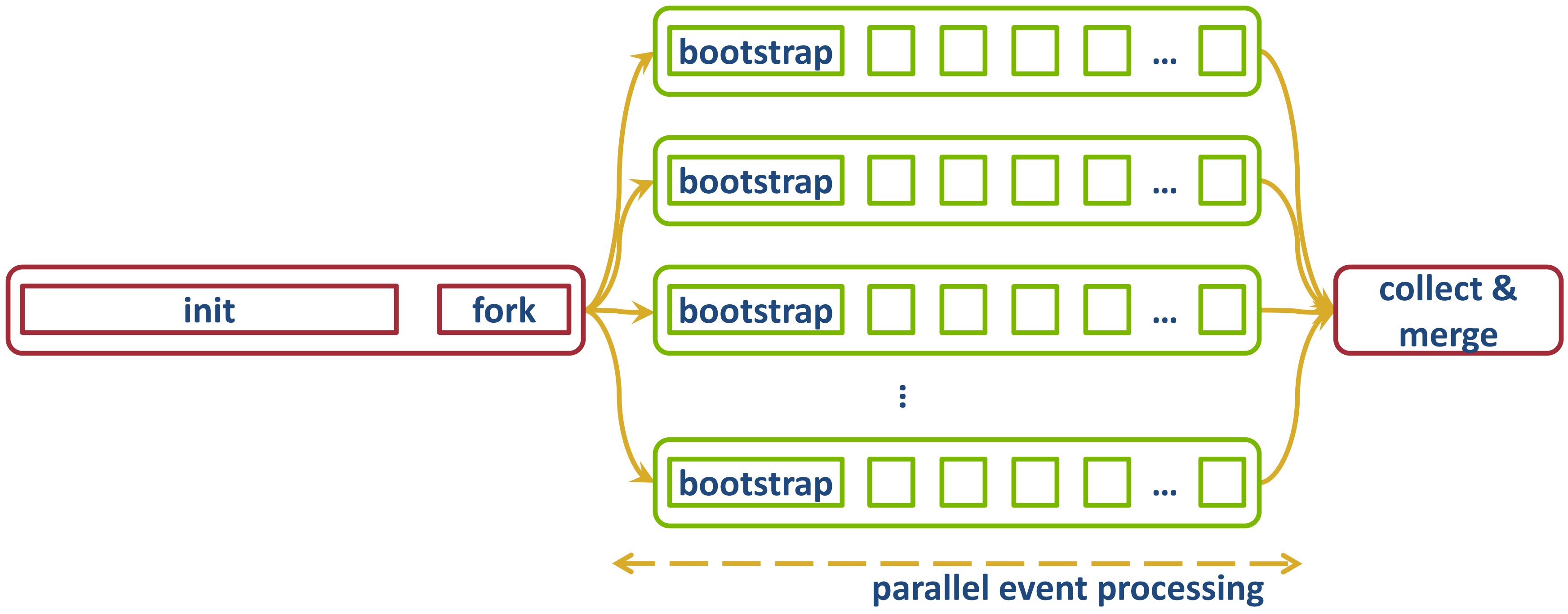
## Abstract

A critical component of any multicore/manycore application architecture is the handling of input and output. Even in the simplest of models, design decisions interact both in obvious and in subtle ways with persistence strategies. When multiple workers handle I/O independently using distinct instances of a serial I/O framework, for example, it may happen that because of the way data from consecutive events are compressed together, there may be serious inefficiencies, with workers redundantly reading the same buffers, or multiple instances thereof. With shared reader strategies, caching and buffer management by the persistence infrastructure and by the control framework may have decisive performance implications for a variety of design choices. Providing the next event may seem straightforward when all event data are contiguously stored in a block, but there may be performance penalties to such strategies when only a subset of a given event's data are needed; conversely, when event data are partitioned by type in persistent storage, providing the next event becomes more complicated, requiring marshalling of data from many I/O buffers. Output strategies pose similarly subtle problems, with complications that may lead to significant serialization and the possibility of serial bottlenecks, either during writing or in post-processing, e.g., during data stream merging. In this paper we describe the I/O components of AthenaMP, the multicore implementation of the ATLAS control framework, and the considerations that have led to the current design, with attention to how these I/O components interact with ATLAS persistent data organization and infrastructure.

## Outline

- Current AthenaMP I/O infrastructure
- Data storage
  - Raw data
  - Simulated, Reconstructed, Derived Data
- Handicaps of current I/O on multicore platforms
  - Read data
  - Write data
- Scatter / Gather architecture for multicore I/O
  - Shared reader strategies and alternatives
    - ByteStream
    - POOL / ROOT
  - Shared writer
- Conclusions / Outlook

## Current AthenaMP I/O infrastructure

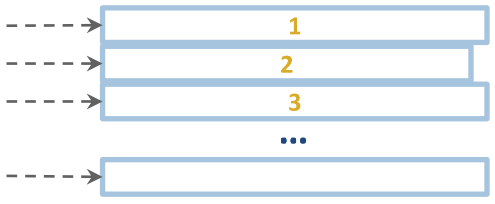


- Multiple workers handle **I/O independently** using **distinct instances of a serial I/O framework**
  - Each worker process produces its own output file, which need to be merged after all workers are done.

# Data storage

## Raw Data: ByteStream

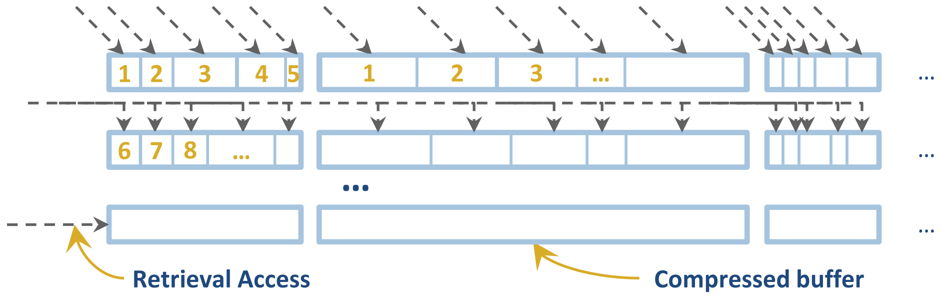
- Simple C-style structure
- Since 2011, ATLAS **compresses Raw data events**
  - Saves about 50% disk storage
  - Event-wise (not file) compression preserves efficient single event reading.



#: Event Number

## Simulated, Reconstructed, Derived Data: (POOL) / ROOT

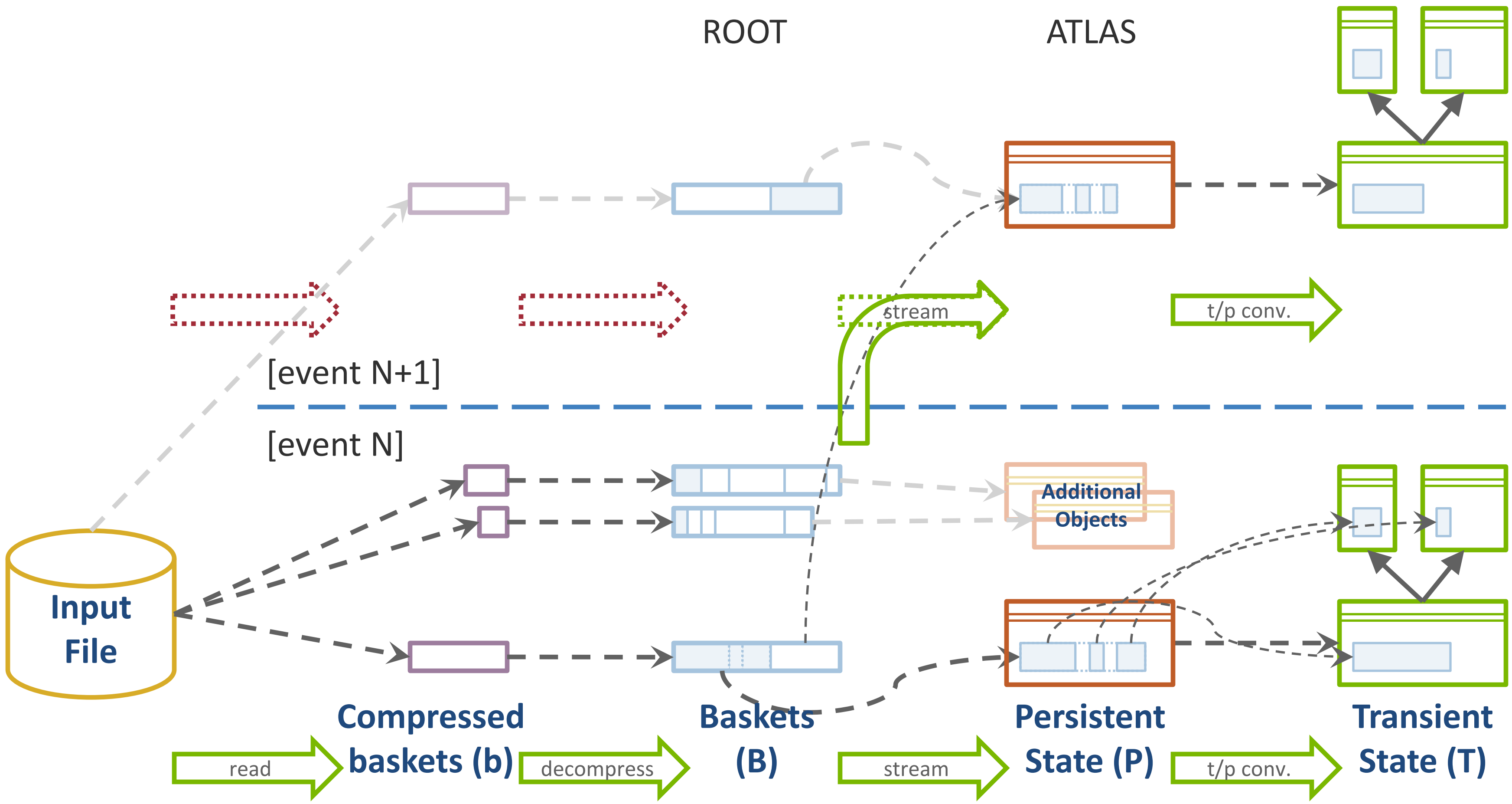
- Object data
  - made simpler and schema evolvable using **Trans.-Pers. conversion**.
- Collections of objects in ROOT TBranches
  - Read separately on demand
- Column-wise **compressed into Baskets**
  - Depending on data product, multiple events share a Basket



# Handicaps of current I/O infrastructure on multicore platforms for ROOT data

- **Read data**: A process (initialization, event execute,…) reads part of the input file (e.g., to retrieve one event, or collection).
  - All workers use the same input file.
    - Multiple accesses may mean **poor read performance**, especially if events are not consecutive.
    - For POOL / ROOT, a worker may retrieve a collection, after the same collection for a later event has already been processed by a different worker. These '**back reads**' hurt I/O performance.
- **Decompress / Stream ROOT baskets**: Each worker will retrieve its own event data, which means reading many ROOT baskets, decompressing them and streaming them into persistent objects.
  - ROOT baskets contain object member of several events, so multiple worker use the same baskets and each of them will decompress them independently:
    - Wastes **CPU time** (multiple decompress of the same data)
    - Wastes **memory** (multiple copies of the same Basket, not shared)

# Multicore reading of ROOT data

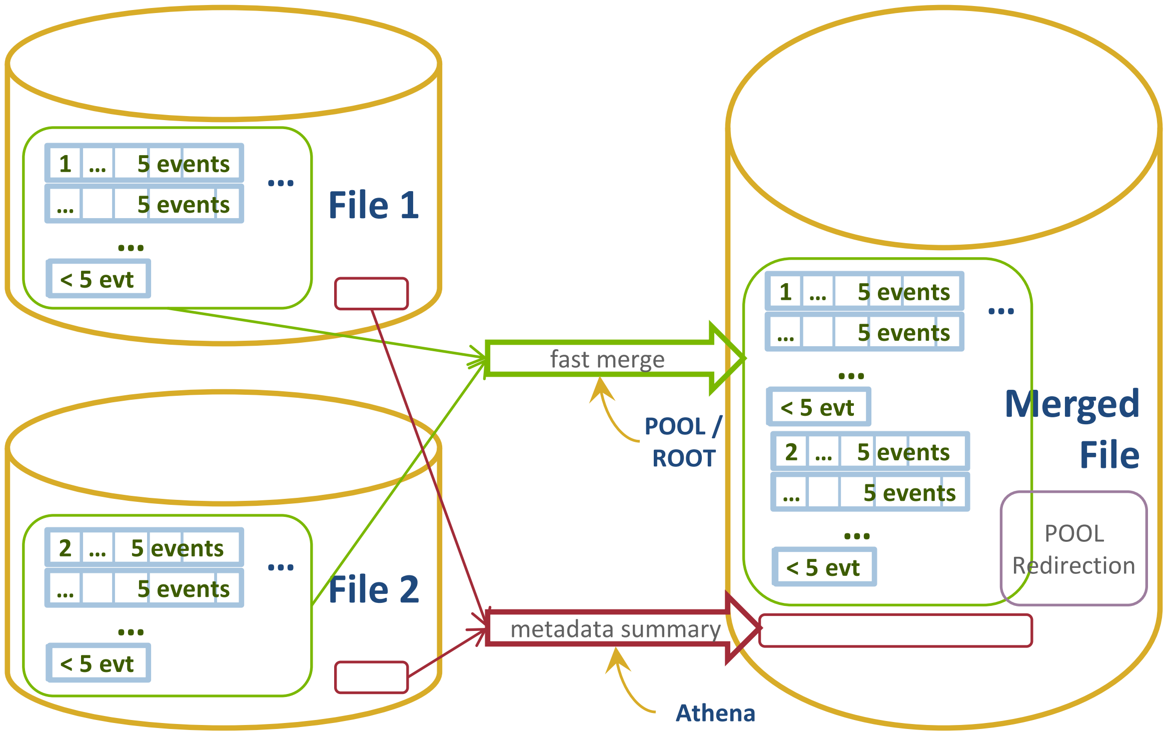


# Handicaps of current I/O infrastructure on multicore platforms for ROOT data

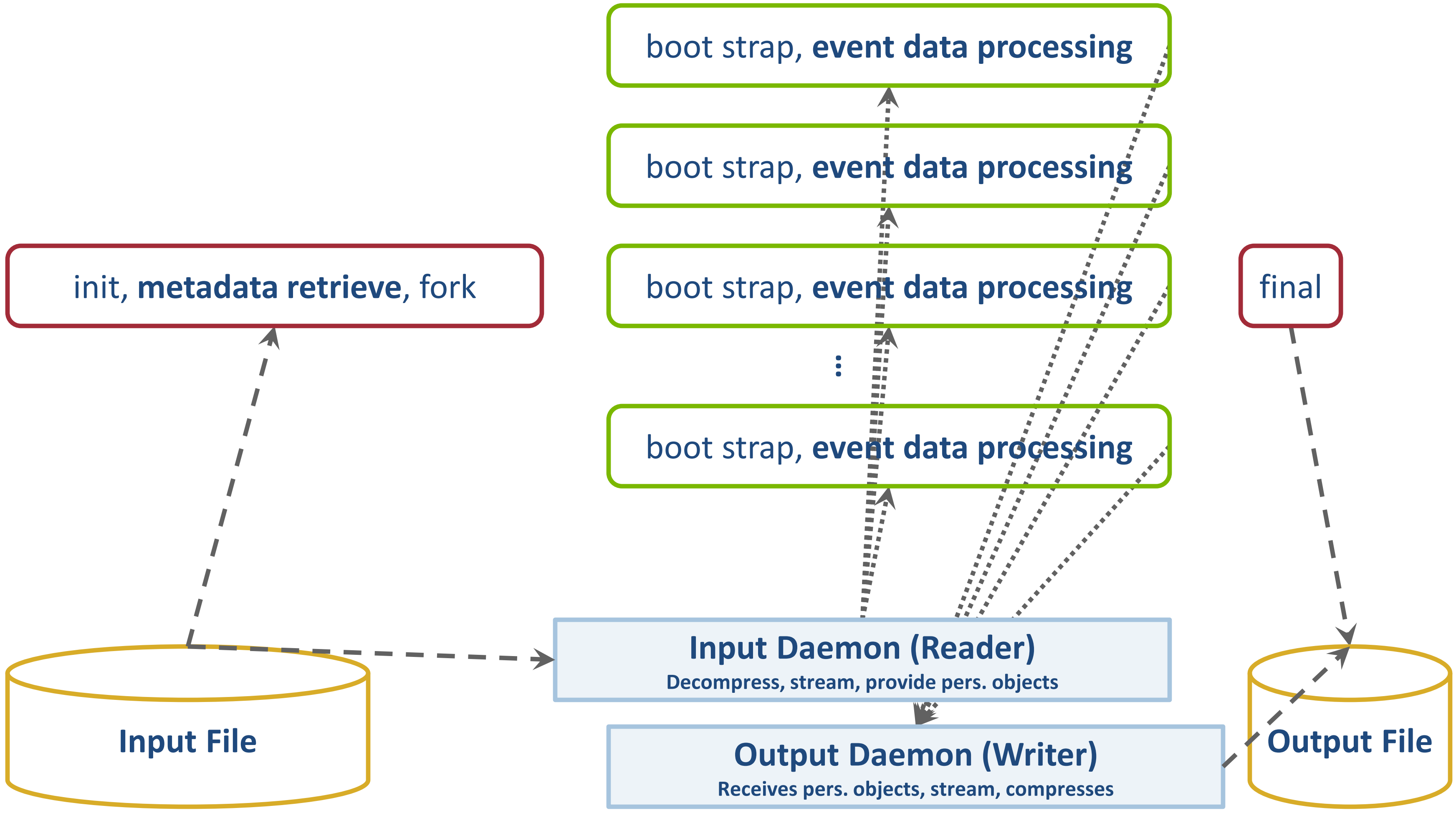
- **Write data**: Each process writes its own output file.
  - Output needs to be merged in **serial**.
- **Compress / Stream to ROOT baskets**: Writers compress data separately.
  - Suboptimal compression factor (which will cost **storage and CPU time** at subsequent reads)
  - Wastes **memory** (each worker needs its own set of output buffer)
- **Merging**: The first implementation of AthenaMP used a separate full Athena job to merge the workers output data.
  - This job would read (decompress, stream, p->t) all data from all files, do nothing , and re-write (t->p, stream, compress) the data into a merged file.
    - Compared to processing this takes about **5% of CPU time**, but because it had to be done in serial, this was the **instant performance bottleneck**.
  - A **fast merge** was developed that appends compressed baskets and maintains a navigational redirection layer (to keep externalized references valid).
    - In-File metadata is still propagated using full Athena framework.
    - Combined event and metadata merging is **5 – 10 times faster**.

# Fast merge utility

- The ATLAS fast merge utility appends event data **without decompressing the baskets** and uses the Athena framework to summarize metadata.

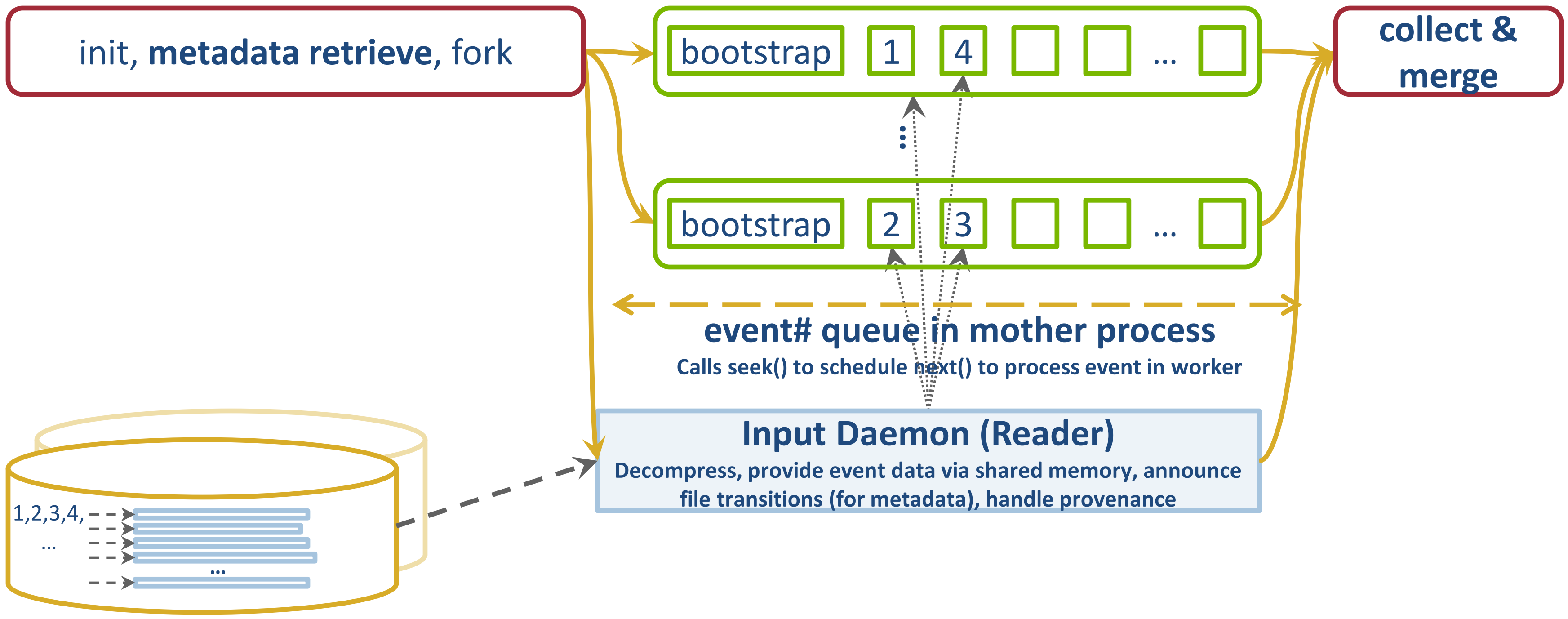


# Scatter / Gather architecture for multicore I/O


boot strap, **event data processing**
boot strap, **event data processing**
init, **metadata retrieve**, fork
boot strap, **event data processing**
final
⋮
boot strap, **event data processing**
Input Daemon (Reader)
Decompress, stream, provide pers. objects
Input File
Output Daemon (Writer)
Receives pers. objects, stream, compresses
Output File

# Shared reader strategies: ByteStream data

- For Raw data, providing the next event is rather straightforward as all event data are contiguously stored in a single block which is read entirely.
  - At the same time, benefit of a single reader for ByteStream is small as Raw events can be decompressed individually.



init, **metadata retrieve**, fork
bootstrap
1
4
…
bootstrap
2
3
…
collect & merge
event# queue in mother process
Calls seek() to schedule next() to process event in worker
Input Daemon (Reader)
Decompress, provide event data via shared memory, announce file transitions (for metadata), handle provenance
1,2,3,4,
…
…

## Challenges for shared ByteStream reader

- Metadata propagation
  - As for multicore processing in general, metadata poses one of the key challenges for the shared ByteStream reader.
  - In this architecture the reader is the only component that detects file transitions.
    - It will need to inform worker to fire incidents, which control metadata propagation.
    - Each worker will then fire the End File Incident after processing its last event from the file and the Begin File Incident before processing the next event.
  - Input metadata needs to be provided to all worker processes.
    - For Raw data, ATLAS uses in-file metadata which is very limited in content and structure.
- Provenance
  - When reading Raw data, the ByteStream reader creates an artificial provenance record that needs to be communicated to the event worker.

# Shared reader strategies: POOL / ROOT data

- For simulated, reconstructed or derived data, there may be performance penalties to event scatter strategies (such as shown for ByteStream).
  - POOL / ROOT data is accessed for each collection on demand and not all are needed for each job.
  - Notation of a persistent event may not be natural on a flat event store?
- However, scattering individual collections of data objects for each event becomes more complicated, requiring marshalling of data from many I/O buffers and increased communication between the reader process and the workers.
  - ATLAS stores several hundreds of collections per event.
    - Can use existing Athena I/O and StoreGate object retrieval mechanism to send a request to the reader.
    - Advanced strategies, such as learning what container are needed, could be used to improve performance.
  - Persistent objects could be passed through shared memory.

## Alternative reader strategies for POOL / ROOT data

- Since 2011, ATLAS uses the ROOT auto-flush feature for their main event data TTree to write clusters of 5 / 10 events (depending on data product).
  - To avoid baskets getting too small for efficient I/O, splitting was switched off. So all the data of a collection is streamed member-wise into a single TBranch.
- Compressed baskets contain data from only 5 / 10 subsequent events
- If the entire cluster of events is processed by the same worker, most of the performance disadvantages from slide 6 are solved.
  - Some of the inefficiencies will remain due to reading of auxiliary TTrees, that are not in sync with event numbers (less than 5% of data).
- Expected to speed up reading by 20 – 50% in CPU time.
  - Better utilization of memory: no duplicated buffer on different workers.
- Not that simple: Fast merged files will destroy event cluster organization.
  - Last cluster for each file can have fewer than 5 / 10 events.

## Writing

- Output strategies pose similarly subtle problems, with complications that may lead to significant serialization and the possibility of serial bottlenecks, either during writing or in post-processing, e.g., during data stream merging.

## TMemFile

- ROOT also prepares for multi-core I/O.
  - E.G.: TMemFile, ROOT way of combining/merging several TTrees to the same output TTree.
- With the migration away from POOL and potentially closer to ROOT, ATLAS may decide to leverage these features directly.
- However, there still are open questions:
  - Biggest obstacle for ATLAS: Inserting TTree entry in TMemFile will not return valid entry number, so ATLAS cannot easily create an external Token.
    - Tokens are the basis for the navigational infrastructure
  - Not clear whether type specific function calls could be sufficient to merge metadata objects
  - And what about multiple tree synchronization?

## Summary and Outlook

- A multi-process **control framework** (like AthenaMP)) to enable HEP event processing on multicore computing architectures is an important step, but it is only one of many steps that need to be accomplished.
- Optimizing **event data storage**, so that it can be efficiently retrieved in the granularity needed by the multiple worker processes is key to avoiding performance penalties during reading.
  - The 2011 change to member-wise streaming with a small number of entries per basket will help ATLAS to tackle inefficiencies in multi-process reading of ROOT data.
  - The data layout also must ensure that data produced by multiple processes can be:
    - efficiently combined, as merging is typically done serially,
    - the resulting output is as efficient to read and store
- ATLAS is in the process of developing an **I/O architecture and components** that can efficiently support even higher numbers of parallel worker processes.
- First prototypes are being tested, but much work remains.