MuTMuT: Efficient Exploration for **Mu**tation **T**esting of **Mu**lti**T**hreaded Code

Milos Gligoric, **Vilas Jagannath** and Darko Marinov

UPCRC Illinois Research Seminar

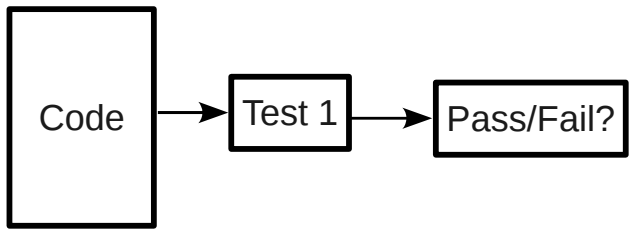
February 18th, 2010

Outline

Testing multithreaded code is costly

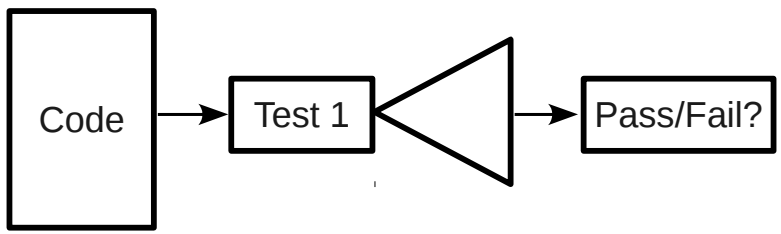
Traditional (sequential) testing:

- *input* $\longrightarrow$ *code* $\longrightarrow$ *expected output*
- Input has only one execution path (deterministic code)



Testing multithreaded code:

- *input*, **schedule** $\longrightarrow$ *code* $\longrightarrow$ *expected output*
- Input has many possible execution paths
- Exploration of large number of possible schedules



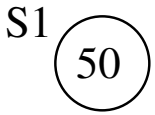
A Simple Banking Simulation

```
class Account {

    double balance;

    Account(double amount) {
        this.balance = amount;
    }

    synchronized void withdraw(double amount) {
        if (balance >= amount)
            balance -= amount;
    }

    synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

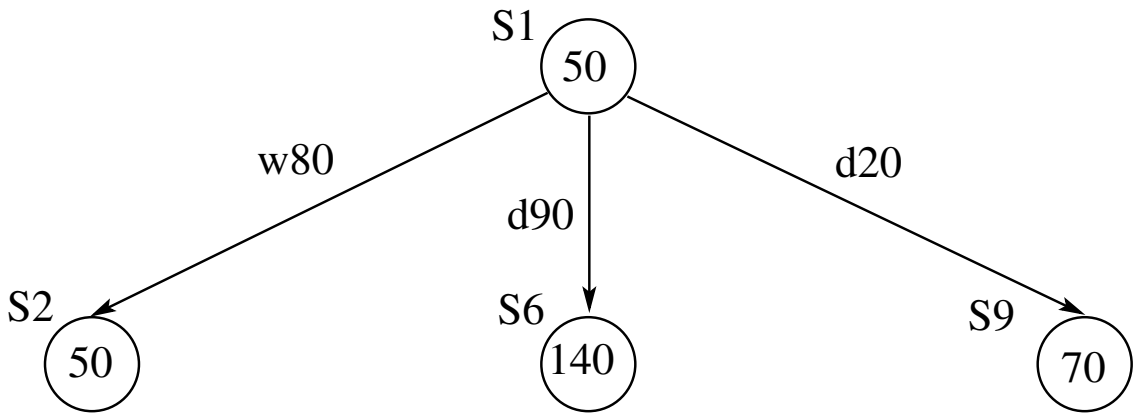
A Multithreaded Test for Account

```
class AccountTest {
    void testDepositWithdraw() {
        final Account a = new Account(50.00);
        Thread t1 = new Thread() {
            public void run() {
                a.withdraw(80.00); } };
        Thread t2 = new Thread() {
            public void run() {
                a.deposit(90.00); } };
        Thread t3 = new Thread() {
            public void run() {
                a.deposit(20.00); } };
        t1.start(); t2.start(); t3.start();
        t1.join(); t2.join(); t3.join();
        assert 80.00 == a.balance() || 160.00 == a.balance();
    }
    void testDepositDeposit() { ... }
    ...
}
```

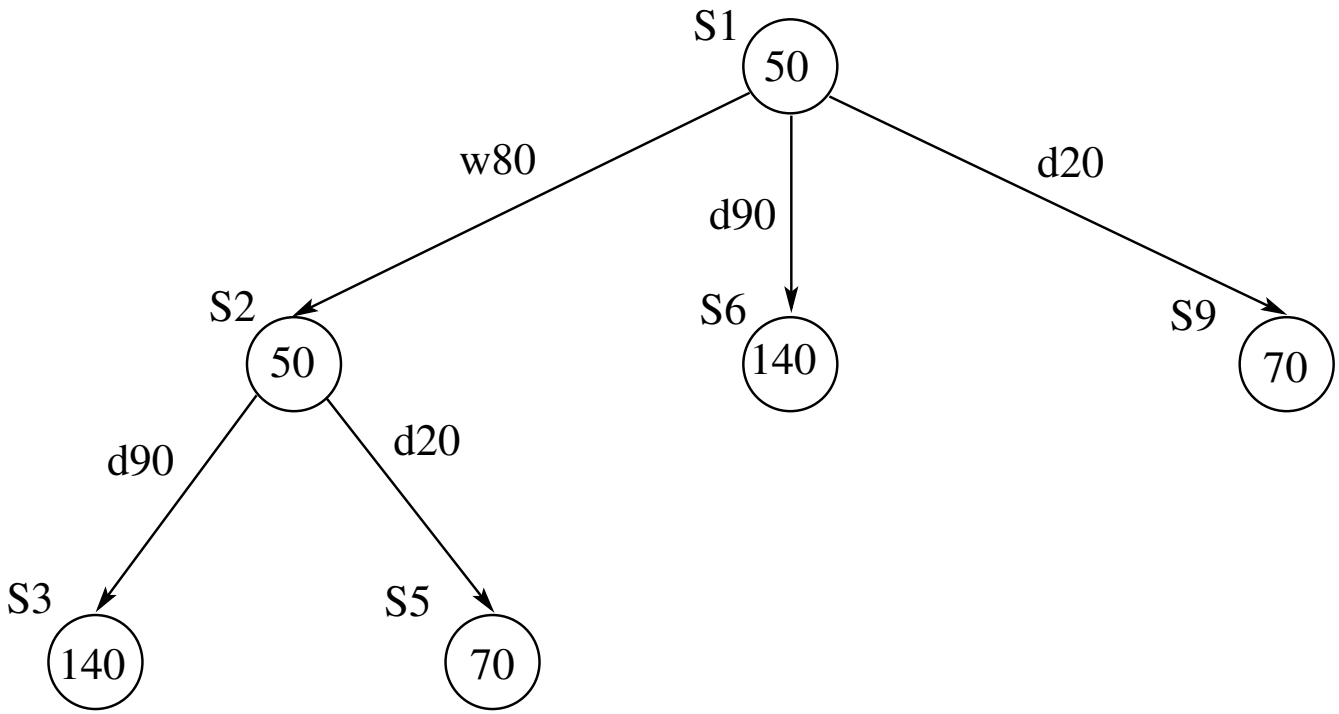
Test Execution/Exploration State Space



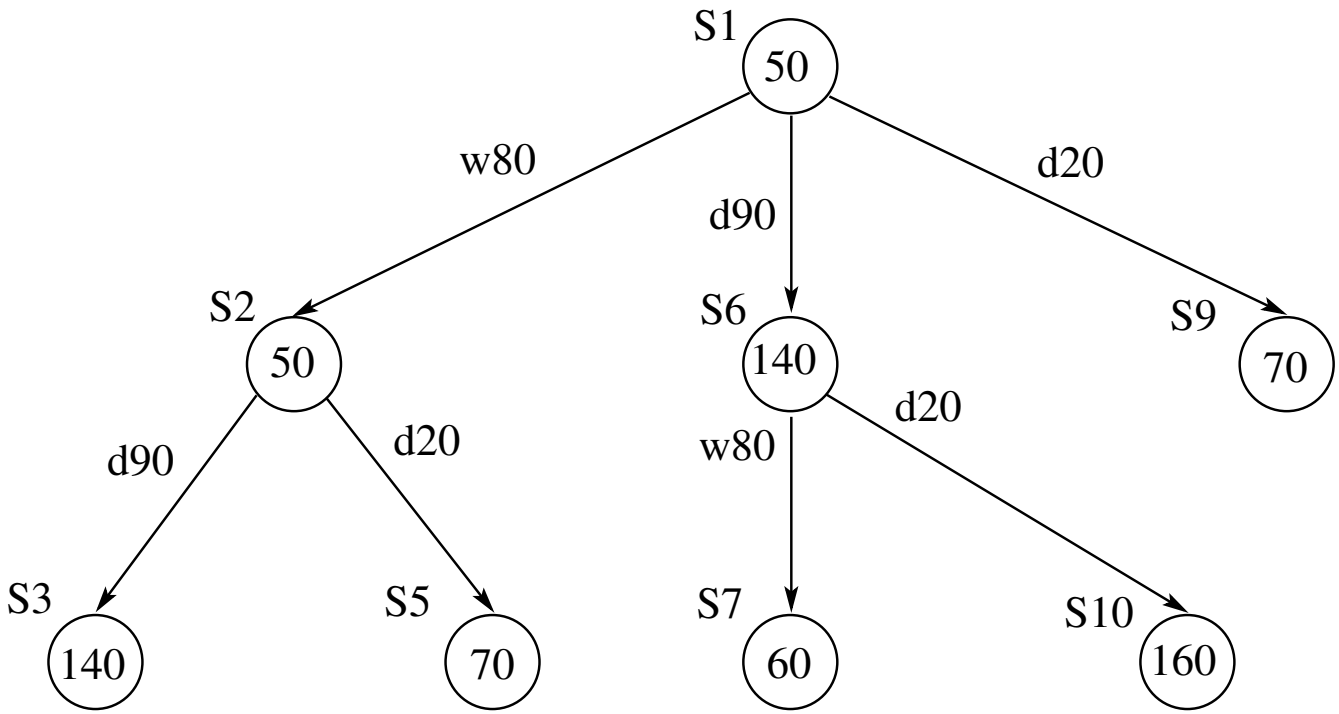
Test Execution/Exploration State Space



Test Execution/Exploration State Space

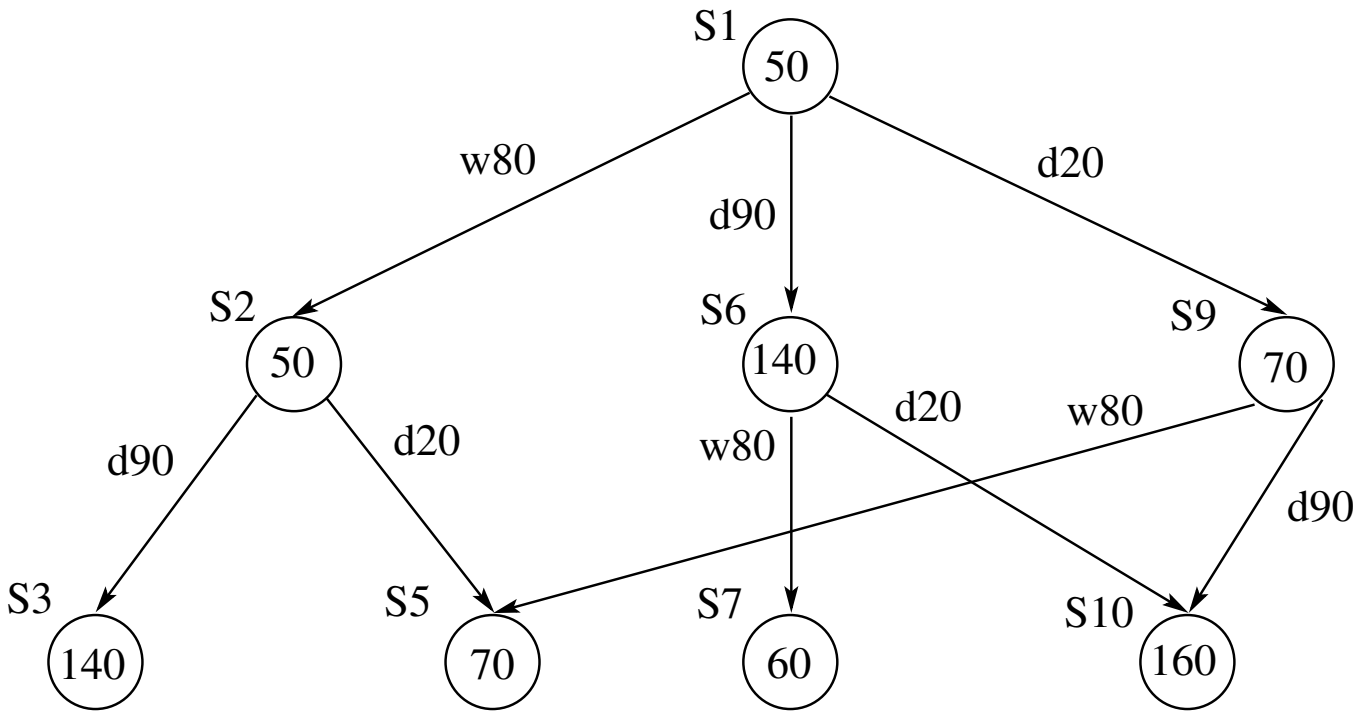


Test Execution/Exploration State Space

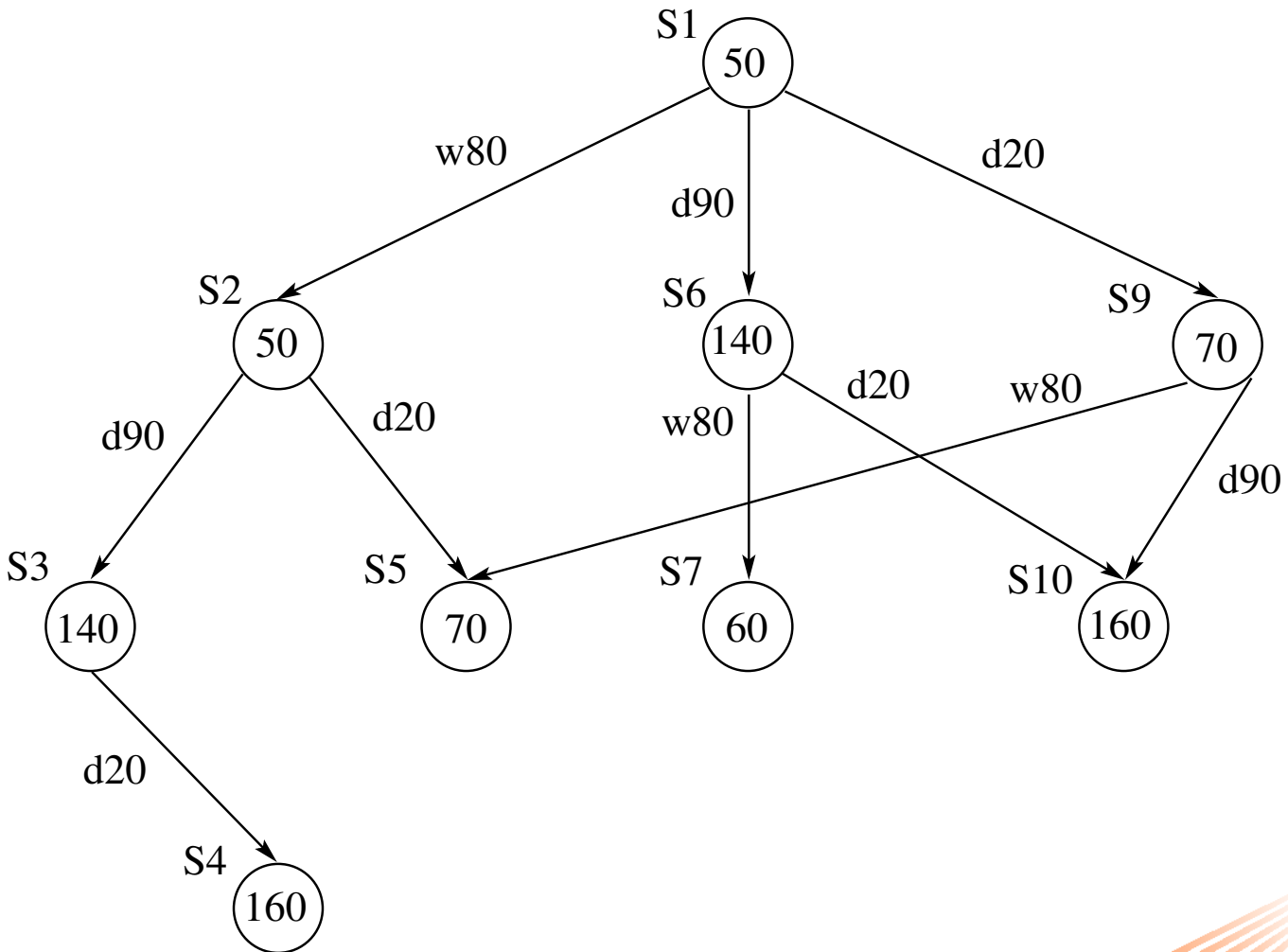


UPCRC Illinois
Universal Parallel Computing Research Center

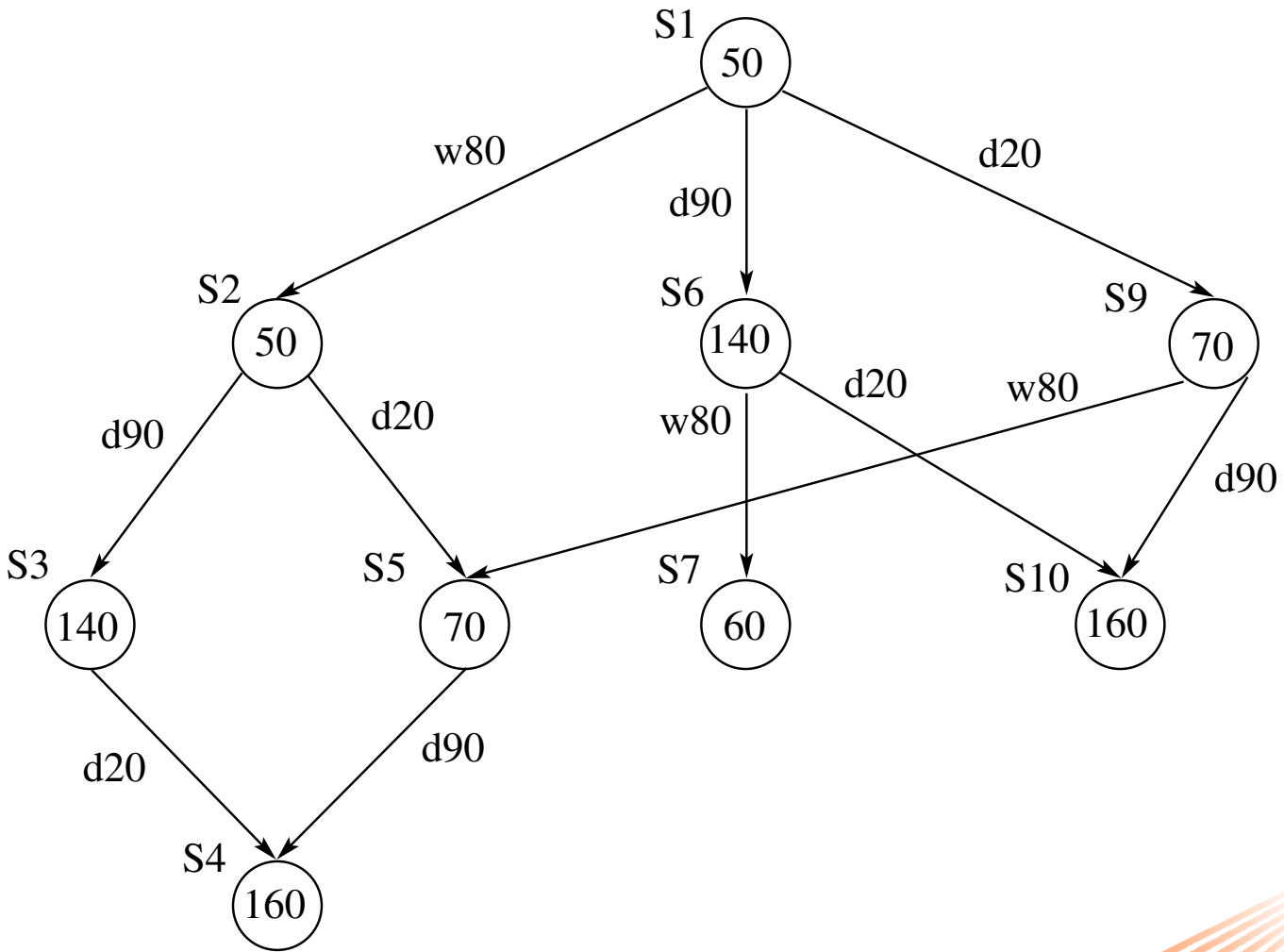
Test Execution/Exploration State Space



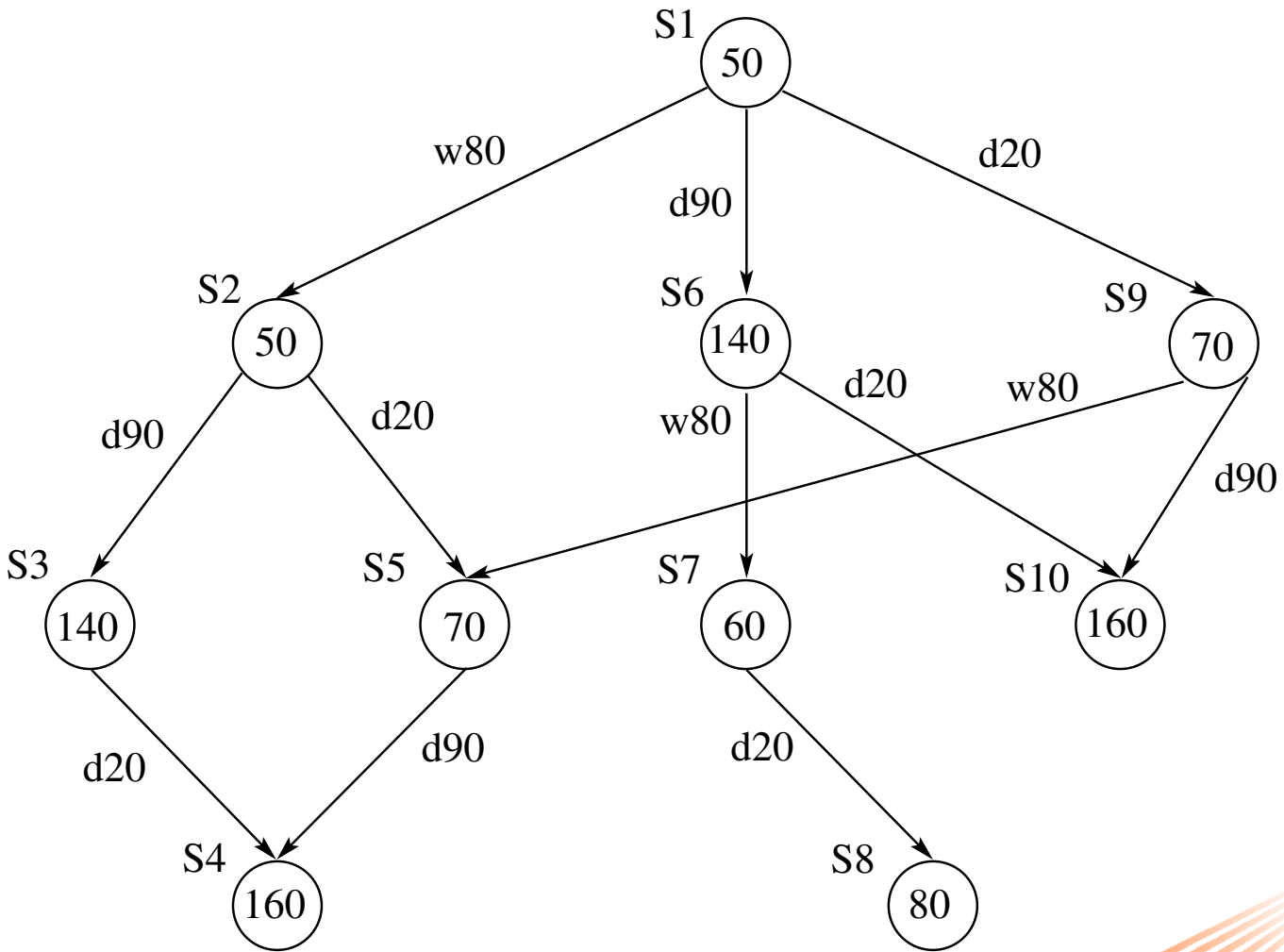
Test Execution/Exploration State Space



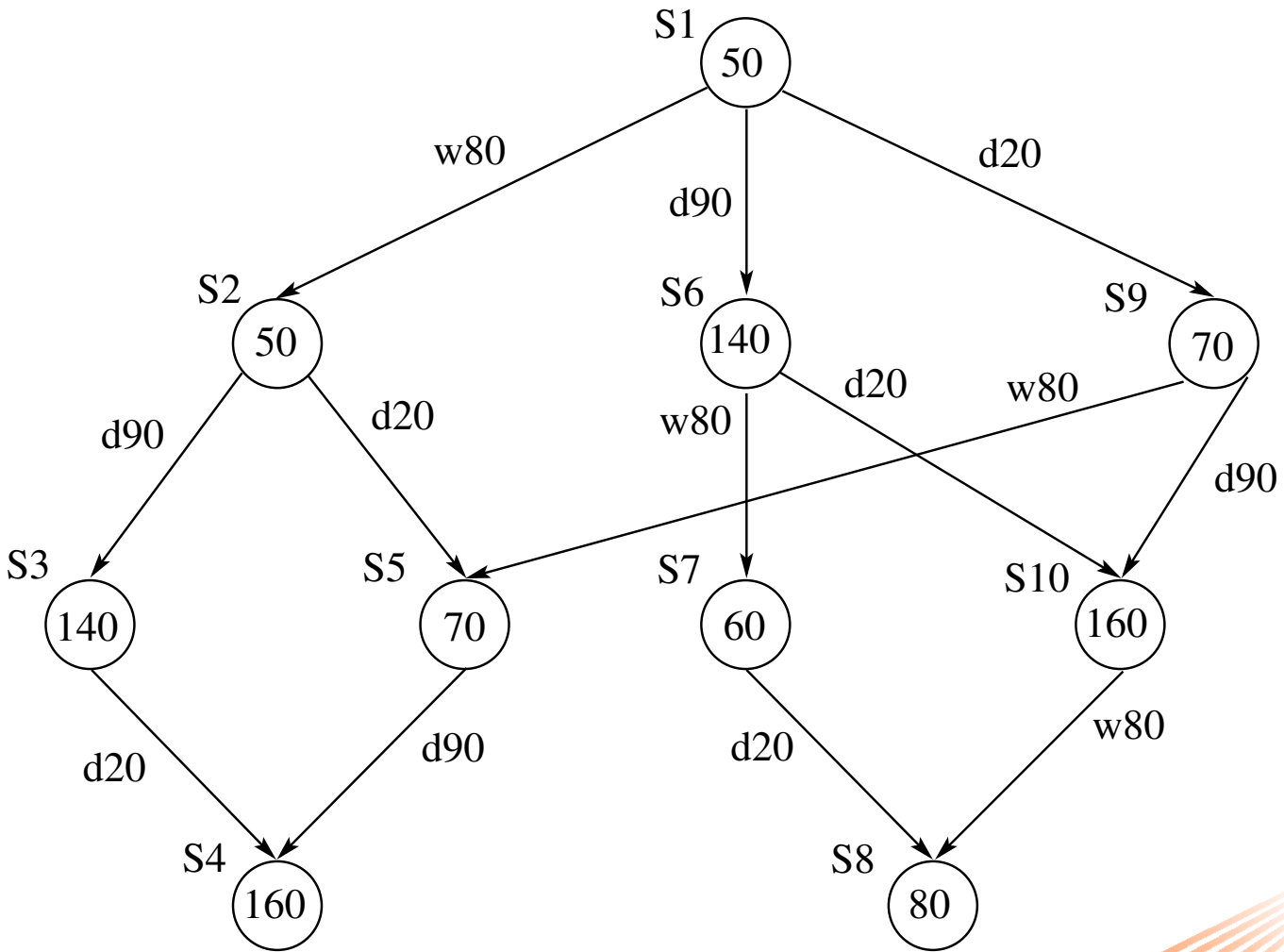
Test Execution/Exploration State Space



Test Execution/Exploration State Space



Test Execution/Exploration State Space



Current techniques focus on improving exploration

Some examples:

- CalFuzzer
- CHESS
- ConTest
- CTrigger
- ...

Focus on exploration of **one** version

However code evolves!

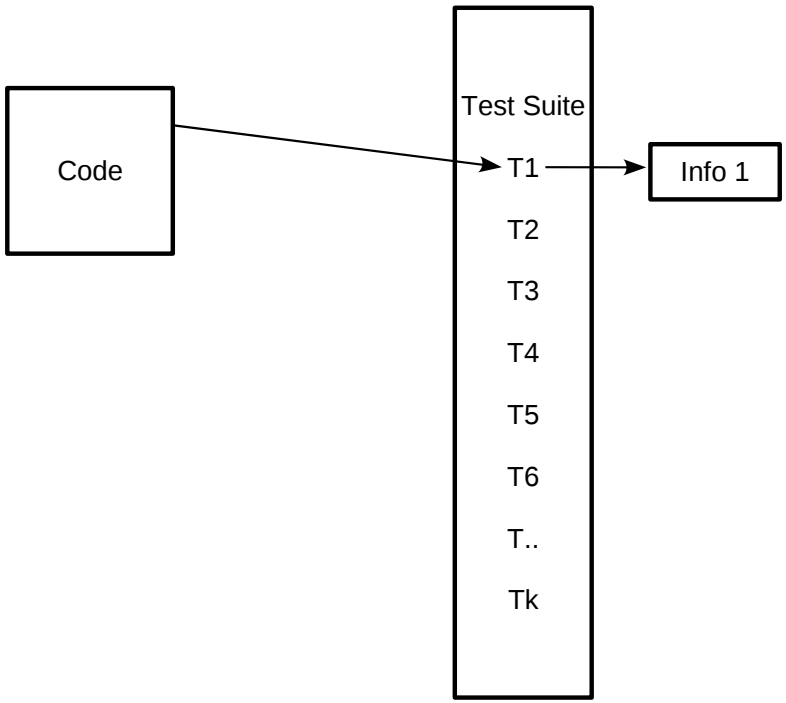
Exploration costs add up over time with each new version

Expensive to repeat complete testing for each new version

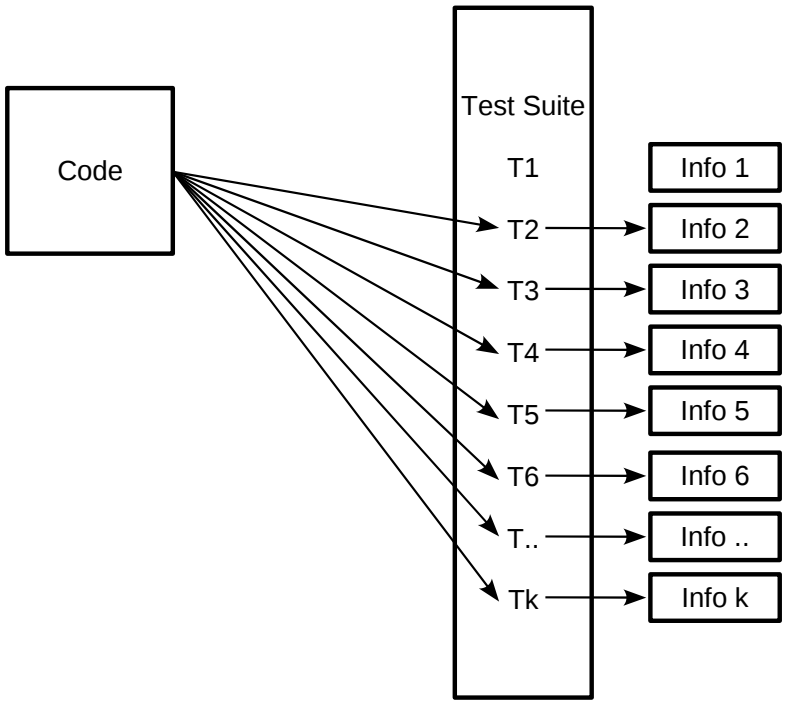
Reducing these costs thoroughly researched for sequential code, i.e., improving regression testing for sequential code

- Test selection [Graves et al. 2001, Harrold et al. 2001...]
- Test prioritization [Rothermel et al. 2001, Elbaum et al. 2002...]
- Impact analysis [Law et al. 2002, Ren et al. 2003...]
- Unitizing system tests [Orso et al. 2005, Saff et al. 2005...]

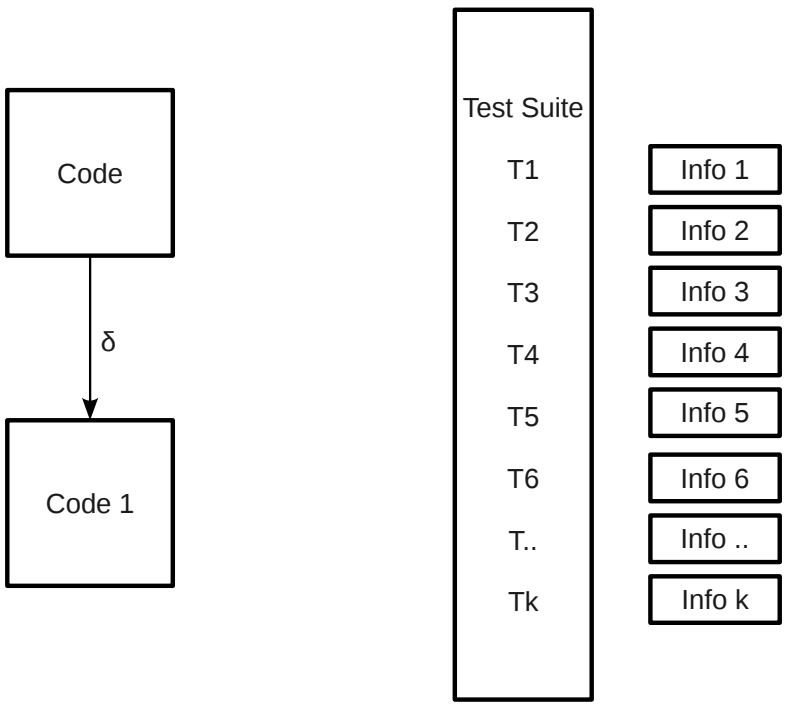
Regression testing for sequential code



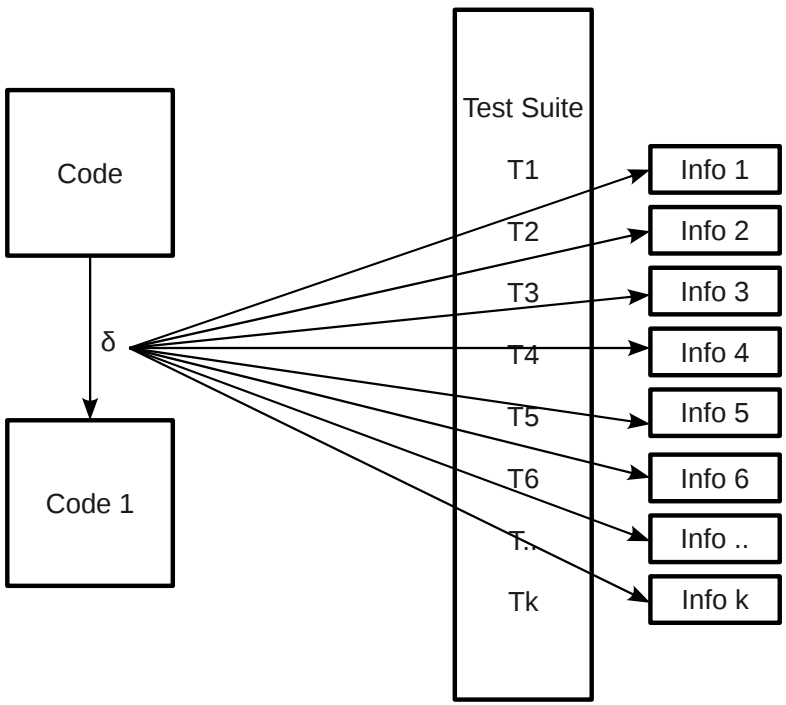
Regression testing for sequential code



Regression testing for sequential code

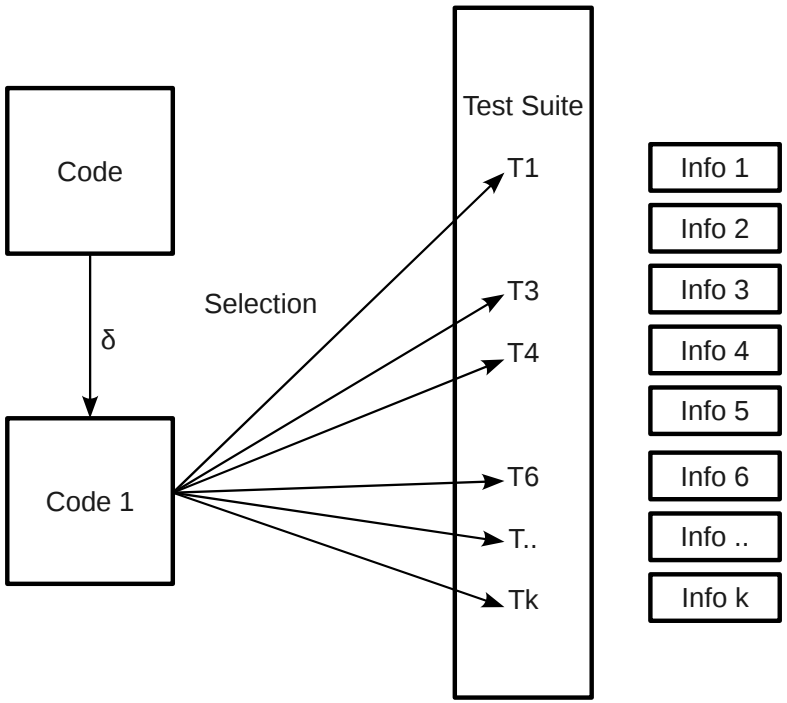


Regression testing for sequential code

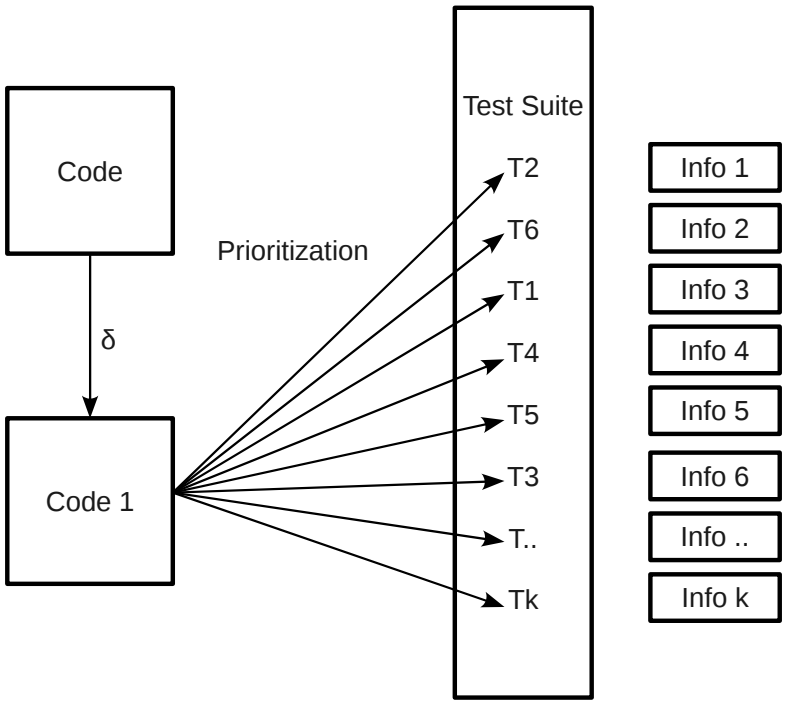


UPCRC Illinois
Universal Parallel Computing Research Center

Regression testing for sequential code



Regression testing for sequential code

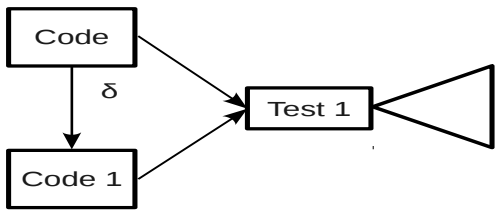


Previous work on incremental exploration

Reducing model checking and analysis costs for evolving code:

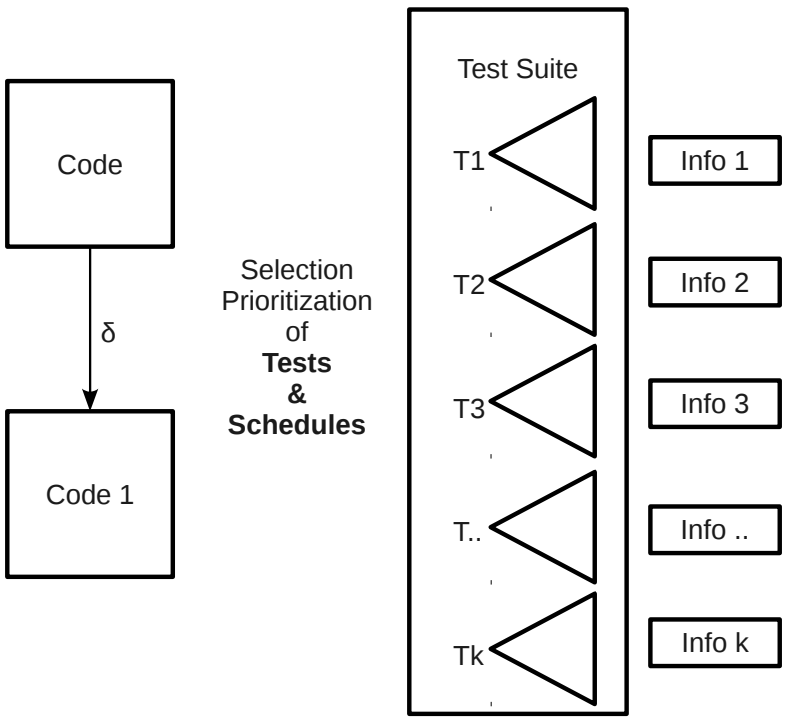
- Modal mu-calculus [Sokolowski and Smolka, 1994]
- Fixed-point properties [Makowsky and Rawe, 1995]
- Practical importance [Sistla, 1996]
- Extreme model checking [Henzinger et al. 2003]
- Checking safety properties [Conway et al. 2005]
- Incremental state space exploration

  [Lauterburg et al. 2008]



Most work does not handle multithreaded code and considers only one test

Regression testing for multithreaded code



Challenges for evaluation

Heuristics need to be evaluated with controlled experiments

Hard to collect artifacts (many versions, many tests, many bugs)

Sample costs for JBoss (sequential tests),
courtesy of Gregg Rothermel (UNL):

- 116K LOC, some existing JUnit tests, 10 versions
- 300 hours basic setup for build/experimentation
- 150 hours for fault documentation and seeding
- 120 hours for code instrumentation and other tools

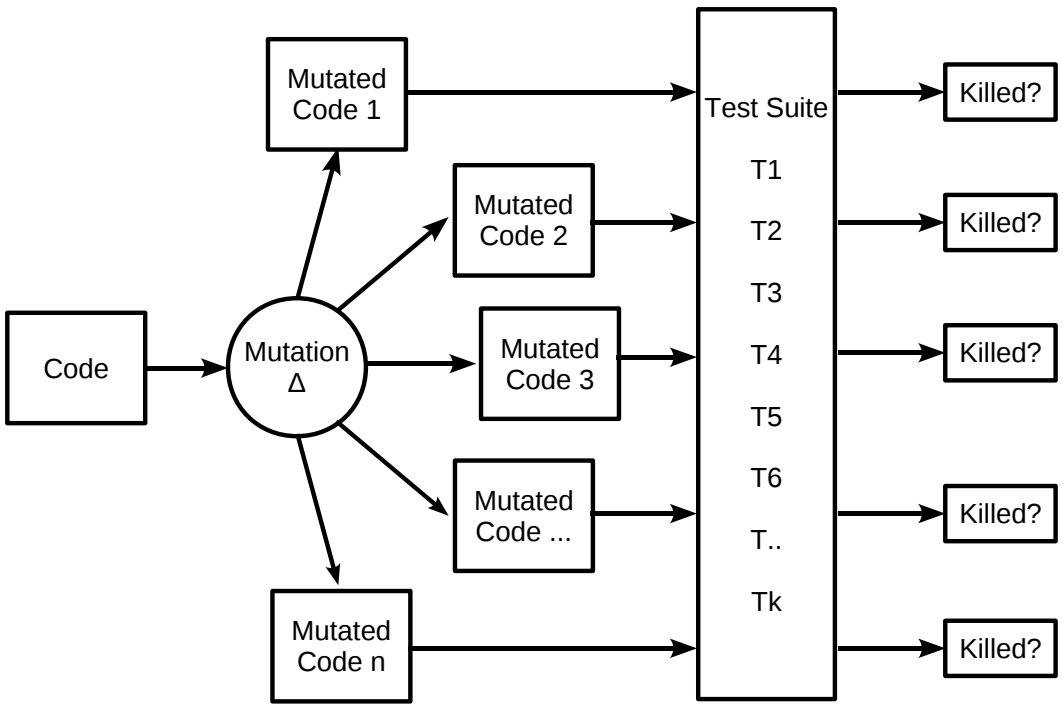
Current benchmarks include one version, one$\sim$ test, one$\sim$ bug

Today's talk: Simulate versions with mutation testing

Outline

What is Mutation Testing?

Methodology to **measure** and **improve** quality of a test suite



Example Mutations

For sequential programs:

- Negate boolean expression: $boolean\_expr \rightarrow !(boolean\_expr)$
- Change arithmetic operator: $expr + expr \rightarrow expr - expr$
- Add static modifier: $variable\_decl \rightarrow static\ variable\_decl$

Example Mutations

For sequential programs:

- Negate boolean expression: *boolean_expr* $\longrightarrow$!(*boolean_expr*)
- Change arithmetic operator: *expr* + *expr* $\longrightarrow$ *expr* − *expr*
- Add static modifier: *variable_decl* $\longrightarrow$ *static variable_decl*

For multi-threaded programs: [Bradbury et al. 2006]

- Replace notifyAll() with notify()
- Change timeout parameter: sleep(200) $\longrightarrow$ sleep(200*10)
- Remove synchronized keyword from method

Application of Mutations - Example

Original code

```
void withdraw(double amount) {
    if (balance >= amount)
        balance -= amount;
}
```

Some Mutants

```
// replace amount with 0
void withdraw(double amount) {
    if (balance >= amount)
        balance -= 0;
}

// replace -= with +=
void withdraw(double amount) {
    if (balance >= amount)
        balance += amount;
}

// replace amount with balance
void withdraw(double amount) {
    if (balance >= amount)
        balance -= balance;
}
```

Compilation Cost

Each mutation creates a new version of the code

Each mutant version has to be compiled

Reduced using mutant schemata [Untch et al. 1993]

Mutant schemata encode all mutations into one code version

Mutation Schemata - Example

without schemata

```
// original
void withdraw(double amount) {
    if (balance >= amount)
        balance -= amount;
}

// replace amount with 0
void withdraw(double amount) {
    if (balance >= amount)
        balance -= 0;
}

// replace -= with +=
void withdraw(double amount) {
    if (balance >= amount)
        balance += amount;
}

// replace amount with balance
void withdraw(double amount) {
    if (balance >= amount)
        balance -= balance;
}
```

with schemata

```
void withdraw(double amount) {
    if (balance >= amount) {
        if (MUTATION_ID == 0) {
            // original
            balance -= amount;
        } else if (MUTATION_ID == 1) {
            // replace amount with 0
            balance -= 0;
        } else if (MUTATION_ID == 2) {
            // replace -= with +=
            balance += amount;
        } else if (MUTATION_ID == 3) {
            // replace amount with balance
            balance -= balance
        }
    }
}
```

Execution/Exploration Cost i.e. MuTMuT motivation

Mutant schemata reduces static (compilation) cost

However, the dynamic cost (execution) remains

All tests have to be executed for each mutant

Moreover, for multithreaded code each test needs to be explored!

Execution/Exploration Cost i.e. MuTMuT motivation

Mutant schemata reduces static (compilation) cost

However, the dynamic cost (execution) remains

All tests have to be executed for each mutant

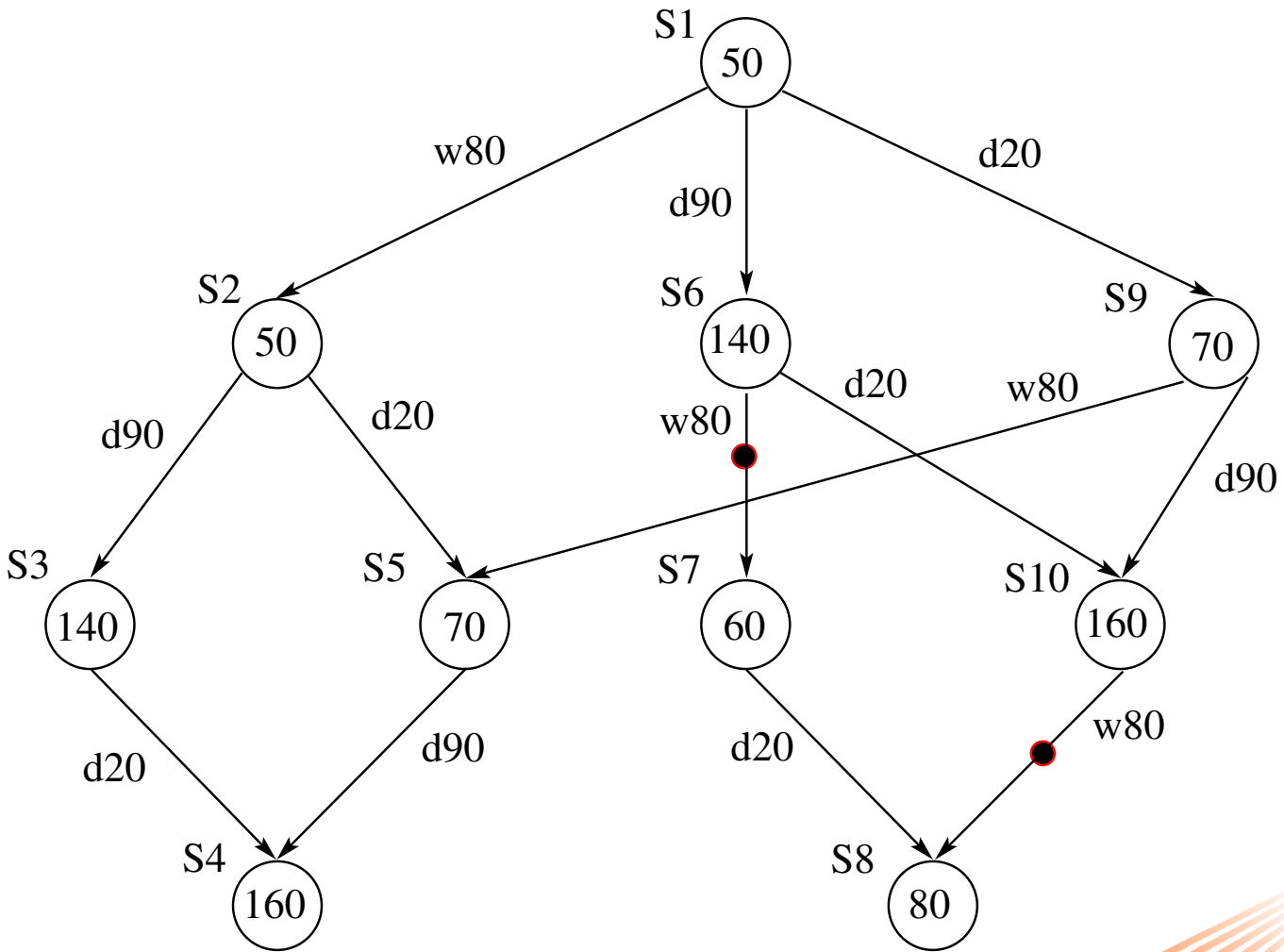
Moreover, for multithreaded code each test needs to be explored!

MuTMuT deals with this cost (upto 77% savings)

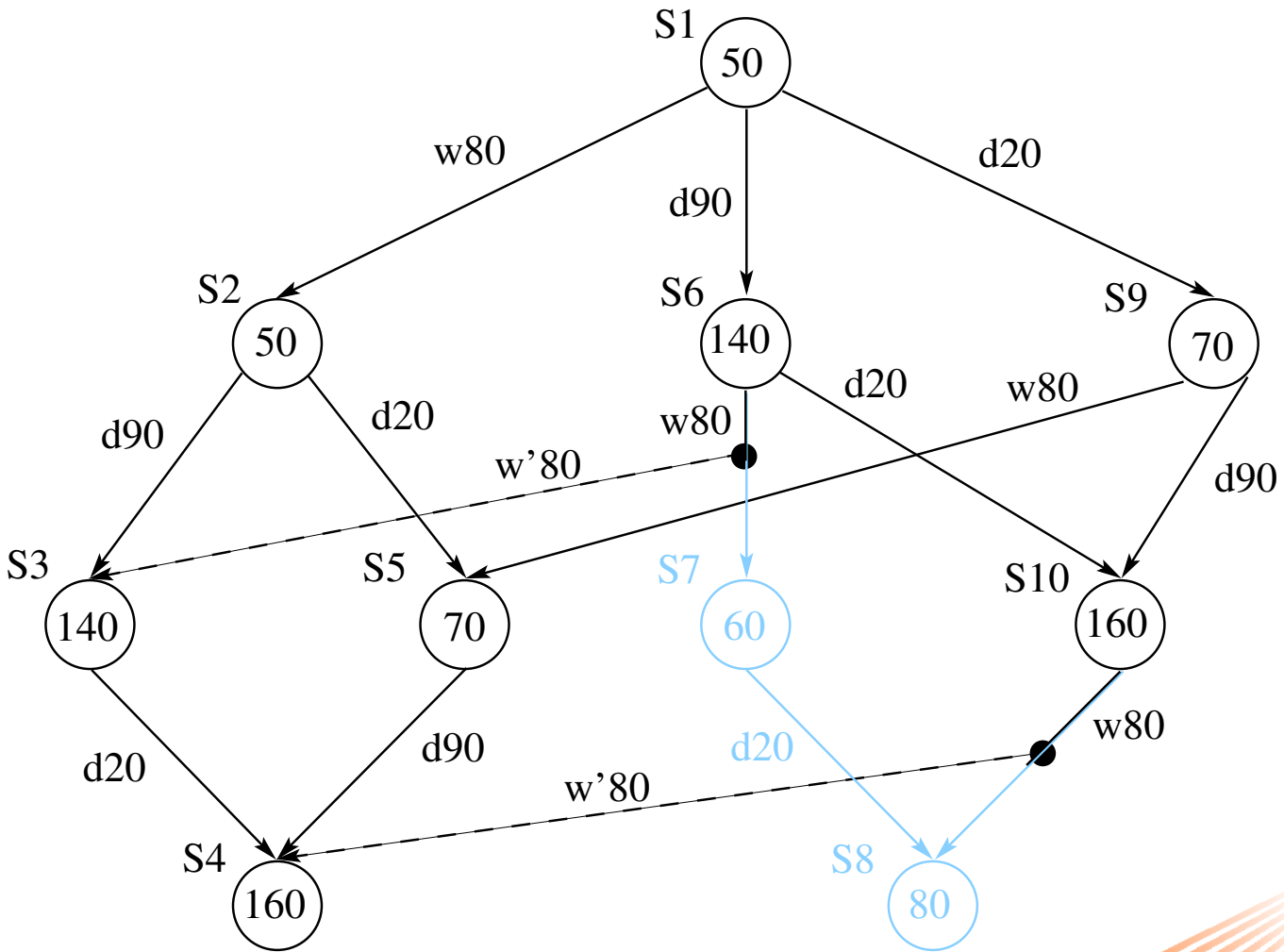
Mutated Banking Simulation

```
class Account {

    double balance;

    Account(double amount) {
        this.balance = amount;
    }

    synchronized void withdraw(double amount) {
        if (balance >= amount) {
            if (MUTATION_ID == 0) {
                balance -= amount;
            } else if (MUTATION_ID == 1) {
                balance -= 0;
            }
        }
    }

    synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

Transitions affected by mutation



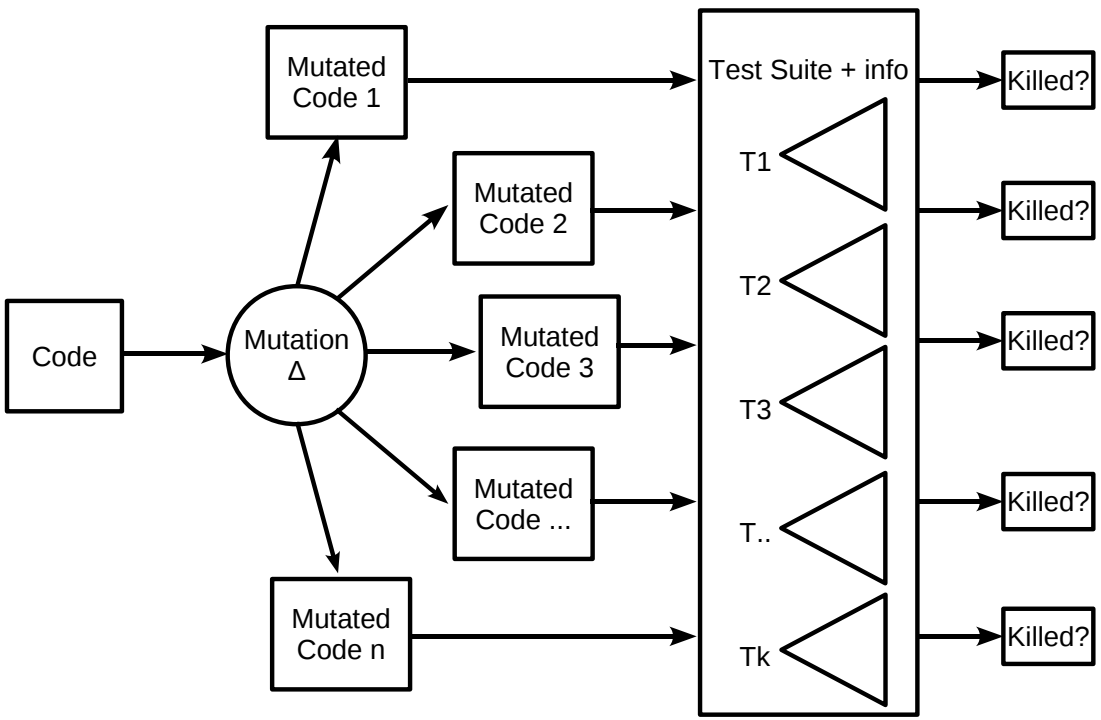
Transitions created by mutation



Outline

MuTMuT

Framework for mutation testing of multithreaded programs



UPCRC Illinois
Universal Parallel Computing Research Center

MuTMuT

Explores both original and mutated code

Reports which mutants were killed

Naive approach: re-explore each test for each mutated version

Optimizations to report same killed mutants faster!

Supports different exploration modes (complete, CHESS...)

Optimizations

Collect **info** during original run and **reuse** during mutant run

Avoid executing unnecessary tests and unnecessary exploration

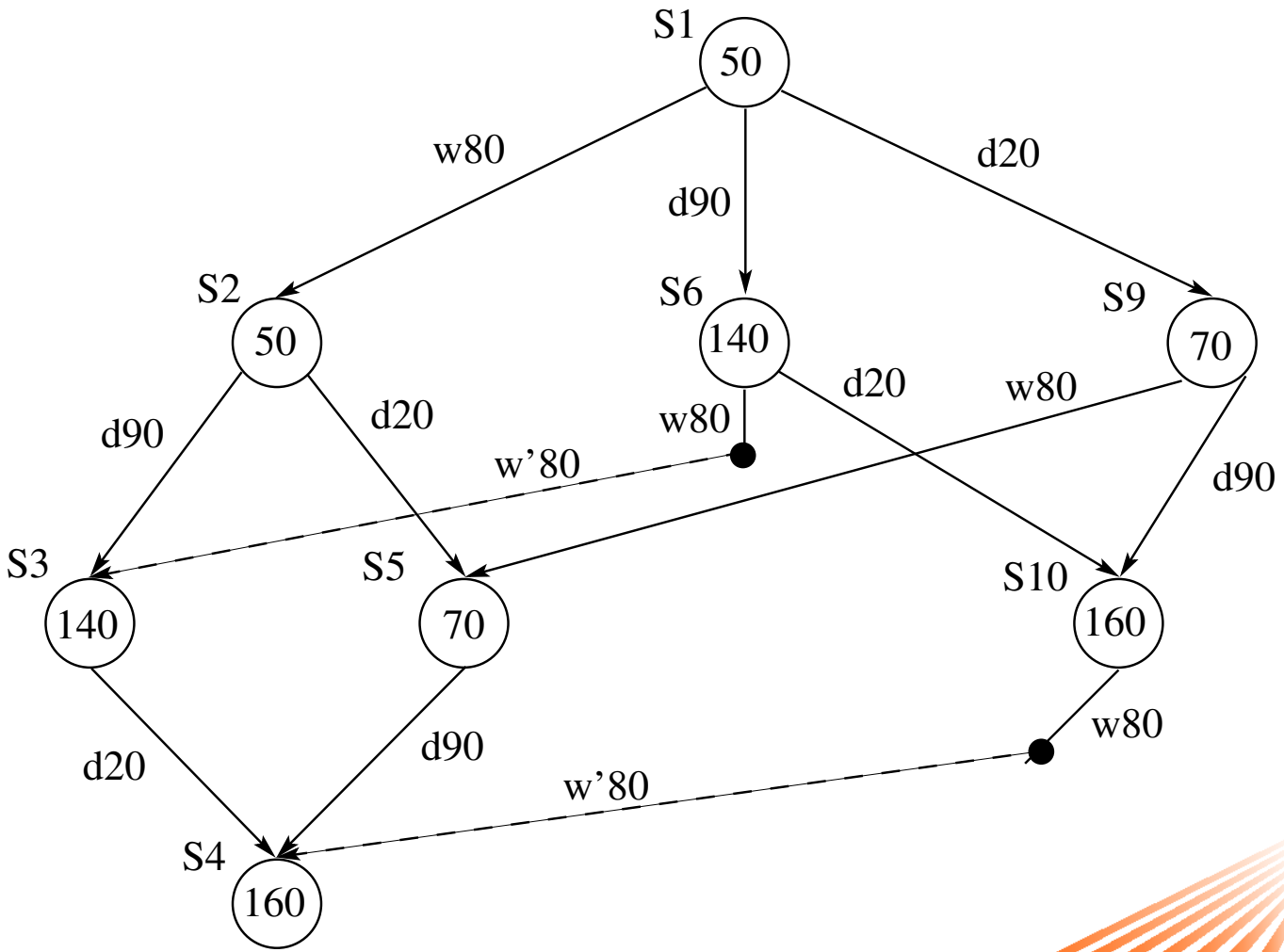
Tradeoff collection time for savings in exploration

4 optimizations:

- Basic
- OneState
- StateSet
- AllStates

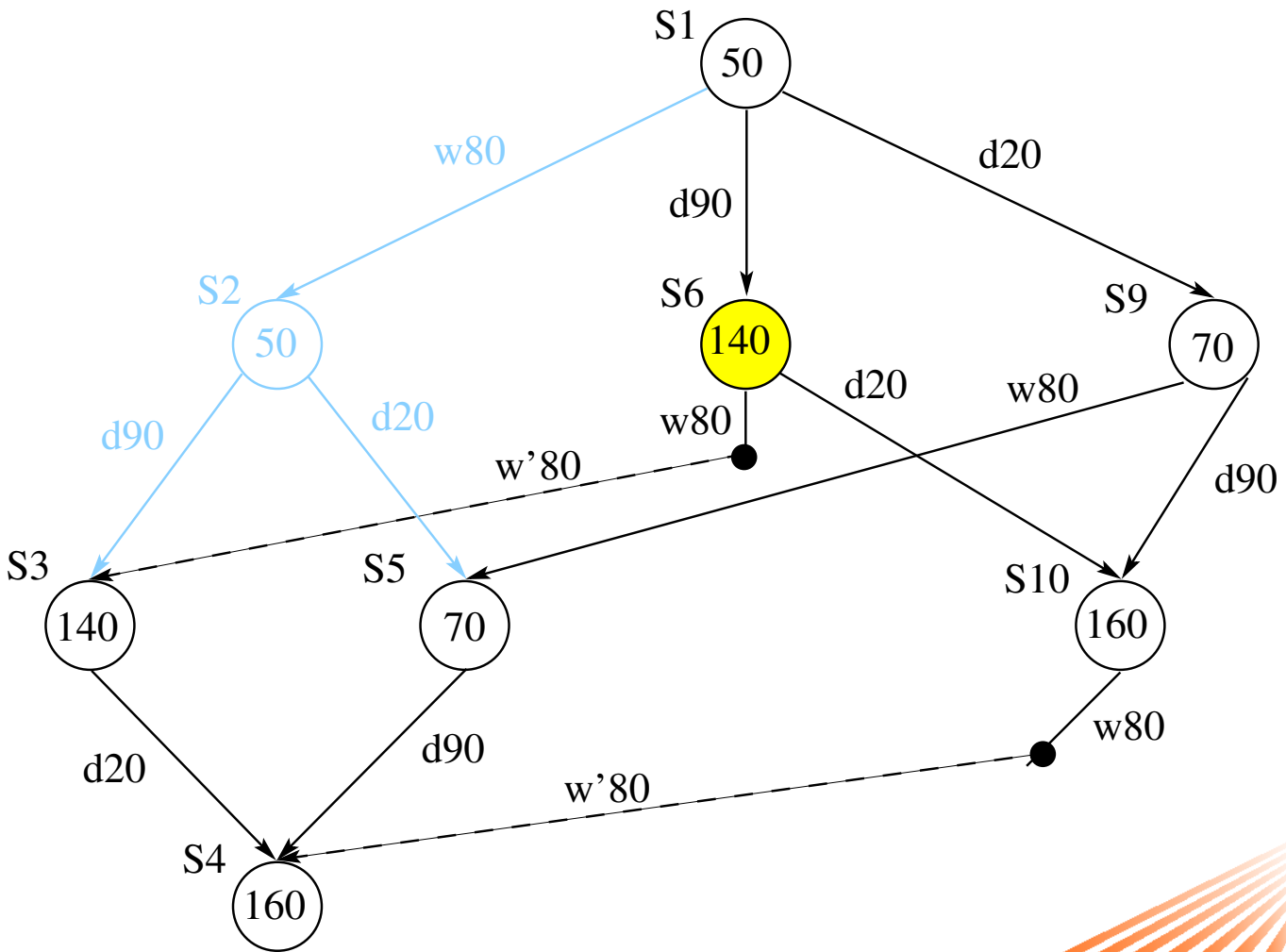
Basic Execution/Exploration

Did the test exploration reach the mutant?



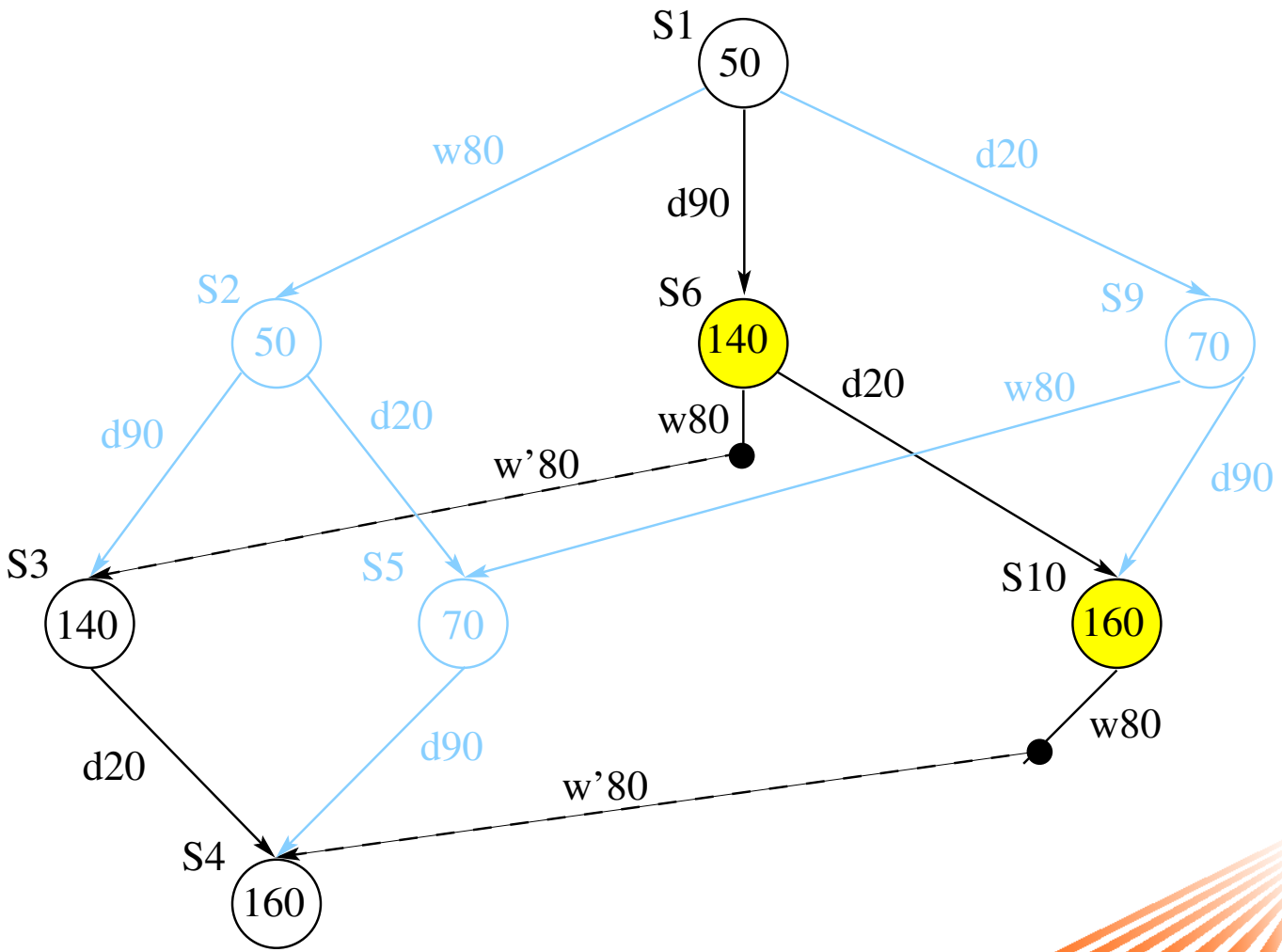
OneState Execution/Exploration

First state in the entire exploration which reaches



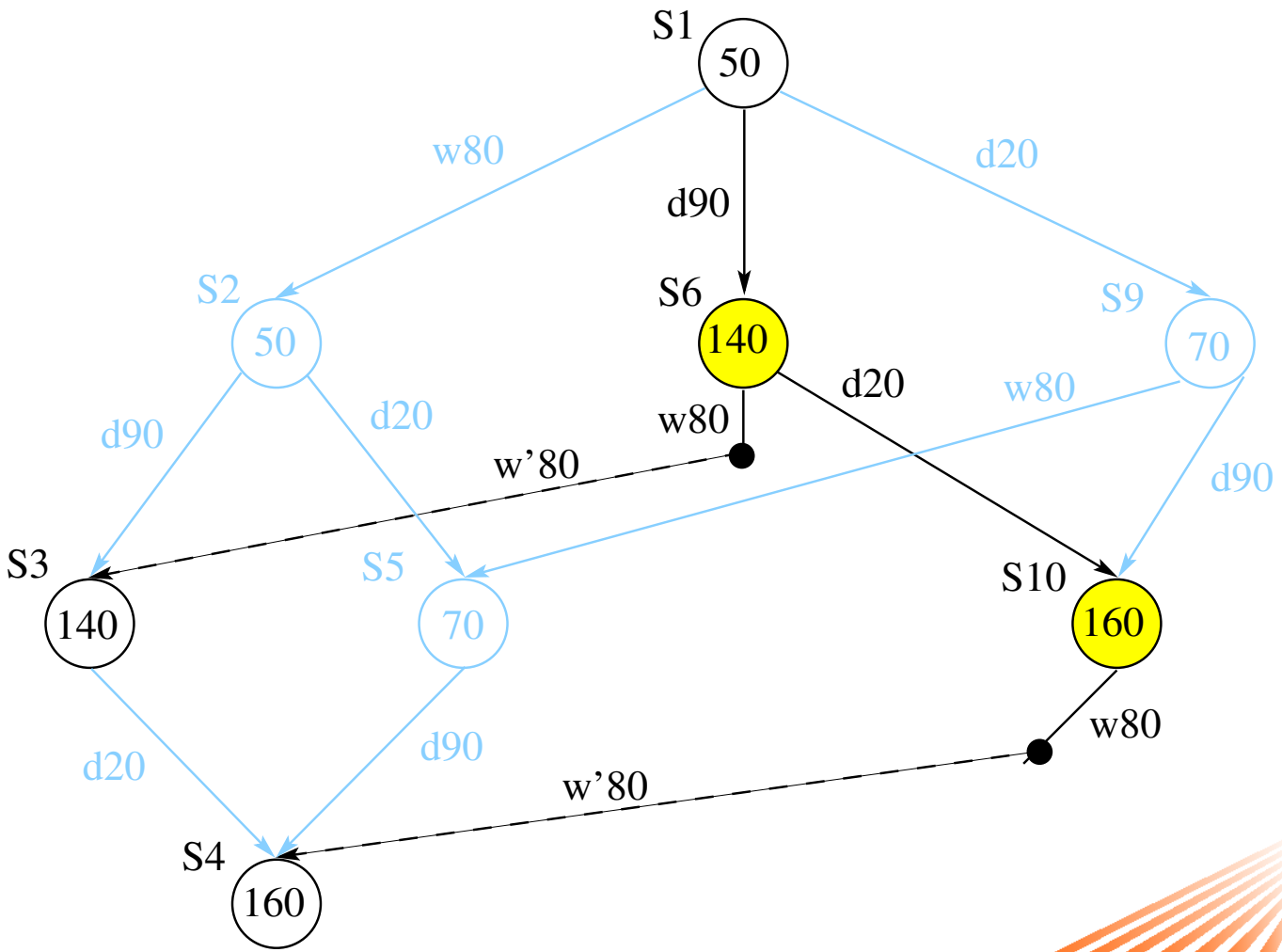
StateSet Execution/Exploration

First states in each exploration path that reaches



AllStates Execution/Exploration

All states in the exploration that reach the mutant



Outline

Experimental Subjects

program	# of tests	# of mutants	# killed	# reached
Account	13	4	4	4
Airline	9	31	18	31
Allocation	9	36	8	29
BubbleSort	6	32	28	32
Buffer	4	*15	13	15
LinkedList	16	23	14	18
NoRollback	4	66	51	66
TreeBag	9	*19	17	19

* manually created five multithreaded mutants

Exploration Time Results Basic vs StateSet

program	Exhaustive			CHESS(2)		
	Basic	StateSet	speedup	Basic	StateSet	speedup
Account	1:17:10	0:44:33	43%	02:32	01:49	28%
Airline	0:19:08	0:11:29	44%	11:52	07:46	35%
Allocation	1:13:51	0:52:52	28%	31:46	24:04	24%
BubbleSort	0:59:34	0:21:47	63%	28:00	12:32	55%
Buffer	0:57:14	0:17:22	70%	03:01	01:08	62%
LinkedList	1:16:51	0:43:57	43%	32:23	18:28	43%
NoRollback	0:25:45	0:05:41	77%	04:26	01:35	72%
TreeBag	0:14:17	0:10:33	26%	01:15	00:58	22%

Outline

Conclusion

Testing multithreaded code is expensive

Testing multiple versions of them is even more so

We are working on this problem; started with mutation testing

Able to achieve significant savings with MuTMuT optimizations

Now onto other forms of testing, especially unit testing

One challenge: collecting parallel code (with versions, tests, bugs)