ParSym: Parallel Symbolic Execution

Junaid Haroon Siddiqui Sarfraz Khurshid
The University of Texas at Austin
Austin, TX 78712
{*jsiddiqui,khurshid*}*@ece.utexas.edu*

***Abstract*—Scaling software analysis techniques based on source-code, such as symbolic execution and data flow analyses, remains a challenging problem for systematically checking software systems. The increasing availability of clusters of commodity machines provides novel opportunities to scale these techniques using parallel algorithms.**

This paper presents ParSym, a novel parallel algorithm for scaling symbolic execution using a parallel implementation. In every iteration ParSym explores multiple branches of a path condition in parallel by distributing them among available workers resulting in an efficient parallel version of symbolic execution. Experimental results show that symbolic execution is highly scalable using parallel algorithms: using 512 processors, more than two orders of magnitude speedup are observed.

I. Introduction

Inefficacy of traditional manual and random testing has led to much recent progress in automated and systematic testing techniques. One particularly promising technique is *symbolic execution*. Although the basic idea dates back at least three decades [15], recent advances [23], [10], [5] have enabled symbolic execution to work for unmodified software. The key idea behind symbolic execution is that instead of exploring concrete program executions, the program is executed on *symbolic inputs* by defining program operations to perform symbolic manipulations. When a branch statement is encountered during symbolic execution, both branches are considered while recording the constraint on symbolic inputs required to follow the corresponding branch. At program termination or at hitting a potential bug, the collection of constraints on symbolic inputs for that path (called *path condition*) is solved (if feasible) to generate concrete values that lead the (deterministic) program along that same path on a concrete execution.

Covering all paths is impossible for all but trivial programs. Thus, a practical search of the program's behaviors needs to be *depth-bounded* or time capped. This has proven to be an effective strategy for checking small units of code. However, checking larger systems may require hours or even days to achieve acceptable code coverage. A novel opportunity for scaling symbolic execution arises from the increasing availability of cheap commodity (multi-core) processors by developing parallel algorithms for symbolic execution, thereby increasing the efficacy of symbolic execution to many more programs.

This paper presents ParSym, an algorithm for parallel symbolic execution. ParSym supports the combined symbolic and concrete execution model [23], [10]. In this model, the given program is executed on an initial input (either random or some initial values like zeroes, null, empty string etc.). A path constraint of the execution is built (as for pure symbolic execution). Each constraint in the path condition represents a branch on a symbolic input. The last constraint is negated and the path condition is solved for new concrete inputs. This leads to the execution of the *not taken* side of the last branch statement. When no new branches are encountered and the last constraint has already been negated once, the exploration backtracks on the path constraint to explore previous branches.

Our insight into parallel symbolic execution is two-fold: One, symbolic execution requires the negation of several branch conditions, each of which must be explored; and Two, the time to solve path conditions dominates the time to execute a path. Symbolic execution is, therefore, likely to benefit significantly from parallel algorithms, which can effectively distribute the workload among different workers.

This paper makes the following contributions: 1) a novel parallel algorithm for executing symbolic execution on a parallel cluster, 2) an efficient implementation of our algorithm, and 3) evaluation of our algorithm on GREP, a C program with 15K lines of code, and a program to perform bounded exhaustive testing of binary trees.

II. Algorithm

Symbolic execution works with path conditions often stored as complex expression trees. The main problem in parallelizing symbolic execution is to avoid transport of these structures between parallel nodes. Concrete inputs, on the other hand, are usually small and can be further compressed by using input domains (pre-decided sets of values). We divide work such that each node gets a set of inputs, executes the program concretely while observing the symbolic path constraint built. It then negates each constraint turn by turn, solves the path condition for concrete inputs and sends the input indices to a parallel node. Thus we minimize communication overhead.

There are three parts of our parallel symbolic execution algorithm. The core symbolic execution engine that concretely runs programs, forms path conditions, and solves constraints,

```
function ParSym( workItem )
    (depth, pos, inputs) $\leftarrow$ workItem
    (constraints) $\leftarrow$ ExecAndObserve(testProgram)
    while depth>0 $\wedge$ pos<Size(constraints) do
        Negate(constraints[pos])
        (success, inputs) $\leftarrow$ Solve(constraints[0...pos])
        if success then
            depth $\leftarrow$ depth-1
            newWork $\leftarrow$ (depth, pos+1, inputs)
            NewWorkItem(newWork)
        end if
        Negate(constraints[pos])
        pos $\leftarrow$ pos+1
    end while
end function
```

Figure 1: Parallel Symbolic Execution.

a central symbolic execution monitor that monitors the overall process, and a symbolic execution agent that helps the core engine communicate with the monitor.

A. Symbolic Execution Engine

Our parallel symbolic execution algorithm is based on combined symbolic and concrete execution [23], [10]. We start by executing the program on an initial input (zeroes, null pointers, empty strings etc.) and observe the path it traverses. We then negate the first constraint and solve it for concrete values. We then move to the next constraint, negate it, and solve all constraints up to the negated constraint in the path condition. We repeat until we get enough solvable constraints (depth bound) or until all constraints are solved.

Baseline symbolic execution will re-run the program after solving the first constraint and then work on the new path condition formed (which should only differ in the part after the negated condition). It only backtracks to the original path condition when it has explored all newer path conditions exhaustively (under given depth bound).

In Figure 1, we show implementation of core engine in function ParSym. It receives inputs and uses them for a concrete execution and observes the path condition like standard symbolic execution. It then checks the path condition from the $\texttt{pos}^{th}$ position and works until all constraints are traversed or enough new items are produced (according to depth bound). At each position the constraint is negated, if the resulting path condition (up to that constraint) is solvable, it produces a new work item that is at the given depth, whose constraints before `pos+1` have been negated once, and whose inputs are the result of solving.

We based our implementation an an open source symbolic execution implementation [3], the CIL instrumentation library [21], and the CVC3 SMT solver [1]. The Solve method in Figure 1 uses the CVC3 library to solve the constraint.

The key idea in this implementation is that the path constraints need not be transferred to other nodes. Only program inputs have to be transferred. Also the function proceeds from solving smaller (and therefore easier and quicker to solve) constraints to longer (and time taking) constraints. Thus the quicker a work can be dispatched to another node, the quicker it is done.

The initial work item contains the maximum depth specified by the user, index of first constraint in path condition to be negated (zero), nothing as input (zeroes and nulls are used as default by instrumentation). The function ExecAndObserve runs the instrumented program and returns the path condition. The path condition is used by rest of the algorithm.

B. Symbolic Execution Monitor

Symbolic Execution Monitor provides the central authority that carries out symbolic execution of a program. It starts by distributing the executable to be tested to all nodes running symbolic execution engines using agents running on each machine. After that it serves as master in a typical master slave configuration [11]. It contains a work queue of inputs to be used for symbolic execution. All other nodes have a symbolic execution engine that contacts the monitor for an input to process using the agent. Any new items generated by the engines are sent back to the monitor, and the monitor enqueues them.

The algorithm for monitor is given in Figure 2. The monitor maintains a list of agents that want to work (`agentQ`), agents that have no more work to do (`exitQ`), and a list of pending work items (`workQ`). This algorithm supports double buffering at the client (receiving the next work item while processing the previous one) and that's why two queues (`agentQ` and `exitQ`) are needed. When a agent has finished work and has not received new work in the background, it requests for addition to the `exitQ`.

Double buffering is important to minimize wasted time at the nodes performing actual symbolic execution. By the time they finish with one input, another is ready for consumption. This eliminates the need to have larger work divisions.

The initialWorkItem (underlined) for symbolic execution is an initial input that can be random or some predetermined initial values (like zeroes, nulls, empty strings etc.). It also contains depth of current symbolic execution iteration and position of the next constraint to be negated along with concrete inputs to the program. The initial input contains the depth bound and zeroth position to explore all parts of the constraint. The algorithm loops and serves messages from the agents.

There are three types of messages from agents to monitor:

- Queue: is used to ask the monitor to add new work items to the work queue.
- Dequeue: is used to ask the monitor for a new work item to process. If a work item is not readily available, the agent is remembered in agent queue. Whenever

```
function Monitor(agentCount, initialWorkItem)
    workQ ← createQueue( )
    agentQ ← createQueue( )
    exitQ ← createQueue( )
    Queue(workQ, initialWorkItem)

    while Size(agentQ) $\neq$agentCount
        $\wedge$ Size(exitQ) $\neq$ agentCount do
        (agent, cmd, workItem) ← Recv(any)
        if cmd = Queue then
            if Empty(agentQ) then
                Queue(workQ, workItem)
            else
                agent′ ← Dequeue(agentQ)
                Send(agent′, Work, workItem)
            end if
        else if cmd = Dequeue then
            Remove(exitQ, agent)
            if Empty(workQ) then
                Queue(agentQ, agent)
            else
                workItem ← Dequeue(workQ)
                Send(agent, Work, workItem)
            end if
        else if cmd = Exit then
            Queue(exitQ, agent)
        end if
    end while
    for all $s$ ← agentQ do
        Send($s$, Exit)
    end for
end function
```

Figure 2: Algorithm for symbolic execution monitor.

work items become available, they are sent to these free agents instead of being queued in work queue.

- Exit: is used to tell the monitor that the agent has finished all work and is safe to exit. However this message does not mean that the agent *will* exit. In fact it may get more work from the monitor soon.

The monitor sends only two messages back to the agents:

- Work: is used to give new work to the agent. It is sent in response to a Dequeue request so the agent should be ready and willing to receive it.
- Exit: is used to ask the agent to exit. It is sent only when the agent has expressed will to exit by an Exit message in the other direction. Therefore it should be able to safely exit immediately.

Safe termination of the algorithm assumes ordering of messages between monitor and a particular agent. Simultaneous occurrence of a agent in `exitQ` and `agentQ` requires that it send an Exit message when it is in the `agentQ` (due to a previous Dequeue message). Any later Dequeue message would first remove it from `exitQ`. Also on a Dequeue message, the monitor immediately serves a agent if work is available. Thus simultaneous occurrence of *all* agents in both these queues means that no work is available in the `workQ` and all agents have finished their assigned work. At this time the monitor terminates its loop and sends an Exit message to all agents.

```
function agent( )
    workItem ← createBuffer( )
    workItemBack ← createBuffer( )

    Send(monitor, Dequeue)
    Send(monitor, Exit)
    (cmd, workItem) ← Recv(monitor)
    while cmd = work do
        Send(monitor, Dequeue)
        (cmd, workItemBack) ← RecvStart(monitor)

        ParSym(workItem)

        if not RecvFinish( )then
            Send(monitor, Exit)
            WaitForRecvFinish( )
        end if
        workItem ← workItemBack
    end while
end function

function NewWorkItem(workItem)
    Send(monitor, Queue, workItem)
end function
```

Figure 3: Algorithm for agent processors.

C. Symbolic Execution Agent

Symbolic execution agents are responsible for providing work items to the core symbolic execution engine. Algorithm for agent processors is given in Figure 3. The two buffers `workItem` and `workItemBack`) allow double buffering. When one is being processed, data from the monitor is received in the other. This allows hiding communication overhead and latency and avoids any wasted time for the core engine.

At the start, every agent sends the monitor processor a Dequeue message followed by an Exit message. If work is given to the agent, the Exit message is benign (see Section II-B). However if there are few work items, and the particular agent will never get any work to do, the pair of messages will allow the monitor to remember that this agent is free.

The agent loops until it receives Work messages (an Exit message will break the loop). To process a work item (inputs for symbolic execution), the function ParSym in Figure 1 is used. This function traverses the search graph and finds new inputs for symbolic execution. Figure 3 also shows NewWorkItem, a wrapper for sending an item to the monitor.

III. Evaluation

We evaluated parallel symbolic execution on two problems. One is symbolic execution of GREP 2.2. The other is using symbolic execution to perform bounded exhaustive

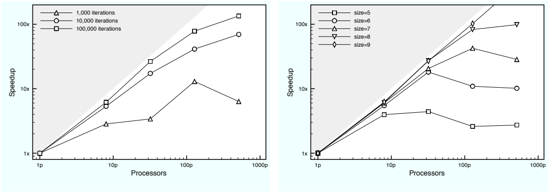
Iterations	Serial	Parallel Time (Speedup)								
	Time	2p	8p		32p		128p		512p	
1,000	0:00:28	0:00:33	0:00:10	(3x)	0:00:08	(3x)	0:00:02	(13x)	0:00:04	(6x)
10,000	0:05:15	0:05:48	0:00:59	(5x)	0:00:18	(17x)	0:00:08	(41x)	0:00:04	(70x)
100,000	1:09:23	1:17:10	0:11:14	(6x)	0:02:37	(26x)	0:00:53	(78x)	0:00:31	(135x)
650,000	TIMEOUT	TIMEOUT	TIMEOUT		3:47:05		0:52:35		0:13:14	

(a) Symbolic Execution of GREP 2.2

Size	Serial	Parallel Time (Speedup)								
	Time	2p	8p		32p		128p		512p	
5	0:00:29	0:00:34	0:00:09	(4x)	0:00:08	(4x)	0:00:13	(3x)	0:00:10	(3x)
6	0:02:05	0:02:22	0:00:23	(5x)	0:00:07	(18x)	0:00:11	(11x)	0:00:12	(10x)
7	0:08:57	0:10:08	0:01:29	(6x)	0:00:26	(21x)	0:00:13	(42x)	0:00:19	(28x)
8	0:38:18	0:43:01	0:06:18	(6x)	0:01:24	(27x)	0:00:28	(83x)	0:00:23	(99x)
9	2:44:12	3:04:37	0:26:18	(6x)	0:06:04	(27x)	0:01:36	(102x)	0:00:24	(405x)
10	TIMEOUT	TIMEOUT	1:41:09		0:25:02		0:06:14		0:01:34	

(b) Bounded Exhaustive Testing for Binary Trees

Table I: Performance data for using parallel symbolic execution.



(a)

(b)

Figure 4: Plot of speedups versus number of processors.

testing of binary trees. We used Lonestar, a Linux cluster containing Xeon 2.66GHz processors with more than 5000 cores and an InfiniBand interconnect, graciously provided by Texas Advanced Computing Center (TACC). We tested our executions on up to 512 processors (maximum allowance).

GREP 2.2 is a 15K lines of C code application and sufficiently complex due to processing of regular expressions and other search patterns on the input. We provided it with 10 symbolic characters to be searched within a string of 40 symbolic characters [3].

Bounded exhaustive testing generates all valid test inputs within given bounds. Specialized solver Korat [2] enables bounded exhaustive testing for complex structures. To perform this test, we added support for pointers to the open source CREST tool and generated all valid binary trees from size 5 to 10. Parallel symbolic execution enables bounded exhaustive testing to process larger bounds and thus further increase the confidence on the program under test.

Performance results for all our test subjects are given in Table I. We observe that high speedups are achieved. We achieved up to 135x speedup for GREP 2.2 whereas an even higher 405x for bounded exhaustive testing. By looking at the last line in both tables, we can see that this would give an even higher speedup, but it cannot be measured due to TIMEOUT on serial execution. TIMEOUT is set to 5 hours. For generation of binary trees of 10 nodes, 512 processors finished the work in 90 seconds whereas serial processor timed out in 5 hours. We also observe that efficiency increases with problem size. For 8 processors, it ranged between 3–6x (7x maximum possible as there are 7 agents) and for 32 processors it ranged between 17–27x except for the smallest problems.

We plot number of processors versus speedup on a logarithmic scale. The plots for both problems are shown in Figure 4. The grey area represents super-linear speedup. Since both scales are logarithmic, processors (1, 8, 32, 128, and 512) and their speedup are nicely spaced. The key information from this graph is that scalability improves with problem size. Smaller problem can deteriorate in performance when given too many processors. Also note that the best performance is close to linear. This means that given a big enough problem, we utilize the resources efficiently.

IV. RELATED WORK

The idea of symbolic execution dates back to over three decades [15]. Renewed interest in symbolic execution is seen in the last decade [4], [7], [9]. Generalized Symbolic Execution [14] extended the concept to concurrent programs and complex structures.

Symbolic execution for large or complex units is difficult due to solving complex constraints for test generation. Larson and Austin [17] combined symbolic execution with concrete execution to overcome this limitation. They used symbolic execution to make the path constraint of a concrete execution and find other input values that lead to errors along the same path.

DART (Directed Automated Random Testing) [10] is one of the first tools to systematically combine symbolic execution and concrete execution. After forming a path constraint during concrete execution, they backtrack on the path constraint by negating clauses, solve the new constraints, and re-run concrete execution expecting it to follow a new path. When it is not feasible to solve the modified constraints, they substitute random concrete values. Another simultaneous effort was EGT (Execution Guided Test Cases) [6] using a similar approach. Lastly, CUTE (Concolic Unit Testing Engine for C) [23], another tool using similar approach, can handle pointers and complex structures. Another symbolic execution tool CREST [3] was introduced for comparing various search strategies. This is the tool we are basing our implementation upon.

In our own previous work, we parallelized the Korat algorithm [20], [24]. Parallel model checkers have also been introduced. Stern and Dill's parallel Murϕ [25] is an example of a parallel model checker. A similar technique was used by Lerda and Visser [26] to parallelize the Java PathFinder model checker [19]. Parallel version of the SPIN model checker [12] was produced by Lerda and Sisto [18]. More work has been done in load balancing and reducing worker communication in these algorithms [22], [16], [13].

Parallel Randomized State Space Search for JPF by Dwyer et al. [8] takes a different approach with workers exploring randomly different parts of the state space.

V. Conclusions

We presented ParSym a novel algorithm for parallel symbolic execution. We discussed the potential of parallelization in symbolic execution even for small problems. We exploited this parallelism in our parallel symbolic execution algorithm. We evaluated our algorithm on widely used GREP tool which contains 15K lines of C code. We also tested it for generation of complex structures. We tested it on a cluster and observed high speedups. We observed very high efficiency on up to 32 processors for all but the smallest of problems. We were able to finish tasks in a couple minutes using 512 processors whereas serially it was taking hours. We believe these optimizations will allow symbolic execution to scale to many more problems and its impact on testing real software will increase.

Acknowledgments

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] C. Barrett and C. Tinelli, "CVC3," in *Proc. 19th Int. Conf. Comput. Aided Verification (CAV)*, 2007, pp. 298–302.

[2] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing based on Java Predicates," in *Proc. 2002 Int. Symp. Softw. Testing and Analysis (ISSTA)*, 2002, pp. 123–133.

[3] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *Proc. 23rd Int. Conf. Automated Softw. Eng. (ASE)*, 2008, pp. 443–446.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. 8th Symp. Operating Syst. Design and Implementation (OSDI)*, 2008, pp. 209–224.

[6] C. Cadar *et al.*, "EXE: Automatically Generating Inputs of Death," in *Proc. 13th Conf. Comput. and Commun. Security (CCS)*, 2006, pp. 322–335.

[7] A. Coen-Porisini *et al.*, "Using Symbolic Execution for Verifying Safety-critical Systems," in *Proc. 3rd joint meeting of the Euro. Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE)*, 2001, pp. 142–151.

[8] M. B. Dwyer *et al.*, "Parallel Randomized State-Space Search," in *Proc. 2007 Int. Conf. Softw. Eng. (ICSE)*, 2007, pp. 3–12.

[9] C. Flanagan *et al.*, "Extended Static Checking for Java," in *Proc. 2002 Conf. Prog. Lang. Design and Implementation (PLDI)*, 2002, pp. 234–245.

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proc. 2005 Conf. Prog. Lang. Design and Implementation (PLDI)*, 2005, pp. 213–223.

[11] A. Grama *et al.*, *Introduction to Parallel Computing*, 2nd ed. Addison Wesley, 2003.

[12] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[13] M. D. Jones and J. Sorber, "Parallel Search for LTL Violations," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 1, pp. 31–42, Feb. 2005.

[14] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Proc. 9th Int. Conf. Tools and Algorithms for the Construction and Analysis of Syst. (TACAS)*, 2003, pp. 553–568.

[15] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[16] R. Kumar and E. G. Mercer, "Load Balancing Parallel Explicit State Model Checking," *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 3, pp. 19–34, Apr. 2005.

[17] E. Larson and T. Austin, "High Coverage Detection of Input-related Security Faults," in *Proc. 12th USENIX Security Symp.*, 2003, p. 9.

[18] F. Lerda and R. Sisto, "Distributed-Memory Model Checking with SPIN," in *Proc. 5th Int. SPIN Workshop onModel Checking of Softw.*, 1999, pp. 22–39.

[19] F. Lerda and W. Visser, "Addressing Dynamic Issues of Program Model Checking," in *Proc. 8th Int. SPIN Workshop onModel Checking of Softw.*, 2001, pp. 80–102.

[20] S. Misailovic *et al.*, "Parallel Test Generation and Execution with Korat," in *Proc. 6th joint meeting of the Euro. Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE)*, 2007, pp. 135–144.

[21] G. C. Necula *et al.*, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proc. 11th Int. Conf. Compiler Construction (CC)*, 2002, pp. 213–228.

[22] R. Palmer and G. Gopalakrishnan, "A Distributed Partial Order Reduction Algorithm," in *Proc. 22nd Int. Conf. Formal Techniques for Networked and Distributed Syst. (FORTE)*, 2002, p. 370.

[23] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proc. 5th joint meeting of the Euro. Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE)*, 2005, pp. 263–272.

[24] J. H. Siddiqui and S. Khurshid, "PKorat: Parallel Generation of Structurally Complex Test Inputs," in *Proc. 2nd Int. Conf. Softw. Testing Verification and Validation (ICST)*, 2009, pp. 250–259.

[25] U. Stern and D. L. Dill, "Parallelizing the Murphi Verifier," in *Proc. 9th Int. Conf. Comput. Aided Verification*, 1997, pp. 256–278.

[26] W. Visser *et al.*, "Model Checking Programs," *Automated Softw. Eng. J.*, vol. 10, no. 2, pp. 203–232, Apr. 2003.