Reduction of Redundancy in Directed Random Verification Through Checkpointing

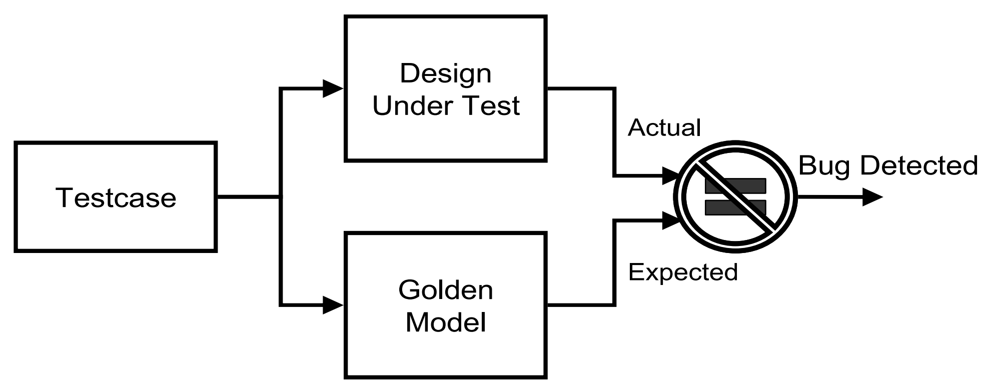
Jesse Craig
Master of Science
Specializing in Computer Science
Advisor – Craig A. Damon, Ph.D.

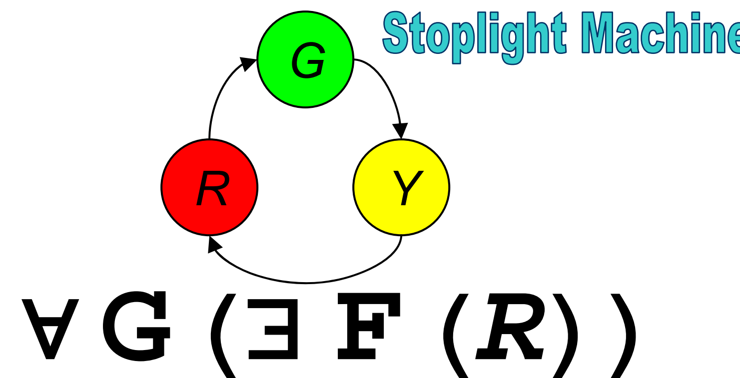
Overview

- Introduction
- Method of Redundancy Reduction
- The Reduction of Redundancy (RoR) Verification System
- Experimentation
- Conclusions and Further Work
- RoR Verification System Demonstration

Digital Logic Verification Techniques

- Functional Verification
  - Testcase applied to design
  - Golden Model creates expected result
    - *Golden Model may be wrong!*
  - Design and Golden Model disagreement indicates bug
- Formal Model Checking
  - Mathematically proves design correct
  - Tests constraints and ensures properties hold
  - Massive computational complexity

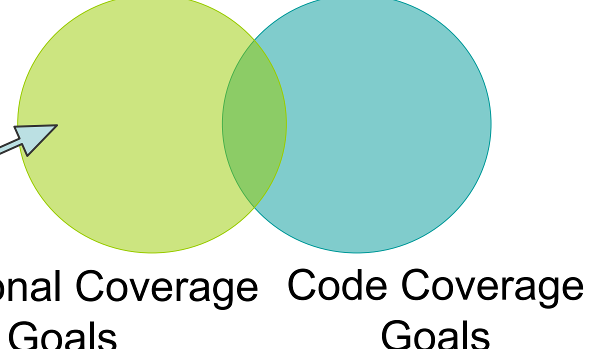




Coverage Goals

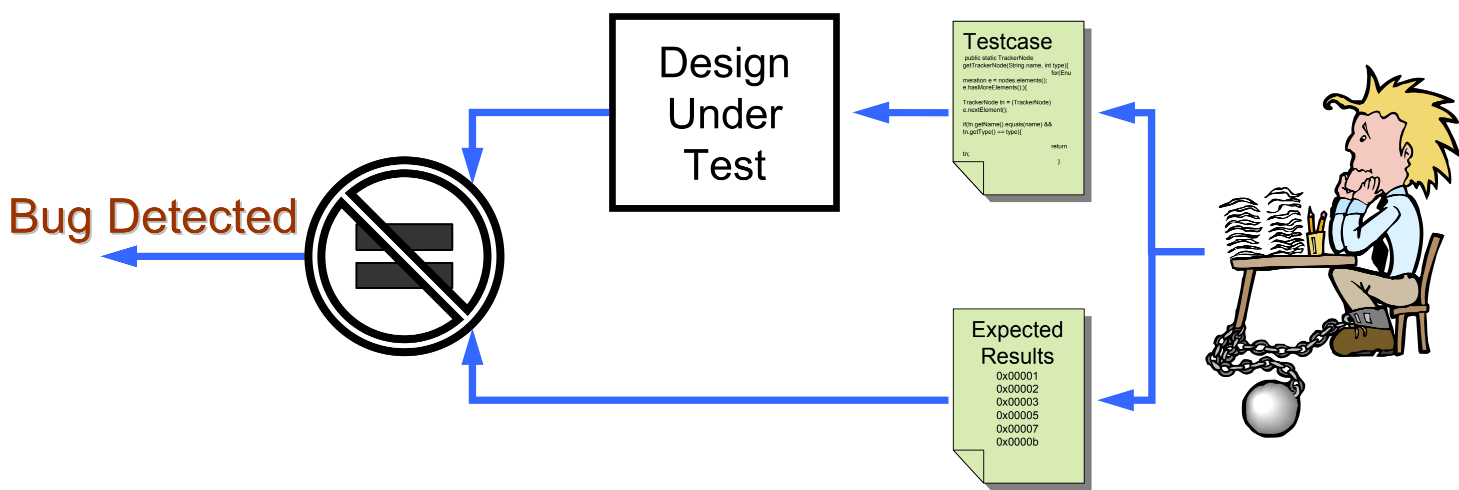
- Functional Coverage Goals
  - Targets the execution of functions offered by the design
  - Includes permutations of sequences of functions
    - Cross-coverage goals
  - Design dependent – but not implementation dependent
    - Coverage goals are the same for most implementations
- Code Coverage Goals
  - Targets the execution of statements in the design's source code
  - Design independent
    - Goal is to execute a certain percent of code statements

Focus of Experimentation



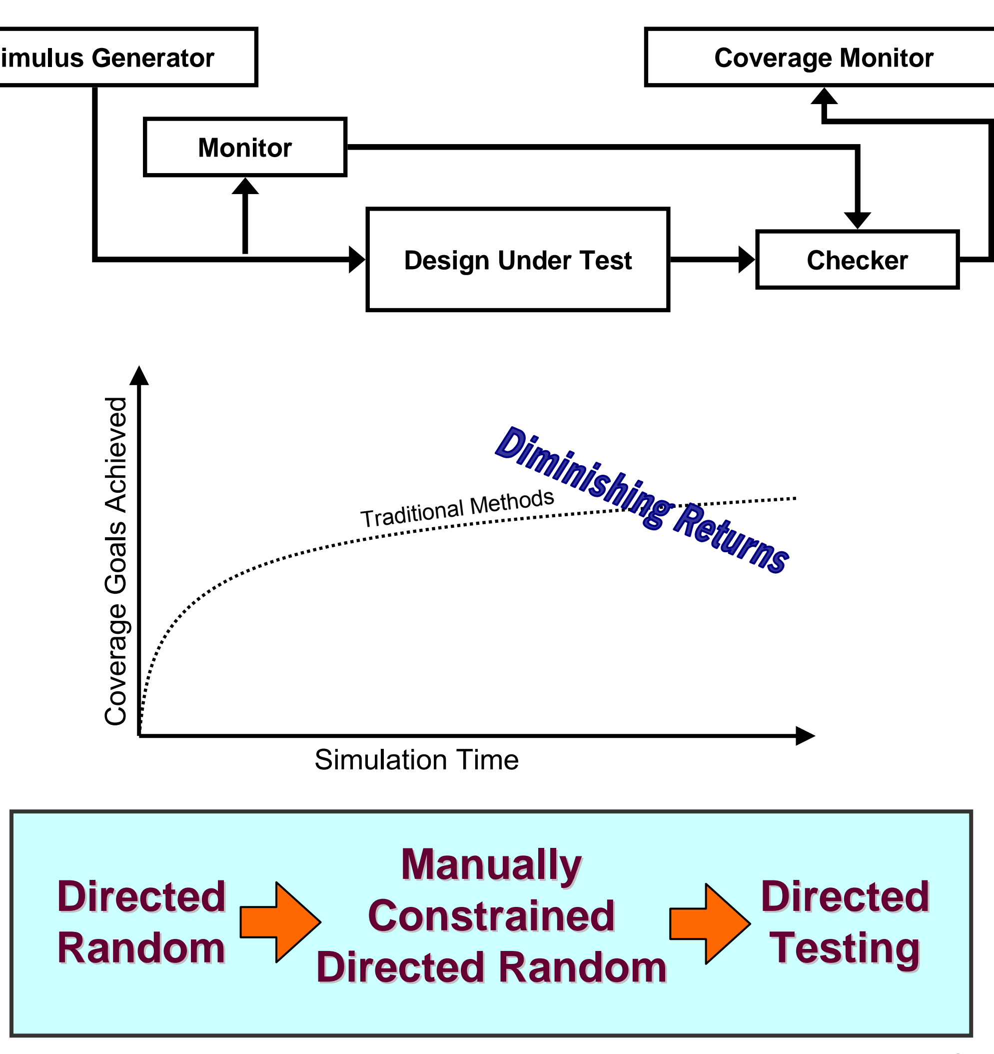
Directed Testing

- Testcase and Golden Model are generated manually
  - Time consuming
  - Costly – requires skilled verification engineers
- Can target specific, hard to achieve, coverage goals
- Good for regression testing



Directed Random Verification

- Utilizes a Stimulus Generator composed of random decisions
  - Stimulus Generator is design dependent – implementation independent
- Chronically exhibits diminishing returns
- Verification engineers forced to use manual methods – increased costs



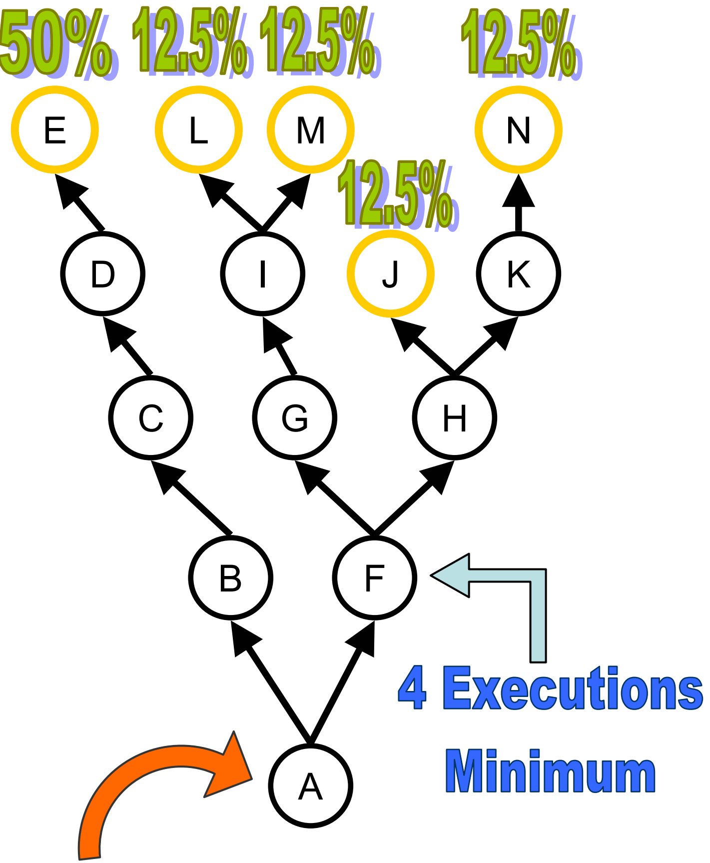
A Simple Example

```
stimulus_generator
begin
A;
if( random_boolean() == true ){
    B; C; D; E;
}else{
    F;
    if( random_boolean() == true ){
        G; I;
        if( random_boolean() == true ){
            L;
        }else{
            M;
        }
    }else{
        H;
        if( random_boolean() == true ){
            J;
        }else{
            K; N;
        }
    }
}
end
```

One testcase can achieve all the goals!

Coverage Goals are {E, L, M, J, and N}

Redundancy



- Design under test is deterministic
  - multiple executions are redundant!
- Random and unbiased system executes A→*E too many times
  - {A→*E}:{A→*N} is 4:1
- Actions A and F are redundantly executed multiple times
- Redundancy is wasted simulation time

Measuring Redundancy

Root→A→F→H→K→N

Partial Redundancy | Unique

Root→A→F→H→J

Root→A→B→C→D→E

Full Redundancy

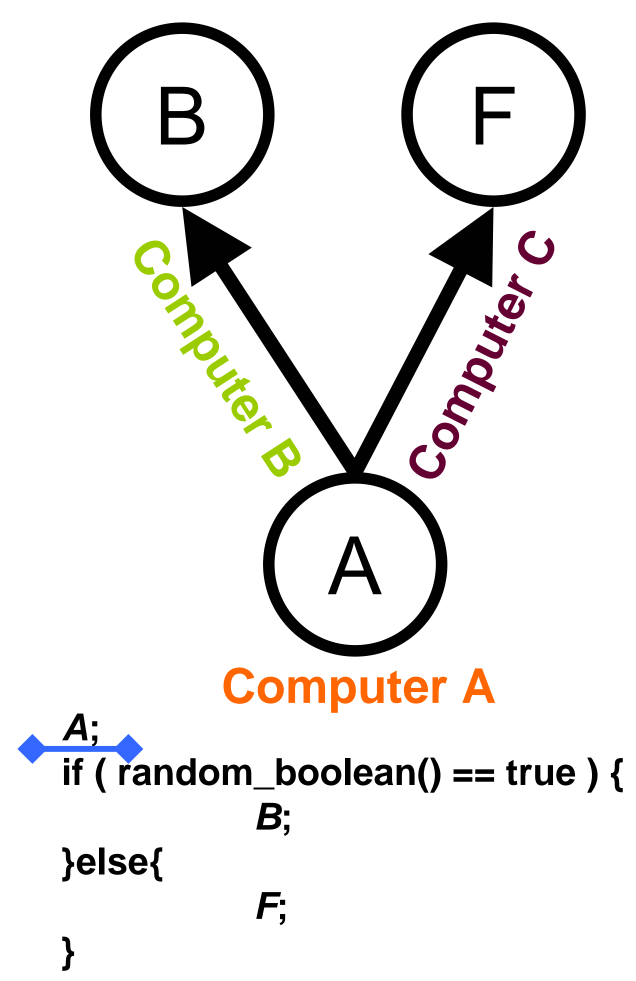
Root→A→B→C→D→E

- Full Redundancy
  - Only counts the first execution of each path as unique, all future executions are fully redundant
- Partial Redundancy
  - Only counts the first execution of each shared sub-path, all other executions are partially redundant
  - Shared sub-paths must begin at the ‘Root’ node – ensures the execution is equivalent

Method of Redundancy Reduction

- Checkpoints the entire verification environment at a decision point
  - Includes both the design simulation and the verification components
- Enumerates the possible outcomes and exhaustively explores each outcome from the checkpoint
  - Outcomes can be explored on one or more computers

`random()` ➔ `fork_random()`



Decision Points

- Directed Random Verification is based on Randomly Controlled Decision Points
  - Outcome is decided by some random function or randomly initialized variable

```
if ( random_boolean() == true ) {
        α ;
}else{
        β ;
}
```

Coverage Goals?

- Unlike standard decision points, all outcomes are valid and desired
  - Standard decision points have only one valid outcome – based on the state

```
if ( packetqueue.depth() > 0 ) {
        databus.send( packetqueue.pop() );
}else{
        databus.send( new Packet() );
}
```

Only one valid outcome!

Related Work

- Verification Optimizations
  - Anti-random Verification [Malaiya95] [Mayrhause98]
    - Only attacks the problem of full redundancy
  - Coverage Driven Verification
    - Several techniques
      - Bayesian Networks [Fine03]
      - Neural Networks [Hao05]
      - Genetic Algorithms [Faye00] [Bose01]
    - These methods can work with Redundancy Reduction
  - Semi-formal Methods [Yuan01] [Krohm96]

Related Work (cont.)

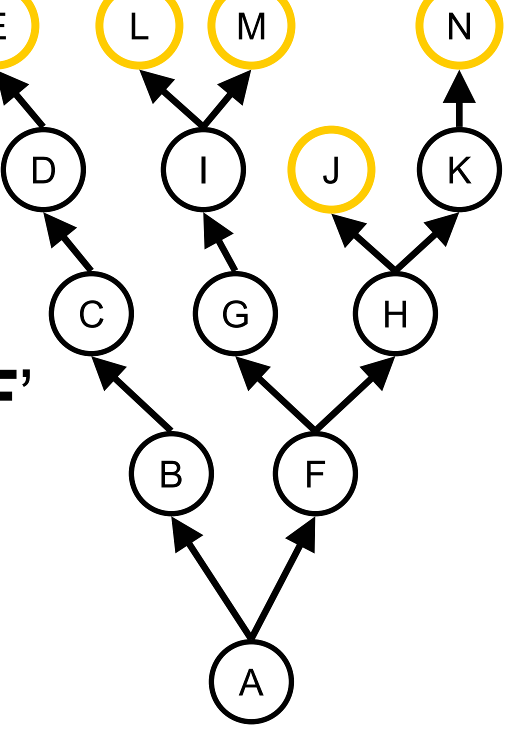
- Other uses of checkpointing
  - Hardware/Software Co-Simulation [Yang00] [Beece90]
    - Checkpoints migrate simulation from hardware to software
  - Recovery from crashes in distributed systems [Shum97]
    - Checkpoints made prior to the crash are recovered on stable resources
- Shows that checkpointing is practical

Introduction Key Points

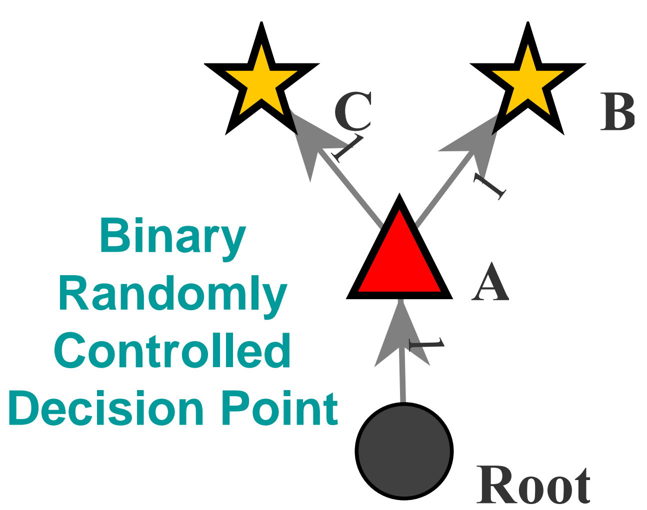
- Directed Random Verification has diminishing returns
  - The attainment of coverage goals is the measure of progress for functional verification
  - Diminishing returns force costly and time consuming directed testing techniques to be used
- Diminishing returns are caused by redundant execution
  - Redundant execution is classified into full redundancy and partial redundancy
- Redundancy Reduction checkpoints decision points and exhaustively explores the outcomes
  - Only randomly controlled decision points are candidates

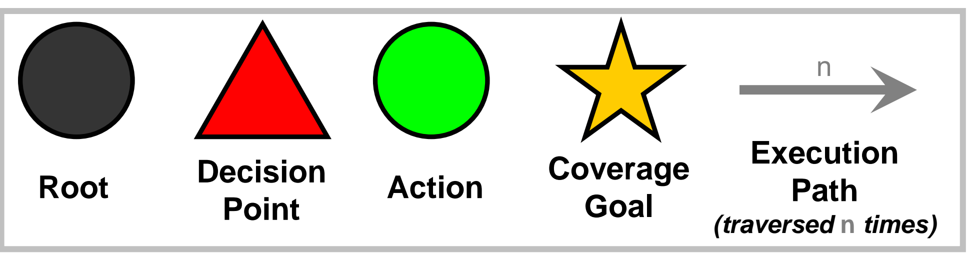
Simple Tree Experiments

- Five Versions:
  - No redundancy reduction (*Traditional*)
  - Checkpointing at ‘**F**’ only
  - Checkpointing at ‘**A**’ and ‘**F**’
  - Checkpoint once at ‘**A**’ and at ‘**F**’ every time
  - Checkpointing at every Randomly Controlled Decision Point (*Exhaustive*)



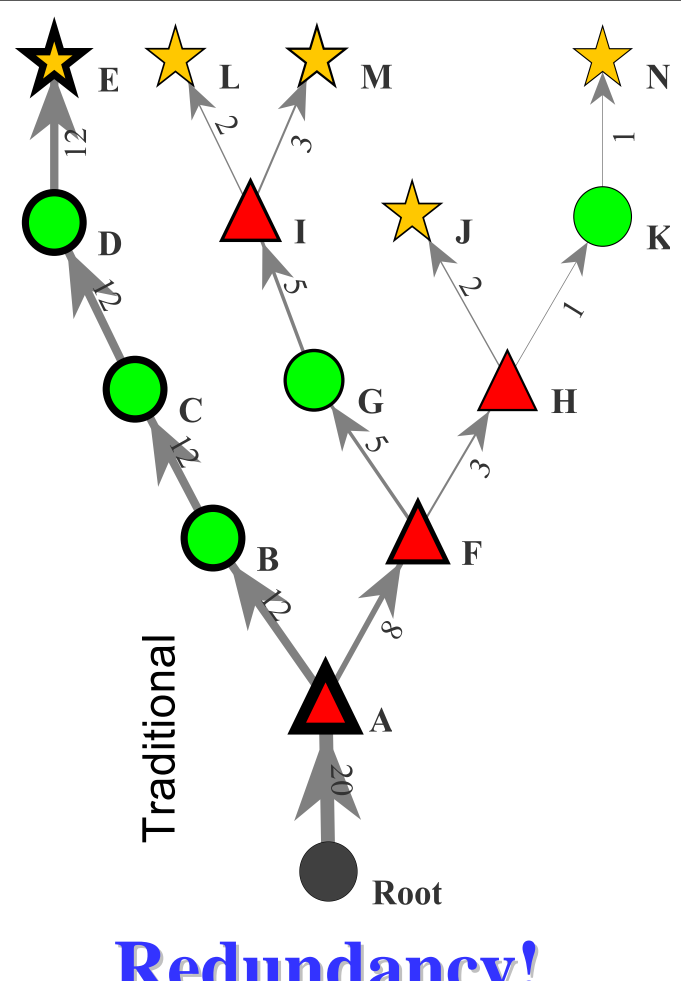
Decision Diagrams

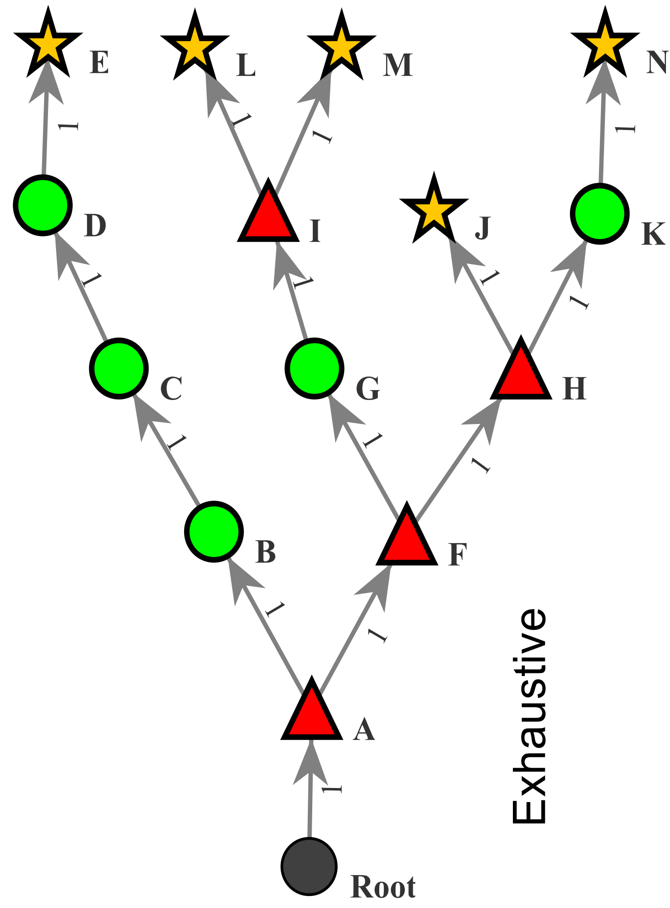




- Visually describes the possible outcomes of the stimulus generator
- A digraph showing actions, decision points, coverage goals, and a summary of the paths executed
  - Edge thickness and border thickness based on normalized execution
  - Thickness emphasizes redundancy visually

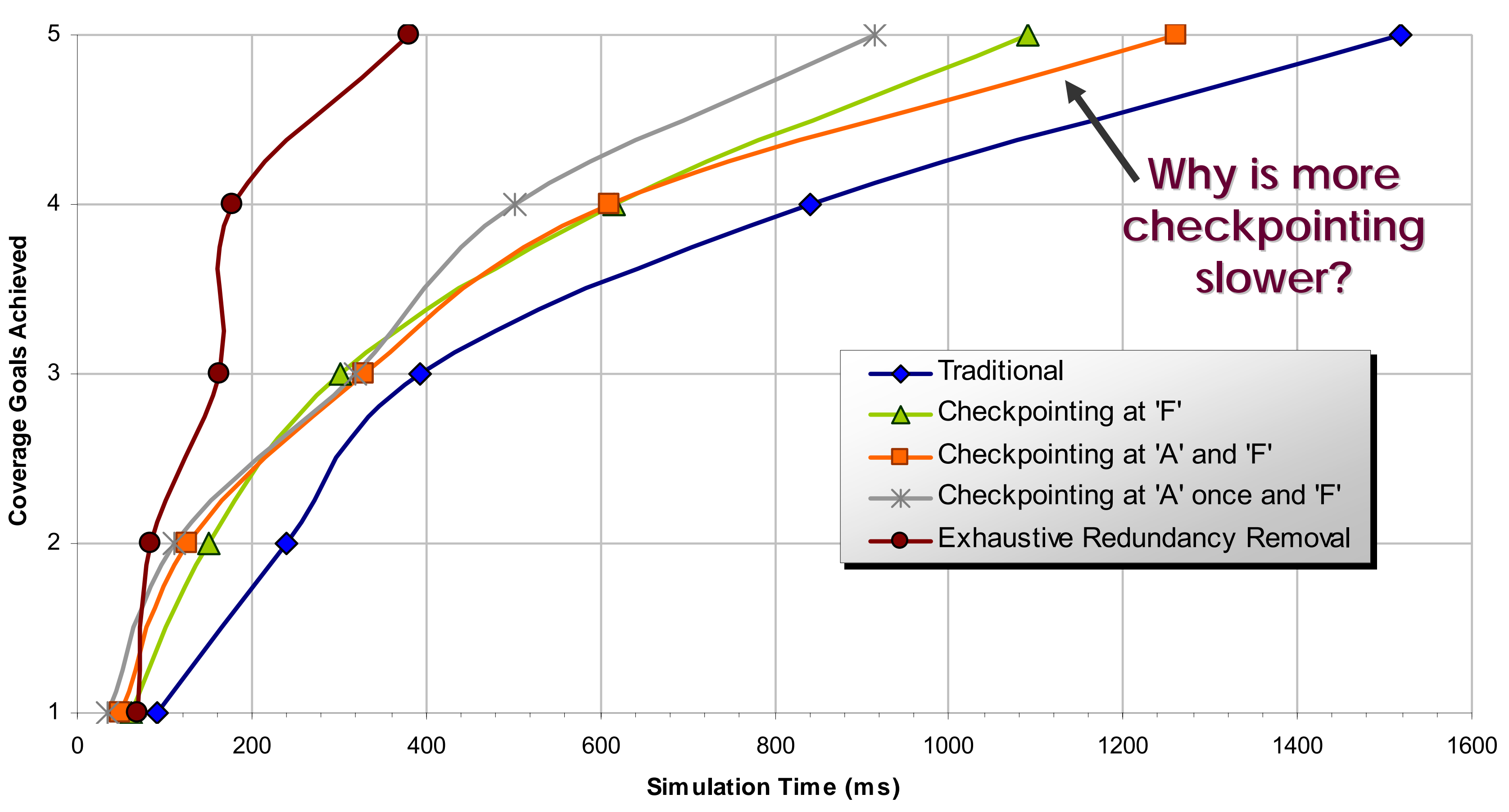
Simple Tree Experiment Decision Diagrams





No Redundancy!

Verification Performance of the Simple Tree Example



Statistical Analysis of Simple Tree Experiments

Path	*Traditional*	*Checkpointing at 'A' and 'F'*	*Checkpointing at 'A' once and 'F'*	*Checkpointing at 'F'*	*Exhaustive Redundancy Removal*
A→*E	50%	100%	100% / 50%	50%	100%
A→*L	12.5%	50%	50%	25%	100%
A→*M	12.5%	50%	50%	25%	100%
A→*J	12.5%	50%	50%	25%	100%
A→*N	12.5%	50%	50%	25%	100%

- Shows the probability of a given path being executed for each testcase starting from the initial state
- Checkpointing increases the odds of executing paths containing the checkpoint
  - Testcases using checkpointing execute multiple paths
- High, well-balanced, probabilities yield the greatest verification performance
  - Checkpointing at ‘**F**’ beats checkpointing at ‘**A**’ and ‘**F**’ yet has lower probabilities
  - Poor balancing causes full redundancy

Simple Tree Results

- In general, as redundancy is decreased, verification performance is increased
  - Cost is a measure of the simulation effort needed to achieve the coverage goals
- Checkpointing at ‘**A**’ and ‘**F**’ increases full redundancy (*versus checkpointing at ‘F’ only*)
- Checkpointing must target the coverage goals

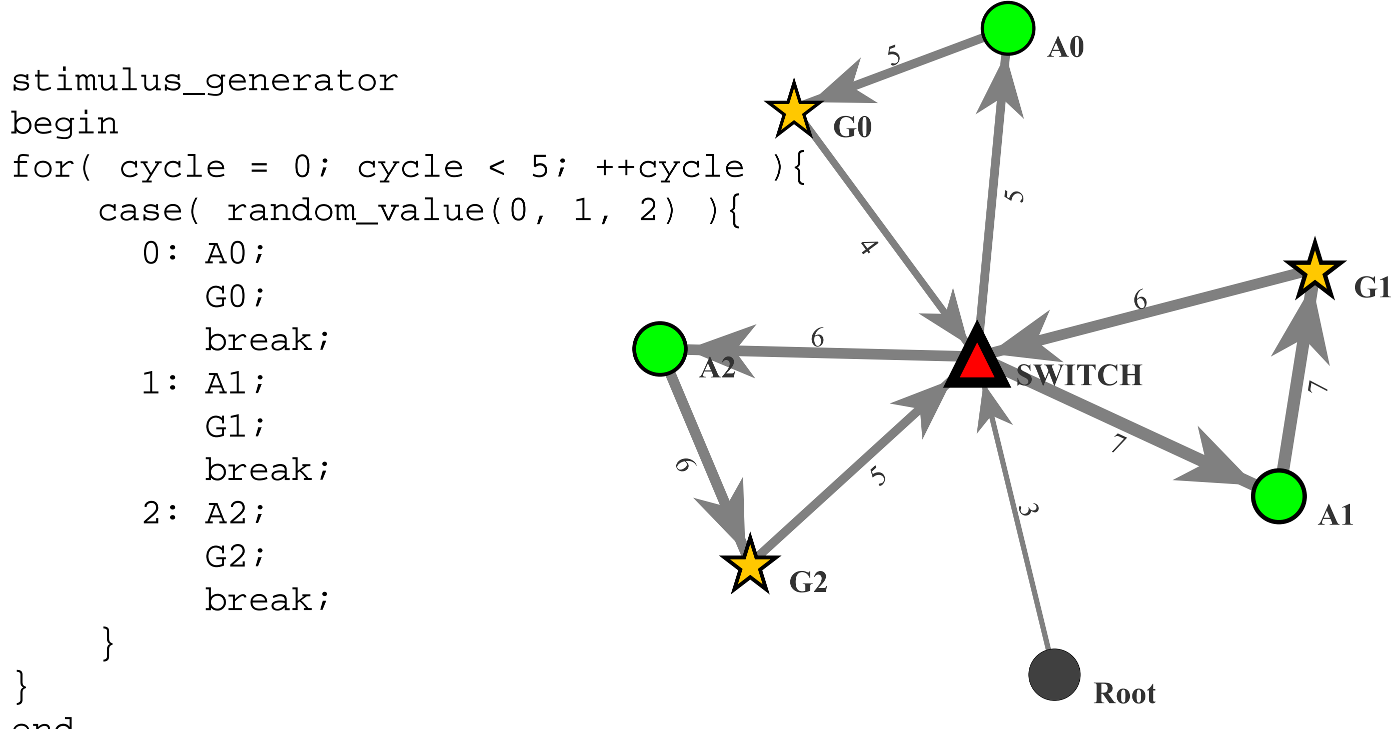
Method	*Total Testcases Executed*	*Total Execution Cost*	*Full Redundancy Cost*	*Percent of Fully Redundant Execution*	*Partial Redundancy Cost*	*Percent of Partially Redundant Execution*
Traditional	13.3	64.9	40.9	63%	10.0	15%
Checkpoint at 'F'	11.3	46.7	26.1	56%	6.6	14%
Checkpoint at 'A' and 'F'	14.1	49.9	29.8	60%	5.1	10%
Checkpointing once at 'A' and 'F'	10.5	37.3	18.2	49%	5.1	14%
Exhaustive Checkpointing	5.0	14.0	0.0	0%	0.0	0%

Simple Tree Example Key Points

- Empirical data shows significant redundancy is traditional directed random verification
- The redundancy reduction method is able to reduce, and even completely remove, this redundancy
  - The exhaustive version has no redundancy
  - This reduction increase verification performance
- Deciding which decision points to checkpoint is not a trivial decision
  - The probabilities of executing the possible paths must be both high and well-balanced

Simple Loop Example

- Common structure for stimulus generators – central control loop
- Two versions: standard coverage goals, and cross-coverage goals



Simple Loop Results

Coverage Goals	*Method*	*Simulation Time to Complete Coverage (ms)*	*Percent of Fully Redundant Execution*	*Percent of Partially Redundant Execution*
Standard Coverage	Traditional	350	0%	1%
	Redundancy Reduction	266	0%	0%
Cross-Coverage	Traditional	11,636	8%	42%
	Redundancy Reduction	5,695	0%	0%

No Redundancy

- Cross-coverage goals by definition have partial redundancy
- The method is excellent at achieving cross-coverage goals
- Without redundancy, the method's exhaustive exploration is still useful
  - Similar to anti-random methods

Queue Sort Methods

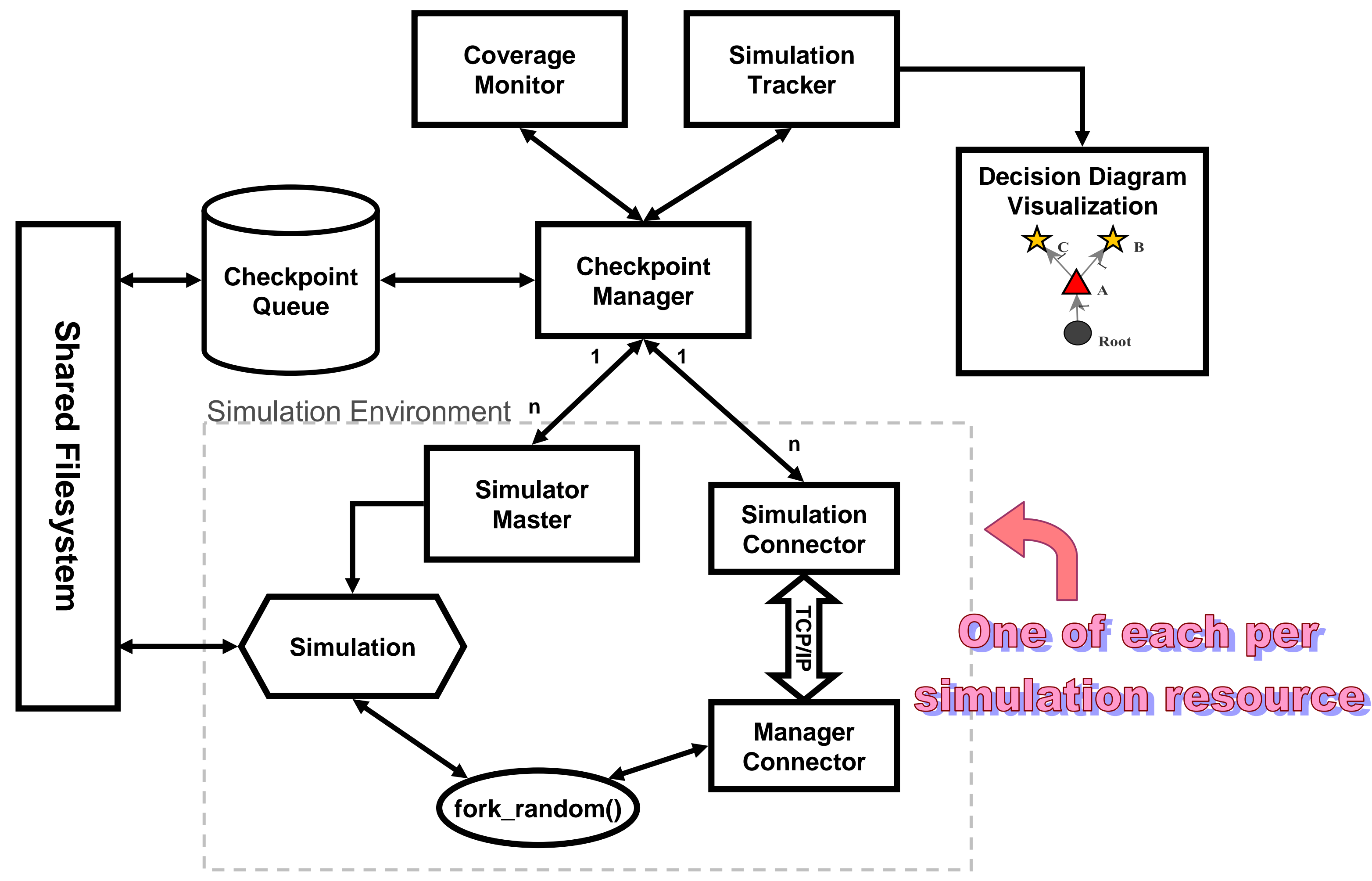
- Three sort methods explored: (*sort both checkpoints and outcomes*)
  - **FIFO** – First-in-first-out
  - **LIFO** – Last-in-last-out
  - **Random** – Randomly explore checkpoints and outcomes
- Verification environments may have a susceptibility or proclivity for certain ordering methods
- Sort methods effect the number of open (*pending*) checkpoints
  - LIFO will never have more open checkpoints than the depth of the decision diagram in decision points
  - Random haphazardly leaves checkpoints with one or two outcomes unexplored – leading to the largest amount of open checkpoints

Verification Environment	*Checkpoint Queue Sort Method*	*Average Number of Open Checkpoints*	*Maximum Number of Open Checkpoints*	*Total Number of Checkpoints*	*Simulation Time for 100% Coverage (ms)*
Simple Tree	FIFO	1.69	3	4	400
	LIFO	1.24	2	4	421
	Random Order	1.44	2.6	4	406
Simple Loops	FIFO	23.06	27	39	2555
	LIFO	3.36	5	15	3014
	Random Order	30.45	42.6	62.2	6457

Simple Loop and Queue Sorting Key Points

- Redundancy reduction is excellent at achieving cross-coverage goals
  - The application of checkpointing should target cross-coverage goals
- Without redundancy, the method's exhaustive exploration is still useful
- The choice of queue sorting methods is design dependant
  - No one method is best for all applications
  - LIFO yeilds the fewest open checkpoints
  - Random yeilds the most open checkpoints

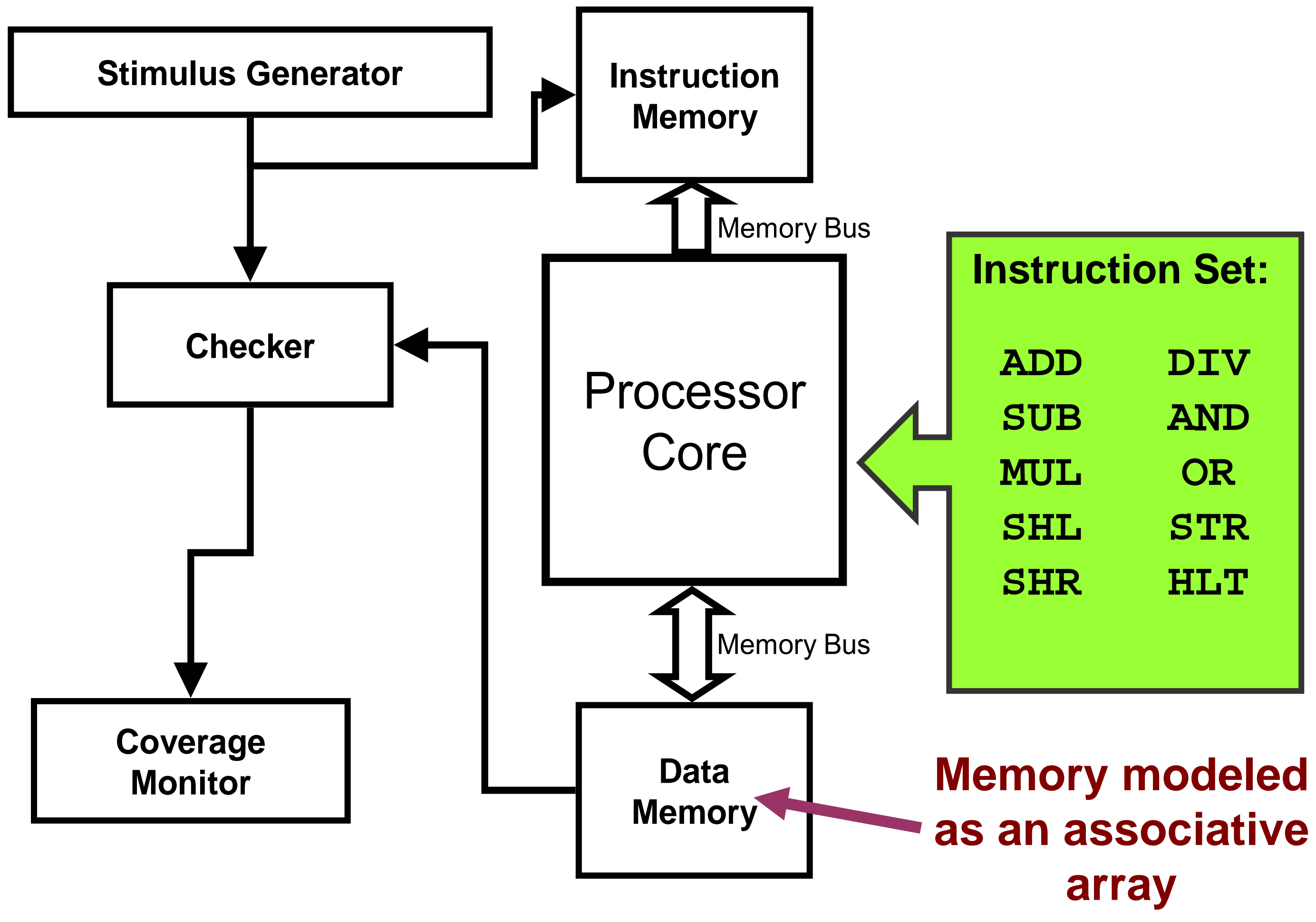
RoR System



Experimentation

- Processor Model Experiment
  - Verifies a harvard architecture processor
  - Similar to verification done on modern embedded processors
- Bus Protocol Model Experiment
  - Bus Bridge design bridging a high-speed serial bus to a processor local bus
  - Common design in System-on-a-chip applications
  - Similar to a USB Host Controller or PCI Express design

Processor Model Verification Environment



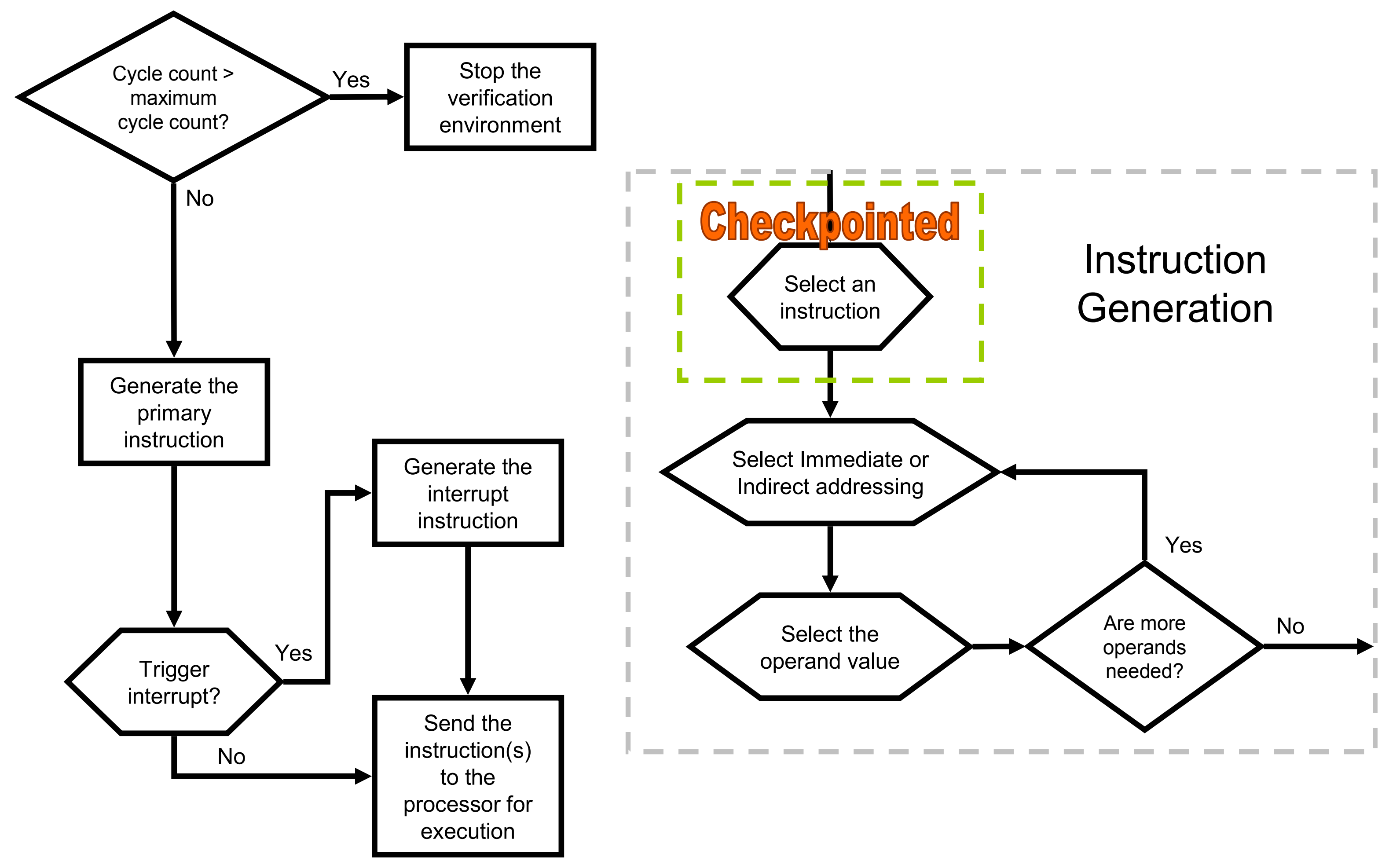
Processor Model Coverage Goals

- Coverage goals include testing instructions, sequences of instructions, and testing exceptions
- Majority of goals are cross-coverage goals

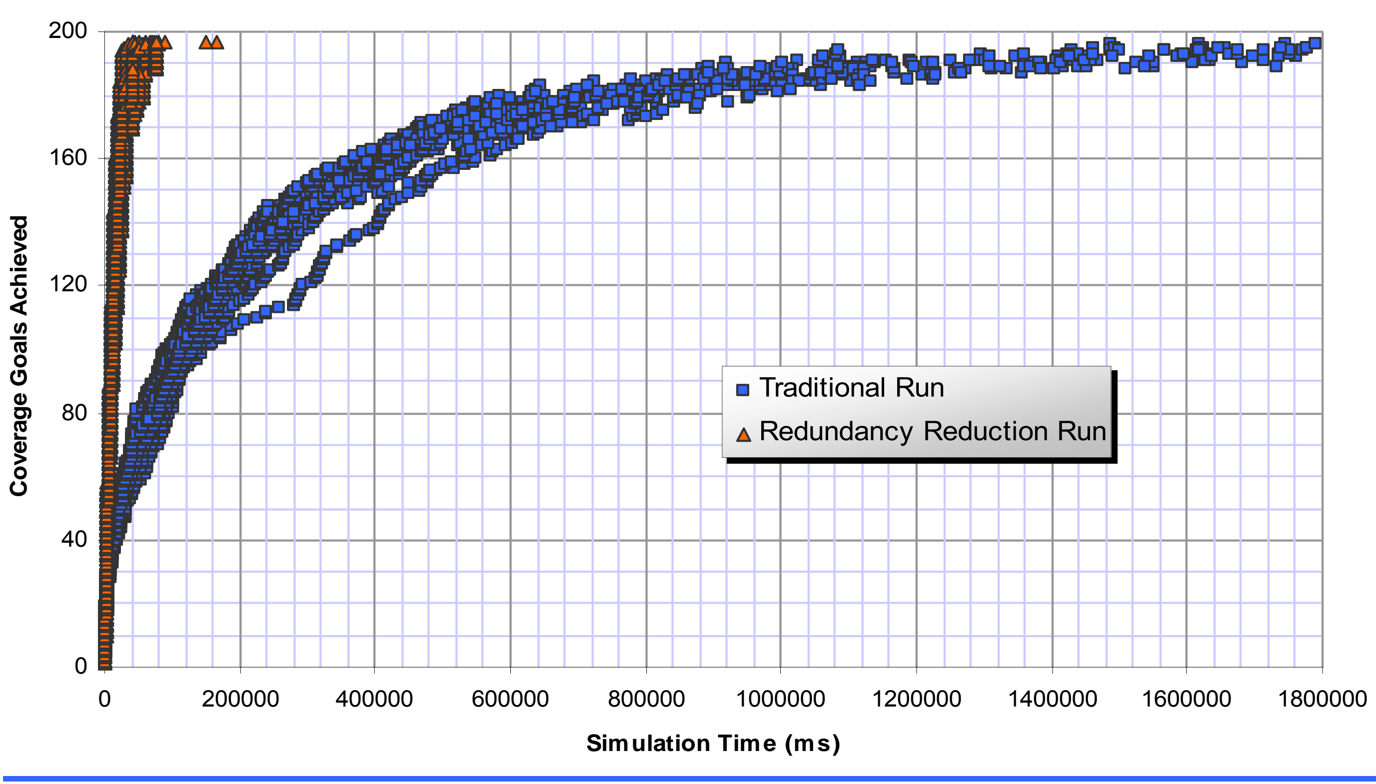
Coverage Goal	Number of Goals	*Goal Type*
Processor Halt	1	Standard
Interrupt	1	Standard
Indirect Addressing	1	Standard
Data Memory Read	1	Standard
Data Memory Write	1	Standard
Instruction Memory Read	1	Standard
Null Pointer Exception	1	Standard
Divide by Zero Exception	1	Standard
Instruction with Indirect Addressing	9	Standard
Pairs of Instructions	90	Cross-coverage
Pairs of Instructions with Interrupt	90	Cross-coverage

Target these goals!

Processor Model Stimulus Generator

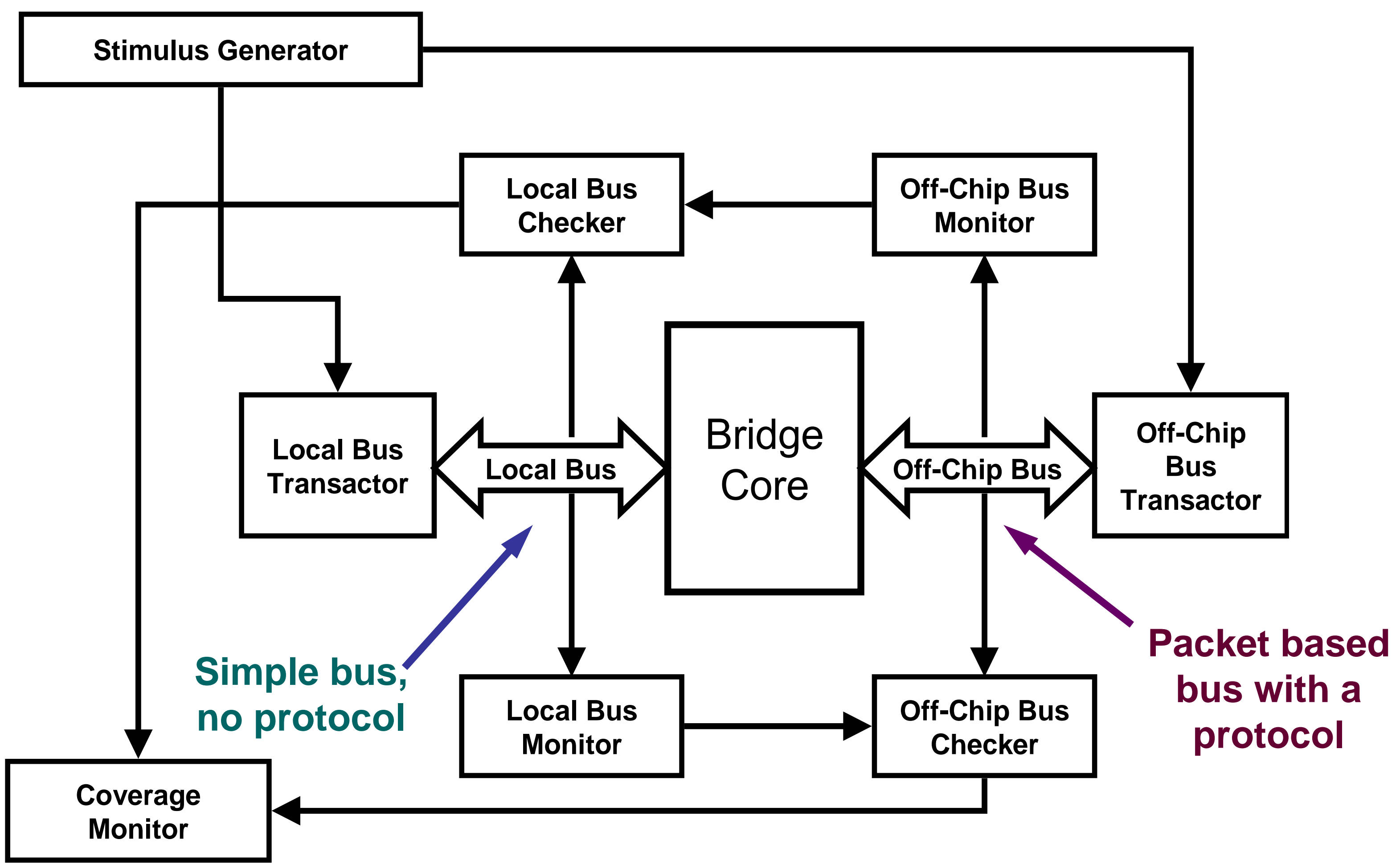


Processor Model Results

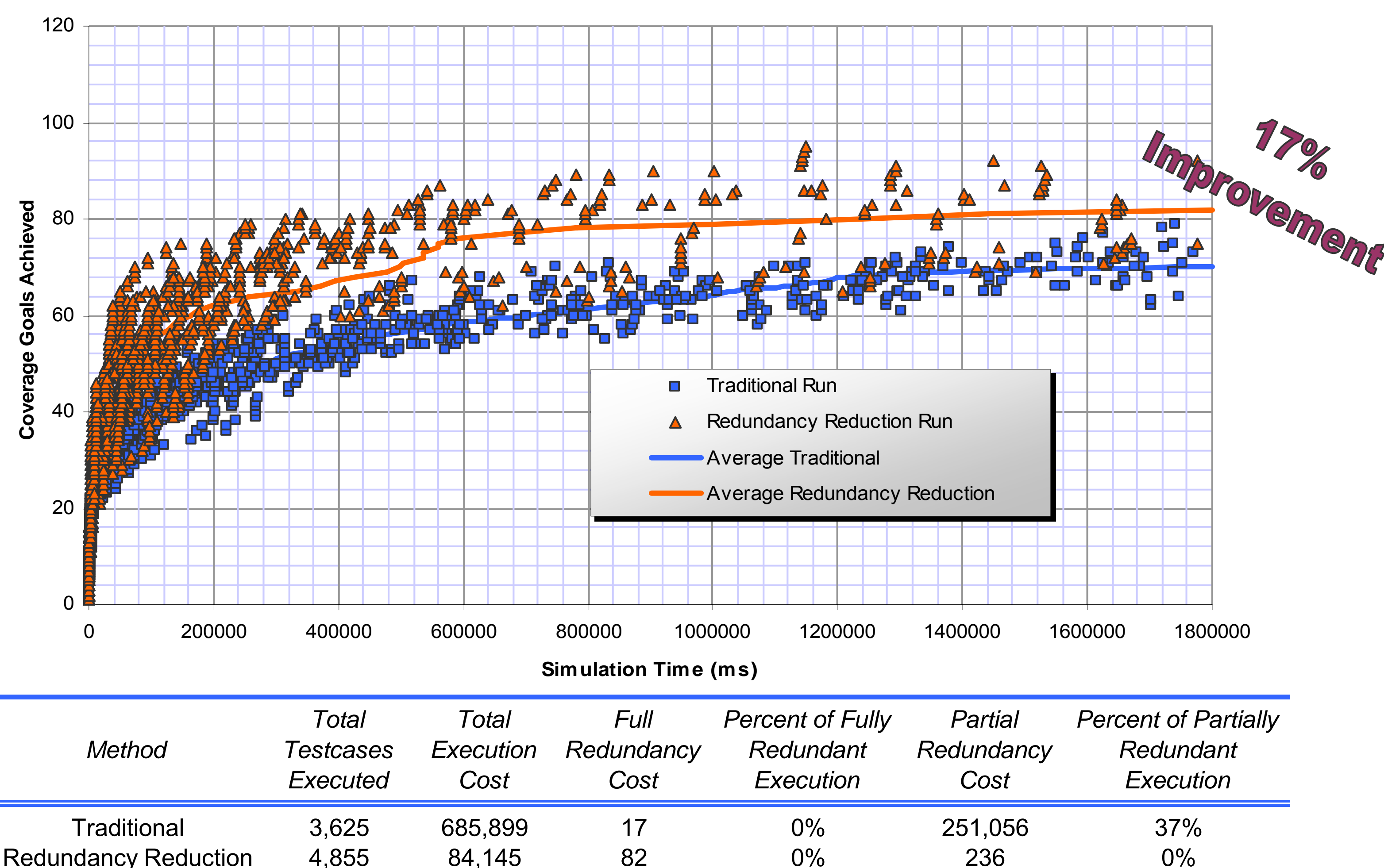


Method	*Total Testcases Executed*	*Total Execution Cost*	*Full Redundancy Cost*	*Percent of Fully Redundant Execution*	*Partial Redundancy Cost*	*Percent of Partially Redundant Execution*
Traditional	7,341	245,187	106,087	43%	71,076	29%
Redundancy Reduction	6,878	104,210	4,494	4%	5,842	6%

Bus Protocol Model Verification Environment



Bus Protocol Model Results



Method	Total Testcases Executed	Total Execution Cost	Full Redundancy Cost	Percent of Fully Redundant Execution	Partial Redundancy Cost	Percent of Partially Redundant Execution
Traditional	3,625	685,899	17	0%	251,056	37%
Redundancy Reduction	4,855	84,145	82	0%	236	0%

Summary of Results

Verification Environment	Checkpoint Queue Sort Method	Maximum Number of Open Checkpoints	Total Number of Checkpoints	Percent of Fully Redundant Execution	Percent of Partially Redundant Execution	Coverage Goals Achieved	Simulation Time to Complete Coverage (ms)
Simple Tree Example Exhaustive Version	Traditional	–	–	63%	15%	5	1,519
	FIFO	3	4	–	–	5	400
	LIFO	2	4	–	–	5	421
	Random Order	3	4	0%	0%	5	406
Simple Loop Example Cross-Coverage Version	Traditional	–	–	8%	42%	30	11,636
	FIFO	27	39	–	–	30	2555
	LIFO	5	15	–	–	30	3014
	Random Order	43	62	0%	0%	30	6457
Bus Protocol Model	Traditional	–	–	0%	37%	70	–
	FIFO	13	5,567	–	–	83	–
	LIFO	3	5,726	0%	0%	81	–
	Random Order	10	5,624	–	–	82	–
Processor Model	Traditional	–	–	43%	29%	194	–
	FIFO	4	5,048	–	–	195	–
	LIFO	3	980	4%	6%	197	72,745
	Random Order	6	2,363	–	–	197	286,966

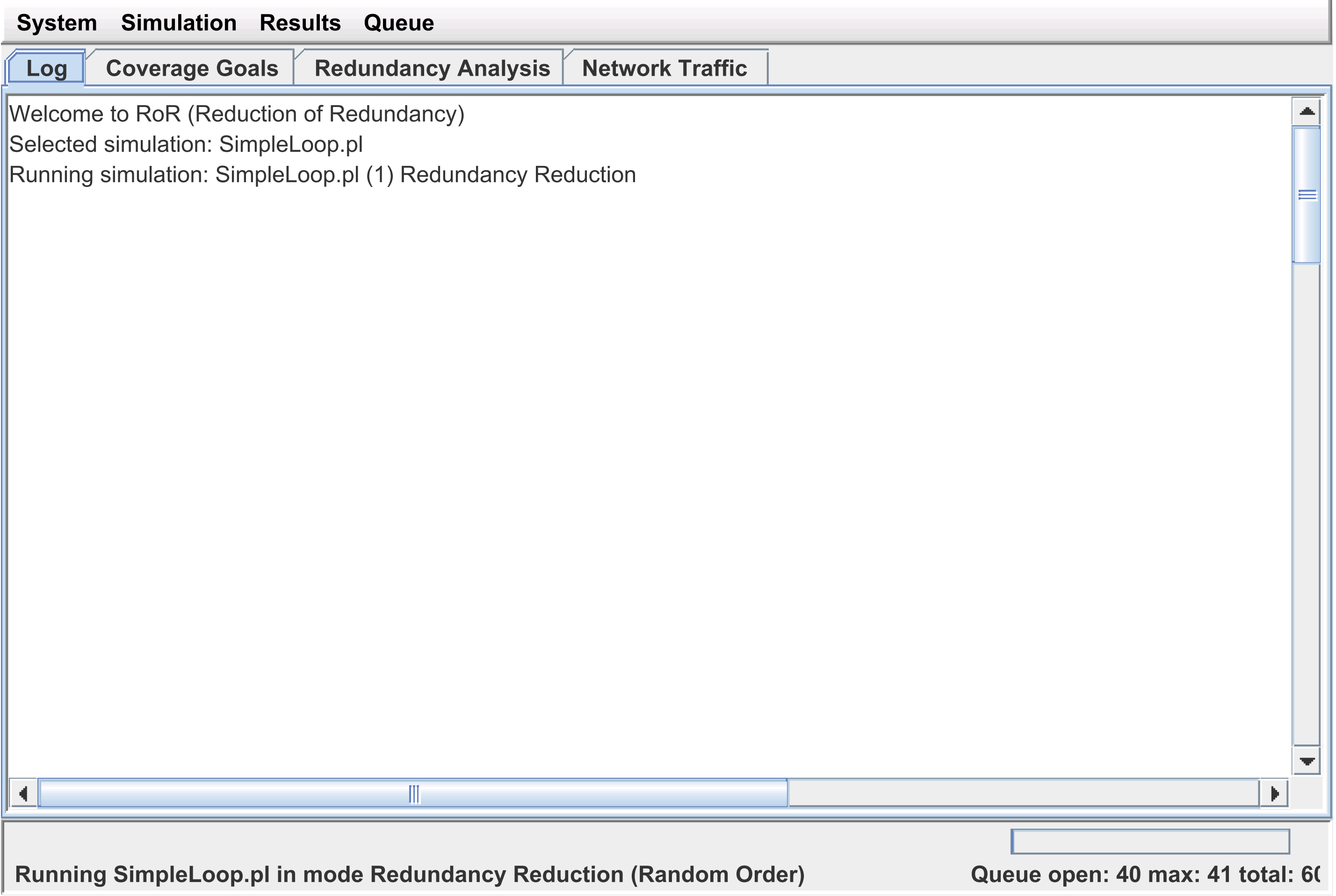
Conclusions

- Redundancy reduction does increase verification performance
  - Easy to integrate into current verification environments
- Redundancy reduction must target the coverage goals
  - Choosing where to checkpoint requires in-depth knowledge of the verification environment
- The performance of the sort methods is application dependent
  - Fewest open checkpoints and fewest total checkpoints are key indicators

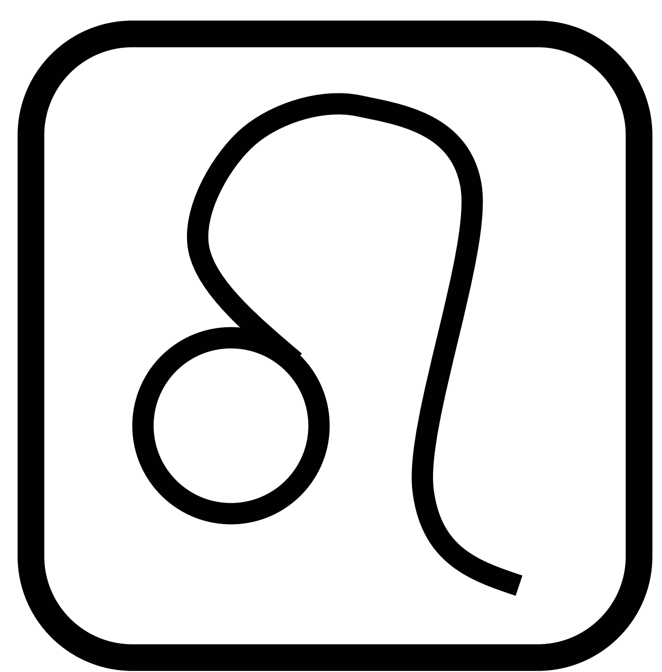
Further Work

- Combing redundancy reduction with coverage driven techniques and anti-random
  - Apply the technique to the queue sorting problem
- Add checkpointing as a feature in modern hardware verification languages (HVLs)
- Discover techniques to better apply redundancy reduction
  - Get the processor model performance gains with the bus protocol model

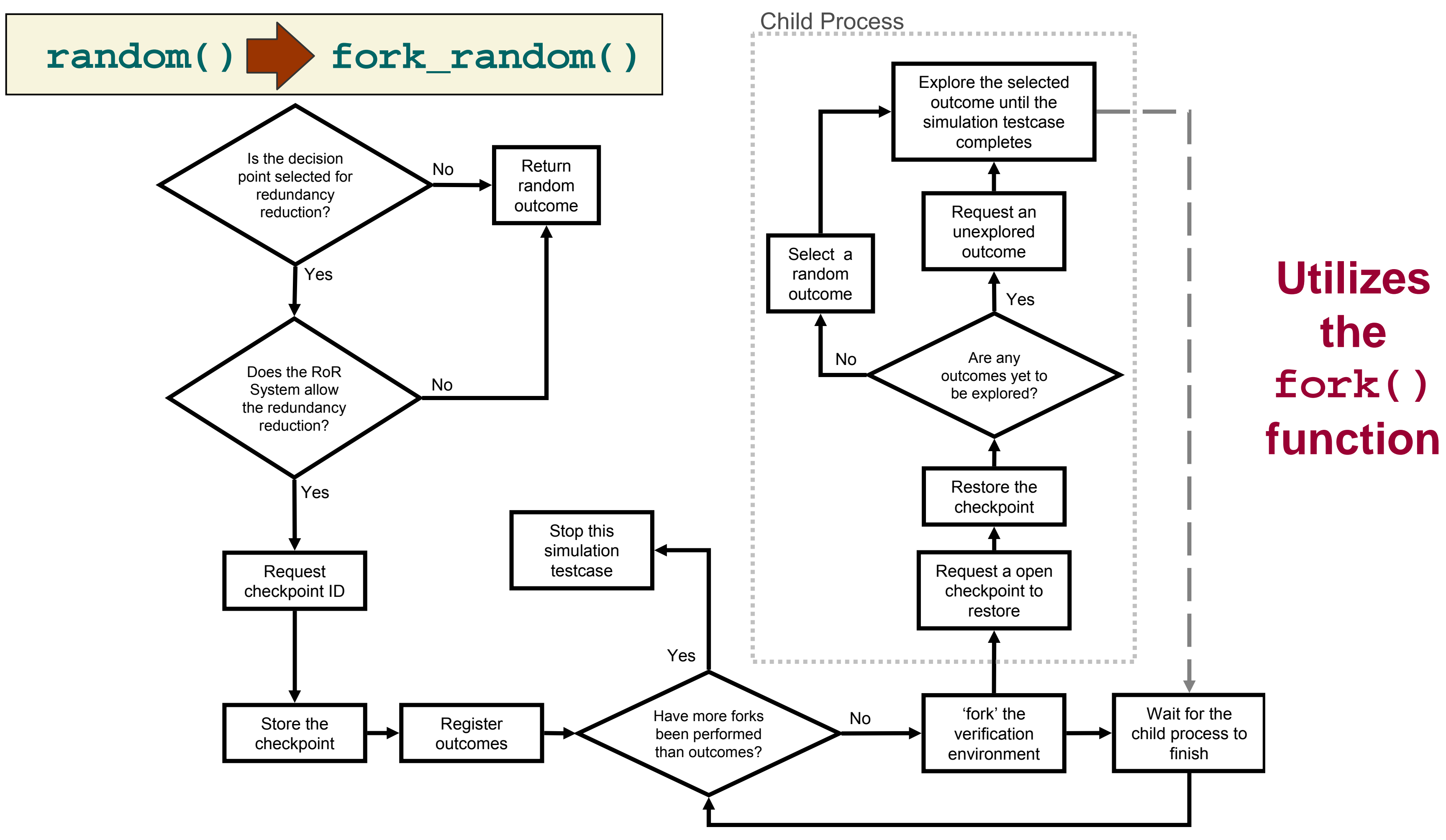
RoR System Demo



Backup Slides



fork_random()



Applying Redundancy Reduction to the Processor Model

Method	*Coverage Goals Achieved*	*Simulation Time to Complete Coverage (ms)*	*Average Number of Open Checkpoints*	*Maximum Number of Open Checkpoints*	*Total Number of Checkpoints*
Trigger an Interrupt, Select a Primary Instruction	197	72,745	3	4	980
Trigger an Interrupt, Select a Primary Instruction, Select the Operand Value	4	–	8	8	71
Trigger an Interrupt	197	1,116,756	1	2	5,556
Trigger an Interrupt, Select a Primary Instruction, Select an Interrupt Instruction, Select Immediate or Indirect Addressing of Instructions	39	–	12	16	6,855

- Checkpointing that does not target the coverage goals does poorly – over-verifies
- Checkpointing at just ‘Trigger an Interrupt’ performs better than traditional methods, but worse than the standard redundancy reduction version

Queue Sort Methods with the Processor Model

Checkpoint Queue Sort Method	*Average Number of Open Checkpoints*	*Maximum Number of Open Checkpoints*	*Total Number of Checkpoints*	*Coverage Goals Achieved*	*Simulation Time to Complete Coverage (ms)*
Traditional	–	–	–	194	–
FIFO	4	15	5,048	195	–
LIFO	3	4	980	197	72,745
Random Order	6	15	2,363	197	286,966

- All three sort orders perform better than the traditional version
- The verification environment is sensitized to the FIFO sort order
- LIFO has the best performance
  - Least simulation time consumed for 100% coverage
  - Fewest pending checkpoints
  - Fewest total checkpoints

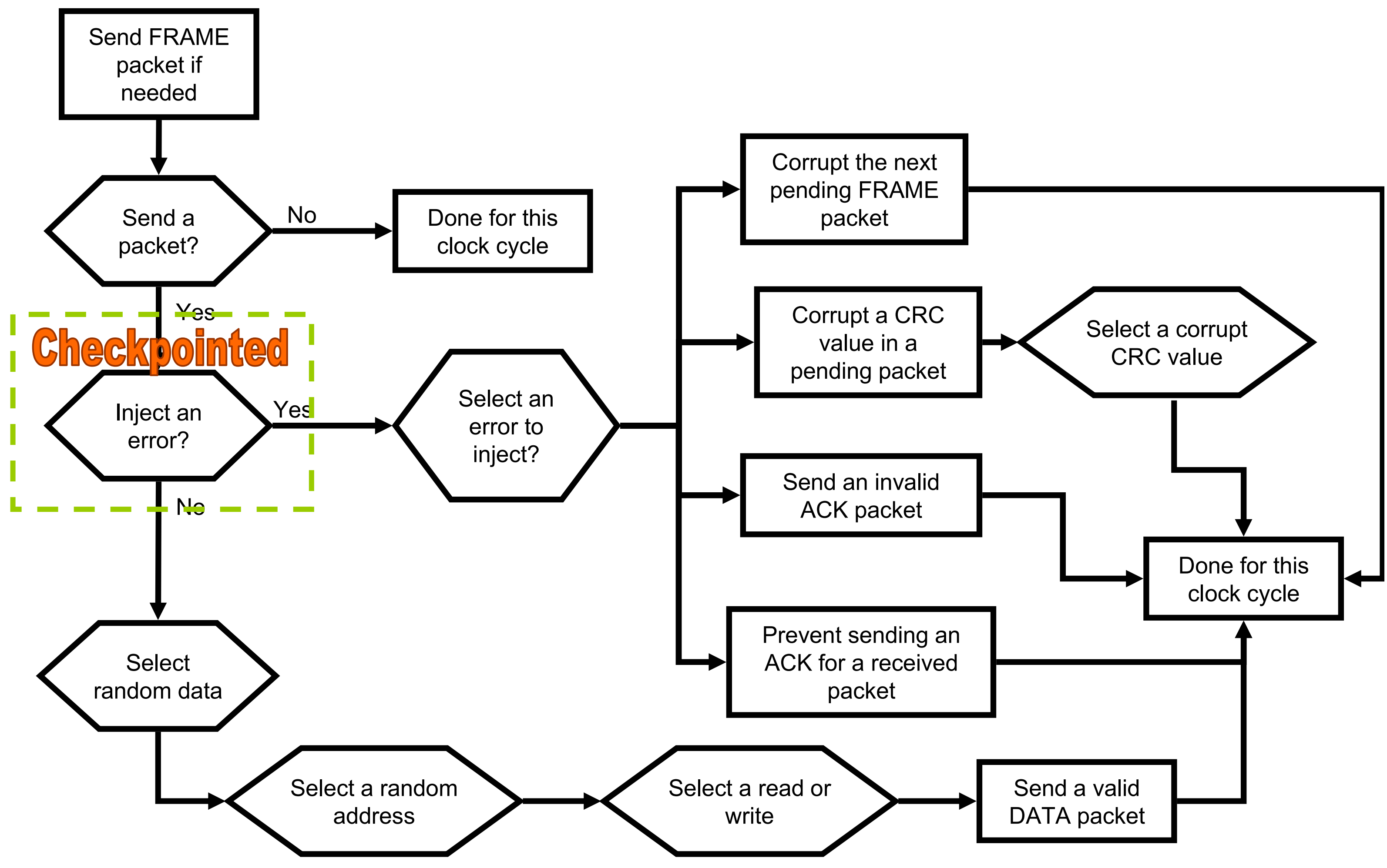
Bus Protocol Model Coverage Goals

Coverage Goal	*Bus Used*	*Purpose*
Reset	—	A reset of the Bridge Core
Data Sent	Local Bus	Sending a transaction on the local bus
Data Received	Local Bus	Receiving a transaction from the local bus
Bad CRC	Off-chip Bus	Receiving a packet with a CRC miscompare
Double Bad CRC	Off-chip Bus	Receiving two packets in a row with a CRC miscompare
Good then Bad CRC	Off-chip Bus	Receiving a good packet followed by one with a CRC miscompare
Bad ACK then Bad CRC	Off-chip Bus	Receiving a bad ACK packet followed by a CRC miscompare
Three Good then Bad CRC	Off-chip Bus	Receiving three good packets followed by one with a CRC miscompare
Five Good then Bad CRC	Off-chip Bus	Receiving five good packets followed by one with a CRC miscompare
One Good Packet Received	Off-chip Bus	Receiving a good packet
Double Good Packets Received	Off-chip Bus	Receiving two good packets in a row
Three Good Packets Received	Off-chip Bus	Receiving three good packets with no bad packets in-between
Five Good Packets Received	Off-chip Bus	Receiving five good packets with no bad packets in-between
Three Bad then Good Packet Received	Off-chip Bus	Receiving three bad packets followed by a good packet
Five Bad then Good Packet Received	Off-chip Bus	Receiving five bad packets followed by a good packet
Good ACK	Off-chip Bus	Receiving a valid ACK packet
Double Bad ACK	Off-chip Bus	Receiving two bad ACK packets in a row
Good then Bad ACK	Off-chip Bus	Receiving a good packet followed by a bad ACK
Bad CRC then Bad ACK	Off-chip Bus	Receiving a packet with a CRC miscompare followed by a bad ACK
Three Good Bad ACK	Off-chip Bus	Receiving three good packets followed by a bad ACK packet
Five Good Bad ACK	Off-chip Bus	Receiving five good packets followed by a bad ACK packet
Bad ACK	Off-chip Bus	Receiving a bad ACK packet
Frame Received	Off-chip Bus	Receiving a FRAME packet in the correct amount of time
Frame Sent	Off-chip Bus	Sending a FRAME packet in the correct amount of time
Frame Reset	Off-chip Bus	Resetting the Bridge Core because no FRAME packet was received in time
Packet Sent	Off-chip Bus	Sending any type of packet
Timeout Resend	Off-chip Bus	Resending a packet that received no ACK or NAK within two clock cycles
Length Seven Patterns of Good and Bad Packets	Off-chip Bus	Receiving patterns of good and bad packets

Local Bus Goals

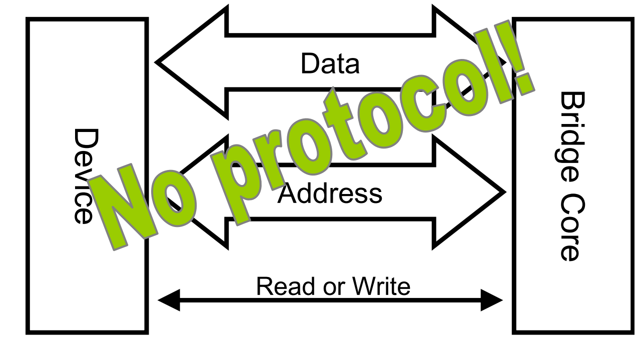
Cross-Coverage Goal

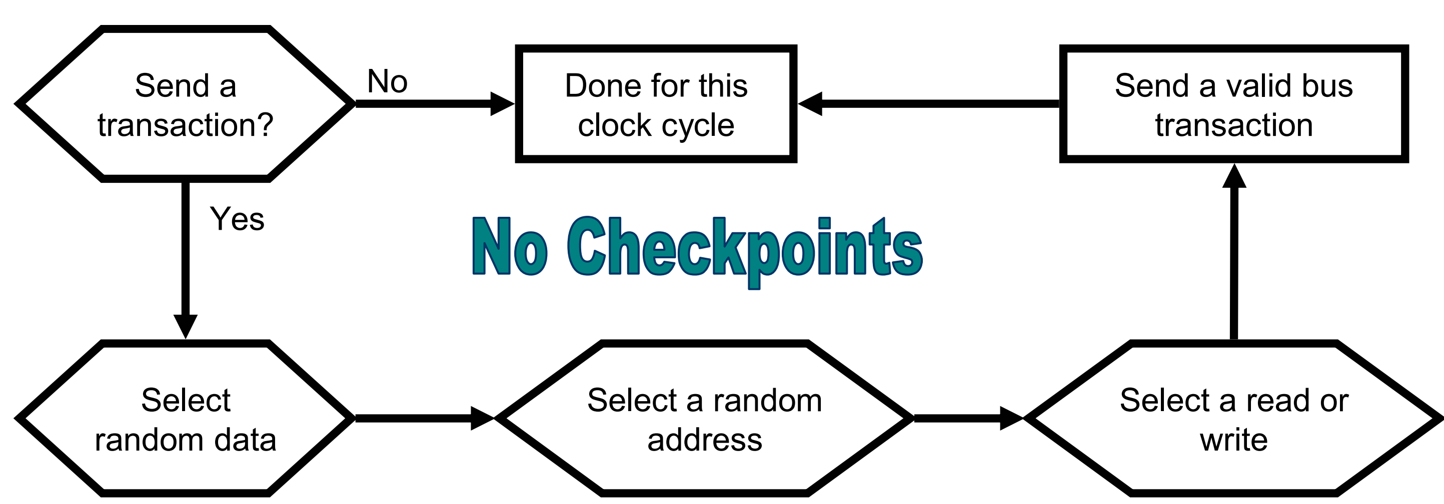
Off-Chip Bus Stimulus Generator



Local Bus Stimulus Generator

- Simpler than the Off-chip bus
- No ‘ACK/NAK’ protocol
- Bus is data corruption free





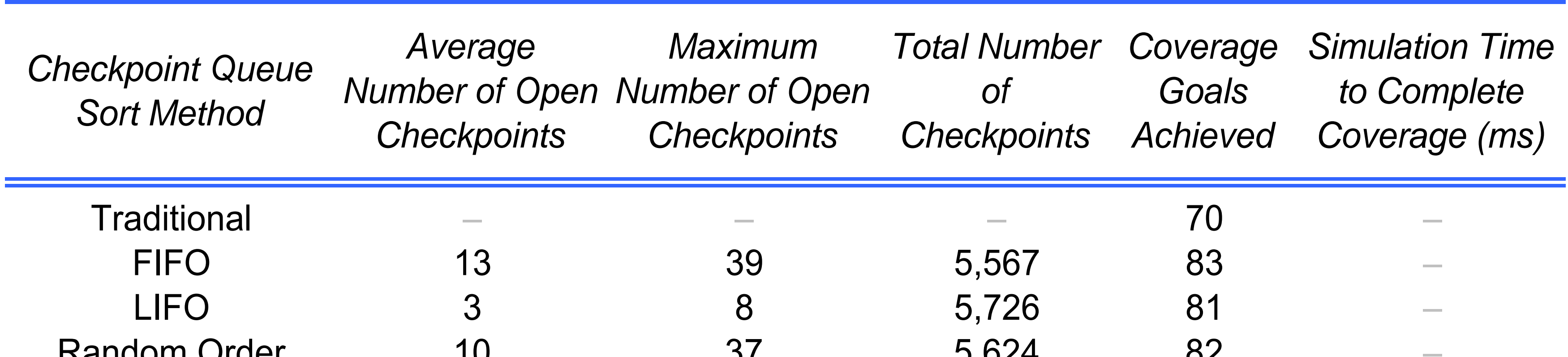
Applying Redundancy Reduction to the Bus Protocol Model

Method	*Coverage Goals Achieved*	*Simulation Time to Complete Coverage (ms)*	*Average Number of Open Checkpoints*	*Maximum Number of Open Checkpoints*	*Total Number of Checkpoints*
Inject an Error	81	–	13	39	5,567
Inject an Error, Select an Error Type	36	–	6	9	5,367
Send a Local Bus Transaction	46	–	4	10	6,049

- Again, the checkpointing must target the coverage goals
- ‘Select an Error Type’ over-verifies the sequences of good and bad packets
  - All bad packets are equivalent for that coverage goal
- Targeting the local bus coverage goals is a waste of simulation time
  - Only 2 local bus coverage goals

Queue Sort Methods with the Bus Protocol Model

- All three sort order methods perform better than the traditional version
- FIFO outperforms LIFO and Random
  - Most coverage goals achieved
  - Fewest checkpoints created
  - Most pending checkpoints ?



Checkpoint Queue Sort Method	*Average Number of Open Checkpoints*	*Maximum Number of Open Checkpoints*	*Total Number of Checkpoints*	*Coverage Goals Achieved*	*Simulation Time to Complete Coverage (ms)*
Traditional	–	–	–	70	–
FIFO	13	39	5,567	83	–
LIFO	3	8	5,726	81	–
Random Order	10	37	5,624	82	–