Deterministic Replay of Java Multithreaded Applications

Jong-Deok Choi and

Harini Srinivasan

slides made by Qing Zhang

- Non-Determinism in Java Threads
  - Threads and concurrency constructs in java introduce non-determinism to a program's execution.
  - Non-determinism in execution behavior makes it impossible to use execution replay for debugging, performance monitoring, or visualization.

- That's Why DejaVu was Introduced

- What is DejaVu?
  - DejaVu is a tool for Java
  - Uses the record/replay model
  - Provides deterministic replay of multithreaded Java applications

- DajaVu Quick facts
  - Developed as an extension of the Sun Microsystems' JVM
  - Runs in Two Modes
    - Record Mode – records the logical thread schedule information of the execution while the Java Program runs.
    - Replay Mode – reproduces the execution behavior of the program by enforcing the recorded logical thread schedule.

- DejaVu needs to capture the thread schedule information during one execution of the program
  - What is thread schedule?
    - A sequence of time intervals each containing the execution events of a single thread.
  - Capturing the actual thread schedule is not always possible.

- How to go around the problem?
  - Rather than relying on the underlying physical thread scheduler, use *logical thread schedule* information that can be computed without any help from the thread scheduler.

- Capturing *logical thread schedule*
- Basic Terms
  - Critical Pts - *synchronization events and shared variable accesses*
  - Logical Thread schedule – *is a sequence of intervals of critical events, wherein each interval corresponds to the critical and non-critical events executing consecutively in a specific thread.*

- Tracing critical events
  - It is not necessary to trace all critical points
  - One only needs to trace the first and last critical events in a critical event interval.
  - i.e. tracing critical event intervals rather than critical points
  - This will suffice to keep track of the logical thread schedule
  - Will improve performance

- Properties of Logical Schedule Interval
  - All critical events of the logical schedule interval belong to the same thread
  - Given any two critical events Ci and Cj of the logical interval everything between Ci and Cj also belong to this logical scheduled interval
  - No two adjacent intervals belong to the same thread

- Technique in Capturing Logical Thread Schedule
  - The Global Clock
    - The technique is based on a Global Clock
    - The global Clock ticks at each execution of a critical event to identify each critical event

– Representation of Schedule Interval

- Each schedule interval can be represented by
  <FirstCriticalEvent(i), LastCriticalEvent(i)>

- Where *FirstCriticalEvent* and *LastCriticalEvent* can be represented by their global clock values

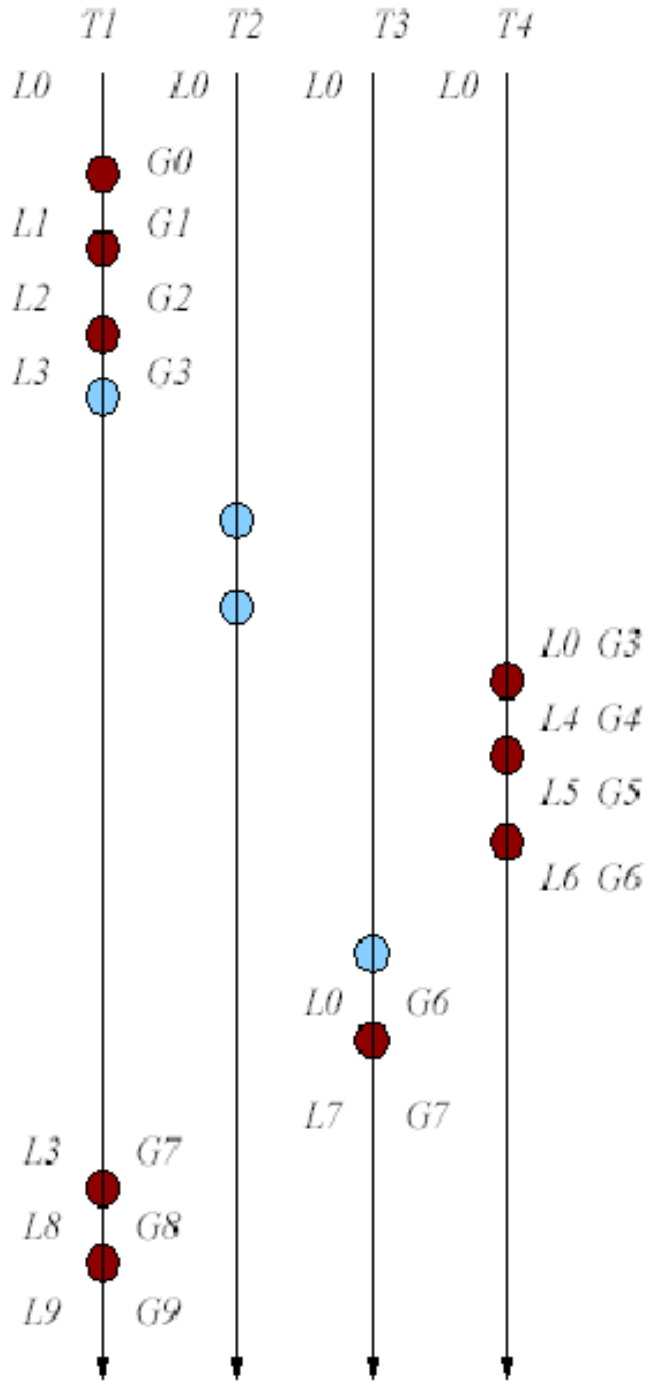
- Each thread captures the *FirstCriticalEvent* and *LastCriticalEvent* pair of each of it's scheduled intervals

- Identifying Schedule Intervals
  - Global clock vs. Local Clock
    - They start at the same value
    - Local clock will stay behind when a different thread executes a critical event
    - A thread will execute a critical event and increment the global clock
    - When a thread executes a critical event it can compare the values of the local clock and the global clock to detect the end of a previous schedule interval.

● : thread-shared variable

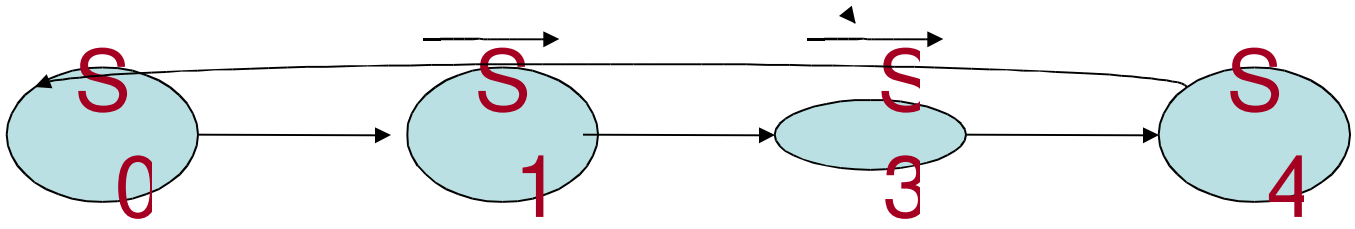
● : thread-local variable

- Gi: Global Clock Val
- Li: Local Clock Val
- T1 <0,2>, <7,8>
- T2
- T3 <6,6>
- T4 <3,5>



- In order to identify the schedule intervals three operations must be atomic
  - Assign Global Clock
    - Assigning the global clock value to a crit. event
  - Update Global Clock
    - Incrementing the global clock
  - Critical Event
    - Execution of a critical event

- Replaying Execution
  - At the end of the record mode dejaVu creates a file that contains all the thread schedule information.
  - The replay mode will read the file and execute the replay according to the files.
  - Replay will use the information as transitions rules of the following DFA



- S0 - Load FirstCriticalEvent and LastCriticalEvent
- S1 - Yield thread schedule
- Arrow 1 - if global clock < FirstCriticalEvent of current thread
- S3 - Execute event - If CriticalEvent increment Global Clock
- Arrow 2 - if global clock <= LastCriticalEvent of current thread
- S4 - Start next thread

- Implementaion and performance Results
  - dejaVu was implemented on java_g interpreter
  - Java_g is mostly written in c and easy to instrument
  - However, it suffers from slower performance than regular java interpreters such as JIT

Application		Number of Threads	Native Execution (sec)	Instrumented Execution		Replay Execution		Trace Size	Critical Events	Thread Switch Intervals
				Time (sec)	Ovrhd	Time (sec)	Ovrhd			
Synthetic Work Load	Chaos	16	11.6	21.7	87.6%	18.3	57.8%	4KB	10M	0.5K
		32	22.6	42.5	88.1%	35.5	57.1%	8KB	19M	1K
		64	47.6	84.5	77.5%	71.7	50.6%	16KB	39M	2K
	MM	4	26.8	43.9	63.8%	35.8	33.6%	4KB	15M	0.5K
		16	26.6	44.1	65.8%	35.9	35.0%	4KB	15M	0.5K
		64	26.8	44.5	66.0%	35.8	33.6%	5KB	15M	0.7K
	SOR	4	5.1	8.4	64.7%	6.8	33.3%	1.3KB	2.3M	0.2K
		16	5.1	8.5	66.6%	7.0	37.3%	4KB	2.3M	0.5K
		64	5.5	8.8	60.0%	7.4	34.5%	8KB	2.4M	2K
	MTD	4	20.1	20.1		20.1		0.5KB	3K	0.1K
		16	20.1	20.1		20.1		1KB	5K	0.1K
		64	20.1	20.1		20.1		2KB	12K	0.3K
SPLASH Kernels	Radix	4	53.7	77.6	44.5%	74.5	38.7%	16KB	2.3M	2K
		16	56.7	81.4	43.6%	76.1	34.2%	17KB	2.6M	2.3K
		64	66.8	104.7	56.7%	99.8	49.4%	36KB	40M	4.5K
	FFT	16	4.98	5.95	19.5%	6.55	31.5%	6KB	1M	0.5K
		32	5.15	6.18	20.0%	8.52	65.4%	9KB	1M	1K
		64	5.82	6.81	17.0%	8.04	38.1%	18KB	1M	2.4K

Chaos – high non-deterministic multi-threaded program

MM – matrix multiply (few threads)

SOR – Successive Over Relaxation for financial application (few Threads)

MTD – Thread Schedule Simulator (lots of synchronization)

SPLASH – Stanford Parallel Applications for Shared Memory

- Future Work
  - dejaVu will run on multi processors with great overhead due to more intervals
  - Implementation of multiple Global Clocks to reduce overhead in parallel CPU systems