Studying Optimal Spilling in the light of SSA

Quentin Colombet, Florian Brandner and Alain Darte
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon

Journées compilation, Rennes, France, June 18-20 2012

Outline

Outline

Introduction

Register Allocation

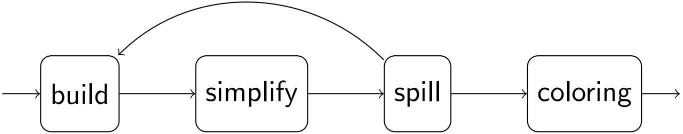
Map an unlimited number of virtual variables to actual physical registers.

Introduction

Register Allocation

Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach

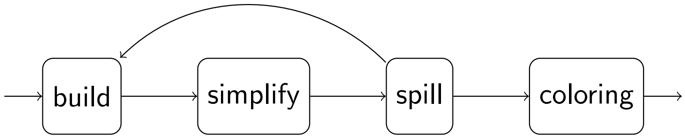


Introduction

Register Allocation

Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach



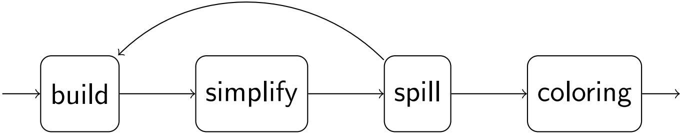
  - Build: Build the interference graph (IG)
  - Simplify: Apply Kempe's Algorithm
  - Spill: Evict one variable into memory (spill-everywhere)
  - Coloring: Assign color using order from simplify

Introduction

Register Allocation

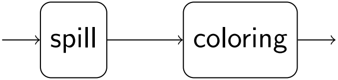
Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach



  - Build: Build the interference graph (IG)
  - Simplify: Apply Kempe's Algorithm
  - Spill: Evict one variable into memory (spill-everywhere)
  - Coloring: Assign color using order from simplify

- Decoupled approach

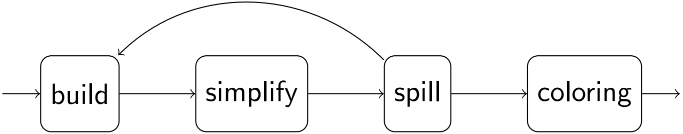


Introduction

Register Allocation

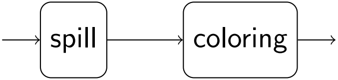
Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach



  - Build: Build the interference graph (IG)
  - Simplify: Apply Kempe's Algorithm
  - Spill: Evict one variable into memory (spill-everywhere)
  - Coloring: Assign color using order from simplify

- Decoupled approach



  - Spill: #live variables $\leq K$ for each program point
  - Coloring: Assign variables colors

Problematic

Context

- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)

Problematic

Context

- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)

Motivations

- Study impact of SSA on spilling
  - Chordal interference graphs help for assignment
  - Does SSA help for spilling too?
- Evaluate existing spilling heuristic

Problematic

Context

- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)

Motivations

- Study impact of SSA on spilling
  - Chordal interference graphs help for assignment
  - Does SSA help for spilling too?
- Evaluate existing spilling heuristic

Contributions

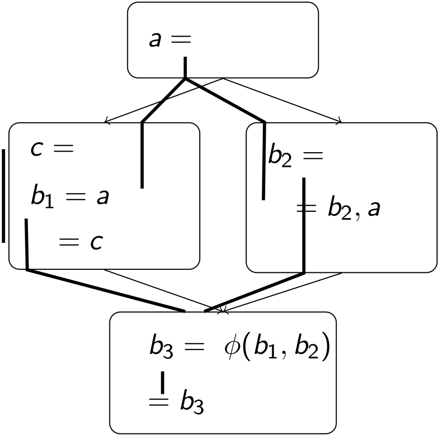
- Provide an exact formulation
- Exploit variable-to-variable copies
- Discuss existing spilling models

Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.

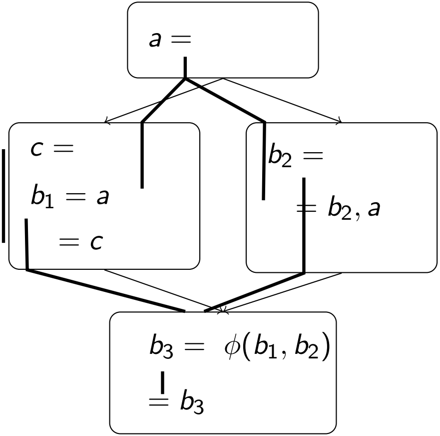
Static Single Assignment (SSA)

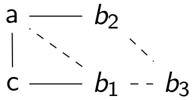
SSA provides sufficient split points, unless pre-coloring or aliasing is involved.



Static Single Assignment (SSA)

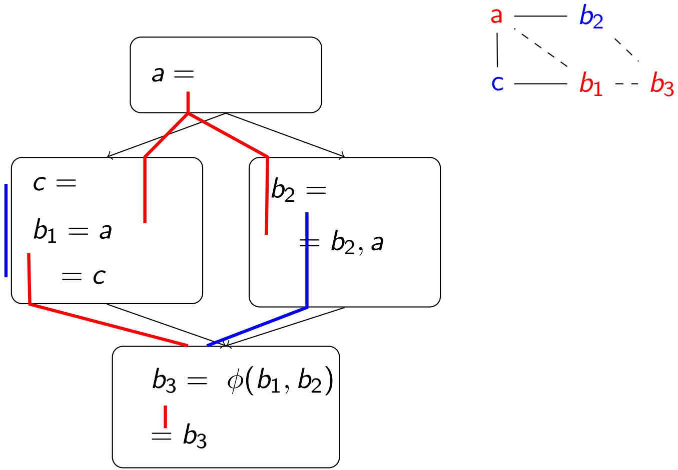
SSA provides sufficient split points, unless pre-coloring or aliasing is involved.





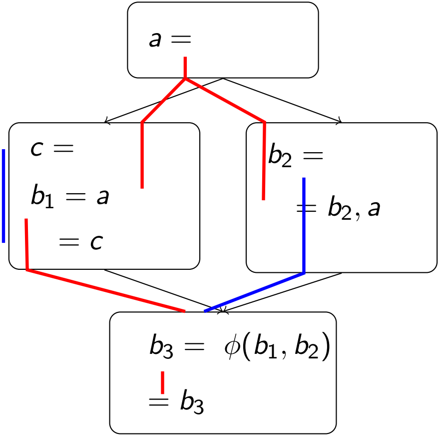
Static Single Assignment (SSA)

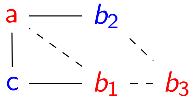
SSA provides sufficient split points, unless pre-coloring or aliasing is involved.



Static Single Assignment (SSA)

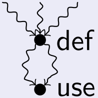
SSA provides sufficient split points, unless pre-coloring or aliasing is involved.





Properties

- Every use has at most one reaching definition
- For strict SSA: A definition dominates all its uses.
  I.e. it does not exist a path from the function entry to $v's$ use that does not traverse $v's$ definition.



Outline

Existing Approaches

Various 'optimal' approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)
- Multi-Commodity Network Flow
  - Koes & Goldstein
- Constrained Min-Cut
  - Ebner & Scholz & Krall

Existing Approaches

Various 'optimal' approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)
- Multi-Commodity Network Flow
  - Koes & Goldstein
- Constrained Min-Cut
  - Ebner & Scholz & Krall
- In the end these approaches rely on ILP.

Existing Approaches

Various 'optimal' approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)
- Multi-Commodity Network Flow
  - Koes & Goldstein
- Constrained Min-Cut
  - Ebner & Scholz & Krall
- In the end these approaches rely on ILP.

All of these formulations have *surprising* flaws!

Flaws: Liveness[1]

```
a = ...
b = a + 1 ↯
...
   = a
```

(a) before spilling

```
a = ...
b = a + 1 ↯
store a
...
load a
   = a
```

(b) ineffective spilling

Problem:

- Variables are either available in memory or register (exclusive)
- Load/store required to change availability
- Artificial interference between `a` and `b`

---

[1] Applies to: Appel, Koes; in other form also Goodwin

Flaws: Spurious Spill Code[2]

```
a = ...
while(...){
  if (...)
    store a
    ↯
    load a
       = a
  else
       = a
}
```

(a) Koes 1

[2] Applies to: Appel, Koes

Flaws: Spurious Spill Code[2]

```
a = ...
while(...){
  if (...)
    store a
    ↯
    load a
       = a
  else
       = a
}
```

(a) Koes 1

```
a = ...
store a
while(...){
  if (...)
    ↯
    load a
       = a
  else
    load a
       = a
  store a
}
```

(b) Koes 2

[2] Applies to: Appel, Koes

Flaws: Spurious Spill Code[2]

```
a = ...
while(...){
  if (...)
    store a
    ↯
    load a
       = a
  else
       = a
}
```

(a) Koes 1

```
a = ...
store a
while(...){
  if (...)
    ↯
    load a
       = a
  else
    load a
       = a
  store a
}
```

(b) Koes 2

```
a = ...
store a
load a
while(...){
  if (...)
    store a
    ↯
    load a
       = a
  else
       = a
}
```

(c) Koes 3

[2] Applies to: Appel, Koes

Limitations

Limitations of existing approaches:

- Rematerialization
  - Appel & George: None
  - related Goodwin & Wilken: Simple and partial
  - Koes & Goldstein: Simple and partial
  - Ebner & Scholz & Krall: None
- Support of SSA
  - Ebner & Scholz & Krall: Partial
  - Others: None

Limitations

Limitations of existing approaches:

- Rematerialization
  - Appel & George: None
  - related Goodwin & Wilken: Simple and partial
  - Koes & Goldstein: Simple and partial
  - Ebner & Scholz & Krall: None
- Support of SSA
  - Ebner & Scholz & Krall: Partial
  - Others: None

We design a new model

Outline

Our Formulation - Concepts

Express spilling using ILP:

- Availability around program points
  - Available in memory / in register
  - Non-exclusive!!
- Actions on program points
  - Load, store, rematerialization, ...
- Propagation along points
  - Along edges in the control flow graph (CFG)
  - Between operations within basic blocks accounting for uses/definitions

Our Formulation - Features

Basic

- Load/Store placement
- Simple rematerialization

This model can emulate all existing approaches.

Our Formulation - Features

Basic

- Load/Store placement
- Simple rematerialization

This model can emulate all existing approaches.

Extended

- Features of basic model
- Copy/SSA handling
- Generalized rematerialization

The extended model is able to emulate the basic one.

SSA Specificities

ϕ-Operations represent implicit copies:

$$a = \phi(b, c)$$
$$e = \phi(b, d)$$

(a) SSA form

SSA Specificities

ϕ-Operations represent implicit copies:

$$\begin{array}{c} \searrow \quad \swarrow \\ a = \phi(b, c) \\ e = \phi(b, d) \end{array}$$

(a) SSA form

$$\begin{array}{ccc} (a_\phi, e_\phi) = (b, b) & & (a_\phi, e_\phi) = (c, d) \\ \searrow & & \swarrow \\ & (a, e) = (a_\phi, e_\phi) & \end{array}$$

(b) Transform ϕ-operations

Simple approach: spilling as if not under SSA form.

SSA Specificities

ϕ-Operations represent implicit copies:

$$\begin{array}{c} \searrow \quad \swarrow \\ a = \phi(b, c) \\ e = \phi(b, d) \end{array}$$

(a) SSA form

$$\begin{array}{ccc} (a_\phi, e_\phi) = (b, b) & & (a_\phi, e_\phi) = (c, d) \\ \searrow & & \swarrow \\ & (a, e) = (a_\phi, e_\phi) & \end{array}$$

(b) Transform ϕ-operations

Simple approach: spilling as if not under SSA form.

Example: Appel & George with and without SSA:

- Spill cost:
  - 5% worse on average under SSA
  - Best improvement: 2%
  - Worst case: 50%

SSA Specificities

ϕ-Operations represent implicit copies:

$$\begin{array}{c} \searrow \quad \swarrow \\ a = \phi(b, c) \\ e = \phi(b, d) \end{array}$$

(a) SSA form

$$\begin{array}{ccc} (a_\phi, e_\phi) = (b, b) & & (a_\phi, e_\phi) = (c, d) \\ \searrow & & \swarrow \\ & (a, e) = (a_\phi, e_\phi) & \end{array}$$

(b) Transform ϕ-operations

Simple approach: spilling as if not under SSA form.

Example: Appel & George with and without SSA:

- Spill cost:
  - 5% worse on average under SSA
  - Best improvement: 2%
  - Worst case: 50%

$\Rightarrow$ Copies force variables to be in register.

Handling ϕ and Copy Operations

- Basic:
  - Replace ϕ-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations

Handling ϕ and Copy Operations

- Basic:
  - Replace ϕ-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations
- Optimistic:
  - Copies/ϕs are virtual operations
  - Propagate locations through them
  - Coalesce memory slots afterwards
  - Repair when memory slots cannot be shared

Handling ϕ and Copy Operations

- Basic:
  - Replace ϕ-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations
- Optimistic:
  - Copies/ϕs are virtual operations
  - Propagate locations through them
  - Coalesce memory slots afterwards
  - Repair when memory slots cannot be shared
- Pessimistic:
  - Replace ϕ-operations by copies
  - Copies are virtual operations
  - Propagate locations through a subset of copies
  - Coalesce remaing memory slots afterwards

Outline

Experiments

Setup:

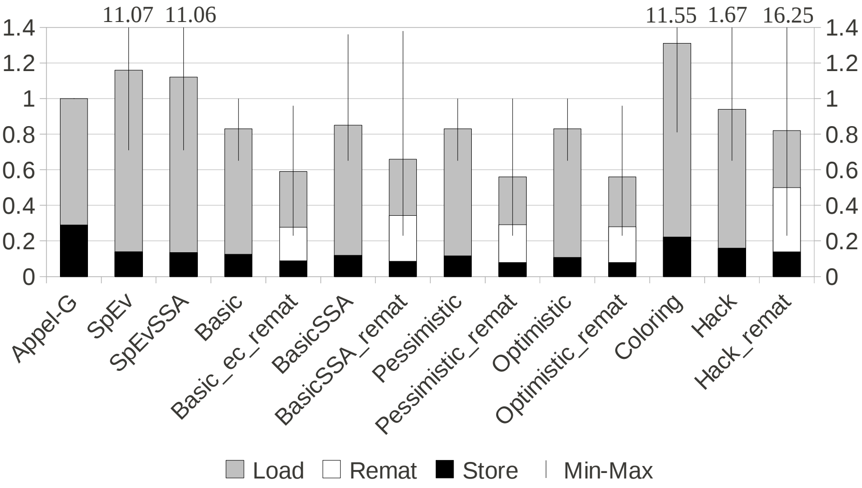
- Production compiler for STMicroelectronics ST2xx VLIW
  - 4-way parallel
  - 32KB direct mapped I-cache
  - 32KB 4-way set associative D-cache
  - 1 load/store per cycle
  - 3 cycles load-use delay
- Restricted to 8 registers
- SPEC2000 and EEMBC v1.1 benchmarks
- IBM CPLEX 12.2 with 1000s time limit

Experiments (2)

Configurations:

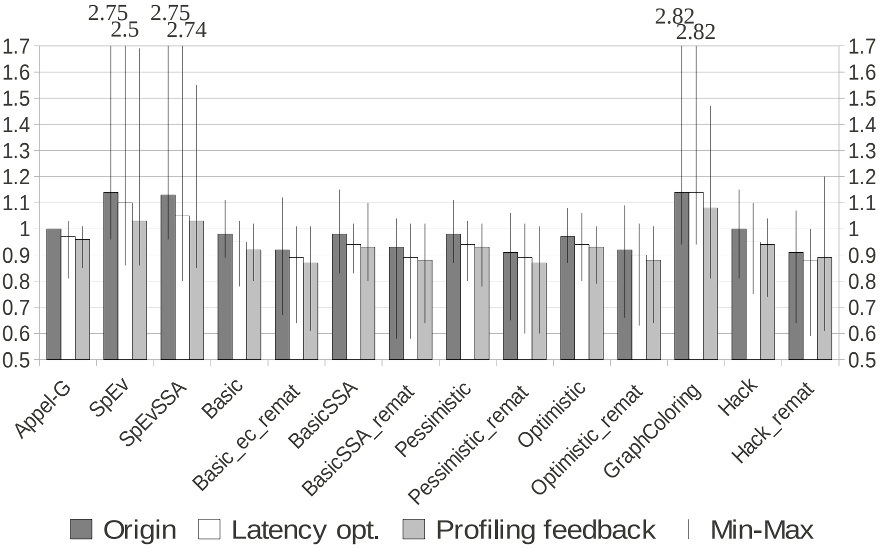
Appel-G.	Appel and George's ILP Formulation
Coloring	Heuristic using iterated register coalescing
SpEv	Basic formulation emulating spill everywhere
Basic	Our basic formulation
BasicSSA	Naive handling of SSA
Pessimistic	Extended formulation, pessimistic coalescing sets
Optimistic	Extended formulation, optimistic coalescing sets
SpEv_ssa	Emulation of spill everywhere under SSA
Hack	Hack's SSA-based spilling heuristic

Spill Costs (EEMBC)



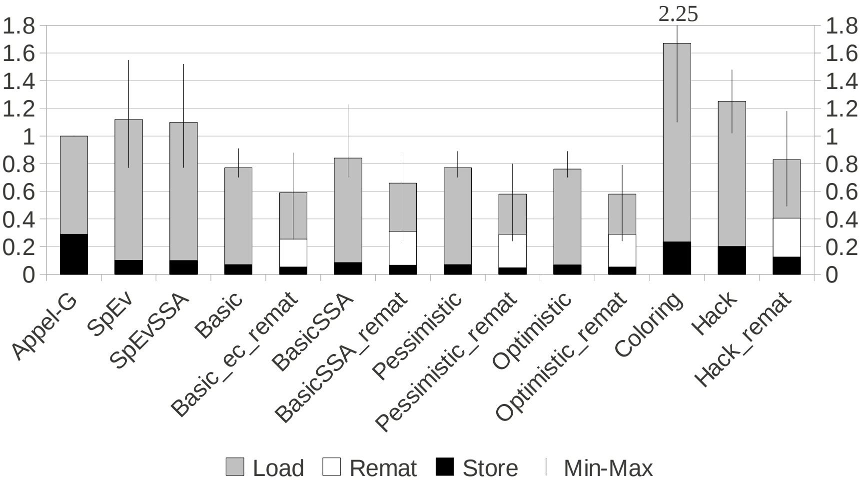
(Lower is better)

Runtime (EEMBC)



(Lower is better)

Spill Costs (SPEC)



(Lower is better)

Outline

Conclusion

- Accurate ILP formulation for spilling
  - Copy-relations and coalescing
  - Emulation of other approaches

Conclusion

- Accurate ILP formulation for spilling
    - Copy-relations and coalescing
    - Emulation of other approaches
- SSA form complicates matters
    - Parallel ϕ-semantics and memory coalescing
    - Ignoring ϕs gives unpredictable behavior

Conclusion

- Accurate ILP formulation for spilling
  - Copy-relations and coalescing
  - Emulation of other approaches
- SSA form complicates matters
  - Parallel ϕ-semantics and memory coalescing
  - Ignoring ϕs gives unpredictable behavior
- Placement of spill code is important
  - Spill costs alone are a bad metric
  - State of pipeline and memory subsystem have to be considered

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$
\begin{array}{ccc}
(a_\phi, e_\phi) = (b, b) & & (a_\phi, e_\phi) = (c, d) \\
\searrow & & \swarrow \\
& (a, e) = (a_\phi, e_\phi) &
\end{array}
$$

(a) **Transform ϕ-operations**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$

$$\searrow \qquad \swarrow$$

$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$(a_\phi, e_\phi) = (b, b) \qquad (a_\phi, e_\phi) = (c, d)$$

$$\searrow \qquad \swarrow$$

$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

$$\begin{gathered} b = ld @_b \\ (a_\phi, e_\phi) = (b, b) \\ @_{aa_\phi c} = st\ a_\phi \\ @_{ee_\phi d} = st\ e_\phi \end{gathered}$$

$$\searrow \qquad \swarrow$$

(c) **Spill**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program
- *Optimistic*: Always

$$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$
$$\searrow \qquad\qquad \swarrow$$
$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

$$b = ld@_b$$
$$(a_\phi, e_\phi) = (b, b)$$
$$@_{aa_\phi c} = st\ a_\phi$$
$$@_{ee_\phi d} = st\ e_\phi$$
$$\searrow \qquad \swarrow$$

(c) **Spill**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$

$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

$$b = ld@_b$$
$$(a_\phi, e_\phi) = (b, b)$$
$$@_{aa_\phi c} = st\ a_\phi$$
$$@_{ee_\phi d} = st\ e_\phi$$

(c) **Spill**

- *Optimistic*: Always

$$a = \phi(b, c)$$
$$e = \phi(b, d)$$

(a) **SSA form**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$
$$\searrow \qquad \swarrow$$
$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

$$b = ld\,@_b$$
$$(a_\phi, e_\phi) = (b, b)$$
$$@_{aa_\phi c} = st\ a_\phi$$
$$@_{ee_\phi d} = st\ e_\phi$$
$$\searrow \qquad \swarrow$$

(c) **Spill**

- *Optimistic*: Always

$$\searrow \qquad \swarrow$$
$$a = \phi(b, c)$$
$$e = \phi(b, d)$$

(a) **SSA form**

$$\searrow \qquad \swarrow$$
$$@_a = \phi(@_b, @_c)$$
$$@_e = \phi(@_b, @_d)$$

(b) **Spill**

Handling ϕ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- *Pessimistic*: If they do not interfer in the original program

$$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$

$$\searrow \qquad \swarrow$$

$$(a, e) = (a_\phi, e_\phi)$$

(a) **Transform ϕ-operations**

$$\{a, a_\phi, c\}\ \{e, e_\phi, d\}\ \{b\}$$

(b) **Build coalescing classes**

$$b = ld @_b$$
$$(a_\phi, e_\phi) = (b, b)$$
$$@_{aa_\phi c} = st\ a_\phi$$
$$@_{ee_\phi d} = st\ e_\phi$$
$$\searrow \qquad \swarrow$$

(c) **Spill**

- *Optimistic*: Always

$$\searrow \qquad \swarrow$$
$$a = \phi(b, c)$$
$$e = \phi(b, d)$$

(a) **SSA form**

$$\searrow \qquad \swarrow$$
$$@_a = \phi(@_b, @_c)$$
$$@_e = \phi(@_b, @_d)$$

(b) **Spill**

$$v_1 = ld @_b$$
$$@_{ac} = st\ v_1$$
$$v_2 = ld @_b$$
$$@_{ed} = st\ v_2$$
$$\searrow \qquad \swarrow$$

(c) **Coalescing and repairing**

Partial Rematerialization Support

```
a = remat
while(...){
    = a
    ↯

}
```

(a) Origin

```
a = remat
while(...){
    = a
    ↯
    a = remat
}
```

(b) Partial support

```

while(...){
    a = remat
    = a
    ↯
}
```

(c) Optimal

Partial SSA Support: Ebner et al.

- No particular constraints on ϕ-operations.
- Deal with ϕ-operations with mixed type of operands.
- $\Rightarrow$ Repairing cost not in the model.

Example:

$$
\begin{aligned}
@_a &= \phi(@_b, c) \\
@_e &= \phi(@_b, d)
\end{aligned}
$$

Program Point and ILP Variables

$$
\begin{array}{lcl}
\texttt{store a} & & \rho_{p,\texttt{a}} = ? \quad \mu_{p,\texttt{a}} = ? \\
l_{p,\texttt{a}} = ? \quad s_{p,\texttt{a}} = ? & \bullet p & \\
\texttt{load a} & & \bar{\rho}_{p,\texttt{a}} = 1 \quad \bar{\mu}_{p,\texttt{a}} = ? \\
 & \texttt{b = a + 1} &
\end{array}
$$

(a) A program point and its ILP variables

$$
\begin{array}{lccc}
 & \bullet p & \bar{\rho}_{p,\texttt{a}} = 1 & \bar{\mu}_{p,\texttt{a}} = ? \\
 & \texttt{b = a + 1} & \geq & \geq \\
\rho_{q,\texttt{b}} = 1 \quad \mu_{q,\texttt{b}} = 0 & \bullet q & \rho_{q,\texttt{a}} = ? & \mu_{q,\texttt{a}} = ?
\end{array}
$$

(b) Program points surrounding an instruction

Emulating other Approaches

Constraints to emulate Appel & George:

$$(\text{Appel})\ \bar{\mu}_{p,v} + \bar{\rho}_{p,v} = 1$$

Alternatively:

$$(\text{Appel}_l)\ l_{p,v} + \bar{\mu}_{p,v} \leq 1 \qquad (\text{Appel}_s)\ s_{p,v} + \bar{\rho}_{p,v} \leq 1$$

Constraints to emulate Koes & Goldstein:

$$(\text{Appel}_s)\ s_{p,v} + \bar{\rho}_{p,v} \leq 1$$

Discussion

- Huge gains in spill costs
  - Compared to 'optimal' techniques
  - Mostly due to elimination of stores
- Dynamic metrics
  - Lower cache miss rates
  - Lower number of loads/stores (-20%)
  - Lower number of operations executed (-8%)
  - Lower number of instruction bundles
- Marginal improvements in actual runtime
  - Costs of stores 'over-weighted'
  - Costs of secondary effects are missing (**pipeline**, cache, code layout)

Optimal Coalescing

```
a = ...
b = ...
↯
   = b
if (...)
   ↯
endif
c = φ(a, b)
↯
```

(a) Original

```
store a at @c
store b at @b
↯
load b
   = b
if (...)
   ↯
   mem_dup c = b
endif
c = φ(a, b)
```

(b) Optimistic/pessimistic

```
store a at @c
store b at @b
↯
load b
   = b
if (...)
   store b at @c
   ↯
endif
c = φ(a, b)
```

(c) Optimal