

Script generated by TTT

Title: Seidl: Programoptimierung (19.12.2012)

Date: Wed Dec 19 08:30:51 CET 2012

Duration: 90:29 min

Pages: 40

Remark:

- Intersection graphs for tree fragments are also known as **cordal graphs** ...
- A cordal graph is an undirected graph where every cycle with more than three nodes contains a **cord** :-)
- Cordal graphs are another sub-class of **perfect graphs** :-))
- Cheap register allocation comes at a price:
when transforming into **SSA** form, we have introduced parallel register-register moves :-((

636

Problem

The parallel register assignment:

$$\psi_1 = R_1 = R_2 \mid R_2 = R_1$$

is meant to exchange the registers R_1 and R_2 :-)

There are at least two ways of implementing this exchange ...

637

Problem

The parallel register assignment:

$$\psi_1 = R_1 = R_2 \mid R_2 = R_1$$

is meant to exchange the registers R_1 and R_2 :-)

There are at least two ways of implementing this exchange ...

(1) Using an auxiliary register:

$$R = R_1;$$

$$R_1 = R_2;$$

$$R_2 = R;$$

638

(2) XOR:

$$\begin{aligned}R_1 &= R_1 \oplus R_2; \\R_2 &= R_1 \oplus R_2; \\R_1 &= R_1 \oplus R_2;\end{aligned}$$

639

(2) XOR:

$$\begin{aligned}R_1 &= R_1 \oplus R_2; \\R_2 &= R_1 \oplus R_2; \\R_1 &= R_1 \oplus R_2;\end{aligned}$$

(3) Harder Support

639

(2) XOR:

$$\begin{aligned}R_1 &= R_1 \oplus R_2; \\R_2 &= R_1 \oplus R_2; \\R_1 &= R_1 \oplus R_2;\end{aligned}$$

But what about cyclic shifts such as:

$$\psi_k = R_1 = R_2 \mid \dots \mid R_{k-1} = R_k \mid R_k = R_1$$

for $k > 2$??

640

(2) XOR:

$$\begin{aligned}R_1 &= R_1 \oplus R_2; \\R_2 &= R_1 \oplus R_2; \\R_1 &= R_1 \oplus R_2;\end{aligned}$$

But what about cyclic shifts such as:

$$\psi_k = R_1 = R_2 \mid \dots \mid R_{k-1} = R_k \mid R_k = R_1$$

for $k > 2$??

Then at most $k - 1$ swaps of two registers are needed:

$$\begin{aligned}\psi_k &= R_1 \leftrightarrow R_2; \\&R_2 \leftrightarrow R_3; \\&\dots \\&R_{k-1} \leftrightarrow R_k;\end{aligned}$$

641

(2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

But what about cyclic shifts such as:

$$\psi_k = R_1 = R_2 \mid \dots \mid R_{k-1} = R_k \mid R_k = R_1$$

for $k > 2$??

Then at most $k - 1$ swaps of two registers are needed:

$$\begin{aligned} \psi_k &= R_1 \leftrightarrow R_2; \\ &R_2 \leftrightarrow R_3; \\ &\dots \\ &R_{k-1} \leftrightarrow R_k; \end{aligned}$$

641

Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts :-)
- Any permutation of n registers with r shifts can be realized by $n - r$ swaps ...

642

Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts :-)
- Any permutation of n registers with r shifts can be realized by $n - r$ swaps ...

Example

$$\psi = R_1 = R_2 \mid R_2 = R_5 \mid R_3 = R_4 \mid R_4 = R_3 \mid R_5 = R_1$$

consists of the cycles (R_1, R_2, R_5) and (R_3, R_4) . Therefore:

$$\begin{aligned} \psi &= R_1 \leftrightarrow R_2; \\ &R_2 \leftrightarrow R_5; \\ &R_3 \leftrightarrow R_4; \end{aligned}$$

643

The general case:

- Every register receives its value at most once.
- The assignment therefore can be decomposed into a permutation together with tree-like assignments (directed towards the leaves) ...

Example

$$\psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3$$

The parallel assignment realizes the linear register moves for R_1, R_2 and R_4 together with the cyclic shift for R_3 and R_5 :

$$\begin{aligned} \psi &= R_1 = R_2; \\ &R_2 = R_4; \\ &R_3 \leftrightarrow R_5; \end{aligned}$$

644

The general case:

- Every register receives its value at most once.
- The assignment therefore can be decomposed into a permutation together with tree-like assignments (directed towards the leaves) ...

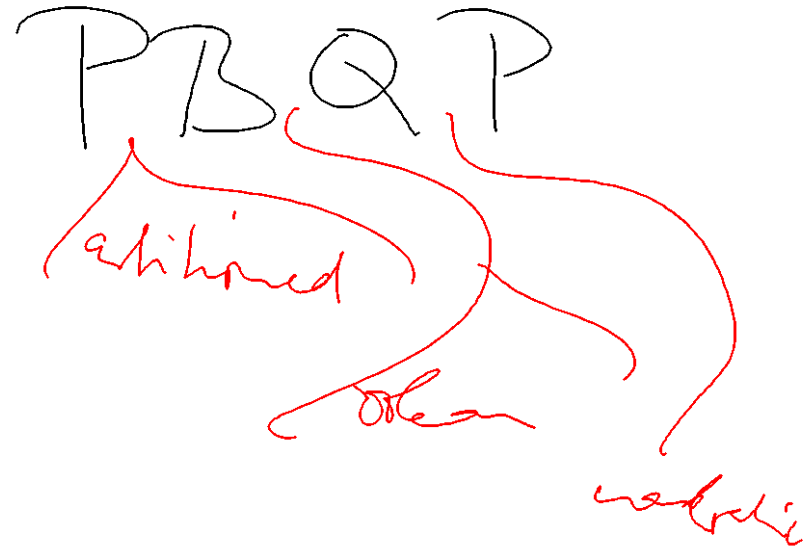
Example

$$\psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3$$

The parallel assignment realizes the linear register moves for R_1 , R_2 and R_4 together with the cyclic shift for R_3 and R_5 :

$$\begin{aligned} \psi &= R_1 = R_2; \\ &R_2 = R_4; \\ &R_3 \leftrightarrow R_5; \end{aligned}$$

644



Interprocedural Register Allocation:

- For every local variable, there is an entry in the stack frame.
- Before calling a function, the locals must be saved into the stack frame and be restored after the call.
- Sometimes there is hardware support :-)
Then the call is transparent for all registers.
- If it is our responsibility to save and restore, we may ...
 - save only registers which are over-written :-)
 - restore overwritten registers only.
- Alternatively, we save only registers which are still live after the call — and then possibly into different registers \implies reduction of life ranges :-)

645

Interprocedural Register Allocation:

- For every local variable, there is an entry in the stack frame.
- Before calling a function, the locals must be saved into the stack frame and be restored after the call.
- Sometimes there is hardware support :-)
Then the call is transparent for all registers.
- If it is our responsibility to save and restore, we may ...
 - save only registers which are over-written :-)
 - restore overwritten registers only.
- Alternatively, we save only registers which are still live after the call — and then possibly into different registers \implies reduction of life ranges :-)

645

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

646

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

646

VLIW:

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

Pipelining:

Instruction execution may overlap.

Example:

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

647


VLIW:

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

Pipelining:

Instruction execution may overlap.

Example:

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


647

Warning:

- Instructions occupy hardware resources.
- Instructions may access the same busses/registers \implies 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.

648

VLIW:

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

Pipelining:

Instruction execution may overlap.

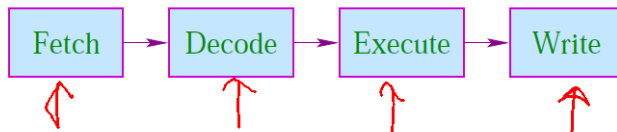
Example:

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

647

Warning:

- Instructions occupy hardware resources.
- Instructions may access the same busses/registers \implies 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.

648

Warning:

- Instructions occupy hardware resources.
- Instructions may access the same busses/registers \implies 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.

648

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** :-)

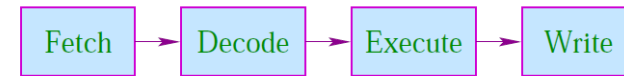
Examples for Constraints:

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

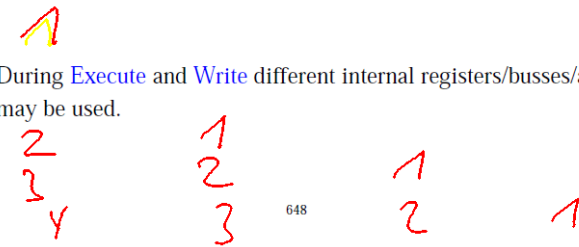
649

Warning:

- Instructions occupy hardware resources.
- Instructions may access the same busses/registers \implies 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.



648

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:

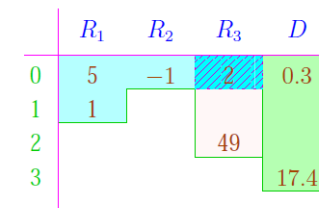
- (1) at most one load/store per word;
- (2) at most one jump;
- (3) at most one write into the same register.

649

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 :-)

650

VLIW:

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

Pipelining:

Instruction execution may overlap.

Example:

1 3 2

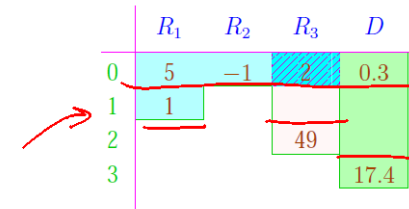
$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

647

Example Timing:

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

Timing Diagram:



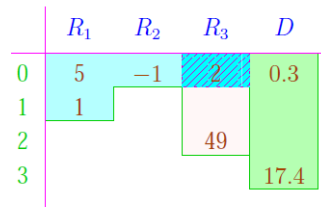
R_3 is over-written, after the addition has fetched 2 :-)

650

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 :-)

650

VLIW:

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

Pipelining:

Instruction execution may overlap.

Example:

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

647

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

Conflicts:

Read-Read: A register is simultaneously read.

⇒ 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!

651

Write-Write: A register is simultaneously written to.

⇒ 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

...

652

Idea: Data Dependence Graph

Vertices	Instructions
Edges	Dependencies

Example:

- (1) $x = x + 1;$
- (2) $y = M[A];$
- (3) $t = z;$
- (4) $z = M[A + x];$
- (5) $t = y + z;$

653

Idea: Data Dependence Graph

Vertices	Instructions
Edges	Dependencies

Example:

- (1) $x = x + 1;$
- (2) $y = M[A];$
- (3) $t = z;$
- (4) $z = M[A + x];$
- (5) $t = y + z;$

653

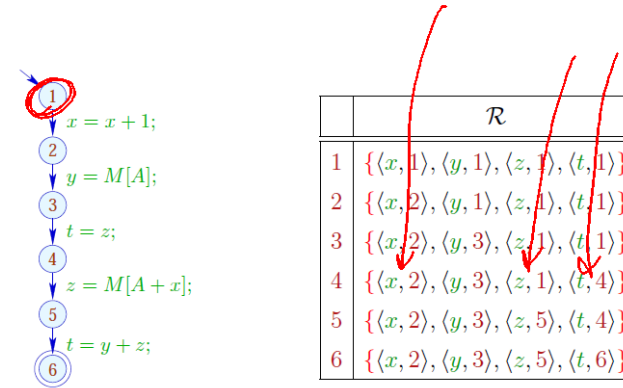
Possible Dependencies:

- Definition → Use // Reaching Definitions
- Use → Definition // ???
- Definition → Definition // Reaching Definitions

Reaching Definitions:

Determine for each u which definitions may reach \implies can be determined by means of a system of constraints \therefore

... in the Example:

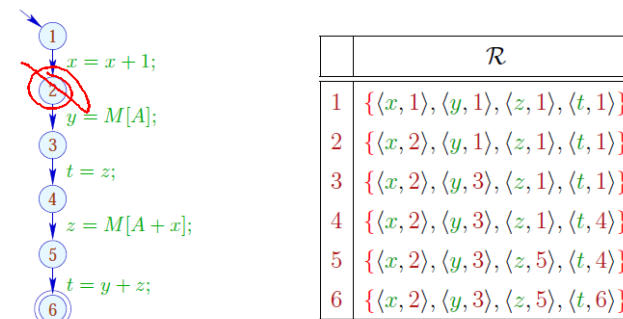
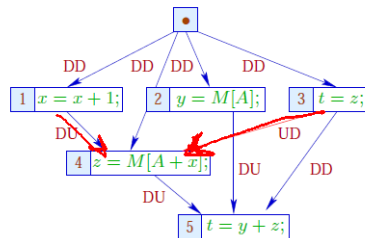


Let U_i, D_i denote the sets of variables which are used or defined at the edge outgoing from u_i . Then:

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

... 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	$t = y + z;$	$\{t\}$	$\{y, z\}$



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	$t = y + z;$	$\{t\}$	$\{y, z\}$

