Checkpoint/Restart for Large-Scale Attack and Compliance Graphs

Noah L. Schrick
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
noah-schrick@utulsa.edu

Peter J. Hawrylak
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
peter-hawrylak@utulsa.edu

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—Attack Graph; Compliance Graph; MPI; High-Performance Computing; Checkpoint/Restart;**

I. Introduction

II. Related Work

III. Implementing C/R

A. Introduction to Intermediate Database Storage

Chapter 2 and the author of [**?**] discuss the challenges of attack graph generation in regards to its scalability. Specifically, the author of [**?**] displays results from generations based on 11 assets and 11 exploits that lead to 14,200 total states. Generating an attack or compliance graph based on a large network with a multitude of assets and involving a more thorough exploit or compliance violation checking will prevent the entire graph from being stored in memory as originally designed. This Section discusses the challenges of graph generation in regards to memory, and a solution through the implementation of intermediate database storage.

B. Memory Constraint Difficulties

Previous works with RAGE have been designed around maximizing performance to limit the longer runtime caused by the state space explosion, such as the works seen by the authors of [**?**], [**?**], and [**?**]. To this end, the output graph is contained in memory during the generation process to minimize disk writing and reading. This also allows for leveraging the performance benefits of memory operations, since graph computation relies less on processor speed and more on data dependency complexity, parallelism coarseness, and memory access time [**?**], [**?**], [**?**]. The author of [**?**] does incorporate PostgreSQL as an initial and final storage mechanism to write the starting and resulting graph information, but no intermediate storage is otherwise conducted.

While the design decision to not use intermediate storage maximizes performance for graph generation, it introduces a few complications. When generating large graphs, the system runs the risk of running out of memory. This typically does not occur when generation is conducted on small graphs, and is especially true when relatively small graphs are generated on an HPC system with substantial amounts of memory. However, when running on local systems or when the graph is large, memory can quickly be depleted due to state space explosion. The memory depletion is due to two primary memory consumption points: the frontier which contains all of the states that still need to be explored, and the graph instance which holds all of the states and their information, as well as all of the edges.

The frontier quickly becomes a problem point with large graphs that have a large height value, and contain many layers before reaching leaf nodes. During the generation process, RAGE works on a Breadth-First Search approach, and new states are continuously discovered each time a state from the frontier is explored. In almost all cases, this means that for every state that is removed from the frontier, several more are added, leading to an ever-growing frontier that can not be adequately reduced for large networks. Simultaneously, the graph instance continues to grow as states are explored. When the network contains numerous assets, each with their own large sets of qualities, the size of each state becomes noticeably larger. With some graphs containing millions of nodes and billions of edges like those mentioned by the authors of [**?**], it becomes increasingly unlikely that the graph can be fully contained within system memory.

C. Maximizing Performance with Intermediate Database Storage

Rather than a static implementation of storing to the database on disk at a set interval or a set size, the goal was to dynamically store to the database only when necessary. This would allow for proper utilization of systems with greater memory, and would reduce fine-tuning of a maximum size variable before database writes on different systems. Since there is an associated cost with preparing the writes to disk, the communication cost across nodes, the writing to disk itself, and a cost for retrieving items from disk, it is desirable to store as much in memory for as long as possible and only write when necessary. When running RAGE, a new argument can be passed *(-a <double>)* to specify the amount of memory the tool should use before writing to disk. This argument is a

value between 0 and 0.99 to specify a percentage. This double is immediately reduced by 10%. For instance, if 0.6 is passed, it is immediately reduced to 0.5. This acts as a buffer for PostgreSQL. Since queries will consume a variable amount of memory through parsing or preparation, an additional 10% is saved as a precaution. This can be changed later as needed or desired for future optimizations. Specific to the graph data, the statement is made that the frontier is allowed to consume half of the allocated memory, and that the graph instance is allowed to consume the other half.

To decide when to store to the database instead of memory, two separate checks are made. The first check is for the frontier. If the size of the frontier consumes equal to or more than the allowed allocated memory, then all new states are stored into a new table in the database called "unexplored states". Each new state from this point forward is stored in the table, regardless of if room is freed in the frontier. This is to ensure proper ordering of the FIFO queue. The only time new states are stored directly into the frontier is when the unexplored states table is empty. Once the frontier has been completely emptied, new states are then pulled from the database into the frontier. To pull from the database, the parent loop for the generator process has been altered. Instead of a while loop for when the frontier is not empty, it has been adjusted to when the frontier is not empty or the unexplored states table is not empty. Due to C++ using short-circuit evaluation where the first argument is completely evaluated before processing the second, some performance is gained. The performance gained is due to not having to pass a SQL statement to disk to check the size of the unexplored states table unless the frontier is empty. The original generation design stored new states into the frontier during the critical section to avoid testing on already-explored states. To follow this design decision, writing new states to the database is also performed during the critical section.

For the graph instance, a check in the critical section determines if the size of the graph instance consumes more than its allocated share of the memory. If it does, the edges, network states, and network state items are written to the database, and are then removed from memory.

However, a new issue arose with database storage. The original design was to save staging, preparation, and communication cost by writing all the data in one query (as in, writing all of the network states in one query, all the network state items in one query, and all the edges in one query). While this was best option in terms of performance, it was also not feasible when the amount of data to store was large in relation to system memory. Building the SQL queries themselves quickly began depleting the already constrained memory with large storage requests. As a result, the storage process would consume too much memory and crash the generator tool. To combat this, all queries had to be broken up into multiple queries. As previously mentioned, an extra 10% buffer was saved for the storage process. SQL query strings are now built until they consume the 10% buffer, where they are then processed by PostgreSQL, cleared, and the query building process resumes.

D. Portability

The intermediate database storage is greatly advantageous in increasing the portability of RAGE across various systems, while still allowing for performance benefits. By allowing for a user-defined argument, users can safely assign a value that allows for other processes and for the host OS to continue their workloads. While the "total memory" component currently utilizes the Linux *sysconf()* function, this is not rigid and is easily adjustable. When working on an HPC cluster, using this function could lead to difficulties since multiple users may be working on the same nodes, which prevents RAGE from fully using all system memory. This could be prevented by using a job scheduler argument such as Slurm's "–exclusive" option, but this may not be desirable. Instead, a user could pass in the amount of total memory to use (and can be reused from a job scheduler's memory allocation request option), and the intermediate database storage process would function in the same fashion.

E. Implementation Approach

SCR: Adam Moody, Greg Bronevetsky, Kathryn Mohror, Bronis R. de Supinski, Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, LLNL-CONF-427742, Supercomputing 2010, New Orleans, LA, November 2010.

F. Implementation Challenges

IV. Results

V. Conclusion

Acknowledgment

References

References