UNM SCHOOL of ENGINEERING
Department of Computer Science

Scalable Systems Lab

Exploring the Impact of Overlay Network Topology on Tool and Application Performance

Whit Schonbein

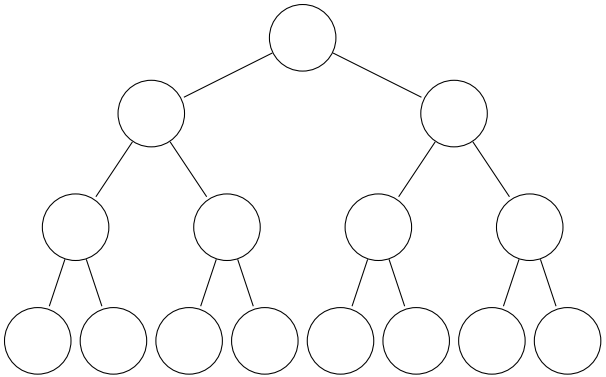
University of New Mexico

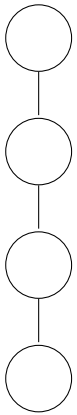
2016-03-30

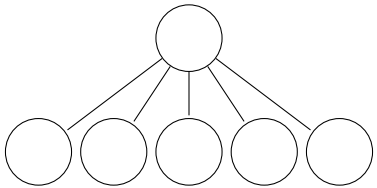
Advisor: Dorian Arnold

Introduction

- HPC monitoring, debugging, and analysis tools often share a common communication structure: Trees.







Introduction

- HPC monitoring, debugging, and analysis tools often share a common communication structure: Trees.
- Tree-based overlay networks provide lightweight communication infrastructure for facilitating this type of communication.
- Tool performance depends on overlay network performance.

Introduction

- While tree structures are common to many software architectures…
- … there are limited studies of how tools perform under different workloads and tree topologies.

Example: Latency-bound

- Task: Querying nodes for core temperature, filtering only those that exceed a threshold.
  - Simple (low-latency) aggregation filter.
  - Small packet size.
  - Low frequency.

Example: Bandwidth-bound

- Task: Collecting stack traces at milliseconds latency.
  - More computationally substantive filter (merge traces).
  - Potentially larger packet size.
  - High frequency data stream.

Introduction

- There are limited studies of how tools perform under different types of workloads and tree topologies.
  - Arnold et. al. (2006), Goehner et. al. (2013), Groves et. al. (2015)

- Goal: Start to fill in this gap in our knowledge.

Methodology

- Infrastructure: MRNet
  - In use in a variety of tools (TotalView, STAT, AutomaDeD)
  - Data aggregation capabilities via filters

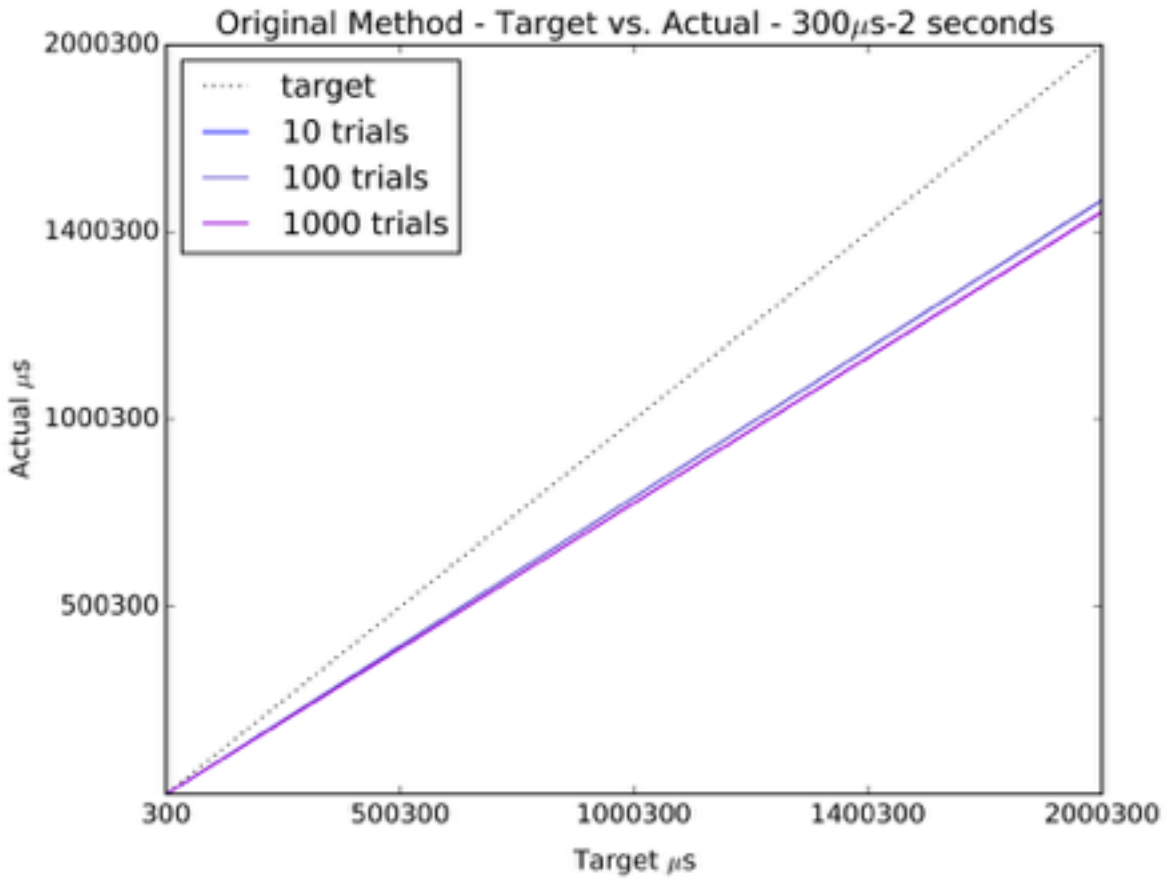
Methodology

- Benchmarking: MRNetBench
  - User-defined topologies
    - Structure
    - Number of nodes
  - Packet size
  - Communication frequency
  - Latency incurred by filters

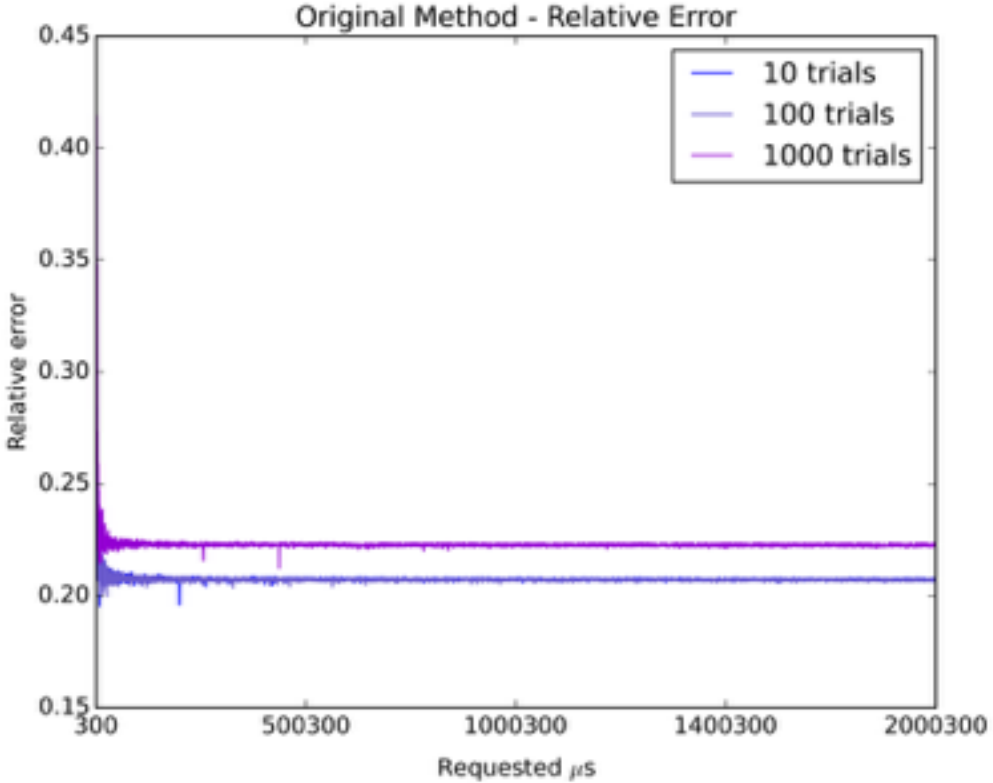
Modeling Filter Latency

- Original Method:
  - Calibration phase:
    - For each calibration trial, measure duration of matrix multiplication, resting between each trial.
    - Get average duration across trials.
  - Usage phase:
    - Calculate number of iterations of matrix multiplication required given the average duration and target latency.

Modeling Filter Latency



Modeling Filter Latency



	10 trials	100 trials	1000 trials
Mean	2.076×10^{-1}	2.073×10^{-1}	2.229×10^{-1}
Variance	6.200×10^{-8}	1.168×10^{-7}	3.778×10^{-7}
Std. Dev.	2.490×10^{-4}	3.418×10^{-4}	6.146×10^{-4}

Modeling Filter Latency: (1) New Method

- New Method
  - Calibration phase:
    - For each calibration trial, measure duration of operation performed for n cycles, resting between each trial.
    - Get average duration across trials.
  - Usage phase:
    - Calculate number of cycles required given average duration and target latency.

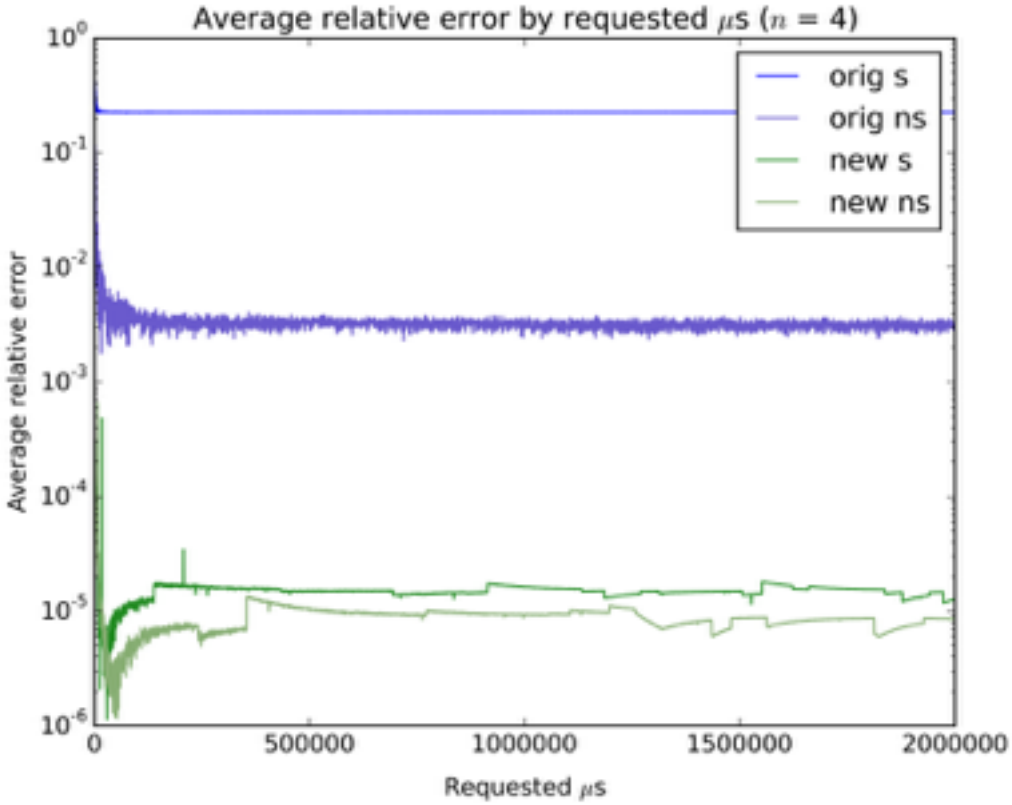
Modeling Filter Latency: (1) New Method

```
clock_gettime( CLOCK_MONOTONIC, &start )

uint64_t _sc, _tc, _cc, _sh, _sl, _ch, _cl;
float _bf = 21.1198213341;
float _o;
__asm__ __volatile__ (  "CPUID          ;"
                        "RDTSC          ;"
                        "mov %%rdx, %0  ;"
                        "mov %%rax, %1  ;"
                        : "=r" (_sh),
                          "=r" (_sl) : :
                          "%rax",
                          "%rbx",
                          "%rcx",
                          "%rdx");
_sc = ( ((uint64_t)_sh << 32) | _sl );
_tc = _sc + loop_num;
do {
__asm__ __volatile__ ("flds %3;"
                      "fmulp;"
                      "CPUID;"
                      "RDTSC;"
                      "mov %%rdx, %1;"
                      "mov %%rax, %2;"
                      : "=&t" (_o),
                        "=r" (_ch),
                        "=r" (_cl) :
                        "m" (_bf),
                        "0" (5.35667) :
                        "st(1)",
                        "%eax",
                        "%ebx",
                        "%rax",
                        "%rdx" );
_cc = ( ((uint64_t)_ch << 32) | _cl );
} while (_cc <= _tc);

clock_gettime( CLOCK_MONOTONIC, &end ) == -1 ) {

rest()
```

Modeling Filter Latency: (2) Avoid Sleeping

- Original method uses sleep(1) to rest
  - Solution: Change to a ‘nosleep’ resting method.

Modeling Filter Latency: Results

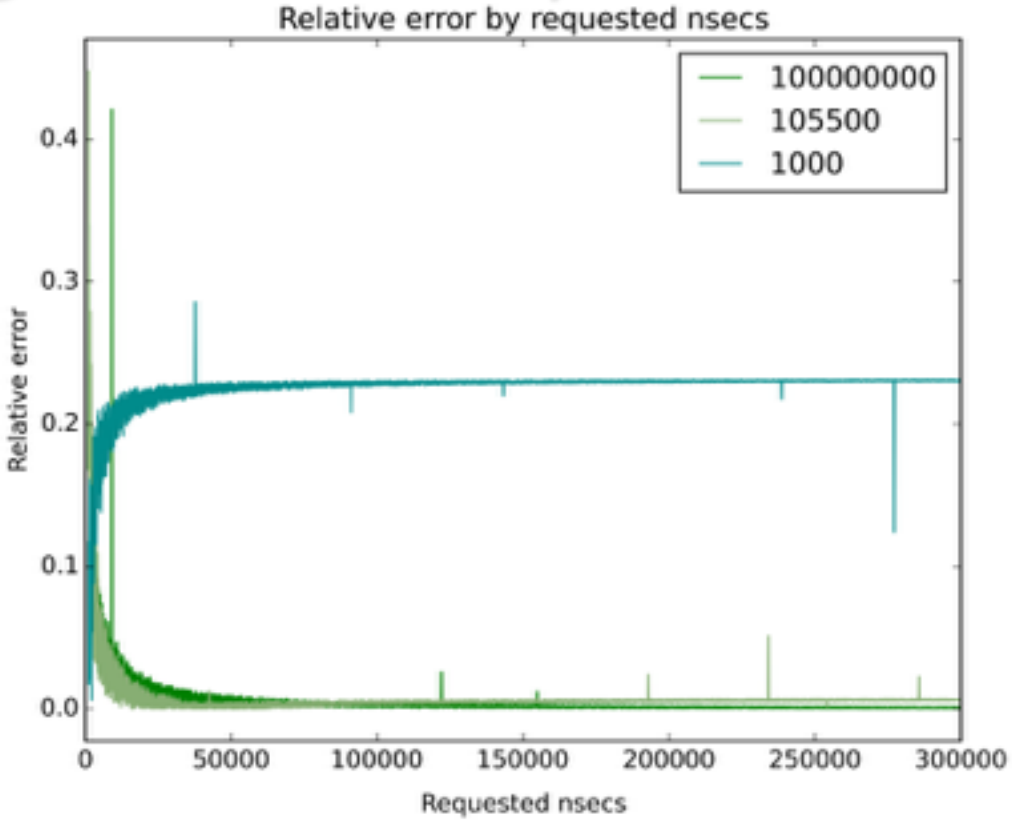


	original s	original ns	new s	new ns
Mean	2.258×10^{-1}	3.430×10^{-3}	1.564×10^{-5}	9.215×10^{-6}
Variance	4.159×10^{-5}	2.375×10^{-5}	9.150×10^{-10}	5.800×10^{-10}
Std. Dev.	6.449×10^{-3}	4.873×10^{-3}	3.025×10^{-5}	2.407×10^{-5}

Modeling Filter Latency: Results

100 calibration trials

1 x 10^8 cycles per trial



	1.0×10^3 cycles	1.505×10^5 cycles	1.0×10^8 cycles
Mean	2.263×10^{-1}	7.316×10^{-3}	5.638×10^{-3}
Variance	1.441×10^{-5}	6.661×10^{-7}	2.740×10^{-5}
Std. Dev.	3.796×10^{-3}	8.162×10^{-4}	5.235×10^{-3}

Modeling Filter Latency: Summary

- Avoiding sleep rests during calibration can reduce error in original method (why?)
- But with new method:
  - Error can be further reduced
  - Finer temporal grain
  - Can deploy different kinds of operation (logical, integer, floating point)

Conclusion

- Goal: Fill in the gap in our knowledge of how tree-based overlay network topologies impact tool and application performance.
- I've filled a gap so that we can start to fill this gap.
- Next: put it to use.

Thanks: Dorian Arnold, Evan Dye, Tyler Groves, Samuel Gutierrez, and everyone else I've pestered over this project.