# NanoLog: A Nanosecond Scale Logging System

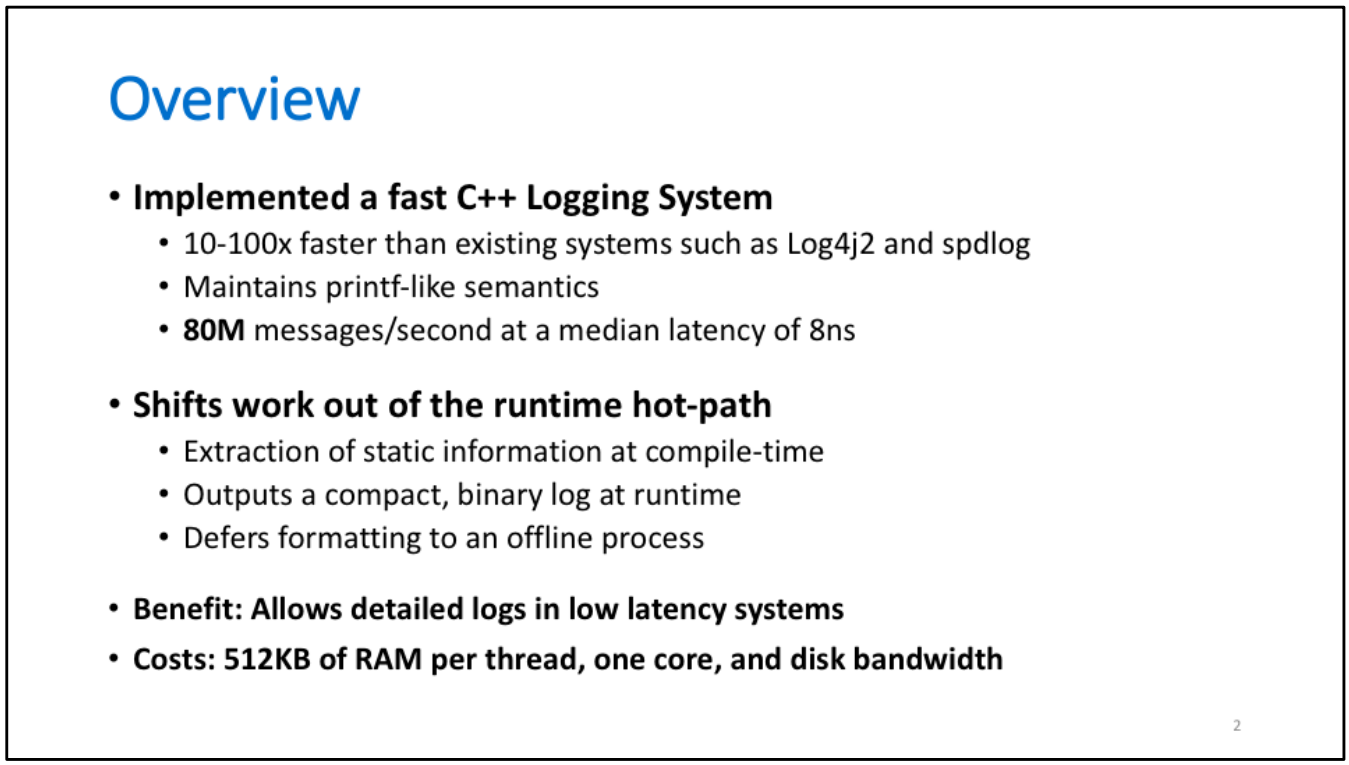
**Stephen Yang**, Seo Jin Park,
John Ousterhout

Stanford University

PLATFORMLAB

ATC 2018 1 July 12th, 2018

Hello, my name is Stephen Yang and I'm a PhD Candidate from Stanford University. Today, I'll be giving a talk on NanoLog, a nanosecond scale logging system that we developed for C++ to help debug the low latency systems in our lab.




NanoLog is a new logging system that's 1-2 orders of magnitude faster than existing solutions out there like Log4j2, spdlog, or Event Tracing for Windows all while maintaining a fairly intuitive printf-like API. The system has a throughput of up to 80 M log messages per second at a median latency of just 8 nanoseconds.

NanoLog achieves this insane level of performance by shifting work out of the **runtime hot path** into the compilation and post-execution phases of the application. More specifically, it slims down user log invocations by extracting static content at compile-time, rewrites the logging code to only emit dynamic information at runtime, and relies on a post processor to reflate the logs, but only if necessary. More on that later.

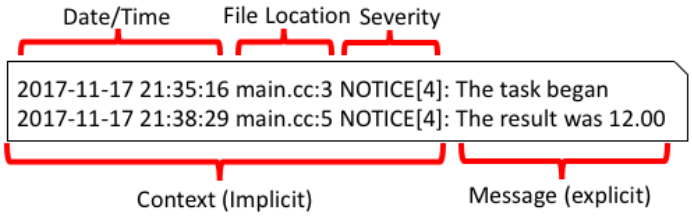
And what this buys us is the ability to log in more detail, log more often, and even do it in production where response times are critical. But this system does come with a cost of requiring about 0.5MB of RAM per user thread, one core for processing, and reliable disk bandwidth is a must.

# What is Logging?

- **printf-like messages with dynamic information**
  - Written at development time; output at runtime
  - Developer specifies severity, log message + dynamic data
  - Logging system inserts invocation date/time, file location

main.cc

```
1  void main() {
2    LOG(NOTICE, "The task began");
3    float result = doSomething();
4
5    LOG(NOTICE,
6      "The result was %4.2f", result);
```



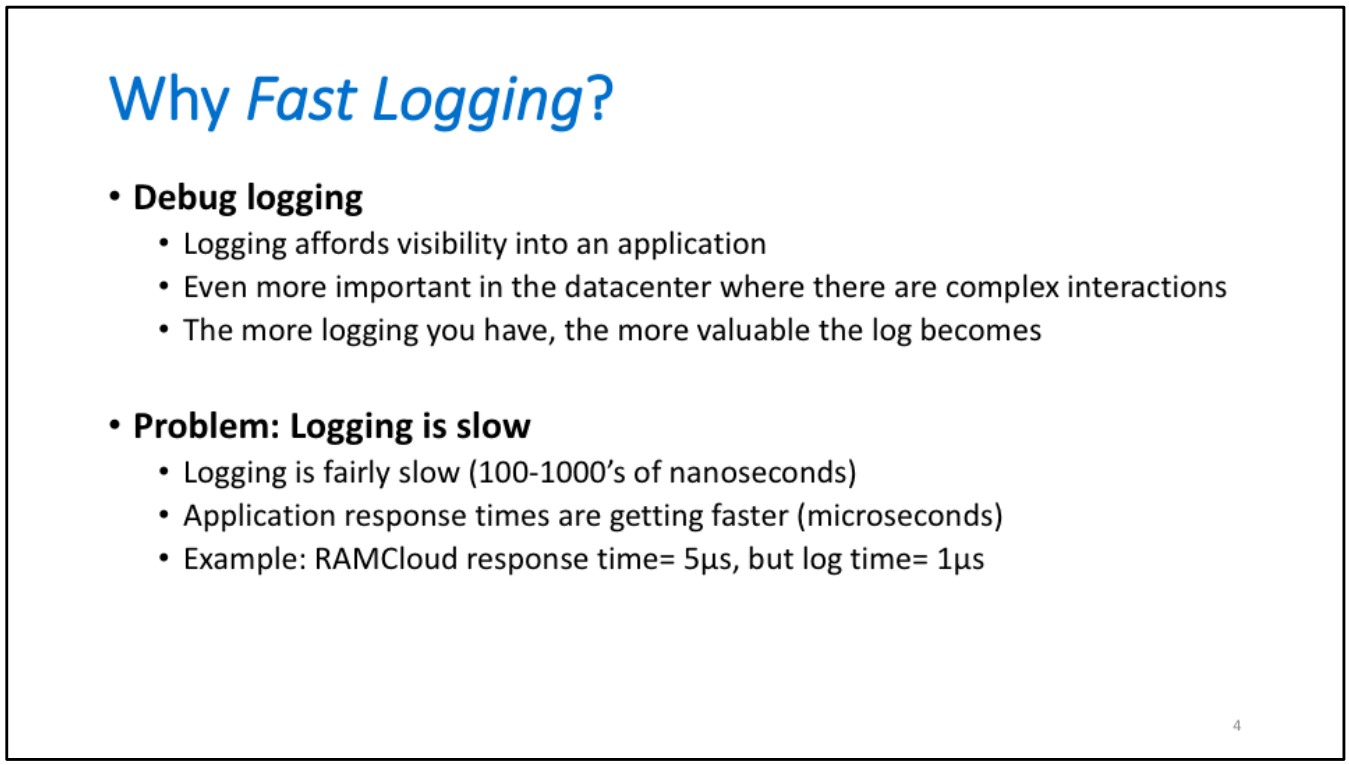

3

Before we begin, let me first clarify what logging means in the context of this talk. When I say logging, I mean log messages that are output by the application as the result of the developer specifying printf-like statements in the code.

So for example, in the left side of the slide, I have an application with two messages, one at line 2 to indicate a task began and one at line 5 to indicate the result of the task. When the application runs, the two statements execute, and it generates a log file similar to what we have on the right.

Each log statement produces a line of output that specifies when and where the log message occurred along with the original message specified by the developer. The logging system typically fills in the invocation time and file location automatically, so the developer only has to specify the severity and log message.

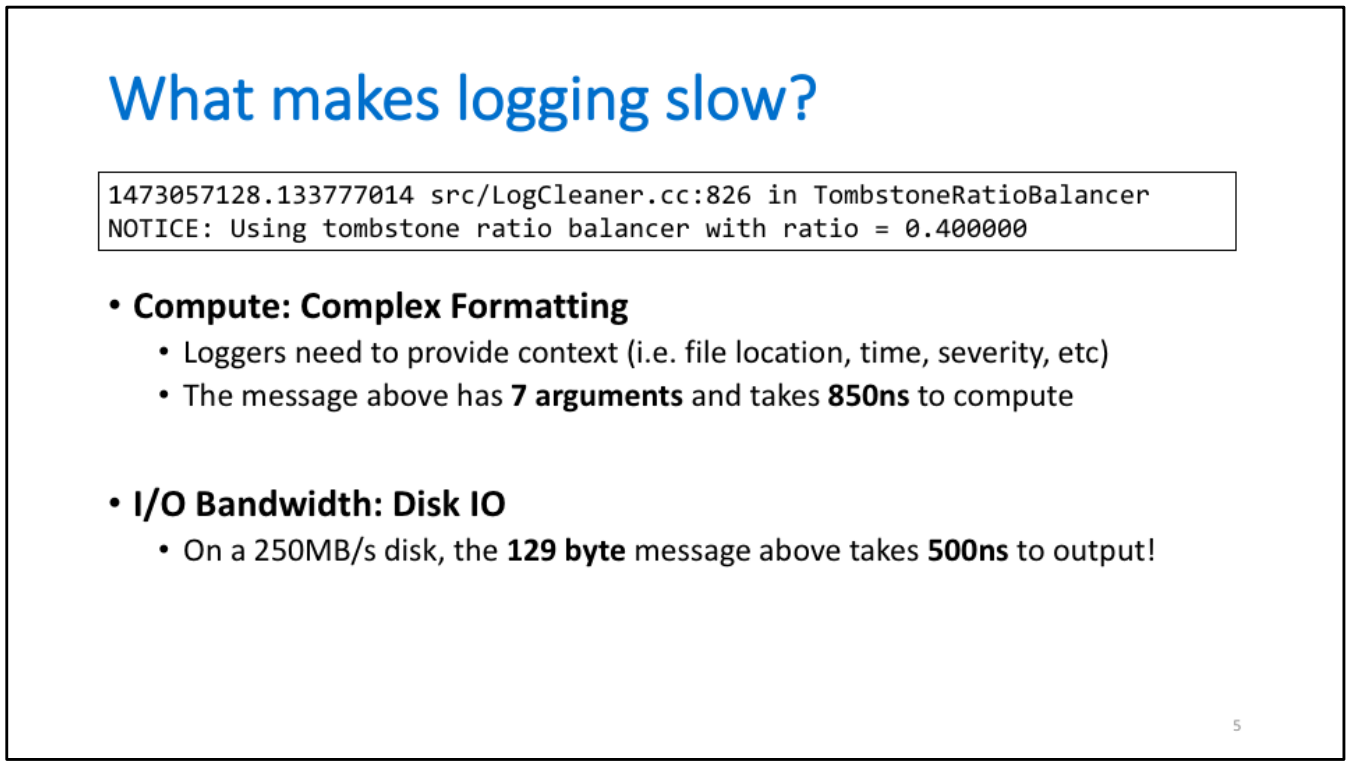
So when I say we've built a logging system that's 1-2 orders of magnitude faster, I mean that the code on the left executes significantly faster and the production of the log file on the right is deferred to a post-processor.




So at this point, you may be wondering, why logging? why is it important?

Well logging plays an extremely important role in debugging. It affords you visibility into what an application is doing at runtime and is an incredibly useful tool for root cause-analysis or tracing what went wrong in an execution. And it's becoming even more important in the datacenter, where there are so many interlocking processes and timing dependencies, that it may not make sense to debug these systems any other way, like interactively with GDB.
This is so important. In fact, there was an interesting conversation we had with a Google employee a year or two ago whereby they stated that at Google, everything needed to be instrumented at the request level or they simply won't use it.

However, the unfortunate fact is that logging is fairly expensive. In the systems I've measured, the median latency for the operation is in the order of hundreds of nanoseconds, and in some cases microseconds. While this wouldn't have been too bad a decade ago where millisecond responses times dominated, it's becoming a problem now as we move towards more granular computing whereby interesting computation is taking less and less time. And thus the relative overhead of logging is increasing.

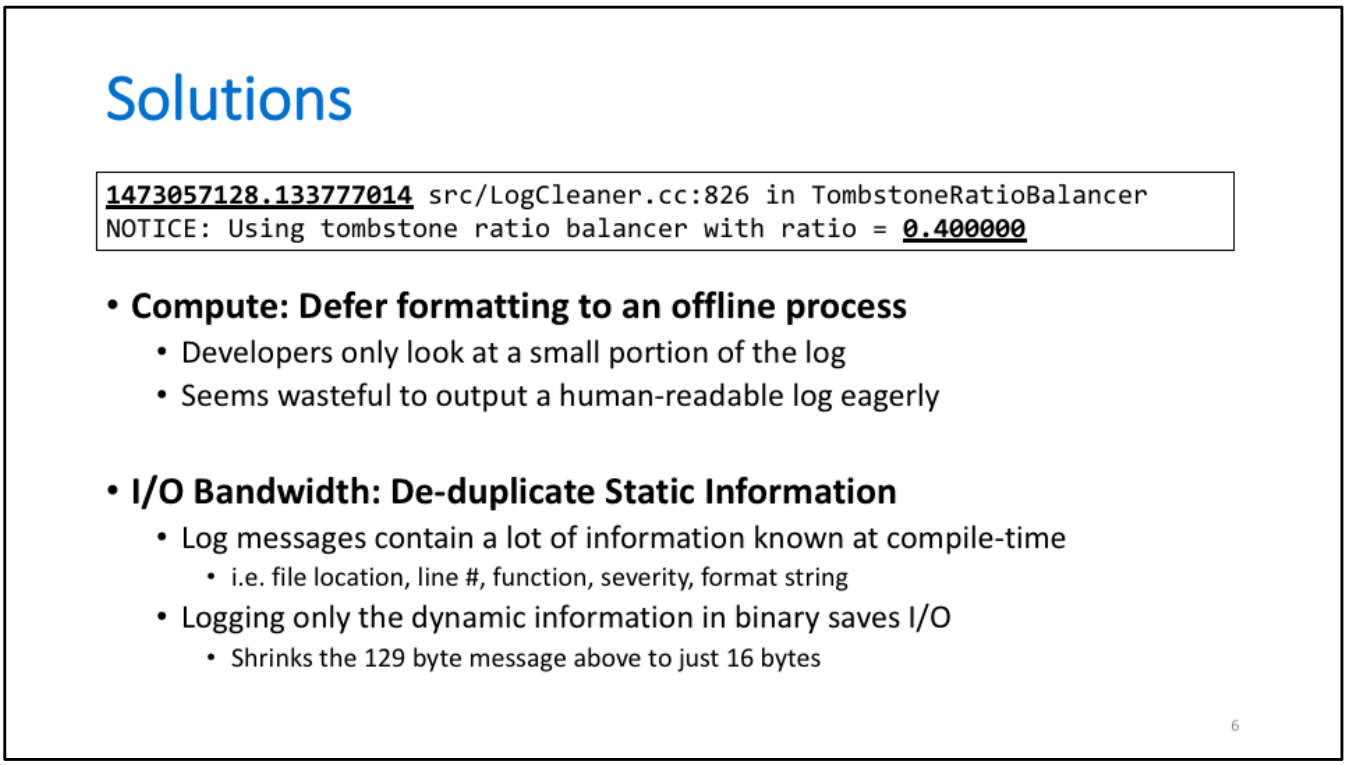



And so that begs the question, what makes logging so slow? Well, software overheads aside, the two **most expensive** operations a logging platform has to perform are (a) formatting the log message and (b) outputting the formatted message.

Take for example, this typical log message taken from the RAMCloud project. For context, the logger has to format the time, the filename, line number, the function, and the severity level associated with the log message along with the original message itself with its own specifiers. In total that's 7 parameters it has to format. Even running it in a tight loop with -O3 optimizations in C++, this operation alone takes **850ns**!

Then after all that work, the message STILL needs to be saved on disk or over the network which, on a 250MB/s disk, this 129byte message would take about **500ns** to output at least. That means, end to end, this SINGLE log message takes at least 1.3µs of CPU time.

So then that begs the question, can we improve on either of these operations? Well yes, trivially you could get faster hardware and these operations will naturally get faster, but I think there are a few tricks we can play to squeeze out **even more** performance.

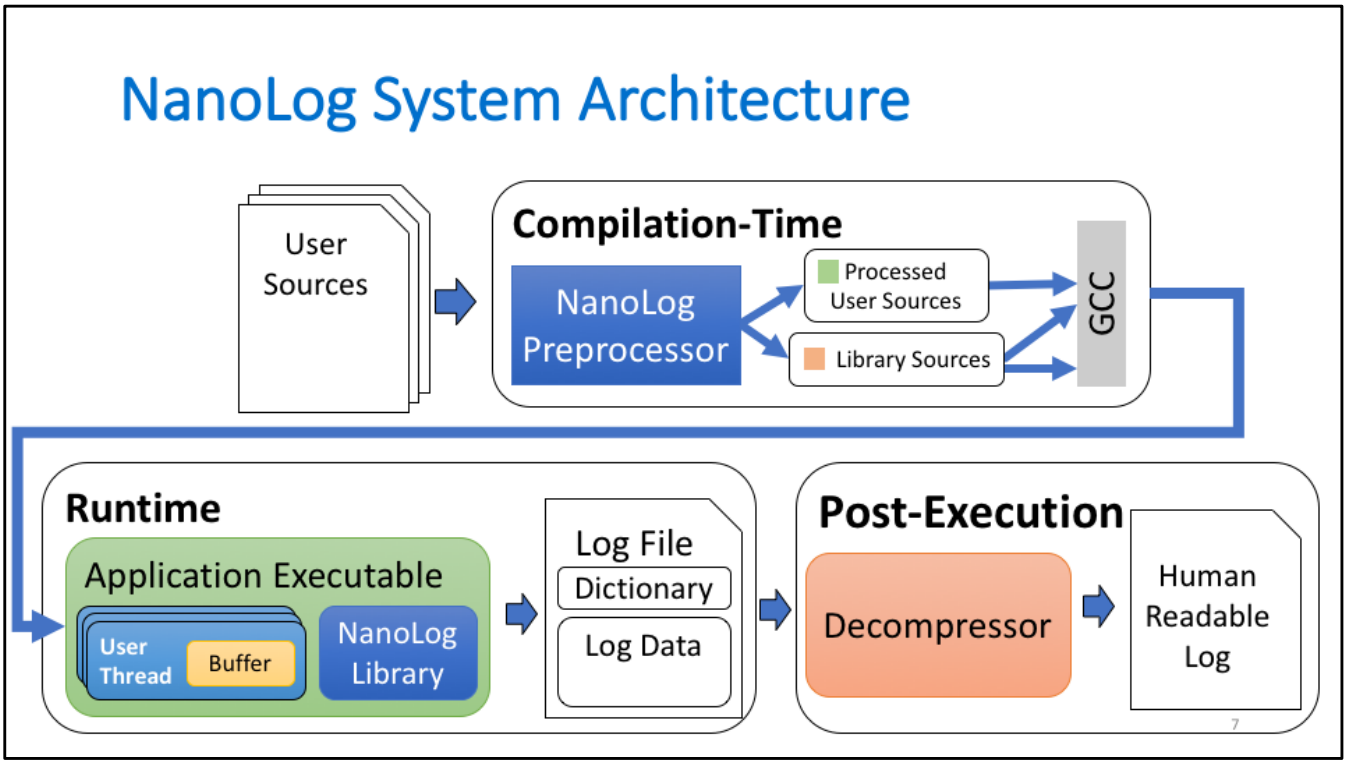



First, let me ask simple but surprisingly deceptive question: Does it even make sense to output the logs immediately in a human readable format? Or can we get away with outputting in an intermediate binary format and deferring the formatting to a later point? I mean if you think about it, a lot of the logs in production aren't even read by humans. They're almost always first consumed by computers, whether it be to aggregate key metrics or filter out error conditions, and THEN when something goes wrong do developers finally look at the log. So it seems outputting in a human-readable format eagerly is a little wasteful in the normal case.

And secondly, on the IO front, notice that there's actually a lot of information in the log message that's known at compile time. For example, in this log message, the file position, function, severity level, and even the user format string are known at compile-time… The only portions of this log message that's dynamic, is the time at the front and the 0.4 double at the end, which are underlined above.

Trivially, if we only outputted JUST these two numbers that would shrink our One Hundred 29 byte message to just 29 bytes, and if we take that one step further and output it in a binary format, that'd be 8 bytes for each parameter or 16 bytes total.

So then with these thoughts in mind, could we build a system that would speed up logging by shifting work out of the runtime? Well, yes. I've built it. It's called NanoLog. Let's talk about it.

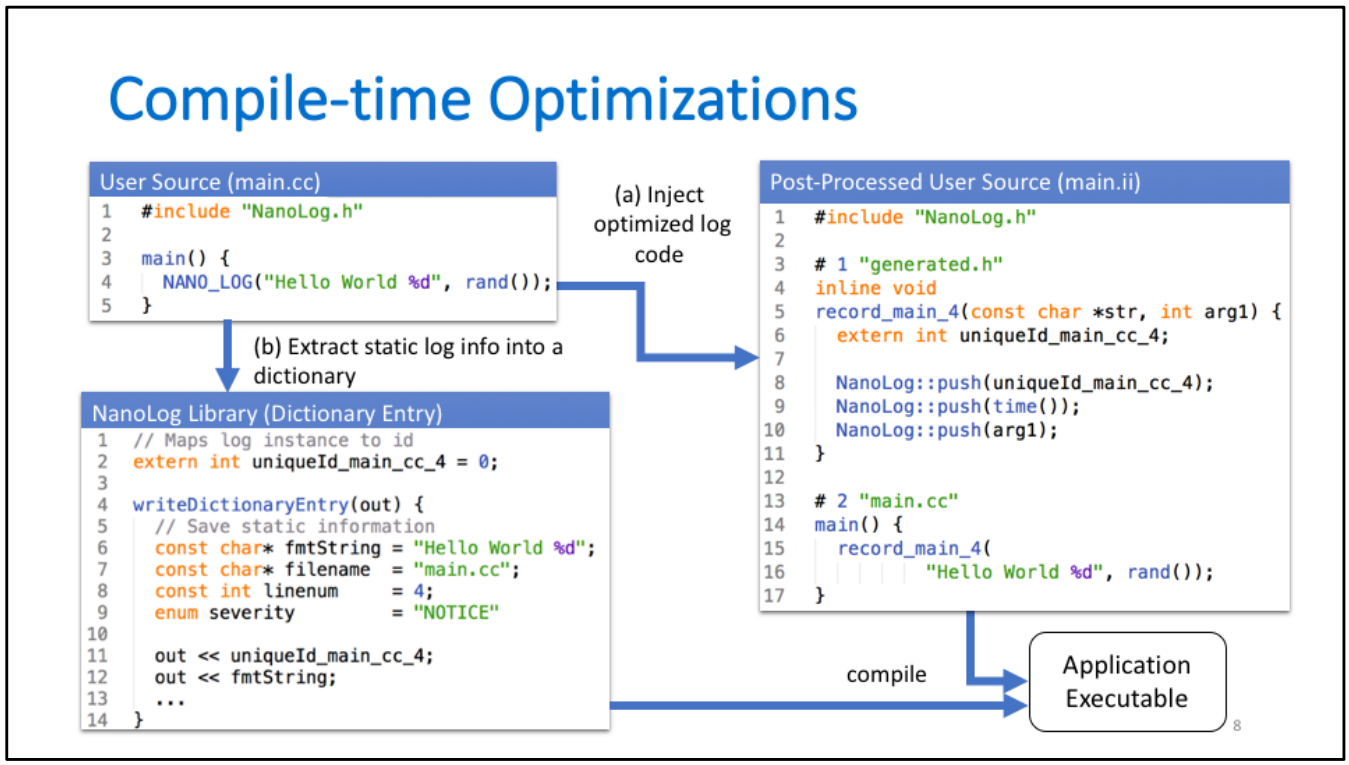



So this is the simplified architecture of NanoLog. How it differs from traditional logging systems is that it uses a preprocessor to modify the user sources at compile-time, emits a compact, binary log at runtime, and uses a decompressor to reinterpret the log at post-execution.

So going through it all, you start with the user sources on the top left, and they contain printf-like log statements. When you want to compile, the NanoLog Preprocessor will take in your sources, extract all the static log information into a dictionary, and rewrite the log statements to save only dynamic information, before passing it into gcc for compilation.

Then at runtime, the bottom left, the application interacts with the NanoLog Library to output the dictionary of static information just once at the beginning of the log file, which then frees all subsequent log invocations to log only the dynamic information. This special log file can then be passed into the post-processor, or decompressor, to make it human-readable again.

Now let's go through the system in more detail, stepping through how an application would go through this process.




As I mentioned earlier, the primary role of the NanoLog Preprocessor is to extract as much static information as possible from the log messages, and replace the user log invocations with more optimized code.

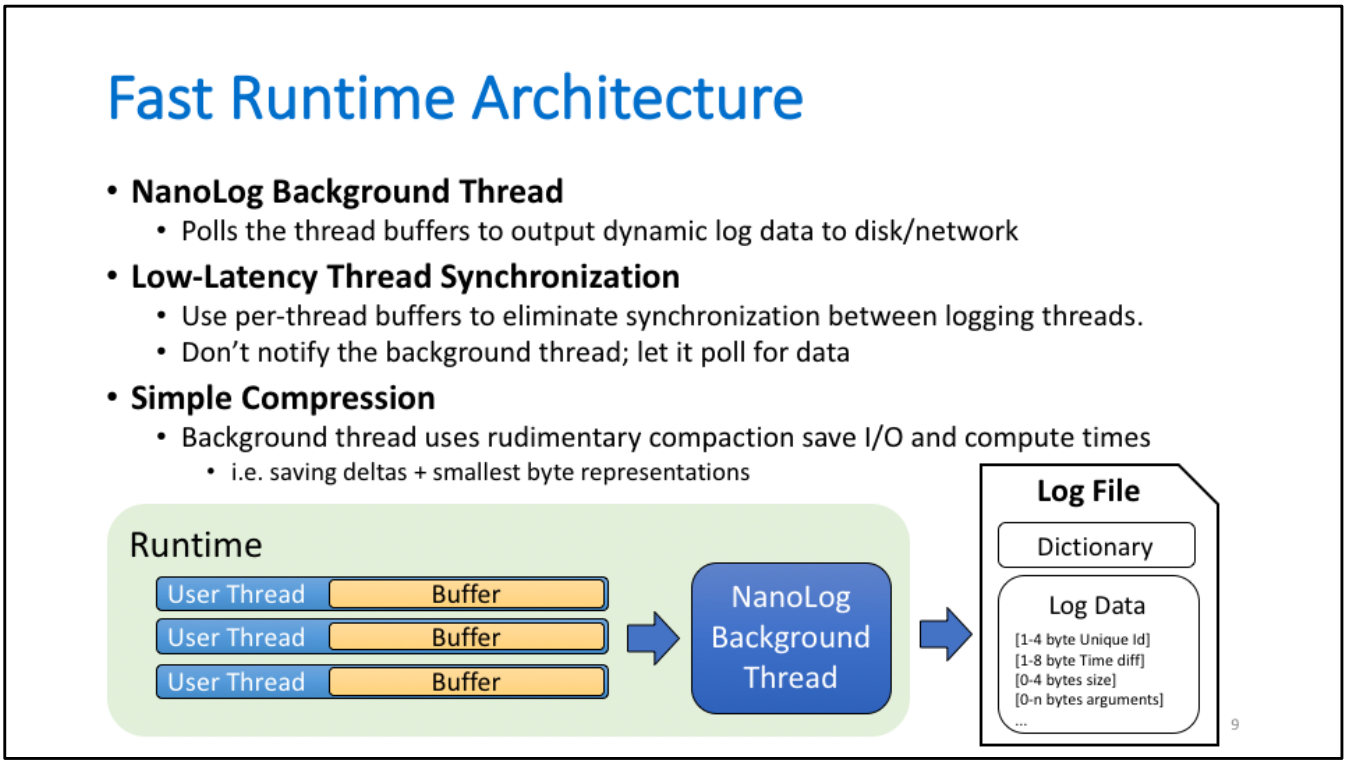
So stepping through the process, the original source is to the top left; it's just a simple program that prints "Hello World" followed by a random number.

After going through the preprocessor, it ends up looking a little bit like the file to the right. Here the preprocessor found our "Hello World" log invocation and injected a function to perform the minimal amount of work needed for that log, which is to save a symbolic reference for the static information, the time of invocation, and the log's argument, that random number. And it replaces the original log invocation in the source down here, with a call to this function instead.

The preprocessor then takes all the static information associated with this log message and generates a function to output a dictionary entry mapping the symbolic reference to the static information. This can be seen on the bottom left where the log message's format string, filename, line number, and severity level is in-lined into the function.

Going back to the broader picture, the dictionary function on the left is used by the runtime to output all the static information into the beginning of the log, and the function on the right generates the dynamic log data.

After this transformation is done to all the user source files, they're passed through gcc to generate the user application.



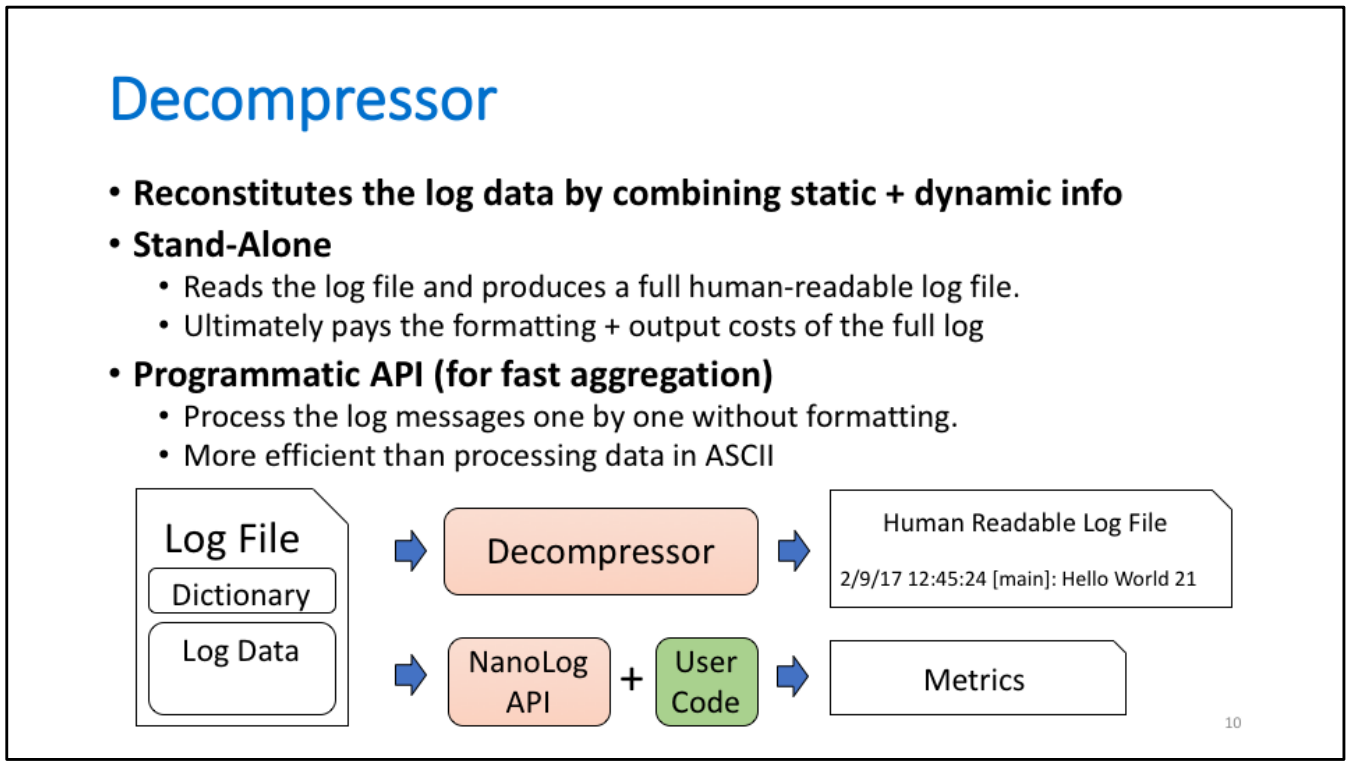

Now that we have our binary, let's see what happens in the runtime hot-path... Whenever the application threads execute the generated log code, they push dynamic data into some buffers, and a background thread will come by, poll the buffers, and output the contents along with the dictionary to disk.

The main consideration in this component is speed. How do we achieve low latency?

Well first, we isolate the application's logging threads from each other by using per-thread buffer queues. And this allows multiple threads to put data into the buffers without synchronization in the normal case. Then to avoid signaling costs, the logging threads also don't notify the background thread that there's data and the background thread instead has to poll for it. With these two techniques, synchronization only needs to occur when either the buffer is first created or the background thread can't keep up and the buffer fills up.

Speaking of the background thread, the story gets a bit more interesting with IO. On one hand, disk IO is expensive so it should try to compress the data as much as possible to reduce the IO time, BUT if it spends too much time on compression the buffers will back up and latency will increase again. So the compromise here is to only do rudimentary compaction such as only recording deltas or finding the smallest byte representation for integers. This results in a file that's much smaller with minimal computation. You can find more information about how we do compaction in the paper.

Okay, so at this point we have the executed of the user application and it outputted a compacted log. Let's see what we can do with it.

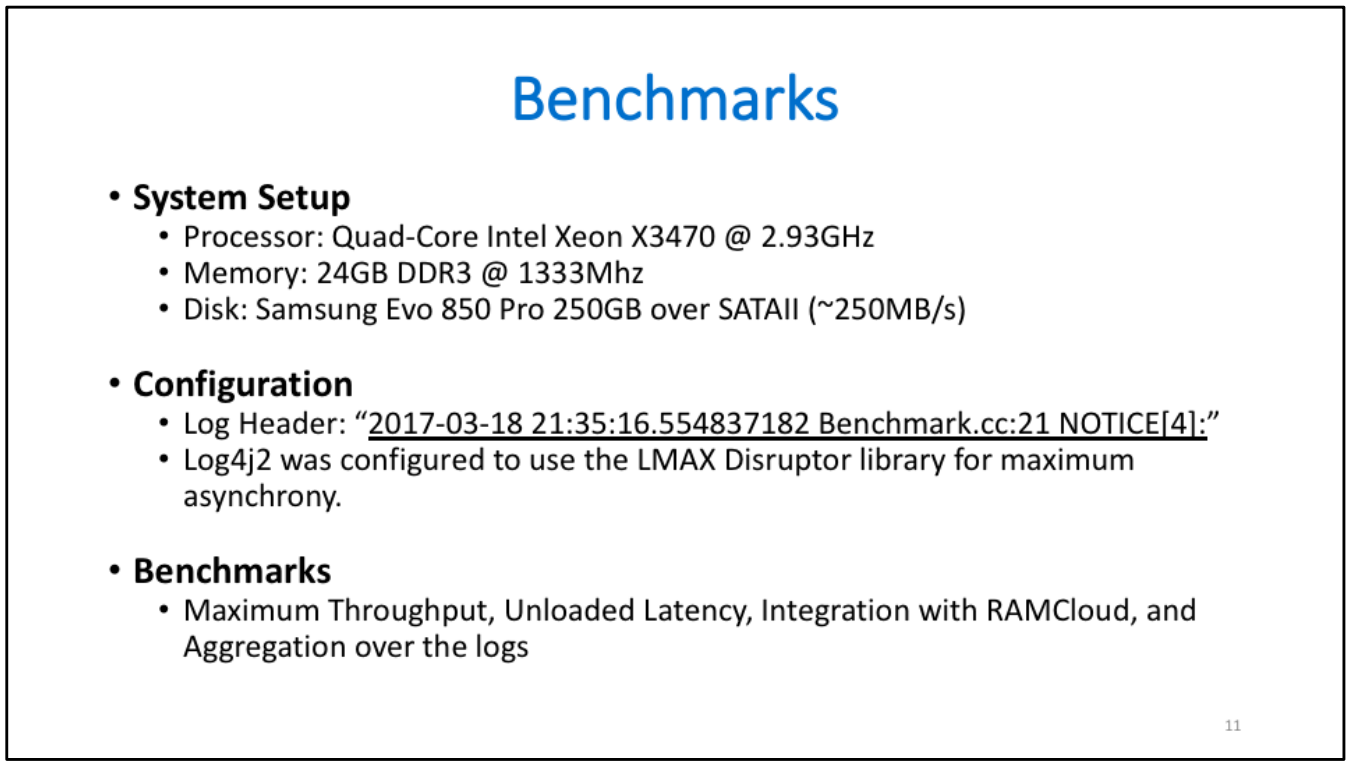



The final component of NanoLog is the post-processor or decompressor. Its primary purpose is to reconstitute the log data by parsing the dictionary at the beginning of the log and interpreting the dynamic log data that follows. In other words, it recombines the static information we separated at compile-time with the dynamic information generated at runtime.

There are two ways to use the decompressor. Used as a stand-alone application, it will reconstitute the full-human readable log and ultimately pays the two costs we've deferred so far, which are formatting the log message and outputting the messages in its full format.

The decompressor also offers a programmatic API to consume the log messages without formatting the them first. With this API, you can process the log messages one by one and query its static and dynamic arguments in a binary format, which is way more efficient that processing them in an ASCII. And we'll see the benefits of doing so in a microbenchmark later.

To summarize, the NanoLog system rips out the static log information at compile time, injects code to log only the essentials to disk at runtime, and then users can run offline executable to make the log human-readable again. This shifts computation out of the hot runtime path and into the compilation and post-execution phases.

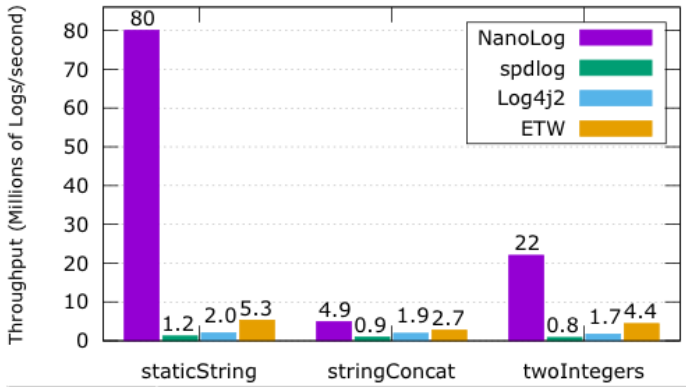



Okay! Onto the more interesting portion of the talk, benchmarks! How do the techniques we've talked about actually perform? For these tests, we're running on a Intel Xeon quad core processor with hyper threading. The machine has 24GB of RAM and more importantly an SSD that averages about 250MB/s in write throughput.

All the logging systems I measured against were configured to output the log messages with the time/date, filename, line number, severity, and thread id associated with the log invocation. It looks a little something like what I have underlined on the slide and adds an extra 50 bytes to each log message for traditional logging systems.

The benchmarks I'm about to show are all in the paper, but they may be simplified here to save time.

# Microbenchmark - Throughput




## Evaluation

- Repeated logged 1 message with no inter-log delay in a thread
- Varied the number of logging threads to maximize throughput

## Conclusions

- NanoLog always faster
- Even better with fewer dynamic arguments.

| ID | Example Output | NL Msg Size |
|---|---|---|
| staticString | Starting backup replica garbage collector thread | 3-4 Bytes |
| stringConcat | Opened session with coordinator at basic+udp:host=192.168.1.140,port=12246 | 43 Bytes |
| twoIntegers | buffer has consumed 1032024 bytes of extra storage, current allocation: 1016544 bytes | 10 Bytes |

underlined indicates dynamic data

12

The first thing we did was measure the throughput of NanoLog vs. existing logging systems like spdlog, Log4j2, and Event Tracing for Windows. In this experiment, we repeatedly logged 1 message back-to-back with no delay in single logging thread and then ramped up the number of logging threads until the peak throughput was achieved for each system.

We repeated the experiment for 3 different log messages and the results are shown in the figure to the left. On the y-axis, we have the peak throughput in millions of log messages/second, so higher is better. The clusters of bars represent different log messages used with a table below describing what they are. In the table, the center column shows the log message and the dynamic data is underlined. Finally, the different colors in the graph represent different logging systems with Purple being NanoLog.

And the first thing to notice is that NanoLog is consistently faster than the other systems by 3-140x, depending on the log message used. The primary reason for this is because we're logging at such a high rate that all systems, including NanoLog, fill up their internal buffers and have to flush to disk. And since NanoLog's output is both in binary and compacted vs. the full ASCII in other systems, NanoLog performs significantly better.

Digging a little deeper we find that NanoLog performs better with fewer dynamic arguments, and this is makes sense since the primary speedup mechanism for NanoLog is removing static data. So it performs the best when the message is completely static (like what we have in the left most cluster), the worst when there's a lot of dynamic data (like what we have in the middle where a log message that contains a large 39-byte string). And the right most column is what we consider average, where the user logs a few numbers.

Overall NanoLog is pretty performant on throughput. Now let's look at latency.

## Microbenchmark - Unloaded Latency

- **Goal: Measure minimum processing delay**
- **Setup:**
  - 100,000 iterations of logging 1 log message with a 650ns delay between each to eliminate blocking due to I/O
  - Percentile times reported below in nanoseconds

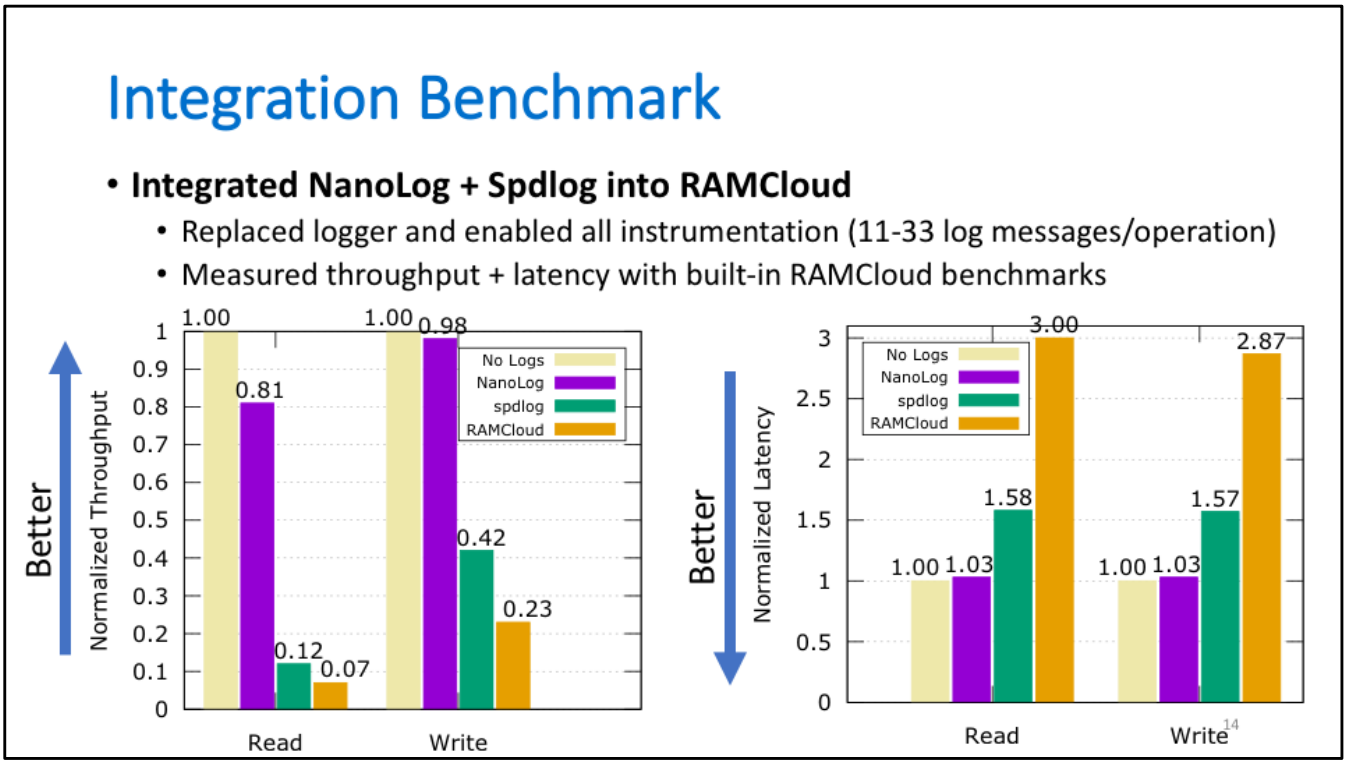
| System | NanoLog | spdlog | Log4j2 | ETW | | NanoLog | spdlog | Log4js | ETW |
|---|---|---|---|---|---|---|---|---|---|
| Log Msgs | Median Latency (ns) | | | | | 99.9 Percentile Latency (ns) | | | |
| staticString | **8** | 230 | 192 | 180 | | **33** | 473 | 1868 | 726 |
| stringConcat | **8** | 436 | 230 | 208 | | **33** | 1614 | 6171 | 2954 |
| twoIntegers | **7** | 674 | 160 | 200 | | **44** | 1335 | 1992 | 761 |

13

Next, we wanted to measure the time delay experienced by the application when it invokes the logging API. To do this, we measured the time to log the same 3 log messages from before, but only with a single thread this time and with a 650ns delay between each log message to ensure no blocking occurs due to I/O. In the table below, left chunk of numbers show the median invocation latency and the right chunk of numbers show the 99.9$^{th}$ percentile latencies. NanoLog is in the column that's bolded.

The important trend to notice is that NanoLog is extremely fast. In all cases, its invocation latency is over an order of magnitude faster than the other systems. At the median, NanoLog takes around 7-8 nanoseconds per log message whereas the other systems start at at least 160 nanoseconds. A pretty big difference. At the 99.9$^{th}$ latency to the right, NanoLog still maintains a latency within tens of nanoseconds ranging from 33-44, while the other systems start dipping in into the multiple microsecond range.

Comparing the two sides, one cool thing to notice is that even at the 99.9$^{th}$ percentile, NanoLog is still faster than the median of the other systems! Pretty cool right?



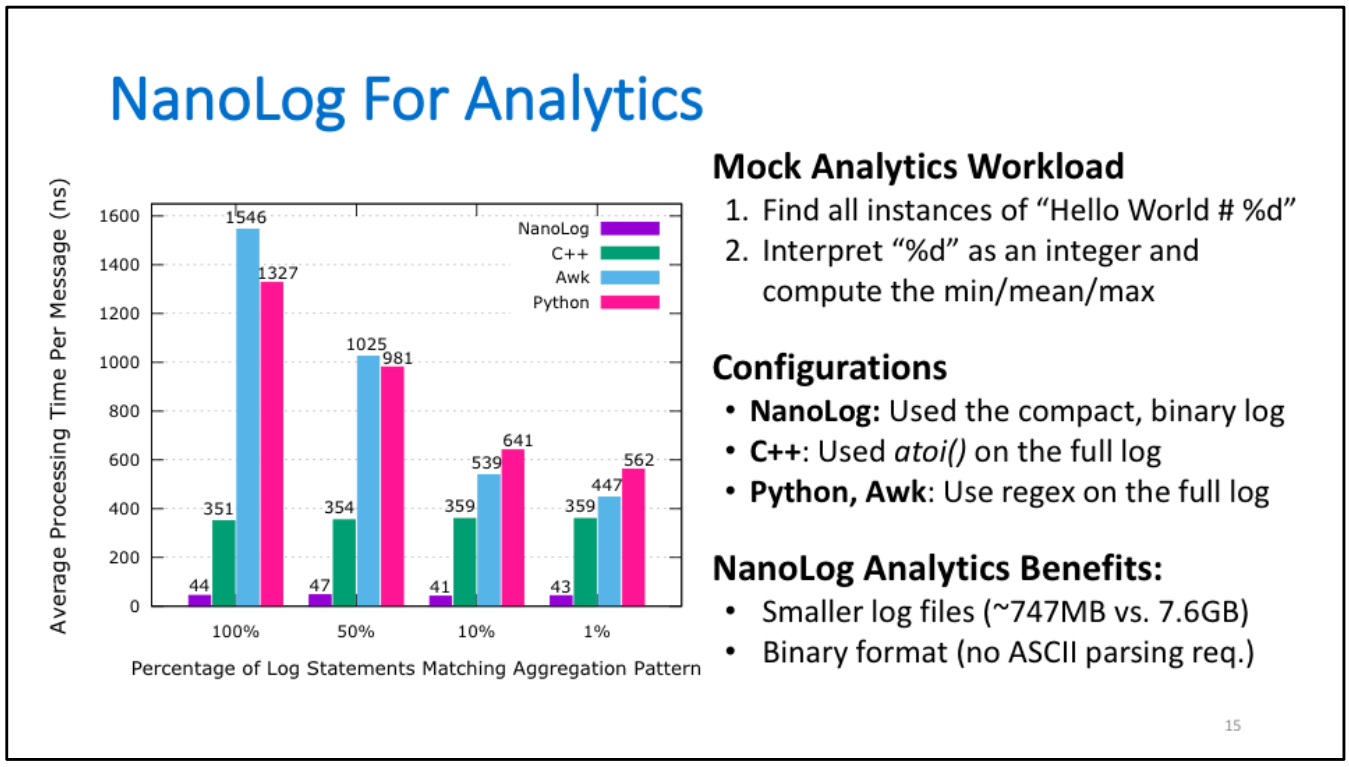

Okay, so at this point you may be thinking: great, you have microbenchmarks, but how does NanoLog actually perform in a real system with a real workload? To answer this question, we took an open-source, low-latency key value store, RAMCloud, and replaced it's internal logger with either NanoLog or Spdlog. We then measured the performance of RAMCloud WITH RAMCloud's own benchmarks and enabled all existing verbose logging and instrumentation already in RAMCloud. We didn't add any new log statements or benchmarks of our own.

On the bottom left, we have the throughput of RAMCloud with different logging systems and on the right we have the same for latency. The two clusters in each figure represent read/write performance and the different colors represent different systems. Yellow is the baseline with no logging, purple is NanoLog, green is spdlog, and orange is using RAMCloud's internal logger. All results are normalized to RAMCloud with logging completely disabled, i.e. the yellow bar.

And the thing to notice is that NanoLog performs the best in all cases. On the left, NanoLog retains 98% of the write throughput performance while the other systems drop it to less than half. With read throughput, NanoLog drops a bit more to 81% of the original, but that's far better than spdlog degrading performance down to a tenth of the original, making the system practically unusable.

On the right, we have normalized latency and it shows a similar story. NanoLog only imposes a 3% increase in read/write latency, while the other systems increase it by at least 55%

Overall, NanoLog imposes a minimal performance impact when compared to traditional logging systems, even when integrated into low-latency applications.

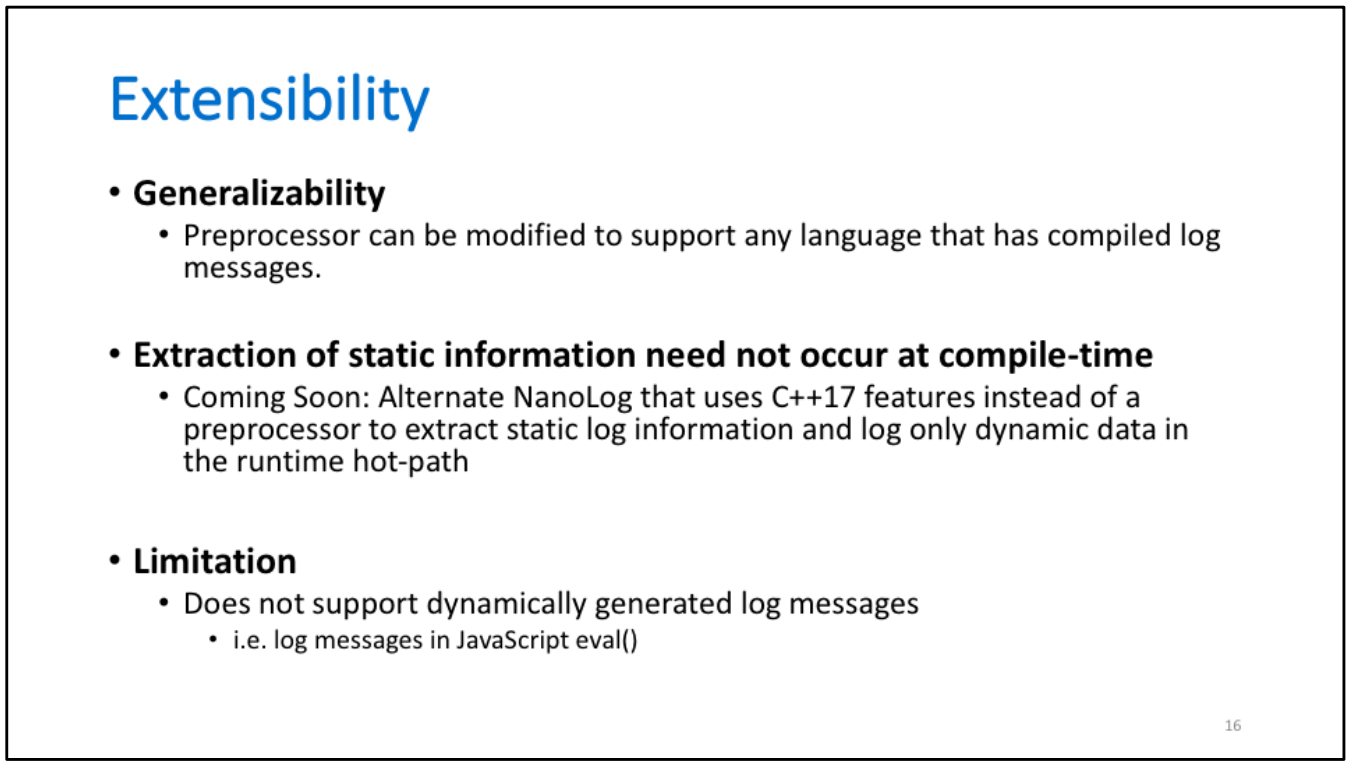



So we've seen the benefits of NanoLog at runtime, what about at post-execution? Well, I made the claim earlier that NanoLog's compact, binary logs are great for analytics, or consumption by computers. So to demonstrate this, I created a benchmark that mimics what an analytics application might do: which is find all instances of a log message matching "Hello world " followed by a number, and perform a minMeanMax aggregation on that number.

I implemented this in 4 ways: once with the NanoLog Decompressor which operated on the compact, binary logs, and once in C++, Awk, and Python which operated on the full, human readable logs. The results are shown on the figure to the left. The y-axis measures the average time to process a single log message (so lower is better), the clusters of bars vary how often our target message occurs in the log (so 100% means the entire log file matches "hello world"), and the different bars represents different systems with NanoLog in Purple again.

The primary thing to notice is that again, NanoLog is significantly faster; it has a average processing time of just 44 nanoseconds per log message vs. at least 350 nanoseconds by the other methods. The primary speedup comes from NanoLog's compact, binary logs, which are both smaller in size (about about $1/10^{th}$ of the human-readable log), and already contain log data in a binary format, meaning we don't need to perform expensive ASCII parsing to ascertain the value.

Overall, NanoLog's compact, binary log has benefits that extend beyond the runtime, and shows great promise for log analytics as well.

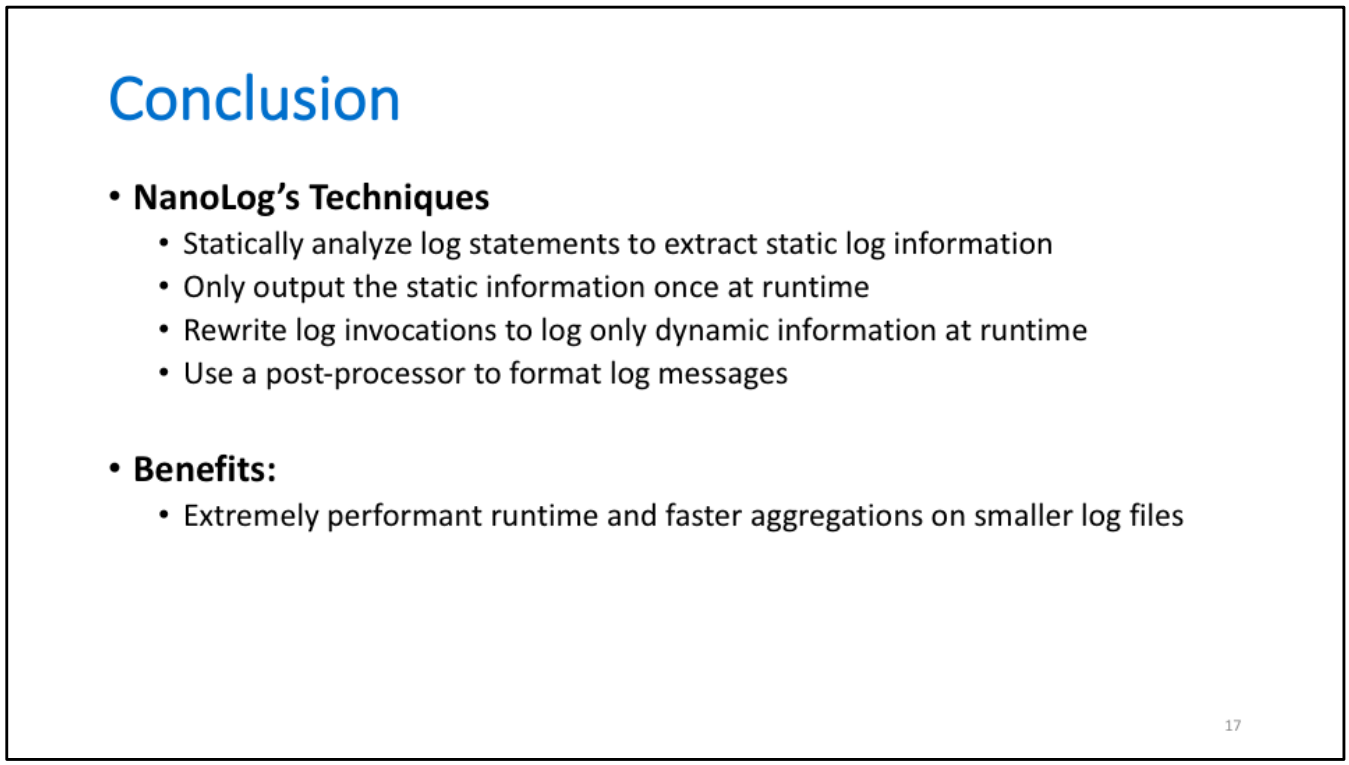



And that's all the time we have for the benchmarks. Before I start concluding, there's one more thought I'd like to leave you with, which is...

While I've spent most of the time describing the C++ implementation we have, the ideas of NanoLog are quite portable. The preprocessor can be modified to support any language that uses compiled log messages (like Java), and formatting could be deferred.

Furthermore, the deduplication of static log information doesn't necessarily need to occur at compile-time, it can be done at runtime with a small performance penalty. In fact, there's going to be an implementation of NanoLog coming to the GitHub repository in the next month or so that uses C++17 features instead of a preprocessor achieve similar goals.

The only limitation of NanoLog is that it requires some portion of the log message to be static, so dynamically generated log messages don't work that well.




So in conclusion, NanoLog is a new logging system that is over an order of magnitude faster than existing logging systems. It speeds up logging by using a preprocessor to remove static log information at compile-time, injecting optimized code to log only the dynamic information in the runtime hot path, and deferring formatting of log messages to a post-execution application.

The benefits of using this system are that it is extremely performant at runtime and can provide fast aggregations at post execution. The only caveat is that you may not have your logs immediately in a human-readable format.


NanoLog is an open-source project and you can give the system a try at github.com/PlatformLab/NanoLog and you can e-mail me any questions at the e-mail on the slide.

And with that, I will now open the floor up for comments and questions.

# NanoLog: A Nanosecond Scale Logging System

**Stephen Yang**, Seo Jin Park,

John Ousterhout

Stanford University

PLATFORMLAB

# Overview

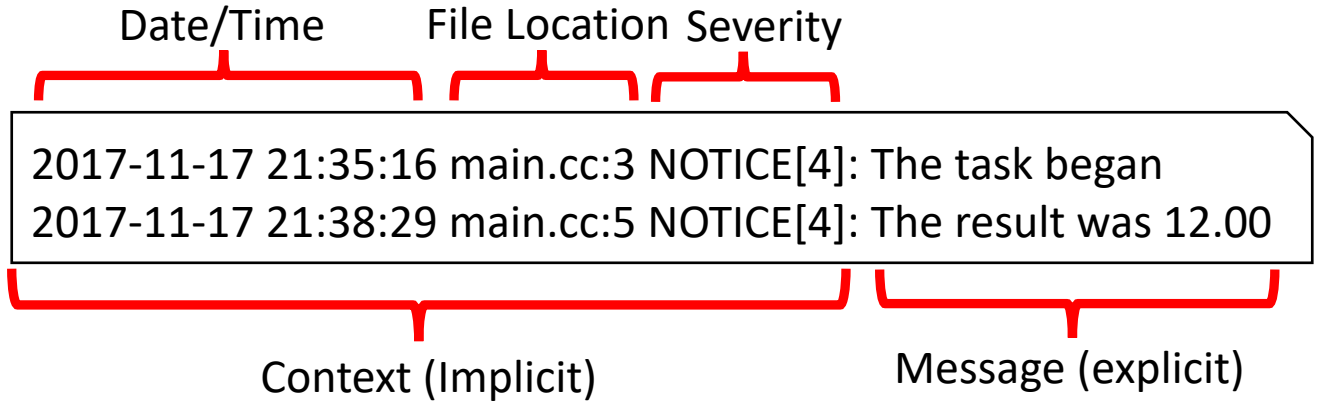
- **Implemented a fast C++ Logging System**
  - 10-100x faster than existing systems such as Log4j2 and spdlog
  - Maintains printf-like semantics
  - **80M** messages/second at a median latency of 8ns
- **Shifts work out of the runtime hot-path**
  - Extraction of static information at compile-time
  - Outputs a compact, binary log at runtime
  - Defers formatting to an offline process
- **Benefit: Allows detailed logs in low latency systems**
- **Costs: 512KB of RAM per thread, one core, and disk bandwidth**

# What is Logging?

- **printf-like messages with dynamic information**
  - Written at development time; output at runtime
  - Developer specifies severity, log message + dynamic data
  - Logging system inserts invocation date/time, file location

**main.cc**

```
1  void main() {
2    LOG(NOTICE, "The task began");
3    float result = doSomething();
4
5    LOG(NOTICE,
6      "The result was %4.2f", result);
```



# Why *Fast Logging*?

- **Debug logging**
  - Logging affords visibility into an application
  - Even more important in the datacenter where there are complex interactions
  - The more logging you have, the more valuable the log becomes

- **Problem: Logging is slow**
  - Logging is fairly slow (100-1000's of nanoseconds)
  - Application response times are getting faster (microseconds)
  - Example: RAMCloud response time= 5µs, but log time= 1µs

# What makes logging slow?

```
1473057128.133777014 src/LogCleaner.cc:826 in TombstoneRatioBalancer
NOTICE: Using tombstone ratio balancer with ratio = 0.400000
```

- **Compute: Complex Formatting**
  - Loggers need to provide context (i.e. file location, time, severity, etc)
  - The message above has **7 arguments** and takes **850ns** to compute

- **I/O Bandwidth: Disk IO**
  - On a 250MB/s disk, the **129 byte** message above takes **500ns** to output!

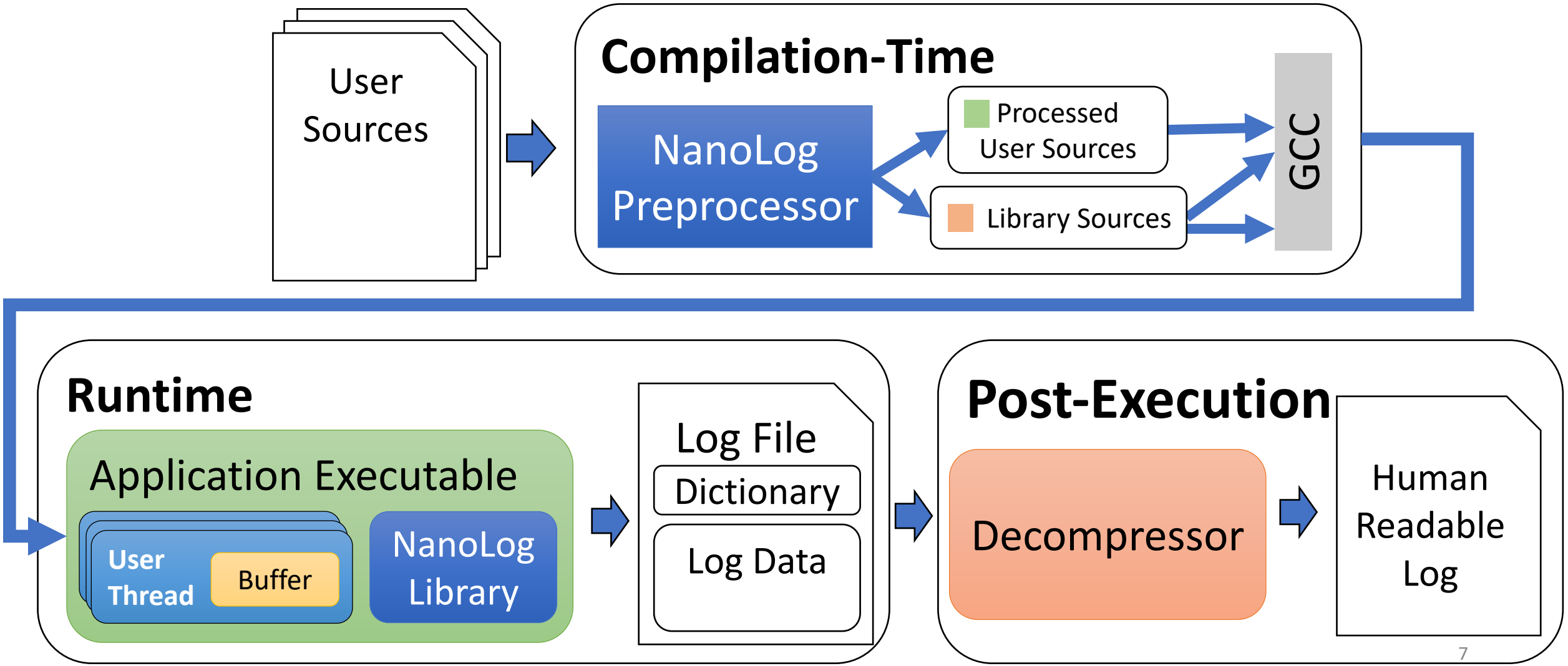
# Solutions

```
1473057128.133777014 src/LogCleaner.cc:826 in TombstoneRatioBalancer
NOTICE: Using tombstone ratio balancer with ratio = 0.400000
```

- **Compute: Defer formatting to an offline process**
  - Developers only look at a small portion of the log
  - Seems wasteful to output a human-readable log eagerly

- **I/O Bandwidth: De-duplicate Static Information**
  - Log messages contain a lot of information known at compile-time
    - i.e. file location, line #, function, severity, format string
  - Logging only the dynamic information in binary saves I/O
    - Shrinks the 129 byte message above to just 16 bytes

# NanoLog System Architecture



# Compile-time Optimizations

**User Source (main.cc)**

```
1  #include "NanoLog.h"
2
3  main() {
4    NANO_LOG("Hello World %d", rand());
5  }
```

(a) Inject optimized log code

**Post-Processed User Source (main.ii)**

```
1  #include "NanoLog.h"
2
3  # 1 "generated.h"
4  inline void
5  record_main_4(const char *str, int arg1) {
6    extern int uniqueId_main_cc_4;
7
8    NanoLog::push(uniqueId_main_cc_4);
9    NanoLog::push(time());
10   NanoLog::push(arg1);
11 }
12
13 # 2 "main.cc"
14 main() {
15   record_main_4(
16                 "Hello World %d", rand());
17 }
```

(b) Extract static log info into a dictionary

**NanoLog Library (Dictionary Entry)**

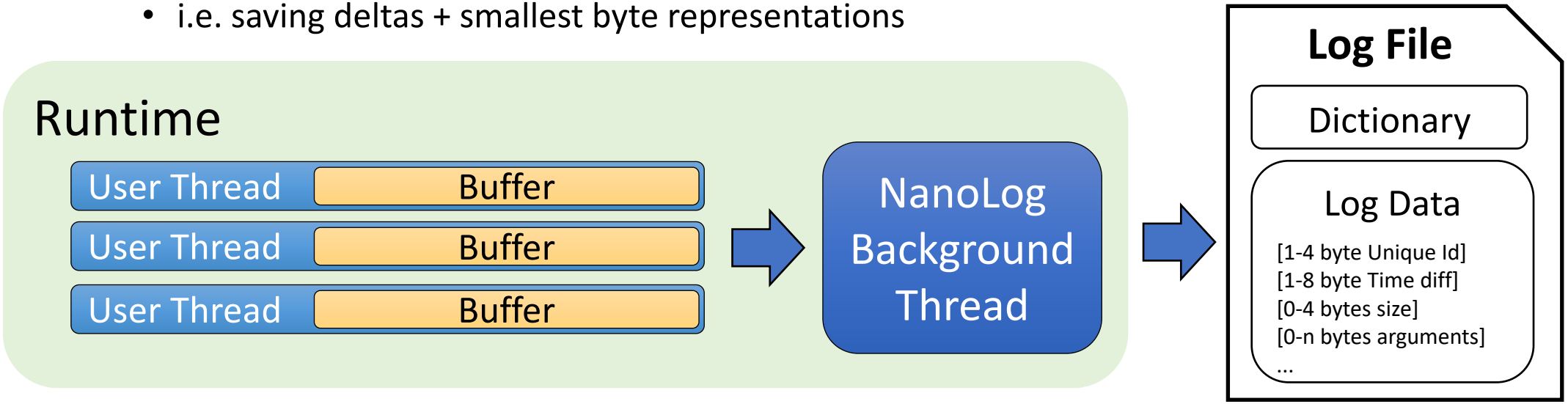
```
1  // Maps log instance to id
2  extern int uniqueId_main_cc_4 = 0;
3
4  writeDictionaryEntry(out) {
5    // Save static information
6    const char* fmtString = "Hello World %d";
7    const char* filename  = "main.cc";
8    const int linenum     = 4;
9    enum severity         = "NOTICE"
10
11   out << uniqueId_main_cc_4;
12   out << fmtString;
13   ...
14 }
```

compile

Application Executable

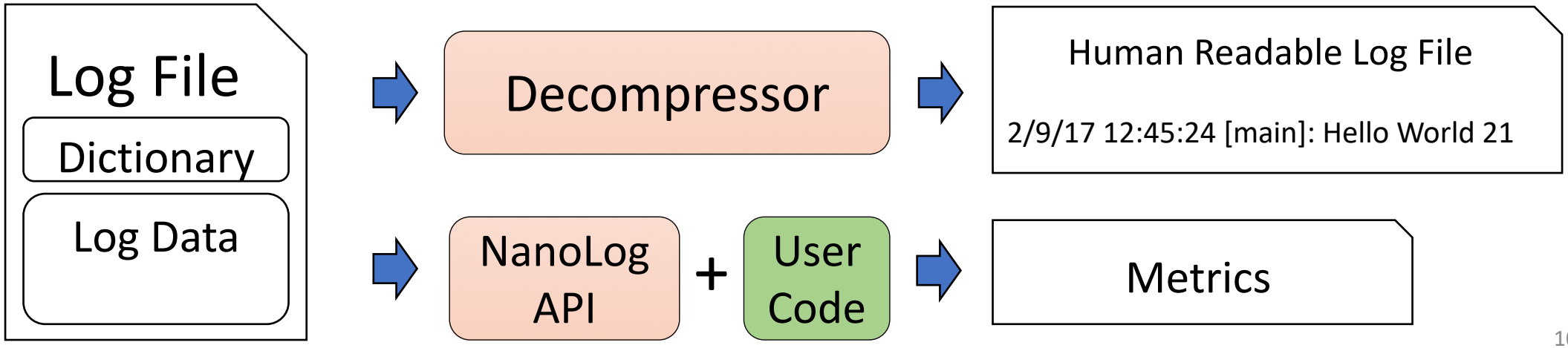
# Fast Runtime Architecture

- **NanoLog Background Thread**
  - Polls the thread buffers to output dynamic log data to disk/network
- **Low-Latency Thread Synchronization**
  - Use per-thread buffers to eliminate synchronization between logging threads.
  - Don’t notify the background thread; let it poll for data
- **Simple Compression**
  - Background thread uses rudimentary compaction save I/O and compute times
    - i.e. saving deltas + smallest byte representations



# Decompressor

- **Reconstitutes the log data by combining static + dynamic info**
- **Stand-Alone**
  - Reads the log file and produces a full human-readable log file.
  - Ultimately pays the formatting + output costs of the full log
- **Programmatic API (for fast aggregation)**
  - Process the log messages one by one without formatting.
  - More efficient than processing data in ASCII



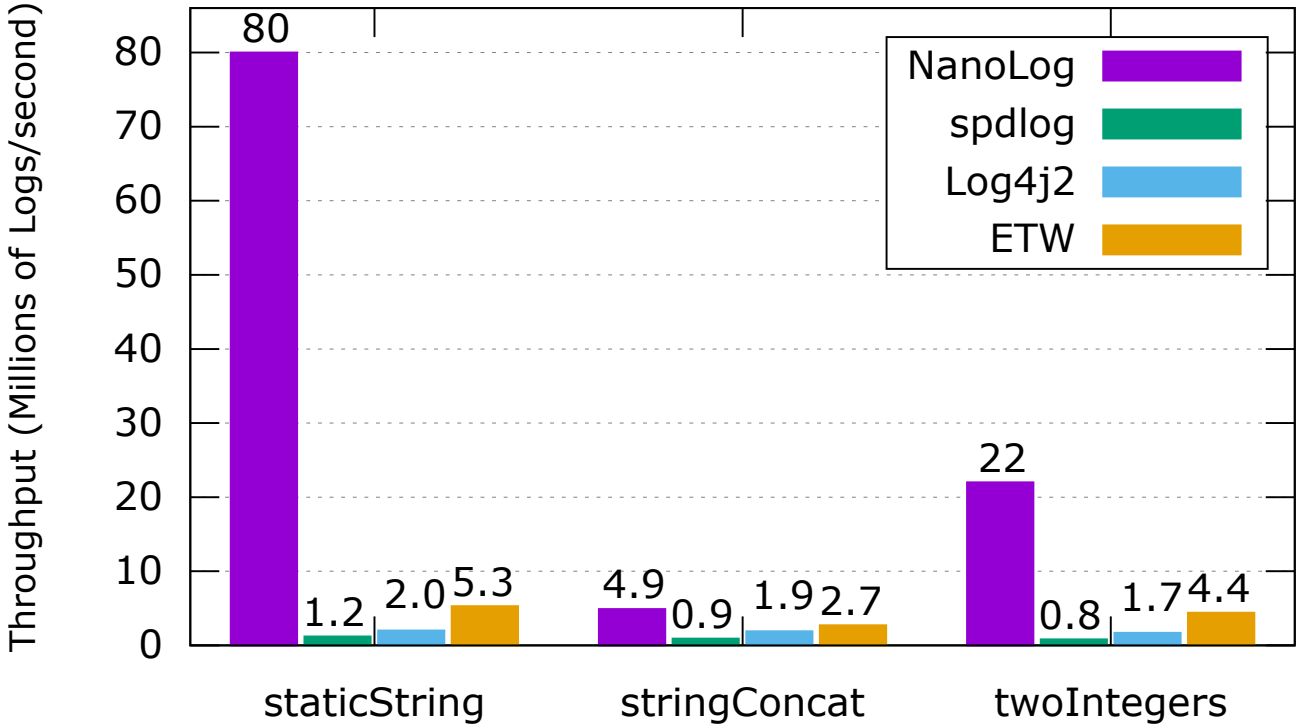
# Benchmarks

- **System Setup**
  - Processor: Quad-Core Intel Xeon X3470 @ 2.93GHz
  - Memory: 24GB DDR3 @ 1333Mhz
  - Disk: Samsung Evo 850 Pro 250GB over SATAII (~250MB/s)

- **Configuration**
  - Log Header: "2017-03-18 21:35:16.554837182 Benchmark.cc:21 NOTICE[4]:"
  - Log4j2 was configured to use the LMAX Disruptor library for maximum asynchrony.

- **Benchmarks**
  - Maximum Throughput, Unloaded Latency, Integration with RAMCloud, and Aggregation over the logs

# Microbenchmark - Throughput




## Evaluation

- Repeated logged 1 message with no inter-log delay in a thread
- Varied the number of logging threads to maximize throughput

## Conclusions

- NanoLog always faster
- Even better with fewer dynamic arguments.

| ID | Example Output | NL Msg Size |
|---|---|---|
| staticString | Starting backup replica garbage collector thread | 3-4 Bytes |
| stringConcat | Opened session with coordinator at basic+udp:host=192.168.1.140,port=12246 | 43 Bytes |
| twoIntegers | buffer has consumed 1032024 bytes of extra storage, current allocation: 1016544 bytes | 10 Bytes |

underlined indicates dynamic data

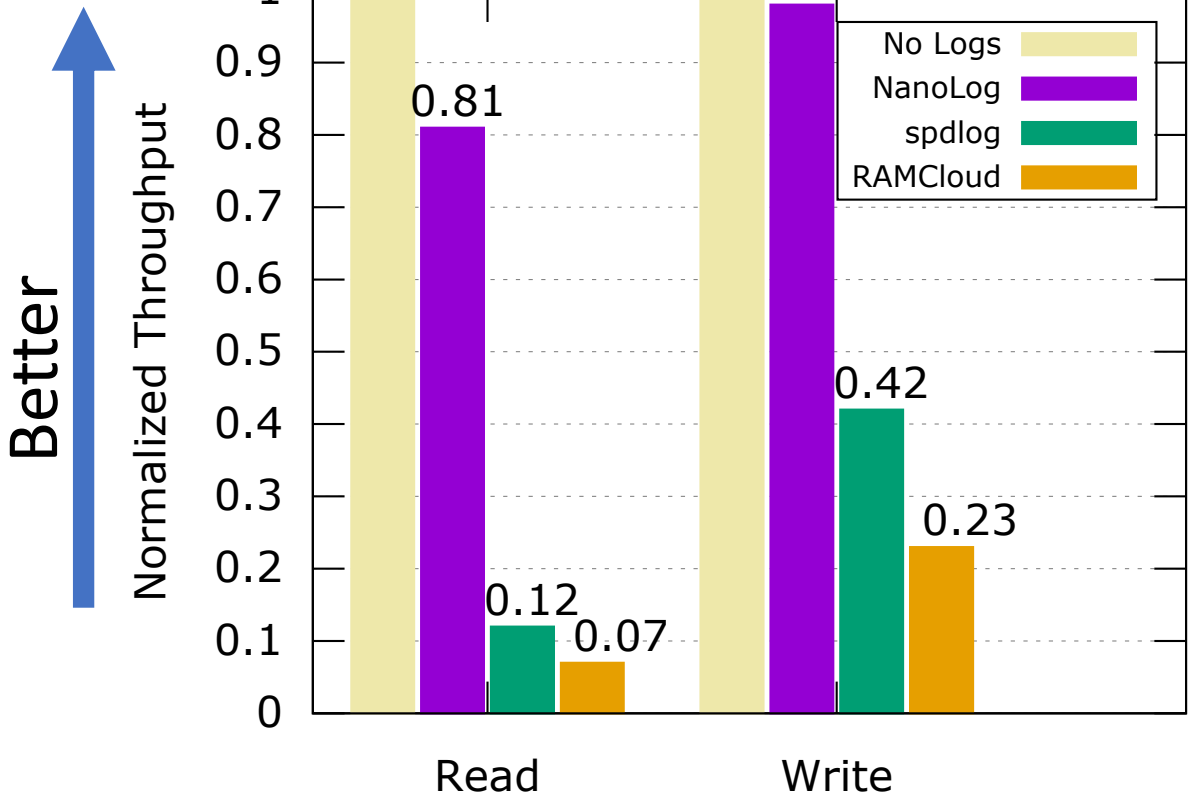
# Microbenchmark - Unloaded Latency

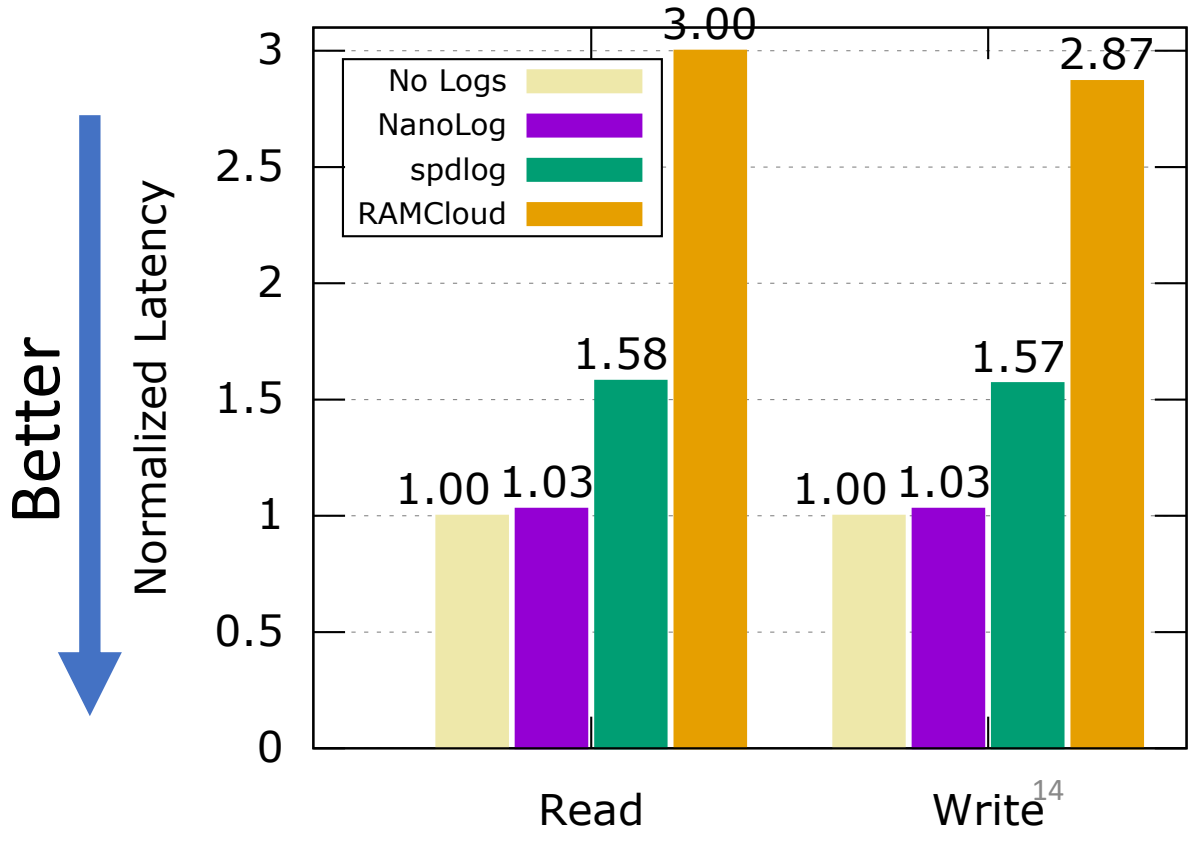
- **Goal: Measure minimum processing delay**
- **Setup:**
  - 100,000 iterations of logging 1 log message with a 650ns delay between each to eliminate blocking due to I/O
  - Percentile times reported below in nanoseconds

| System | NanoLog | spdlog | Log4j2 | ETW | | NanoLog | spdlog | Log4js | ETW |
|---|---|---|---|---|---|---|---|---|---|
| Log Msgs | Median Latency (ns) | | | | | 99.9 Percentile Latency (ns) | | | |
| staticString | **8** | 230 | 192 | 180 | | **33** | 473 | 1868 | 726 |
| stringConcat | **8** | 436 | 230 | 208 | | **33** | 1614 | 6171 | 2954 |
| twoIntegers | **7** | 674 | 160 | 200 | | **44** | 1335 | 1992 | 761 |

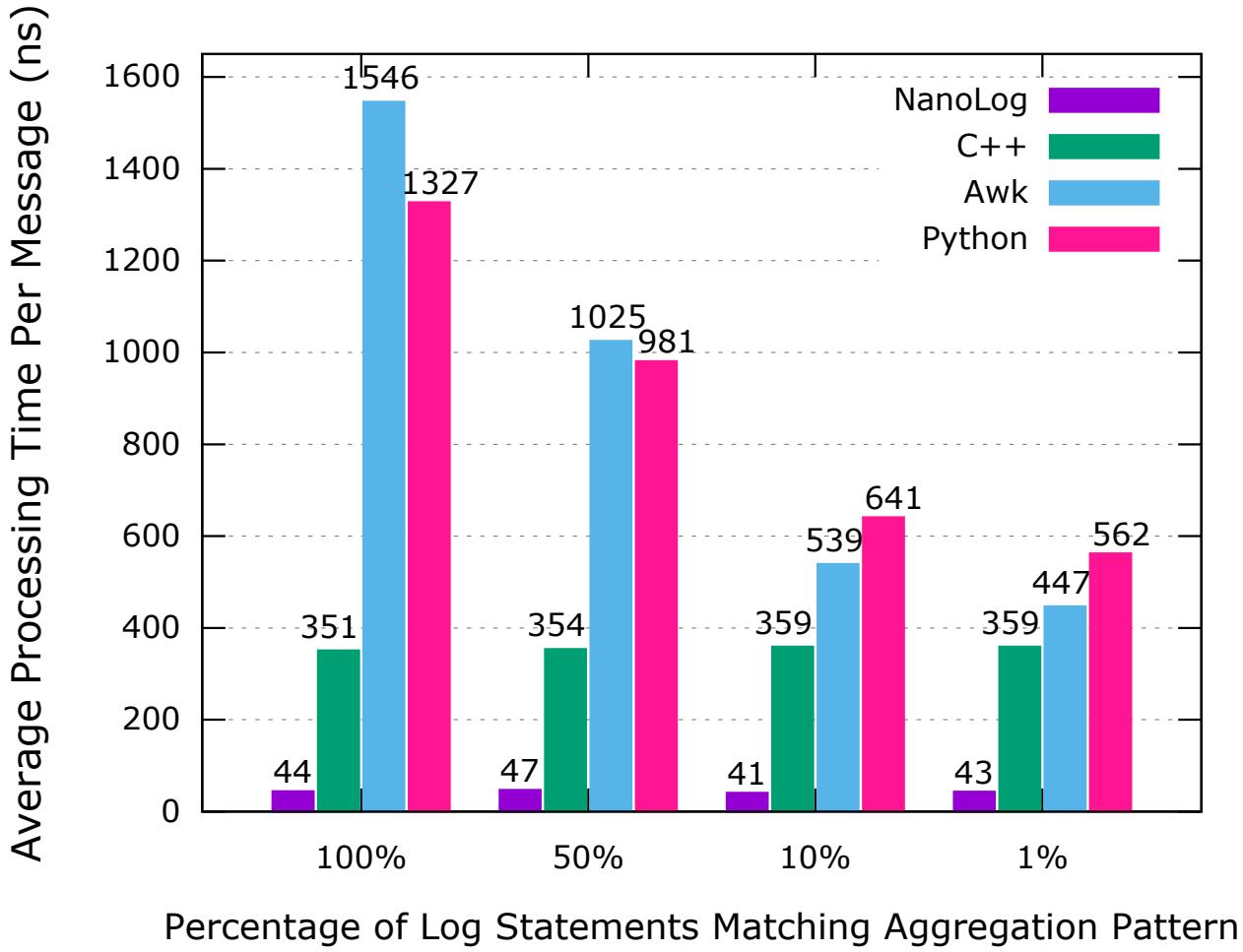
# Integration Benchmark

- **Integrated NanoLog + Spdlog into RAMCloud**
  - Replaced logger and enabled all instrumentation (11-33 log messages/operation)
  - Measured throughput + latency with built-in RAMCloud benchmarks



# NanoLog For Analytics




## Mock Analytics Workload

1. Find all instances of “Hello World # %d”
2. Interpret “%d” as an integer and compute the min/mean/max

## Configurations

- **NanoLog:** Used the compact, binary log
- **C++**: Used *atoi()* on the full log
- **Python, Awk**: Use regex on the full log

## NanoLog Analytics Benefits:

- Smaller log files (~747MB vs. 7.6GB)
- Binary format (no ASCII parsing req.)

# Extensibility

- **Generalizability**
  - Preprocessor can be modified to support any language that has compiled log messages.

- **Extraction of static information need not occur at compile-time**
  - Coming Soon: Alternate NanoLog that uses C++17 features instead of a preprocessor to extract static log information and log only dynamic data in the runtime hot-path

- **Limitation**
  - Does not support dynamically generated log messages
    - i.e. log messages in JavaScript eval()

# Conclusion

- **NanoLog's Techniques**
  - Statically analyze log statements to extract static log information
  - Only output the static information once at runtime
  - Rewrite log invocations to log only dynamic information at runtime
  - Use a post-processor to format log messages

- **Benefits:**
  - Extremely performant runtime and faster aggregations on smaller log files

# Thanks!

- **Contact Information**
  - **Github**: https://github.com/PlatformLab/NanoLog
  - **Primary Author**: Stephen Yang ( syang0@cs.stanford.edu )