PyTrace: High-Performance Tracing With Python

Andrew Candelaresi, Joshua Killinger, Austin Longo, Ryan Sullivan

Department of Computer Science, University of Colorado Boulder

Andrew.Candelaresi@colorado.edu

Joshua.Killinger@colorado.edu

Austin.Longo@colorado.edu

Ryan.N.Sullivan@colorado.edu

Abstract

Often times, a high degree of introspection is needed for debugging programs. One common and effective approach is logging program output. A high degree of program output, however, can lead to indeterminate changes in behavior or running characteristics of the program under scrutiny. Not only can this alter the behavior of the bug being investigated, it can also be a substantial disruption to performance sensitive applications.

We propose a model for utilizing high-performance, in-memory, binary tracing to give fine-grained details of Python programs while being as lightweight as possible. To achieve this, we utilize the quick access times of memory by storing the traced data on the heap, and save processing time by only writing out the raw binary representation of each value and leaving the actual "stringification" of those values to a post-processing phase we call decoding.

I. Introduction

Software logging is a critical component to the development process and supportability of small to large scale software applications. The use of runtime logging is pervasive across nearly the entire industry, with many open-source software applications utilizing large amounts of program output statements [2].

One of the more common uses for runtime logging is debugging. This approach to debugging is often an effective one, but has performance implications associated with it. Not only can output statements cause a slowdown in performance and hinder some performance sensitive applications, it can also introduce a change in running characteristics that might mask an issue that is being debugged [1]. This phenomenon is often referred to as a Heisenbug, a software bug that appears to change its behavior or disappear entirely when efforts are made to observe it [3].

The need to write to a log for debugging can add significant overhead to an applications throughput. It is not uncommon in heavy io bound applications to see upwards of 200 MB/s spent on logging alone [1], which is about two thirds of the sustained sequential write of a standard ssd drive. When excessive cycles are spent writing a log to disk the overall performance of an application can take a hit. Since we know that logging is crucial for speed and ease in debugging and we would like to avoid paying these heavy performance penalties of logging to disk we must come up with an alternative. This is where binary trace logging can be applied.

II. Related Work

Much time and effort has been devoted to creating high performance logging frameworks. The primary inspiration for our work is a C++11 implementation developed by T. Göckel and M. McMullen for use in SolidFire's all-flash storage system [1]. The use case here being one that relies heavily on high performance in order to trace individual IO operations through the system. Such an application can have millions of operations occurring every second and logging detailed information on each one of them can have a serious performance impact if not done in the most performant way possible.

Another example of this is Pantheois [8]. Pantheois is a C++ based logging API library, that leverages existing transport mechanisms of feature-rich diagnostic logging libraries such as ACE, Log4cplus, or log4cxx with the peerless features afforded by the Pantheois architecture: 100% type safety, high efficiency, not paying for what you don't use. These optimizations provide significant speed ups to logging but fail to address the underlying issue of the time spent writing logs.

Other implementations were also investigated to determine the best possible approach for our Python solution [4][5][6].

III. Implementation

AST and Compiler Changes

For our project we were not interested in implementing a logging framework. We therefore did not concern ourselves with the source language for what to log, but instead decided to implicitly log all function definitions and all function calls. We did this by creating two new AST nodes with the following signatures.

```
class LogDef(Node):
    def __init__(self, expr):
        self.expr = expr
    def __repr__(self):
        return "LogDef({})".format(self.expr)
    def getChildren(self):
        return (self.expr)

class LogCall(Node):
    def __init__(self, expr):
        self.expr = expr
    def __repr__(self):
        return "LogCall({})".format(self.expr)
    def getChildren(self):
        return (self.expr)
```

We add these nodes during the function closure pass of our compiler. When we create the function closure we have both the name of the function and the pointer to the function code block in the assembly. So this is when we add our LogDef AST node into the object code. This signifies a call to the runtime to create a mapping from the function name to the pointer. Then when we see a function call we add our LogCall AST node into the object code. This signifies a call to the runtime to log the function call.

In our flatten phase we process the log nodes and package them for our select x86 phase. For the LogDef node we simply add a line to our flattened instructions to call `trace_define_func` from the run time with the function ptr created in the closure, the name of the function and the number of args the function takes. For LogCall we add a line into our flattened instructions to call `trace_log_call` from the runtime with the function pointer returned from `get_fun_ptr`, the number of args the function takes and a pointer to the last argument.

In our select x86 phase we parse the flatten instructions and create the correct assembly instructions to create logs. For the LogDef node we need to push the function name, pointer and the number of arguments onto the stack and call the runtime to create a log for that function definition. For the LogCall we push the parameters and the stack pointer to the runtime logging call, where the stack pointer is used as a reference to the position of the last parameter. We then push the number of args and the function pointer onto the stack and call the runtime to create a log for the function call. In this phase we also insert an instruction to call `trace_init` which tells the runtime to create and open the log file. We also had to add our function names as strings in the .data section of the assembly, so that we can push them as strings to the `trace_define_func` call. This call creates an entry for that function mapping the function pointer to the actual name for use in decoding the logs later.

We added a log level command line argument to our compiler to support no logging, binary tracing, and verbose logging. This was done by passing a flag to the flatten phase and based on that flag including or omitting the calls to the runtime to log in our output assembly. Our compiler implementation relies on the run time to do the heavy lifting of creating the logs which will be discussed in the following sections.

Runtime Interface

In order to implement these features into our compiler, we extended the runtime by implementing a lightweight interface for logging data to disk. During execution, we track only the values of function pointers, values of parameters, and a timestamp for the function call. Logs are stored in binary form, and not formatted into strings as they would be for a human readable log. Information for decoding the function calls into a human-readable format is also stored.

Initially this metadata was stored in a header for the log file. In C-like implementations, this approach is sufficient. However, it presented an issue in a dynamic language such as Python, where functions can be defined inside other functions. Since we save the data for decoding into a human readable form when functions are defined, functions could still be defined after a function had been called, thus corrupting the data. While one solution to this issue could have been to perform a static analysis of the program and injecting header definitions at the start of the main function instead of during the process for creating a closure, we instead chose the simpler solution of splitting the header and log into two separate files. This approach also simplifies the process of decoding. In addition, separating the header and the log allows for a wider variety of type definitions in the log, without increasing the complexity of how data is defined and stored. This is detailed further in Future Work.

Prior to all logging and definition calls, a special call must be made to the runtime in order to initialize the log files. This call opens a file to store the header data, and a file to store the log data using stdio. Buffer specification can be changed, but a 2 kilobyte buffer was used for our testing. We rely on the underlying implementation of stdio to flush and close the files on program termination.

The runtime provides calls for both compact and verbose logging, in order to acquire benchmarking results for both forms. Both methods require calls to define the functions before the functions are used. In the compact logging, the

definitions are written to the header file. In verbose logging, a hashtable of function pointer to a string with a human readable name for the function is kept in memory. In this hashtable, no collisions are possible, since the function pointer is treated as the hash. This allowed for simpler, faster, hashtable usage.

When a function call is logged, the function pointer is used to look up the function name in the hashtable. The function name, followed by each parameter, plus the timestamp, are formatted into human readable ASCII strings, and written to the log file in CSV format. Compact logging simply writes these values in binary format.

The interface definition for tracing is defined as follows:

```
int trace_init();
int trace_define_string(char* str_ptr);
int trace_define_func(void* fun_ptr, char*
name_ptr, int argc);
int trace_define_func_verbose(void* fun_ptr, char*
name_ptr, int argc);
int trace_log_call(void* fun_ptr, int argc, pyobj*
argv);
int trace_verbose_call(void* fun_ptr, int argc,
pyobj* argv);
```

The signatures of the verbose and binary logging functions are identical in order to simplify implementation in the compiler.

Varargs

One requirement for implementing the logging functions was the ability to handle any number of values as arguments, since there is no limit to the number of parameters that can be passed to a function call in Python. In order to implement this, we treat the stack as if it were an array.

		Stack	
	previous frame	. . .	
		arg2	
	function args	arg1	
		arg0	<-argv points here
		argv	
	tracelog args	3	argc
		$fun_ptr	

Fig 1. Snapshot of a representation of the stack during a call to trace_log_call

Before calling a trace log function, the parameters to the function being logged are pushed onto the stack as if the function were being called. We then push the memory address of the first argument, which is `pyobj* argv` in the trace function. The number of arguments is `int argc`, and the pointer to the function being logged is `void* fun_ptr`. With the pointer to the first stack location of the logged function's arguments, and the number of arguments, we iterate over `argv` using C's array notation. Figure 1 shows an example of a call stack when tracing a function with 3 arguments.

Since the arguments to the logged function are already on the stack, the compiler can perform a small optimization to only pop the formal arguments to the logging function before calling the logged function.

Data Formats

We allowed for both strings and functions to be defined in the header. Both are able to be decoded by referencing their pointer. All entries into the header file begin with the pointer to the entity. The pointer is followed by a null terminated string in both cases, but string definitions and function definitions are differentiated by function definitions having a null byte separating the pointer value and the string. Function definitions then contain an integer specifying the number of arguments the function takes, followed by an additional null byte.

Call log entries begin with the function pointer. The pointer is immediately followed by the argument values. The argument values are in tagged form to preserve semantics of the data. After the arguments, the log entry is finalized with an 8-byte timestamp from a call to `rdtsc()`.

Timestamping

Our timestamps are read directly from the processor's time-stamp counter using RDTSC, or Read Time-Stamp Counter[7]. The timestamp represents the number of clock cycles since startup. This is a 64-bit value that gets stored across the EDX and EAX registers. We implemented an inline function in the runtime for retrieving this value.

```
static __inline__ unsigned long long rdtsc(void)
{
       unsigned hi, lo;
       __asm__ __volatile__ ("rdtsc" : "=a"(lo),
"=d"(hi));
       return ( (unsigned long long)lo)|(
((unsigned long long)hi)<<32 );
}
```

The use of a clock-cycle timestamp makes for the most performant approach, but also limits the portability of our implementation. In order to decode these timestamps into human-readable form, one must know the exact clock frequency of the CPU they were generated on to determine how many ticks translate to how many milliseconds or seconds. Ideally, the user would decode the trace logs on the same CPU they were generated from in order to know this

information during post-processing. This information could also be stored in the header file.

Decoding

With a well-defined data format, decoding logs becomes relatively simple, and since decoding is not required at runtime, it has no impact on runtime performance. As a result, the decoder can be implemented in the language and framework of choice. For our purposes, the decoder was written in C# on the Mono framework.

The implementation is fairly straight-forward, with opening and parsing the header file into a dictionary of pointers to function definitions. The log file is then opened and parsed using the function dictionary in order to convert the log into a CSV format. This decoded format is identical to the format used during verbose logging. Any format could be used for the decoded log, but we chose CSV for simplicity.

IV. Project Evaluation

To evaluate our binary tracing implementation we utilized functions and while loops from the P1 and P2 subsets, respectively. Given our implementation does implicit logging on each function invocation, we knew we would need functions to be utilized. With our limited supported subset of Python, we chose to structure our tests within while loops to generate a sufficient amount of log statements in an attempt to stress the system. Our benchmarking test is shown below, along with a snippet of its decoded binary trace log.

pytrace-test1.py:

```
def plus_one(x):
        return x + 1
def negate(x):
        return -x
def plus_two(x):
        return x + 2
x = 0
while x != 400000:
        x = plus_one(x)
        x = negate(x)
        x = negate(x)
        x = plus_one(x)
        x = plus_two(x)
print x
```

Output:

```
lambda_1, 0, 53307436507581
lambda_2, 1, 53307436514201
lambda_2,-1, 53307436516242
lambda_1, 1, 53307436518327
lambda_3, 2, 53307436520257
lambda_1, 4, 53307436522213
...
```

A second test was used as an additional benchmark. The function performed by this test was computing a matrix product. Matrix multiplication was deemed a valuable operation to test as it is both a common operation, and a computationally intensive one in practice.

Our evaluation was done by measuring runtimes of the test program with verbose logging, binary tracing, and no logging. Several iterations of each test were run to get a statistically relevant averaged result across runs. Varying iterations of the program loop -- 4000, 40000, and 400000 -- were also used to measure the impact of increased stress on the system. In the case of the matrix product program loop, the numbers of iterations used were one-tenth of the aforementioned due to the increased depth of the while loops, and thus, frequency of logging per iteration.

V. Results

Our results showed an average runtime decrease of 50% when comparing binary tracing to verbose logging. At the same time, binary tracing introduced a roughly 40% overhead when compared to no logging. Fig 2 below shows these results.

These disparities only became clearly evident at higher iterations of the loop, and the chart below indicates that verbose logging has a much sharper rise in performance degradation as the number of log statements increases.

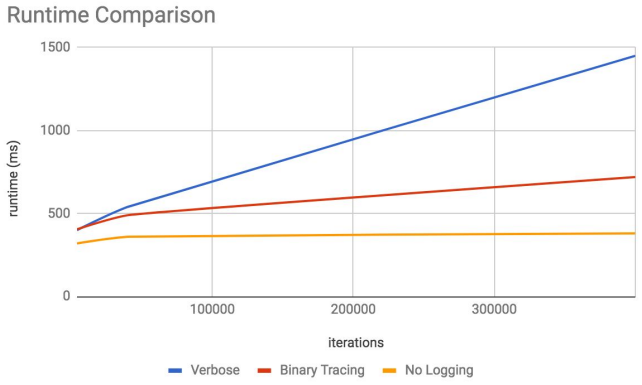

Fig 2. Comparisons of verbose logging, binary trace logging, and no logging on a simple P2 python program using while loops and functions.

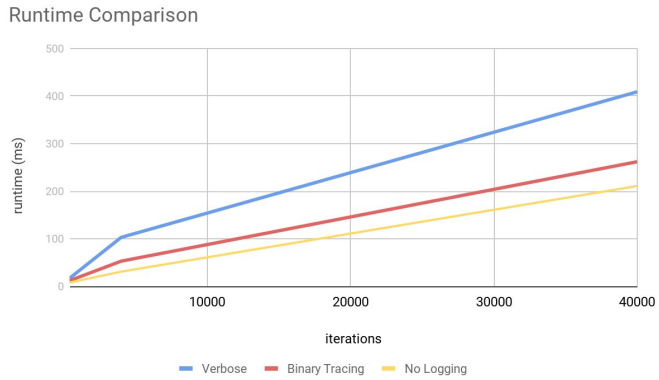

Fig 3. Comparisons of verbose logging, binary trace logging, and no logging on a complex P2 python program using many nested while loops to compute the matrix product.

VI. Conclusion

Runtime logging is a crucial and pervasive practice in the development of software applications. It is often an effective tool in debugging anomalies, but can also introduce indeterminate changes in behavior. Given its usefulness, attempts to mitigate the performance hit taken from logging provide an interesting and worthwhile area for research. From our results it is apparent that there is an advantage to be gained from using binary trace logging. We can see from the graph above that the binary trace logging beat the verbose logging by almost a full second as the number of iterations grew towards 400,000. While tracing did add general overhead to the performance time, logging is a necessity in most production systems.

VII. Future Work

Strings

While the initial implementation includes the ability to define string constants, it does not yet utilize this feature. The feature was intended as a supporting mechanism for classes, since our compiler uses strings to reference class attributes. It could be expanded upon, however, in order to handle user-defined string constants, useful for marking certain locations in code to improve speed of debugging.

Classes and Objects

Implementing logging for classes and objects would present a bit more of a challenge, more through specification in the log and header than on the logging side. More robust differentiation would need to be created for specifying entries in the header file. This should not present too much of a hurdle, since the header file should not be consistently written to. Leading the header entries with a type specification character would be sufficiently lightweight for purposes of keeping formatting and file write costs down. Dictionary support would also be very helpful for class and object support.

User Specified Dumps

Another useful addition to our tracing system would be to allow users to specify explicit calls to write values to the log. This could potentially be done for booleans and integers by writing a null pointer to the log, followed by the value.

Lists

A current shortcoming of our implementation that would also be very useful is the ability to write lists to the log. As above, this could be done on the header side by simply creating an entry that is a pointer to the list. When an entry in the log begins with this pointer, it can be followed by the number of elements in the list, followed by each element's value. In order to keep runtime costs down, the lists would have to be explicitly dumped to the log at user-defined locations. Recognizing the usage of a list and automatically dumping it would be detrimental to runtime performance.

Dictionaries

Dictionaries present a greater challenge, as the implementation for them does not currently expose a way to get all key-value pairs in the dictionary. The dictionary implementation, as well as the underlying hashtable implementation, would have to be extended to expose this information. Exposing this is not atypical for high-level language dictionary implementations. Like lists, the header entry would simply be the pointer to the dictionary. Writing dictionaries to the log would require explicit calls by the user to dump them. The log would write the pointer for the dictionary, followed by the number of entries, then values of each key and its corresponding value in pairs. Extending the dictionary interface would also open the door for extending the implemented language subset to include operations over keys, values, and key-value pairs as a collection, without requiring knowledge of the keys.

VIII. References

[1] T. Göckel and M. McMullen. "High-Performance Tracing With C++11". [Online] Available: https://github.com/tgockel/binary-tracing/blob/master/binary-tracing.pdf

[2] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 102-112.

[3] Gray, Jim. (1985) "Why Do Computers Stop and What Can Be Done About It?" Cupertino: Tandem Technical

Report. 85.7.

[4] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In Proceedings of the 2nd international conference on Virtual execution environments (VEE '06). ACM, New York, NY, USA, 154-163. DOI=http://dx.doi.org/10.1145/1134760.1220164

[5] IBM. "Binary Logging." *IBM Knowledge Center*. N.p., 21 Aug. 2017. Web. 04 Dec. 2017.

[6] Cisco. "Event Tracer." *Cisco*. N.p., 18 Mar. 2015. Web. 03 Dec. 2017.

[7] x86 Instruction Set Reference (RDTSC) https://c9x.me/x86/html/file_module_x86_id_278.html

[8] Anon. Pantheios. Retrieved December 03, 2017 from http://www.pantheios.org/essentials.html