UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Corso di Tecniche automatiche per la correttezza del software

Verifica di programmi periodici

Presentato da: Sheykina Alexandra

Prof: Salvatore La Torre

Outline

1. Lavori analizzati
2. Proprietà safety-critical
3. Programmi periodici
4. Esperimenti
5. Conclusione
6. Riferimenti

Lavori analizzati

- *S. Chaki, A. Gurfinkel, and N. Sinha, “Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots,”*

- *S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, “Compositional Sequentialization of Periodic Programs,”*

- *S. Chaki, A. Gurfinkel, and O. Strichman, “Verifying Periodic Programs with Priority Inheritance Locks,”*

Autori

Sagar Chaki (CMU)

interested in specification, verification, and validation of software, with particular focus on concurrent software, real-time and cyber-physical systems, and software security.

Arie Gurfinkel (CMU)

primary focus areas are Automated Program Analysis, Software Model Checking, Automated Reasoning, and Abstract Interpretation

Ofer Strichman (IIT)

Formal verification of finite-state systems, model-checking and bounded model-checking. Decision procedures for first-order theories in the Satisfiability Modulo Theories (SMT) framework; SAT and CSP; Program equivalence checking

Safety-critical o life-critical

Sistemi safety -critical o life critical, ovvero tali che un fallimento può comportare danni in termini di ferimenti o perdita di vite umane (macchine per dialisi, sistemi di guida automatica per autoveicoli, treni o aerei, controllo delle centrali nucleari ecc.)

Esempi errori

1. Razzo Arian 5 esploso dopo 39 sec.
   - Errore nella programma di bordo durante la trasformazione di un reale di 64bit in un intero di 16bit. Danno >&600 mln
2. Intel pentium
   - Rilasciato un chip con un errore nel firmware. Sostituzione di milioni di processori difettosi. Danno > &500 mln
3. Therac-25
   - Dispositivo di radioterapia. Programma integratoper la gestione dei dispositivi consentiva il trasferimento in modalità con una dose molto elevata di radiazione. In alcuni (rari) lazioni del personale, i pazienti ricevevano sovra-dosaggio Due morti, diverse persone remasti invalidi.
4. Errore nel programma del missile Patriot abbattuto il suo Tornado
5. Robot Talon
6. Sonda Fobos-1 su Marte

Periodic Embedded Real-Time Software

Un PP è costituito da un insieme finito di compiti, ciascuno in esecuzione nel proprio thread

Task	*Period*
Engine control	10ms
Aibag	40ms
Bracking	40ms
Cruise Control	50ms
Collision Detection	50ms
Entertainment	80ms

Periodic Program (PP)

- Un programma periodico PP è un insieme finito di Task

$$\{\tau_1, \tau_2, \ldots, \tau_N\}$$

- Task τ_i è una tupla $(\pi_i, J_i, P_i, C_i, A_i)$

  π_i Priorità di esecuzione

  J_i Codice del task (con $J_{i,j}$ indichiamo il j-imo job dell' i-imo task)

  P_i Periodo

  C_i worst case execution time

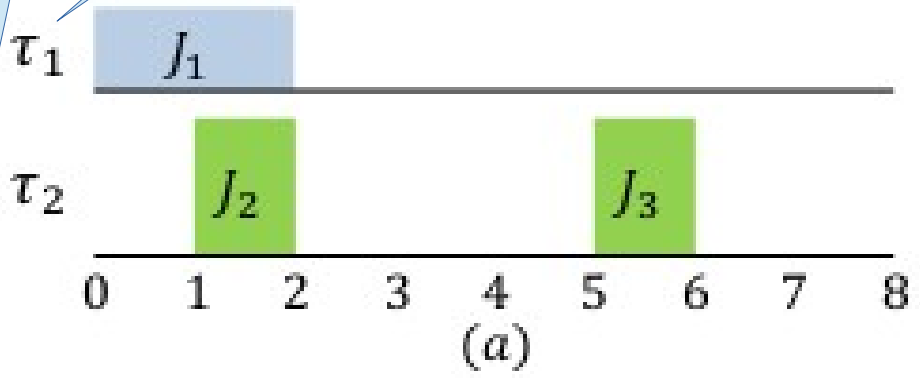
  A_i tempo d'arrivo

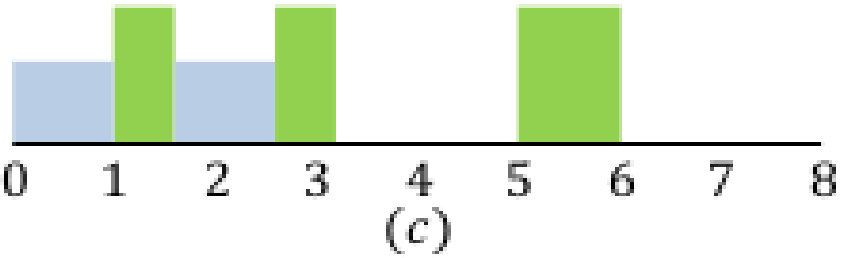
Esempio PP

Task con Alta priorità

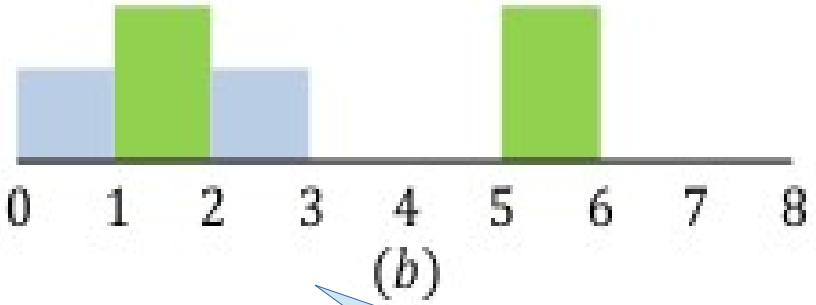
Task con Bassa priorità

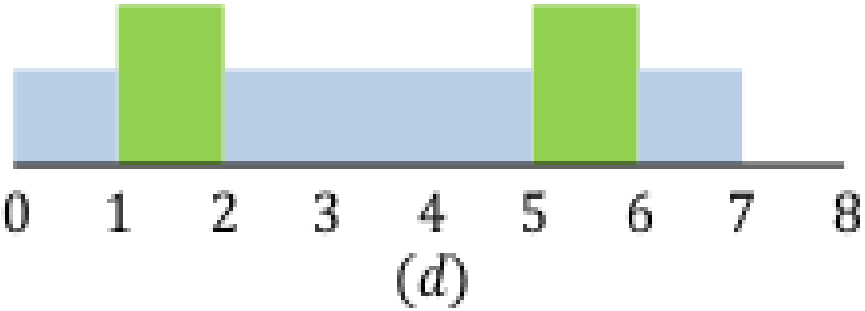
Esecuzione illegale





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





Esecuzione legale

Esecuzione illegale

Efficient Verification of Periodic Programs Using Sequential Consistency and Snapshots

- Contesto:
  - Programma periodico
  - Time-Bounded Verification
- Generazione della Verification Condition
  - L'orologio di Lamport
  - Locks
  - Snapshotting

Time-Bounded Verification

- Input (Periodic Program – collezione dei task periodici)
  - Eseguire jobs con prelazione basata sulle priorità
  - Le priorità rispetto RMS (Ogni processo deve essere completato entro il periodo di tempo stabilito;Non vi sono dipendenze tra processi;In ogni esecuzione i processi richiedono la sessa quantità' di tempo;I processi non periodici non hanno scadenze temporali;)
  - Comunicazione attraverso la memoria condivisa

- Problem
  - Proprietà A violata dopo X ms dallo stato iniziale I
  - A, X, I specificate dall'utente

- Solution
  - Generare condizione di verifica
  - Usare SMT Solver per risolvere soddisfacibilità

Memory Consistency

"Un multiprocessore si dice sequential consistent (SC), se l'esecuzione di un qualsiasi programma concorrente, produce lo stesso risultato che verrebbe prodotto da una qualche esecuzione sequenziale, e le operazioni in tale sequenza, avvengono nello stesso ordine dettato dal programma originario."

Laslie Lamport

- Lamport's (hierarchical) Clock

  Ogni evento (es. accesso ad una variabile condivisa) ha associato un timestamp

  Timestamp = interi simbolici che servono ad ordinare

  - Program order
  - Operazioni di read/write
  - Operazioni di prelazione

Verification Condition

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

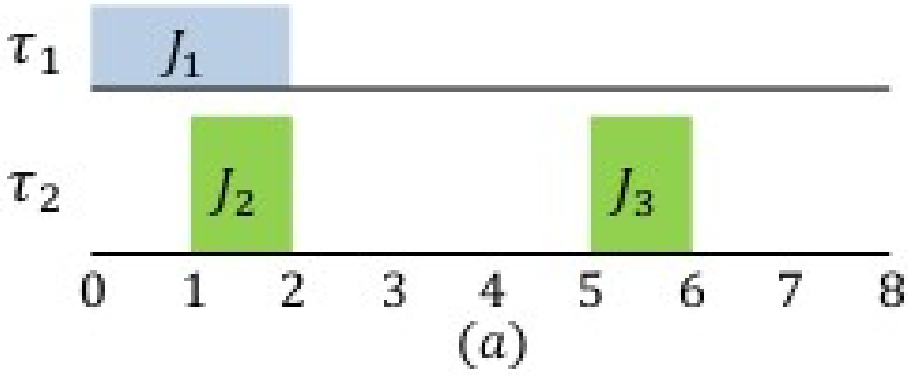
$$VC_{sec} = VC(J_1) \wedge VC(J_2) \wedge VC(J_3) \ldots \wedge VC(J_i)$$

VC_{seq} - Codifica calcolo Job-locale. Valore read /write da ogni accesso alla variabile condivisa rappresentato da una variabile nuova.

VC_{clk} - Associa ad ogni accesso della variabile condivisa l'orologio gerarchico di Lamport. I valori di Clock basate sui tempi e la priorità. Inoltre assicura il corretto susseguirsi dei round (timestamp) delle azioni.

VC_{obs} - Si collega valore letto ad ogni "read" per il valore scritto dai più recenti "write" secondo l'orologio Lamport.

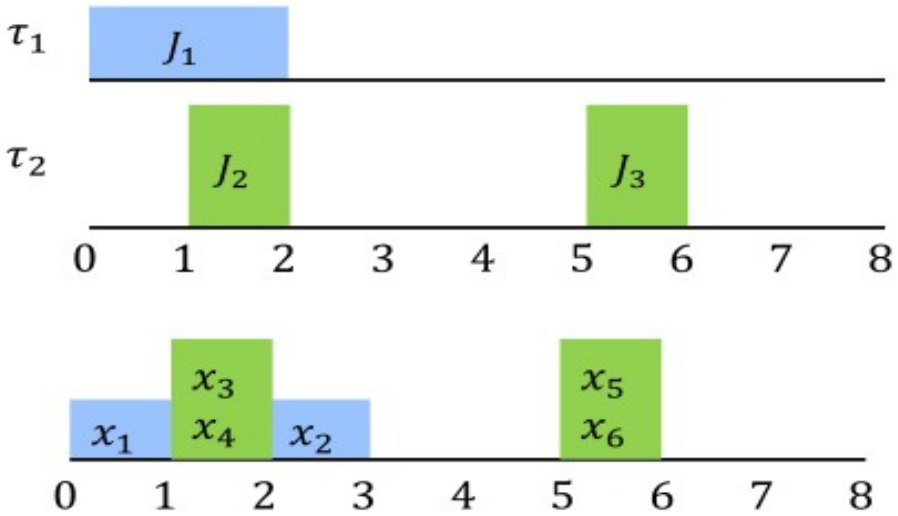
Verification Condition VC_{seq}



Stessa condizione di verifica per il programma sequenziale con eccezione che sia per read che per write si assegnano nuovi variabili

$$\left.\begin{array}{l} J_1()\,x := x+1;\rightarrow x_2 = x_1+1 \\ \wedge \\ J_2()\,x := x+1;\rightarrow x_4 = x_3+1 \\ \wedge \\ J_3()\,x := x+1;\rightarrow x_6 = x_5+1 \end{array}\right\} VC_{seq}$$

Verification Condition VC_{clk}



Osservazione: x_i è accessibile prima x_j se e solo se

$$(R_i, \pi_i, \iota_i) < (R_j, \pi_j, \iota_j)$$

dove < ordine lessicografico

Questo vale per tutti esecuzioni legali

Quindi: Associare ad ogni x_i orologio gerarchico di Lamport

$$\kappa_i = (R_i, \pi_i, \iota_i)$$

VC_{clk}

- π_i — Priorità d'esecuzione

$$\pi_1 = \pi_2 = 1, \pi_3 = \ldots = \pi_6 = 2$$

- R_i — # jobs finite prima di x_i

$$R_1 = R_3 = R_4 = 0, R_2 = 1, R_5 = R_6 = 2$$

- ι_i — Indice di istruzione di accesso in ordine topologico di CFG

$$\iota_1 = \iota_3 = \iota_5 = 1, \iota_2 = \iota_4 = \iota_6 = 2$$

Verification Condition VC_{obs}

J_i – Job in cui si accede alla variabile x_i

$J \sqsubset J'$ – se J completa sempre prima che inizia J'

$\kappa_i = (R_{\tau}, \pi_i, \iota_i)$

– Per ogni read x_i abbiamo

$$W_i = \{ x_j | x_j \text{ è una write} \wedge \neg (J_i \sqsubset J_j) \}$$

, l'insieme di tutti write che possono essere osservati

VC_{obs} Il valore di ogni x_i accessibili da una lettura è uguale al valore di x_j tale che

$$\kappa_j = max \{ \kappa_k | \kappa_k < \kappa_i \, e \, x_k \in W_i \}$$

, dove max{} è valore iniziale di *x*

Verification Condition VC_{obs}

- Per ogni read di x_i introduciamo $\tilde{\kappa}$ il clock della write action osservata

$$VC_{obs} \equiv \bigwedge_{x_j \in W_i} \kappa_j < \kappa_i \Rightarrow \kappa_j \leq \tilde{\kappa}_i \qquad (1)$$

$$\bigwedge \left(\left(VC^1_{obs} \right) \vee \left(\bigvee_{(x_j \in W_i)} VC^2_{obs}(j) \right) \right) \qquad (2)$$

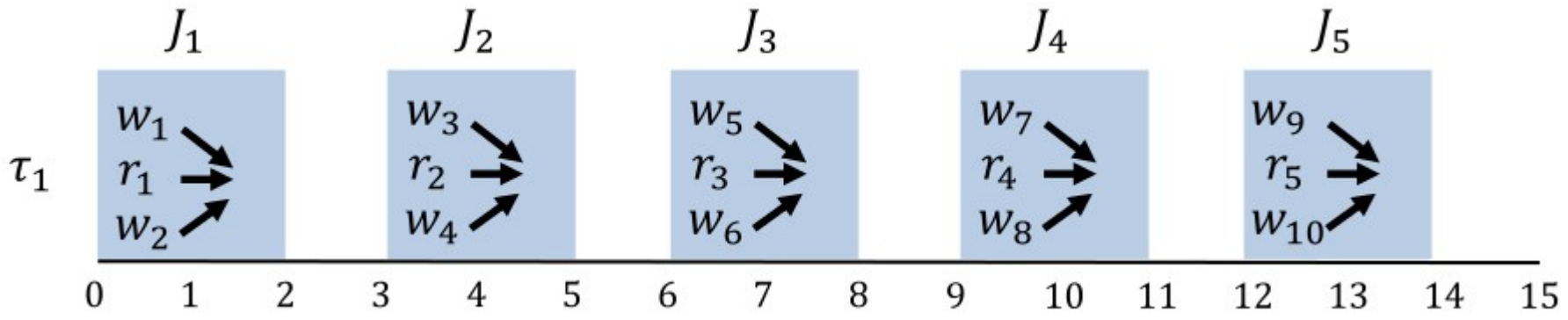
$$VC^1_{obs} \equiv \left(\bigvee_{(x_j \in W_i)} \kappa_j \geq \kappa_i \right) \wedge \left(x_i = x_{Init} \right) \quad VC^2_{obs}(j) \equiv \left(\kappa_j < \kappa_i \wedge \kappa_j = \tilde{\kappa}_i \right) \wedge x_i = x_j$$

(1) assicura che la write action osservata da ***i*** deve essere eseguita prima di ***j***

(2) assicura che ***j*** legge effettivamente il valore scritto dalla write action che osserva

Snapshotting:Problema

w_i = *write*, r_i = *read*



- *Osservazione:*

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

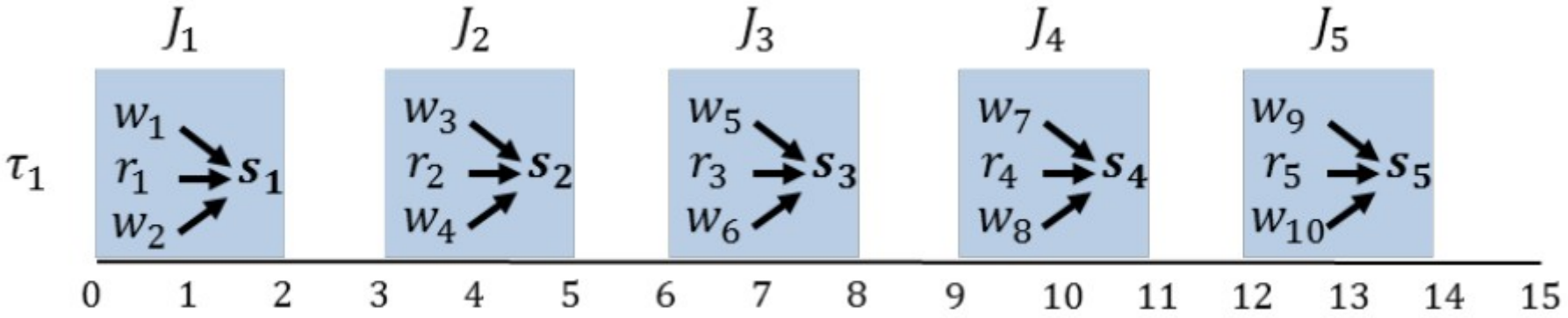
- *Risultato:*

  problema per $r_{(i-1)}$ ricodifica (e re-solver SMT) come la parte del problema r_i

- SMT solvers non scala all'aumentare numero di jobs.

Snapshotting:Soluzione

Fare uno snapshot significa fare, in maniera atomica, una read seguita da una write.



S_i = *snapsnot*

- *Osservazione:*

$$W(r_1)=W(s_1)=\{w_1, w_2\}, W(r_2)=W(s_2)=\{s_1, w_3, w_4\}, W(r_3)=W(s_3)=\{s_2, w_5, w_6\}, \ldots$$

- Snapshotting elimina gran parte di questa codifica e risoluzione ridondante
- SMT solver scala
- Scelta delle variabili di snapshot: (i) tutti variabili; (ii)solo scritti da job

Verification Condition VC_{obs} con Snapshotting

- *Input:* *Snaps(J) = insieme delle variabili osservati da* ***J***
- *Calcolare:* Relazione $J \uparrow J'$ se e solo se ***J*** può essere prelazionato da ***J'***, allora non può essere eseguito fin quando ***J'*** resta attivo

  $\Psi_{\sqsubset}(J, g)$ l'inseme massimale delle jobs di variabili osservati g, precedenti a ***J***, secondo ordine $\sqsubset$

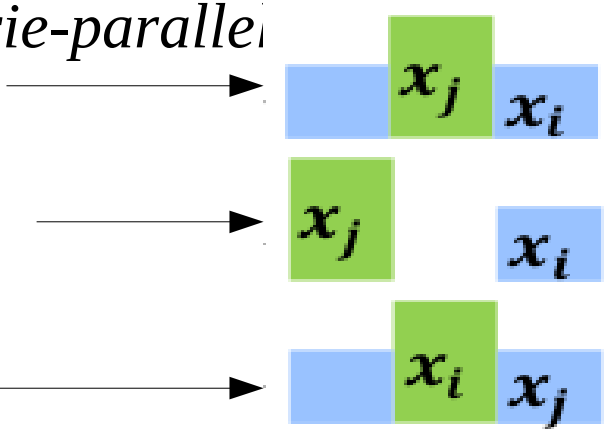
$$\Psi_{\uparrow}(J, g) = \{J' \mid J \uparrow J' \wedge g \in Snaps(J')\} \quad \Psi_{\downarrow}(J) = \{J' \mid J' = J \vee J' \uparrow J\}$$

- *Queste relazioni catturano la struttura serie-paralle*

$$W_i = \{x_j \mid x_j \text{ è un snapshot} \wedge J_j \in \Psi_{\uparrow}(J_i, g)\} \cup$$

$$\{x_j \mid x_j \text{ è un snapshot} \wedge J_j \in \Psi_{\sqsubset}(J_i, g)\} \cup$$

$$\{x_j \mid x_j \text{ è un write} \wedge J_j \in \Psi_{\downarrow}(J_i, g)\}$$



Handling locks

PCP lock (Priority ceiling protocol)

- Ogni thread ha una priorità base = priorità del task che viene eseguito
- Ad ogni lock PCP l è associato la priorità $\pi(l)$
- la priorità del thread = max (la sua priorità di base, le priorità di tutte le serrature PCP in suo possesso)
- scheduling disabilitato per tutti i task con priorità inferiore a quella del task che ha acquisito il lock

CPU lock

- scheduling completamente disabilitato, resta in esecuzione solo il task che lo ha acquisito (caso specifico di un PCP lock)
- Un CPU lock è un PCP lock tale che $\pi(l) = \infty$

Handling locks

- Operazioni di PCP-lock codificati con: *priority test and set actions* PTAS = dove: (J, pc, π_t, L_a, L_r)
- J è un job
- pc ∈ Z è una locazione
- Tutti lock devono avere la priorità inferiore a π_t
- L_a , L_r i lock acquisiti/rilasciati
- PTAS codificate nella VC_{clk} e VC_{obs} per assicurare che:

  1. test: i lock attivi hanno priorità maggiore di π
  2. set: i lock in L_a vengono acquisiti, quelli in L_r rilasciati

Resultati (Time in secondi)

- 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

	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**	*

Resultati (Time in secondi)

	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

Taglia Osservabilità

- AVGOBS(P)
  - avg. no. of reads observing each write or snapshot
- |W(P)|
  - total no. of snapshot and write variables

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

Taglia Osservabilità

	AvgObs($\mathcal{P}$)			$\lvert W(\mathcal{P})\rvert$		
	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

Compositional Sequentialization of Periodic Programs

- *Obiettivo*:

  Migliorare la metodologia di sequenzializzazione

- *Idea:*

  Sfruttare la separazione temporale dei job

  Generare esecuzioni non spurie

  Abbattere ulteriormente i tempi di verifica

  Mostrare l’impatto di tale metodologia sui programmi armonici

- *Soluzione:*

  Usare come time bound un hyperperiod (hiperperiod = mcm(di tutti periodi))

  Derivare da ciò un upper bound al numero di job che ogni task può eseguire

  Sequenzializzare il risultante job bounded programma

  Usare CBMC per la verifica

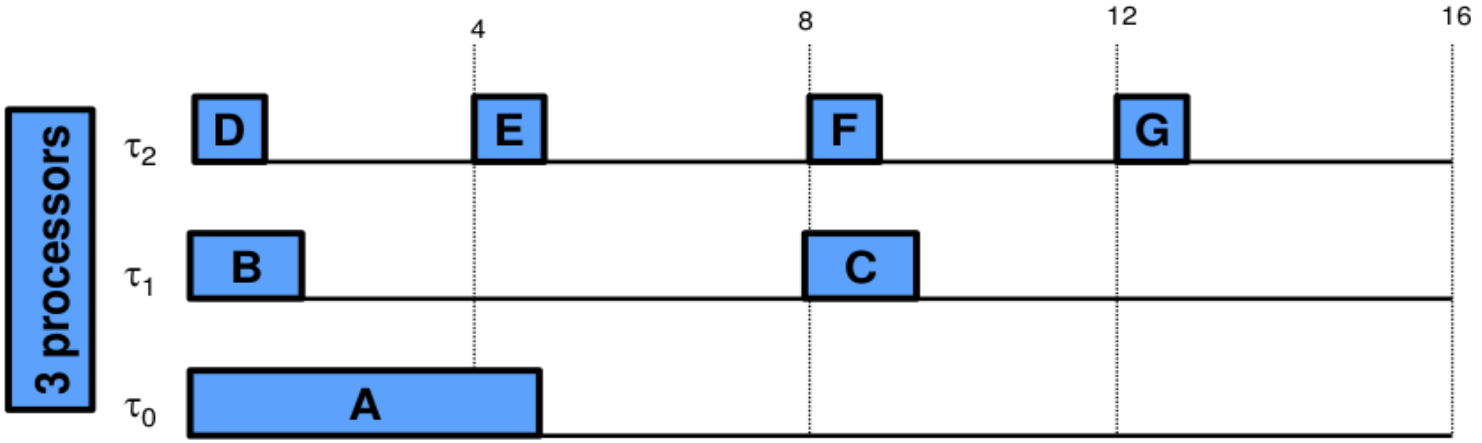
  Il nuovo approccio viene chiamato CompSeq, quello precedente MonoSeq

Harmonic PP

- Un programma periodico PP = $\{\tau_1, \tau_2, \ldots, \tau_N\}$ si dice armonico se per ogni coppia (τ_i, τ_j) abbiamo che

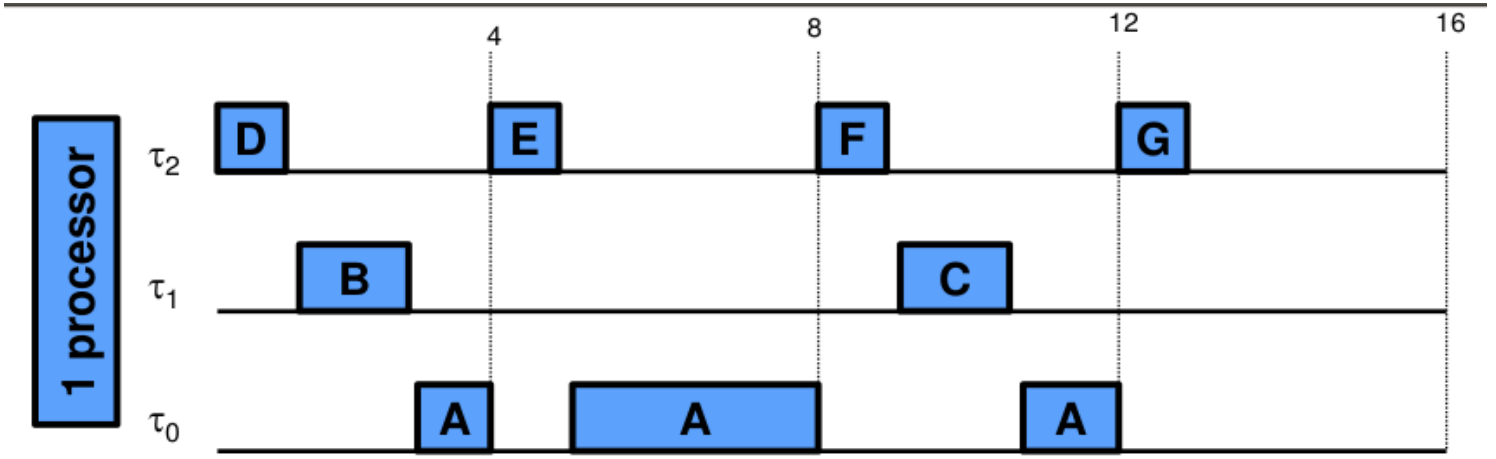
$$\text{mcm}(P_i, P_j) = \text{MAX}\{P_i, P_j\}$$

Ad esempio $P_1 = 2, P_2 = 4, P_3 = 8, P_4 = 16, \ldots P_k = 2k$



Task	WCET (C_i)	Period (P_i)	Arrival Time (A_i)
τ_2	1	4	0
τ_1	2	8	0
τ_0	5	16	0

Harmonic PP



Task	WCET (C_i)	Period (P_i)	Arrival Time (A_i)
τ_2	1	4	0
τ_1	2	8	0
τ_0	5	16	0

Perché avere programmi periodici che siano armonici?

- Migliore sfruttamento della CPU
- Migliore utilizzo della batteria su sistemi come reti di sensori
- Numero di round necessari per la verifica esponenzialmente inferiore

Hiperperiod

- Time bound usato è l’ hyperperiod ossia il mimino comune multiplo dei periodi di tutti i thread del programma, e la separazione sarà di tipo ***intra-period*** o ***inter-period***
- ***Intra-Hyper-Period***
  - Tra i jobs all'interno della stessa ***hyperperiod***
  - Previene alcuni posti di lavoro nello stesso iper-periodo dal interleaving in base alle loro priorità, tempi di arrivo e tempi di esecuzione nel caso peggiore
- ***Inter-Hyper-Period***
  - Tra i jobs in diversi ***hyperperiod***
  - Impedisce interleaving tra i jobs di diversi ***hyperperiod***
  - Per ipotesi A2, tutti i jobs in hyperperiod ***i*** completo prima che qualsiasi job in iper-periodo (***i + 1***) si avvia.

Job Ordering

$$J_1 \rhd J_2 \equiv (\pi(J_1) \leq \pi(J_2) \wedge D(J_1) \leq D(J_2)) \vee (\pi(J_1) > \pi(J_2) \wedge A(J_1) \leq A(J_2))$$

J_1 *termina sempre prima dell'inizio di* J_2

$$J_1 \uparrow J_2 \equiv (\pi(J_1) < \pi(J_2) \wedge A(J_1) < A(J_2) < D(J_1))$$

$$J_1 \sqsubset J_2 \equiv (J_1 \rhd J_2) \vee (J_1 \uparrow J_2)$$

OR logico delle precedenti

Verifying Periodic Programs with Priority Inheritance Locks

Obiettivo

- Prendere un programma periodico C
- Convertire C in un programma sequenziale S
- Verificare che S sia safety e deadlock-free con uso di CBMC

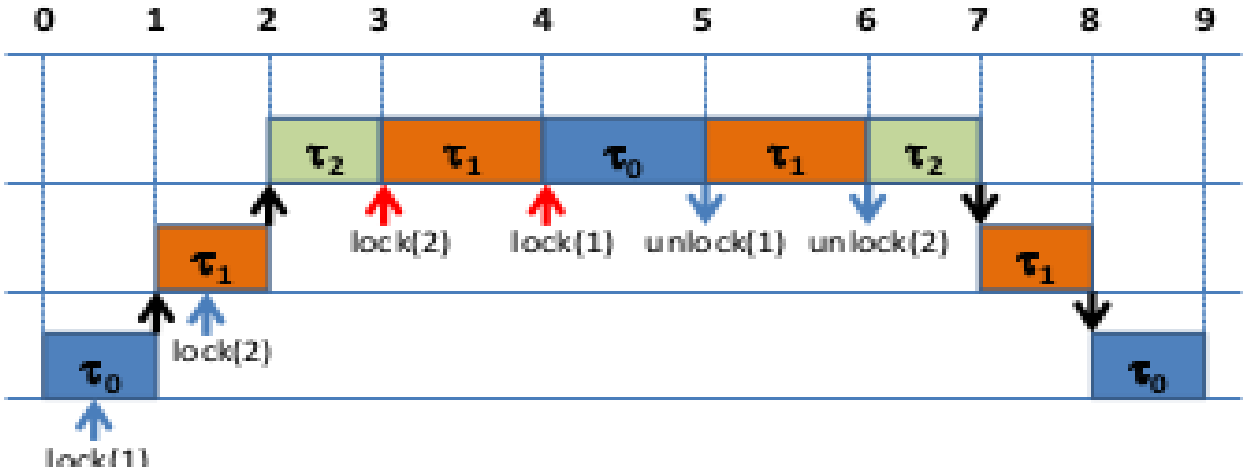
Come convertire C in un programma sequenziale S?

- Suddividere l’esecuzione in un numero fissato di round
- Assegnare ad ogni job starting ed ending round
- Eseguire i job in ordine di priorità
- Simulare la prelazione tramite passaggio (non deterministico) ad altro round
- Verificare che vincoli ed asserzioni siano state rispettate

Sequenzializzazione

- l’esecuzione viene suddivisa in round
- un round inizia al tempo 0
- un altro round inizia quando quello precedente termina, oppure un job viene bloccato nel tentativo di acquisire un lock

Task	I_i	A_i	C_i	P_i
τ_2	2	2	2	10
τ_1	1	1	4	20
τ_0	0	0	3	40



Verificare l’assenza di deadlock

Verificare l’assenza di deadlock: il task-resource graph TRG:

- Il TRG è un grafo bipartito i cui vertici sono partizionati in 2 sottoinsiemi:
- T: insieme dei task
- R: insieme delle risorse (o variabili condivise)
- Un arco da un vertice di T ad uno di R indica richiesta pendente di assegnazione
- Un arco da un vertice di R ad uno di T indica risorsa assegnata
- La presenza di deadlock si tradurrà in una ciclicità su tale grafo

- Node=task/lock; Edge = blocking/ownership; Cycle = deadlock;
- Chiusura transitiva di TRG mantenuta e aggiornata in modo dinamico
- Programma si interrompe se in TRG si trova un ciclico (cioè, chiusura transitiva ha auto-loop)

Verificare la safety

Verificare la safety - l’algoritmo iterativo PipVerif

- Mantiene un numero di round (inizialmente pari al numero dei job)
- Cerca un controesempio (esecuzione che viola la safety) entro gli R round
- Se lo trova, restituisce UNSAFE
- Se non lo trova incrementa i numeri di round entro cui cercare ed itera
- Se una tale esecuzione non esiste, restituisce SAFE

Algoritmo PipVerif()

function PipVerif(C)

R:= |J|

loop

x:= VerifRounds(C, R)

if x = INCROUNDS then R := R+1

else return x

function VerifRounds(C, R)

if [[S a (C, R)]] ≠ ∅ then return UNSAFE

if [[S b (C, R)]] ≠ ∅ then return INCROUNDS

then return SAFE

S a (C, R) ha il compito di trovare una esecuzione non legale entro R round.

Esperimenti

File	T	J	Rn	Vars	Cls	SAT	Result
nxt.bug1a.c	29	15	15	1.4M	4.3M	26	UNSAFE
nxt.bug1b.c	58	15	15	2.5M	7.5M	54	UNSAFE
nxt.bug1c.c	61	15	15	2.6M	8.1M	57	UNSAFE
nxt.ok1.c	746	15	17	2.9M	9.0M	714	SAFE
aso.bug1a.c	73	15	15	2.7M	8.3M	68	UNSAFE
aso.bug1b.c	64	15	15	2.6M	8.0M	59	UNSAFE
aso.bug1c.c	33	15	15	1.7M	5.1M	29	UNSAFE
aso.ok1.c	4148	15	19	3.5M	10.9M	4,088	SAFE
aso.bug2a.c	43	15	15	1.6M	4.9M	39	UNSAFE
aso.bug3a.c	48	15	15	1.7M	5.1M	45	UNSAFE
aso.bug3b.c	35	15	15	1.5M	4.6M	32	UNSAFE
aso.bug3c.c	55	15	15	1.6M	4.9M	52	UNSAFE
aso.ok3.c	879	15	16	1.8M	5.5M	866	SAFE
aso.bug4a.c	63	15	15	2.0M	6.1M	58	UNSAFE
aso.bug4b.c	908	15	16	2.1M	6.4M	898	UNSAFE
aso.ok4.c	3047	15	17	2.2M	6.7M	3,027	SAFE

Conclusioni

- Un metodo di verifica dei programmi periodici basato sulIl nostro algoritmo è basato su sequenzializzazione - riducendo la verifica di un programma concorrente a quella di verificare un equivalente (non deterministico) programma sequenziale., con timebound definito su hyperperiod, verificato tramite CBMC
- Presntato un algoritmo iterativo per verificare la sicurezza e assenza deadlock in programmi periodici. Si estende lavoro precedente in questa zona gestendo la sincronizzazione via Inheritance priorità Protocol (PIP) serrature, migliorando la scalabilità
- Un metodo di verifica dei programmi periodici basato su paradigma BMC-MC, composto di due fasi: (i) generare una Verification Condition VC; (Ii) risolvere il VC utilizzando un solver SMT. Generazione il VC, addattando “memory consistency” di Lamport alla semantica di PP. A migliorare la scalabilità, sviluppata una strategia chiamata snapshot, che pota a generare VC con meno ridondanti sotto-formule.

Lavori analizzati

- *S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, “Compositional Sequentialization of Periodic Programs,”*

- *S. Chaki, A. Gurfinkel, and O. Strichman, “Verifying Periodic Programs with Priority Inheritance Locks,”*

- *S. Chaki, A. Gurfinkel, and N. Sinha, “Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots,”*