Example Transformations on SSA Form

Dead, Clean, and Constant Propagation

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Citation numbers refer to entries in the EaC2e bibliography.
Dead Code Elimination

Three distinct problems

• Useless operations
  ♦ Any operation whose value is not used in some visible way
  ♦ Use the SSA-based mark/sweep algorithm (DEAD)

• Useless control flow
  ♦ Branches to branches, empty blocks
  ♦ Simple CFG-based algorithm (CLEAN)

• Unreachable blocks
  ♦ No path from $n_0$ to $b \Rightarrow b$ cannot execute
  ♦ Simple graph reachability problem
The “DEAD” Algorithm

Using SSA – Dead code elimination

---

**Mark**
for each op i
  clear i’s mark
  if i is critical then
    mark i
    add i to WorkList

while (Worklist ≠ Ø)
  remove i from WorkList
  (i has form “x←y op z”)
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList
  for each b ∈ RDF(block(i))
    mark the block-ending branch in b
    add it to WorkList

**Sweep**
for each op i
  if i is not marked then
    if i is a branch then
      rewrite with a jump to i’s nearest useful post-dominator
    if i is not a jump then
      delete i

---

**Notes:**
- Eliminates some branches
- Reconnects dead branches to the remaining live code
- Find useful post-dominator by walking post-dom tree
  > Entry & exit nodes are always “useful”
Eliminating Useless Control Flow

**The Problem**
- After optimization, the CFG can contain empty blocks
- “Empty” blocks still end with either a branch or a jump
- Produces jump to jump, which wastes time & space
- Need to simplify the CFG & eliminate these

**The CLEAN Algorithm**
- Use four distinct transformations
- Apply them in a carefully selected order
- Iterate until done

Devised by Rob Shillingsburg (1992), documented by John Lu (1994)
Eliminating Useless Control Flow

Transformations in CLEAN

Both sides of branch target $B_i$
- Neither block must be empty
- Replace it with a jump to $B_i$
- Simple rewrite of last op in $B_1$

How does this happen?
- Rewriting other branches

How do we find it?
- Check each branch
Eliminating Useless Control Flow

Transformations in CLEAN

- Merging an empty block
  - Empty $B_1$ ends in a jump
  - Coalesce $B_1$ with $B_2$
  - Move $B_1$’s incoming edges
  - Eliminates extraneous jump
  - Faster, smaller code

How does this happen?
- Eliminate operations in $B_1$

How do we find it?
- Test for empty block
Eliminating Useless Control Flow

Transformations in CLEAN

Coalescing blocks
- Neither block must be empty
- $B_1$ ends with a jump
- $B_2$ has 1 predecessor
- Combine the two blocks
- Eliminates a jump

How does this happen?
- Simplifying edges out of $B_1$

How do we find it?
- Check target of jump $|\text{preds}|$

Combining non-empty blocks

$B_1$ and $B_2$ should be a single basic block
If one executes, both execute, in linear order.
Eliminating Useless Control Flow

Transformations in CLEAN

- Jump to a branch
  - $B_1$ ends with jump, $B_2$ is empty
  - Eliminates pointless jump
  - Copy branch into end of $B_1$
  - Might make $B_2$ unreachable

How does this happen?
- Eliminating operations in $B_2$

How do we find this?
- Jump to empty block

Hoisting branches from empty blocks
Eliminating Useless Control Flow

**Putting the transformations together**

- Process the blocks in postorder
  - Clean up $B_i$’s successors before $B_i$
  - Simplifies implementation & understanding
- At each node, apply transformations in a fixed order
  - Eliminate *redundant* branch
  - Eliminate *empty* block
  - Merge block with successor
  - Hoist branch from empty successor
- May need to iterate
  - Postorder $\Rightarrow$ unprocessed successors along back edges
  - Can bound iterations, but a deriving tight bound is hard
  - Must recompute postorder between iterations

Montonicity is not obvious
Eliminating Useless Control Flow

What about an empty loop?

• By itself, **CLEAN** cannot eliminate the loop
• Loop body branches to itself
  ♦ Branch is **not** redundant *
  ♦ Doesn’t end with a jump
  ♦ Hoisting does not help *
• Key is to eliminate self-loop
  ♦ Add a new transformation? *
  ♦ Then, \( B_1 \) merges with \( B_2 \) *

New transformation must recognize that \( B_1 \) is empty. Presumably, it has code to test exit condition & (probably) increment an induction variable.

This requires looking at code inside \( B_1 \) and doing some sophisticated pattern matching. This seems awfully complicated.
Eliminating Useless Control Flow

What about an empty loop?
• How to eliminate \(<B_1, B_1>\) ?
  ♦ Pattern matching ?
  ♦ Useless code elimination ?
Eliminating Useless Control Flow

What about an empty loop?
• How to eliminate $<B_1, B_1>$?
  ♦ Pattern matching?
  ♦ Useless code elimination?

• What does DEAD do to $B_1$?
  ♦ Remember, it is empty
  ♦ Contains only the branch
  ♦ $B_1$ has only one exit
  ♦ So, $B_1 \notin RDF(B_2)$
  ♦ $B_1$’s branch is useless
  ♦ DEAD rewrites it as a jump to $B_2$
Using SSA – Dead code elimination

Mark
for each op i
  clear i’s mark
  if i is critical then
    mark i
    add i to WorkList

while (Worklist ≠ Ø)
  remove i from WorkList
  (i has form “x ← y op z”)
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList
  for each b ∈ RDF(block(i))
    mark the block-ending branch in b
    add it to WorkList

Sweep
for each op i
  if i is not marked then
    if i is a branch then
      rewrite with a jump to i’s nearest useful post-dominator
    if i is not a jump then
      delete i

Notes:
• Eliminates some branches
• Reconnects dead branches to the remaining live code
• Find useful post-dominator by walking post-dom tree
  > Entry & exit nodes are always “useful”

Added to the algorithm by Shillingsburg
Eliminating Useless Control Flow

What about an empty loop?
• How to eliminate $<B_1,B_1>$?
  ♦ Pattern matching?
  ♦ Useless code elimination?

• What does **DEAD** do to $B_1$?
  ♦ Remember, it is empty
  ♦ Contains only the branch
  ♦ $B_1$ has only one exit
  ♦ So, $B_1 \notin RDF(B_2)$
  ♦ $B_1$’s branch is **useless**
  ♦ **DEAD** rewrites it as a jump to $B_2$

**DEAD** converts the empty loop to a form where **CLEAN** handles it!
Eliminating Useless Control Flow

The Algorithm

CleanPass()
  for each block $i$, in postorder
    if $i$ ends in a branch then
      if both targets are identical then
        rewrite with a jump
    if $i$ ends in a jump to $j$ then
      if $i$ is empty then
        merge $i$ with $j$
      else if $j$ has only one predecessor
        merge $i$ with $j$
      else if $j$ is empty & $j$ has a branch then
        rewrite $i$’s jump with $j$’s branch
Clean()
  until CFG stops changing
  compute postorder
  CleanPass()

Summary

• Simple, structural algorithm
• Limited transformation set
• Cooperates with DEAD
• In practice, its quite fast

How many calls to CleanPass are needed before CLEAN halts?
• Clearly a fixed point algorithm
• Answer is not obvious
Eliminating Useless Control Flow

Putting the transformations together

• Process the blocks in postorder
  ♦ Clean up $B_i$’s successors before $B_i$
  ♦ Simplifies implementation & understanding

• At each node, apply transformations in a fixed order
  ♦ Eliminate redundant branch
  ♦ Eliminate empty block
  ♦ Merge block with successor
  ♦ Hoist branch from empty successor

• May need to iterate
  ♦ Postorder $\Rightarrow$ unprocessed successors along back edges
  ♦ Can bound iterations, but deriving tight bound is hard
  ♦ Must recompute postorder between iterations

Montonicity is not obvious

*
Eliminating Unreachable Code

The Problem
• Block with no entering edge
• Situation created by other optimizations

The Cure
• Compute reachability & delete unreachable code
• Simple mark/sweep algorithm on CFG
• Mark during computation of postorder, reverse postorder ...
• In MSCP, importing ILOC did this (every time)
Dead Code Elimination

Summary

- Useless Computations $\Rightarrow$ DEAD
- Useless Control-flow $\Rightarrow$ CLEAN
- Unreachable Blocks $\Rightarrow$ Simple housekeeping

Other Transformations that eliminate dead code

- Constant propagation can eliminate some branch targets
- Algebraic identities & redundancy elimination make some operations useless or outright remove them (depends on implementation style)

Use of SSA Form

- DEAD used SSA form as a convenient way to get DEF-USE chains
- CLEAN operated on the CFG without much regard to contents of a block
Constant Propagation

We have seen two formulations of constant propagation

• Classical formulation as a global data-flow problem
  ♦ Annotate each node in the CFG (each block) with a \texttt{CONSTANTS} set
  ♦ Complicated transfer functions to model effect of single op
  ♦ Compose transfer function of individual ops to get function for entire block
    → \textit{Resembles a symbolic interpretation}
  ♦ Verdict: conceptually complex and (potentially) slow

• Sparse formulation over the graph formed by \texttt{DEF-USE} chains
  ♦ Each value is treated separately — propagated along chain and used in a meet operation with the values of other defs that reach the same use
  ♦ Algorithm is conceptually simple
  ♦ Argument for termination and speed are based on lattice height, not some transfer function and the \texttt{CFG} structure (e.g., \( d(G) + 3 \) passes a la Kam-Ullman)
  ♦ Verdict: conceptually simpler and (arguably) faster
Constant Propagation

Safety

• Proves that name always has known value at point p
• Specializes code around that value
  ♦ Moves some computations to compile time (⇒ code motion)
  ♦ Exposes some unreachable blocks (⇒ dead code)

Opportunity

• Value ≠ ⊥ signifies an opportunity

Profitability

• Compile-time evaluation is cheaper than run-time evaluation
• Branch removal may lead to block coalescing (CLEAN)
  ♦ If not, it still avoids the test & makes branch predictable
Constant Propagation over DEF-USE Chains

Worklist ← ∅

For i ← 1 to number of operations
  if $in_1$ of operation $i$ is a constant $c_i$
    then $\text{Value}(in_1,i) ← c_i$
    else $\text{Value}(in_1,i) ← T$
  if $in_2$ of operation $i$ is a constant $c_j$
    then $\text{Value}(in_2,i) ← c_j$
    else $\text{Value}(in_2,i) ← T$
  if ($\text{Value}(in_1,i)$ is a constant &
    $\text{Value}(in_2,i)$ is a constant)
    then $\text{Value}(out,i) ← \text{evaluate op } i$
    Worklist ← Worklist ∪ {i}
    else $\text{Value}(out,i) ← T$

  Initialization Step

while (Worklist ≠ ∅)
  remove a definition $i$ from WorkList
  for each $j ∈ \text{USES}(out,i)$
    set $x$ so that $\text{out}$ of $i$ is $in_x$ of $j$
    $\text{Value}(in_x,j) ← \text{Value}(in_x,j) \land \text{Value}(out,,i)$
    if ($\text{Value}(in_1,j)$ is a constant &
      $\text{Value}(in_2,j)$ is a constant)
      then $\text{Value}(out,j) ← \text{evaluate op } j$
      Worklist ← Worklist ∪ {j}
    else if ($\text{Value}(in_1,j)$ is ⊥ or
      $\text{Value}(in_2,j)$ is ⊥)
      then $\text{Value}(out,j) ← ⊥$
      Worklist ← Worklist ∪ {j}

  Propagation Step
Last time we saw constant propagation, we had this algorithm over DEF-USE chains ... 

Constant Propagation over DEF-USE Chains

**Initialization Step**

Worklist $\leftarrow \emptyset$

for $i \leftarrow 1$ to number of operations

if $in_1$ of operation $i$ is a constant $c_i$
then $Value(in_1, i) \leftarrow c_i$
else $Value(in_1, i) \leftarrow T$

if $in_2$ of operation $i$ is a constant $c_j$
then $Value(in_2, i) \leftarrow c_j$
else $Value(in_2, i) \leftarrow T$

if $(Value(in_1, i)$ is a constant &
$Value(in_2, i)$ is a constant)
then $Value(out, i) \leftarrow$ evaluate op $i$
Worklist $\leftarrow$ Worklist $\cup \{i\}$
else $Value(out, i) \leftarrow T$

**Propagation Step**

while (Worklist $\neq \emptyset$)
remove a definition $i$ from WorkList

for each $j \in \text{USES}(out, i)$
let $x$ be operand where $j$ occurs

$Value(in_x, j) \leftarrow Value(in_x, j) \land Value(out, i)$

if $(Value(in_1, j)$ is a constant &
$Value(in_2, j)$ is a constant)
then $Value(out, j) \leftarrow$ evaluate op $j$
Worklist $\leftarrow$ Worklist $\cup \{j\}$
else if $(Value(in_1, j)$ is $\bot$ or
$Value(in_2, j)$ is $\bot$)
then $Value(out, j) \leftarrow \bot$
Worklist $\leftarrow$ Worklist $\cup \{j\}$
**Using SSA — Sparse Constant Propagation** [W&Z, 347]

\[ \forall \text{ expression, } e \]
\[ \text{Value}(e) \leftarrow \begin{cases} \text{TOP} & \text{if its value is unknown} \\ \text{c}_i & \text{if its value is known} \\ \text{BOT} & \text{if its value is known to vary} \end{cases} \]

\[ \forall \text{ SSA edge } s = <u,v> \]
\[ \text{if } \text{Value}(u) \neq \text{TOP} \text{ then} \]
\[ \text{add } s \text{ to WorkList} \]

while (WorkList \neq \emptyset)
\[ \text{remove } s = <u,v> \text{ from WorkList} \]
\[ \text{let } o \text{ be the operation that uses } v \]
\[ \text{if } \text{Value}(o) \neq \text{BOT} \text{ then} \]
\[ t \leftarrow \text{result of evaluating } o \]
\[ \text{if } t \neq \text{Value}(o) \text{ then} \]
\[ \forall \text{ SSA edge } <o,x> \]
\[ \text{add } <o,x> \text{ to WorkList} \]

---

**Evaluating a Ø-function:**
\[ \emptyset(x_1,x_2,x_3, \ldots x_n) \text{ is} \]
\[ \text{Value}(x_1) \land \text{Value}(x_2) \land \text{Value}(x_3) \]
\[ \land \ldots \land \text{Value}(x_n) \]
\[ \text{where} \]
\[ \text{TOP} \land x = x \quad \forall x \]
\[ c_i \land c_j = c_i \quad \text{if } c_i = c_j \]
\[ c_i \land c_j = \text{BOT} \quad \text{if } c_i \neq c_j \]
\[ \text{BOT} \land x = \text{BOT} \quad \forall x \]

---

*Same result, fewer \land operations*

*Performs \land only at \emptyset nodes*
Using SSA — Sparse Constant Propagation

How long does this algorithm take to halt?

• Initialization is two passes
  ♦ $|\text{ops}| + 2 \times |\text{ops}|$ edges

• In propagation, Value(x) can take on 3 values
  ♦ TOP, $c_i$, BOT
  ♦ Each use can be on WorkList twice
  ♦ $2 \times |\text{args}| = 4 \times |\text{ops}|$ evaluations, WorkList pushes & pops

This algorithm is much simpler than the DEF-USE version
Constant Propagation over DEF-USE Chains

Optimism versus Pessimism

Optimism
- This version of the algorithm is an optimistic formulation
- Initializes values to
- Prior version used $\perp^T$ (pessimism)

Clear that $i$ is always 12 at def of $x$

$\text{while ( \ldots )}$

$\text{\ldots}$

$x \leftarrow i \ast 17$

$j \leftarrow i$

$i \leftarrow \ldots$

$\ldots$

$i \leftarrow j$

Constant Propagation over DEF-USE Chains

Optimism versus Pessimism

Optimism

- This version of the algorithm is an optimistic formulation
- Initializes values to $\top$
- Prior version used $\bot$ (pessimism)

Pessimistic initializations

\[
12 \quad i \leftarrow 12 \\
\text{while} \ (\ldots) \\
\ldots \\
\downarrow \\
x \leftarrow i \ast 17 \\
downarrow \\
j \leftarrow i \\
downarrow \\
i \leftarrow \ldots \\
downarrow \\
i \leftarrow j
\]

Leads to

\[
i \leftarrow 12 \land \bot = \bot
\]

Constant Propagation over DEF-USE Chains

Optimism versus Pessimism

Optimism
- This version of the algorithm is an optimistic formulation
- Initializes values to $\top$
- Prior version used $\bot$ (pessimism)

In general
- Optimism helps inside loops
- Determined by the initial value

Sparse Constant Propagation

What happens when SCP propagates a value into a branch?

• **TOP** ⇒ we gain no knowledge
• **BOT** ⇒ either path can execute
• **TRUE** or **FALSE** ⇒ only one path can execute

But, the algorithm does not use this knowledge ...

Using this observation, we can add an element of refining feasible paths to the algorithm that will take it beyond the standard limits of **DFA**

→ Until a block can execute, treat it as unreachable
→ Optimistic initializations allow analysis to proceed with unevaluated blocks

Result is an analysis that can use *limited symbolic evaluation* to combine constant propagation with unreachable code elimination
Sparse Conditional Constant Propagation

Can use constant-valued control predicates to refine the CFG

- If compiler knows the value of $x$, it can eliminate either the then or the else case
  - $(x > 0) \Rightarrow y$ is 17 in $B_3$
  - $(x > 0) \Rightarrow B_2$ is unreachable

- This approach combines constant propagation with CFG reachability analysis to produce better results in each

- Example of Click’s notion of “combining optimizations”
  - Predated & motivated Click

Classic DFA assumes that all paths can be taken at runtime, including $(B_0, B_2, B_3)$
Aside on Combining Optimizations

Sometimes, combining two optimizations can produce solutions that cannot be obtained by solving them independently.

- Requires bilateral interactions between optimizations
  - C. Click and K.D. Cooper, “Combining Analyses, Combining Optimizations”, TOPLAS 17(2), March 1995 [86]

Sparse Conditional Constant Propagation is an example
- Combines constant propagation and unreachable code elimination
- Achieves results that no combination of the two can reach independently
- In the paper, they also suggest combining inline substitution
  - While that idea is nice, it does not achieve the kind of same synergy
  - Inlining followed by SCCP would achieve the same results

Interdependence versus a phase ordering problem
Sparse Constant Propagation

To work simplification of conditionals into the algorithm, requires several modifications:

• Use two worklists:
  ♦ SSAWorkList
    → Holds edges in the SSA graph
    → SSA worklist propagates changing values
  ♦ CFGWorkList
    → Holds edges in the control-flow graph
    → CFG worklist propagates information on reachability

• Do not evaluate operations until block is reachable
• When algorithm marks a block as reachable, must evaluate all operations in the block and propagate their effects forward
Sparse Conditional Constant Propagation

SSAWorkList $\leftarrow \emptyset$
CFGWorkList $\leftarrow n_0$

\forall \text{ block } b
\quad \text{clear } b’s \text{ mark}
\forall \text{ operation } o \text{ in } b
\quad \text{Value}(o) \leftarrow \text{TOP}

Initialization Step

To evaluate a branch
if arg is \text{BOT} then
\quad \text{put both targets on CFGWorkList}
else if arg is \text{TRUE} then
\quad \text{put } \text{TRUE} \text{ target on CFGWorkList}
else if arg is \text{FALSE} then
\quad \text{put } \text{FALSE} \text{ target on CFGWorkList}

To evaluate a jump
\quad \text{place its target on CFGWorkList}

while \((\text{CFGWorkList} \cup \text{SSAWorkList}) \neq \emptyset)\)
while(\text{CFGWorkList} \neq \emptyset)
\quad \text{remove } b \text{ from CFGWorkList}
\quad \text{mark } b
\quad \text{evaluate each } \emptyset\text{-function in } b
\quad \text{evaluate each } o \text{ in } b, \text{ in order}
\quad \forall \text{ SSA edge } <o,x>
\quad \quad \text{if } \text{block}(x) \text{ is marked}
\quad \quad \quad \text{add } <o,x> \text{ to SSAWorkList}

while(\text{SSAWorkList} \neq \emptyset)
\quad \text{remove } s = <u,v> \text{ from WorkList}
\quad \text{let } o \text{ be the operation that contains } v
\quad t \leftarrow \text{result of evaluating } o
\quad \text{if } t \neq \text{Value}(o) \text{ then}
\quad \quad \text{Value}(o) \leftarrow t
\quad \forall \text{ SSA edge } <o,x>
\quad \quad \text{if } \text{block}(x) \text{ is marked, then}
\quad \quad \quad \text{add } <o,x> \text{ to SSAWorkList}

Propagation Step

The statement of this algorithm in EaC1e is mangled. It is fixed in EaC2e.
Sparse Conditional Constant Propagation

There are some subtle points

• Branch conditions should not be **TOP** when evaluated
  - Indicates an upwards-exposed use
  - Hard to envision compiler producing such code

  *(no initial value)*

• Initialize Value attribute for each operation to **TOP**
  - Block processing will fill in the non-top initial values
  - Unreachable paths contribute **TOP** to Ø-functions

  *(correctness)*

• Code shows **CFG** edges first, then SSA edges
  - Can intermix them in arbitrary order
  - Taking **CFG** edges first may help with speed

  *(minor effect)*
Sparse Conditional Constant Propagation

More subtle points

• TOP * BOT → TOP
  ♦ If TOP becomes 0, then 0 * BOT → 0
  ♦ This prevents non-monotonic behavior for the result value
  ♦ Uses of the result value might go irretrievably to BOT
  ♦ Similar effects with any operation that has a “zero”

• Some values reveal simplifications, rather than constants
  ♦ BOT * \( c_i \) → BOT, but might turn into shifts & adds (\( c_i = 2, \text{ BOT} \geq 0 \))
    → Multiply to shift removes commutativity
      \[ \text{(reassociation)} \]
  ♦ BOT**2 → BOT * BOT
    \[ \text{(vs. series or call to library)} \]

• cbr TRUE → \( L_1, L_2 \) becomes br → \( L_1 \)
  ♦ Method discovers this; it must rewrite the code, too!
Sparse Conditional Constant

Unreachable Code

```plaintext
17  i ← 17
   if (i > 0) then
10    j₁ ← 10
   else
20    j₂ ← 20
⊥    j₃ ← ∅(j₁, j₂)
⊥    k ← j₃ * 17
```

Assume that all paths execute

Optimism

- Initialization to **TOP** is still important
- Unreachable code keeps **TOP**
- ∧ with **TOP** has desired result
Sparse Conditional Constant

Unreachable Code

```
17    i ← 17
     if (i > 0) then
         j₁ ← 10
     else
         j₂ ← 20
         j₃ ← Ø(j₁, j₂)
         k ← j₃ * 17
```

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- ∧ with TOP has desired result
Sparse Conditional Constant

Unreachable Code

17  
i ← 17
17  
if (i > 0) then
10    
j₁ ← 10
10  
else
10    
j₂ ← 20
10  
j₃ ← ∅(j₁, j₂)
170  
k ← j₃ * 17

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- ∧ with TOP has desired result
Sparse Conditional Constant

Unreachable Code

```
17  i ← 17
17  if (i > 0) then
   10  j₁ ← 10
else
   TOP
   j₂ ← 20
10  j₃ ← ∅(j₁, j₂)
170 k ← j₃ * 17
```

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- ∧ with TOP has desired result

Cannot get this result with separate transformations

- DEAD cannot test (i > 0)
- DEAD marks j₂ as useful

In general, combining two optimizations can lead to answers that cannot be produced by any combination of running them separately.

This algorithm is one example of that general principle.

Combining allocation & scheduling is another ...

COMP 512, Rice University
Sparse Conditional Constant Propagation

And one more thing ...

- Wegman and Zadeck proposed integrating inline substitution into SCCP
- They were aware of the difficulty of the decision problem for inlining
  - The “einey, meiney, miney, moe” problem

They proposed a simple solution:

*Inline during SCCP when known constants propagate into a call site*

- Constant-valued parameters & globals are one important source of improvement with inline substitution (see Ball [31])
- Compiler might inline for analysis and undo transformation if it did not find significant opportunities for simplification — constant folding, loop invariant code motion, redundancy expression

I know of no experimental evaluation of this idea.