Outline

How to coalesce variables?

Two alternatives

- Use a working interference graph where, in case of coalescing, the corresponding nodes are merged. $O(1)$ interference query.
- Manipulate congruence classes, i.e., sets of coalesced variables. Interferences must be tested between sets.

Chaitin, Sreedhar, Budimlić use congruence classes. Also useful to avoid interference graph. Naive algorithm: quadratic complexity.

How to coalesce variables?

Two alternatives

- Use a working interference graph where, in case of coalescing, the corresponding nodes are merged. $O(1)$ interference query.
- Manipulate congruence classes, i.e., sets of coalesced variables. Interferences must be tested between sets.

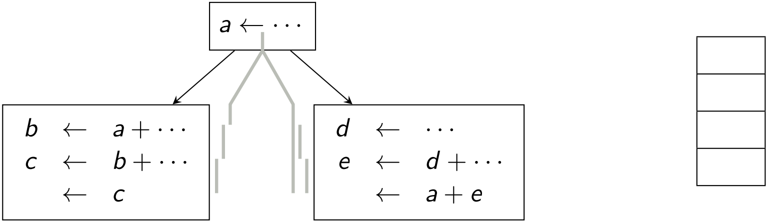
Chaitin, Sreedhar, Budimlić use congruence classes. Also useful to avoid interference graph. Naive algorithm: quadratic complexity.

Key properties for linear-complexity live range intersection

- 2 variables intersect if one is live at the definition of the other.
- In this case, the first definition dominates the second one.
- Budimlić: a set contains 2 intersecting variables if it contains a variable that intersects its "parent dominating" variable.

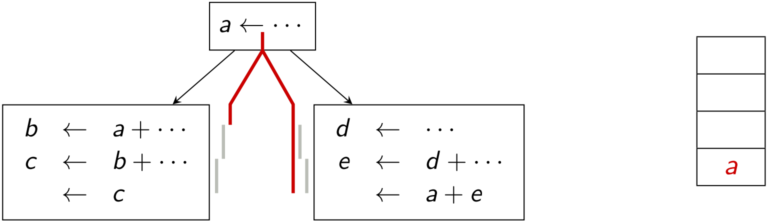
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.



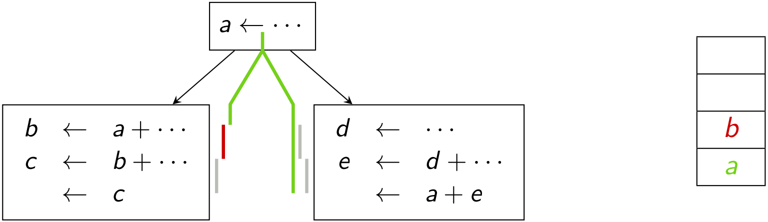
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.



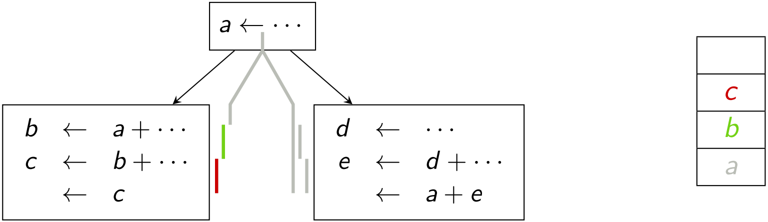
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.



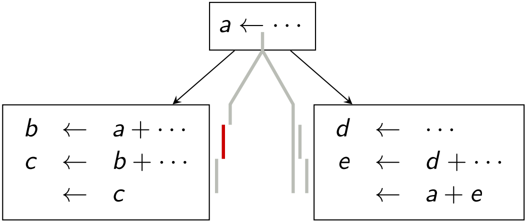
Fast interference test for a set of variables

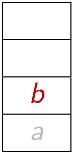
- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.



Fast interference test for a set of variables

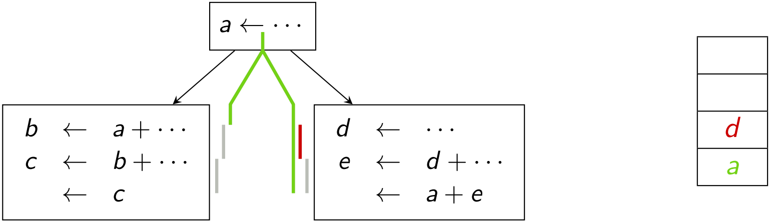
- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.





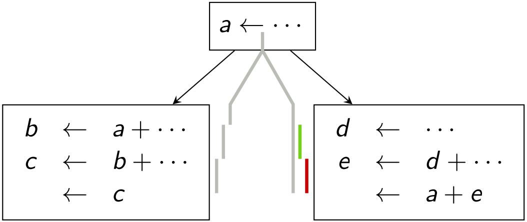
Fast interference test for a set of variables

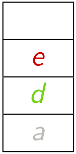
- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.



Fast interference test for a set of variables

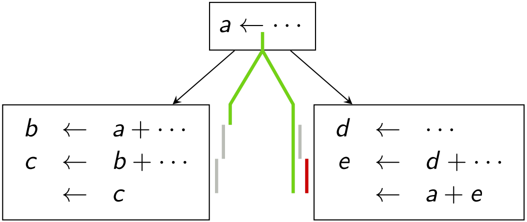
- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.

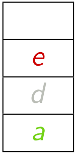




Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with "parent dominating" variable.





Algorithm 2: Check intersection in a set of variables

Data: `list` sorted according to a pre-DFS order of the dominance tree
Output: Returns TRUE if the list contains an interference

```
dom ← empty_stack ; i ← 0 ;                              /* stack of the traversal */
while i < list.size() do
    current ← list(i++) ;
    other ← dom.top() ;                                  /* NULL if dom is empty */
    while (other ≠ NULL) and dominate(other, current) = FALSE do
        dom.pop() ;                                      /* not the desired parent, remove */
        other ← dom.top() ;                              /* consider next one */
    parent ← other ;
    if (parent ≠ NULL) and (intersect(current, parent) = TRUE) then
    return TRUE ;                                        /* intersection detected */
    dom.push(current) ;                                  /* otherwise, keep checking */
return FALSE ;
```

Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one ☛ linear number of tests.

Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one ☛ linear number of tests.

$i_r \leftarrow 0$; $i_b \leftarrow 0$;
while ($i_r <$ red.size() and $i_b <$ blue.size()) **do**
    if blue(i_b) $\prec$ red(i_r) **then** current $\leftarrow$ blue(i_b++) **else** current $\leftarrow$ red(i_r++)
while($i_r <$ red.size() and $n_b > 0$) **do** current $\leftarrow$ red(i_r++) */* still n_b blue in stack */*
while($i_b <$ blue.size() and $n_r > 0$) **do** current $\leftarrow$ blue(i_b++) */* still n_r red in stack */*

Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one ☛ linear number of tests.

- No need to test intersection of variables in the same set.
- Take values into account for value-based interference: need links of "equal ancestors", which may increase complexity.
- Sort in linear time the resulting set, in case of coalescing.

Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one ☛ linear number of tests.

- No need to test intersection of variables in the same set.
- Take values into account for value-based interference: need links of "equal ancestors", which may increase complexity.
- Sort in linear time the resulting set, in case of coalescing.

Fewer intersection tests ☛ possible now to use more expensive queries for intersection/liveness and to avoid interference graph:

- Budimlić intersection test, still using liveness sets.
- Fast liveness checking of Boissinot et al. (CGO'08).

Algorithm 3: interference(a, b)

Data: A variable a and its parent b in the dominance tree

Output: Returns TRUE if a interferes (i.e., intersects and has a different value) with an already-visited variable. Also, update `equal_anc` information

```
/* a and b are assumed to not be equal to NULL                                  */
equal_anc_out(a) ← NULL ;                                      /* initialization */
if a and b are in the same set then
    b ← equal_anc_out(b) ;                           /* check/update in other set */
if value(a) ≠ value(b) then
    return chain_intersect(a, b) ;    /* check with b and its equal intersecting
    ancestors in the other set */
else
    update_equal_anc_out(a, b) ;    /* update equal intersecting ancestor going
    up in the other set */
    return FALSE ;                                            /* no interference */
```

Algorithm 4: update_equal_anc_out(a, b)

Data: Variables a and b, same value, but in different sets

Output: Set nearest intersecting ancestor of a, in other set, with same value (NULL if does not exist)

tmp $\leftarrow$ b ;
while (tmp $\neq$ NULL) and (intersect(a, tmp) $=$ FALSE) **do**
  tmp $\leftarrow$ equal_anc_in(tmp) ; /* *follow the chain of equal intersecting ancestors in the other set* */
equal_anc_out(a) $\leftarrow$ tmp ; /* *tmp intersects a or* NULL */

Algorithm 5: chain_intersect(a, b)

Data: Variables a and b, different value, in different sets

Output: Returns TRUE if a intersects b or one of its equal intersecting ancestors in the same set

tmp $\leftarrow$ b ;
while (tmp $\neq$ NULL) and (intersect(a, tmp) $=$ FALSE) **do**
  tmp $\leftarrow$ equal_anc_in(tmp) ; /* *follow the chain of equal intersecting ancestors* */
if tmp $=$ NULL **then return** FALSE **else return** TRUE ;

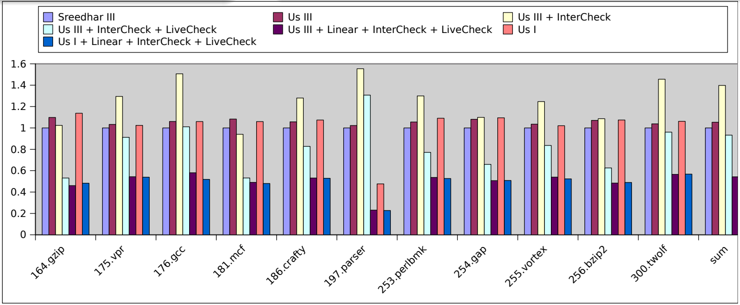
Speed-up for SPEC CINT2000: x2

General scheme

- `Sreedhar III`: w. virtualization.
- `Us I`, `Us III`: our proposal, w.o./w. virtualization.

Interference checks

- Default: liveness sets + interference graph.
- `InterCheck`: Budimlić with liveness sets.
- `LiveCheck`: Fast liveness checking.
- `Linear`: Linear check instead of quadratic.

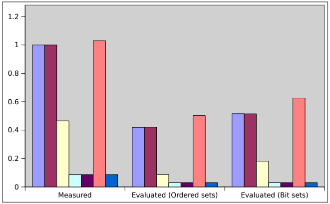


Memory footprint reduction for SPEC CINT2000: x10

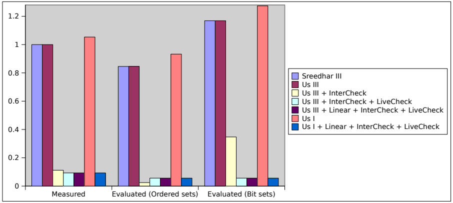
- Interference graph: half-size bit matrix.
- Liveness sets: enumerated sets. Does not count construction.
- Livenesss check: bit sets. Construction taken into account.

Data structures grow during virtualization. "Perfect memory" evaluated, with both enumerated/bit sets for liveness sets.

Sum of memory footprint



Max of memory footprint



General framework

- Correctness clarified even for complex cases
- Two-phases solution, based on coalescing

Results

- Value-based interferences, for free, as good as Sreedhar III
- Fast algorithm: Speed-up x2, memory reduction x10.

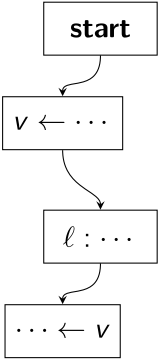
Implementation

- No need to virtualize (at least for us)
- Simpler implementation

Outline

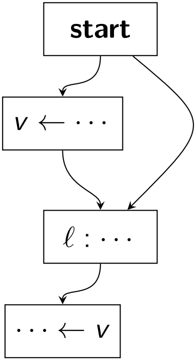
Dominance, liveness, and interference

- A variable v is live(-in) at program point p if there is a path, not containing the definition of v, from q to a use of v.
- Each instruction ℓ, where v is live, is dominated by def(v) the definition point of v: $def(\mathtt{v}) \succeq \ell$.



Dominance, liveness, and interference

- A variable v is live(-in) at program point p if there is a path, not containing the definition of v, from q to a use of v.
- Each instruction ℓ, where v is live, is dominated by def(v) the definition point of v: $\mathit{def}(\mathtt{v}) \succeq \ell$.

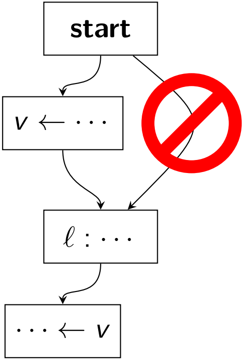


Proof: if ℓ is not dominated by $\mathit{def}(\mathtt{v})$

- there is a path from **start** to ℓ that does not visit $\mathit{def}(\mathtt{v})$.
- v is live at ℓ: there is a path from ℓ to a use of v that does not visit $\mathit{def}(\mathtt{v})$.
- Thus, there is a path from **start** to a use of v that does not visit $\mathit{def}(\mathtt{v})$.

Dominance, liveness, and interference

- A variable v is live(-in) at program point p if there is a path, not containing the definition of v, from q to a use of v.
- Each instruction ℓ, where v is live, is dominated by def(v) the definition point of v: $def(\mathrm{v}) \succeq \ell$.



Proof: if ℓ is not dominated by $def(\mathrm{v})$

- there is a path from **start** to ℓ that does not visit $def(\mathrm{v})$.
- v is live at ℓ: there is a path from ℓ to a use of v that does not visit $def(\mathrm{v})$.
- Thus, there is a path from **start** to a use of v that does not visit $def(\mathrm{v})$.

No: each use of v is dominated by $def(\mathrm{v})$.

Dominance, liveness, and interference

- Assume that v and w are both live at some instruction ℓ.
- Then, $\mathit{def}(\mathtt{v}) \succeq \ell$ and $\mathit{def}(\mathtt{w}) \succeq \ell$.
- Dominance = tree:
  - either $\mathit{def}(\mathtt{v}) \succeq \mathit{def}(\mathtt{w})$ (and, in this case, v is live at $\mathit{def}(\mathtt{w})$);
  - or $\mathit{def}(\mathtt{w}) \succeq \mathit{def}(\mathtt{v})$ (and, in this case, w is live at $\mathit{def}(\mathtt{v})$).

☛ interference can be directed according to dominance.

Dominance, liveness, and interference

- Assume that v and w are both live at some instruction ℓ.
- Then, $\mathit{def}(\mathtt{v}) \succeq \ell$ and $\mathit{def}(\mathtt{w}) \succeq \ell$.
- Dominance = tree:
  - either $\mathit{def}(\mathtt{v}) \succeq \mathit{def}(\mathtt{w})$ (and, in this case, v is live at $\mathit{def}(\mathtt{w})$);
  - or $\mathit{def}(\mathtt{w}) \succeq \mathit{def}(\mathtt{v})$ (and, in this case, w is live at $\mathit{def}(\mathtt{v})$).

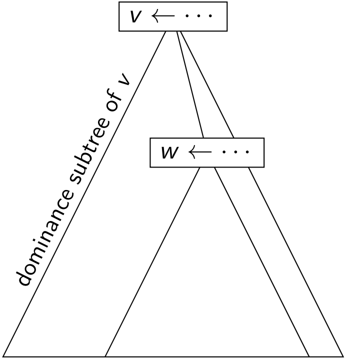
☛ interference can be directed according to dominance.

Consequences

- Strictness implies two equivalent notions of interferences:
  - live ranges intersect;
  - one variable is live at the definition of the other.
- Assume no equality among intersecting variables: then, the interference graph of an SSA program is chordal/triangulated.

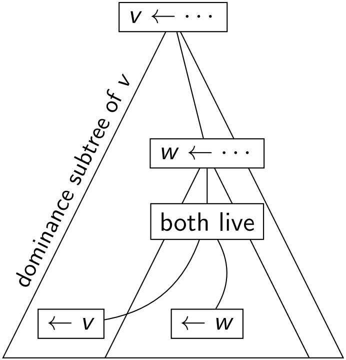
Intersecting live ranges, subtrees of a tree

- Assume $v \xrightarrow{\text{dom}} w$
- Then, v is live at $def(\mathtt{w})$



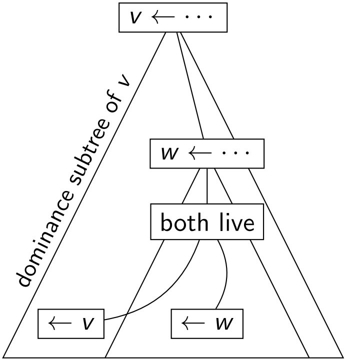
Intersecting live ranges, subtrees of a tree

- Assume $v \xrightarrow{\text{dom}} w$
- Then, v is live at $def(\mathtt{w})$



Intersecting live ranges, subtrees of a tree

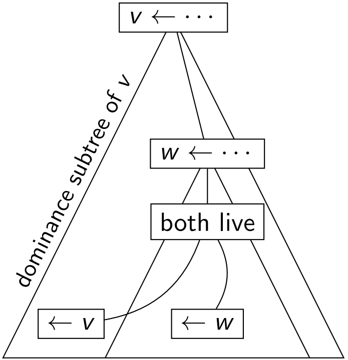
- Assume $v \xrightarrow{\text{dom}} w$
- Then, v is live at $def(\mathtt{w})$



- Live ranges of variables can be represented as subtrees of the dominance tree ☛ intersection graph = chordal graph.

Intersecting live ranges, subtrees of a tree

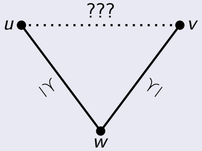
- Assume $v \xrightarrow{\text{dom}} w$
- Then, v is live at $def(\texttt{w})$



- Live ranges of variables can be represented as subtrees of the dominance tree ☛ intersection graph = chordal graph.

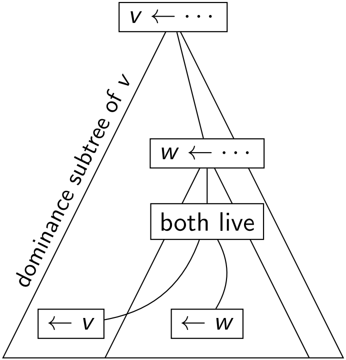
Other proof: no chordless cycle

Consider a cycle in the interference graph. There must be three vertices u, v, w, such that:



Intersecting live ranges, subtrees of a tree

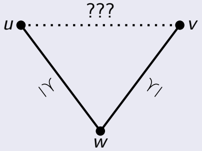
- Assume $v \overset{\text{dom}}{\bullet\!\!-\!\!-\!\!-\!\!-\!\!\bullet} w$
- Then, v is live at *def*(`w`)



- Live ranges of variables can be represented as subtrees of the dominance tree ☛ intersection graph = chordal graph.

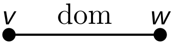
Other proof: no chordless cycle

Consider a cycle in the interference graph. There must be three vertices u, v, w, such that:

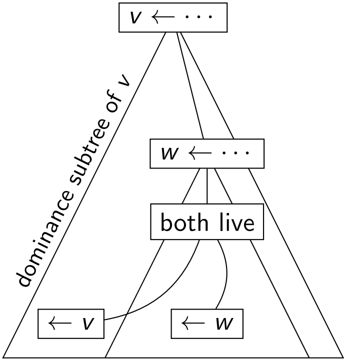


u and v are both live at *def*(`w`).

Intersecting live ranges, subtrees of a tree

- Assume 

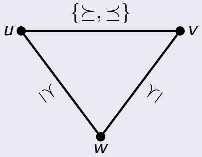
- Then, v is live at $def(\mathtt{w})$



- Live ranges of variables can be represented as subtrees of the dominance tree ☛ intersection graph = chordal graph.

Other proof: no chordless cycle

Consider a cycle in the interference graph. There must be three vertices u, v, w, such that:



u and v are both live at $def(\mathtt{w})$. They thus interfere (chord).

SSA versus non-SSA interference graphs

Program

$a \leftarrow \cdots$

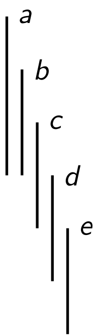
$b \leftarrow \cdots$

$c \leftarrow \cdots$

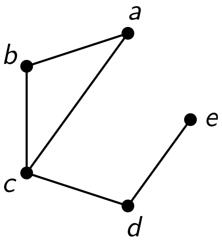
$d \leftarrow a + b$

$e \leftarrow c + 1$

Live Ranges



Interference Graph



- How can we create a 4-cycle $\{a, c, d, e\}$?

SSA versus non-SSA interference graphs

Program

$a \leftarrow \cdots$

$b \leftarrow \cdots$

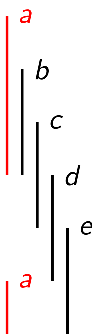
$c \leftarrow \cdots$

$d \leftarrow a + b$

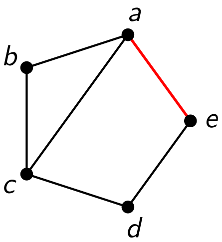
$e \leftarrow c + 1$

$a \leftarrow \cdots$

Live Ranges



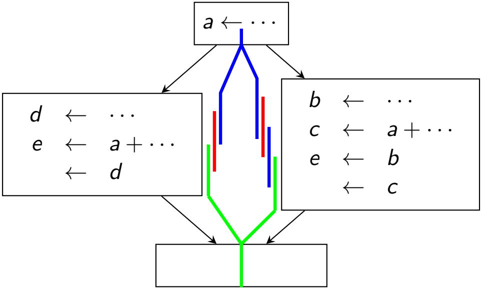
Interference Graph



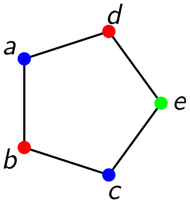
- How can we create a 4-cycle $\{a, c, d, e\}$?
- Redefine $a \implies$ SSA violated!

SSA versus non-SSA interference graphs

Program and live ranges

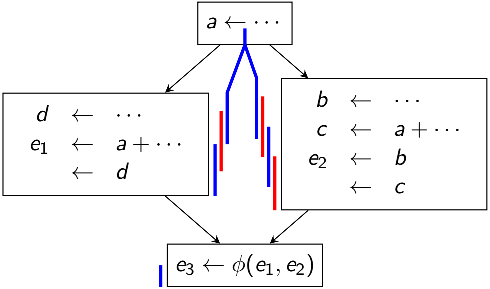


Interference Graph

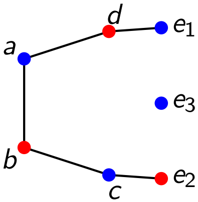


SSA versus non-SSA interference graphs

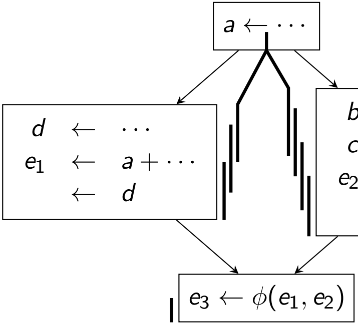
Program and live ranges

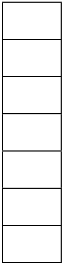


Interference Graph



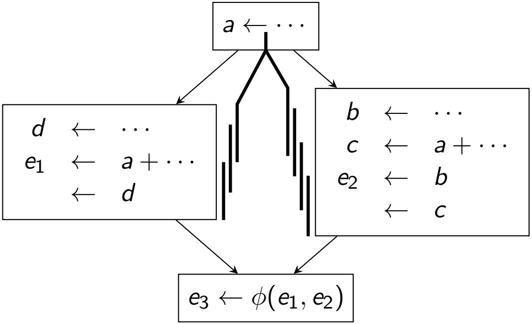
Chordal k-colorable: greedy-k-colorable & tree-scan

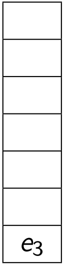




- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

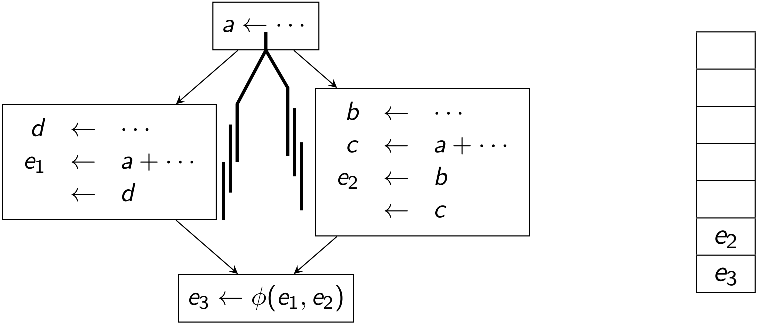
Chordal k-colorable: greedy-k-colorable & tree-scan





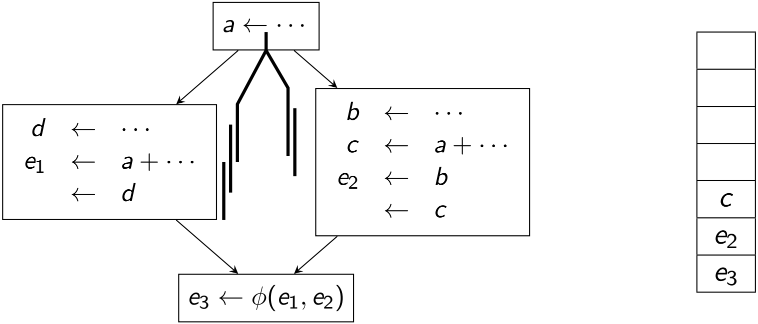
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

Chordal k-colorable: greedy-k-colorable & tree-scan



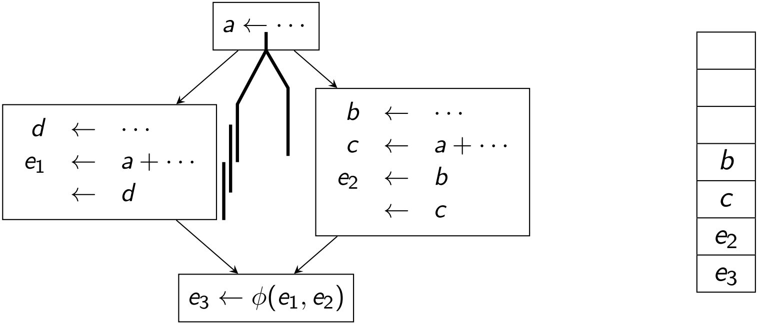
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

Chordal k-colorable: greedy-k-colorable & tree-scan



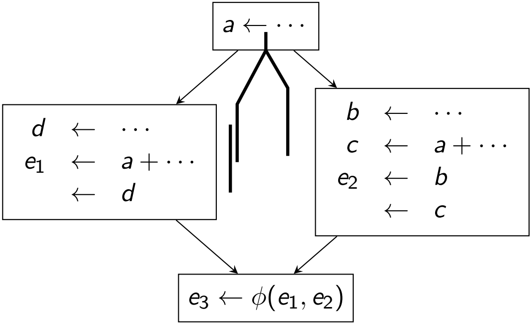
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

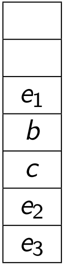
Chordal k-colorable: greedy-k-colorable & tree-scan



- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

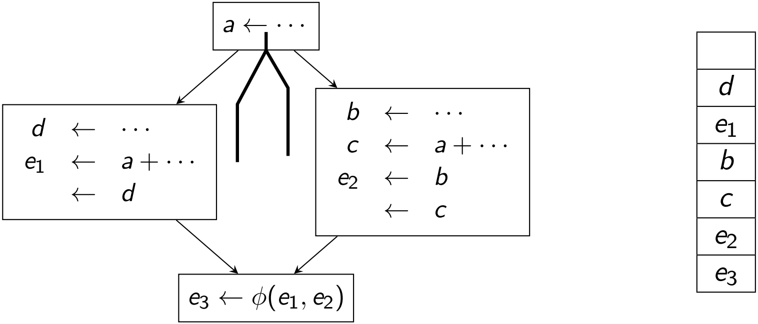
Chordal k-colorable: greedy-k-colorable & tree-scan





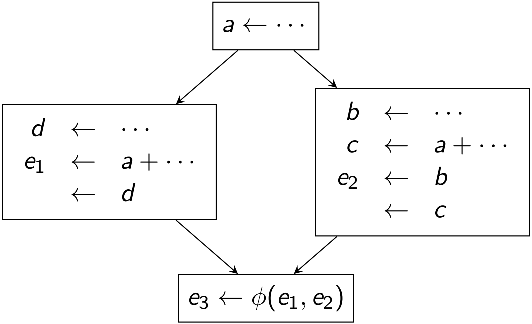
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

Chordal k-colorable: greedy-k-colorable & tree-scan



- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.

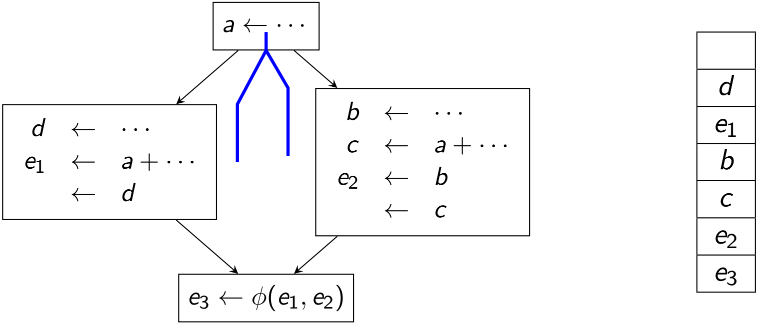
Chordal k-colorable: greedy-k-colorable & tree-scan



a
d
e_1
b
c
e_2
e_3

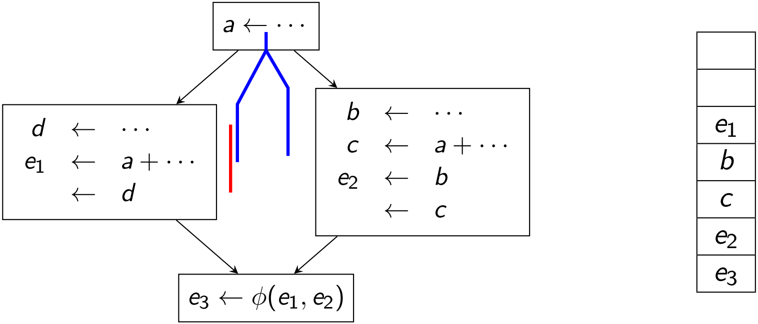
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

Chordal k-colorable: greedy-k-colorable & tree-scan



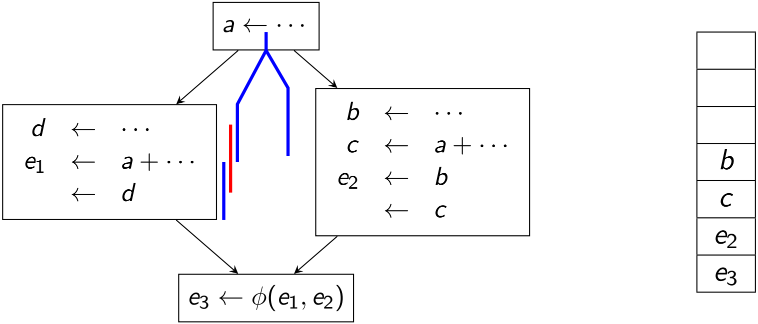
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

Chordal k-colorable: greedy-k-colorable & tree-scan



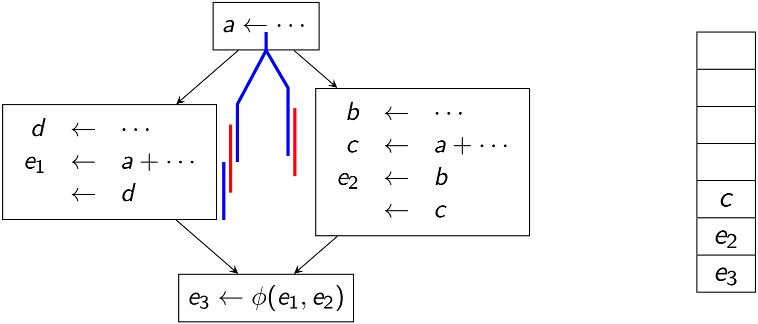
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

Chordal k-colorable: greedy-k-colorable & tree-scan



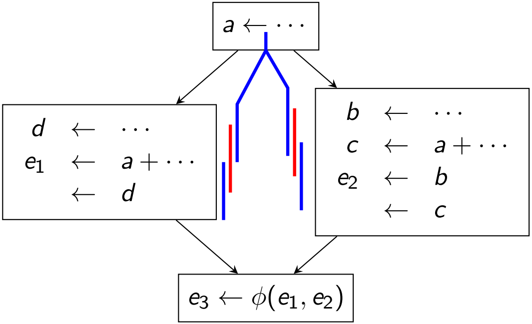
- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

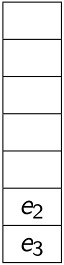
Chordal k-colorable: greedy-k-colorable & tree-scan



- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

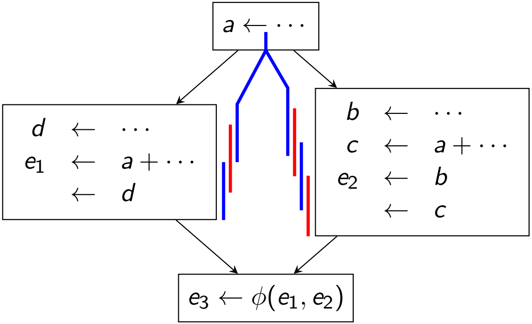
Chordal k-colorable: greedy-k-colorable & tree-scan

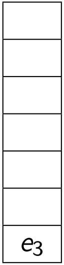




- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

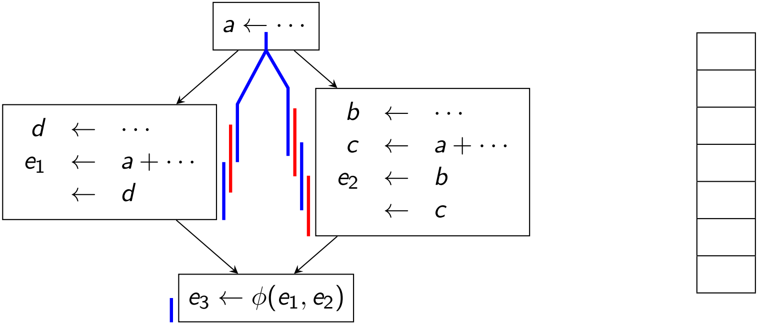
Chordal k-colorable: greedy-k-colorable & tree-scan





- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

Chordal k-colorable: greedy-k-colorable & tree-scan



- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-k-colorable: all vertices can be successively "simplified" ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

Outline

Traditional fixed-point data-flow analysis

Equations

$$
\begin{aligned}
\mathrm{LiveIn}(B) &= \mathrm{PhiDefs}(B) \cup \mathrm{UpwardExposed}(B) \cup (\mathrm{LiveOut}(B) \setminus \mathrm{Defs}(B)) \\
\mathrm{LiveOut}(B) &= \textstyle\bigcup_{S \in \mathrm{succs}(B)} (\mathrm{LiveIn}(S) \setminus \mathrm{PhiDefs}(S)) \cup \mathrm{PhiUses}(B)
\end{aligned}
$$

- $\mathrm{PhiDefs}(B)$: variables defined by ϕ-operations at entry of B.
- $\mathrm{PhiUses}(B)$: used by ϕ-operations at a successor block of B.
- $\mathrm{UpwardExposed}(B)$: used in B but not defined earlier in B.

Traditional fixed-point data-flow analysis

Equations

$$\begin{aligned}\mathrm{LiveIn}(B) &= \mathrm{PhiDefs}(B) \cup \mathrm{UpwardExposed}(B) \cup (\mathrm{LiveOut}(B) \setminus \mathrm{Defs}(B))\\ \mathrm{LiveOut}(B) &= \bigcup_{S \in \mathrm{succs}(B)} (\mathrm{LiveIn}(S) \setminus \mathrm{PhiDefs}(S)) \cup \mathrm{PhiUses}(B)\end{aligned}$$

- $\mathrm{PhiDefs}(B)$: variables defined by ϕ-operations at entry of B.
- $\mathrm{PhiUses}(B)$: used by ϕ-operations at a successor block of B.
- $\mathrm{UpwardExposed}(B)$: used in B but not defined earlier in B.

Complexity

- W: non-local variables (i.e., not fully in a block), P: program.
- $G = (V, E)$: CFG with $|V| - 1 \leq |E| \leq |V|^2$.

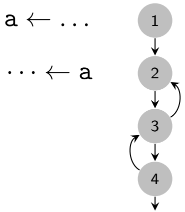
$O(|P|) + O(|W|) \times$ number of iterations, i.e., $O(|E||W|)$ for worklist algorithms and $O(|E|(d(G, T) + 3))$ for round robin.

$d(G, T)$: max. number of back edges (for a DFS tree T), in a cycle-free path of G.

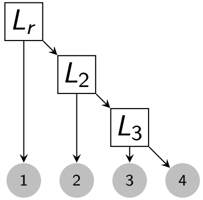
Exploiting loop structure

- $G = (V, E)$: reducible CFG with strict SSA.
- $\mathcal{F}_{\mathcal{L}}(G)$: DAG obtained by removing loop-edges.

Bad case for iterative data-flow analysis:



Control-flow graph



Loop-nesting forest

Principles to avoid iteration:

- Compute liveness information, traversing $\mathcal{F}_{\mathcal{L}}(G)$ bottom-up.
- Refine liveness by exploiting loop structure.

Key lemmas related to loop structure

Lemma 1

Let G be a reducible CFG, v an SSA variable, and d its definition. If L is a maximal loop not containing d, then v is live-in at the loop-header h of L iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, not containing d, from h to a use of v.

Lemma 2

Let G be a reducible CFG, v an SSA variable, and d its definition. Let p be a node of G such that all loops containing p also contain d. Then v is live-in at p iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, from p to a use of v, not containing d.

Key lemmas related to loop structure (cont'd)

- Propagating liveness along $\mathcal{F}_{\mathcal{L}}(G)$ can only mark live-in variables that are indeed live-in.
- If, after this propagation, v is missing at p, p belongs to a loop that does not contain the definition of v (Lemma 2).
- If L is such a maximal loop, v is correctly marked as live-in at the header of L (Lemma 1).

Lemma 3

Consider L a loop and v *an SSA variable. If* v *is live-in at the loop-header of L, it is live-in and live-out at every node in L.*

☛ Propagating inside loops is enough to patch the liveness sets.

Partial liveness, with postorder traversal

Algorithm 6: DAG_DFS(block B)

for each $S \in \mathrm{succs}(B)$ if (B, S) is not a loop-edge **do**
    if S is unprocessed **then**
        DAG_DFS(S)

$\mathit{Live} = \mathrm{PhiUses}(B)$ /* *used by ϕ-functions in B's successors* */
for each $S \in \mathrm{succs}(B)$ if (B, S) is not a loop-edge **do**
    $\mathit{Live} = \mathit{Live} \cup (\mathrm{LiveIn}(S) \setminus \mathrm{PhiDefs}(S))$

$\mathrm{LiveOut}(B) = \mathit{Live}$;
for each program point p in B, backward **do**
    remove variables defined at p from Live;
    add uses at p in Live

$\mathrm{LiveIn}(B) = \mathit{Live} \cup \mathrm{PhiDefs}(B)$;
mark B as processed

Propagate live variables within loop bodies

Algorithm 7: LoopTree_DFS(node N of the loop forest)

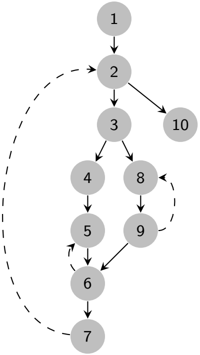
```
if N is a loop node then
    Let B_N = Block(N) /* the loop-header of N */
    Let LiveLoop = LiveIn(B_N) \ PhiDefs(B_N);
    for each M ∈ LoopTree_children(N) do
        Let B_M = Block(M) /* loop-header or block */
        LiveIn(B_M) = LiveIn(B_M) ∪ LiveLoop;
        LiveOut(B_M) = LiveOut(B_M) ∪ LiveLoop;
        LoopTree_DFS(M)
```

Algorithm 8: Compute_LiveSets_SSA_Reducible(CFG)

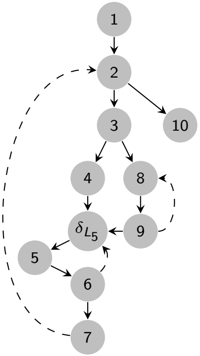
```
for each basic block B do
    mark B as unprocessed
DAG_DFS(R) /* R is the CFG root node */
for each root node L of the loop-nesting forest do
    LoopTree_DFS(L)
```

Transformation of an irreducible CFG into a reducible one

$$\begin{aligned} E' \;=\; & E \setminus LoopEdges(L) \setminus EntryEdges(L) \cup \{(s, \delta_L) \mid s \in PreEntries(L)\} \\ & \cup \{(s, \delta_L) \mid \exists (s, h) \in LoopEdges(L)\} \cup \{(\delta_L, h) \mid h \in LoopHeaders(L)\} \end{aligned}$$



G: Irreducible



$\Psi_L(G)$: Reducible

Key results to analyze liveness in irreducible CFGs

Lemma 4

If d dominates u in G, d dominates u in $\Psi_L(G)$.

Theorem 5

Let v be an SSA variable, G a CFG, transformed into $\Psi_L(G)$ when considering a loop L of a loop forest of G. Then, for each node q of G, v is live-in (resp. live-out) at q in G iff v is live-in (resp. live-out) at q in $\Psi_L(G)$.

☛ HnCA(B,S): highest non common ancestor (in the loop forest) of B and S, i.e., highest ancestor of S that is not ancestor of B.

Algorithm 9: DAG_DFS(block B) /* if loops have one header */

for each $S \in \mathrm{succs}(B)$ if (B, S) is not a loop-edge **do**
    if S is unprocessed **then**
        $T = \text{HnCA}(B, S)$;
        DAG_DFS(T)

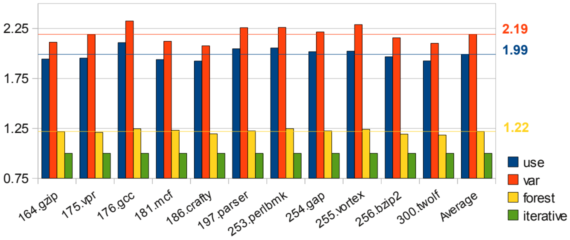
$\mathit{Live} = \mathrm{PhiUses}(B)$ /* *used by ϕ-functions in B's successors* */
for each $S \in \mathrm{succs}(B)$ if (B, S) is not a loop-edge **do**
    $T = \text{HnCA}(B, S)$;
    $\mathit{Live} = \mathit{Live} \cup (\mathrm{LiveIn}(T) \setminus \mathrm{PhiDefs}(T))$

$\mathrm{LiveOut}(B) = \mathit{Live}$;
for each program point p in B, backward **do**
    remove variables defined at p from *Live*;
    add uses at p in *Live*

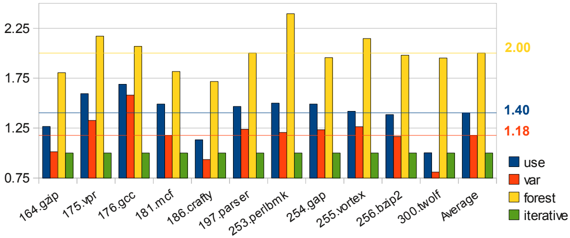
$\mathrm{LiveIn}(B) = \mathit{Live} \cup \mathrm{PhiDefs}(B)$;
mark B as processed

Experimental results

Speed-up w.r.t. iterative data-flow, unoptimized programs, bitsets.

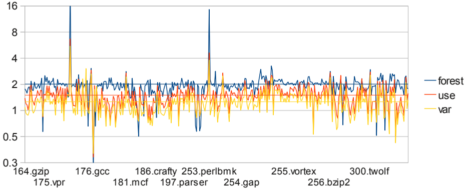


Speed-up w.r.t. iterative data-flow, optimized programs, bitsets.



Experimental results

Speed-up w.r.t iterative data-flow, for optimized programs, with bitsets.



Ratio of the different phases in the forest-based algorithm (forward & backward passes, computation of PhiUses & PhiDefs sets, initialization), bitsets, unoptimized & optimized programs.