Monitoring Extreme-scale Lustre Toolkit

Michael J. Brim
Joshua K. Lothian

Computer Science Research Group
Computer Science & Mathematics Division
Oak Ridge National Laboratory
Tree-Based Overlay Networks (TBONs)

- Designed to address scalability problems in master-worker tool/application architectures

- Overlay network structured as a tree graph
  - provides logarithmic scaling for multicast/gather communication
  - provides distributed data processing (e.g., filtering, reductions)

- Distributed Data Processing
  - distribute processing across subtrees to reduce master load
  - for streaming data, pipeline parallelism on paths from leaves to root

- Tree topology can be optimized based on communication and data processing needs
  - Balanced: equal fan-out from all vertices at a given tree depth
    - good for load-balanced distributed aggregation
  - Binomial: good for streaming throughput
MRNet (since 2003)

- **General-purpose TBŌN API (C++)**
  - **Network**: user-defined topology
  - **Stream**: logical data channel
    - to a set of back-ends
    - multicast, gather, and custom filter reduction
  - **Packet**: collection of data
  - **Filter**: stream data operator
    - synchronization
    - transformation
- Tool developer writes front-end (FE), back-end (BE), and Stream Filter code using library API
- MRNet provides communication process (CP) executable
@Brad Settlemyer - Hey Mike, do you think it would be possible to build an MRNet-based tool to diagnose Lustre locking issues?

@Mike - Sure, assuming the problem can be tackled using hierarchical data aggregation.

… a couple months pass …

@Brad - Could you use the same infrastructure to continuously monitor Lustre performance and detect problems?

@Mike - That sounds a bit like my parallel top tool, only more Lustre oriented.

@Mike - But in its current state, you can’t use MRNet across separate network domains.
Current Lustre Performance Monitoring

- General-purpose host monitoring
  - Collectl
  - Ganglia
  - NAGIOS

- Lustre-specific performance monitoring
  - LLNL LMT
    - server-side monitoring (OSS, MDS, LNET)
    - realtime monitoring via top-like display
    - uses a real database to store historical data!!
    - dependent on LLNL Cerebro, multicast can be hard to deploy
  - TACC lltop/xltop
    - server-side monitoring (OSS, MDS)
    - integrates with batch job system to display per-job information
    - direct ssh/socket connections between master and server daemons => limits scalability
  - Collectl plugin for Lustre
    - single host information for clients, OSS, and MDS
    - detailed info available on clients and OSS
Limitations of Current Lustre Monitoring

- Limitations of current toolkits include one or more of:
  - problem analysis is generally post-mortem
  - hard to correlate measurements:
    • across clients within a job or application
    • across servers used by a job or application
    • across servers used by a given client
    • ...
  - lack of insight into MDS, LNET, etc.
  - scalability (# of monitored nodes)
  - center-wide monitoring
Lustre Monitoring Grand Vision

• Full visibility
  – clients, MDS, OSS, OST, LNET
  – storage devices (if possible)

• Support for center-wide deployments
  – multiple compute systems sharing one or more Lustre filesystems

• Two usage modes
  1. always on, low-overhead monitoring
     • with active problem detection and alerting
  2. on demand, in-depth problem inspection and diagnosis
     • aka “Right Now Queries”
Monitoring Extreme-scale Lustre Toolkit (MELT)

• Collects Lustre performance metrics
  – on clients, OSS, MDS, LNET

• Uses SNOflake overlay network to:
  – aggregate metric data into performance summaries
    • for clients and LNET routers of each compute cluster
    • for OSS and MDS servers of each storage cluster
  – correlate data within and across compute/storage domains
    • within compute domain: e.g., app-level or job-level aggregation
    • across compute/storage domains: identify server or filesystem contention
MELT Command-line Interface

melt [options] target mode classes [mode-opts]

- **Targets** - specifies information source
  - `fs`: filesystem-level information
  - `job`: information for a given job
  - `oss`: information for a given OSS server
  - `mds`: information for a given MDS server or all MDS
  - `clnt`: information for a given client
MELT Command-line Interface

melt [options] target mode classes [mode-opts]

- **Modes** - controls how information aggregated
  - `status`: min/max/sum/avg (default is sum)
  - `top`: show top-k entries for a given metric and k-value

- **Metric Classes** - which metrics to gather
  - `io`, `lock`, `meta`, `rpc`, `client`, `op`, `path`
  - each class has a set of associated metrics
    - e.g., `IO_RD_BW`, `META_OP_RATE`, `RPC_PENDING`
## MELT CLI Example – Filesystems Status

```
% melt fs status io,meta -delay=1m \\  
  -metrics=IO_RD_BW,IO_WR_BW,META_OP_RATE
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>FILESYS</th>
<th>RD_BW</th>
<th>WR_BW</th>
<th>MD_RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30:32</td>
<td>knot1</td>
<td>217 MB/s</td>
<td>133 MB/s</td>
<td>7 op/s</td>
</tr>
<tr>
<td>08:30:33</td>
<td>knot2</td>
<td>49 MB/s</td>
<td>7.6 GB/s</td>
<td>43 op/s</td>
</tr>
<tr>
<td>08:31:33</td>
<td>knot1</td>
<td>183 MB/s</td>
<td>94 MB/s</td>
<td>0 op/s</td>
</tr>
<tr>
<td>08:31:35</td>
<td>knot2</td>
<td>53 MB/s</td>
<td>7.8 GB/s</td>
<td>61 op/s</td>
</tr>
</tbody>
</table>

...
MELT CLI Example – Job Status

```bash
% melt job=tait.1234 status io,meta -delay=5m \
   -metrics=IO_RD_BW,IO_WR_BW,META_OP_RATE

<table>
<thead>
<tr>
<th>TIME</th>
<th>RD_BW</th>
<th>WR_BW</th>
<th>MD_RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:40:32</td>
<td>692 MB/s</td>
<td>0 B/s</td>
<td>75 op/s</td>
</tr>
<tr>
<td>08:45:33</td>
<td>117 MB/s</td>
<td>13 MB/s</td>
<td>33 op/s</td>
</tr>
<tr>
<td>08:50:32</td>
<td>0 B/s</td>
<td>9 MB/s</td>
<td>13 op/s</td>
</tr>
<tr>
<td>08:55:32</td>
<td>0 B/s</td>
<td>8 MB/s</td>
<td>14 op/s</td>
</tr>
<tr>
<td>09:00:33</td>
<td>153 MB/s</td>
<td>2 MB/s</td>
<td>47 op/s</td>
</tr>
</tbody>
</table>
```

...
MELT CLI Example - Filesystem Status

```bash
% melt fs=knot2 status io,rpc -delay=10s \
   -metrics=IO_RD_BW,IO_CLNT_DIRTY,RPC_PENDING
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>WR_BW</th>
<th>CL_DIRTY</th>
<th>RPC_PEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:34:16</td>
<td>7.7 GB/s</td>
<td>1.32 TB</td>
<td>32345</td>
</tr>
<tr>
<td>08:34:26</td>
<td>7.8 GB/s</td>
<td>1.30 TB</td>
<td>30178</td>
</tr>
<tr>
<td>08:34:35</td>
<td>7.4 GB/s</td>
<td>1.29 TB</td>
<td>29006</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>TIME</th>
<th>WR_BW</th>
<th>CL_DIRTY</th>
<th>RPC_PEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:36:45</td>
<td>7.9 GB/s</td>
<td>91.7 GB</td>
<td>2456</td>
</tr>
<tr>
<td>08:36:56</td>
<td>3.3 GB/s</td>
<td>7.85 GB</td>
<td>913</td>
</tr>
<tr>
<td>08:37:06</td>
<td>127 MB/s</td>
<td>372 MB</td>
<td>123</td>
</tr>
</tbody>
</table>
MELT CLI Example - Filesystem Top Jobs

```plaintext
% melt -group=job fs=knot2 top io \ 
  -topk=5 -topmetric=IO_RD_BW \ 
  -metrics=IO_RD_BW,IO_CLNT_AVG_RD_SZ,\ 
  IO_CLNT_AVG_RD_TIME

<table>
<thead>
<tr>
<th>JOB</th>
<th>RD_BW</th>
<th>RD_SZ</th>
<th>RD_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>conway.2789</td>
<td>12 GB/s</td>
<td>127 MB</td>
<td>63.9 ms</td>
</tr>
<tr>
<td>tait.4321</td>
<td>7.8 GB/s</td>
<td>156 MB</td>
<td>72.3 ms</td>
</tr>
<tr>
<td>euler.22397</td>
<td>7.2 GB/s</td>
<td>112 MB</td>
<td>64.5 ms</td>
</tr>
<tr>
<td>tait.4334</td>
<td>3.4 GB/s</td>
<td>354 MB</td>
<td>283 ms</td>
</tr>
<tr>
<td>euler.22388</td>
<td>780 MB/s</td>
<td>31.9 MB</td>
<td>54.7 ms</td>
</tr>
</tbody>
</table>
```
MELT CLI Example - Job Performance Log

% melt -group=job -format=log fs status io \
   -delay=5m

Jan 15 11:22:33 skein melt[123]: job=tait.1111 IO_RD_BW=20M/s
   IO_WR_BW=476M/s IO_CLNT_NUM=256 IO_CLNT_DIRTY=4.3G
   IO_CLNT_AVG_RD_SZ=776K IO_CLNT_AVG_WR_SZ=1M ...

Jan 15 11:22:33 skein melt[123]: job=tait.1113 IO_RD_BW=89M/s
   IO_WR_BW=21M/s IO_CLNT_NUM=64 IO_CLNT_DIRTY=1.2G
   IO_CLNT_AVG_RD_SZ=507K IO_CLNT_AVG_WR_SZ=123K ...

Jan 15 11:22:33 skein melt[123]: job=tait.1114 IO_RD_BW=364M/s
   IO_WR_BW=28M/s IO_CLNT_NUM=32 IO_CLNT_DIRTY=86M
   IO_CLNT_AVG_RD_SZ=1.4M IO_CLNT_AVG_WR_SZ=67K ...

...
SNOflake - Scalable Network Overlay

• General-purpose overlay network infrastructure for constructing distributed services, tools, and apps
  – bootstrapping and distributed launching
    • system-level and user-level
    • deployments spanning intra-network domains
  – peer and group communication
    • leverage advanced network capabilities (e.g., RDMA or collectives)
  – integrated, customizable data analysis and aggregation

• Real Scalability: no changes to core design/architecture required for use on future “extreme scale” systems

• Real Resilience: overlay network should persist as long as any of the constituent distributed systems are operational
SNOflake Design Characteristics

• Support for cross-domain overlay deployments

• Simple yet flexible API in C
  – *Session* represents an overlay shared among clients
  – each *Session* supports many logical *Services*
  – each *Service* supports many data *Streams*
  – *Streams* used to transfer/process opaque *Data Buffers*, rather than formatted Packets
  – *Filter Graph* instead of single filter per Stream

• Ability to leverage advanced networking capabilities
  – incorporate layers such as the Common Communication Interface (CCI) or the Universal Common Communication Substrate (UCCS)
SNOflake Architecture Overview

- Deploy TBŌNs on separate resource domains
  - place *Tree Managers* (i.e., TBŌN roots) on hosts with inter-domain communication capability
  - use separate trees for distinct resource classes within same distributed system (e.g., compute, management, storage)

- Ring of Tree Managers
  - data routing between TBŌNs
  - state replication within ring for fault tolerance

- “SNOflake as a Service”
  - at-boot SNOflake provides bootstrap/launch service for scalable deployment of additional SNOflake-based services, tools, and apps
SNOflake Architecture
MELT Architecture Overview

• Uses SNOflake overlay network for:
  – aggregating metric data into performance summaries for each domain
  – correlating data within and across domains

• Deploys monitoring services and associated backend agents on clients, servers, and LNET routers
  – intended as an on-boot infrastructure
MELT Continuous Monitoring

• *meltmon frontend*
  – controls default aggregations and sampling rates for all the metrics
  – periodically polls the job scheduling system(s) to associate compute nodes with jobs
    • multicasts the job=>{node,…} mappings to client agents
  – dumps aggregated metrics to logs
MELT On-Demand Investigation

• CLI tool attaches to MELT session as additional frontend

• Tool may:
  – subscribe to existing service data streams
    • no additional transmission of performance data vs. meltmon
  – create new streams that use different metric aggregations (e.g., to filter on a specific job)
    • performance data from backends will be sent on multiple streams

• Backends sample at the highest requested rate for a given metric
MELT – backend data collection

• Considered methods
  1. read directly from Lustre /proc files
     • first-party, likely most efficient method
     • high development/maintenance cost (e.g., procfs to sysfs) ✗
  2. leverage Collectl Lustre plugin
     • already used at a number of sites, integrates well with other monitoring (e.g., Ganglia)
     • ongoing support a concern
     • overhead of Perl a concern
  3. use persistent lctl and periodic queries
     • @Andreas Dilger – lctl is “the path forward” for reading metrics
     • improvements/fixes will be integrated into ongoing releases
     • overhead of third-party collection a concern
MELT – backend data collection

• Choice between collectl and lctl
• Experiment to monitor overheads on a single host
  – sample client per-OST statistics
    • polling 56 separate entries in /proc (one per OST)
  – Collectl default sampling rate is every 10 seconds
  – simulate whole-day collection (8640 total samples) by decreasing inter-sample delay to 0
  – measure walltime, CPU & memory usage
    • via /usr/bin/time, which uses wait4() to get rusage data
MELT - collectl vs. lctl overhead

• Collectl
  – average time per sample ~ 8.8ms
  – average CPU load 99%
    • ~ .087% scaled to normal sampling
  – maximum resident memory ~ 79MB

• MELT querying lctl
  – average time per sample ~ 6.7ms
  – average CPU load 27.2%
    • ~ .018% scaled to normal sampling
  – maximum resident memory ~ 124MB

Time spent over 8640 samples

Memory used over 8640 samples
MELT – lctl bugs and improvements

- Using a persistent lctl and periodically querying it has revealed a few usability issues
  - no clear marker to indicate end of query response
    - have a quick fix, still need to submit patch
  - initial request determines query buffer size, so subsequent longer requests are truncated
    - already fixed (by others) in git head
  - query command options ignored in subsequent requests
SNOflake Implementation Status

• Complete
  – bootstrapping over multiple domains
  – core communication (for base TCP sockets)
  – basic data filtering

• Under Construction
  – frontend/backend client API and request servicing
  – service-launching service

• Future Work
  – ring-state replication
  – TBON recover after overlay process failure
  – integration of advanced network abstraction layers
MELT Implementation Status

• Backend agents
  – collecting an initial set of relevant metrics
  – on clients, OSS, MDS, and LNET routers

• Metric data aggregations
  – implementing metric-specific performance summaries (min, max, sum, avg) as data filter aggregations
  – considering other aggregations such as histograms

• Under construction
  – meltmon frontend
  – CLI frontend
You’re the experts - Please advise

• Still a work-in-progress
  – you can influence delivered capabilities

• What metrics are you most interested in?
  – are there new metrics you would like added to Lustre?

• Besides instantaneous performance summaries and a historical record of such summaries, what else?
Future Directions: Performance Alerts

• With continuous monitoring, opportunity to detect anomalous performance and notify

• Challenges
  – what’s anomalous: need a baseline
    • for any metric that you wish to alert on
  – performance is dependent on offered load
  – changing workloads could move the baseline
Future Directions: Oracle Mode

• Assuming MELT command-line tools allow experienced admins to find root causes of performance problems, can we embed that expertise in the tools

• Add a new “oracle” mode that searches for common problems on a filesystem or server level

• Challenges
  – copying the brains of expert admins
  – what level of overhead is acceptable for oracle mode?
  – is this something you could give to users for job-level problem diagnosis?