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

Jyothi Krishna V S
IIT Madras
jkrishna@cse.iitm.ac.in

Vassily Litvinov
Cray Inc.
vass@cray.com

***Abstract*—**
Programmers use `begin` constructs in Chapel to create *fire and forget*-style tasks, which do not perform any implicit synchronization with the parent task. While this provides a good facility to invoke parallel tasks, it poses issues when the child task accesses a variable declared in the scope of its ancestor. If the parent task exits before the child, its scope is deallocated and the child may end up accessing memory location that is no longer valid. The child task must synchronize with the parent task to ensure legal access to its variables, for example by means of atomic variables, `sync` statements, or `sync` and `single` synchronization variables. In this work, we address the above issue with a compile-time partial inter-procedural analysis for outer variable accesses in `begin` tasks to identify and report potentially dangerous accesses. We make use of a Concurrent Control Flow Graph to generate all possible run-time Parallel Program States (PPS). All outer variable accesses that are potentially dangerous in the generated PPS-es are then reported to the user for rectification.

***Keywords*-Concurrent CFG, PPS, `begin` statement**

I. Introduction

Chapel [1] supports a task based parallel programming model allowing users to design create-and-forget tasks. This is achieved using the `begin` keyword as shown in Figure 1. The tasks thus created are devoid of any implicit synchronization with the parent task. This minimizes the inter-task communication, thereby increasing effective parallelism. Tasks are put into a common task pool at runtime, from which the threads execute the tasks in unspecified order. If no explicit ordering is forced by the users, the parent task of a `begin` task may finish execution before the `begin` task.

A `begin` task can refer to memory locations which are not defined inside its scope. We define such accesses as Outer Variable (OV) accesses. The scope defining the outer variable is labelled as the *parent scope* of the variable. The `begin` tasks with OV accesses should be synchronized with the parent scope, failing which the accesses might result in fetching invalid memory locations. For example in Figure 1 the access of outer variable x in Task B (Line 10) can happen after

This issue gets exacerbated with nested `begin` tasks. Chapel allow task nesting with a `begin` task can create new `begin` tasks inside it. In Figure 1, Task B is a nested `begin` task as it is declared inside Task A. The nested task thus

```
L#
1  proc outerVarUse( ) {
2     var x: int = 10;
3     var doneA$: sync bool;
4     begin with (ref x) { // TASK A
5        // safe access
6        writeln(x++);
7        var doneB$: sync bool;
8        begin with (ref x){ // TASK B
9        // potentially dangerous access.
10          writeln(x);
11          doneB$ = true;
12       }
13       writeln(x);// unsafe access
14       doneA$ = true;
15       doneB$;
16    }
17    doneA$;
18    begin with (in x){ // TASK C
19      writeln(x);
20    }
21 }
```

Figure 1: Chapel code with three tasks with outer variable access. The outer variable `x` is accessed in Lines 6, 10 and 13. The `sync` variables `doneA$` and `doneB$` is used to synchronize with the parent task. The access in Line 10 may happen after the parent task has exited and hence is potentially dangerous.

created lacks any implicit synchronization with the parent `begin` task or any of its ancestor tasks.

Outer variables can be passed to `begin` like a normal function, with the `ref` keyword used to denote variables which are passed by reference. If the `begin` task is interested only in the value of the variable (at the time of task definition) users can make use of the `in` keyword which will ensure that all accesses to the variable inside the `begin` task are made on a local copy and operations executed on the variable are not visible outside the task.

Another challenge while dealing with the nested task is Chapel allows function nesting where a procedure can be defined inside another procedure. All live variables of the parent procedure at the time of definition are accessible in the child procedure. Such nested functions can be invoked from a `begin` task, thus resulting in outer variable accesses inside the `begin` task without passing a reference of the

variable to the `begin` task.

Chapel allows users to have a point-to-point synchronization between the parent task and `begin` using special synchronization variables, `sync` and `single`. This allows the users to control and delay the synchronization point between two tasks and judiciously allow the parent task to progress as much as possible. Apart from the `sync` and `single` variable Chapel also makes use of atomic integer operations for point-to-point synchronization. In this work, we limit our discussion to just the synchronization variables.

Consider the Chapel program given in Figure 1, with three `begin` statements starting at Lines 4 (Task A), 8 (Task B) and 18 (Task C). The outer variable to these begin tasks `x` is defined in Line 2. Task C accesses a local copy of the variable `x` (pass by value), ensuring the safety of all accesses of `x` in the task. Task A and the nested Task B refer to the original memory location of the variable and accesses of the memory location inside these tasks need to be properly synchronized with the parent task to ensure the validity of the address location.

Our aim is to identify outer variable uses inside `begin` tasks without proper synchronization with the parent scope of the variable. In Figure 1, the outer variable `x` is accessed in lines 6, 10 and 13 each of which should be synchronized properly with the parent task. The safety of variable accesses in Lines 6 and 13 (in Task A) is ensured as the parent task would wait until the Line 14 is executed in Task A. However, the variable access in Line 10 is dangerous as there is no scenario where the parent task, is waiting for the Task B. This could cause the parent task to exit and invalidate the memory location of variable `x` before the task B access the location. If we swap positions of lines 14 and 15 in Task A, a wait chain from Task B to the parent task (line 11 $\rightarrow$ line 14 $\rightarrow$ line 15 $\rightarrow$ line 17) will ensure the safety of the variable access in line 10. If there were multiple paths that the program execution can follow (due to branches) our analysis ensure that all accesses are safe in all possible execution paths.

For identifying such dangerous accesses we abstract out the use of outer variable uses, creation of `begin` tasks and synchronization events in the given program into a Concurrent Control Flow Graph (CCFG) [2]. In CCFG, the nodes store the details of outer variable uses which is linked to its parent scope, along with the synchronization events. The edges provide the information on the control flow and `begin` task creation. For the CCFG abstraction generated a conservative approximate set of all Parallel Program State (PPS) that are possible at execution-time is computed. The compiler use these PPS-es to identify the outer variable accesses that are not properly synchronized with the parent scope. If such occurrences are found, the compiler reports the point of outer variable access along with the outer variable under consideration to the user.

The paper makes the following contributions:

- To the best of our knowledge, this is the first work providing a solution for identifying the accesses-after-free of external memory locations in a task parallel environment enabled with point-to-point synchronization.
- We extend the solution to handle hidden external variable accesses in nested functions where the external variables are not passed on as an argument.

From our experiments, our analysis is able to identify 63 use-after-free instances of outer variable uses in the Chapel version 1.11 [3] test suite.

Section II gives a brief explanation of Chapel language constructs used in the paper. Section III explains our compiler pass in detail. Section IV gives details about the implementation. Section VI provides a brief discussion about the related work. Section V presents experimental evaluation. Section VII lists the conclusions and possible future work.

II. Chapel Background

In this section, we give brief description about synchronization options available for `begin` in Chapel. The `begin` tasks with outer variable accesses may use point-to-point synchronization to synchronize with the parent scope of the outer variable.

`single` and `sync` are specialized synchronization variable types available in Chapel. Programmers use `single` variable to synchronize a single point in child task to multiple points in parent task and a `sync` variable to reuse the same `sync` variable to synchronize multiple events. A synchronization variable can have two states: *full* and *empty*. Full state is equivalent to a set flag, whereas the empty state indicates an unset flag. A variable can change from empty to full when a value is written into it. A `single` variable, once filled, will not change to empty state whereas a read on a `sync` variable will change the state from full to empty, clearing the value written into it. The synchronization variables (of type `sync` and `single`) are need not be passed to the `begin` task as they are universally visible. The synchronization variables are named with a suffixed $ by convention to distinguish them from normal variables. This helps to reduce the odds of users introducing reads and writes on synchronization variables that can result in infinite waiting loops. The default initialization of all synchronization variables is the empty state. The synchronization variables can be explicitly initialized to the full state. The reads and writes to `sync` variables are internally executed using special functions readFE and writeEF respectively. The *readFE* method blocks the execution of the current task until the `sync` variable is full. The value of the `sync` variable is cleared and the state set to empty when this method completes. The *writeEF* method blocks until the `sync` variable is empty. The state of the `sync` variable is set to full when this method completes. The reads on a `single` variable are executed using a readFF function. The *readFF* method blocks until the variable is full, but will

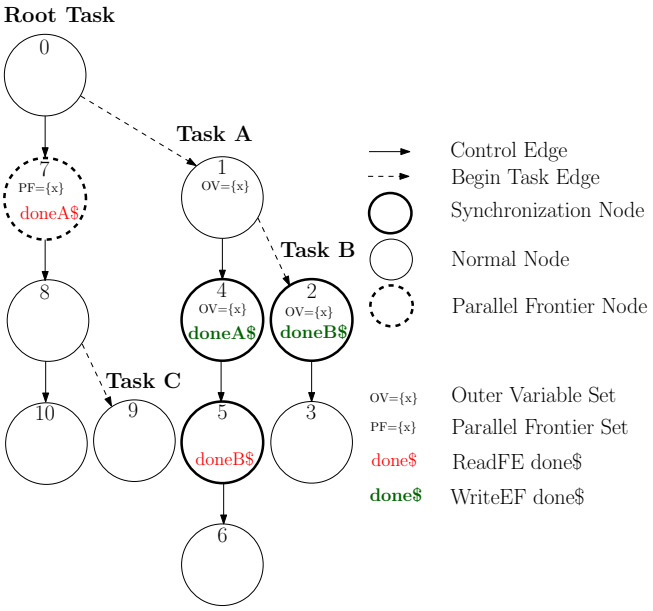
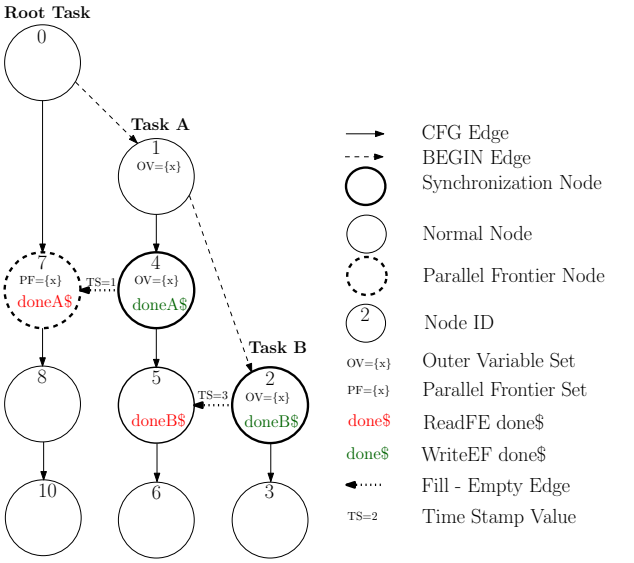


Figure 2: CCFG generated for procedure **`outerVarUse`** in Figure 1. Empty Outer Variable sets are not shown.



ID	TS	ASN	OV	SV	states	Remark
0	0	{2,4,7}	ϕ	ϕ	doneA$=E, doneB$=E	$\xrightarrow[Node\#4]{rule\#3}$ 1
1	1	{2, 5, 7}	$\{x_1,x_4\}$	ϕ	doneA$=F	PF(x), $\xrightarrow[N\#7]{r\#2}$ 2
2	2	{2,5}	ϕ	$\{x_1,x_4\}$	doneA$=E	$\xrightarrow[N\#2]{r\#3}$ 3
3	3	{5}	$\{x_2\}$	$\{x_1,x_4\}$	doneB$=F	$\xrightarrow[N\#2]{r\#3}$ 4
4	4	ϕ	$\{x_2\}$	$\{x_1,x_4\}$	doneB$=E	Sink PPS

Figure 3: Sync variables paired for procedure `outerVarUse` in Figure 1. Some of the safe transitions are omitted. The subscript of the variable in OV set denotes the node corresponding to the access. In state column E means empty and F means full.

retain the full state and the value of the variable when the method is complete. Multiple writes to a `single` variable are prohibited as per Chapel specifications [1].

In Figure 1, two `sync` variables `doneA$` and `doneB$` are defined to protect the accesses of the outer variable x, in Tasks A and B respectively. Since, no initialization state is specified, both `doneA$` and `doneB$` are initialized to the empty state. The state of `sync` variable `doneA$` is set to full in line 14 (inside Task A). Since initial state of the `doneA$` is empty, Line 14 can be executed without any wait time. Line 17 in parent task, waits until the state of `doneA$` is full and then changes its state to empty. Hence, the parent task waits until the Task A statement in line 14 is completed. Similarly, Task A (at line 15) waits for Task B (at line 11) due to the synchronization variable `doneB$`.

As the number of `begin` tasks increases the point-to-point synchronization might become costlier. In such cases the programmer can create a `sync` block bounding all the `begin` tasks. A `sync` block creates a *fence* that blocks the parent task at the end of the `sync` block until all child tasks declared inside the `sync` block complete. This ensures that the parent task does not progress until the `begin` task finishes its execution, and thereby ensures that the outer variable of concern is alive at the time of access.

III. Outer Variable Use After Lexical Scope

In this section, we explain the compiler pass in detail. The compiler pass is executed on the Chapel intermediate code representation, where the special read/write functions for `sync` and `single` are embedded in. To improve the analysis scalability, we restrict ourselves to a partial inter-procedural analysis where the outermost procedures containing the `begin` tasks are analyzed separately.

The overall compiler analysis can be divided into two parts. For performing our analysis, we need to know the control flow and the synchronization flow in a concurrent settings. We capture these information along with the outer variable accesses using a concurrent control flow graph (CCFG). The algorithm for finding unsafe memory accesses run on the constructed CCFG.

A. CCFG Construction

A CCFG is an abstraction of the input Chapel program capturing the concurrent control flow details and outer variable accesses. Each node in the CCFG is bounded by a Concurrent Control flow event like encountering the `begin` statement, read/write on a synchronization variable (readFE, readFF or writeEF) or sequential control flow events like branches. The CCFG thus created is then subjected to a

pruning process based on a set of predefined rules for removing safe nodes.

Figure 2 represents the CCFG generated from the Chapel code given in Figure 1. In Figure 2 Nodes 0,7,8,10 the represent the parent task strand (1,4,5,6) comprehends the Task A, Nodes 2 and 3 comprehends Task B and node 9 represents task C.

Each node is associated with an Outer Variable (OV) set, which maintains the set of outer variable uses inside the node. For example in Figure 2, the OV set of Node 1 is {x}. This represents the use of outer variable x by Task A in Line 6 in Figure 1. Similarly the x in the OV set of Node 4 represents the use of x in Line 13. Each outer variable use is associated with the declaration information of the variable. The declaration information will contain the details of the parent scope. For instance, the outer variable uses in Nodes 1, 2, 4 are associated with the declaration information of x which stores the end of parent scope details (Node 10).

The sub-graphs of the nested functions of the parent task are maintained separately and are also inlined at the call sites. This allows our analysis to identify all hidden outer variable accesses in the `begin` task, that might occur due to a call to a nested function from the `begin` task. Inlining the nested functions makes our analysis context sensitive and improves the precision of analysis. As a result of inlining, multiple copies of the same begin task can be present in the CCFG. During inlining, we maintain a call stack of the nested function to identify recursion and prevent infinite expansion. We stop inlining as soon as we identify a recursion.

Nodes which contain operations on synchronization variables are classified as *sync* nodes. As soon as we encounter a synchronization event a new node is generated. Hence the CCFG can have have at-most one synchronization operation per sync node. The sync nodes store the synchronization variable and their potential state change. In Figure 2 the nodes 2, 4, 5 and 7 represents the sync nodes.

The edges can be classified as either control edges or task edges. The task edges denote the creation of a new `begin` task. In Figure 2 the dotted arrow lines represent the task edges (for example edge $0 \rightarrow 1$). The control edges illustrate the control flow direction (for example edge $0 \rightarrow 7$).

CCFG pruning: Not all tasks need to be analyzed for OV accesses. In particular, tasks that do not have external memory accesses and synchronization events which will affect the relative execution of rest of the tasks are *safe*. In Figure 2 the node 9 representing Task C can be pruned out since it does not contain any external memory access or synchronization event. Our analysis identifies safe tasks for CCFG pruning. The tasks identified as safe are pruned without affecting the correctness of the analysis.

Our analysis uses the following rules to identify safe tasks.

Rule A. A `begin` task that does not contain any nested task or refers to any outer variable.

Rule B. A `begin` task, which is immediately encapsulated by a `sync` statement, provided all nested tasks are safe.

Rule C. A `begin` task, in which the scope of all external variables accessed by the task is protected by a `sync` block.

Rule D. A `begin` task, in which all nested tasks are safe and is by itself not referring to any outer variable.

Figure 3 shows the pruned CCFG. Here, Node 9 (representing Task C) is pruned using Rule A.

Analyzing CCFG to identify tasks that satisfy one of these rules necessitates maintaining the live `sync` blocks along with their scope information. For this, we maintain a *synced-scope* list while generating the CCFG, which stores the scopes of live `sync` blocks.

The outer variables can be passed as an argument to the root function / task. Usually programmers ensure that the calls to such functions are enclosed by `sync` statements as it ensures safety while maintaining simplicity. If all call sites from which the root function is invoked are enclosed by `sync` statements, then all the outer variable accesses passed as argument to the root (parent) function is safe. Our compiler add such root (parent) function to synced-scope list. This ensures that the compiler identifies all accesses to outer variables that are passed as arguments to the root function as safe.

B. Identifying Unsafe Uses

Next step of our analysis works on the pruned CCFG to identify potentially dangerous OV accesses due to lack of or improper use of synchronization variables. This step makes use of an abstraction called Parallel Program State (PPS) which represents an execution-time program state partial ordering [4] is used to identify the potentially dangerous outer variable accesses. We may have numerous possible PPS-es if we consider all possible serialization of the nodes in CCFG. Instead of considering all possible PPS-es, we can conservatively consider all possible serialization of synchronization events (abstracted to sync nodes and task edges in CCFG). A write (writeEF) on a synchronization variable in synchronization node j, can be paired with a corresponding read (readFE / readFF) in node k, if in any of the CCFG paths the value written in node j can be consumed by the read in node k at run time, giving a partial ordering of the nodes in CCFG similar to Lamport's time-stamp [4] (A signal-wait pair with writeEF modelled as a signal and readFE modelled as a wait).

Formally, a PPS is identified by:

1) A set of sync nodes called Active Sync Node (ASN) set, which represents the sync nodes which are next in line to be executed
2) A table storing the state of all synchronization variables at the PPS called the state table (ST) and

3) A set of outer variable accesses which are safe, Safe Set (SV).
4) A set of all outer variable accesses (OV) which *must have* happened before reaching the current PPS, excluding the set of outer variable accesses in SV. For any given PPS, the OV ∪ SV set contains all outer variable accesses which must have happened before the last synchronization event in the execution path taken.

   SV ∩ OV = ϕ.

In Figure 3, each row in the table represents a possible PPS. At time stamp 0, we have three sync nodes (Nodes 2, 4, 7) available for execution (PPS 0). The PPS 0 contains sync nodes 2,4,7 will be available in ASN set. Both the OV set and the SV set of the PPS 0 are empty. A sync node can be executed if we can apply one of the below stated rules.

Rule 5 (SINGLE-READ)**.** A read on a `single` variable is visited if the current state of the variable is full.

Rule 6 (READ)**.** 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 7 (WRITE)**.** 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.

Of these, Node 7 cannot be executed since the current state of `doneA$` variable was empty. The subset of ASN nodes which can be executed next, is termed as the *candidate* set. We can apply WRITE rule on Nodes 2 to 4. Every application of one of the rules on the current PPS with a non-empty ASN set results in a different PPS which is appended to the work-list. We proceed by applying WRITE rule on Node 4 ($S_i = 4$). This updates the state table entry of doneA$ from EMPTY to FULL. We add the outer variable accesses between the previous sync node (S_{prev} = the root node) and S_i (Node 4) into the OV set. The path contains three nodes (Nodes 0, 1, 4) and contains two outer variable accesses (`x` in Nodes 1 and 4) which are added to the OV set (the subscript in OV set in Figure 3 refers to the Node corresponding to the access). Before adding an outer variable access into OV set, the variable access is searched in both the OV set and the SV set to avoid duplicate additions. Once node 4 is executed, we move forward in Task A, and sync node 5 is now added to the ASN set (PPS 1).

In PPS 1, the candidate set includes Nodes 2 and 7 only. A PPS with an empty ASN set (*sink-PPS*), denotes we have visited all synchronization events which are part of an execution path (PPS 4 in Figure 3). Figure 3 shows one possible execution path. We recursively follow all possible execution paths in the execution tree, one path for each of the nodes available in candidate set. For example, { (2, 4, 5, 7) , (2, 4, 7, 5) , (4, 2, 7, 5) , (4, 2, 5, 7) } are a few of the possible sync-node execution paths where the use of variable `x` in Node 2 is safe.

In CCFG, the last sync node (readFE/writeEF/writeFF) encountered in a path inside the parent scope for an outer variable x is classified as the Parallel Frontier of x (PF(x)) in that path. If there are multiple paths from the start of the parent scope to the end of the scope, there can be multiple PF nodes one for each path. In Figure 2 the set PF(x) contains a single Node 7.

Property 1. 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.

The parallel frontier represents the last synchronization point to which all the OV accesses can synchronize. The Parallel Frontiers can change based on the CFG path chosen in the parent task. As soon as a PF(x) node becomes part of the candidate set of a PPS, all the accesses to variable x in the OV, which were synchronized before PPS are cleared from the OV set and added to the safe set (SV) as those accesses are accepted to be safe in the current path of execution. In Figure 3, the Node 7, becomes part of the candidate set in PPS 1. The outer variable accesses in OV set (x_1, x_4) are moved to SV, in the next PPS (PPS 2).

Once a node (S_i) is processed, we generate a new PPS by finding the successor of the processed node in the CCFG and appending it to the remaining sync nodes in the previous ASN set. We repeat until no new PPS can be generated. A PPS with an empty ASN set is called the sink PPS. The outer variable accesses in OV set at sink PPS, were added after the PF(x) was available for execution and hence there exists an execution path where these accesses can happen after the end of scope of the variable. All outer variable accesses which are part of OV set at sink PPS, are deemed unsafe and reported to the user. In Figure 3 the outer variable access x_2 was added after execution of PF(x) (Node 7) making it a potentially dangerous access of variable x. If lines 14 and 15 in the example program were to exchange position, the generated graph would have had the synchronization variables of Nodes 4 and 5 swapped. In the resultant CCFG, we can only follow a single execution path (Node 2 → Node 4 → Node 5 → Node 7), and all the outer variable accesses would have been added to the OV set before the PF Node 7 ca be executed. This would make all accesses to variable x safe.

C. Algorithm

The algorithm for identifying the unsafe variable accesses (`checkForUnsafeUse`) is provided in Figure 4. In lines 5-10, we report the potentially dangerous variable accesses to the users.

```
L#  checkForUnsafeUse(ppsWL)
1   SL: filled single variable List
2   ppsWL: PPS Worklist.
3     while(! ppsWL.size() == 0):
4     pps <- ppsWL.pop()
5       if pps.ASN == φ:
6         ∀ ev_i ϵ pps.OV:
7           Throw warning( ev_i )
8         ∀ ev_i !(visited):
9           Throw warning( ev_i )
10        continue
11      ∀ S_i ϵ pps.ASN:
12        //SINGLE READ
13        ∀ S_i.syncVar ϵ pps.SL:
14          sli.add(S_i)
15      if sli.size() > 0:
16        findNewPPS(ppsWL, sli, pps, FULL)
17      ∀ S_i ϵ pps.ASN:
18        // READ
19        if S_i.syncVar.op == writeEF &&
20         pps.status(S_i.syncVar) == EMPTY:
21          findNewPPS(ppsWL, S_i, pps, FULL)
22        // WRITE
23        else if S_i.syncVar.op == readFE &&
24         pps.status(S_i.syncVar) == FULL:
25          findNewPPS(ppsWL, S_i, pps, EMPTY)
```

Figure 4: Algorithm for finding unsafe accesses. Procedure `checkForUnsafeUse` iterates over the PPS work-list and reports the unsafe access points.

Since the SINGLE-READ rule reflects reading a `single` variable that is already in the *full* state, the synchronization event is non-blocking. Also, the SINGLE-READ rule applied to a sync node does not block other nodes in the current ASN set to be processed by applying the same rule. To maintain correctness, we apply rule SINGLE-READ whenever possible as a bunch, before proceeding to apply the READ and WRITE rules. The rest of the non blocking synchronization events can also be incorporated similar to SINGLE-READ. Lines 11-16 in `checkForUnsafeUse` handle SINGLE-READ, the non blocking synchronization event.

The `findNewPPS` procedure (displayed in Figure 5) generates new PPS and updates the OV set. The `findNewPPS` procedure also updates the ASN set, the state table (Lines 7 to 16) and the OV set (Lines 18-21) of the new PPS . If the current executed node is a Parallel Frontier of a given variable `x`, then we move all accesses of the variable `x` in OV set to safe set (SV) (Lines 23-26). The newly generated PPS, is appended to the work-list (Line 30).

If multiple branches are present, different PPS-es are generated with each of the possible branches (Lines 7 - 12). All newly available `begin` tasks are added to the list of paths searched for new sync nodes.

Optimization: As an optimization, we merge two or more PPS-es if their ASN sets and the state tables (ST) are identical. In the merged PPS the OV set will be union of

```
L#  findNewPPS(PPSWL,sV,pps,state)
1   sV: set of sync nodes visited
2   state: new state.
3    (partialSet,OV,sT)<-(pps.(ASN,OV,sT))
4    partialSet -= sV
5    sT[sV.syncVar] <- state
6
7    ∀ S_i ∈ sV:
8      S_j <- nextSyncNode(S_i)
9      if S_i to S_j contains branches:
10        ∀ b_l in branches:
11       findNewPPS(PPSWL, sV+ b_l − S_i,
12          pps, state);
13     if S_i to S_j contains begin:
14       ∀ t_l in new begin(s):
15         sV.append(t_l)
16     partialSet.append(S_j)
17  // update OV set
18    ∀ S_i ∈ sV:
19      ∀ N_k from S_prev to S_i:
20        if N_k !(visited) :
21          OV = OV ∪ N_k.OV
22      // update safe set
23      ∀ S_i ϵ partialSet:
24      //check if PF(x) is in candidate set
25      if canVisitNext(S_i, sT)
26       &&  S_i ϵ PF(x):
27         ev(x) -> safe //xsafe set
28
29   (newpps.(ASN,ev,sT))<-(partialSet,ev,sT)
30   ppsWL.append(newpps)
```

Figure 5: `findNewPPS` updates the work-list with new PPS, and updates the safe accesses on encountering a parallel frontier. S_{prev} represents the previous sync node, in the path we encountered S_i.

the original OV sets. Since the ASN set and the state table are identical, the candidate set of the original PPS-es are also equivalent. The safe set (SV), of the combined PPS will be the intersection of the original SV sets. The PPS is set to unprocessed if at least one of the original PPS was unprocessed.

The analysis does not maintain the candidate set of PPS as it can be easily generated when required, using the ASN set and the state table. The algorithm can identify a variable access ev_i as unsafe in more than one execution path. Since, we are interested only in finding the dangerous accesses, the algorithm removes the newly identified dangerous access ev_i from further analysis.

D. Conditional Branching

Consider Figure 6 which presents a Chapel code where we have branches inside `begin` tasks (Line 7). The branch condition (the value of flag in Figure 6) could be unknown at compile time and our analysis conservatively consider all possible paths that can be chosen at runtime, while generating the PPS. Figure 7 presents the CCFG of the program along with the various PPS obtained.

```
L#
1  config const flag = true;
2  proc multipleUse() {
3     var x: int = 10;
4     var done$: sync bool;
5     // Task A
6     begin with (ref x) {
7       if(flag) {
8         // Task B
9         begin with (ref x){
10          writeln(x);
11          done$ = true;
12          done$;
13        }
14      }
15      done$ = true;
16    }
17    done$;
18 }
```

Figure 6: Chapel code with multiple paths and rules. If the flag is true and the branch is taken the access of variable x in Line 10 in Task B may be potentially dangerous.

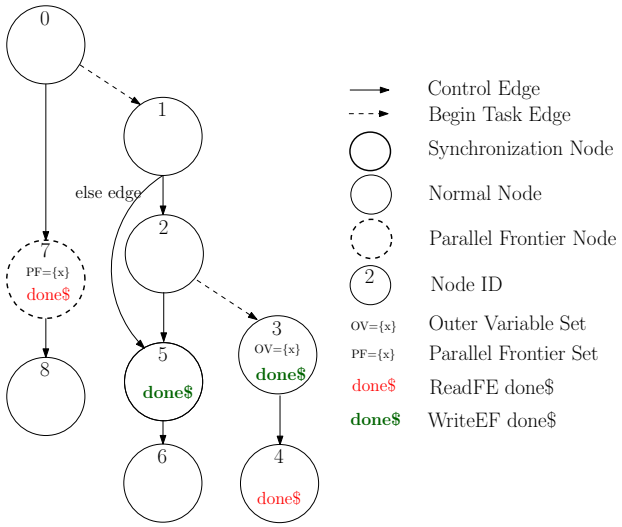
At run-time we can either take or skip the if-branch. We generate the PPS initially, considering the if condition (PPS 0 in Figure 7) and latter one omitting the if condition (PPS 8). Both are maintained at time-stamp 0. While considering the if condition, at stage 0, we have two possible applications of Rules 3 and 2 (Nodes 3,7 and Nodes 5,7) with the nodes in the ASN set. Both possibilities are separately considered which further generates multiple PPS-es. When we visit PF node (Node 7), in PPS ID 4 (to generate 5) the OV set is empty and hence none of the accesses of x are made safe. Later, (PPS ID 5) when the algorithm visits Node 3, the outer variable access in Node 3 (Line 10 in Figure 6) is added to OV list. This access is identified as unsafe we reach sink-PPS (PPS ID 7).

IV. Implementation

The proposed compiler analysis is implemented as a compiler pass on the latest Chapel compiler. The analysis works on Chapel intermediate code representation. The compiler code with our pass is available online on github [5]. All potentially dangerous accesses identified by our analysis are reported to the user as a compiler warning for manual verification and rectification.

A. Scope Limitations

Currently our analysis does not support loops which contain a sync node or a `begin` task edge inside the loop body. Loops with just the outer variable accesses are treated as a single node if no synchronization event happens between the variable access in the first iteration and the variable access in the last iteration in the current task. The rest of the loops are subsumed into a single node in the generated CCFG.



ID	TS	ASN	OV	states	Remark
0	0	{3,5,7}	ϕ	done$=E	IF, $\xrightarrow[Node\#3]{rule\#3}$ 1 $\xrightarrow[Node\#5]{rule\#3}$ 4
1	1	{4, 5, 7}	$\{x_3\}$	done$=F	PF(x), $\xrightarrow[N\#7]{r\#2}$ 2
2	2	{4,5}	ϕ	done$=E	$\xrightarrow[N\#5,4]{r\#3,2}$ 3
3	4	ϕ	ϕ	done$=E	safe
4	1	{3,7}	ϕ	done$=F	PF(x), $\xrightarrow[N\#7]{r\#2}$ 5
5	2	{3}	ϕ	done$=E	$\xrightarrow[N\#3]{r\#3}$ 6
6	3	{4}	$\{x_3\}$	done$=F	$\xrightarrow[N\#4]{r\#2}$ 7
7	4	ϕ	$\{x_3\}$	done$=E	unsafe
8	0	5,7	ϕ	done$=E	ELSE, $\xrightarrow[N\#5,7]{r\#3,2}$ 9
9	2	ϕ	ϕ	,,	safe

Figure 7: The CCFG and the PPS set generated for the Chapel program given in Figure 6. The PPS-es 1-7 represent the case where the IF branch is taken where as PPS-es (8,9) represent the case where the IF branch is not taken.

Of the multiple non-blocking synchronization events, only SINGLE-READ is currently handled. The atomic integers can also be used to synchronize `begin` tasks with the parent task. Writes to atomic integers can be taken as a non-blocking fill event (state changing from EMPTY to FULL), and the corresponding read can be considered equivalent to SINGLE-READ event. In the current implementation of the analysis, the atomic integer read-writes are not handled.

V. Results

For evaluation purpose we use the the test suite available with Chapel version 1.11 [3], which had many programs with potentially dangerous outer variable accesses. Our

Table I: Results of running use-after-free check in Chapel version 1.11 test suite.

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%

observations are given in Table I. From the list of memory accesses reported by our compiler, we were able to manually verify 63 potentially dangerous memory accesses in the test suite.

The low percentage of true positives (14.4%) reported is due to the non-handling of atomic integers, non blocking synchronization events (other than readFF) and lack of complete inter procedural analysis.

VI. Related Work

In this section we discuss the related work, including some works on static MHP analysis and static race detection, both closely related problems.

In X10 [6] and HJ [7], the `async` (equivalent to `begin`) tasks which have references to outer memory locations are enforced to be enclosed within a `finish` block (equivalent to `sync` block). This approach of blocking the parent task can be heavily restrictive and inefficient at times, especially when the parent task has a sufficient amount of work after the `sync` statement. Also, the `sync` block has no effect on the tasks defined outside its scope, prompting the programmers to create multiple `sync` blocks which can result in high run-time cost.

Compile time identification of concurrent program issues is a challenging and fascinating area. Multiple works in the field of compile time may-happen-in-parallel (MHP) [8]–[12] and static race detection [13]–[16] have been proposed recently with great success.

We could perform our analysis using an efficient, sound and precise MHP oracle by relying on the following property: any outer variable access is potentially dangerous if the end of the variable scope may-happen-in-parallel with the access or any preceding statement. The MHP oracle will be inefficient if the accesses may happen after the parent task exited.

Different variations of CCFG are used (like segment graphs [8], Parallel Reachability Graph [12]) for capturing and comprehending the program control flow for the static MHP analysis.

By restricting the analysis to an environment containing only the `sync` blocks and `begin` tasks the CCFG for MHP analysis can be comprehended into a tree structure (Program Structure Tree (PST) [10], [11]) where the `begin` task nodes can be attached as a child node to the immediately enclosing `sync` block. In MHP analysis for a thread parallel model [9] a Thread Creation Tree (TCT) is constructed where multiple call sites of the same thread are maintained separately to retain the context sensitivity of the analysis. In our model we copy the entire sub-graph of the embedded function (in-lining) at all call sites to maintain the context sensitivity. Parallel Reachability Graph (PRG) [12] maintains the target task (spawn and merge) information along with each node which helps reduce the individual query time of MHP for each node to O(V+E) where V is the number of vertices in PRG and E the number of edges. None of the above mentioned algorithms handle point-to-point synchronization, allowing them to simplify the representation graphs and provide polynomial-time guarantee on MHP algorithms. In segment graphs [8] the parallel tasks are further divided into segments to improve the precision of the MHP analysis.

Static race detection is another well studied area which has some overlapping with the outer variable detection. For example, Lockset based algorithms [13], [15] developed for static race detection can be used to optimize our state table of synchronization variables. The full and empty states along with synchronization variable names can be hashed into a set of virtual locks which need to be acquired to execute the blocking synchronization event. The function summarization techniques used in [14] can be used to extend the precision of the inter-procedural analysis for non nested function calls.

VII. Conclusions

We proposed a partial inter-procedural compiler pass in Chapel to identify access to variables that could happen after the base task has exited the scope of the variable, rendering the address void. Such accesses are potentially dangerous and must be avoided. We have discussed the generation of a CCFG and parsing the CCFG to identify improperly synchronized variable accesses that can happen after the Parallel Frontier (PF) of the variable. This will help the users to identify the potentially dangerous access and add synchronization statements to ensure the safety. In future, we would like to extend the analysis to handle loops and atomic integers. The analysis can be extended to optimize the amount and position of synchronization points required, and identify potential deadlock points.

Acknowledgment

The authors would like to thank Dr. Rupesh Nasre and Chandramohan T. N. for spending their valuable time in reviewing this work. We would also like to thank our initial reviewers of the workshop for their helpful comments and Dr. Cosmin Oancea for guiding through the shepherding process.

REFERENCES

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[2] S. Rajasekaran and J. Reif, *Handbook of Parallel Computing: Models, Algorithms and Applications (Chapman & Hall/Crc Computer & Information Science Series)*, 1st ed.

[3] "Chapel Evolution," http://chapel.cray.com/docs/latest/language/evolution.html#version-1-11-april-2015, accessed: 2017-02-28.

[4] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[5] "A productive parallel programming language http://chapel.cray.com/," https://github.com/jkrishnavs/chapel, accessed: 2017-02-28.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[7] Z. Budimlić, V. Cavé, R. Raman, J. Shirako, S. Taşirlar, J. Zhao, and V. Sarkar, "The Design and Implementation of the Habanero-java Parallel Programming Language," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 185–186. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048198

[8] W. Chen, X. Han, and R. Dömer, "May-happen-in-parallel Analysis Based on Segment Graphs for Safe ESL Models," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 287:1–287:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2617025

[9] R. Barik, "Efficient Computation of May-happen-in-parallel Information for Concurrent Java Programs," in *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 152–169. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69330-7_11

[10] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel Analysis of X10 Programs," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 183–193. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229471

[11] A. Sankar, S. Chakraborty, and V. K. Nandivada, "Improved MHP Analysis," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 207–217. [Online]. Available: http://doi.acm.org/10.1145/2892208.2897144

[12] C. Chen, W. Huo, and X. Feng, "Making It Practical and Effective: Fast and Precise May-happen-in-parallel Analysis," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 469–470. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370900

[13] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 308–319. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134018

[14] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and Accurate Static Data-race Detection for Concurrent Programs," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 226–239. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770351.1770386

[15] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 205–214. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287654

[16] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 258–269. [Online]. Available: http://doi.acm.org/10.1145/512529.512560