SSA Destruction after Register Allocation

Tutorial on Register Allocation under SSA ; Part 5

Florent Bouchez
florent.bouchez@ens-lyon.org

Indian Institute of Science
Bangalore

LCPC ’09 — University of Delaware
How to get the hell out of SSA?
Tutorial on Register Allocation under SSA ; Part 5

Florent Bouchez
florent.bouchez@ens-lyon.org

Indian Institute of Science
Bangalore

LCPC ’09 — University of Delaware
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Cytron et al.’s out-of-SSA


*Efficiently computing static single assignment form and the control dependence graph.*

Naïvely, a $k$-input $\phi$-function at entrance to a node $X$ can be replaced by $k$ ordinary assignments, one at the end of each control flow predecessor of $X$.

*This is always correct.*
Cytron et al.’s out-of-SSA

*Efficiently computing static single assignment form and the control dependence graph.*

Example

```
a ← ...  b ← ...
```

```
c ← ϕ(a, b)
```
Cytron et al.’s out-of-SSA


*Efficiently computing static single assignment form and the control dependence graph.*

**Example**

\[
\begin{align*}
  a & \leftarrow \ldots \\
  b & \leftarrow \ldots \\
  c & \leftarrow \phi(a, b)
\end{align*}
\]
Briggs’s, Cooper’s out-of-SSA

Briggs, Cooper, Harvey, Simpson (1998)
Practical improvements to the construction and destruction of static single assignment form.

Cytron et al.’s out-of-SSA works after some optimizations:
- constant propagation
- dead code elimination
Briggs’s, Cooper’s out-of-SSA

Briggs, Cooper, Harvey, Simpson (1998)
*Practical improvements to the construction and destruction of static single assignment form.*

Cytron et al.’s out-of-SSA works after some optimizations:
- constant propagation
- dead code elimination

But NOT if done after more aggressive ones:
- copy folding
- value numbering
Lost copy problem

Example

\[
\begin{align*}
    a & \leftarrow 1 \\
    c & \leftarrow \phi(a, b) \\
    b & \leftarrow c + 1 \\
    \cdots & \leftarrow c
\end{align*}
\]
Lost copy problem

Example

\[
\begin{align*}
a &\leftarrow 1 \\
c &\leftarrow \phi(a, b) \\
b &\leftarrow c + 1 \\
\cdots &\leftarrow c
\end{align*}
\]

\[
\begin{align*}
a &\leftarrow 1 \\
c &\leftarrow a \\
b &\leftarrow c + 1 \\
\cdots &\leftarrow c
\end{align*}
\]

This is due to the critical edge.
Lost copy problem

Example

\[
a \leftarrow 1
\]
\[
c \leftarrow \phi(a, b)
\]
\[
b \leftarrow c + 1
\]
\[
\ldots \leftarrow c
\]

\[
a \leftarrow 1
\]
\[
c \leftarrow a
\]
\[
b \leftarrow c + 1
\]
\[
c \leftarrow b
\]
\[
\ldots \leftarrow c
\]

This is due to the critical edge.
Swap problem

Example

\[
\begin{align*}
  a & \leftarrow \phi(a, b) \\
  b & \leftarrow \phi(b, a)
\end{align*}
\]
Swap problem

Example

\[ a \leftarrow \phi(a, b) \]
\[ b \leftarrow \phi(b, a) \]

\[ a \leftarrow a \]
\[ b \leftarrow b \]

\[ a \leftarrow b \]
\[ b \leftarrow a \]
Swap problem

Example

```
<table>
<thead>
<tr>
<th></th>
<th>a ← a</th>
<th>a ← b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b ← b</td>
<td>b ← a</td>
</tr>
<tr>
<td>a ← φ(a, b)</td>
<td>b ← φ(b, a)</td>
<td></td>
</tr>
</tbody>
</table>
```

Copies induced by $\phi$-functions must be executed in parallel. We call them //copies.
Sreedhar et al.’s out-of-SSA

Sreedhar, Ju, Gillies, Santhanam (1999)
*Translating out of Static Single Assignment form.*

**Example**

\[
\begin{align*}
a & \leftarrow \ldots \\
\phi(a, b) & \leftarrow \ldots \\
c & \leftarrow \phi(a, b)
\end{align*}
\]
Sreedhar et al.’s out-of-SSA

Sreedhar, Ju, Gillies, Santhanam (1999)
*Translating out of Static Single Assignment form.*

Example

\[
\begin{align*}
    a & \leftarrow \ldots \\
    b & \leftarrow \ldots \\
    c & \leftarrow \phi(a, b)
\end{align*}
\]

\[
\begin{align*}
    a & \leftarrow \ldots \\
    a' & \leftarrow a \\
    b' & \leftarrow b \\
    c' & \leftarrow \phi(a', b') \\
    c & \leftarrow c'
\end{align*}
\]

Adding primed variables ➞ conventional SSA.
Sreedhar et al.’s out-of-SSA

Sreedhar, Ju, Gillies, Santhanam (1999)
Translating out of Static Single Assignment form.

Example

\[a \leftarrow \ldots\]
\[b \leftarrow \ldots\]
\[X \leftarrow a\]
\[X \leftarrow b\]
\[c \leftarrow \phi(a, b)\]
\[c \leftarrow X\]

Adding primed variables \(\mapsto\) conventional SSA.
Then, all variables of the same \(\phi\)-congruence class can be replaced by a common name.
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Flow of SSA vs. classical register allocation

- Interim. repr. (IR)
- SSA conversion
- SSA form IR
- SSA destruction
- post SSA IR
- Register allocation
- Assembly

Only registers remain when translating out-of-SSA! «not possible to add new variables.»

F. Bouchez (IISc – India) How to get the hell out of SSA? LCPC ’09
Flow of SSA vs. classical register allocation

Interm. repr. (IR) -> SSA conversion -> SSA form IR -> SSA destruction -> post SSA IR -> Register allocation -> Assembly

Program optim.

Interm. repr. (IR) -> SSA conversion -> SSA form IR -> Assembly

Program optim.
Flow of SSA vs. classical register allocation

Interm. repr. (IR) → SSA conversion → SSA form IR → SSA destruction → post SSA IR → Register allocation → Assembly

Interm. repr. (IR) → SSA conversion → SSA form IR → Register allocation → Assembly

F. Bouchez (IISc – India)

How to get the hell out of SSA?

LCPC '09
Flow of SSA vs. classical register allocation

Interm. repr. (IR) -> SSA conversion -> SSA form IR

Prog. optim. -> SSA destruction -> post SSA IR

Register allocation -> Assembly

Interm. repr. (IR) -> SSA conversion -> SSA form IR

Register allocation -> SSA destruction -> Assembly

Out-of-SSA after RA

The problem

Moving copies

Experiments

Conclusion
Flow of SSA vs. classical register allocation

Only registers remain when translating out-of-SSA!

⇒ not possible to add new variables.
Deal with the differences

Without temporary variables:

- Imitate Sreedhar using available registers as new variables. But what if there are not enough registers? spill? ➔ bad idea.
Deal with the differences

Without temporary variables:

- Imitate Sreedhar using available registers as new variables. But what if there are not enough registers? spill? bad idea.
- Problems are back:
  - "swap problem" since copies order is important
  - critical edge poses the problem of where to put copies
Deal with the differences

Without temporary variables:

- Imitate Sreedhar using available registers as new variables. But what if there are not enough registers? spill? ➔ bad idea.
- Problems are back:
  - ”swap problem” since copies order is important
  - critical edge poses the problem of where to put copies

Back to semantics of φ-function: depends on the flow
➔ split all incoming edges to place //copies, but performance problem!
Splitting edges can be bad

Folk assumption that splitting edges is a bad thing to do:

- adds a jump instruction
- prevents mapping on hardware accelerator
- code cannot be scheduled
Splitting edges can be bad

Folk assumption that splitting edges is a bad thing to do:

- adds a jump instruction
- prevents mapping on hardware accelerator
- code cannot be scheduled

We would like to have a way to not split (at least some) edges. Is it possible and how to do it?
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Problem statement

Goal: reduce the overhead due to the added copies and edge splitting during SSA destruction.

We suppose there still remains //copies on edges after coalescing.

Problems

- “parallel” problem: //copies must be sequentialized
  - how to find a correct ordering?
- if a //copy is placed on an edge, it must be split
  - possible to move //copies out of edges?
Modeling of //copies

Simplifying fact: register allocation is done. 
//copies involves registers, not variables.

⇒ register transfer graphs (RTG), a graphical representation:

```
R1
R2
R3
```

Properties of an RTG:
• Maximum in-degree equals 1
• 3 types of RTG (tree, cycle, windmill (general case))
Modeling of //copies

Simplifying fact: register allocation is done.
//copies involves registers, not variables.

→ register transfer graphs (RTG), a graphical representation:

Properties of an RTG:
- Maximum in-degree equals 1
- 3 types of RTG (tree, cycle, windmill)
Modeling of //copies

Simplifying fact: register allocation is done. //copies involves registers, not variables.

_register transfer graphs (RTG), a graphical representation:

Properties of an RTG:
- Maximum in-degree equals 1
- 3 types of RTG (tree, cycle, windmill(general case))
Modeling of //copies

Simplifying fact: register allocation is done.
//copies involves registers, not variables.

⇒ register transfer graphs (RTG), a graphical representation:

Properties of an RTG:
- Maximum in-degree equals 1
- 3 types of RTG (tree, cycle, windmill (general case))
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Is it possible to move \(/\)copies out of edges?

On a simple edge (i.e., non-critical): code can be placed on predecessor or successor basic block.

Example

\[ B_s \xrightarrow{\text{\#copy}} B_d \]
Is it possible to move //copies out of edges?

On a simple edge (i.e., non-critical): code can be placed on predecessor or successor basic block.

Example

\[
\begin{align*}
B_s & \xrightarrow{\text{//copy}} B_d \\
& \quad \xrightarrow{\text{//copy}} B_s \xrightarrow{\text{move up}} B_d
\end{align*}
\]
Is it possible to move //copies out of edges?

On a simple edge (i.e., non-critical): code can be placed on predecessor or successor basic block.

Example

```
Bs  //copy
   ↓
Bs

Bd
```

But what if Bs has more than one successor? Bd has more than one predecessor?
Is it possible to move //copies out of edges?

On a simple edge (i.e., non-critical): code can be placed on predecessor or successor basic block.

Example

But what if $B_s$ has more than one successor?
$B_d$ has more than one predecessor?
//copies are hard to move out of critical edges

Example

A \rightarrow B \rightarrow C \rightarrow D
//copies are hard to move out of critical edges

Example

A → B

C ← φ(y, z) → D

Example graph with nodes A, B, C, and D, and an edge from A to B, and another from C to D, with a label φ(y, z) on the edge from C.
//copies are hard to move out of critical edges

Example

\[ R_1 \leftarrow \phi(R_2, R_3) \]
//copies are hard to move out of critical edges

Example

F. Bouchez (IISc – India)
//copies are hard to move out of critical edges

Example

Moving down erases $R_1$ if path comes from D.
//copies are hard to move out of critical edges

Moving up might erase a live variable in $R_1$. 

Example

\[
\begin{array}{c}
A \quad \cdots \leftarrow R_1 \\
\text{B} \quad R_1 \leftarrow \cdots \leftarrow R_1 \leftarrow R_2 \\
\text{C} \\
\text{D} \quad R_1 \leftarrow R_3 \\
\end{array}
\]
Reversing the effect of parallel copies

Idea: a //copy can be moved if its effects are cancelled on the other edges.
Reversing the effect of parallel copies

Idea: a //copy can be moved if its effects are cancelled on the other edges.

Example (Cancelling $R_1 \leftarrow R_2$)

- **A**: $R_1 \leftarrow R_3$
- **B**: $R_1 \leftarrow \ldots$  
  $R_3 \leftarrow R_1$
  $R_1 \leftarrow R_2$
- **C**: $R_1 \leftarrow \phi(R_2, R_3)$
- **D**: $R_1 \leftarrow R_3$

F. Bouchez (IISc – India)  How to get the hell out of SSA?  LCPC '09
Reversing the effect of parallel copies

Idea: a //copy can be moved if its effects are cancelled on the other edges.

Example (Cancelling $R_1 \leftarrow R_2$)

```
A

B
  \( R_1 \leftarrow \ldots \)  

C
  \( R_1 \leftarrow R_2 \)
  \( R_1 \leftarrow \phi(R_2, R_3) \)

D
  \( R_1 \leftarrow R_3 \)
  \( R_2 \leftarrow R_1 \)
```
Reversing the effect of parallel copies

Idea: a //copy can be moved if its effects are cancelled on the other edges.

Example (Duplication problem)

F. Bouchez (IISc – India)  How to get the hell out of SSA?  LCPC ’09
Reversing the effect of parallel copies

Idea: a \//copy can be moved if its effects are cancelled on the other edges.

- Effects must be reversed if moving up.
- Effects must be pre-reversed if moving down.

Problems

- \//copies involving overwriting of existing value must be carefully moved.
  - //copy motion is highly dependent on the liveness.
- \//copies with duplications cannot be pre-reversed.
Extracting duplications from //copies

Duplications can only be moved up. We want to extract them to //copies to get rid of them. This can be done using free registers.

Example

\[
R_1 \rightarrow R_3 = R_1 \rightarrow R_2 \rightarrow R_3 \circ R_1 \rightarrow R_2 \rightarrow R_3
\]
Extracting duplications from //copies

Duplications can only be moved up. We want to extract them to //copies to get rid of them. This can be done using free registers.

Example

A  B  C  D

R₁  R₂  R₁ ← R₂
R₂  R₃  R₂ ← R₃
R₁ ← φ(R₁, R₂)
R₂ ← φ(R₁, R₃)
Extracting duplications from //copies

Duplications can only be moved up. We want to extract them to //copies to get rid of them. This can be done using free registers.

Example

R₁ ← φ(R₁, R₂)
R₂ ← φ(R₁, R₃)
R₁ ← R₂
R₂ ← R₃
Extracting duplications from //copies

Duplications can only be moved up. We want to extract them to //copies to get rid of them. This can be done using free registers.

- Extracted duplications are moved up.
- If preceding basic block has not enough free registers, one needs to split or spill.
Extracting duplications from //copies

Duplications can only be moved up. We want to extract them to //copies to get rid of them. This can be done using free registers.

- Extracted duplications are moved up.
- If preceding basic block has not enough free registers, one needs to split or spill.
- There remains a reversible //copy: max out-degree = 1.
Permutations to easily deal with liveness

Liveness is important when moving //copies. Registers holding live variables must not be overwritten.

We convert reversible //copies to permutations:

→ just close the open cycles in the RTG.
Permutations to easily deal with liveness

Liveness is important when moving \parallel copies. Registers holding live variables must not be overwritten.

We convert reversible \parallel copies to permutations:

\rightarrow just close the open cycles in the RTG.

Example

\begin{tikzpicture}
  \node (R1) at (0,0) {$R_1$};
  \node (R2) at (1,-1) {$R_2$};
  \node (R3) at (2,0) {$R_3$};
  \node (R4) at (3,0) {$R_4$};
  \node (R5) at (2,-1) {$R_5$};
  \node (R6) at (3,-1) {$R_6$};

  \path[->] (R1) edge (R2);
  \path[->] (R2) edge (R3);
  \path[->] (R3) edge (R1);
  \path[->] (R4) edge (R5);
  \path[->] (R5) edge (R6);
  \path[->] (R6) edge (R4);
\end{tikzpicture}
Permutations to easily deal with liveness

Liveness is important when moving //copies. Registers holding live variables must not be overwritten.

We convert reversible //copies to permutations:

→ just close the open cycles in the RTG.

Example

```
R1 → R3 → R4

R2 ← R3 ← R5

R6
```
π-motion out of critical edges

Example
\( \pi \)-motion out of critical edges

Example

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D}
\end{array}
\]

\[
\begin{array}{c}
R_1 \\
R_2 \\
R_3
\end{array}
\]

This critical edge is weak: its adjacent edges are not critical.
\( \pi \)-motion out of critical edges

Example

\[ \begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D}
\end{array} \]

\[ \begin{array}{c}
R_1 \\
R_2 \\
R_3
\end{array} \]
π-motion out of critical edges

Example
The problem

π-motion out of critical edges

Example

This critical edge is weak: its adjacent edges are not critical.
\( \pi \)-motion follows weak critical edges

Example

Weakness is a property propagated along connected edges.
π-motion follows weak critical edges

Example

Weakness is a property propagated along connected edges.
\( \pi \)-motion follows weak critical edges

Weakness is a property propagated along connected edges.
$\pi$-motion follows weak critical edges

Example

Weakness is a property propagated along connected edges.
**Intrinsic difficulty strong critical edges**

Connected critical edges forming a cycle are **strong**.

---

**Example**

![Diagram of connected critical edges forming a cycle](image)
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

Example
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

- $\pi$-motion is not possible on strong critical edges
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

- $\pi$-motion is not possible on strong critical edges
- removing the $//copies$ on such a multiplexing region is NP-complete
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

- $\pi$-motion is not possible on strong critical edges
- removing the //copies on such a multiplexing region is NP-complete

Example
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

- $\pi$-motion is not possible on strong critical edges
- removing the $//\text{copies}$ on such a multiplexing region is NP-complete

Example
Intrinsic difficulty strong critical edges

Connected critical edges forming a cycle are strong.

- π-motion is not possible on strong critical edges
- removing the \( // \) copies on such a multiplexing region is NP-complete

Example

```
Example

\[
\begin{array}{c}
\text{switch} \\
B_{\text{root}} \\
B_{a,c} \\
B_{b,d} \\
B_{c,d} \\
B_{a,b} \\
B_a, B_b, B_c, B_d \\
\end{array}
\]

```

```
Example

a ← 0
b ← 1
x ← a + b

a ← 3
c ← 3
x ← a + c

b ← 6
d ← 6
x ← b + d

c ← 9
d ← 9
x ← c + d

return a + x
return b + x
return c + x
return d + x
```

```
Example

a ← 0
b ← 1
x ← a + b

a ← 3
c ← 3
x ← a + c

b ← 6
d ← 6
x ← b + d

c ← 9
d ← 9
x ← c + d

return a + x
return b + x
return c + x
return d + x
```

```
Example

\[
\begin{array}{c}
\text{switch} \\
B_{\text{root}} \\
B_{a,c} \\
B_{b,d} \\
B_{c,d} \\
B_{a,b} \\
B_a, B_b, B_c, B_d \\
\end{array}
\]

```

```
Example

a ← 0
b ← 1
x ← a + b

a ← 3
c ← 3
x ← a + c

b ← 6
d ← 6
x ← b + d

c ← 9
d ← 9
x ← c + d

return a + x
return b + x
return c + x
return d + x
```

```
Example

\[
\begin{array}{c}
\text{switch} \\
B_{\text{root}} \\
B_{a,c} \\
B_{b,d} \\
B_{c,d} \\
B_{a,b} \\
B_a, B_b, B_c, B_d \\
\end{array}
\]

```

```
Example

a ← 0
b ← 1
x ← a + b

a ← 3
c ← 3
x ← a + c

b ← 6
d ← 6
x ← b + d

c ← 9
d ← 9
x ← c + d

return a + x
return b + x
return c + x
return d + x
```
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 1.
Liveness projection
Non Live = \{R_5, R_7\}
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 2.
Sequentialize chains
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 2.
Sequentialize chains

\[ R_7 \leftarrow R_6 \]
Sequentialize permutations
Permutations must eventually be converted to actual copies.

Step 2.
Sequentialize chains

\[ R_7 \leftarrow R_6 \]
\[ R_5 \leftarrow R_4 \]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 2.
Sequentialize chains

- $R_7 \leftarrow R_6$
- $R_5 \leftarrow R_4$
- $R_4 \leftarrow R_3$
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 2.
Sequentialize chains

\[ R_7 \leftarrow R_6 \]
\[ R_5 \leftarrow R_4 \]
\[ R_4 \leftarrow R_3 \]
\[ R_3 \leftarrow R_2 \]
Sequentialize permutations

Permutations must eventually be converted to actual copies.
Sequentialize permutations
Permutations must eventually be converted to actual copies.

Step 3.
Mark root of chain as “free”

\[
R_7 \leftarrow R_6 \\
R_5 \leftarrow R_4 \\
R_4 \leftarrow R_3 \\
R_3 \leftarrow R_2 \\
R_2 \leftarrow R_1
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4.
Use free register for cycles

\[
\begin{align*}
R_7 & \leftarrow R_6 \\
R_5 & \leftarrow R_4 \\
R_4 & \leftarrow R_3 \\
R_3 & \leftarrow R_2 \\
R_2 & \leftarrow R_1 \\
R_1 & \leftarrow R_{10}
\end{align*}
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4.
Use free register for cycles

\[
\begin{align*}
R_7 & \leftarrow R_6 \\
R_5 & \leftarrow R_4 \\
R_4 & \leftarrow R_3 \\
R_3 & \leftarrow R_2 \\
R_2 & \leftarrow R_1 \\
R_1 & \leftarrow R_{10} \\
R_{10} & \leftarrow R_9
\end{align*}
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4.
Use free register for cycles

\[
\begin{align*}
R_7 & \leftarrow R_6 \\
R_5 & \leftarrow R_4 \\
R_4 & \leftarrow R_3 \\
R_3 & \leftarrow R_2 \\
R_2 & \leftarrow R_1 \\
R_1 & \leftarrow R_{10} \\
R_{10} & \leftarrow R_9 \\
R_9 & \leftarrow R_8
\end{align*}
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4.
Use free register for cycles

\[
\begin{align*}
R_7 &\leftarrow R_6 \\
R_5 &\leftarrow R_4 \\
R_4 &\leftarrow R_3 \\
R_3 &\leftarrow R_2 \\
R_2 &\leftarrow R_1 \\
R_1 &\leftarrow R_{10} \\
R_{10} &\leftarrow R_9 \\
R_9 &\leftarrow R_8 \\
R_8 &\leftarrow R_1
\end{align*}
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4’. (only cycles)
No free register $\rightarrow$ swap
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4’. (only cycles)
No free register → swap

swap($R_1, R_5$)
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4’. (only cycles)
No free register ➔ swap

\[
\text{swap}(R_1, R_5) \\
\text{swap}(R_2, R_5) \\
\text{swap}(R_3, R_5) \\
\text{swap}(R_4, R_5)
\]
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4’. (only cycles)
No free register ➡️ swap

swap($R_1, R_5$)
swap($R_2, R_5$)
swap($R_3, R_5$)
Sequentialize permutations

Permutations must eventually be converted to actual copies.

Step 4’. (only cycles)
No free register \( \Rightarrow \) swap

\[
\text{swap}(R_1, R_5) \\
\text{swap}(R_2, R_5) \\
\text{swap}(R_3, R_5) \\
\text{swap}(R_4, R_5)
\]

Swaps can be hardware or emulated (by using 3 XOR).

If swapping is impossible, spilling is required.
Outline

Background: classical out-of-SSA

Differences with out-of-SSA after register allocation under SSA

Statement of the problem

Moving parallel copies out of edges

Experiments
Experiments

Different approaches to deal with //copies on edges:

- *split*: edges are split by default
- \(\pi\)-*motion*: //copies are moved out of edges whenever possible
- done after coalescing (IRC\(^1\)) or not

<table>
<thead>
<tr>
<th>Number of split edges in SPEC2000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>After coalescing</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>split</td>
</tr>
<tr>
<td>257</td>
</tr>
<tr>
<td>(\pi)-motion</td>
</tr>
<tr>
<td>45.5</td>
</tr>
</tbody>
</table>

For about 180 000 total edges.

\(^1\)Iterated Register Coalescing of Appel&George
Conclusion

SSA destruction after register allocation means:

- new variable creation not possible
- reappearance of “swap problem” and “critical edge problem”

Emergency instructions to get out-of-SSA

1. $\phi$-functions are replaced by //copies on incoming edges
2. //copies are decomposed into duplications (moved up) and reversible //copies
3. //copies are converted to permutations for easier handling
4. permutations can be moved out of weak critical edges
5. permutations are converted back to //copies and sequentialized in a working order.
That’s all for today