A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation

Moonzoo Kim, Yunho Kim
Department of Computer Science
KAIST, South Korea
moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr

Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
grother@cse.unl.edu

***Abstract*—Although testing is a standard method for improving the quality of software, conventional testing methods often fail to detect faults. Concolic testing attempts to remedy this by automatically generating test cases to explore execution paths in a program under test, helping testers achieve greater coverage of program behavior in a more automated fashion. Concolic testing, however, consumes a significant amount of computing time to explore execution paths, which is an obstacle toward its practical application. To address this limitation, we have developed a scalable distributed concolic testing framework that utilizes large numbers of computing nodes to generate test cases in a scalable manner. In this paper, we present the results of an empirical study that shows that the proposed framework can achieve a several orders-of-magnitude increase in test case generation speed compared to the original concolic approach, and also demonstrates clear potential for scalability.**

I. Introduction

Dynamic testing is a de-facto standard method for improving the quality of software in industry. Conventional testing methods, however, often fail to detect faults in programs. One reason for this is that a program can have an enormous number of different execution paths due to conditional and loop statements. Thus, it is practically infeasible for a test engineer to manually create test cases sufficient to detect subtle bugs in specific execution paths. In addition, it is a technically challenging task to generate test cases that cover different paths in an automated manner.

To address such limitations, concolic (CONCrete + symbOLIC) testing [1] (also known as dynamic symbolic execution [2] and white-box fuzzing [3]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore execution paths of a program, to achieve full path coverage (or at least, coverage of paths up to some bound). However, concolic testing may consume a significant amount of time exploring execution paths, and this is an obstacle toward its practical application [4].

To address this limitation, we have developed the Scalable COncolic testing for REliable software (SCORE) framework [5]. The SCORE framework employs a distributed concolic algorithm that can utilize a large number of computing nodes in a scalable manner so as to achieve:

- a linear increase in the speed of test case generation as the number of distributed nodes increases;
- low communication overhead among distributed nodes.

In this paper, we present the SCORE framework and the results of an empirical study that shows that the SCORE framework can achieve a several orders-of-magnitude increase in test case generation speed compared to the original concolic approach, and also demonstrates clear potential for scalability. To investigate the effectiveness (with regard to generation of test cases that cover different paths in a given time) and scalability of the SCORE framework, we conducted a controlled experiment in which we applied the framework to six C programs in the SIR benchmark suite [6] (three of them large real-world applications), using numbers of Amazon EC2 nodes ranging up to 256. Our results show that SCORE can greatly increase the effectiveness of test case generation, and that it is scalable as the numbers of nodes utilized increases.

The rest of the paper is organized as follows. Section II describes related work. Section III describes the SCORE framework. Section IV presents our empirical study, and Section V discusses observations from the study. Section VI concludes and discusses future work.

II. Related Work

The core idea behind concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers. Various concolic testing tools have been implemented to realize this core idea (see Pasareanu et al. [7] for a survey). Existing tools can be classified in terms of the approach they use to extract symbolic path formulas from concrete executions.

The first approach for extracting symbolic path formulas is to use modified virtual machines. The concolic testing tools that use this approach are implemented as modified virtual machines on which target programs execute. An advantage of this approach is that the tools can exploit all execution information at run-time, since the virtual machine possesses all necessary information. PEX [2] targets C# programs that are compiled into Microsoft .Net binaries, KLEE [8] targets LLVM [9] binaries, and jFuzz [10] targets Java bytecode on top of Java PathFinder [11], [12].

The second approach for extracting symbolic path formulas is to instrument the target source code to insert probes that extract formulas from concrete executions at run-time. Compared to the first approach, this approach is

more light-weight, because adding probes is simpler than modifying virtual machines. Tools that use this approach include CUTE [1], DART [13], and CREST [14], which operate on C programs, and jCUTE [15], which operates on Java programs. SCORE uses this approach.

Because concolic testing has been studied for a relatively short period of time, there has been little research on employing distributed platforms to improve the scalability of concolic testing techniques. Staats et al. [16] propose a *static partitioning* technique for parallelizing symbolic execution that uses pre-conditions/prefixes of symbolic executions to partition the symbolic execution tree. They have implemented the technique on top of Java PathFinder [11] using the Symbolic PathFinder extension [12]. A limitation of this approach is that the resulting partitioned symbolic execution trees are *not well-balanced*, because the technique statically partitions a symbolic execution tree based only on its prefixes. Thus, some nodes may finish exploring symbolic execution paths quickly and become idle while other nodes take long times to complete exploration, which degrades overall performance.

In contrast, King [17], ParSym [18], and Cloud9 [19] utilize *dynamic partitioning* of target program executions. King's master's thesis [17] describes a distributed symbolic execution framework for Java [20]. King populates a queue of symbolic execution subtrees dynamically, but the resulting speedup decreases as the number of nodes increases beyond six. ParSym [18] uses a central server that collects test cases generated from nodes and distributes the test cases to the other nodes whose queues are empty. ParSym demonstrates speedup on `grep` 2.2 and a binary tree program on up to 512 nodes, but does not achieve linear speedup. Cloud9 [19] is a testing service framework based on parallel symbolic execution techniques implemented on KLEE. Cloud9 uses dynamic partitioning to ensure that the job queue lengths of all nodes stay within a given range and shows linear speed-up as the number of nodes increases up to 48. Similar to these techniques, SCORE distributes test cases among multiple nodes in a dynamic on-demand manner (Section III) and achieves linear speed-up on up to 256 nodes (Section IV-E). Note that Cloud9 and SCORE utilize different parallelization techniques. In Cloud9, when an execution meets a branch containing symbolic values, two parallel executions are *forked* with a corresponding clone of the program state and distributed to nodes to continue (Cloud9 can obtain parallel executions since Cloud9 operates as a virtual machine). In contrast, SCORE generates whole execution paths one by one on distributed nodes in a systematic manner while preventing redundant test cases (Algorithm 1).

III. THE SCORE FRAMEWORK

The goal of the SCORE framework is to address the limitation of scalability problem in the original sequential concolic testing framework by parallelization. We describe the SCORE framework, beginning with a description of the distributed concolic algorithm (Section III-A), followed by details on communications between nodes (Section III-B). Section III-C then discusses our implementation of the framework.

A. Distributed Concolic Algorithm

Algorithm 1 [21] describes how the SCORE framework generates test cases on each node n among a set of distributed nodes. The algorithm controls each node n to generate *non-redundant* test cases (i.e., test cases that cover different paths) *independently* from the other nodes; n communicates with another node n' only when n cannot generate test cases due to having an empty job queue. Note that this independent generation of test cases on each node is a key condition for achieving *linear speedup* of test case generation with scalability.

We assume that there exists one startup node that runs $DstrConcolic(startup)$ with $startup$ as true (line 1). This startup node running $DstrConcolic()$ generates and stores a *test case pair* (tc,neg_limit) in the test case pair queue q_{tcp} (lines 5-6), where neg_limit is used to prevent the algorithm from generating redundant test cases based on a test case tc. For the startup node, tc is a randomly generated test case and $neg_limit = 1$. The other non-startup nodes running $DstrConcolic(false)$ wait until they receive test case pairs from another node n' (lines 8-9).[1]

Next, a node n generates further test case pairs (lines 20-30) based on a (tc,neg_limit) that is removed from q_{tcp} (line 14) and stores the generated test case pairs in q_{tcp} (line 26). Node n repeats this process until q_{tcp} becomes empty (lines 13-31).[2] The detail of the process is as follows. First, the algorithm removes (tc,neg_limit) from q_{tcp} (line 14) and obtains a symbolic path formula ϕ (line 18) from the concrete execution on tc (line 16). Then, the algorithm generates further symbolic path formulas ψs by negating path conditions (i.e., $c_1, c_2, ..., c_{|\phi|}$) in ϕ one by one from $c_{|\phi|}$ to $c_{|neg_limit|}$ in decreasing order through the while loop (lines 20-30). Note that these ψs indicate new execution paths to explore. Each ψ is solved by an SMT solver (line 24) and the corresponding solution to ψ is stored in q_{tcp} as a new test case pair $(tc, j+1)$ (line 26).

If q_{tcp} is empty (exiting the loop of lines 13-31) and the q_{tcp}s of all distributed nodes are empty, the algorithm terminates (line 37). Otherwise (i.e., there is another node n''

[1] For the sake of simplicity, in Algorithm 1 the communication between nodes is described abstractly or omitted. For example, at lines 8-9 of Algorithm 1, n in fact sends a request to the central server. Then, the server finds an appropriate n' and asks n' to send test case pairs to n. Details of this communication are given in Section III-B.

[2] q_{tcp} can be empty, because n may be unable to generate further test case pairs from (tc, neg_limit) such that $neg_limit >| \phi |$ or none of the negated symbolic path formula ψs generated from ϕ is satisfiable, where ϕ is a symbolic path formula of an execution path on tc. However, q_{tcp} rarely becomes empty, since n usually generates multiple test case pairs (lines 20-30) based on a test case pair in q_{tcp} (line 14).

that has test case pairs), a current node n requests test case pairs from n'' (line 33) and receives test case pairs from n'' (line 34). The received test case pairs are then added into q_{tcp} (line 35) and the algorithm continues from line 13 again.

Note that Algorithm 1 traverses all possible execution paths and does *not* generate redundant test cases (test cases that cover the same path) with the assumption that ϕ truly reflects $path$ and $Solve(\psi)$ can solve ψ.[3] Complete details on Algorithm 1, including a comparison with the original concolic algorithm and the properties of the algorithm such as traversal of all possible execution paths and its unique test case generation, are provided in a [22].

B. Communication between Nodes

Figure 1 illustrates the communication among nodes in the SCORE framework. SCORE operates on distributed computing nodes where one node operates as a server and the other nodes operate as clients. In addition, one client is designated as a startup client, which initiates distributed concolic testing (lines 4-6 in Algorithm 1).

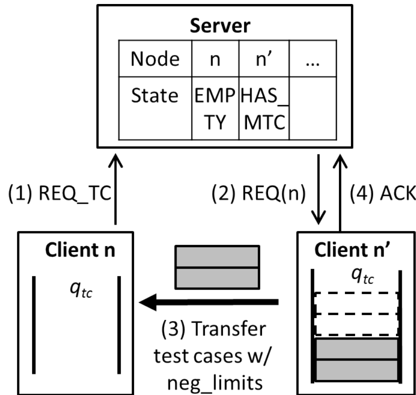

Figure 1. Communication among clients and the server

Input:
$startup$: a flag to indicate whether a current node n is a startup node or not.
Output:
TC: a set of test cases generated at a current node n (i.e., tcs of line 24)

1 $DstrConcolic(startup)$ {
2 $q_{tcp} = \emptyset$; // a queue containing (tc, neg_limit)s
3 $TC = \emptyset$; // a set of generated test cases
4 **if** *startup* **then**
5 $\quad$ tc = random value; // initial test case
6 $\quad$ Add $(tc, 1)$ to q_{tcp};
7 **else**
8 $\quad$ Send a request for test cases to another node n';
9 $\quad$ Receive (tc, neg_limit)s from n';
10 $\quad$ Add (tc, neg_limit)s to q_{tcp};
11 **end**
12 **while** *true* **do**
13 $\quad$ **while** $| q_{tcp} | > 0$ **do**
14 $\quad\quad$ Remove (tc, neg_limit) from q_{tcp};
15 $\quad\quad$ // Execute target program on tc
16 $\quad\quad$ $path$ = an execution path on tc;
17 $\quad\quad$ // Obtain a symbolic path formula ϕ
18 $\quad\quad$ ϕ = a symbolic path formula of $path$ (i.e., $c_1 \wedge c_2 \wedge ... \wedge c_{|\phi|}$) ;
19 $\quad\quad$ $j = | \phi |$;
20 $\quad\quad$ **while** $j >= neg_limit$ **do**
21 $\quad\quad\quad$ // Generate ψ for the next input values
22 $\quad\quad\quad$ $\psi = c_1 \wedge ... \wedge c_{j-1} \wedge \neg c_j$;
23 $\quad\quad\quad$ // Select the next input values
24 $\quad\quad\quad$ $tc = Solve(\psi)$;
25 $\quad\quad\quad$ **if** tc is not NULL **then**
26 $\quad\quad\quad\quad$ Add $(tc, j + 1)$ to q_{tcp};
27 $\quad\quad\quad\quad$ $TC = TC \cup \{tc\}$;
28 $\quad\quad\quad$ **end**
29 $\quad\quad\quad$ $j = j - 1$;
30 $\quad\quad$ **end**
31 $\quad$ **end**
32 $\quad$ **if** *there is a test case in another node* n'' **then**
33 $\quad\quad$ Send a request for test cases to another node n'';
34 $\quad\quad$ Receive (tc, neg_limit)s from n'';
35 $\quad\quad$ Add (tc, neg_limit)s to q_{tcp};
36 $\quad$ **else**
37 $\quad\quad$ Halt; // no test cases exist in all nodes
38 $\quad$ **end**
39 **end**
40 }

Algorithm 1: Distributed concolic algorithm

We assume that all clients are connected to the server properly and the network does not lose or corrupt its messages or change message delivery order. There are six different control packets sent between a server and clients to coordinate distributed clients that generate test cases.

- An `REQ_TC` packet is sent to the server from a client n that has no test case pair in its q_{tcp}.
- An `REQ` packet is sent from the server to a client n' that has test case pairs and that is not currently serving any `REQ` packet.
- An `ACK` packet is sent from an n' that received an `REQ` packet and has transferred test case pairs to an n that sent an `REQ_TC` packet to the server. In a similar manner, an `ACK` packet is sent from an n that sent an

[3]In practice, a program P may contain complex arithmetic or binary library calls that cannot be reasoned about by SMT solvers. Thus, Algorithm 1 generates symbolic path formulas without such conditions, and in these cases this may result in redundant test cases.

REQ_TC to the server after n receives test case pairs.

- A NACK packet is sent from an n' that has received an REQ packet and could not transfer test case pairs, since $|q_{tcp}| \leq 1$.
- A HAS_MTC_AGAIN packet is sent from an n' that sent NACK but has $|q_{tcp}| > 1$ now.
- A STOP packet is sent from the server to clients when every client has no test case pair (i.e., the distributed concolic testing process is completed).

To identify which client has test case pairs to transfer, the server maintains a state table of all clients. From the viewpoint of the server, the state of a client must be one of the following four states:

1) EMPTY when $| q_{tcp} |= 0$. EMPTY indicates that the client has no test case pairs to execute.
2) HAS_1TC when $| q_{tcp} |= 1$ and the client is not serving a request from the server. HAS_1TC indicates that the client cannot provide test case pairs to another client.
3) HAS_MTC when $| q_{tcp} |> 1$ and the client is not serving a request from the server. HAS_MTC indicates that the client is ready to provide test case pairs to another client, since the client has multiple test cases.
4) SERVING_REQ when the client is serving a request from the server. In other words, the client received an REQ but has not yet sent an ACK or NACK to the server. SERVING_REQ indicates that the client is not currently able to send test case pairs to another client.

By tracking control messages between the server and the client, the server knows the states of clients. If the server receives an REQ_TC from a client n, then the state of n is considered EMPTY. If the server receives an ACK from a client n' that sent test case pairs to n, then the server considers the state of n' to be HAS_MTC. In a similar manner, if the server receives an ACK from a client n, then the server considers the state of n to be HAS_MTC. If the server receives a NACK from a client n' whose state is HAS_MTC, then the server considers the state of n' to be HAS_1TC. If the server receives an HAS_MTC_AGAIN from a client n', then the state of n' is considered HAS_MTC. If the server has sent an REQ to a client and has not yet received an ACK or NACK, then the state of the client is SERVING_REQ.

Figure 1 illustrates the following scenario. When the server receives an REQ_TC from a client n, it searches the client status table and finds a client n' whose status is HAS_MTC. Then, it sends an REQ to n' with destination n. If n' has m test case pairs (i.e., $| q_{tcp} |= m$), it sends $\lfloor m/2 \rfloor$ test case pairs to n and then sends an ACK to the server. Client n also sends an ACK to the server after it receives test case pairs. If client n' has one or zero test case pairs, it sends NACK to the server. Then, the server tries another client whose status is HAS_MTC in a round-robin manner. If client n' that sent NACK has more than one test case pair now, n' sends HAS_MTC_AGAIN to the server and the server updates the state of n' as HAS_MTC. If no client is in HAS_MTC state, the server waits until at least one client enters the HAS_MTC state. If all clients are EMPTY, the server sends a STOP message to the clients, and the distributed concolic algorithm terminates. When clients receive a STOP from the server, the clients transfer generated test cases, covered path/branch information, and statistics on testing activities to the server.

Note that communication between clients n and n' occurs only when q_{tcp} of n is empty. Because q_{tcp} is non-empty for most of the testing time, the number of communications is small compared to the number of test cases generated (Table V). Furthermore, test case pairs are directly transferred between clients without causing heavy load on the server. Otherwise, the server would become a bottleneck. This efficient communication protocol is another key condition for achieving linear speedup with scalability.

C. The SCORE Implementation

The SCORE framework is implemented to operate on distributed computers connected through TCP/IP networks. Thus, SCORE can operate on a large number of computing nodes such as on a cloud computing platform or any computers connected through the internet. The SCORE framework uses an extended version of CREST with bit-vector (BV) support. The original CREST [14] supports only linear-integer arithmetic (LIA) formulas so that non-linear arithmetic operations in a target program cannot be analyzed symbolically. To overcome this limitation, we have developed CREST-BV based on CREST 0.1.1 to support bit-vector symbolic path formulas by using the Z3 2.19 SMT solver [23]. SCORE is written in C/C++ and contains 7600 lines of code with 24 classes and 265 functions [5].

For m clients, the server creates m threads, each of which communicates with one client. To minimize communication overhead, each client is implemented as two separate threads. One thread generates and consumes test case pairs in q_{tcp} as described in Algorithm 1 while the other thread handles communication with other nodes such as receiving test case pairs into q_{tcp} or sending test case pairs from q_{tcp} to another node per request. All communications in the SCORE framework are implemented using TCP sockets, since the framework may be deployed on a large scale computing platform where communication might not be reliable. In addition to the communication between clients and the server described in Section III-B, as logs, clients periodically report to the server the current status of testing activities such as the number of test cases generated, the size of q_{tcp}, the number of test cases received from another node, and so forth. Each client stores testing outcomes such as test cases generated, covered branches, and covered execution paths on

the local hard disk. When the testing process terminates or a user sends a command to stop the concolic testing, these outcomes are collected by the server automatically.

IV. Empirical Study

There are two overall methodologies that we could use to empirically study the SCORE framework. The first methodology involves empirically comparing SCORE to other approaches for parallelizing dynamic symbolic execution such as those of Staats et al. [16], King [17], ParSym [18] and Cloud9 [19], discussed in Section II. At this time, however, such a comparison would be difficult, because Staats' approach is implemented only for Java, and the other approaches were not available in implemented form at the time of the study. Moreover, comparisons of implementations created in different contexts pose many threats to internal validity in terms of ensuring the comparability of techniques.

The second methodology we can use to assess SCORE involves comparing the framework to baseline approaches that allow direct assessment of the benefits of SCORE's distributed aspects. For example, we can compare SCORE to the original concolic testing approach applied on distributed nodes. If this comparison does *not* show that SCORE is effective and scalable, then the results obviate the need to perform expensive implementations of other techniques. We thus chose this approach.

The SCORE framework is meant to increase the *effectiveness* of concolic testing at generating potentially useful test inputs by distributing workload over large numbers of client nodes. However, the efficiency of distributed algorithms tends to decrease as the number of client nodes increases due to redundant computations, overhead related to increasing communications, and unbalanced workloads. These issues impact the *scalability* of these algorithms. The degree to which the framework can achieve these attributes on real workloads, however, must be assessed empirically.

To provide such an assessment, we designed an empirical study addressing the following research questions:

- **RQ1**: To what extent does the SCORE framework increase the effectiveness of test case generation?
- **RQ2**: To what extent does the SCORE framework achieve scalability?

A. Target Benchmark Programs

As objects of study, we selected six programs (see Table I) from the SIR repository [6], including three of the Siemens programs [24], and three non-trivial real-world programs (`grep` 2.0, `sed` 1.17, and `vim` 5.0). We selected these programs because they were written in C and thus can be processed by our tools, and because they do not utilize intensive numbers of floating point arithmetic statements that cannot be analyzed by bit-vector SMT solvers.

Table I
Experiment Objects

Program	Functions	LOC	Branches	Test cases
`grep`	126	12562	3768	808
`ptok1`	19	725	284	4140
`ptok2`	24	569	168	4140
`repl`	2	563	210	5543
`sed`	70	8678	2690	389
`vim`	3049	111227	33486	975

To perform concolic testing on these programs, we needed to decide what sizes of inputs to utilize for symbolic variables. To make a realistic decision, we reviewed all of the test cases provided for each program in the SIR repository. We selected symbolic input sizes as the maximum size of the 90% of the smallest test cases in the SIR repository, since there is often a large gap between that size and the sizes of the remaining 10% of the test cases. Table II shows the sizes of symbolic inputs chosen for each program.

Table II
Size of Symbolic Inputs (Bytes)

`grep`	`ptok1`	`ptok2`	`repl`			`sed`	`vim`
pattern	target text	target text	from	to	target text	com-mand	script
21	82	82	23	28	64	148	76

We provided two symbolic options for `grep`, because `grep` uses three different algorithms for pattern matching based on a given option. Also, 90% of the test cases for `grep` in the SIR repository contain two or fewer options; for an option with an argument, we assigned one symbolic character as a corresponding argument. Finally, for `grep` and `sed`, we chose to use target text files provided in the SIR repository as concrete inputs rather than to generate file contents symbolically, because meaningful files are too large (greater than 100k) to treat as symbolic input.

B. Variables and Measures

To address our research questions, our experiment manipulated two independent variables:

IV1: Test case generation technique
To investigate RQ1 we study two techniques: the distributed algorithm implemented in the SCORE framework, and the non-distributed concolic testing algorithm [14]. Further, to facilitate comparisons we apply the non-distributed (baseline) algorithm in two ways: (1) running a single instance of the algorithm on a single node, and (2) running multiple instances, one on each client node, with different random seeds, and aggregating results from each node in the end. The former approach lets us assess the effectiveness of SCORE relative to current practice, while the latter lets us assess whether the particular parallelization solution

used by SCORE is more effective than the naive approach of simply running the original algorithm on the same number of nodes.

IV2: Client number level
To investigate scalability issues such as those posed by RQ2, as well as to examine technique effectiveness over a wide range of distribution settings, we need to apply techniques using different numbers of client nodes. We chose to use client number levels 1, 64, 128, 192, and 256.

To measure effectiveness and scalability we selected five dependent variables. The first variable tracks effectiveness in terms of numbers of test cases that can be generated in a given time. An issue that arises in this context, however, concerns redundancy. As explained earlier, in the context of concolic testing, where test cases are generated to cover paths, test cases that cover the same paths are redundant. In theory, neither the SCORE algorithm nor the original concolic algorithm can generate redundant test cases [22]. However, limitations in concolic algorithms or limitations in symbolic execution engines can lead in practice to cases where redundancy does occur. Since the different techniques that we compare may differ in terms of the number of redundant test cases that they create, a fair comparison of their effectiveness must exclude redundant test cases. Thus, our first dependent variable focuses on creation of non-redundant test cases.

DV1: Number of non-redundant test cases generated in time τ
We chose $\tau = 5$ minutes, because exploratory studies with smaller and larger times suggested that increases beyond 5 minutes had negligible effects on results.

To track scalability we use four variables, each of which represents a different important aspect of performance in the context of distributed algorithms.

DV2: Number of communicated messages
We measured the total number of messages communicated between nodes (both server and client).

DV3: Communication overhead in terms of waiting time
We measured the elapsed waiting time between lines 33 and 35 of Algorithm 1 due to empty q_{tc}, for each client node.

DV4: Workload assigned to each client
We measured the number of test cases generated by each client node.

DV5: Efficiency of the SCORE framework
We measured the efficiency of the framework by calculating its effectiveness ratios (i.e., # of test cases generated by SCORE over # of test cases generated by the non-distributed concolic algorithm) over the number of clients, $\frac{\text{effectiveness ratio}}{\text{\# of clients nodes}}$.

C. Experiment Operation

For each target program, we executed the non-distributed CREST-BV algorithm and the distributed SCORE algorithm on each of the five client number levels. To control for potential differences in runs due to the randomization inherent in the techniques, we repeated this process 30 times.

To count the non-redundant test cases in a generated test suite, we stored the sequence of branch IDs $B_i = [b_{i_1}, b_{i_2}, ...b_{i_n}]$ that were executed on each test case tc_i. Then, because the total number and sizes of the B_is are large (e.g., we had more than 0.2 million B_is consuming 500 gigabytes for VIM on 256 nodes (see Table III)) we applied the SHA1 algorithm [25] to each (B_i) to obtain a representative hash value for each B_i. In our experiment runs, the number of hash collisions was negligible. As a sample, we compared all B_is directly in one entire set of technique runs on 256 nodes, and found no hash value collisions. Finally, we compared SHA1(B_i) values with each other to remove redundant test cases, keeping exactly one test case from each set of test cases that execute identical paths.

All experiments were performed on the Amazon EC2 cloud computing platform. The server of the SCORE framework ran on a virtual node that had seven gigabytes of memory and eight CPU cores of 20 ECU computing power in total (1 ECU is equivalent to a 1Ghz Xeon processor). Each client ran on a virtual node that was equipped with 1.7 gigabytes of memory and one CPU core of 1 ECU in total. The server and clients ran on Fedora Core Linux 8. All virtual nodes are connected through a 1 gigabps Ethernet.

D. Threats to Validity

The primary threat to external validity for our study involves the representativeness of our object programs, since we have examined only six C programs (although three of them are real-world applications). Furthermore, we have chosen programs that are amenable to concolic testing, and thus, do not reveal cases in which program characteristics might hinder that approach. As a second threat, we have employed only the CREST-BV tool as an example of an original concolic algorithm implementation; results obtained with other implementations may differ. A third threat to validity is the limited power of the underlying symbolic execution engine used in SCORE. For example, SCORE does not analyze floating point arithmetic operations, since most SMT solvers do not support them. Also, external binary libraries used by a target program cannot be analyzed. We believe, however, that the use of a more powerful SMT solver in the future would not affect assessments of the scalability of SCORE, because SCORE's scalability is affected primarily by the distribution of independent workloads and communication cost, and SCORE's communication protocol is independent of the SMT solver (see Section III-B).

The primary threat to internal validity is possible faults in the implementation of our algorithms and in tools we use to collect metrics. We controlled for this threat through extensive functional testing of our tools. A second threat pertains to differences in the implementations compared; we limited this threat by using the same underlying concolic test case generation tool in all cases.

Where construct validity is concerned, there are other metrics that could be pertinent to the effects studied. In particular, as an effectiveness measure we consider only numbers of non-redundant test cases generated (which correlates with path coverage achieved). In contrast, studies of Cloud9 [19] relied on statement coverage as an effectiveness measure. We believe, however, that our effectiveness measure is more appropriate for assessing concolic testing, since concolic testing targets path coverage, not statement or branch coverage. As another potential metric for assessing effectiveness, numbers of faults detected could also be considered. Further, as another potential metric for assessing scalability to measure scalability, the time required to reach a fixed level of statement/branch coverage or the time consumed to explore an execution subtree in a k-depth bound completely could be considered. These metrics, however, have weaknesses. First, concolic testing may not be able to generate test cases to reach a given fixed level of coverage. In addition, it is difficult to accurately estimate the time required to reach a fixed level of coverage in advance, causing difficulty for experiment design. Yet another metric for scalability in terms of workload distribution, also employed in [19], is to measure the number of target program instructions executed per node. This might be a valid performance indicator for parallel algorithms in general, but when the goal of such algorithms is test case generation, we believe that it is not the best measure.

E. Results and Analysis

We now present and analyze our results.

1) RQ1: To what extent does SCORE increase the effectiveness of concolic testing?: We begin by comparing SCORE to the non-distributed algorithm run on a single node. Table III displays the mean total numbers of non-redundant test cases generated by both the non-distributed and distributed algorithms, for each of the object programs, per client number level, across all 30 runs of the algorithm at that level. As the table shows, the number of test cases generated by SCORE increased substantially as the number of client nodes increased.

Figure 2 illustrates the effectiveness (i.e., number of non-redundant test cases generated) increase achieved by SCORE as the client number level increased, in a manner that compares the two algorithms. The figure compares effectiveness results obtained by the distributed algorithm at all five client number levels to the results obtained by the non-distributed algorithm on a single node, per program, in terms of the ratio of numbers of test cases generated by each. Effectiveness appears to increase *linearly* with client number level, but the rate of increase does vary per program.

Table III
TOTAL # OF NON-REDUNDANT TESTS GENERATED PER CLIENT NUMBER LEVEL, NON-DISTRIBUTED AND DISTRIBUTED ALGORITHMS

	CREST-BV	SCORE				
	1	1	64	128	192	256
`grep`	441	474	21684	51485	85707	128337
`ptok1`	354	356	31566	76794	106332	127667
`ptok2`	3350	3377	243330	509142	759004	959633
`repl`	5310	5505	384588	742251	1156691	1547508
`sed`	1189	1201	92165	191845	303615	377621
`vim`	886	817	60276	116448	177964	217460

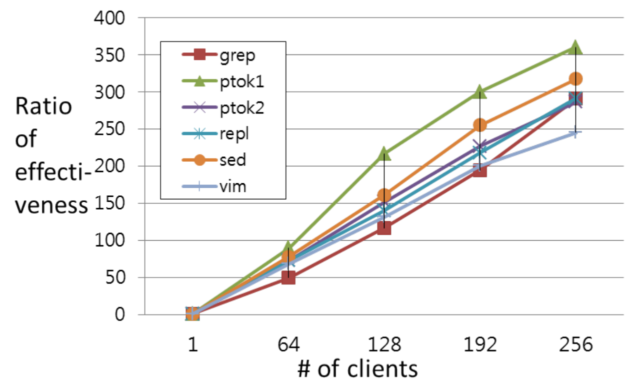

Figure 2. Ratio of effectiveness between non-distributed and distributed algorithms, per client number level

To assess whether the observed differences in performance were statistically significantly different, we applied Wilcoxon tests [26] to the effectiveness data achieved at each client number level, per program, by SCORE and the non-distributed algorithm, with $\alpha = 0.05$ as confidence level. In every case above client number level 1 the differences were statistically significant.

It is worth noting that the effectiveness ratios for all the target programs except `vim` at client number level 256 are greater than the number of client nodes. We believe that the reason for this is that the average length of the symbolic path formulas analyzed to generate test cases by one client node using the non-distributed algorithm was larger than the average length of the symbolic path formulas analyzed to generate the test cases by 256 client nodes using the distributed algorithm. The relatively low effectiveness ratio (i.e., 245.4 = 217460/886) for `vim` at client number level 256 can be explained similarly.

Note also that the numbers of test cases generated by the non-distributed algorithm and SCORE did differ when the algorithms were each run on one node (see the second and third columns of Table III), because the test cases generated by the algorithms on any given run can differ. However,

the magnitude of the difference in performance is relatively small, ranging from 0.6% (on ptok1) to 7.8% (on vim).

We next compare SCORE to the use of the non-distributed algorithm on the same numbers of client nodes. Table IV shows the mean numbers of non-redundant test cases generated by SCORE (S) and multiple runs of CREST-BV (MC) across the 30 runs performed in each case, as well as the ratios between those means.

Table IV
NUMBERS OF NON-REDUNDANT TEST CASES

# of nodes		grep	ptok1	ptok2	repl	sed	vim
64	S	21684	31566	243330	384588	92165	60276
	MC	1937	9244	208843	111384	141264	1383
	S/MC	**11.19**	**3.41**	**1.17**	**3.45**	**0.65**	**43.58**
128	S	51485	76794	509142	742251	191845	116448
	MC	3952	17504	331865	178262	273359	1514
	S/MC	**13.03**	**4.39**	**1.53**	**4.16**	**0.70**	**76.91**
192	S	85707	106332	759004	1156691	303615	177964
	MC	5807	25182	475168	237711	400194	1524
	S/MC	**14.76**	**4.22**	**1.60**	**4.87**	**0.76**	**116.77**
256	S	128337	127667	959633	1547508	377621	217460
	MC	7626	26891	590085	305025	487802	1560
	S/MC	**16.83**	**4.75**	**1.63**	**5.07**	**0.77**	**139.40**

As the data shows, the number of test cases generated by SCORE ranged from 17% (ptok2 on 64 nodes) to 13840% (vim on 256 nodes) more than the number generated by multiple CREST-BV runs, except sed. In addition, the S/MC ratios have a tendency to increase over the number of nodes utilized. For example, for ptok1, the ratios increase from 3.41 on 64 nodes to 4.75 on 256 nodes. (The only case in which the ratios do not increase involves ptok1, going from 128 to 192). We conjecture that the reason for this S/MC increase over increasing number of client nodes is that uncontrolled multiple runs of CREST-BV will generate more redundant test cases as more test cases are generated (i.e., the probability of generating redundant test cases among 1000 test cases is higher than among 10 test cases).

Again, Wilcoxon tests applied to the effectiveness data achieved at each client number level, per program, by SCORE and by the non-distributed algorithm, using $\alpha = 0.05$ as the confidence level, showed that in every case the differences were statistically significant. SCORE thus increased effectiveness more than running multiple instances of the non-distributed concolic algorithm.

2) RQ2: To what extent does the SCORE framework achieve scalability?: Table V shows the number of communicated messages (left) and the communication overhead (right) for the distributed algorithm, for each object program, for client number levels greater than 1. The number of communications is small compared to the number of generated non-redundant test cases. For example, using 64 client nodes on grep resulted in 362 messages being communicated between client nodes and server, which is equivalent to 5.7 (362/64) messages per client node. In other words, each client node communicates 5.7 messages while generating 339 (21684/64) test cases.

Table V
STATISTICS ON COMMUNICATION

	# of total messages and the ratios of # of msgs/# of TCs				Comm. overhead (%)			
	64	128	192	256	64	128	192	256
grep	362 (1.67%)	846 (1.64%)	1070 (1.25%)	1984 (1.55%)	0.55	0.51	0.71	0.90
ptok1	412 (1.31%)	930 (1.21%)	1314 (1.24%)	1692 (1.33%)	0.16	0.26	0.33	0.30
ptok2	362 (0.15%)	664 (0.13%)	998 (0.13%)	1310 (0.14%)	0.09	0.18	0.16	0.19
repl	322 (0.08%)	618 (0.08%)	960 (0.08%)	1234 (0.08%)	0.15	0.21	0.20	0.16
sed	347 (0.38%)	692 (0.36%)	1014 (0.33%)	1357 (0.36%)	0.12	0.20	0.23	0.19
vim	344 (0.57%)	690 (0.59%)	1028 (0.58%)	1364 (0.63%)	0.45	0.77	0.56	0.63

As shown on the right side of the table, total communication overhead was less than 0.9% of total execution time. Since generating a new test case (including concolic execution and solving symbolic path formulas) takes much more time than communication, the overhead of waiting time including communication time is negligible.

Figure 3 presents boxplots showing workload distributions for all six object programs. The horizontal axes indicate client number levels, and the vertical axes indicate workload (number of test cases generated per client node). The central 50% of the data points (those denoted by boxes) exhibit relatively small variance, and results do not vary widely as client number levels increase. This provides further evidence of scalability.

Finally, Table VI depicts the efficiency (i.e., $\frac{\text{effectiveness ratio}}{\text{\# of clients}}$) of the SCORE framework across different client number levels, per program. The efficiency of the distributed algorithm does not decrease, but remains almost constant over different client number levels. For example, the efficiencies for repl are 1.13, 1.09, 1.13, and 1.14 for 64, 128, 192, and 256 clients, respectively.

Table VI
EFFICIENCY OF CLIENTS AT GENERATING TEST CASES

$\frac{\text{effectiveness ratio}}{\text{\# of clients}}$	64	128	192	256
grep	0.82	0.98	1.08	1.22
ptok1	1.39	1.69	1.56	1.41
ptok2	1.13	1.19	1.18	1.12
repl	1.13	1.09	1.13	1.14
sed	1.21	1.26	1.33	1.24
vim	1.06	1.03	1.05	0.96

V. DISCUSSION

Concolic Testing for Branch Coverage : In our experiment we generated test cases for path coverage, because path coverage is the target coverage of concolic testing. However, since branch coverage is more common in industrial practice, we also measured the percentages of branches covered in our study; we report these in Table VII. The branch coverages achieved did not increase much (and in four cases, each of which on grep, repl, sed, and vim they decreased) as

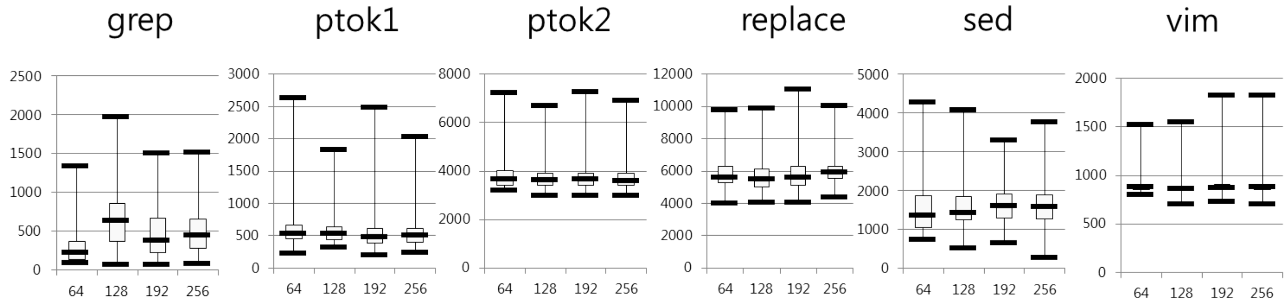

Figure 3. Distribution of numbers of test cases generated at each client node for the six object programs

Table VII
BRANCH COVERAGE ACHIEVED (%)

	1	64	128	192	256	SIR
grep	31.1	35.4	33.8	37.9	38.0	50.3
ptok1	41.5	41.5	41.5	41.5	41.5	93.6
ptok2	62.5	62.5	62.5	62.5	62.5	98.2
repl	51.0	65.4	62.4	67.1	75.5	93.9
sed	19.0	23.2	22.4	23.3	24.1	47.3
vim	9.2	9.7	11.9	11.5	11.9	35.8

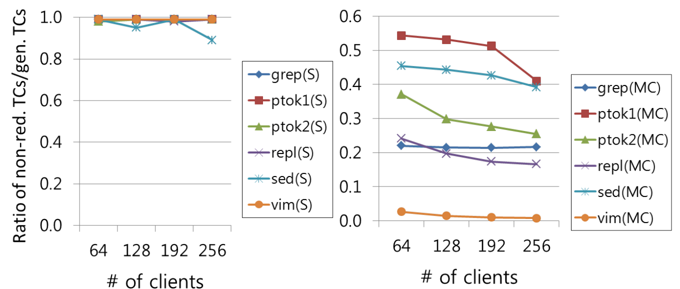

Figure 4. Ratios of non-redundant TCs over generated TCs

the number of clients used increased. We believe that this is a result of the concolic testing approach's focus on path coverage, which allow it to continue to explore new paths even though they do not cover new branches.

Note also that the branch coverages achieved in our experiment are lower than those achieved by the manually generated test cases in the SIR repository (rightmost column of the table). We believe that this too is related to the focus of the concolic approach on path coverage, though it may also be due to difficulties in solving constraints related to paths that reach particular branches. Nonetheless, given that the test cases for `grep`, `sed`, and `vim` found in the SIR repository were built over several weeks by students, and that the test cases for the other subjects were built from large pools of tests provided by the authors of Hutchins et al. [24], the mechanically achieved branch coverage attained by running SCORE for five minutes has value.

Limitations of Concolic Testing in Practice: As discussed earlier, concolic testing tools suffer from the limitations related to unavailability of library code, and the limitations of symbolic execution engines. Thus, in practice, concolic testing may not achieve full path coverage and may generate redundant test cases. To help examine this issue, Figure 4 shows the ratios of non-redundant test cases to total test cases generated for SCORE (S) (left) and multiple CREST-BV runs (MC) (right).

The ratios of non-redundant test cases for SCORE are almost one (except `sed`). In contrast, the ratios for multiple CREST-BV runs are lower than those for SCORE. In addition, the ratios decrease over increasing numbers of nodes. For example, on `sed` the ratios decrease from 0.45 (64 nodes) to 0.39 (256 nodes).

An additional limitation involves test oracles. Generating enormous numbers of test cases carries with it the complication that oracles are required for these test cases. In cases where oracles must be generated on a per test-case basis (e.g., creating "expected outputs" for each test case), this may involve excessive effort. Note that this problem arises with any large-scale automated test case generation effort. Therefore, to utilize automated test case generation frameworks to generate large numbers of test cases, we need to use oracle approaches whose cost is *not* proportional to the number of test cases. For example, we can utilize oracles that can be automated such as detecting system crashes and exceptional behavior or monitoring behavior for violations of user-given `assert` statements, which succeed in detecting many faults [3], [8], [13].

VI. CONCLUSION AND FUTURE WORK

We have developed the SCORE framework to enable distributed nodes to generate test cases independently. The resulting framework achieves linear speedup with large scalability. We demonstrated the increased effectiveness of the framework, as well as its scalability, through an empirical study on six C programs from the SIR repository, including three real-world applications. In the empirical study, SCORE demonstrated a several orders-of-magnitude increase in test

case generation speed compared to the original concolic approach, and also demonstrated clear potential for scalability. As future work, we intend to apply SCORE to additional applications, to analyze advantages and weakness of the framework in practice. In addition, we plan to conduct a comparative study on SCORE and Cloud9, since they have several common characteristics, but with different symbolic execution models. Finally, we plan to add fault-tolerant capability to SCORE.

ACKNOWLEDGEMENTS

This research was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology / National Research Foundation of Korea (Grant 2011-0000978), and Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (2010-0005498).

REFERENCES

[1] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[2] N. Tillmann and W. Schulte, "Parameterized unit tests," in *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[3] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Network and Distributed Systems Security*, 2008.

[4] M. Kim and Y. Kim, "Concolic testing of the multi-sector read operation for flash memory file system," in *Brazilian Symposium on Formal Methods*, 2009.

[5] Y. Kim and M. Kim, "SCORE: a scalable concolic testing tool for reliable embedded software," in *European Software Engineering Conference/Foundations of Software Engineering*, Szeged, Hungary, September 5-9 2011, pp. 420–423, tool demonstration track.

[6] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[7] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating System Design and Implementation*, 2008.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Intl. Symp. on Code Generation and Optimization*, 2004.

[10] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for Java," in *NASA Formal Methods Symposium*, 2009.

[11] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Automated Software Engineering*, Sep. 2000.

[12] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *International Symposium on Software Testing and Analysis*, 2008.

[13] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Programming Language Design and Implementation*, 2005.

[14] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-123, Sep 2008.

[15] K. Sen and G. Agha, "CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, 2006.

[16] M. Staats and C. Pasareanu, "Parallel symbolic execution for structural test generation," in *International Symposium on Software Testing and Analysis*, 2010.

[17] A. King, "Distributed parallel symbolic execution," Kansas State University, Tech. Rep., 2009, MS thesis.

[18] J. H. Siddiqui and S. Khurshid, "ParSym: Parallel Symbolic Execution," in *International Conference on Software Technology and Engineering*, 2010.

[19] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *6th ACM SIGOPS/EuroSys*, 2011.

[20] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Automated Software Engineering*, 2006.

[21] Y. Kim, M. Kim, and N. Dang, "Scalable distributed concolic testing: a case study on a flash storage platform," in *Intl. Conf. on Theoretical Aspects of Computing*, 2010.

[22] M. Kim, Y. Kim, and G. Rothermel, "Distributed concolic algorithm of the SCORE framework," KAIST, Tech. Rep., 2011, http://pswlab.kaist.ac.kr/publications/2012/whitepaper-score.pdf.

[23] L. Moura and N. Bjorner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *International Conference on Software Engineering*, 1994, pp. 191–200.

[25] National Security Agency (NSA), "FIPS 180-3: Secure hash standard (SHS)," 2008.

[26] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.