In Search of *Sweet-Spots* in Parallel Performance Monitoring

Aroon Nataraj, Allen D. Malony, Alan Morris

University of Oregon

{anataraj,malony,amorris}@cs.uoregon.edu

Dorian C. Arnold, Barton P. Miller

University of Wisconsin, Madison

dorian.arnold@gmail.com

bart@cs.wisc.edu

Motivation

- Performance problem analysis increasingly complex
  - Multi-core, heterogeneous, and extreme scale computing
- Shift of performance measurement and analysis perspective
  - Static, offline analysis ⇨ dynamic, online analysis
  - Support for *performance monitoring* (measurement + query)
  - Enabling of adaptive applications with performance feedback
- Prerequisites for performance measurement
  - Low overhead and low intrusion
  - Runtime analysis antithetical to performance tool orthodoxy
- Neo-performance perspective
  - Co-allocation of additional (tool specific) system resources
  - Make dynamic, performance-driven optimization viable

Monitoring for Performance Dynamics

- Runtime access to parallel performance data
  - Scalable and lightweight
  - Support for performance-adaptive, dynamic applications
- Raises vital concerns of overhead and intrusion
  - Bigger issue in online systems due to global effects
- *Alternative 1*: Extend existing performance measurement
  - Create own integrated monitoring infrastructure
  - Disadvantage: maintain own monitoring framework
- *Alternative 2*: Couple with other monitoring infrastructure
  - Leverage scalable middleware from other supported projects
  - Challenge: measurement system / monitor integration
  - ***TAU over Supemon*** (ToS) (UO, LANL)
  - ***TAU over MRNet*** (ToM) (UO, University of Wisconsin)

Performance Monitoring: Contradictory Goals

- Application semantics dictate monitoring scheme
  - Performance across iterations / phases
- Requirements determining a performance monitoring scheme
  - Performance events to monitor
  - Access frequency
  - Data analysis operation (reduction type)
- Performance monitoring costs
  - Intrusion to application
  - Extra monitoring resource allocation (# processors)
- Opposing goals (leads to trade-offs)
  - Acceptable performance data granularity (temporal / spatial)
  - Acceptable level of application intrusion from offloads
  - Acceptable monitoring resource requirements

Scalable Low-Overhead Performance Monitoring

- Key is to match ...
  - Application monitoring demands with ...
  - Effective operating range of monitoring infrastructure
- Over-provisioning (more monitor resources assigned)
  - Leads to wasted resources and lost performance potential
- Under-provisioning (less monitor resources assigned)
  - Poor scalability, high overhead, low performance data quality
- Not simply a question of # processors - but *configuration*
  - Transport topology
  - Transport-level reduction operations
- Try to find the monitoring ***Sweet-Spot***
  - "Area on a bat or racket where it makes most effective contact with the ball" (New Oxford English Dictionary)

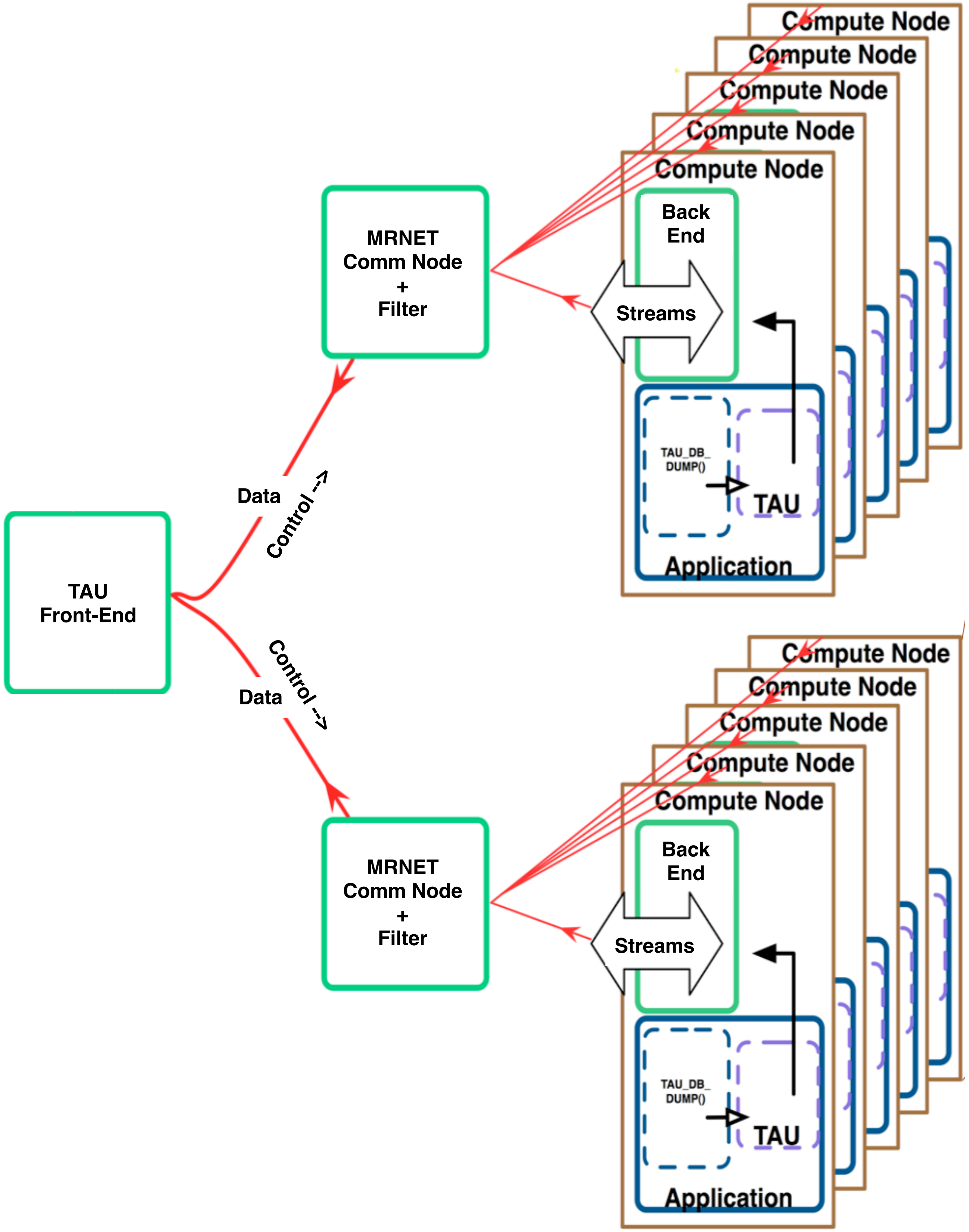
Talk Outline

- Motivation
- Monitoring for performance dynamics
- Contradictory goals of performance monitoring
- Key to scalable low-overhead performance monitoring
- TAUoverMRNet (*ToM*)
- Naive monitoring choices and consequences
- Estimating the bottleneck
- Characterization and finding *sweet-spots*
- Future plans and conclusion

What is MRNet?

- ***M***ulticast ***R***eduction ***Net***work (University of Wisconsin)
  - Software infrastructure, API, utilities (written in C++)
  - Create and manage network overlay trees (TBON model)
  - Efficient control through root-to-leaf multicast path
  - Reductions (transformations) on leaf-to-root data path
  - Packed binary data representation
- Uses *thread-per-connection* model
  - Supports multiple concurrent "streams"
- Filters on intermediate nodes (upstream, downstream)
  - Default filters (e.g., sum, average)
  - Loads custom filters through shared-object interface
- MRNet-base tools (Paradyn, STAT debugger, ToM)

TAUoverMRNet (ToM)

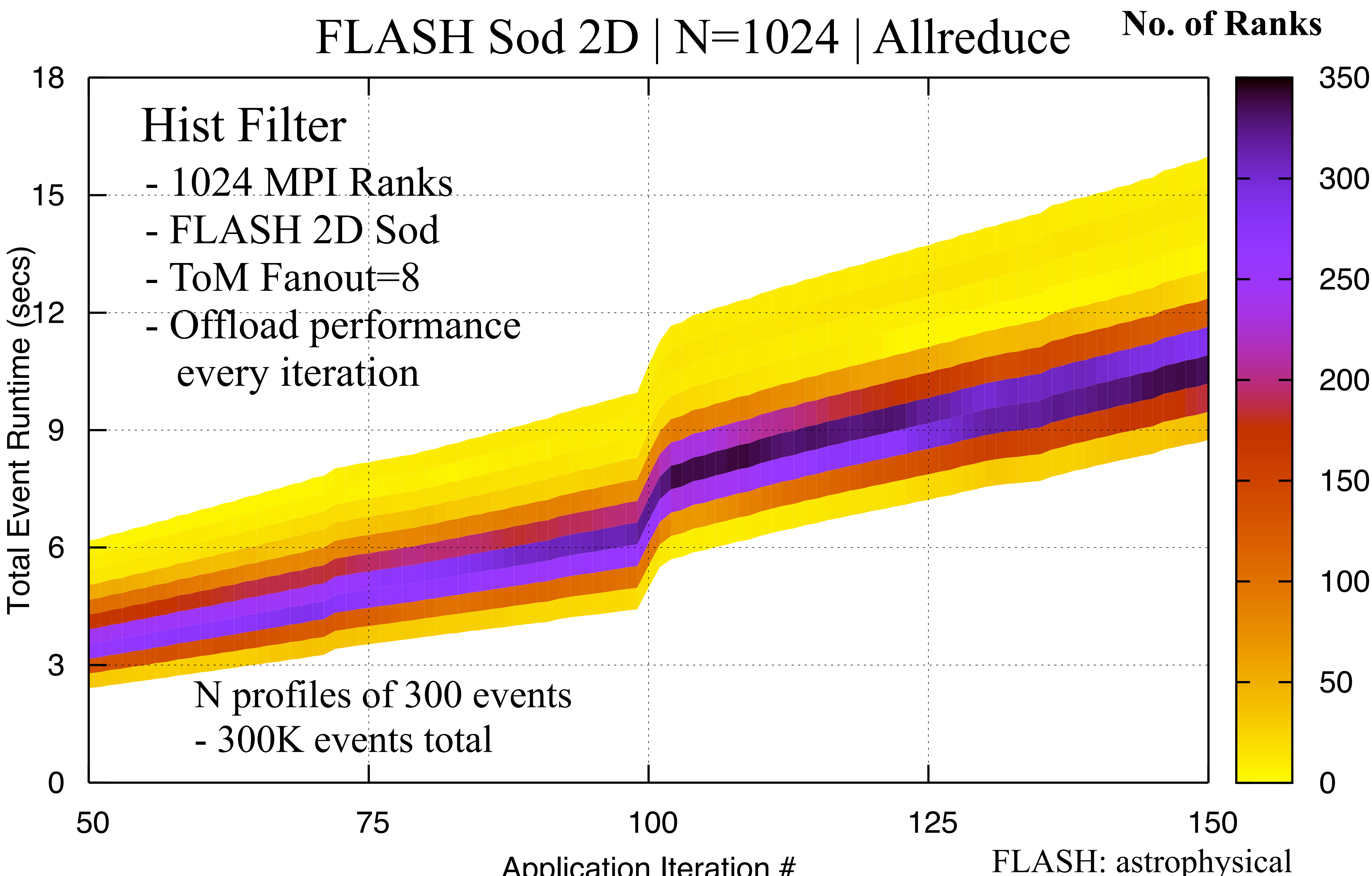


- Back-End (BE) TAU adapter offloads performance data
- Filters
  - reduction
  - distributed analysis
  - upstream / downstream
- Front-End (FE) unpacks, interprets, stores
- Paths
  - reverse data reduction path
  - multicast control path
- Push-Pull model
  - source pushes, sink pulls

ToM Filters

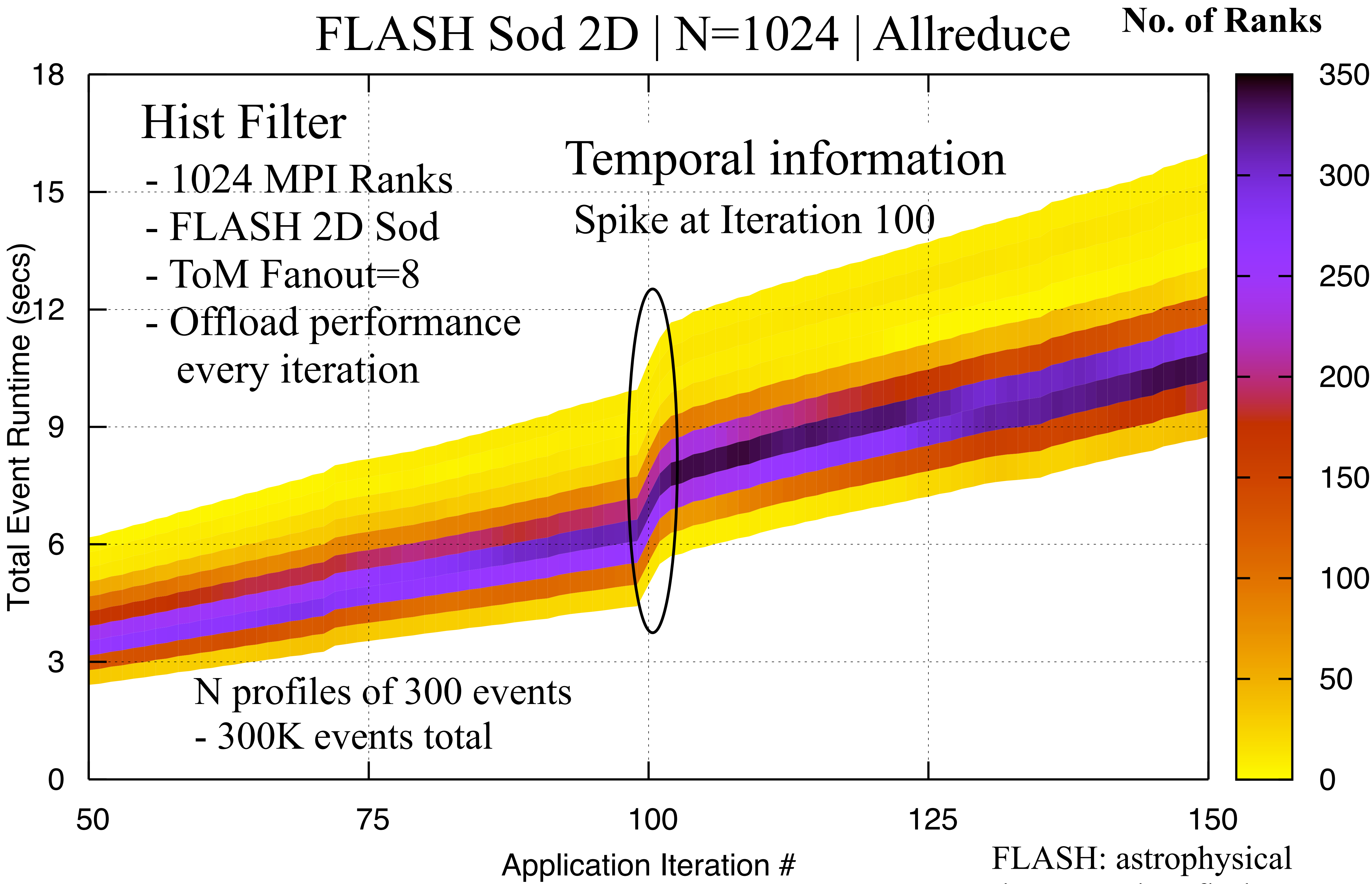
- Ideally there would be no need for filtering
  - Retrieve and store *all* performance data provided
  - Acceptability depends on performance monitor use
- High application intrusion, transport and storage costs
  - Need to trade-off queried performance data granularity
  - Which events, time intervals, application ranks?
- Reduce performance data as it flows through transport
  - Distribute FE analysis out to intermediate filters
- Three filtering schemes
  - *1-phase*: summary
  - *3-phase*: histogram, classification histogram
  - Progressive temporal/spatial detail with added complexity

Histogram Filter (FLASH)



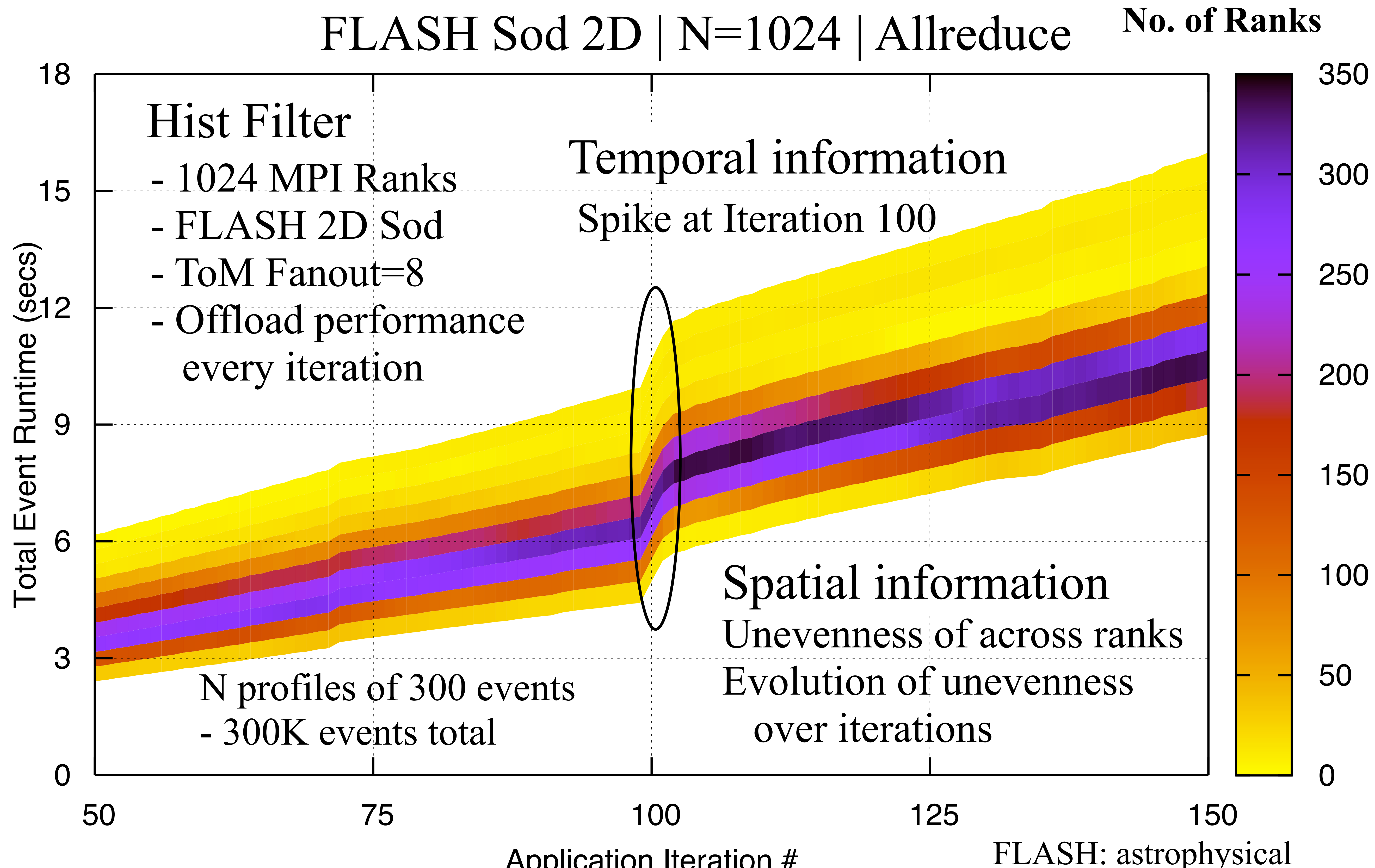
FLASH: astrophysical thermonuclear flashes

Histogram Filter (FLASH)



FLASH: astrophysical thermonuclear flashes

Histogram Filter (FLASH)



FLASH: astrophysical thermonuclear flashes

Evaluating Monitoring Choices

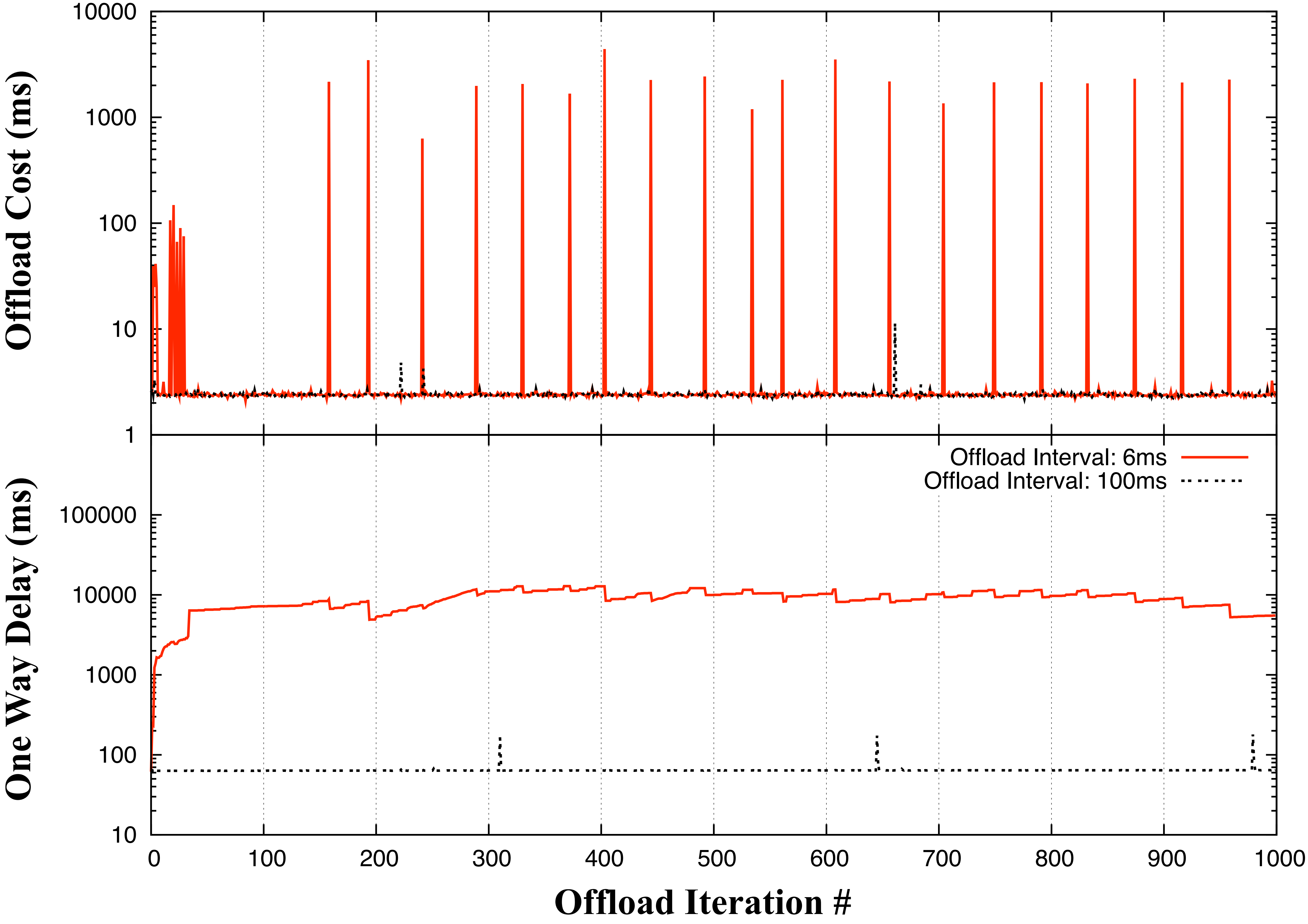
- Simple SPMD workload

  outputs profile to ToM

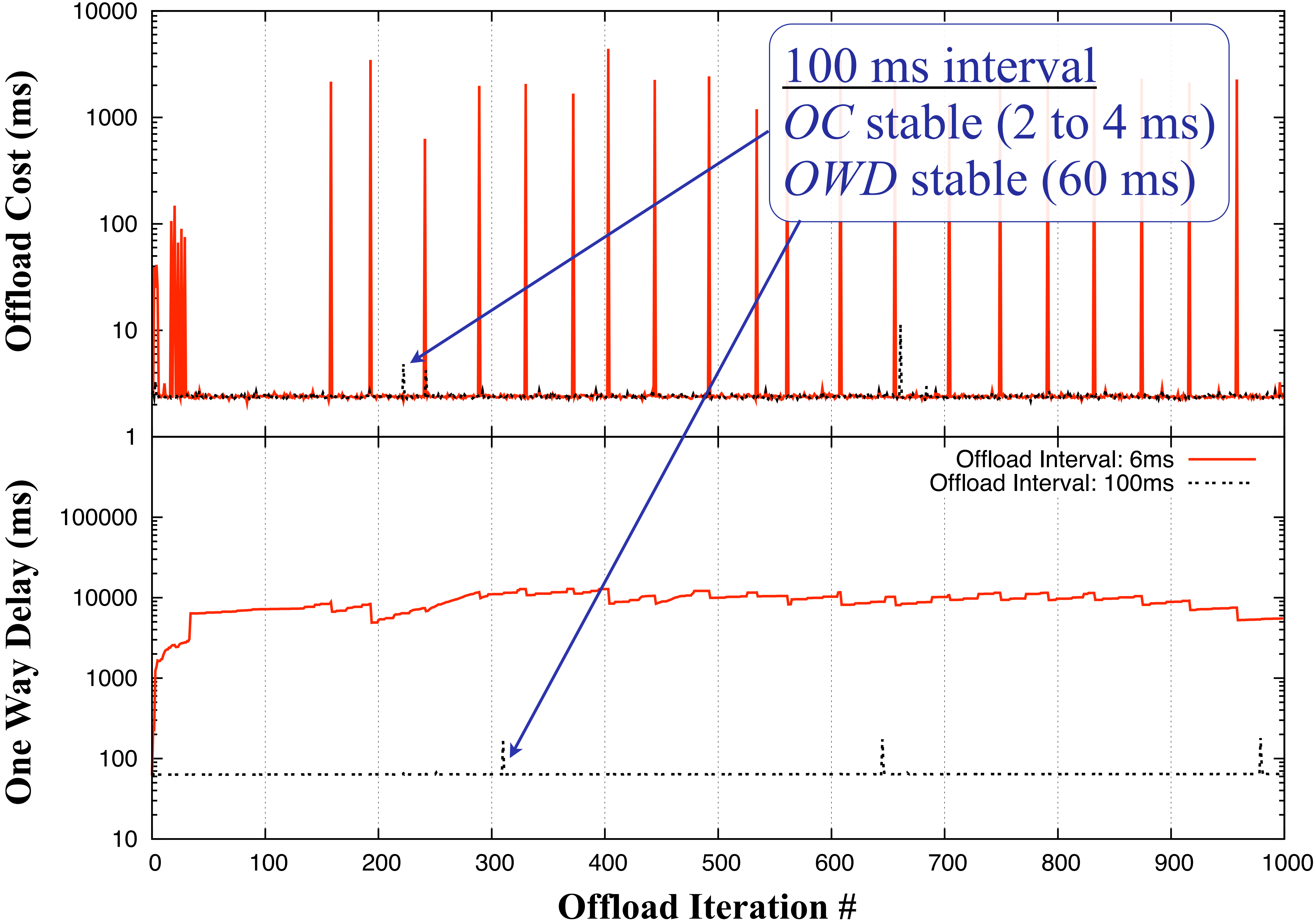
  ```
  for (i=0; i<iterations; i++) {
    work (usecs);   TAU_DB_DUMP();   MPI_Barrier();
  }
  ```

  - Number of profile events fixed (64)
  - Monitoring offload interval (usecs) = 100ms and 6ms
  - # of application ranks = 64, # iterations = 1000
- Simple (1-phase) statistics filter
  - ToM Fanout (FO) = 8   (two-level tree)
- *Offload Cost* (OC) metric
  - Maximum time within offload operation across ranks
- *One Way Delay* (OWD) metric
  - Time difference between sink receive and earliest *BE* send

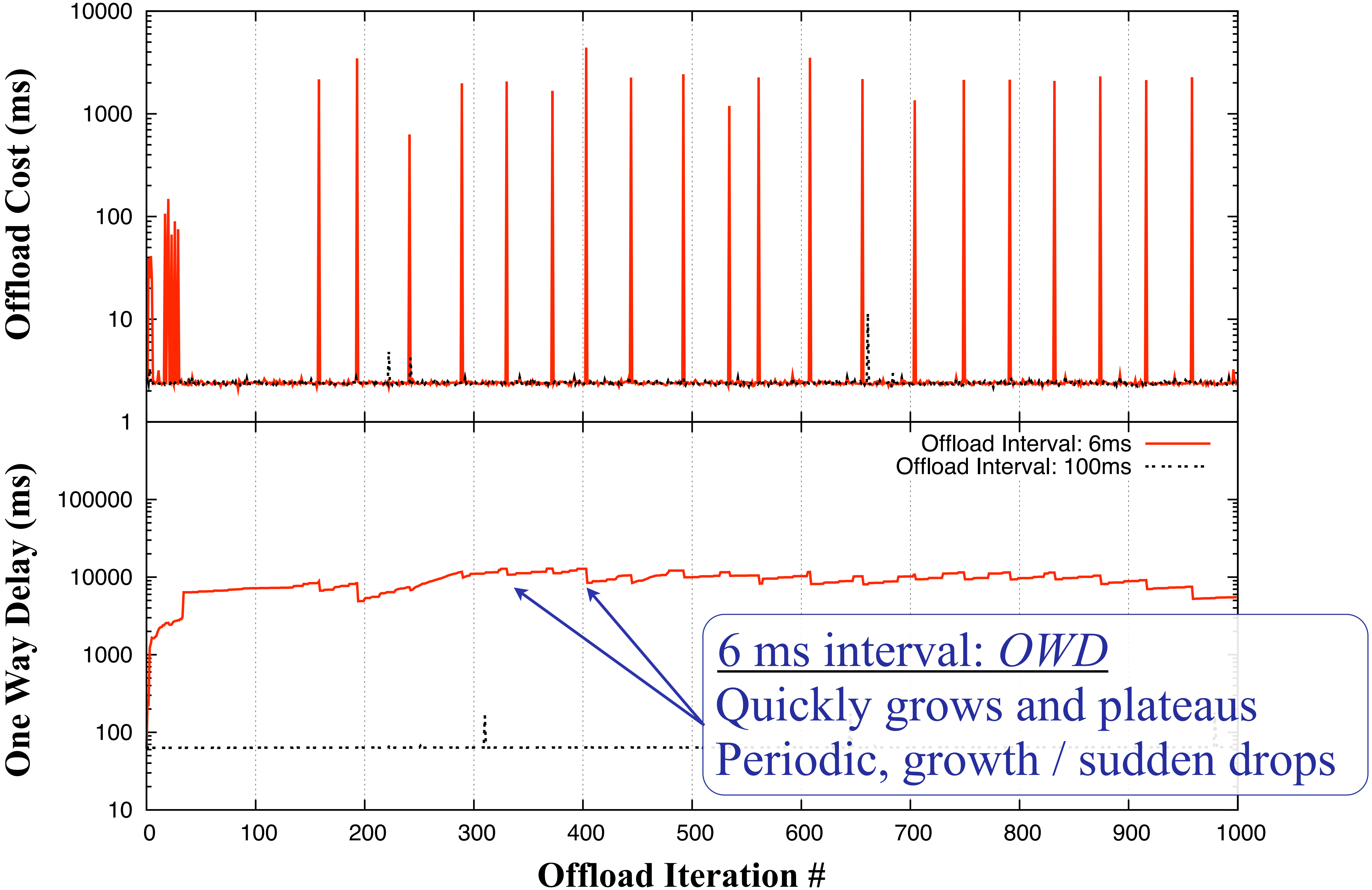
Naive Monitoring : Consequence



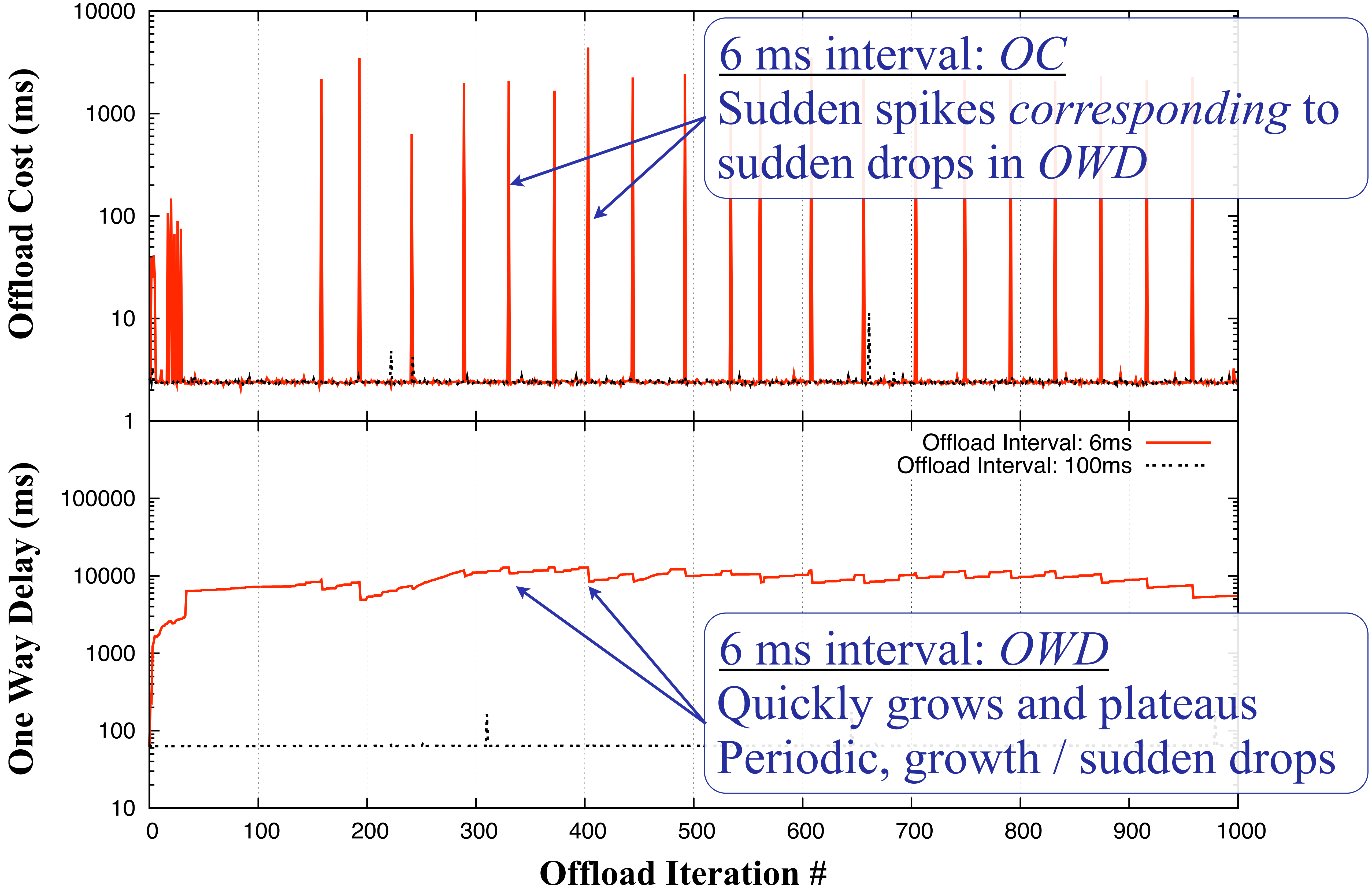
Naive Monitoring : Consequence



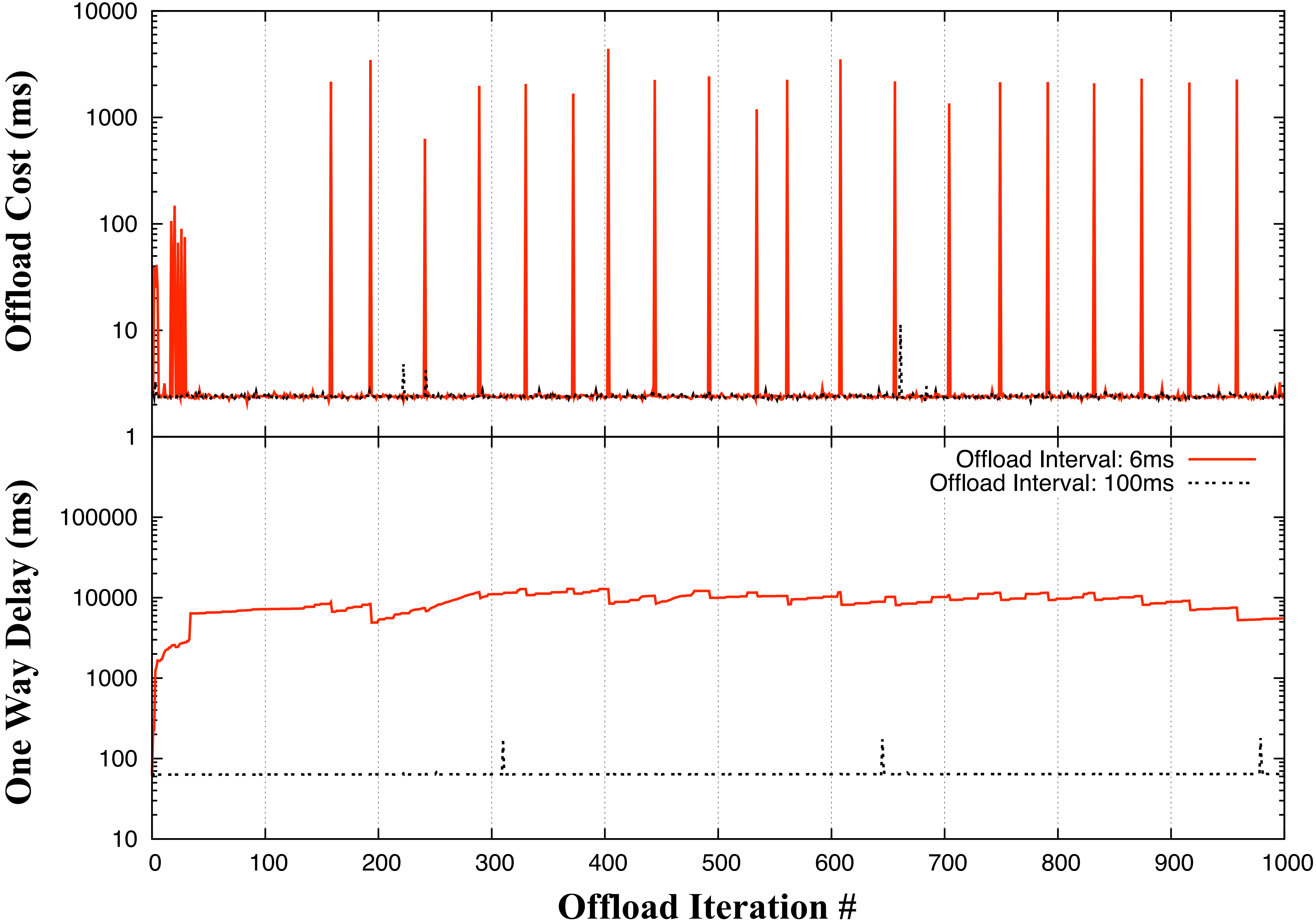
Naive Monitoring : Consequence



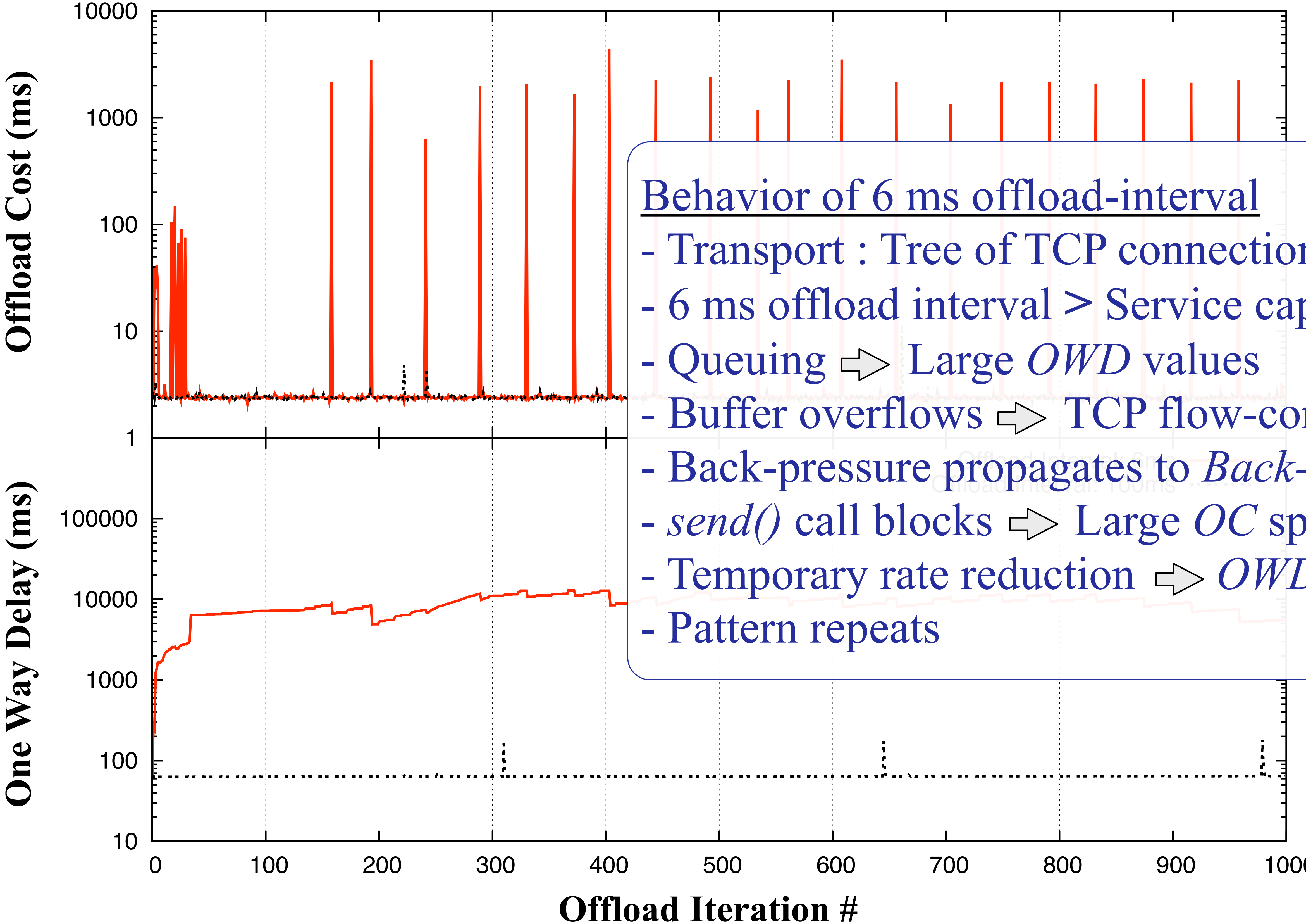
Naive Monitoring : Consequence



Naive Monitoring : Consequence



Naive Monitoring : Consequence



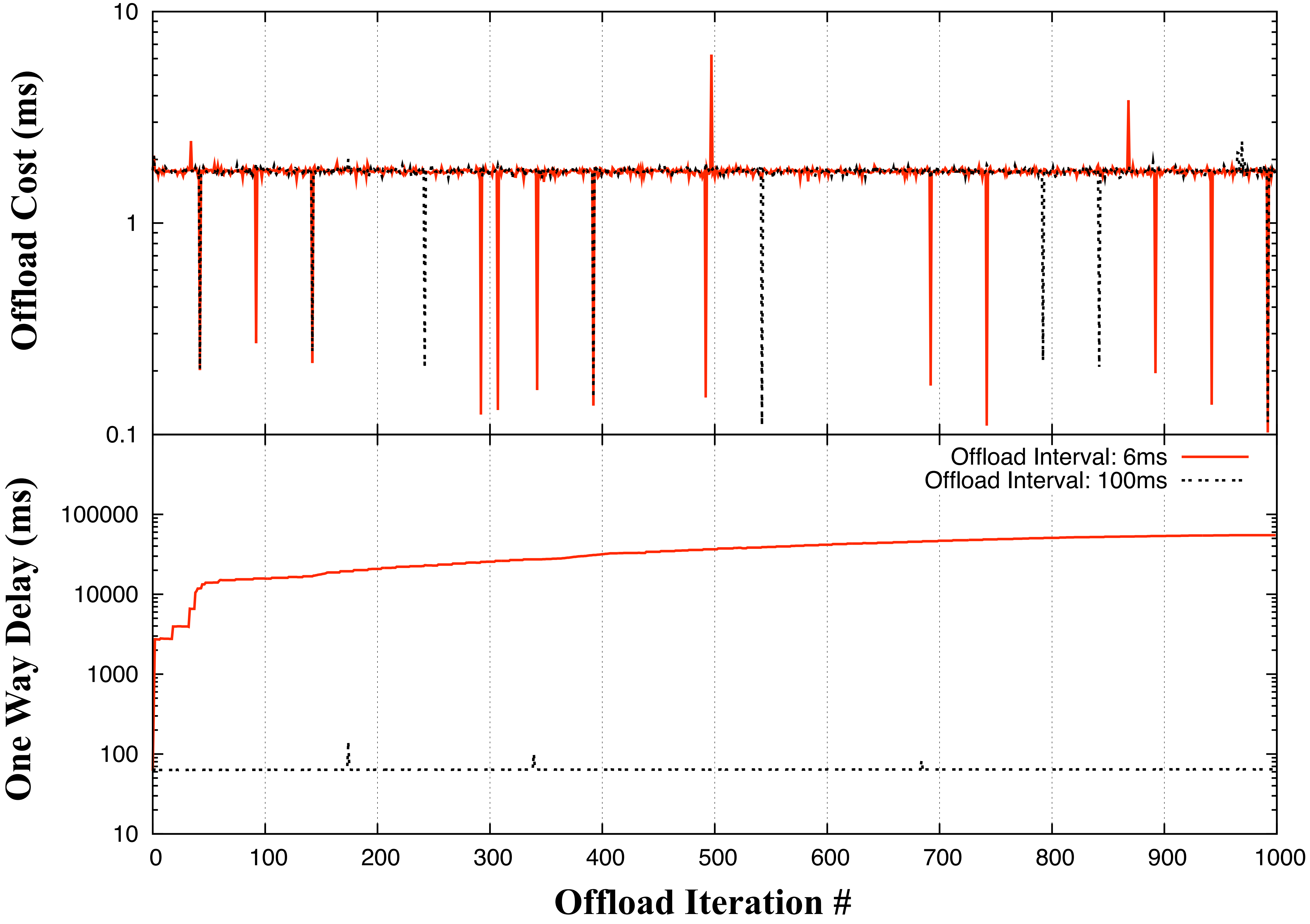
Behavior of 6 ms offload-interval

- Transport : Tree of TCP connections
- 6 ms offload interval > Service capacity
- Queuing ⇨ Large *OWD* values
- Buffer overflows ⇨ TCP flow-control
- Back-pressure propagates to *Back-End*
- *send()* call blocks ⇨ Large *OC* spike
- Temporary rate reduction ⇨ *OWD* drop
- Pattern repeats

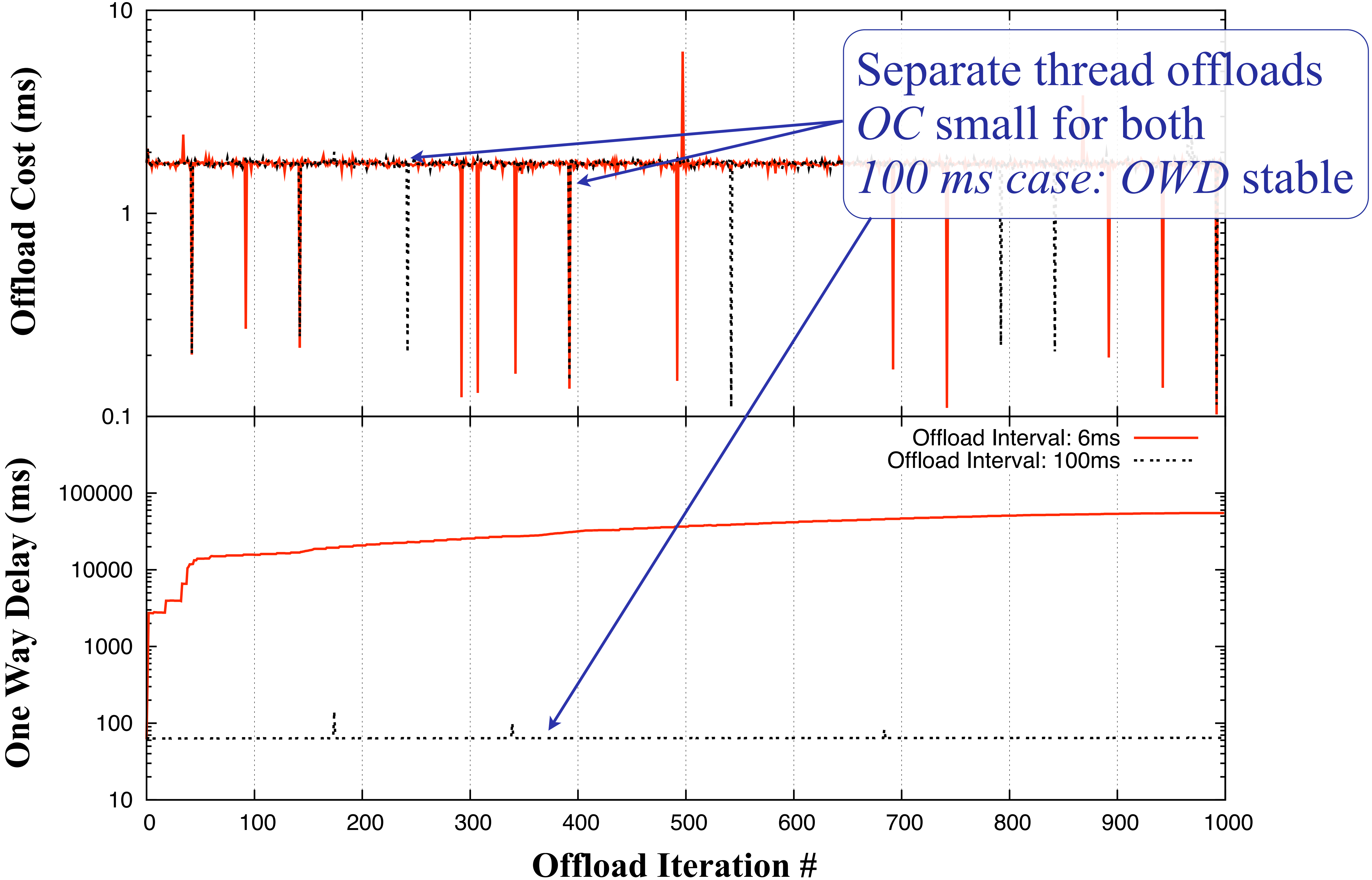
Non-Blocking Monitoring Scheme

- Impact to application directly from blocking *send()*
  - Large *OC* spikes
  - 779.4% overhead due to large blocking time
- Will simple non-blocking approach help?
- Decouple application from offloading
  - Separate consumer thread performs actual offload
  - Application places profile into shared, unbounded buffer
  - Latency hiding
- Repeat same experiments, with non-blocking scheme

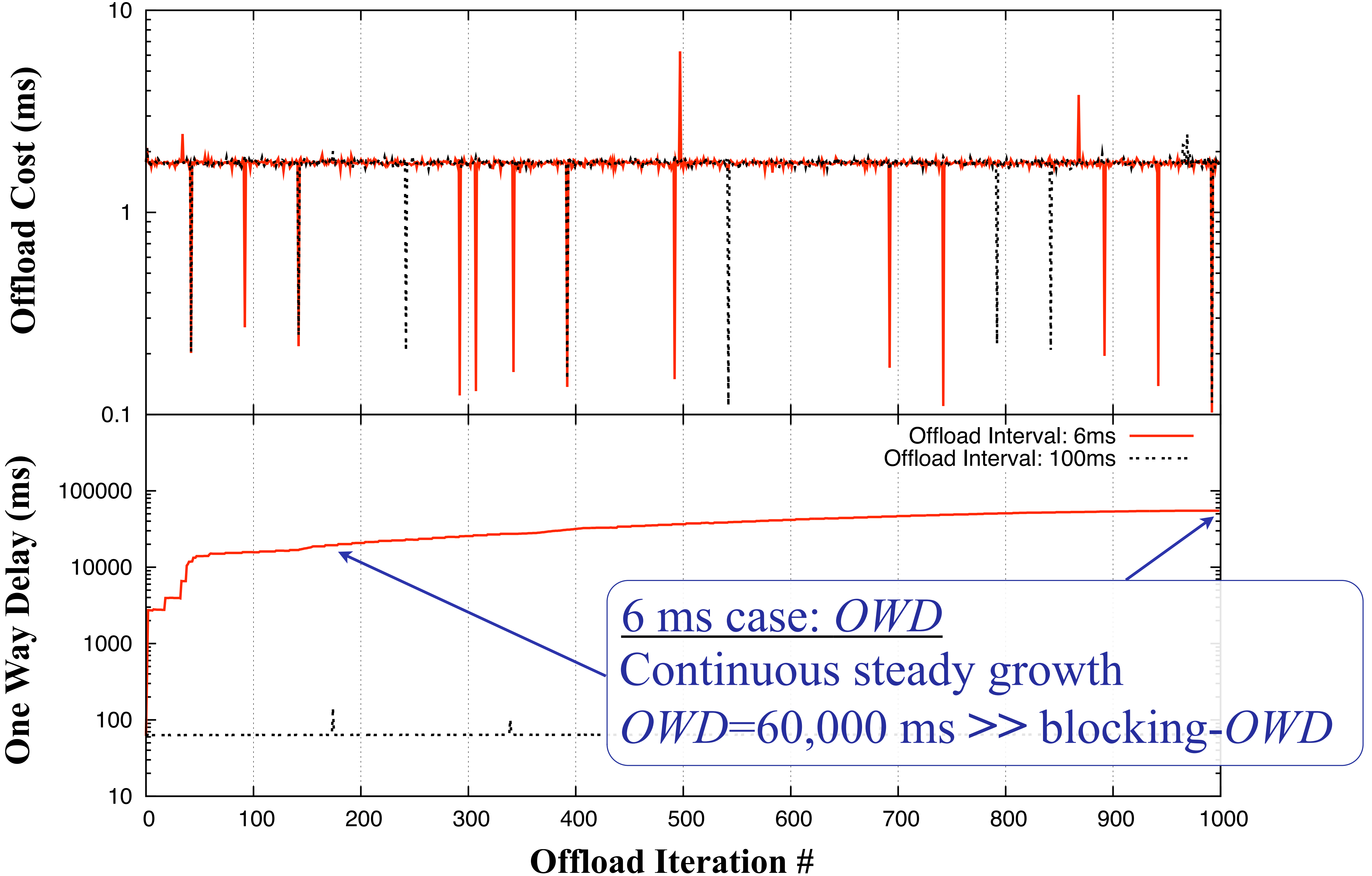
Non-Blocking Monitoring : Consequence



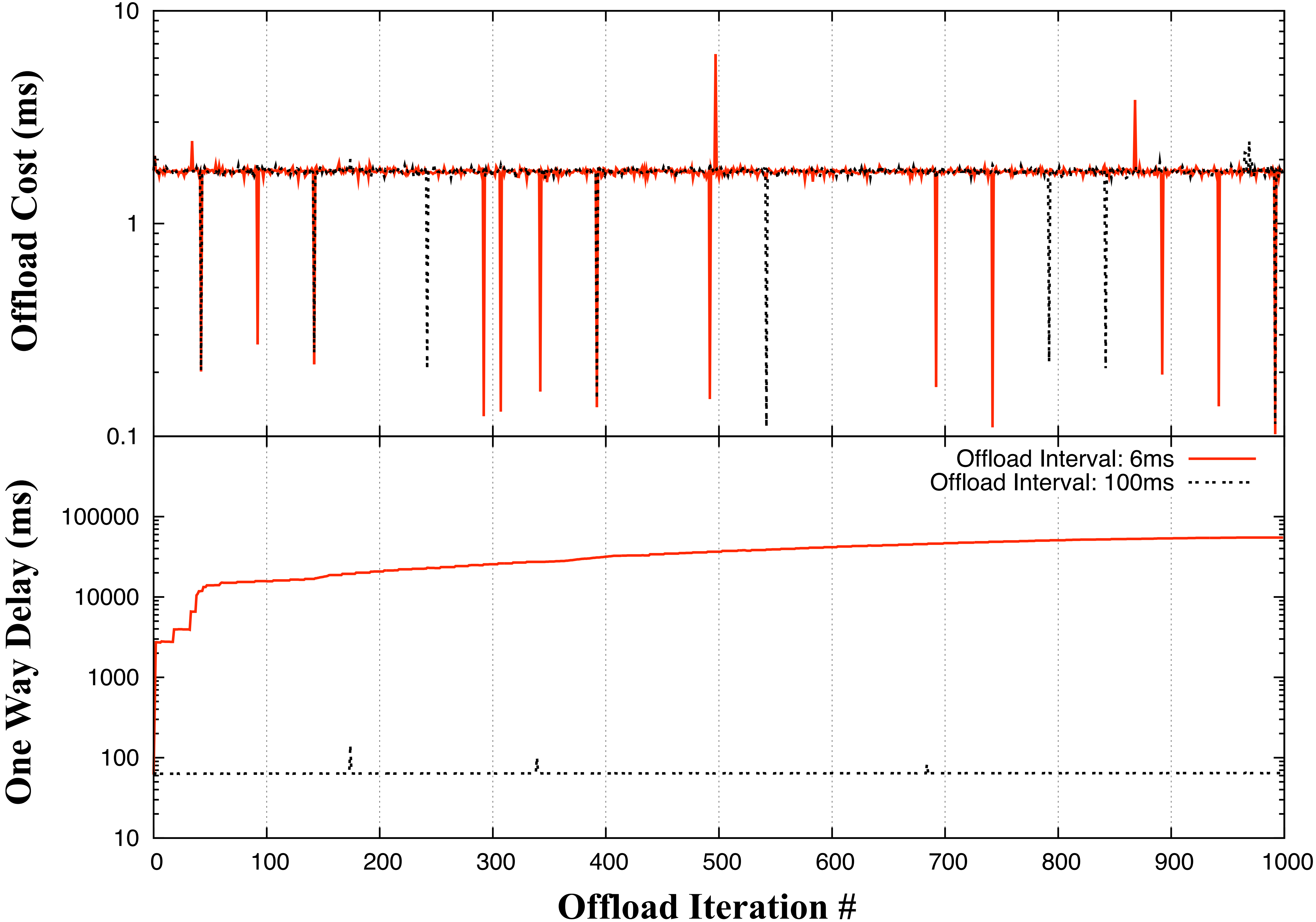
Non-Blocking Monitoring : Consequence



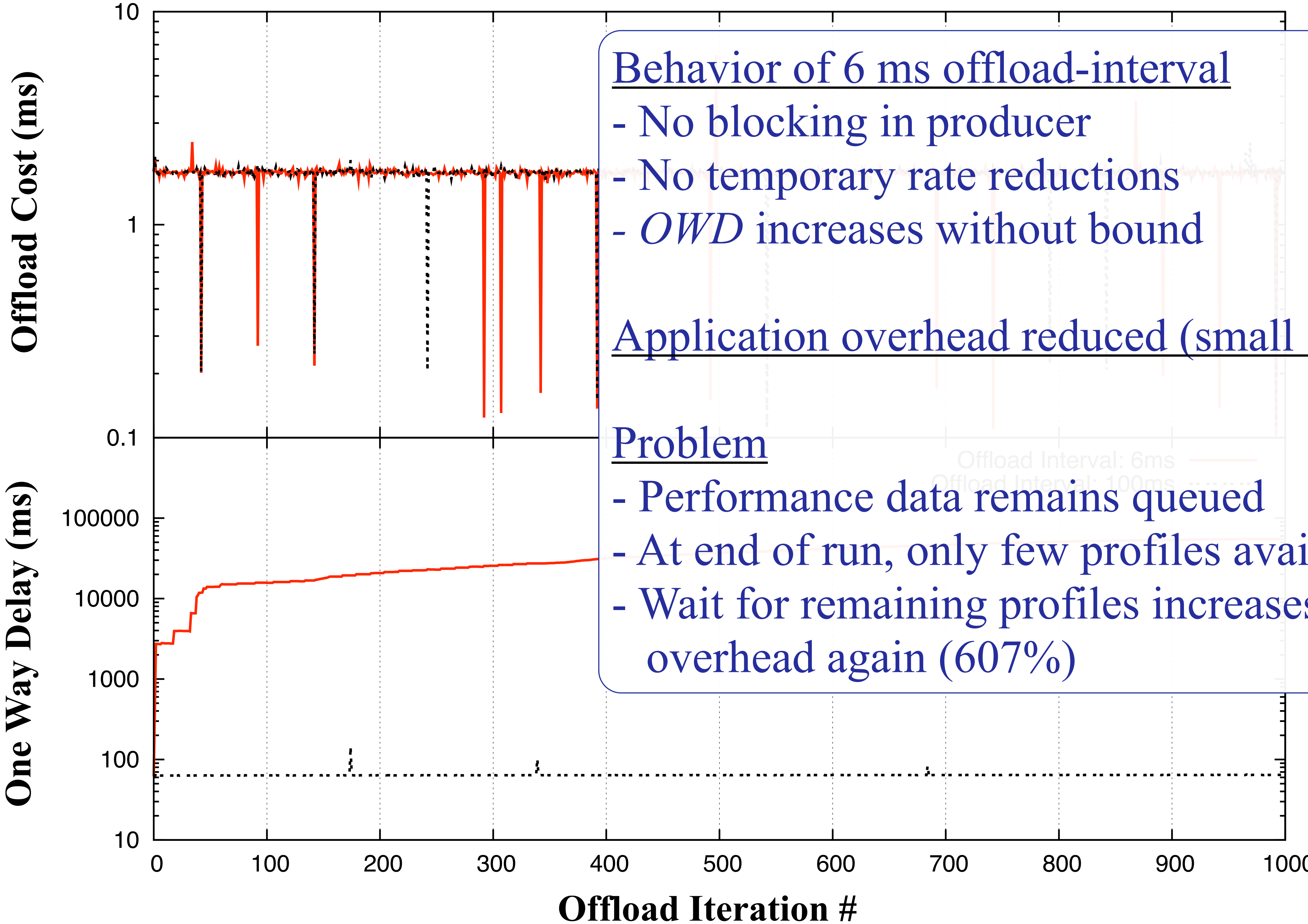
Non-Blocking Monitoring : Consequence



Non-Blocking Monitoring : Consequence



Non-Blocking Monitoring : Consequence



Behavior of 6 ms offload-interval

- No blocking in producer
- No temporary rate reductions
- *OWD* increases without bound

Application overhead reduced (small *OC*)

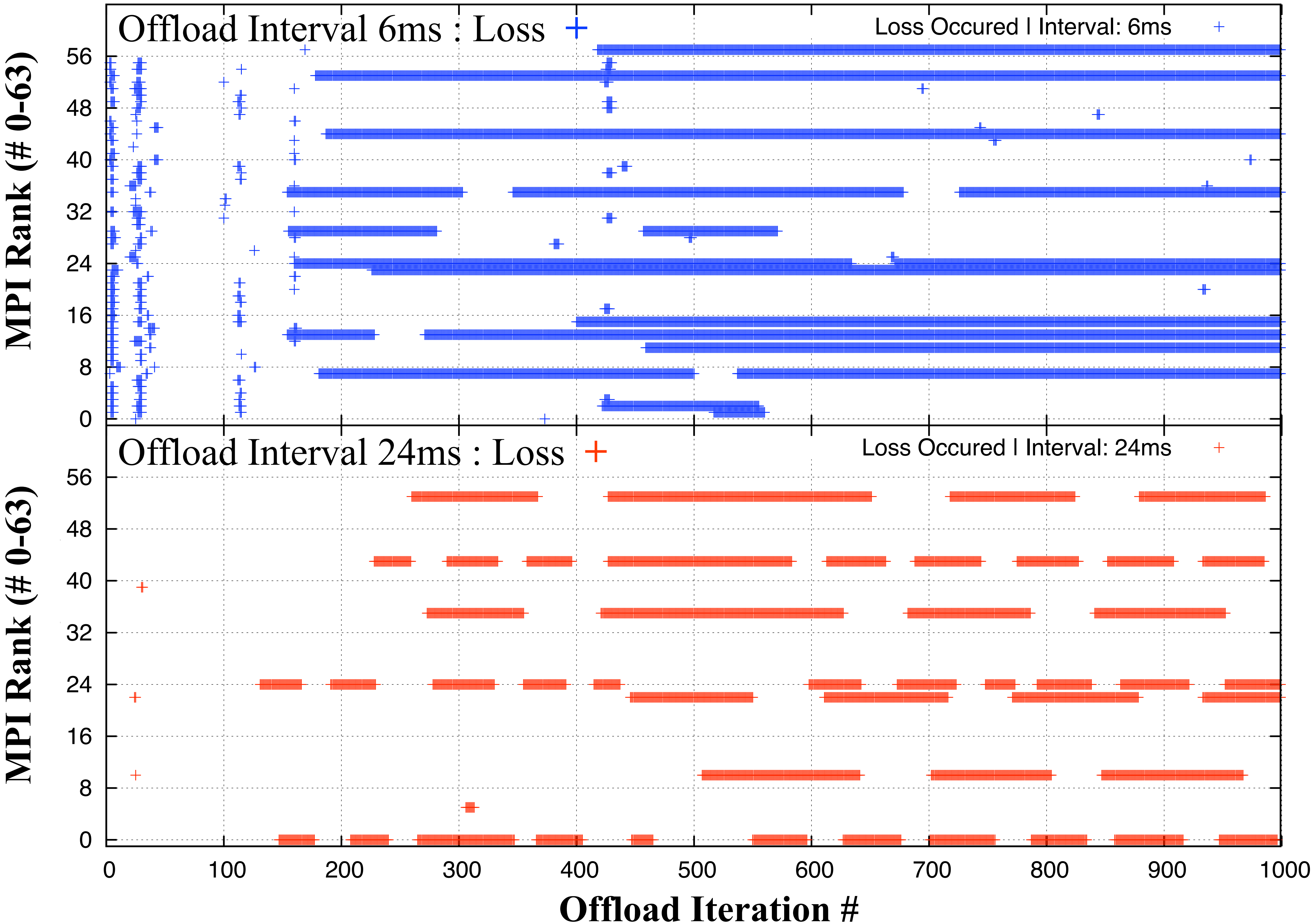
Problem

- Performance data remains queued
- At end of run, only few profiles available
- Wait for remaining profiles increases overhead again (607%)

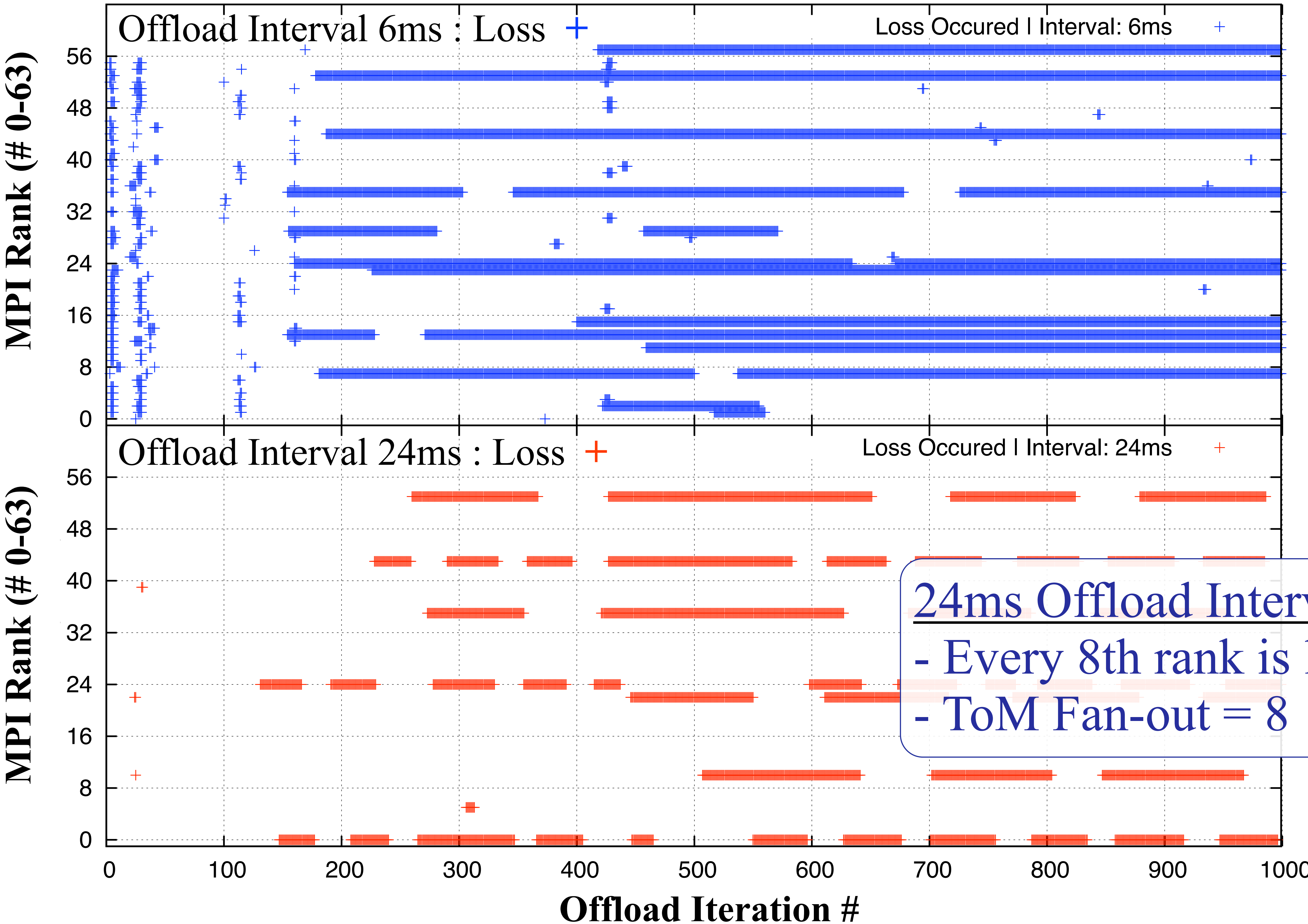
Non-Blocking and Lossy Monitoring Scheme

- Previous schemes do not *reduce number of offloads*
  - Non-blocking simply delays the problem
- Application can detect problems
  - *Blocking case*: Locally detect spike in *OC*
  - *Non-blocking case*: Locally detect full buffer
- Application can do something
  - *Locally back-off*
  - Drop the current profile instead of offloading
- Lossy, non-blocking scheme
  - With bounded buffers
  - With local back-off

Lossy, Non-Blocking : Loss Map

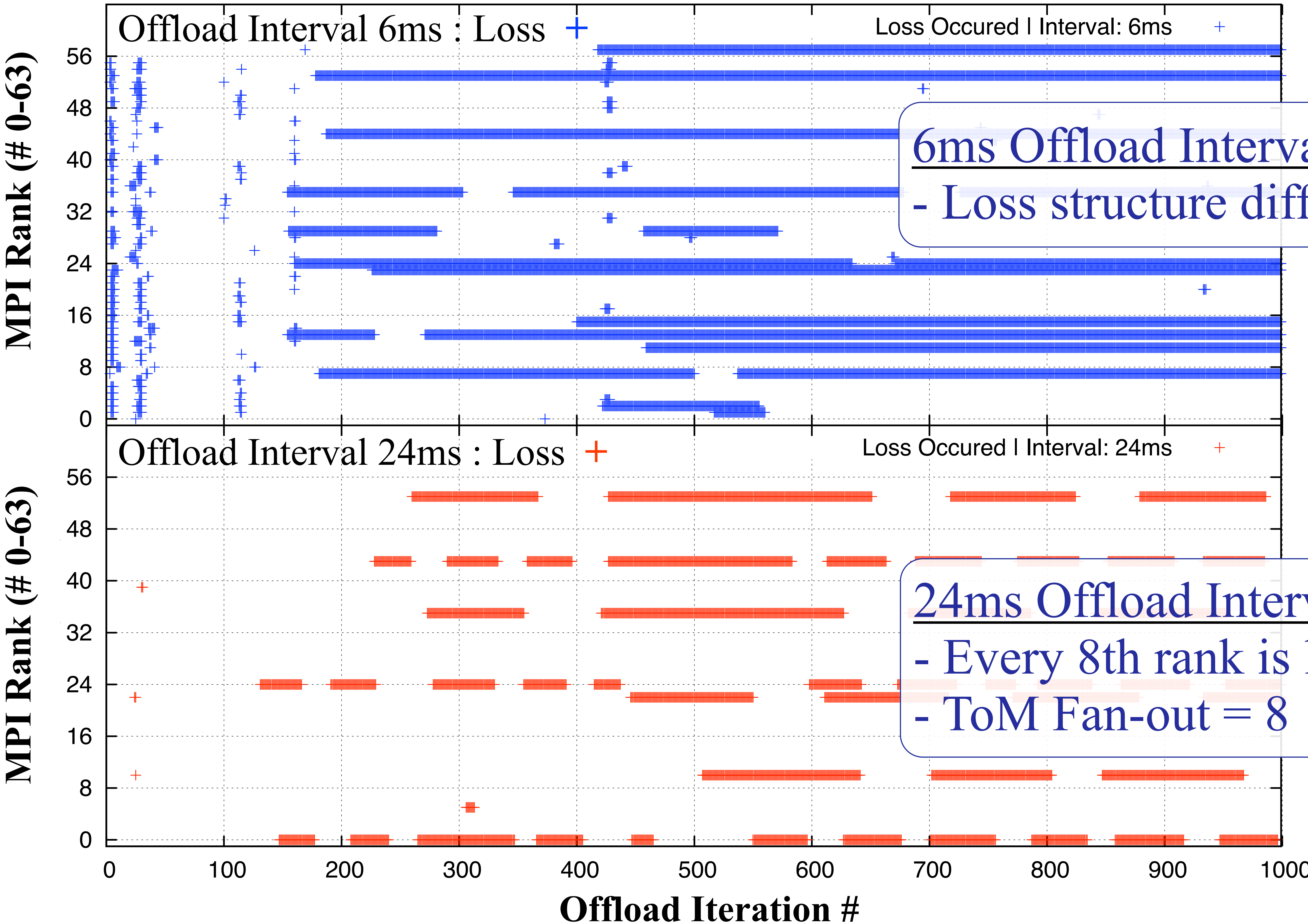


Lossy, Non-Blocking : Loss Map



<u>24ms Offload Interval</u>
- Every 8th rank is lossy
- ToM Fan-out = 8

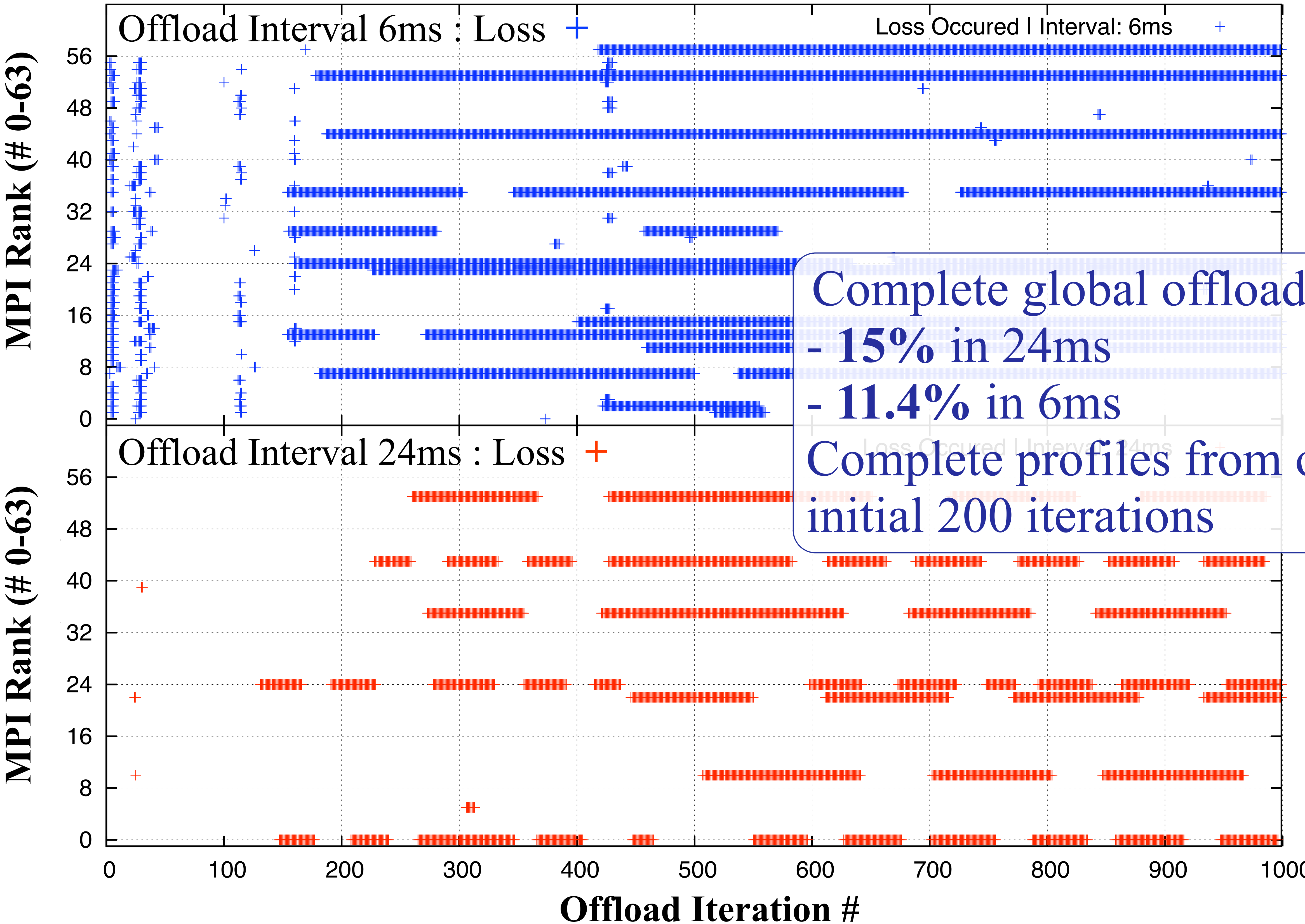
Lossy, Non-Blocking : Loss Map



6ms Offload Interval
- Loss structure different

24ms Offload Interval
- Every 8th rank is lossy
- ToM Fan-out = 8

Lossy, Non-Blocking : Loss Map



Complete global offloads:
- **15%** in 24ms
- **11.4%** in 6ms

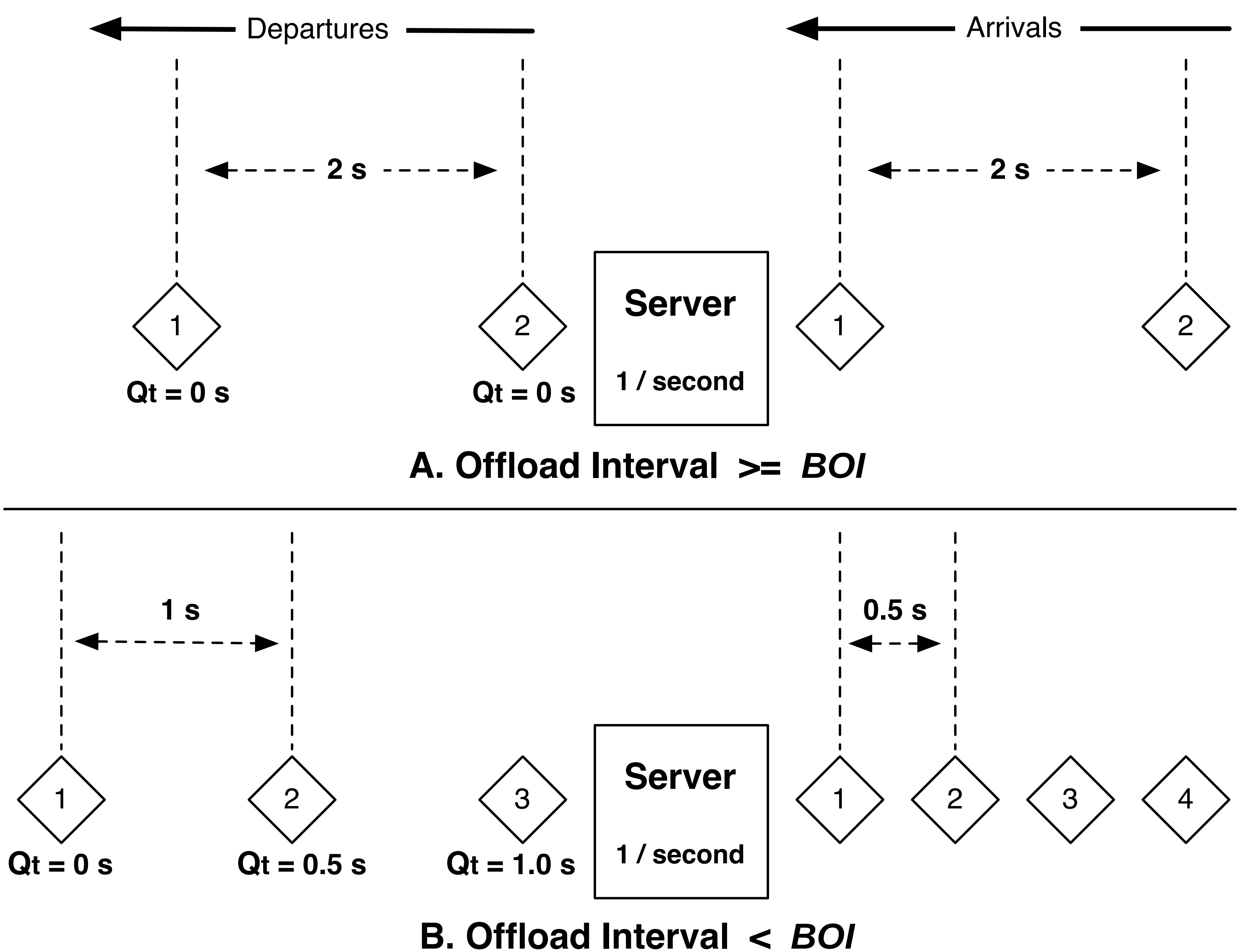
Complete profiles from only initial 200 iterations

Inconsistent, Incomplete Performance Data

- OC spike / full-buffer signal is inconsistent
  - Some ranks repeatedly penalized
  - Some ranks are never penalized
  - Bursts of loss - large intervals unmonitored
    - ⇨ *Local backoff* reaction inconsistent
  - Late reaction - OC spike / full-buffer implies damage done
- ***Spatially*** **and** ***temporally*** **inconsistent performance views**
  - Which *ranks* are monitored when? No control.
  - Which *intervals / iterations* are monitored? No control.
- Need globally consistent performance views
- <u>Need globally consistent method to determine *ToM* capacity</u>

Queueing Theory 101

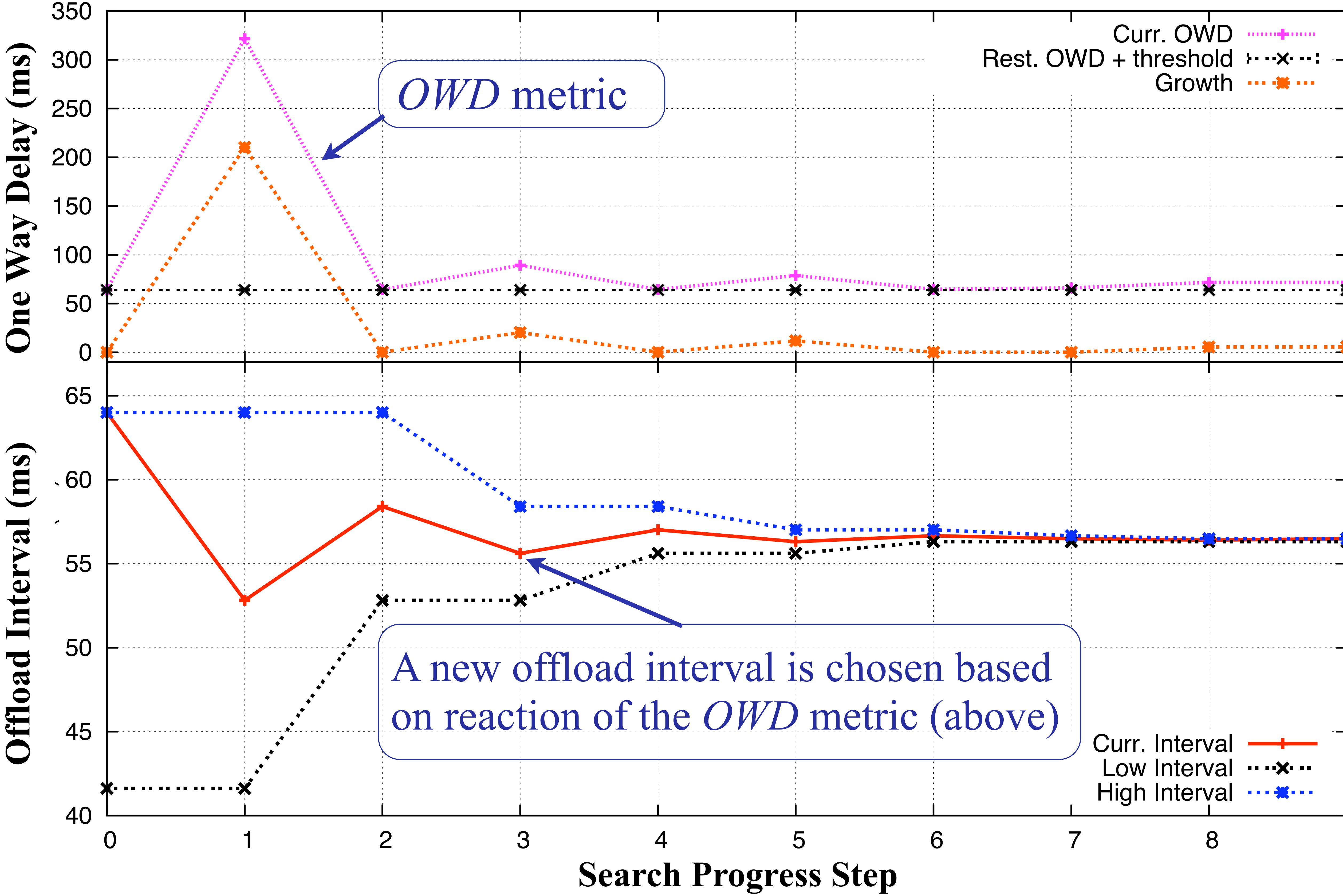
BOI: bottleneck offload interval



Estimation of Bottleneck Interval

- Need an estimator of operating capacity
  - Minimum offload interval without queueing
  - Per profile size and filter type
  - Metric: *Bottleneck Offload Interval (BOI)*
- *Estimation Method 1*
  - If offload interval < *BOI*, then departure interval ≈ *BOI*
  - Set offload interval = 0
  - Measure departure interval
- *Estimation Method 2*
  - If offload interval < *BOI,* then queue builds, *OWD* increases
  - If offload interval ≥ *BOI,* then *OWD* stable or decreases
  - Use increasing *OWD* as heuristic to *binary*-*search* for *BOI*

BOI Binary Search: Example



Evaluating BOI Estimation

- *Bottleneck Offload Interval* depends on many factors
  - Underlying network latencies and bandwidth
  - TCP / IP stack processing
  - MRNet (de)packetization
  - Intermediate custom filter and sink operations
  - TAU wrapper processing
- Cannot use standard capacity estimation tools (e.g., Nettimer)
  - Probes will not encounter all costs involved in *ToM*
  - Meant for one-to-one paths, not many-to-one *ToM* trees
- Instead *corroborate BOI* estimates from Methods 1 and 2
  - Test under various configurations
  - Result: *Estimates agree to within 10%*

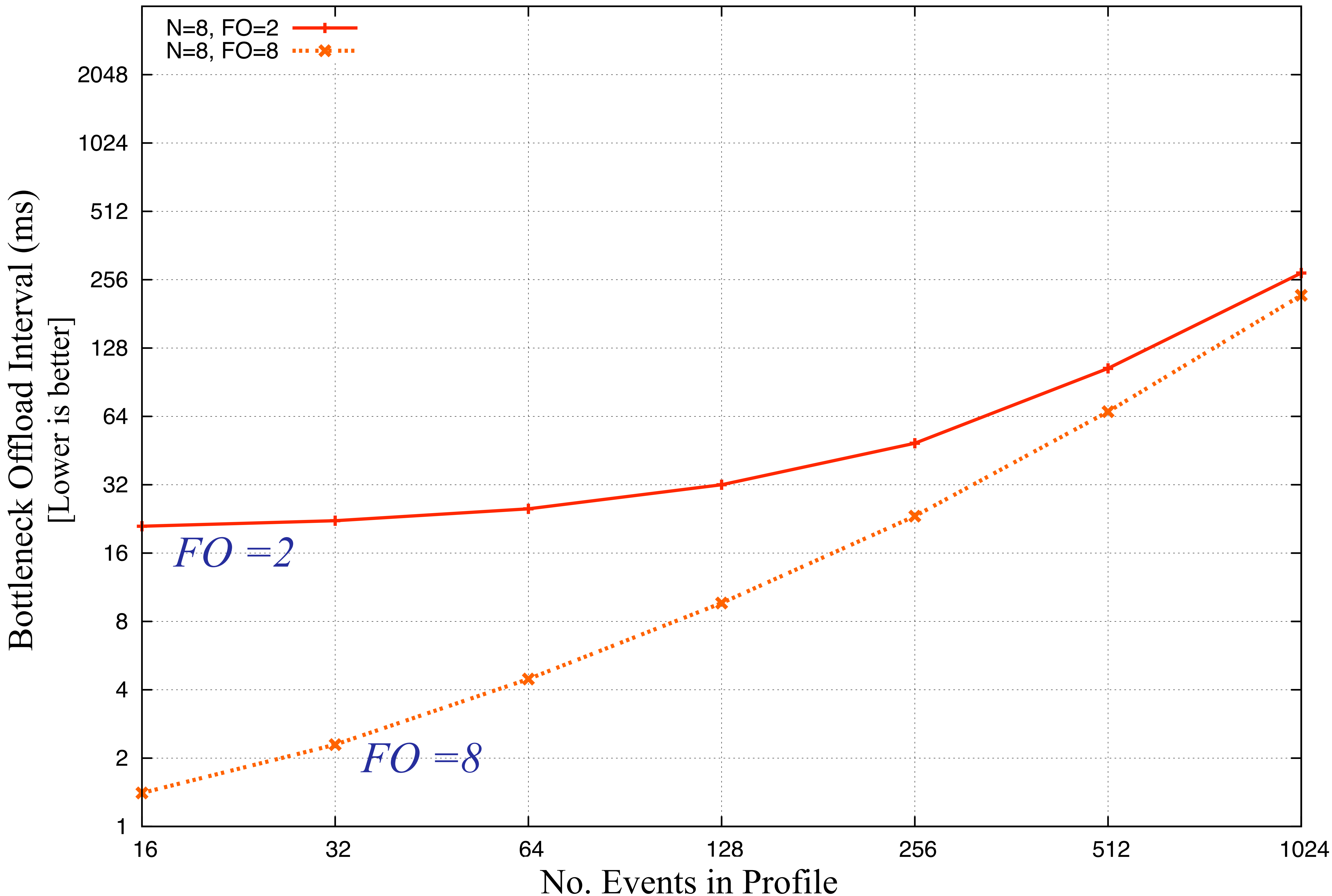
Once BOI is determined, how should it be used?

- *ToM* provides estimation APIs to query
  - *BOI*, *OWD*, *OC* and several other metrics
- Application can use metrics to decide profile granularity
- Example: Iterative application - 75 ms per iteration
  - Estimated *BOI* provided by *ToM* = 100 ms
  - All application ranks can decide to *drop every 4th profile*
  - Average *offload interval* (over 4 rounds) will match *BOI*
- How to bridge monitoring requirements and costs?
  - Given application size and type of reductions ...
  - How to choose: ToM fanout, offload intervals, # profile events
- Answer: *Characterization*
  - Need to characterize various ToM configurations using *BOI*

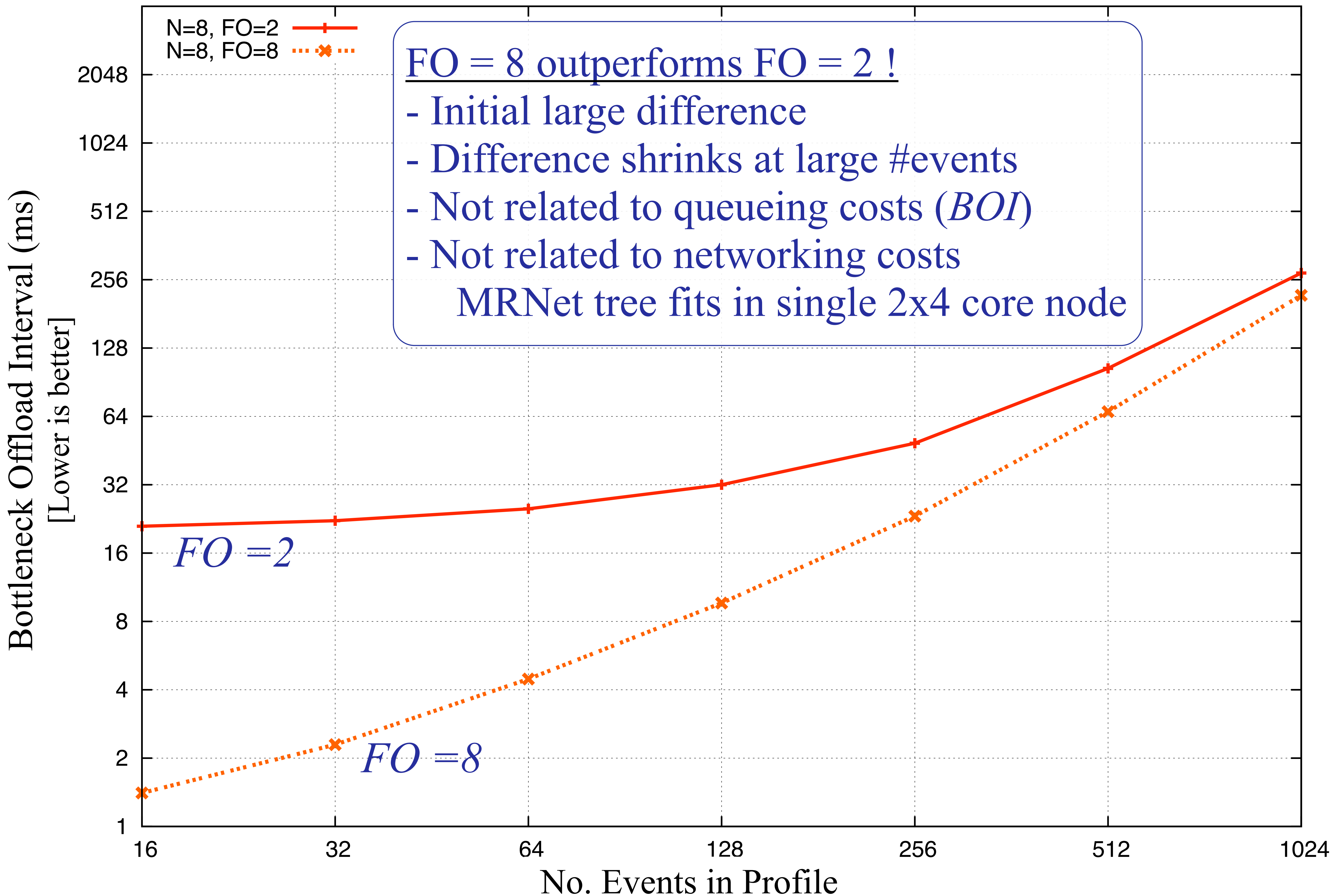
Characterizations using BOI

- Configurations
  - Simple statistics filter:
    - mean, min, max, standard deviation
  - Profile sizes
    - 16 to 1024 events with power of two increments
  - Application size N = 8
    - ToM fanouts FO = 2, 8
  - Application size N = 16
    - ToM fanouts FO = 2, 4, 16
  - Application size N = 64
    - ToM fanouts FO = 2, 4, 8, 64
- Estimate *BOI* for each configuration
  - Pick median of 3 trials for each point

Application Size N = 8; Fanouts FO = 2, 8



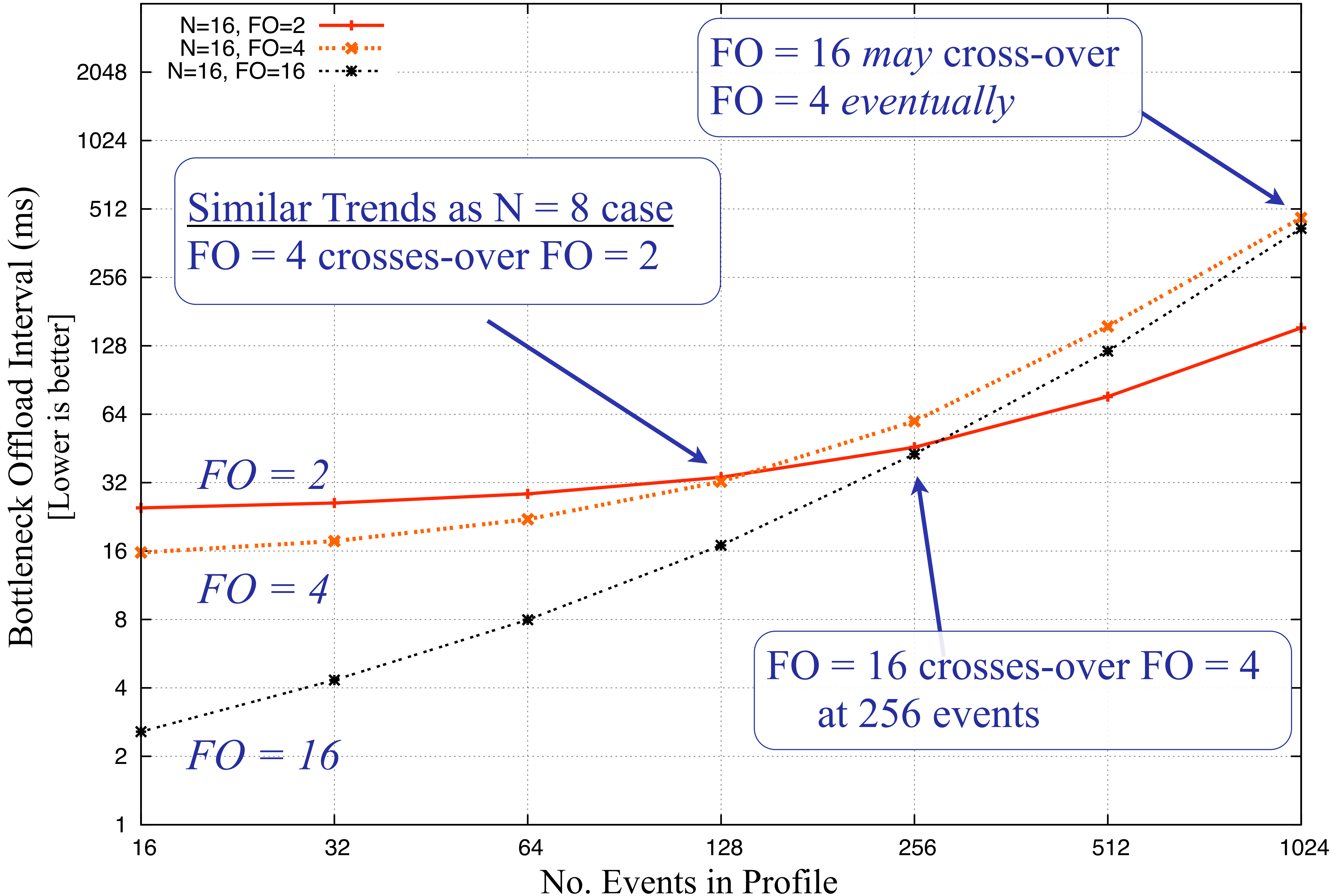
Application Size N = 8; Fanouts FO = 2, 8



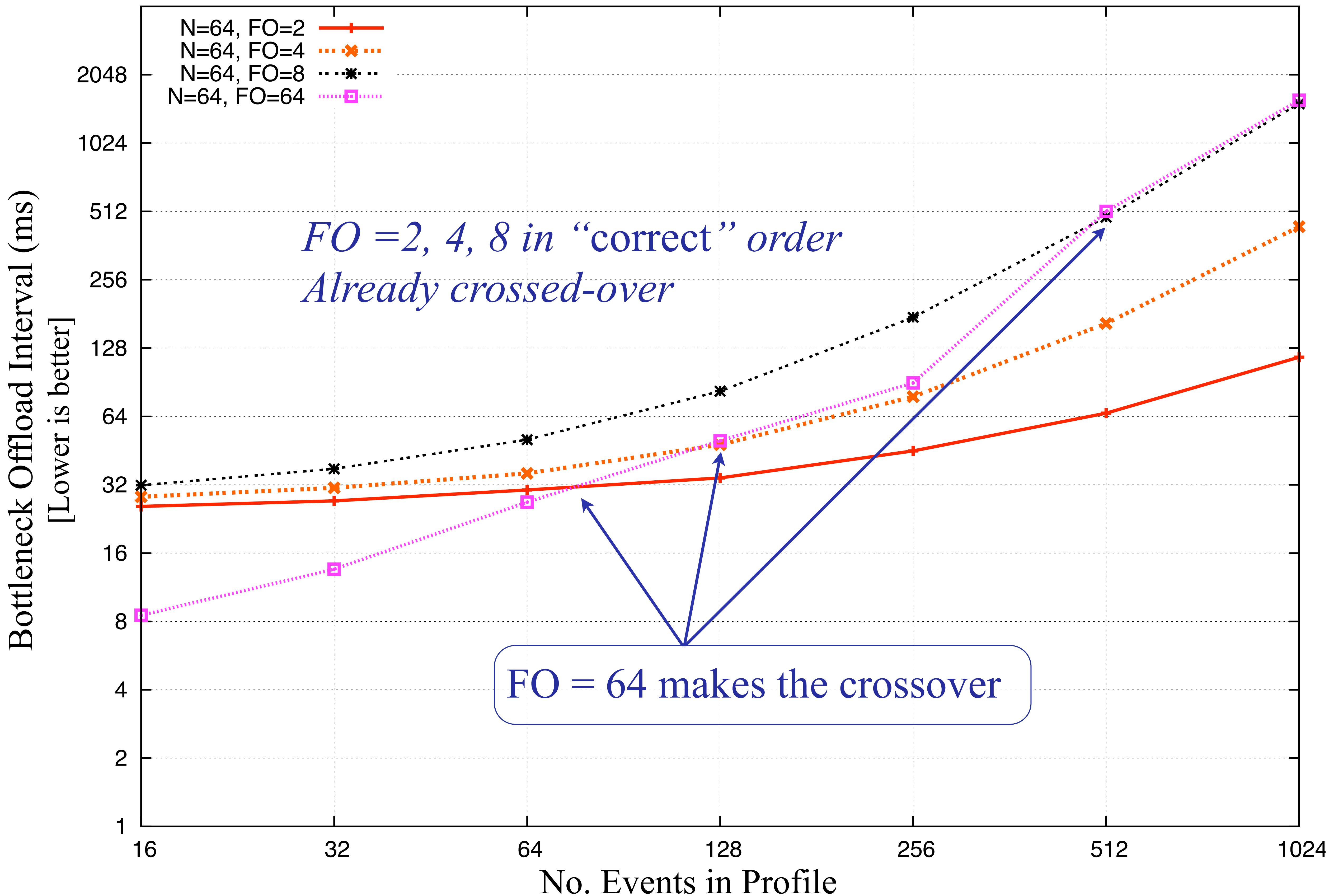
Reduction and (De)Packetization Costs

- Reduction Cost ($\boldsymbol{T_R}$) for single binary reduction operation
  - For N = 8, 7*$\boldsymbol{T_R}$ cycles for reduction
  - Reduction performed on *arrival of last profile*
  - FO = 8: single thread performs all 7*$\boldsymbol{T_R}$ cycles serially
  - FO = 2: 7*$\boldsymbol{T_R}$ cycles split across 7 threads in tree
- (De)Packetization Cost ($\boldsymbol{T_P}$) to (un)pack intermediate profile
  - FO = 8: No intermediate (de)packetizations
  - FO = 2: Every-level in tree adds (de)packetizations
- At small #events, $\boldsymbol{T_P}$ dominates costs in FO = 2
- FO = 8 $\boldsymbol{T_R}$ costs quickly rise as single processor saturated
- Intuitively: *Too many resources* allocated in FO = 2
  - Serial costs and parallelization overheads dominate

Application Size N = 16; Fanouts FO = 2, 4, 16



Application Size N = 64; Fanouts FO = 2, 4, 8, 64



Monitoring Requirements and Infrastructure Costs

- *BOI* characterization shows importance of careful match
  - Under-provisioning may be bad for performance
  - But so can *over-provisioning* !
- Other metrics (direct costs, limiting overhead) also in paper
- Sweet-spot configurations for specific requirements exist
- *ToM* helps find these *sweet-spots*
  - APIs
  - Framework
  - Metrics and evaluation methodology

Conclusion and Future Work

- ***Sweet-spot***
  - "Spot on a bat that produces the least shock when a ball is hit" (New Oxford American Dictionary)
- Parallel performance monitoring must meet overhead, latency, responsiveness, data consistency requirements
- *Sweet-spots* are those configuration choices that meet the requirements or allow acceptable trade-offs
- Methodology and framework help find *sweet-spots* and make informed monitoring decisions
- Would like to extend the characterizations to other filters
- Analysis of irregular, less periodic or non-uniform behaviors
- Dynamic estimation and feedback to application to stay within sweet-spot during execution

Credits

- University of Oregon
  - Aroon Nataraj
  - Alan Morris
  - Allen D. Malony
  - TAU group members
  - "Extreme Performance Scalable Operating Systems," DOE FastOS-2 grant, with Argonne National Lab

- University of Wisconsin
  - Dorian C. Arnold (soon to be at University of New Mexico)
  - Michael Brim
  - Barton P. Miller