Efficient Concolic Testing of MPI Applications

Hongbo Li
CSE Department
Univ. of California, Riverside
Riverside, CA, USA
hli035@cs.ucr.edu

Zizhong Chen
CSE Department
Univ. of California, Riverside
Riverside, CA, USA
chen@cs.ucr.edu

Rajiv Gupta
CSE Department
Univ. of California, Riverside
Riverside, CA, USA
gupta@cs.ucr.edu

ABSTRACT

Software testing is widely used in industry, but its application in the High Performance Computing area has been scarce. Concolic testing, that automates testing via generation of inputs, has been highly successful for desktop applications and thus recent work on the COMPI [29] tool has extended it to MPI programs. However, COMPI has two limitations. First, it requires the user to specify an upper limit on input size – if the chosen limit is too big, considerable time is wasted and if the chosen limit is too small, the branch coverage achieved is limited. Second, COMPI does not support floating point arithmetic that is common in HPC applications.

In this paper, we overcome the above limitations as follows. We propose *input tuning* that eliminates the need for users to set hard limits and generates inputs such that the testing achieves high coverage while avoiding waste of testing time by selecting suitable input sizes. Moreover, we enable handling of *floating point* data types and operations and demonstrate that the efficiency of constraint solving can be improved if we rely on the use of reals instead of floating point values. Our evaluation demonstrates that with *input tuning* the coverage we achieve in 10 minutes is typically higher than the coverage achieved in 1 hour when input tuning is not used. Without input tuning, 9.6-57.1% loss in coverage occurs for a real-world physics simulation program. For the physics simulation program, using our *floating-point extension that uses reals* covers 46 more branches than without using the extension. Also, we cover 122 more branches when solving floating-point constraints using *reals* rather than directly using floating-point numbers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

MPI, Concolic Testing, Floating point, Efficiency

ACM Reference Format:
Hongbo Li, Zizhong Chen, and Rajiv Gupta. 2019. Efficient Concolic Testing of MPI Applications. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307353

CC '19, February 16–17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6277-1/19/02…$15.00
https://doi.org/10.1145/3302516.3307353

1 INTRODUCTION

Software testing is widely used to improve software quality — a program is executed on range of inputs that collectively exercise the program thoroughly. The efficacy of testing for a program is commonly evaluated using a code coverage criteria such as statements, branches, execution paths, etc. A higher code coverage by executions of tests typically corresponds to reduced likelihood of the tested software containing an undetected bug. The goal of testing is to attain a certain code coverage before releasing the software. In practice, *branch coverage* is the most commonly used code coverage criteria.

However, the study as well as the application of *testing* in the field of High Performance Computing (HPC) is scarce. While scientists may be interested delivering a tool of high quality, lack of effective and efficient testing techniques for HPC applications hinders their ability to test their code rigorously. It is thus not unexpected that the quality of HPC code is often lacking [22]. The lack of testing is ultimately the result of inadequate knowledge transfer between the Software Engineering discipline and High Performance Computing [21, 22]. Some notable works in this less-studied area include: message perturbation to improve the testing coverage of non-determinism [48], Automated Testing System (ATS) for regression testing [1], FortranTestGenerator for generating unit tests for legacy HPC applications written in Fortran [20], and GKLEE for concolic testing for GPU programs [28, 30]. However, none are aimed at improving the code coverage of MPI applications, though MPI has been a widely-used parallel programming model on distributed memory systems for decades.

Recently COMPI [29] applied *concolic testing* [17, 42] to boost the *branch coverage* of MPI applications. Concolic testing automatically generates test inputs via combination of concrete execution and symbolic execution. Since its birth, concolic testing has been a great success for testing various applications [8, 9, 18, 28, 38, 40, 47]. Its use also extends to the software industry [18, 24, 25, 47]. It works as follows. Given a target program, developers mark input-taking variables that dominates the execution path. Then the program is instrumented to perform *symbolic execution*. Afterwards, the program is executed iteratively with automatically generated inputs. In each execution, a series of *symbolic constraints* that map to encountered branches along the execution path are recorded. New input values that drive the next execution are generated via solving constraints that map to a prefix of the path followed by a negation of the next (last) constraint — the negation helps guide execution along a new path. For example, if a branch outcome generated a constraint $x > 0$,

to change the branch outcome in the next execution, we solve its negation, i.e., $x \leq 0$.

Following the above methodology, COMPI [29] proposed a concolic testing framework for MPI applications with adaptations enabling practical testing via controlling the cost of testing MPI programs. It performs symbolic execution only on one *focus* process in each execution and records branch coverage across all processes. Based on the same input, it can dynamically vary the *number of processes* (i.e., the size of `MPI_COMM_WORLD`), as well as the focus such that it can cover branches whose conditional statement depends on the size of `MPI_COMM_WORLD` or MPI rank such as the statement `if (rank == 0)`.

1.1 Limitations of COMPI

COMPI has two major limitations that hinder its testing effectiveness. First, the input values generated by COMPI do not guarantee cost-effective testing. As COMPI needs to repeatedly run target programs in testing, the time cost per execution greatly impacts the testing efficiency — the higher the time cost per execution is the less efficient the testing is. Previous research controls the execution cost via limiting the number of input values as the execution time cost of many applications increases as the number of input values increases. For example, Burnim and Sen [8] evaluated CREST using three string manipulation programs using short strings – each character in the string is one symbolic value. Kim et al. [24] applied CREST-BV and KLEE [9] to an image processing application using small image files – each pixel is a input value. However, this method does not apply to MPI programs whose execution cost is directly related to input values instead of the number of input values. It is common the larger the input value is presented to an MPI program, the more time-consuming is the execution. If an excessively large value is generated for a variable that is closely related to the size of the problem, the testing cost can be exorbitant. To address this problem, COMPI proposes a technique, known as *Input Capping*, allowing developers to set an upper limit, referred to as the *cap*, for the input generation of each variable. Its underlying idea is that with a *well-selected* smaller cap values, inputs generated achieve branch coverages that are comparable to larger cap values at a far less testing cost. However, selecting such good caps is challenging. Excessively large caps ensure good coverage but incur exorbitantly high testing cost. Conversely, too small caps ensure the overhead per program execution is low but this comes at the cost of lower coverage because some constraints may have no solution under the *cap* limits and thus some branches cannot be explored. For simple programs manual inspection of the constraints of all branches can help developers find caps such that the caps do not prevent the constraints from being solved. *However, manual inspection is infeasible for complex or large programs and thus an automated approach is essential.*

Second, COMPI does not support floating-point types and operations that are commonly used in HPC applications. Using COMPI to test an MPI program that reads many floating-point values requires developers to manually fix the floating-point variables to selected values. But fixing the variables to certain values prevents testing from covering branches depending on these variables (e.g., the `true` side of conditional statement `if (x < 1)` cannot be exercised if we fix x to 2.0). Furthermore, floating-point operations are either ignored or recorded imprecisely (e.g., assignment statement `x = y + 1.5` is ignored as expression `y + 1.5` is a floating point operations). *The lack of floating-point support can cause some constraints not to be recorded or solved, and branches related to the use of floating-point types and operations may never be covered during testing.*

1.2 Our Contributions

We strengthen concolic testing for MPI applications via overcoming the above limitations. We propose *input tunning* to make testing cost-effective while avoiding the need for user to manually set hard *cap* limits. Its overall idea is as follows. COMPI generates new input values via solving a subset of dependent constraints (details in Section 2.1) — the new values are consumed by the variables appearing in these constraints in the next test run. Input tuning aims to make these values as small as possible as follows. It identifies the largest value L in the generated values and then, via binary search over the range $(0, L]$, it finds the smallest values for the involved variables such that the constraints can still be satisfied and thus uses them to drive the next test run. That is to say, we can achieve cost-effective testing via searching for small values to drive the testing as (1) the search does not disrupt the constraint solving unlike hard *cap* limits, and (2) they are small enough to ensure the least-expensive execution during testing.

We also extend COMPI to support *floating-point data types and operations* and show that the efficiency of constraint solving can be greatly improved if we rely on the use of reals instead of floating point values. Satisfiability modulo theories (SMT) solvers like Z3 [13] have begun to support floating point reasoning due to the recent advances of the solver technology. This leads to the incorporation of floating-point reasoning into concolic testing [32]. However, solving constraints over floating-point numbers is far slower than over reals. Though approximating floating-point arithmetic using real arithmetic sacrifices precision, we show that the high efficiency of the approximation outweighs the imprecision in terms of achieving higher testing coverage in practice.

The main contributions of this paper include:

- We present *input tuning* to achieve the most cost-effective testing via automatically searching for the smallest values that satisfy the collected constraints and thus eliminate the need for manually setting hard *cap* limits.
- We support *floating-point* data types and operations and demonstrate significant improvement in constraint solving and testing efficiency by approximating floating-point arithmetic using real arithmetic.
- We evaluate input tuning for HPL, IMB-MPI1, and SUSY-HMC based on one-hour of testing. For HPL, with input tuning we cover 1865 branches in less than 10 minutes which is 6× faster than the time it takes to achieve the same coverage without using input tuning. For IMB-MPI1, with input tuning we achieve coverage of 766 branches in less than 8 minutes while without it only 735 branches are covered in one hour. For SUSY-HMC, with input tuning we achieve the highest coverage, while with input capping 9.6-57.1% coverage loss occurs in other settings.

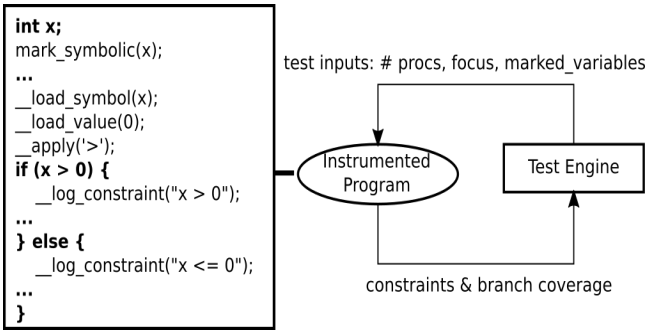

Figure 1: Concolic testing of MPI programs: (1) on the left is a segment of one instrumented program with the code lines in bold being the original code, `mark_symbolic()` being inserted by developers, and the remaining being the symbolic execution code inserted automatically; and (2) on the right shows how the test engine tests the instrumented program.

- We evaluate our floating-point extension using SUSY-HMC physics simulation program with one-hour of testing. With our floating point extension using reals we cover 46 more branches than without it. Also we cover 122 more branches when solving floating point constraints using reals rather than directly using floating point numbers during solving.

2 BACKGROUND AND OVERVIEW OF SOLUTIONS

Here we briely describe the concolic testing process of COMPI for MPI programs and the incremental constraint solving approach used by COMPI. We also illustrate with examples the limitations of the current concolic testing tool for MPI programs as well as overview our proposed solutions.

2.1 Concolic Testing of MPI Programs

Testing Process. The concolic testing of a given MPI program consist of two major steps: *instrumentation* and *iterative testing*.

In the instrumentation step, developers manually mark variables that read input values and dominate the program execution, then the marked program is transformed into a simplified program in C Intermediate Language (CIL) [33], and finally the simplified program is instrumented with symbolic execution code as shown in Figure 1. With the simplification, branch statements like loops and `switch` are all translated into `goto` and `if` statements. Each `if` statement only contains a simple condition, e.g., $\{x > 0\}$ instead of $\{x > 0$ and $x < 10\}$, and is always accompanied with an `else` statement. The true/false *branch* outcome causes the execution of the `if`-side/`else`-side of the conditional statement. The *branch coverage* metric represents the number of branch outcomes covered during testing. Note the term *branch coverage* used in this paper refers to the branch coverage of the simplified CIL program.

Next, iterative testing (i.e., iteratively executing the target program with generated inputs) is performed so as to increase the branch coverage and potentially uncover software bugs. At the end of each execution, a series of symbolic constraints mapped to the branches along the program execution path are recorded.

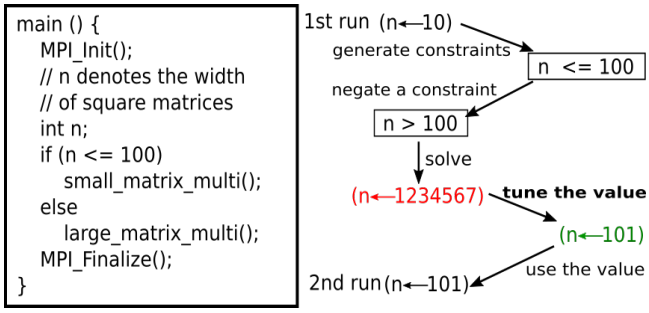

Figure 2: Input tuning achieves cost-effective testing: (1) on the left is an MPI program performing square matrix multiplication with n denoting the matrix width; (2) on the right input tuning helps avoiding expensive execution via replacing 1234567 with 101.

The testing tool can generate a set of input values via solving constraints in a prefix of the execution path followed by a *negation* of the next constraint in the prefix. Due to the negation, the new inputs can potentially cover a new branch outcome. Among these inputs, some are used to determine the number of processes to be used as well as which process should be the *focus* process — the focus is the only process on which symbolic execution is performed while all the other processes only perform concrete execution (e.g., the code lines in bold in Figure 1). Based on these, the test engine can configure the right number of processes and the focus when launching the program. The remaining input values are passed to the marked variables at runtime, e.g., variable x takes one value via `mark_symbolic()` in Figure 1.

Incremental Constraint Solving. Incremental constraint solving is a widely used approach in many concolic testing tools due to its efficiency when solving similar constraints repeatedly. CREST [8], on which COMPI is built, benefits from it as well. Its basic idea is to exploit the similarity between two constraint sets being solving consecutively to speedup the solving process. It works as follows: (1) it only solves subset of constraints — the *target negated constraint* as well as *constraints depending on it* [1] — such that new values are generated for variables appearing in these constraints; and (2) it assign old values from previous input to all the other variables that do not appear in the constraints. Since each time only a subset of, instead of all, the constraints are solved, this technique greatly speedups the constraint solving.

We observe that a property inherent to this technique is: *An input value generated for a variable remains unchanged as long as the variable does not appear in the incrementally solved constraints.*

2.2 Overview of Our Solutions

Input Tuning. Though COMPI's input capping relieves the issue to a certain degree, it is very challenging to select a good set of *cap* limits. Consider the MPI program performing square matrix multiplication shown in Figure 2 where variable n representing the

[1] Two symbolic constraints are claimed to be dependent if only they share the same variables.

```
main () {
  ...
  int a; float b;
  COMPI_int(a);
  ...
  if (b > 1.1 && b < 1.2) f1();
  ...
  float c = a * 1.1;
  if (c > 2) f2();
  ...
}
```

Figure 3: Concolic testing of a program without support for floating-point data types and operations.

matrix width determines the execution time. The program is designed to use different strategies for different range of matrix widths to optimize performance — `small_matrix_multi()` is invoked when $n \leq 100$ and `large_matrix_multi()` is invoked otherwise. If the upper *cap* limit is set to 50 (i.e., $n \leq 50$), `large_matrix_multi()` will not be explored during testing. On the other hand, if the upper limit is set to 500, the testing could be very expensive as the matrix width could be as high as 500. The property of incremental solving as discussed earlier in Section 2.1 exacerbates the high cost problem — once a large width value is generated it could stay unchanged for a long time and thus repeated time-consuming executions will be performed.

With our input tuning technique we can achieve the best cost-effective testing without the need for finding the best upper limits as input tuning always finds the smallest value to satisfy a given constraint. In Figure 2, the input tunning technique is illustrated. Suppose in the first run a random value is generated $n \leftarrow 10$. After execution, constraint $n \leq 100$ is obtained. Via negating it ($n > 100$) the testing aims to cover the branch outcome that invokes `large_matrix_multi()`. The solver can generate any value like $n \leftarrow 1234567$ to satisfy $n > 100$. This obviously is the worst scene the testing needs to avoid. With input tuning, we can find that $n \leftarrow 101$ also satisfy $n > 100$. This small value ensures `large_matrix_multi()` is invoked with the minimum possible execution time.

Floating-Point Support. We exemplify the consequence of missing floating point support with the example shown in Figure 3. In this program, variable a and b read inputs from users. We mark a as symbolic. As there is no marking interface to support marking of b, a `float` variable, as symbolic in COMPI, we can only fix it to a selected value (e.g., 1.1). Variable c is also a `float` and its value is derived from a. Suppose a is initialized to 1 in the first test. However, function `f1()` cannot be explored as $b = 1.1$ does not satisfy `b > 1.1 && b < 1.2`, and `f2()` cannot be explored as the symbolic constraint $a * 1.1 \leq 2$ is not recorded, which is ultimately due to the fact that floating-point multiplication like $a * 1.1$ is ignored by COMPI's symbolic execution component.

To overcome the above limitation, we provide an interface to developers for marking floating-point variables and allow floating-point arithmetic in the symbolic execution component. Our extension helps cover branches related to the use of floating-point calculations.

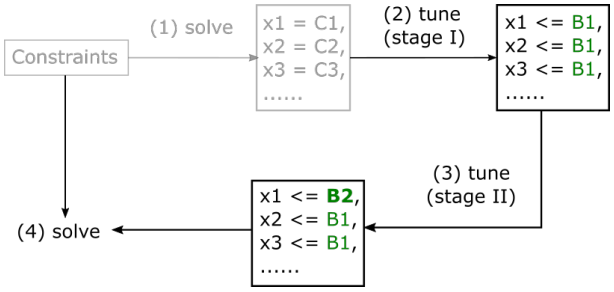

Figure 4: Two-stage tuning is applied on the solution generated by the solver — the solution contains the values generated for variables $x1$, $x2$, and $x3$, which are respectively $C1$, $C2$, and $C3$. After Stage I tuning, the smallest upper bound, $B1$, is found for all involved variables (i.e., $x \leq B1$ for $\forall x \in \{x1, x2, x3, ...\}$ with $B1 \leq max\{C1, C2, C3, ...\}$). After Stage II, the smallest bound is found for variable $x1$ if $x1$ is the single variable in the target constraint (i.e., $x1 \leq B2$ with $B2 \leq B1$). Within the limits of these bounds, the constraints are solved to get the optimized solution.

3 INPUT TUNING

Directly applying the values generated by the constraint solver often incurs high testing cost that is not necessary. Though setting upper limits relieves this problem to a certain degree, it is challenging to manually find the best limits with which the testing achieves a high coverage yet incurs the least time cost. We thus propose *input tuning* as a solution to achieve effective testing that eliminates the challenge of setting hard *cap* limits.

3.1 Design of Our Approach

The tuning process consists of two stages: (Stage I) *Multi-variable tuning* that optimizes all variables appearing in the *dependent constraints* such that their values are no bigger than the detected smallest upper bound; and (Stage II) *Single-variable tuning* that optimizes the single variable in the target negated constraint, i.e., the target negated constraint only contains one variable, under the limit of the detected bound. Figure 4 illustrates how the two-stage tuning optimizes the solution, i.e., input values. These two stages are complimentary. Stage I ensures dependent variables, like $x2$ and $x3$ in Figure 4, are not significantly increased when tuning the single variable in the target constraint, like $x1$. Stage II ensures the single variable gets the smallest value under the upper bound detected in Stage I.

Consider the application of concolic testing to the following code segment.

```
...
if ( x - y > 100 )
{ // b1
  if ( y > 0 ) ... // b2
  else ... // b3
}
...
```

This code segment has three branches $b1$, $b2$, and $b3$. In the i−th test, suppose that $b1$ and $b3$ are covered, i.e., the execution path is $[..., b1, b3, ...]$. In the $(i+1)$−th test, we hope to force execution

along a new path $[..., b1, b2, ...]$ and thus cover $b2$ by satisfying constraints $[x - y > 100, y > 0]$. Directly applying the solver to the above constraints could probably generate values such as $[x = 4321, y = 1234]$. Our tuning strategy can optimize these generated values as follows. In stage I, we find common upper limit 102 for both x and y such that $[x-y > 100, y > 10, x \leq 102, y \leq 102]$ is still solvable. In stage II, we find the minimal upper limit 11 for only y such that $[x - y > 100, y > 10, x \leq 102, y \leq 11]$ is still solvable. Thus, we are able to decrease the satisfying input values from $[x = 4321, y = 1234]$ to $[x = 102, y = 11]$ and cover $b2$ with far less testing cost.

Next we details the input tuning algorithm. Algorithm 1 overviews the two-stage tuning process. Algorithm 2 and Algorithm 3 are two utility functions, where the former finds the largest input value of a solution and the latter illustrates the upper bound search process.

Stage I.. Suppose *target* is the negated constraint, *cstrs* stands for the constraint set including *target* as well as the constraints depending on *target* (see Section 2.1), *excls* is a set of *symbolic symbols* [2] that do not need tuning, and *soln* stores the generated values for *symbols* appearing in *cstrs* as *key-value pairs* with *key* being a symbolic symbol and *value* being the generated value for symbol *key*. The goal of Stage I tuning is to find the lowest upper bound for all symbols not appearing in *excls*, i.e., we exclude symbols/variables that do not need to be tuned. This process is composed of the following steps:

- **Decide to tune or not** (Algorithm 1 and 2). At first, we find the largest value, denoted as *bound* (Algorithm 2), among the generated values, stored in *soln*, for symbols not appearing in *excls*. If the largest value is too small (i.e., $bound < 2$), we directly return *soln* as the input values are already small enough and there is no need to tune them further (Algorithm 1: lines 4-5).
- **Fix variables requiring no tuning** (Algorithm 1). We fix the variables that do not need tuning, i.e., those appearing in *excls*, to the generated values in *soln* to avoid any value changes caused by the tuning (lines 6-12).
- **Search the lowest upper bound** (Algorithm3). We search for the smallest upper bound for symbols/variables to be tuned using binary search in the range of $(0, bound]$ (lines 2-21). In the search, we construct new constraints via `new_cstr()` that specifies tuned variables are no greater than *mid*, where *mid* is the average of the lower and upper bound (lines 6-14). Then we check if the new constraints *cstrs_* are consistent with old ones *cstrs* (line 15), and set the upper bound as *mid* if they are consistent (line 16) and the lower bound as *mid* otherwise (line 18). The lowest bound is obtained after the search is complete.
- **Set upper bound** (Algorithm 1). We set an upper bound for tuned variables via constructing new constraints specifying their values must be no larger than the detected bound (lines 15-19).

Stage II. . Stage II aims to optimize the value for the single variable within the restriction of the upper bound detected in Stage

[2]Each *symbolic symbol* represents a *variable* marked in the tested program. In the paper, we use the term symbol and variable interchangeably

Algorithm 1 Input Tuning

Input: *target* ▷ target negated constraint
cstrs ▷ dependent constraints including *target*
excls ▷ variables requiring no tuning
soln ▷ a solution generated by the solver
Output: *opt_soln* ▷ an optimized solution

```
1:  function TUNE(target, cstrs, excls, soln)
2:      /* * * STEP 1: optimize a group of variables * * */
3:      bound ← get_largest(soln, excls)
4:      if bound < 2 then return soln                          ▷ 1.1
5:      end if
6:      _cstrs ← cstrs                                         ▷ 1.2
7:      // fix the values of symbols in excls
8:      for all s.key ∈ excls do
9:          // construct new constraint: s.key = s.value
10:         c ← new_cstr(" = ", s.key, s.value)
11:         _cstrs ← _cstrs ∪ {c}
12:     end for
13:     opt_bound ← optimize_multi(_cstrs, excls,              ▷ 1.3
14:          soln, bound);
15:     // set upper bounds                                    ▷ 1.4
16:     for all s ∈ soln AND s.key ∉ excls do
17:         c ← new_cstr(" <= ", s.key, opt_bound)
18:         _cstrs ← _cstrs ∪ {c}
19:     end for
20:     /* * * STEP 2: optimize a single variable * * */
21:     if target contains more than one variable then         ▷ 2.1
22:         return solve(_cstrs)
23:     end if
24:     if opt_bound < 2 then
25:         return solve(_cstrs)
26:     end if
27:     symb ← single symbolic symbol in target
28:     opt_bound2 ← optimize_single(_cstrs, excls,            ▷ 2.2
29:          opt_soln, opt_bound, symb)
30:     if opt_bound2 < opt_bound then                         ▷ 2.3
31:         c ← new_cstr(" <= ", symb, opt_bound2)
32:         _cstrs ← _cstrs ∪ {c}
33:     end if
34:     return solve(_cstrs)                                   ▷ 2.4
35: end function
```

Algorithm 2 Get the Largest Input Value

```
1:  function GET_LARGEST(soln, exls)
2:      bound ← −1
3:      for all s ∈ soln do
4:          // largest not in excls
5:          if s.key ∉ excls and s.value > bound then
6:              bound ← s.value
7:          end if
8:      end for
9:      return bound
10: end function
```

Algorithm 3 Search for the lowest upper bound

```
1:  function OPTIMIZE_MULTI(cstrs, excls, soln, bound)
2:      /* * * optimize variables * * */
3:      lower ← 0, upper ← bound
4:      prev_upper ← upper
5:      while lower + 1 < upper do
6:          mid ← lower + (upper − lower)/2
7:          cstrs_ ← ∅
8:          for all s ∈ soln do
9:              if s.key ∉ excls then
10:                 // construct constraint: s.key ≤ mid
11:                 c ← new_cstr(" <= ", s.key, mid)
12:                 cstrs_ ← cstrs_ ∪ {c}
13:             end if
14:         end for
15:         if cstrs_ is consistent with cstrs then
16:             upper ← mid
17:         else
18:             lower ← mid
19:         end if
20:     end while
21:     return upper
22: end function
```

I only if the variable is the single variable in the target negated constraint. It consists of similar steps.

- **Decide to tune or not** (Algorithm 1). We check if the target negated constraint, namely *target*, only contains single variable (lines 21-23) and if the detected bound is already small enough (lines 24-26). If either is not satisfied, we directly solve and return; otherwise, we proceed to the next step.
- **Search for the lowest upper bound** (Algorithm 1). This is the same to the search in Stage I except that it optimize only a single variable (lines 27-29).
- **Update upper bound** (Algorithm 1). If the new bound is smaller than the older one, we update the upper bound for the single variable (lines 30-33).
- **Generate optimized values** (Algorithm 1). We solve the updated constraints, i.e., *_cstrs*, to get the optimized values (line 34).

Additional setup. As no constraints are available prior to the first test, input generation for the first test is not available. We need to assign input values. We make all the initial values as the smallest positive integer for integer varaiables (i.e., 1). This setting makes not only the first test as well as latter tests efficient enough considering the value persistence property of incremental constraint solving.

3.2 Applicability

Input tuning is effective for tuning input values for a variable when the following conditions are satisfied: (1) the variable is of integer type like char, int, and long; (2) the larger the value of the variable is the longer the execution takes; and (3) eligible values allowing the program performing its function must be positive, e.g., for the square matrix multiplication program the matrix width must be positive so as to perform valid matrix multiplication. Below we detail how we deal with the cases when one of the conditions is not satisfied.

Floating-point variables do not satisfy condition (1). We do not perform input tuning for floating-point variables as usually the values of integer variables, like matrix width in the matrix multiplication program, determine the problem size. However, this technique can be applied for floating-point variables as well.

There two types of variables that do not satisfy condition (2): Type-1 variables whose values are unrelated to execution time and Type-2 variables for which increase in value leads to shorter execution. We do not differentiate type-1 when applying the tuning as tuning it does not have much side-effect other than the tuning cost — the tuning cost is trivial considering constraint solving including the tuning accounts for less than 3% of the total testing time in our evaluation. To deal with type-2, we allow developers to mark variables that need to be excluded from the tuning process. A variable representing the number of processes, i.e., the size of `MPI_COMM_WORLD`, is a good example of Type-2. As aforementioned, the testing also generates input values used for determining the number of processes: for the same workload the execution takes more time when more processes are used to run the program. In this paper, we only mark variables representing the number of processes in *exclusion*. If it is found the appearance of Type-2 variables in real-world MPI programs becomes common, we can automate the marking process as follows. For each variable, we can measure the execution time cost by giving differing values. By comparing the cost, we can easily know whether smaller values or larger values lead to longer execution. Hence, we can automatically exclude the Type-2 variables in our tuning.

We do not tune the variables violating condition (3) as for the majority of, if not all, HPC applications only positive values are meaningful.

4 FLOATING POINT SUPPORT

Enabling floating-point data types and operations in concolic testing requires adapting three components of COMPI: instrumentation module, symbolic execution library, and the constraints solving component.

Instrumentation module. guides the insertion of symbolic execution code into the target program. The instrumentation is performed at instruction level. For example, the instruction $x = y + z$ needs to be inserted with four code sequences to achieve symbolic execution: two for loading the symbolic expressions of x and y, one for applying the add operation, and one for storing the symbolic expression for $y + z$ into x. The instrumentation module of COMPI only instruments integer variables and operations. We adapted the module such that it also instrument floating-point variables and operations.

Symbolic execution library. defines all the instrumentation functions — the instrumented instructions discussed above are function calls to the functions of the library. These functions manipulate symbolic expressions according to the original instructions. In COMPI, all symbolic expressions only represent linear arithmetic operations as

$$E = C + \sum_{i=0}^{i=N-1} C_i * x_i,$$

Table 1: Time cost (unit: seconds) of floating-point constraint solving using reals and floating-point values based 100 iterative tests of a simple synthetic program.

Expression →	x	$x+y$	$x+y+z$
Float →	31.4	75.0	91.2
Real →	8.2	8.1	8.2

where E is a symbolic expression, C is a constant, x_i denotes a symbolic symbol representing one input-taking variable, C_i is the coefficient of x_i, and N is the number of symbolic symbols in E. COMPI ensures linear constraints via replacing symbolic expressions with concrete values as needed. For example, consider the multiplication of two symbolic expressions: $x * y$ with both x and y being symbolic expressions. To avoid non-linear operation $x * y$ being recorded, COMPI substitutes the symbolic expression of y with the concrete value of y like 2 such that the result expression is $2x$ which is still linear. As COMPI only targets integers, it records C and C_i using 64-bit integers.

Our floating-point extension requires us to represent integer expressions in the same way, but for a floating-point expression we record C and C_i using double-precision floating point values. Also, the extension also needs conversion between floating-point and integer expressions. We convert a integer expression into a floating -point expression via converting C and C_i from 64-bit integers to double precision floating point numbers. We convert a floating-point expression into an integer expression via converting the concrete value of the floating-point expression into an 64-bit integer, i.e., after the conversion the integer expression is a concrete value instead of symbolic expression. In addition, we provide the marking functions for developers to mark variables of data type `float` and `double` as symbolic such that these variables can also be involved in the symbolic execution.

Constraints solving component. solves constraints to generate new inputs that are used in the next test run and this process is used repeatedly during iterative testing. For incremental solving, this component finds all constraints depending on the target negated constraint, and uses Yices-1.0 [4], an SMT solver, to solve the dependent constraints. In COMPI, the component is only able to solve integer constraints.

As SMT solvers like *Z3* [13] has begun to support floating-point reasoning, concolic testing is also able to solve constraints with floating-point arithmetic based on the floating-point reasoning of Z3. However, the floating-point reasoning is known for its high cost — the cost of solving floating-point constraints is hundreds of times the cost of solving integer constraints [49]. Therefore, instead we propose simulating floating-point arithmetic using real arithmetic that is far less expensive. To compare the efficiency between solving using reals and using floating-point values, we created two versions of COMPI: one solves constraints using floating-point reasoning of Z3, and the other solves constraints using real arithmetic of Z3. We use the two versions of the tool to test a simple synthetic program with 3 `if` statements below:

```
if (expr == 0) ...
if (expr < 0) ...
if (expr <= 0) ...
```

where *expr* stands for an C floating-point expression. In the testing, the program can generates 6 constraints including $expr = 0$, $expr \neq 0$, $expr < 0$, $expr \geq 0$, $expr \leq 0$, and $expr > 0$ such that all the relational operators are covered.

Based on 100 iterative tests, we measured the time cost of constraints solving using reals and using floating-point values based on three expressions: x, $x+y$, $x+y+z$ (the data type of x, y and z are all `float`). We observed that the solving time using floating point values is 3.8× to 11.1× times the solving time using reals. Also the solving time using floating point value grows as the number of variables in the expression grows, while the solving time using reals stays almost the same. Hence, we believe the efficiency of solving floating point constraints using *reals* makes it a better fit for practical testing. It should be noted that as our approximation trades off accuracy for performance our tool may fail to satisfy a floating-point comparison demanding high accuracy and thus fail to cover the corresponding branch.

5 EVALUATION

We evaluate *input tuning* and *floating point* extension of concolic testing based on three non-trivial MPI applications.

Hardware and Tool setup. The evaluation is performed on a computer equipped with two Intel E5607 CPUs with total of 8 cores and 32 GB memory. In the evaluation, COMPI tool uses Z3 instead of Yices-1.0 as its constraint solver due to the floating point extension. By default, the tool runs the target program with 8 processes with the focus being rank 0 in the first test. Additionally, the number of processes is restricted to no more than 16 during dynamic variation as without it the computer can crash when running with too many processes. Our tool sets all input values to 1 for the first test run for both input tuning and input capping techniques for fair comparison. The decision on which constraint to negate is made by the search strategy — COMPI uses is *BoundedDFS*. BoundedDFS explores the execution tree using a variation of depth-first search (DFS) strategy which skips constraints as well as branches that are deeper than a *specified depth bound* in the execution tree. The depth bound is selected to ensure that COMPI has the ability to explore the entire execution tree. The testing process using BoundedDFS (1) applies x tests without setting a bound first so that the maximal number of constraints M can be observed and (2) performs the testing with a selected bound B, which is obtained via rounding up M to the next hundred. In the default setting, we perform 100 tests to detect the bound, i.e., $x = 100$.

Evaluation goals and applications. Our evaluation aims to show that input tuning is more effective than input capping, i.e., it achieves *higher coverage at lower testing cost.* We use HPL [2], IMB-MPI1 [3], and SUSY-HMC [37] to evaluate input tuning as they all have integer inputs. For floating-point support, we aim to show that testing with floating-point extension achieves higher coverage than without it and solving floating-point constraints using real values saves testing time without sacrificing branch coverage. This evaluation uses only SUSY-HMC as it has multiple floating-point inputs while HPL has only one and IMB-MPI1 has none.

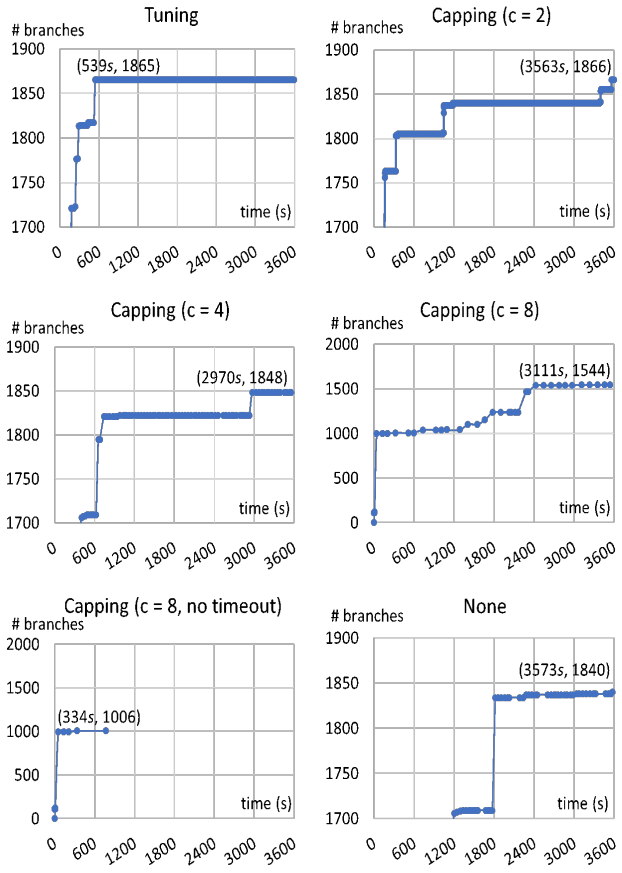

Figure 5: Branch coverage progress over one-hour of testing of HPL using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y.

5.1 HPL

HPL [2] is a high-performance Linpack benchmark for distributed memory computers. It solves a dense linear system using LU factorization. Many of the algorithm features can be exploited by configuring the abundant parameters it provides. To enable concolic testing, we need to mark variables for which the testing tool is to generate input values. HPL read inputs from a designated file, marking variables requires us to insert the marking lines as well as commenting out the reading from the file. For HPL we mark 23 integer variables (the variable can also be an array) by inserting 23 lines of code as well as commenting out the same amount of lines. The depth bound for *BoundedDFS* is 500 based on the observations in the first 100 tests.

We compare *input tuning* with four *input capping* settings as well as the case where neither input tuning or input capping is used (called *None*). In the input capping evaluations, we set the same cap (or upper bound limit), denoted as c, for all variables, and use three caps: $c = 2$, $c = 4$, and $c = 8$. We also evaluate $c = 8$ without the timeout mechanism — the tool by default uses timeout to identify excessively long executions such as those caused by infinite loops — to avoid the interference from timeout as many large input values cause the execution to timeout when $c = 8$. We allow each of the above configurations to *test for one hour*.

Table 2: Comparison among Tuning, Capping (C2, C4, C8, and C8 using No Timeout), and None based on HPL with two metrics: the time costs (unit: seconds) of covering 1860 branches and the number of tests completed in one hour.

Metric ↓	T	C2	C4	C8	C8_NT	N
Cost (1860)	539	3563	–	–	–	–
# tests	390	1717	231	63	32	215

Figure 5 shows that using *input tuning*, the testing covers 1865 branches, which is only 1 less the highest coverage. Using input capping with $c = 2$, the testing achieves the highest coverage but the time cost to achieve such coverage is almost 1 hour while the time cost of covering 1865 branches using input tuning is less than 10 minutes. This is because, for $c = 2$ the values of all variables must be smaller than 2, and thus very often the constraints have no solution. Using capping with $c = 4$, the testing coverage is 17 branches fewer than when using *input tuning*. Using capping with $c = 8$ in the default setting, 321 branches fail to be covered as larger upper bound permits larger values and larger values make the execution unnecessarily long such that many executions are killed by the timeout mechanism. Using capping with $c = 8$ without the timeout scheme, the coverage obtained is even less due to the same reason — too large values can cause one program execution to take tens of minutes (the program execution that started after 768 seconds did not finish till finally 1 hour expired). The *None* configuration that directly uses the values generated by the solver (i.e., neither tuning nor capping is used) delivers coverage of 1840 branches after running for over 30 minutes. This is not only worse coverage than *input tuning* but also at a much higher execution time cost (10 minutes vs. 30 minutes).

Table 2 demonstrates the high efficiency of testing using input tuning. The time it takes to cover 1860 branches using *input tuning* is 539 seconds which is only 15.1% cost of using capping with $c = 2$. In all other configurations the coverages and time costs are significantly worse. This high efficiency is the result of *input tuning* preferring smaller values and only using larger values when necessary. Thus, input tuning ensures testing makes progress at a good pace. Table 2 also shows the efficient testing using *input tuning* executed 390 tests in one hour. All other configurations, except input capping with $c = 2$, perform fewer tests in one hour because unnecessarily long executions are involved. Although input capping with $c = 2$ executes many more test cases with short runs, it still takes about one hour to deliver nearly the same coverage because frequently constraints have no solution. In other words, *input tuning* choses neither too small nor too large inputs and as a result execution runs are just long enough to keep delivering solvable constraints and thus higher and higher coverage.

5.2 IMB-MPI1

IMB-MPI1 [3] is a major component of Intel MPI Benchmarks (IMB) and is used for benchmarking MPI-1 functions. It reads inputs by parsing the command line. We mark 14 integer variables by commenting out the whole code block that parses command line

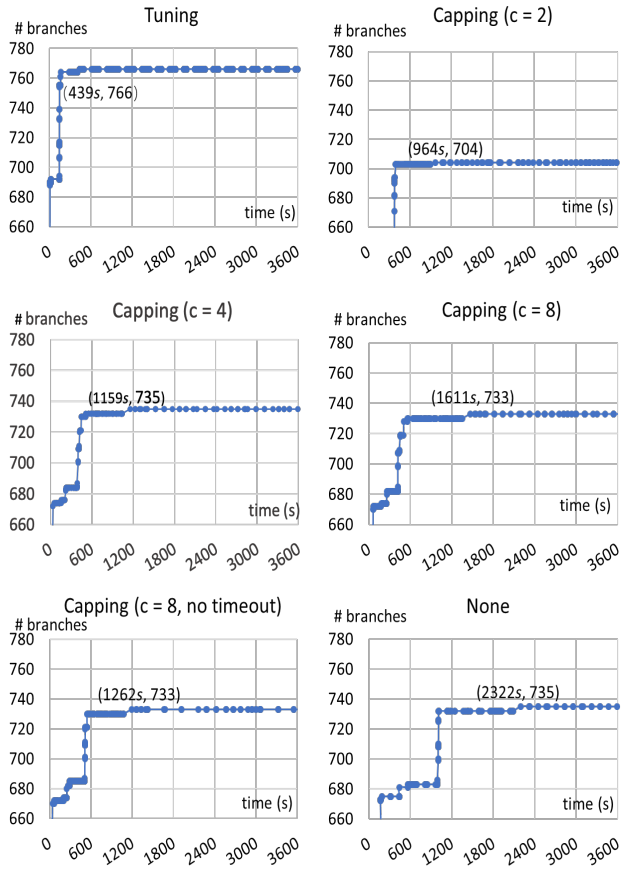

Figure 6: Branch coverage progress over one-hour of testing of IMB-MPI1 using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y.

Table 3: Comparison among Tuning, Capping (C2, C4, C8, and C8 using No Timeout), and None based on IMB-MPI1 with two metrics: the time costs (unit: seconds) of covering 730 branches and the number of tests completed in one hour.

Metric ↓	T	C2	C4	C8	C8_NT	N
Cost (730)	142	–	449	559	545	1011
# tests	2806	280	246	244	240	224

and inserting 30 lines with about half of them being the marking lines and the others being sanity checks on the inputs. The depth bound for *BoundedDFS* is 200 based on the observation in the first 100 tests. We compare input tuning with input capping as well as the case where neither tuning nor capping is used. In the *input capping* evaluation, we set the same cap for all variables, and use three configurations: $c = 2$, $c = 4$, $c = 8$. We also evaluate $c = 8$ without using timeout. Once again we perform *testing for one-hour testing* in each configuration.

Figure 6 shows using *input tuning*, we cover the most branches, i.e., 766 branches. Using capping with $c = 2$, we cover about 700 branches as the cap limit is too small. When we use bigger cap limits like $c = 4$ and $c = 8$ in the default setting, the coverage is over 30 branches less than the coverage based on input tuning. Using capping when $c = 8$ without timeout scheme, the coverage does not improve since without timeout expensive tests costing several minutes are used and thus one hour is not enough to explore the branches. Without using either input tuning or capping technique (*None*), the coverage is less than the coverage using *input tuning* as frequently long executions are killed by the timeout mechanism of COMPI.

Most importantly, the efficiency of *input tuning* is justified — with input tuning we cover 766 branches in only 439 seconds, which cannot be achieved in any other configurations. Further Table 3 shows with input tuning we cover 730 branches in 142 seconds, which is only 14.0-31.6% the time cost of other techniques. Still this is because input tuning permits large values as well as long program executions only when necessary. Table 3 shows the number of tests performed by testing using input tuning is far bigger than the number of tests performed by testing in input capping configurations. This result from the fact that if using input capping we need to carefully find the most appropriate cap limit for each variable to achieve cost-effective testing, while the default limits for all variables are set to the same value. It is obviously very challenging to make input capping cost-effective as selecting cap limits is hard. On the other hand, *input tuning* eliminates the need for setting limits.

5.3 SUSY-HMC

SUSY-HMC [37] is a major component in SUSY LATTICE — a physics simulation program for Rational Hybrid Monte Carlo simulations of extended-supersymmetric Yang–Mills theories in four dimensions. It reads inputs from standard input stream. We consider 13 integer variables and 7 double-precision floating point variables and mark different numbers of variables depending on the evaluation goals. The marking is achieved by commenting out the code block that reads values from standard input stream and inserting around 23 lines of marking code. The depth bound for *BoundedDFS* is 500 based on the observation in the first 100 tests.

Input tuning. We mark 13 integer variables using COMPI that does not have floating-point support. We compare input tuning with three input capping settings ($c = 2$, $c = 8$, and $c = 8$ without the timeout scheme). Our tool aborts in two configurations: capping with $c = 4$ and the case where neither input tuning nor input capping are used. Thus, results in these configurations are not shown. Each configuration is evaluated over one-hour of testing.

Figure 7 demonstrates with *input tuning* we obtain the highest coverage, i.e., 1662 branches. Using input capping with $c = 2$, we only cover 713 branches. Using capping with $c = 8$, we covers at most 1503 branches regardless of whether the timeout scheme is used or not. Using input capping, the loss in coverage ranges from 9.6% to 57.1%. Obviously, the serious coverage loss using input capping is caused by a bad *cap* limit. On the other hand, *input tuning* delivers high coverage without requiring users to find a good *cap* limit.

Table 4 shows that in just about one minute using input tuning we cover 1600 branches, which is about 100 more branches than the

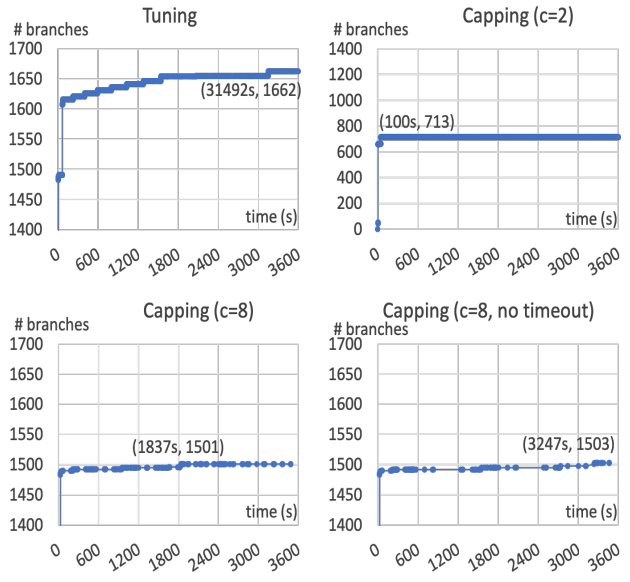

Figure 7: Branch coverage progress over one-hour testing of SUSY-HMC using *input tuning* and *input capping*: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y.

Table 4: Comparison between Tuning and Capping (C2, C8, and C8 using No Timeout) based on SUSY-HMC with two metrics: the time costs (unit: seconds) of covering 1600 branches and the number of tests completed in one hour.

Metric ↓	T	C2	C8	C8_NT
Cost (1600)	64	–	–	–
# tests	1411	8074	190	131

coverage of one-hour testing using input capping with $c = 8$ and about 900 branches higher than the one-hour coverage based on input capping with $c = 2$. The high efficiency of testing based on input tuning results from the fact that input tuning chooses small enough input values yet without disrupting the constraint solving process. Hence input tuning makes testing progresses at a good pace (1411 tests in one hour as shown in Table 4), which is neither too fast (8074 tests when $c = 2$) nor too slow (190 and 131 tests when $c = 8$) just like the evaluation of HPL.

Floating-point support. On the basis of input tuning, we evaluate our floating point extension by comparing three versions of the testing tool: one that only considers integers (Int); one with floating point extension using reals (Real); and one with floating point extension that directly uses floating point numbers (Float). When using Real and Float, we mark all the identified 13 integer variables and 7 floating-point variables. When using Int, we (1) only mark the 13 integer variables as floating-point variables cannot be marked in the Int version, and (2) fix the floating-point variables to 1 for fair comparison, considering all COMPI versions set all input values to 1 in the first test run. We perform one-hour of testing using each version of the tool.

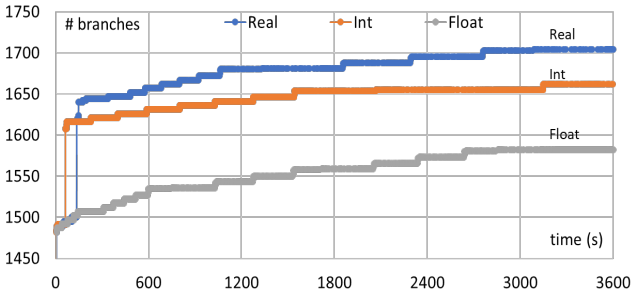

Figure 8: Branch coverage progress of testing of SUSY-HMC based on 3 versions of COMPI: (Int) only integers; (Real) with floating point extension using reals; and (Float) with floating point extension directly using floating point numbers.

Figure 8 shows that Real achieves the best coverage after 200 seconds of testing. Real covers 1704 branches, Int covers 1662 branches, and Float only covers 1582 branches. We find that Real covers 42 more branches than Int. This demonstrates that floating-point extension help testing achieve greater coverage. Also, we find Float achieves the worst coverage though it also support floating point arithmetic. This results from the fact that constraint solving directly using floating-point arithmetic is inefficient — the constraint solving cost of Float accounts for 10.9% in the total testing time while the cost of constraint solving with Real only accounts for 1.7%. Thus, constraint solving using reals is efficient and thus more practical for testing in comparison to solving constraints by directly using floating-point numbers.

6 RELATED WORK

Concolic testing. Concolic testing [17, 42], also known as dynamic symbolic execution, automatically generates test inputs via performing symbolic execution dynamically along with a concrete execution. Since its birth, concolic testing has been a great success to test a variety of *sequential* applications like web applications [5], sensor network applications [36], Unix utilities [9], database applications [14, 34], and embedded software [24]. Also concolic testing has been applied to shared-memory parallel programs including GPU programs [11, 28, 30, 31], multi-threading programs in C and Java [15, 40, 41]. None of the above tackle distributed-memory applications. jCUTE [39] instead applies concolic testing to boost the branch coverage as well as to detect deadlock in distributed Java programs. While jCUTE does not tackle large scale MPI programs that are commonly used in the HPC area, Hermes [23] detects communication deadlocks in MPI programs using dynamic symbolic verification that is similar to concolic testing. MPISE [16] and MPI-SV [51] deals with the non-determinism of message passing in MPI programs and mainly focus on communication deadlock detection based on concolic testing, and they are both built on top of CLOUD9 [7] — a parallel version of concolic testing tool KLEE [9]. Unlike the above, COMPI [29], built on CREST [8], aims to be a practical concolic testing tool that can help MPI program developers improve their code quality via achieving high branch coverage and manifesting runtime bugs like assertion violation, segmentation faults, and infinite loops.

Efficiency. The efficiency of concoclic testing is mainly determined by the search strategy and the time cost per test/execution (the time cost to evaluate a execution path). A variety of search strategies have been proposed. They generally fall into two categories: (1) search strategies guiding the search to branches that lead to the largest coverage gain [10, 19, 35, 50] or yet uncovered branches [8], and (2) strategies skipping redundant constraints [19, 29, 43]. Regarding the method of controlling time cost per execution, previous research mainly limits the execution time by restricting the number of symbolic input values as the execution time cost of many applications increases as the the number of symbolic input values used increase. For example, Burnim and Sen [8] evaluated CREST using three string manipulation programs using short strings – each character in the string is one symbolic value. Kim et al. [24] applied CREST-BV and KLEE to an image processing application using small image files – each pixel is a input value. However, these methods do not apply to MPI programs like HPL, IMB-MPI1, and SUSY-HMC whose execution time is directly related to input values instead of the number of input values. Though COMPI [29] alleviated the issue by limiting the input values generated by the constraint solver, it is challenging for developers to select the right limits as detailed above. Our work hence proposed input tuning to greatly improve the testing efficiency via automatically search for small input values for each test, which avoids the manual selection of limits for developers.

Floating-point support. Floating-point computations are commonly used in MPI applications. Thus it is a must to enable the floating-point reasoning in a practical concolic testing tool for MPI programs. For general programs, the study of floating-point support in concolic testing can be classified into two categories based on their testing goals. The first category of is the research that checks a certain property or type of bug, e.g. KLEE-FP [11] and KLEE-CL [12] supports floating-point reasoning to evaluate program equivalence and Ariadne [6] makes use of real arithmetic to expose floating-point exceptions. Unlike these, the goal of this work is to support floating-point arithmetic to strengthen the test input generation and thus improve program coverage. The second category that includes CORAL [46] and FloPSy [27] instead shares the same goal as our work. Our work differs from CORAL and FloPSy in that we approximate the floating-point constraint solving using real values while they use search-based methods. It is worth mentioning that directly solving floating-point constraints has been supported to perform general-purpose concolic testing recently [32]. This work pursues accuracy while our work trades off accuracy for performance.

Other symbolic execution based techniques for MPI applications. A variety of approaches exist to aid the defect detection in MPI programs, e.g., dynamic analysis [26], formal methods [44, 52], and symbolic execution [45, 52]. Symbolic execution based methods are most closely related to our work. CIVL [52] and TASS [45] are symbolic execution based tools that uncover software bugs such as data races, deadlocks and assertion violations in MPI programs. These works are complimentary to our approach as they employ symbolic execution alone as opposed to concolic testing. Concolic testing distinguishes itself from traditional symbolic execution via simplification of symbolic constraints by using concrete values. The cost of concolic testing is the sacrifice of completeness as some constraints can be lost or not recorded precisely due to the use of concrete values. What distinguishes our work from above works is that it focuses on efficiently performing branch coverage based testing of real-world MPI applications.

7 CONCLUSION

COMPI faces two major limitations that hinder its testing effectiveness. First, the input values generated by COMPI do not guarantee cost-effective testing. Second, floating-point data types and operations are not supported and thus coverage loss can occur. We propose *input tuning* to achieve cost-effective testing by favoring small input values. Also, we provide *floating-point* support and argue that the efficiency of constraint solving as well as testing could be significantly boosted if solving using reals instead of floating-point numbers. Evaluation results demonstrate that *input tuning* achieves high branch coverages much quicker than when it is not used. We further demonstrate that floating-point extension using reals helps us achieve higher coverage and solving constraints using reals is a better fit for practical testing compared with direct use of floating-point numbers.

ACKNOWLEDGMENTS

This work is supported by the NSF grants CNS-1617424, CCF-1524852, and CCF-1513201 to UC Riverside.

REFERENCES

[1] [n. d.]. Catching bugs with the Automated Testing System. https://computation.llnl.gov/research/mission-support/WCI/automated-testing-system
[2] [n. d.]. HPL: a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/
[3] [n. d.]. Intel MPI Benchmarks User Guide. https://software.intel.com/en-us/imb-user-guide
[4] [n. d.]. The Yices SMT Solver. http://yices.csl.sri.com/
[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 261–272.
[6] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 549–560.
[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the conference on Computer Systems*. ACM, 183–198.
[8] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 443–446.
[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.
[10] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1244–1254.
[11] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2011. Symbolic cross-checking of floating-point and SIMD code. In *Proceedings of the sixth conference on Computer systems*. ACM, 315–328.
[12] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2011. Symbolic testing of OpenCL code. In *Haifa Verification Conference*. Springer, 203–218.
[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[14] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 151–162.
[15] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 353–365.

[16] Xianjin Fu, Zhenbang Chen, Yufeng Zhang, Chun Huang, Wei Dong, and Ji Wang. 2015. MPISE: Symbolic execution of MPI programs. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*. IEEE, 181–188.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

[18] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[19] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[20] Christian Hovy and Julian M Kunkel. 2016. Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code.. In *SE-HPCCSE@ SC*. 1–8.

[21] Upulee Kanewala and James M Bieman. 2014. Testing scientific software: A systematic literature review. *Information and software technology* 56, 10 (2014), 1219–1232.

[22] Diane Kelly and Rebecca Sanders. 2008. The challenge of testing scientific software. In *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)*. Citeseer, 30–36.

[23] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. 2018. Dynamic Symbolic Verification of MPI Programs. In *International Symposium on Formal Methods*. Springer, 466–484.

[24] Yunho Kim, Moonzoo Kim, Young Joo Kim, and Yoonkyu Jang. 2012. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 1143–1152.

[25] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. 2013. Automated unit testing of large industrial embedded software using concolic testing. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 519–528.

[26] Ignacio Laguna, Dong H Ahn, Bronis R de Supinski, Todd Gamblin, Gregory L Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, et al. 2015. Debugging high-performance computing applications at massive scales. *Commun. ACM* 58, 9 (2015), 72–81.

[27] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. 2010. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*. Springer, 142–157.

[28] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 215–224.

[29] Hongbo Li, Zizhong Chen, and Rajiv Gupta. 2018. COMPI: Concolic Testing for MPI Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, 865–874.

[30] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2012. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 1–10.

[31] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical symbolic race checking of GPU programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 179–190.

[32] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-point symbolic execution: A case study in N-version programming. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 601–612.

[33] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.

[34] Kai Pan, Xintao Wu, and Tao Xie. 2011. Generating program inputs for database application testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 73–82.

[35] Sangmin Park, BM Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. 2012. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 35.

[36] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 186–196.

[37] David Schaich and Thomas DeGrand. 2015. Parallel software for lattice N= 4 supersymmetric Yang–Mills theory. *Computer Physics Communications* 190 (2015), 200–212.

[38] Koushik Sen and Gul Agha. 2006. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 339–356.

[39] Koushik Sen and Gul Agha. 2006. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 339–356.

[40] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*. Springer, 419–423.

[41] Koushik Sen and Gul A Agha. 2006. *Concolic testing of multithreaded programs and its application to testing security protocols*. Technical Report.

[42] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.

[43] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2635868.2635872

[44] Stephen F Siegel. 2007. Model checking nonblocking MPI programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 44–58.

[45] Stephen F Siegel and Timothy K Zirkel. 2011. TASS: The toolkit for accurate scientific software. *Mathematics in Computer Science* 5, 4 (2011), 395–426.

[46] Matheus Souza, Mateus Borges, Marcelo dâĂŹAmorim, and Corina S Păsăreanu. 2011. CORAL: solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*. Springer, 359–374.

[47] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs (TAP'08)*. Springer-Verlag, Berlin, Heidelberg, 134–153. http://dl.acm.org/citation.cfm?id=1792786.1792798

[48] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis De Supinski, and Andreas Sæbjørnsen. 2006. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. ACM, 27–36.

[49] X. Wu, Z. Xu, D. Yan, T. Wu, J. Yan, and J. Zhang. 2016. The Floating-Point Extension of Symbolic Execution Engine for Bug Detection. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 265–272.

[50] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. Citeseer, 359–368.

[51] Hengbiao Yu. 2018. Combining symbolic execution and model checking to verify MPI programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 527–530.

[52] Manchun Zheng, Michael S Rogers, Ziqing Luo, Matthew B Dwyer, and Stephen F Siegel. 2015. CIVL: formal verification of parallel programs. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 830–835.