Liveness Analysis in Explicitly-Parallel Programs

Alain Darte **Alexandre Isoard** Tomofumi Yuki

Compsys, LIP, Lyon

6th International Workshop on
Polyhedral Compilation Techniques
January 19, 2016
Prague, Czech Republic

Use of liveness analysis

Necessary for memory reuse:

- Register allocation: interference graph
- Array contraction: conflicting relation
- Wire usage: bitwidth analysis

Important information for:

- Communication: live-in/live-out sets (inlining, offloading)
- Memory footprint: cache prediction
- Lower/upper bounds on memory usage

Why revisit liveness analysis?

Several variants:

- Value-based or memory-based analysis
- Liveness sets or interference graphs
- Control flow graphs: basic blocks, SSA, SSI, etc...

What about task graphs? Or parallel specifications in general?

- Alpha, OpenStream
- CUDA/OpenCL
- OpenMP (loop parallelism), OpenMP 4.0 (dependent tasks)
- X10 (async, finish, clocks)
- ...

Key contribution

Liveness analysis based on "happens-before" relations.

Key remarks:

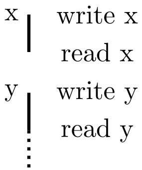
- No global notion of time
- Polyhedral fragments of OpenMP, X10, ... can be handled
- Room for approximations

Outline

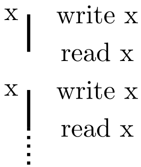
- Introduction
- Recap of sequential case
- Direct extensions
- Using happens-before relation
- Some properties
- Conclusion

Register allocation

```
x = ...;
y = x + ...;
... = y;
```




```
x = ...;
x = x + ...;
... = x;
```



Array folding

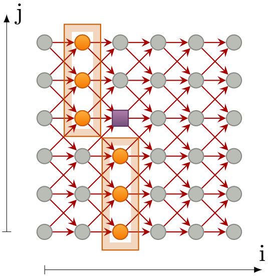
```
c[0] = ...;
for(i=0; i<n; ++i)
   c[i+1] = c[i] + ...;
```

```
c = ...;
for(i=0; i<n; ++i)
   c = c + ...;
```

	$\vdots$		$\vdots$
c[i-1]	write c_{i-1}	c	write c
	read c_{i-1}		read c
c[i]	write c_i	c	write c
	read c_i		read c
c[i+1]	write c_{i+1}	c	write c
	read c_{i+1}		read c
	$\vdots$		$\vdots$

Jacobi-1D: Sequential

```
for(i=0; j<n; ++i)
  for(j=0; j<n; ++j)
    A[i+1][j] = A[i][j-1] + A[i][j] + A[i][j+1];
```



A[i][j] $\mapsto$ A[(j-i)%(n+1)]

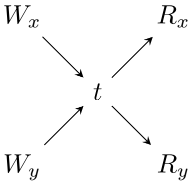
Simultaneously live: "Crossproduct"

Definition (Conflict)

Two memory cells x and y conflicts iff there exists a time step t where they are both live.

W_x write of x

R_x read of x



$\iff x \bowtie y$

Liveness at a given time step with iscc

```
# Inputs
Params := [n] -> { : n >= 0 };
Domain := [n] -> { S[i,j] : 0 <= i, j < n };
Read := [n] -> { S[i,j] -> A[i-1,j-1]; S[i,j] -> A[i-1,j];
                 S[i,j] -> A[i-1,j+1] } * Domain;
Write := [n] -> { S[i,j] -> A[i,j] } * Domain;
Sched := [n] -> { S[i,j] -> [i,j] };

# Operators
Prev := { [i,j]->[k,l]: i<k or (i=k and j<l) };
Preveq := { [i,j]->[k,l]: i<k or (i=k and j<=l) };
WriteBeforeTStep := (Prev^-1).(Sched^-1).Write;
ReadAfterTStep := Preveq.(Sched^-1).Read;

# Liveness and conflicts
Live := WriteBeforeTStep * ReadAfterTStep;
Conflict := (Live^-1).Live;
Delta := deltas Conflict;
```

☛
$$\text{Delta}(n) = \begin{array}{l} \{(1, i_1) \mid i_1 \leq 0,\ n \geq 3,\ i_1 \geq 1 - n\} \cup \\ \{(0, i_1) \mid i_1 \geq 1 - n,\ n \geq 2,\ i_1 \leq -1 + n\} \cup \\ \{(-1, i_1) \mid i_1 \geq 0,\ n \geq 3,\ i_1 \leq -1 + n\} \end{array}$$

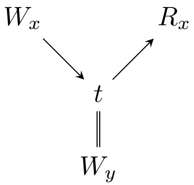
Simultaneously live: "Triangle" (Register allocation)

Definition (Conflict)

Two memory cells x and y conflicts iff one is live at a write of the other.

W_x write of x

R_x read of x

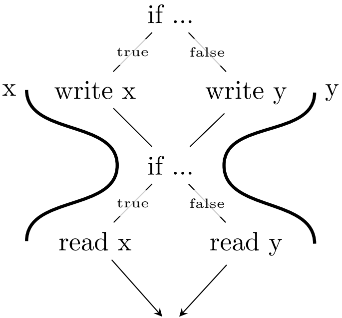


$\vee$ sym $\iff$ $x \bowtie y$

“Crossproduct” vs “Triangle”

```
if(...) x = ...;
else    y = ...;

if(...) ... = x;
else    ... = y;
```



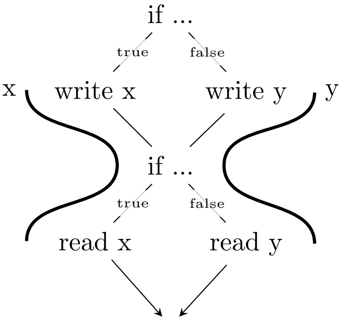
Crossproduct Will detect a conflict

Triangle Will not detect a conflict

"Crossproduct" vs "Triangle"

```
if(...) x = ...;
else    y = ...;

if(...) ... = x;
else    ... = y;
```



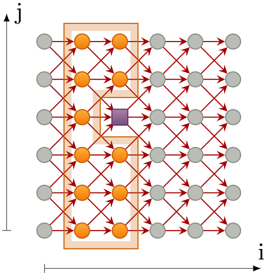
Crossproduct Will detect a conflict

Triangle Will not detect a conflict

☛ Valid because no legal trace are affected

Jacobi-1D: Parallel

```
for(i=0; j<n; ++i)
#pragma omp parallel for
   for(j=0; j<n; ++j)
      A[i+1][j] = A[i][j-1] + A[i][j] + A[i][j+1];
```



A[i][j] $\mapsto$ A[i%2][j]

How general?

Inner parallelism Almost the same as sequential.

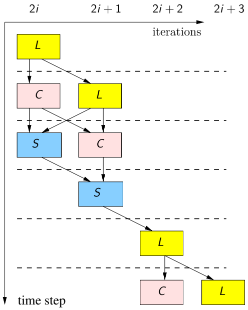
Series parallel Can use a careful hierarchical approach.

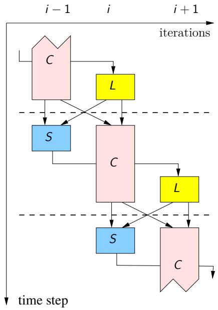
How general?

Inner parallelism Almost the same as sequential.

Series parallel Can use a careful hierarchical approach.

Software pipelining Harder to get a concept of “time”.



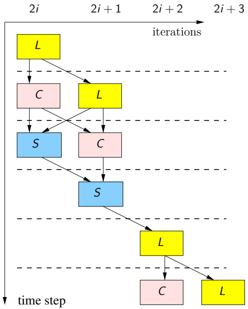


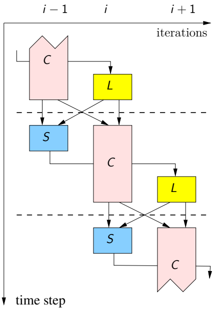
How general?

Inner parallelism Almost the same as sequential.

Series parallel Can use a careful hierarchical approach.

Software pipelining Harder to get a concept of "time".





$S(i-1) \bowtie C(i)$ and $C(i) \bowtie L(i+1)$ but not $S(i-1) \bowtie L(i+1)$.

☛ Not a clique!

Potentially simultaneously live

Definition (Conflict)

Two memory cells x and y conflicts iff there exists a trace where one is live at a write of the other.

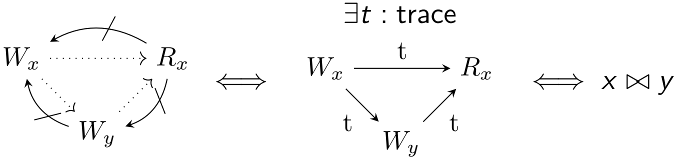
Definition (Happens-before)

a happens-before b iff, in all traces where a and b are executed, a is executed before b.

If:

- A trace is assumed possible iff it is allowed by happens-before
- Happens-before is a partial order (transitive closure)

then:



Potentially simultaneously live

Definition (Conflict)

Two memory cells x and y conflicts iff there exists a trace where one is live at a write of the other.

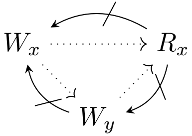
Definition (Happens-before)

a happens-before b iff, in all traces where a and b are executed, a is executed before b.

If:

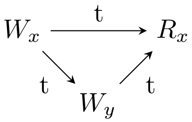
- A trace is assumed possible iff it is allowed by happens-before

then:



$\Longleftarrow$

$\exists t$: trace



$\Longleftrightarrow x \bowtie y$

Folk corollary

Corollary (when happens-before is a partial order)

A source-to-source memory transformation that respects the conflicts preserves <u>all</u> the parallelism captured by the happens-before relation.

```
if(b)       x = ...;          if(b)       x = ...;
if(not b) y = ...;     =      if(c)       ... = x;
if(c)       ... = x;          if(not b) y = ...;
if(not c) ... = y;            if(not c) ... = y;
```

Folk corollary

Corollary (when happens-before is a partial order)

A source-to-source memory transformation that respects the conflicts preserves all the parallelism captured by the happens-before relation.

```
if(b)      x = ...;
if(not b)  y = ...;
if(c)      ... = x;
if(not c)  ... = y;
```

=

```
if(b)      x = ...;
if(c)      ... = x;
if(not b)  y = ...;
if(not c)  ... = y;
```

traces: $W_x \to R_x$ $\quad W_x \to R_y$ $\quad W_y \to R_x$ $\quad W_y \to R_y$

Folk corollary

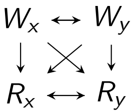
Corollary (when happens-before is a partial order)

A source-to-source memory transformation that respects the conflicts preserves <u>all</u> the parallelism captured by the happens-before relation.

```
if(b)       x = ...;              if(b)       x = ...;
if(not b)   y = ...;      =       if(c)       ... = x;
if(c)       ... = x;              if(not b)   y = ...;
if(not c)   ... = y;              if(not c)   ... = y;
```

traces: $\begin{array}{cccc} W_x & W_x & W_y & W_y \\ \downarrow & \downarrow & \downarrow & \downarrow \\ R_x & R_y & R_x & R_y \end{array}$

happens-before:



Folk corollary

Corollary (when happens-before is a partial order)

A source-to-source memory transformation that respects the conflicts preserves all the parallelism captured by the happens-before relation.

```
if(b)      x = ...;
if(not b)  x = ...;
if(c)      ... = x;
if(not c)  ... = x;
```

$\neq$

```
if(b)      x = ...;
if(c)      ... = x;
if(not b)  x = ...;
if(not c)  ... = x;
```

traces: $\begin{matrix} W_x & W_x & W_y & W_y \\ \downarrow & \downarrow & \downarrow & \downarrow \\ R_x & R_y & R_x & R_y \end{matrix}$

happens-before: $\begin{matrix} W_x & \leftrightarrow & W_y \\ \downarrow & \times & \downarrow \\ R_x & \leftrightarrow & R_y \end{matrix}$

Theorem

Theorem (when happens-before is a partial order)

If no dead code, no undefined read, but possibly races, the interference graph is the complement of a comparability graph: the reuse graph.

Consequences:

- Perfect graph: max color = max clique;
- Dilworth theorem: coloring polynomially computable;
- Link with "reuse graph" of work on (Q)UOV.

But not particularly useful in the polyhedral framework: would require enumeration of iterations.

Wrap-up

Trace-independent: if allocation respects $\bowtie$ it is valid for any trace.

Happens-before: quite general, handle if conditions (conservatively), do not handle critical sections (will assume possible conflict).

Optimality: size = max clique, polynomially computable (Dilworth) if graph is given in extension (unlike polyhedral optimization).

Source-to-source transformation: contraction can be expressed in the same specification model, without constraining parallelism further.

Conclusion

Possible future work:

- Critical sections are not captured by happens-before
  ☛ hierarchical happens-before?
- Explicit handling of control ☛ directly exploiting CFG?
- Code generation from happens-before relation?

☛ Towards a better understanding of parallel languages: semantics, static analysis, and links with the runtime.

Buffer Sizes

Sequential Memory Size	Pipelined Memory Size
jacobi-1d-imper	
$\mathtt{A}[2s_1 + s_2]$ $\mathtt{B}[2s_1 + s_2 - 1]$	$\mathtt{A}[2s_1 + 2s_2]$ $\mathtt{B}[2s_1 + 2s_2 - 2]$
jacobi-2d-imper	
$\mathtt{A}[2s_1 + s_2, \min(2s_1, s_2 + 1) + s_3]$ $\mathtt{B}[2s_1 + s_2 - 1, \min(2s_1, s_2 + 1) + s_3 - 1]$	$\mathtt{A}[2s_1 + s_2, \min(2s_1, s_2 + 1) + 2s_3]$ $\mathtt{B}[2s_1 + s_2 - 1, \min(2s_1, s_2 + 1) + 2s_3 - 2]$
seidel-2d	
$\mathtt{A}\begin{bmatrix} s_1 + s_2 + 1, \\ \min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + s_3 \end{bmatrix}$	$\mathtt{A}\begin{bmatrix} s_1 + s_2 + 1, \\ \min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + 2s_3 \end{bmatrix}$
gemm	
$\mathtt{A}[s_1, s_3]$ $\mathtt{B}[s_3, s_2]$ $\mathtt{C}[s_1, s_2]$	$\mathtt{A}[s_1, 2s_3]$ $\mathtt{B}[2s_3, s_2]$ $\mathtt{C}[s_1, s_2]$
floyd-warshall	
$\mathtt{path}\begin{bmatrix} \max(k + 1, n - k), \\ \max(k + 1, n - k) \end{bmatrix}$	$\mathtt{path}\begin{bmatrix} \max(k + 1, n - k), \\ \max(k + 1, n - k, 2s_2) \end{bmatrix}$