



An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile

Hucheng Zhou
Tsinghua University
June 2011

Joint work with: Wenguang Chen (Tsinghua University),
Fred Chow (ICube Technology Corp.)



Contents

Basic Concepts

PRE

SSA

SSAPRE

Speculative Code Motion

MC-SSAPRE

Algorithm

Complexity

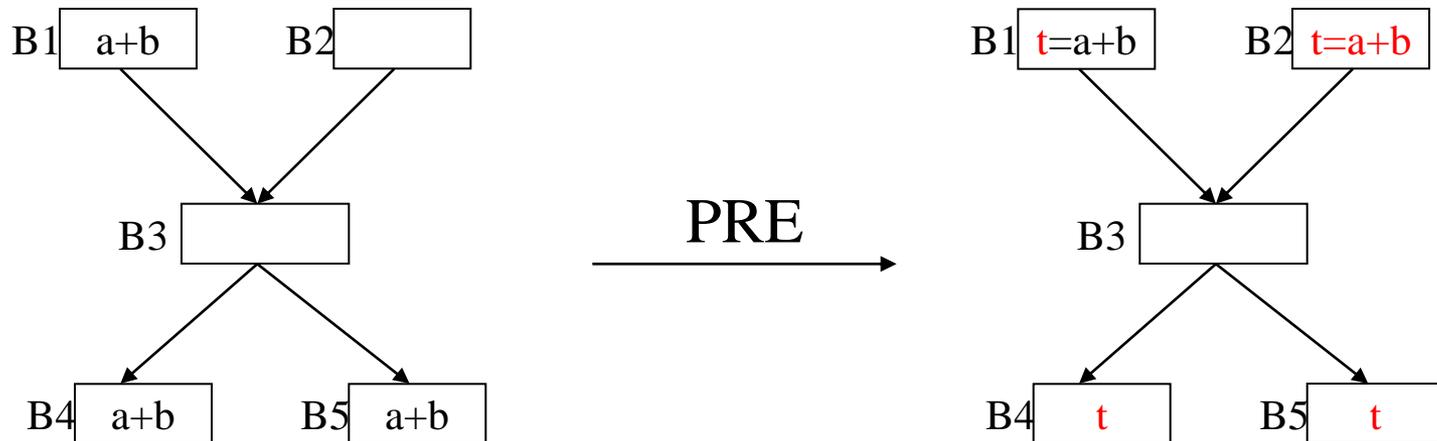
Experiments

Conclusion



Partial Redundancy Elimination (PRE)

- Eliminates expressions redundant on some (not necessarily all) paths
- One of the most important and widely applied target-independent global optimization
- Subsumes global common subexpression and loop invariant code motion





PRE Facts

- Applied to each lexically identified expression independently
 - e.g $(a+b)$, $(a-b)$, $(a*c)$
- Formulated as a Placement problem:
 - Step 1 – Determine where to perform insertions
 - Render more computations fully redundant
 - Step 2 – Delete fully redundant computations
- Main challenge is in Step 1



The Most Popular PRE Algorithms

Lazy Code Motion (Knoop *et. al*)

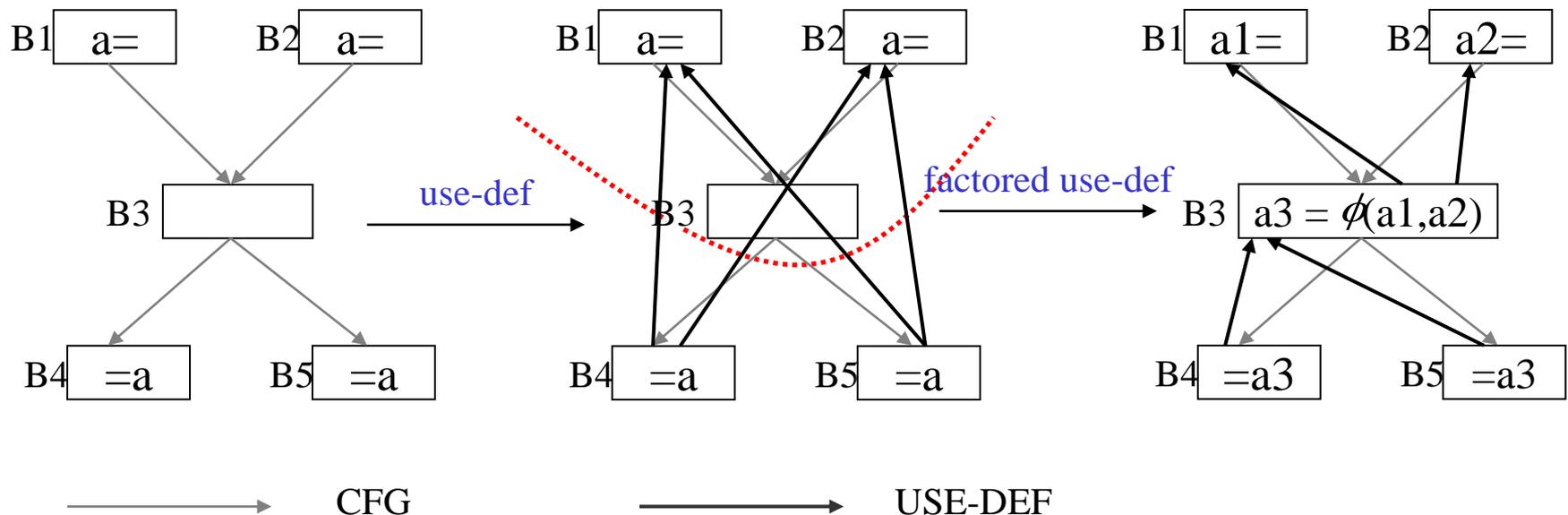
- Computationally and Life-time Optimal
- Ordinary program representation
- Bit-vector-based iterative data flow analyses

SSAPRE

- Computationally and Life-time Optimal
- SSA form of program representation
- Sparse solution of data flow properties
- Subsumes local common subexpression
 - Insensitive to basic block boundaries

Static Single Assignment (SSA)

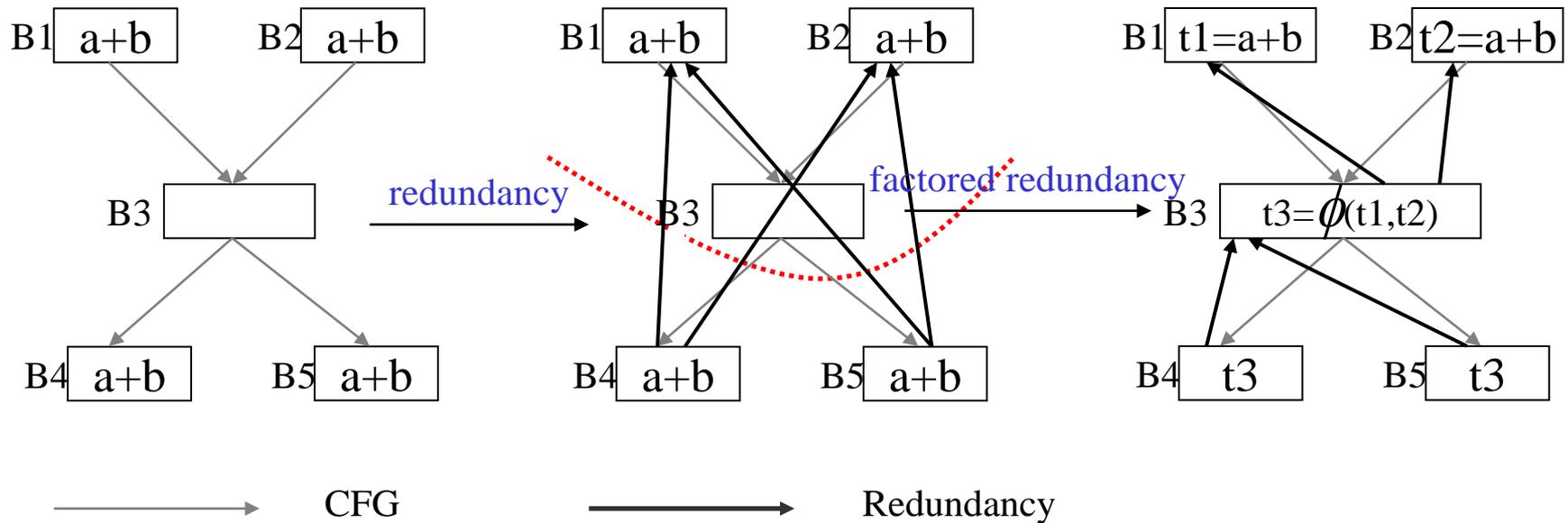
- Program representation with *built-in* use-def information
- Use-def edges factored at join points in CFG
- Use-def implicitly represented via unique names
- Each renamed variable has only one definition





Factored Redundancy Graph (FRG)

- Used in SSAPRE to represent redundant relationships among occurrences of the same expression via edges
- The redundancy edges are factored as in SSA
- Can view as SSA applied to expressions
 - Effectively put the t storing the expression after PRE in SSA form



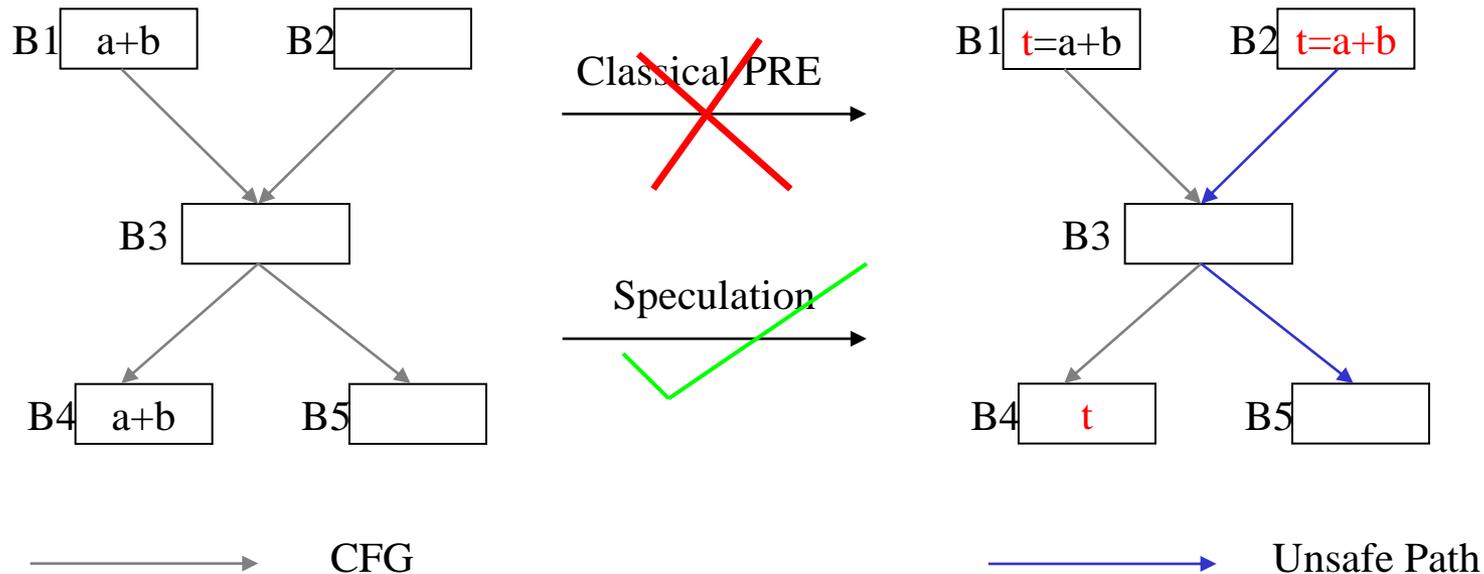
Speculative Code Motion

Classical PRE only inserts at places where the expression is anticipated (down-safe)

- Many redundant computations cannot be eliminated

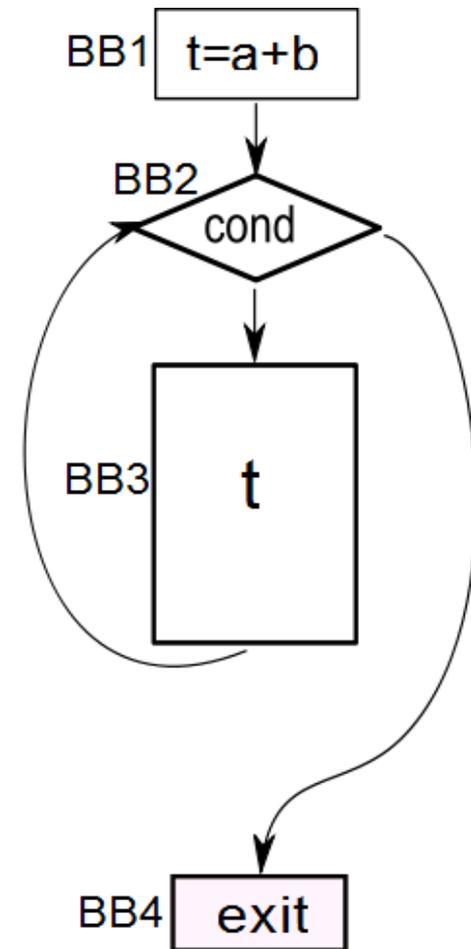
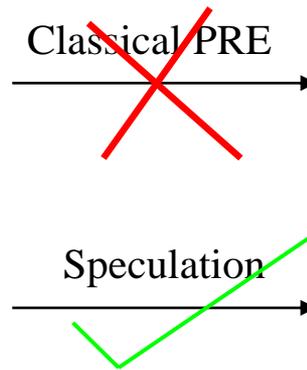
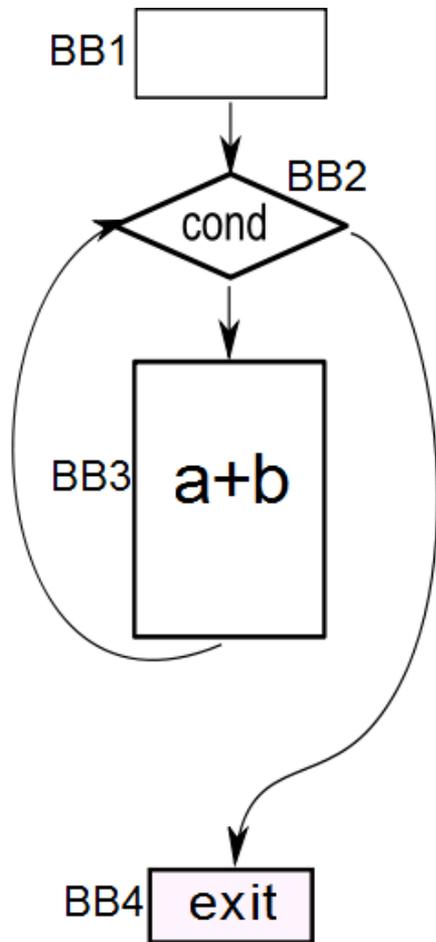
Speculative code motion *ignores* safety constraint

- Can remove more redundancies
- Not applicable to computations that may trigger runtime exceptions



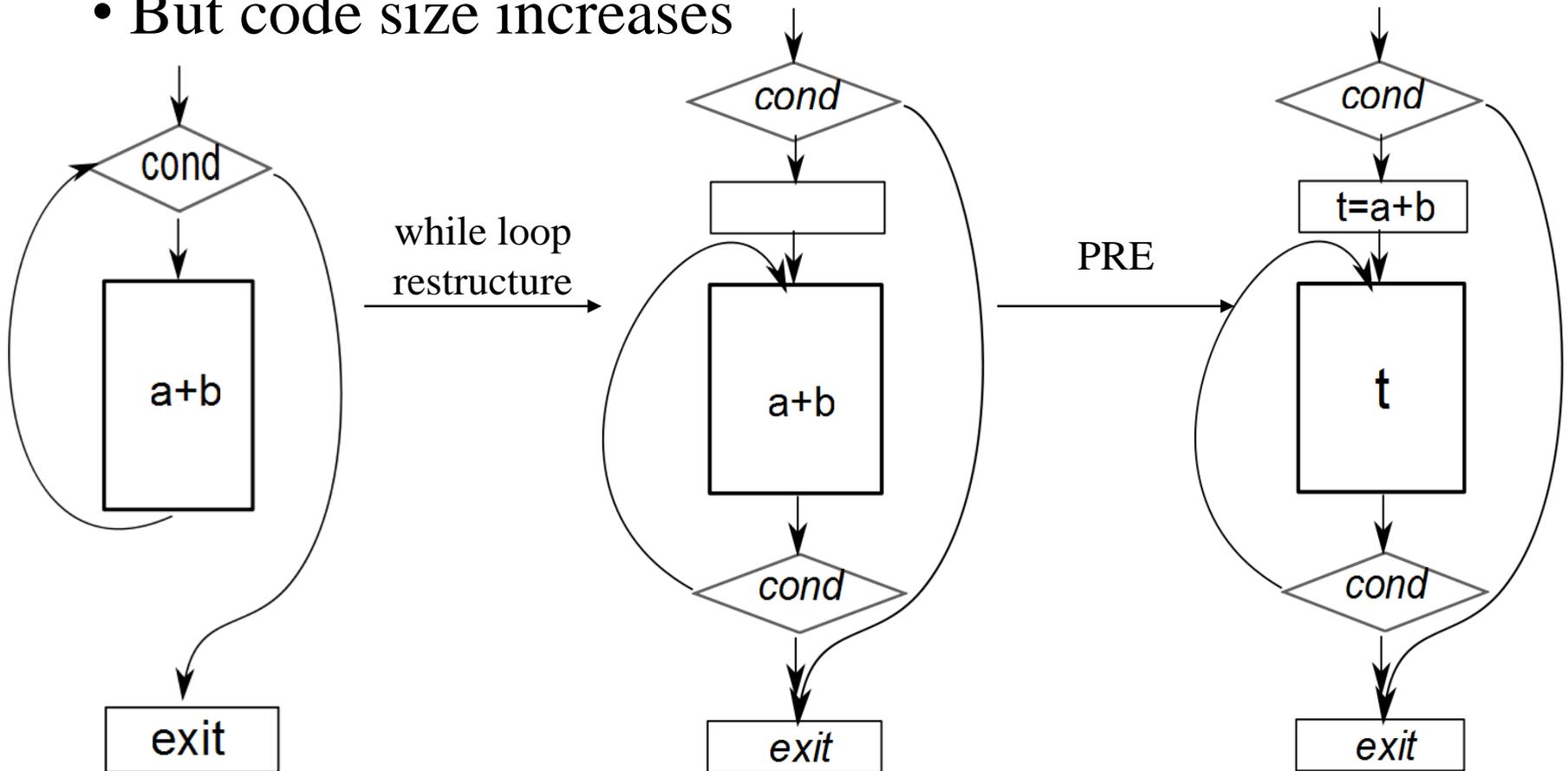
While Loop Example

Invariant code motion involves speculation



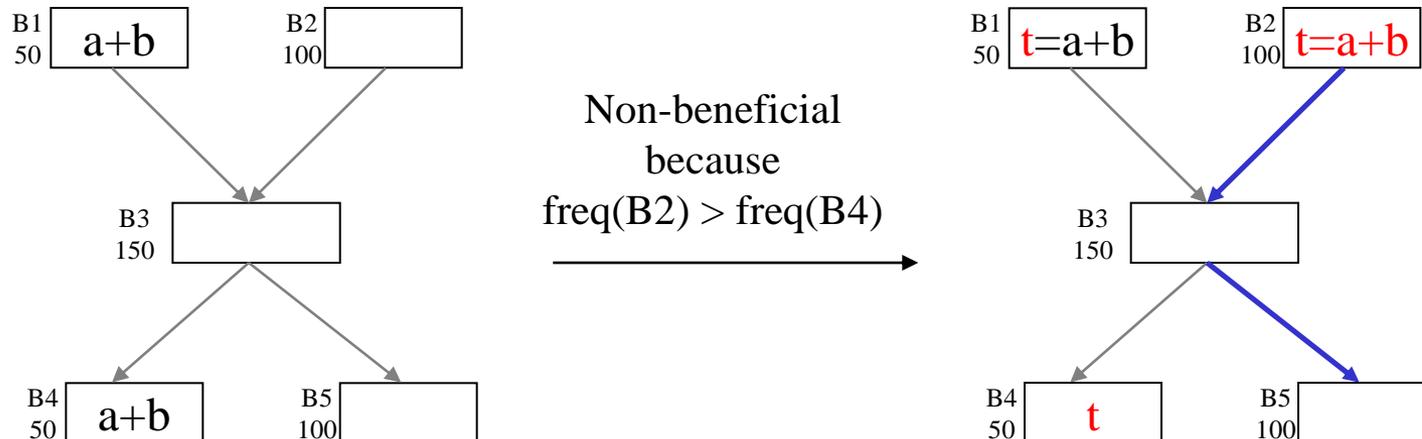
While Loop Restructuring

- The common solution
- Speculation no longer necessary
- But code size increases



Speculation not always beneficial

- Useless computations introduced for some paths
- Beneficial only if removed computations executed more frequently than inserted computations
- Requires execution frequency information





Problem Statement

How to minimize the dynamic execution count of an expression under an execution profile

- A more aggressive form of PRE
 - Classical PRE beneficial regardless of execution frequencies
- Cai and Xue (2003, 2006) first to apply min-cut to solve this problem optimally
 - Algorithm called MC-PRE
 - Uses bit-vector-based data flow analyses
 - Min-cut applied to CFG
- No SSA-based technique exists yet



Topic of this Paper

MC-SSAPRE – a new algorithm that yields optimal code placement under the SSAPRE framework

Overview:

- Form an essential flow graph (EFG) out of the FRG
- Map the BB execution frequencies to the EFG nodes
- Apply min-cut to the EFG



Algorithm Steps

SSAPRE Steps

- Construct FRG
 - Φ insertion
 - Rename
- Data Flow Attributes
 - DownSafety
 - WillBeAvail
- Book-keeping
 - Finalize
 - CodeMotion

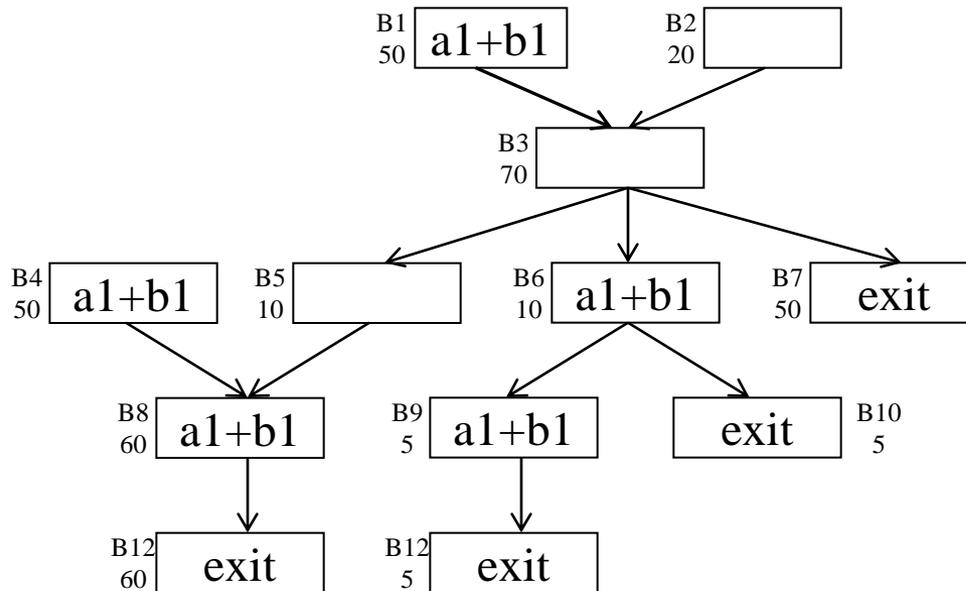
MC-SSAPRE Steps

- Construct FRG
 - Φ insertion
 - Rename
- Form EFG and perform min-cut
 - Data flow
 - Graph reduction
 - Single source
 - Single sink
 - Minimum cut
 - WillBeAvail
- Book-keeping
 - Finalize
 - CodeMotion



Running example in SSA Form

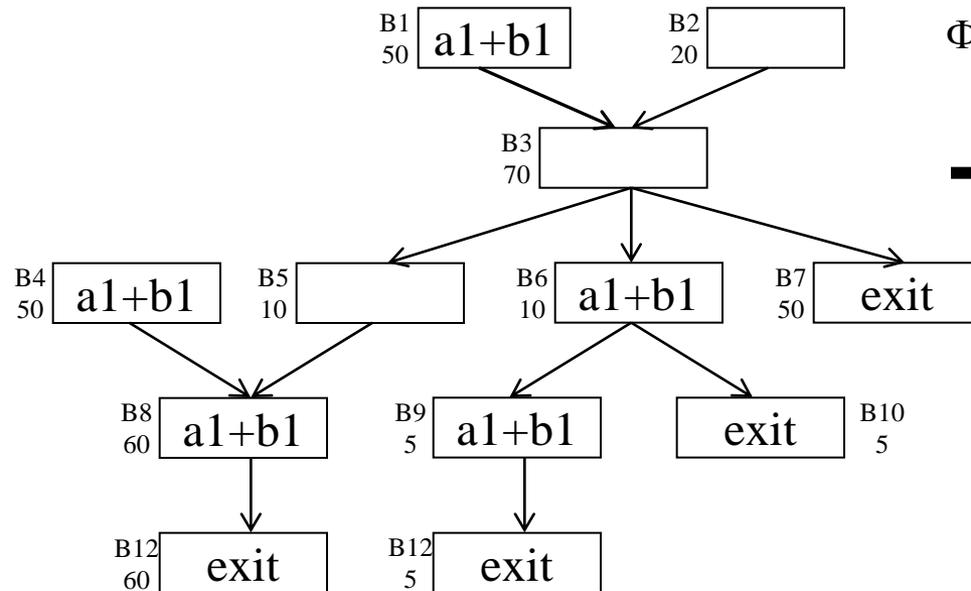
Input Program



FRG for Running Example

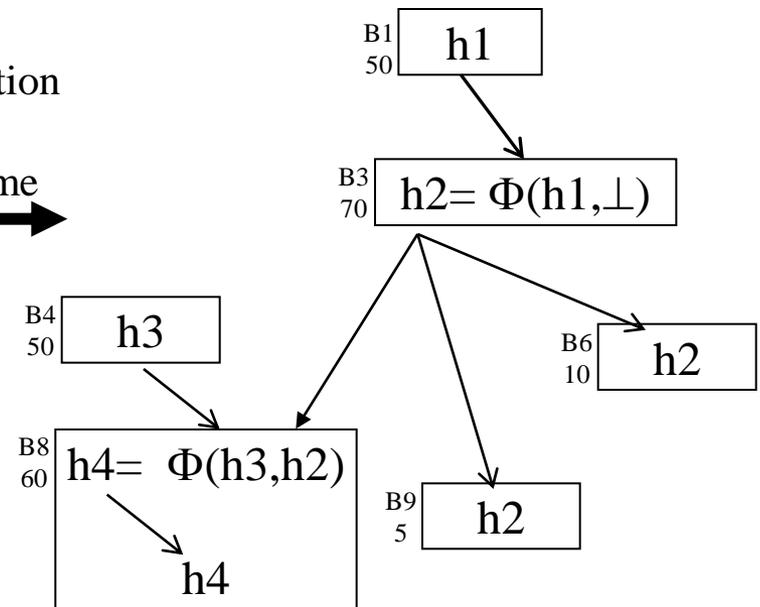
Introduce h so the FRG can be viewed from an SSA perspective

Input Program



Φ Insertion
and
Rename
→

FRG





Roles of Factored Redundancy Graph

- Insertions need to be considered only at Φ 's
 - associated with the Φ operands
- Medium to compute data flow properties to disqualify more Φ 's from being insertion candidates
- SSA form for t (temporary to store the computed value) will be carved out of the FRG
- Three kinds of nodes:
 1. Real occurrences in original program
 - Def – always non-redundant
 - Use – partially redundant (including fully redundant)
 2. Φ (def)
 3. Φ operand (use) – can be \perp



Data Flow Properties for MC-SSAPRE

Fully available

- Insertions at these Φ 's always unnecessary because the computed values are available

Partially anticipated

- Insertions should only be at these Φ 's
- otherwise, the inserted computation would have no use



Graph Reduction

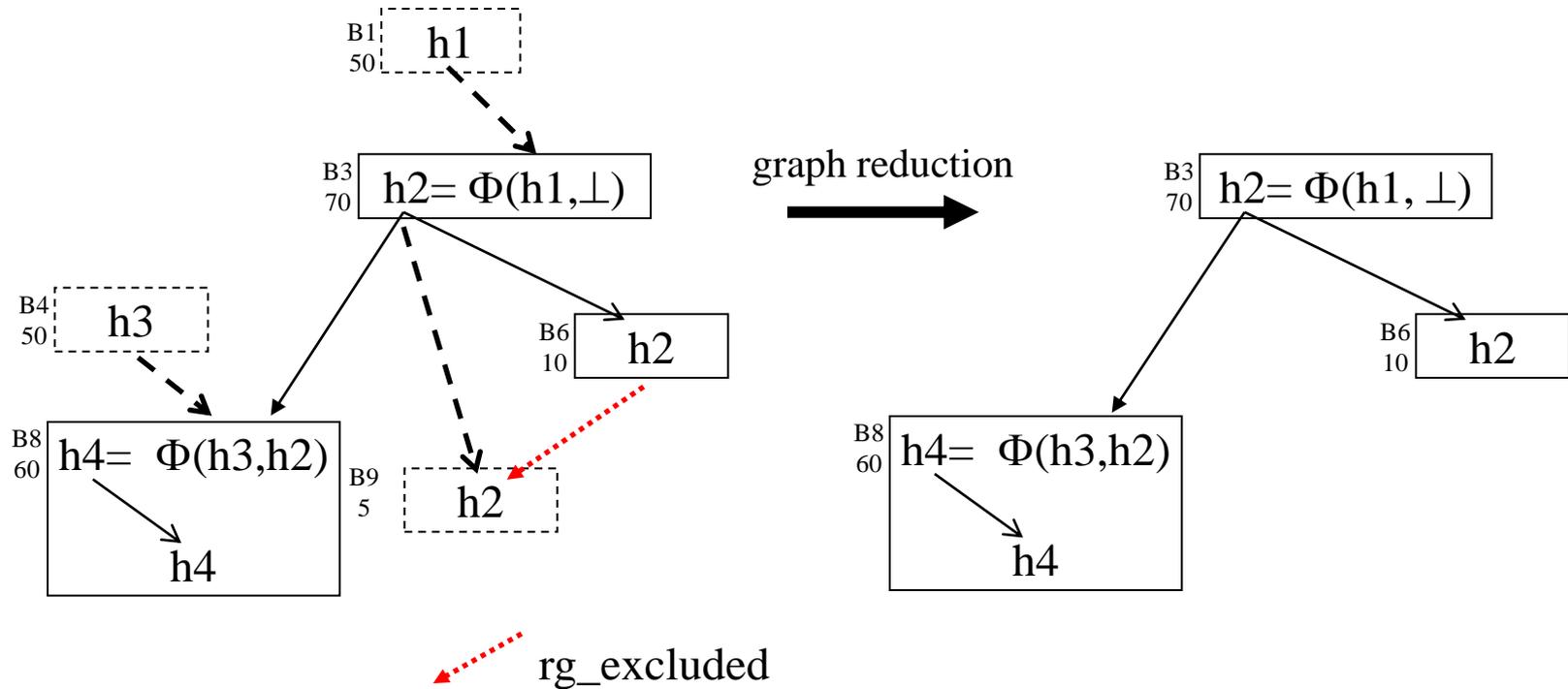
Use computed data flow properties to further narrow down the Φ candidates for insertion

Delete:

- Φ 's that are fully available
- Φ 's that are not partial anticipated
- Use nodes (real occurrences or Φ operands) that are fully redundant
- Edges from/to above nodes



Graph Reduction for Running Example

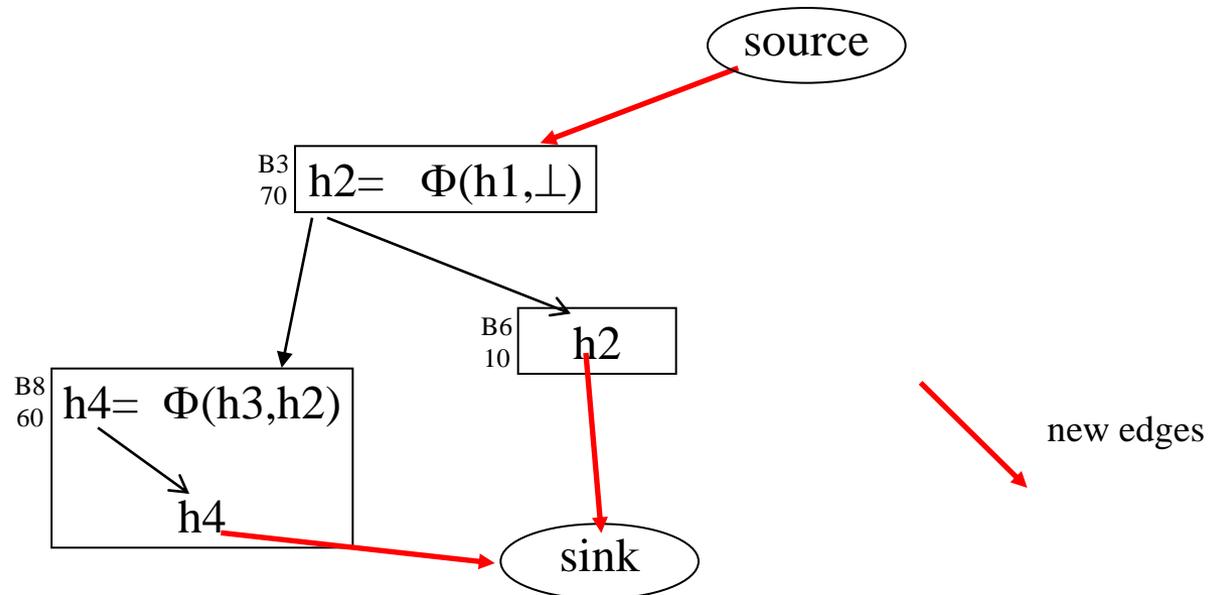


`rg_excluded` – fully redundant occurrences determined during Renaming



Form Essential Flow Graph (EFG)

- Introduce a virtual source node
 - Add an edge from it to each $\perp \Phi$ operand
- Introduce a virtual sink node
 - Add an edge from each real occurrence to it
- Result is a complete flow network



Edges in EFG

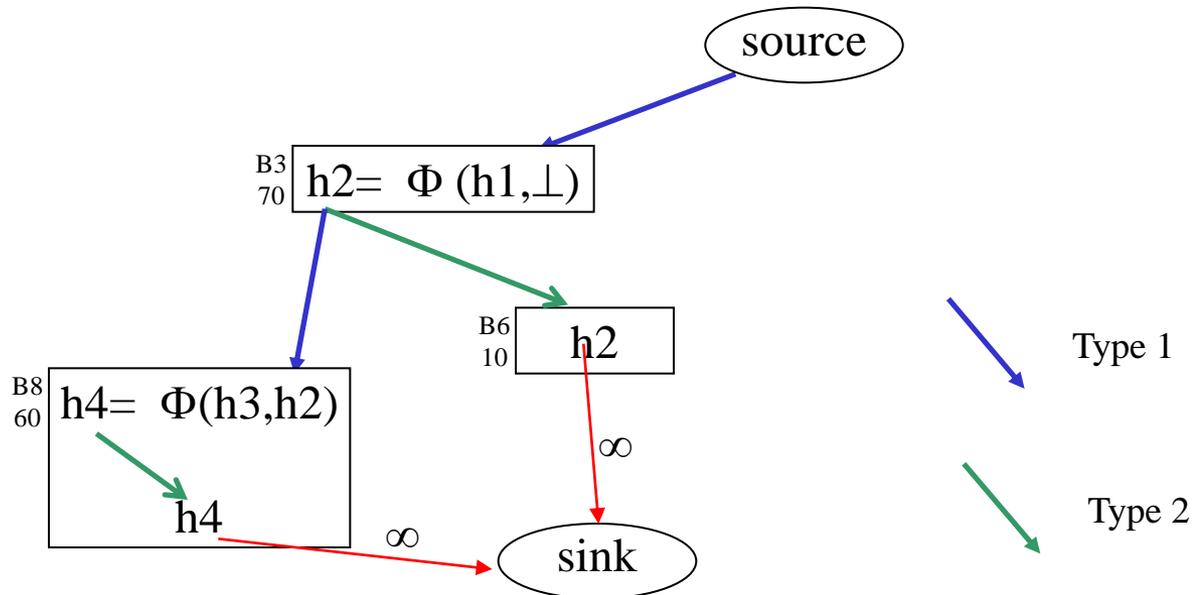
Edges to the sink are never insertion candidate

- Mark with ∞ frequency

Other edges are:

Type 1 edge – Edges ending at a Φ operand

Type 2 edge – Edges from a Φ to a real occurrence



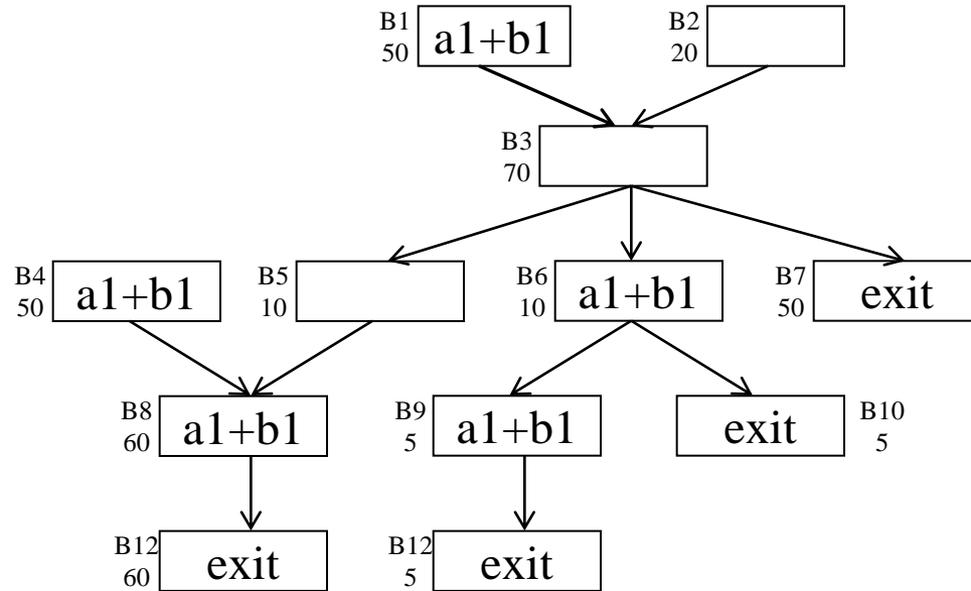


Mapping Frequencies to EFG Edges

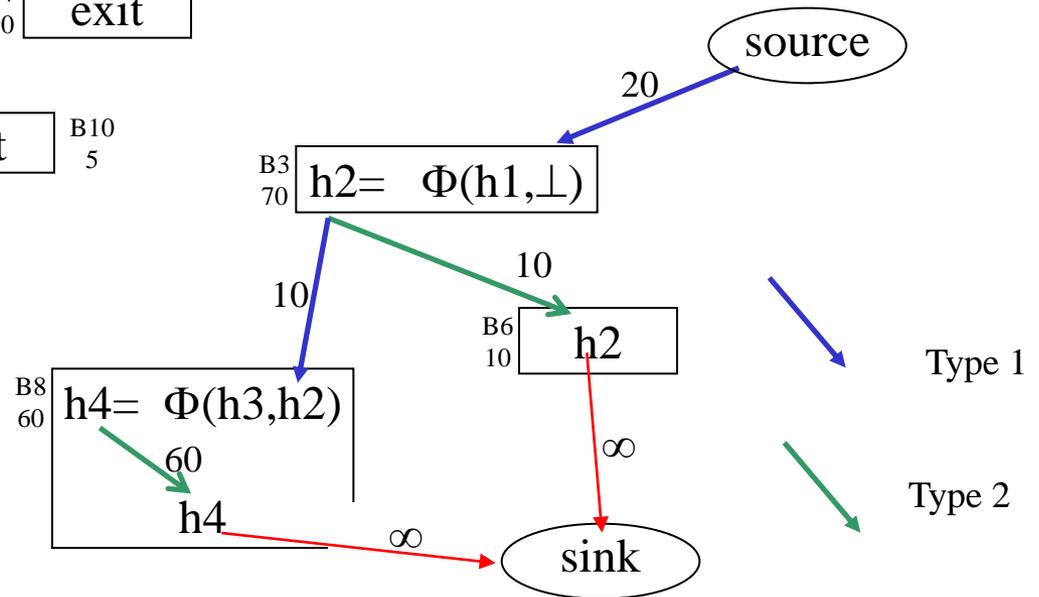
- Model insertion at a Type 1 edge by inserting at exit of the predecessor BB corresponding to the Φ operand
 - Annotate the Type 1 edge by the node frequency of that predecessor BB
- Insertion at a Type 2 edge means performing the computation *in place*
 - Annotate the Type 2 edge by the frequency of the real occurrence

EFG annotated with Frequencies

Original Program



Final EFG





Performing Minimum Cut

A minimum cut

- *separates the flow network into two halves, such that*
- *the sum of the weights of the cut edges is minimized*

By performing insertions at the cut edges, the number of execution of the computation is minimized

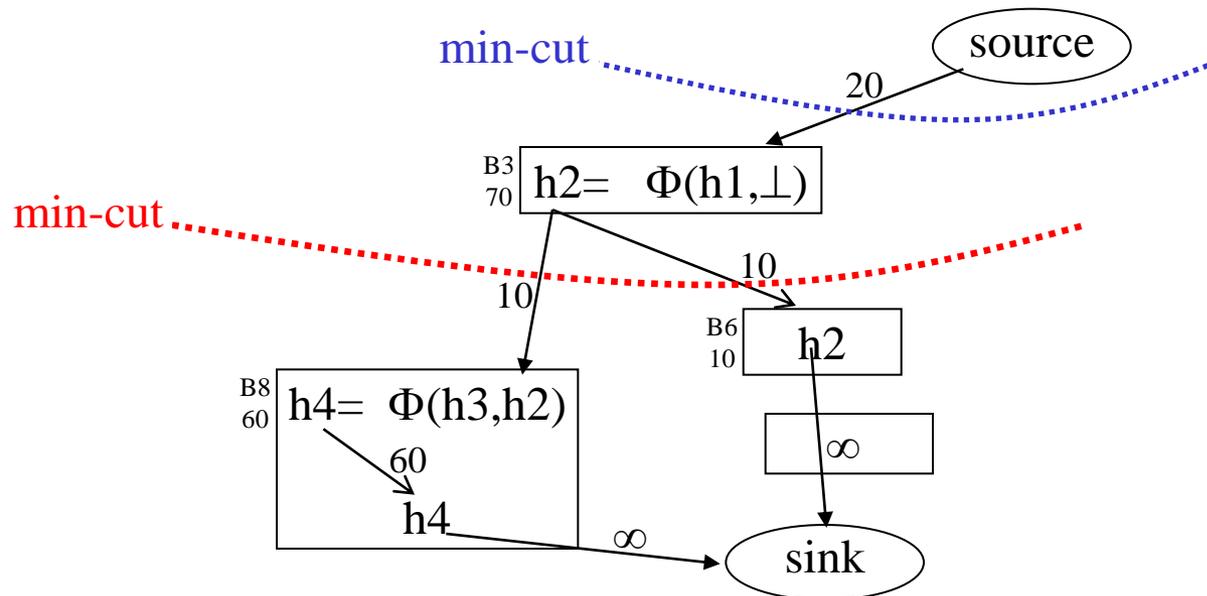
– Implies computational optimality

If min-cut not unique, choose the cut nearest the sink

– Induces life-time optimality

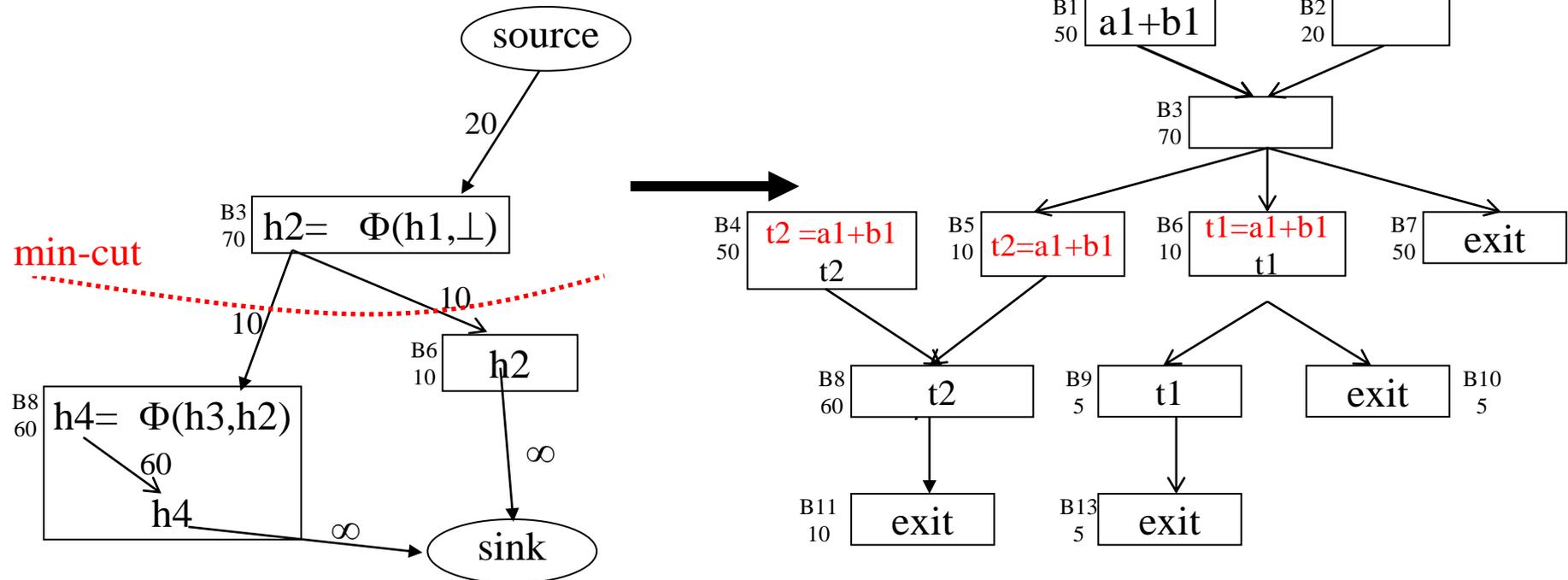
Our Example

- Two possible min-cuts
- Pick later red one



Final Result

final transformed program





Complexity of MC-SSAPRE

MC-SSAPRE Steps

- Construct FRG
 - Φ insertion
 - Rename
- Form EFG and perform min-cut
 - Data flow
 - Graph reduction
 - Single source
 - Single sink
 - Minimum cut
 - WillBeAvail
- Book-keeping
 - Finalize
 - CodeMotion

V – number of FRG nodes

E – number of FRG edges

- Except the minimum cut step, all the steps are $O(V+E)$
- Performing minimum cut is $O(V^2\sqrt{E})$
- In general,

$$V_{\text{cfg}} > V_{\text{frg}} > V_{\text{efg}}$$



Our Implementation

- Implemented MC-SSAPRE in the open source Path64 compiler, a descendent of the compiler with the original SSAPRE
- Leveraged existing SSAPRE infrastructure
- Resulting compiler will perform:
 - SSAPRE when no profile available
 - Perform speculation for loop-invariant computations
 - MC-SSAPRE with profile data
- Compiler always restructures **while** loops

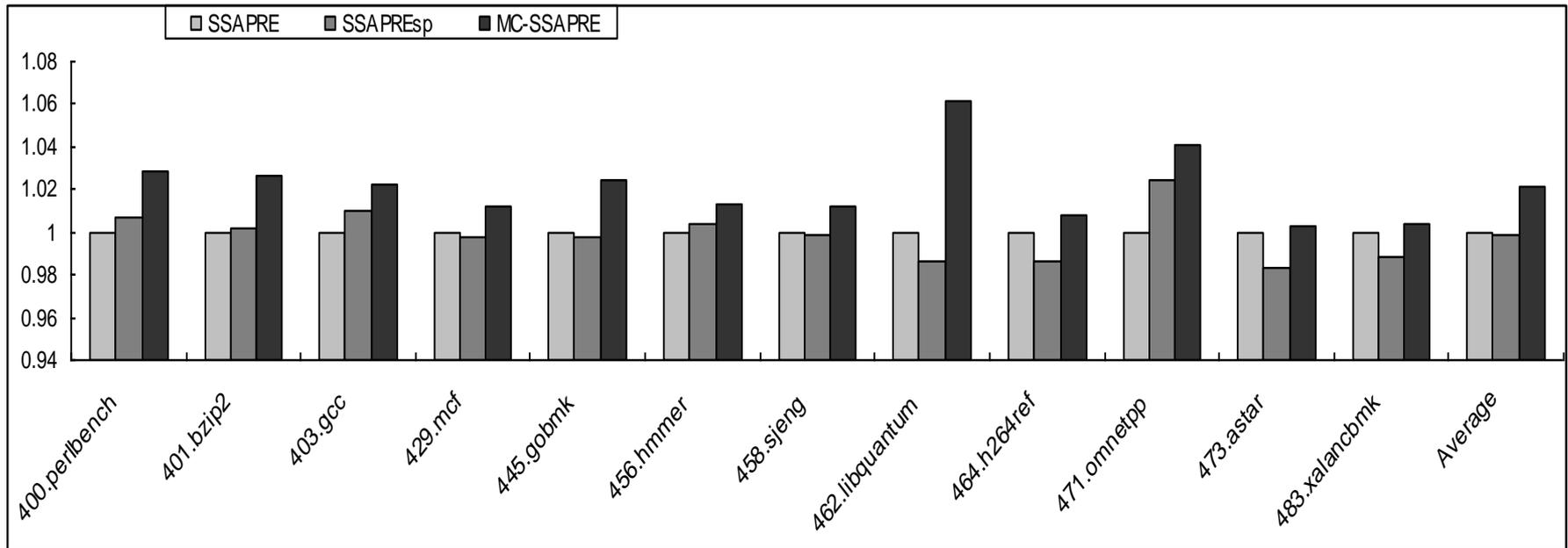


Setup of Experiment 1

- Target is Intel Core™ i7-970 at 2.67GHz with 8MB cache
- Ubuntu 9.10
- With 6GB on board memory
- Compare run-time performances of all of SPEC CPU2006 (29 benchmarks)
- The 3 runs:
 - SSAPRE – no speculation, no profile data
 - SSAPRE_{sp} – loop-based speculation, no profile data
 - MC-SSAPRE – speculation based on profile data

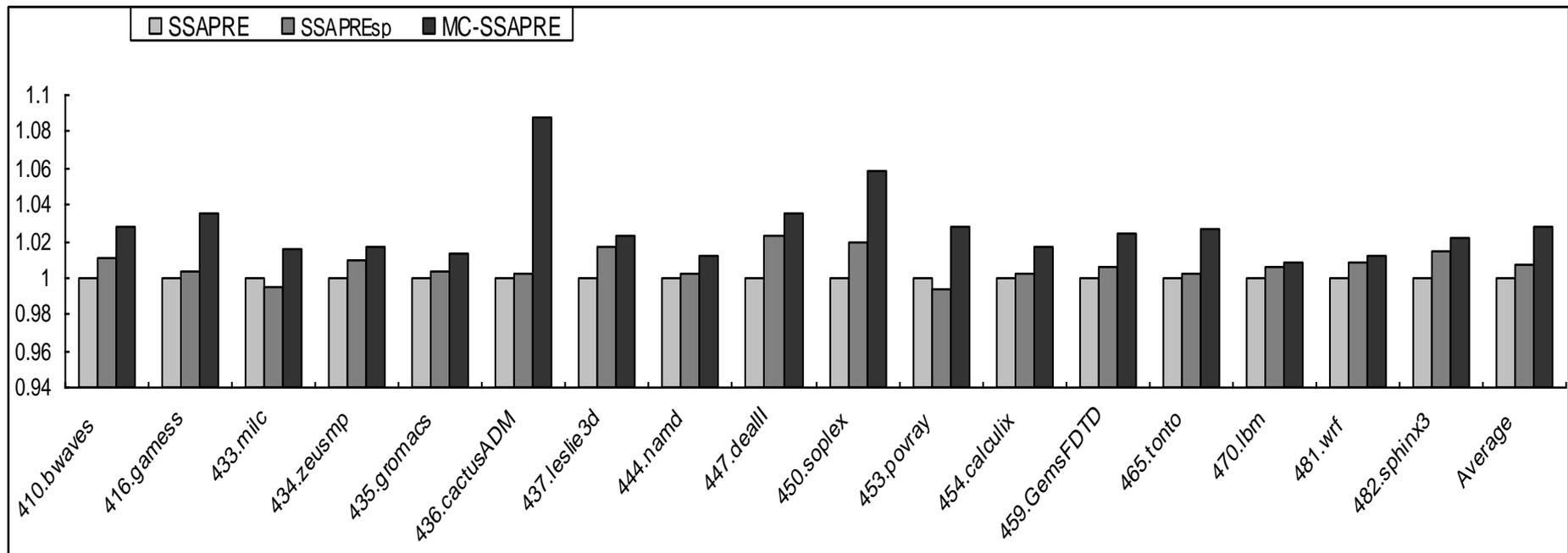
Experimental Results – CINT2006

- Average speedup of 2.13% over SSAPRE
- Average speedup of 2.25% over SSAPRE_{sp}



Experimental Results – CFP2006

- Average speedup of 2.76% over SSAPRE
- Average speedup of 1.96% over SSAPRE_{sp}



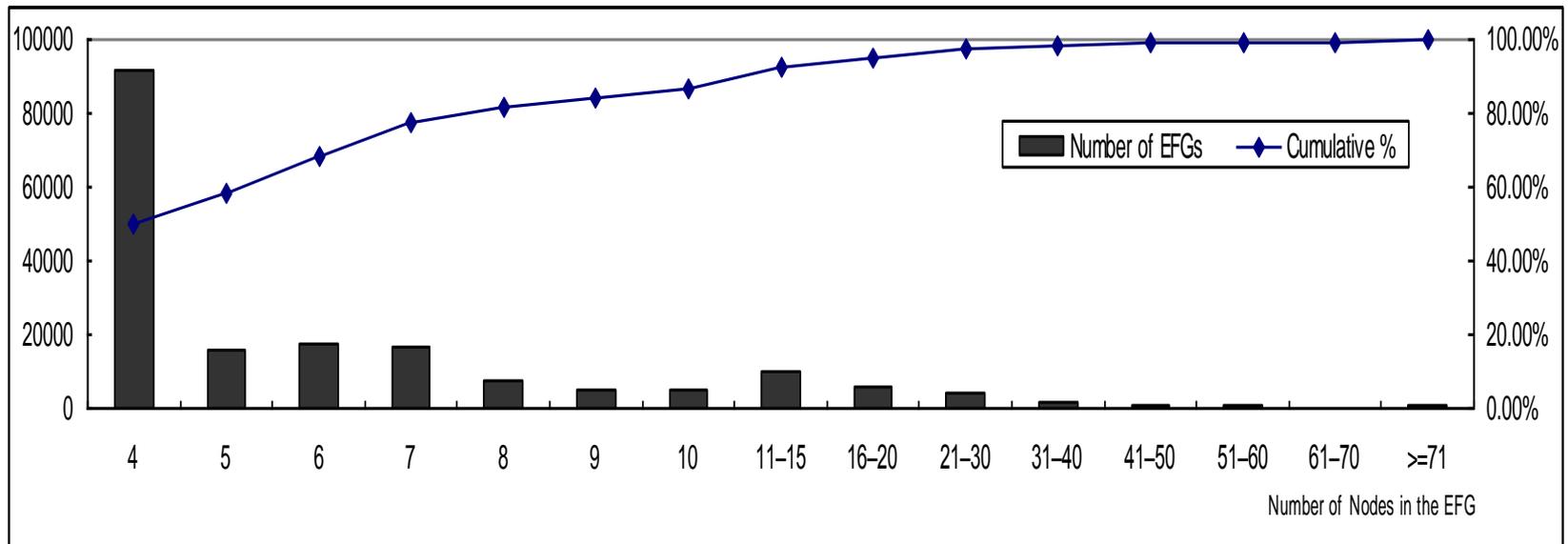


Setup of Experiment 2

- Calculate size of EFGs formed during MC-SSAPRE
- Same 29 SPEC CPU2006 benchmarks
- Target-independent
- Show
 - Optimization overhead in MC-SSAPRE
 - Impact of sparse approach
- Exclude empty EFGs
- Smallest EFG is 4 nodes:
 - Source, sink, Φ , real occurrence

Sizes of EFGs

- 183152 EFGs in the 29 SPEC CPU2006 benchmarks
- Near 50% of EFGs are only 4 nodes
- 86.5% of EFGs are less than 10 nodes
- 99.0% of EFGs are less than 50 nodes
- 24 EFGs larger than 300 nodes (largest size is 805)





Conclusion

- The minimum-cut technique for flow networks can effectively be applied to SSA graphs
- SSA-based compilers can apply MC-SSAPRE to achieve optimal speculative code motion under an execution profile
- The sparse approach is effective in reducing the problem sizes
- The polynomial time complexity of Min-cut only has limited effect on MC-SSAPRE's optimization efficiency
- MC-SSAPRE always improves program performance over SSAPRE



Questions?