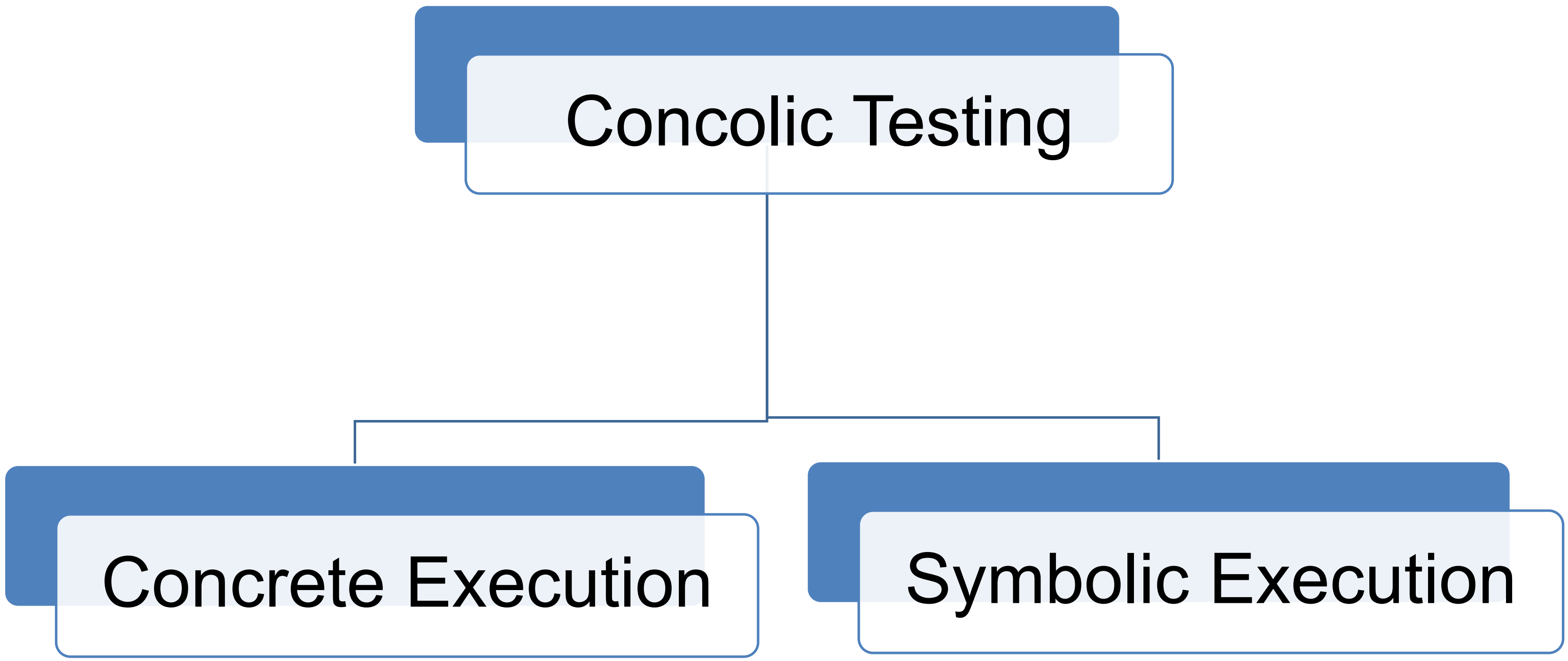
Efficient Concolic Testing of MPI Applications

Hongbo Li

Zizhong Chen Rajiv Gupta

CC'19, Washington DC, USA

Feb. 17, 2019


Concolic Testing
Concrete Execution
Symbolic Execution

It Is Popular

- Programming languages:
  - Binary machine code, C, Java, and JavaScript.
- Application types:
  - web applications, sensor network applications, Unix utilities, database applications, and embedded software, GPU programs, image processing software, and so on
- Various tools:
  - KLEE, DART, SAGE, PEX, jCute, CREST, Jalangi, etc.

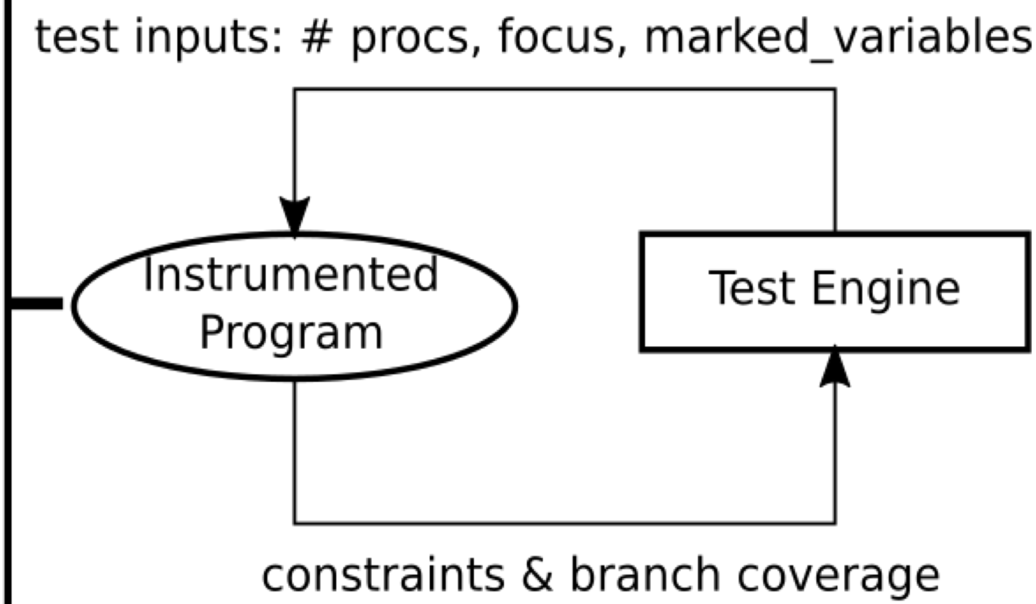
COMPI [IPDPS 2018]

- COMPI is a concolic testing tool for MPI programs with following major features:
  - Deals with basic MPI semantics
  - Deals with high testing cost caused by input values, parallelism, and loops

- COMPI achieves 69-86% branch coverage within a few hours

Concolic Testing

```
int x;
mark_symbolic(x);
...
__load_symbol(x);
__load_value(0);
__apply('>');
if (x > 0) {
    __log_constraint("x > 0");
...
} else {
    __log_constraint("x <= 0");
...
}
```



(1) Instrumentation

(2) Iterative testing

Issues of COMPI

Our Solutions

Evaluation

Issues of COMPI

Our Solutions

Evaluation

Issues of COMPI

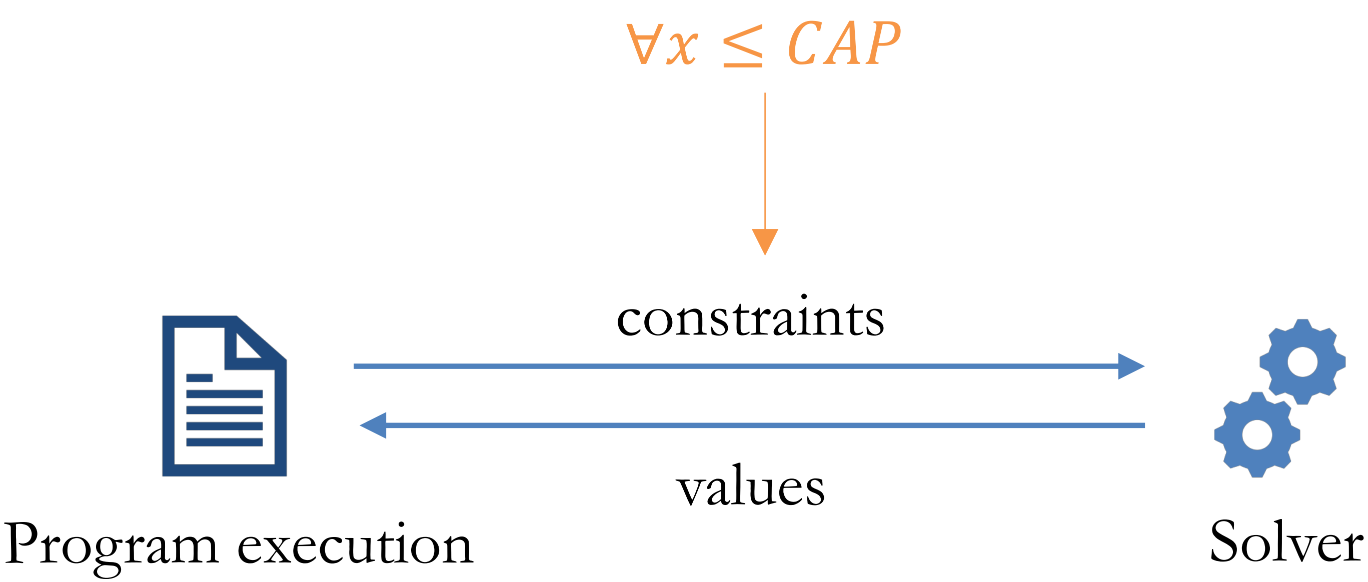
- **Input generation** does not guarantee cost-effective testing

- **Floating point** data types and operations are not supported

Issue I

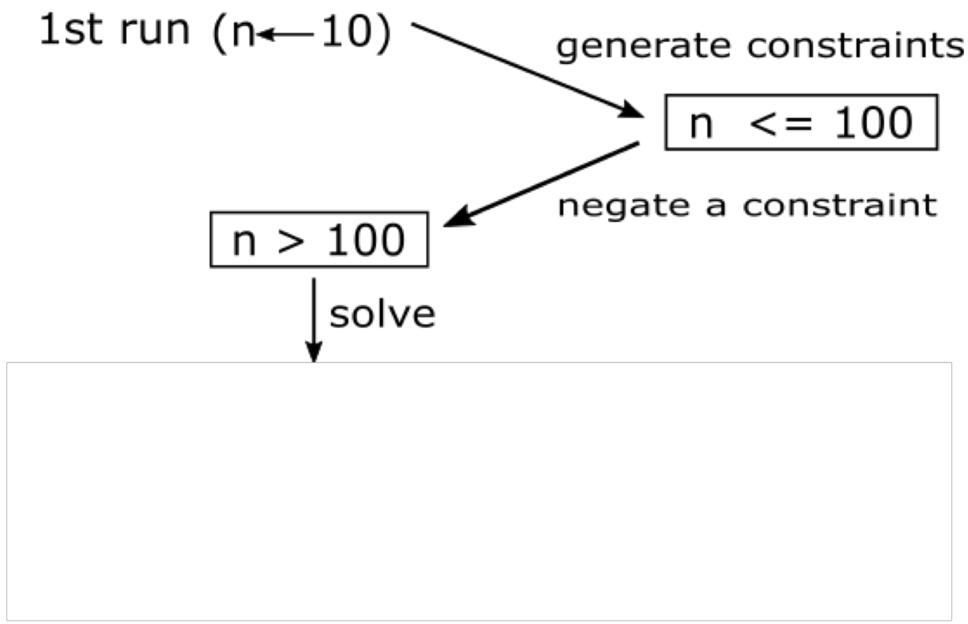
- Larger input values → Longer execution

- Solution of COMPI: Input Capping



Example

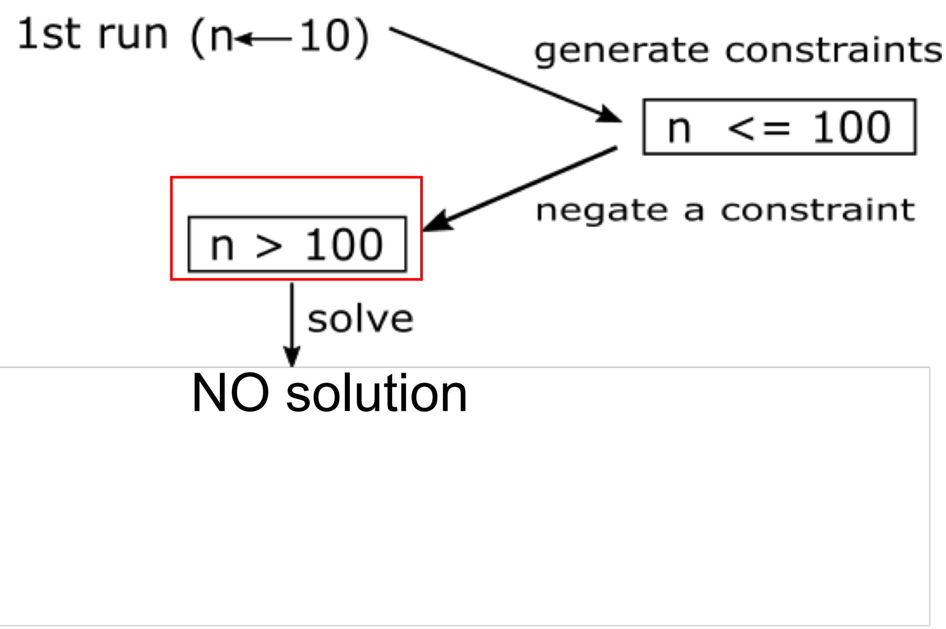
```
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```



An MPI program performing matrix multiplication.

Small Cap

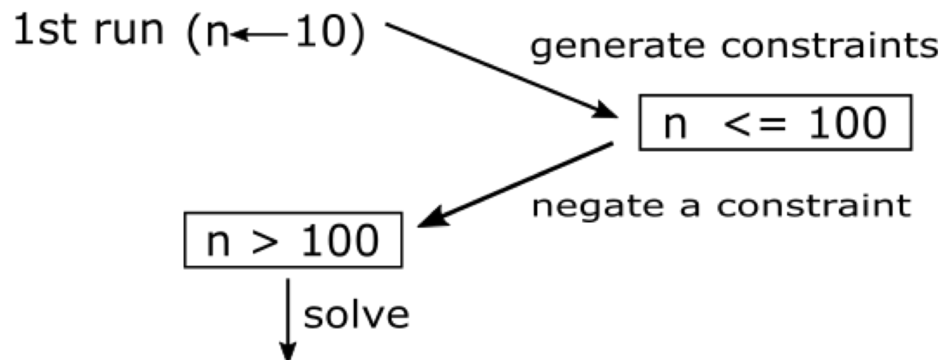
```
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```



$CAP = 50$ → Fail to cover the *else* branch

Big Cap

```
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```

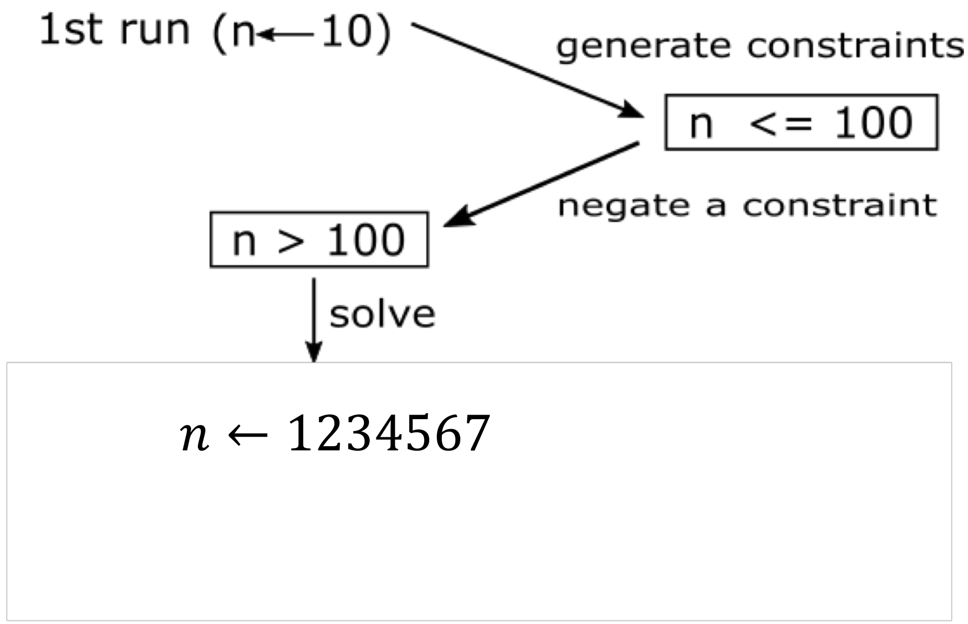


$n \leftarrow$ C, where $100 < C \leq 500$

$CAP = 500$ → High testing cost

No Input Capping

```
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```



No Capping → Execution failure

Issue II

- Floating-point data types and operations are not supported

```
main () {
    ...
    int a; float b;
    COMPI_int(a);
    ...
    if (b > 1.1) f1();
    else f2();
    float c = a * 1.1;
    if (c > 2) f3();
    else f4();
}
```

- Unable to mark b
  - Fixing it to a value → either *f1* or *f2* could not be explored
- Unable to record $a * 1.1$
  - Symbolic representation of c is not existing → either *f3* or *f4* could not be explored

Issues of COMPI

Our Solutions

Evaluation

Our Solutions

- **Input tuning →** cost effective testing

- **Floating-point extension →** exploration of branches related to the use of floating-point arithmetic

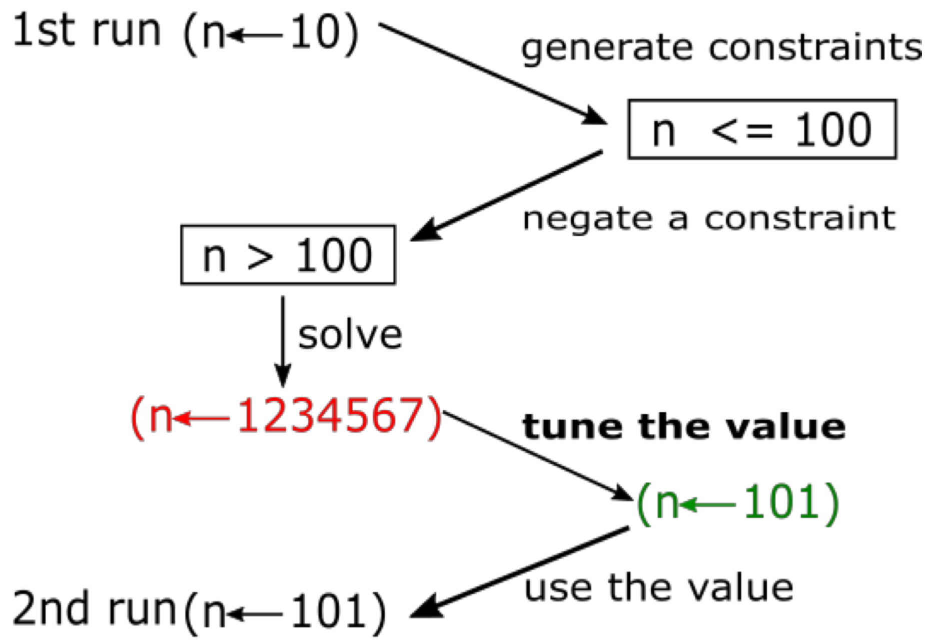
Our Solutions

- **Input tuning →** cost effective testing

- **Floating-point extension →** exploration of branches related to the use of floating-point calculations
  - Constraint solving using reals instead of floating-point numbers → faster constraint solving

Input Tuning

```
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```



Input Tuning

Binary search of $\boldsymbol{upper}$ in $(0, 1234567)$ satisfying:

$\{n > 100\} \cup \{n \leq \boldsymbol{upper}\}$ is solvable

AND $\{n > 100\} \cup \{n \leq \boldsymbol{upper} - \mathbf{1}\}$ is unsolvable

Tuning

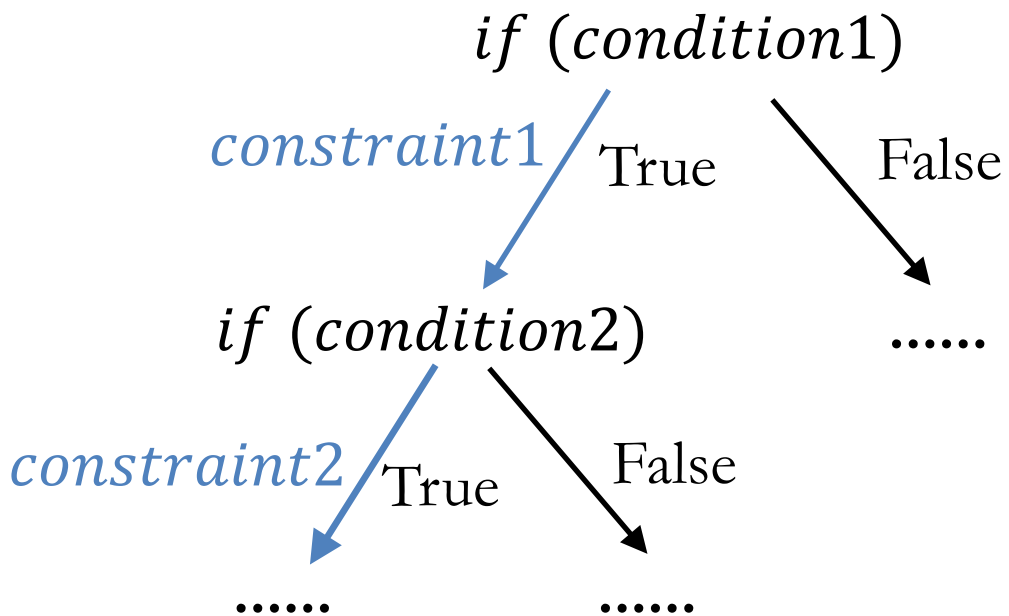
$\boldsymbol{n} \leftarrow \mathbf{1234567}$

Solution by solver for $\{n > 100\}$

$\boldsymbol{n} \leftarrow \mathbf{101}$

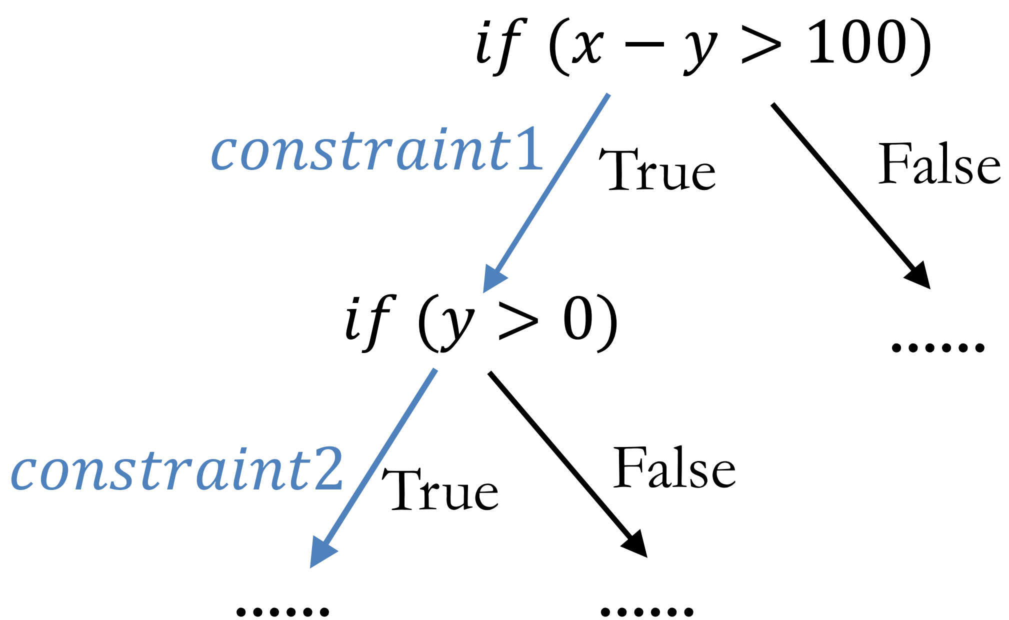
Tuned solution

Input Tuning for Multi-variable Multi-constraint Case



Need to solve $\{constraint1, constraint2\}$

Input Tuning for Multi-variable Multi-constraint Case



Need to solve $\{x - y > 100, y > 0\}$

Input Tuning for Multi-variable Multi-constraint Case

$\{x = 4321, y = 1234\}$

Stage I Tuning

$\{x = 4321, y = 1234, x \leq upper1, y \leq upper1\}$

$\min\{upper\} = 102$

Stage II Tuning

$\{x = 4321, y = 1234, x \leq upper1, y \leq upper2\}$

$\min\{upper2\} = 1$

$\{x = 102, y = 1\}$

Tuning for $\{\boldsymbol{x - y > 100, y > 0}\}$

Input Tuning -- Summary

- Stage I avoids too large values being generated for **ALL** variables appearing in dependent constraints

- Stage II ensures the smallest value is generated for the **SINGLE** variable appearing in the target constraint based on Stage I

Our Solutions

- **Input tuning →** cost effective testing

- **Floating-point extension →** exploration of branches related to the use of floating-point arithmetic

Floating-Point Extension

- Two floating-point data types supported: *float*, *double*
- The extension adopts the design methodology of symbolic reasoning for integers
  - Instrument floating-point operations
  - Records only *linear constraints*
    - Non-linear constraints are simplified using concrete values, e.g., $x * y$ is recorded as $C * x$ with C being the concrete value of y

Floating-Point Extension

- Two solvers: Real v.s. Float
  - Accuracy: Real < Float
  - Solving speed: Real > Float

Real v.s. Float based on 100 iterative tests of a synthetic program that compares expression e with constant 0.

$e =$	x	$x + y$	$x + y + z$
Cost (float)	31.4s	75.0s	91.2s
Cost (real)	8.2s	8.1s	8.2s

3.8-11.1X faster

Limitations of COMPI

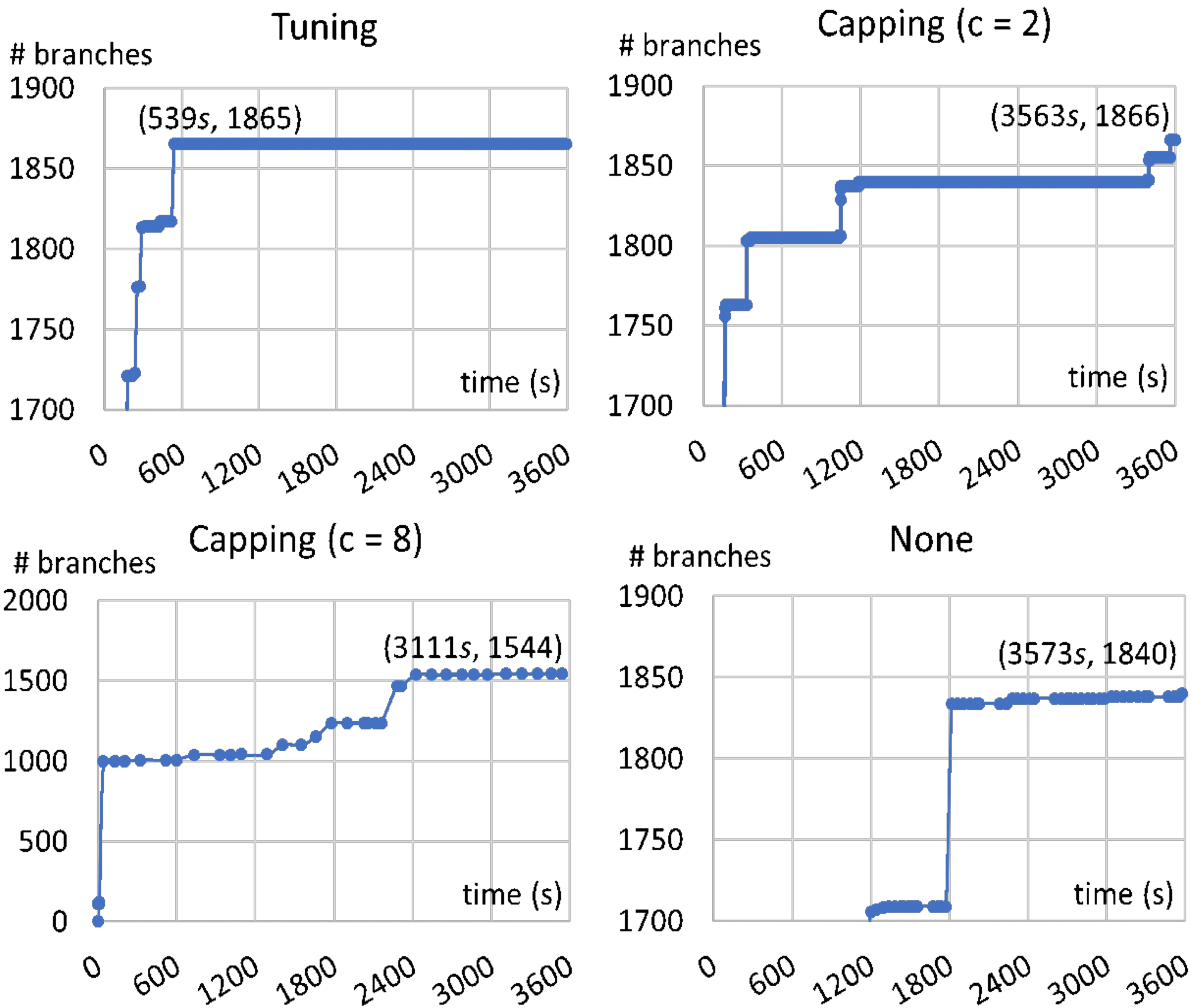
Our Solutions

Evaluation

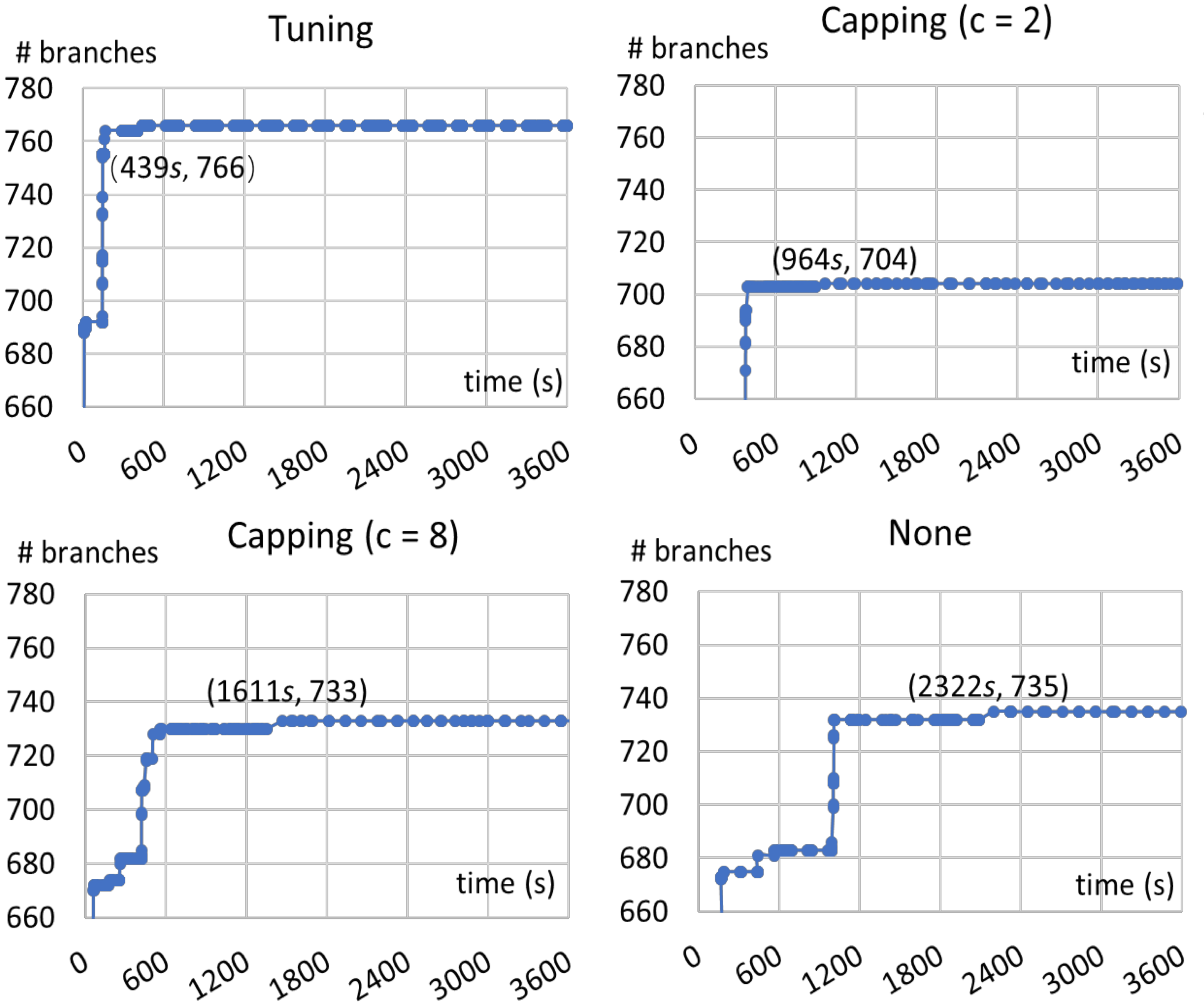
Evaluation

- Input tuning is evaluated using HPL, IMB-MPI1, and SUSY-HMC
- Floating-point extension is evaluated using SUSY-HMC
- One hour testing at each configuration
- Initial input values are 1 for all variable in the first test
- In the evaluation of input capping, we selects the same cap for all variables

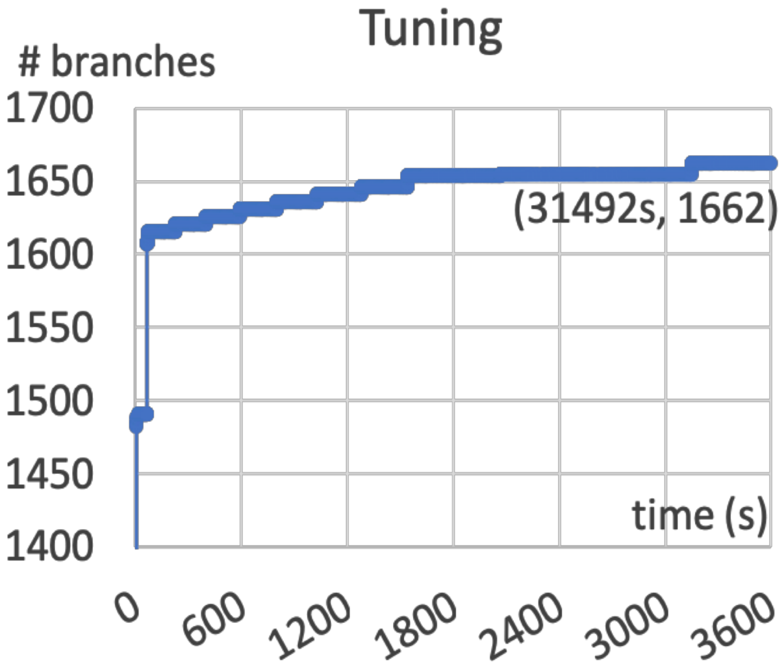
Input Tuning on HPL

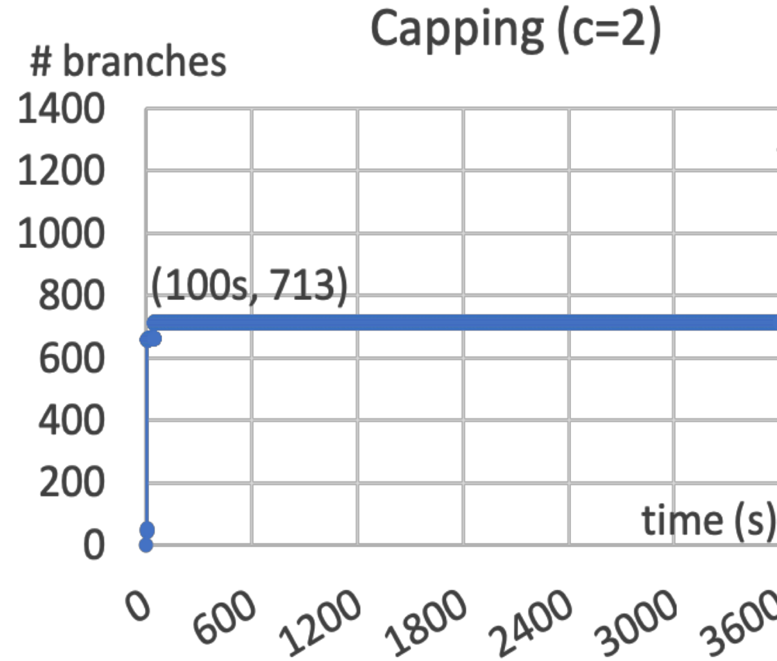


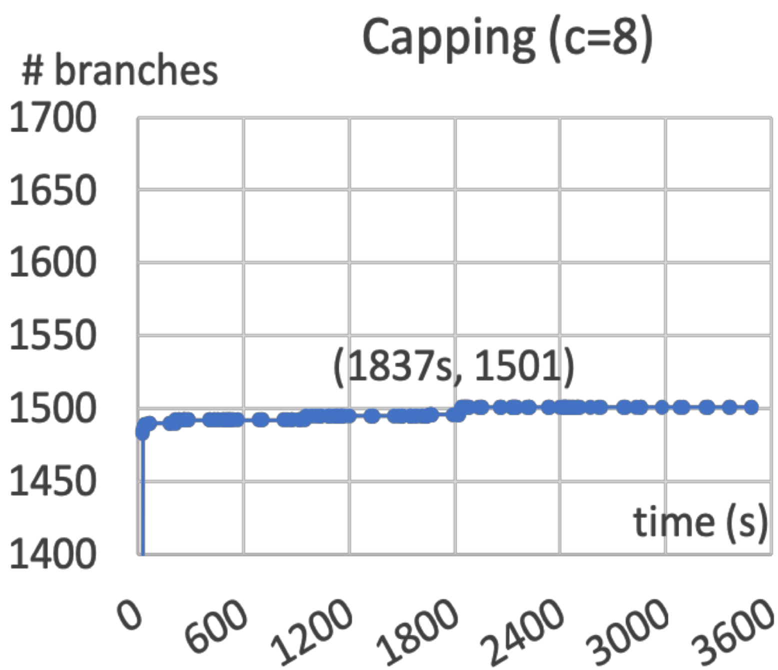
Input Tuning on IMB-MPI1



Input Tuning on SUSY-HMC





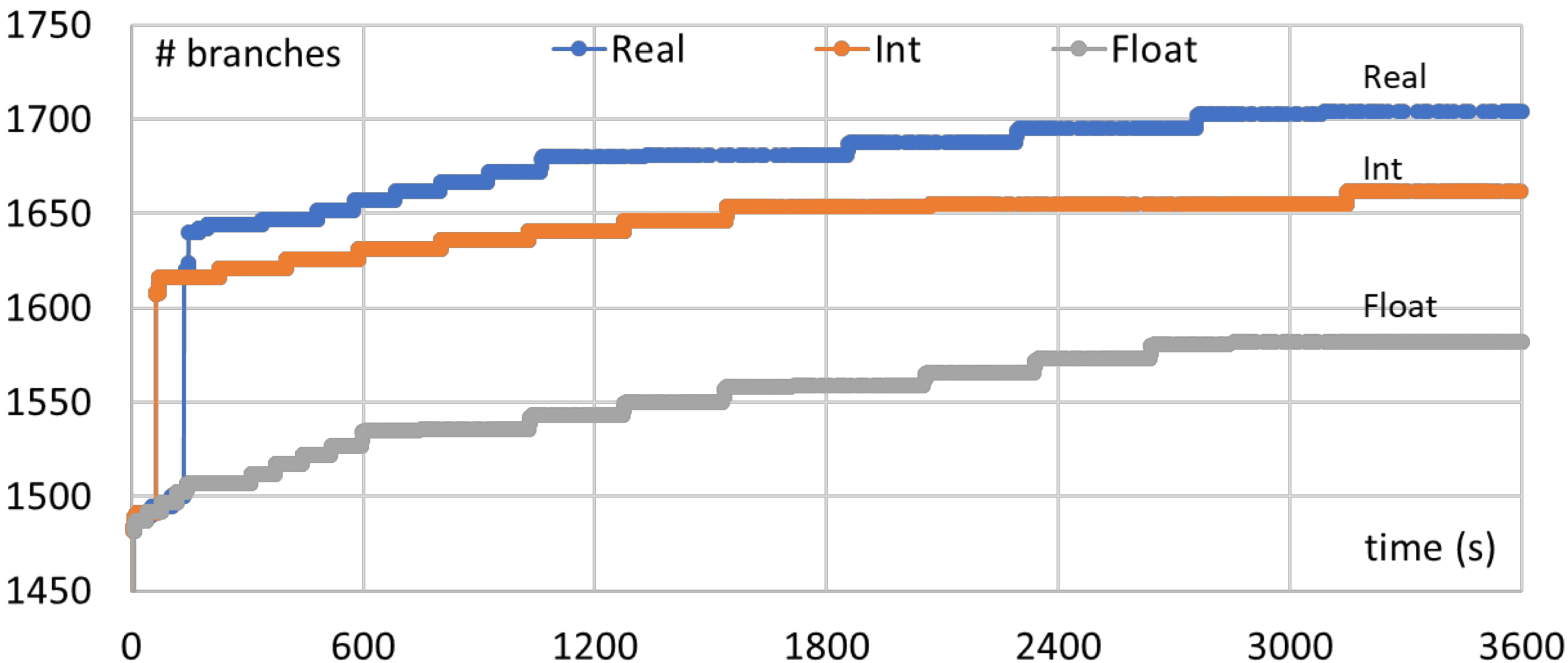


Input Tuning

- **10-minute** coverage (input tuning) ≥ **1-hour** coverage (other methods)

- SUSY-HMC: 1-hour coverage (input tuning) is about **1.2-2.3X** higher than 1-hour coverage (other methods)

Floating-point Extension



Floating-point Extension

- Real (1704) > Int (1662) > Float (1582)

- Constraint solving time of Real (1.7%) < Constraint solving time of Float (10.9%)

Conclusion

- Input tuning
  - **10-minute** coverage (input tuning) ≥ **1-hour** coverage (other methods)
  - SUSY-HMC: 1-hour coverage (input tuning) is about **1.2-2.3X** higher than 1-hour coverage (other methods)

- Floating-point Extension
  - Floating-point extension using reals achieve 42-122 more branches than the other two

Thank you!