Managing Asynchronous Data in ATLAS's Concurrent Framework

Charles Leggett

Gaudi Workshop 2016

Conditions Handling in ATLAS

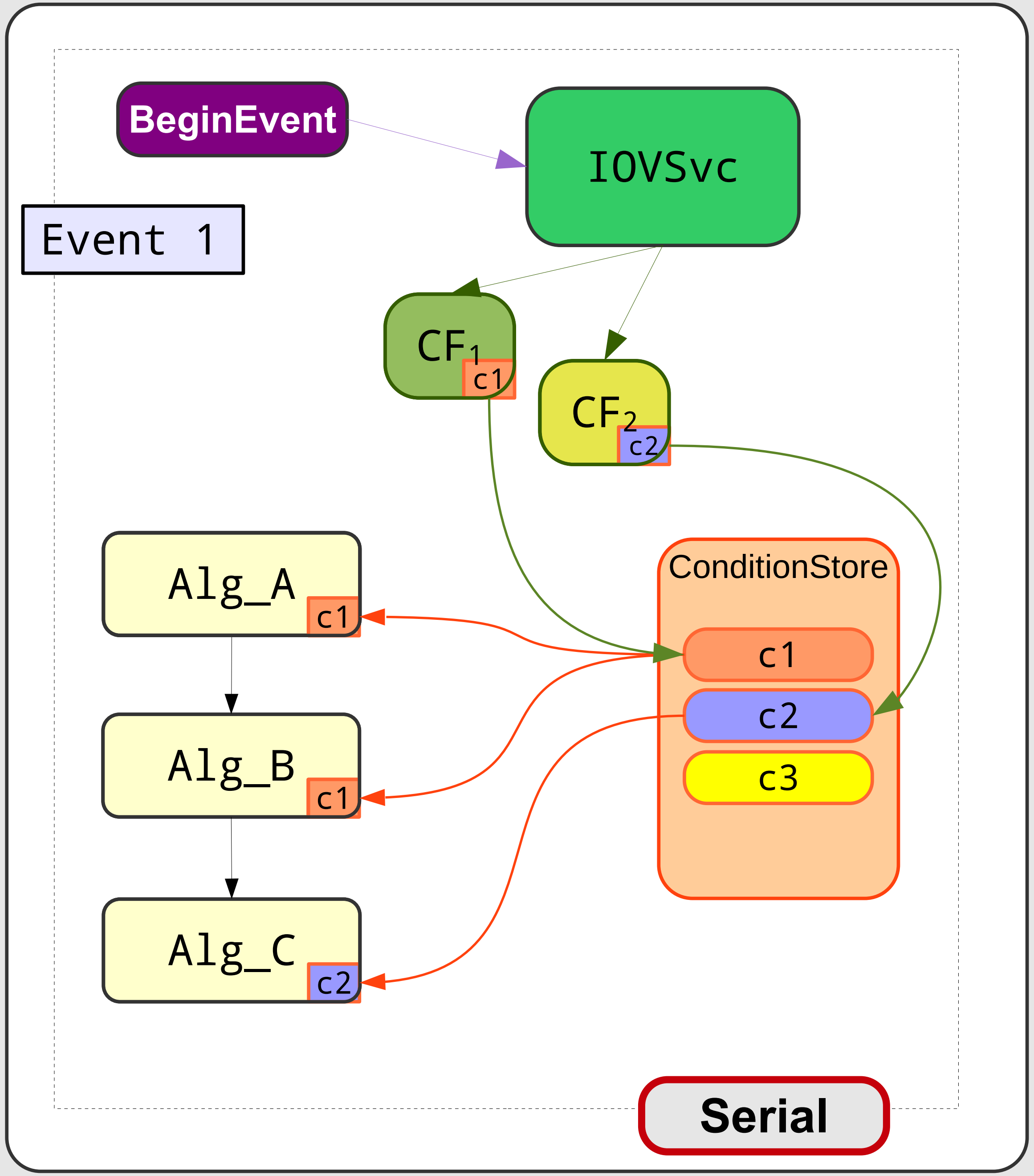
- **Conditions**: data which changes over the course of a job, at a frequency which is not event based
  - can be simple constants read from a dB ("raw")
  - can be derived, which requires some form of processing ("calibrated")

  - in ATLAS, managed by the IOVSvc and IOVDbSvc
    - "derived" conditions produced by callback functions which are registered with the IOVSvc
      - usually AlgTools, but can be anything with the right signature
    - dependencies can be registered against each other in a hierarchical graph

- Detector Alignments follows the same patterns
  - we should not try to solve the problem twice!

- Depending on the type of job, the frequency of updates can be anywhere from once per job, to once per event.
  - can be hundreds of objects updated regularly in a job

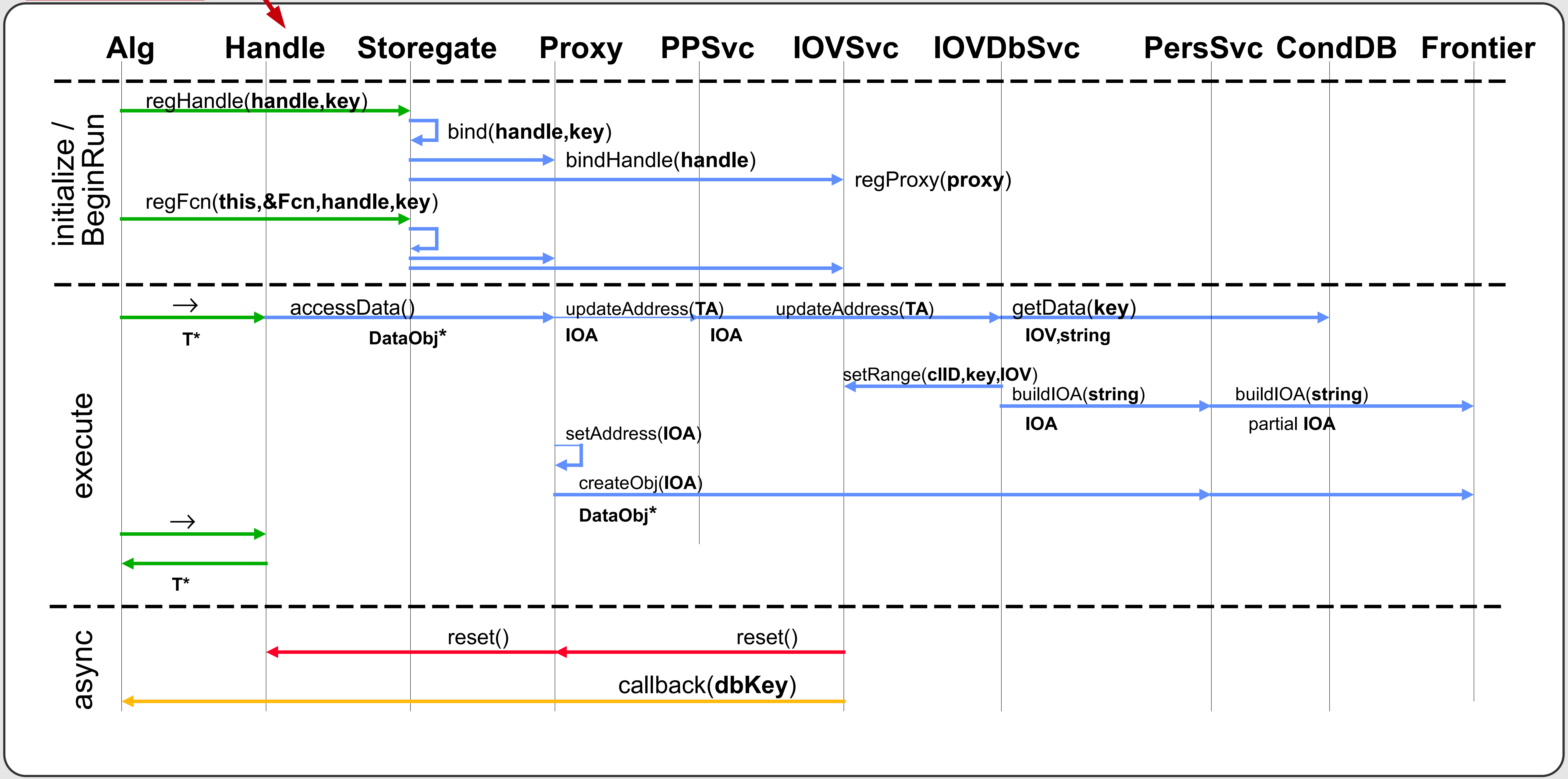
Serial Processing with Conditions



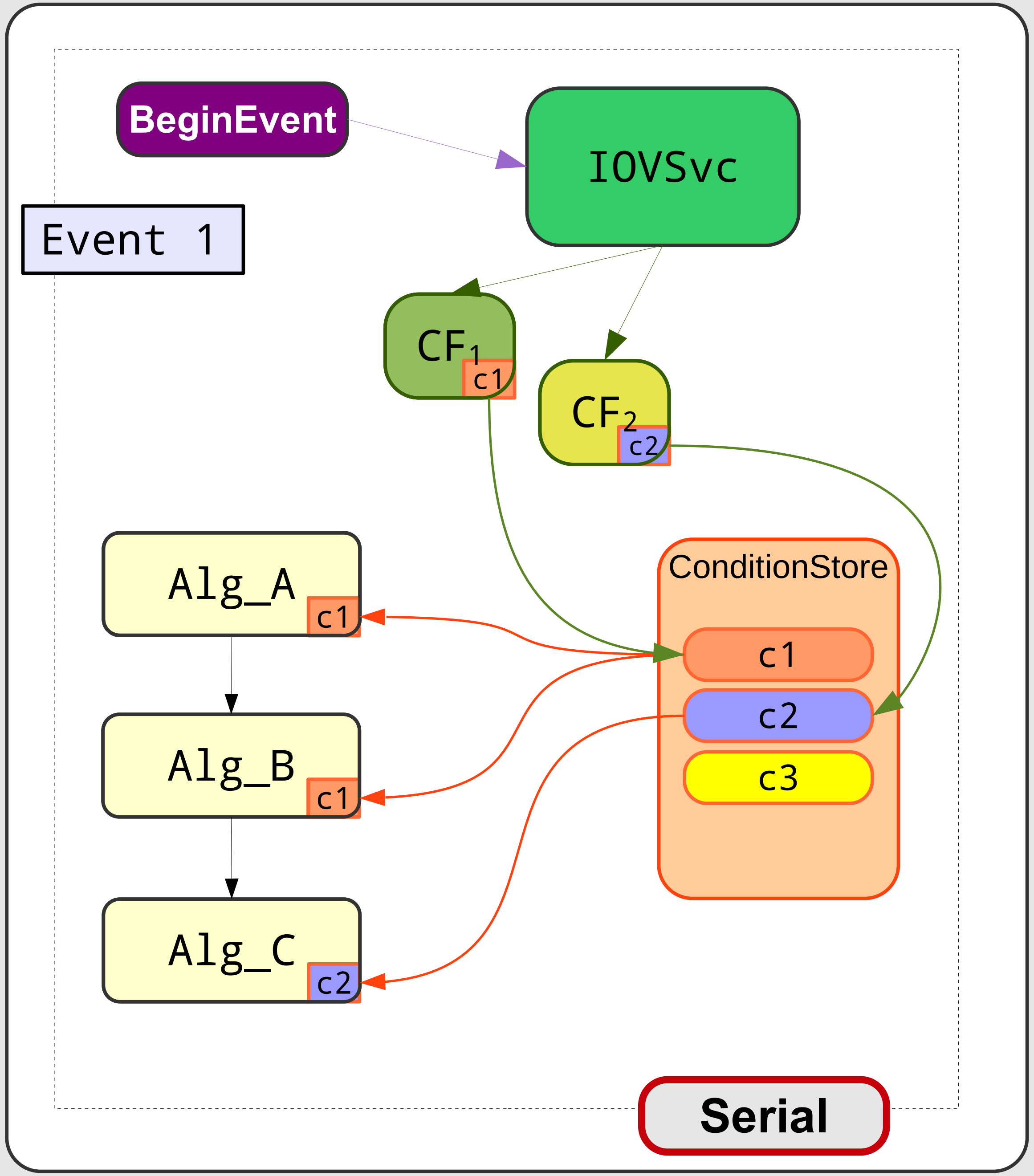
- All framework elements process data from the same IOV
- Algorithms are blind to the IOV, retrieve data from **ConditionStore**
- At the start of every Event, **IOVSvc** checks IOVs, and triggers any necessary updates
  - handled by the Callback Functions
  - Callback Functions are **shared** instances
- Only one copy of any Conditions object is maintained in the Store

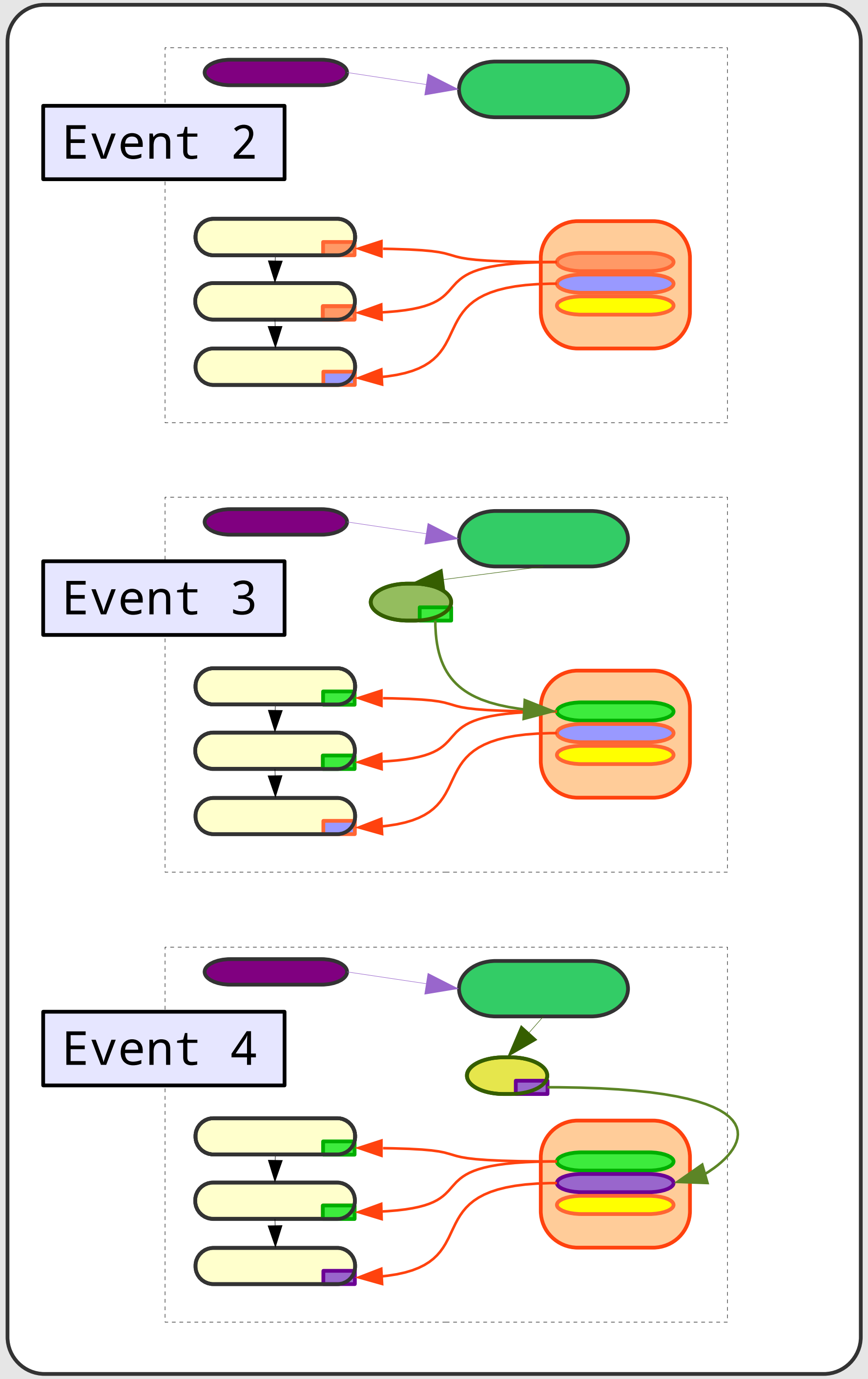
Serial Access Pattern

a different
DataHandle

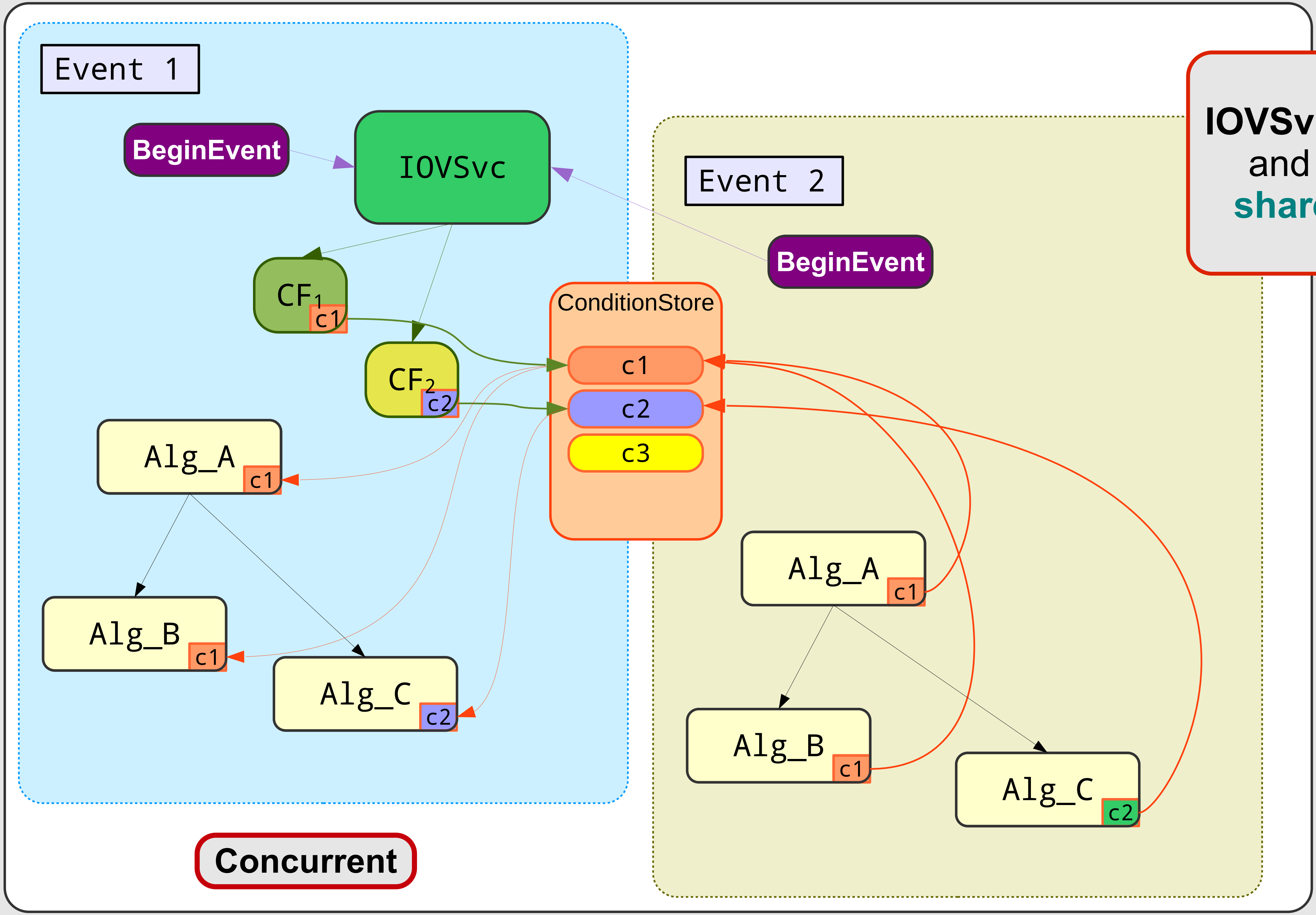


Serial Processing with Conditions



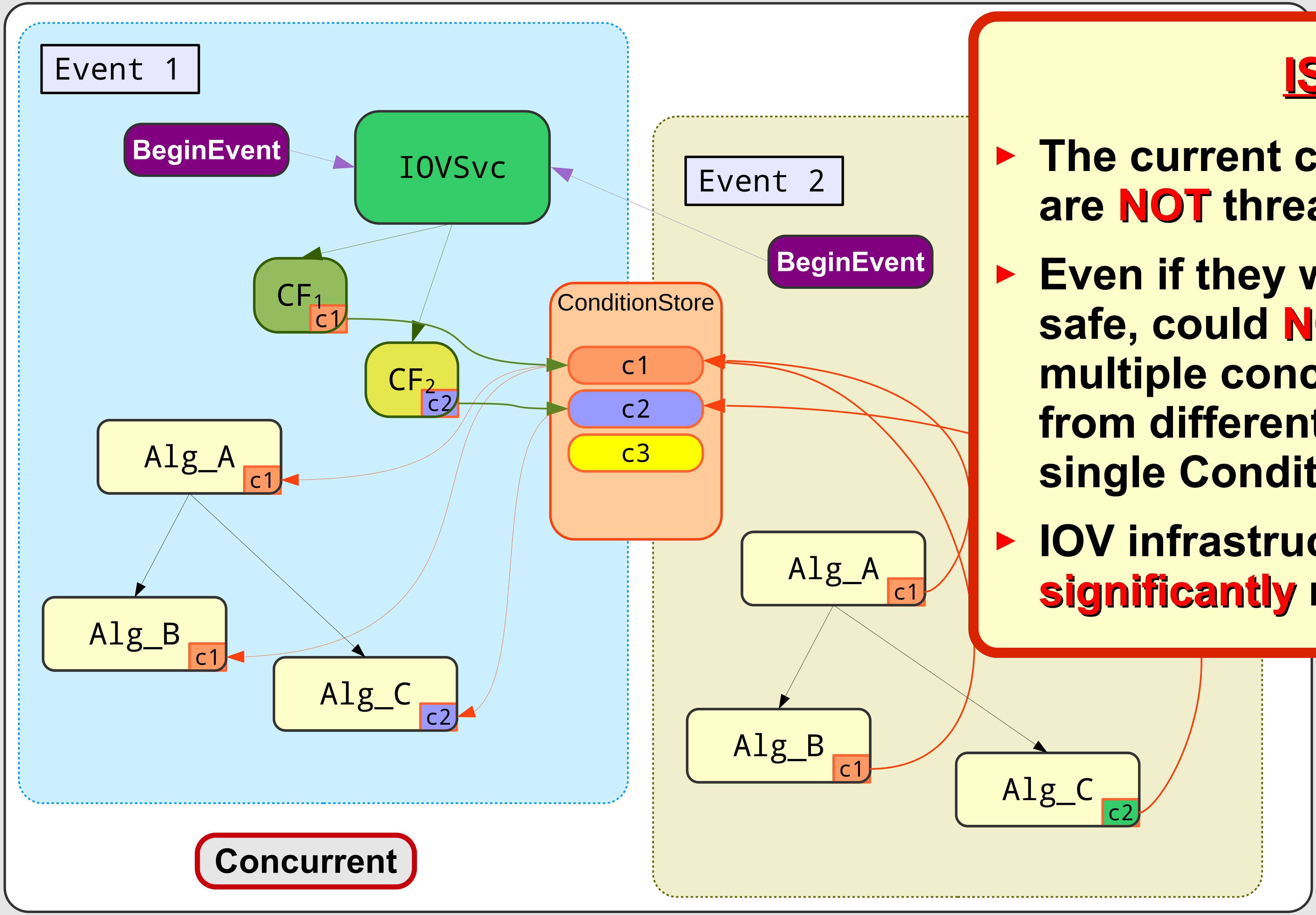


Concurrent Processing with Conditions



IOVSvc, **Callback Functions** and **ConditionStore** are **shared** between all Events

Concurrent Processing with Conditions



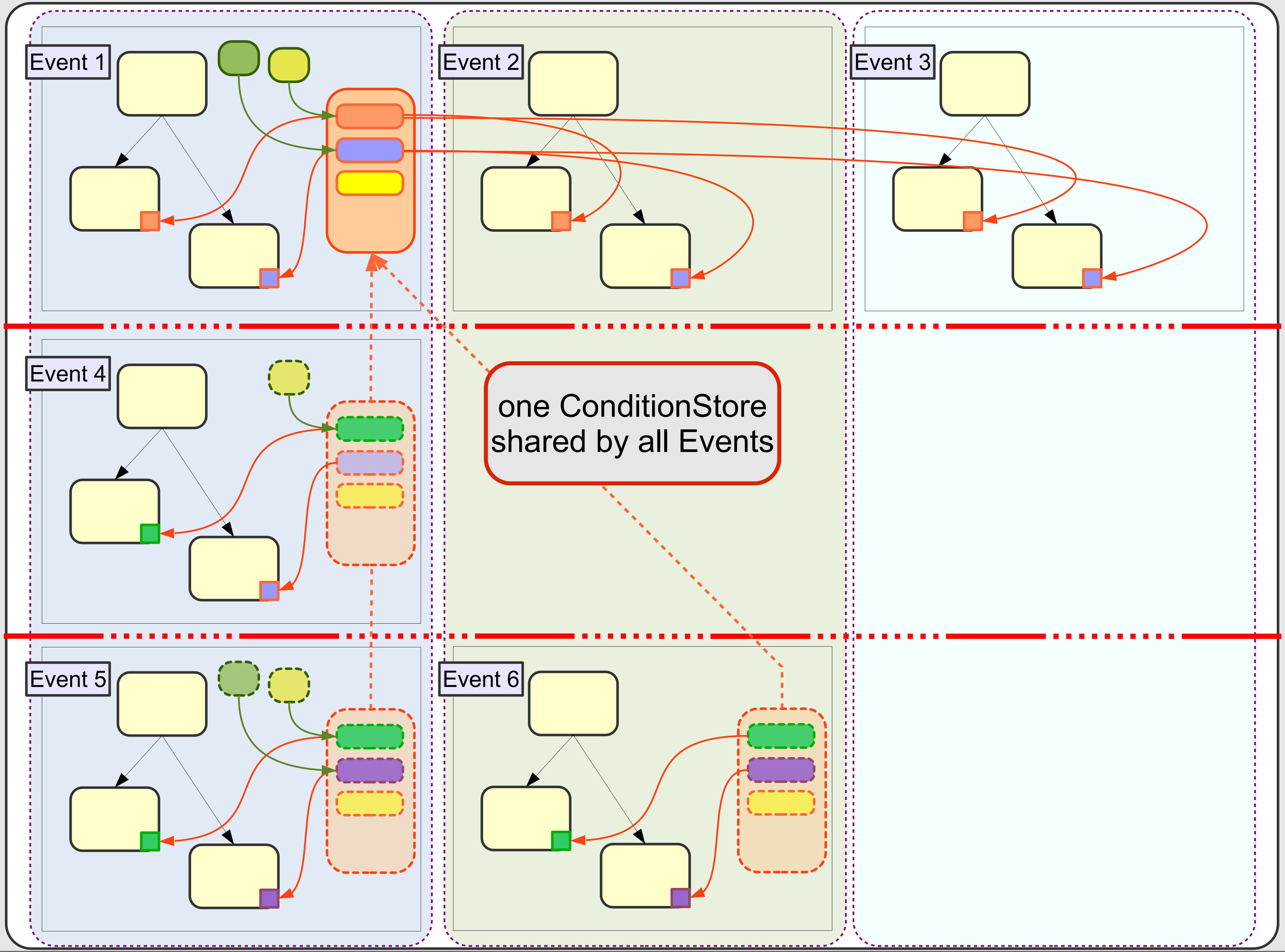
ISSUES

- The current callback functions are **NOT** thread-safe
- Even if they were made thread-safe, could **NOT** run with multiple concurrent Events from different IOVs due to the single ConditionStore
- IOV infrastructure needs to be **significantly** modified for MT

Relevant Requirements for AthenaMT

- **Requirement**: Try to minimize changes to User code
  - there's lots and lots of it!
  - avoid forcing Users to implement fully thread-safe code by handling most thread-safety issues at the framework / Services level
- **Requirement**: All access to Event data *via* **DataHandles**, which also declare data dependency relationship to the framework
  - we can use this by forcing migration to **ConditionHandles** as well

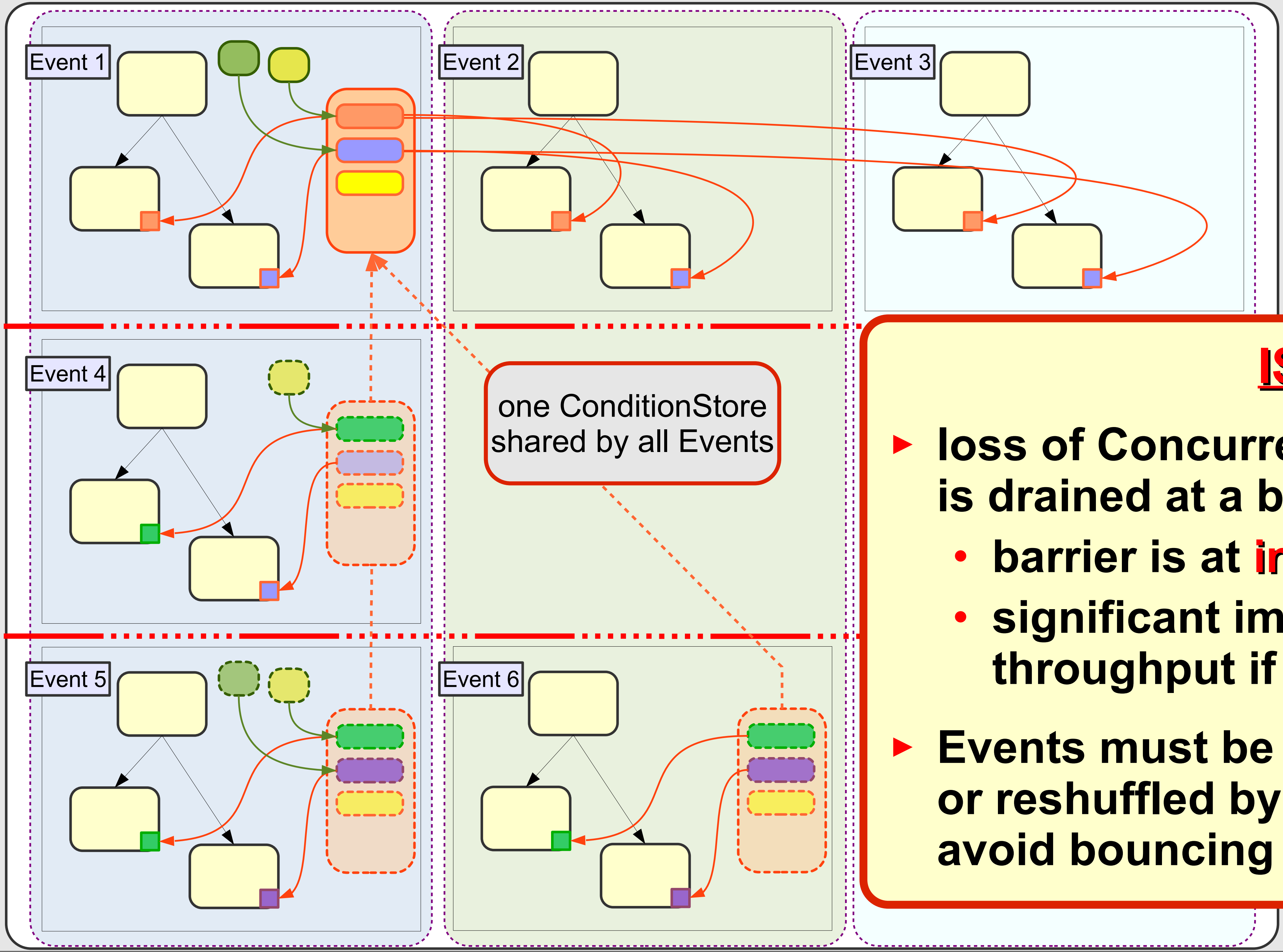
Concurrent: Scheduling Barrier



Scheduler can only **concurrently** process events which have **all** Conditions in the **same** IOV

NO changes required in User code and minimal changes in IOV code

Concurrent: Scheduling Barrier

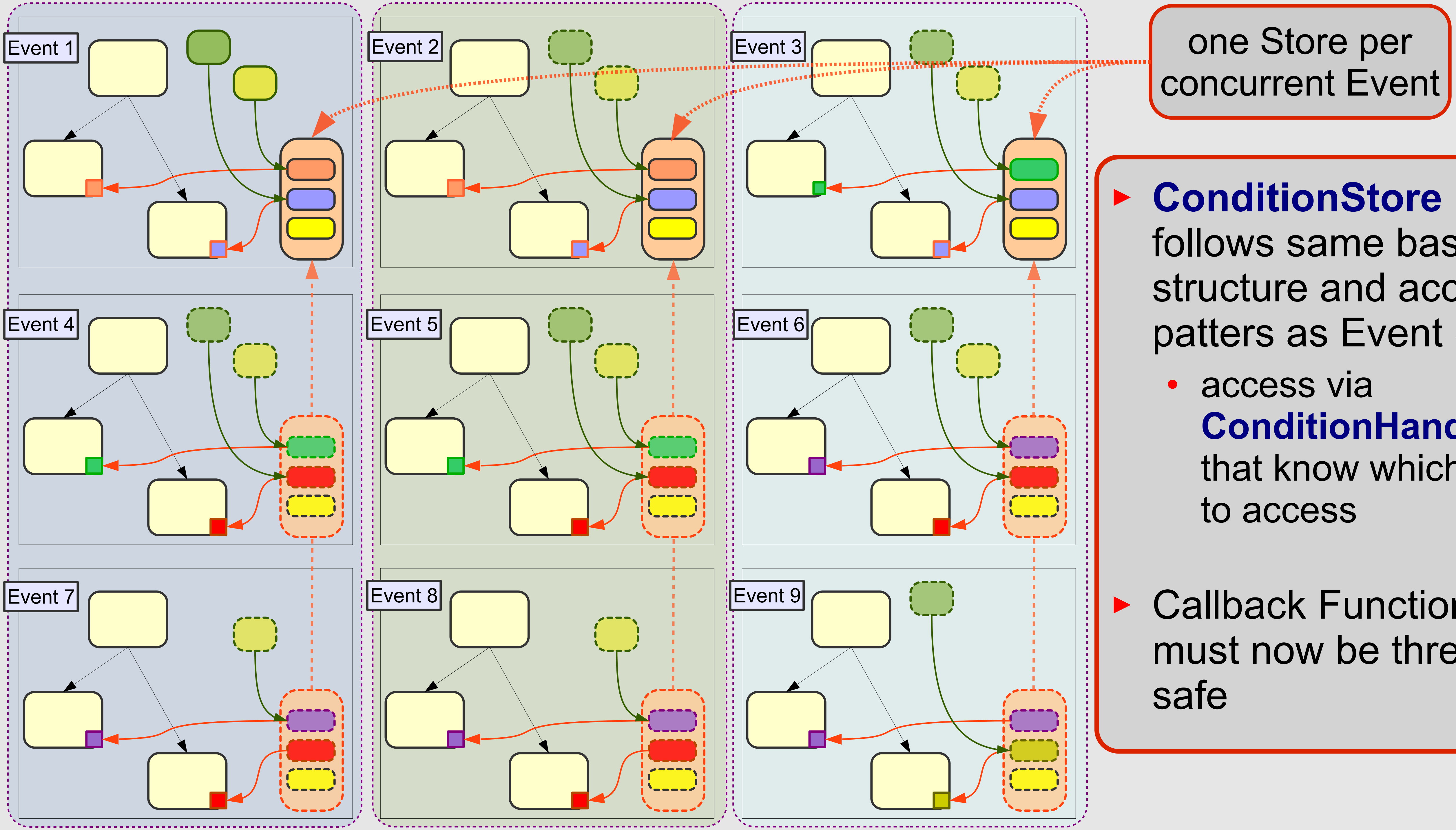


Scheduler can only **concurrently** process events which have **all** Conditions in the **same** IOV

ISSUES

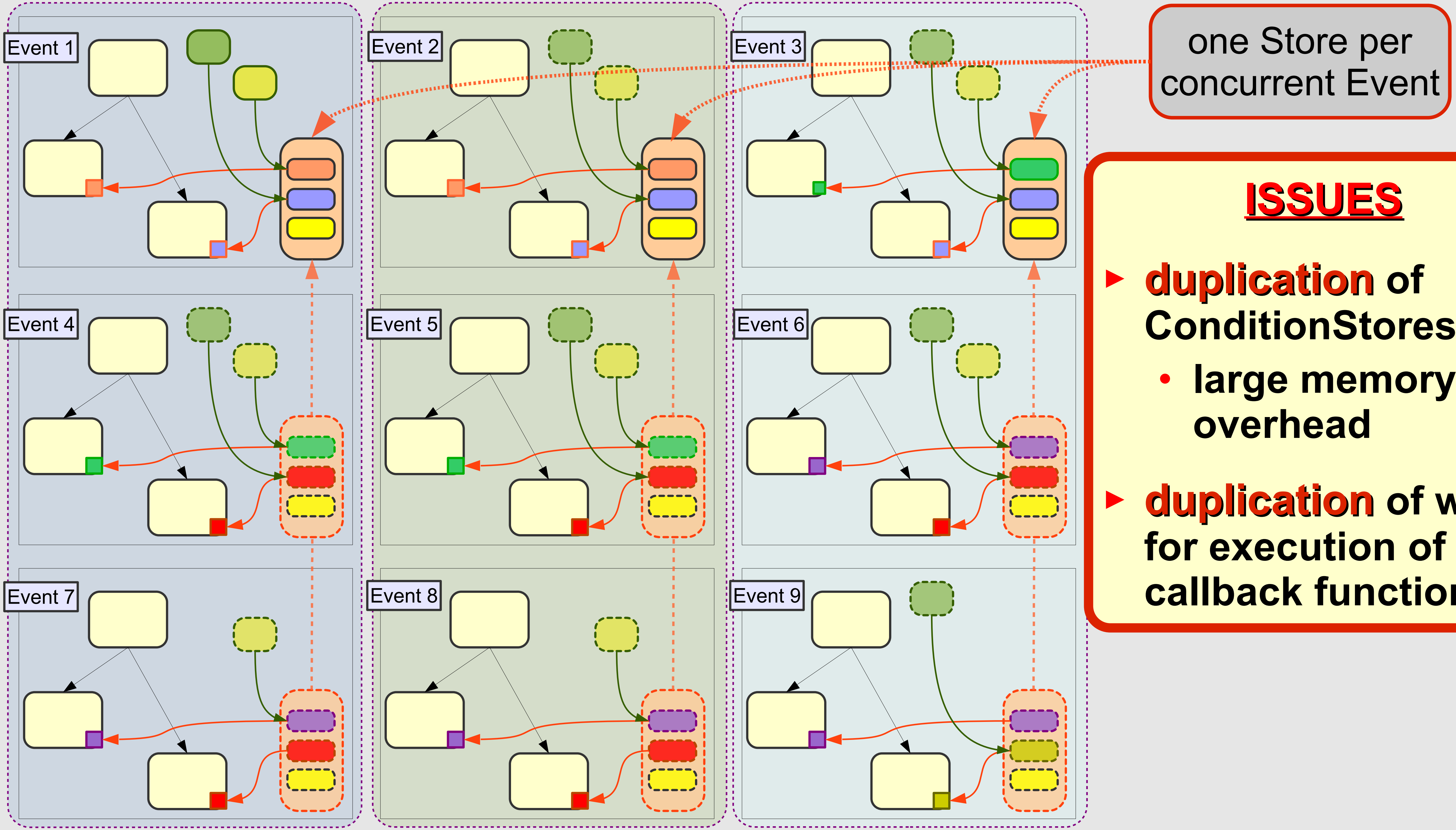
- **loss of Concurrency when Scheduler is drained at a barrier**
  - **barrier is at intersection of all IOVs**
  - **significant impact on Event throughput if IOVs change often**
- **Events must be processed in order, or reshuffled by the Scheduler to avoid bouncing back and forth**

Concurrent: Multiple Condition Stores



- **ConditionStore** follows same basic structure and access patters as Event Store
  - access via **ConditionHandles** that know which store to access
- Callback Functions must now be thread safe

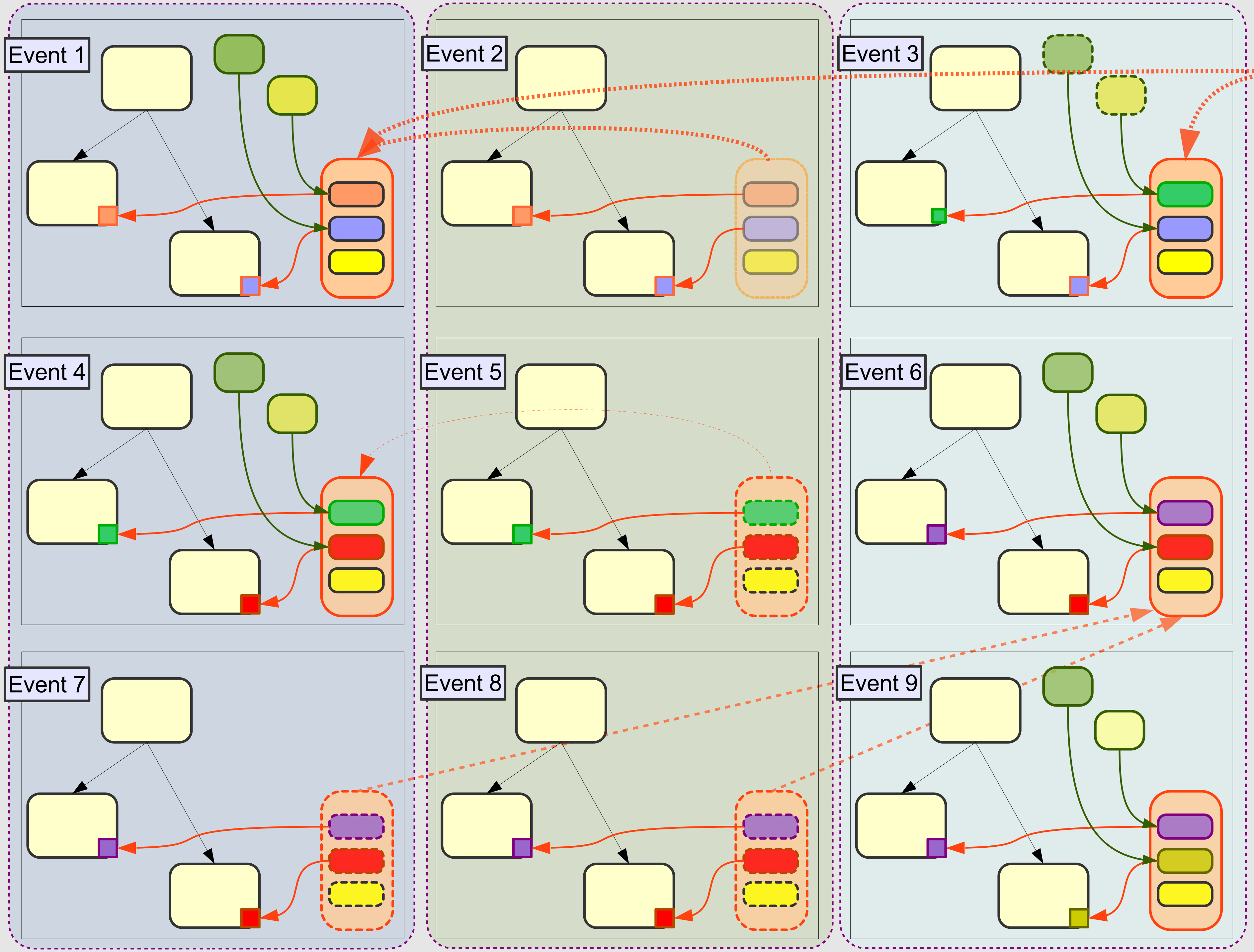
Concurrent: Multiple Condition Stores



ISSUES

- **duplication of ConditionStores**
  - **large memory overhead**
- **duplication of work for execution of callback functions**

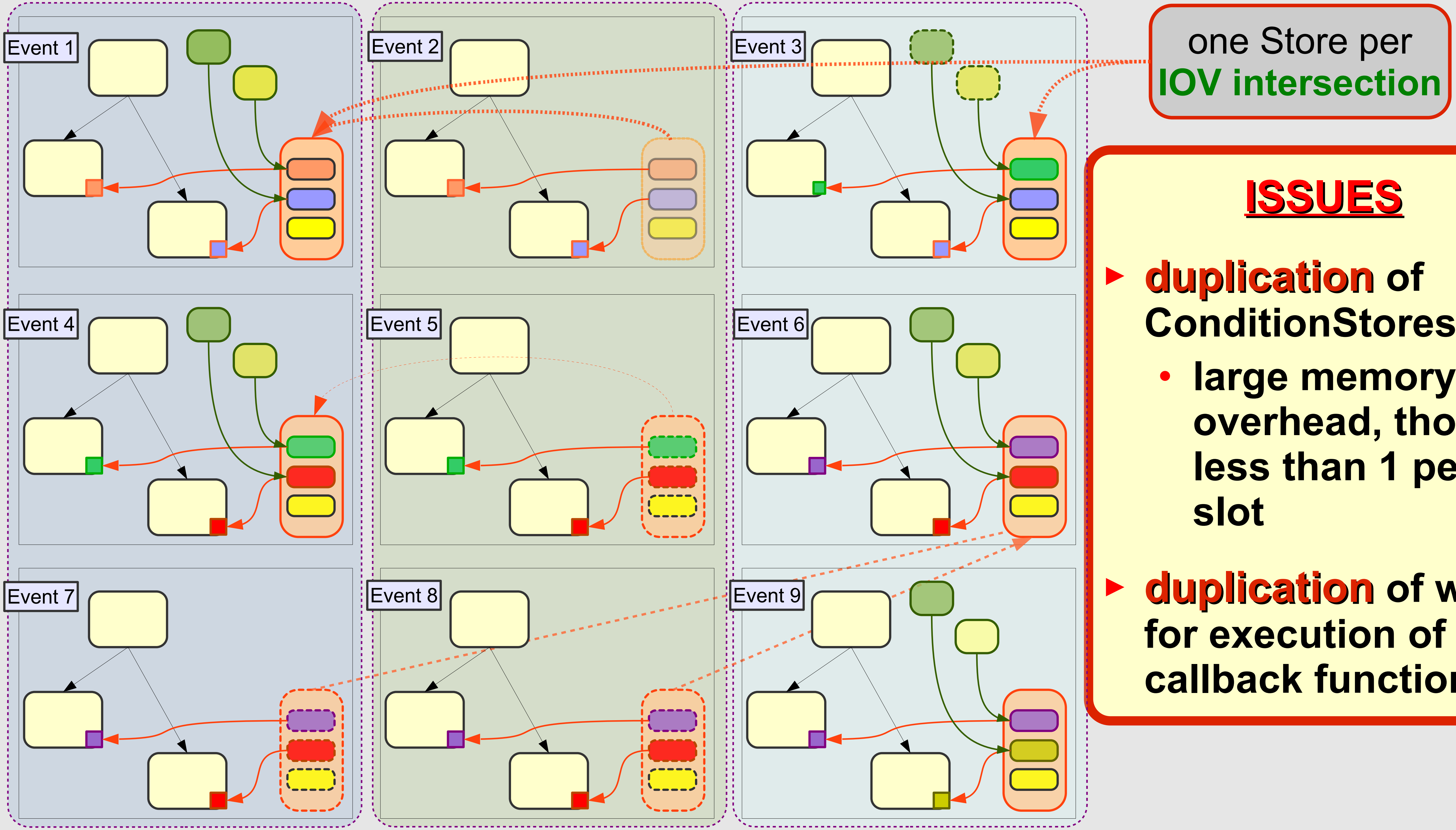
Multiple Condition Stores : IOV Intersection



one Store per
IOV intersection

- take **AND** of all IOVs
  - ConditionSlot
  - all condition objs static for all events in this range
  - can have 1 to N active slots, where N == number of concurrent events
- CondHandles know which Condition Slot to point to

Multiple Condition Stores : IOV Intersection



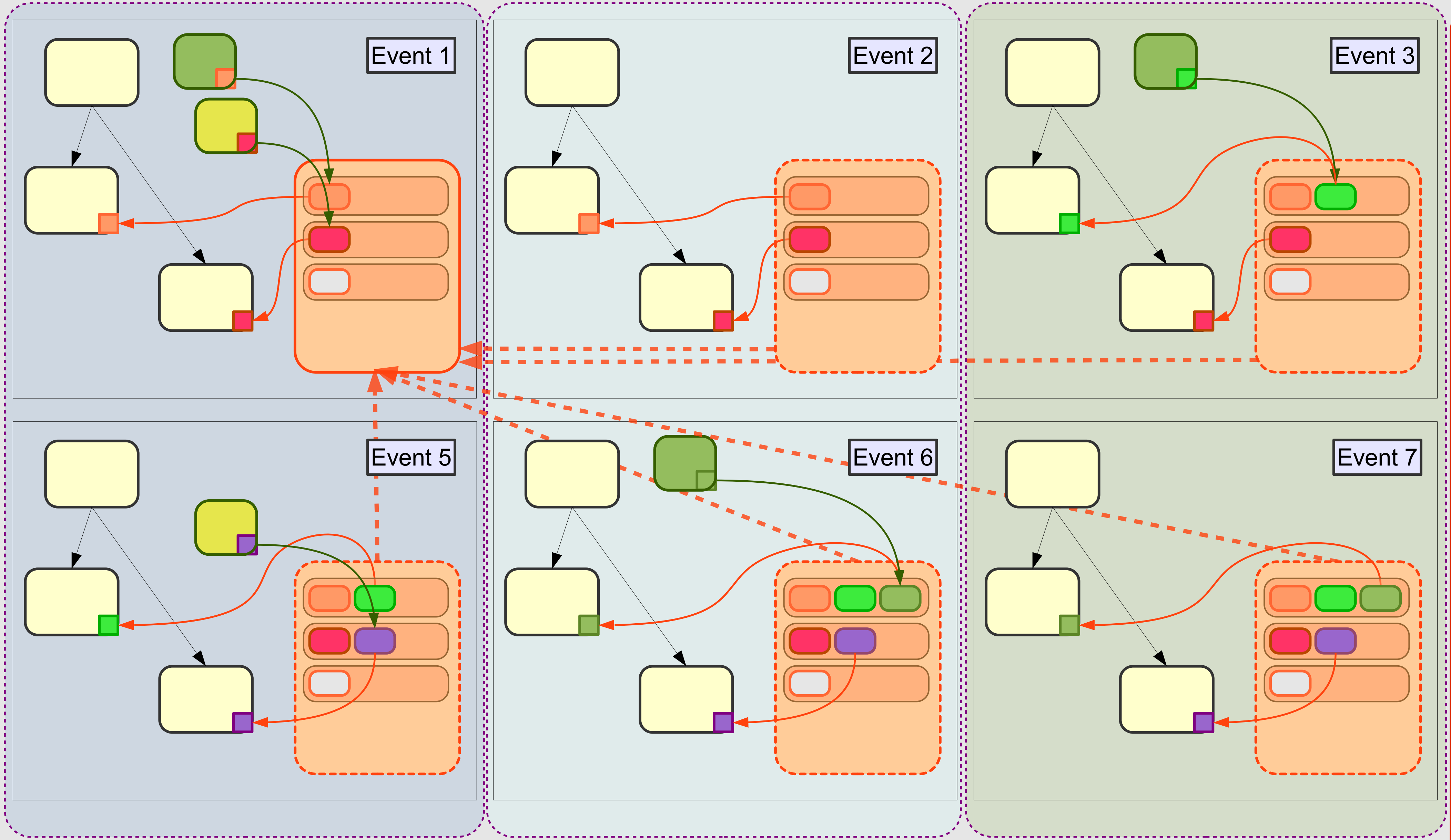
ISSUES

- **duplication of ConditionStores**
  - **large memory overhead, though less than 1 per evt slot**
- **duplication of work for execution of callback functions**

Multi-Cache Condition Store

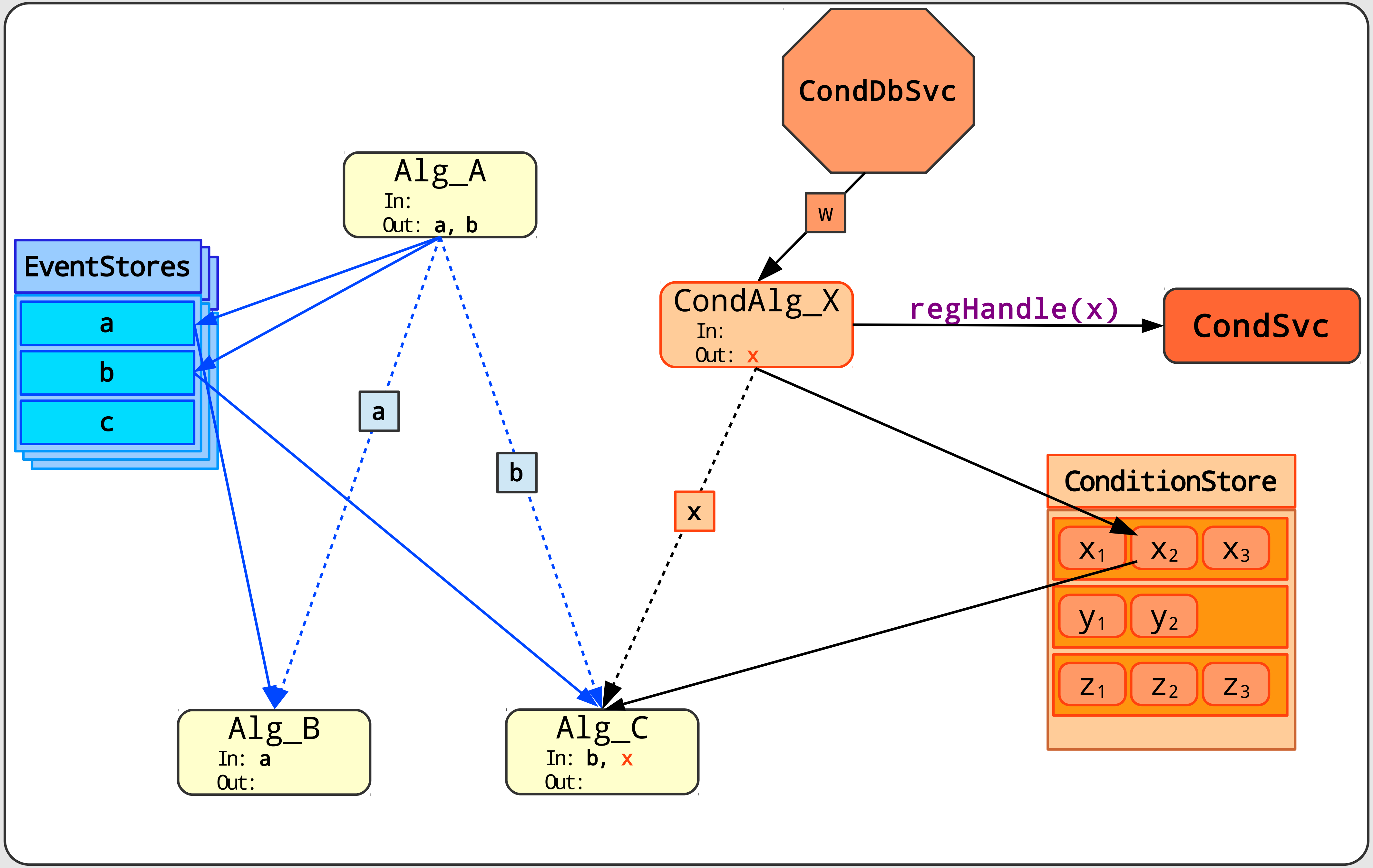
- **Single multi-cache Store for Conditions data**
- Each Store element is a **container** that holds multiple instances of the Condition data objects (**ConditionContainer**), one per IOV
- Clients access the data via smart **ConditionHandles**, that point to the appropriate entry in the ConditionContainer objects for a given Event
  - ConditionHandles are constructed with an **EventContext** object
  - from the Client's point of view, these objects look like any other object in the EventStore (keyed with a unique identifier)
    - Client Algorithms declare a data dependency on the conditions data object
- Updating functions are scheduled by the framework, that load new elements as needed from the DB, and perform any necessary computations
  - IOVSvc callback functions are migrated into **ConditionAlgorithms**
  - these Algorithms are only scheduled when they enter a new IOV

Concurrent: Single Multi-Cache Condition Store



- **One** ConditionStore, shared by all Events.
  - no wasted memory
  - no duplicate calls
  - Store elements are **ConditionContainers**, with one entry per IOV
- Data access via **ConditionHandles** that point to appropriate entry
- Callback Functions become **Algorithms**, scheduled by framework

ConditionHandles



Functionality

- During initialize, CondAlgs register their WriteCondHandles with the CondSvc

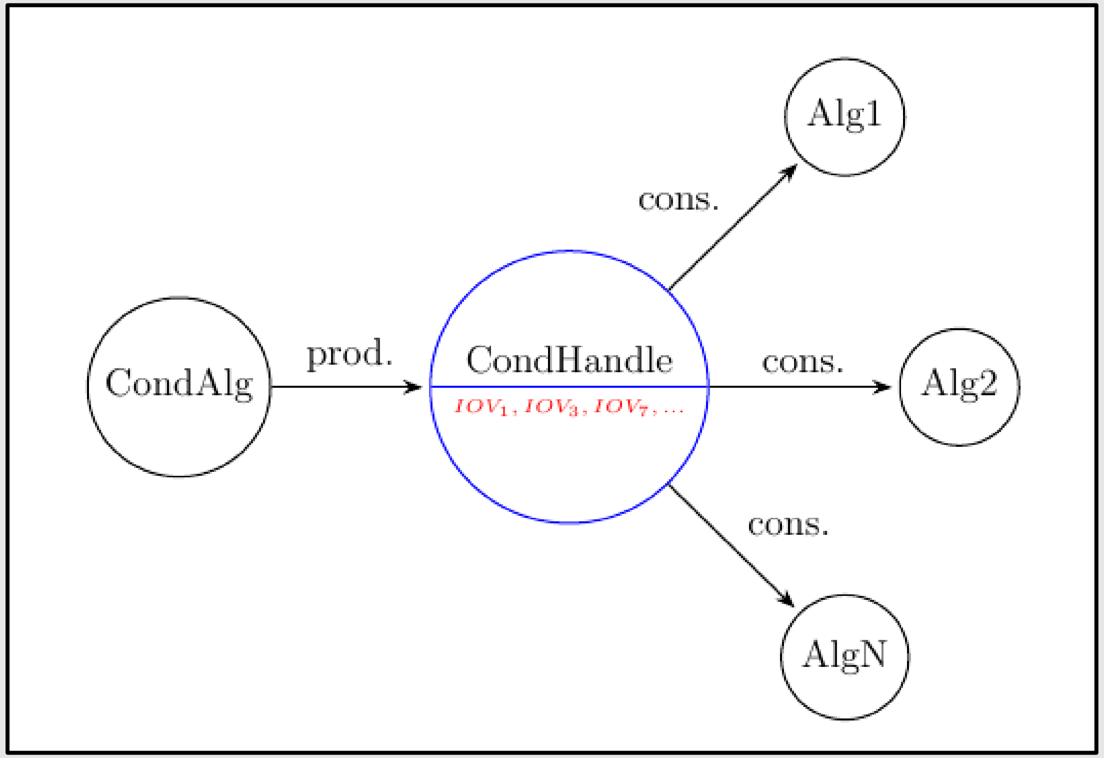
```
StatusCode ICondSvc::regHandle(IAlgorithm* alg,
                               const Gaudi::DataHandle& id,
                               const std::string& dBkey);
```

- At the start of each event, the ForwardScheduler will:
  - query CondSvc to determine which CondObjIDs are valid/invalid
  - query ExecutionFlowGraph to find producer CondAlg of these objects
    - we could build this locally once since it's fixed, but the EFG is pretty efficient
  - if any objects produced by a CondAlg is invalid, schedule the Alg to execute, otherwise mark it as already executed
  - update data catalog with all valid CondObjIDs
- Only CondAlgs that produce new data (ie, the CondObj has entered a new validity range) will execute

- Can make this scheme even simpler if we integrate condition object validities into the Scheduler

CondHandle Node in Scheduler

- Augment the regular data flow rules with a new type of node.
- CondHandle node will be a bit smarter than a regular data object node.
  - carry the list of IOVs, objects for which are currently available in the ConditionStore.
  - when a node visitor, which is event-time-aware, enters a CondHandle node on request from, e.g., Alg1, it can figure out whether to declare this CondHandle as valid for Alg1, or trigger the parental CondAlg to load the missing object/IOV to ConditionStore.
  - When it is loaded, another visitor is launched by CondAlg in the backwards direction to notify all related Alg consumers of this change.
- In this approach, the graph of precedence rules is a singleton across all events.



Conditions Data With No Callback

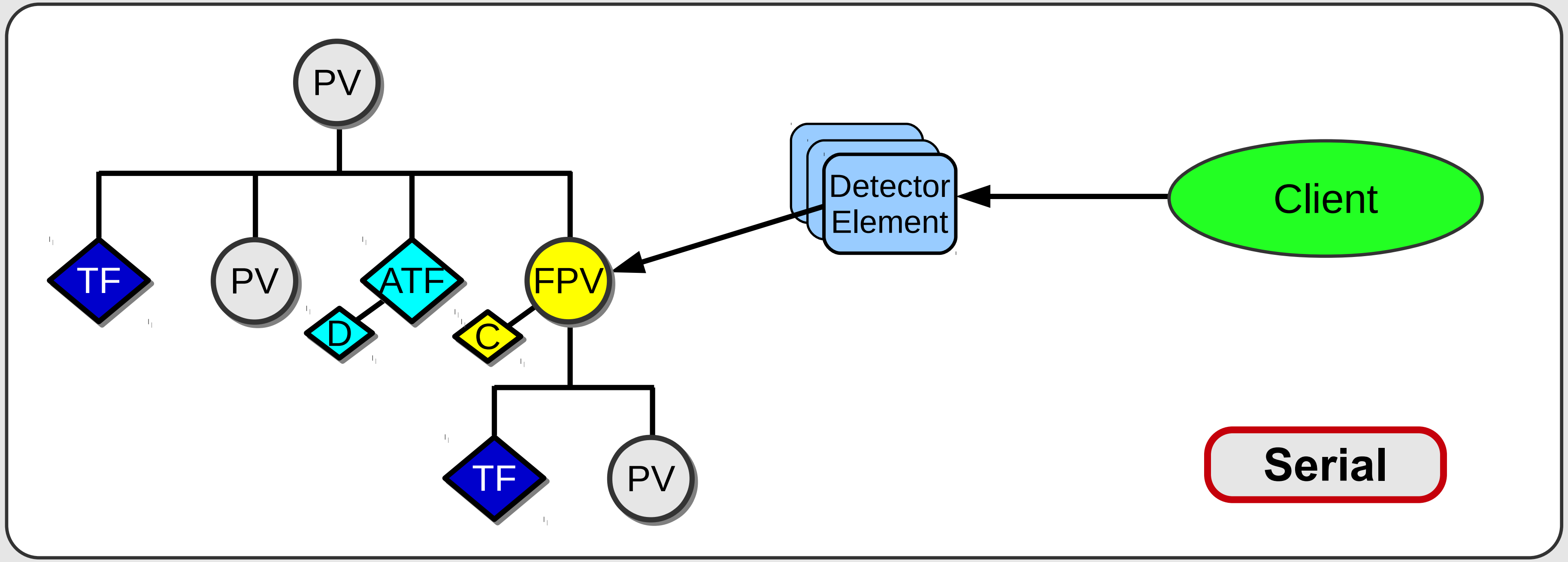
- Significant fraction of Conditions data has no associated callback function ("raw")
  - big overhead if we have to create a new CondAlg for each one!
  - want to just read them in, provide WriteCondHandles for them (to satisfy downstream data dependencies), update the handle when it gets into a new validity range

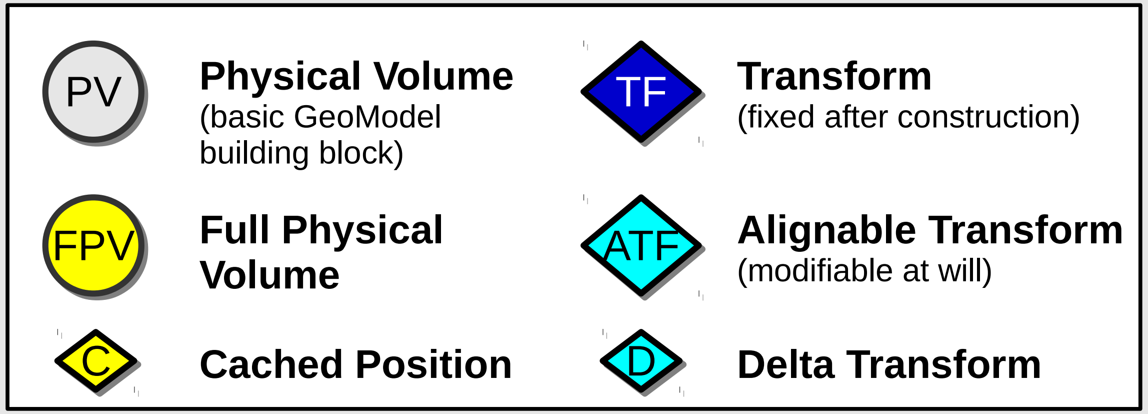
- generic alg `IOVSvc/CondInputLoader`
  - supply with list of db items (folders) to be loaded, just like with the IOVDbSvc

```
from IOVSvc.IOVSvcConf import CondInputLoader
topSequence += CondInputLoader( "CondInputLoader" )

topSequence.CondInputLoader.load += [
    ('AthenaAttributeList', '/path/to/DB/folder1'),
    ('AthenaAttributeList', '/path/to/DB/folder2'),
    ('CaloLocalHadCoeff',   '/CALO/HadCalib/CaloEmFrac') ]
```

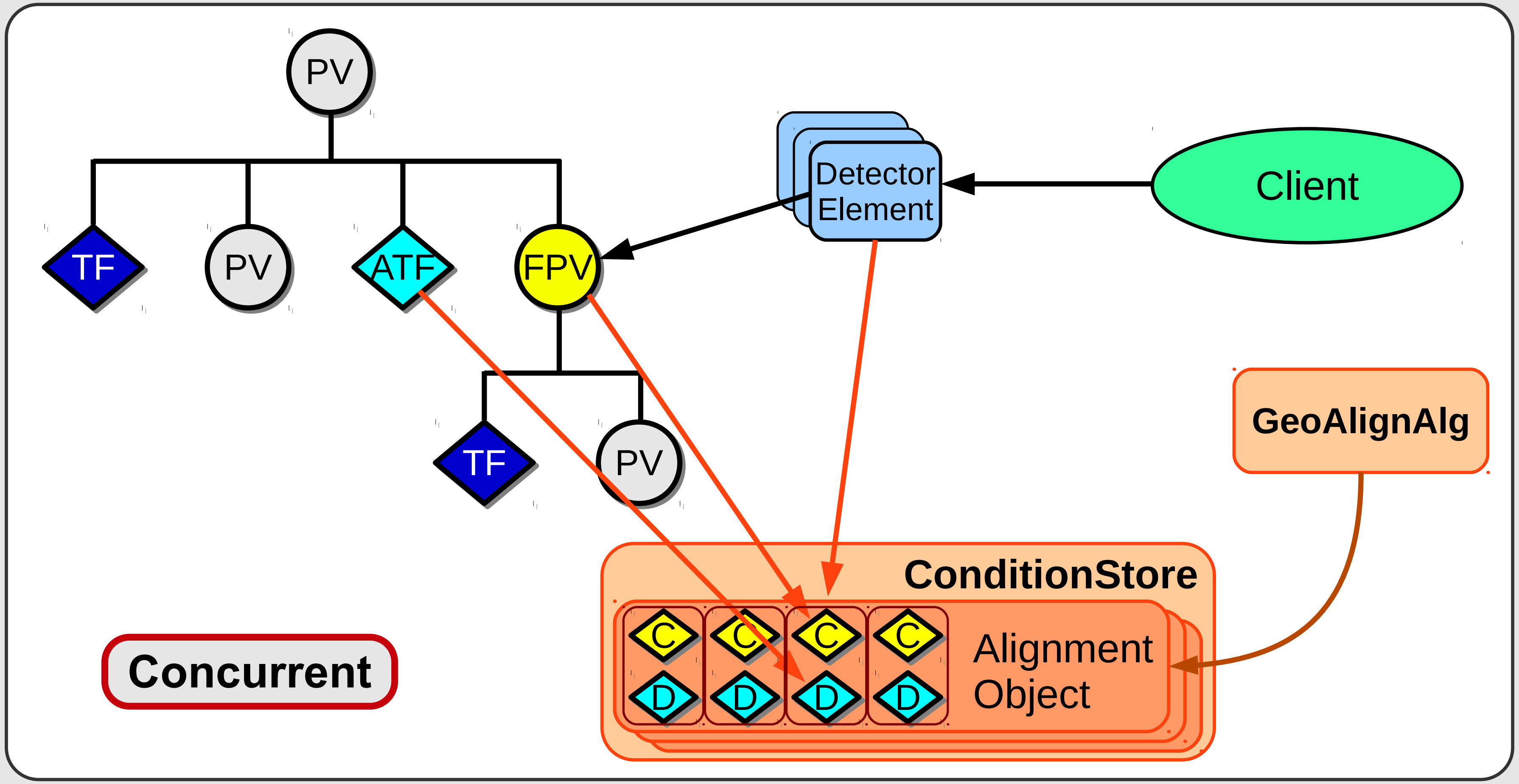
Detector Geometry Alignment





- ATLAS's geometry model (**GeoModel**) is not exposed to Detector Description clients
- Readout geometry layer consists of subsystem specific **Detector Elements**
- Each Detector Element has a pointer to **Full Physical Volume**

Geometry Alignments in AthenaMT



- The **Alignment Object** is a regular ConditionObject in a ConditionContainer, so it should be handled as any other ConditionObject in AthenaMT
  - Created by a **ConditionAlgorithm** (replacement of current callback function)
  - Accessed from the FPV and ATF via **ConditionHandle**
- By making Detector Elements aware of the Alignment Objects we can make the transition transparent to Detector Description clients

Conclusions

- Have to be able to handle rapidly changing conditions with short IOVs
- Use concept of Handles to manage data dependencies and hide implementation details
  - clients are blind to condition updates once they use CondHandles
- Re-use existing components
  - Algorithms for processing units
  - Data dependencies
  - Scheduler to do updates on demand
- Detector Alignments can use same infrastructure
- Minimize memory
  - only one Condition Store
  - objects are held in containers, one entry per IOV
    - garbage collection can be done on a per-object level

Extras

Memory Management

- While a multi-cache store makes optimal use of memory (no duplication of objects), the store will continue to grow with time
- Depending on memory constraints, may become necessary to perform garbage collection
  - prune ConditionContainers of old, unused entries
- Possible pruning techniques:
  - only keep N copies
  - keep reference count of which entries are in use, purge old entries

CondInputLoader Functionality

```
CondInputLoader::initialize() {
  // do translation of FolderName to SGKey
  …
  // set the Write dependencies via linking Property "Load" to ExtraOutputDeps()
  …

  // register Write DataHandles with CondSvc
  for (auto &e : m_load) {
    if (e.key() == "") {
      sc = StatusCode::FAILURE;
      ATH_MSG_ERROR("   ERROR: empty key is not allowed!);
    } else {
      Gaudi::DataHandle dh(e, Gaudi::DataHandle::Writer, this);
      if (m_condSvc->regHandle(this, dh, e.key()).isFailure()) {
        ATH_MSG_ERROR("Unable to register WriteCondHandle " << dh.fullKey());
        sc = StatusCode::FAILURE;
      }
      // remove proxy reset control from old IOVSvc
      m_IOVSvc->ignoreProxy(dh.fullKey().clid(), e.key());
    }
  }
}
```

CondInputLoader Functionality

```
CondInputLoader::execute() {
    for (auto &obj: m_load) {

    CondContBase* ccb(0);
    if (! m_condStore->retrieve(ccb, obj.key()).isSuccess()) {
      ATH_MSG_ERROR( "unable to get CondContBase* for " << obj
                     << " from ConditionStore" );
      continue;
    }

    if (! ccb->valid(now)) {
       if (m_IOVSvc->createCondObj( ccb, obj, now ).isFailure()) {
          std::string dbKey = m_folderKeyMap[obj.key()];
          ATH_MSG_ERROR("unable to create Cond object for " << obj << " dbKey: "
                        << dbKey);
          return StatusCode::FAILURE;
    } else {
       ATH_MSG_INFO( "  CondObj " << obj << " is still valid at " << now );
    }
    evtStore()->addedNewTransObject(obj.clid(), obj.key());
}
```

IOVSvc::createCondObj

```
IOVSvc::createCondObj(CondContBase* ccb, const DataObjID& id,
                      const EventIDBase& now) {
   if (getRangeFromDB(id.clid(), id.key(), t_now, range, tag, ioa).isFailure()) {
    ATH_MSG_ERROR( "unable to get range from db for "
                   << id.clid() << " " << id.key() );
    return StatusCode::FAILURE;
  }

  DataProxy *dp = ccb->proxy();
  DataObject* dobj(0);
  void* v(0);
  if (dp->loader()->createObj(ioa, dobj).isFailure()) {
    ATH_MSG_ERROR(" could not create a new DataObject ");
    return StatusCode::FAILURE;
  } else {
    v = SG::Storable_cast(dobj, id.clid());
  }
  EventIDRange r2(EventIDBase(range.start().run(), range.start().event()),
                  EventIDBase(range.stop().run(),  range.stop().event()));

  if (!ccb->insert( r2, v)) {
    ATH_MSG_ERROR("unable to insert Object at " << v << " into CondCont "
                  << ccb->id() << " for range " << r2 );
    return StatusCode::FAILURE;
  }
  return StatusCode::SUCCESS;
}
```