Parallel Symbolic Execution: Merging In-Flight Requests

Martin Nowack(✉), Katja Tietze,
and Christof Fetzer

Technische Universität Dresden, Dresden, Germany
martin_nowack@tu-dresden.de

Abstract. The strength of symbolic execution is the systematic analysis and validation of all possible control flow paths of a program and their respective properties, which is done by use of a solver component. Thus, it can be used for program testing in many different domains, e.g. test generation, fault discovery, information leakage detection, or energy consumption analysis. But major challenges remain, notably the huge (up to infinite) number of possible paths and the high computation costs generated by the solver to check the satisfiability of the constraints imposed by the paths. To tackle these challenges, researchers proposed the parallelization of symbolic execution by dividing the state space and handling the parts independently. Although this approach scales out well, we can further improve it by proposing a thread-based parallelized approach. It allows us to reuse shared resources like caches more efficiently – a vital part to reduce the solving costs. More importantly, this architecture enables us to use a new technique, which merges parallel incoming solver requests, leveraging incremental solving capabilities provided by modern solvers. Our results show a reduction of the solver time up to 50 % over the multi-threaded execution.

1 Introduction

Symbolic Execution [12] is a method to automatically and thoroughly test software and generate test cases. Lately it has also received attention in other domains like estimation of power consumption [10], analysis of mobile applications [1,14], security, and taint tracking [3,8]. Despite these advances, two major problems remain: the state space explosion problem and high solver times.

To tackle these issues, researchers introduced *process-based parallelization* [4,7,11,18] to symbolic execution engines in order to scale to multiple cores, multiple machines, and cloud environments [7]. However, this approach typically involves higher communication costs between concurrent components, e.g., using Java Remote Method Invocation (RMI) [18] or communicating with the manager process [4], which distributes and load-balances the single jobs of the symbolic execution. To reduce these costs, the search space is partitioned, so that the individual components work independently. But this way, partial solutions (e.g., previously solved path constraints that apply to multiple paths) are less likely

N. Piterman (Ed.): HVC 2015, LNCS 9434, pp. 120–135, 2015.
DOI: 10.1007/978-3-319-26287-1_8

to be reused, which results in re-execution, and thus, in redundant solver calls. Additionally, the partitioning itself imposes communication overhead for the manager process to find the best search space segmentation.

Still, solving costs are one major part of the computational expenses - i.e. as high as 40 % [16] or more of the execution time. Reducing them allows symbolic execution to explore more of the state space.

In this paper, we propose to use *thread-based parallelization* as an orthogonal approach to existing solutions. We introduce techniques like batching of parallel-running solver requests to minimize the number of solver calls and merging of solver requests to avoid redundant constraint analysis. Consequently, our approach reduces the overall solving time.

Our major contributions are:

- A multi-threaded implementation of a symbolic execution engine (we modified KLEE [5]); and
- An extension, which allows combining multiple solver calls, thus reducing the average time spent per solver request.

The remainder of this paper is structured as follows: We describe symbolic execution in Sect. 2, extending single threaded execution to our multi-threaded solution. In Sect. 3 we continue with the approach to batch and merge solver requests. We detail our implementation based on KLEE in Sect. 4, continue with the evaluation of our approach in Sect. 5, and finish with related work and a conclusion in Sects. 6 and 7.

2 Multi-threaded Symbolic Execution

In this section, first we introduce the standard single-threaded symbolic execution and point out challenges regarding the current use of process-based parallelization. Next we will present our approach to improve parallelization by applying a thread-based approach.

2.1 Symbolic Execution

Symbolic execution systematically executes a program by assuming arbitrary values for input parameters. By observing the behavior of a specific program path, it collects logic formulas (*constraints, C*) that describe this path. The collected path constraints ($PC = C_1 \wedge \cdots \wedge C_n$) are sent to a solver: (a) if the program reaches a conditional control flow statement (e.g., an `if(cond)` statement) or (b) to check a property of the current program state (e.g., if the following memory access can be a null pointer, if no out-of-bounds access or overflow occurs). In both cases, the constraint (C_{cond}) or its negation ($\neg C_{cond}$) is checked combined with the other path constraints. For the conditional control flow, if both outcomes are possible, the program state is duplicated with C_{cond} added to one state and $\neg C_{cond}$ to the other; further work on the states is done independently. If the set of constraints is impossible to fulfill the state can be removed. For the

second case (to check properties), the solver can find possible input values for the program that trigger the condition to be true or false. This greatly helps a programmer, e.g., for debugging, by providing him with a concrete input that can be used to follow a particular path, possibly to reach a bug. However, even for small non-trivial applications (e.g., containing endless-loops) this approach can lead to a huge or infinite number of paths, the so-called state space explosion problem. Besides, checking satisfiability by the solver is computationally expensive, accounting for the major part of the runtime costs.

2.2 Single-Threaded Symbolic Execution

We based our prototype on KLEE [5], a state-of-the-art implementation of symbolic execution. Other implementations are structured similarly, so our findings will be easy to transfer. The main work flow of a symbolic execution engine is depicted in Fig. 1. First, an execution unit selects a state from the pool of available states (*state selection*). The execution engine uses the state and interprets its next instructions (*interpretation*) until: (i) a final instruction of the interpreted program is reached and the state is terminated; (ii) a bug is found, so the state must be terminated prematurely; or (iii) another state is selected. When path constraints must be solved or conditions need to be checked, the interpreter invokes the solver (*solving*).

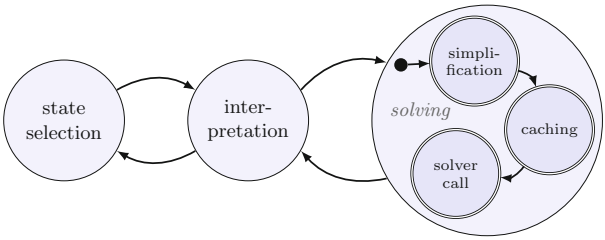


Fig. 1. The basic symbolic execution flow: First, a state is selected and then interpreted. If needed, collected constraints during interpretation are solved. To reduce the solver overhead, several optimizations like simplification and caching are done before the actual solver call.

The solver is a major, computationally expensive part of the symbolic execution. The feasibility of formulas has to be calculated, which is typically NP complete. Hence, a chain of additional optimizations (*simplification*, like independence optimization or *caching* [5]) is executed before the solver is called.

Symbolic execution engines like KLEE typically use existing solvers [9,16]. But these can suffer from issues like segmentation faults or memory leakage. To improve the reliability, KLEE forks and runs the solver in the child process from where it returns the result through shared memory. As expected, the forking imposes non-negligible overhead as we show in the evaluation.

2.3 Going Multi-Threaded

A major challenge of parallelizing symbolic execution is to efficiently partition and search the execution tree without knowing in the first place which parts are costly. First, if a leaf node is explored, the size of the subtree it expands is typically unknown. For example, if only a few expansions are possible, the subtree will be terminated earlier than if it must be expanded repeatedly. Second, the individual cost of exploring each node of a subtree is highly dependent on the costs of solving the path constraints, which are not easy to predict.

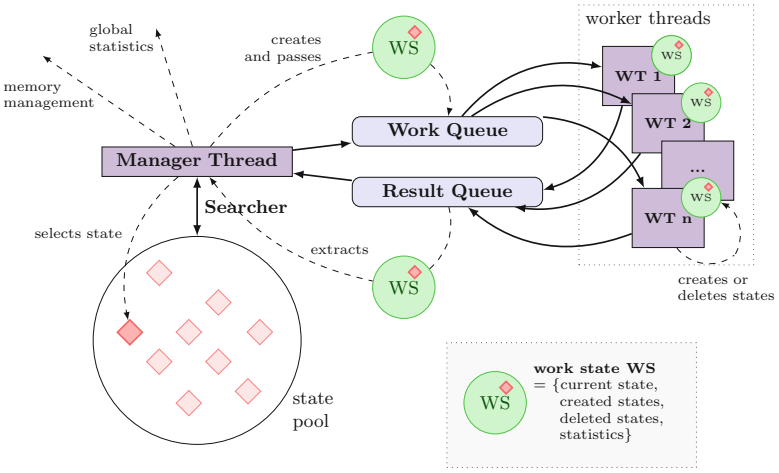


Fig. 2. Basic general architecture of a multi-threaded symbolic execution. A manager thread prioritizes states and puts them in a working queue. Worker threads select states from the queue and work independently on them, each calls the solver individually.

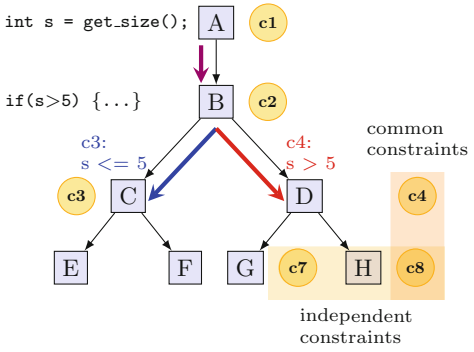
We propose the architecture depicted in Fig. 2. It resembles the steps from Sect. 2.2 Fig. 1 and assigns them to different threads. A *manager thread* is decoupled from a group of *worker threads* by using two queues. While the *work queue* buffers the states that are to be handled by the workers, the *result queue* is filled with states returned by the workers. The manager selects the states from the state pool and puts them in the work queue. With the order in which states get inserted, the managing thread can steer the exploration process. This selection is based on the statistics that are maintained by symbolic execution engines. They include information about the code checked during operation, e.g., which lines of code were already covered. The manager further extracts finished states from the result queue and updates the global statistics. In case not enough memory is available, e.g., due to state space explosion, the manager deletes the least important states (e.g., the state furthest away from uncovered instructions. Each worker takes a state from the work queue and runs the symbolic execution by iterating over the code. In case the solver is called, the worker executes it in a forked process.

The *current state* to be worked on is wrapped in a *worker state* used by only one worker at a time and is thread local. Workers add new states (which were found by their current exploration) to it, mark states as deleted, and update local statistics. Later, these information are used by the manager thread to update the state pool and global statistics. This way, workers manage their updates locally which improves their access time and reduces pressure on the CPU cache.

The major advantage of our thread-based approach is an improved utilization of the CPU cores while additional communication costs between different processes are avoided. At the same time, components like caches are shared. New solutions found by the solver are added to the cache which avoids solving them again.

3 Reducing Overall Solver Costs

In the previous section, we explained how we use more cores for solving, but the overall solver costs are still significant. Fortunately, our multi-threading architecture allows new ways of combining solver requests to reduce the number of solver calls. For our approach we distinguish between *unique constraints* and *common constraints* as depicted in Fig. 3(a). While unique constraints are state specific (i.e., request specific), common constraints are true for multiple states. Hence, solutions for common constraints can be reused without recalculation. The figure also shows how constraints are gathered along a path of the code tree.



(a) State exploration

```
(push) # start common scope
# Common constraints
(assert c1)
(assert c2)
# Individual constraints
# for first worker
(push)
(assert c3)
(check-sat)
(pop)
# Individual constraints
# for second worker
(push)
(assert c4)
(check-sat)
(pop)
(pop) # finish common scope
```

(b) Incremental SMT solving

Fig. 3. Example of state exploration (a) and incremental solving using push/pop (b). If two workers work on C and D independently, they still share a common path. In that case, to expand C respectively D, the solver can reuse solutions.

We leverage two key observations (i) the likelihood that two or more requests are worked on in parallel is quite high; and (ii) requests from different paths can share a common history, hence they are likely to share common path constraints. More precisely, we address the first observation by batching requests and the second by reusing the partial solutions for common constraints, as we explain in Sects. 3.1 and 3.2.

For the second option, we leverage the feature of recent SMT solvers to *incrementally* find a solution to solver requests (e.g. [6,15]). Instead of solving all requests independently, we first start a new scope (`push`) and put all common constraints in it (Fig. 3(b)). After that, for the remaining constraints, we open a new scope (`push`) for each individual request, solve the constraint (`check-sat`) and close the scope again (`pop`). This way, specific lemmas learned get removed, common lemmas are kept (e.g. for value forward propagation).

3.1 Batching Solver Requests

We enable a rendezvous of solver requests by a new layer in the solver chain as depicted in Fig. 4. The *waiting barrier* (before the solver call) consists of a set of slots, each describing a possible rendezvous point.

Our waiting barrier is a ring buffer with multiple slots. Each slot can gather a number of queries. All requests in one slot will be combined to be handled by one solver request, instead of each provoking a solver call individually. This batching reduces the number of solver calls, thus reducing the overall costs.

Apart from the set of queries, each slot contains a *busy flag*, which indicates whether a solving process is currently running for the queries gathered in the slot. A global *open slot pointer* indicates the next open slot in the ring. Open means that the solving process for the queries in this slot has not yet started, so other workers are still allowed to add their query to the slot.

We distinguish worker threads and, along them, one *primary* worker thread per slot, which acts as a master and manages the solving process. The course of a solving process is depicted in Fig. 5. The first thread to enter an open slot will become the primary. Then it starts spinning until a *waiting timeout* has

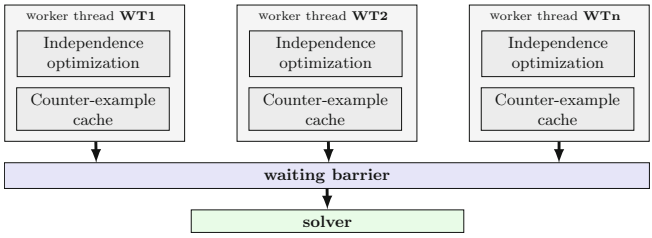


Fig. 4. Additional rendezvous layer: Instead of every worker calling the solver individually, each thread waits briefly at a waiting barrier for potential other workers to call the solver. In case a worker arrives in the meanwhile, their requests are combined.

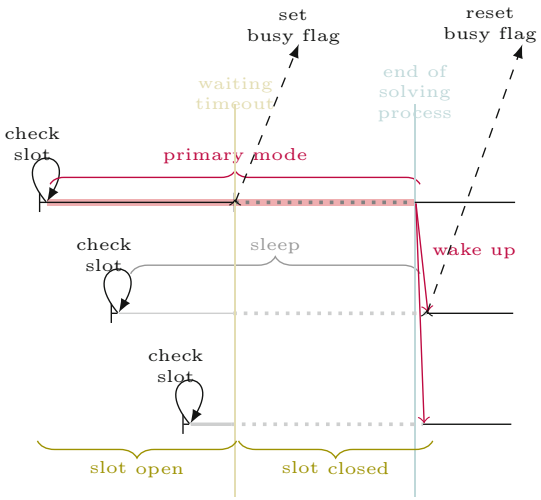


Fig. 5. Temporal course of the solving process. The first worker to arrive at a slot becomes primary and waits for others. Later arriving workers, enqueue their solver request in the same slot and sleep. After a timeout, the first thread closes the slot, solves all queries, returns the result to all sleeping workers, and wakes them up.

exceeded. Subsequently it sets the busy flag, increments the open slot pointer, and triggers the solving process.

Every worker thread that enters a slot with an existing primary goes to sleep immediately. When the solving process is finished, the primary wakes up the other workers, hands them their solver results, and all threads leave the slot, thus leaving the waiting barrier.

Currently, our matching is trivial: as long as the current slot is still open, new requests can be matched to others in this slot. However, for future work we plan to improve this and follow a multiple lane approach, in which we use multiple ring buffers and assign threads to slots based on the complexity of the request ("slow track" vs. "fast track"). For this refinement the waiting time depends on (i) the simplicity of the request and, therefore, the benefit of waiting for another request to arrive and (ii) the fork time needed to call the solver. If a request is simple its solution can be calculated quickly, so waiting is too expensive ("fast track"). As shown by [16], for many benchmarks most solver requests take 1 ms or less, which we could confirm for our own experiments. We estimate the complexity of requests resembling [18]: we weight the constraints depending on their costs. More expensive request wait longer for potential candidates to combine them with ("slow track").

3.2 Merging and Solving Requests

Once the primary worker's timeout has exceeded, it sets the busy flag (to prevent other workers from entering the slot) and start the solving process. It consists

of two steps, which facilitate the capabilities of incremental solvers. First, it calculates the intersection of all constraints, thus determining the *common constraints*, which are shared by all queries. The solver will cache this solution for the common constraints to subsequently reuse it for solving all queries without recalculation. Second, the requests of all worker threads are handled iteratively: for each worker, the primary pushes the unique constraints on the working stack, solves the request, and combines it with the previously calculated solution for the common constraints.

Once the primary finished solving all worker requests, it wakes the sleeping worker threads of this batch and passes them their respective results. The threads leave the slot, thus splitting up again.

Currently we only determine the common constraints of all requests. However, in future work we plan to extend on this and also calculate subsets of constraints that are shared by multiple (but not all) workers in a slot. This way, solver time could be reduced further.

3.3 Merging vs. Caching

One could argue that a global cache to store common constraints and their solutions should be sufficient and simpler in comparison to our proposed solution. A major challenge we see is the identification of common constraints. One cannot know easily upfront the number of common constraints for two or more requests. Therefore, to make a global cache efficient, one has to save subsets of constraints for requests. This poses two challenges, first to subsequently match, the subset should not contain specific constraints otherwise subsequent hits are less likely; second, choosing the right size of a subset influences the number of entries in the cache and its hit rate. The smaller entries should be, the larger the cache has to be. Otherwise, entries might get evicted too often.

4 Implementation

Our prototype is based on the current version of KLEE [5][1]. As a symbolic execution engine for running C and C++ programs, KLEE is built on top of LLVM [13] (a tool chain for building compilers and interpreters) and uses STP [9] as a solver for quantifier-free first order logic with support for bit-vectors and arrays.

We parallelized KLEE by implementing the architecture described in Sect. 2.3. By converting KLEE to C++11, it enabled us to leverage support for multi-threading (e.g. using `std::shared_ptr`, `std::atomics`).

Based on the basic architecture, we added our waiting barrier as an additional stage before the final solver as depicted in Fig. 4. Precisely, the waiting barrier is a ring buffer with a number of slots whose contents consist of a set of queries and a busy flag. The total number of slots in the ring buffer is equal to the number

[1] https://github.com/klee/klee.

of CPUs, so every worker thread running in parallel is guaranteed to fit into a slot.

Once a thread enters an open slot, it will be assigned one of two roles. The first thread to enter runs in *primary* mode, thus following a separate code path and executing primary tasks. All threads that enter a slot with an existing primary become *secondaries*.

Flexible waiting timeout: Previously, we explained that threads are matched to each other by being assigned to the same slot in the waiting barrier. While our concept allows for different approaches of a waiting timeout, our prototype implements the following batching process: Every time the primary detects a newly arrived secondary, it will decrease the waiting timeout by interval I, with N being the number of possible parallel threads (i.e., the number of CPUs) and T_w being the length of the waiting timeout: $I = T_w/N$. Hence, the new waiting timeout T'_w is $T'_w = T_w - I$.

This flexible waiting timeout allows to include multiple threads in the batching while reducing the overall idle time. However, in the meantime between the end of the waiting timeout and the slot actually being closed, other threads can still enter the slot.

In future work, we will focus on a more sophisticated batching process. First, we intend to experiment with varying waiting timeouts and matching approaches. Second, we will implement a multiple lane approach to allow different complexity tracks as described in Sect. 3.2.

Resetting the busy flag: When the solving process is finished, the primary wakes up the secondaries and passes them their respective solutions. To avoid unnecessary delays of single threads and additional overhead, the primary gives up its role right afterwards. The last thread to leave the barrier resets the busy flag and, thus, opens the slot again.

Limitations to the approach and the prototype: Calls to external libraries have to be synchronized. Our implementation resembles the original KLEE, e.g., if a worker calls an external function (e.g. to print to screen), all memory objects contained in the current state will be written to the real process memory, the external call will be executed, and changes will be written back. If other workers also need to execute external calls they will be stalled.

5 Evaluation

For our experiments we used an Intel E5405 2 GHz with 8 cores and 8 GB of RAM running Ubuntu 14.04.02. To evaluate our approach, we used Coreutils suite[2] 6.10 as our benchmark suite, the same version as in [4,5,16]. Coreutils contains around 90 basic utilities for text, file, and shell manipulation and are part of most Linux-based systems. Written in C, the utilities often have system interactions

[2] https://www.gnu.org/software/coreutils/.

and contain low-level bit manipulations. We used the same input parameters for KLEE as in [5] as far as they are available in the current unmodified KLEE version. The major difference of our experiment setup is that we run the 64 bit version of the test applications, our KLEE-based implementation builds on top of LLVM 3.4, and we limit the maximum memory to use to 4 GB. We allow a solver request to take up to a maximum of 30 s to execute.

Each of the test tools from the benchmark suite was run 3 times and up to 1h each for every version of our implementation.

5.1 Time Spent by the Solver

To be able to show the improvement of the solving time, we first need to have a closer look at the solving time spent in the case of the single-threaded execution. We ran the Coreutils benchmark with the original version of KLEE and recorded the generated solver requests and the time it took (t_{klee}). In a second step, we used the solver independently of KLEE and re-ran each request individually using the solver front end `KLEAVER`. We disabled the forking mechanism or any other optimization happened before the solver call, and recorded the solving time (t_{real}). To our surprise, the solving times varied vastly ($t_{diff} = t_{klee} - t_{real}$).

One example run is depicted in Fig. 6 for `csplit`. The graph depicts solver request and their execution time over finished instructions made during 1 h. The green dots depict the real solving time (t_{real}). The blue dots depict the additional time (t_{diff}) spent for a request. The solving time is measured in seconds with maximum of 21.89 s. The allocated memory (red points) is depicted in the same graph as well. We normalized the measured values to a 0–1 range with 1 being equivalent to a 4 GB allocation.

As one can see, the real solver time is distributed in three lanes. The major part of requests is below 10 ms with a slight concentration in the 1st third of the time; followed by up to 100 ms quite equally distributed over time; with only minor number of requests above more concentrated during the last two third of the execution.

As one can see, the solver time difference (t_{diff}) increase over time highly correlates with the total size of allocated memory (red points). The higher the allocation, the higher is t_{diff}.

We could reproduce similar behavior with a very simple toy program executed natively that has two phases: first, it allocates an amount of memory and writes into it, and in the second phase, a few thousand `forks()` are executed. The average time a fork takes increases linearly with the amount of allocated memory. If multiple threads run in parallel in the second phase, the fork time will not change significantly over the single-threaded execution.

In essence, the less often we need to fork, especially under memory pressure (e.g. due to state space explosion), the more time can be spent in solving. Therefore, if many solver requests can be merged and combined to one, the overall time spent in forking can be reduced.

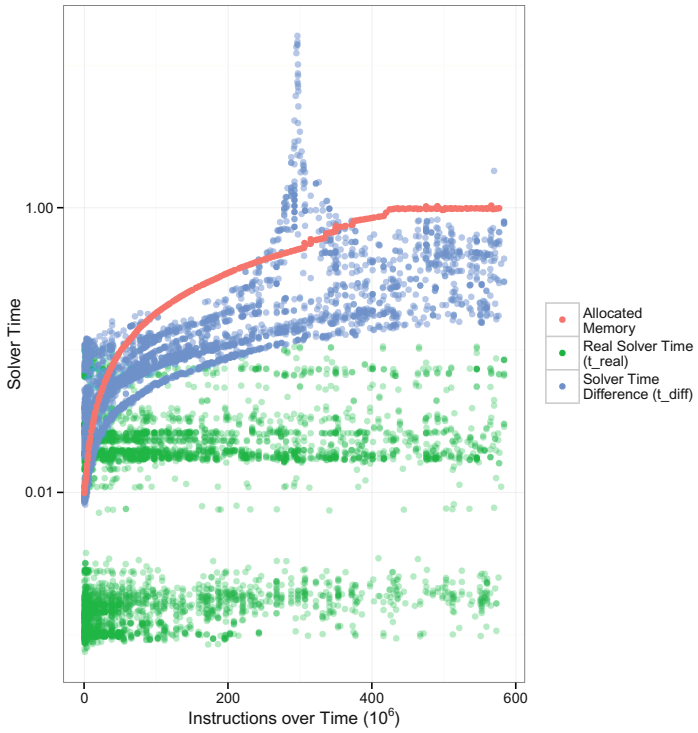


Fig. 6. Example of one run for `csplit` and the time spent for solving the requests (Color figure online).

5.2 Thread-Based Parallel Symbolic Execution

Our major contribution is combining in-flight solver requests which is built on top of a multi-threaded implementation of a symbolic execution engine. Before we evaluate the combining, we evaluated the thread-based symbolic execution.

To show the effectiveness of our multi-threaded implementation, we need to compare it with the single-threaded execution. To do so, we use two metrics: the *achieved line coverage* and the *instructions executed*.

Multi-threading adds a big source of randomness, which makes it hard to compare single-threaded and multi-threaded runs. Already for original single-threaded KLEE, its behavior is highly random, making it hard to repeat experiments. We took similar measures as [16] to disable random influences (e.g., deactivate address space random layout). To show the effectiveness (e.g. using line coverage criterion) of our multi-threaded implementation, we could not use non-random searchers like depth-first search or breadth-first search. It would be a disadvantage for single-threaded implementations, as already with a second worker thread covering an additional path, a multi-threaded implementation has an advantage over a single threaded implementation. We used a searcher which selects states randomly from the state space but prefers states close to an uncovered instruction.

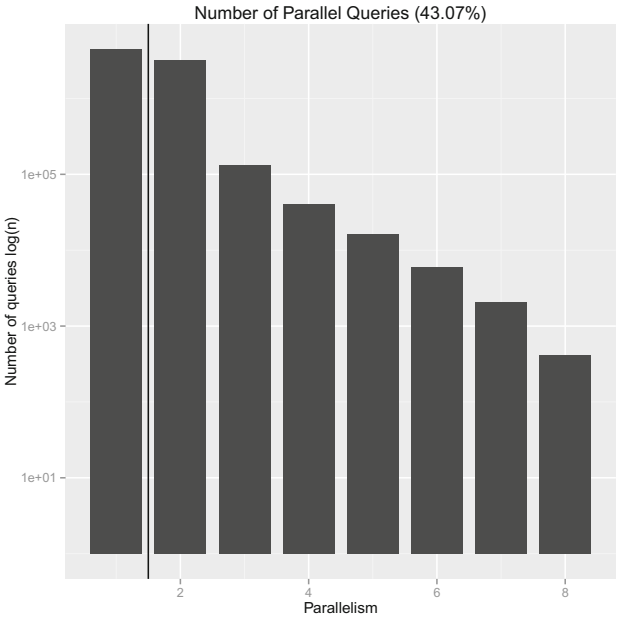
Our results for the coverage show that for the group of applications of Coreutils, which can be fully covered during the execution of 1 h, the multi-threaded execution achieves the full coverage in most of the runs earlier up to 6.7x faster for 8 threads. For the second group of applications, for which full coverage cannot be achieved during 1 h, the multi-threaded implementation achieved on average a higher coverage faster in comparison to the single-threaded execution. Still, the coverage metric is highly influenced by selecting the right states to explore as early as possible. Using *instructions executed* as a metric is also fragile. If a taken path covers instructions for which the constraints are hard to solve, less instructions can be executed. In contrast, if constraints can be solved fast, more instructions can be executed. Still our experiments show an improvement for 14 out of 40 experiments executing at least 2 billion more instructions and up to 5 billions more at most (e.g. `md5sum`).

5.3 Batching and Merging Parallel Requests

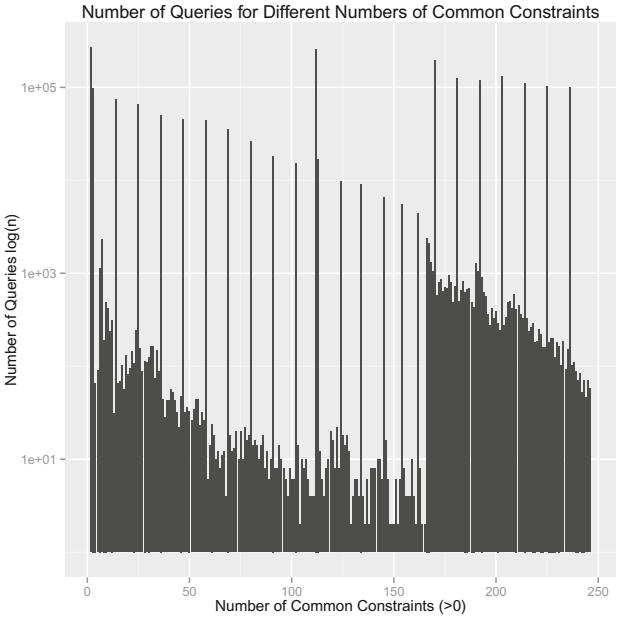
The success of our method depends highly on how many requests can be batched. Obviously, this depends on the time a thread in primary mode is waiting at the barrier for other threads to arrive and the average time it takes to solve a request. The second is often specific to the application under test (Sect. 3.1). In our tests, we varied the waiting time between 1 ms and 100 ms. As a result, different applications were favored depending on the timeout of the waiting barrier.

How often are solver requests batched and how many together? We run all the Coreutils suite with our implementation of merging in-flight requests. We recorded the solver requests which have been made. The experiments generated 7.8 million queries per different waiting configuration. Figure 7(a) shows the distribution of combined queries. For every observed number of queries involved for batching (x axis - 1 to 8), we depict the number of how often this happened. Our testing machines has 8 cores and up to 8 queries were batched, therefore up to 8 threads arrived at the waiting barrier and used the same slot at one time. The majority of requests do not execute in parallel (57 %). For the remaining requests, two threads share requests most often. It is an order of magnitude less for 3, down to a couple of hundreds for 8.

How many constraints did batched requests have in common? The number of shared constraints for batched requests varied highly. Figure 7(b) shows the distribution over the number of common constraints. It varies between 0 and 246 in the example of the Coreutils experiment. On average, requests had 69.9 constraints in common. This indicates a good applicability of our method. Reason why the graph has the ragged shape are the way a program is explored. Our used searcher prefers states close to uncovered instructions. If instructions from a code region are uncovered, branch points in the code enable different workers to work on similar states. This makes it more likely to have common constraints. Also loops in code containing switches make it more likely that several workers work on different switch cases but still sharing common constraints.



(a) Distribution of combined queries encountered.



(b) Distribution of common constraints for combined queries.

Fig. 7. Results for Coreutils having a barrier waiting time of up to 10 ms. Total number of queries: 7.8 million.

Results show a saving of solver costs up to 50 % over the multi-threaded only version, which is almost 7 h in case of the time spent for solver requests on Coreutils.

5.4 Prototype Limitations

External Calls Calling external functions (i.e. system calls) is essential for testing system applications like the ones in Coreutils The problem is that changes made by calls to external libraries (i.e. modifications of the memory) cannot be observed by KLEE. Therefore, the current implementation of KLEE works around the issue by, first, writing all memory associated to the execution state to their native positions, second, executing the external call, and last, writing back all changes from the native memory to the execution state. For our prototype, to avoid inconsistencies, we made calling external calls mutual exclusive. This has an impact on the execution time of the application, we measured waiting costs up to 8 % of the total execution time for each thread (i.e. for `md5sum`). On average it was 1.8 % per application and thread.

5.5 Threats to Validity

We verified our implementation and experiment setup to the best of our knowledge. Still it is hard to cope with inherent randomness of symbolic execution plus the multi-threaded implementation. We tried to mitigate the problem by running the experiments multiple times and calculating the average. We validated our approach comparing the code coverage of the checked applications, which is similar or better.

6 Related Work

To the best of our knowledge, no existing solutions follow a thread-based approach as proposed by us. All solutions we know build on a process-based approach to utilize multiple cores for symbolic execution.

Staats et al. [18] propose an extension to parallelize the symbolic execution part of Java Pathfinder [2]. They partition the search tree statically upfront. First, they perform a symbolic search with iterative deepening to depth n and collect the sets of constraints. Second, they reassemble the constraints according to complexity and use them to split the domains of the input variables in independent pieces. In a third step, they assign these pieces to different workers, which work on them separately to reduce costs. Bucur et al. [4] propose a dynamic approach named Cloud 9. The search tree is split dynamically and a load balancer observes the workers. As soon as a worker is over-utilized, its search tree is split and parts of the tree are transfered to the load balancer, which distributes them to under-utilized nodes. Our approach can be combined with both solutions to help each worker accelerate its progress, thus allowing

a better overall throughput. Additionally, our thread-based parallelization can also enhance the light-weight symbolic execution proposed by Staats et al. [18].

An interesting alternative is using a portfolio solver as proposed by Palikareva et al. [16]. A request is sent to different SMT solvers in parallel and the fastest and/or best solution will be used. This can efficiently utilize multiple cores available on the same machine. We see our work as an orthogonal approach.

Our solution for merging solver requests is based on similar ideas (see Pötzl et al. [17]) for solver preprocessing, e.g., Push/Pop Encoding.

7 Conclusion

We presented a new way to combine solver requests in symbolic execution: We leverage multi-threaded execution and merge parallel occurring solver requests. Our prototype shows a reduction of the solver time by up to 50 %. We consider our work a viable extension of existing approaches for parallelized symbolic execution.

Acknowledgment. We thank the anonymous reviewers for their insightful comments. This work was partially founded by the German Research Foundation (DFG) under grant FE 1035/1-2.

References

1. Anand, S., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone apps. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012, pp. 59:1–59:11. ACM, New York (2012). http://doi.acm.org/10.1145/2393596.2393666
2. Anand, S., Păsăreanu, C.S., Visser, W.: JPF–SE: a symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
3. Avancini, A., Ceccato, M.: Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. Inf. Softw. Tech. **55**(12), 2209–2222 (2013). http://dx.doi.org/10.1016/j.infsof.2013.08.001
4. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: EuroSys 2011: Proceedings of the Sixth Conference on Computer Systems, pp. 183–198. ACM Request Permissions, New York, April 2011. http://portal.acm.org/citation.cfm?doid=1966445.1966463
5. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 209–224. USENIX Association, Berkeley (2008). http://dl.acm.org/citation.cfm?id=1855741.1855756
6. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)

7. Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., Candea, G.: Cloud9: a software testing service. ACM SIGOPS Operat. Syst. Rev. **43**(4), 5–10 (2010). http://dl.acm.org/citation.cfm?id=1713254.1713257
8. Corin, R., Manzano, F.A.: Taint analysis of security code in the KLEE symbolic execution engine. In: Chim, T.W., Yuen, T.H. (eds.) ICICS 2012. LNCS, vol. 7618, pp. 264–275. Springer, Heidelberg (2012)
9. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
10. Hönig, T., Eibel, C., Kapitza, R., Schröder-Preikschat, W.: SEEP: exploiting symbolic execution for energy-aware programming. Operat. Syst. Rev. **45**(3), 58–62 (2011). http://doi.acm.org/10.1145/2094091.2094106
11. King, A.: Distributed Parallel Symbolic Execution. Master's thesis, August 2009
12. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). http://doi.acm.org/10.1145/360248.360252
13. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. pp. 75–86. IEEE Computer Society, March 2004. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281665
14. Mirzaei, N., Malek, S., Pasareanu, C.S., Esfahani, N., Mahmood, R.: Testing android apps through symbolic execution. ACM SIGSOFT Softw. Eng. Not. **37**(6), 1–5 (2012). http://doi.acm.org/10.1145/2382756.2382798
15. de Moura, L., Bjørner, N.S.: Z3: an efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer, Heidelberg (2013)
17. Pötzl, D., Holzer, A.: Solving constraints for generational search. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 197–213. Springer, Heidelberg (2013)
18. Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: The 19th International Symposium, p. 183. ACM Press, New York (2010). http://portal.acm.org/citation.cfm?doid=1831708.1831732