Microsoft Research
微软亚洲研究院

Log^2: A Cost-Aware Logging Mechanism for Performance Diagnosis

Rui Ding*, Hucheng Zhou*, Jian-Guang Lou*, Hongyu Zhang*, Qingwei Lin*, Qiang Fu**, Dongmei Zhang*, Tao Xie***

* Microsoft Research ** Microsoft

*** University of Illinois at Urbana-Champaign

July 2015

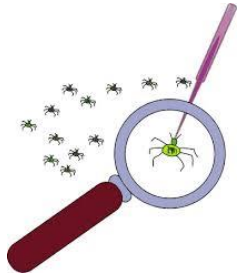
Performance logging in practice

- Recording begin/end time of a specific function call
- Such a function (*foo1*) is named Monitored Code Region (MCR)

Performance logging APIs

```
…
PerfLogger.Begin ("foo1", DateTime.Now, "process start", …); //begin
foo1();
PerfLogger.End ("foo1", DateTime.Now, "process finished", …); //end
…
```

Monitoring system state



Diagnosing perf. issues

#1	process start	foo1	00:00:00	ID1
#2	process finished	foo1	00:00:55	ID1
#3	process start	foo2	00:01:34	ID2
#4	process finished	foo2	00:01:39	ID2

……

Logging cost matters

- Logging introduces overhead
  - I/O bandwidth
  - Storage
  - CPU
  - Memory
  - …
- Potential impact on normal execution

  Example: When logging is fully conducted, latency increases 16.3% and average throughput decreases 1.48% for a search engine*
- Potential burden on log analysis
  - Long log processing time
  - Useful logs hidden among huge number of irrelevant logs

*SIGELMAN et al., Dapper, a large-scale distributed systems tracing infrastructure. In Google technical report (2010).

An empirical study of logging cost in Microsoft

- We conducted a survey of 82 logging experts from 5 divisions in Microsoft
- 80% agreed that logging cost is a non-negligible issue
- 59% experienced negative consequences of heavy logging cost

Category	Example of Reported Experiences
Storage (59%)	• *"Other process that needs disk space may crash and even logging system could crash"*
I/O bandwidth (58%)	• *"Overuse of I/O caused perception of interference with core functionality"*
CPU (56%)	• *"Service is slowed down significantly once the CPU usage of logging is increased to double digits"*
Memory (32%)	• *"Unexpected increases of memory usage of logging system once caused service incident"*

Practice of controlling logging cost

What are the most common solutions to control logging cost?

Solutions to control logging cost
(93%) Adjusting logging level
(64%) Removing unnecessary logs
(43%) Archiving local log files periodically

Are these solutions satisfactory?

Overall satisfactory
(65%) Not satisfied, needs improvement
(32%) Satisfied
(3%) I don't know

Insight from our survey

(83%) Agreed that "*many log messages are not useful to the issue under investigation*"

Eliminating useless logs has little effect on diagnostic purpose

A solution on reducing logging cost while preserving effectiveness is **feasible**

Log^2: Cost-aware logging mechanism for performance diagnosis

Problem definition

Given a pre-specified budget on I/O bandwidth*, provide a **lightweight** logging mechanism to dynamically eliminate irrelevant logs, such that the preserved logs are

1. **Budget-compliant**
2. **Effective for performance diagnosis**

* Defined as the maximum volume of logs allowed to be output in a time interval

Challenges

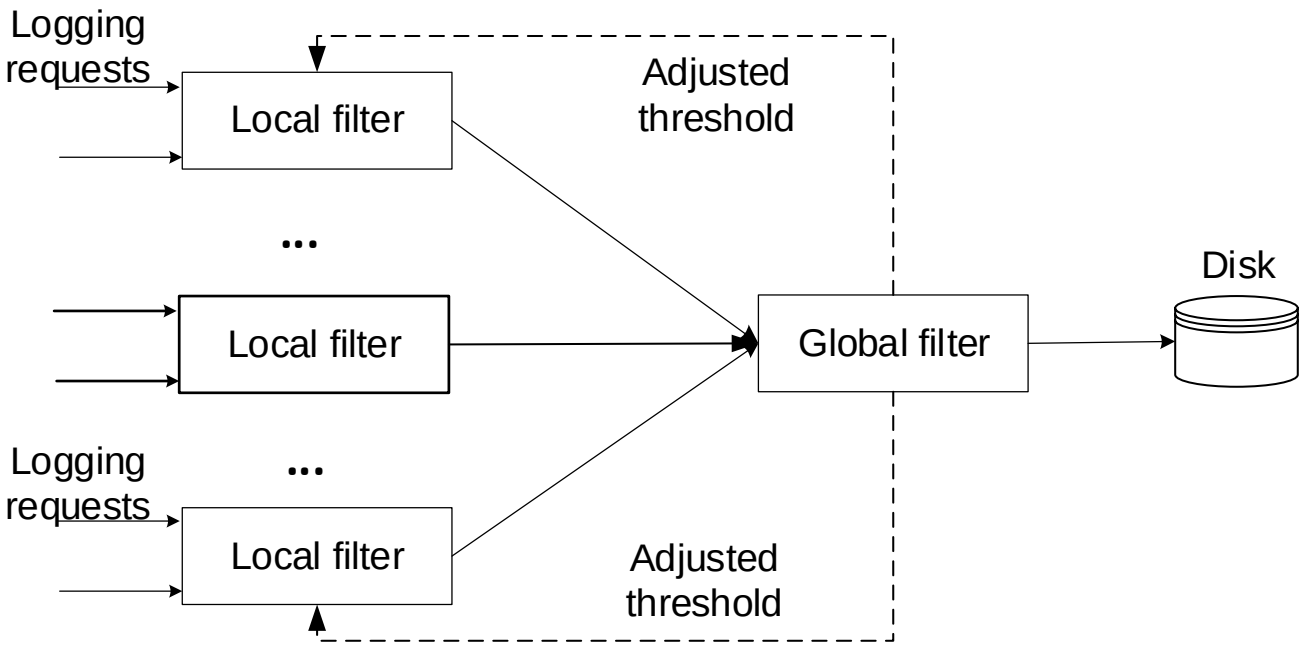
- Retaining budget-compliant while preserving logging effectiveness
- Incurring low additional overhead (i.e., low CPU and Memory usage)
- Self-adapting to system changes (e.g., workload changes, or configuration changes, etc.)

Overview of Log^2

Log^2 makes "whether to log" decision through a two-phase filtering mechanism

Local filter: a large number of irrelevant logs are discarded efficiently

Global filter: useful logs are cached and output while complying with logging budget



System diagram of Log^2

Illustrated examples of “whether to log”

The execution time of function *foo* is 55ms → I know that normally, *foo* is executed within 20~100ms → This log looks normal, not necessarily to be dumped to disk → This log is discarded

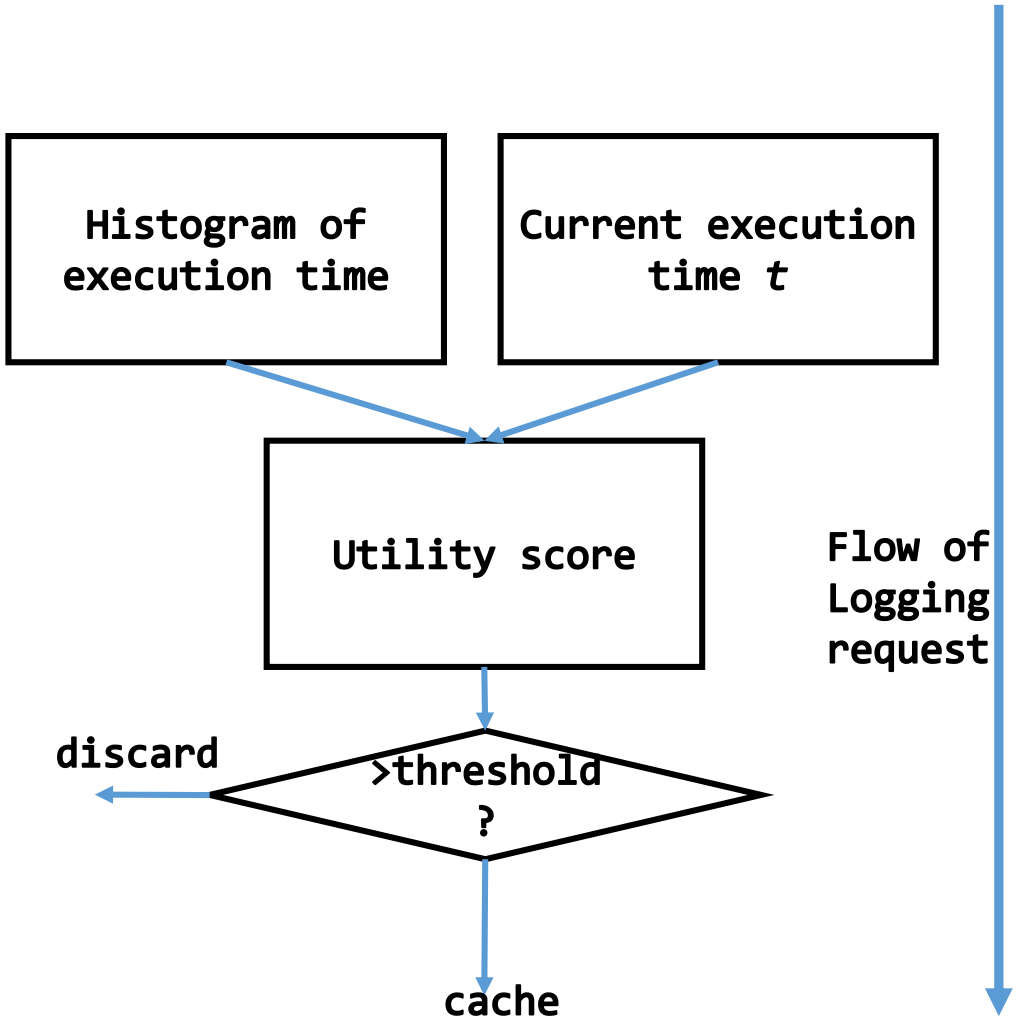
Local filter

Current cache of logs are 600KB → Budget is 500KB/interval → Identifying best 500KB logs from cache to dump → Notifying local filters to raise the bar of log discarding

Global filter

Local filter

- Utility score calculation
  - Measuring *usefulness* of logging request
  - Taking current execution time *t* and its histogram into account
- Log discarding
  - Based on a global threshold *th*
  - *th* is updated dynamically by global filter



Design of utility score

Design goals	Approach
Capturing normal behavior of each MCR	• Adopting *method of moments* to represent histogram of execution time • Memory complexity O(1)
Tackling slow-varying system states	Adopting exponential smoothing to estimate *moments*
Minimizing additional computational cost of Log^2	• Incremental update with time complexity O(1) • Asynchronized updating to avoid lock contention
Flexibility to tackle various types of performance issues	Configurable definition of "utility score"

Example of three utility scores

$$utility = \frac{t - \mu - \tau}{\sigma}$$

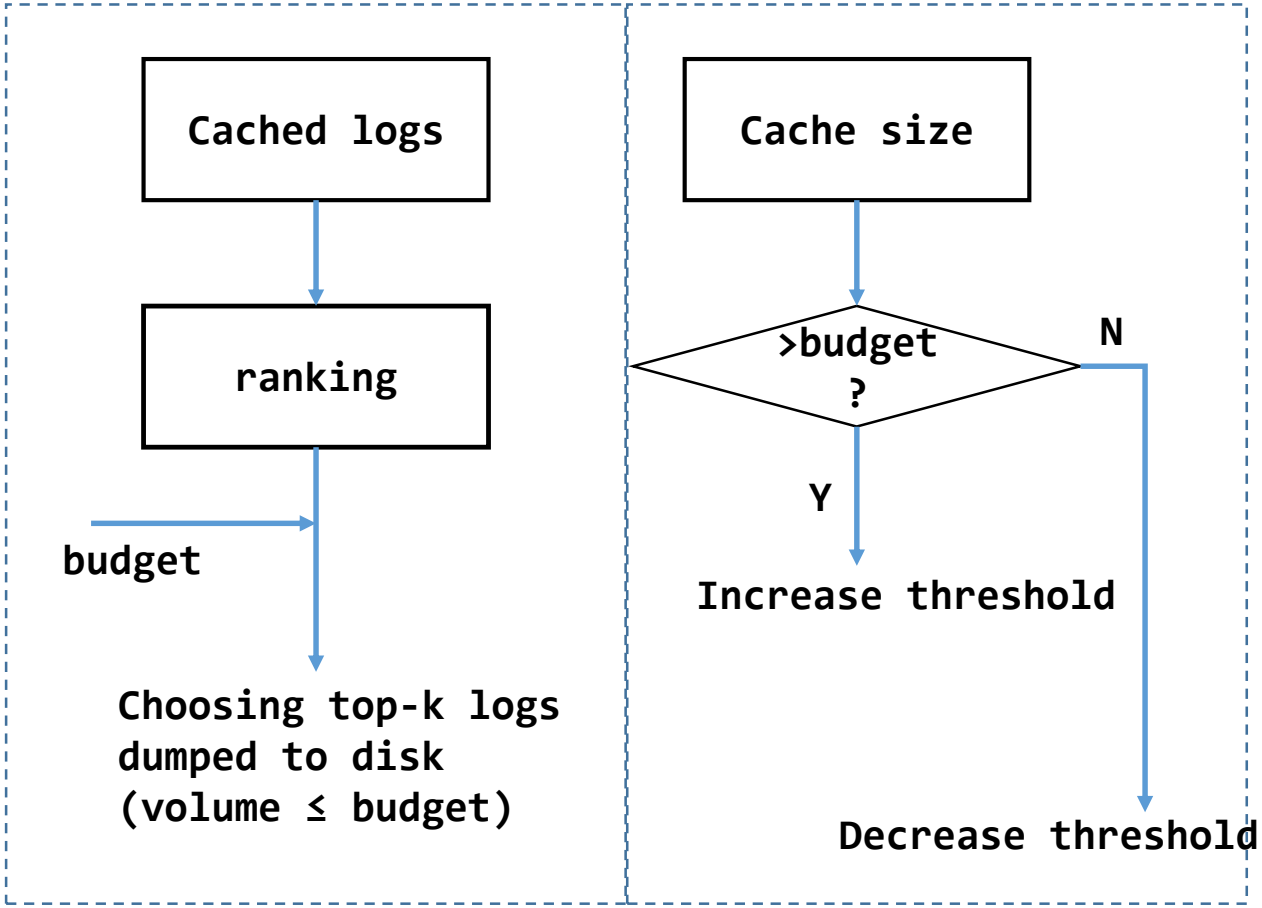
$$utility = t$$

$$utility = t - \mu - \tau$$

t: execution time
μ: mean of execution time
σ: standard deviation of execution time
τ: tolerance factor

Global filter

- Log flushing
  - Identifying and dumping most useful logs
  - Compliant with budget
- Utility threshold adjustment
  - Adjusting threshold *th* to control cache size within budget
  - Informing local filter of threshold *th* changes



Log flushing

Threshold adjustment

Design of threshold adjustment

Design goals	Approach
Bounding memory usage of Log^2	Discarding more logs when memory usage increases
• Self-adapting to workload changes • Supporting budget change on-the-fly	Adjusting threshold *th* using reinforcement learning
Agile to the workload or budget change	Adopting Secant-Method to achieve super-linear convergence rate
Minimizing additional computational cost of Log^2	Incremental threshold update with time complexity O(1)

- Modeling of threshold adjustment
  - Assumption: workload changes slowly, distribution of utility score varies slowly
  - The volume of logs passing through local filter
    - $V(T,t) = L(t)\int_T^{\infty} f(\theta_t, u)du$
  - $L(t)$: workload
  - $f(\theta_t, u)$: probability distribution of utility score
  - T_n: threshold for utility score at time n
  - V_n: volume of logs dumped at time n
  - B: budget
- Updating method
  - Estimating T^*, $s.t.\ B = L(t)\int_{T^*}^{\infty} f(\theta_t, u)du$
  - Adopting Secant Method
    - $T_n = T_{n-1} - (V_{n-1} - B)\frac{T_{n-1}-T_{n-2}}{V_{n-1}-V_{n-2}}$

Evaluation

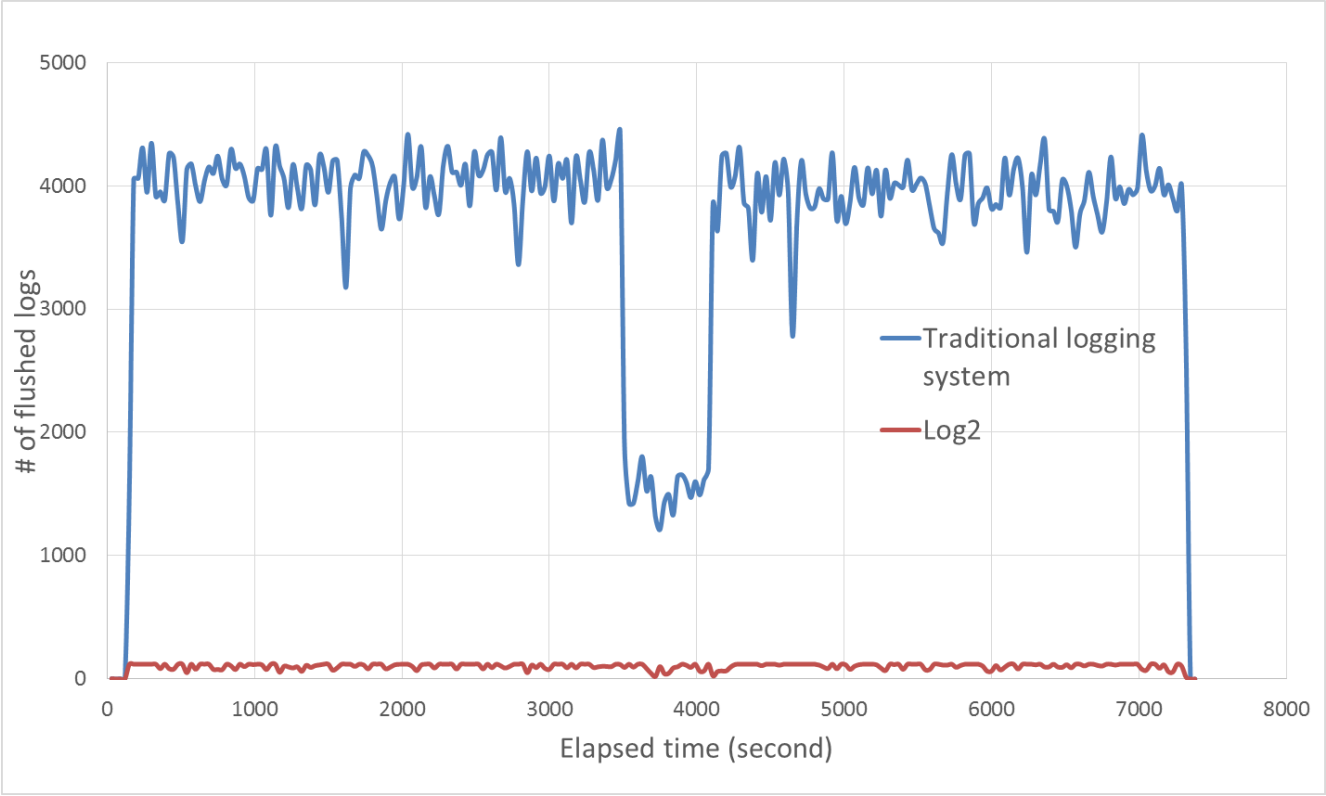
- Evaluation subjects
  - BlogEngine
    - A popular open-source blogging platform
    - More than 1,000,000 downloads as of January 2015
  - Microsoft ServiceX
    - A 3-tier large-scale online service system
    - Serving millions of users globally
- Evaluation metrics
  - **Logging throughput**: I/O throughput of by Log^2 vs. existing logging system
  - **Logging effectiveness***: Effectiveness of capturing performance issues by Log^2
  - **Additional overhead**: Additional CPU and memory usage of Log^2

* measured as the percentage of performance issues that can be captured by the flushed logs.

Evaluation on BlogEngine

- Instrumentation
  - Three types of Monitoring Code Regions (MCRs)
    - Expensive system level API
    - Loop blocks
    - Cross-boundary function calls
  - ~1000 individual MCRs in total are instrumented
- Performance issue injection
  - Uploading a large file
  - Searching a strange term
  - Exhausting CPU usage by other process
- Environment setup
  - ~100 synthetic users concurrently accessing BlogEngine
  - Each round of experiment lasts two hours
  - 7 rounds

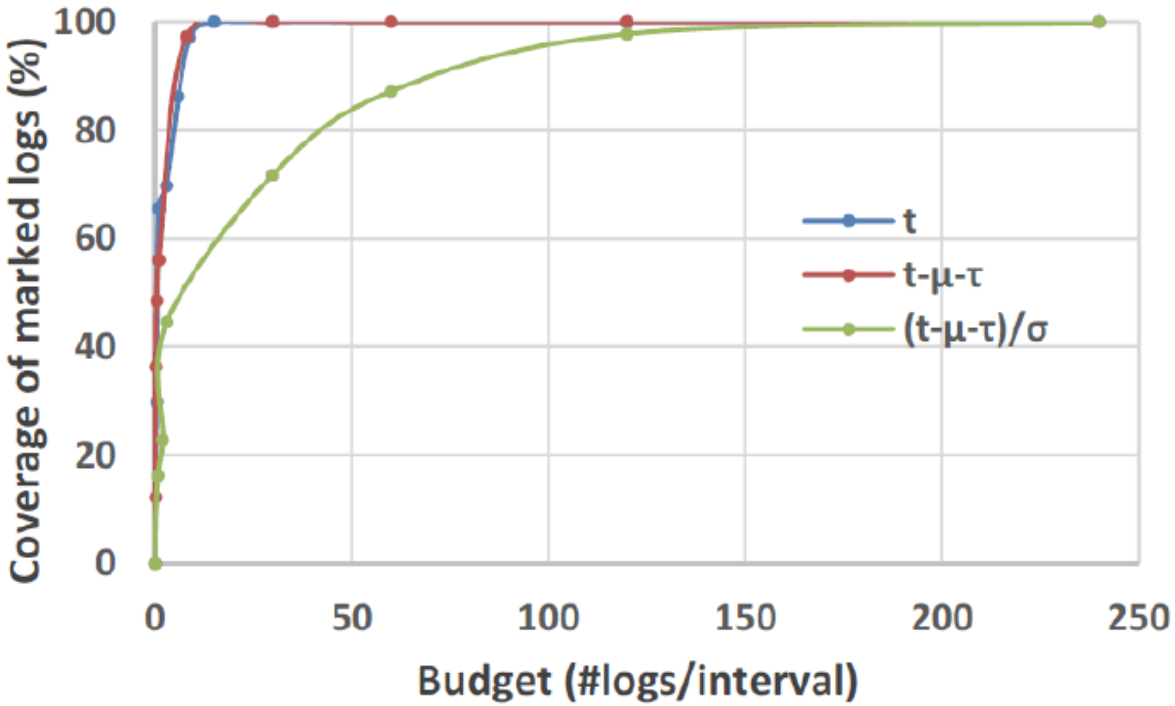
Evaluation results on I/O throughput



Comparison of logging throughput. (budget = 120 logs/interval)

Average reduction on logging throughput: >97%

Evaluation results on logging effectiveness



Logging effectiveness vs. budget size

Coverage close to ~100% when budget size reaching ~ 100 logs/interval (~2.5% sampling rate) for all three utility scores

Evaluation results on additional overhead

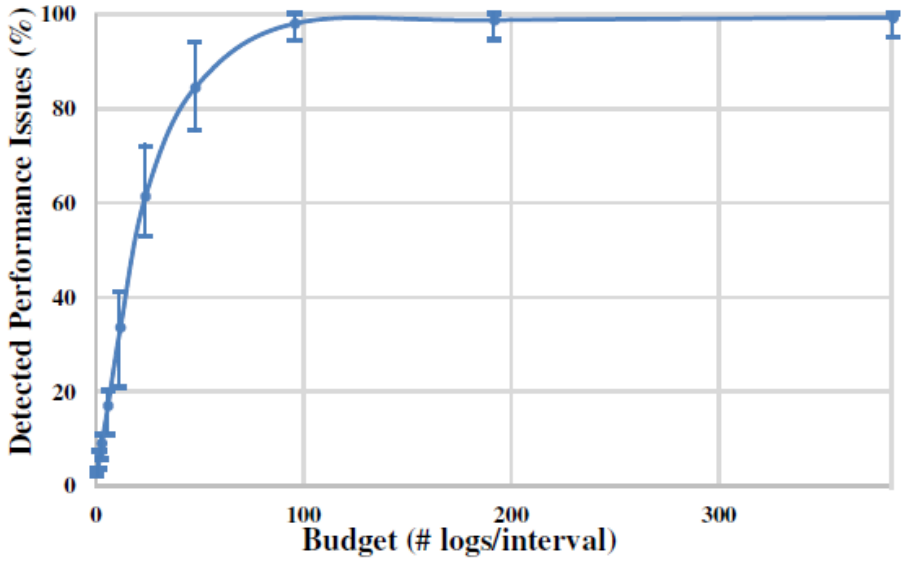
Logging system	Memory (GB)	CPU(%)
Log^2	4.74±(0.21)	63.4±3.0
Baseline	4.70±(0.25)	70.6±4.1

CPU and memory usage comparison of Log^2 vs. traditional logging system

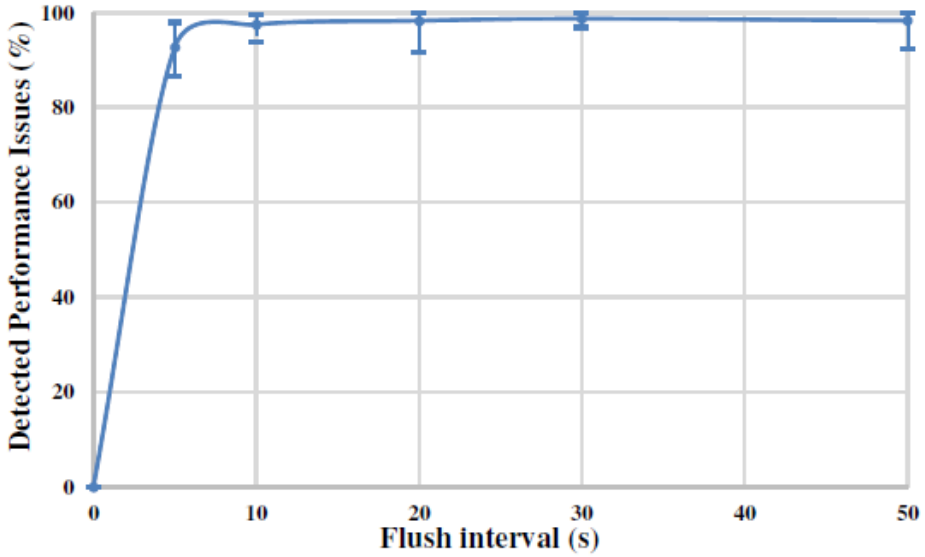
- Additional memory usage is not noticeable
- Average CPU usage of Log^2 is slightly lower than that of baseline logging system

Evaluation on Microsoft ServiceX

- Randomly selected real logs
- Performance hotspots labeling
  - Logs with top 0.3% longest execution time are identified as performance hotspots
- Results
  - Coverage quickly reaching 100% with small budget
  - ~98% coverage with ~0.77% sampling rate



Logging effectiveness vs. budget



Logging effectiveness vs. flush interval

Discussion and future directions

- Budget control for multiple services
- Supporting more types of performance analysis
- Multiple objectives
- Where to log
- Leveraging non-performance logs
- Extension to failure diagnosis

Conclusion

- We propose a cost-aware logging mechanism, Log^2, to make optimal "whether to log" decisions
  - Selectively recording useful logs
  - Compliant with budget
  - Self-adapt to system changes
- Experimental results show that Log^2 can control logging overhead while preserving logging effectiveness.

Microsoft Research
微软亚洲研究院

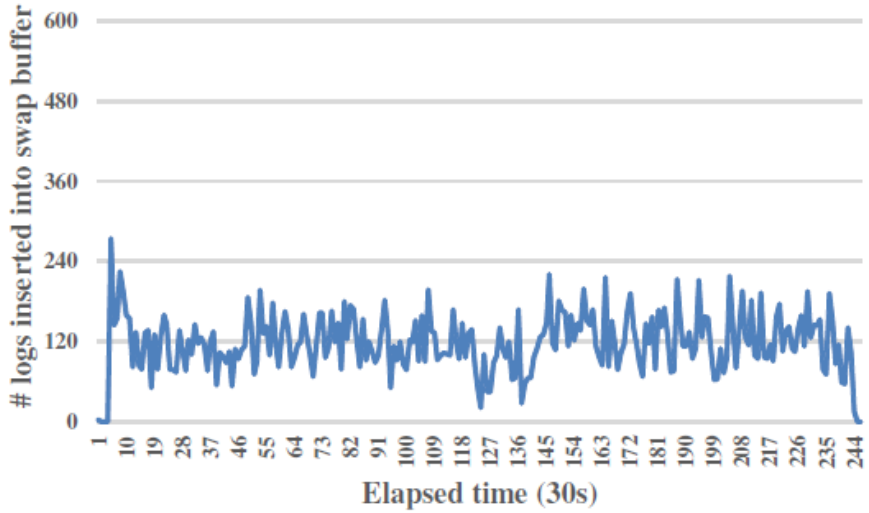
Thanks!

Q&A

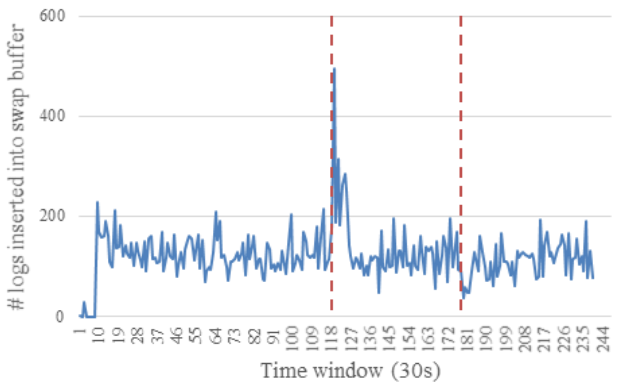
Implementation details

- Bounded memory usage
  - Maximum memory usage of Log^2 is configurable (currently set to 50MB)
  - Logs are discarded directly when memory usage reaches upper bound
- Handling system idle time
  - Overshoot avoidance
- Nested instrumentation
  - Thread-local stack is adopted
- Window size selection
  - Currently, Log^2 sets adjust interval to 30 seconds, same as flush interval

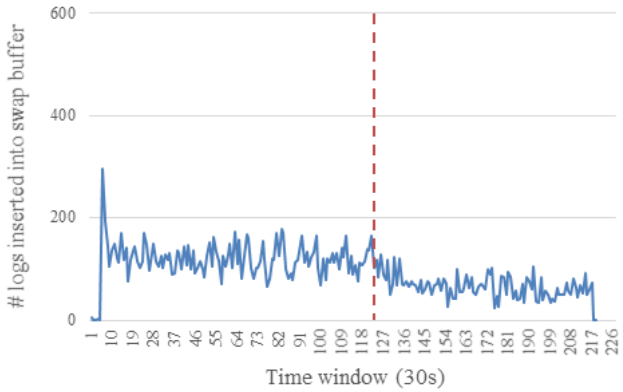
Additional experiment results



Dynamics of swap buffer size



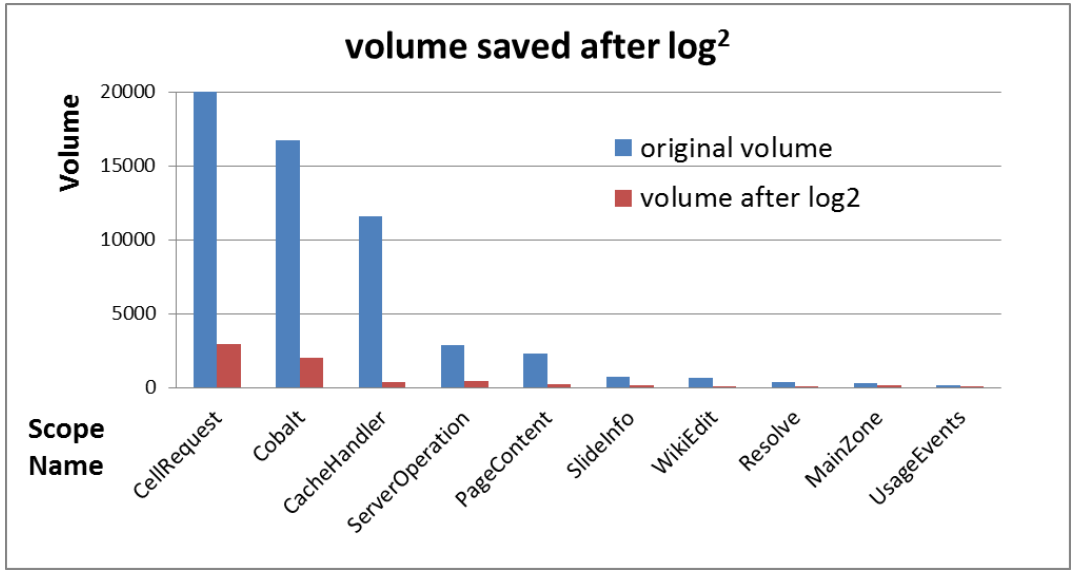
Buffer size vs. workload changes

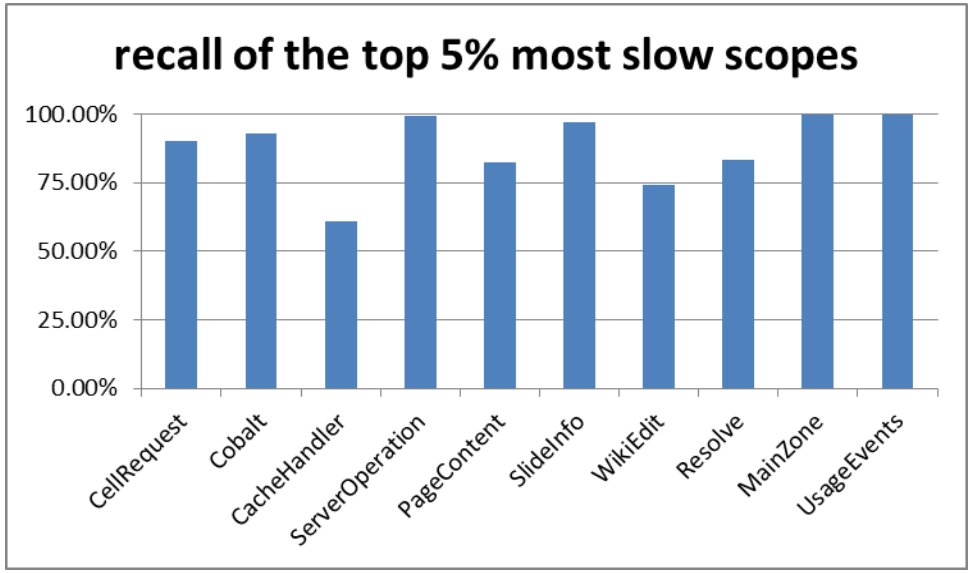


Buffer size vs. budget changes

Additional evaluation results on ServiceX

- Results





Total volume reduced: 90.1%

Illustration of effectiveness on top-5 “hot” monitored-scopes

Budget assurance cache

- Assuring budget compliance
- High efficiency data structure
  - Using swap buffer to reduce lock contention