Dominator Trees
and incremental updates that transcend time

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

CFG (Control Flow Graph)
Introduction

Dominance:

CFG (Control Flow Graph)
Introduction

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.
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.
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.
Dominance:

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

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

Immediate dominators
Postdominator Tree:

Immediate postdominators
Postdominator Tree:

Multiple exits: D, G, H

Roots: D, G, H
Postdominator Tree:

Roots: D, G, H

Multiple exits: D, G, H

(entry)

nullptr

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

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

DFS In/Out numbers – calculated lazily

Textual representation
(for debugging)

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->'
  - 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

L. Georgiadis et al.
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

---

L. Georgiadis et al.  
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 – $O(n^2)$
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

```
entry
    switch
    /   |
   /    |
default two four five exit
```

CFG

```
entry
  switch
def 2 4 5
```

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

1. [ADCE] make %two the only successor of %switch
Depth Based Search confused

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

1. [ADCE] make %two the only successor of %switch
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
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
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
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
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
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
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
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
• Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

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

\[\begin{align*}
\text{CFG}' &= \text{CFG} \setminus \text{Updates}[3:4] \\
\text{CFG}'' &= \text{CFG} \setminus \text{Updates}[2:4] \\
\text{CFG}''' &= \text{CFG} \setminus \text{Updates}[1:4] \\
\text{CFG}'''' &= \text{CFG} \setminus \text{Updates}[0:4]
\end{align*}\]
Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

Updates = \{
{\text{Insert, } C, D}, \nn{\text{Insert, } E, D}, \nn{\text{Delete, } E, C}, \nn{\text{Insert, } F, G}\} \n
\text{CFG}' = \text{CFG} \setminus \text{Updates}[3:4] \n\text{CFG}'' = \text{CFG} \setminus \text{Updates}[2:4] \n\text{CFG}''' = \text{CFG} \setminus \text{Updates}[1:4] \n\text{CFG}'''' = \text{CFG} \setminus \text{Updates}[0:4]
• Reverse-apply updates to the CFG from the future to get the snapshots of the CFG in the past

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

\[
\begin{align*}
\text{CFG}' & = \text{CFG} \setminus \text{Updates}[3:4] \\
\text{CFG}'' & = \text{CFG} \setminus \text{Updates}[2:4] \\
\text{CFG}''' & = \text{CFG} \setminus \text{Updates}[1:4] \\
\text{CFG}'''' & = \text{CFG} \setminus \text{Updates}[0:4]
\end{align*}
\]

Because every permutation of a sequence of updates yields the same DominatorTree, we are free to reorder them internally.
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
- **verifyReachability** – 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
- **verifyDFSNumer** – 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
Reported: 2017-08-28 10:26 PDT by Zhenfeng Su
Modified: 2017-08-28 15:14 PDT (History)
Status: RESOLVED FIXED

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

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
Polly::Isl::CodeGen::OpenMP::two-parallel-loops-reference-outer-invar

Incorrect DFS numbers for:
   Parent `xfor.body13.body13` (18, 23)
   Child `%cleanup.loopexit65` (21, 22)
   All children: `%cleanup.loopexit65` (21, 22),

-------------------
DomTree verification failed
-------------------

Inorder Dominator Tree:
1. Mentry {0, 59} [0]
   2. `while.cond.l` (1, 58) [1]
      3. `%ZN11_sanitizerIsInternal_strlenEPKc.exit` (2, 57) [2]
      4. `%ZN11_sanitizerIsInternal_strlenEPKcm.exit` (3, 48) [3]
Postdominators and infinite loops

Postdominator Tree

Roots: B
Postdominators and infinite loops

Postdominator Tree

A

<virtual exit>

B

A

entry

entry

A

T

F

B

C

D

T

F

E

F

G
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

<virtual exit>

B

A

entry

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

Diagram:

- Entry node
- Nodes A, B, F, C, G, E, D, T, F, B, C, D, E, F, G
- Connections between nodes

Diagram on the right:

- Entry node
- Nodes A, T, F, B, C, D, T, F, E, F, G
- Connections between nodes
### Recalculations – currently, with the incremental API

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
<th>DomTree recalculation</th>
<th>DomTree: CFG nodes visited</th>
<th>Nodes visited per second</th>
<th>Recalculation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>June 27 2017</strong></td>
<td>before switching to Semi-NCA</td>
<td>1,020,000</td>
<td>48,100,000</td>
<td>1,705,673</td>
<td>28.2s / 15m 15s → 3.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>October 16 2017</strong></td>
<td>with incremental batch updates</td>
<td>1,040,000</td>
<td>49,500,000</td>
<td>1,718,750</td>
<td>28.8s / 18m 52s → 2.54%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Update time: 0.6s / 18m 52s → 0.05%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
<th>PostDomTree recalculation</th>
<th>PostDomTree: CFG nodes visited</th>
<th>Nodes visited per second</th>
<th>Recalculation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>June 27 2017</strong></td>
<td>before switching to Semi-NCA</td>
<td>50,000</td>
<td>2,800,000</td>
<td>1,818,181</td>
<td>1.54s / 15m 15s → 0.16%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>October 16 2017</strong></td>
<td>with incremental batch updates</td>
<td>50,000</td>
<td>5,800,000</td>
<td>2,761,905</td>
<td>2.1s / 18m 52s → 0.19%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Optimization time:</td>
</tr>
</tbody>
</table>


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 remaining 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