Efficient Verification of Periodic Programs Using Sequential Consistency and Snapshots

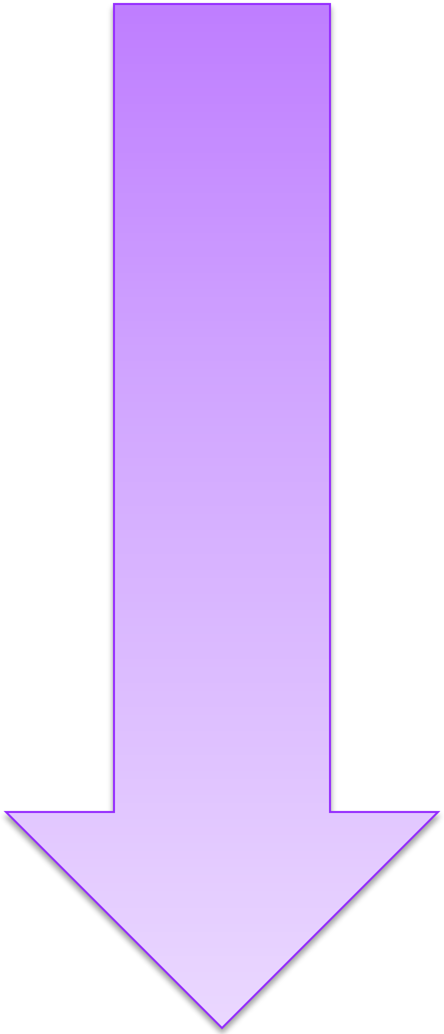
Sagar Chaki, Arie Gurfinkel, Nishant Sinha
October 24, 2014

FMCAD'14, Lausanne, Switzerland

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0001817

Outline



- Context
    - Periodic Programs
    - Time-Bounded Verification

- Verification Condition Generation
    - Hierarchical Lamport Clocks
    - Locks
    - Snapshotting

- Experimental Results

- Related Work

Periodic Embedded Real-Time Software

Technical Name

Periodic Fixed-Priority Software with Preemptive Rate Monotonic Scheduling

Task	Period
Engine control	10ms
Airbag	40ms
Braking	40ms
Cruise Control	50ms
Collision Detection	50ms
Entertainment	80ms

***Domains:* Avionics, Automotive**

***OS:* OSEK, VxWorks, RTEMS**

We call them periodic programs

Time-Bounded Verification [FMCAD'11&'14, VMCAI'13]

Input: Periodic Program

- Collection of periodic tasks
  - Execute concurrently with preemptive priority-based scheduling
  - Priorities respect RMS
  - Communicate through shared memory

Problem: Time-Bounded Verification

- Assertion A violated within X ms of a system's execution from initial state I?
  - A, X , I are user specified
  - Time bounds map naturally to program's functionality (e.g., air bags)

Solution: Bounded Model Checking

- Generate Verification Condition (SMT Formula over Bit-Vectors)
- Use SMT Solver to check satisfiability

Main focus of this paper

Periodic Program (PP)

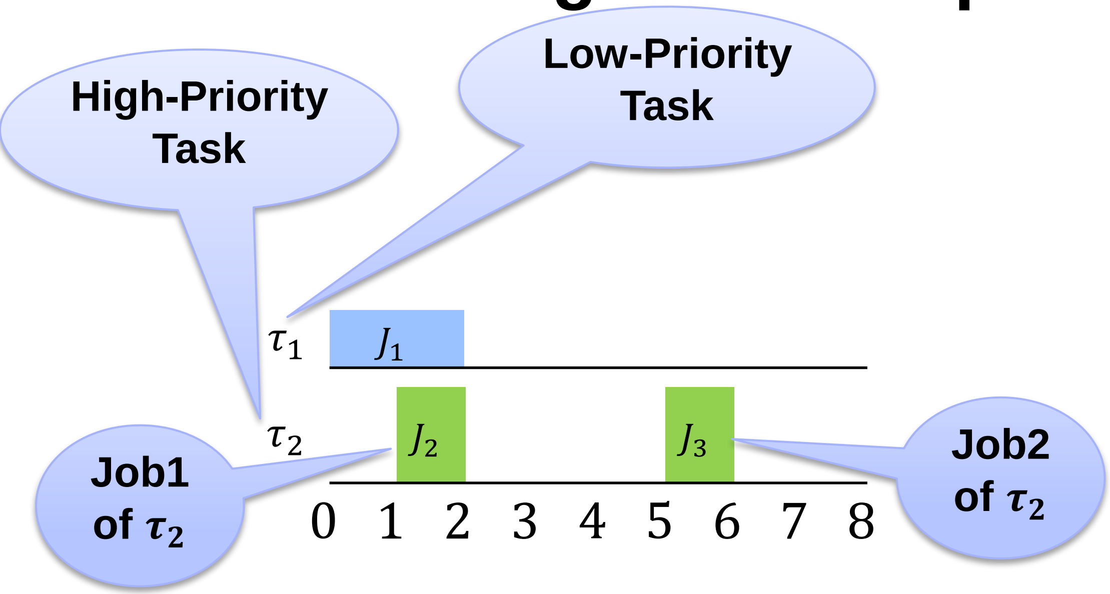
An N-task periodic program PP is a set of tasks $\{\tau_1, \ldots, \tau_N\}$

A task τ is a tuple $\langle I, T, P, C, A \rangle$, where

- I is a task identifier = its priority
- T is a task body (i.e., code)
- P is a period
- C is the worst-case execution time
- A is the *release time:* the time at which task becomes first enabled

Semantics of PP bounded by time $X \equiv$ asynchronous concurrent program:

Periodic Program Example

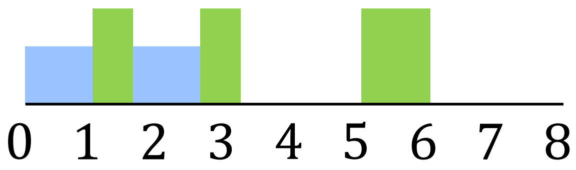

High-Priority Task

Low-Priority Task

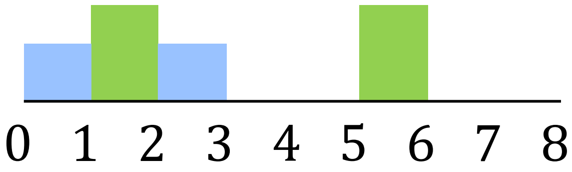
Job1 of τ_2

Job2 of τ_2

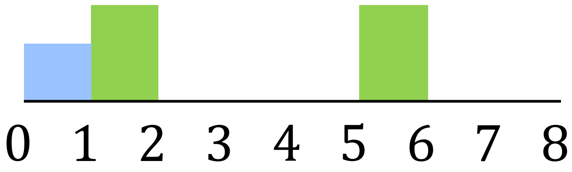
Illegal Execution – τ_1 preempts τ_2



$$\tau_1 = \langle 1, J_1, 8, 2, 0\rangle, \quad \tau_2 = \langle 2, J_2 = J_3, 4, 1, 1\rangle$$



Legal Execution – τ_1 executes for 2 units



Another Legal Execution – τ_1 executes for 1 units

Verification Condition

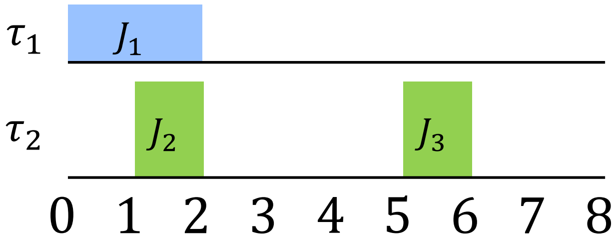
$$VC = VC_{seq} \wedge VC_{clk} \wedge VC_{obs}$$

Encodes Purely Job-local computation. Value read/written by each Shared Variable access represented by a fresh variable.

Associates each shared variable access with a hierarchical Lamport Clock. Constraints values of Clock components based on timing and priority.

Connects value read at each "read" to the value written by most recent "write" according to the Lamport Clock.

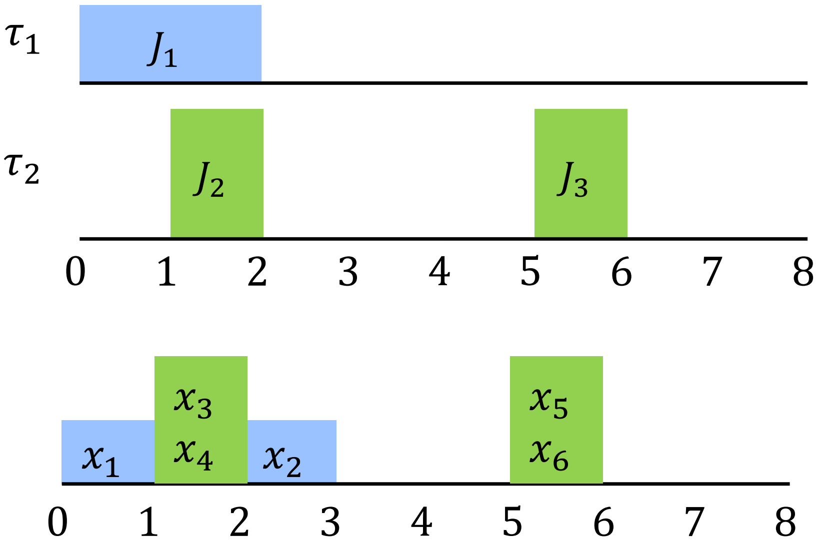
Verification Condition VC_{seq}



Same as verification condition for sequential program except that both reads and writes are given fresh variables

$$\left.\begin{array}{lcl}
J_1()\,\{\,x := x + 1;\,\} & \xrightarrow{\hspace{3cm}} & x_2 = x_1 + 1 \\
 & & \wedge \\
J_2()\,\{\,x := x + 1;\,\} & \xrightarrow{\hspace{3cm}} & x_4 = x_3 + 1 \\
 & & \wedge \\
J_3()\,\{\,x := x + 1;\,\} & \xrightarrow{\hspace{3cm}} & x_6 = x_5 + 1
\end{array}\right\} VC_{seq}$$

Verification Condition VC_{clk}



Observe: x_i is accessed before x_j iff $(R_i, \pi_i, \iota_i) < (R_j, \pi_j, \iota_j)$

where $<$ is lexicographic ordering

Claim/Intuition: This holds for all legal executions, not just this one.

Therefore: Associate x_i with hierarchical Lamport clock $\kappa_i = (R_i, \pi_i, \iota_i)$

- $\pi_i = priority\ of\ job\ accessing\ x_i$
  - $\pi_1 = \pi_2 = 1, \pi_3 = \cdots = \pi_6 = 2$
- $R_i = \#of\ jobs\ finished\ before\ x_i\ accessed$
  - $R_1 = R_3 = R_4 = 0, R_2 = 1, R_5 = R_6 = 2$
- $\iota_i = index\ of\ instruction\ accessing\ x_i\ in\ topological\ ordering\ of\ CFG$
  - $\iota_1 = \iota_3 = \iota_5 = 1, \iota_2 = \iota_4 = \iota_6 = 2$

$\Bigg\}\ VC_{clk}$

Verification Condition VC_{obs}

Let J_i = job in which x_i is accessed

Compute: $J \sqsubset J'$ if J always completes before J' starts

Recall $\kappa_i = (R_i, \pi_i, \iota_i)$. For each read x_i, let

$W_i = \{x_j | x_j\ is\ a\ write \wedge \neg(J_i \sqsubset J_j)\}$, i.e., the set of all writes that x_i "may observe"

$VC_{obs} \equiv$

The value of each x_i accessed by a read equals the value of x_j such that $\kappa_j = max\{\kappa_k | \kappa_k < \kappa_i\ and\ x_k \in W_i\}$, where $max\{\} =$ initial value of x.

Verification Condition VC_{obs}

For each read x_i introduce $\tilde{\kappa}_i$ = clock of write action observed

$$VC_{obs} \equiv$$
$$\bigwedge_{x_j \in W_i} \kappa_j < \kappa_i \Rightarrow \kappa_j \leq \tilde{\kappa}_i$$
$$\wedge$$
$$\left(\left(VC^1_{obs}\right) \vee \left(\bigvee_{x_j \in W_i} VC^2_{Obs}(j)\right)\right)$$

x_i observes initial value x_{Init} of x

$$VC^1_{obs} \equiv \left(\bigwedge_{x_j \in W_i} \kappa_j \geq \kappa_i\right) \wedge \left(x_i = x_{Init}\right)$$

$$VC^2_{Obs}(j) \equiv \left(\kappa_j < \kappa_i \wedge \kappa_j = \tilde{\kappa}_i\right) \wedge x_i = x_j$$

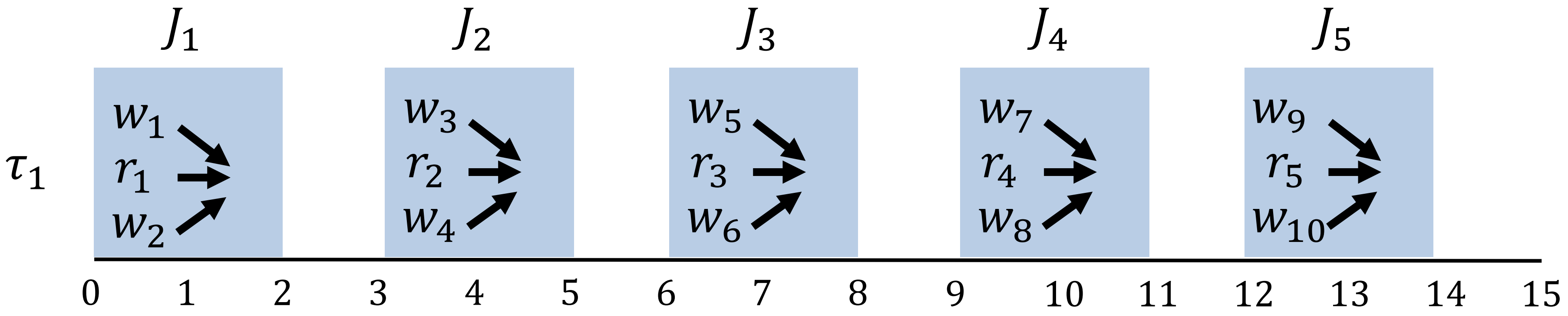
x_i observes x_j

In the paper, we handle multiple shared variables.

Snapshotting: Problem

$w_i = write, r_i = read$

Sequence of jobs. Each job writes to a variable multiple times.



Series-Parallel Structure

Observe: $W(r_1) = \{w_1, w_2\}, W(r_2) = \{w_1, w_2, w_3, w_4\}, W(r_3) = \{w_1, w_2, w_3, w_4, w_5, w_6\}, \ldots$

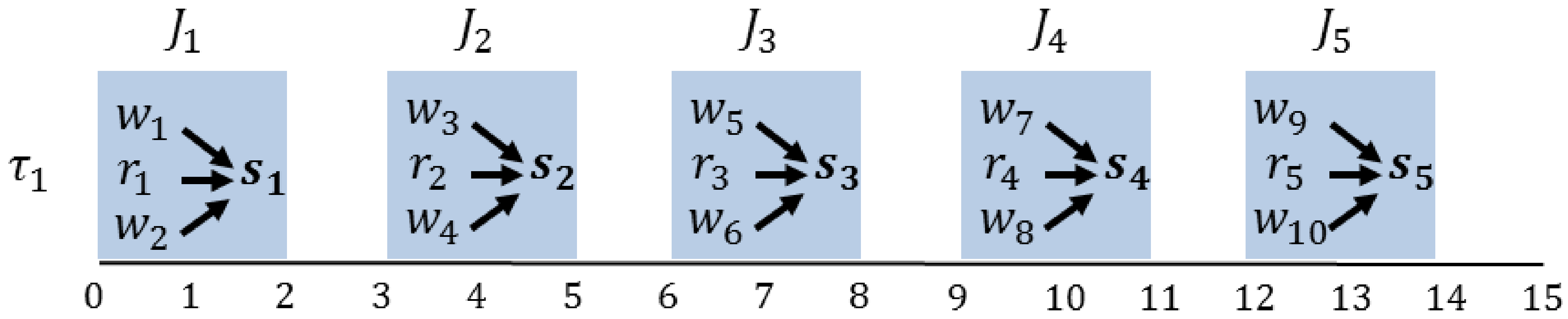
Result: Problem for $r_{<i}$ gets re-encoded (and resolved) as part of problem for r_i

Empirically: SMT solvers do not scale beyond small number of jobs

Snapshotting: Solution

$w_i = write, r_i = read$
$s_i = snapshot$

Snapshot: Atomically read and write variable at the end of the job. Dominates all other access in the job.



<u>Now:</u> $W(r_1) = W(s_1) = \{w_1, w_2\}, W(r_2) = W(s_2) = \{s_1, w_3, w_4\},$
$W(r_3) = W(s_3) = \{s_2, w_5, w_6\}, \ldots$

<u>Result:</u> Solving VC_{obs} involves fewer redundant computation

<u>Empirically:</u> SMT solvers scale beyond small number of jobs

<u>Choice</u> of variables to snapshot: (i) all variables (ii) only written by the job

Verification Condition VC_{obs} with Snapshotting

Input: $Snaps(J)$ = set of variables snapshotted by J

Compute: Relation $J \uparrow J'$ iff J can be preempted by J'

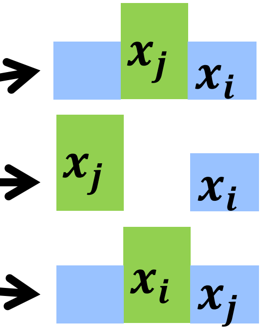
Let $\Psi_{\sqsubset}(J, g)$ = maximal jobs less that J that snapshot g

Let $\Psi_{\uparrow}(J, g) = \{J' \mid J \uparrow J' \wedge g \in Snaps(J')\}$

Let $\Psi_{\downarrow}(J) = \{J' \mid J' = J \vee J' \uparrow J\}$

These relations capture the series-parallel structure

$$
\begin{aligned}
W_i = & \left\{x_j \,\middle|\, x_j \; is\ a\ snapshot \wedge J_j \in \Psi_{\uparrow}(J_i, g)\right\} \cup \\
& \left\{x_j \,\middle|\, x_j \; is\ a\ snapshot \wedge J_j \in \Psi_{\sqsubset}(J_i, g)\right\} \cup \\
& \left\{x_j \,\middle|\, x_j \; is\ a\ write \wedge J_j \in \Psi_{\downarrow}(J_i, g)\right\}
\end{aligned}
$$



$VC_{obs} \equiv$ same as before with the new definition of W_i above

Handling Locks

We handle two types of locks (both involve changing priorities)

- Each thread has a base priority = priority of task it executes
- Each PCP lock l is associated with priority $\pi(l)$
  - A CPU lock is a PCP lock such that $\pi(l) = \infty$
- Thread's priority = max (its base priority, priorities of all PCP locks it holds)

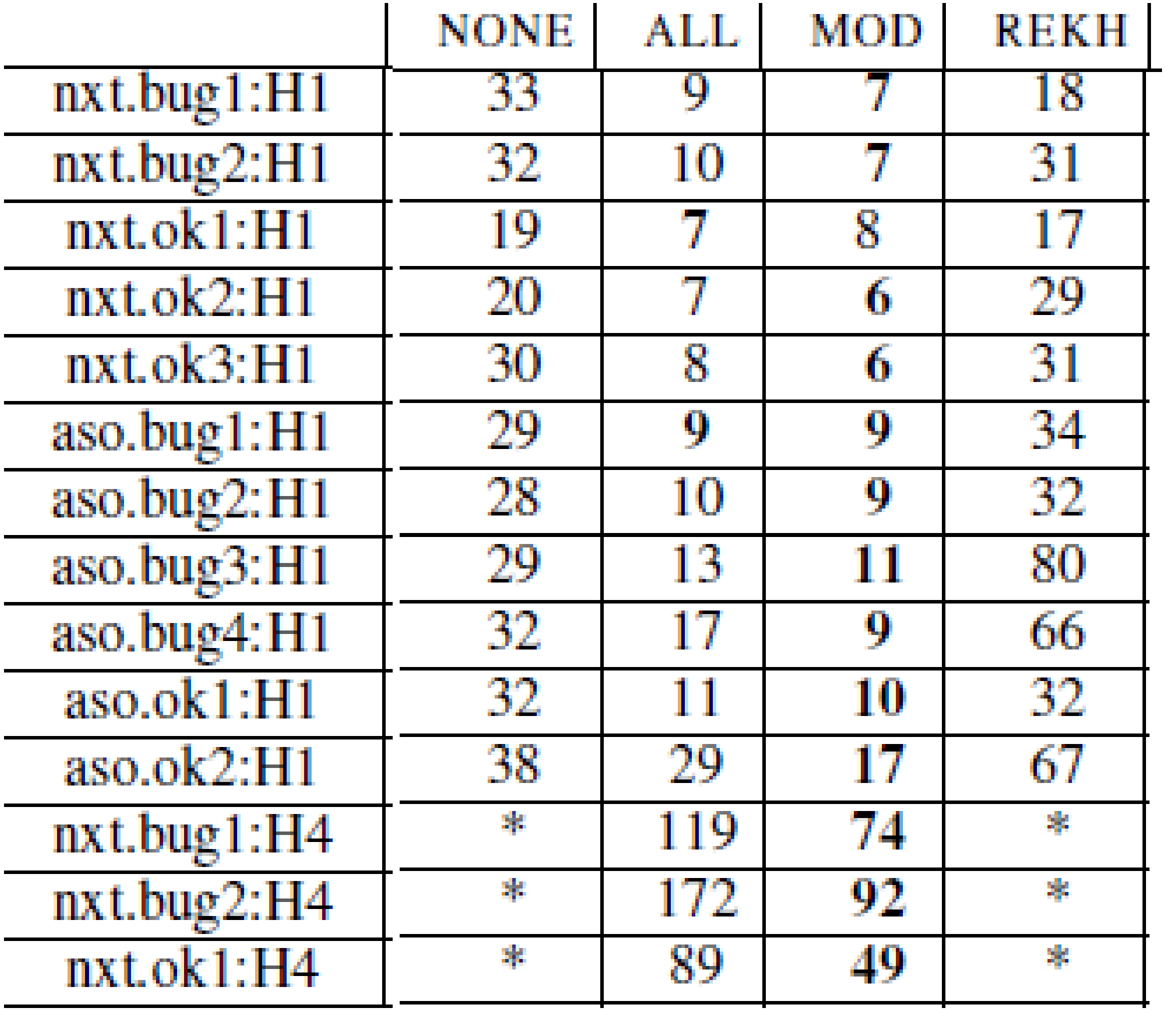
Lock operation encoded by "priority-test-and-set" action (J, pc, π_t, L_r, L_a)

- Guard: All held locks must have priority less than π_t
- Command: Locks in L_r are released; Locks in L_a are acquired
- Encode by updating VC_{clk} and VC_{obs} appropriately

Note: To handle locks, we generalize VC-Gen to support operations that read and write program state (in this case held locks) atomically

- Atomic operations handled similarly to snapshots

Results (Time in seconds)



	NONE	ALL	MOD	REKH
nxt.bug1:H1	33	9	7	18
nxt.bug2:H1	32	10	7	31
nxt.ok1:H1	19	7	8	17
nxt.ok2:H1	20	7	6	29
nxt.ok3:H1	30	8	6	31
aso.bug1:H1	29	9	9	34
aso.bug2:H1	28	10	9	32
aso.bug3:H1	29	13	11	80
aso.bug4:H1	32	17	9	66
aso.ok1:H1	32	11	10	32
aso.ok2:H1	38	29	17	67
nxt.bug1:H4	*	119	74	*
nxt.bug2:H4	*	172	92	*
nxt.ok1:H4	*	89	49	*

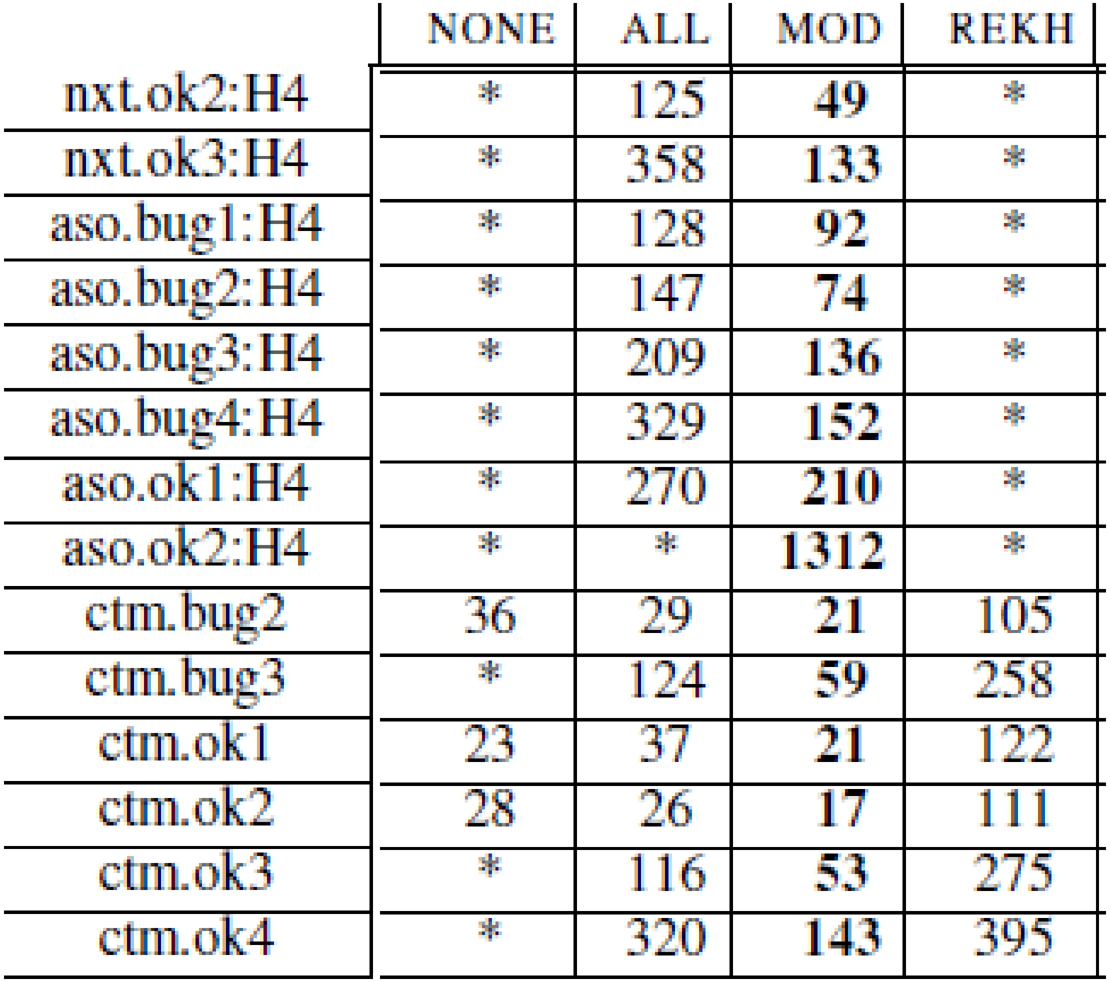
2GB Memory Limit

60min Time Limit

Solver=STP

NONE=No snapshotting, ALL=Snapshot all variables, MOD=Snapshot only modified variables, REKH=Previous tool based on sequentialization

Results (Time in seconds)



	NONE	ALL	MOD	REKH
nxt.ok2:H4	*	125	**49**	*
nxt.ok3:H4	*	358	**133**	*
aso.bug1:H4	*	128	**92**	*
aso.bug2:H4	*	147	**74**	*
aso.bug3:H4	*	209	**136**	*
aso.bug4:H4	*	329	**152**	*
aso.ok1:H4	*	270	**210**	*
aso.ok2:H4	*	*	**1312**	*
ctm.bug2	36	29	**21**	105
ctm.bug3	*	124	**59**	258
ctm.ok1	23	37	**21**	122
ctm.ok2	28	26	**17**	111
ctm.ok3	*	116	**53**	275
ctm.ok4	*	320	**143**	395

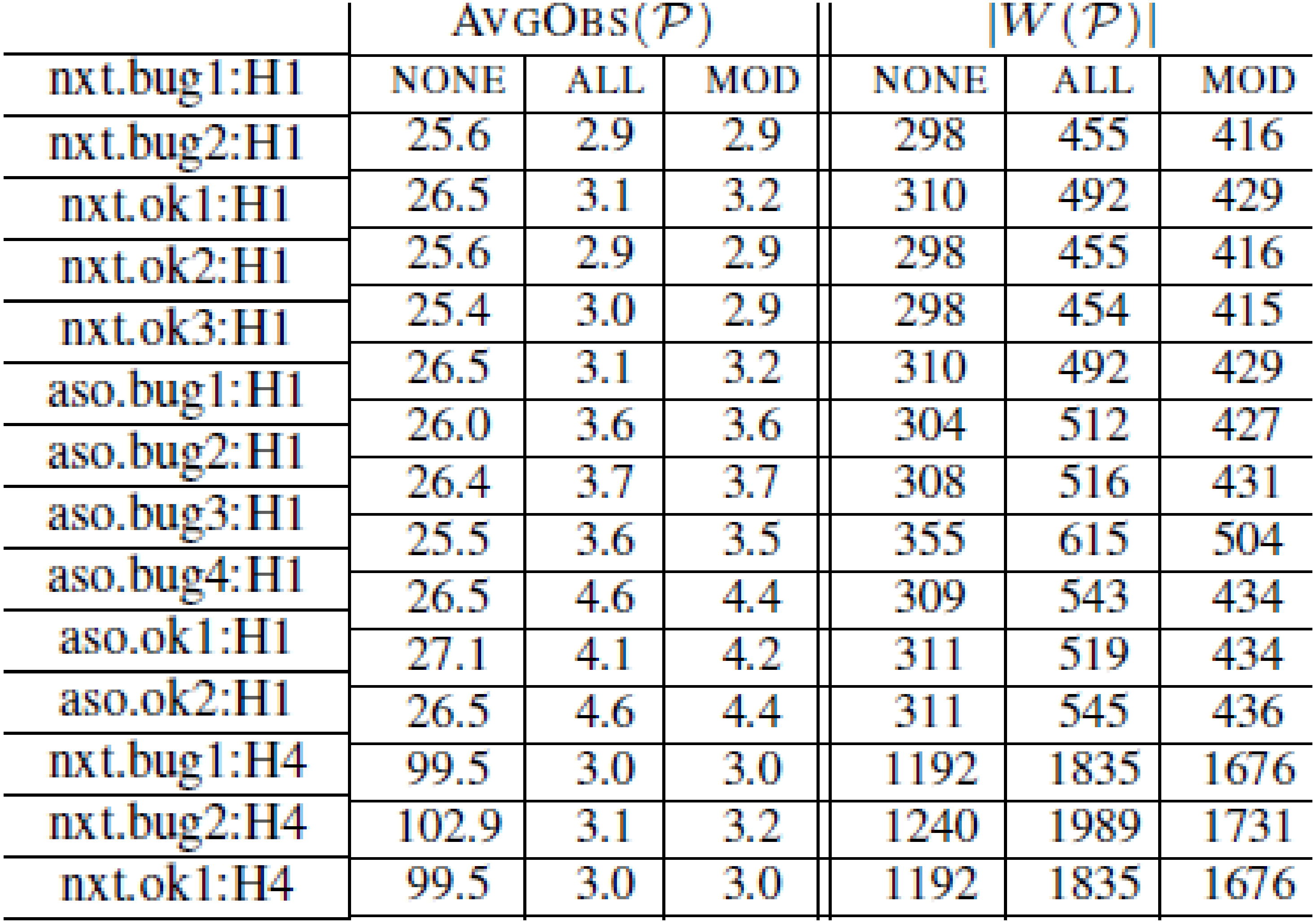
2GB Memory Limit

60min Time Limit

Solver=STP

NONE=No snapshotting, ALL=Snapshot all variables, MOD=Snapshot only modified variables, REKH=Previous tool based on sequentialization

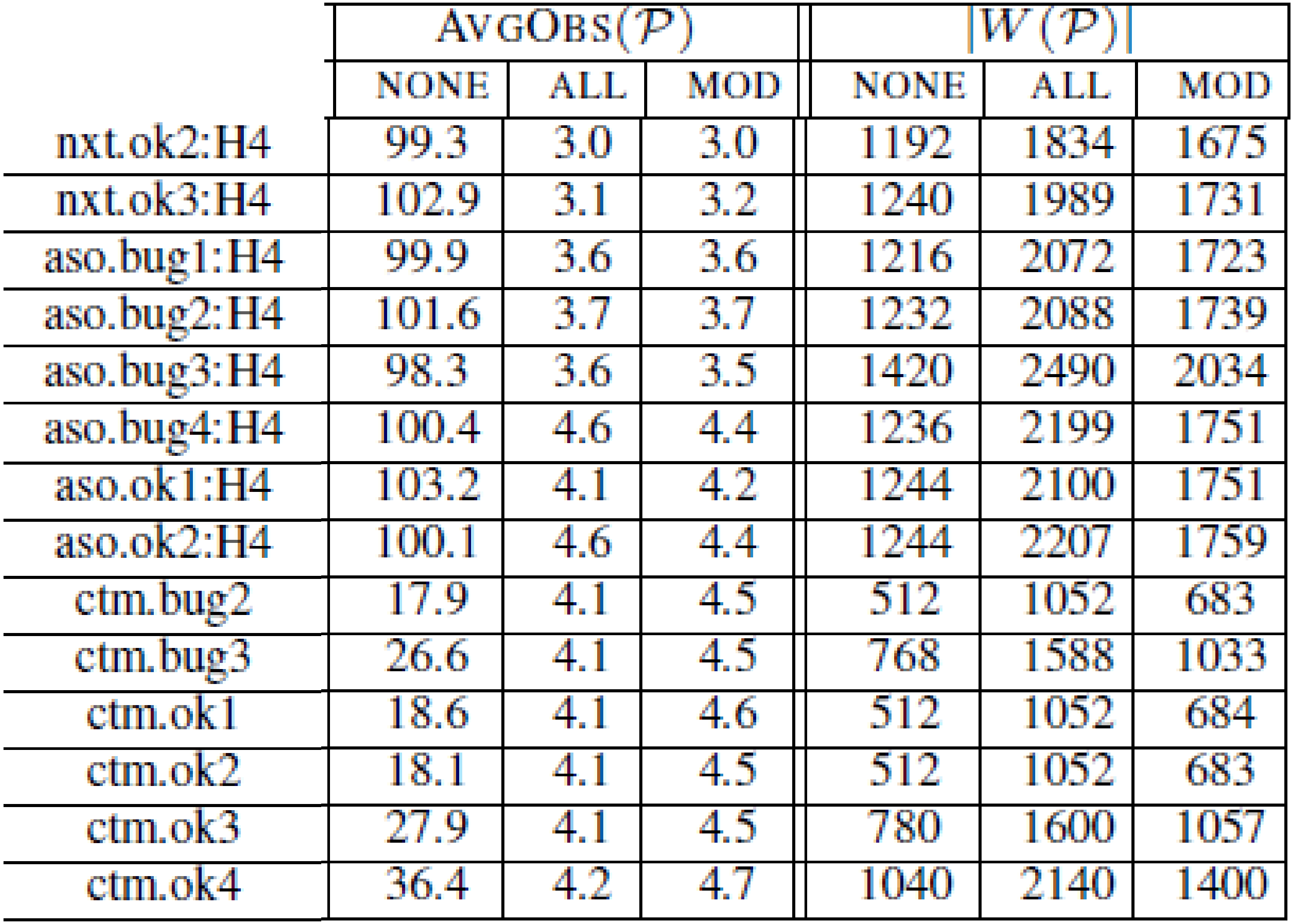
Observability Sizes



	$\text{AvgObs}(\mathcal{P})$			$\lvert W(\mathcal{P})\rvert$		
	NONE	ALL	MOD	NONE	ALL	MOD
nxt.bug1:H1	25.6	2.9	2.9	298	455	416
nxt.bug2:H1	26.5	3.1	3.2	310	492	429
nxt.ok1:H1	25.6	2.9	2.9	298	455	416
nxt.ok2:H1	25.4	3.0	2.9	298	454	415
nxt.ok3:H1	26.5	3.1	3.2	310	492	429
aso.bug1:H1	26.0	3.6	3.6	304	512	427
aso.bug2:H1	26.4	3.7	3.7	308	516	431
aso.bug3:H1	25.5	3.6	3.5	355	615	504
aso.bug4:H1	26.5	4.6	4.4	309	543	434
aso.ok1:H1	27.1	4.1	4.2	311	519	434
aso.ok2:H1	26.5	4.6	4.4	311	545	436
nxt.bug1:H4	99.5	3.0	3.0	1192	1835	1676
nxt.bug2:H4	102.9	3.1	3.2	1240	1989	1731
nxt.ok1:H4	99.5	3.0	3.0	1192	1835	1676

$AVGOBS(P)$ = avg. no. of reads observing each write or snapshot
$\lvert W(P)\rvert$ = total no. of snapshot and write variables

Observability Sizes



	AvgObs($\mathcal{P}$)			$\|W(\mathcal{P})\|$		
	NONE	ALL	MOD	NONE	ALL	MOD
nxt.ok2:H4	99.3	3.0	3.0	1192	1834	1675
nxt.ok3:H4	102.9	3.1	3.2	1240	1989	1731
aso.bug1:H4	99.9	3.6	3.6	1216	2072	1723
aso.bug2:H4	101.6	3.7	3.7	1232	2088	1739
aso.bug3:H4	98.3	3.6	3.5	1420	2490	2034
aso.bug4:H4	100.4	4.6	4.4	1236	2199	1751
aso.ok1:H4	103.2	4.1	4.2	1244	2100	1751
aso.ok2:H4	100.1	4.6	4.4	1244	2207	1759
ctm.bug2	17.9	4.1	4.5	512	1052	683
ctm.bug3	26.6	4.1	4.5	768	1588	1033
ctm.ok1	18.6	4.1	4.6	512	1052	684
ctm.ok2	18.1	4.1	4.5	512	1052	683
ctm.ok3	27.9	4.1	4.5	780	1600	1057
ctm.ok4	36.4	4.2	4.7	1040	2140	1400

$AVGOBS(P)$ = avg. no. of reads observing each write or snapshot

$|W(P)|$ = total no. of snapshot and write variables

Related Work and Concluding Thoughts

Generate Verification Condition by Encoding Dataflow between Reads and Writes Using Lamport Clocks

- Nishant Sinha, Chao Wang: Staged concurrent program analysis. SIGSOFT FSE 2010: 47-56
- Jade Alglave, Daniel Kroening, Michael Tautschnig: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. CAV 2013: 141-157

Generate Verification Condition per Scheduling round using prophecy variables, and ensure that output of one round equals input to the next

- Akash Lal, Thomas W. Reps: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. CAV 2008: 37-51

- **Snapshotting combines both ideas**
- **Interplay between Logical Clocks and Prophecy Variables**
    - **Both due to Lamport**
- **We encode both program variables and clocks as bit-vectors**
    - **Clocks can be encoded as integers, but then we have a mixed theory**

QUESTIONS?

Contact Information Slide Format

Sagar Chaki
Principal Researcher
SSD/CSC
Telephone: +1 412-268-1436
Email: chaki@sei.cmu.edu

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Web
www.sei.cmu.edu
www.sei.cmu.edu/contact.cfm

Customer Relations
Email: info@sei.cmu.edu

Telephone:	+1 412-268-5800
SEI Phone:	+1 412-268-5800
SEI Fax:	+1 412-268-6257