CMS and ROOT I/O

Dan Riley (Cornell)

ROOT I/O Workshop
2018-06-20

ROOT Output Serial Bottlenecks

ROOT output is currently the largest single bottleneck for CMS multi-threaded production jobs—but IO characteristics vary:

- AOD/MINIAOD
  - Relatively small data volumes, infrequent flushes, expensive compression, many branches
  - Compression is the main bottleneck
- RECO
  - Large data volume, frequent flushes, faster compression, many branches
  - Bottleneck is more complicated!
- GENSIM
  - Moderate data volume, moderate flush frequency, expensive compression, few branches
  - Also complicated, not addressed in this talk

Mitigation Approaches

Two strategies for addressing the bottlenecks:

- ROOT Implicit Multi-Threading (IMT)
  - IMT parallelizes branch buffer compression into TBB tasks
  - Helps most with many branches and expensive compressions
- CMS ParallelPoolOutputModule (PPOM) & ROOT TBufferMerger
  - Concurrency is limited to avoid excessive resource allocation
  - PPOM keeps a pool of output TBufferMergerFiles (derived from TMemFile)
  - Output is written to the available TBufferMergerFile with the most entries
  - Full TBufferMergerFiles are copied to a buffer and merged to the output file

TBufferMerger Versions

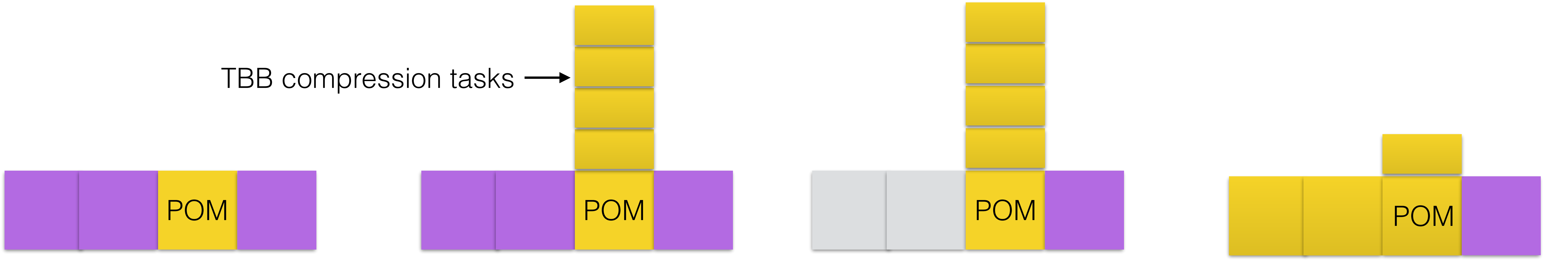
Not using the standard TBufferMerger

- **Standard version uses an auxiliary thread for the merge operation**
  - Due to compression of the branch keys during autosave operations, the merge operation can take enough CPU time to throw off our scheduling and oversubscribe resource allocations
- **Instead, using a slightly modified version of PR#1737 from mid-March**
  - “Make TBufferMerger agnostic about user's model for parallelism” does the merge on the caller’s thread
  - Good for CMS, but immediately reverted due to lack of parallelism when IMT is not used
  - Modified version does an `std::try_to_lock` on the merge mutex, adds to the queue instead of waiting if a merge is in progress
  - Some discussion in April about addressing the autosave CPU usage and other CMS requests. Status?

IMT in Schematic Form

IMT takes advantage of threads that would otherwise stall

- **IMT creates TBB tasks to compress branch buffers**
- **TBB tasks are queued on the PoolOutputModule thread's task queue**
- **If another thread has no work on its task queue, it will "steal" work from the PoolOutputModule queue**
    - This is invisible to the framework—it cannot distinguish idle threads from threads gainfully employed compressing branch buffers
    - IMT can't use threads that are blocked (e.g., on a mutex)



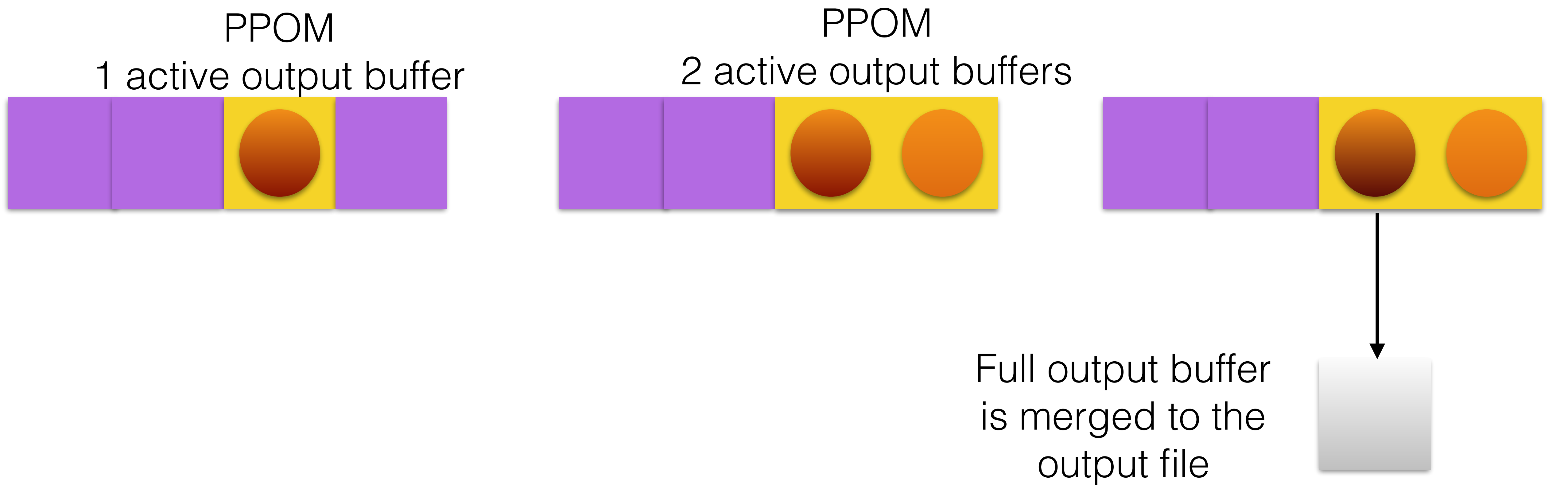
Streams stall waiting on PoolOutputModule

Threads for stalled streams "steal" compression tasks

ParallelPoolOutputModule Schematic

ParallelPoolOutputModule creates TBufferMergerFiles on demand

- **limited::OutputModule to limit the # of TBufferMergerFiles created**
  - Framework needs to know about the limit so it can schedule accordingly
- **Always fill the available TBufferMergerFile with the most entries**
  - Avoids synchronization effects, minimizes tail effects, approximates serial ordering
- **Branch buffer compression happens on the PPOM thread**
  - Possibly using IMT—can lead to non-trivial interactions



Framework interactions with IMT

Using TBB tasks for IMT can lead to unexpected interactions

- **Example: GEN-SIM production**
  - GEN-SIM has time consuming GEANT simulation tasks
  - Output file has few branches
- **Scenario:**
  - PoolOutputModule does a TTree::Fill() that results in a flush operation
  - IMT parallelizes the compression of the (small number of) branch buffers
  - Output module thread gets a relatively small buffer to compress, finishes early, and has to wait for other tasks to finish branch buffer compression
  - Starved for work, output module thread “steals” a GEANT simulation task
  - Output module task is blocked until the GEANT simulation task finishes

Solution/workaround

- `tbb::this_task_arena::isolate( [&]{ tree_->Fill(); } );`
  - Keeps the output module thread “honest” (no task stealing)

Other Developments

Other changes since the last workshop—significantly reduced lock contention:

- **Went hunting for unnecessary lock acquisitions elsewhere in CMSSW**
  - Expression parser in "lazy" evaluation mode was calling TClass::GetClass() excessively
  - One module was creating new instances of the StringCutObjectSelector every event
- **Creating TBufferMergerFile instances "on demand" resulted in lots of lock activity while the trees and branches were created**
  - Modified the ParallelPoolOutputModule to create the instances up front in a serial section

Philosophical(?) Digression

ROOT is a toolkit used in a variety of computations

- In the CMSSW framework, we end up with a mixture of very large tasks from the framework scheduler and relatively small ones from IMT
- This can lead to scheduling inefficiencies when a thread that initiated a set of IMT tasks steals a heavy-weight CMSSW task
  - It can also lead to bugs with thread locals, e.g. with recursive entry to the legacy TMinuit fitter
- We can mitigate this on a case-by-case basis via TBB isolation (or the "SERIAL" option for fits)
  - But that depends on knowing where IMT is used, which seems likely to expand
- It would be useful for CMS if there were an option for all IMT TBB tasks to use TBB isolation
  - Since the TBB pieces are well hidden the code changes would be fairly modest
  - Could be off by default to preserve the current task stealing behavior

Comparison Tests

Test setup:

- CMSSW_10_2_0_pre5 with CMS ROOT 6.12/07
- CMS workflow 500202.0: 13GeV TTBar, run2 conditions, semi-realistic pileup
    - RECO step, writing RECO, AOD and MINIAOD, standard compression levels
- Platform: 32 core Skylake-SP Gold 6130 CPU @ 2.10GHz
    - 32 threads and streams
    - System configured to be representative of what we expect for the next generation of CMS HLT and prompt-RECO farm systems

Tests:

- Normal PoolOutputModule with and without IMT
- ParallelPoolOutputModule with IMT
    - RECO output concurrency 6, AOD 6, MINIAOD 3 (6x6x3)
    - RECO with standard PoolOutputModule, AOD concurrency 6, MINIAOD 3 (1x6x3)
    - Tests to isolate performance issues: “no write” and “no fill”

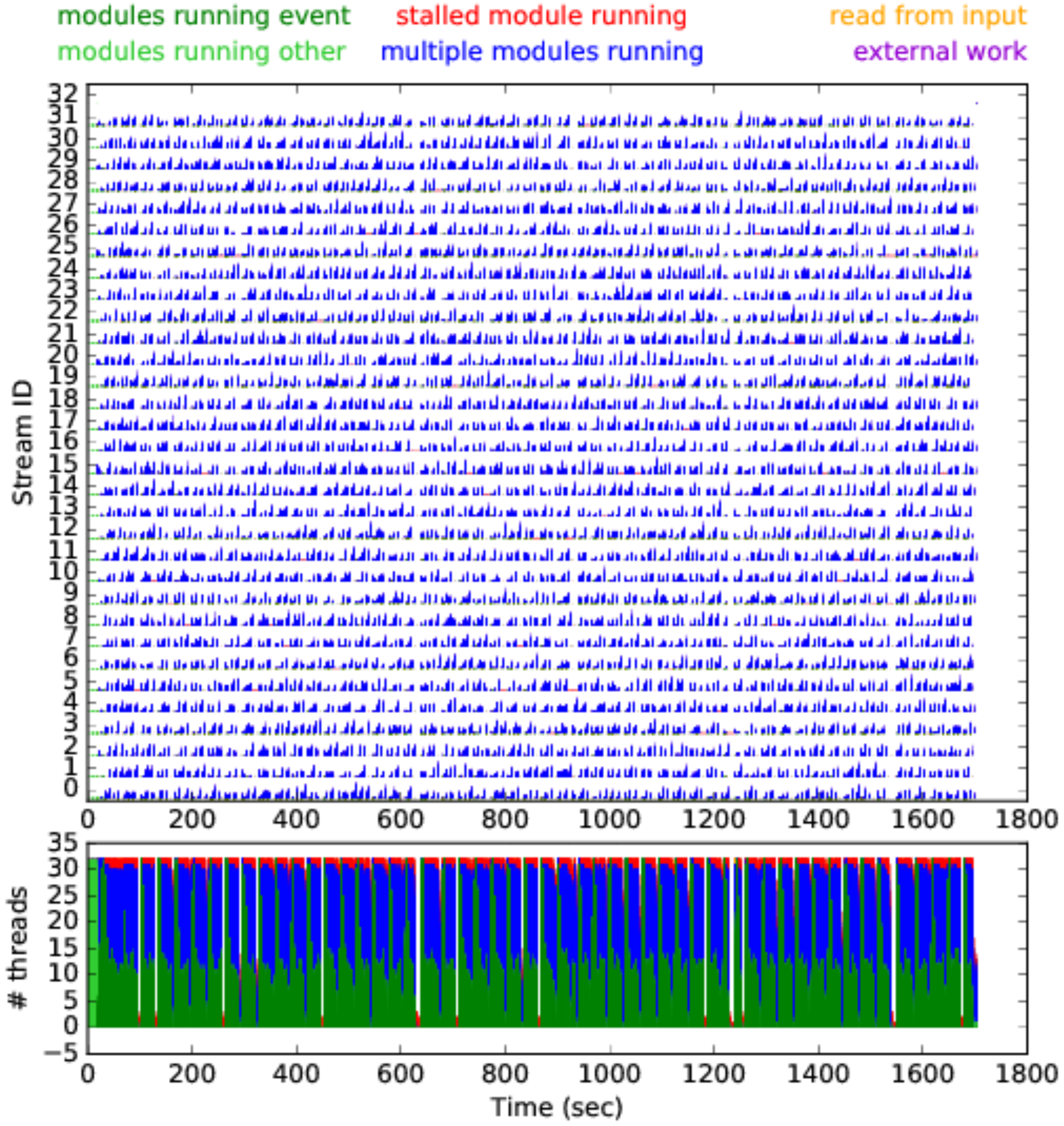
"No Write" and "No Fill"

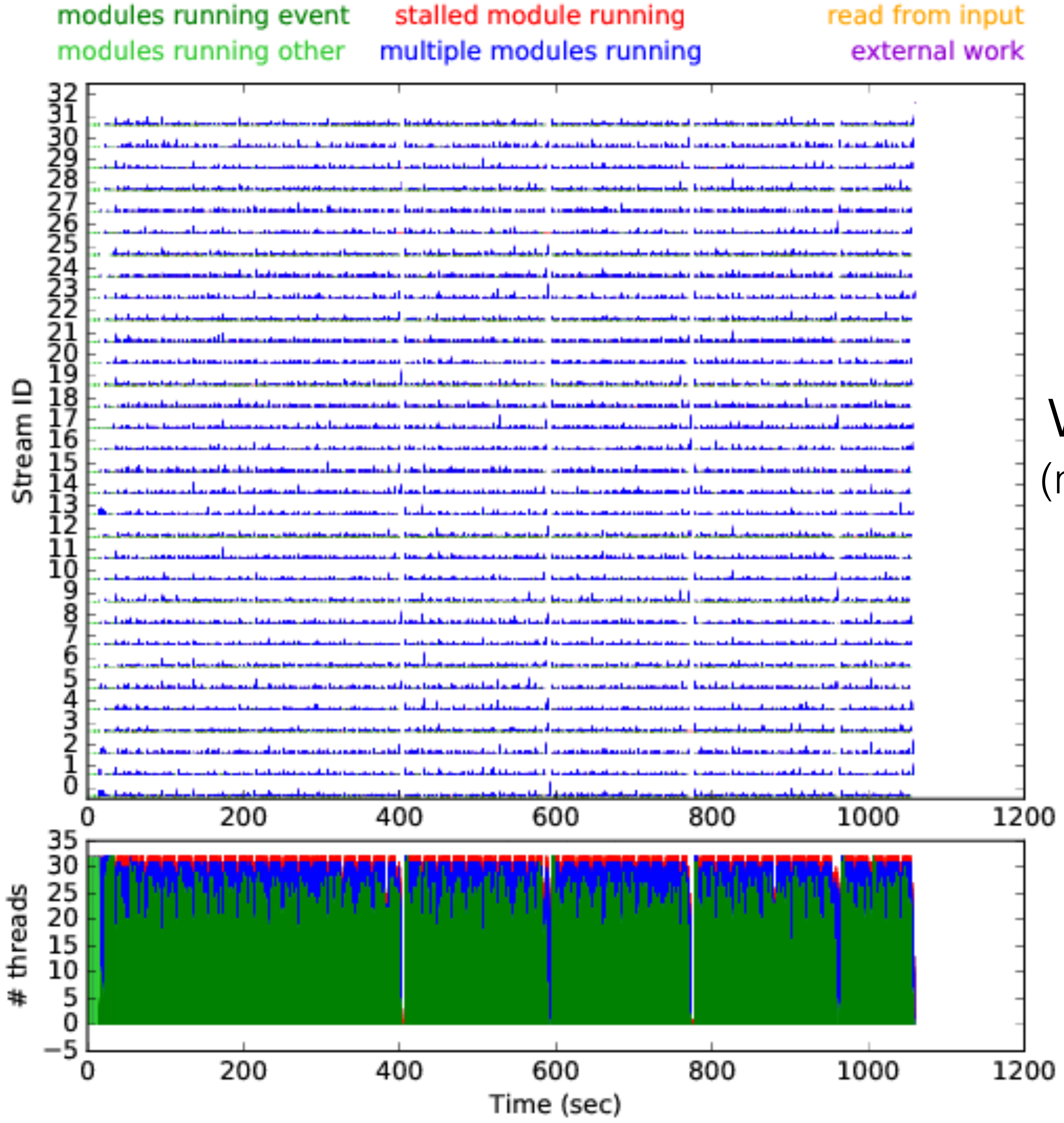
The intent of these configurations is to isolate factors in the performance

- NoWrite skips the merge step—instead of doing the TBufferMergerFile::Write() to queue to the merge, it just does the ResetAfterMerge()
  - Trees and branches are still filled, so this separates the cost of filling from the merge step
- NoFill skips TTree::Fill(). With this set the ParallelPoolOutputModule does some bookkeeping operations and updates the metadata, but skips filling the branches
  - This is close to the limiting case where the output module takes no time at all
  - Bookkeeping operations are non-blocking, so should be no (or little) lock contention

Standard output, no IMT vs w/IMT

No IMT

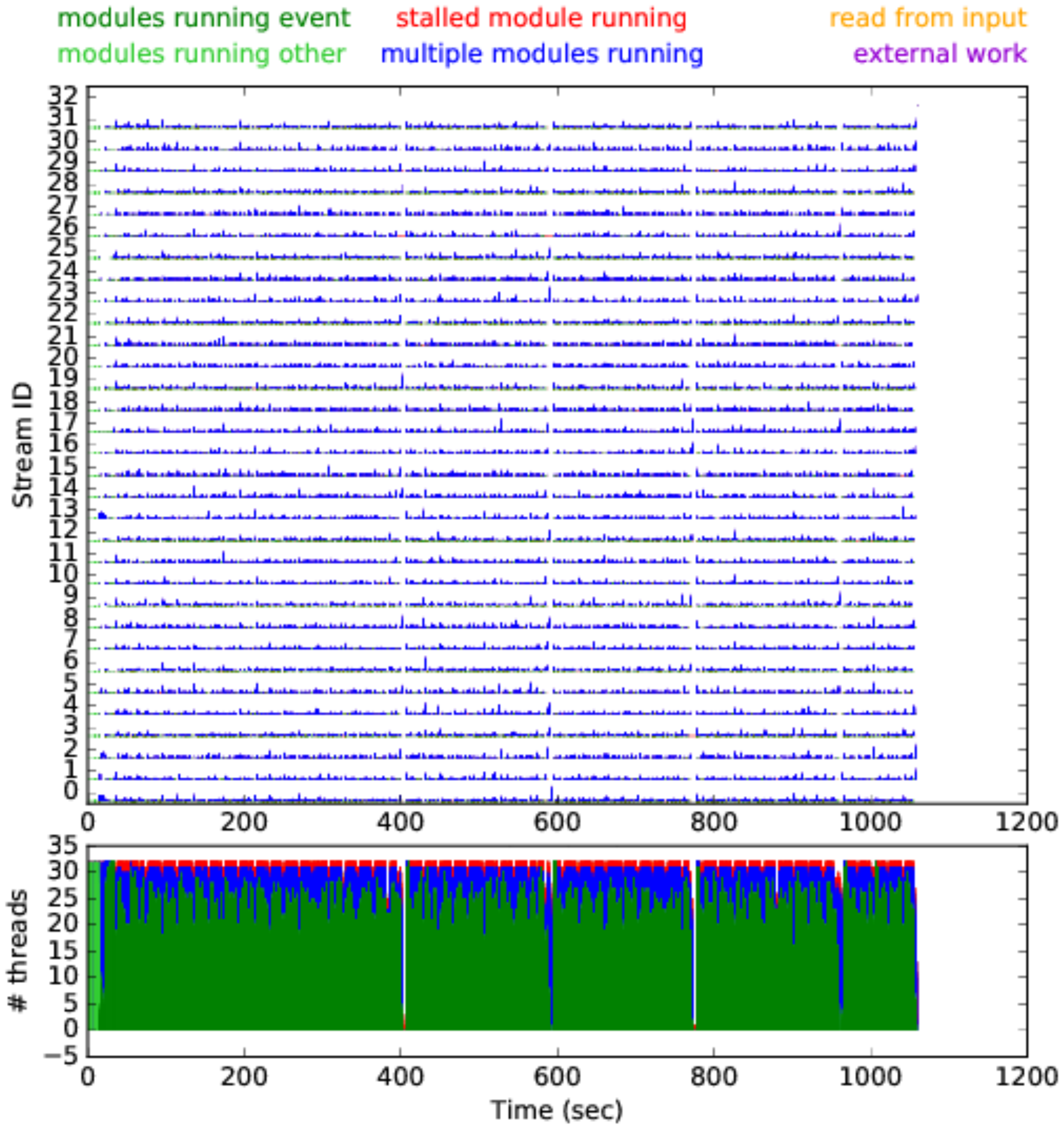


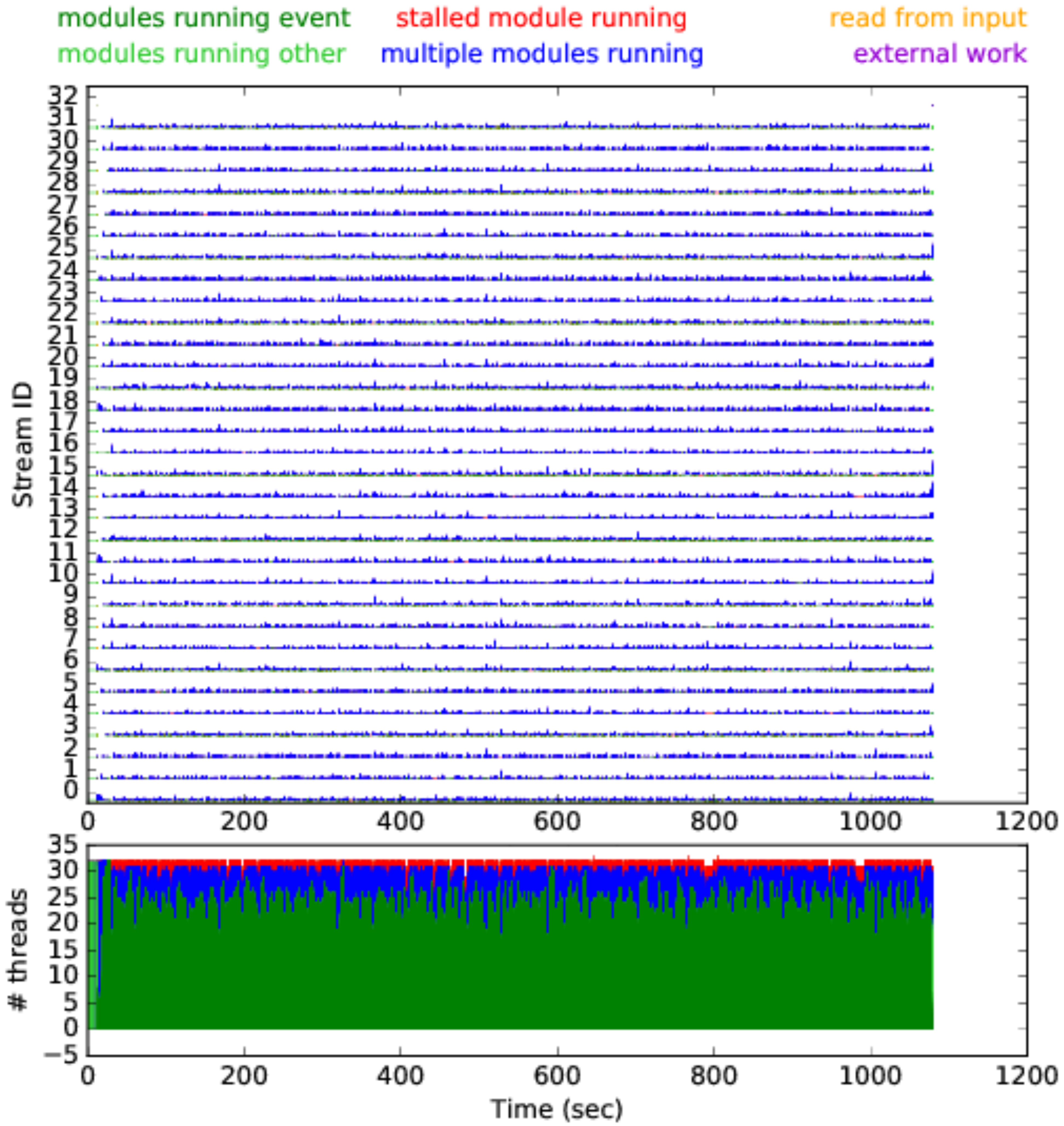


w/IMT
(note scale change)

Standard output vs. parallel merger (both w/IMT)

IMT





Parallel Merger (1x6x3)

32 thread RECO-AOD-MINIAOD

Module	Total Loop Time	Total Loop CPU	CPU Utilization	Events/ Second	RSS
Standard w/o IMT	1701	33989	0.62	2.94	9454
Standard w/IMT	1187	32076	0.84	4.21	8981
Parallel 6x6x3	1119	33722	0.92	4.47	13817
Parallel 1x6x3	1088	33396	0.95	4.59	10745
NoWrite	1075	33116	0.96	4.65	12140
NoFill	924	26987	0.91	5.41	7201

Understanding the RECO Anomaly

To get a handle on why 1x6x3 does better than 6x6x3, look at the PPOM concurrency distribution

- Histogram the output module concurrency level on every event write
- PPOM 6x6x3:
  - AOD and MINIAOD rarely use their full concurrency limits
  - RECO uses full concurrency much more frequently
- NoWrite 6x6x3:
  - Concurrency histograms are slightly lower, but very similar to 6x6x3 with write/merge ops
  - TBufferMerger write and merge operations are likely not the reason RECO does worse
- NoFill:
  - Concurrency is never greater than 1, so no contention
- Speculation: contention is primarily in TTree::Fill()
  - Main source of observed lock contention is TBranchElement::SetAddress() (CMS changes the object pointer every event)
  - RECO has lots of branches, spends relatively less time/byte in compression, flushes frequently
  - RECO gives IMT lots of tasks, while parallel output module leads to more contention

Conclusions

Progress:

- IMT is a clear win for CMS
  - Does more on some data tiers than others
- Reducing mutex contention and other improvements have helped improve scaling for the parallel output module
  - Could be even better if the TBranchElement::SetAddress() mutex could be eliminated (previously identified, see ROOT-9253)
- The combination of IMT and the parallel output module does better than either alone
  - TBB task isolation was essential for eliminating interaction anomalies
  - Combined these can dramatically improve output scaling for most (all?) CMS data tiers
  - But finding the right combination isn't fully understood

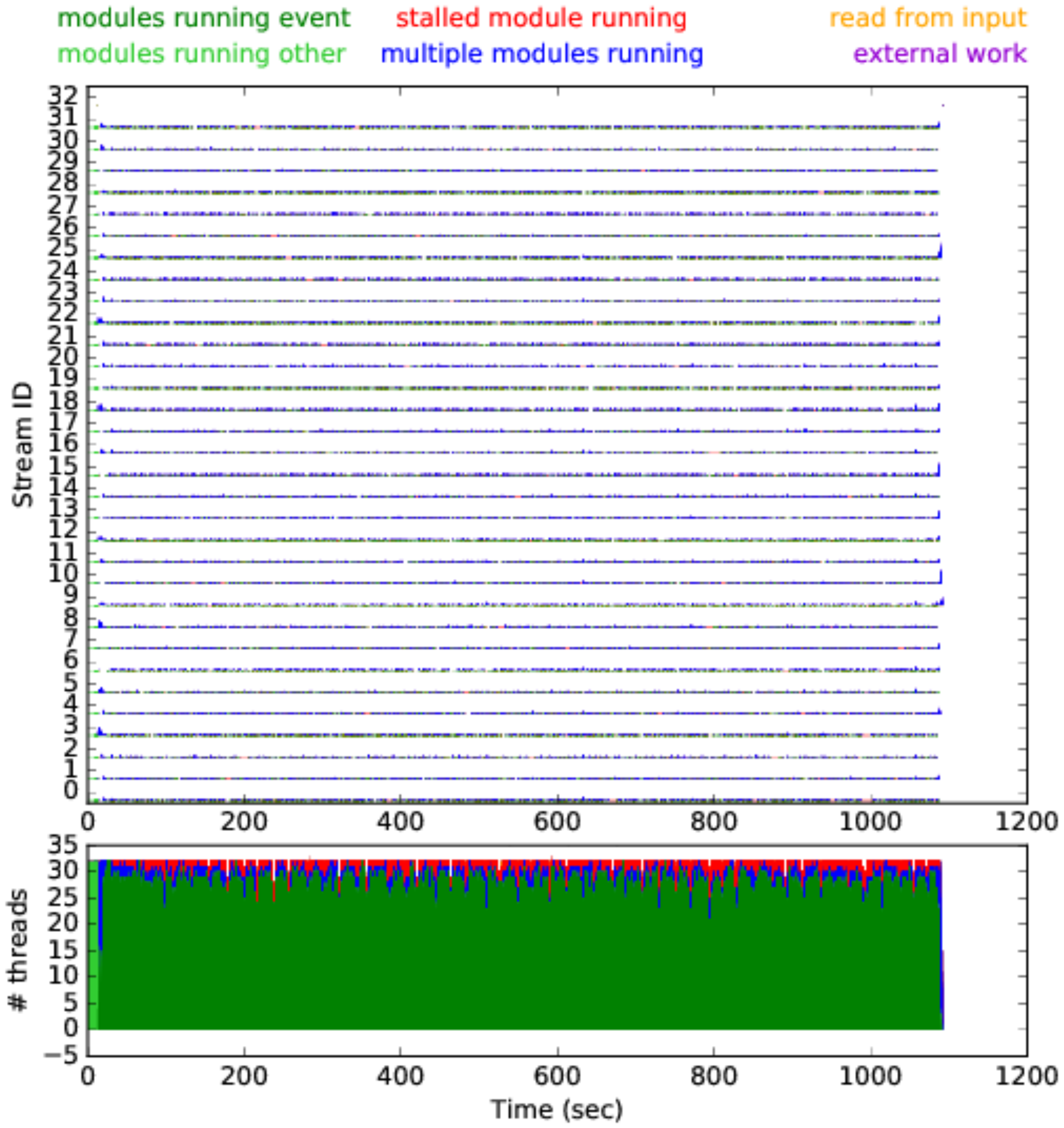
Todo:

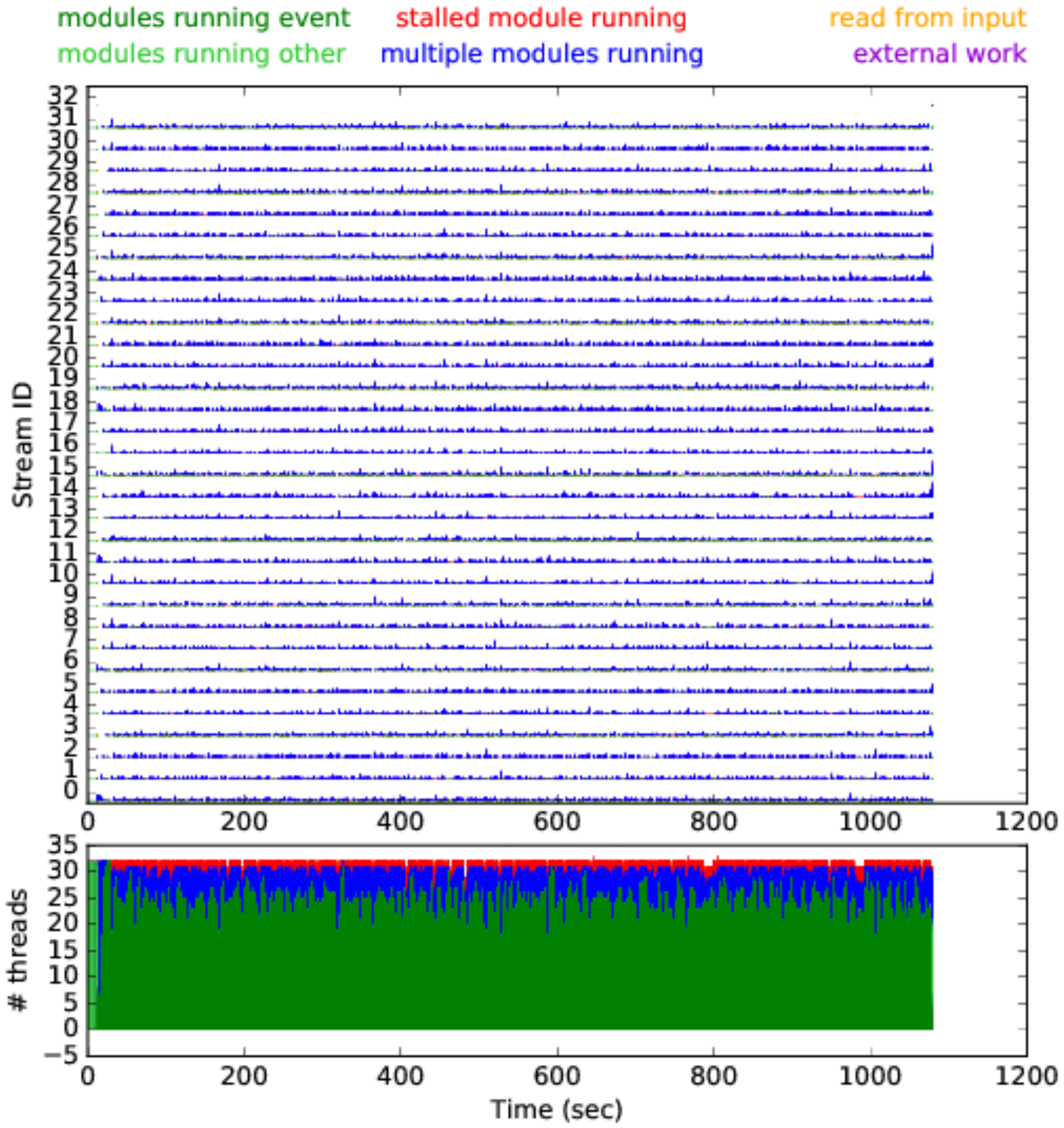
- Finish loose ends in the parallel output module implementation (mostly metadata)
- Work on more fully characterizing (and automating) the best configuration for a job

BACKUP SLIDES

6x6x3 vs 1x6x3

Parallel Merger (6x6x3)





Parallel Merger (1x6x3)