From Bottlenecks to Breakthroughs : Scaling Pointer Analysis using Coalescing Algorithm

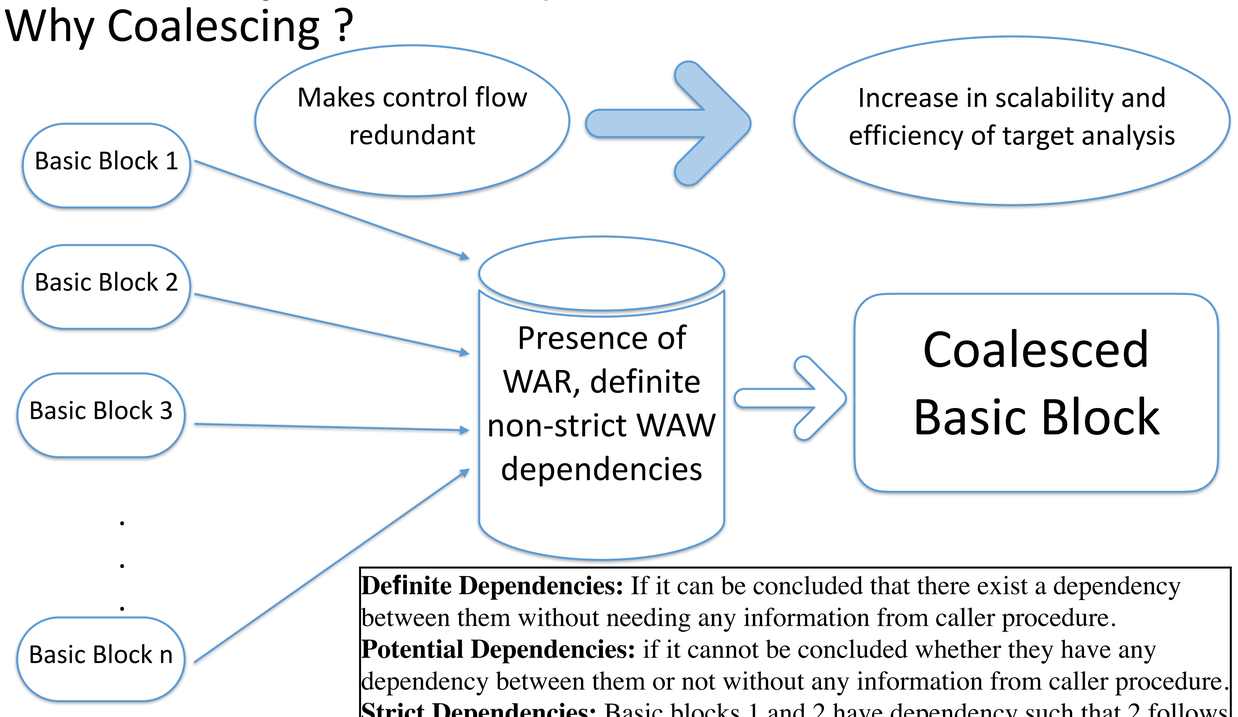
Priyanshi Gupta
Prof. Uday Khedker
Department of Computer Science and Engineering, IIT Bombay

IIT Bombay

A fundamental challenge : Scalability vs. Precision

A major tradeoff in pointer analysis is achieving both **scalability** (handling large programs efficiently) and **precision** (minimizing false positives and false negatives) as improving one often degrades the other. Flow-insensitive pointer analysis **ignores the order of execution** of statements and computes a **single, program-wide points-to set**. While this improves **scalability**, it also introduces **redundant control flow**, affecting both **soundness** and **precision.**

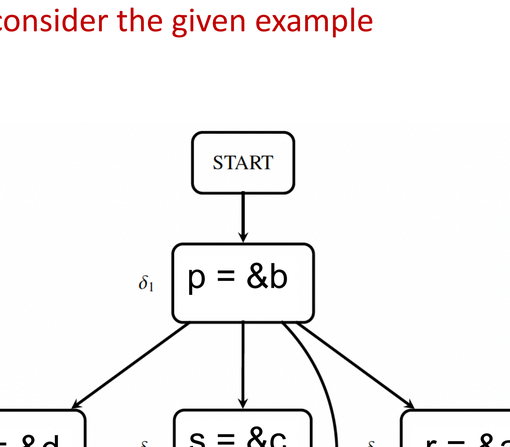
Why Coalescing ?



Definite Dependencies: If it can be concluded that there exist a dependency between them without needing any information from caller procedure.
Potential Dependencies: if it cannot be concluded whether they have any dependency between them or not without any information from caller procedure.
Strict Dependencies: Basic blocks 1 and 2 have dependency such that 2 follows 1 along every control path from 1 to 2.
Non-Strict Dependencies: Basic blocks 1 and 2 has dependency such that 2 follows 1 along some control-flow path from 1 to 2.

Conservative Nature of Data-flow equation

The data-dependency equation which is used checks for potential RaW and WaW dependency. But it doesn't distinguish betweenWaW dependencies which are strict and which are non-strict. But, Coalescing in presence of WaW dependency which are non-strict should be allowed.



The Basic Block 1 and 2 have a potential non-strict WaW dependency. They are not coalesced due to this potential dependency. But coalescing the two basic blocks would be sound and precise because non-strict WaW dependency are preserved by the control flow of basic block semantic.

Confluence Operator as AND

Primarily, a basic block can only be coalesced with one of its successor/predecessor if it can be coalesced with all its successors / predecessors

But, consider the given example

Due to a potential WaW dependency between δ1 and δ4 they cannot be coalesced. But basic blocks δ1, δ2, δ3 and δ5 can all be coalesced together as they have no potential data dependency between them. But due to confluence being AND, this coalescing of basic blocks cannot be done due to potential data dependency between δ1 and δ4.

Changes in Data-flow Equations

$$\text{gpuFlow(m, n)} = \begin{cases} \phi & \neg\ \text{ColIn} \ \vee\ \text{DDep}(\text{GpuOut}_m, \delta_n) \\ \text{GpuOut}_m & \text{otherwise} \end{cases}$$

Current data-flow equations checks for Potential RaW or Potential WaW dependency between the pointer statements of the two basic blocks. Using the above equation gave incorrect result for some cases, which would result in missing out on possible chance of coalescing of two basic blocks.

Updated data-flow equation:

$$\text{DDep}(X, Y) \iff \big(\text{TIndierctDef}(Y) \cup \text{TIndirectRef}(Y)\big) \cap \text{TDirectDef}(X - Y) \neq \phi$$
$$\vee\big(\text{TRef}(Y) \cup \text{TDef}(Y)\big) \cap \text{TIndirectDef}(X - Y) \neq \phi$$

TDef: is used to identify the types of set of locations being defined
TRef : is used to identify the types of set of locations being read.

TDef and TRef are split further to identify the set of locations which are being defined directly and indirectly similarly, set of locations being read directly and indirectly.
Above equation gives correct result for checking data dependency.

Updated Coalescing Algorithm

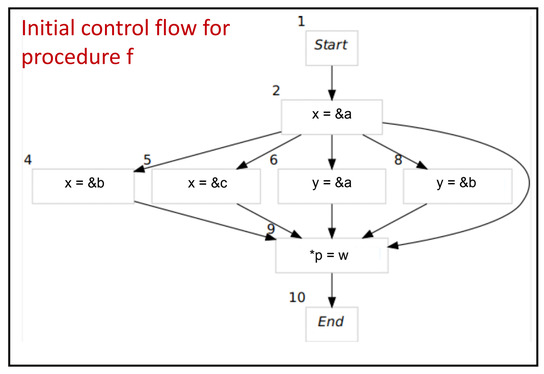
Changes made :

- **store a map for every basic block which stores whether there exist a potential dependency between the pair of basic blocks.**
- **Used an updated equation to check for potential dependency between pointer statements.**
- **An updated algorithm which uses BFS which explicitly creates partition of basic blocks.**
- **Accounting of information which stores the dependency relation between the current basic block and all other basic blocks.**

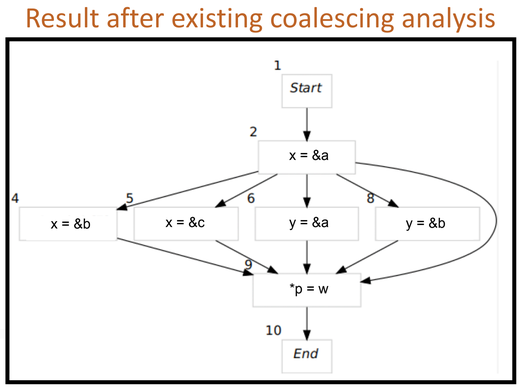
Example :

```
// int *x, *y, *w, **p, **q, a, b, c;
void f()
{
   x = &a;
  switch(a)
  {
     case 1:
        x = &b;
        break;

     case 2:
        x = &c;
        break;

     case 3:
        y = &a;
        break;

     case 5:
        break;

     default :
        y = &b;
        break;
  }
  *p = w;
}
```

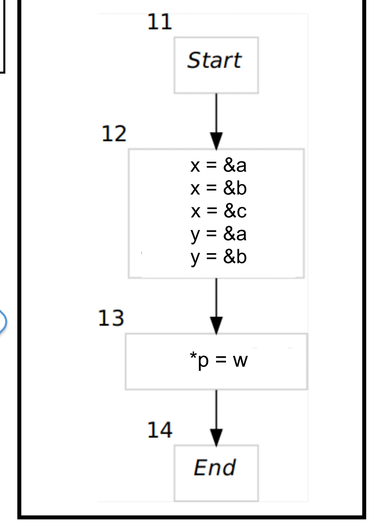
Initial control flow for procedure f



Result after existing coalescing analysis



Result after updated coalescing analysis



Target Analysis

Apart from points to analysis, the updated coalescing algorithm can be used in various other fields of optimisation in terms of scalability. Some of them are :

1: **Alias Analysis** - Distinguishing between must-alias and may-alias
2: **Call Graph Analysis** - Coalescing similar function call contexts to reduce the number of distinct analysis states.
3: **Heap and Memory Allocation Analysis** - Coalescing objects allocated at similar sites when their differentiation is not critical.