Formal Verification of Spilling Algorithms

Bachelor Talk

Julian Rosemann
Advisors: Prof. Gert Smolka, Sigurd Schneider

Saarland University
Department of Computer Science

Friday 24$^{\text{th}}$ February, 2017

Outline

Subproblems of Register Allocation

	r_1	r_2

```
let y  := 1  in
let z  := x  in


if z   >= w
then
  z
else

  w  + y
```

- Spilling: determine whether a variable is in the registers or in memory
- Register Assignment: determine in which register a variable resides
- Coalescing: reduce copy-instructions

Subproblems of Register Allocation

	r_1	r_2

```
let W  := w  in
let y  := 1  in
let z  := x  in
let Y  := y  in
let w  := W  in
if z   >= w
then
  z
else
  let y  := Y in
  w  + y
```

- Spilling: determine whether a variable is in the registers or in memory
- Register Assignment: determine in which register a variable resides
- Coalescing: reduce copy-instructions

Subproblems of Register Allocation

	r_1	r_2
	w	x
let W := w^1 in	w	⋮
let y^1 := 1 in	y	x
let z^2 := x^2 in	⋮	z
let Y := y^1 in	y	⋮
let w^1 := W in	w	⋮
if z^2 >= w^1	⋮	⋮
then		
z^2	⋮	z
else		
let y^2 := Y in	⋮	y
w^1 + y^2	w	y

- Spilling: determine whether a variable is in the registers or in memory
- Register Assignment: determine in which register a variable resides
- Coalescing: reduce copy-instructions

Subproblems of Register Allocation

	r_1	r_2
	w	x
let W := w^1 in	w	⋮
let y^1 := 1 in	y	⋮
let Y := y^1 in	y	⋮
let w^1 := W in	w	⋮
if x^2 >= w^1	⋮	⋮
then		
x^2	⋮	x
else		
let y^2 := Y in	⋮	y
w^1 + y^2	w	y

- Spilling: determine whether a variable is in the registers or in memory
- Register Assignment: determine in which register a variable resides
- Coalescing: reduce copy-instructions

Subproblems of Register Allocation

	r_1	r_2
	w	x
`let W := w`^{1} `in`	w	⋮
`let y`^{1} `:= 1 in`	y	⋮
`let Y := y`^{1} `in`	y	⋮
`let w`^{1} `:= W in`	w	⋮
`if x`^{2} `>= w`^{1}	⋮	⋮
`then`		
`x`^{2}	⋮	x
`else`		
`let y`^{2} `:= Y in`	⋮	y
`w`^{1} `+ y`^{2}	w	y

- Spilling: determine whether a variable is in the registers or in memory
- Register Assignment: determine in which register a variable resides
- Coalescing: reduce copy-instructions

only possible in SSA

Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

Global Register Allocation

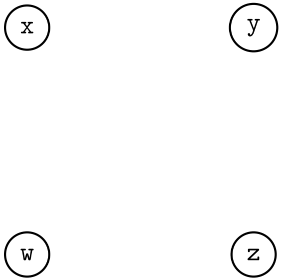
- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```

Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

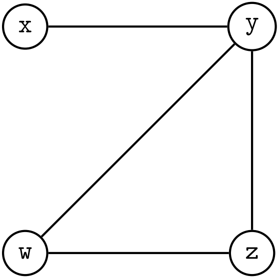
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

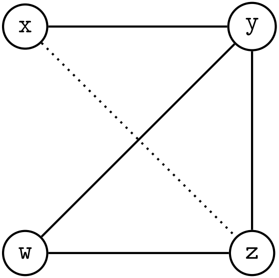
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

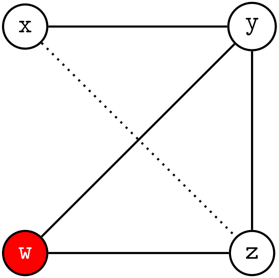
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

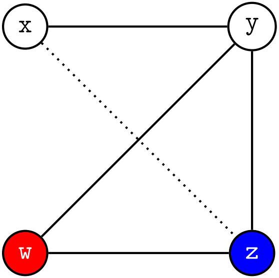
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

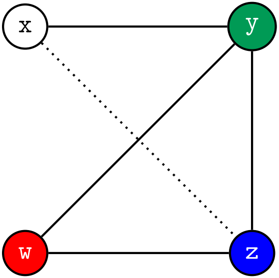
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

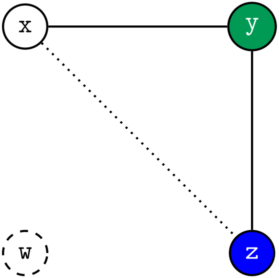
```
let y := 1 in
let z := x in

if z >= w
then
  z
else

  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

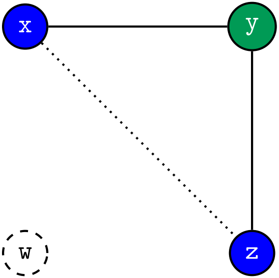
```
let y := 1 in
let z := x in
let w := W in
if z >= w
then
  z
else
  let w:= in
  w + y
```



Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

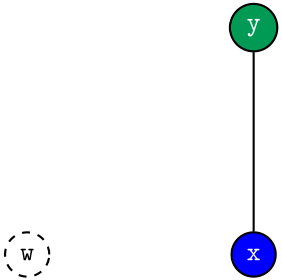
```
let y := 1 in

let w := W in
if x >= w
then
  x
else
  let w:= in
  w + y
```



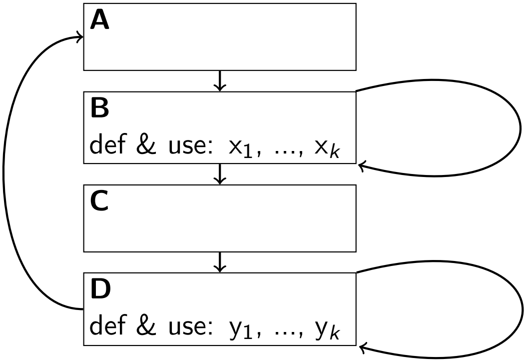
Global Register Allocation

- register assignment is independent of program point
- graph coloring algorithm *by Chaitin (1981)*

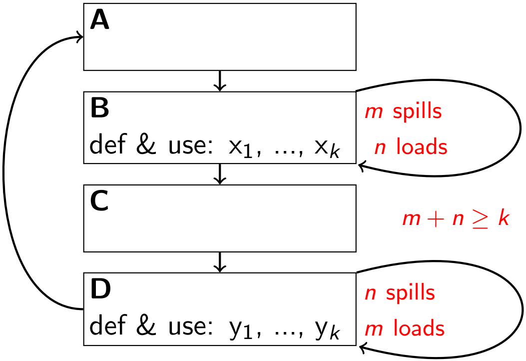
```
let y := 1 in

let w := W in
if z >= w
then
  z
else
  let w:= in
  w + y
```



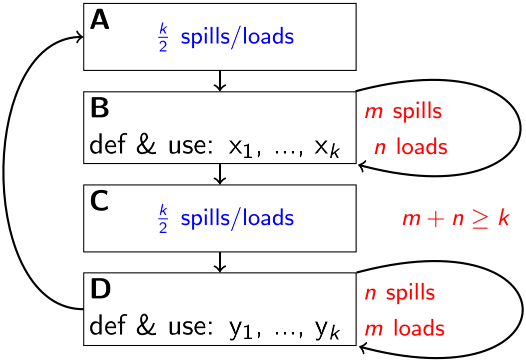
Problems with Global Register Allocation



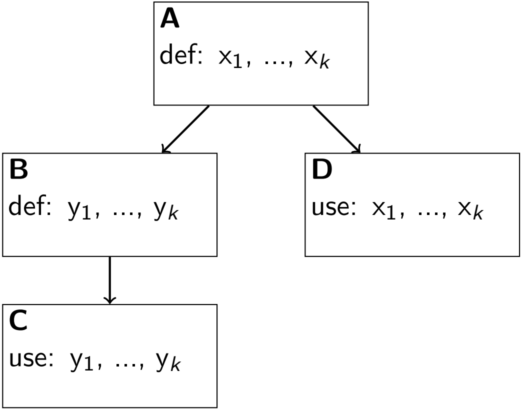
Problems with Global Register Allocation



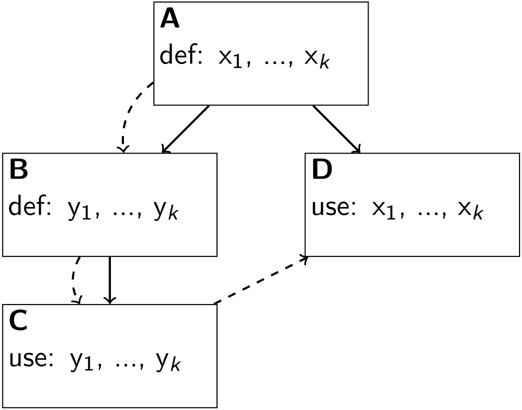
Problems with Global Register Allocation



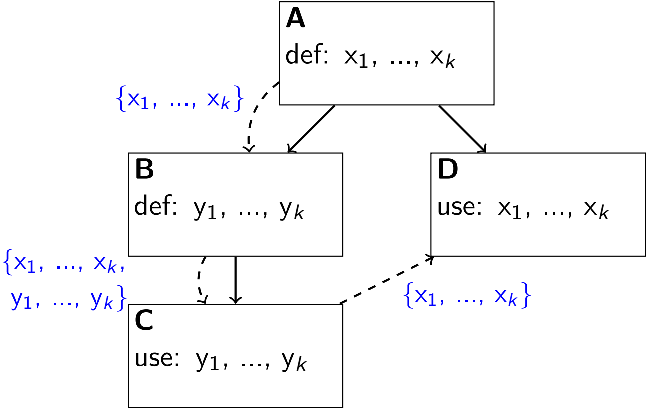
Linear Scan *by Poletto & Sarkar (1999)*



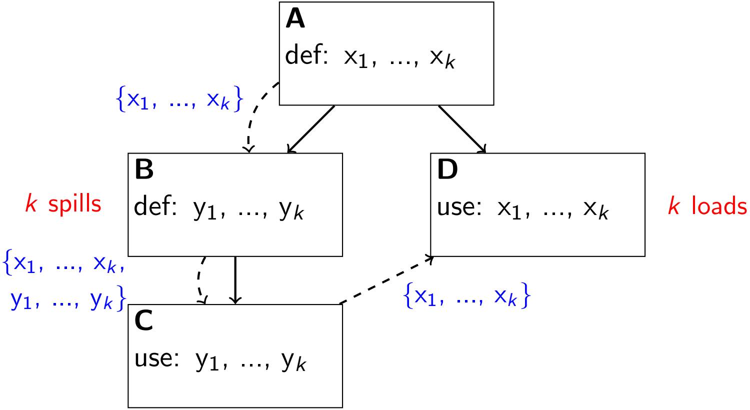
Linear Scan *by Poletto & Sarkar (1999)*



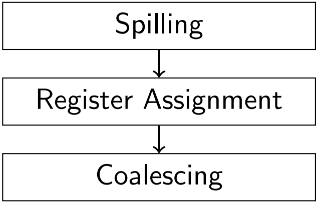
Linear Scan *by Poletto & Sarkar (1999)*



Linear Scan *by Poletto & Sarkar (1999)*



Register Allocation in SSA *by Hack et al. (2006)*

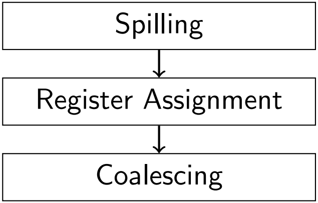


Register Allocation in SSA *by Hack et al. (2006)*

Spilling
↓
Register Assignment
↓
Coalescing

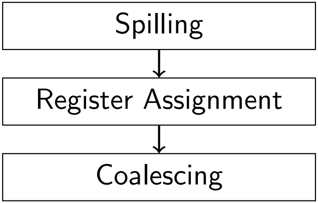
- inference graphs of SSA-programs are chordal

Register Allocation in SSA *by Hack et al. (2006)*



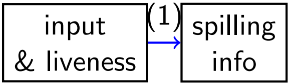
- inference graphs of SSA-programs are chordal
- efficient algorithm for register assignment

Register Allocation in SSA *by Hack et al. (2006)*

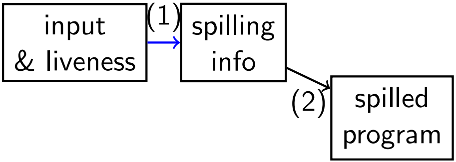


- inference graphs of SSA-programs are chordal
- efficient algorithm for register assignment
- coalescing using a heuristic

input
& liveness

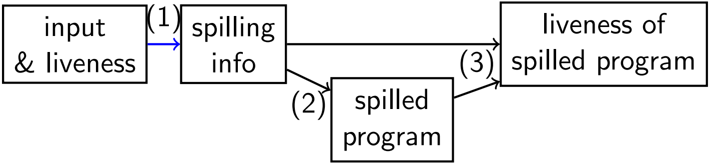


(1) spilling algorithm



(1) spilling algorithm

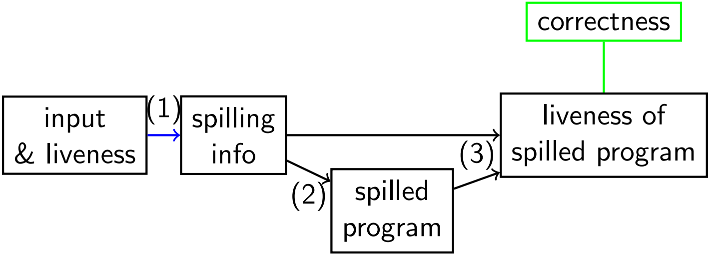
(2) doSpill



(1) spilling algorithm

(2) doSpill

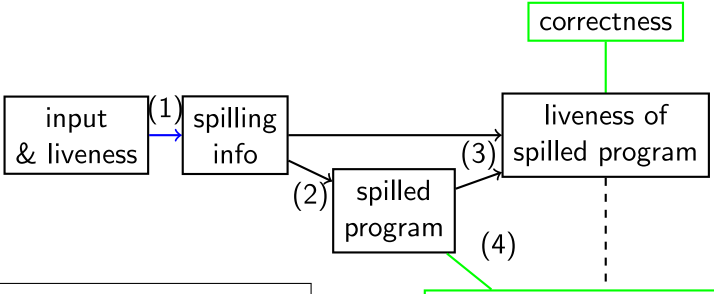
(3) spilledLive



(1) spilling algorithm

(2) doSpill

(3) spilledLive



(1) spilling algorithm

(2) doSpill

(3) spilledLive

(4) correctness predicate

valid spilling

(a) variables in registers

(b) register bound

(c) program equivalence

Spilling Algorithms

StupSpill:

SimplSpill:

Spilling Algorithms

StupSpill:

- at any statement:

SimplSpill:

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything

SimplSpill:

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything

SimplSpill:

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate $<$80 l.

SimplSpill:

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate $<$80 l.

SimplSpill:

- at any statement:

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate $<$80 l.

SimplSpill:

- at any statement:
  - load as little as possible

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate $<$80 l.

SimplSpill:

- at any statement:
  - load as little as possible
  - spill as little as possible

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate $<$80 l.

SimplSpill:

- at any statement:
  - load as little as possible
  - spill as little as possible

Spilling Algorithms

StupSpill:

- at any statement:
  - load everything
  - spill everything
- specification of algorithm 30 l.
- verifying the predicate <80 l.

SimplSpill:

- at any statement:
  - load as little as possible
  - spill as little as possible
- specification of algorithm 70 l.
- verifying the predicate < 500 l.

Compute the Spilled Program

$$\text{doSpill}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}})) =$$

Compute the Spilled Program

$$\text{doSpill}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}})) = \begin{array}{l} \texttt{let } X_1 = x_1 \texttt{ in } \texttt{...} \\ \texttt{let } y_1 = Y_1 \texttt{ in } \texttt{...} \\ s \end{array}$$

Compute the Spilled Program

$$\text{doSpill}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}})) = \begin{array}{l} \texttt{let } X_1 = x_1 \texttt{ in } ... \\ \texttt{let } y_1 = Y_1 \texttt{ in } ... \\ s \end{array}$$

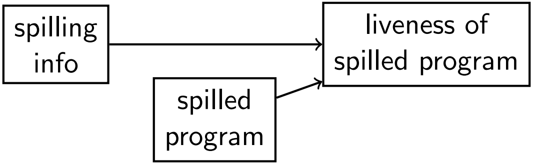
- `doSpill` is called recursively on every substatement

Compute the Spilled Program

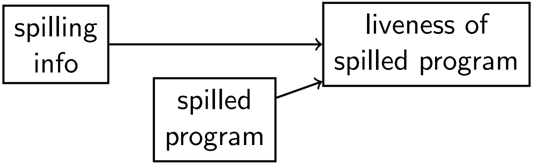
$$\text{doSpill}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}})) = \begin{array}{l}\texttt{let } X_1 = x_1 \texttt{ in } \texttt{...} \\ \texttt{let } y_1 = Y_1 \texttt{ in } \texttt{...} \\ s\end{array}$$

- `doSpill` is called recursively on every substatement
- the parameters are adjusted to the spilling information

Liveness of the spilled program

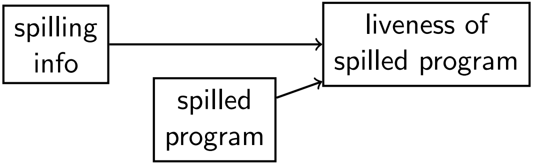


Liveness of the spilled program



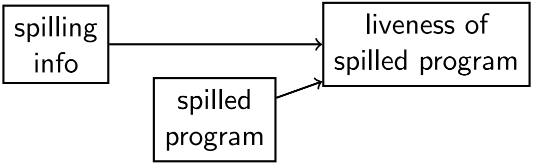
- spilling information includes liveness at function definitions

Liveness of the spilled program



- spilling information includes liveness at function definitions
- construct liveness in one pass over the program

Liveness of the spilled program



- spilling information includes liveness at function definitions
- construct liveness in one pass over the program
- proof of correctness was involving

Correctness Predicate

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

Z
Σ
$x \in R$
$x \in M$
k
s
S
L
$\_$

Correctness Predicate

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

Z	list of parameters of defined functions
Σ	list of expected live variables at function heads
$x \in R$	$:\Leftrightarrow$ current value is in a register
$x \in M$	$:\Leftrightarrow$ current value is in the memory
k	
s	
S	
L	
$\_$	

Correctness Predicate

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

Z	list of parameters of defined functions
Σ	list of expected live variables at function heads
$x \in R$	:⇔ current value is in a register
$x \in M$	:⇔ current value is in the memory
k	register bound
s	
S	
L	
$\_$	

Correctness Predicate

$$Z \mid \Sigma \mid R \mid M \vdash \textbf{spill}_{\mathbf{k}}\ \ s : (S, L, \_)$$

Z	list of parameters of defined functions
Σ	list of expected live variables at function heads
$x \in R$	:⇔ current value is in a register
$x \in M$	:⇔ current value is in the memory
k	register bound
s	source program
S	
L	
$\_$	

Correctness Predicate

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

Z	list of parameters of defined functions
Σ	list of expected live variables at function heads
$x \in R$	$:\Leftrightarrow$ current value is in a register
$x \in M$	$:\Leftrightarrow$ current value is in the memory
k	register bound
s	source program
S	variables to be spilled
L	variables to be loaded
$\_$	additional information for functions

Soundness of the Predicate

Let s be renamed apart, $|R| \leq k$ and applications only have variables as arguments. If

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

then

Soundness of the Predicate

Let s be renamed apart, $|R| \leq k$ and applications only have variables as arguments. If

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

then

- any variable is in a register whenever it is used

Soundness of the Predicate

Let s be renamed apart, $|R| \leq k$ and applications only have variables as arguments. If

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

then

- any variable is in a register whenever it is used
- at any program point the register contains at most k variables

Soundness of the Predicate

Let s be renamed apart, $|R| \leq k$ and applications only have variables as arguments. If

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$$

then

- any variable is in a register whenever it is used
- at any program point the register contains at most k variables
- the translation preserves program equivalence

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm: Verification:

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill "furthest-first"

Verification:

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill "furthest-first"
- pull spills and loads out of loops

Verification:

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill "furthest-first"
- pull spills and loads out of loops

Verification:

- verify the method computing distance to next-use

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill “furthest-first”
- pull spills and loads out of loops

Verification:

- verify the method computing distance to next-use
- keep track of register and memory state

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill "furthest-first"
- pull spills and loads out of loops

Verification:

- verify the method computing distance to next-use
- keep track of register and memory state
- induction on the statement

Verification of other Spilling Algorithms

e.g.: spilling algorithm *by Braun and Hack (2009)*:

Concepts of the algorithm:

- spill "furthest-first"
- pull spills and loads out of loops

Verification:

- verify the method computing distance to next-use
- keep track of register and memory state
- induction on the statement
- more involving than verification of SimplSpill, but similar

Translation Validation with Success Guarantee

Translation Validation with Success Guarantee

- untrusted spilling algorithm computes spilling

Translation Validation with Success Guarantee

- untrusted spilling algorithm computes spilling
- iterate over program flow, liveness and spilling information:
  - if free variables of the expression are not in the registers, load them
  - if register set exceeds bound, spill variables

Translation Validation with Success Guarantee

- untrusted spilling algorithm computes spilling
- iterate over program flow, liveness and spilling information:
  - if free variables of the expression are not in the registers, load them
  - if register set exceeds bound, spill variables
- Properties:
  - yields always a valid spilling
  - if input is a valid spilling it is not modified

References

Matthias Braun and Sebastian Hack. "Register spilling and live-range splitting for SSA-form programs". In: 2009.

Gregory J Chaitin et al. "Register allocation via coloring". In: 1981.

Sebastian Hack, Daniel Grund, and Gerhard Goos. "Towards Register Allocation for Programs in SSA-Form". In: 2005.

Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation". In: 1999.

Sigurd Schneider, Gert Smolka, and Sebastian Hack. "A Linear First-Order Functional Intermediate Language for Verified Compilers". In: 2015.

Liveness

$$\frac{\mathsf{fv}\, e \subseteq X \quad X_s \setminus \{x\} \subseteq X \quad x \in X_s \quad Z \,|\, \Lambda \vdash \textbf{live}\ s : X_s}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s) : X}\ \textsc{LiveLet}$$

$$\frac{\mathsf{fv}\, e \subseteq X}{Z \,|\, \Lambda \vdash \textbf{live}\ e : X}\ \textsc{LiveReturn}$$

$$\frac{\mathsf{fv}\, e \cup X_{s_1} \cup X_{s_2} \subseteq X \quad Z \,|\, \Lambda \vdash \textbf{live}\ s_1 : X_{s_1} \quad Z \,|\, \Lambda \vdash \textbf{live}\ s_2 : X_{s_2}}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) : X}\ \textsc{LiveIf}$$

$$\frac{\mathsf{fv}\, \overline{e} \subseteq X \quad \Lambda_f \setminus Z_f \subseteq X}{Z \,|\, \Lambda \vdash \textbf{live}\ f\ \ \overline{e} : X}\ \textsc{LiveApp}$$

$$\frac{X_{s_2} \subseteq X \quad \overline{x} \subseteq X_{s_1} \quad f : \overline{x}; Z \,|\, X_{s_1} :: \Lambda \vdash \textbf{live}\ s_1 : X_{s_1} \quad f : \overline{x}; Z \,|\, X_{s_2} :: \Lambda \vdash \textbf{live}\ s_2 : X_{s_2}}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s_1\ \texttt{in}\ s_2) : X}$$

$$\textsc{LiveFun}$$

Correctness Predicate

$$\frac{L \subseteq M \qquad |R \setminus K \cup L| \leq k \qquad Z \mid \Sigma \mid R \setminus K \cup L \mid M \vdash \mathbf{spill_k}\ \ s : (\emptyset, \emptyset, \_)}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (\emptyset, L, \_)}\ \textsc{SpillLoad}$$

$$\frac{S \subseteq R \qquad Z \mid \Sigma \mid R \mid M \cup S \vdash \mathbf{spill_k}\ \ s : (\emptyset, L, \_)}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)}\ \textsc{SpillSpill}$$

$$\frac{\mathrm{fv}\, e \subseteq R \qquad |R \setminus K_x \cup \{x\}| \leq k \qquad Z \mid \Sigma \mid R \setminus K_x \cup \{x\} \mid M \vdash \mathbf{spill_k}\ \ s}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s) : (\emptyset, \emptyset)}\ \textsc{SpillLet}$$

$$\frac{\mathrm{fv}\, e \subseteq R}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ e : (\emptyset, \emptyset)}\ \textsc{SpillReturn}$$

$$\frac{(R_f, M_f) := \Sigma_f \qquad R_f \setminus Z_f \subseteq R \qquad M_f \setminus Z_f \subseteq M \qquad \mathrm{fv}\, \overline{y} \subseteq R \cup M}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (f\ \ \overline{y}) : (\emptyset, \emptyset, (R, M))}\ \textsc{SpillApp}$$

$$\frac{|R_f| \leq k \qquad Z \mid \mathrm{merge}\, \Sigma \vdash \mathbf{live}\ s_1 : R_f \cup M_f \qquad f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R_f \mid M_f \vdash \mathbf{spill_k}\ \ s_1 \qquad f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s_2}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\texttt{fun}\ f\ \ \overline{x}\ \texttt{:=}\ s_1\ \texttt{in}\ s_2) : (\emptyset, \emptyset, (R_f, M_f))}\ \textsc{SpillFun}$$

$$\mathrm{merge}\ \ \mathrm{nil} := \mathrm{nil}$$

$$\mathrm{merge}\ (R_f, M_f); \Sigma := R_f \cup M_f; \mathrm{merge}\, \Sigma$$

vir-Predicate

$$\frac{x \in \mathcal{V}_R \quad y \in \mathcal{V}_M \quad \textbf{vir}\, s}{\textbf{vir}\, \texttt{let}\ x\ \texttt{:=}\ y\ \texttt{in}\ s}\ \textsc{ViRLoad}$$

$$\frac{\mathrm{fv}\, e \subseteq \mathcal{V}_R \quad \textbf{vir}\, s}{\textbf{vir}\, \texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s}\ \textsc{ViRLet}$$

$$\frac{\mathrm{fv}\, e \subseteq \mathcal{V}_R}{\textbf{vir}\, e}\ \textsc{ViRReturn}$$

$$\frac{\mathrm{fv}\, e \subseteq \mathcal{V}_R \quad \textbf{vir}\, s \quad \textbf{vir}\, t}{\textbf{vir}\, \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t}\ \textsc{ViRIf}$$

$$\frac{}{\textbf{vir}\, f\ \overline{y}}\ \textsc{ViRApp}$$

$$\frac{\textbf{vir}\, s \quad \textbf{vir}\, t}{\textbf{vir}\, \texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s\ \texttt{in}\ t}\ \textsc{ViRFun}$$