Dominator Trees

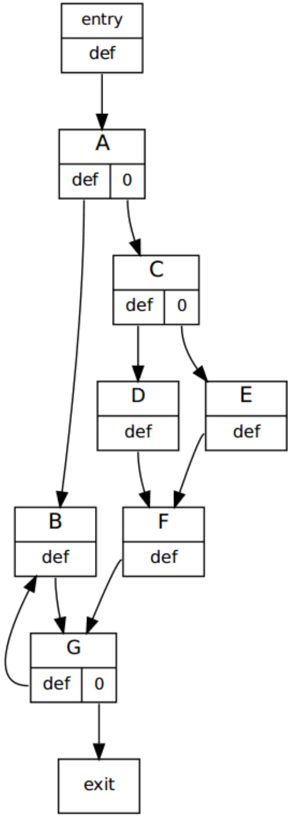
and incremental updates that transcend time

Jakub (Kuba) Kuderski
kubakuderski@gmail.com
University of Waterloo

2017 US LLVM Developers' Meeting – San Jose

Introduction

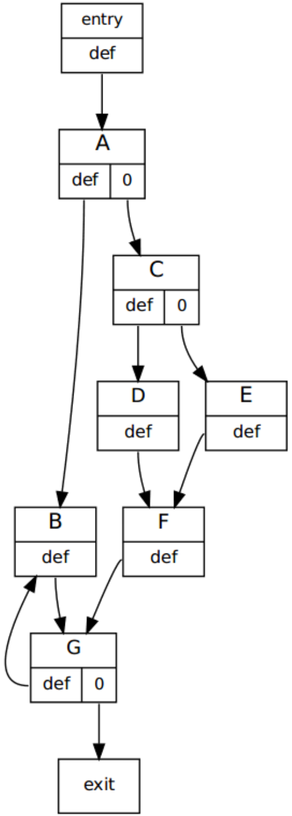
CFG (Control Flow Graph)



Introduction

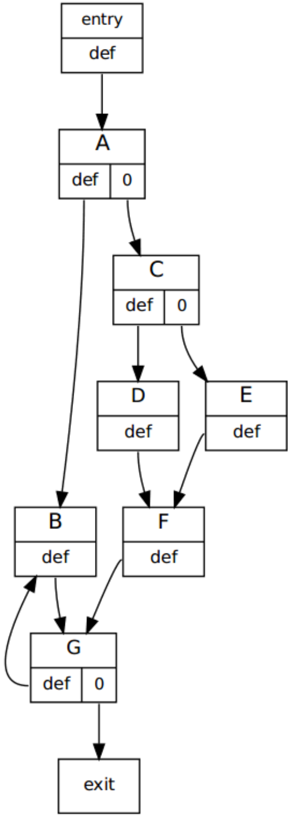
Dominance:

CFG (Control Flow Graph)



Introduction

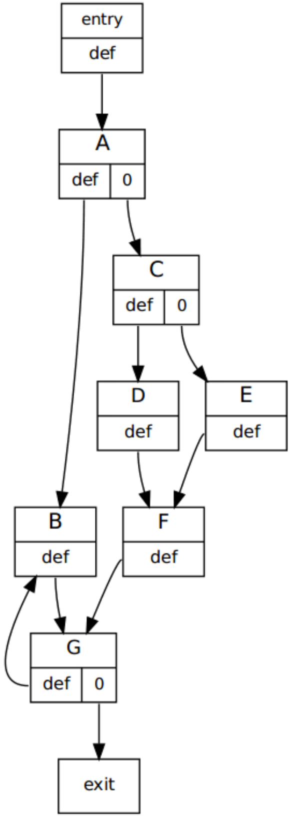
CFG (Control Flow Graph)



Dominance:

Node X dominates node Y iff all paths from the entry to Y go through X.

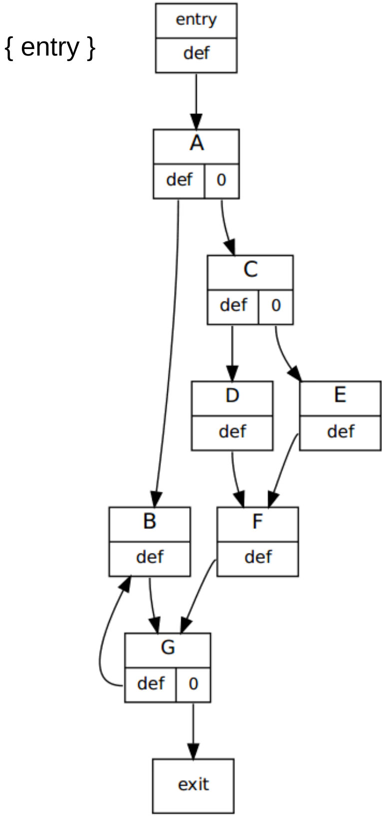
Dominators



Dominance:

Node X dominates node Y iff all paths from the entry to Y go through X.

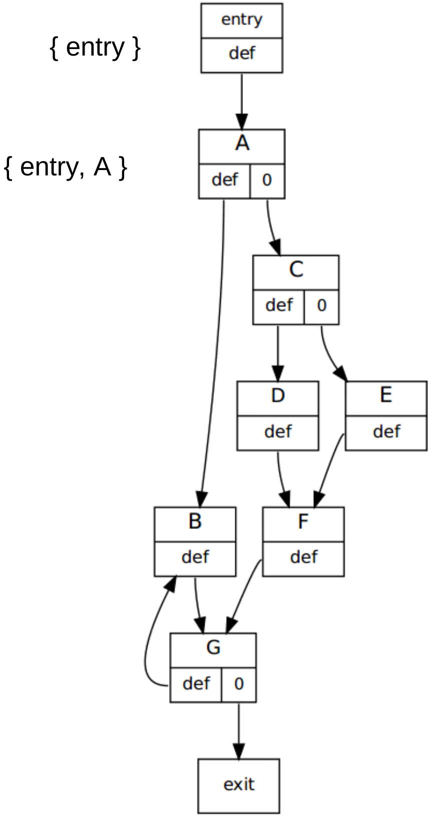
Dominators



Dominance:

Node X dominates node Y iff all paths from the entry to Y go through X.

Dominators



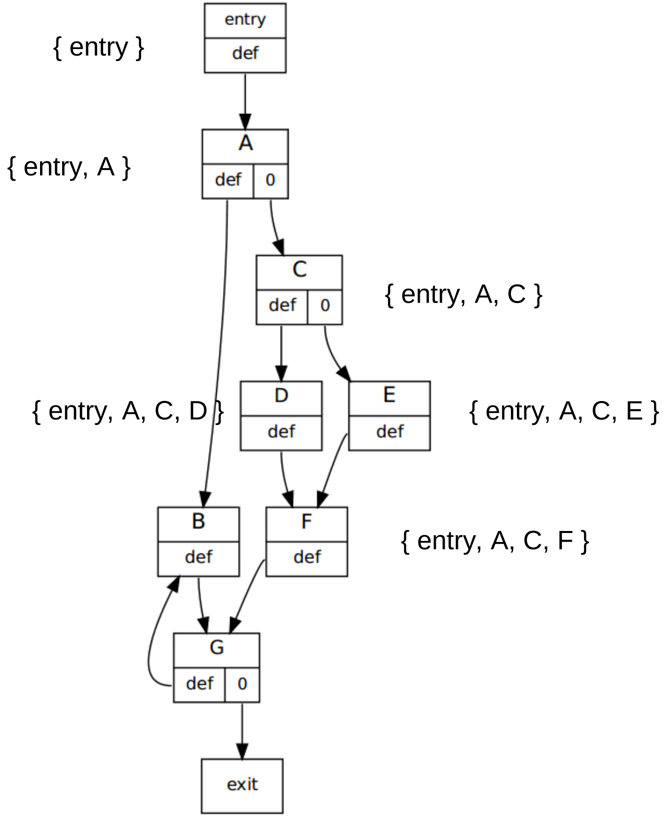
Dominance:

Node X dominates node Y iff all paths from the entry to Y go through X.

Dominators

Dominance:

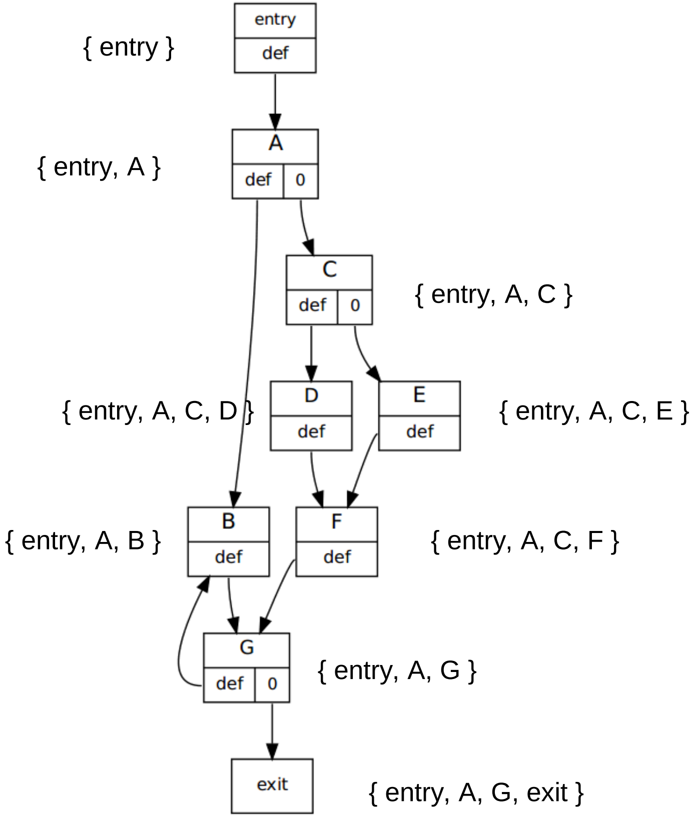
Node X dominates node Y iff all paths from the entry to Y go through X.



Dominators

Dominance:

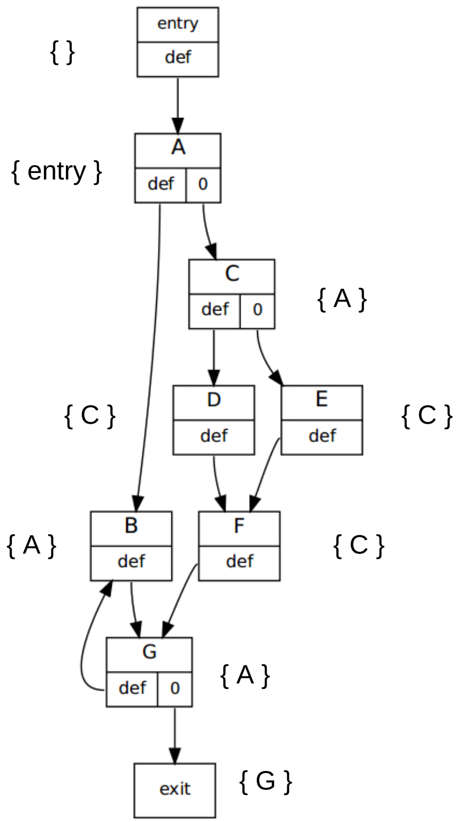
Node X dominates node Y iff all paths from the entry to Y go through X.



Dominance:

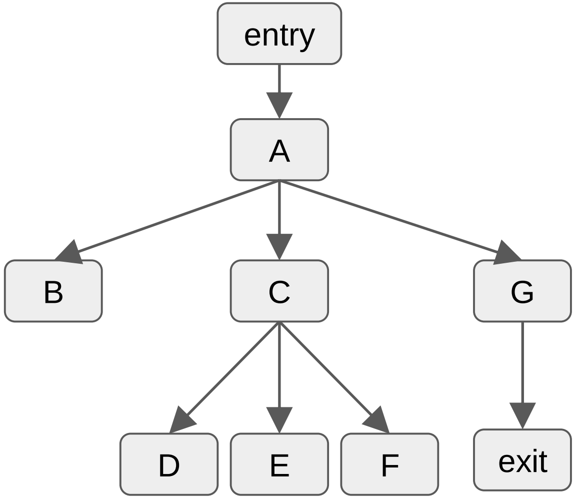
Node X dominates node Y iff all paths from the entry to Y go through X.

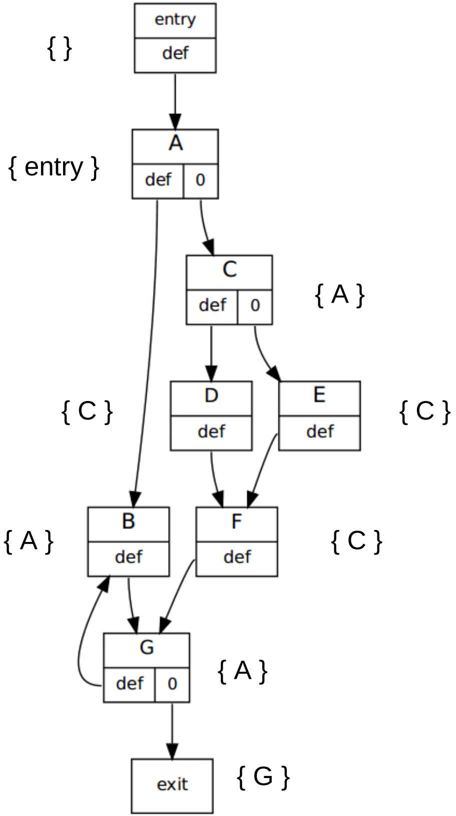
Immediate dominators



Immediate dominators

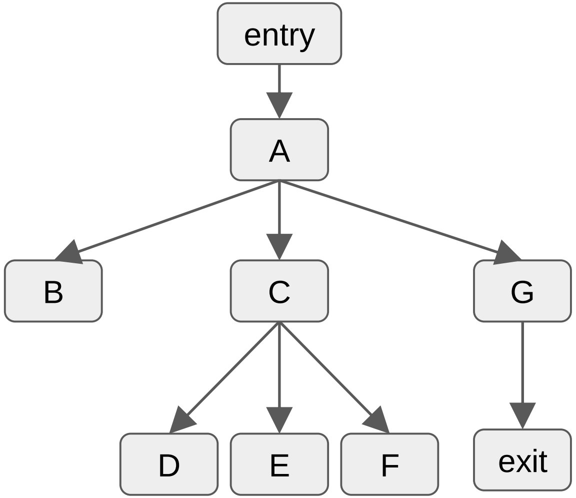
Dominator Tree:

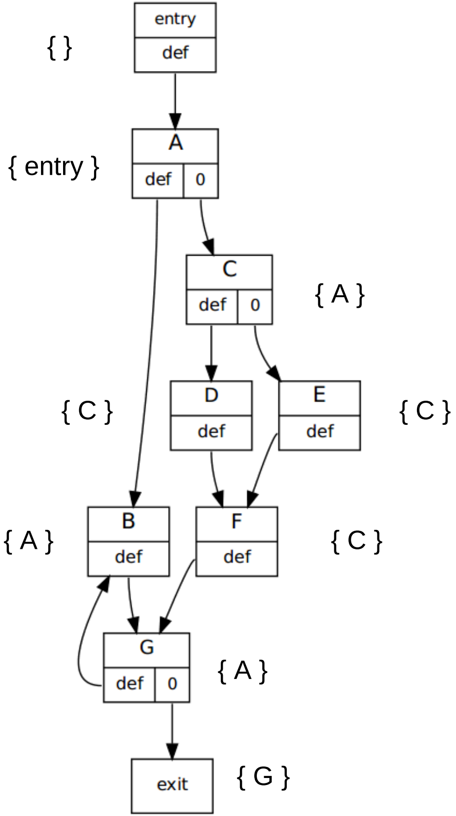




Immediate dominators

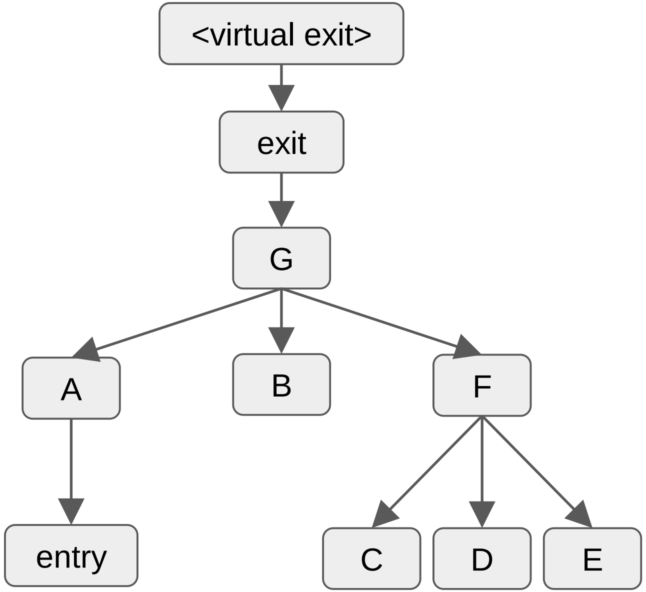
Tree **T** is the **dominator tree** if and only if
it has the **parent** and the **sibling** properties.

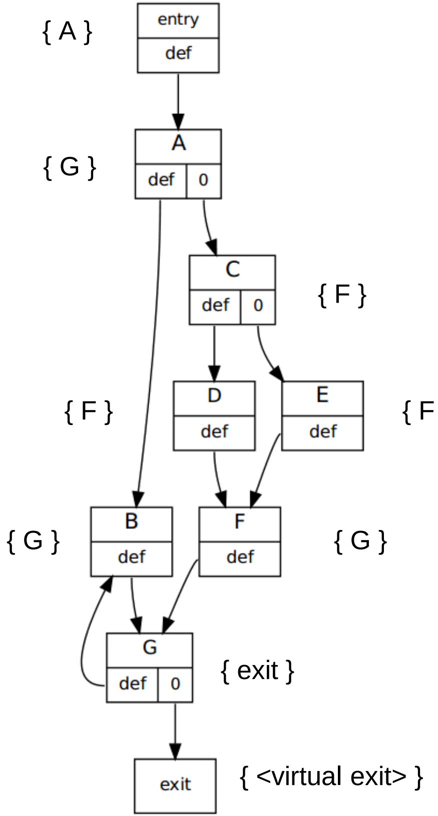




Immediate postdominators

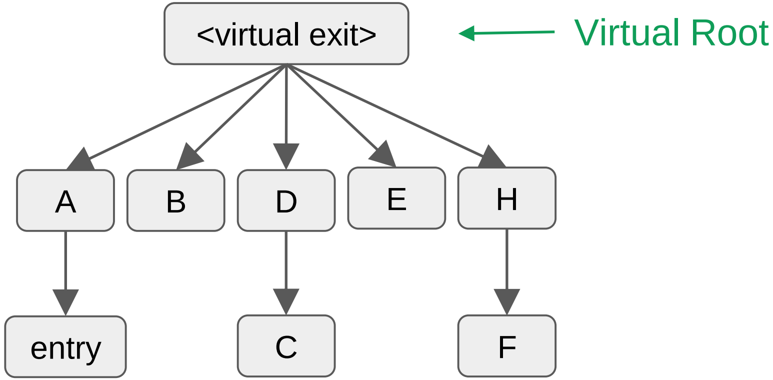
Postdominator Tree:



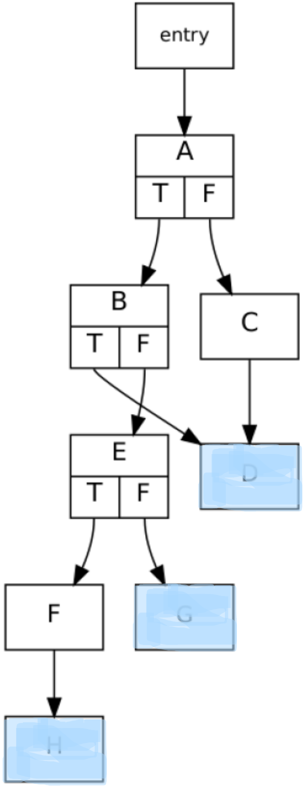


Multiple exits: D, G, H

Postdominator Tree:

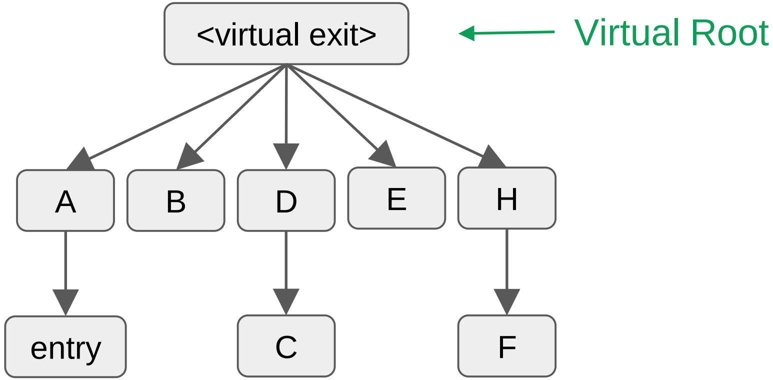


Roots: D, G, H

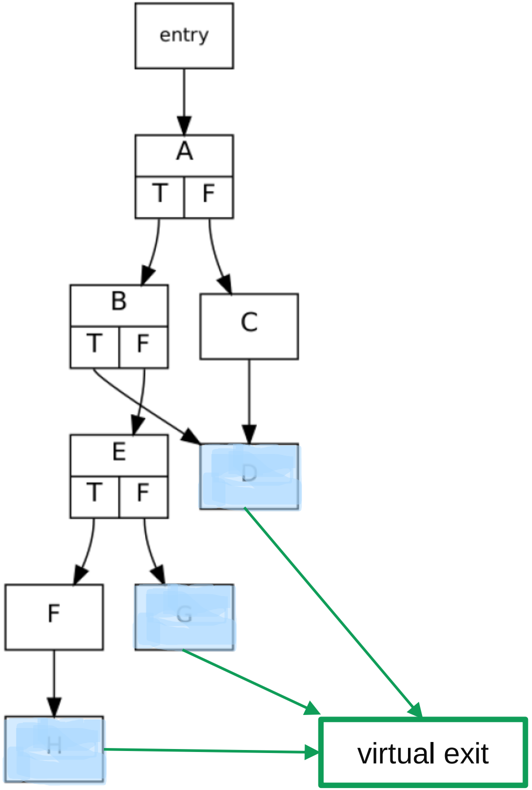


Multiple exits: D, G, H

Postdominator Tree:

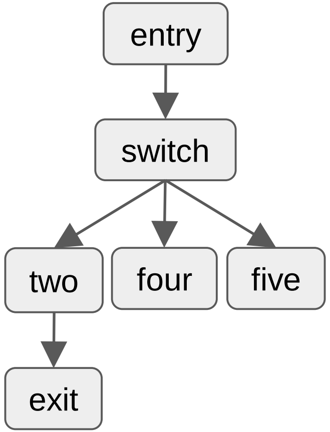


Roots: D, G, H



`(nullptr)`

Dominator Tree



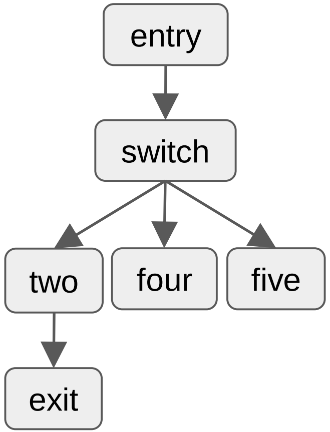
Textual representation
(for debugging)

```
Inorder Dominator Tree: DFSNumbers invalid: 0 slow queries.
  [1] %entry {4294967295,4294967295} [0]
    [2] %switch {4294967295,4294967295} [1]
      [3] %five {4294967295,4294967295} [2]
      [3] %two {4294967295,4294967295} [2]
        [4] %exit {4294967295,4294967295} [3]
      [3] %four {4294967295,4294967295} [2]
```

calculated level

level stored in the tree node

Dominator Tree



Textual representation
(for debugging)

DFS In/Out numbers – calculated lazily

```
Inorder Dominator Tree:
  [1] %entry {1,12} [0]
    [2] %switch {2,11} [1]
      [3] %five {3,4} [2]
      [3] %two {5,8} [2]
        [4] %exit {6,7} [3]
      [3] %four {9,10} [2]
```

calculated level

level stored in the tree node

Dominators are important in SSA

- Every def must dominate its uses
  - ... in a valid piece of IR
- Dominators are used to compute the optimal placement of PHI nodes
  - DominanceFrontier

Use of dominators in LLVM

- Used with BasicBlocks
  - DominatorTree, PostDominatorTree
  - DominatorTreeWrapperPass, PostDominatorTreeWrapperPass
  - DominanceFrontier, IteratedDominanceFrontier
- Also with MachineBasicBlocks and Clang's CFG

Use of dominators in LLVM

- `grep -r 'Dominator'`
  - ?
- `grep -r 'Dominance'`
  - ?
- `grep -r 'dominates'`
  - ?
- `grep -rE 'DT\.|DT->'` `DT.` and `DT->`
  - ?

Use of dominators in LLVM

- `grep -r 'Dominator'`
  - 2600
- `grep -r 'Dominance'`
  - 320
- `grep -r 'dominates'`
  - 660
- `grep -rE 'DT\.|DT->'` `DT.` and `DT->`
  - 1200

Problems

- There was no API for automatically updating the DominatorTree
  - Very low-level API for performing manual updates
  - Frequent DominatorTree recalculations
    (1 million recalculations when optimizing clang fullLTO, ~3.2% of total optimization time)
- PostDominatorTree was virtually impossible to update manually
  - Too costly to maintain
  - Not used widely in practice

Goals

- Make updating the DominatorTree easy
    - To get rid of numerous extremely subtle bugs scattered across the whole optimizer
    - Reduce the number of recalculations
- Make the PostDominatorTree more viable to use
    - By making it possible to update it without doing full recalculations

Incremental dominator tree updater

- Depth Based Search algorithm
  - Uses Semi-NCA tree construction algorithm
  - Splits updates into 4 categories and tries to bound the search of affected subtrees using tree level information

arXiv:1604.02711v1 [cs.DS] 10 Apr 2016

An Experimental Study of Dynamic Dominators*

Loukas Georgiadis[1] Giuseppe F. Italiano[2] Luigi Laura[3] Federico Santaroni[2]

April 12, 2016

Abstract

Motivated by recent applications of dominator computations, we consider the problem of dynamically maintaining the dominators of flow graphs through a sequence of insertions and deletions of edges. Our main theoretical contribution is a simple incremental algorithm that maintains the dominator tree of a flow graph with n vertices through a sequence of k edge insertions in $O(m \min\{n,k\} + kn)$ time, where m is the total number of edges after all insertions. Moreover, we can test in constant time if a vertex u dominates a vertex v, for any pair of query vertices u and v. Next, we present a new decremental algorithm to update a dominator tree through a sequence of edge deletions. Although our new decremental algorithm is not asymptotically faster than repeated applications of a static algorithm, i.e., it runs in $O(mk)$ time for k edge deletions, it performs well in practice. By combining our new incremental and decremental algorithms we obtain a fully dynamic algorithm that maintains the dominator tree through intermixed sequence of insertions and deletions of edges. Finally, we present efficient implementations of our new algorithms as well as of existing algorithms, and conduct an extensive experimental study on real-world graphs taken from a variety of application areas.

1 Introduction

A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex $s \in V$. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no such path exists. The *dominator relation* on G is defined for the set of reachable vertices as follows. A vertex w *dominates* a vertex v if every path from s to v includes w. We let $Dom(v)$ denote the set of all vertices that dominate v. If v is reachable then $Dom(v) \supseteq \{s, v\}$; otherwise $Dom(v) = \emptyset$. For a reachable vertex v, s and v are its *trivial dominators*. A vertex $w \in Dom(v) - v$ is a *proper dominator* of v. The *immediate dominator* of a vertex $v \neq s$, denoted $d(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all vertices in $Dom(v) - v$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* D: u dominates w if and only if u is an ancestor of w in D. To form D, we make each reachable vertex $v \neq s$ a child of its immediate dominator.

The problem of finding dominators has been extensively studied, as it occurs in several applications. The dominator tree is a central tool in program optimization and code generation [12].

[1]Department of Computer Science & Engineering, University of Ioannina, Greece. E-mail: loukas@cs.uoi.gr.
[2]Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata", Roma, Italy. E-mail: giuseppe.italiano@uniroma2.it, santaroni@uniroma2.it.
[3]Dipartimento di Ingegneria Informatica, Automatica e Gestionale e Centro di Ricerca per il Trasporto e la Logistica (CTL), "Sapienza" Università di Roma, Roma, Italy. E-mail: laura@dis.uniroma1.it.
*A preliminary version of this paper appeared in the *Proceedings of the 20th Annual European Symposium on Algorithms*, pages 491–502, 2012.

1

L. Georgiadis et al.
https://arxiv.org/pdf/1604.02711.pdf

Incremental dominator tree updater

- Depth Based Search algorithm
  - Uses Semi-NCA tree construction algorithm
  - Splits updates into 4 categories and tries to bound the search of affected subtrees using tree level information
- What we have done:
  - Cleaned up existing implementation of the DominatorTree
  - Switched from Simple Lengauer-Tarjan to Semi-NCA
  - Adapted the Depth Based Search algorithm to LLVM
  - Made improvements to the PostDominatorTree

arXiv:1604.02711v1 [cs.DS] 10 Apr 2016

An Experimental Study of Dynamic Dominators*

Loukas Georgiadis[1] Giuseppe F. Italiano[2] Luigi Laura[3] Federico Santaroni[2]

April 12, 2016

Abstract

Motivated by recent applications of dominator computations, we consider the problem of dynamically maintaining the dominators of flow graphs through a sequence of insertions and deletions of edges. Our main theoretical contribution is a simple incremental algorithm that maintains the dominator tree of a flow graph with n vertices through a sequence of k edge insertions in $O(m \min\{n, k\} + kn)$ time, where m is the total number of edges after all insertions. Moreover, we can test in constant time if a vertex u dominates a vertex v, for any pair of query vertices u and v. Next, we present a new decremental algorithm to update a dominator tree through a sequence of edge deletions. Although our new decremental algorithm is not asymptotically faster than repeated applications of a static algorithm, i.e., it runs in $O(mk)$ time for k edge deletions, it performs well in practice. By combining our new incremental and decremental algorithms we obtain a fully dynamic algorithm that maintains the dominator tree through intermixed sequence of insertions and deletions of edges. Finally, we present efficient implementations of our new algorithms as well as of existing algorithms, and conduct an extensive experimental study on real-world graphs taken from a variety of application areas.

1 Introduction

A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex $s \in V$. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no such path exists. The *dominator relation* on G is defined for the set of reachable vertices as follows. A vertex w *dominates* a vertex v if every path from s to v includes w. We let $Dom(v)$ denote the set of all vertices that dominate v. If v is reachable then $Dom(v) \supseteq \{s, v\}$; otherwise $Dom(v) = \emptyset$. For a reachable vertex v, s and v are its *trivial dominators*. A vertex $w \in Dom(v) - v$ is a *proper dominator* of v. The *immediate dominator* of a vertex $v \neq s$, denoted $d(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all vertices in $Dom(v) - v$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* D: u dominates w if and only if u is an ancestor of w in D. To form D, we make each reachable vertex $v \neq s$ a child of its immediate dominator.

The problem of finding dominators has been extensively studied, as it occurs in several applications. The dominator tree is a central tool in program optimization and code generation [12].

[1]Department of Computer Science & Engineering, University of Ioannina, Greece. E-mail: loukas@cs.uoi.gr.
[2]Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata", Roma, Italy. E-mail: giuseppe.italiano@uniroma2.it, santaroni@uniroma2.it.
[3]Dipartimento di Ingegneria Informatica, Automatica e Gestionale e Centro di Ricerca per il Trasporto e la Logistica (CTL), "Sapienza" Università di Roma, Roma, Italy. E-mail: laura@dis.uniroma1.it.
*A preliminary version of this paper appeared in the *Proceedings of the 20th Annual European Symposium on Algorithms*, pages 491–502, 2012.

1

L. Georgiadis et al.
https://arxiv.org/pdf/1604.02711.pdf

Semi-NCA dominator tree construction algorithm

- Simpler to implement than Simple Lengauer-Tarjan
  - Does not perform path compression
  - Stores levels (depth in tree) in nodes
- Worse computational complexity, but faster in practice
  - Simple Lengauer-Tarjan – $O(n \log(n))$
  - Semi-NCA – $O(n^2)$

Semi-NCA dominator tree construction algorithm

- Simpler to implement than Simple Lengauer-Tarjan
  - Does not perform path compression
  - Stores levels (depth in tree) in nodes
- Worse computational complexity, but faster in practice
  - Simple Lengauer-Tarjan – O(n log(n))
  - Semi-NCA – $O(n^2)$

Delta (%)

Project	SLT (release)	Semi-NCA (release)	SLT (debug)	Semi-NCA (debug)
Clang fullLTO	0	-6.3	0	0.8
Sqlite	0	-9.7	0	-18.2
Oggenc	0	-18.7	0	-1.7

Incremental update API

- Two new functions:
  - `DT.insertEdge(From, To)`
  - `DT.deleteEdge(From, To)`
- Following transforms taught to use the new API and preserve dominators:
  - Loop Deletion
  - Loop Rerolling
  - Loop Unswitching
  - Break Critical Edges
  - Aggressive Dead Code Elimination

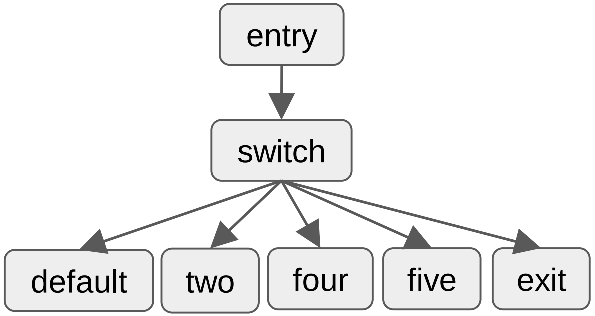
Incremental update API

- Two new functions:
  - `DT.insertEdge(From, To)`
  - `DT.deleteEdge(From, To)`
- Following transforms taught to use the new API and preserve dominators:
  - ~~Loop Deletion~~
  - ~~Loop Rerolling~~
  - ~~Loop Unswitching~~
  - ~~Break Critical Edges~~
  - ~~Aggressive Dead Code Elimination~~

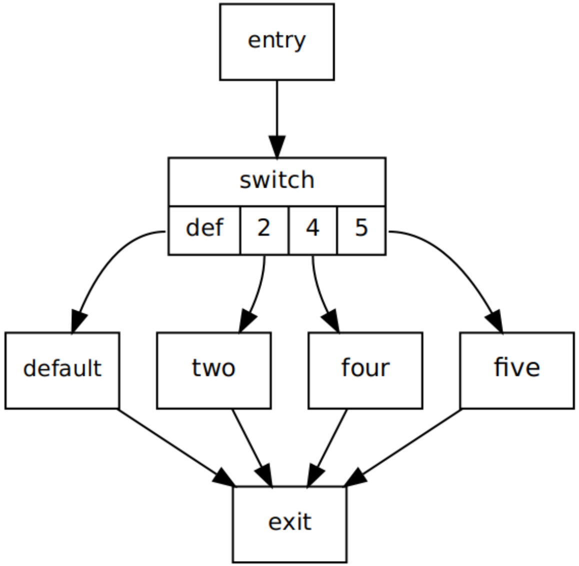
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five
```

Dominator Tree



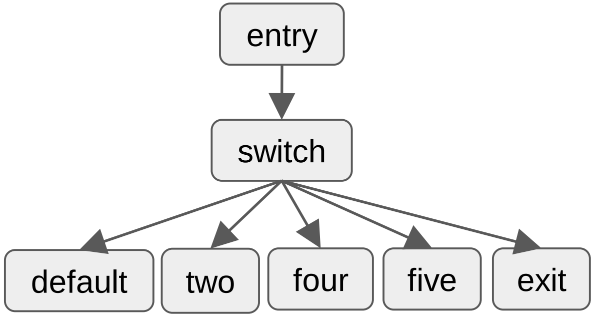
CFG



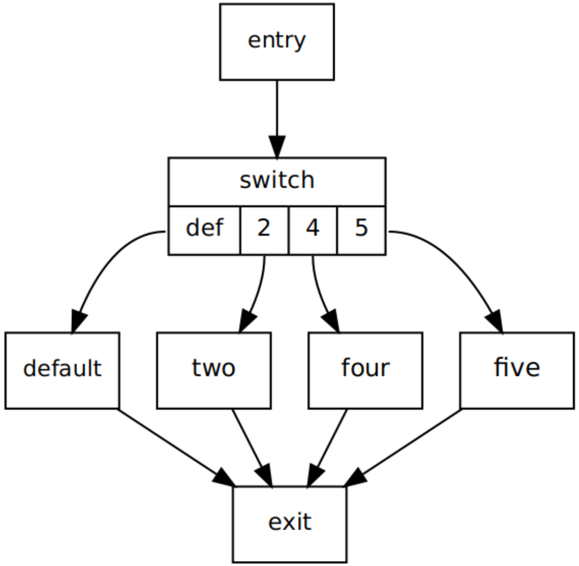
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch
```

Dominator Tree



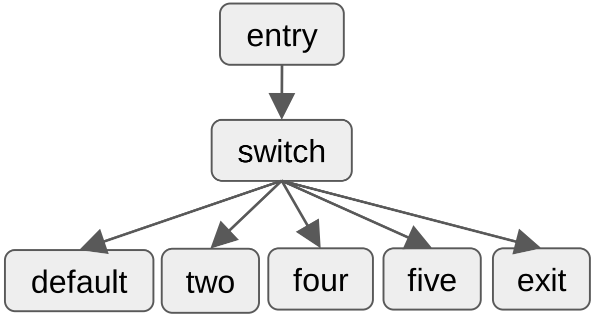
CFG



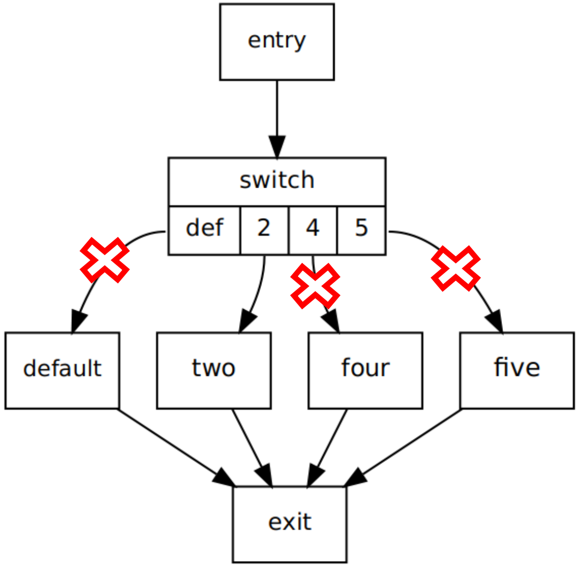
Depth Based Search confused

0. [ADCE] final dead block: %default, %two, %four, %five
1. [ADCE] make %two the only successor of %switch

Dominator Tree



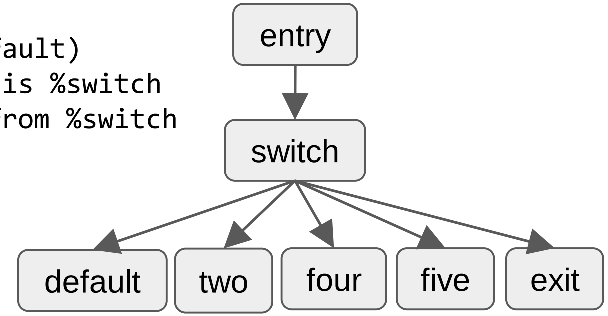
CFG



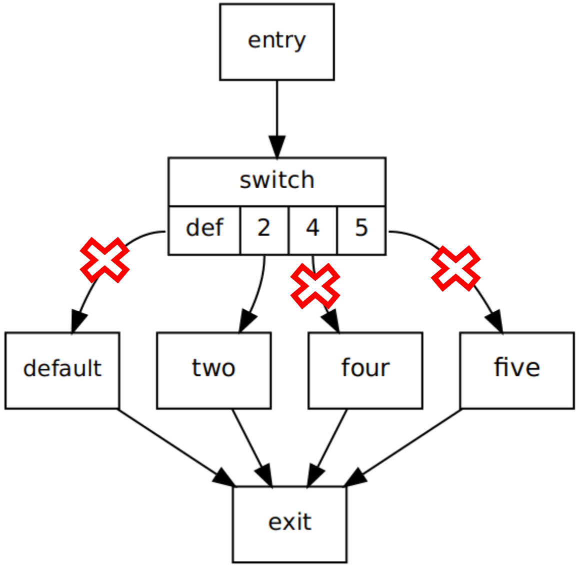
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
```

Dominator Tree



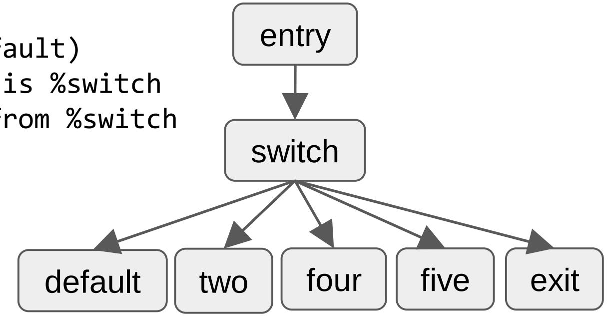
CFG



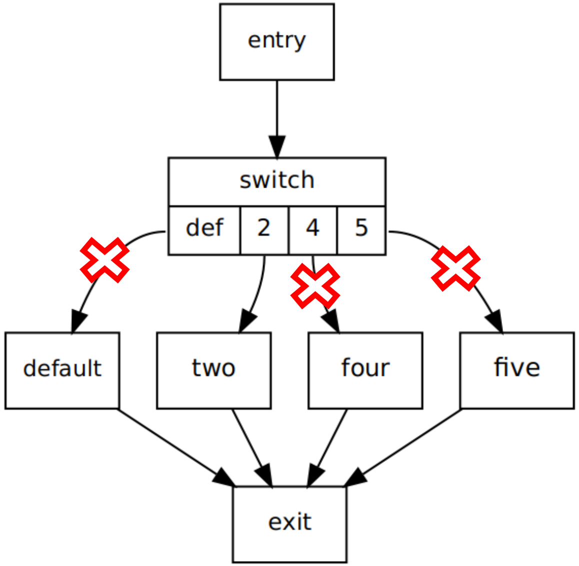
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two
```

Dominator Tree



CFG



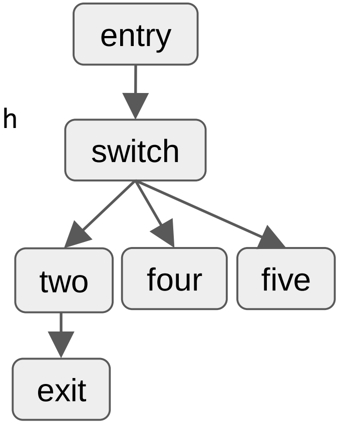
Depth Based Search confused

0. [ADCE] final dead block:
   %default, %two, %four, %five

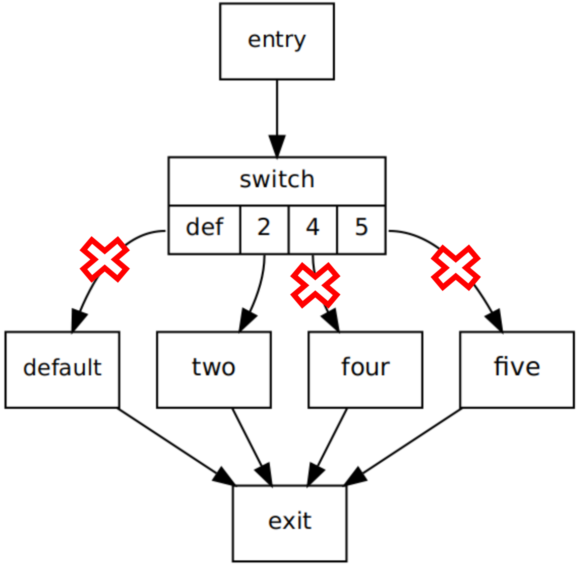
1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two

Dominator Tree



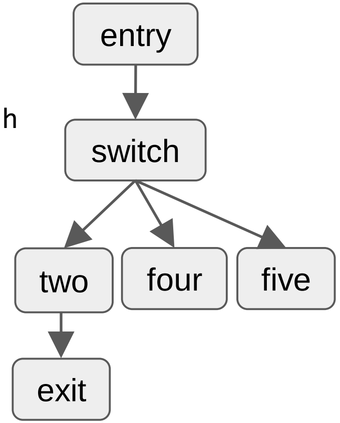
CFG



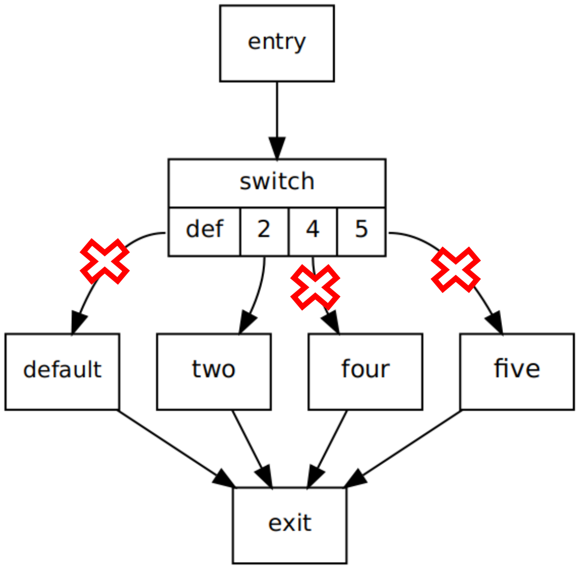
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two
3. [ADCE] DT.deleteEdge(%switch, %four)
   [DT] NCD(%switch, IDom(%four)) is %switch
   [DT] %four was only reachable from %switch
   [DT] delete subtree %four
```

Dominator Tree



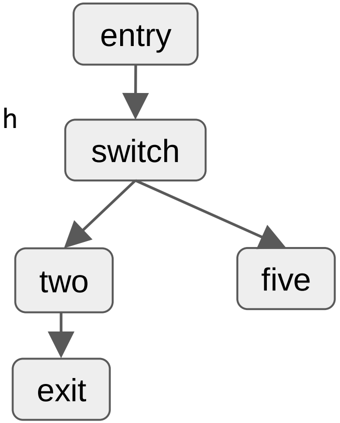
CFG



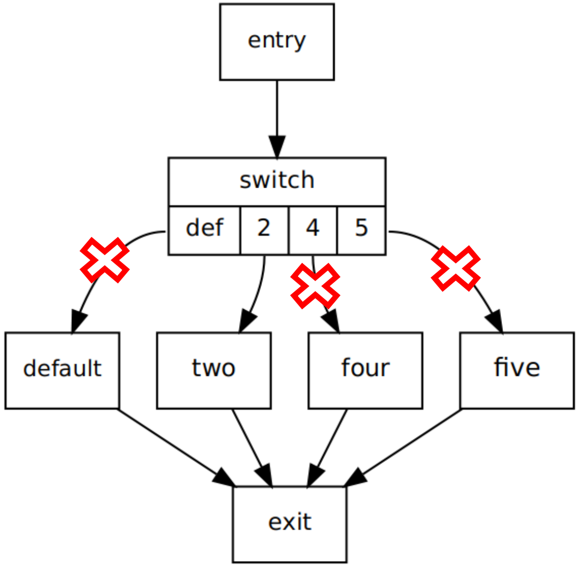
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two
3. [ADCE] DT.deleteEdge(%switch, %four)
   [DT] NCD(%switch, IDom(%four)) is %switch
   [DT] %four was only reachable from %switch
   [DT] delete subtree %four
```

Dominator Tree



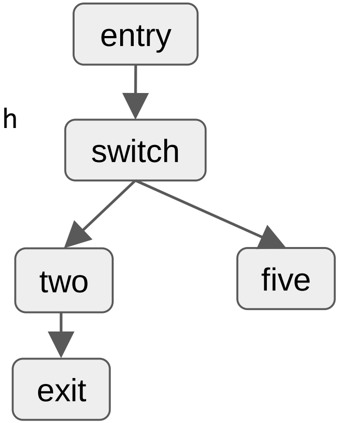
CFG



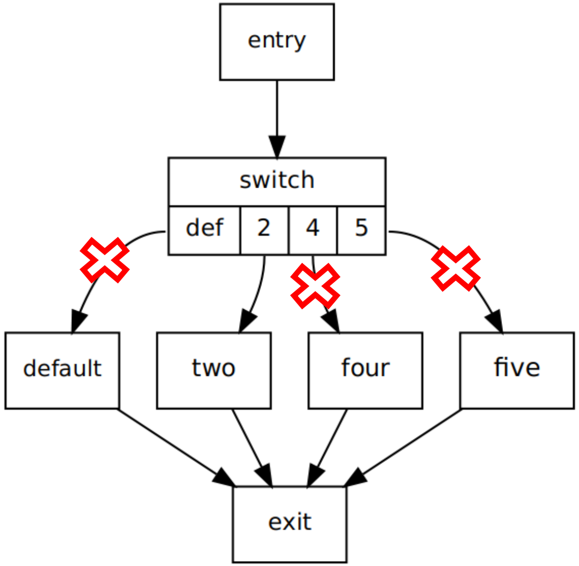
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two
3. [ADCE] DT.deleteEdge(%switch, %four)
   [DT] NCD(%switch, IDom(%four)) is %switch
   [DT] %four was only reachable from %switch
   [DT] delete subtree %four
   [DT] %exit is %four's successor and
        Level(%exit) == Level(%four) + 1,
        so it must be in %four's subtree
```

Dominator Tree



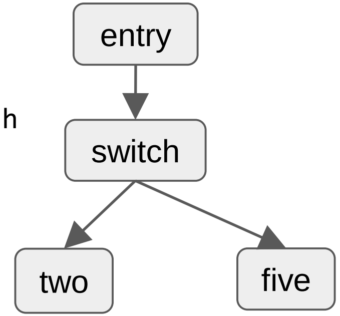
CFG



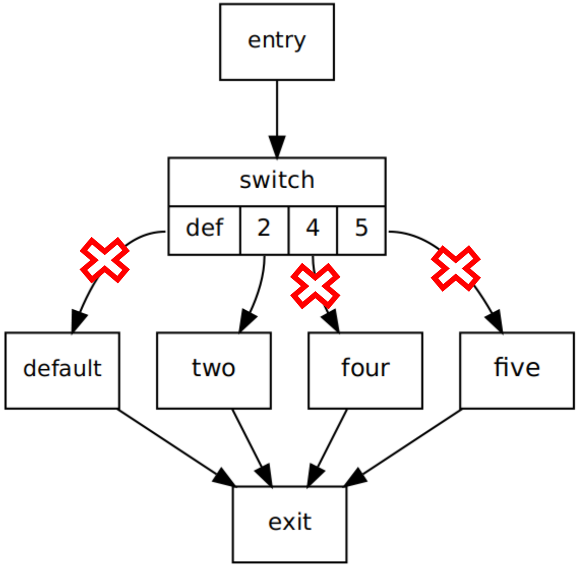
Depth Based Search confused

```
0. [ADCE] final dead block:
   %default, %two, %four, %five

1. [ADCE] make %two the only
   successor of %switch

2. [ADCE] DT.deleteEdge(%switch, %default)
   [DT] NCD(%switch, IDom(%default)) is %switch
   [DT] %default was only reachable from %switch
   [DT] delete subtree %default
   [DT] attach %exit to its only
        predecessor reachable from
        %switch – to %two
3. [ADCE] DT.deleteEdge(%switch, %four)
   [DT] NCD(%switch, IDom(%four)) is %switch
   [DT] %four was only reachable from %switch
   [DT] delete subtree %four
   [DT] %exit is %four's successor and
        Level(%exit) == Level(%four) + 1,
        so it must be in %four's subtree
   [DT] delete %exit
```

Dominator Tree



CFG



Batch updates

- Depth Based Search needs to see a snapshot of the CFG just after each update
- We do not want to store different versions of the same CFG in DominatorTree

Batch updates

- Depth Based Search needs to see a snapshot of the CFG just after each update
- We do not want to store different versions of the same CFG in DominatorTree
- We need to have a way to ‘diff’ CFG between batch updates

Batch updates

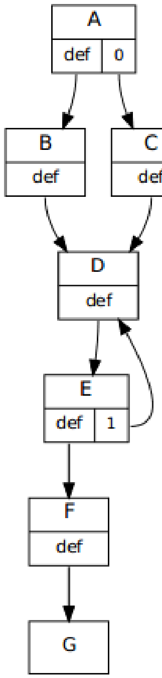
- Depth Based Search needs to see a snapshot of the CFG just after each update
- We do not want to store different versions of the same CFG in DominatorTree
- We need to have a way to ‘diff’ CFG between batch updates
- The list of updates to perform is also the full list of changes to the CFG

Batch update algorithm

- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

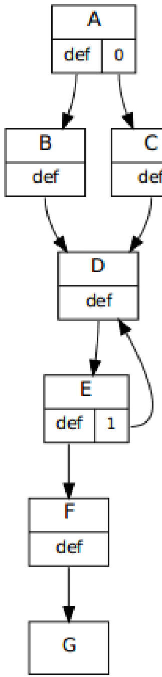
Current CFG



- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

```
Updates = {{Insert, C, D},
           {Insert, E, D},
           {Delete, E, C},
           {Insert, F, G}}
```

Current CFG

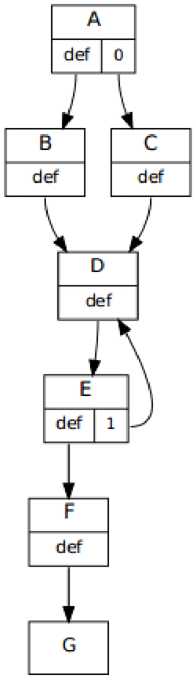


- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

```
Updates = {{Insert, C, D},
           {Insert, E, D},
           {Delete, E, C},
           {Insert, F, G}}
```

```
CFG'    = CFG \ Updates[3:4]
CFG''   = CFG \ Updates[2:4]
CFG'''  = CFG \ Updates[1:4]
CFG'''' = CFG \ Updates[0:4]
```

Current CFG

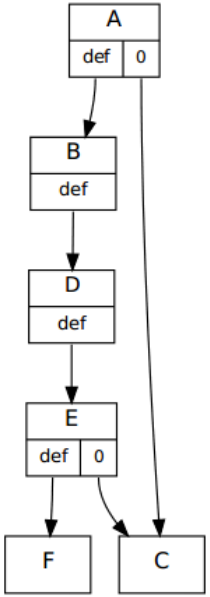


- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

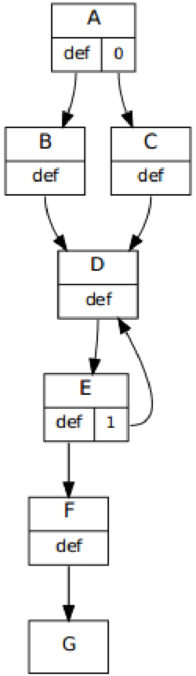
```
Updates = {{Insert, C, D},
           {Insert, E, D},
           {Delete, E, C},
           {Insert, F, G}}
```

```
CFG'    = CFG \ Updates[3:4]
CFG''   = CFG \ Updates[2:4]
CFG'''  = CFG \ Updates[1:4]
CFG'''' = CFG \ Updates[0:4]
```

CFG''''



Current CFG



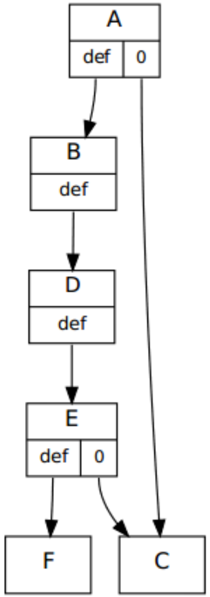
- Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

```
Updates = {{Insert, C, D},
           {Insert, E, D},
           {Delete, E, C},
           {Insert, F, G}}
```

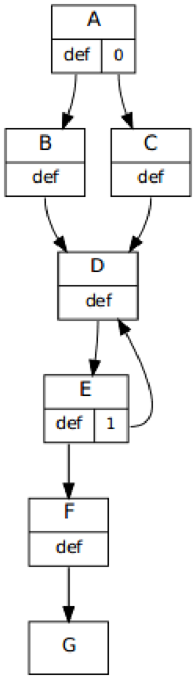
```
CFG'    = CFG \ Updates[3:4]
CFG''   = CFG \ Updates[2:4]
CFG'''  = CFG \ Updates[1:4]
CFG'''' = CFG \ Updates[0:4]
```

Because every permutation of a sequence of updates yields the same DominatorTree, we are free to reorder them internally.

CFG''''



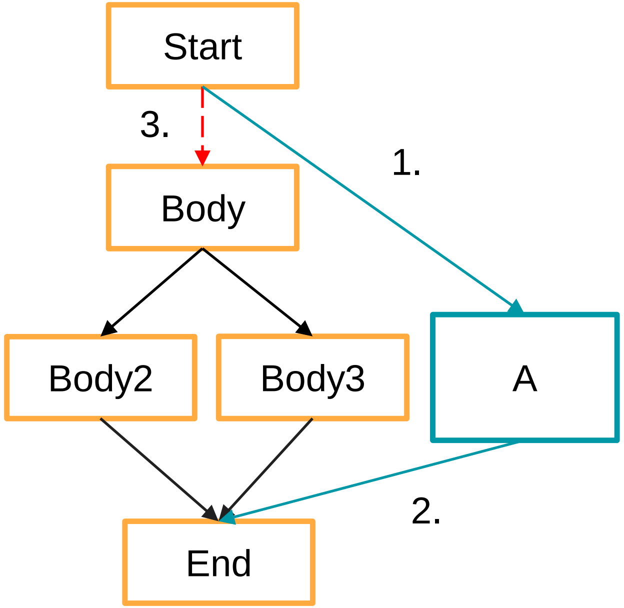
Current CFG



Batch update API

- `DT.applyUpdates(Updates)`
- In action:

```
0.   SmallVector<DominatorTree::UpdateType, 3> Updates;
1.   Updates.push_back({DT::Insert, Start, A   });
2.   Updates.push_back({DT::Insert, A,     End });
3.   Updates.push_back({DT::Delete, Start, Body});
4.   DT.applyUpdates(Updates);
```



Batch update API

- Used to preserve dominators in:
  - LoopRerolling
  - LoopUnswitching
  - BreakCriticalEdges
  - AggressiveDeadCodeElimination
  - JumpThreading (by Samsung Research)

Verifiers

- Old validation: builds a new DominatorTree and checks if it compares equal
    - `DT.verifyDominatorTree()`

  \- Not able validate the PostDominatorTree

  \- Does not check correctness of a freshly calculated tree

  \+ Relatively cheap

Verifiers

- Old validation: builds a new DominatorTree and checks if it compares equal
  - `DT.verifyDominatorTree()`

  \- Not able validate the PostDominatorTree

  \- Does not check correctness of a freshly calculated tree

  \+ Relatively cheap

- New validation: validates every bit of information in the DominatorTree!
  - `DT.verify()`

  \+ Able to check both dominators and postdominators

  \+ Able to validate freshly calculated trees

  \- Expensive – $O(n^3)$

New validation

- `verifyRoots` – checks if roots correspond to the CFG
- `verifyReachablility` – checks if the same nodes are in the CFG and in the DT
- `verifyParentProperty` – ensures the parent property holds – $O(n^2)$
- `verifySiblingProperty` – ensures the sibling property holds – $O(n^3)$
- `verifyLevels` – checks if the tree levels stored in tree nodes are consistent
- `verifyDFSNumbers` – ensures that (not invalidated) DFS numbers are correct

verifyDFSNumbers – bugs possible to find

Bug 34466 - opt crashes with "opt -instcombine -adce -newgvn -gvn-hoist ": Assertion `DT->dominates(NewBB, OldBB) && "invalid path"' failed

Status: RESOLVED FIXED

Bug 34355 - opt crashes with "opt -gvn -gvn-hoist -instcombine -gvn-hoist -instcombine -adce -loop-vectorize": Assertion `Headers.size() >= 2 && "Expected irreducible CFG; -loop-info is likely invalid"' failed

Status: RESOLVED FIXED

Reported: 2017-08-28 22:06 PDT by Zhendong Su
Modified: 2017-09-26 15:14 PDT (History)

Bug 34461 - opt crashes with "opt -gvn -inline -slp-vectorizer -adce -gvn-hoist -sroa": Assertion `UBB == DBB' failed

Status: RESOLVED FIXED

Bug 34345 - MemorySSA crashes when using ADCE preserved dominators. Assertion `dominates(MP, U) && "Memory PHI does not dominate it's uses"' failed.

Failing Tests (2):
    Polly :: Isl/CodeGen/OpenMP/reference-argument-from-non-affine-region.ll
    Polly :: Isl/CodeGen/OpenMP/two-parallel-loops-reference-outer-indvar.ll

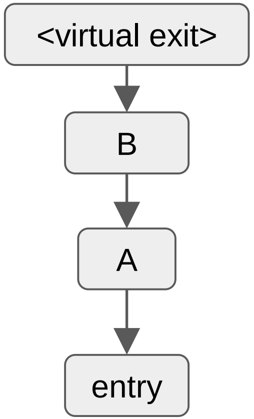
```
Incorrect DFS numbers for:
        Parent %for.body13.i53 {18, 23}
        Child %cleanup.loopexit65 {21, 22}
All children: %cleanup.loopexit65 {21, 22},

~~~~~~~~~~~
                DomTree verification failed!
~~~~~~~~~~~
=============================--------------------------------
Inorder Dominator Tree:
  [1] %entry {0,59} [0]
    [2] %while.cond.i {1,58} [1]
      [3] %_ZN11__sanitizer15internal_strlenEPKc.exit {2,57} [2]
        [4] %_ZN11__sanitizer16internal_strnlenEPKcm.exit {3,48} [3]
```

[llvm] r314847 - [Dominators] Make eraseNode invalidate DFS numbers   llvm x   llvm/llvm-commits x

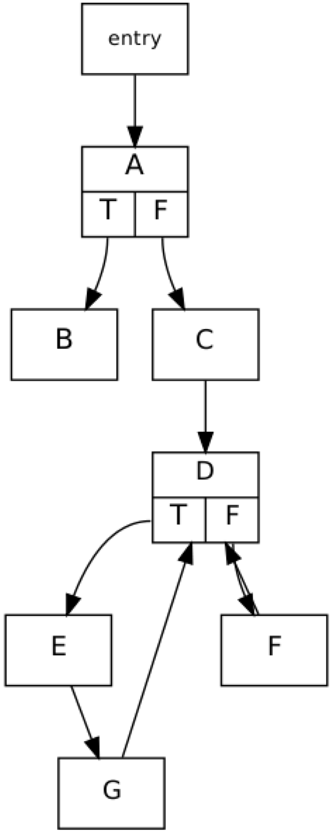
[llvm] r314254 - [Dominators] Invalidate DFS numbers upon edge deletions   llvm x   llvm/llvm-commits x

Postdominators and infinite loops

Postdominator Tree

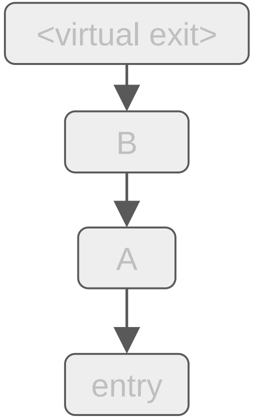


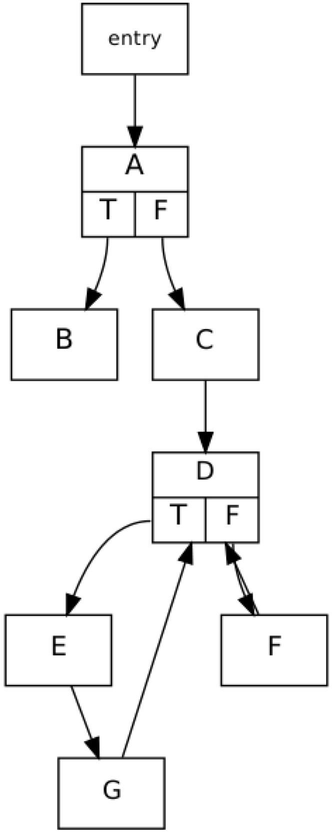
Roots: B



Postdominators and infinite loops

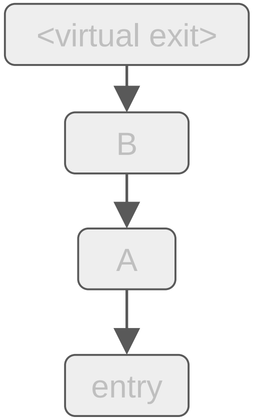
Postdominator Tree



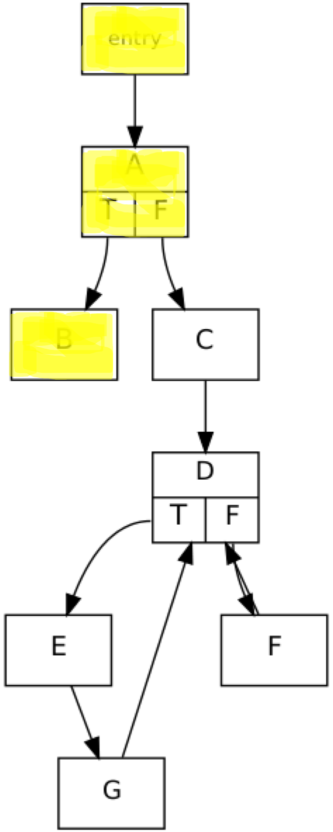


Postdominators and infinite loops

Postdominator Tree

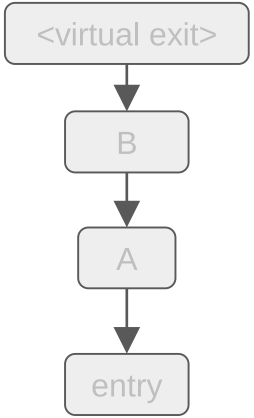


Roots: B

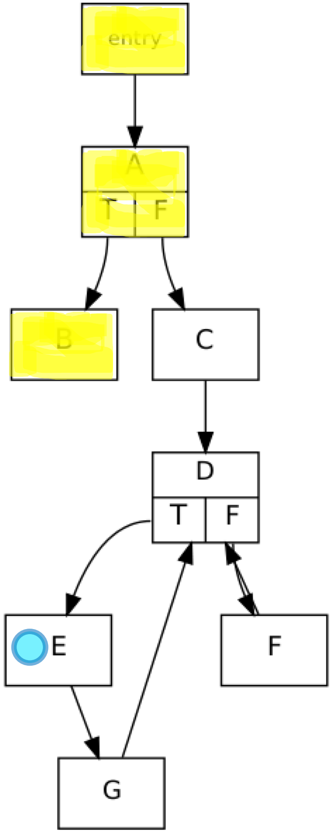


Postdominators and infinite loops

Postdominator Tree

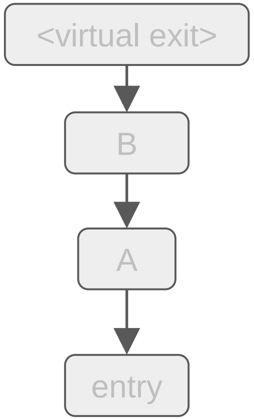


Roots: B

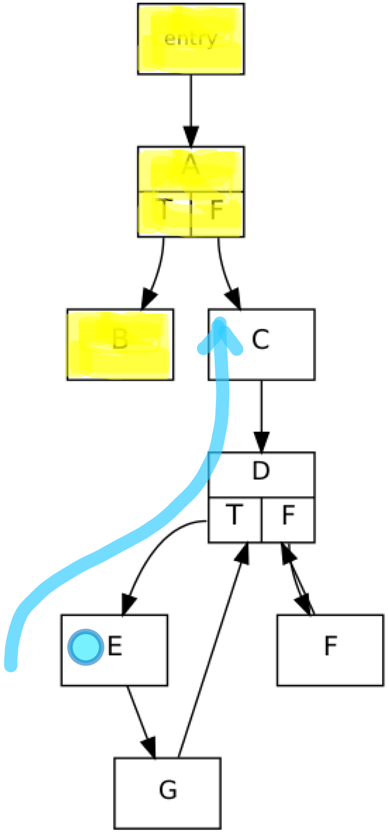


Postdominators and infinite loops

Postdominator Tree

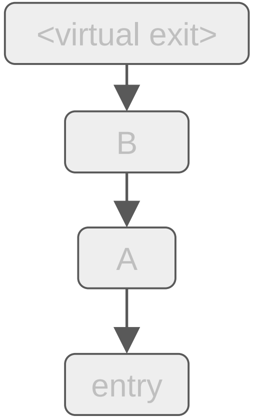


Roots: B

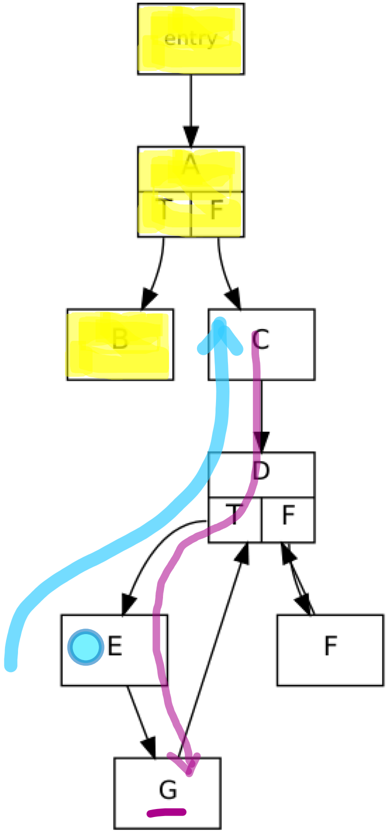


Postdominators and infinite loops

Postdominator Tree

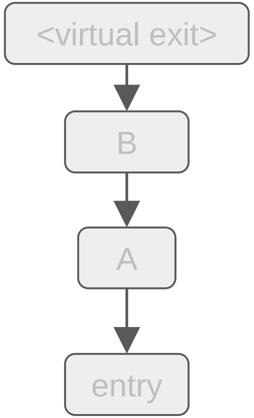


Roots: B , G

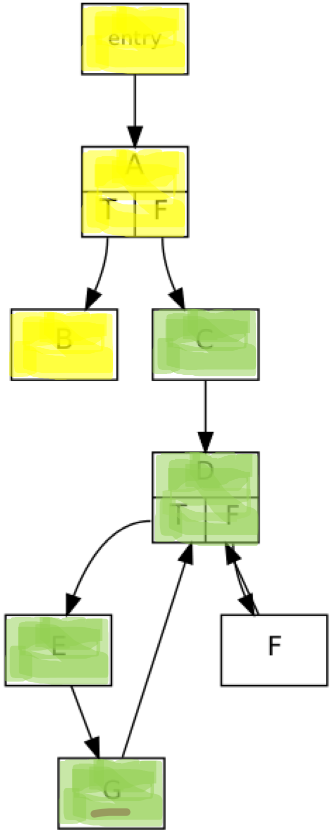


Postdominators and infinite loops

Postdominator Tree

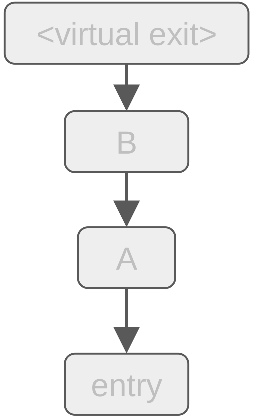


Roots: B , G

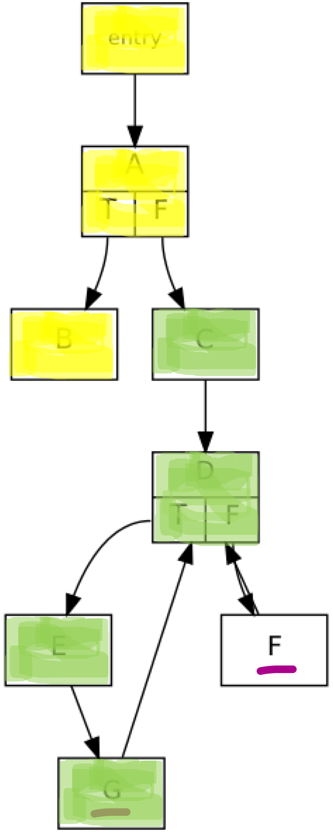


Postdominators and infinite loops

Postdominator Tree

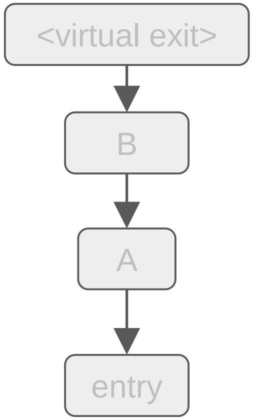


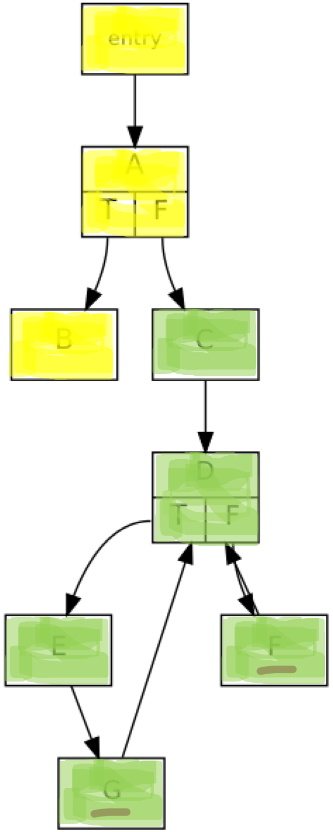
Roots: B , G, F



Postdominators and infinite loops

Postdominator Tree

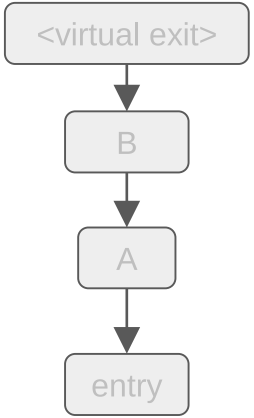


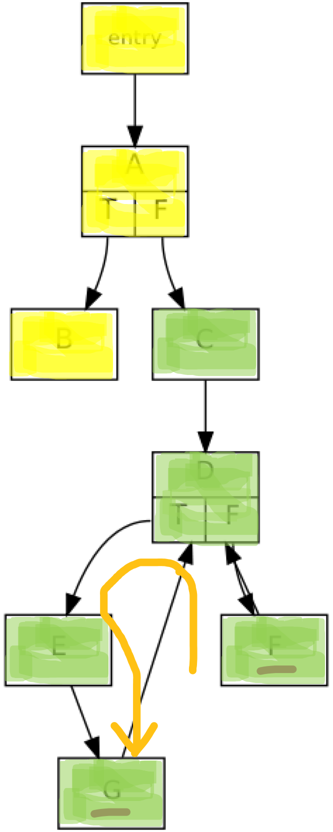


Roots: B , G, F

Postdominators and infinite loops

Postdominator Tree

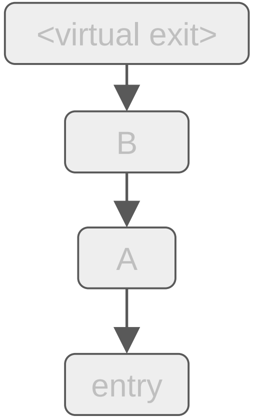




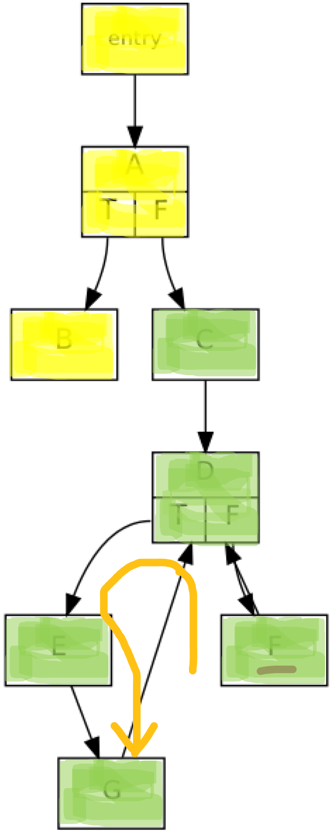
Roots: B , G, F

Postdominators and infinite loops

Postdominator Tree

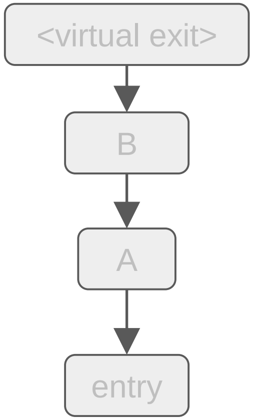


Roots: B , F

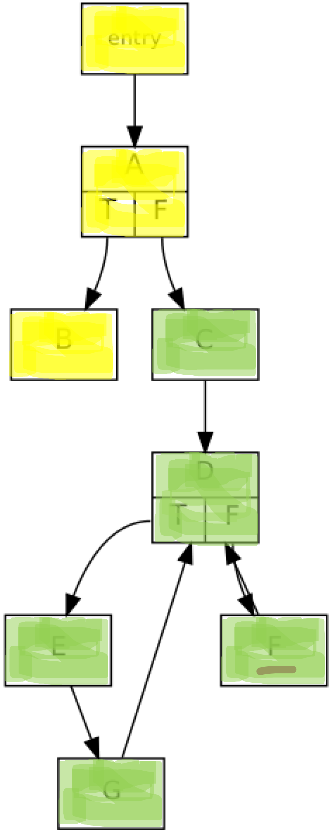


Postdominators and infinite loops

Postdominator Tree

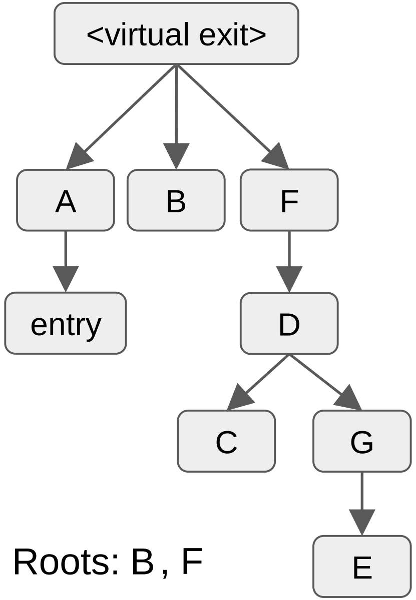


Roots: B , F

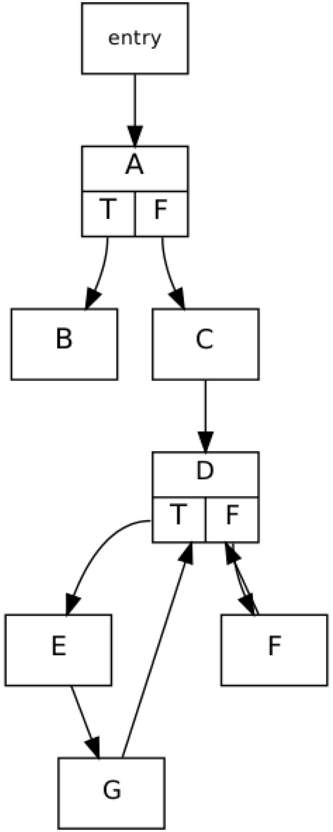


Postdominators and infinite loops

Postdominator Tree



Roots: B , F



Recalculations – currently, with the incremental API

Optimizing a fullLTO clang bitcode with -O3, assertions enabled. (Experiments run on 2x E5-2670 CPU)

June 27 2017 – before switching to Semi-NCA

DomTree recalculations:	1,020,000
DomTree: CFG nodes visited:	48,100,000
Nodes visited per second:	1,705,673
Recalculation time:	28.2s / 15m 15s → 3.1%
PostDomTree recalculations:	50,000
PostDomTree: CFG nodes visited:	2,800,000
Nodes visited per second:	1,818,181
Recalculation time:	1.54s / 15m 15s → 0.16%

October 16 2017 – with incremental batch updates

DomTree recalculations:	1,040,000
DomTree updates:	163,500
DomTree: CFG nodes visited:	49,500,000
Nodes visited per second:	1,718,750
Recalculation time:	28.8s / 18m 52s → 2.54%
Update time:	0.6s / 18m 52s → 0.05%
PostDomTree recalculations:	50,000
PostDomTree: CFG nodes visited:	5,800,000
Nodes visited per second:	2,761,905
Optimization time:	2.1s / 18m 52s → 0.19%

TL;DR

- Use the incremental API `DT.applyUpdates()` instead of `DT.changeImmediateDominator(…)`
  - May be slower, but works for both dominators and postdominators
  - Is guaranteed to be correct
  - If it's too slow, let me know!
  - When in doubt, add `assert(DT.verify())` when working on your pass

Remaining problems

- Interface for incremental updates CFG-level, not IR-level
  - Operates on changed edges
  - Each transform has to collect affected edges on its own
  - Not easily expressible common idioms, e.g. ReplaceAllUsesWith

- After performing incremental updates, next pass may invalidate the Dominator Tree
  - It will be recalculated anyway

Future work

- Converting remainig passes to use the incremental updater
- Simpler interface – a single updater object able to update both the DominatorTree and PostDominatorTree
- Deferred batch updates – applied lazily when actually needed
- Properly profile and optimize the batch updater

Thank you

Questions?

Jakub (Kuba) Kuderski
kubakuderski@gmail.com