Distributed eXplode: A High-Performance Model Checking Engine to Scale Up State-Space Coverage

Nageswar Keetha, Leon Wu, Gail Kaiser, Junfeng Yang
Department of Computer Science, Columbia University, New York NY 10027
nk2340@columbia.edu, {leon, kaiser, junfeng}@cs.columbia.edu

Abstract

Model checking the state space (all possible behaviors) of software systems is a promising technique for verification and validation. Bugs such as security vulnerabilities, file storage issues, deadlocks and data races can occur anywhere in the state space and are often triggered by corner cases; therefore, it becomes important to explore and model check all runtime choices. However, large and complex software systems generate huge numbers of behaviors leading to 'state explosion'. eXplode is a lightweight, deterministic and depth-bound model checker that explores all dynamic choices at runtime. Given an application-specific test-harness, eXplode performs state search in a serialized fashion - which limits its scalability and performance. This paper proposes a distributed eXplode engine that uses multiple host machines concurrently in order to achieve more state space coverage in less time, and is very helpful to scale up the software verification and validation effort. Test results show that Distributed eXplode runs several times faster and covers more state space than the standalone eXplode.

1. Introduction

Model checking medium to large programs by taking the code as the model is challenging because of exponential growth in dynamic states[4, 5], which quickly depletes computing resources. Even though it is practically impossible for model checkers to fully explore the states of large programs within available resources of memory and CPU time, several heuristics in reachability analysis are proposed to confront the state-explosion problem [3, 4, 5, 10]. While these tools can get good coverage on selected applications, it is still an open question whether complete state coverage can be achieved consistently. Hence, improving performance of model checkers by reducing memory requirement and employing multiple processors is important and is an active research topic. In their seminal work Stern and Dill [15] reported on parallelizing murphi verifier, utilizing distributed memory and multiprocessors on reachable state-space partitions. Their work is the basis for all other techniques in the distributed explicit state model checking literature, e.g., [16, 17, 20].

eXplode[1] runs in a single-thread of execution exploring one state at a time with one instance of eXplode per one application's state space; hence, it doesn't scale up to large programs. To reduce memory, eXplode takes a light weight snapshot of the state consisting of state's signature (a hash compaction of an actual state), the trace (the sequence of return values from its path decision function). To restore the state, it replays the sequence of choices from the initial state, however, reconstructing states is a slow and CPU intensive process, especially when traces are deeper.

By designing an engine to reduce runtime using parallel processing, we propose a fast performing distributed eXplode that supports multiple eXplode instances in parallel, each instance exploring unvisited states or subset of the generated state-space. Distributed eXplode has the following advantages 1) We can employ several hosts on demand to reconstruct and clone the states from their traces concurrently and explore them on different hosts, 2) Checkpoint of an actual application state is also distributable around other hosts, in addition, it paves a way to distribute high overhead checkpoints as live OS processes using thin virtualization systems[12], 3) In addition, it facilitates the use of distributed hash tables[7] treating the light weight states as network objects to achieve fair load balancing when hosts join and leave on the fly.

The rest of the paper is organized as follows. Section II provides an overview on eXplode, section III Provides detailed description of proposed solution and its implementation. Section IV Provides the feasibility

evaluation on an example, Section V provides related work and section VI provides future work and concludes.

2. eXplode Overview

Let a system model M be a state transition graph (typically a Kripke Structure) on environment E, then given a property P, the model checking problem is to verify if M in E satisfies P.

eXplode is easier to setup and verifies real programs by performing stateful search. It treats code as a transition system and provides a choose(N) operation as shown in figure 1, a serialized simulation of a N-way fork, that allows the model checker to fork at every decision point during the exploration of every possible operation. Users can code a lightweight test harness in which definition of guarded transitions are provided.

eXplode can perform more invasive white box checks if we have access to source code by instrumenting the code without modifying it, if no source code is available then it can attach to live applications at runtime through the test harness and perform black box tests. It attempts to explore as many behaviors as possible by focusing on precision and determinism, if the tool reports an error property, then it is a real error and can't be a false positive. Once an error is found, it reports/logs the trace leading to the error.

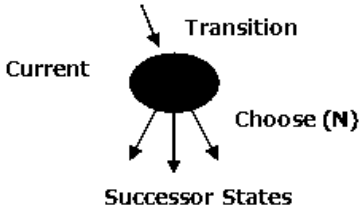

Figure 1

3. Proposed Solution Architecture

Tools like VeriSoft [4], CHESS [10] employ stateless techniques and have low in-memory overhead but demand more CPU time. Whereas, stateful model checkers like eXplode [1], FiSc [2], CMC [6], Java Path Finder(JPF) have high in-memory overhead as states have to be check pointed. eXplode defines a lightweight state S consisting of {signature, trace},where a signature is an hash compaction of current snapshot of state data and is a unique fixed size bit string obtained by MD4/MD5 hashing, capturing the signature can be overridden in the test harness by the user if needed. Trace is the transition sequence consisting of returned values from eXplode's Choose (N) at every decision point. Hence, by using this lightweight state, we can distribute it with less communication overhead and re-compute the actual state to its clone from the trace of choices made when the original state was constructed. This is expensive, however, we can reconstruct large states which otherwise would be difficult to be sent across network in original form. In other techniques[15], whole data of state(even if the state is large) needs to be sent across the network as expanding states from (hash) signature is not possible anyway, however, in distributed eXplode due to the availability of state's trace it is possible to reconstruct the state from the trace and handle large states as well.

Optionally, explode takes checkpoints of actual state data as well. A checkpoint CP consists of {signature, data}; where data = {v1, v2, vn} is an instance of actual state variables and signature is the hash value of data digest. Hence, a state can be represented either as an actual checkpoint or as a light weight object. Hashing each state is also expensive; however, this effort is also implicitly distributed.

Distributed eXplode is developed on Linux as well as on Windows. On windows we have implemented it using Microsoft Messaging Queuing (MSMQ) /COM+ application server. Each host maintains a local queue which is publicly visible to other hosts. Seen-set is deployed as a COM+ process on each host. When a host picks a new state, if it's not seen, then it will update its seen set and processes the state by running all reachable transitions defined in the test harness from that state, if transitions run with no bugs and generate new reachable states, it will assign the states among participating hosts based on a hash function and forward the states to respective hosts. To trigger the state space generation, a designated master host captures the initial state and sends it to its hash mapped host.

If an eXplode instance finds violations or bugs while exploring transitions from the current state it would place its trace in a log and either continues to explore other states in the queue until preset maximum number of bugs (violations) are found or its depth-bound is reached. The generated workload on a host is a function of exploration time, network overhead, and state partitioning techniques. Hence, workload balancing is desired among participating nodes but that needs the knowledge of the state space which is the very problem we're trying to solve. However, there are several techniques which can be employed such as caching, dynamic partitioning functions to reduce network overhead and achieve fair distributions [16].

In addition, distributed eXplode can be integrated with chord DHT[7] for state distribution to achieve fair load balancing as the state is a lightweight object in eXplode that can be treated like a low overhead network object.

The proposed conceptual architecture of distributed eXplode is shown in Figure 2. Each host has a local state queue, seen-set (hash-map/distributed hash map), and a local service to manage eXplode instances.

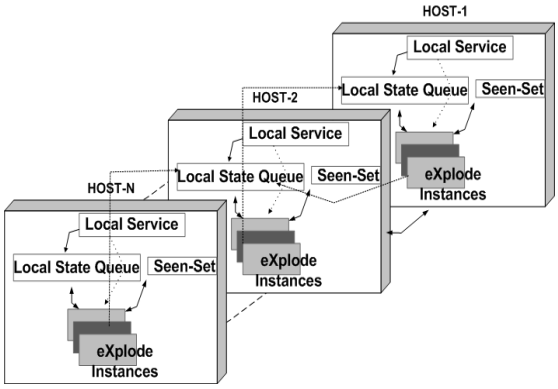

Figure 2

We have implemented a lightweight and independent local service, shown in figure 2, installed on each host for the following reasons 1) to manage eXplode clients based on the generated workload in local queues and the underlying resources available on the host so that thrashing on the system can be avoided by limiting maximum number of clients per host. This service senses the workload and resizes number of eXplode instances on the host, this is useful because if host has multiple processors with shared memory then service can scale-up model checking instances on the host. The seen-set is maintained per host, so if we increase the parallel instances the lock on local seen set reduces the scalability on the host to some degree, 2) Model checking instances can be automatically restarted gracefully by the local service after every time a bug is found or their preconfigured time span expires by doing so the new instances are clean and reliable with no resource leak issues and can start exploring new states in an incremental fashion. In other words, instantiating the processes on the fly or instructing the model checking processes to live only for a particular period of time improves the reliability.

Distributed eXplode has an option to checkpoint the states and distribute the original state data via messaging. A checkpoint is a high overhead object, so currently we have implemented in-memory data of variables to be check pointed, not the state of the environment (such as opened files or connection sessions). User can choose whether a state can be check-pointed or not in the test-harness.

State space of a program is equivalent to a Graph that captures all possible behaviors whether it's generated by one instance or several instances of the program. So the power of proposed distributed engine can be exploited fully by attaching an image of application to each instance of eXplode to model check subsets of the targeted state space. Model checking centralized applications service in a black box approach may not scale up if the service itself is the bottleneck and is not scalable. In that case, we can install a copy of service on each host and test them on pre-production scenarios. If centralized services under verification are scalable, then distributed eXplode performs better when checking the applications on production environment

In Distributed eXplode only the model checking effort is concurrent, it can't test multithreaded applications with heavy global data inter-leavings. However, by building the Lamport's [9] happens before graph, we can convert multi-threaded application into an inter-leavings graph which can be searched serially, and then we can apply distributed eXplode to model check this graph in parallel to scale up checking the multi threaded applications as well. However, as is eXplode's model checking engine doesn't address multi threaded applications directly.

4. Feasibility and Evaluation

In this study we have used a simple example to prove the feasibility of the distributed eXplode. As shown in figure 3, let's say x and y be integer variables and each can take values up to a MAX number. By creating a transition as shown in the code below in the test harness, using choose (2) that returns random values either 0 or 1. This transition increments x if the random choice is 0 else increments y if choice is 1, then on every choice made it calls a test function which dependent on x, y. Test function has some memory related operations such as malloc and memory checks based on values of x and y, and consumes Memory and CPU cycles to simulate the test. If a choice of x and y produces error then explode reports the error. By just modifying MAX we can change the size of state space in order to test the performance of standalone eXplode vs. Distributed eXplode with several instances.

```
void run_one_transition(void)
{
        int op = choose(2);
```

```
        switch (op) {
        case 0:
                x += 1;
                x = x% MAX;
                break;
        case 1:
                y += 1;
                y =y% MAX;
                break;
        }
        if(!RunMemoryFunctions(x,y )
        {
                LogErrorTrace;
        }
}
```

Figure 3

We have used three Dell PowerEdge 2650 servers each with two Intel® Xeon™ processors at 2.4GHz, 4GB DDR SDRAM running windows 2000 server. There are 6 parallel processors in total in this configuration.

We ran one instance of eXplode on 20000 states for the transition in Figure 3 and the results are shown in the Figure 4. If number of states increase, the performance degrades for two reasons 1) the seen-set size increases 2) State trace depth increases.

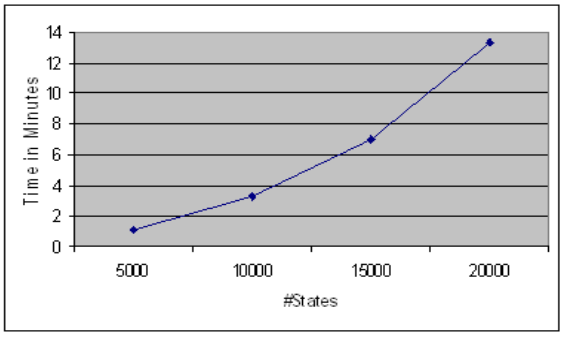

Figure 4: #Processed States vs. Time taken by one instance

We ran multiple instances up to 6 instances as there 6 CPUs and obtained the results in Figure 5. Performance is improved several times. If there are 2 instances in parallel each processed roughly 10000 states in 3 minutes 43 seconds and is consistent with Figure 4 where one instance took 3 minutes 28 seconds to process 10000 states.

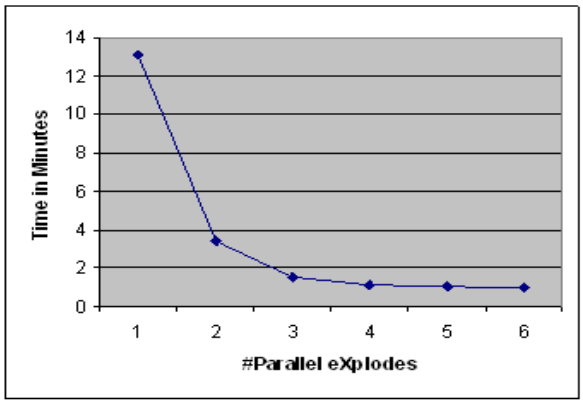

Figure 5: Searching time with #instances

Explosive population of states would still challenge the system, as we'll be limited by maximum number of clients we can instantiate dynamically at some point in the testing process. However, eXplode does depth bound search so we can manage the state space explosion to some extent, by limiting the depth, number of states per process, or total time to explore.

5. Related Work

Software Model Checking and Reachability Analysis: Model checkers [1, 2, 5, 9] are used to find errors in software systems code. VeriSoft [4], CHESS [10] employ stateless techniques require low in-memory where as SPIN[5], eXplode [1], FiSc [2], CMC [6], Java Path Finder(JPF) are explicit state model checkers and have high in-memory overhead and some of these are more concerned with solving the reachability, depth or context bounding and state reduction techniques. eXplode[1] in particular is a lightweight and generalized model checker because it has reduced memory requirement by defining a lightweight state, hence these objects can be distributed and load balanced with low communication overhead. In addition, eXplode checks user space applications and can be easily ported to several environments. eXplode runs in a single thread of execution and its N-Way decision fork is actually a serialized execution where it explores one state at a time, hence would not scale well for very large systems. However, the version proposed in this paper would improve the performance of model checking via parallelized eXplode.

Parallelized Software Model Checking: This category attacks the state space with distributed memory and multiprocessors via available parallelism.

In their work Stern and Dill [15] reported on parallelizing murphi verifier to check protocols, distributing reachable state-space partitions on parallel processors. Their work is the basis for all other techniques in the distributed explicit state model checking research, e.g., [16, 17, 20]. Distributing the actual checkpoints as large states (in terms of several MBs in size) of user space applications is still a challenge due to communication overhead involved. However, in distributed eXplode, we can distribute the workload by creating traces and reconstructing the states maintaining low communication overhead. Hence, states of any size can even be distributed over HTTP on the Internet. As search time increases, so is the size of the local seen-set and locking and updating the seen-sets limits the scalability to some degree if several eXplode instances are run per host. The technique proposed in [20] is implemented using JPF which avoids the lock on seen-set but is randomized possibly leading to redundant work.

6. Conclusion and Future Work

The main advantage of model checking is that we can capture system's behavior at any point in time as a 'State' then try to search the whole state space to hit interesting states(possibly with deviating properties as bugs). If we're lucky to exhaust the state space then we verify the system and find issues if exist, if not, we can check suboptimal state space, by bounding the search. Further, by Distributed eXplode presented in this paper, we have attempted to scale up the performance to several folds. Distributed eXplode can not only work for bug identification for user space applications but also can be used for design verifications, protocol verifications and module level contract verifications.

Checkpointing a live process as a state is a daunting task because of environment issues and high overhead involved in migration, and distribution; so yet another interesting direction for our work is to incorporate process virtualization techniques such as live process check pointing and migration via low overhead thin virtualization techniques [12]. We have integrated the local seen-set with OpenDHT [18] service that works for feasibility study with no lock on seen-set, however, we plan to study and evaluate the performance with local installation of Chord [7]. We plan to incorporate DHT techniques to load balance, to avoid lock on the local seen-set, to avoid idling, and to handle leaving and joining hosts gracefully.

7. Acknowledgments

The authors thank Jason Nieh for his assistance. Keetha, Wu, and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH 1 U54 CA121852-01A1. Yang is a member of the Reliable Computer Systems Laboratory.

7. References

[1] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI), Seattle, CA, November 2006.

[2] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. ACMTransactions on Computer Systems (TOCS), 24(4):393–423, 2006.

[3] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS 1032. Springer-Verlag, 1996.

[4] Patrice Godefroid. Model checking for programming languages using Verisoft. In POPL 97: Principles of Programming Languages, pages 174–186. ACM Press, 1997.

[5] G. J. Holzmann. The model checker spin. IEEE Trans. Softw. Eng., 23(5):279–295, 1997

[6] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, David L. Dill. CMC: A pragmatic approach to model checking real code, Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, December 2002

[7] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan., Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, In the Proceedings of ACM SIGCOMM 2001, San Deigo, CA, August 2001

[8] E. Clarke, O. Grumberg, and D. Peled. Model Checking.MIT Press, 1999.

[9] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 7 (1978), 558– 565.

[10] Madanlal Musuvathi, Shaz Qadeer, Tom Ball, Gerard Basler, P. Arumuga Nainar, Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs, OSDI '08.

[11] Madanlal Musuvathi, Shaz Qadeer.Iterative Context Bounding for Systematic Testing of Multithreaded Programs,. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), June 2007.

[12] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments, Proceedings of the Fifth Symposium on Operating Systems Design

and Implementation (OSDI 2002), Boston, MA, December 2002.

[13] U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In Correct Hardware Design and Verification Methods, volume 987, pages 206–224, Stanford University, USA, 1995. Springer-Verlag.

[14] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. FormalMethods in System Design, 9(1/2):105–131, August 1996.

[15] U. Stern and D. L. Dill. Parallelizing the Murphi verifier. pages 256–278, 1997

[16] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification, pages 19–34, Apr. 2005.

[17] R. Kumar, M. Jones, J. Lesuer, and E. Mercer. Exploring dynamic partitioning schemes in hopper. Technical Report 3, Verification and Validation Laboratory, Computer Science Department, Brigham Young University, Provo, Utah, September 2003.

[18] OpenDHT: A Public DHT Service and Its Uses", Sean Rhea, P. Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu, In Proceedings of ACM SIGCOMM'05, Philadelphia, PA, August 2005.

[19] Alex Groce, Rajeev Josh. Random Testing and Model Checking: Building a Common Framework for Nondeterministic Exploration, ACM WODA – Workshop on Dynamic Analysis, July, 2008.

[20] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, Rahul Purandare, Parallel Randomized State-space Search.29th International Conference on Software Engineering (ICSE'07)

[21] Xiaoying Bai, Wei-Tek Tsai. WSDL-Based Automatic Test Case Generation for Web Services Testing, Proceedings of the 2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE’05).

[22] Jeff Offutt, Wuzhi Xu. Generating Test Cases for Web Services Using Data Perturbation, In TAV-WEB Proceedings/ACM. SIGSOFT SEN, vol. 29, number 5, September, 2004.