NanoLog: A Nanosecond Scale Logging System

Stephen Yang, Seo Jin Park, and John Ousterhout, ***Stanford University***

https://www.usenix.org/conference/atc18/presentation/yang-stephen

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

NanoLog: A Nanosecond Scale Logging System

Stephen Yang
Stanford University

Seo Jin Park
Stanford University

John Ousterhout
Stanford University

Abstract

NanoLog is a nanosecond scale logging system that is 1-2 orders of magnitude faster than existing logging systems such as Log4j2, spdlog, Boost log or Event Tracing for Windows. The system achieves a throughput up to 80 million log messages per second for simple messages and has a typical log invocation overhead of 8 nanoseconds in microbenchmarks and 18 nanoseconds in applications, despite exposing a traditional printf-like API. NanoLog achieves this low latency and high throughput by shifting work out of the runtime hot path and into the compilation and post-execution phases of the application. More specifically, it slims down user log messages at compile-time by extracting static log components, outputs the log in a compacted, binary format at runtime, and utilizes an offline process to re-inflate the compacted logs. Additionally, log analytic applications can directly consume the compacted log and see a performance improvement of over 8x due to I/O savings. Overall, the lower cost of NanoLog allows developers to log more often, log in more detail, and use logging in low-latency production settings where traditional logging mechanisms are too expensive.

1 Introduction

Logging plays an important role in production software systems, and it is particularly important for large-scale distributed systems running in datacenters. Log messages record interesting events during the execution of a system, which serve several purposes. After a crash, logs are often the best available tool for debugging the root cause. In addition, logs can be analyzed to provide visibility into a system's behavior, including its load and performance, the effectiveness of its policies, and rates of recoverable errors. Logs also provide a valuable source of data about user behavior and preferences, which can be mined for business purposes. The more events that are recorded in a log, the more valuable it becomes.

Unfortunately, logging today is expensive. Just formatting a simple log message takes on the order of one microsecond in typical logging systems. Additionally, each log message typically occupies 50-100 bytes, so available I/O bandwidth also limits the rate at which log messages can be recorded. As a result, developers are often forced to make painful choices about which events to log; this impacts their ability to debug problems and understand system behavior.

Slow logging is such a problem today that software development organizations find themselves removing valuable log messages to maintain performance. According to our contacts at Google[7] and VMware[28], a considerable amount of time is spent in code reviews discussing whether to keep log messages or remove them for performance. Additionally, this process culls a lot of useful debugging information, resulting in many more person hours spent later debugging. Logging itself is expensive, but lacking proper logging is *very* expensive.

The problem is exacerbated by the current trend towards low-latency applications and micro-services. Systems such as Redis [34], FaRM [4], MICA[18] and RAMCloud [29] can process requests in as little as 1-2 microseconds; with today's logging systems, these systems cannot log events at the granularity of individual requests. This mismatch makes it difficult or impossible for companies to deploy low-latency services. One industry partner informed us that their company will not deploy low latency systems until there are logging systems fast enough to be used with them [7].

NanoLog is a new, open-source [47] logging system that is 1-2 orders of magnitude faster than existing systems such as Log4j2 [43], spdlog [38], glog [11], Boost Log [2], or Event Tracing for Windows [31]. NanoLog retains the convenient printf[33]-like API of existing logging systems, but it offers a throughput of around 80 million messages per second for simple log messages, with a caller latency of only 8 nanoseconds in microbenchmarks. For reference, Log4j2 only achieves a throughput of 1.5 million messages per second with latencies in the hundreds of nanoseconds for the same microbenchmark.

NanoLog achieves this performance by shifting work out of the runtime hot path and into the compilation and post-execution phases of the application:

- It rewrites logging statements at compile time to remove static information and defers expensive message formatting until the post-execution phase. This dramatically reduces the computation and I/O bandwidth requirements at runtime.
- It compiles specialized code for each log message to handle its dynamic arguments efficiently. This avoids runtime parsing of log messages and encoding argument types.
- It uses a lightweight compaction scheme and outputs the log out-of-order to save I/O and processing at runtime.
- It uses a postprocessor to combine compacted log data with extracted static information to generate

```
NANO_LOG(NOTICE, "Creating table '%s' with id %d", name, tableId);

2017/3/18 21:35:16.554575617 TableManager.cc:1031 NOTICE[4]: Creating table 'orders' with id 11
```

Figure 1: A typical logging statement (top) and the resulting output in the log file (bottom). "NOTICE" is a log severity level and "[4]" is a thread identifier.

human-readable logs. In addition, aggregation and analytics can be performed directly on the compacted log, which improves throughput by over 8x.

2 Background and Motivation

Logging systems allow developers to generate a human-readable trace of an application during its execution. Most logging systems provide facilities similar to those in Figure 1. The developer annotates system code with logging statements. Each logging statement uses a printf-like interface[33] to specify a static string indicating what just happened and also some runtime data associated with the event. The logging system then adds supplemental information such as the time when the event occurred, the source code file and line number of the logging statement, a severity level, and the identifier of the logging thread.

The simplest implementation of logging is to output each log message synchronously, inline with the execution of the application. This approach has relatively low performance, for two reasons. First, formatting a log message typically takes 0.5-1 µs (1000-2000 cycles). In a low latency server, this could represent a significant fraction of the total service time for a request. Second, the I/O is expensive. Log messages are typically 50-100 bytes long, so a flash drive with 250 Mbytes/sec bandwidth can only absorb a few million messages per second. In addition, the application will occasionally have to make kernel calls to flush the log buffer, which will introduce additional delays.

The most common solution to these problems is to move the expensive operations to a separate thread. For example, I/O can be performed in a background thread: the main application thread writes log messages to a buffer in memory, and the background thread makes the kernel calls to write the buffer to a file. This allows I/O to happen in parallel with application execution. Some systems, such as TimeTrace in PerfUtils [32], also offload the formatting to the background thread by packaging all the arguments into an executable lambda, which is evaluated by the background thread to format the message.

Unfortunately, moving operations to a background thread has limited benefit because the operations must still be carried out while the application is running. If log messages are generated at a rate faster than the background thread can process them (either because of I/O or CPU limitations), then either the application must eventually block, or it must discard log messages. Neither of these options is attractive. Blocking is particularly unappealing for low-latency systems because it can result in long tail latencies or even, in some situations, the appearance that a server has crashed.

In general, developers must ensure that an application doesn't generate log messages faster than they can be processed. One approach is to filter log messages according to their severity level; the threshold might be higher in a production environment than when testing. Another possible approach is to sample log messages at random, but this may cause key messages (such as those identifying a crash) to be lost. The final (but not uncommon) recourse is a social process whereby developers determine which log messages are most important and remove the less critical ones to improve performance. Unfortunately, all of these approaches compromise visibility to get around the limitations of the logging system.

The design of NanoLog grew out of two observations about logging. The first observation is that fully-formatted human-readable messages don't necessarily need to be produced inside the application. Instead, the application could log the raw components of each message and the human-readable messages could be generated later, if/when a human needs them. Many logs are never read by humans, in which case the message formatting step could be skipped. When logs are read, only a small fraction of the messages are typically examined, such as those around the time of a crash, so only a small fraction of logs needs to be formatted. And finally, many logs are processed by analytics engines. In this case, it is much faster to process the raw data than a human-readable version of the log.

The second observation is that log messages are fairly redundant and most of their content is static. For example, in the log message in Figure 1, the only dynamic parts of the message are the time, the thread identifier, and the values of the `name` and `tableId` variables. All of the other information is known at compile-time and is repeated in every invocation of that logging statement. It should be possible to catalog all the static information at compile-time and output it just once for the postprocessor. The postprocessor can reincorporate the static information when it formats the human-readable messages. This approach dramatically reduces the amount of information that the application must log, thereby allowing the application to log messages at a much higher rate.

The remainder of this paper describes how NanoLog capitalizes on these observations to improve logging performance by 1-2 orders of magnitude.

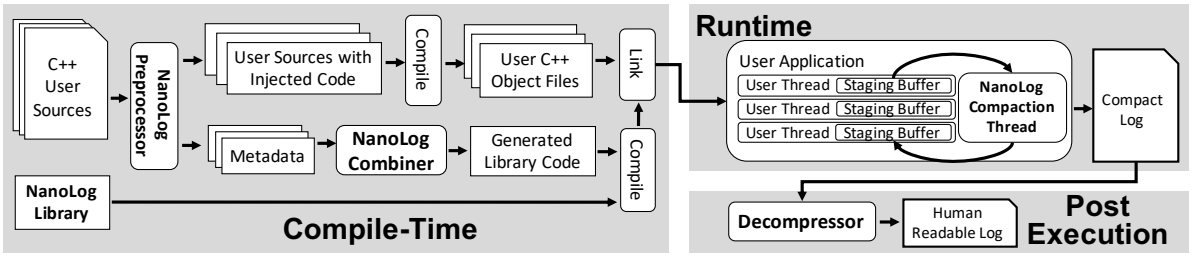

Figure 2: Overview of the NanoLog system. At compile time, the user sources are passed through the NanoLog preprocessor, which injects optimized logging code into the application and generates a metadata file for each source file. The modified user code is then compiled to produce C++ object files. The metadata files are aggregated by the NanoLog combiner to build a portion of the NanoLog Library. The NanoLog library is then compiled and linked with the user object files to create an application executable and a decompressor application. At runtime, the user application threads interact with the NanoLog staging buffers and background compaction thread to produce a compact log. At post execution, the compact log is passed into the decompressor to generate a final, human-readable log file.

3 Overview

NanoLog's low latency comes from performing work at compile-time to extract static components from log messages and deferring formating to an off-line process. As a result, the NanoLog system decomposes into three components as shown in Figure 2:

Preprocessor/Combiner: extracts and catalogs static components from log messages at compile-time, replaces original logging statements with optimized code, generates a unique compaction function for each log message, and generates a function to output the dictionary of static information.

Runtime Library: provides the infrastructure to buffer log messages from multiple logging threads and outputs the log in a compact, binary format using the generated compaction and dictionary functions.

Decompressor: recombines the compact, binary log file with the static information in the dictionary to either inflate the logs to a human-readable format or run analytics over the log contents.

Users of NanoLog interact with the system in the following fashion. First, they embed `NANO_LOG()` function calls in their C++ applications where they'd like log messages. The function has a signature similar to printf [17, 33] and supports all the features of printf with the exception of the "`%n`" specifier, which requires dynamic computation. Next, users integrate into their GNUmakefiles [40] a macro provided by NanoLog that serves as a drop-in replacement for a compiler invocation, such as g++. This macro will invoke the NanoLog preprocessor and combiner on the user's behalf and generate two executables: the user application linked against the NanoLog library, and a decompressor executable to inflate/run analytics over the compact log files. As the application runs, a compacted log is generated. Finally, the NanoLog decompressor can be invoked to read the compacted log and produce a human-readable log.

4 Detailed Design

We implemented the NanoLog system for C++ applications and this section describes the design in detail.

4.1 Preprocessor

The NanoLog preprocessor interposes in the compilation process of the user application (Figure 2). It processes the user source files and generates a *metadata file* and a modified source file for each user source file. The modified source files are then compiled into object files. Before the final link step for the application, the NanoLog combiner reads all the metadata files and generates an additional C++ source file that is compiled into the NanoLog Runtime Library. This library is then linked into the modified user application.

In order to improve the performance of logging, the NanoLog preprocessor analyzes the `NANO_LOG()` statements in the source code and replaces them with faster code. The replacement code provides three benefits. First, it reduces I/O bandwidth by logging only information that cannot be known until runtime. Second, NanoLog logs information in a compacted form. Third, the replacement code executes much more quickly than the original code. For example, it need not combine the dynamic data with the format string, or convert binary values to strings; data is logged in a binary format. The preprocessor also extracts type and order information from the format string (e.g., a `"%d %f"` format string indicates that the log function should encode an integer followed by a float). This allows the preprocessor to generate more efficient code that accepts and processes exactly the arguments provided to the log message. Type safety is ensured by leveraging the GNU `format` attribute compiler extension [10].

The NanoLog preprocessor generates two functions for each `NANO_LOG()` statement. The first function, `record()`, is invoked in lieu of the original `NANO_LOG()` statement. It records the dynamic information associated with the log message into an in-

```
inline void record(buffer, name, tableId) {
  // Unique identifier for this log statement;
  // the actual value is computed by the combiner.
  extern const int _logId_TableManager_cc_line1031;

  buffer.push<int>(_logId_TableManager_cc_line1031);
  buffer.pushTime();
  buffer.pushString(name);
  buffer.push<int>(tableId);
}

inline void compact(buffer, char *out) {
  pack<int>(buffer, out); // logId
  packTime(buffer, out); // time
  packString(buffer, out); // string name
  pack<int>(buffer, out); // tableId
}
```

Figure 3: Sample code generated by the NanoLog preprocessor and combiner for the log message in Figure 1. The `record()` function stores the dynamic log data to a buffer and `compact()` compacts the buffer's contents to an output character array.

memory buffer. The second function, `compact()`, is invoked by the NanoLog background compaction thread to compact the recorded data for more efficient I/O.

Figure 3 shows slightly simplified versions of the functions generated for the `NANO_LOG()` statement in Figure 1. The `record()` function performs the absolute minimum amount of work needed to save the log statement's dynamic data in a buffer. The invocation time is read using Intel's RDTSC instruction, which utilizes the CPU's fine grain Time Stamp Counter [30]. The only static information it records is a unique identifier for the `NANO_LOG()` statement, which is used by the NanoLog runtime background thread to invoke the appropriate `compact()` function and the decompressor to retrieve the statement's static information. The types of `name` and `tableId` were determined at compile-time by the preprocessor by analyzing the "`%s`" and "`%d`" specifiers in the format string, so `record()` can invoke type-specific methods to record them.

The purpose of the `compact()` function is to reduce the number of bytes occupied by the logged data, in order to save I/O bandwidth. The preprocessor has already determined the type of each item of data, so `compact()` simply invokes a type-specific compaction method for each value. Section 4.2.2 discusses the kinds of compaction that NanoLog performs and the trade-off between compute time and compaction efficiency.

In addition to the `record()` and `compact()` functions, the preprocessor creates a dictionary entry containing all of the static information for the log statement. This includes the file name and line number of the `NANO_LOG()` statement, the severity level and format string for the log message, the types of all the dynamic values that will be logged, and the name of a variable that will hold the unique identifier for this statement.

After generating this information, the preprocessor replaces the original `NANO_LOG()` invocation in the user source with an invocation to the `record()` function. It also stores the `compact()` function and the dictionary information in a metadata file specific to the original source file.

The NanoLog combiner executes after the preprocessor has processed all the user files (Figure 2); it reads all of the metadata files created by the preprocessor and generates additional code that will become part of the NanoLog runtime library. First, the combiner assigns unique identifier values for log statements. It generates code that defines and initializes one variable to hold the identifier for each log statement (the name of the variable was specified by the preprocessor in the metatadata file). Deferring identifier assignment to the combiner allows for a tight and contiguous packing of values while allowing multiple instances of the preprocessor to process client sources in parallel without synchronization. Second, the combiner places all of the `compact()` functions from the metadata files into a function array for the NanoLog runtime to use. Third, the combiner collects the dictionary information for all of the log statements and generates code that will run during application startup and write the dictionary into the log.

4.2 NanoLog Runtime

The NanoLog runtime is a statically linked library that runs as part of the user application and decouples the low-latency application threads executing the `record()` function from high latency operations like disk I/O. It achieves this by offering low-latency staging buffers to store the results of `record()` and a background compaction thread to compress the buffers' contents and issue disk I/O.

4.2.1 Low Latency Staging Buffers

Staging buffers store the result of `record()`, which is executed by the application logging threads, and make the data available to the background compaction thread. Staging buffers have a crucial impact on performance as they are the primary interface between the logging and background threads. Thus, they must be as low latency as possible and avoid thread interactions, which can result in lock contention and cache coherency overheads.

Locking is avoided in the staging buffers by allocating a separate staging buffer per logging thread and implementing the buffers as single producer, single consumer circular queues [24]. The allocation scheme allows multiple logging threads to store data into the staging buffers without synchronization between them and the implementation allows the logging thread and background thread to operate in parallel without locking the entire structure. This design also provides a throughput benefit as the source and drain operations on a buffer can be overlapped in time.

However, even with a lockless design, the threads' accesses to shared data can still cause cache coherency de-

lays in the CPU. More concretely, the circular queue implementation has to maintain a head position for where the log data starts and a tail position for where the data ends. The background thread modifies the head position to consume data and the logging thread modifies the tail position to add data. However, for either thread to query how much space it can use, it needs to access both variables, resulting in expensive cache coherency traffic.

NanoLog reduces cache coherency traffic in the staging buffers by performing multiple inserts or removes for each cache miss. For example, after the logging thread reads the head pointer, which probably results in a cache coherency miss since its modified by the background thread, it saves a copy in a local variable and uses the copy until all available space has been consumed. Only then does it read the head pointer again. The compaction thread caches the tail pointer in a similar fashion, so it can process all available log messages before incurring another cache miss on the tail pointer. This mechanism is safe because there is only a single reader and a single writer for each staging buffer.

Finally, the logging and background threads store their private variables on separate cachelines to avoid false sharing [1].

4.2.2 High Throughput Background Thread

To prevent the buffers from running out of space and blocking, the background thread must consume the log messages placed in the staging staging buffer as fast as possible. It achieves this by deferring expensive log processing to the post-execution decompressor application and compacting the log messages to save I/O.

The NanoLog background thread defers log formatting and chronology sorting to the post-execution application to reduce log processing latency. For comparison, consider a traditional logging system; it outputs the log messages in a human-readable format and in chronological order. The runtime formatting incurs computation costs and bloats the log message. And maintaining chronology means the background thread must either serialize all logging or sort the log messages from concurrent logging threads at runtime. Both of these operations are expensive, so the background thread performs neither of these tasks. The NanoLog background thread simply iterates through the staging buffers in round-robin fashion and for each buffer, processes the buffer's entire contents and outputs the results to disk. The processing is also non-quiescent, meaning a logging thread can record new log messages while the background thread is processing its staging buffer's contents.

Additionally, the background thread needs to perform some sort of compression on the log messages to reduce I/O latency. However, compression only makes sense if it reduces the overall end-to-end latency. In our measurements, we found that while existing compression schemes like the LZ77 algorithm [49] used by gzip [9] were very effective at reducing file size, their computation times were too high; it was often faster to output the raw log messages than to perform any sort of compression. Thus, we developed our own lightweight compression mechanism for use in the `compact()` function.

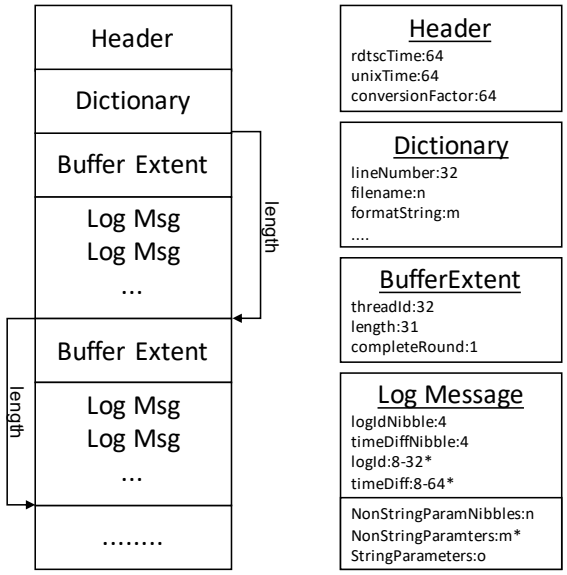

Figure 4: Layout of a compacted log file produced by the NanoLog runtime at a high level (left) and at the component level (right). As indicated by the diagram on the left, the NanoLog output file always starts with a Header and a Dictionary. The rest of the file consists of Buffer Extents. Each Buffer Extent contains log messages. On the right, the smaller text indicates field names and the digits after the colon indicate how many bits are required to represent the field. An asterisk (*) represents integer values that have been compacted and thus have a variable byte length. The lower box of "Log Message" indicates fields that are variable length (and sometimes omitted) depending on the log message's arguments.

NanoLog attempts to compact the integer types by finding the fewest number of bytes needed to represent that integer. The assumptions here are that integers are the most commonly logged type, and most integers are fairly small and do not use all the bits specified by its type. For example, a 4 byte integer of value 200 can be represented with 1 byte, so we encode it as such. To keep track of the number of bytes used for the integer, we add a *nibble* (4-bits) of metadata. Three bits of the nibble inform the algorithm how many bytes are used to encode the integer and the last bit is used to indicate a negation. The negation is useful for when small negative numbers are encoded. For example a -1 can be represented in 1 byte without ambiguity if the negation bit was set. A limitation of this scheme is that an extra half byte (nibble) is wasted in cases where the integer cannot be compacted.

Applying these techniques, the background thread produces a log file that resembles Figure 4. The first

component is a header which maps the machine's Intel Time Stamp Counter [30] (TSC) value to a wall time and the conversion factor between the two. This allows the log messages to contain raw invocation TSC values and avoids wall time conversion at runtime. The header also includes the dictionary containing static information for the log messages. Following this structure are *buffer extents* which represent contiguous staging buffers that have been output at runtime and contained within them are log messages. Each buffer extent records the runtime thread id and the size of the extent (with the log messages). This allows log messages to omit the thread id and inherit it from the extent, saving bytes.

The log messages themselves are variable sized due to compaction and the number of parameters needed for the message. However, all log messages will contain at least a compacted log identifier and a compacted log invocation time relative to the last log message. This means that a simple log message with no parameters can be as small as 3 bytes (2 nibbles and 1 byte each for the log identifier and time difference). If the log message contains additional parameters, they will be encoded after the time difference in the order of all nibbles, followed by all non-string parameters (compacted and uncompacted), followed by all string parameters. The ordering of the nibbles and non-string parameters is determined by the preprocessor's generated code, but the nibbles are placed together to consolidate them. The strings are also null terminated so that we do not need to explicitly store a length for each.

4.3 Decompressor/Aggregator

The final component of the NanoLog system is the decompressor/aggregator, which takes as input the compacted log file generated by the runtime and either outputs a human-readable log file or runs aggregations over the compacted log messages. The decompressor reads the dictionary information from the log header, then it processes each of the log messages in turn. For each message, it uses the log id embedded in the file to find the corresponding dictionary entry. It then decompresses the log data as indicated in the dictionary entry and combines that data with static information from the dictionary to generate a human-readable log message. If the decompressor is being used for aggregation, it skips the message formatting step and passes the decompressed log data, along with the dictionary information, to an aggregation function.

One challenge the NanoLog decompressor has to deal with is outputting the log messages in chronological order. Recall from earlier, the NanoLog runtime outputs the log messages in staging buffer chunks called buffer extents. Each logging thread uses its own staging buffer, so log messages are ordered chronologically within an extent, but the extents for different threads can overlap in time. The decompressor must collate log entries from different extents in order to output a properly ordered log. The round-robin approach used by the compaction thread means that extents in the log are roughly ordered by time. Thus, the decompressor can process the log file sequentially. To perform the merge correctly, it must buffer two sequential extents for each logging thread at a time.

Aside from the reordering, one of the most interesting aspects of this component is the promise it holds for faster analytics. Most analytics engines have to gather the human-readable logs, parse the log messages into a binary format, and then compute on the data. Almost all the time is spent reading and parsing the log. The NanoLog aggregator speeds this up in two ways. First, the intermediate log file is extremely compact compared to its human-readable format (typically over an order of magnitude) which saves on bandwidth to read the logs. Second, the intermediate log file already stores the dynamic portions of the log in a binary format. This means that the analytics engine does not need to perform expensive string parsing. These two features mean that the aggregator component will run faster than a traditional analytics engine operating on human-readable logs.

4.4 Alternate Implementation: C++17 NanoLog

While the techniques shown in the previous section are generalizable to any programming language that exposes its source, some languages such as C++17 offer strong compile-time computation features that can be leveraged to build NanoLog without an additional preprocessor. In this section, we briefly present such an implementation for C++17. The full source for this implementation is available in our GitHub repository[47], so we will only highlight the key features here.

The primary tasks that the NanoLog preprocessor performs are (a) generating optimized functions to *record* and *compact* arguments based on types, (b) assigning unique log identifiers to each `NANO_LOG()` invocation site and (c) generating a dictionary of static log information for the postprocessor.

For the first task, we can leverage inlined variadic function templates in C++ to build optimized functions to *record* and *compact* arguments based on their types. C++11 introduced functionality to build generic functions that would specialize on the types of the arguments passed in. One variation, called "variadic templates", allows one to build functions that can accept an unbounded number of arguments and process them recursively based on type. Using these features, we can express meta `record()` and `compact()` functions which accept any number of arguments and the C++ compiler will automatically select the correct function to invoke for each argument based on type.

One problem with this mechanism is that an argument of type "`char*`" can correspond to either a "`%s`" speci-

CPU	Xeon X3470 (4x2.93 GHz cores)
RAM	24 GB DDR3 at 800 MHz
Flash	2x Samsung 850 PRO (250GB) SSDs
OS	Debian 8.3 with Linux kernel 3.16.7
OS for ETW	Windows 10 Pro 1709, Build 16299.192

Table 1: The server configuration used for benchmarking.

fier (string) or a "`%p`" specifier (pointer), which are handled differently. To address this issue, we leverage constant expression functions in C++17 to analyze the static format string at compile-time and build a constant expression structure that can be checked in `record()` to selectively save a pointer or string. This mechanism makes it unnecessary for NanoLog to perform the expensive format string parsing at runtime and reduces the runtime cost to a single if-check.

The second task is assignment of unique identifiers. C++17 NanoLog must discover all the `NANO_LOG()` invocation sites dynamically and associate a unique identifier with each. To do this, we leverage scoped static variables in C++; `NANO_LOG()` is defined as a macro that expands to a new scope with a static identifier variable initialized to indicate that no identifier has been assigned yet. This variable is passed by reference to the `record()` function, which checks its value and assigns a unique identifier during the first call. Future calls for this invocation site pay only for an if-check to confirm that the identifier has been assigned. The scoping of the identifier keeps it private to the invocation site and the static keyword ensures that the value persists across all invocations for the lifetime of the application.

The third task is to generate the dictionary required by the postprocessor and write it to the log. The dictionary cannot be included in the log header, since the NanoLog runtime has no knowledge of a log statement until it executes for the first time. Thus, C++17 NanoLog outputs dictionary information to the log in an incremental fashion. Whenever the runtime assigns a new unique identifier, it also collects the dictionary information for that log statement. This information is passed to the compaction thread and output in the header of the next Buffer Extent that contains the first instance of this log message. This scheme ensures that the decompressor encounters the dictionary information for a log statement before it encounters any data records for that log statement.

The benefit of this C++17 implementation is that it is easier to deploy (users no longer have to integrate the NanoLog preprocessor into their build chain), but the downsides are that it is language specific and performs slightly more work at runtime.

5 Evaluation

We implemented the NanoLog system for C++ applications. The NanoLog preprocessor and combiner comprise of 1319 lines of Python code and the NanoLog runtime library consists of 3657 lines of C++ code.

System Name	Static Chars	Integers	Floats	Strings	Others	Logs
Memcached	56.04	0.49	0.00	0.23	0.04	378
httpd	49.38	0.29	0.01	0.75	0.03	3711
linux	35.52	0.98	0.00	0.57	0.10	135119
Spark	43.32	n/a	n/a	n/a	n/a	2717
RAMCloud	46.65	1.08	0.07	0.47	0.02	1167

Table 2: Shows the average number of static characters (Static Chars) and dynamic variables in formatted log statements for five open source systems. These numbers were obtained by applying a set of heuristics to identify log statements in the source files and analyzing the embedded format strings; the numbers do not necessarily reflect runtime usage and may not include every log invocation. The "Logs" column counts the total number of log messages found. The dynamic counts are omitted for Spark since their logging system does not use format specifiers, and thus argument types could not be easily extracted. The static characters column omits format specifiers and variable references (i.e. $variables in Spark), and represents the number of characters that would be trivially saved by using NanoLog.

We evaluated the NanoLog system to answer the following questions:

- How do NanoLog's log throughput and latency compare to other modern logging systems?
- What is the throughput of the decompressor?
- How efficient is it to query the compacted log file?
- How does NanoLog perform in a real system?
- What are NanoLog's throughput bottlenecks?
- How does NanoLog's compaction scheme compare to other compression algorithms?

All experiments were conducted on quad-core machines with SATA SSDs that had a measured throughput of about 250MB/s for large writes (Table 1).

5.1 System Comparison

To compare the performance of NanoLog with other systems, we ran microbenchmarks with six log messages (shown in Table 3) selected from an open-source datacenter storage system [29].

5.1.1 Test Setup

We chose to benchmark NanoLog against Log4j2 [43], spdlog [38], glog [11], Boost log [2], and Event Tracing for Windows (ETW) [31]. We chose Log4j2 for its popularity in industry; we configured it for low latency and high throughput by using asynchronous loggers and appenders and including the LMAX Disruptor library [20]. We chose spdlog because it was the first result in an Internet search for "Fast C++ Logger"; we configured spdlog with a buffer size of 8192 entries (or 832KB). We chose glog because it is used by Google and configured it to buffer up to 30 seconds of logs. We chose Boost logging because of the popularity of Boost libraries in the C++ community; we configured Boost to use asynchronous sinks. We chose ETW because of its similarity to NanoLog; when used with Windows Software

ID	Example Output
staticString	Starting backup replica garbage collector thread
stringConcat	Opened session with coordinator at basic+udp:host=192.168.1.140,port=12246
singleInteger	Backup storage speeds (min): 181 MB/s read
twoIntegers	buffer has consumed 1032024 bytes of extra storage, current allocation: 1016544 bytes
singleDouble	Using tombstone ratio balancer with ratio = 0.4
complexFormat	Initialized InfUdDriver buffers: 50000 receive buffers (97 MB), 50 transmit buffers (0 MB), took 26.2 ms

Table 3: Log messages used to generate Figure 5 and Table 4. The underlines indicate dynamic data generated at runtime. *staticString* is a completely static log message, *stringConcat* contains a large dynamic string, and other messages are a combination of integer and floating point types. Additionally, the logging systems were configured to output each message with the context "YY-MM-DD HH:MM:SS.ns Benchmark.cc:20 DEBUG[0]:" prepended to it.

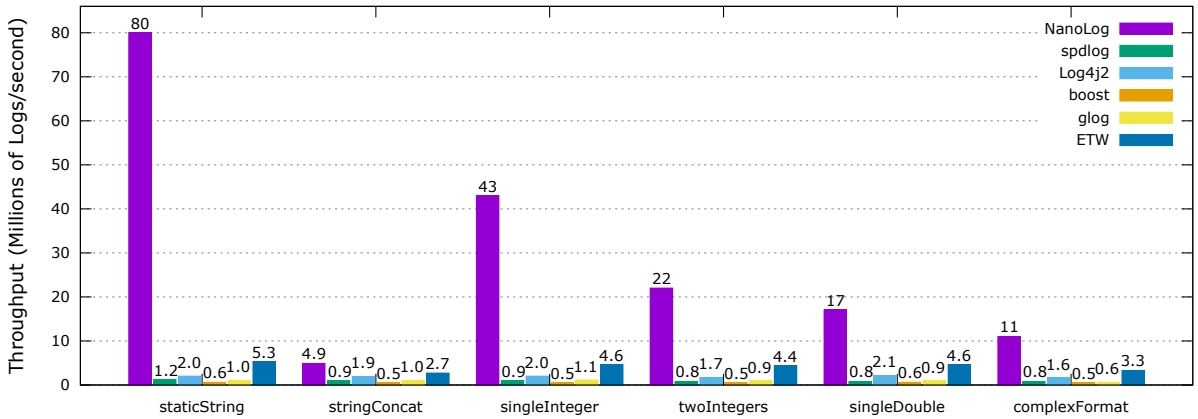

Figure 5: Shows the maximum throughput attained by various logging systems when logging a single message repeatedly. Log4j2, Boost, spdlog, and Google glog logged the message 1 million times; ETW and NanoLog logged the message 8 and 100 million times repectively to generate a log file of comparable size. The number of logging threads varied between 1-16 and the maximum throughput achieved is reported. All systems except Log4j2 include the time to flush the messages to disk in its throughput calculations (Log4j2 did not provide an API to flush the log without shutting down the logging service). The message labels on the x-axis are explained in Table 3.

Trace PreProcessor [23], the log statements are rewritten to record only variable binary data at runtime. We configured ETW with the default buffer size of 64 KB; increasing it to 1 MB did not improve its steady-state performance.

We configured each system to output similar metadata information with each log message; they prepend a date/-time, code location, log level, and thread id to each log message as shown in Figure 1. However, there are implementation differences in each system. In the time field, NanoLog and spdlog computed the fractional seconds with 9 digits of precision (nanoseconds) vs 6 for Boost-/glog and 3 for Log4j2 and ETW. In addition, Log4j2's code location information (ex. "Benchmark.cc:20") was manually encoded due to inefficiencies in its code location mechanism [45]. The other systems use the GNU C++ preprocessor macros "`__LINE__`" and "`__FILE__`" to encode the code location information.

To ensure the log messages we chose were representative of real world usage, we statically analyzed log statements from five open source systems[22, 42, 19, 44, 29]. Table 2 shows that log messages have around 45 characters of static content on average and that integers are the most common dynamic type. Strings are the second most common type, but upon closer inspection, most strings used could benefit from NanoLog's static extraction methods. They contain pretty print error messages, enumerations, object variables, and other static/formatted types. This static information could in theory be also extracted by NanoLog and replaced with an identifier. However, we leave this additional extraction of static content this to future work.

5.1.2 Throughput

Figure 5 shows the maximum throughput achieved by NanoLog, spdlog [38], Log4j2 [43], Boost [2], Google glog [11], and ETW [31]. NanoLog is faster than the other systems by 1.8x-133x. The largest performance gap between NanoLog and the other systems occurs with *staticString* and the smallest occurs with *stringConcat*.

NanoLog performs best when there is little dynamic information in the log message. This is reflected by *staticString*, a static message, in the throughput benchmark. Here, NanoLog only needs to output about 3-4 bytes per log message due to its compaction and static extraction techniques. Other systems require over an order of magnitude more bytes to represent the messages (41-90 bytes). Even ETW, which uses a preprocessor to strip messages, requires at least 41 bytes in the static string

ID	NanoLog				spdlog				Log4j2				glog				Boost				ETW			
Percentiles	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*
staticString	8	9	29	33	230	236	323	473	192	311	470	1868	1201	1229	3451	5231	1619	2338	3138	4413	180	187	242	726
stringConcat	8	9	29	33	436	494	1579	1641	230	1711	3110	6171	1235	1272	3469	5728	1833	2621	3387	5547	208	218	282	2954
singleInteger	8	9	29	35	353	358	408	824	223	321	458	1869	1250	1268	3543	5458	1963	2775	3396	7040	189	195	237	720
twoIntegers	7	8	29	44	674	698	807	1335	160	297	550	1992	1369	1420	3554	5737	2255	3167	3932	7775	200	207	237	761
singleDouble	8	9	29	34	607	637	685	1548	157	252	358	1494	2077	2135	4329	6995	2830	3479	3885	7176	187	193	248	720
complexFormat	8	8	28	33	1234	1261	1425	3360	146	233	346	1500	2570	2722	5167	8589	4175	4621	5189	9637	242	252	304	1070

Table 4: Unloaded tail latencies of NanoLog and other popular logging frameworks, measured by logging 100,000 log messages from Table 3 with a 600 nanosecond delay between log invocations to ensure that I/O is not a bottleneck. Each datum represents the 50th/90th/99th/99.9th percentile latencies measured in nanoseconds.

case. NanoLog excels with static messages, reaching a throughput of 80 million log messages per second.

NanoLog performs the worst when there's a large amount of dynamic information. This is reflected in *stringConcat*, which logs a large 39 byte dynamic string. NanoLog performs no compaction on string arguments and thus must log the entire string. This results in an output of 41-42 bytes per log message and drops throughput to about 4.9 million log messages per second.

Overall, NanoLog is faster than all other logging systems tested. This is primarily due to NanoLog consistently outputting fewer bytes per message and secondarily because NanoLog defers the formatting and sorting of log messages.

5.1.3 Latency

NanoLog lowers the logging thread's invocation latency by deferring the formatting of log messages. This effect can be seen in Table 4. NanoLog's invocation latency is 18-500x lower than other systems. In fact, NanoLog's 50/90/99th percentile latencies are all within tens of nanoseconds while the median latencies for the other systems *start* at hundreds of nanoseconds.

All of the other systems except ETW require the logging thread to either fully or partially materialize the human-readable log message before transferring control to the background thread, resulting in higher invocation latencies. NanoLog on the other hand, performs no formatting and simply pushes all arguments to the staging buffer. This means less computation and fewer bytes copied, resulting in a lower invocation latency.

Although ETW employs techniques similar to NanoLog, its latencies are much higher than those of NanoLog. We are unsure why ETW is slower than NanoLog, but one hint is that the even with the preprocessor, ETW log messages are larger than NanoLog (41 vs. 4 bytes for staticString). ETW emits extra log information such as process ids and does not use the efficient compaction mechanism of NanoLog to reduce its output.

Overall, NanoLog's unloaded invocation latency is extremely low.

5.2 Decompression

Since the NanoLog runtime outputs the log in a binary format, it is also important to understand the performance implications of transforming it back into a human readable log format.

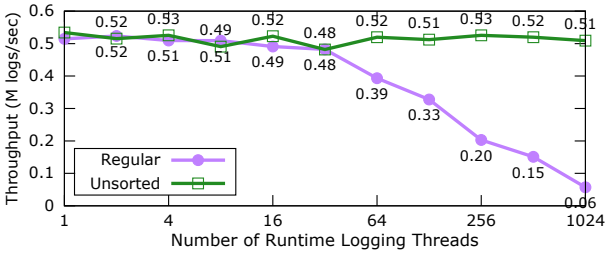

Figure 6: Impact on NanoLog's decompressor performance as the number of runtime logging threads increases. We decompressed a log file containing 2^{24} log messages (about 16M) in the format of "2017-04-06 02:03:25.000472519 Benchmark.cc:65 NOTICE[0]: Simple log message with 0 parameters". The compacted log file was 49MB and the resulting decompressed log output was 1.5GB. In the "Unsorted" measurements, the decompressor did not collate the log entries from different threads into a single chronological order.

The decompressor currently uses a simple single-threaded implementation, which can decompress at a peak of about 0.5M log messages/sec (Figure 6). Traditional systems such as Log4j2 can achieve a higher throughput of over 2M log messages/second at runtime since they utilize all their logging threads for formatting. NanoLog's decompressor can be modified to use multiple threads to achieve higher throughput.

The throughput of the decompressor can drop if there were many runtime logging threads in the application. The reason is that the log is divided into different extents for each logging thread, and the decompressor must collate the log messages from multiple extents into a single chronological order. Figure 6 shows that decompressor can handle up to about 32 logging threads with no impact on its throughput, but throughput drops with more than 32 logging threads. This is because the decompressor uses a simple collation algorithm that compares the times for the next message from each active buffer extent (one per logging thread) in order to pick the next message to print; thus the cost per message increases linearly with the number of logging threads. Performance could be improved by using a heap for collation.

Collation is only needed if order matters during decompression. For some applications, such as analytics,

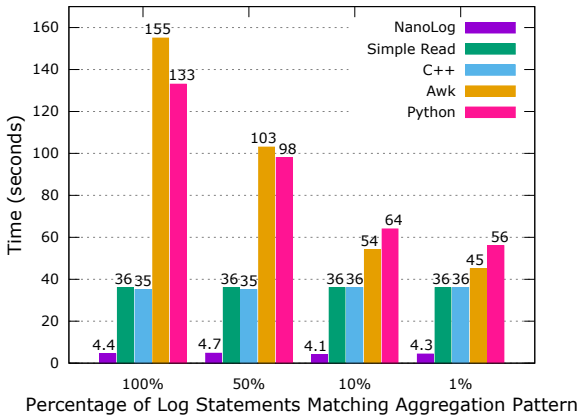

Figure 7: Execution time for a min/mean/max aggregation using various systems over 100 million log messages with a percentage of the log messages matching the target aggregation pattern "`Hello World # %d`" and the rest "`UnrelatedLog #%d`". The NanoLog system operated on a compacted file (∼747MB) and the remaining systems operated on the full, uncompressed log (∼7.6GB). The C++ application searched for the "`Hello World #`" prefix and utilized atoi() on the next word to parse the integer. The Awk and Python applications used a simple regular expression matching the prefix: "`.*Hello World # (\d+)`". "Simple Read" reads the entire log file and discards the contents. The file system cache was flushed before each run.

the order in which log messages are processed is unimportant. In these cases, collation can be skipped; Figure 6 shows that decompression throughput in this case is unaffected by the number of logging threads.

5.3 Aggregation Performance

NanoLog's compact, binary log output promises more efficient log aggregation/analytics than its full, uncompressed counterpart. To demonstrate this, we implemented a simple min/mean/max aggregation in four systems, NanoLog, C++, Awk, and Python. Conceptually, they all perform the same task; they search for the target log message "`Hello World #%d`" and perform a min/mean/max aggregation over the "`%d`" integer argument. The difference is that the latter three systems operate on the full, uncompressed version of the log while the NanoLog aggregator operates directly on the output from the NanoLog runtime.

Figure 7 shows the execution time for this aggregation over 100M log messages. NanoLog is nearly an order of magnitude faster than the other systems, taking on average 4.4 seconds to aggregate the compact log file vs. 35+ seconds for the other systems. The primary reason for NanoLog's low execution time is disk bandwidth. The compact log file only amounted to about 747MB vs. 7.6GB for the uncompressed log file. In other words, the aggregation was disk bandwidth limited and NanoLog used the least amount of disk IO. We verified this assumption with a simple C++ application that performs no aggregation and simply reads the file ("Simple Read" in the figure); its execution time lines up with the "C++" aggregator at around 36 seconds.

We also varied how often the target log message "`Hello World #%d`" occurred in the log file to see if it affects aggregation time. The compiled systems (NanoLog and C++) have a near constant cost for aggregating the log file while the interpreted systems (Awk and Python) have processing costs correlated to how often the target message occurred. More specifically, the more frequent the target message, the longer the execution time for Awk and Python. We suspect the reason is because the regular expression systems used by Awk and Python can quickly disqualify non-matching strings, but perform more expensive parsing when a match occurs. However, we did not investigate further.

Overall, the compactness of the NanoLog binary log file allows for fast aggregation.

		No Logs	NanoLog	spdlog	RAMCloud
Throughput (kop/s)	Read	994 (100%)	809 (81%)	122 (12%)	67 (7%)
	Write	140 (100%)	137 (98%)	59 (42%)	32 (23%)
Read Latency (μs)	50%	5.19 (1.00x)	5.33 (1.03x)	8.21 (1.58x)	15.55 (3.00x)
	90%	5.56 (1.00x)	5.53 (0.99x)	8.71 (1.57x)	16.66 (3.00x)
	99%	6.15 (1.00x)	6.15 (1.00x)	9.60 (1.56x)	17.82 (2.90x)
Write Latency (μs)	50%	15.85 (1.00x)	16.33 (1.03x)	24.88 (1.57x)	45.53 (2.87x)
	90%	16.50 (1.00x)	17.08 (1.04x)	26.42 (1.60x)	47.50 (2.88x)
	99%	22.87 (1.00x)	23.74 (1.04x)	33.05 (1.45x)	59.17 (2.59x)

Table 5: Shows the impact on RAMCloud [29] performance when more intensive instrumentation is enabled. The instrumentation adds about 11-33 log statements per read/write request with 1-3 integer log arguments each. "No Logs" represents the baseline with no logging enabled. "RAMCloud" uses the internal logger while "NanoLog" and "spdlog" supplant the internal logger with their own. The percentages next to Read-/Write Latency represent percentiles and all results were measured with RAMCloud's internal benchmarks with 16 clients used in the throughput measurements. Throughput benchmarks were run for 10 seconds and latency benchmarks measured 2M operations. Each configurations was run 15 times and the best case is presented.

5.4 Integration Benchmark

We integrated NanoLog and spdlog into a well instrumented open-source key value store, RAMCloud[29], and evaluated the logging systems' impact on performance using existing RAMCloud benchmarks. In keeping with the goal of increasing visibility, we enabled verbose logging and changed existing performance sampling statements in RAMCloud (normally compiled out) to always-on log statements. This added an additional 11-33 log statements per read/write request in the system. With this heavily instrumented system, we could answer the following questions: (1) how much of an improvement does NanoLog provide over other state-of-the-art systems in this scenario, (2) how does NanoLog perform in a real system compared to microbenchmarks

and (3) how much does NanoLog slowdown compilation and increase binary size?

Table 5 shows that, with NanoLog, the additional instrumentation introduces only a small performance penalty. Median read-write latencies increased only by about 3-4% relative to an uninstrumented system and write throughput decreased by 2%. Read throughput sees a larger degradation (about 19%); we believe this is because read throughput is bottlenecked by RAMCloud's dispatch thread [29], which performs most of the logging. In contrast, the other logging systems incur such a high performance penalty that this level of instrumentation would probably be impractical in production: latencies increase by 1.6-3x, write throughput drops by more than half, and read throughput is reduced to roughly a tenth of the uninstrumented system (8-14x). These results show that NanoLog supports a higher level of instrumentation than other logging systems.

Using this benchmark, we can also estimate NanoLog's invocation latency when integrated in a low-latency system. For RAMCloud's read operation, the critical path emits 8 log messages out of the 11 enabled. On average, each log message increased latency by (5.33-5.19)/8 = 17.5ns. For RAMCloud's write operation, the critical path emits 27 log messages, suggesting an average latency cost of 17.7ns. These numbers are higher than the median latency of 7-8ns reported by the microbenchmarks, but they are still reasonably fast.

Lastly, we compared the compilation time and binary size of RAMCloud with and without NanoLog. Without NanoLog, building RAMCloud takes 6 minutes and results in a binary with the size of 123 MB. With NanoLog, the build time increased by 25 seconds (+7%), and the size of the binary increased to 130 MB (+6%). The dictionary of static log information amounted to 229KB for 922 log statements ($\sim$ 248B/message). The log message count differs from Table 2 because RAMCloud compiles out log messages depending on build parameters.

5.5 Throughput Bottlenecks

NanoLog's performance is limited by I/O bandwidth in two ways. First, the I/O bandwidth itself is a bottleneck. Second, the compaction that NanoLog performs in order to reduce the I/O cost can make NanoLog compute bound as I/O speeds improve. Figure 8 explores the limits of the system by removing these bottlenecks.

Compaction plays a large role in improving NanoLog's throughput, even for our relatively fast flash devices (250MB/s). The "Full System" as described in the paper achieves a throughput of nearly 77 million operations per second while the "No Compact" system only achieves about 13 million operations per second. This is due to the 5x difference in I/O size; the full system outputs 3-4 bytes per message while the no compaction system outputs about 16 bytes per message.

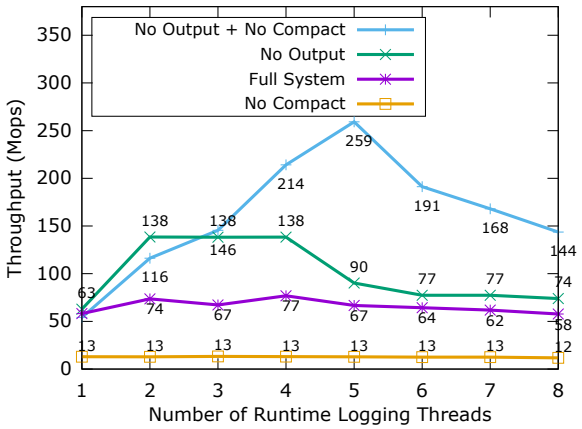

Figure 8: Runtime log message throughput acheived by the NanoLog system as the number of logging threads is increased. For each point, 2^{27} (about 134M) static messages were logged. The *Full System* is the NanoLog system as described in this paper, *No Output* pipes the log output to /dev/null, *No Compact* omits compaction in the NanoLog compression thread and directly outputs the staging buffers' contents, and *No Output + No Compact* is a combination of the the last two.

If we remove the I/O bottleneck altogether by redirecting the log file to /dev/null, NanoLog "No Output" achieves an even higher peak throughput of 138 million logs per second. At this point, the compaction becomes the bottleneck of the system. Removing both compaction and I/O allows the "No Output + No Compact" system to push upwards of 259 million operations per second.

Since the "Full System" throughput was achieved with a 250MB/s disk and the "No Output" has roughly twice the throughput, one might assume that compaction would become the bottleneck with I/O devices twice as fast as ours (500MB/s). However, that would be incorrect. To maintain the 138 million logs per second without compaction, one would need an I/O device capable of 2.24GB/s (138e6 logs/sec x 16B).

Lastly, we suspect we were unable to measure the maximum processing potential of the NanoLog compaction thread in "No Output + No Compact." Our machines only had 4 physical cores with 2 hyperthreads each; beyond 4-5, the logging threads start competing with the background thread for physical CPU resources, lowering throughput.

5.6 Compression Efficiency

NanoLog's compression mechanism is not very sophisticated in comparison to alternatives such as gzip [9] and Google snappy [37]. However, in this section we show that for logging applications, NanoLog's approach provides a better overall balance between compression efficiency and execution time.

Figure 9 compares NanoLog, gzip, and snappy using 93 test cases with varying argument types and lengths chosen to cover a range of log messages and show the

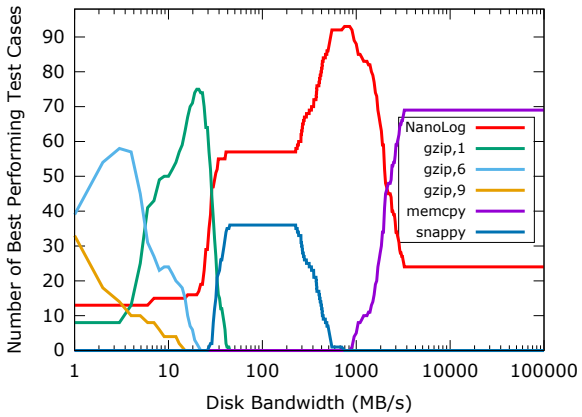

Figure 9: Shows the number of test cases (out of 93) for which a compression algorithm attained the highest throughput. Here, throughput is defined as the minimum of an algorithm's compression throughput and I/O throughput (determined by output size and bandwidth). The numbers after the "gzip" labels indicate compression level and "memcpy" represents "no compression". The input test cases were 64MB chunks of binary NanoLog logs with arguments varied in 4 dimensions: argument type (int/long/double/string), number of arguments, entropy, and value range. Strings had [10, 15, 20, 30, 45, 60, 100] characters and an entropy of "random", "zipfian" (θ=0.99), and "Top1000" (sentences generated using the top 1000 words from [26]). The numeric types had [1,2,3,4,6,10] arguments, an entropy of "random" or "sequential," and value ranges of "up to 2 bytes" and "at least half the container".

best and worst of each algorithm. For each test case and compression algorithm combination, we measured the total logging throughput at a given I/O bandwidth. Here, the throughput is determined by the lower of the compression throughput and I/O throughput (i.e. time to output the compressed data). Since the background thread overlaps the two operations, the slower operation is ultimately the bottleneck. We then counted the number of test cases where an algorithm produced highest throughput of all algorithms at a given I/O bandwidth and graphed the results in Figure 9.

From Figure 9 we see that aggressive compression only makes sense in low bandwidth situations; gzip,9 produces the best compression, but it uses so much CPU time that it only makes sense for very low bandwidth I/O devices. As I/O bandwidth increases, gzip's CPU time quickly becomes the bottleneck for throughput, and compression algorithms that don't compress as much but operate more quickly become more attractive.

NanoLog provides the highest logging throughput for most test cases in the bandwidth range for modern disks and flash drives (30–2200 MB/s). The cases where NanoLog is not the best are those involving strings and doubles, which NanoLog does not compact; snappy is better for these cases. Surprisingly, NanoLog is sometimes better than memcpy even for devices with extremely high I/O throughput. We suspect this is due to out-of-order execution[16], which can occasionally overlap NanoLog's compression with load/stores of the arguments; this makes NanoLog's compaction effectively free. Overall, NanoLog's compaction scheme is the most efficient given the capability of current I/O devices.

6 Related Work

Many frameworks and libraries have been created to increase visibility in software systems.

The system most similar to NanoLog is Event Tracing for Windows (ETW) [31] with the Windows Software Trace PreProcessor (WPP) [23], which was designed for logging inside the Windows kernel. This system was unbeknownst to us when we designed NanoLog, but WPP appears to use compilation techniques similar to NanoLog. Both use a preprocessor to rewrite log statements to record only binary data at runtime and both utilize a postprocessor to interpret logs. However, ETW with WPP does not appear to be as performant as NanoLog; in fact, it's on par with traditional logging systems with median latencies at 180ns and a throughput of 5.3Mop/s for static strings. Additionally, its postprocessor can only process messages at a rate of 10k/second while NanoLog performs at a rate of 500k/second.

There are five main differences between ETW with WPP and NanoLog: (1) ETW is a non-guaranteed logger (meaning it can drop log messages) whereas NanoLog is guaranteed. (2) ETW logs to kernel buffers and uses a separate kernel process to persist them vs. NanoLog's in-application solution. (3) The ETW postprocessor interprets a separate trace message format file to parse the logs whereas NanoLog uses dictionary information embedded in the log. (4) WPP appears to be targeted at Windows Driver Development (only available in WDK), whereas NanoLog is targeted at applications. Finally, (5) NanoLog is an open-source library [47] with public techniques that can ported to other platforms and languages while ETW is proprietary and locked to Windows only. There may be other differences (such as the use of compression) that we cannot ascertain from the documentation since ETW is closed source.

There are also general purpose, application-level loggers such as Log4j2 [43], spdlog [38], glog [11], and Boost log [2]. Like NanoLog, these systems enable applications to specify arbitrarily formatted log statements in code and provide the mechanism to persist the statements to disk. However these systems are slower than NanoLog; they materialize the human-readable log at runtime instead of deferring to post-execution (resulting in a larger log) and do not employ static analysis to generate low-latency, log specific code.

There are also implementations that attempt to provide ultra low-latency logging by restricting the data types or the number of arguments that can be logged [24, 32].

This technique reduces the amount of compute that must occur at runtime, lowering latency. However, NanoLog is able to reach the same level of performance without sacrificing flexibility by employing code generation.

Moving beyond a single machine, there are also distributed tracing tools such as Google Dapper [35], Twitter Zipkin [48], X-Trace [8], and Apache's HTrace [41]. These systems handle the additional complexity of tracking requests as they propagate across software boundaries, such as between machines or processes. In essence, these systems track causality by attaching unique request identifiers with log messages. However, these systems do not accelerate the actual runtime logging mechanism.

Once the logs are created, there are systems and machine learning services that aggregate them to provide analytics and insights [39, 3, 25, 27]. However, for compatibility, these systems typically aggregate full, human-readable logs to perform analytics. The NanoLog aggregator may be able to improve their performance by operating directly on compacted, binary logs, which saves I/O and processing time.

There are also systems that employ dynamic instrumentation [15] to gain visibility into applications at runtime such as Dtrace [14], Pivot Tracing [21], Fay [6], and Enhanced Berkley Packet Filters [13]. These systems eschew the practice of embedding static log statement at compile-time and allow for dynamic modification of the code. They allow for post-compilation insertion of instrumentation and faster iterative debugging, but the downside is that instrumentation must already be in place to enable post mortem debugging.

Lastly, it's worth mentioning that the techniques used by NanoLog and ETW are extremely similar to low-latency RPC/serialization libraries such as Thrift [36], gRPC [12], and Google Protocol Buffers [46]. These systems use a static message specification to name symbolic variables and types (not unlike NanoLog's printf format string) and generate application code to encode/decode the data into succinct I/O optimized formats (similiar to how NanoLog generates the record and compact functions). In summary, the goals and techniques used by NanoLog and RPC systems are similar in flavor, but are applied to different mediums (disk vs. network).

7 Limitations

One limitation of NanoLog is that it currently can only operate on static printf-like format strings. This means that dynamic format strings, C++ streams, and toString() methods would not benefit from NanoLog. While we don't have a performant solution for dynamic format strings, we believe that a stronger preprocessor/compiler extension may be able to extract patterns from C++ streams by looking at types and/or provide a snprintf-like function for toString() methods to generate a intermediate representation for NanoLog.

Additionally, while NanoLog is implemented in C++, we believe it can be extended to any language that exposes source code, since preprocessing and code replacement can be performed in almost any language. The only true limitation is that we would be unable to optimize any logs that are dynamically generated and evaluated (such as with JavaScript's `eval()` [5]).

NanoLog's preprocessor-based approach also creates some deployment issues, since it requires the preprocessor to be integrated in the development tool chain. C++17 NanoLog eliminates this issue using compile-time computation facilities, but not all languages can support this approach.

Lastly, NanoLog currently assumes that logs are stored in a local filesystem. However, it could easily be modified to store logs remotely (either to remotely replicated files or to a remote database). In this case, the throughput of NanoLog will be limited by the throughput of the network and/or remote storage mechanism. Most structured storage systems, such as databases or even main-memory stores, are slow enough that they would severely limit NanoLog performance.

8 Conclusion

NanoLog outperforms traditional logging systems by 1-2 orders of magnitude, both in terms of throughput and latency. It achieves this high performance by statically generating and injecting optimized, log-specific logic into the user application and deferring traditional runtime work, such as formatting and sorting of log messages, to an off-line process. This results in an optimized runtime that only needs to output a compact, binary log file, saving I/O and compute. Furthermore, this log file can be directly consumed by aggregation and log analytics applications, resulting in over an order of magnitude performance improvement due to I/O savings.

With traditional logging systems, developers often have to choose between application visibility or application performance. With the lower overhead of NanoLog, we hope developers will be able to log more often and log in more detail, making the next generation of applications more understandable.

9 Acknowledgements

We would like to thank our shepherd, Patrick Stuedi, and our anonymous reviewers for helping us improve this paper. Thanks to Collin Lee and Henry Qin for providing feedback on the design of NanoLog and Bob Felderman for refusing to use our low latency systems until we had developed better instrumentation. Lastly, thank you to the Industrial Affiliates of the Stanford Platform Lab for supporting this work. Seo Jin Park was additionally supported by Samsung Scholarship.

References

[1] BOLOSKY, W. J., AND SCOTT, M. L. False Sharing and Its Effect on Shared Memory Performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (Berkeley, CA, USA, 1993), Sedms'93, USENIX Association, pp. 3–3.

[2] boost C++ libraries. `http://www.boost.org`.

[3] Datadog. `https://www.datadoghq.com`.

[4] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[5] ECMASCRIPT, ECMA AND EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION AND OTHERS. Ecmascript language specification, 2011.

[6] ERLINGSSON, Ú., PEINADO, M., PETER, S., BUDIU, M., AND MAINAR-RUIZ, G. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 13.

[7] FELDERMAN, B. Personal communication, June 2015. Google.

[8] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation* (2007), USENIX Association, pp. 20–20.

[9] GAILLY, J.-L., AND ADLER, M. gzip. `http://www.gzip.org`.

[10] GNU COMMUNITY. Using the GNU Compiler Collection: Declaring Attributes of Functions. `https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html`, 2002.

[11] GOOGLE. glog: Google Logging Module. `https://github.com/google/glog`.

[12] GOOGLE. gRPC: A high performance, open-source universal RPC framework. `http://www.grpc.io`.

[13] GREGG, B. Linux bpf superpowers. `http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html`, 2016.

[14] GREGG, B., AND MAURO, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.

[15] HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the* (1994), IEEE, pp. 841–850.

[16] INTEL, R. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, June* (2016).

[17] JTC, I. SC22/WG14. ISO/IEC 9899: 2011. *Information Technology Programming languages C*. (2011).

[18] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. USENIX.

[19] The Linux Kernel Organization. `https://www.kernel.org/nonprofit.html`, May 2018.

[20] LMAX Disruptor: High Performance Inter-Thread Messaging Library. `http://lmax-exchange.github.io/disruptor/`.

[21] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 378–393.

[22] memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`, Jan. 2011.

[23] MICROSOFT. Wpp software tracing. `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/wpp-software-tracing`, 2007.

[24] MORTORAY, E. Wait-free queueing and ultra-low latency logging. `https://mortoray.com/2014/05/29/wait-free-queueing-and-ultra-low-latency-logging/`, 2014.

[25] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 26–26.

[26] NORVIG, P. *Natural Language Corpus Data: Beautiful Data*, 2011 (accessed January 3, 2018).

[27] OLINER, A., GANAPATHI, A., AND XU, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61.

[28] OTT, D. Personal communication, June 2015. VMWare.

[29] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7.

[30] PAOLONI, G. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation, September 123* (2010).

[31] PARK, I., AND BUCH, R. Improve Debugging And Performance Tuning With ETW. *MSDN Magazine*, April 2007 (2007).

[32] Performance Utilities. `https://github.com/PlatformLab/PerfUtils`.

[33] printf - C++ Reference. `http://www.cplusplus.com/reference/cstdio/printf/`.

[34] Redis. `http://redis.io`.

[35] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, 2010.

[36] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. *Facebook White Paper 5*, 8 (2007).

[37] Snappy, a fast compressor/decompressor. `https://github.com/google/snappy`.

[38] spdlog: A Super fast C++ logging library. `https://github.com/gabime/spdlog`.

[39] Splunk. `https://www.splunk.com`.

[40] STALLMAN, R. M., MCGRATH, R., AND SMITH, P. *GNU Make: A Program for Directed Compilation*. Free software foundation, 2002.

[41] THE APACHE SOFTWARE FOUNDATION. Apache HTrace: A tracing framework for use with distributed systems. `http://htrace.incubator.apache.org`.

[42] THE APACHE SOFTWARE FOUNDATION. Apache HTTP Server Project. `http://httpd.apache.org`.

[43] THE APACHE SOFTWARE FOUNDATION. Apache Log4j 2. `https://logging.apache.org/log4j/log4j-2.3/manual/async.html`.

[44] THE APACHE SOFTWARE FOUNDATION. Apache Spark. `https://spark.apache.org`.

[45] THE APACHE SOFTWARE FOUNDATION. Log4j2 Location Information. `https://logging.apache.org/log4j/2.x/manual/layouts.html#LocationInformation`.

[46] VARDA, K. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).

[47] YANG, S. NanoLog: an extremely performant nanosecond scale logging system for C++ that exposes a simple printf-like API. `https://github.com/PlatformLab/NanoLog`.

[48] Twitter Zipkin. `http://zipkin.io`.

[49] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.