Identifying Use-After-Free Variables in Fire-and-Forget Tasks

Jyothi Krishna V S & Vassily Litvinov
jkrishna@cse.iitm.ac.in

IIT Madras & Cray Inc.

June 2, 2017

`begin` construct in Chapel

- Creates a dynamic task with an unstructured lifetime.
- *Fire-and-forget*
- Low synchronization and scheduling cost.

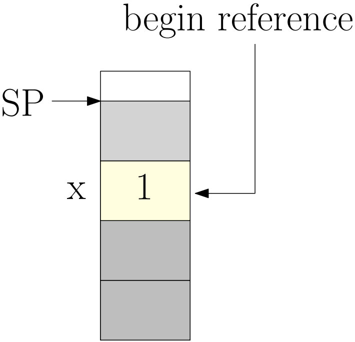
`begin` construct in Chapel

- Creates a dynamic task with an unstructured lifetime.
- *Fire-and-forget*
- Low synchronization and scheduling cost.

```
...
begin write("hello ");
write("world ");
...
```

```
Either outputs:
hello world
world hello
```

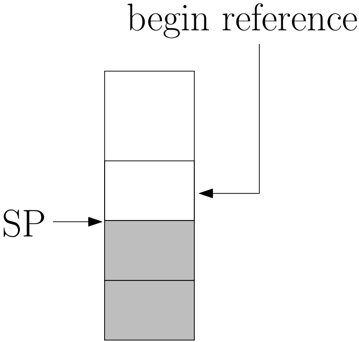
Use-After-Free Variables



```
{
 var x = 1;
 begin (ref x)
  {
   if x == 0 then
    writeln("chaos");
  }
}
...
{
  var y = 0;
}
```

`begin` task has a reference to variable x (outer variable).

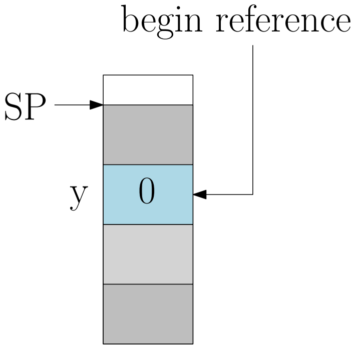
Use-After-Free Variables



```
{
 var x = 1;
 begin (ref x)
  {
   if x == 0 then
    writeln("chaos");
  }
}
...
{
  var y  = 0;
}
```

End of scope of variable x and is removed from the Stack.

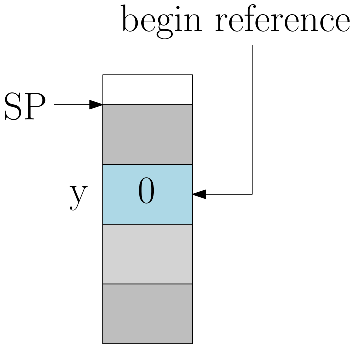
Use-After-Free Variables



```
{
 var x = 1;
 begin (ref x)
  {
   if x == 0 then
    writeln("chaos");
  }
}
...
{
  var y = 0;
}
```

New variable y added.

Use-After-Free Variables



```
{
 var x = 1;
 begin (ref x)
  {
   if x == 0 then
    writeln("chaos");
  }
}
....
{
  var y = 0;
}
```

Incorrect value of x seen by `begin` task. We need to avoid these in our programs.

Use-After-Free Access: Sources

- Lack of synchronization.
  - Programs written for older versions of Chapel.
- Improper synchronization.
  - Programmer skills/ Programming speed / Complexity of the Program.

[1] image source:shuttershock.com

Use-After-Free Access: Sources

- Lack of synchronization.
  - Programs written for older versions of Chapel.
- Improper synchronization.
  - Programmer skills/ Programming speed / Complexity of the Program.

[1]

[1] image source:shuttershock.com

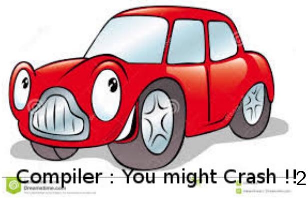
Talk Overview

- Extract relevant constructs and outer variables access into CCFG.
- Subset Representation of execution time states: Parallel Program States (PPS).
- Identify possible Use-After-Free variable accesses.
- Results and Conclusions

[2] image source:dreamstime.com

Talk Overview

- Extract relevant constructs and outer variables access into CCFG.
- Subset Representation of execution time states: Parallel Program States (PPS).
- Identify possible Use-After-Free variable accesses.
- Results and Conclusions



[2]

[2] image source:dreamstime.com

Synchronization Constructs in Chapel

- `sync` variable: One-to-one synchronization
- `single` variable: One-to-many synchronization
- `sync` block: Many-to-one synchronization.
- atomic variables.

```
1   proc outerVarUse( ) {
2    var x: int = 10;
3    var doneA$: sync bool;
4    begin with (ref x) { // A
5     writeln(x);
6     var doneB$:  sync bool;
7     begin with (ref x){ // B
8       writeln(x);
9       doneB$ = true;
10    }
11    writeln(x);
12    doneA$ = true;
13    doneB$;
14   }
15   doneA$;
16   begin with (in x){ // C
17    writeln(x);
18   }
19  }
```

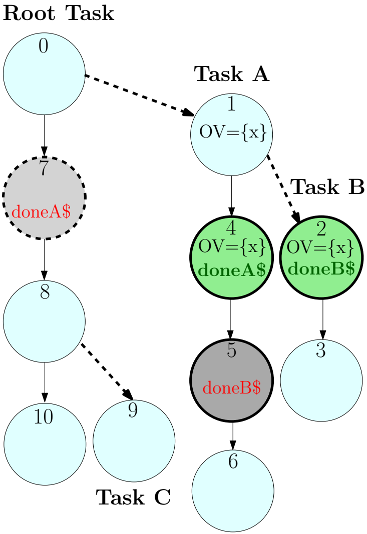
- Task A Line 4.
- Task B: nested task, at Line 7.
- Task C: at Line 16. Pass by value.
- sync variables:
- doneA$: Task A and Root Task.
- doneB$: Task B and Task A.
- Outer variable: x.

Concurrent Control Flow Graph (CCFG)

- CCFG Node bounded by a Concurrent Control Flow event.
    - Encounter `begin` statement.
    - Read/Write on a synchronization variable.
    - Control Flow event
- A CCFG Node
    - Outer Variable Set: OV.
    - Synchronization type.
    - Synchronization variable.
- Sub graph of nested functions expanded at call site.
- A live set of `sync` block scope is maintained.
    - Safe OV accesses are removed.

CCFG

```
proc outerVarUse( ) {
 var x: int = 10;
 var doneA$: sync bool;
 begin with (ref x) { // A
  writeln(x++);
  var doneB$:  sync bool;
  begin with (ref x){ // B
    writeln(x);
    doneB$ = true;
  }
  writeln(x);
  doneA$ = true;
  doneB$;
 }
 doneA$;
 begin with (in x){ // C
  writeln(x);
 }}
```



CCFG pruning

1. Remove empty nodes at the end of each task.
   Eg: Node 10.
2. A `begin` task that does not contain any nested task or does not refers to any outer variable.
   Eg. Task C.
3. A `begin` task, in which the scope of all outer variables accessed by the task is protected by a `sync` block.
   Eg:

   ```
   sync begin (ref x) { ...}
   ```

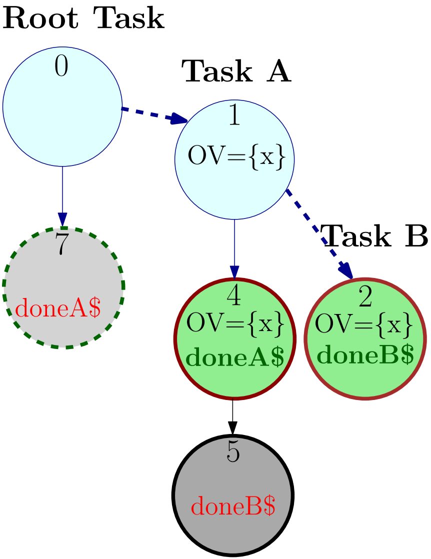
CCFG pruning

1. Remove empty nodes at the end of each task.
   Eg: Node 10.
2. A `begin` task that does not contain any nested task or does not refers to any outer variable.
   Eg. Task C.
3. A `begin` task, in which the scope of all outer variables accessed by the task is protected by a `sync` block.
   Eg:

```
sync begin (ref x) { ...}
```

- Recursively apply these three rules.

Pruned CCFG



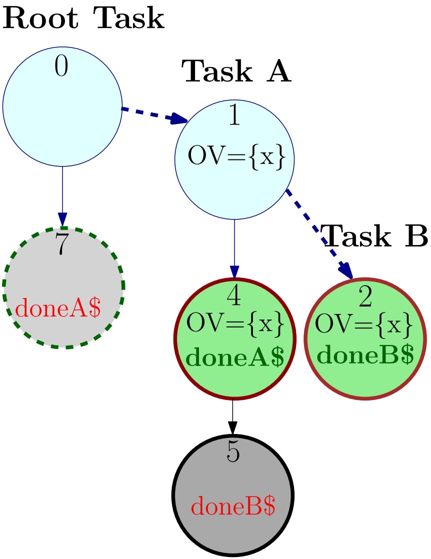
- Active `sync` nodes: 2, 4, 7
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

PPS

- A program state that captures a possible relationship between synchronization nodes.
- A Parallel Program State (PPS):
    - Active Sync Node (ASN): Set of nodes which are next in line to be executed.
    - State Table (ST): State of all live synchronization variables.
    - Safe access set(SV): A set of outer variable accesses which are safe.
    - Live access set (LA): A set of OV accesses which *must have* happened before reaching the current PPS, excluding the set of outer variable accesses in SV.
        - SV $\cap$ LA $= \phi$.

PPS 0



PPS 0:

- ASN = {2, 4, 7 }
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

- SV = ϕ
- LA = ϕ

Parallel Frontier

- Checking for Use-After-Free Variable in each PPS is costly.
- Parallel Frontier: The last sync node encountered in a path in parent scope.
- Defined for every OV, `x`: PF(x).
- Multiple paths could lead to Multiple PF.
- The safety checks limited at PF.

Theorem

A statement that accesses an outer variable x is potentially unsafe if there exists an execution path serialization where the corresponding Parallel Frontier node is executed before the statement.

Next PPS ?

- Design a set of rules to travel in CCFG to generate next PPS.
- Rules designed based on synchronization variables' behaviour.
- Priority : Non-blocking $>$ blocking.

Next PPS ?

- Design a set of rules to travel in CCFG to generate next PPS.
- Rules designed based on synchronization variables' behaviour.
- Priority : Non-blocking $>$ blocking.

Rule (SINGLE-READ (Non blocking))

A read on a `single` *variable is visited if the current state of the variable is full.*

Next PPS ?

- Design a set of rules to travel in CCFG to generate next PPS.
- Rules designed based on synchronization variables' behaviour.
- Priority : Non-blocking $>$ blocking.

Rule (SINGLE-READ (Non blocking))

A read on a `single` variable is visited if the current state of the variable is full.

Rule (READ (blocking))

A read of a `sync` variable can be visited if the current state of the variable is full. The state of the variable is changed to empty.

Next PPS ?

- Design a set of rules to travel in CCFG to generate next PPS.
- Rules designed based on synchronization variables' behaviour.
- Priority : Non-blocking $>$ blocking.

Rule (SINGLE-READ (Non blocking))

A read on a `single` variable is visited if the current state of the variable is full.

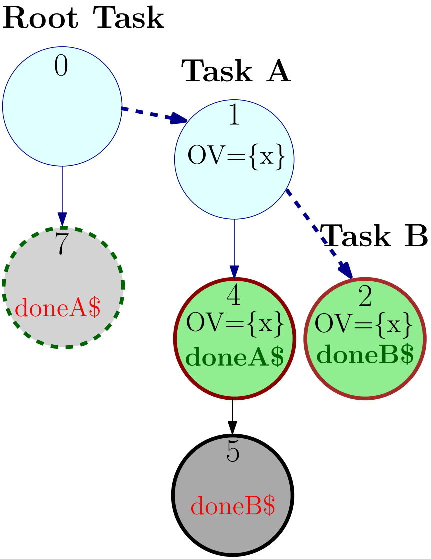
Rule (READ (blocking))

A read of a `sync` variable can be visited if the current state of the variable is full. The state of the variable is changed to empty.

Rule (WRITE (blocking))

A write on `single` or `sync` variable can be visited if the current state of the variable is empty. The state of the variable is changed to full.

PPS 0



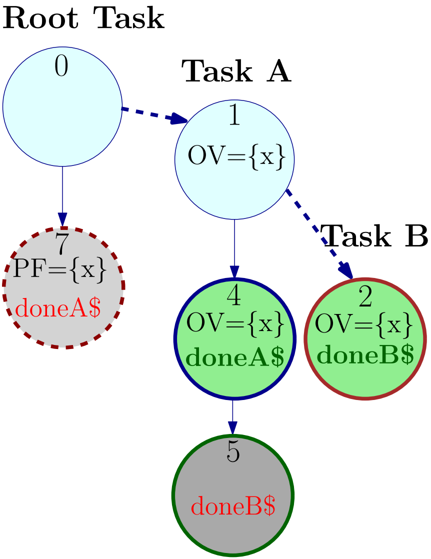
PPS 0:

- ASN = {2, 4, 7 }
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

- SV = ϕ
- LA = ϕ

Execute Node 4



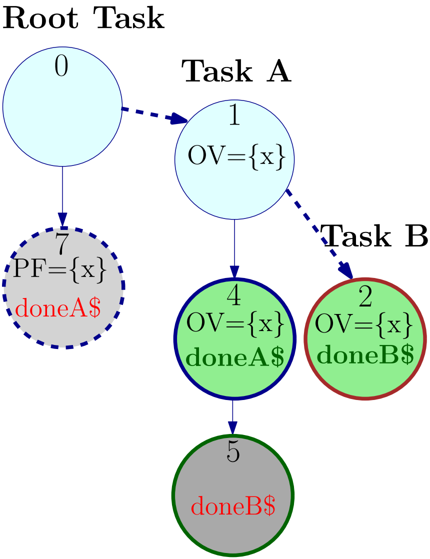
PPS 1:

- ASN = $\{2, 5, 7\}$
- State Table

var	state
doneA$	full
doneB$	empty

- SV = ϕ
- LA = $\{x_1, x_4\}$

Execute Node 7



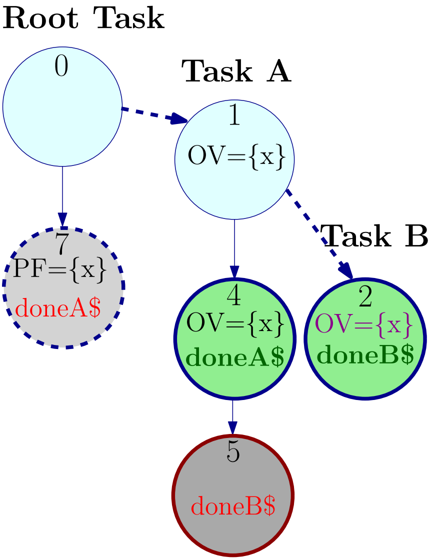
PPS 2:

- ASN = $\{2, 5\}$
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

- SV = $\{x_1, x_4\}$
- LA = ϕ

Execute Node 2



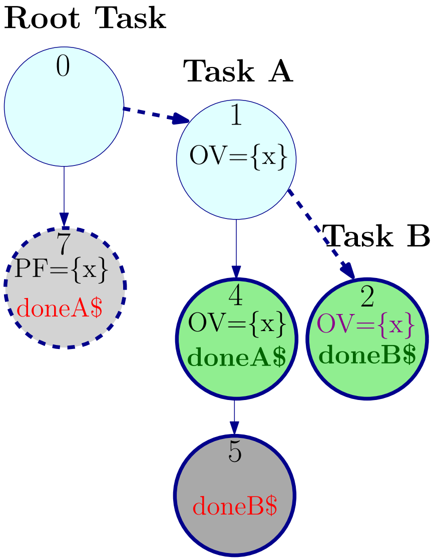
PPS 3:

- ASN = { 5 }
- State Table

var	state
doneA$	`empty`
doneB$	`full`

- SV = $\{x_1, x_4\}$
- LA = $\{x_2\}$

Execute Node 5



PPS 4:

- ASN $= \phi$
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

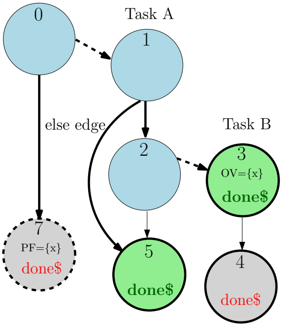
- SV $= \{x_1, x_4\}$
- LA $= \{ x_2 \}$
- **Report** x_2.

Conditional Nodes , Loops

- Conditional Nodes: Both branches are explored separately.
- Loops:
  - Just OV accesses: treated as single node with OV access

Source with Conditional Node

```
var x: int = 10;
var done$: sync bool;
begin with (ref x) { // A
if(flag)
  begin with (ref x) { // B
    writeln(x);
    done$ = true;
    done$;
  }
  done$ = true;
}
done$;
```



Optimizations & Limitations

Optimizations

- Run algorithm only for functions containing `begin` tasks.
- Merging multiple PPS:
  - Requirement: Identical State table & ASN set.
  - Resultant PPS: SV : $SV_i \cap SV_j$, LA : $LA_i \cup LA_j$.
- Combine same variable accesses inside node.

Optimizations & Limitations

Optimizations

- Run algorithm only for functions containing `begin` tasks.
- Merging multiple PPS:
  - Requirement: Identical State table & ASN set.
  - Resultant PPS: SV : $SV_i \cap SV_j$, LA : $LA_i \cup LA_j$.
- Combine same variable accesses inside node.

Limitations: Not Handled

- Non blocking sync events: atomic
- Recursion
- Loops containing `begin` or synchronization node.

Results

Table : Results of running use-after-free check over Chapel test suite (version 1.11).

Total test cases	5127
Test cases with `begin` tasks	218
Test cases with Use-After-Free warnings	38
Number of warnings reported	437
True positives	63
Percentage of true positives	14.4%

`Source Code:` https://github.com/jkrishnavs/chapel

Conclusions

- Partial inter-procedural analysis to identify and report potentially dangerous Outer Variable accesses to the user.
- Results reported on chapel test suite.
- More test cases on https://github.com/jkrishnavs/chapel_workspace

Future Work

- Atomic variable synchronization
- Loops & recursion