Online Appendix to: Compiler assisted memory management for safety-critical hard-real time applications

Avinash Malik, University of Auckland
HeeJong Park, University of Auckland
Muhammad Nadeem, University of Auckland
Zoran Salcic, University of Auckland

A. DATA-FLOW ANALYSIS PSEUDO-ALGORITHMS

Figure 15 gives the pseudo-code implementing the data-flow analysis algorithm for compile time memory allocation.

B. THE AGRC ANALYSIS ALGORITHM

The algorithm for obtaining *may* and *must* relationships between the AGRC nodes is shown in Algorithm 1. The input to this algorithm is an AGRC, which is a tuple (V, E) as explained previously. The first part of the algorithm, lines 1-5, groups all the nodes, which are belonging to the same clock-domain. For instance, lines 3-4 initializes a set V_r, which consists of a root node of all clock-domains in AGRC. The algorithm then traverses every clock-domain graph, starting from these root nodes via a recursive function `collect_cd_nodes`(v, E) (line 5). A result is a set of set of AGRC nodes V_{cd}. Next *May* and *Must* analysis is performed (lines 6-10). Every node in V_{cd} is given as an input to the function `create_node`(v, E) (line 9), which traverses the graph backward starting from the input until it reaches the root node. Function `create_node` first maps the input node to a set of edges E_i using the lambda function (line 14). It is then divided into three parts:

(1) If the current node type is an `JDN` node (line 15) and
   (a) the number of element in E_i is 1, then the only parent node v_i *must* be called before v_j. Then a new node of type $mNode$ is created with its field $Must$ initialized to the result of recursive call of `create_note`.
   (b) the number of element in E_i is greater than 1, then any parent nodes $v_i \in E_i$ *may* be called before v_j. A set of $mList$, consisting return results of `create_node` of each parent node $v_i \in E_i$, is assigned to the field May of a newly created $mNode$.
   (c) the number of element in E_i is 0, then a new $mNode$ is created with both its $Must$ and May fields initialized to $Null$.
(2) If the current node type is `Join`, this means that there are two or more parallel reactions forked at the `Fork` node, which *must* be found in one of the recursive parents of the current node. Hence, the algorithm must continue the backward traversal from this `Fork` node. During compilation, the compiler has already constructed a Hashtable which maps each `Join` node to its corresponding `Fork` node as shown in line 28.
(3) If the current node type is other than `JDN` (line 31) and
   (a) the number of parent node ($|E_i|$) is 0 , then *Null* is returned
   (b) the number of parent node is 1, then a result of `create_node` is returned.

DOI: http://dx.doi.org/10.1145/0000000.0000000

```
Input: program: Program in RTL format
/* pp is the set of program points */
let global pp ← ∅;
let M ← main;
let M ← ReachableMethods (main) ∪ {M};
let size ← sizeof (signal class);
foreach m ∈ M do
    let pp ← (size, get in m, program points
    initializing signals);
end
Step2 (program,pp,main);
```

(a) Step1:Scan signal initialization

```
Input: program: Program in RTL format
Input: pp: Set of program points
Input: method: Method to scan for signal
       emissions
let S ← signalSet (pp);
let C ← code (method);
let PPC ← ∅;
foreach c ∈ C do
    if emits (c,S) then
        PPC ← PPC ∪ {c};
    end
end
Step3 (program,PPC,C);
foreach c ∈ C do
    if invokesMethod (c) then
        Step2 (program,c,getInvoked (c));
    end
end
```

(b) Step2: Scan setValue method calls

```
Input: program: Program in RTL format
Input: PPC: Set of program points affected
Input: C: code of the method containing PPC
foreach ppc ∈ PPC do
    if affectsFields (getExpr (ppc)) then
        let pp ← pp ∪ Step4
        (program,ppc,getExpr (ppc),C);
    else if affectsVars (getExpr (ppc)) then
        let pp ← pp ∪ {Step5
        (program,ppc,getExpr (ppc),C)};
    else raise error
end
```

(c) Step3: For every affected field/variable get the object allocation program points

```
Input: program: Program in RTL format
Input: ppc: Affected program point
Input: pexpr: Expression affecting field
Input: C: code of the method containing ppc
let f ← fieldOf (pexr);
let (size,defs) ← Step6 (program,ppc,f,C);
if size ≠ 0 then return (size,defs);
foreach d ∈ defs do
    if !escape (program,C,d) then
        if hasRefFields (f) then
            let PPC ← getAffectedFields
            (getFields (f));
            Step3 (program,PPC,C);
        end
    else raise error;
end
let size ← max (sizeof (defs));
return (size,defs);
```

(d) Step4: Get object allocation program points for affected field

Fig. 15: Data-flow analysis pseudo-algorithm

```
Input: program: Program in RTL format
Input: ppc: Affected program point
Input: pexpr: Expression affecting field
Input: C: code of the method containing ppc
1  let v ← varOf (pexr);
2  let defs ← Step7 (program,ppc,v,C);
3  foreach d ∈ defs do
4      if !escape (program,C,d) then
5          if hasRefFields (v) then
6              let PPC ← getAffectedFields
               (getFields (v));
7              Step3 (program,PPC,C);
8          end
9      else raise error;
10 end
11 let size ← max (sizeof (defs));
12 return (size,defs);
```

(e) Step5: Get object allocation program points for affected variables

```
Input: program: Program in RTL format
Input: ppc: Affected program point
Input: f: Field affected
Input: C: code of the method containing ppc
1  let ofields ← getAllFields (C) \ {f} ;
2  let pcs ← reachDef (ppc,f);
3  if pcs ≠ ∅ then
4      let defs ← ∅;
5      foreach pc ∈ pcs do
6          let vf ← getVarSetting (f,pc);
7          let tt ← Step7 (program,pc,vf,C);
8          if all other field ∈ ofields have same tt
           then
               /* Concat in set defs          */
9              defs ← defs ∪ tt;
10         else /* Append a new set in defs    */
11             defs ← defs ∪ {tt};
12     end
13     return (0,defs);
14 else
       /* Inter-procedural analysis          */
       /* Get all callers, including from
       AGRC analysis                         */
15     let Callers ← getCallers (C);
16     if Callers = ∅ then raise Not_init f;
17     let defs ← ∅;
18     let sizes ← ∅;
19     foreach caller ∈ Callers do
20         let defs ← defs ∪ Step6
           (program,getCallerPPC (C),f,caller);
           /* Similar to Figure 15d
           lines 4-12, to fill in the sizes
           set.                              */
21     end
22     let size ← max(sizes);
23     return (size,defs);
24 end
```

(f) Step6: Search object allocation program points for affected field

```
Input: program: Program in RTL format
Input: ppc: Affected program point
Input: v: Variable affected
Input: C: code of the method containing ppc
1  let pcs ← reachDef (ppc,v);
2  if pcs = ∅ then raise Not_init;
3  let defs ← ∅;
4  foreach pc ∈ pcs do
5      if getExpr (pc) is New or getExpr (pc) is
       NewArray then
6          defs ← defs ∪ {pc};
7      else if getExpr (pc) is AffectField then
8          let expr ← getExpr (pc);
9          let (_,defs) ← Step4
           (program,pc,expr,C);
10     else if getExpr (pc) is AffectVar then
11         let v ← varOf (getExpr (pc));
12         let (_,defs) ← Step7 (program,pc,v,C);
13     else
           /* Cannot handle new outside
           current method allocation          */
14         raise Not_handled
15     end
16 end
   /* Make sure that definitions are not
   reachable from each other                  */
17 if |defs| > 1 then
18     assert(notReachable (defs));
19 end
20 return (defs);
```

(g) Step7: Search object allocation point(s) for method local affected variable

Fig. 15: Data-flow analysis pseudo-algorithm

```
Input: AGRC : (V, E)
Output: mList: a set of set of mNode
Data: V_cd: a set of set of clock-domain nodes
Data: mNode = [Name:string, Must:mNode, May:set of mNode]
/* Grouping nodes for each clock-domain                                   */
let V_r = ∅ ∪ V
let V_cd = ∅
foreach v ∈ V do
    foreach (v_i, v_j) ∈ E do if v_j = v then V_r = V_r \ {v}
foreach v ∈ V_r do V_cd = V_cd ∪ {collect_cd_nodes(v, E)}
/* Perform May and Must analysis                                          */
forall the V ∈ V_cd do
    let V_temp = ∅
    foreach v ∈ V do
        V_temp = V_temp ∪ {create_node(v, E)}
    mList = mList ∪ {V_temp}

Function create_node(v, E):
begin
    let E_i = λ(v)
    if v = ActionNode then
        if |E_i| = 1 then
            let {(v_i, v_j)} = E_i
            let newNode = [Name ← get_name(v_j); Must ← create_node(v_i, E); May ← Null]
            return newNode
        else if |E_i| > 1 then
            let V_temp = ∅
            foreach (v_i, v_j) ∈ E_i do
                V_temp = V_temp ∪ {create_node(v_i, E)}
            return [Name ← get_name(v_j); Must ← Null; May ← V_temp]
        else if |E_i| = 0 then
            return [Name ← get_name(v_j); Must ← Null; May ← Null]
    else if v = JoinNode then
        let v = find_matching_fork(v)
        let {(v_i, v_j)} = λ(v)
        return create_node(v_i, E)
    else
        if |E_i| = 0 then
            return Null
        else if |E_i| = 1 then
            return create_node(v_i, E)

Function collect_cd_nodes(v, E):
begin
    let V_temp = ∅
    foreach (v_i, v_j) ∈ E do
        if v = v_i then
            V_temp = V_temp ∪ {v_j}
            V_temp = V_temp ∪ collect_cd_nodes(v_j, E)
    return V_temp
```

ALGORITHM 1: May and Must analysis