Model Checking Real Time Java --- Wrap Up Report

SCHOOL OF COMPUTING

NASA Ames Research Center
Robust Software Systems Group

Gary Lindstrom
Willem Visser
Peter C. Mehlitz

THE UNIVERSITY OF UTAH

SCHOOL OF COMPUTING

Outline

- Review
  - Motivation and approach
- RTSJ implementation strategy
  - In pure Java
  - Exploiting JPF
- Examples
  - Multiprogramming operating system
  - Cars crossing intersection
- Status and *To Do* list
  - Gleams in our eyes

THE UNIVERSITY OF UTAH

What Is Real Time Java?

- A Java extension supporting real time applications
  - Defined in *Realtime Specification for Java* (RTSJ)
  - New classes, e.g. **`RealtimeThread`**
  - New semantics for existing classes, e.g., **`java.lang.Thread`**
  - No language extensions, e.g., new syntax or keywords

Key Features of RTSJ

- Threads
  - Real time threads
- User definable schedulers
  - Default: priority based (32 levels)
  - Priority inversion avoided by *priority inheritance*
- Events and event handlers
  - Events bound to clocks and timers
  - Or external *happenings*

Other Features of RTSJ

- Memory areas
  - Scoped, immortal, physical
  - **`NoHeapRealtimeThreads`** not impeded by GC
- Asynchronous transfer of control (ATC)
  - Asynchronous interrupt handling
  - Within defined scopes
    - Those throwing **`AsynchronousInterruptException`**

RTSJ Validation Challenges

- More dynamic program behavior
  - Consequence of *no language extension*
  - Means more possible exceptions
- Role of time complicates correctness
  - Will a handler complete by its deadline?
  - If not, will overrun events be generated?
- How do we model environment in which RTSJ code runs?

Model Checking RTSJ

- System under test has 2 components
  - *Embedded* code: RTSJ code itself
    - e.g., flight control software
  - *Embedding* code: modeling environment
    - e.g., the airplane sensors and actuators
  - Both can be complex and difficult to specify
  - Both must be tested and validated

RTSJ Under JPF

- *Option 1*:
  - Extend JPF JVM to support RTSJ features
    - Custom scheduling, memory areas, ATC, …
- *Option 2*:
  - Don't change JPF JVM
  - Instead, add features external to JPF
  - *Bonus:* will run under ordinary Java

RTSJ Under JPF (cont'd)

- *Example:* custom schedulers
  - *Option 1*: make JPF scheduler programmable
    - But this is integral to JPF state management
  - *Option 2*: constrain the JPF scheduler to become deterministic
    - Run real time threads as coroutines
    - Define customized scheduler logic externally

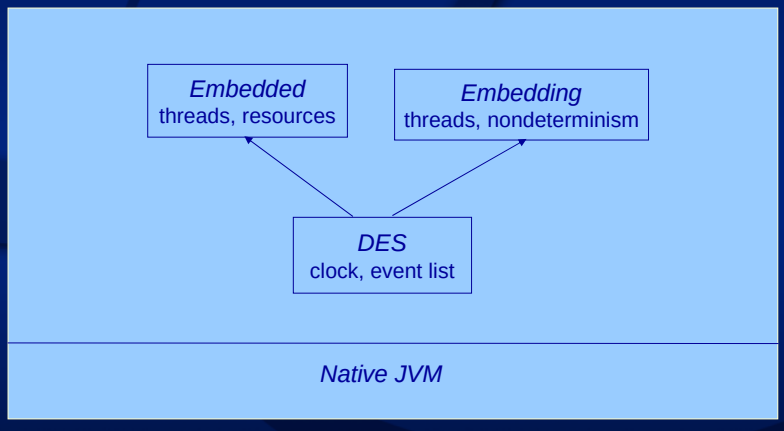
Our First Implementation Follows Option 2

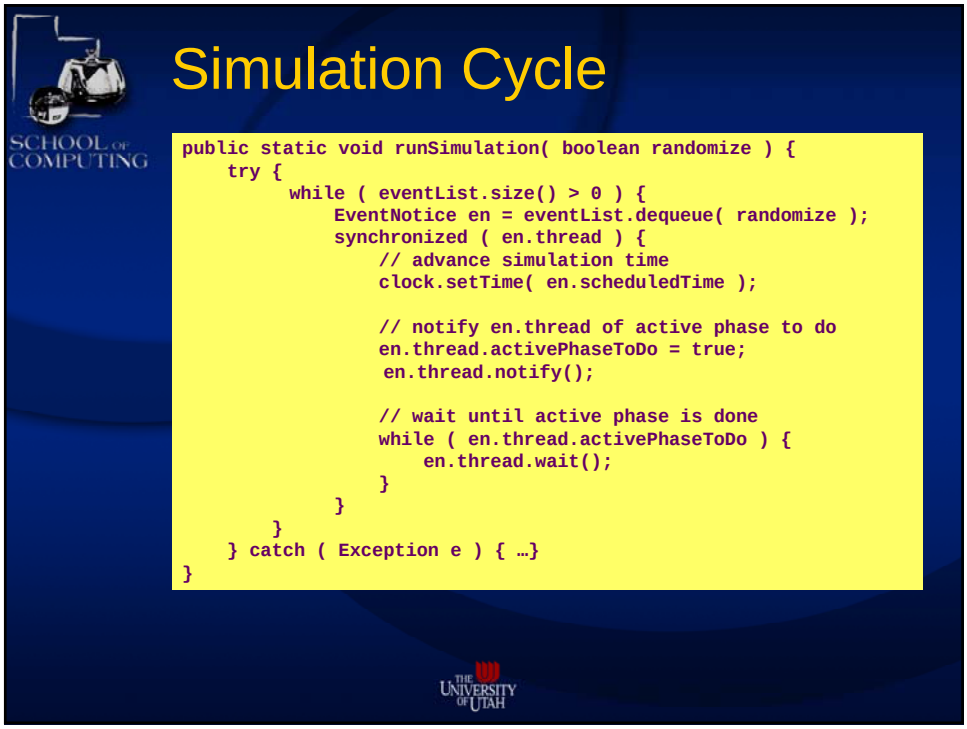
- Requirements
  - Discrete event simulation (DES) framework models time
  - 100% Java
  - Runs under native Java *or* JPF
    - *Native Java* => speed, portability
    - *JPF* => value added
      - Nondeterministic state exploration
      - Cost modeling, using JPF JVM instrumentation

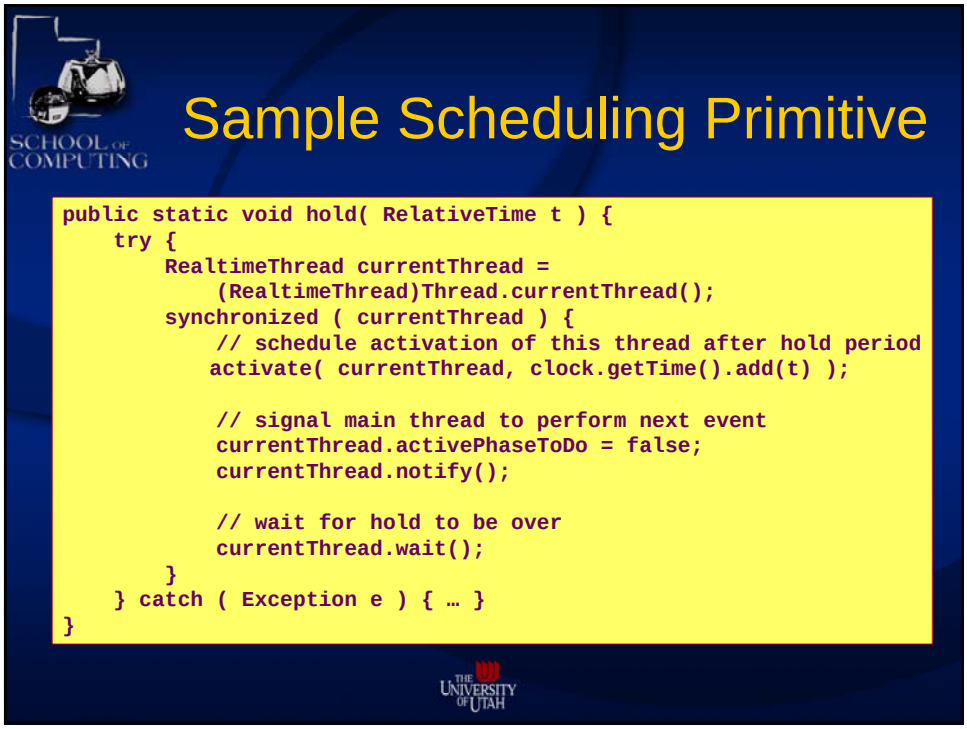
Key To External Scheduler: Resource Objects

- Focus of thread interactions
- Serializes thread possession
  - **`seize()`**: wait for resource to be free
    - Wait set is priority queue
    - Priority inheritance (PI) on holder thread
  - **`release()`**: relinquish resource
    - Dynamic priority may decrease due to PI
- Most vital application:
  - The CPU, which for which threads contend
  - This is how external scheduler mimics RTSJ default scheduler

Native Java Implementation



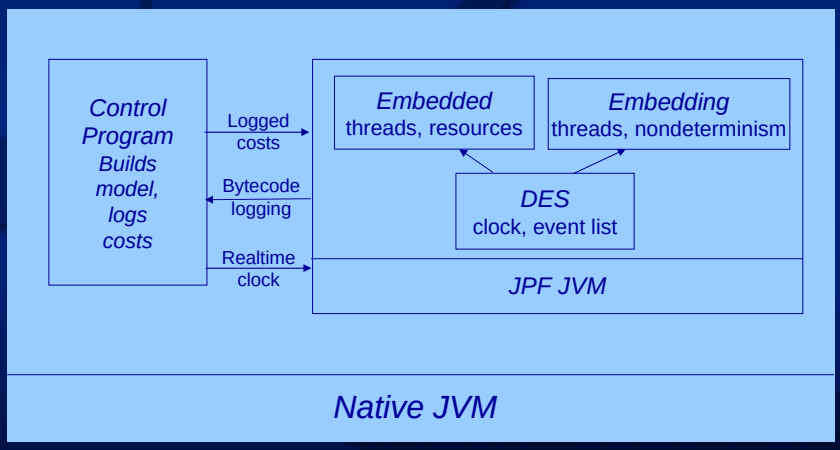

SCHOOL OF COMPUTING
Simulation Cycle
public static void runSimulation(boolean randomize) {
try {
while (eventList.size() > 0) {
EventNotice en = eventList.dequeue(randomize);
synchronized (en.thread) {
// advance simulation time
clock.setTime(en.scheduledTime);
// notify en.thread of active phase to do
en.thread.activePhaseToDo = true;
en.thread.notify();
// wait until active phase is done
while (en.thread.activePhaseToDo) {
en.thread.wait();
}
}
}
} catch (Exception e) { …}
}
THE UNIVERSITY OF UTAH


SCHOOL OF COMPUTING
Sample Scheduling Primitive
public static void hold(RelativeTime t) {
try {
RealtimeThread currentThread =
(RealtimeThread)Thread.currentThread();
synchronized (currentThread) {
// schedule activation of this thread after hold period
activate(currentThread, clock.getTime().add(t));
// signal main thread to perform next event
currentThread.activePhaseToDo = false;
currentThread.notify();
// wait for hold to be over
currentThread.wait();
}
} catch (Exception e) { … }
}
THE UNIVERSITY OF UTAH

JPF Implementation

- Two layer architecture
  - *Native Java JVM layer*
    - Runs start up program and JPF model checker
    - Listeners logging execution costs
  - *JPF value-added JVM layer*
    - State exploration
    - Property checking
    - Cost instrumentation

RTSJ / JPF Architecture



Listener Utilization

```
public static void main( String args[] ) {
        …
        JPF.addVMListener(theTestClient);
        …
}

public void instructionExecuted (VM vm) {
        JVM jvm = (JVM)vm;
        Instruction instruction = jvm.getLastInstruction();
        int byteCode = instruction.getByteCode();

        // catch opcodes without costs
        assert byteCodeCosts[byteCode] > 0;

        instructionCount++;
        byteCodeCounts[byteCode]++;
        totalCost += byteCodeCosts[byteCode];
}
```

MJI Utilization

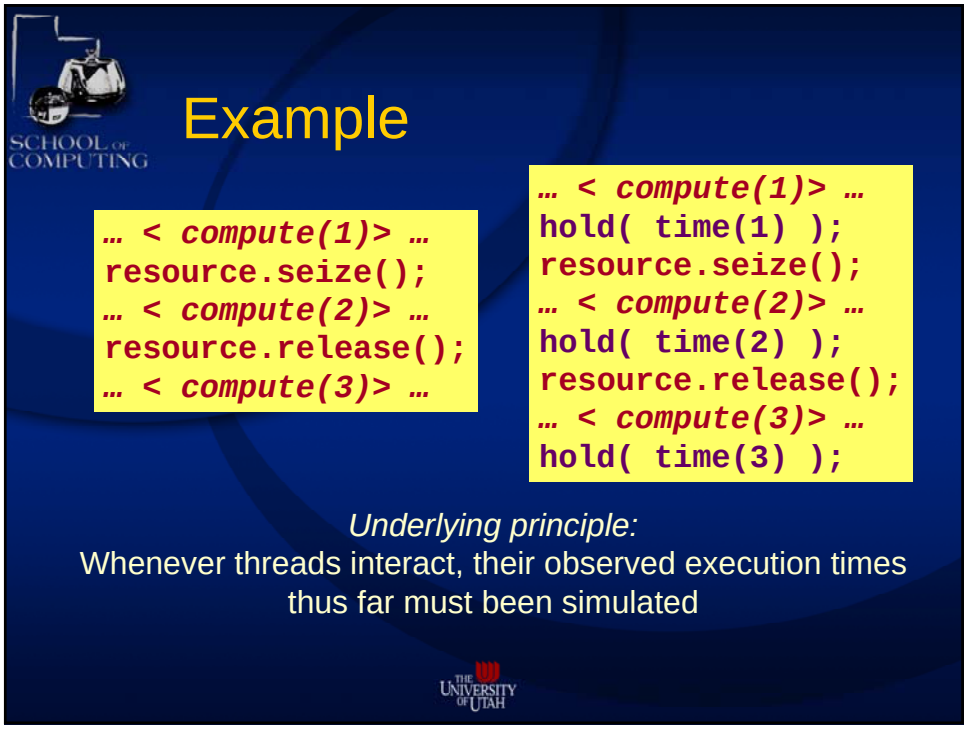
```
public class JPF_gov_nasa_jpf_rtsj_TestClient{

        public static int getCost(
                        MJIEnv env, int objRef ) {
                return gov.nasa.jpf.TestClient.getCost();
        }

        /*
         * so model can have access to true real time clock
         */
        public static long millisSinceEpoch(
                        MJIEnv env, int objRef ) {
                GregorianCalendar gc = new GregorianCalendar();
                return gc.getTimeInMillis();
        }
```

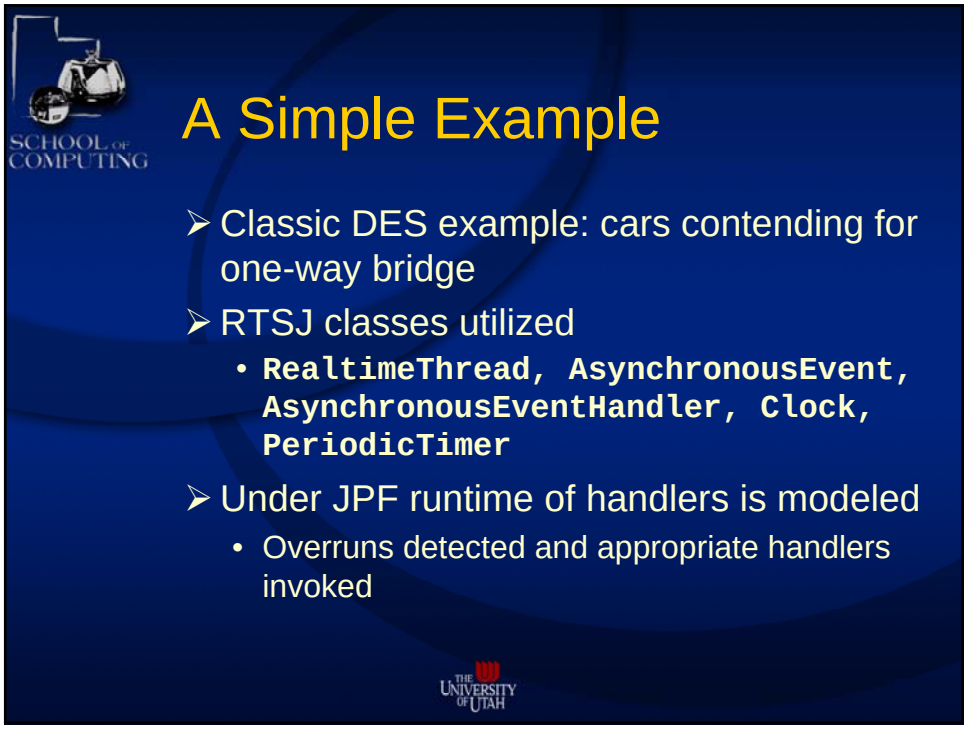
A Delicate Issue: Time

- In real systems, only time is real time
- In our system, we have 2 times:
  1. Simulated time
  2. Time (cost) of RTSJ code execution
- Under most simulations, second time is ignored
  - *Not here*, because ability to meet time deadlines is a crucial correctness issue
  - *Plus*, we have byte code logging capability

Options on Reconciling Simulated and Logged Times

1. Assume RTSJ code runs in zero time
   - Ignores important correctness issues
2. Log and simulate delays for each byte code
   - Costly, and invasive to JPF code base
3. The *Goldilocks* (just right) solution
   - Do delays for accumulated time at points where threads could interact


SCHOOL OF COMPUTING
Example
… < compute(1)> …
resource.seize();
… < compute(2)> …
resource.release();
… < compute(3)> …
… < compute(1)> …
hold(time(1));
resource.seize();
… < compute(2)> …
hold(time(2));
resource.release();
… < compute(3)> …
hold(time(3));
Underlying principle:
Whenever threads interact, their observed execution times thus far must been simulated
THE UNIVERSITY OF UTAH


SCHOOL OF COMPUTING
A Simple Example
Classic DES example: cars contending for one-way bridge
RTSJ classes utilized
RealtimeThread, AsynchronousEvent, AsynchronousEventHandler, Clock, PeriodicTimer
Under JPF runtime of handlers is modeled
Overruns detected and appropriate handlers invoked
THE UNIVERSITY OF UTAH

Performance Comparison

	Native Java	*JPF*
Deterministic	.041	13.94
Nondeterministic	.020	596.758

- Times are in seconds on 786MB Pentium 2 laptop
- *Nondeterministic* means:
  - *Native Java:* pseudo random selection
  - *JPF:* all possibilities -- 175 paths explored
- Single runs – only relative magnitudes are important

Second Example: Asaf Degani's Accident Scenario

- Recall talk two weeks ago
  - Actual commercial airline accident
  - Premature spoiler deployment caused airframe damage
  - Spoilers were not armed during pre-landing checklist
  - Due to delay in landing gear door retraction
  - Manual spoiler deployment was done before landing

Simplified RTSJ Model

- Two dimensional approach
  - 1 dimension horizontal (distance)
  - 1 dimension vertical (altitude)
- Pilot performs scenario
  - Crux is whether gear doors retract before flaps 25 action
- Gear/door deployment time is a normal distribution
  - Mean 28 sec, standard deviation 5.5 sec
  - JPF nondeterministically makes 3 draws
    - random, mean +/- 2 standard deviations

Additional Features

- 5 resource types
  - *FIFO*
  - *Priority*
  - *Priority ceiling*
  - *Priority Inheritance*
  - *Preemption*
- Detailed per thread cost accounting
  - With cost overrun and deadline miss handlers
- Physical memory model
  - With segments

Features Added Since 12/9

- 4 new resource types
  - *FIFO*
  - *Priority*
  - *Priority ceiling*
  - *Preemption*
  - (Had *Priority Inheritance* in 12/9)
- Detailed per thread cost accounting
  - With cost overrun and deadline miss handlers
- Physical memory model
  - With segments

Other Accomplishments

- Port to Open JPF
- JavaDoc
- Configuration control
  - Native Java vs. JPF
  - Choice mode
    - Deterministic
    - Pseudo random
    - Non deterministic (only JPF)

Multiprogramming OS Example

- Jobs contend for CPU resource
  - CPU types FIFO, priority, priority inheritance, priority ceiling, preemption
- Most interesting case is preemption
  - Requires notion of resource interrupt
  - **`hold(t,r)`** requires loop until full time **`t`** has elapsed while holding resource **`r`**
    -- time while **`r`** is stolen does not count

Comparative Results

	Job1(5)	*Job2(6)*	*Job3(4)*	*Job4(3)*	*CPU Busy*
FIFO	80ms/27%	101/40	101/37	101/38	95%
Priority	68/17	72/27	101/35	101/41	88%
PC(6)	68/16	73/26	100/30	101/45	88%
PI	69/16	73/26	101/30	101/41	88%
Preempt	65/19	54/0	101/39	101/62	85%

Notes:

- Job % is resource wait time
- Priority, PC and PI are essentially the same
- FIFO evens run time, but improves CPU usage
- Preempt speeds align with priorities

More Comprehensive Example

- Cars at uncontrolled intersection
  - Can go straight, turn left, or turn right
  - Admissible combinations are familiar
    - Straight through if:
      - Opposite car is not turning left
      - Car on left is not going straight or turning left
      - Car on right is not going straight or turning left or right
    - Similar rules for Left and Right turns

Modeling The Intersection: Four Sector Resources

Northbound shown
-- symmetric for other three directions

Experiment 1: Deadlocks

- How can we let cars in opposite directions both make left turns?
  - But not a car in a cross direction?
- For northbound (others analogous):
  - *Seize SE and NE*
  - *Release SE*
  - *Seize NW*
  - *Release NE and NW*
- What if we seized all three at once?
  - *Deadlock* – found easily by JPF

Experiment 2: Resource Types

- Each car is a real time thread
- Sectors have nominal transit time t
- Cars have priorities p in {1, …, 10}
  - Sector transit time = 10 * t / p
- Various sector resource types:
  - *FIFO, priority, priority inheritance, priority ceiling*
  - *Preempt* is physically impossible!

Sample Results

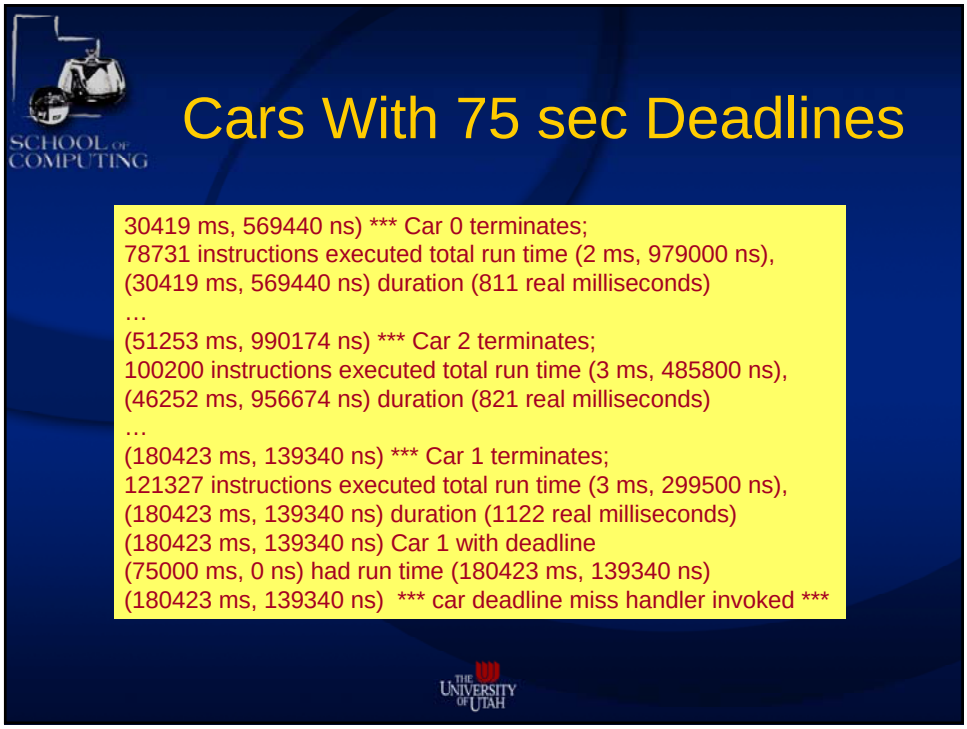
	Car 0 (5/N/S)	*Car 1 (2/S/L)*	*Car 2 (8/E/L delay 5s)*
FIFO	33s/0%	183/18	49/21
Priority	33/0	183/18	49/21
PC(8)	25/0	63/40	45/15
PI	30/0	180/16	46/17

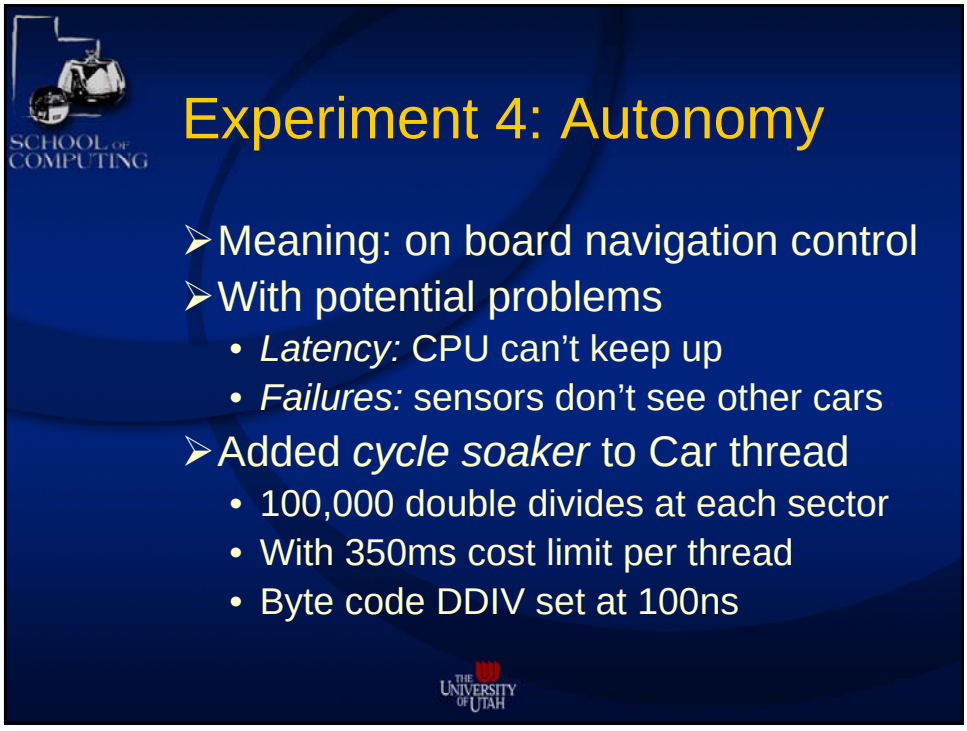
Notes:

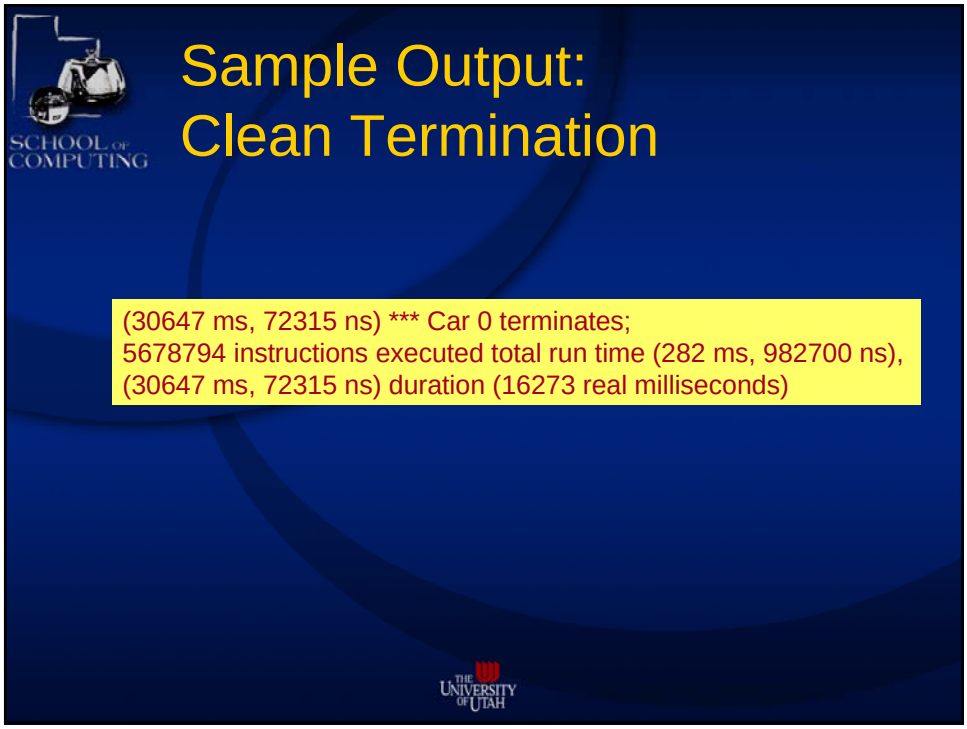
- Nominal sector time 10 sec.
- FIFO, Priority are essentially the same
- Cars 0 and 1 benefit from car speed increase under PC
- Car 1 gets small speed up under PI

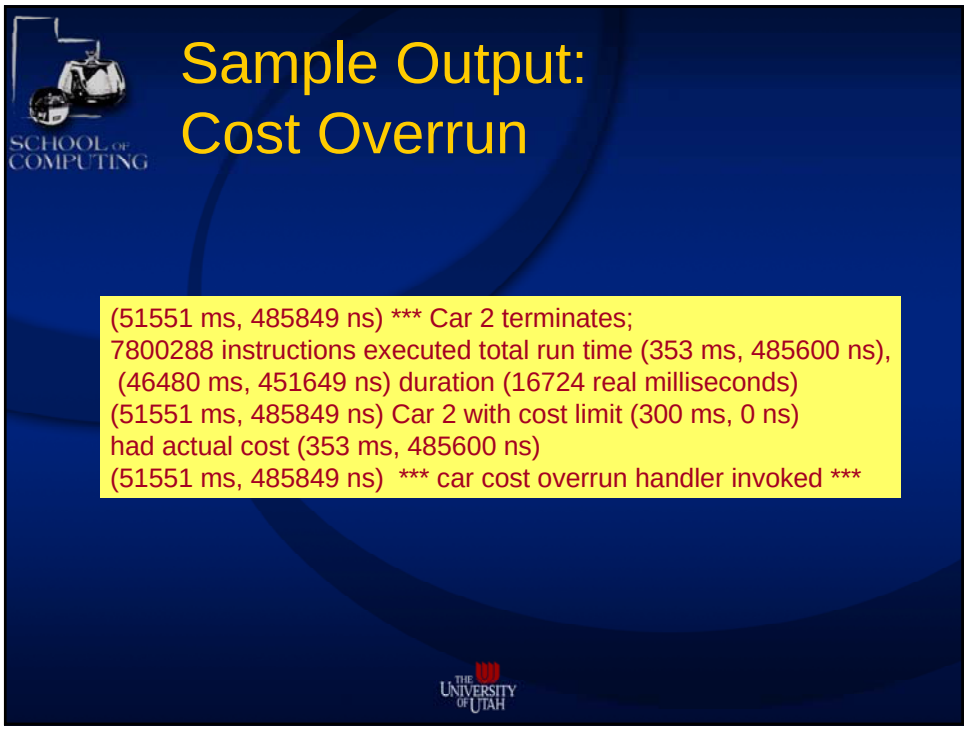
Experiment 3: Deadlines

- Real time threads can be given miss handlers
  - Invoked when deadline is not met on thread completion
- This makes sense in Native Java, as well as JPF
  - Later we will deal with cost overruns, which require JPF listeners on JVM


SCHOOL OF COMPUTING
Cars With 75 sec Deadlines
30419 ms, 569440 ns) *** Car 0 terminates;
78731 instructions executed total run time (2 ms, 979000 ns),
(30419 ms, 569440 ns) duration (811 real milliseconds)
…
(51253 ms, 990174 ns) *** Car 2 terminates;
100200 instructions executed total run time (3 ms, 485800 ns),
(46252 ms, 956674 ns) duration (821 real milliseconds)
…
(180423 ms, 139340 ns) *** Car 1 terminates;
121327 instructions executed total run time (3 ms, 299500 ns),
(180423 ms, 139340 ns) duration (1122 real milliseconds)
(180423 ms, 139340 ns) Car 1 with deadline
(75000 ms, 0 ns) had run time (180423 ms, 139340 ns)
(180423 ms, 139340 ns) *** car deadline miss handler invoked ***
THE UNIVERSITY OF UTAH


SCHOOL OF COMPUTING
Experiment 4: Autonomy
Meaning: on board navigation control
With potential problems
Latency: CPU can't keep up
Failures: sensors don't see other cars
Added cycle soaker to Car thread
100,000 double divides at each sector
With 350ms cost limit per thread
Byte code DDIV set at 100ns
THE UNIVERSITY OF UTAH


SCHOOL OF COMPUTING
Sample Output:
Clean Termination
(30647 ms, 72315 ns) *** Car 0 terminates;
5678794 instructions executed total run time (282 ms, 982700 ns),
(30647 ms, 72315 ns) duration (16273 real milliseconds)
THE UNIVERSITY OF UTAH


SCHOOL OF COMPUTING
Sample Output:
Cost Overrun
(51551 ms, 485849 ns) *** Car 2 terminates;
7800288 instructions executed total run time (353 ms, 485600 ns),
(46480 ms, 451649 ns) duration (16724 real milliseconds)
(51551 ms, 485849 ns) Car 2 with cost limit (300 ms, 0 ns)
had actual cost (353 ms, 485600 ns)
(51551 ms, 485849 ns) *** car cost overrun handler invoked ***
THE UNIVERSITY OF UTAH

Sample Output: Both Cost and Deadline Overruns

```
(180930 ms, 717415 ns) *** Car 1 terminates;
9922129 instructions executed
   total run time (353 ms, 299200 ns),
(180930 ms, 717415 ns) duration (23153 real milliseconds)
(180930 ms, 717415 ns) Car 1 with cost limit (300 ms, 0 ns)
had actual cost (353 ms, 299200 ns)
(180930 ms, 717415 ns)  *** car cost overrun handler invoked ***
(180930 ms, 717415 ns) Car 1 with deadline (75000 ms, 0 ns)
had run time (180930 ms, 717415 ns)
(180930 ms, 717415 ns)  *** car deadline miss handler invoked ***
```

Sample Output Sections

```
[313674:0] altitude 1800, runway 28691 ft.,
    gear change requested from up to down
[330674:0] altitude 1800, runway 24203 ft.,
    gear change from up to down completed
[330674:0] altitude 1800, runway 24203 ft.,
    spoilers armed
…
[470180:0] altitude 0, runway 1 ft., on ground
[470180:0] altitude 0, runway 1 ft., >>> spoilers deploy automatically <<<
```

```
[313674:0] altitude 1800, runway 28691 ft.,
    gear change requested from up to down
[352674:0] altitude 1800, runway 18395 ft.,
    gear change from up to down completed
[352674:0] altitude 1800, runway 18395 ft.,
    doors not retracted -- arm spoilers manually!
…
[452400:0] altitude 0, runway 1 ft., on ground
[452400:0] altitude 0, runway 1 ft., >>> deploy spoilers manually! <<<
```

Gleams In Our Eyes

- Customized state abstractions
  - e.g., focused on event list structure and history
- Symbolic, constraint based analysis
  - Exploit event causality relationships
- Other applications
  - Generation of test scripts
  - Generation of procedures (scripts)

For More Information

- Consult project web site
  - http://www.cs.utah.edu/~gary/RTSJ/
  - Includes:
    - ATVA '05 paper
    - RTSJ API Javadoc
    - Examples
    - These slides