Script generated by TTT

Title: Seidl: Programmoptimierung (13.01.2016)

Date: Wed Jan 13 10:20:06 CET 2016

Duration: 90:50 min

Pages: 54

3.2 Instruction Level Parallelism

Modern processors do not execute one instruction after the other strictly sequentially.

Here, we consider two approaches:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining

VLIW

One instruction simultaneously executes up to k (e.g., 4:-) elementary Instructions.

Pipelining

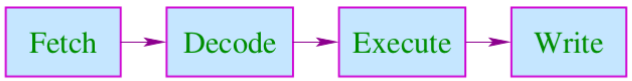
Instruction execution may overlap.

Example

$$w = (R_1 = R_2 + R_3 | D = D_1 * D_2 | R_3 = M[R_4])$$

Caveat

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers $\Longrightarrow$ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:



- During Execute and Write different internal registers/busses/alus may be used.

VLIW

One instruction simultaneously executes up to k (e.g., 4:-) elementary Instructions.

Pipelining

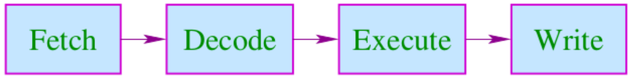
Instruction execution may overlap.

Example

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Caveat

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers $\Longrightarrow$ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:



- During Execute and Write different internal registers/busses/alus may be used.

We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

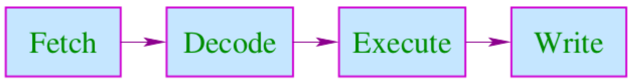
In the following, we ignore the phases Fetch und Decode.

Examples for Constraints

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.

Caveat

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers $\Longrightarrow$ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:



- During Execute and Write different internal registers/busses/alus may be used.

650

We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

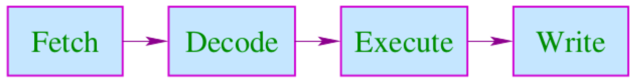
In the following, we ignore the phases Fetch und Decode.

Examples for Constraints

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.

Caveat

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers $\Longrightarrow$ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:



- During Execute and Write different internal registers/busses/alus may be used.

We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

In the following, we ignore the phases Fetch und Decode.

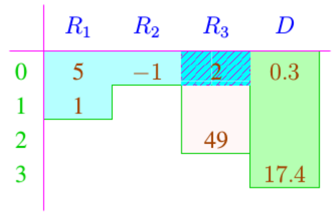
Examples for Constraints

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.

Example Timing:

Floating-point Operation	3
Load/Store	2
Integer Arithmetic	1

Timing Diagram:



R_3 is over-written, after the addition has fetched 2.

VLIW

One instruction simultaneously executes up to k (e.g., 4:-) elementary Instructions.

Pipelining

Instruction execution may overlap.

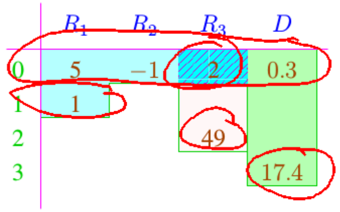
Example

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Example Timing:

Floating-point Operation	3
Load/Store	2
Integer Arithmetic	1

Timing Diagram:



R_3 is over-written, after the addition has fetched 2.

If a register is accessed simultaneously (here: R_3), a strategy of conflict solving is required ...

Conflicts

Read-Read: A register is simultaneously read.

$\Longrightarrow$ in general, unproblematic.

Read-Write: A register is simultaneously read and written.

Conflict Resolution:

- ... ruled out!
- Read is delayed (stalls), until write has terminated!
- Read before write returns old value!

Write-Write: A register is simultaneously written to.

$\Longrightarrow$ in general, unproblematic.

Conflict Resolutions:

- ... ruled out!
- ...

In Our Examples ...

- simultaneous read is permitted;
- simultaneous write/read and write/write is ruled out;
- no stalls are injected.

We first consider basic blocks only, i.e., linear sequences of assignments ...

If a register is accessed simultaneously (here: R_3), a strategy of conflict solving is required ...

Conflicts

Read-Read: A register is simultaneously read.

$\Longrightarrow$ in general, unproblematic.

Read-Write: A register is simultaneously read and written.

Conflict Resolution:

- ... ruled out!
- Read is delayed (stalls), until write has terminated!
- Read before write returns old value!

Write-Write: A register is simultaneously written to.

$\Longrightarrow$ in general, unproblematic.

Conflict Resolutions:

- ... ruled out!
- ...

In Our Examples ...

- simultaneous read is permitted;
- simultaneous write/read and write/write is ruled out;
- no stalls are injected.

We first consider basic blocks only, i.e., linear sequences of assignments ...

Idea: Data Dependence Graph

Vertices	Instructions
Edges	Dependencies

Example

$$
\begin{array}{ll}
(1) & x = x + 1; \\
(2) & y = M[A]; \\
(3) & t = z; \\
(4) & z = M[A + x]; \\
(5) & t = y + z;
\end{array}
$$

Possible Dependencies

Definition $\to$ Use // Reading Definitions

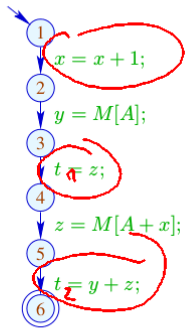
~~Use $\to$ Definition //~~ ???

~~Definition $\to$ Definition //~~ Reaching Definitions

Reaching Definitions:

Determine for each u which definitions may reach $\Longrightarrow$ can be determined by means of a system of constraints.

... in the Example:



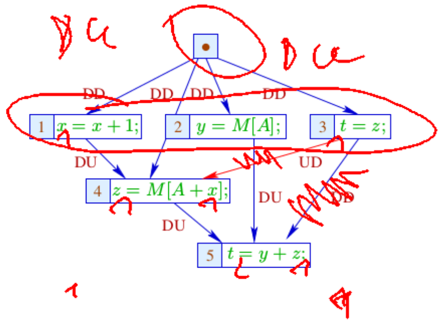
	$\mathcal{R}$
1	$\{\langle x,1\rangle, \langle y,1\rangle, \langle z,1\rangle, \langle t,1\rangle\}$
2	$\{\langle x,2\rangle, \langle y,1\rangle, \langle z,1\rangle, \langle t,1\rangle\}$
3	$\{\langle x,2\rangle, \langle y,3\rangle, \langle z,1\rangle, \langle t,1\rangle\}$
4	$\{\langle x,2\rangle, \langle y,3\rangle, \langle z,1\rangle, \langle t,4\rangle\}$
5	$\{\langle x,2\rangle, \langle y,3\rangle, \langle z,5\rangle, \langle t,4\rangle\}$
6	$\{\langle x,2\rangle, \langle y,3\rangle, \langle z,5\rangle, \langle t,6\rangle\}$

Let U_i, D_i denote the sets of variables which are used or defined at the edge outgoing from u_i. Then:

$$(u_1, u_2) \in DD \quad \text{if} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset$$
$$(u_1, u_2) \in DU \quad \text{if} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset$$

... in the Example:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$



The UD-edge $(3, 4)$ has been inserted to exclude that z is over-written before use.

In the next step, each instruction is annotated with its (required ressources, in particular, its) execution time.

Our goal is a maximally parallel correct sequence of words.

For that, we maintain the current system state:

$$\Sigma : \mathit{Vars} \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{expected delay until } x \text{ is available}$$

Initially:

$$\Sigma(x) = 0$$

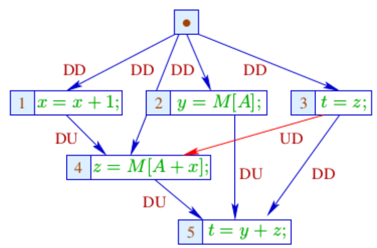
As an invariant, we guarantee on entry of the basic block, that all operations are terminated.

Let U_i, D_i denote the sets of variables which are used or defined at the edge outgoing from u_i. Then:

$$(u_1, u_2) \in DD \quad \text{if} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset$$
$$(u_1, u_2) \in DU \quad \text{if} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset$$

... in the Example:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$



The UD-edge $(3, 4)$ has been inserted to exclude that z is over-written before use.

In the next step, each instruction is annotated with its (required ressources, in particular, its) execution time.

Our goal is a maximally parallel correct sequence of words.

For that, we maintain the current system state:

$$\Sigma : Vars \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{expected delay until } x \text{ is available}$$

Initially:

$$\Sigma(x) = 0$$

As an invariant, we guarantee on entry of the basic block, that all operations are terminated.

Then the slots of the word sequence are successively filled:

- We start with the minimal nodes in the dependence graph.
- If we fail to fill all slots of a word, we insert $;$.
- After every inserted instruction, we re-compute Σ .

Caveat

→ The execution of two VLIWs can overlap !!!

→ Determining an optimal sequence, is NP-hard ...

Example:

Word width $k = 2$

Word		State			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In each cycle, the execution of a new word is triggered.

The state just records the number of cycles still to be waited for the result.

Example:

Word width $k = 2$

Word		State			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In each cycle, the execution of a new word is triggered.

The state just records the number of cycles still to be waited for the result.

Example: Word width $k = 2$

Word		State			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In each cycle, the execution of a new word is triggered.

The state just records the number of cycles still to be waited for the result.

Remark

- If instructions put constraints on future selection, we also record these in Σ.
- Overall, we still distinuish just finitely many system states.
- The computation of the effect of a VLIW onto Σ can be compiled into a finite automaton !!!
- This automaton, though, could be quite huge.
- The challenge of making choices still remains.
- Basic blocks usually are not very large

  $\Longrightarrow$ opportunities for parallelization are limited.

Extension 1: Acyclic Code

if $(x > 1)$ {
  $y = M[A];$
  $z = x - 1;$
} else {
  $y = M[A + 1];$
  $z = x - 1;$
}
$y = y + 1;$

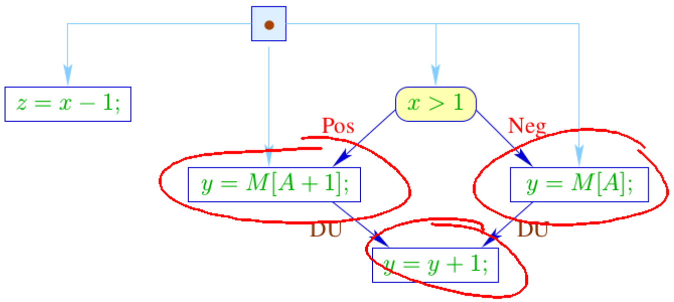
The dependence graph must be enriched with extra control-dependencies ...

Extension 1: Acyclic Code

if $(x > 1)$ {
  $y = M[A];$
  $z = x - 1;$
} else {
  $y = M[A + 1];$
  $z = x - 1;$
}
$y = y + 1;$

The dependence graph must be enriched with extra control-dependencies ...

663



The statement $z = x - 1;$ is executed with the same arguments in both branches and does not modify any of the remaining variables.

We could have moved it before the if anyway.

The following code could be generated:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	
	goto B	
$A:$	$y = M[A+1]$	
$B:$	$y = y + 1$	

At every jump target, we guarantee the invariant.

If we allow several (known) states on entry of a sub-block, we can generate code which complies with all of these.

... in the Example:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	goto B
$A:$	$y = M[A+1]$	
$B:$		
	$y = y + 1$	

If this parallelism is not yet sufficient, we could try to speculatively execute possibly useful tasks ...

For that, we require:

- an idea which alternative is executed more frequently;
- the wrong execution may not end in a catastrophy, i.e., run-time errors such as, e.g., division by 0;
- the wrong execution must allow roll-back (e.g., by delaying a commit) or may not have any observational effects ...

... in the Example:

	$z = x - 1$	$y = M[A]$	if $(x > 0)$ goto B
	$y = M[A+1]$		
$B:$			
	$y = y + 1$		

In the case $x \leq 0$ we have $y = M[A]$ executed in advance. This value, however, is overwritten in the next step ...

In general:

$x = e;$ has no observable effect in a branch if x is dead in this branch.

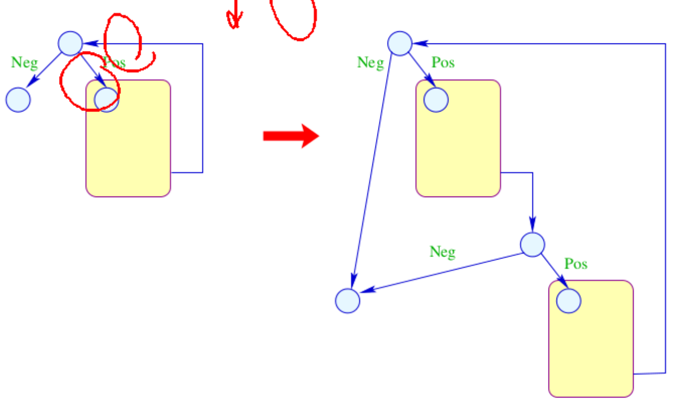
If this parallelism is not yet sufficient, we could try to speculatively execute possibly useful tasks ...

For that, we require:

- an idea which alternative is executed more frequently;
- the wrong execution may not end in a catastrophy, i.e., run-time errors such as, e.g., division by 0;
- the wrong execution must allow roll-back (e.g., by delaying a commit) or may not have any observational effects ...

Extension 2: Unrolling of Loops

We may unrole important, i.e., inner loops several times:



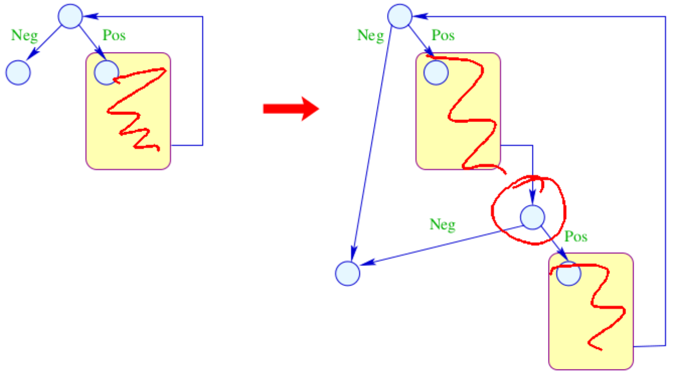
If this parallelism is not yet sufficient, we could try to speculatively execute possibly useful tasks ...

For that, we require:

- an idea which alternative is executed more frequently;
- the wrong execution may not end in a catastrophy, i.e., run-time errors such as, e.g., division by 0;
- the wrong execution must allow roll-back (e.g., by delaying a commit) or may not have any observational effects ...

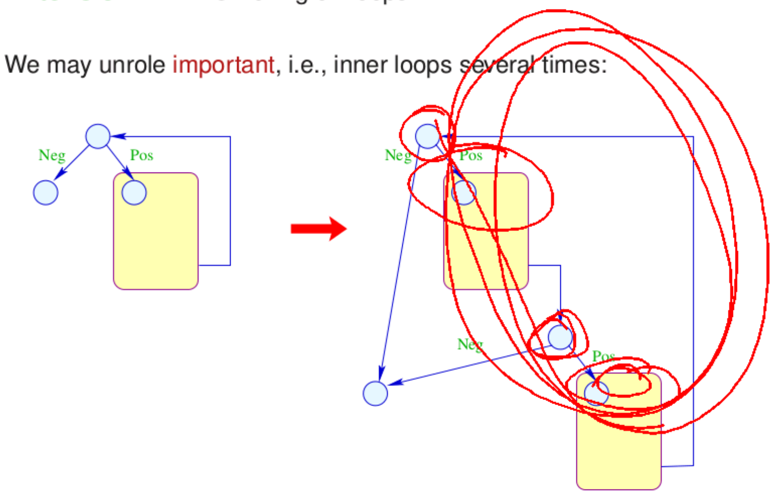
Extension 2: Unrolling of Loops

We may unrole important, i.e., inner loops several times:



Extension 2: Unrolling of Loops

We may unrole important, i.e., inner loops several times:



Now it is clear which side of tests to prefer:

the side which stays within the unroled body of the loop.

Caveat

- The different instances of the body are translated relative to possibly different initial states.
- The code behind the loop must be correct relative to the exit state corresponding to every jump out of the loop!

Now it is clear which side of tests to prefer:

the side which stays within the unroled body of the loop.

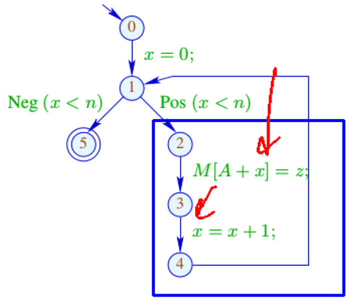
Caveat

- The different instances of the body are translated relative to possibly different initial states.
- The code behind the loop must be correct relative to the exit state corresponding to every jump out of the loop!

Example

for $(x = 0; x < n; x{+}{+})$

$M[A + x] = z;$



Duplication of the body yields:

for $(x = 0; x < n; x{+}{+})$ {

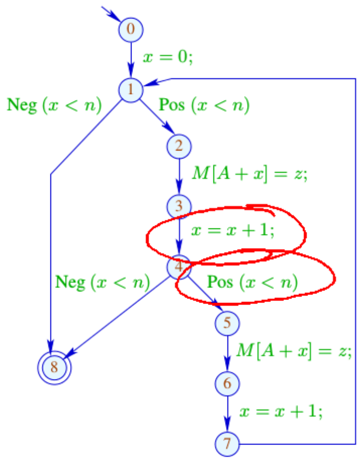
$M[A + x] = z;$

$x = x + 1;$

if $(!(x < n))$ break;

$M[A + x] = z;$

}



It would be better to remove $x = x + 1;$ together with the test in the middle — since these serialize execution of the copies !!

This is possible if $x + 1$ is substituted for x in the second copy, the condition is transformed and compensation code is added:

for $(x = 0; x + 1 < n; x = x + 2)$ {

$M[A + x] = z;$

$M[A + x + 1] = z;$

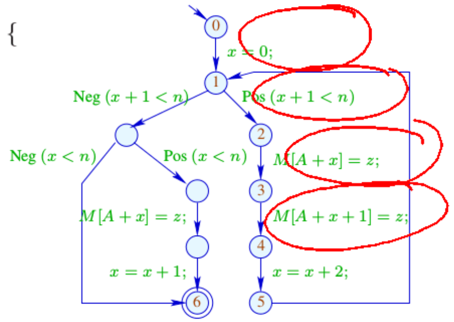
}

if $(x < n)$ {

$M[A + x] = z;$

$x = x + 1;$

}



It would be better to remove $x = x + 1;$ together with the test in the middle — since these serialize execution of the copies !!

This is possible if $x + 1$ is substituted for x in the second copy, the condition is transformed and compensation code is added:

for $(x = 0; x + 1 < n; x = x + 2)$ {

$M[A + x] = z;$

$M[A + x + 1] = z;$

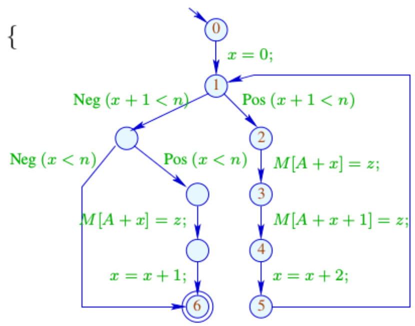
}

if $(x < n)$ {

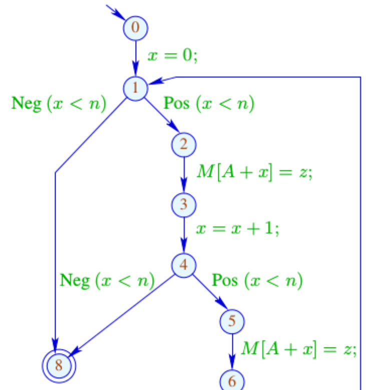
$M[A + x] = z;$

$x = x + 1;$

}



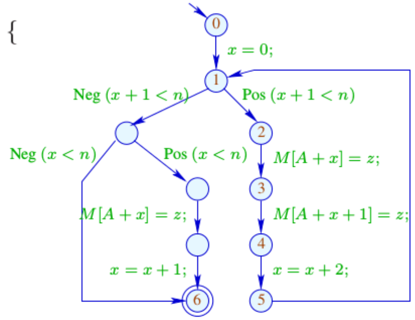
for $(x = 0; x < n; x{+}{+})$ {
$M[A + x] = z;$
$x = x + 1;$
if $(!(x < n))$ break;
$M[A + x] = z;$
}



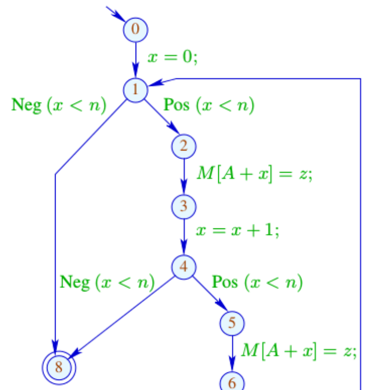
It would be better to remove $x = x + 1;$ together with the test in the middle — since these serialize execution of the copies !!

This is possible if $x + 1$ is substituted for x in the second copy, the condition is transformed and compensation code is added:

for $(x = 0; x + 1 < n; x = x + 2)$ {
$M[A + x] = z;$
$M[A + x + 1] = z;$
}
if $(x < n)$ {
$M[A + x] = z;$
$x = x + 1;$
}



for $(x = 0; x < n; x{+}{+})$ {
$M[A + x] = z;$
$x = x + 1;$
if $(!(x < n))$ break;
$M[A + x] = z;$
}

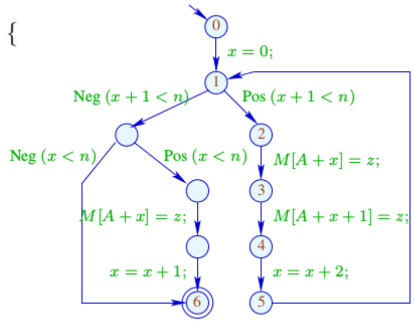


672

It would be better to remove $x = x + 1;$ together with the test in the middle — since these serialize execution of the copies !!

This is possible if $x + 1$ is substituted for x in the second copy, the condition is transformed and compensation code is added:

for $(x = 0; x + 1 < n; x = x + 2)$ {
$M[A + x] = z;$
$M[A + x + 1] = z;$
}
if $(x < n)$ {
$M[A + x] = z;$
$x = x + 1;$
}



673

Discussion

- Elimination of the intermediate test together with the the fusion of all increments at the end reveals that the different loop iterations are in fact independent.
- Nonetheless, we do not gain much since we only allow one store per word.
- If right-hand sides, however, are more complex, we can interleave their evaluation with the stores.

Extension 3

Sometimes, one loop alone does not provide enough opportunities for parallelization.

... but perhaps two successively in a row ...

Example

for $(x = 0; x < n; x{+}{+})$ {
$R = B[x];$
$S = C[x];$
$T_1 = R + S;$
$A[x] = T_1;$
}

for $(x = 0; x < n; x{+}{+})$ {
$R = B[x];$
$S = C[x];$
$T_2 = R - S;$
$C[x] = T_2;$
}

Extension 3

Sometimes, one loop alone does not provide enough opportunities for parallelization.

... but perhaps two successively in a row ...

Example

for $(x = 0; x < n; x{+}{+})$ {
$R = B[x];$
$S = C[x];$
$T_1 = R + S;$
$A[x] = T_1;$
}

for $(x = 0; x < n; x{+}{+})$ {
$R = B[x];$
$S = C[x];$
$T_2 = R - S;$
$C[x] = T_2;$
}

675