ATLAS experience running ATHENA and TDAQ software

Werner Wiedenmann

University of Wisconsin

Workshop on Virtualization and Multi-Core Technologies for LHC

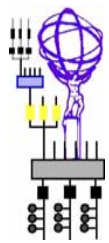
CERN, April 14-16, 2008

Introduction

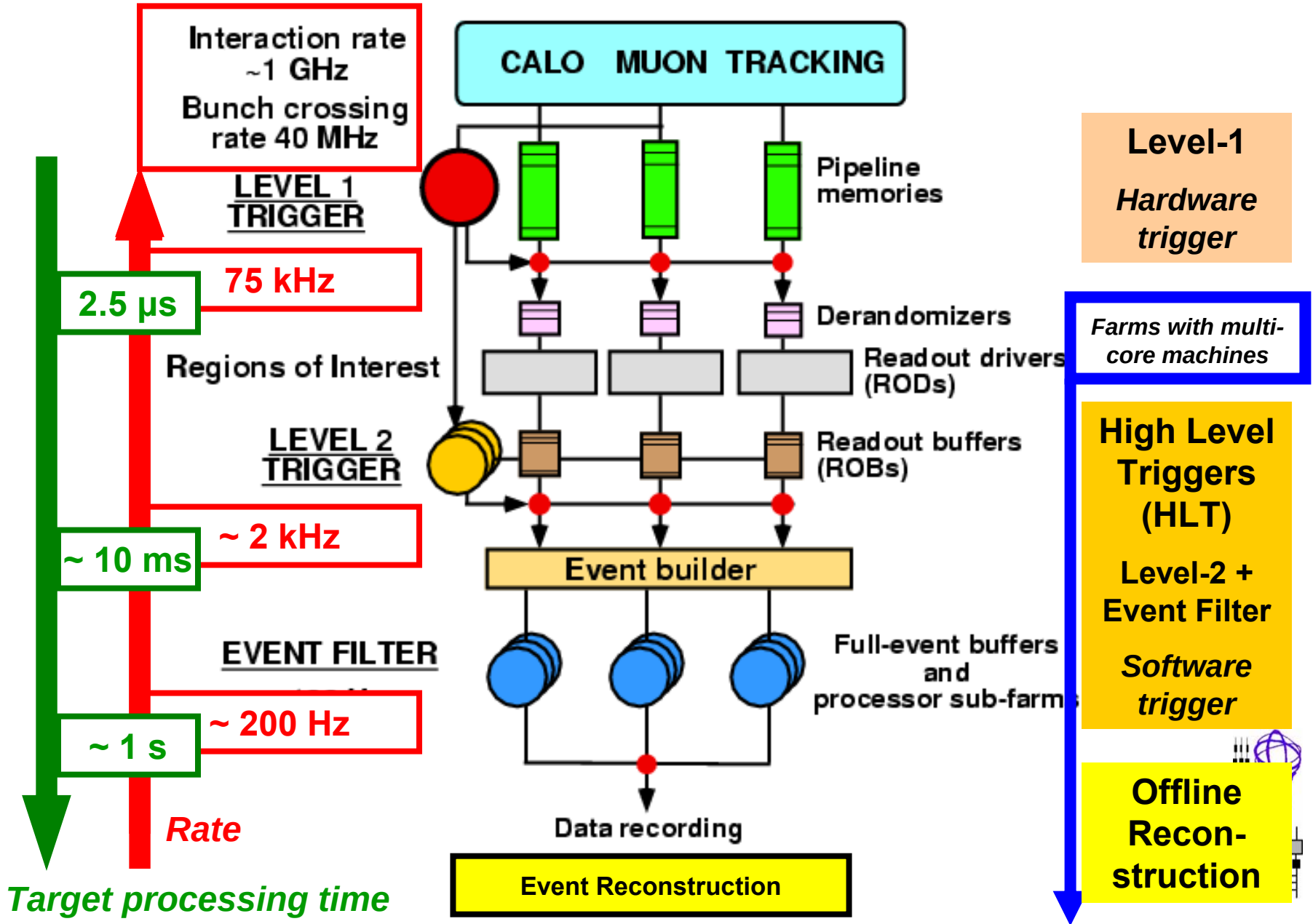
- Atlas software environment
- Experience with multi-threading
- Experience with multiple process instances
- Questions on further technology evolution
- Summary and Plans

Many thanks for contributions to:

A. dos Anjos, A. Bogaerts, P. Calafiura, H. von der Schmitt, S. Snyder



From Detector Data to Offline Reconstruction



High Level Triggers and Offline use Athena Framework

- **Level-2** (75 kHz→2 kHz, 10 ms target processing time)
  - Partial event reconstruction in Regions of Interest (RoI)
  - RoI data are requested over network from readout system
  - HLT selection software runs in the Level-2 Processing Unit (L2PU) in *worker threads*
  - Each thread processes one event (event parallelism)
  - Executable size ~ 1 Gb
- **Event Filter** (2 kHz→200 Hz, 1-2 s target processing time)
  - *Full event reconstruction* (seeded by Level-2 result) with *offline-type algorithms*
  - Independent *Processing Tasks (PT)* run selection software on Event Filter (EF) farm nodes
  - Executable size ~ 1.5 Gb
- **Offline Reconstruction**
  - *Full event reconstruction* with *best calibration*
  - Executable size ~ 1.5 - 2 Gb
- **HLT Selection Software and Offline Reconstruction Software**
  - *Both use Athena/Gaudi framework interfaces and Services*
  - Share many services and algorithms, e.g. detector description, track reconstruction tools etc.

LVL2 and EF Data Flow : Threads and Processes

Werner Wiedenmann, Virtualization and Multi-Core Technologies for LHC 2008/04/15

Level-2

Event processing in Worker threads

Online/Data Collection Framework

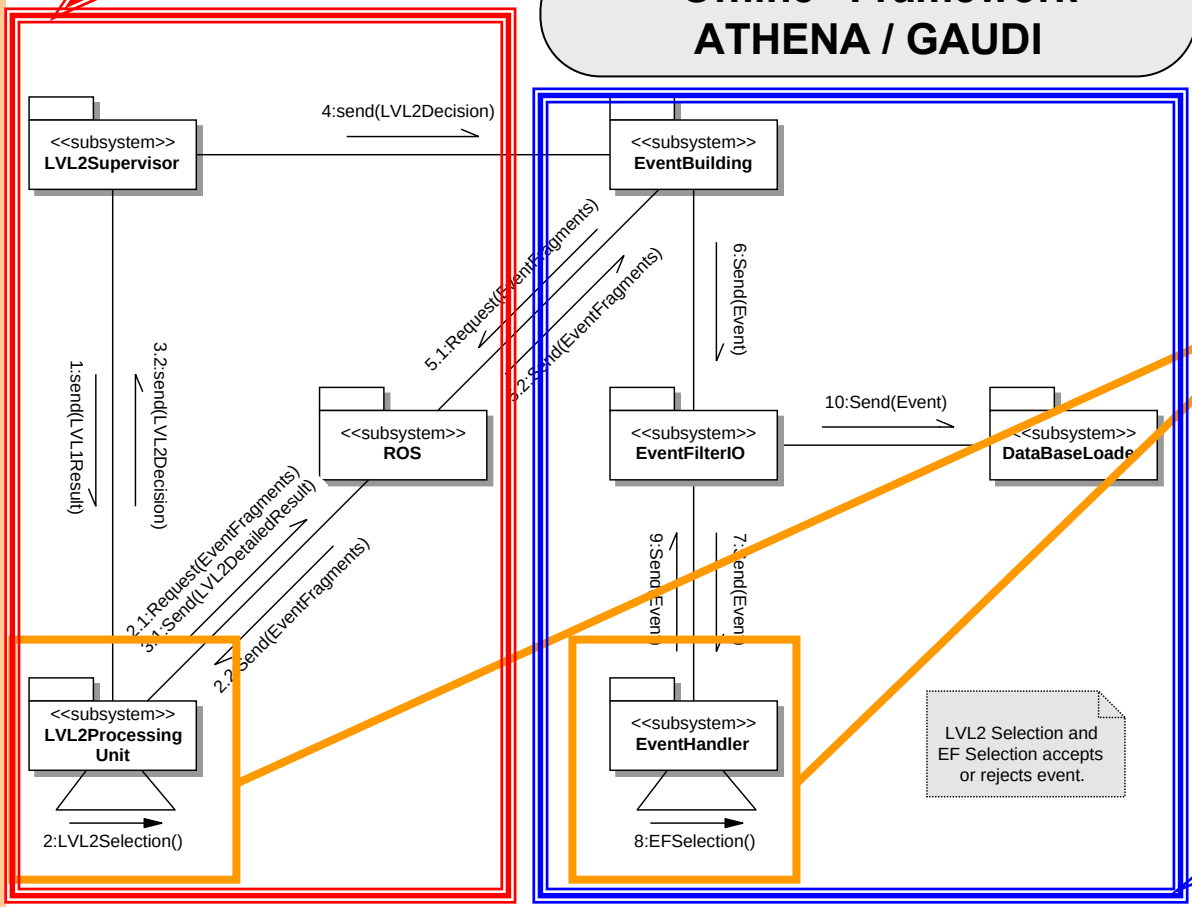
“Offline” Framework ATHENA / GAUDI

Data Collection

L2PU/EFPT

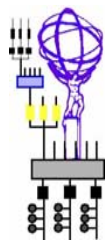
Steering Controller

HLT Event Selection Software

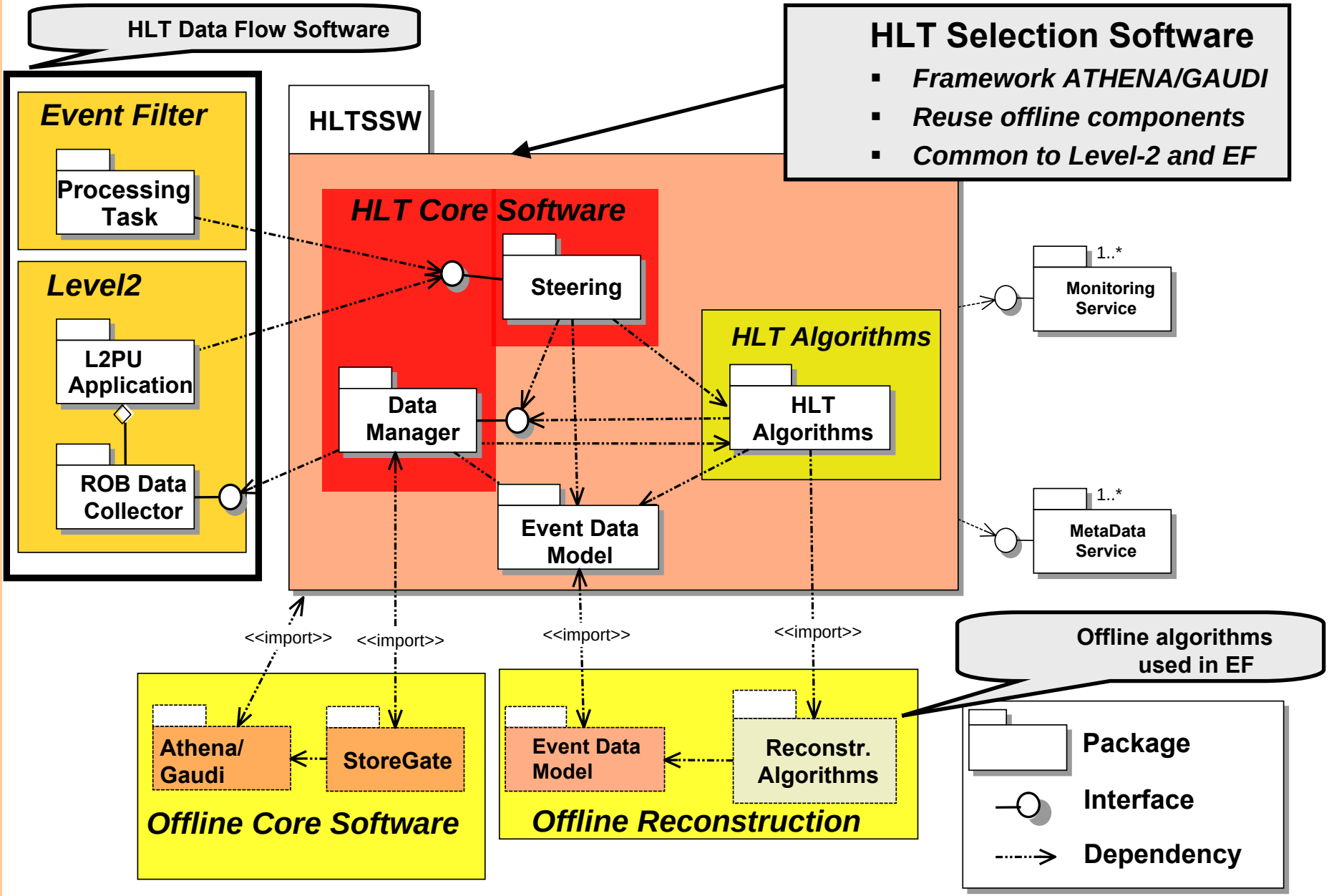


Event Filter

Event processing in Processing Tasks

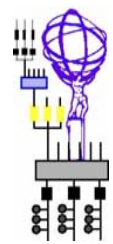


HLT Event Selection and Offline Software



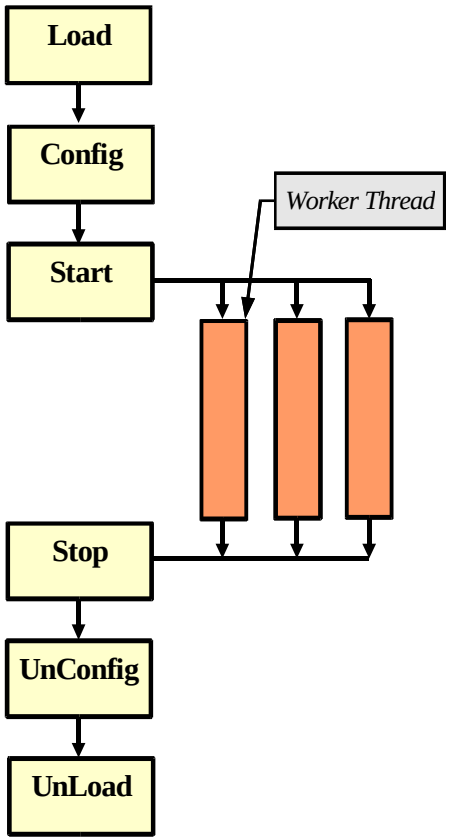
Multi-core machines in ATLAS

- Multi-core machines are used in all big trigger and reconstruction farms in Atlas (e.g. Dual Quad-core processors for trigger farm)
- Large number of CPU cores calls for more parallelism
  - Event parallelism inherent to typical high energy physics selection and reconstruction programs
  - Parallelization inside applications may provide huge speed ups but requires typically also careful (re)design of code → may be demanding for are large existing code basis
- Exploit Parallelism with
  - Multi-threading
  - Multiple processes
- ATLAS has large code basis mostly written and designed in "pre-multi-core era"
  - HLT reconstruction code and offline reconstruction code mainly process based and single threaded
  - However many multi-threaded applications available in TDAQ framework
- Experimented with multi-threading and multiple processes (and mixture of both)
  - Existing code basis implies boundary conditions for future developments



Multi-threading

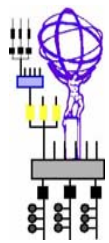
- Code sharing
- Small context switch times → "lightweight processes"
- Automatic sharing of many hardware resources
- Example: Trigger L2 Processing Unit



- *Event processing in multiple worker threads*
- *HLT selection software is controlled by TDAQ framework*
- *Special version of Gaudi/Athena framework to create selection algorithm instances for worker threads*
- *Development and MT tests started on dual processor single core machines, long before multi-core machines were available*

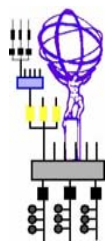
Multi-threading support in Athena framework

- *Multi-threading*: All services and algorithms which modify data have to be thread specific (e.g. StoreGate/EventStore). Threads may however also use “read only” services which are common to all threads (e.g. GeometrySvc, DetectorStore)
- All thread specific instances of services and algorithms are distinguished by *type* and *(generic name)__(thread ID)*. E.g. create an algorithm of type "TriggerSteering" and generic name "TrigStr" for 2 threads:
  - TriggerSteering/TrigStr__0
  - TriggerSteering/TrigStr__1
- Assumption :
  - *Algorithms* (SubAlgorithms) are always thread specific, i.e. for each thread an algorithm copy is generated automatically
  - If a *Service* is run thread specific or common for all threads has to be specified in the configuration
- Modified Athena can also be used for normal offline running (i.e. no thread ID will be appended, number of threads = 0)

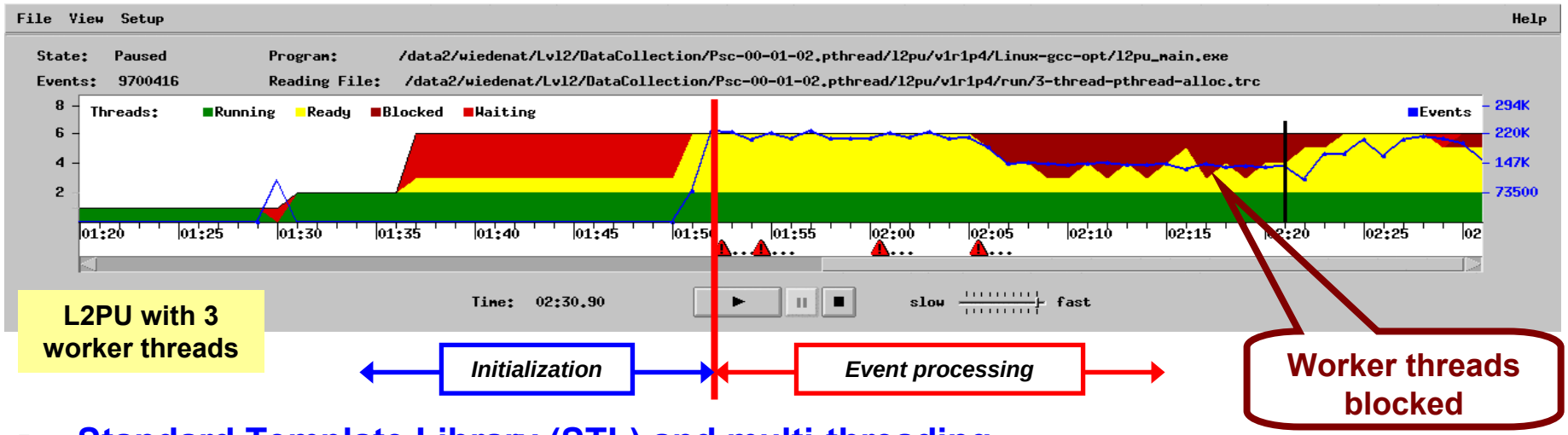


Experiences with Multi-threading (1)

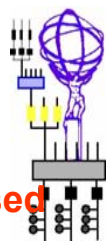
- Created different event selection slices which could run multithreaded
- Some technical issues are historical now but interesting, e.g.
  - Implementation of STL elements with different compiler versions → memory allocation model not optimal for “event parallelism”
  - Thread safe external libraries



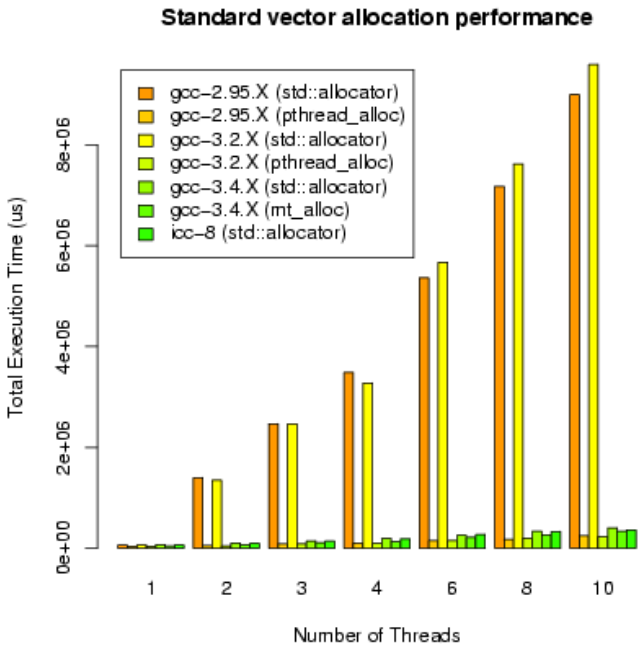
Multi-threading Performance



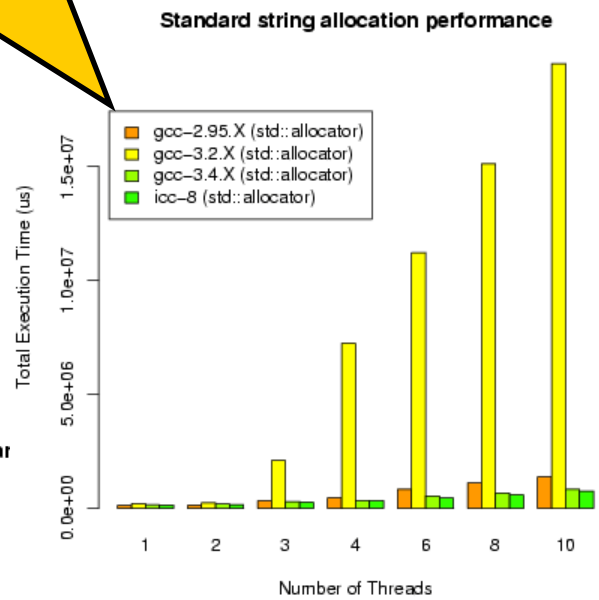
- **Standard Template Library (STL) and multi-threading**
  - L2PU: **independent event processing in each worker thread**
  - **Default STL memory allocation scheme** (common memory pool) for containers is **inefficient** for L2PU processing model → frequent locking
  - L2PU processing model favors **independent memory pools** for each thread
    - Use *pthread allocator/DF_ALLOCATOR* in containers
    - Solution for strings = avoid them
  - Needs changes in offline software + their external sw
    - need to insert DF_ALLOCATOR in containers
    - utility libraries need to be compiled with DF_ALLOCATOR
    - design large containers to **allocate memory once** and **reset data during event processing**.
- **Evaluation of problem with gcc 3.4 and icc 8**
  - Results with simple test programs (→ were also used to understand original findings) indicate considerable improvement (also for strings) in libraries shipped with new compilers
  - **Insertion of special allocator in offline code may be avoided when new compilers are used**

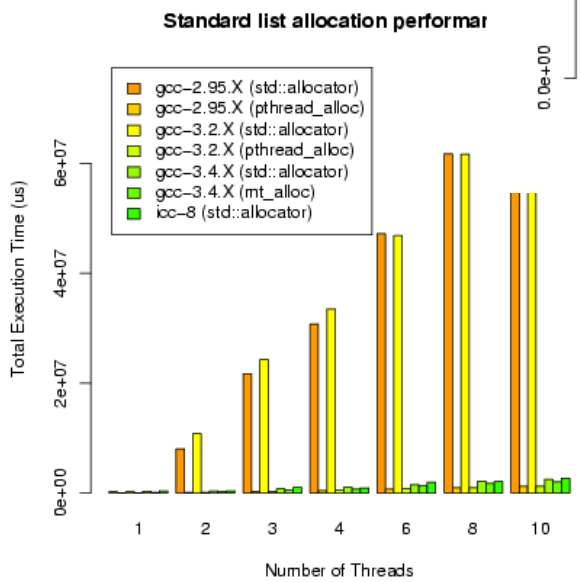


Multithreading: Compiler Comparison (vector, list, string)



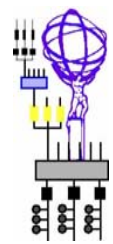
Gcc 2.95 not valid, string not thread safe





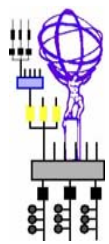
Need technology tracking

- Compilers
- Debuggers
- Performance assessment tools



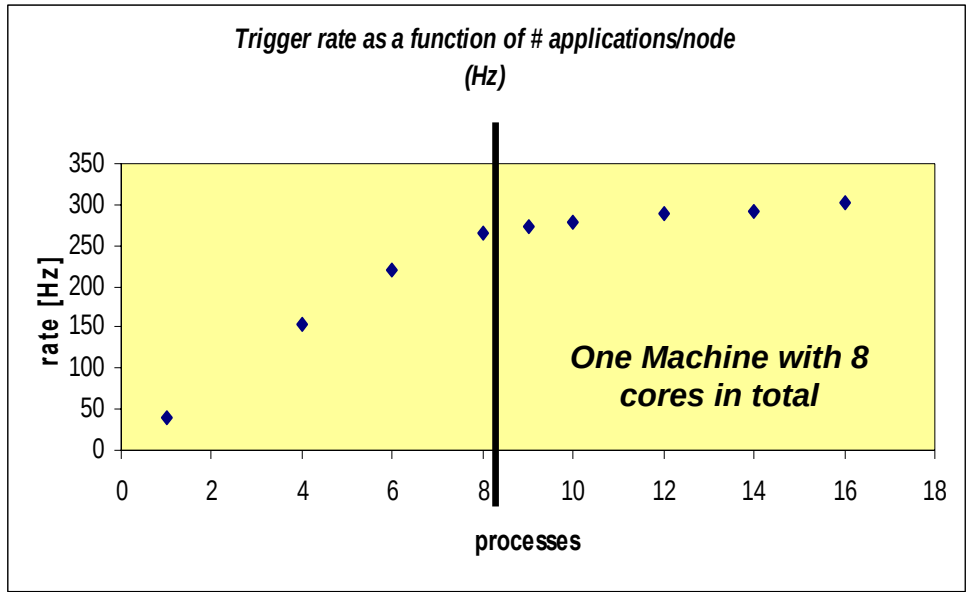
Experiences with Multi-threading (2)

- Software development
  - Developers have to be familiar with thread programming
    - Need special training and knowledge
  - Developers have to take into account for their code the multi threading model of L2 → event parallelism
    - Created emulator *athenaMT* as a development tool/environment for Lvl2 code
  - Synchronization problems for multi-threaded code are tedious to debug
  - Need good tools to assist developers for debugging and optimizing multi-threaded programs
  - Typically selection code changes rapidly due to physics needs → constant need for re-optimization
- Problem: Preserve thread safe and optimized code over release cycles and in a large heterogeneous developer community (→ coupling of different software communities with different goals)
  - Presently we run n (= # cores) instances of L2 Processing Unit on a multi-core machine with one worker thread

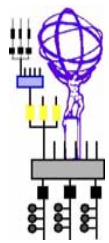


Multiple Processes

- Run n process instances on machine with n cores
  - Easy to do with existing code → a priori no code changes required
  - Observe good scaling with number of cores

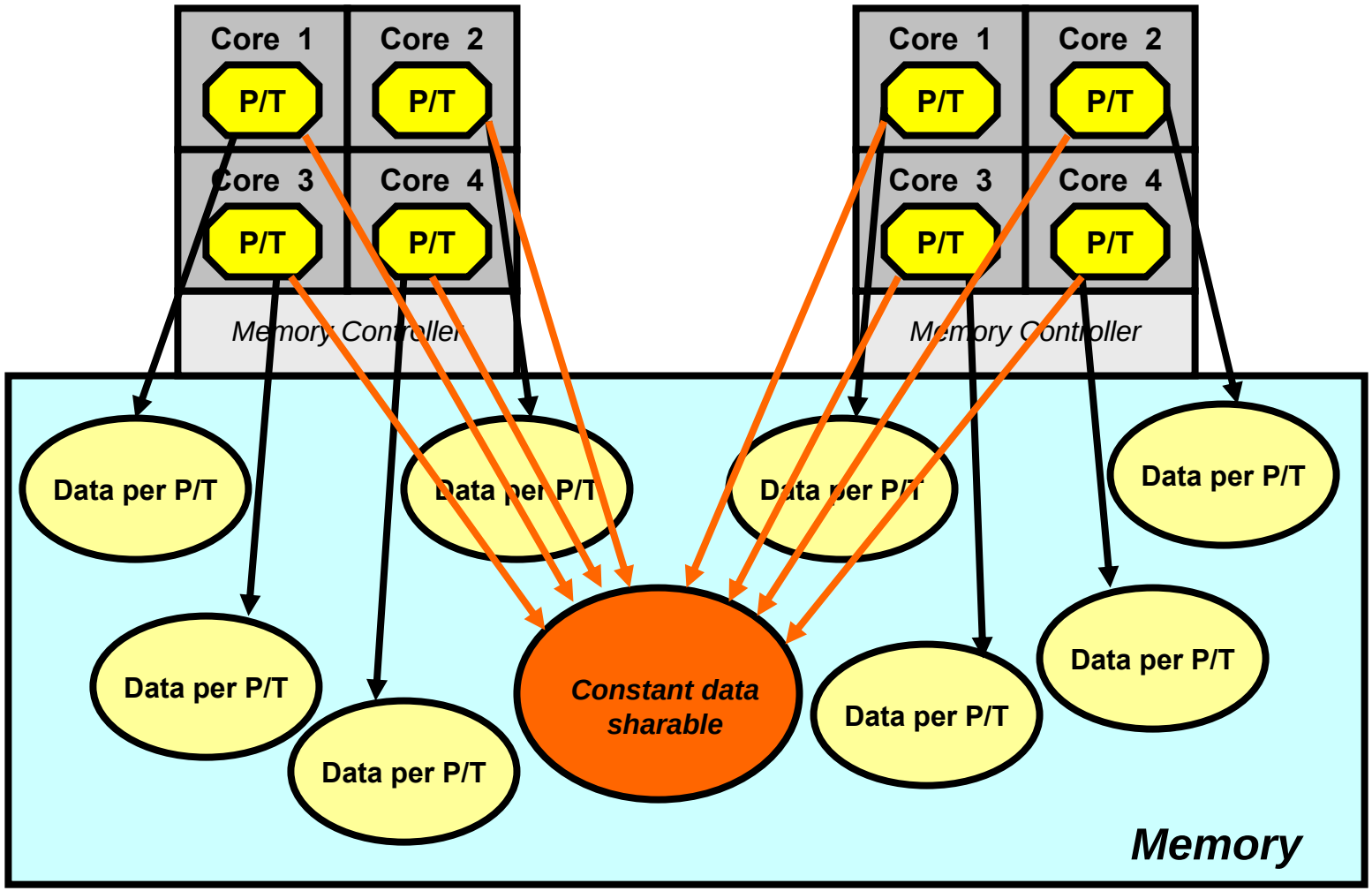


- Disadvantages: Resource sharing and optimization → Resource requirements are multiplied with number of process instances
  - Memory size
  - OS resources: file descriptors, network sockets,….
  - On trigger farms
    - Number of controlled applications
    - Number of network connections to readout system
    - Transfer of same configuration data n times to the same machine
    - Recalculation of the same configuration data n times
    - Optimal CPU utilization → use CPU for event processing while waiting for input data



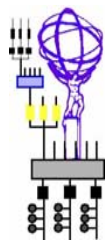
Reduce Memory Size

Typically in HEP applications all processes use a large amount of constant configuration and detector description data. Two approaches tried in prototypes



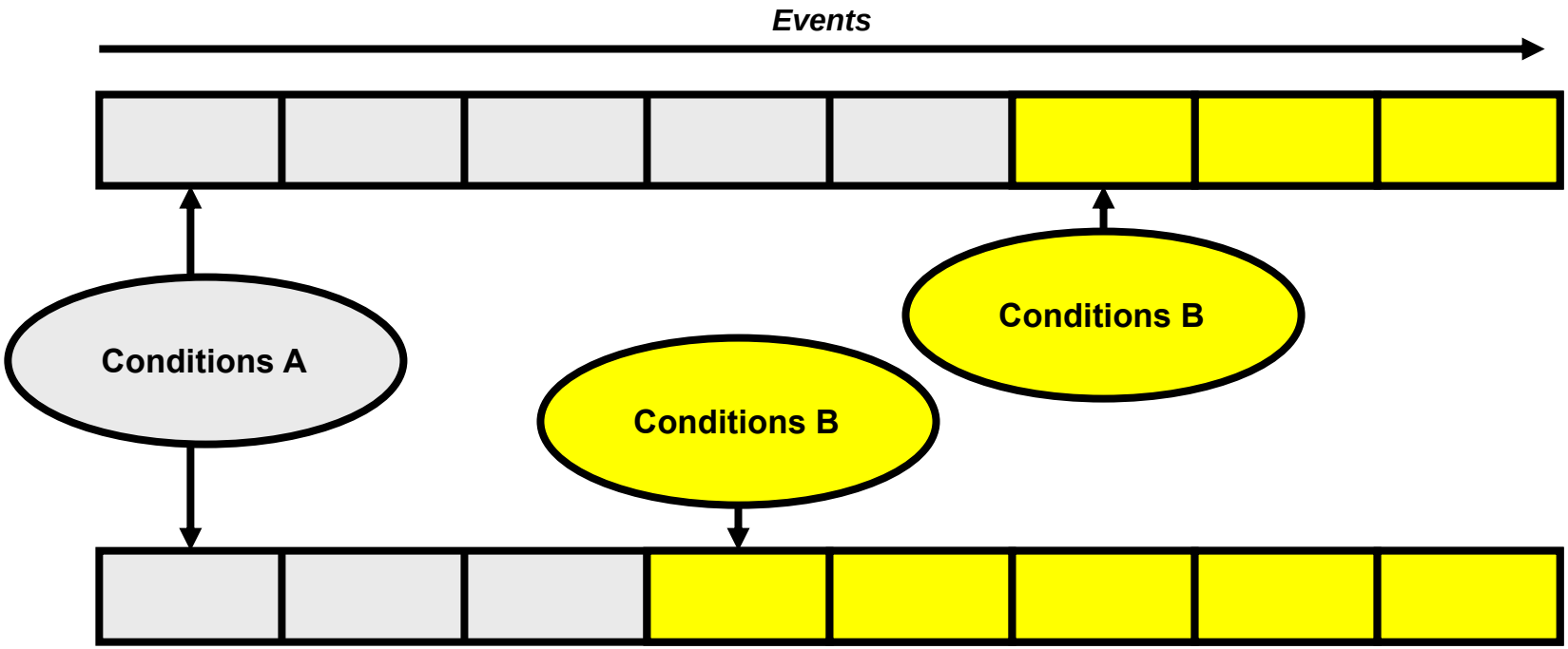
A: Memory sharing via fork (1)

- Idea and prototype by Scott Snyder (BNL)
  - http://indico.cern.ch/getFile.py/access?contribId=132&sessionId=7&resId=0&materialId=slides&confId=5060
  - Following is from Scott's slides
- Basic Ideas
  - Run multiple Athena reconstruction jobs, sharing as much memory as possible
  - Minimize number of required code changes, let the OS do most of the work
  - Use fork()
- Fork()
  - Fork() clones a process, including its entire address space
  - Modern OS, fork() uses "*Copy On Write*" → memory is shared up to the point a process writes to it. Memory will be copied and the affected changes will become unshared.
  - Fork is done after the first event is processed, but before any output is written
  - As much memory as possible is *automatically* shared between processes. Memory which is modified will become unshared. Static configuration data will remain shared.



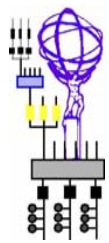
A: Memory sharing via fork (2)

- Advantages
  - All memory that can be shared will be
  - Code changes restricted to few framework packages, bulk of the code remains untouched
  - Don't need to worry about locking
- Disadvantages
  - Memory cannot be re-shared after it became unshared → maybe e.g. problem for conditions data



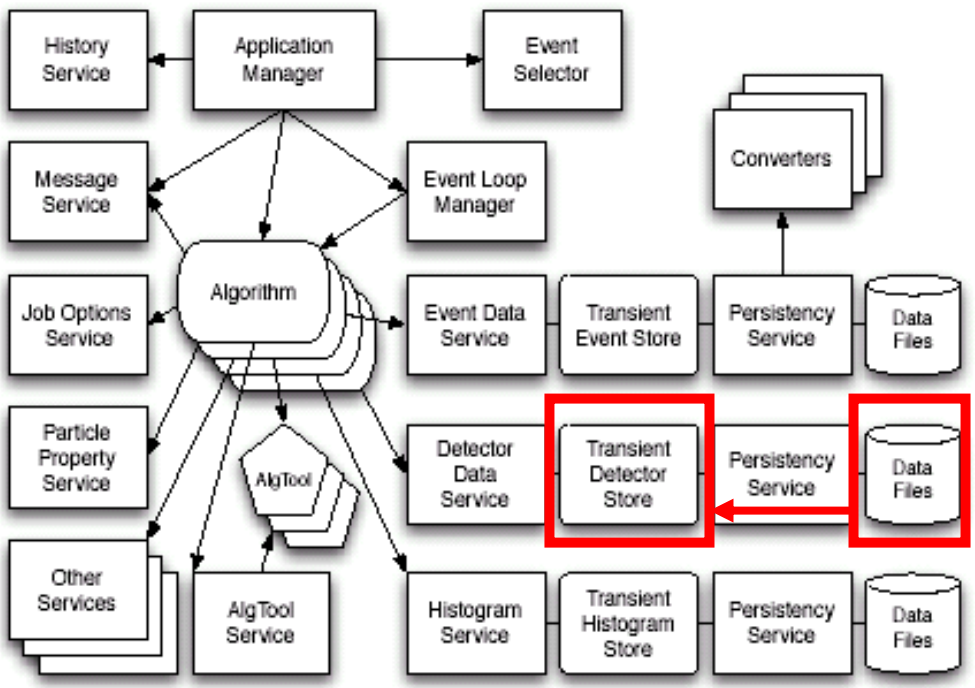
A: Memory sharing via fork (3)

- First prototype results encouraging with a standard reconstruction job from a recent release
  - Input Z→ee Monte Carlo data
  - All detectors used
  - Total job size ~ 1.7 Gb
- Data after running a few events (crude estimate from /proc/…/smaps)
  - Total heap size: 1149 Mb
  - Total heap resident size: 1050 Mb
  - Shared memory: 759 Mb
  - Private memory: 292 Mb
- ~ 2 / 3 of memory remains shared
- With real data frequent conditions data updates may change the results (→ see previous slide)

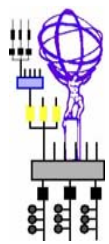


B: Detector Data in Shared Memory (1)

- Idea and prototype by Hans von der Schmitt (*MPI for Physics, Munich*)
  - http://indico.cern.ch/getFile.py/access?contribId=131&sessionId=7&resId=0&materialId=slides&confId=5060
  - Following is from Hans's slides
- DetectorStore sharable

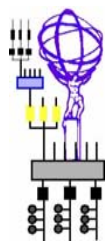


- *DetectorStore fill once and share*
- *Reduce configuration time by avoiding same calculations multiple times*
- *Memory saving ~ O(100 Mb), more possible in future*



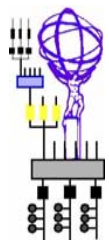
B: Detector Data in Shared Memory (2)

- Athena Storage Manager *architecture very well suited* to support *shm*
  - administration of storable objects on the heap and in *shm*
  - In *shm* the store can not only serve algorithms within one Unix process but also between processes
  - Need additional logic to fill, lock and attach shared memory write-protected to a process → ideally handled in a state change for the reconstruction program / trigger code
- In a prototype the Athena Storage Manager "StoreGate" was modified to be able to use also shared memory to store and retrieve objects
  - Quite some code changes to present implementation
  - Tested with some simple examples
  - Roll out/in of complete *shm* to file seems fine
- Implementation details can be found in presentation mentioned above



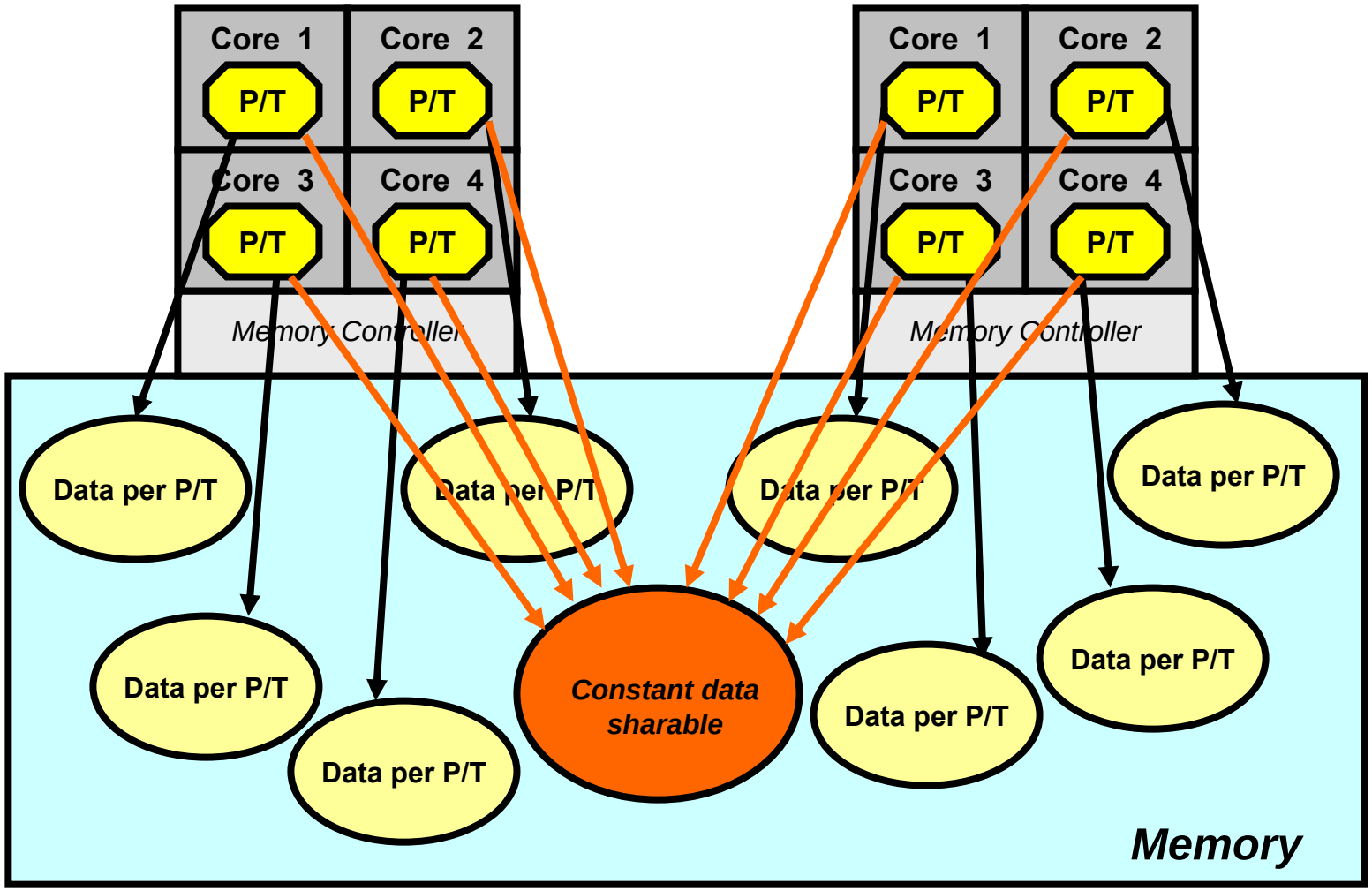
Questions on further Technology Evolution

- What strategy is best for future hardware platforms ?
- Could mixed scenarios be required ? → run multiple processes each multithreaded ?
- Does one have to worry about CPU affinity ?
- Is there a danger to lock in to a certain hardware technology ?
- …



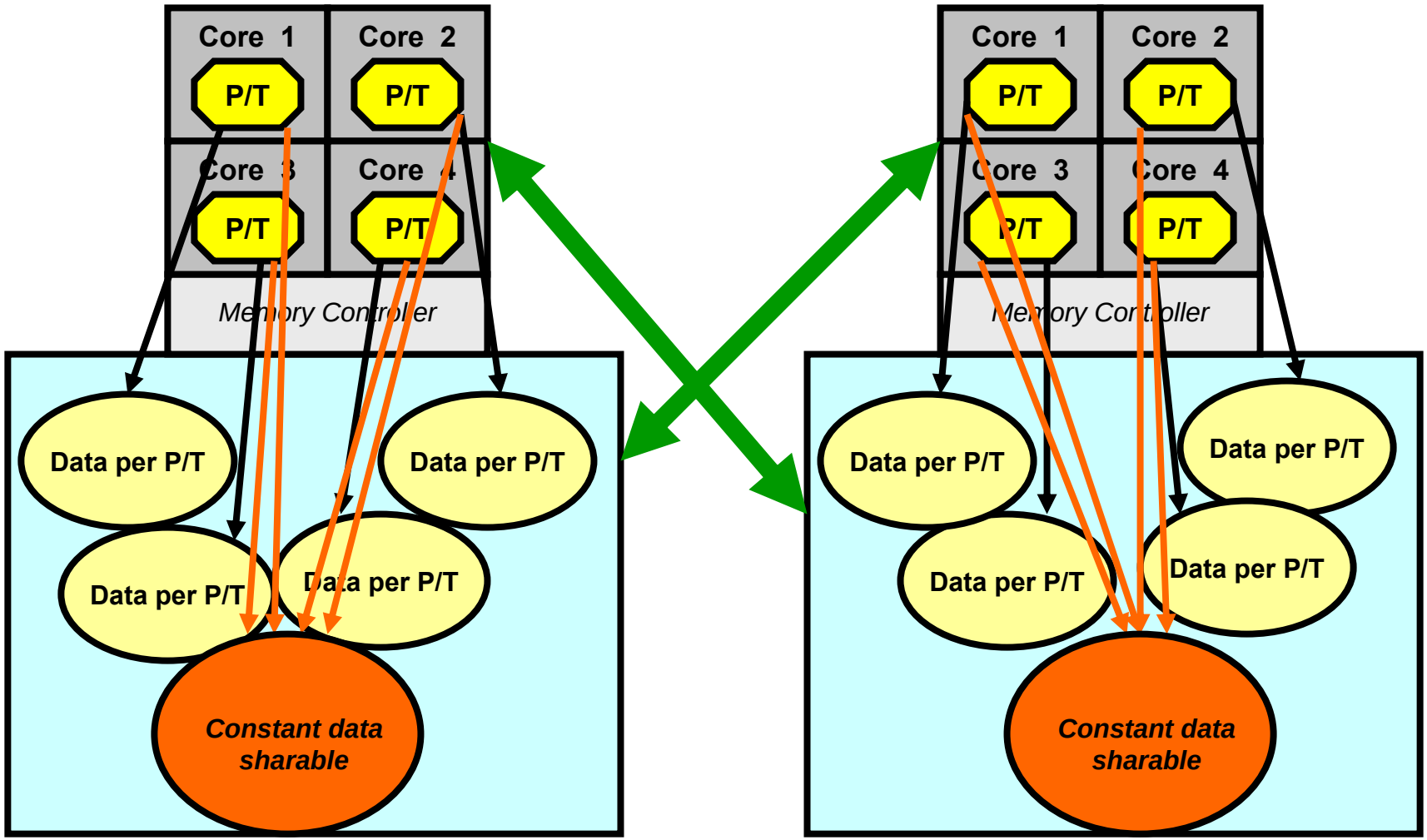
Questions

- all CPU sockets access/share common memory system
  - Pure multithreading → Run one application instance per machine with n worker threads, n >= #CPU sockets * #cores per CPU
  - Multiple processes and one shared memory segment



Questions

- each CPU socket has its own memory subsystem attached
  - Multiple processes, each process multithreaded → Run one application instance per CPU socket with n worker threads, n >= #cores per CPU
  - Multiple processes and multiple shared memory segments



Summary and Plans

- Due to the large code basis written as single threaded applications it is probably best in the near future to explore the multi applications approach first for multi-core machines
  - Most important : reduce memory requirement
  - Investigate resource optimization strategies
  - Compiler support
  - OS support (scheduler, resource sharing, …)
- Explore performance assessment and optimization tools
- Multi-threading may offer big performance gains but is more difficult to realize for a large code basis written over a long time and needs
  - Good support from compilers, external libraries and programming tools
  - Developer training
- In both cases, multi-process and multi-threaded, we would like to have more general support libraries for typical recurring code requirements in HEP application on multi-core machines e.g.
  - Access/manage “logical” file descriptors shared or split among different processes / threads
  - High level “functions” to manage shared memory
  - Guidelines
  - ….