Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency

Benoit Boissinot (LIP), Alain Darte (LIP), Benoît Dupont de Dinechin (STMicro), Christophe Guillon (STMicro), Fabrice Rastello (LIP)

Compsys Team
Laboratoire de l’Informatique du Parallélisme (LIP)
École normale supérieure de Lyon

CGO’09, March 24, 2009, Seattle, WA
Outline

1. SSA foundations
   - Out-of-SSA translation

2. Correctness and code quality
   - Translation with copy insertions
   - Improving code quality
   - Qualitative experiments

3. Speed and memory footprint
   - Linear-time algorithm for coalescing congruence classes
   - Experimental results for speed and memory footprint

4. Conclusion
Outline

1. SSA foundations
   - Out-of-SSA translation

2. Correctness and code quality
   - Translation with copy insertions
   - Improving code quality
   - Qualitative experiments

3. Speed and memory footprint
   - Linear-time algorithm for coalescing congruence classes
   - Experimental results for speed and memory footprint

4. Conclusion
Static single assignment (SSA)

SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce $\phi$-functions at dominance frontier.

Interests of SSA

- Code optimizations: efficient, easy-to-implement, fast;
- Two-phases register allocation;
- Program analysis/verification.
Static single assignment (SSA)

SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.

```
B_0
a = ...
b = ...

B_1
n = b
b = a
a = n
```
Static single assignment (SSA)

SSA with dominance property
- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA
- Need to introduce $\phi$-functions at dominance frontier.
SSA foundations
Correctness and code quality
Speed and memory footprint
Conclusion

Out-of-SSA translation

Static single assignment (SSA)

SSA with dominance property
- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA
- Need to introduce $\phi$-functions at dominance frontier.

\[
\begin{align*}
B_0 & \quad a_1 = \ldots \\
& \quad b_1 = \ldots \\
B_1 & \quad a_2 = \phi(a_1, a_3) \\
& \quad b_2 = \phi(b_1, b_3) \\
& \quad n = b_2 \\
& \quad b_3 = a_2 \\
& \quad a_3 = n
\end{align*}
\]
Static single assignment (SSA)

**SSA with dominance property**
- Unique definition for each variable;
- Each definition dominates its uses.

**Conversion into SSA**
- Need to introduce $\phi$-functions at dominance frontier.

**Interests of SSA**
- Code optimizations: efficient, easy-to-implement, fast;
- Two-phases register allocation;
- Program analysis/verification.

\[
\begin{align*}
B_0 & : \quad a_1 = \ldots \\
    & \quad b_1 = \ldots \\
B_1 & : \quad a_2 = \phi(a_1, a_3) \\
    & \quad b_2 = \phi(b_1, b_3) \\
    & \quad n = b_2 \\
    & \quad b_3 = a_2 \\
    & \quad a_3 = n
\end{align*}
\]
Static single assignment (SSA)

SSA with dominance property
- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA
- Need to introduce $\phi$-functions at dominance frontier.

Interests of SSA
- Code optimizations: efficient, easy-to-implement, fast;
- Two-phases register allocation;
- Program analysis/verification.
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks.

Swap problem

\[
\begin{align*}
B_0 & : a_1 = \ldots \\
   & b_1 = \ldots \\
B_1 & : a_2 = \phi(a_1, b_2) \\
   & b_2 = \phi(b_1, a_2)
\end{align*}
\]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks.

Swap problem:

\[
\begin{align*}
B_0: & \quad a_1 = \ldots \\
& \quad b_1 = \ldots \\
& \quad a_2 = a_1 \\
& \quad b_2 = b_1 \\
B_1: & \quad a_2 = \phi(a_1, b_2) \\
& \quad b_2 = \phi(b_1, a_2) \\
& \quad a_2 = b_2 \\
& \quad b_2 = a_2
\end{align*}
\]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

\[ \begin{align*}
B_0 & : a_1 = \ldots \quad b_1 = \ldots \\
B_1 & : a_2 = a_1 \\
& \quad b_2 = b_1 \\
\end{align*} \]

\[ \begin{align*}
& : a_2 = b_2 \\
& \quad b_2 = a_2 \\
\end{align*} \]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

```
B_0
x = ...

B_1
y = x
x = x + 1
```

Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
Bad understanding of parallel copies.

Briggs et al. (1998): both problems identified. General correctness unclear.

Sreedhar et al. (1999): correct but handling of complex branching instructions unclear; interplay with coalescing unclear; "virtualization" hard to implement.

Lost copy problem

Many SSA optimizations turned off in gcc and Jikes.
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

\[ x_2 = \phi(x_1, x_3) \]
\[ y = x_2 \]
\[ x_3 = x_2 + 1 \]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0 & : x_1 = \ldots \\
B_1 & : x_2 = \phi(x_1, x_3) \\
      & x_3 = x_2 + 1
\end{align*}
\]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0 & : \quad x_1 = \ldots, \\
    & \quad x_2 = x_1
\end{align*}
\]

\[
\begin{align*}
B_1 & : \quad x_2 = \phi(x_1, x_3) \\
    & \quad x_3 = x_2 + 1 \\
    & \quad x_2 = x_3
\end{align*}
\]

\[
x_2
\]
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

![Lost copy problem diagram]

Many SSA optimizations turned off in gcc and Jikes.
Why is out-of-SSA translation difficult?

- **Cytron et al. (1991):** copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

- **Briggs et al. (1998):** both problems identified. General correctness unclear.
Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - “virtualization” hard to implement.
Why is out-of-SSA translation difficult?

- **Cytron et al. (1991):** copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

- **Briggs et al. (1998):** both problems identified. General correctness unclear.

- **Sreedhar et al. (1999):** correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - “virtualization” hard to implement.

- Many SSA optimizations turned off in gcc and Jikes.
**Conventional SSA (CSSA)**

For any \( \phi \)-function \( a_0 = \phi(a_1, \ldots, a_n) \), the variables \( a_0, \ldots, a_n \) can be safely replaced by a common ressource.

**From SSA to CSSA**

\[
B_1 \quad B_i \quad B_n
\]

\[
B_0 \quad a_0 = \phi(a_1, \ldots, a_n)
\]
Conventional SSA (CSSA)

For any $\phi$-function $a_0 = \phi(a_1, \ldots, a_n)$, the variables $a_0, \ldots, a_n$ can be safely replaced by a common resource.

Correctness

Add copies with new local variables around every $\phi$. $\implies$ CSSA

From SSA to CSSA

$B_1$

$B_i$

$B_n$

$B_0$

$$a'_0 = \phi(a'_1, \ldots, a'_n)$$

$B_0$

$$a'_0 = a'_1$$

$$a'_i = a_i$$

$$a'_n = a_n$$

$$a_0 = a'_0$$
Code quality

Removing copies

Useless copies can be removed by standard aggressive coalescing. Using an accurate notion of interference (value-based) gives excellent results.
Removing copies

Useless copies can be removed by standard aggressive coalescing. Using an accurate notion of interference (value-based) gives excellent results.

“Liveness of $\phi$” defined by the $a_i'$. Be careful with potential bugs due to conditional branches that use or define variables.
Coalesced example: the swap problem

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ B_0 \]

\[ a_2 = \phi(a_1, b_2) \]
\[ b_2 = \phi(b_1, a_2) \]

\[ B_1 \]

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ (u_1, v_1) = (a_1, b_1) \]

\[ B_0 \]

\[ u_0 = \phi(u_1, u_2) \]
\[ v_0 = \phi(v_1, v_2) \]
\[ (a_2, b_2) = (u_0, v_0) \]
\[ (u_2, v_2) = (b_2, a_2) \]

\[ B_1 \]
Coalesced example: the swap problem

\[
\begin{align*}
\text{a}_1 & \quad \text{u} = (\text{u}_0, \text{u}_1, \text{u}_2) \quad \text{a}_2 \\
\text{b}_1 & \quad \text{v} = (\text{v}_0, \text{v}_1, \text{v}_2) \quad \text{b}_2
\end{align*}
\]

\[
\begin{align*}
\text{B}_0 & \\
\text{a}_1 & = \ldots \\
\text{b}_1 & = \ldots \\
(\text{u}_1, \text{v}_1) & = (\text{a}_1, \text{b}_1)
\end{align*}
\]

\[
\begin{align*}
\text{B}_1 & \\
\text{u}_0 & = \phi(\text{u}_1, \text{u}_2) \\
\text{v}_0 & = \phi(\text{v}_1, \text{v}_2) \\
(\text{a}_2, \text{b}_2) & = (\text{u}_0, \text{v}_0) \\
(\text{u}_2, \text{v}_2) & = (\text{b}_2, \text{a}_2)
\end{align*}
\]
Coalesced example: the swap problem

\[a_1 \quad u = (u_0, u_1, u_2) \quad a_2\]

\[b_1 \quad v = (v_0, v_1, v_2) \quad b_2\]

\[B_0\]

\[a_1 = \ldots\]
\[b_1 = \ldots\]
\[(u_1, v_1) = (a_1, b_1)\]

\[B_1\]

\[u_0 = \phi(u_1, u_2)\]
\[v_0 = \phi(v_1, v_2)\]
\[(a_2, b_2) = (u_0, v_0)\]
\[(u_2, v_2) = (b_2, a_2)\]
Coalesced example: the swap problem

\[
B_0
\]
\[
a_1 = \ldots
b_1 = \ldots
\]

\[
B_1
\]
\[
(u_2, v_2) = (b_2, a_2)
\]

\[
B_0
\]
\[
(u_1, v_1) = (a_1, b_1)
\]

\[
B_1
\]
\[
u_0 = \phi(u_1, u_2)
\]
\[
v_0 = \phi(v_1, v_2)
\]
\[
(a_2, b_2) = (u_0, v_0)
\]
\[
(u_2, v_2) = (b_2, a_2)
\]
Coalesced example: the swap problem

\[
\begin{align*}
B_0: & \quad a_1 = \ldots \quad b_1 = \ldots \\
B_1: & \quad (u_2, v_2) = (b_2, a_2)
\end{align*}
\]

\[
\begin{align*}
B_0: & \quad a_1 = \ldots \quad b_1 = \ldots \\
B_1: & \quad n = b_1 \quad b_1 = a_1 \quad a_1 = n
\end{align*}
\]
Coalesced example: the lost copy problem

\[ x_1 = \ldots \]

\[ x_2 = \phi(x_1, x_3) \]
\[ x_3 = x_2 + 1 \]

\[ x_2 \]

\[ B_0 \]

\[ B_1 \]

\[ u_0 = \phi(u_1, u_2) \]
\[ x_2 = u_0 \]
\[ x_3 = x_2 + 1 \]
\[ u_2 = x_3 \]

\[ x_2 \]

\[ B_0 \]

\[ B_1 \]
Coalesced example: the lost copy problem

\[ u = (u_0, u_1, u_2) \]

\[ B_0 \]
\[ x_1 = \ldots \]
\[ u_1 = x_1 \]

\[ B_1 \]
\[ u_0 = \phi(u_1, u_2) \]
\[ x_2 = u_0 \]
\[ x_3 = x_2 + 1 \]
\[ u_2 = x_3 \]
Coalesced example: the lost copy problem

Coalesced example: the lost copy problem

\[ u = (u_0, u_1, u_2) \]

\[ x_1 = \ldots \]
\[ u_1 = x_1 \]

\[ x_2 = u_0 \]
\[ x_3 = x_2 + 1 \]

\[ u_0 = \phi(u_1, u_2) \]
\[ u_2 = x_3 \]
Coalesced example: the lost copy problem

\[ B_0 \]
\[ x_1 = \ldots \]

\[ B_1 \]
\[ x_2 = x_1 \]
\[ x_1 = x_2 + 1 \]

\[ B_0 \]
\[ u_1 = x_1 \]

\[ B_1 \]
\[ u_0 = \phi(u_1, u_2) \]
\[ x_2 = u_0 \]
\[ x_3 = x_2 + 1 \]
\[ u_2 = x_3 \]
Exploiting SSA: value-based interferences

Definition (Chaitin interference)
Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

\[ a = \phi(b, c) \]

\[ d = \phi(b, a) \]
Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

\[
\begin{align*}
b &= \ldots \\
a' &= b \\
d' &= b \\
\end{align*}
\]

\[
\begin{align*}
c &= \ldots \\
a' &= c \\
\end{align*}
\]

\[
\begin{align*}
a &= a' \\
d' &= a \\
\end{align*}
\]

\[
d = d'
\]
Exploiting SSA: value-based interferences

**Definition (Chaitin interference)**

Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

![Diagram showing SSA variable interferences](image)
Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

Need to update interference graph after coalescing.
Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

Need to update interference graph after coalescing.

Unique value $V$ of a SSA variable

For a copy $b = a$, $V(b) = V(a)$ (traversal of dominance tree).

Value-based interference

$a$ and $b$ interfere if $V(a) \neq V(b)$ and $\text{Live-range}(a) \cap \text{Live-range}(b) \neq \emptyset$.

For a copy $b = a'$, $V(b) = V(a')$ (traversal of dominance tree).
Exploiting SSA: value-based interferences

Definition (Chaitin interference)
Two variables interfere if one is live at the definition of the other, and it is not a copy of the first.

Need to update interference graph after coalescing.

Unique value $V$ of a SSA variable
For a copy $b = a$, $V(b) = V(a)$ (traversal of dominance tree).

Value-based interference
$a$ and $b$ interfere if $V(a) \neq V(b)$ and $\text{Live-range}(a) \cap \text{Live-range}(b) \neq \emptyset$.
Qualitative experiments with SPEC CINT2000

Number of remaining moves
Qualitative experiments with SPEC CINT2000

Key points of the out-of-SSA translation

- Copy insertion (to go to CSSA and to handle register renaming constraints) followed by coalescing.
- Value-based interferences → coalescing is improved and independent of virtualization (i.e., as in Sreedhar III).
- Parallel copies followed by sequentialization.
Outline

1. SSA foundations
   - Out-of-SSA translation

2. Correctness and code quality
   - Translation with copy insertions
   - Improving code quality
   - Qualitative experiments

3. Speed and memory footprint
   - Linear-time algorithm for coalescing congruence classes
   - Experimental results for speed and memory footprint

4. Conclusion
How to coalesce variables?

Two alternatives

- Use a **working interference graph** where, in case of coalescing, corresponding vertices 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.
Fast interference test for a set of variables

Key properties for linear-complexity live range intersection

- 2 SSA variables intersect if one is live at the definition of the other.
- In this case, the first definition dominates the second one.
- Budimlić: If $a$ and $b$ intersect ($a \text{ dom } b$), then $\forall c$ with $a \text{ dom } c$ and $c \text{ dom } b$: $b$ and $c$ interfere.

$\implies$ For each variable, the only test needed is with the “closest” dominating variable.
Fast interference test for a set of variables

\[
\begin{align*}
  a & \leftarrow \ldots \\
  b & \leftarrow a + \ldots \\
  c & \leftarrow b + \ldots \\
  & \leftarrow c \\
  d & \leftarrow \ldots \\
  e & \leftarrow d + \ldots \\
  & \leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\[ \begin{align*}
  b & \leftarrow a + \cdots \\
  c & \leftarrow b + \cdots \\
  & \leftarrow c \\
  d & \leftarrow \cdots \\
  e & \leftarrow d + \cdots \\
  & \leftarrow a + e
\end{align*} \]
Fast interference test for a set of variables

\[ b \leftarrow a + \cdots \]
\[ c \leftarrow b + \cdots \]
\[ \leftarrow c \]
\[ d \leftarrow \cdots \]
\[ e \leftarrow d + \cdots \]
\[ \leftarrow a + e \]
Fast interference test for a set of variables

\[
\begin{align*}
    b & \leftarrow a + \cdots \\
    c & \leftarrow b + \cdots \\
       & \leftarrow c \\
    d & \leftarrow \cdots \\
    e & \leftarrow d + \cdots \\
       & \leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\[
\begin{align*}
  b & \leftarrow a + \cdots \\
  c & \leftarrow b + \cdots \\
  & \leftarrow c \\
  d & \leftarrow \cdots \\
  e & \leftarrow d + \cdots \\
  & \leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\[
\begin{align*}
  b &\leftarrow a + \cdots \\
  c &\leftarrow b + \cdots \\
  &\leftarrow c \\
  d &\leftarrow \cdots \\
  e &\leftarrow d + \cdots \\
  &\leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\[
b \leftarrow a + \cdots
\]

\[
c \leftarrow b + \cdots
\]

\[
c \leftarrow \cdots
\]

\[
d \leftarrow \cdots
\]

\[
e \leftarrow d + \cdots
\]

\[
e \leftarrow \cdots
\]

\[
a \leftarrow a + e
\]
Fast interference test for a set of variables

\[
\begin{align*}
b & \leftarrow a + \cdots \\
c & \leftarrow b + \cdots \\
& \leftarrow c \\
d & \leftarrow \cdots \\
e & \leftarrow d + \cdots \\
& \leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\[ a \leftarrow \cdots \]

\[ b \leftarrow a + \cdots \]
\[ c \leftarrow b + \cdots \]
\[ \leftarrow c \]

\[ d \leftarrow \cdots \]

\[ e \leftarrow d + \cdots \]
\[ \leftarrow a + e \]
Fast interference test for a set of variables

\[
\begin{align*}
    b & \leftarrow a + \cdots \\
    c & \leftarrow b + \cdots \\
    & \leftarrow c \\
    d & \leftarrow \cdots \\
    e & \leftarrow d + \cdots \\
    & \leftarrow a + e \\
\end{align*}
\]
Fast interference test for a set of variables

\[
\begin{align*}
    b & \leftarrow a + \cdots \\
    c & \leftarrow b + \cdots \\
    & \leftarrow c \\
    d & \leftarrow \cdots \\
    e & \leftarrow d + \cdots \\
    & \leftarrow a + e
\end{align*}
\]
Fast interference test for a set of variables

\begin{align*}
  b &\leftarrow a + \cdots \\
  c &\leftarrow b + \cdots \\
     &\leftarrow c \\
  d &\leftarrow \cdots \\
  e &\leftarrow d + \cdots \\
     &\leftarrow a + e
\end{align*}
Fast interference test for a set of variables

\[
\begin{align*}
    a & \leftarrow \cdots \\
    b & \leftarrow a + \cdots \\
    c & \leftarrow b + \cdots \\
    & \leftarrow c \\
    d & \leftarrow \cdots \\
    e & \leftarrow d + \cdots \\
    & \leftarrow a + e
\end{align*}
\]
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. Also, 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. Also, 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 and liveness and avoid interference graph:

- Budimlić intersection test, still using liveness sets.
- Fast liveness checking of Boissinot et al. (CGO’08).
Speed-up for SPEC CINT2000: x2

Time to go out of SSA (valgrind cycles)
Default: Liveness sets + interference graph
Memory footprint reduction for SPEC CINT2000: x10

- Interference graph: half-size bit matrix.

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

Max of memory footprint
Outline

1. SSA foundations
   - Out-of-SSA translation

2. Correctness and code quality
   - Translation with copy insertions
   - Improving code quality
   - Qualitative experiments

3. Speed and memory footprint
   - Linear-time algorithm for coalescing congruence classes
   - Experimental results for speed and memory footprint

4. Conclusion
General framework

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

Results

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

Implementation

- No need to virtualize (at least for us)
- Simple implementation
### General framework
- Correctness clarified even for complex cases
- Two phases solution, based on coalescing

### Results
- Value-based interference as good as Sreedhar III
- Fast algorithm: Speed-up x2, memory reduction x10.

### Implementation
- No need to virtualize (at least for us)
- Simple implementation

Journal version should contain:
- Virtualization of register renaming constraints.
- All pseudo-codes, e.g., virtualization in general case.
Thank you!
Problems with SSA form: lack of loop unrolling breaks VM

This problem is probably one of the most serious in the RVM currently. When loop unrolling is disabled and SSA enabled the created IR is corrupt. The error has in the past look like we were suffering from the "lost copy" problem, but implementing a naive solution to this didn’t solve the problem. Their is sound logic behind the code so we need to identify a small test case where things are broken and then reason about what’s wrong in leave SSA. This has been attempted once (with the code that removes an element from the live set) but the problem no longer appears to surface here. Currently these optimizations are disabled but by RVM 3.0 they should be re-enable and this bug cured.
Potential bugs with conditional branches

Initial code

u = \ldots
v = \ldots

Br(u, B_3, B_4)

w = \phi(u, v)
\ldots = w

“Blind” Sreedhar III

u = \ldots
v = \ldots

v' = v
Br(u, B_3, B_4)

w = \phi(u, v')
\ldots = w

Wrong output code

w = \ldots

Br(w, B_3, B_4)

\ldots = w
Unfeasible out-of-SSA translation example

Initial code

After optimization

Needs edge splitting

\[
\begin{align*}
B_1: & \\
& u_0 \\
& \quad u_1 = \phi(u_0, u_2) \\
& \quad u_2 = u_1 - 1 \\
& \quad t_0 = u_2 \\
& \quad Br(u_2, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_2: & \\
& t_1 = \phi(t_0, t_2) \\
& t_2 = t_1 + \ldots \\
& Br(t_2, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_3: & \\
& \ldots = u_2 \\
\end{align*}
\]

\[
\begin{align*}
B_1: & \\
& u \\
& \quad Br_{\text{dec}}(u, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_2: & \\
& t_1 = \phi(u, t_2) \\
& t_2 = t_1 + \ldots \\
& Br(t_2, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_3: & \\
& \ldots = u \\
\end{align*}
\]

\[
\begin{align*}
B_1: & \\
& u \\
& \quad Br_{\text{dec}}(u, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_2: & \\
& t_1 = \phi(t_0, t_2) \\
& t_2 = t_1 + \ldots \\
& Br(t_2, B_1, B_2) \\
\end{align*}
\]

\[
\begin{align*}
B_3: & \\
& \ldots = u \\
\end{align*}
\]
Speedup
Qualitatives

![Qualitative Performance Chart]

- Intersection
- Sreedhar I
- Chaitin
- Value
- Value IS
- Sharing

Benoit Boissinot
Revisiting Out-of-SSA Translation