The Journal of Systems and Software 135 (2018) 55–68

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

R-SHT: A state history tree with R-Tree properties for analysis and visualization of highly parallel system traces

L. Prieur-Drevon*, R. Beamonte, M.R. Dagenais**

Ecole Polytechnique de Montreal, Computer and Software Engineering Department, Canada

ARTICLE INFO

Article history:
Received 31 January 2017
Revised 30 July 2017
Accepted 22 September 2017
Available online 23 September 2017

Keywords:
Tracing
Stateful analysis
Tree
Data structure
External memory

ABSTRACT

Understanding the behaviour of distributed computer systems with many threads and resources is a challenging task. Dynamic analysis tools such as tracers have been developed to assist programmers in debugging and optimizing the performance of such systems. However, complex systems can generate huge traces, with billions of events, which are hard to analyze manually. Trace visualization and analysis programs aim to solve this problem. Such software needs fast access to data, which a linear search through the trace cannot provide. Several programs have resorted to stateful analysis to rearrange data into more query friendly structures.

In previous work, we suggested modifications to the State History Tree (SHT) data structure to correct its disk and memory usage. While the improved structure, eSHT, made near optimal disk usage and had reduced memory usage, we found that query performance, while twice as fast, exhibited scaling limitations.

In this paper, we proposed a new structure using R-Tree techniques to improve query performance. We explain the hybrid scheme and algorithms used to optimize the structure to model the expected behaviour. Finally, we benchmark the data structure on highly parallel traces and on a demanding trace visualization use case.

Our results show that the hybrid R-SHT structure retains the eSHT's optimal disk usage properties while providing several orders of magnitude speed up to queries on highly parallel traces.

1. Introduction

Understanding the runtime behavior of complex computer systems is a daunting task. Tracing is one of many runtime analysis methods used to instrument and collect data on systems and applications. Compared to logging, tracers have much lower overhead and can produce hundreds of thousands of events per second, at nanosecond precision, providing extremely detailed information on kernel, process and hardware states.

Tracers produce trace files, a series of chronological events, which are optimized for low overhead and data storage but challenging for human operators to understand. A number of software solutions, called trace visualizers, have been developed to facilitate the understanding of these files by providing graphical visualizations, statistics and detailed analysis of certain use cases. These programs perform stateful analysis, that transform event-based data structures into state-based structures, and reorganize data from lists to trees, for faster access.

Indeed, when traces reach gigabyte or terabyte size, efficient data structures are important for maintaining sustainable performance levels for analysis, and low latency for interactive visualizations. Said data structures must be able to scale horizontally – for tracing programs over a large duration – as well as vertically – for tracing systems with many processors, threads and resources.

Among the existing data structures, some are optimized for disk storage, build time or perhaps query performance. When working on trace visualization, the latter is fairly important. R-Trees are a family of data structures used to index multi-dimensional data sets and offer excellent query performance.

In previous work Prieur-Drevon et al. (2016), we presented a self-defined tree structure, optimized for external memory storage and with satisfactory query performance. However, we found that query performance scaled linearly to the number of components in the system, which led to slowdowns for the analysis of systems with many threads for example.

In this paper, we propose an enhanced, configurable build algorithm that reorganizes data in the sub-trees so that they reflect properties of an efficient R-Tree. The types of queries we optimize

* Corresponding author.
** Corresponding author.
E-mail addresses: loic.prieur-drevon@polymtl.ca (L. Prieur-Drevon), raphael.beamonte@polymtl.ca (R. Beamonte), michel.dagenais@polymtl.ca (M.R. Dagenais).

https://doi.org/10.1016/j.jss.2017.09.023
0164-1212/

for are at least two orders of magnitude faster on R-SHTs with a million time series than the equivalent eSHT. These gains result in a minimum 7 times speedup for the visualization of large traces.

This paper is organized as follows. First we cover related research on trace visualizers and underlying data structures in Section 2. Then we present the architecture of the current data structure in Section 3 as well as that of the evolutions we suggest in Sections 4 and 5. In Section 6, we model the behavior of the query algorithms before benchmarking them on real-life traces in Section 7. Finally, we conclude and suggest future work.

2. Related work

2.1. Trace visualizers

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` (Chan et al., 2008) 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. However, Jumpshot is focused on MPI visualization and doesn't provide detailed analysis capabilities.

The `TAU Performance System` (Shende and Malony, 2006) is a set of Tuning and Analysis Utilities to collect data from function, method, block and statement instrumentation as well as event-based sampling. Its visualizer, `Paraprof` (Bell et al., 2003) aggregates and stores data in a `CUBE` (Geimer et al., 2007) data structure, which is based around a Cube data model, with one dimension for metrics, another for programs and a third dimension for the system. When in memory, Paraprof stores its data as a double level map of vectors, keyed by the metric, then the call path and finally the process number. Despite all its capabilities, Paraprof does not provide stateful information on the systems' performance, rather focusing on metrics.

`Aftermath` (Pop and Cohen, 2013) provides visualization and analysis for traces from task-parallel work-flows. As part of the OpenStream project, it relies heavily on aggregation of trace points from the application as well as the runtime, and performance counters. Its creators state that the software can scale up to traces of several gigabytes in size while remaining fast thanks to the use of augmented interval trees as a backend. However, Aftermath stores the entire trace in memory, thus limiting its scalability.

Google has built tracing into `Chromium` (Google, 2013) to help developers identify slowdowns originating from either JavaScript, C++, or other bottlenecks. The visualizer easily scales to the number of threads used by chrome and the flame-graphs of some deep call stacks. Withal, Chromium Tracing is obviously restricted to analyzing Chrome's performance, yet shows the appeal of tracing and analysis for diversified applications.

`Pajé ViTE` (Coulomb et al., 2012) is developed for Pajé or OTF traces from parallel or distributed applications. It can scale to display millions of events per view and large computing clusters by storing trace events in a balanced binary tree, which is however limited by the size of the main memory.

`Trace Compass` (Côté and Dagenais, 2016) is the extensible trace visualizer and analyzer for traces generated by the `LTTng` (Desnoyers and Dagenais, 2006) 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. It supports a number of different trace formats and offers comprehensive analysis modules. Because of its flexibility, it is equally effective for analysing real-time programs running on a single system, as it is with multi-threading, DSP and GPU architectures, and distributed or virtualized systems.

Distributed systems, which rely on the MPI standard also have a number of dedicated tools to analyse their specificities.

`HPCTraceviewer` (Adhianto et al., 2008) is the visualization component in the HPCToolkit. It is used for performance measurement and analysis on large supercomputers. By relying on a client/server architecture, it avoids moving gigabytes of trace files and benefits from the computing power and memory of MPI nodes to process raw data.

The `VampirTrace` (Müller et al., 2007) visualizer relies on a client/server architecture with parallel servers to scale up for reading large distributed traces. The nodes interact via standard MPI primitives and precompute the required information before sending the results over to the client.

`ScalaTrace` (Noeth et al., 2009) relies on local and global compression to reduce the sizes of MPI traces dramaticaly and preprocess trace comparison. This results in constant size or sublinear growth sizes compared to the number of nodes.

However, when working on huge traces, the aforestated software cannot afford to query directly 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 (Ezzati-Jivan and Dagenais, 2012). For example, the state of the Attribute "thread/42/Status" could be "Sleep" between two specific time-stamps.

2.2. Stateful data structures

In this section, we compare the data structures used by aforementioned trace visualizers and generic data structures used for multidimensional data. The following structures focus on query performance.

`B-Trees` (Comer, 1979) were one of the first index structures developed to accelerate accesses to external memory data structures. B-Trees extend binary search trees by giving each node between d and $2d$ keys as well as $d+1$ to $2d+1$ pointers to children nodes, in which case the tree is of `order d`. All the values in the sub-tree referenced by the ith pointer are larger than the ith key and smaller than the $(i+1)$th.

`Multi-version B-Trees` (Becker et al., 1996) 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 of 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.

`Interval Trees` (Cormen, 2009) are tree structures designed to efficiently find time intervals that overlap a certain timestamp. Different implementations of interval trees exist in the literature. `Aftermath` for example, uses `Augmented Interval Trees` (Har-Peled, 2011) which are based on ordered tree structures. These are typically binary trees or self-balancing binary search trees, where the interval start time is used for ordering. Each node is "augmented" with the latest end time of the associated sub-tree. Knowing the end times of the sub-tree tells the algorithms which nodes they can skip when searching for intervals. The `Centered Interval Tree` implementation is similar to a binary search tree, with each node using a time t as a key such that all the intervals in the left node end before t, all the intervals in the right node start after t, and the node contains all intervals overlapping t. The tree is balanced when the left and