3-1-2019

Efficient and Scalable Event Tracing

Rupika Dikkala
Portland State University

10.15760/honors.780

Efficient and Scalable Event Tracing

by

Rupika Dikkala

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Karen Karavanic

Portland State University

2019

Efficient and Scalable Event Tracing

Rupika Dikkala
Department of Computer Science
Portland State University
rdikkala@pdx.edu

ABSTRACT

In this work, I demonstrate that a time series database can be utilized to store Open Trace Format 2 (OTF2) file metadata for common trace events efficiently and scalably. This paper examines the efficacy of storing event trace data in a time series database, and investigates associated performance overhead compared to the state of the art method using OTF2 trace files. The sample traces used in this project are generated from a parallel hydrodynamic modeling code, *Lulesh,* developed at Lawrence Livermore National Laboratory. In my approach, I first cache common event trace metadata in InfluxDB, a contemporary time series database. Next, I compare the runtime performance of various metrics by executing InfluxQL queries on InfluxDB, and using corresponding one-pass algorithms on the OTF2 trace files. My results reflect an exponential performance improvement benefitting the InfluxDB technique.

KEYWORDS

High performance computing, Open Trace Format 2, time series database, Influx database, trace file

INTRODUCTION

The largest, most powerful computers in the world run scientific applications, data analytics, and commercial applications such as databases and web servers. While these use an immense amount of computing power, the pattern of each application differs. For example, a cell phone network or web server can potentially serve millions of requests largely independent from one another. On the other hand, a parallel science simulation using a message passing interface (MPI), might include thousands of processes (or threads) that exchange intermediate results as the computation progresses, requiring special interconnection networks with low latency. Although high end data analytics involve copious amounts of data, the focus is primarily on bandwidth over latency to accommodate high rates of data for computation.

High Performance Computing (HPC) is defined as:

> the use of super computers and parallel processing techniques for solving complex computational problems, and its ability to deliver a sustained performance through the concurrent use of computing resources [13].

HPC applications are found in a multitude of advances. For example, geographical models rely on predictions facilitated by the time-critical data corresponding to numerous physical, chemical, and biological properties [5]. For example, natural disaster simulations rely on high performance computing for integrated earthquake simulations, that analyze wave propagation and amplification processes in underground structures [4]. HPC is a powerful method of computation consisting of undeniable extremes: even a subtle variance during runtime causes a considerable difference in performance. Reliability and scalability are two of the most critical characteristics in HPC systems as both depend on the size of the application and the platform it runs on [16].

This study will investigate how a time series database can be utilized to store Open Trace Format 2 (OTF2) file metadata for common trace events efficiently and scalably.

BACKGROUND

A common technique employed to optimize performance in HPC applications is to identify performance bottlenecks using trace analyses. A trace analysis is performed by examining event traces, which are a collection of time stamped records of various events [12], output from the application. An event is a runtime occurrence of any of the following program activities: machine instructions, basic block executions, memory references, function calls, or a message send/receive. The challenges associated with storing event traces in a trace file format is described by Mohror et al:

> … collection of event traces presents scalability challenges: the act of measurement perturbs the target application; and the large volume of collected data increases the perturbation, and results in data files that are difficult, or even impossible, to store and analyze [12].

Tremendous amounts of trace data output from event traces abstracted from HPC applications are produced at a high frequency, and must be managed and processed efficiently in order to conduct a definitive performance analysis. As applications and runtime platforms increase in size, the efficiency of trace processing itself is reduced [18]. HPC systems are especially subject to this condition because of their scale - even a single millisecond improvement in runtime performance of a loop iteration can drastically affect efficiency. Technology becoming increasingly interconnected in day-to-day life necessitates HPC applications, therefore creating an urgency to address the performance drawbacks that accompany ineffectual trace processing. More efficient tracing allow for a more accurate performance analysis, and reduce the time to solution.

A time series database (TSDB)[12] is structured for analytical data containing time stamps such as metrics, sensors, or networks. As aforementioned, event traces are a set of time stamped events, and thus justify the use of a TSDB. Time series databases are utilized in various contemporary applications, including the stock market using the ARIMA (Autoregressive Integrated Moving Average) statistical model [1] and in machine learning predictive software [3].

The time series database used in this study is the Influx Database (InfluxDB)[11]. InfluxDB uses a highly efficient data collection technique that provides higher scalability than the Postgres database [13]. For example, InfluxDB is used in SensorCloud [11], a sensor observation service,

and its geospatial consortium utilizes the database to manage and query observation data. However, they have not yet been used for storing HPC performance metrics and event trace metadata. With the current trend of growing data capacities, ensuring optimized runtime and memory overhead are increasingly critical. Combining OTF2's file compression technology that conserves memory [2], and InfluxDB's runtime efficiency has the potential to generate a powerful tool that surpasses the current capabilities of trace file storage.

METHODS

The OTF2 trace files used in this study were provided by Dr. Kevin Huck of the Parallel Processing Performance Research Lab at the University of Oregon. InfluxDB [6] was elected over other time series databases because of its reliability and scalability in professional industry as mentioned in the Background section. Scripts were developed using the Python 3 programming language [14], as it had a readily available and well documented `InfluxDB-Python` client-server library used to transfer metadata from OTF2 file to the database [17]. All code was run on Portland State University's server `ada.cs.pdx.edu`, on a virtual machine with 12 vcpus and 24 gigabytes of RAM. `ada` comprises 2x Intel(R) Xeon(R) CPU E5-2670 [7] version 2 at 2.50 gigahertz CPUs and 256 gigabytes of memory.

Over the course of my research, I first transferred trace file metadata into InfluxDB, and afterward conducted a measurement study comparing the execution run times of various metrics on data in OTF2 files and InfluxDB. I familiarized myself with the OTF2 library [2] and InfluxDB schema as a prerequisite step, and modified OTF2 library source code to distinguish and output event trace metadata from all events occurring in the trace file.

Metadata Transfer from OTF2 Files into InfluxDB

Before streamlining OTF2 trace file metadata into InfluxDB, I first identified the total number of events and their respective counts (Table 1) in the *Lulesh* OTF2 trace file. I executed `otf2-print`, a built-in reader tool in the OTF2 library supplemented by a Python counting script to tally the metadata. Next, I wrote a similar Python script assisting the library file `otf2_reader_example.c`, which contains the source code for reading and printing attributes of each event in the trace file. This check was performed to confirm both the built-in library tool and `otf2_reader_example.c` outputs were consistent with one another, because `otf2_reader_example.c` would be modified to print all the event trace metadata in *Lulesh.*

Event Name	***Count***
mpi_receive	219404
mpi_send	219404
leave	45153394
enter	45153466
rma_win_create	8
metric	893880
TOTAL	91639556

Table 1. Events found in the Lulesh OTF2 trace file with respective counts

Once the check was complete, I used `otf2_reader_example.c` to print the attributes from each event to the command line by writing modified print functions based on the given function for the "Enter" event. I determined the parameters that defined each event attribute by carefully examining the following header files in the OTF2 library:

- `OTF2_GlobalEvtReadervCallbacks.h`
- `OTF2_AttributeList.h`
- `OTF2_GeneralDefinitions.h`

The command used for compiling and printing the trace metadata from `otf2_reader_example.c` is as follows:

```
gcc -g -O0 -o a.out -I ../otf2-2.1.1/include otf2_reader_example.c -L ../otf2-2.1.1/build-backend/.libs/ -lotf2 -lm; ./a.out
```

After printing the event trace metadata from the OTF2 file to the command line, this output was piped in real time through a Python script, `otf_to_influx.py`, that directly inserted trace file data into InfluxDB. `otf_to_influx.py` enlists the Python library `InfluxDB-Python`, which establishes a connection directly to the InfluxDB server, and carries out client-interaction with the database. The OTF2 metadata printed to the command line is extracted as data points, and is stored in an array of JSON (JavaScript Object Notation) [8] structures. These JSON arrays are then injected into `InfluxDB-Python` library's `write_points` function [17], which consequently inserts the points in InfluxDB (Table 2).

measurement	*enter*	*leave*	*mpi_send*	*mpi_receieve*	*metrics*	*rma_win_create*
tags	region id	region id	msg_tag msg_rank id	msg_tag msg_rank id	metric id	win_id id
fields	location	location	comm_id length location	comm_id length location	location metric_num metric_value type_id	location

Table 2. InfluxDB Schema developed for the "`otf_influx`" database. The metadata is classified into six tables, or "measurements", corresponding to each of the events found in Lulesh's OTF2 file, and is further characterized into a "tag" or "field" depending on the data type.

Generating and Testing Metrics

In order to construct a fair comparison for metric execution times between the OTF2 trace files and InfluxDB, I wrote and implemented Python scripts with one-pass algorithms that would extricate metadata from the trace files, and executed comparable InfluxQL queries on InfluxDB (Figure 1). The metrics are split into two categories: event type per second and specific event counts. To ensure consistency, each metric was run 30 times on both approaches. Tests for both the OTF2 trace file and InfluxDB were administered on the command line, and invoke the Linux `time` command [10] to measure the execution runtime of each metric. Sample commands used to run the OTF2 file and InfluxQL query metrics are listed below:

InfluxQL Query:
```
{ time ./influx -database 'otf_influx' -execute "<query>"; } &>> query_output.txt
```

OTF2 File:
```
gcc -g -O0 -o a.out -I ../otf2-2.1.1/include otf2_reader_example.c -L ../otf2-2.1.1/build-backend/.libs/ -lotf2 -lm; ./a.out | { time python3 ../<otf2_algorithm.py>; } &>> otf2_output.txt
```

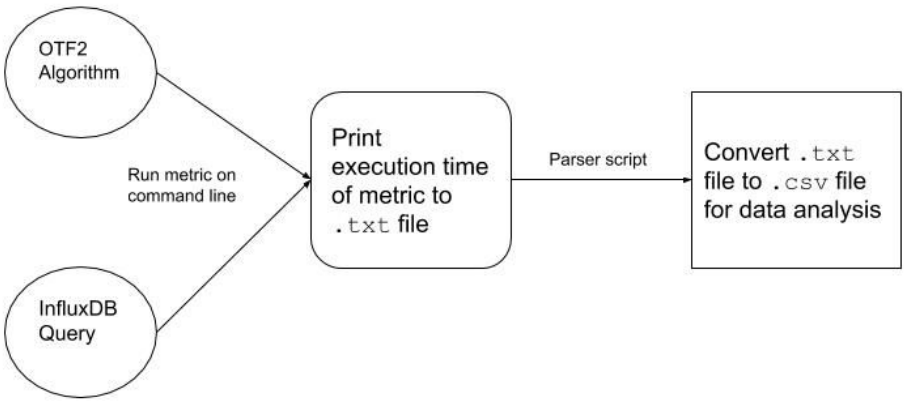

Figure 1. Workflow carried out to measure metrics and gather execution time data

Events Per Second

These metrics measure how frequently an Enter, MPI Send, or MPI Receive event occurs per second. I first noted the time range of the data in the trace, which is as follows: May 21, 2018 18:51:24.69241 to May 21, 2018 18:51:51.348201. These timestamps indicate a ~27 second gap between the first and last data points. I enlisted Unix Shell scripting to increment each timestamp in one second intervals from 24 to 51 seconds. In real time, these metrics can provide valuable insight into the performance of specific blocks of code based on the amount of data being processed.

Sample query: `select count(*) from <enter, mpi_send, mpi_receieve> where time >= '2018-05-21T18:51:24.000Z' and time <= '2018-05-21T18:51:25.000Z'`

Event Counts

The following metrics count unique characteristics pertaining to a specific event, with a focus on the Enter, MPI Receive, and MPI Send events.

Number of Enters per Region - counts the number of enters per region. In real time, this metric can be used as a baseline to count the number of enters per function in a trace file.

Enters per Region Query: `select count(*) from enter group by region`

Number of Messages Sent/Received per Message Tag - Counts the number of MPI messages sent/received per unique message tag. In real time, this metric can be used to track the consistency of MPI message interactions in the trace file.

Send Message Query: `select count(*) from mpi_send group by msg_tag`

Receive Message Query: `select count(*) from mpi_recv group by msg_tag`

Number of Messages Sent/Received per Message Rank - Counts the number of MPI messages sent/received per unique message rank. In real time, this metric could be used to define existing communications between various message ranks

Send Message Query: `select count(*) from mpi_send group by rank`

Receive Message Query: `select count(*) from mpi_recv group by rank`

Number of Unique Locations in a Region - Counts the number of unique locations per region in the Enter event. In real time, this measurement can be used to mimic the depth of the stack of various functions in a program.

Unique Locations per Region Query: `select count(distinct(location)) from enter group by region`

RESULTS

In this section, I run various statistical measures and outline the results from my experiments conducted on the trace file and database.

Metric	OTF2 File			InfluxQL Query			Speed Up Ratio of InfluxQL Query (%)
	Median (ms)	*Mean (ms)*	*Standard Deviation*	*Median (ms)*	*Mean (ms)*	*Standard Deviation*	
Enters per Second	66360.5	68267.5	5165.182	108	108.533	5.045	614.449
Receives per Second	67551.25	70473.85	7047.093	21.25	21.4	2.179	3178.882
Sends per Second	72175.25	74836.45	7355.925	20.5	20.217	1.675	3520.744
Enters per Region	151386	152291.467	8133.053	3792.5	4033.6	959.228	39.917
Sends per Message Rank	116719	116790.733	5138.217	78	82.5	25.366	1496.397
Receives per Message Rank	115091	115077.533	4561.714	64	71.233	22.258	1798.297
Sends per Message Tag	115929	116998.5	5169.947	50.5	58.8	18.224	2295.624
Receives per Message Tag	118175	118205.433	6166.284	52	53.267	12.303	2272.596
Locations per Region	110471	110665.3	4192.704	97374	97304.767	2523.541	1.135

Table 3. Calculated the mean, median, and standard deviation from the execution times of each approach. I later took the ratio of the performance speed up by dividing InfluxQL median time by OTF2 file median time for each metric.

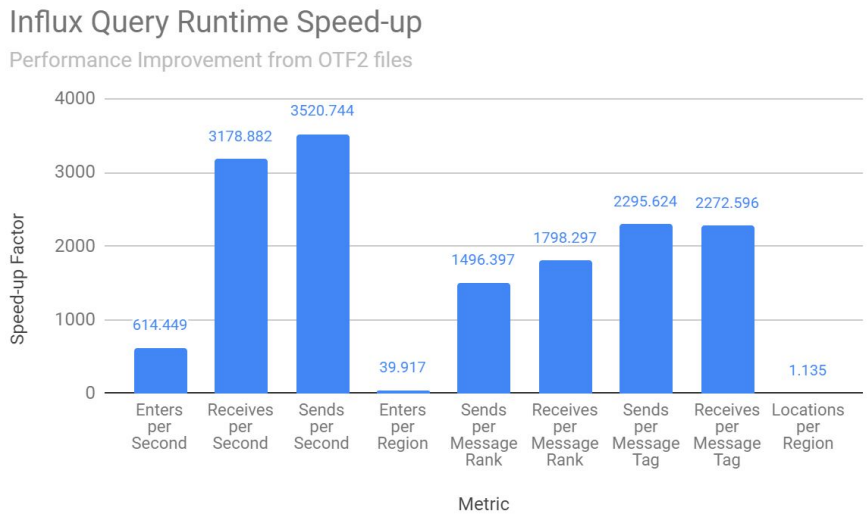

Figure 2. Speed-up graph showcasing the ratio of median InfluxQL query execution time to median OTF2 file execution time on various metric

DISCUSSION

As delineated in Figure 2 and Table 3, the InfluxQL queries outperform OTF2 file algorithms in all the metrics tested. Although there is variance in the data collected, it exhibits a persisting upward trend. The highest performance improvement is indicated in "MPI Sends per Second" with an increase of approximately 3500%. The data presents two outliers, namely "Enters per Region" and "Locations per Region" with speed-ups of approximately 40% and 1.1% respectively. While the query for "Enters per Region" performs up to 40 times faster than the OTF2 algorithm, it does not measure up to the projections illustrated by other metrics. The farthest outlier is "Locations per Region", with its query performing only 1.1% faster than the OTF2 approach. The complexity of this query could have attributed to the lack of indexing in the database; adding the `distinct` filter further complicates the query, and is more likely to retrieve the data manually.

Troubleshooting during Methods

I encountered multiple cardinality issues while streamlining the OTF2 trace file data into InfluxDB. First, I noticed data points were not properly copied into the database: data containing the same tag and timestamp were overwritten. Let us examine the following set of data points written using line protocol, the format used to insert points into InfluxDB [6]:

Point 1: **enter,region=6 location=65 100**
Point 2: **enter,region=6 location=80 100**

where *measurement:* **enter**, *tag:* **region**, *field:* **location**, and *time:* **100**

Despite both points having different location values, the tag and time (region=6 and 100) are the same, and therefore *enter,region=6 location=80 100* will overwrite *enter,region=6 location=65 100***.** This method of handling duplicate points is a design consideration by InfluxDB [6], so I attempted to resolve this issue by adding a unique id `uuid` tag to prevent overwriting.

After incorporating the unique id, I encountered a `max values per tag limit exceeded` error. To combat this inaccuracy, I disabled the `max_values_per_tag` parameter in the configuration file found in InfluxDB source code. The maximum limit is set to 100,000 by default, but terminating this limit was necessary to accommodate the millions of data points found in the *Lulesh* trace file. Subsequently, I encountered a new `max_series_per_database limit exceeded (1,000,000)` error. Although this limit can also be disabled, InfluxDB documentation characterizes this error as a warning to redesign the current schema [6]. I believe this series error arose because of the aforementioned `uuid` tag; the database was sorting tags by `uuid` instead of the tags assigned to trace file metadata (such as "region" from the Enter event), and a series overflow defeats the purpose of indexing by tag because it is not scalable. I circumvented this issue by designing a new schema that inversely grouped `uuid` based on the timestamp of the data: if two or more points had the same timestamp, then the corresponding `uuid`'s are unique.

During data transfer from OTF2 trace files to InfluxDB, there was substantial performance overhead resulting in a sizable bottleneck. As cited in the Methods section, my `otf_to_influx.py` script primarily relied

on the `write_points` function from the `InfluxDB-Python` library to transmit data from the command line. The batching parameter in `write_points` was set to 10,000 as recommended by InfluxDB documentation [6], and it took ~90 minutes to read in the ~92 million points from the OTF2 trace file into InfluxDB. In an effort to optimize my script, I implemented a multithreaded version. The resulting time was approximately 45 minutes to read in just 10 million points — unquestionably slower than before. I initially assumed this resulted from an inefficient threading implementation, however after running `cProfile` [15], a tool for classifying memory usage, on my script, I learned the overhead was caused by the method of reading in OTF2 trace data. Currently event trace data is printed to the command line from the OTF2 file, and that output is parsed by the script in a `for` loop which is the main culprit behind the bottleneck.

FUTURE WORK

Suggestions for moving forward in this study include but are not limited to:

1. Optimize data transfer overhead
Instead of printing the trace file metadata to the command line, this intermediary step can be completely eliminated from the workflow by directly piping JSON structures from `otf_reader_example.c` into `otf_to_influx.py`.

2. Optimize OTF2 algorithms
By maximising the efficiency of OTF2 algorithms, future researchers can derive more consistent results and avoid variance.

3. Test multiple datasets and a wider subset of events
By testing multiple datasets encompassing a wider range of event traces, and therefore events, future researchers can test a larger assortment of metrics of varying degrees of complexity.

CONCLUSION

Over the course of my research, I examined the efficiency of storing common events from OTF2 trace files in a InfluxDB, a time series database. I accomplished this by loading OTF2 event trace metadata into InfluxDB, and subsequently conducting thorough experiments measuring the execution time for obtaining metrics directly from the OTF2 trace file versus executing queries in InfluxDB. My results reflect a significant speedup in extracting metrics from InfluxDB over the OTF2 trace file. Given the limited scope of my project due to time constraints, I only tested one dataset, but my initial hypothesis is proven by the outcomes from a representative subset of trace events used in this study. Access speed and storage efficiency are important properties in HPC platforms and require scalability to process massive amounts of data. My outcomes strongly support utilizing a time series database to store event trace data as a promising approach.

REFERENCES

[1] Angadi, M. C., & Kulkarni, A. P. (2015). Time Series Data Analysis for Stock Market Prediction using Data Mining Techniques with R. *International Journal of Advanced Research in Computer Science, 6*(6).
[2] Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W. E., & Wolf, F. (2011, August). Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *PARCO* (Vol. 22, pp. 481-490).

[3] Fritz, B. A., Chen, Y., Murray-Torres, T. M., Gregory, S., Abdallah, A. B., Kronzer, A., ... & Sharma, A. (2018). Using machine learning techniques to develop forecasting algorithms for postoperative complications: protocol for a retrospective study. *BMJ open*, *8*(4), e020124.

[4] Hori, M., Ichimura, T., Wijerathne, L., Ohtani, H., Chen, J., Fujita, K., & Motoyama, H. (2018). Application of High Performance Computing to Earthquake Hazard and Disaster Estimation in Urban Area. *Frontiers in Built Environment*, *4*, 1.

[5] How HPC is Helping Solve Climate and Weather Forecasting Challenges (2016). *insideHPC*. Retrieved from https://insidehpc.com/2016/01/how-hpc-is-helping-solve-climate-and-weather-forecasting-challenges

[6] InfluxDB. (2019). *InfluxDB 1.7 documentation*. Retrieved from https://docs.influxdata.com/influxdb/v1.7/

[7] Intel. (2019). *Intel® Xeon® Processor E5-2670*. Retrieved from https://ark.intel.com/content/www/us/en/ark/products/64595/intel-xeon-processor-e5-2670-20m-cache-2-60-ghz-8-00-gt-s-intel-qpi.html

[8] JSON. (2019). Retrieved from https://www.json.org/

[9] Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B. L., Cohen, J., Devito, Z., … Still, C. H. (2013). Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*.

[10] Kerrisk, M. (2019). time(1) - Linux manual page. Retrieved from http://man7.org/linux/man-pages/man1/time.1.html

[11] Leighton, B., Cox, S. J., Car, N. J., Stenson, M. P., Vleeshouwer, J., & Hodge, J. (2015, March). A best of both worlds approach to complex, efficient, time series data delivery. In *International Symposium on Environmental Software Systems* (pp. 371-379). Springer, Cham.

[12] Mohror, K., & Karavanic, K. L. (2009, November). Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (p. 55). ACM.

[13] PostgreSQL. (2019). *PostgreSQL*. Retrieved from https://www.postgresql.org/

[14] Python. (2009). *Python v3.0.1 documentation*. Retrieved from https://docs.python.org/3.0/

[15] Python. (2019). The Python Profilers. Retrieved from https://docs.python.org/3.7/library/profile.html

[16] Reliability, scalability and performance – the impact of Intel HPC Orchestrator. (2017). *insideHPC*. Retrieved from https://insidehpc.com/2017/05/the-impact-of-intel-hpc-orchestrator

[17] Shahid, J. (2019). *InfluxDB Documentation*. Retrieved from https://media.readthedocs.org/pdf/influxdb-python/latest/influxdb-python.pdf

[18] Weber, M. (2016). Structural Performance Comparison of Parallel Software Applications. Retrieved from https://tu-dresden.de/ing/informatik/ressourcen/dateien/postgraduales/archiv/fc03badad325524c399c5287c7dc4415.pdf?lang=en