A New Method for MHP Analysis for Dynamic Barrier Languages

Saurabh Joshi, IIT Kanpur
R K Shyamasundar,
Tata Institute of Fundamental Research
Sanjeev K Aggarwal, IIT Kanpur

Problem

- Given a concurrent program *P*, *mhp(S,S')==true*, two statements *S* and *S'* (or instances thereof) may execute in parallel in some execution of program *P*
  - A statement inside a loop gives rise to multiple instances
- From now on it is denoted as *S||S'* and *!(S||S')*
- More helpful in optimization and debugging if we can establish *!(S||S')*

General MHP Analysis

- NP-complete under certain restrictions (Taylor, Acta Informatica 1983)
- Un-decidable if you take terminations
- Important for Compiler Optimizations and checking interference freedom in concurrent fragments

Dynamic Barrier Languages

- *async S* : Spawns a new activity which will execute *S*
- *finish S* : When all the activities transitively spawned from *S* finishes, only then this statement is said to finish its execution
- MHP for Dynamic Barrier Languages
  - X10 (PGAS), Habenaro
  - Shivali Agrawal, Raj Kishore Barik, Vivek Sarkar, RK Shyamasundar, PPoPP 2007

Dynamic Barrier Languages

- Clocks : A dynamic barrier construct, where activities can join and leave the barrier dynamically.
  - Activities participating in a barrier synchronization is not known at the time of barrier creation

Clocks

- *c=new clock()* : new clock is created and the current activity is registered with the clock

- *next* : barrier synchronization
  - Control does not proceed beyond this point, until all activities sharing a clock with this activity reaches corresponding synchronization point

Clocks

- *c.drop()* : activity de-registers itself from clock *c*
  - Once an activity drop out of the clock, it can not re-register again on that clock
- *async (c1,c2,…cn) S* : spawns an activity registered on clocks c1,c2,…,cn
  - This is the only other way to register with a clock apart from creating a clock
  - Parent activity has to be registered on c1,…,cn
  - A clocked *async* can not be in the immediate scope of *finish*

MHP Analysis Extensions

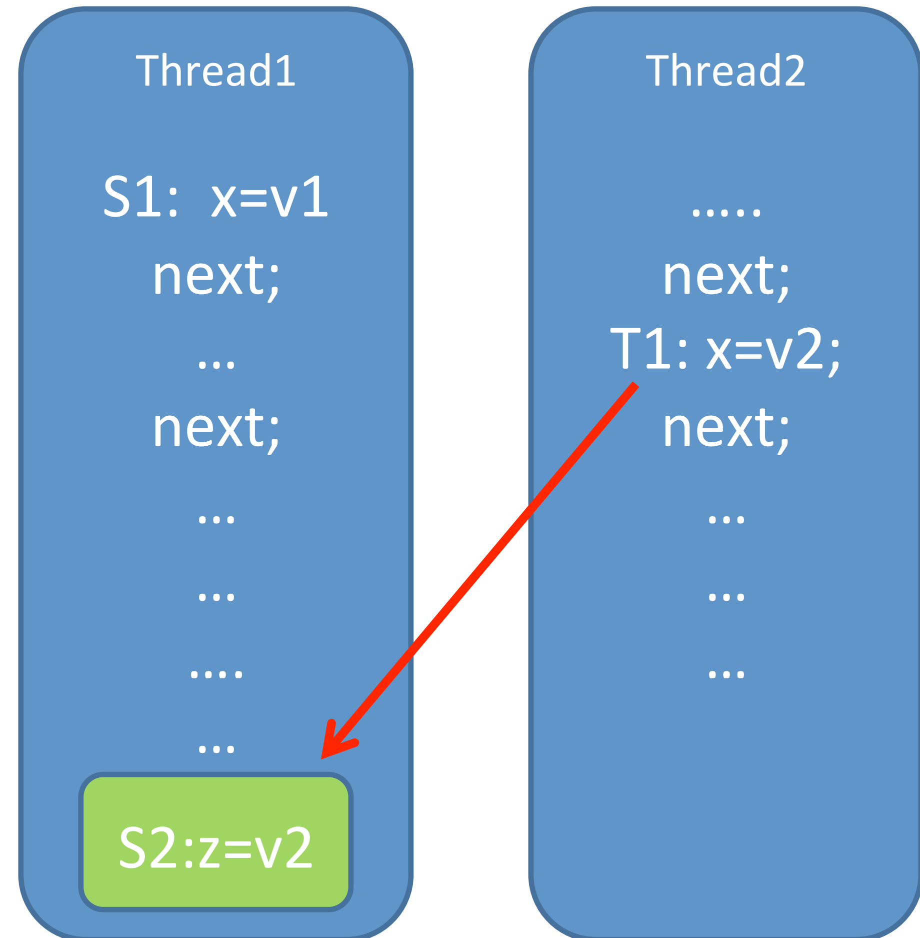
- Harshit Shah, RK Shyamasundar, Pradeep Varma, IPDPS 2009
- Realized in the context of CSSA (concurrent single static assignment of barrier synchronization languages) by constructing hierarchical clock control graph.

Why a new method?

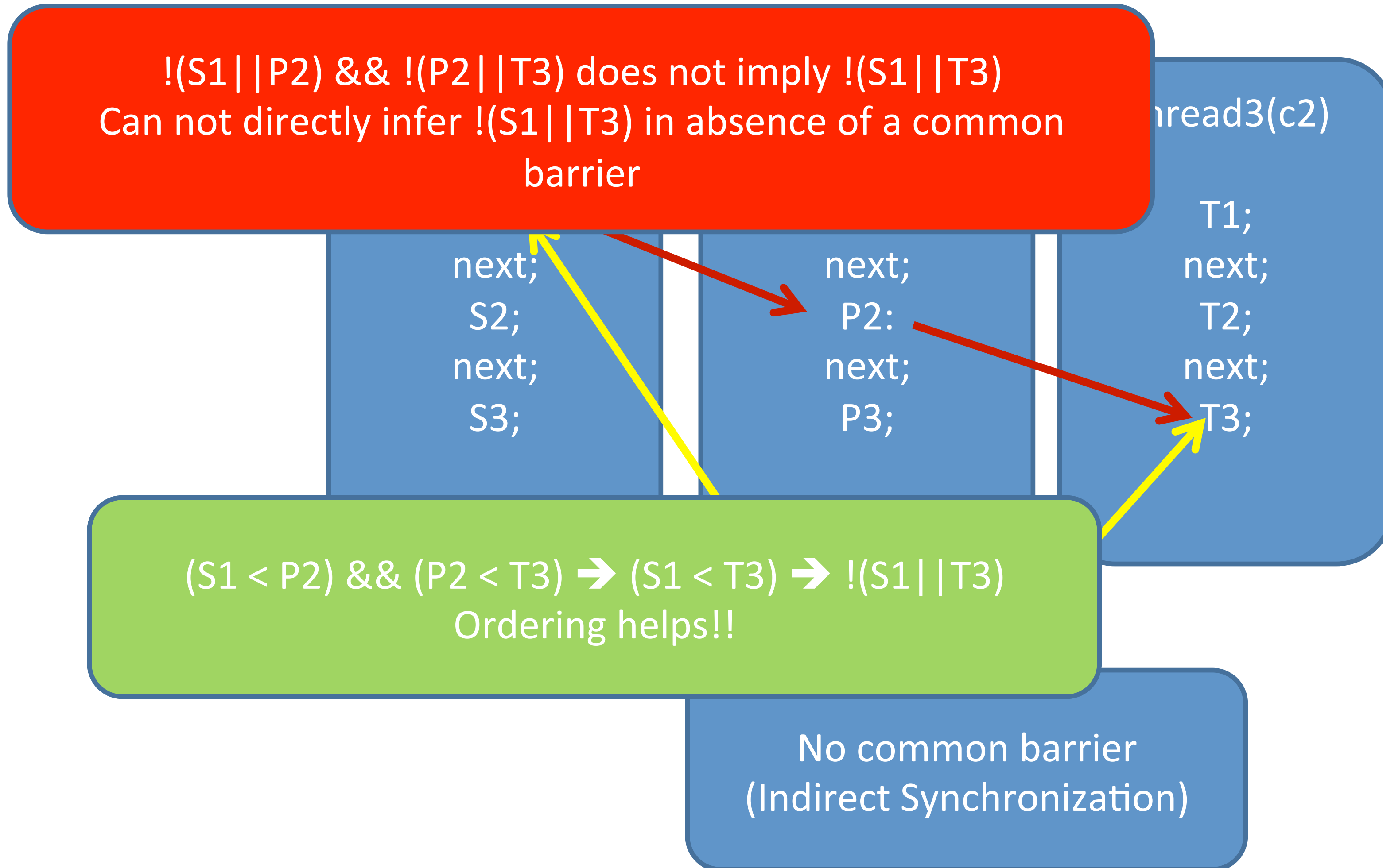
- Earlier approaches directly compute *parallel (n)* for every node *n* , where *parallel(n)* is the set of nodes that may execute in parallel to node *n*
  - It does not retain/compute order between two statements
  - Misses out on opportunities
  - Requires that two statements share a barrier to infer *mhp(S,S')==false*

Example 1

- Set based MHP analysis only infer *! (S1||T1)*
- Misses on optimization
- One can do better if the order can be inferred *(S1 < T1), (T1 < S2)*



Example 2



PIA: only approach to deal with indirect synchronization

Solution?

- What we need is a mechanism to retain or compute order between two statements

Phase Interval Analysis

- Compute $\langle a,b \rangle_c$ for each statement *S* with respect to some reference point *R*
  - Barrier : *c*
  - Lower bound : *a*
  - Upper bound : *b*
- S can only execute from phase *a* to *b* of barrier *c* when *R* is assumed to execute in $\langle 0,0 \rangle$

Phase Interval Analysis

<a,b> s1<a1,b1> <a1,b1> s2 <a2,b2>
-- (SEQ)
<a,b>s1;s2 <a2,b2>

<a1,b1> s1<a2,b2> <a1,b1> s2 <a3,b3>
-- (ITE)
<a,b>if b then s1 else s2 <min(a2,a3),max(b2,b3)>

Here, implicitly, clocks are assumed as a subscript for every interval.
One has to calculate intervals with respect to all clocks.

$$\frac{<a,b>\ \ s\ <a1,b1>}{<a,b>async\ s\ <a,b>}\ (ASYNC)$$

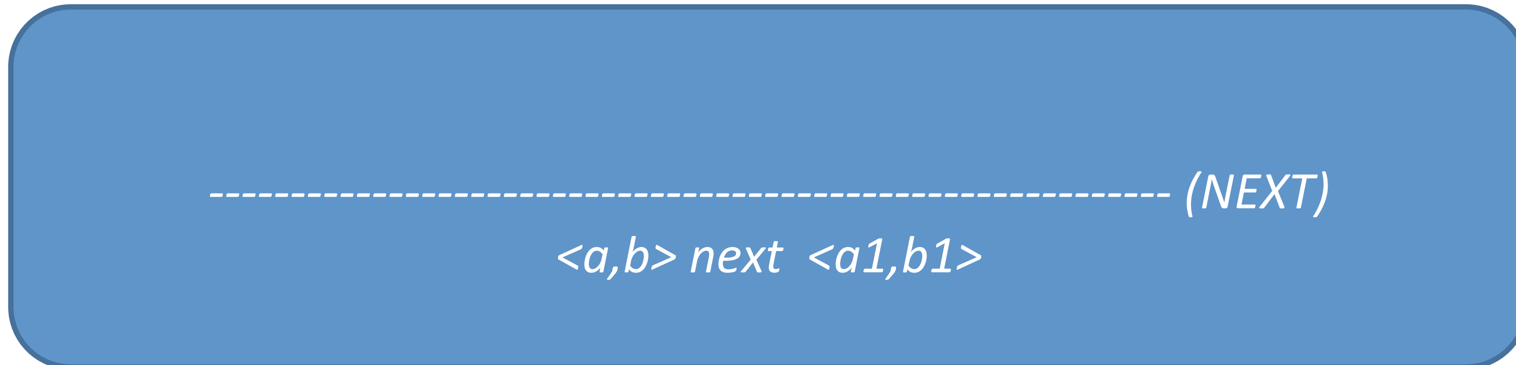
- *async* starts a new control flow, hence the statement following it starts in the interval *<a,b>*

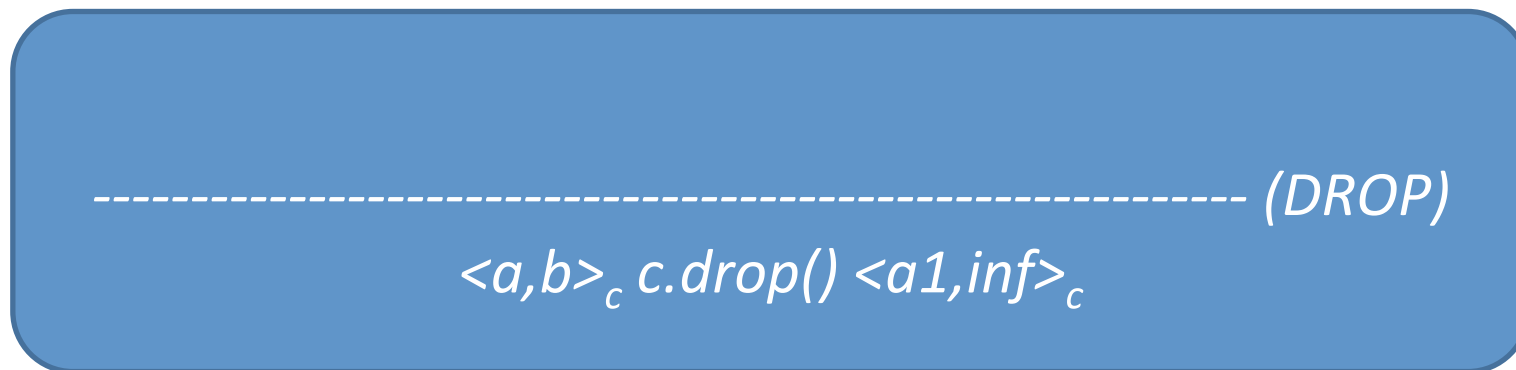
<a,b> s <a1,b1>

-- (FINISH)

<a,b>finish s <a1,b1>

- Follows from the rule for async
  - Note : a clocked *async* can not be in the immediate scope of *finish*





Once the clock is dropped, the current activity can execute in any phase of that clock, hence the upper bound is set to infinity

$$\frac{\langle a,b\rangle \; s \; \langle a1,b1\rangle \qquad loop\ s}{\langle (a1-a)i + a,(b1-b)i+b\rangle \; s^{i} \; \langle (a1-a)(i+1) +a, (b1-b)(i+1)+b\rangle} \; (LOOP\text{-}INST)$$

- For *i*-th instance of a statement, phase interval is computed by assuming that the body of the loop has executed *i* times

<a,b> s <a1,b1>

-- (LOOP-GEN)

<a,b> while b s <a,inf>

- When it is not possible to compute the iteration count of a loop in a static manner, set the lower bound after the loop to *a* (to account for *0* iterations) and upper bound to *infinity* (to account for unbounded iterations)

Following the widening technique by cousot and cousot., as we do not know how many times a loop executes, ,<a,inf> is a safe bound

<a,b> s <a1,b1>

-- (LOOP-INST)

<a,b> do N times s <(a1-a)N +a, (b1-b)N+b>

- When it is possible to compute a loop count *N* statically, phase interval can be computed by assuming that body executes as shown above

"Do N times" is representative of all loops for which loop count N can be computed upfront

From intervals to order

$$\frac{\langle a,b\rangle \;\; s1 \; \langle a1,b1\rangle \; , \; \langle a2,b2\rangle \; s2 \; \langle a3,b3\rangle \; , \; b1<a2}{s1 < s2}$$

- When for all clocks, upperbound of *s1* is smaller than lowerbound of *s2*, we can infer *s1* < *s2*
  - In general, bounds may be functions of iteration vectors, hence the order is conditional to the condition function being true

- To compute an order, we do not need actual phase interval, relative phase interval w.r.t. a common reference point *R* is sufficient
- Lease common ancestor between two statements is a good choice for *R*

Context Sensitivity

- For a method *foo()*, phase intervals can upfront be calculated :
  - *<0,0> foo() <a,b>*
- In any calling context, which starts at <c,d>, new intervals would be :
  - *<c,d> foo() <a+c, b+d>*

- - We can calculate phase intervals for a method in isolation assuming <0,0> at the beginning of the method .
-- Then in any calling context starting at <c,d>, for the entire body of the method,
we just need to add <c,d>, hence we can save a lot of time by not having to analyze a method all over again for every context

Condition function and optimization

- if the condition function for *s1<s2* or *!(s1||s2)* can not be calculated statically, it can be still helpful for optimization

```
If(CondFunc(I,J))
        Optimized Code
Else
        Unoptimized code
```

I and J are iteration vectors for statements S1[I] and S2[J]

Complexity

- *$O(n^2 rh)$*
  - *n* : number of nodes
  - *r* : number of clocks (practically extremely small)
  - *h* : height of the program structure tree (here, maximum nesting depth of loops)
- Does not subsume technique by Harshit et al (2009) (complexity *$O(n^4)$*)

Conclusion

- A new perspective on MHP analysis
- Integrated with Hierarchical Control Flow Graph is powerful for barrier synchronization languages
- Use both approaches (PIA based as well as set based proposed by Harshit et. Al.,) for best results
  - Other applications for correctness

Thank You