Enhanced State History Tree (eSHT) : a Stateful Data Structure for Analysis of Highly Parallel System Traces

Loïc Prieur-Drevon, Raphaël Beamonte, Naser Ezzati-Jivan, Michel R. Dagenais
Department of Computer Engineering
Ecole Polytechnique de Montréal
Montréal, QC, Canada
{loic.prieur-drevon, raphael.beamonte, n.ezzati, michel.dagenais}@polymtl.ca

***Abstract*—Behaviors of distributed systems with many cores and/or many threads are difficult to understand. This is why dynamic analysis tools such as tracers are useful to collect run-time data and help programmers debug and optimize complex programs. However, manual trace analysis on very large traces with billions of events can be a difficult problem which automated trace visualizers and analyzers aim to solve. Trace analysis and visualization software needs fast access to data which it cannot achieve by searching through the entire trace for every query. A number of solutions have adopted stateful analysis which rearranges events into a more query friendly structures after a single pass through the trace.**

In this paper, we look into current implementations and model the behavior of previous work, the State History Tree (SHT), on traces with many thread creation and deletion. This allows us to identify which properties of the SHT are responsible for inefficient disk usage and high memory consumption. We then propose a more efficient data structure, the enhanced State History Tree (eSHT), to store and query computed states, in order to limit disk usage and reduce the query time for any state. Next, we compare the use of SHT and eSHT on traces with many attributes. We finally verify the scalability of our new data structure according to trace size.

As shown by our results, the proposed solution makes near optimal use of disk space, reduces the algorithm's memory usage logarithmically for previously problematic cases, and speeds up queries on traces with many attributes by an order of magnitude. The proposed solution builds upon our previous work, enabling it to easily scale up to traces containing a million threads.

I. INTRODUCTION

Computer system tracing is one of many run-time analysis methods used by programmers to instrument systems and applications. The advent of multithreading, many-core systems and distributed systems creates a need for such tools to understand system behavior, but raises new challenges for trace storage, analysis and visualization.

Tracing is based on the generation of a chronological sequence of events, keeping track of what happened in the traced program. This information is usually stored in a trace file, called trace, but the format of such files is optimized for data storage. Two types of traces exist : event-based logging and state-based logging. In event-based logging, the tracer stores events, which are timestamped descriptions of system actions. In state-based logging, the tracer stores the state of the system using intervals. Intervals are tuples representing the state value of an attribute between two timestamps.

Many tools already exist to convert traces into a human readable format. `Trace Compass` [1] is one of these tools for viewing and analyzing traces with informative views and graphs. For interactive views, such as exploring process states, querying information directly from the trace file tends to be inefficient. Indeed, if we had to compute a given state, we may need to search the entire trace for the events responsible for the beginning and end of that state.

`Trace Compass`, as well as other visualization tools, needs a data structure that can store state information through time in order to avoid recomputing said states from the trace for every query. Because of the overwhelming size of traces, which can reach a terabyte, the data structure must be suited for external memory.

Moreover, because we are dealing with interactive programs and large data files, the data structure must scale well and be efficient for most queries for the user experience to remain satisfactory. To avoid recomputing states, Trace Compass uses the State History Tree (SHT) [2], a data structure which is efficient to query state information.

Our contribution is an enhanced data structure for stateful trace analysis which deals with some worst case scenarios, is scalable to the largest traces and allows multithreaded queries.

The remainder of the paper is organized as follows: after investigating this work's context and related research, we present the architecture of the data structure, and how it compares to other structures. Then, we explain the algorithms, queries used on the structure and also evaluate its performance compared to previous work. Finally, we conclude and outline possible future work.

II. RELATED WORK

The related work is divided into two relevant areas, visualization tools, and multidimensional data structures used by these tools.

A. Trace visualization

In this section, we compare open source trace visualizers that deal with stateful analysis and have a documented data structure to store this information.

`Jumpshot` [3] is the visualization component for the MPI Parallel Environment software package. It displays the nodes' states evolutions over time and the messages that they have exchanged. Jumpshot uses the `slog2` format to reduce the cost of accessing trace data. When using the MPE tracing framework for MPI, users have the option for a state based logging format, in which the tracer directly produces state intervals, as opposed to event based tracing, which produces a list of timestamped events.

`Paraprof` [4] uses tracing and profiling techniques to summarize information, allowing it to scale well to HPC applications. It stores data in a `CUBE` [4] data structure : which is stored as a 3D XML when on disk, and in memory as a double level map (the metric is the first level, the call path is the second level) of vectors (where index is the process number).

`Trace Compass` [1] is the extensible trace visualizer and analyzer for traces generated by the `LTTng` [5] tracer and other tracing tools. It is built using the Eclipse framework and uses State History Trees (SHT) to store state data in a query-efficient structure.

However, when working on huge traces, such software cannot afford to query directly on the trace itself as the query length could grow linearly with the trace size. This is why such programs transform traces into other data structures that are more efficient for querying. Most programs choose to store "stateful" data, i.e., one object per state [6]. For example, the state of the Attribute "thread/42/Status" could be "Sleep" between two specific timestamps.

B. Data structures

In this section, we compare the data structures used by the trace visualizers presented previously and generic data structures used for multi-dimensional data.

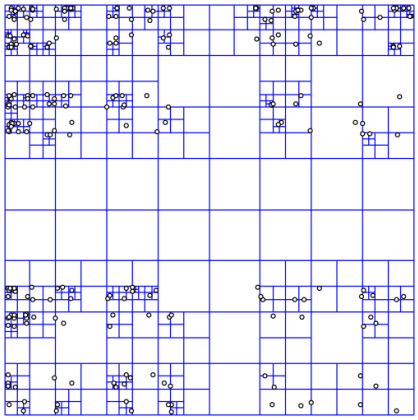


Figure 1: Representation of the `Quad-Tree` data structure [7]

Quad-Trees [8] are widely used in Geographical Information Systems, as their granularity can be adapted to the required resolution for a certain area. Quad-Tree nodes are defined by square bounds of the geographical data they contain. As shown in Figure 1, nodes can either be leaves or have 4 children which are bounded by 4 equal, square subquadrants. Because the quadrants are squares, quadtrees are well adapted for homogeneous data where the two dimensions both represent positions for example.

HV/VH-Trees [9] are used to store Integrated Circuit Layouts. Like other trees, the nodes are defined by the bounds of the data they contain. If a node contains more elements than a predefined threshold, it is split into 2 with either a vertical divider (V node) or a horizontal divider (H node). However, as a V (respectively H) node's children must be H (respectively V) nodes, the structure's flexibility for particularly skewed data is limited.

Multi-version B-Trees [10] store data items of the type $< key, t_{start}, t_{end}, pointer >$ where key is unique for every version and t_{start}, t_{end} are the version numbers for the item's lifespan. It has a number B-Tree root nodes that each stand for an interval of versions. Each operation (insertion or deletion) creates a new version. Versioning uses live blocks which duplicate the open intervals of the old block and have free space to store future values.

R-trees [11] are a type of tree used to store multidimensional data. They use "bounding boxes" to divide multidimensional spaces, grouping the closest objects together. Such bounding boxes become smaller as the depth in the tree increases, which allows queries to search only the relevant nodes. R-Trees can be used for spatio-temporal data by assigning the time to one dimension, though it is most efficient if intervals remain short.

Historical R-Trees (HR-Trees) [12] are another modification of R-Trees to support timestamps, by building an R-Tree for each timestamp and sharing common nodes with linkage between trees. This is more efficient when there are few modifications between a small number of timestamps.

The MV3R-Tree [13] combines an Historical R-Tree and a 3D R-Tree as one is suited to long intervals and the other to short intervals.

The `slog2` data structure [3] uses a tree structure where the root node is the length of the trace and each node is half the length of its parent. Intervals fit in the shortest node that can contain them. Nodes and state intervals cannot overlap, so the challenge is finding the right depth (leaf node length) to obtain a high fill ratio. Figure 2 shows a representation of that data structure.

The `State History Tree` (SHT) structure [2] was designed with trace analysis and visualization storage in mind. Like with slog2, the tree is built in a single pass through the trace. The SHT is also designed to perform well on rotating media, so each node is a multiple of the disk block size. The main difference with slog2 is that the depth of the tree and the length of the intervals are not predefined, so the build starts with a single node. Then,

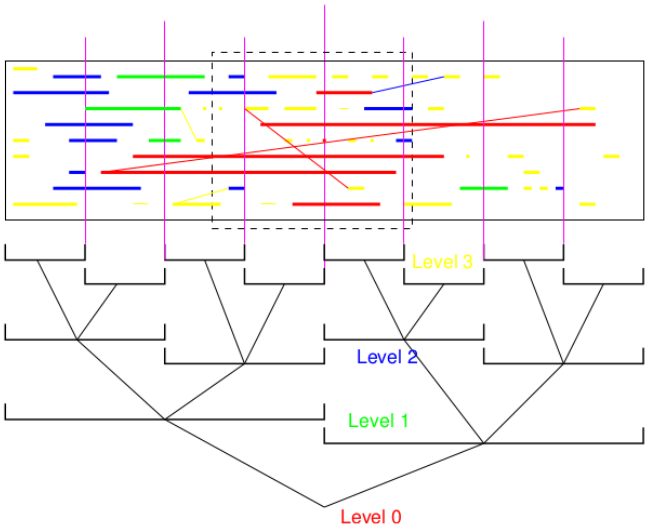


Figure 2: Representation of the `slog2` data structure [3]

siblings and parents are added as the nodes fill up. Sibling nodes are still consecutive, but their start and end times depend on the intervals that they contain.

Finally, multi-dimensional indexes have largely been studied in fields such as Geographical Information Systems, Integrated Circuit Layout, or even graphical displays. However, working on traces and processes' states is fairly different. Indeed, the information we have to store in the data structure is already sorted in time, and aimed to be processed in one pass. Such conditions allows for specific optimizations and tuning.

To our knowledge, there isn't currently any trace viewing software that implements an external memory structure which offers reliable performance, even for very large trace size and number of intervals. In this paper, we propose a scalable data structure, which has performance gains compared to previous implementations, is well suited to parallel and distributed systems, and offers good query times.

III. LIMITATIONS OF THE STATE HISTORY TREE

In this section, we briefly present the current implementation of the State History Tree (SHT), a data structure designed for state storage on external memory [14], and detail the issues that it encounters when dealing with highly parallel traces.

A. Structure of the State History Tree (SHT)

The State History Tree (SHT) [2] is suited for storage of stateful information that is computed while reading through the trace. It keeps the current states in memory with their start times. Every time an event changes a state, we write the ending state interval to external memory and update the current state value.

The SHT is used as a database to store state changes through time. The stored data takes the form of intervals, which consist of an attribute key, a start time, an end time and a value. The start time and end time are specified to the nanosecond level. The attribute key is a unique identifier for the object whose state we are tracking. The value is the payload of the interval which can be a null, a boolean, an integer, a long or a character string.

The SHT is composed of nodes created as the tree is built. A node is defined by a unique sequence number, a start time and an end time. They also contain the sequence numbers of their parent as well as the sequence numbers and start times of their children. Each node also has the ability to store intervals. The unique sequence numbers of the nodes represent the relative position of their block on disk.

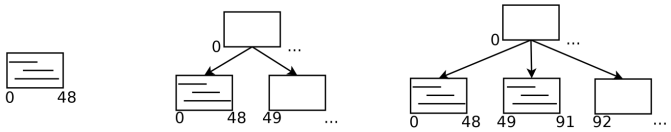


Figure 3: Build steps of the State History Tree using an incremental process [14]

The tree's construction starts from a single node, into which intervals arrive by increasing end times. Once a node is full, it is closed and written to disk as shown in Figure 3. If the node has a parent, we add a sibling node, else we add a new parent and sibling. When the tree becomes deeper, we fit intervals in the shallowest node possible in the current branch, knowing that the time ranges of children nodes are included in that of their parents.

B. Shortcomings on highly parallel trace analysis

Since we store intervals from the beginning for every attribute (such as threads), the State System has many intervals that begin at the SHT's start time. The SHT's design imposes that intervals that begin at the SHT's start time fit in the left most node. Therefore, a trace for a many-threaded program leads to a very deep tree. Indeed, the depth of the tree increases when we try to insert an interval beginning at the tree's start time and it cannot fit into the deeper non filled nodes, as its start time is earlier than theirs. Therefore, it can only fit in the root node, and when the root node is full, another depth level is added.

We have a very deep tree but most intervals fit in leaf nodes, therefore the branches are mostly empty from the left most node to the leaves. As the SHT implements nodes as disk blocks, many empty nodes means a very low disk usage rate (10-14%). As queries on an SHT search down a branch for which the nodes cover the queried time, and most intervals are stored in the leaf nodes, the average query takes a long time.

Because the tree is very deep and the State System stores its **in progress** branch in memory, the SHT con-

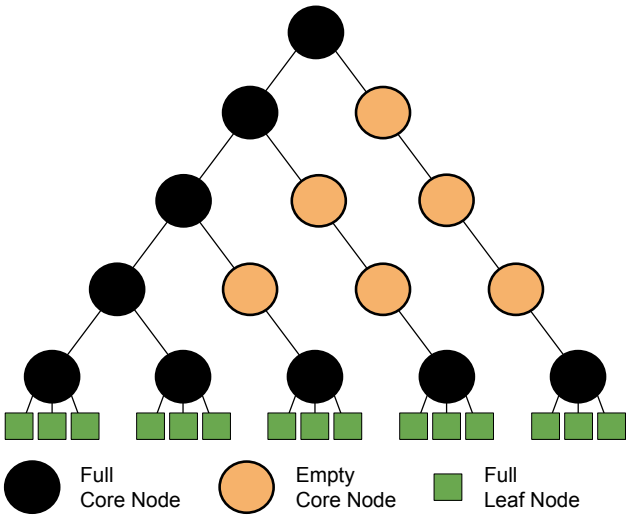
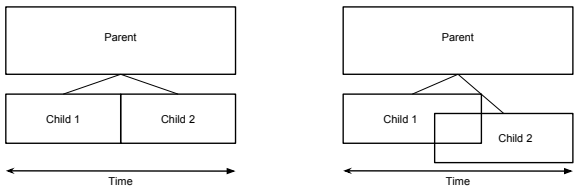


Figure 4: Schematization of a State History Tree for 10k threads

struction may even crash with a JVM OutOfMemoryError on the deepest trees.

IV. THE ENHANCED STATE HISTORY TREE



(a) Representation of consecutive nodes in SHT (b) Representation of overlapping nodes in eSHT

Figure 5: Comparison of the relations between sibling nodes of SHT (left) and eSHT (right)

A. Enhanced SHT structure

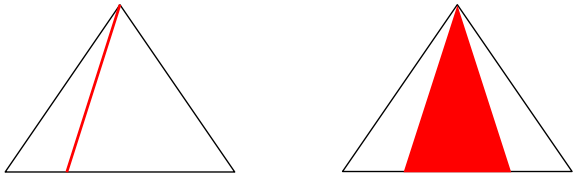
As we have seen previously, the consecutive node constraint is responsible for the tree degenerating into combs. Figure 5(a) shows this constraint on two children nodes. We remove this constraint and replace the new start time for sibling nodes with the start time of the first interval inserted after filling up the previous node. Removing this constraint ensures that incoming intervals will fit into the new leaf nodes, thus preventing the tree's depth from degenerating. However, the data structure and algorithms must be modified to account for this overlap as shown in Figure 5(b).

Because the nodes overlap, we are now querying on sub-trees instead of branches, as shown respectively in Figure 6(b) and 6(a). The query algorithm is also modified, as shown in Algorithm 1. To query the relevant sub-tree, we add children's time ranges to the parent node's header.

There are a number of benefits to allowing the nodes to overlap. We can now fit a large number of attributes that cover the same time ranges, which is typical for highly parallel trace analysis, without degenerating into a list. Therefore the tree is shallower, so queries are shorter and less memory is used. There are also far less empty nodes, so better use of the disk space is made, reducing the number of writes when building the tree (and consequently, build times).

As the data we are handling is multidimensional (time as well as attribute key), we also add attribute key bounds to the header. These represent the minimum and maximum attribute values for intervals in a node. We also store children's bounds in their parents' headers, to help narrow down the number of nodes searched during a query.

B. Query algorithms



(a) Representation of a branch query as used by SHT (b) Representation of a sub-tree query as used by eSHT

Figure 6: Comparison of queries between SHT (left) and eSHT (right)

Unlike for the SHT and slog2, queries on eSHTs cannot simply go down a branch in the tree, since there are potentially several children nodes that contain the relevant time. Therefore, each core node carries an index on the start and end times of each of its children to determine which sub-tree to explore: the method **node.getChildren(t)** produces a list of **node**'s children which contain time t in the following algorithm. The **node.getInterval(k, t)** method retrieves the interval on t with key k when it exists in **node**.

Moreover exploration on a sub-tree allows queries to be multithreaded by assigning a new thread to each children and sharing a queue of the children to be explored between threads. Whereas a path exploration must be sequential as the result exists only in one child.

C. Comparison of query bounds with SHT

The State History Tree aiming at query performance, the number of nodes searched per query is a good measure of the data structure efficiency. In order to prove that queries on our enhanced State History Tree perform

Algorithm 1 Multithreaded Single State Query

```
procedure SINGLEQUERY(key, time)
    interval ← null
    queue ← List(rootNode)
    while interval = null do                ▷ Parallel Section
        node ← queue.pop()
        if node.type() = CoreNode then
            queue.addAll(node.getChildren(t))
        interval ← node.getInterval(k, t)
    return interval
```

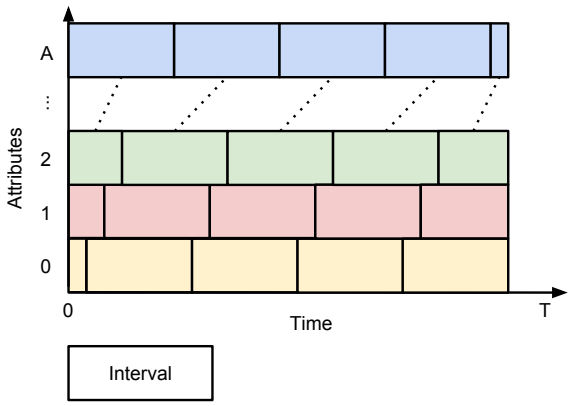


Figure 7: Schematization of the intervals in a node

better than on the original one, we thus will theoretically compare the number of nodes searched in each case.

The theoretical trace used for comparison is represented in Figure 7. That trace is of duration T and contains A different attributes. Each attribute is split into I equal intervals through the trace. Intervals from different attributes are offset by $\frac{T}{AI}$. Each node can store up to n attributes and have up to c children.

In the case of the SHT, the upper bound is the depth of the trace, as explained in IV-B. With our comparison trace, considering a large number A of attributes, we will reach the degenerating tree situation. The query bound Q_{SHT} for SHT can thus be formulated in this way:

$$Q_{SHT} = \left\lceil \frac{A}{n} \right\rceil$$

Where $\left\lceil \frac{A}{n} \right\rceil$ is the depth of the tree.

When it comes to the eSHT, we need to consider the depth of the tree, but also the number of overlapping nodes for each level at the queried time. The query bound Q_{eSHT} for eSHT can then be expressed as the following:

$$Q_{eSHT} \leq h + \left(\frac{A+n}{n+1}\right) \times \left(\frac{1-c^{-h}}{1-c^{-1}}\right)$$

With h being the depth of the eSHT. Considering that in our example, all the data is in the leaf nodes, we can use the standard formula [10] to compute the value of h:

$$h = log_c\left(\frac{AI}{n}\right)$$

The approximation of Q_{eSHT} is obtained by calculating the number of nodes needed to fit A intervals. Because of the build algorithm of eSHT and our worst case theoretical trace, all the intervals reside in leaf nodes. As intervals are inserted by increasing end times, the node duration D is

$$D = \frac{T}{I} + n \times \frac{T}{AI}$$

Where $\frac{T}{I}$ is the interval duration, neglecting border effects. The overlap Θ is then determined by how many nodes in the tree contain a time t, which is the ratio of node duration D over node offset Δt:

$$\Theta = \frac{D}{\Delta t}$$

Which can be developped as:

$$\Theta = \frac{\frac{T}{I} + n \times \frac{T}{AI}}{(n+1) \times \frac{T}{AI}}$$

And finally reduced to:

$$\Theta = \frac{n+A}{n+1}$$

However, we have to search the tree from the root node down to the leaves. We deduce the total number of core nodes from the maximum number of children c referenced by each parent. This relation can be expressed as follows:

$$Q_{eSHT} = \sum_{i=0}^{h} \left\lceil \frac{\Theta}{c^i} \right\rceil$$

Which allows to determine the limit on the maximum value of Q_{eSHT}:

$$Q_{eSHT} \leq \sum_{i=0}^{h} \left(\frac{\Theta}{c^i} + 1\right)$$

That we can then reduced to the following:

$$Q_{eSHT} \leq \Theta \times \frac{1-c^{-h}}{1-c^{-1}} + h$$

While this is slightly larger that the query on an SHT, when we compare the average query size, the average query on an eSHT is half that of the upper bound, as the intervals are uniformly spread over the possible DFS or BFS search path.

Meanwhile, on the SHT, we have to compute the average depth of nodes containing data. According to Figure 4, data is either in the left most nodes, in the deepest core nodes, or in the leaf nodes. If we consider:

- H as the height of the tree
- $N_{\text{leaf}} = c(H-1)$ as the number of **leaf** nodes, of depth H

- $N_{\text{core}} = (H-1)$ as the number of **core** nodes, of depth $H-1$
- $N_{\text{left}} = (H-2)$ as the number of **left** nodes, with increasing depths from 0 to $H-2-2$.

We can express the average depth of nodes as the following:

$$d_{\text{avg}} = \frac{\sum d}{\sum N}$$

Which can be developed as:

$$d_{\text{avg}} = \frac{\sum_{i \in \text{leaf}} d + \sum_{i \in \text{core}} d + \sum_{i \in \text{left}} d}{N_{\text{leaf}} + N_{\text{core}} + N_{\text{left}}}$$

And thus:

$$d_{\text{avg}} = \frac{\sum_{i=0}^{H-2} i + (H-1)^2 + c(H-1)H}{H-2+H-1+c(H-1)}$$

As we know that $H \gg 1$, the equation can finally be reduced to:

$$d_{\text{avg}} \simeq H$$

Therefore, the average eSHT query on problematic traces is faster than the average SHT query.

V. RESULTS

A. Test environment

All experiments were conducted on an Intel Core i5 6500 @ 3.2GHz with 8 GB RAM, a Samsung 850 EVO-Series 250GB Solid State Drive, using Eclipse version 4.5.1 and OpenJDK version 1.8.0_66. The trace files were generated using LTTng version 2.7.0 and the Linux kernel version 4.3.0. The trace files contain the detailed execution trace at kernel level, including all the system calls, scheduling events and interrupts. For the following benchmarks we traced the following program 2, which creates many threads, leading to many attributes in the State System.

Algorithm 2 Manythread Test Program

```
procedure runThreads
    phtread_t threads[NUM_CPUS];
    for i ← 0 to NUM_THREADS do
        for j ← 0 to NUM_CPUS do
            pthread_create(threads[j], BURN_CPU);
        for j ← 0 to NUM_CPUS do
            pthread_join(threads[j]);
```

The analysis behind our benchmarks is `Trace Compass`' kernel analysis. This analysis reads all the events from the trace and tracks different attributes by storing them in the SHT. Among those attributes, multiple information are stored for each thread and CPU of the traced system. This means that the bigger the trace is, and the more activity there was on the system during the trace, the bigger the generated SHT will be. This analysis will thus allow us to perform a thorough comparison between the SHT and eSHT metrics.

B. Tests cases

First, we consider the gains obtained on the pathological case of a trace with 10k threads, with 10 repetitions of the tree build process and a sample of single queries at 10 timestamps for 1000 attributes :

Table I: Gains of using eSHT instead of SHT for a 10k threads trace. Statistics on 10 executions.

	Gains of using		
	SHT	eSHT	Gain
Build (ms)			
min.	8 715	5 848	32.9%
max.	17 986	8 753	51.3%
avg.	14 156	6 991	50.6%
std. dev.	3 002	1 029	
Nodes	23 716	2 788	88.2%
Size (MiB)	1 486	178	88.0%
Node Fill	9%	95%	x10.5
Depth	206	7	96.6%
Query Complexity	84	41	51.2%

We see in Table I that the eSHT is in average twice as fast to build and query than the SHT, while disk size usage is divided by ten. The build time is the only thing that is, and should be, variable, as it depends on the system. On the other side, the other metrics presented in the Table are, and should be, constant. The tree built is always the same. We can also see that the eSHT tree is thirty times shallower than the SHT one. However, since we are querying a few subtrees, instead of a single branch, because the nodes overlap, this information is not as relevant as the rest.

The build time is mainly faster because the program has far less information to write to disk, and thus requires much fewer operations to grow the tree (namely, making it deeper and opening and closing empty nodes).

Single queries are faster, as seen in Table I. Despite querying on a sub-tree instead of a branch, the tree is much shallower and we have additional information on keys contained by the node to further narrow down the query.

The number of Nodes is correlated with the disk size and fill ratio. Since we almost only have full nodes, we do not waste much disk space.

C. Scalability

To evaluate the scalability of the solution, we consider a test bench which generates the intervals from Figure 7 and builds the associated data structures. We generate 20 intervals of equal duration per each attribute and consider the cases from one thousand to one million attributes.

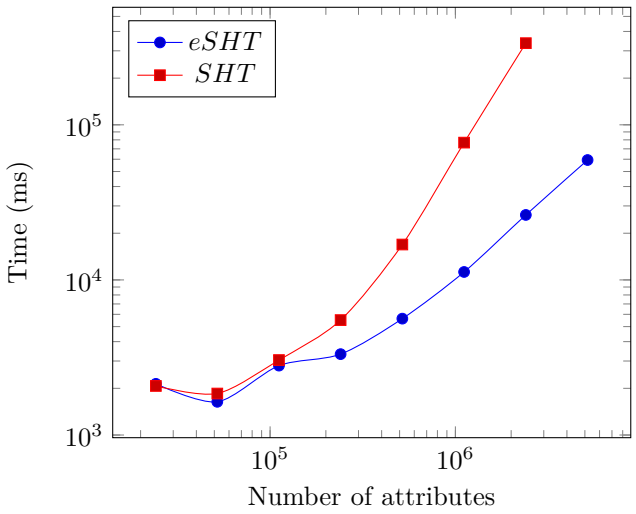


Figure 8: Comparison of SHT and eSHT build times for traces with many attributes. Each data point is the average of 5 executions.

1) Build Times: We look at how long it takes to build the full eSHT compared to the full SHT.

Figure 8 SHT takes longer to build, as the algorithm creates a number of empty nodes, which it has to write to disk. The difference between the SHT and eSHT is significantly bigger on a mechanical hard drive than on an SSD, as write speeds for state intervals are much slower than their generation rate by the CPU.

2) Size on Disk: We compare how much space is occupied on disk by an eSHT versus the SHT. Because of the very low node fill ratio for SHT, we cannot fit the structure on disk for the million threads case.

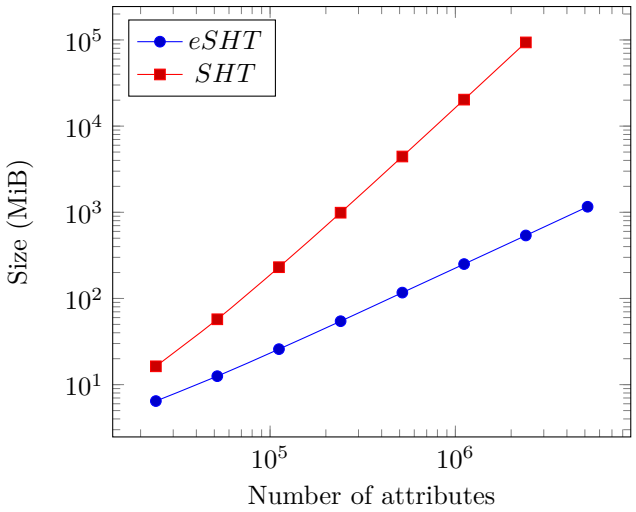


Figure 9: Comparison of the size of SHT and eSHT on disk for traces with many attributes. Each data point is the average of 5 executions.

As we can see in Figure 9, the size on disk is much reduced because we do not waste any space on empty nodes. Indeed, nodes are nearly 96% full for the eSHT, whereas the SHT uses the disk inefficiently, despite storing the same information.

3) Tree Depth: While tree depth is de-correlated from query complexity, it is nonetheless interesting as the algorithm stores the current branch in memory.

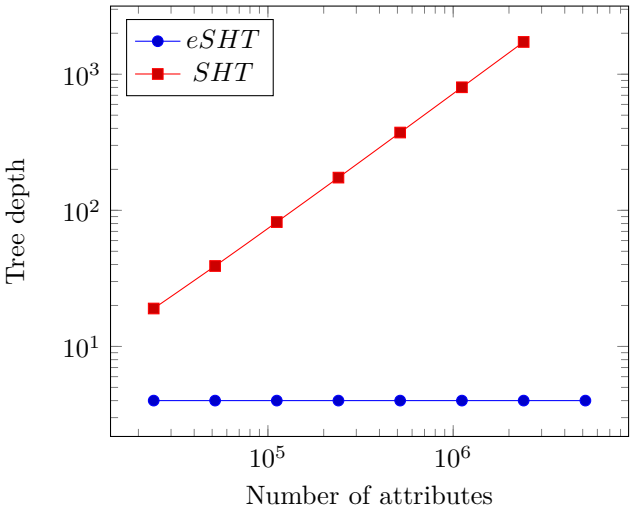


Figure 10: Comparison of SHT and eSHT depths for traces with many attributes. Each data point is the average of 5 executions.

A shallower tree means a smaller branch, and with each node occupying 8 KiB, Figure 10 shows there are substantial memory savings here.

4) Queries: Single queries search for the status of an attribute at a certain time. Full queries return the status of all attributes at desired time.

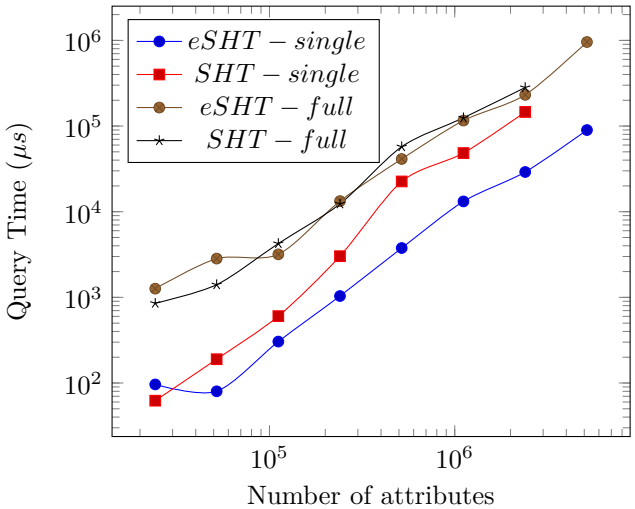


Figure 11: Comparison of SHT and eSHT query times for traces with many attributes. Each data point is the average of 5 executions.

We see in Figure 11 that for the same data, queries on the eSHT are more than an order of magnitude faster. Moreover, the results in Figure 11 match the theoretical results from section IV-C with eSHT single queries being twice as fast as those on SHTs.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the Enhanced State History Tree, an evolution of the State History Tree (SHT), that scales much better to highly parallel systems. By modeling the SHT and eSHT's behaviors, we have proved that the eSHT behaves significantly better than the SHT in queries, being at least twice as fast and featuring near-optimal disk usage. Experiments on highly parallel trace analysis also show that the new data structure behaves better in real scenarios.

We believe that the enhanced SHT structure paves the road for consistently faster (build and query times divided by 2, memory and disk usage divided by 10) analysis and visualization of traces from highly parallel systems. Moreover, said enhancements retain a good performance for use cases that were well suited to the SHT's behavior and already performed well.

Future work could include modifications to the build algorithm that will optimize the tree by minimizing overlap and reducing single query complexity with R-Tree properties. Other modifications to the Trace Compass Framework may insure that the optimizations on the underlying data structure result in visible speedups to the rest of the program.

ACKNOWLEDGMENT

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson and EfficiOS is gratefully acknowledged.

References

[1] M. Côté and M. R. Dagenais, "Problem Detection in Real-Time Systems by Trace Analysis," *Advances in Computer Engineering*, vol. 2016, 2016, article ID 9467181.

[2] A. Montplaisir-Goncalves, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "State history tree: An incremental disk-based data structure for very large interval data," in *Social Computing (SocialCom), 2013 International Conference on*, Sept 2013, pp. 716–724.

[3] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.

[4] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. Wylie, *Scalable Collation and Presentation of Call-Path Profile Data with CUBE*, ser. NIC series. Jülich: John von Neumann Institute for Computing, 2007, vol. 38, pp. 645–652, record converted from VDB: 12.11.2012. [Online]. Available: http://juser.fz-juelich.de/record/58173

[5] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[6] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Advances in Software Engineering*, vol. 2012, p. 6, 2012.

[7] M. T. G. David Eppstein and J. Z. Sun. (2005, Jun.) The Skip Quadtree: A Simple Dynamic Data Structure For Multidimensional Data. Presentation at the 21st ACM Symp. on Computational Geometry, Pisa, June 2005. [Online]. Available: http://www.ics.uci.edu/~eppstein/pubs/EppGooSun-SoCG-05.pdf

[8] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Inf.*, vol. 4, no. 1, pp. 1–9, Mar. 1974. [Online]. Available: http://dx.doi.org/10.1007/BF00288933

[9] G. G. Lai, D. Fussell, and D. F. Wong, "Hv/vh trees: A new spatial data structure for fast region queries," in *Design Automation, 1993. 30th Conference on*, June 1993, pp. 43–47.

[10] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *The VLDB Journal*, vol. 5, no. 4, pp. 264–275, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1007/s007780050028

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, vol. 14, no. 2. New York, NY, USA: ACM, Jun. 1984, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/602259.602266

[12] M. A. Nascimento and J. R. O. Silva, "Towards historical r-trees," in *Proceedings of the 1998 ACM Symposium on Applied Computing*, ser. SAC '98. New York, NY, USA: ACM, 1998, pp. 235–240. [Online]. Available: http://doi.acm.org/10.1145/330560.330692

[13] Y. Tao and D. Papadias, "The mv3r-tree: A spatio-temporal access method for timestamp and interval queries," in *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome*, 2001.

[14] A. Montplaisir, N. E. Jivan, F. Wininger, and M. Dagenais, "Efficient model to query and visualize the system states extracted from trace data," in *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, 2013, pp. 219–234.