COMPILER TECHNIQUES TO EXTRACT PARALLELISM

Slides by: Pedro Tomás

Extracting parallelism from the compiler:

- Local techniques for instruction scheduling
- Global techniques for instruction scheduling
- Hardware support for global techniques
- Dealing with exceptions
- Dealing with Load/Stores
From the previous lesson

Speedup from instruction scheduling

Cont: L.D  F0,0(R2)  

Stall
ADD.D  F2,F0,F1
Stall
S.D  0(R2),F2
DSUBI  R2,R2,#8
BNE  R2,R1,Cont
Stall
9 cycles per iteration

Cont: L.D  F0,0(R2)  

Stall
ADD.D  F2,F0,F1
DSUBI  R2,R2,#8
BNE  R2,R1,Cont
6 cycles per iteration

\[
\text{Speedup} = \frac{9}{6} = 1.5
\]

Cont: L.D  F0,0(R2)
L.D  F2,-8(R2)
L.D  F3,-16(R2)
L.D  F4,-24(R2)
ADD.D  F0,F0,F1
ADD.D  F2,F2,F1
ADD.D  F3,F3,F1
ADD.D  F4,F4,F1
SD  0(R2),F0
SD  -8(R2),F2
SD  -16(R2),F3
SUB  R2,R2,#32
BNE  R2,R1,Cont
S.D  8(R2),F4
3.5 cycles per iteration

\[
\text{Speedup} = \frac{9}{3.5} = 2.57
\]
From the previous lesson

MIPS extension to VLIW

- MIPS-VLIW extension includes 5 different execution unit, which translates into packets of 5 instructions, maximum

<table>
<thead>
<tr>
<th>#</th>
<th>Memory 1</th>
<th>Memory 2</th>
<th>FP 1</th>
<th>FP 2</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0,0(R0)</td>
<td>L.D F2,-8(R0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L.D F3,-16(R0)</td>
<td>L.D F4,-24(R0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L.D F5,-32(R0)</td>
<td>L.D F6,-40(R0)</td>
<td>ADD.D F0,F0,F1</td>
<td>ADD.D F2,F2,F1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L.D F7,-48(R0)</td>
<td>L.D F8,-56(R0)</td>
<td>ADD.D F3,F3,F1</td>
<td>ADD.D F4,F4,F1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L.D F9,-64(R0)</td>
<td>L.D F10,-72(R0)</td>
<td>ADD.D F5,F5,F1</td>
<td>ADD.D F6,F6,F1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>ADD.D F7,F7,F1</td>
<td>ADD.D F8,F8,F1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S.D 0(R0),F0</td>
<td>S.D -8(R0),F2</td>
<td>ADD.D F9,F9,F1</td>
<td>ADD.D F10,F10,F1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S.D -16(R0),F3</td>
<td>S.D -24(R0),F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S.D -32(R0),F5</td>
<td>S.D -40(R0),F6</td>
<td></td>
<td></td>
<td>DSUBI R0,R0,-80</td>
</tr>
<tr>
<td>10</td>
<td>S.D 32(R0),F5</td>
<td>S.D 24(R0),F6</td>
<td></td>
<td></td>
<td>BNE R0,R1,Cont</td>
</tr>
<tr>
<td>11</td>
<td>S.D 16(R0),F5</td>
<td>S.D 8(R0),F6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Speedup vs original case} = \frac{9}{1.1} = 8.18
\]
\[
\text{Speedup vs reordered} = \frac{3.5}{1.1} = 3.18
\]
To extract parallelism from loops, iterations must be independent

**Example A:**

```c
for (i=0; i<=100; i++)
```

**LOOP IS PARALLELIZABLE:**
Operation S1 is independent

The loop can be unrolled ... and the operations can be re-ordered or executed in parallel

```c
for (i=0; i<=100; i+=N) {
    ...
}
```
To extract parallelism from loops, iterations must be independent

Example A:
```c
for (i=0; i<=100; i++)
```

Example B:
```c
for (i=1; i<=100; i++)
```

Example C:
```c
for (i=1; i<100; i++)
    A[i+1] = B[i] + K; /* S1 */
    B[i+1] = A[i] + m; /* S2 */
```

**LOOP IS PARALLELIZABLE:**
Operation S1 is independent

**CLOSED LOOP ➔ LOOP IS NOT PARALLELIZABLE:**
Operation S1 depends on itself

**CLOSED LOOP ➔ NOT PARALLELIZABLE:**
Operation S1 depends on S2 and S2 depends on S1
Parallelizing loops:

Resolving dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

  - Example D:
    ```c
    for (i=0; i < 100; i++) {
      A[i+1] = B[i] + k;  /* S1 */
      C[i] = A[i] + m;    /* S2 */
    }
    ```

  - **Loop Unrolling**

<table>
<thead>
<tr>
<th>Iteration $i$</th>
<th>$A[i+1]$ = $B[i] + k$; /* S1 (i) */</th>
<th>$C[i]$ = $A[i] + m$; /* S2 (i) */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i+1$</td>
<td>$A[i+2]$ = $B[i+1] + k$; /* S1 (i+1) */</td>
<td>$C[i+1]$ = $A[i+1] + m$; /* S2 (i+1) */</td>
</tr>
</tbody>
</table>

  **Read-After-Write (RAW) Dependency** forces iterations to be executed sequentially.
Parallelizing loops:

Resolving dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

  Example D:

  ```
  for (i=0; i < 100; i++) {
      A[i+1] = B[i] + k; /* S1 */
      C[i] = A[i] + m; /* S2 */
  }
  c[0] = A[0] + m;
  for (i=0; i < 99; i++) {
      A[i+1] = B[i] + k; /* S1 */
      C[i+1] = A[i+1] + m; /* S2 */
  }
  A[100] = B[99] + k;
  ```

  OPEN LOOP ➔ DEPENDENCY THAT CAN BE RESOLVED:
  Instructions can be re-written so that loop is parallelizable

  LOOP IS PARALLELIZABLE:
  There are only intra-iteration dependencies
Parallelizing loops:

Name dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

- Example E:

```c
for (i=0; i < 100; i++) {
    A[i] = B[i] + k;  /* S1 */
    C[i] = A[i+1] + m;  /* S2 */
}
```

A Write-After-Read (WAR) Dependency forces iterations to be executed sequentially.

A WAR dependency is also known as an anti-dependency.
Parallelizing loops:

Name dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

  - Example E:

    ```c
    for (i=0; i < 100; i++) {
        A[i] = B[i] + k;  /* S1 */
        C[i] = A[i+1] + m; /* S2 */
    }
    ```

  - **TRANSFORMATION**

    ```c
    for (i=0; i < 100; i++) {
        X[i] = B[i] + k;  /* S1 */
        C[i] = A[i+1] + m; /* S2 */
    }
    A = X;
    ```

- **NAME DEPENDENCY:**
  Name dependency are not true dependencies since they can be resolved by simply renaming the variable

- **LOOP IS PARALLELIZABLE:**
  There are only intra dependencies
Parallelizing loops:

Name dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

- Example F:

```c
for (i=0; i < 100; i++) {
    A[i+1] = B[i] + k; /* S1 */
    A[i] = C[i] + m; /* S2 */
}
```

```
A[i+1] = B[i] + k; /* S1 (i) */
A[i] = C[i] + m; /* S2 (i) */
```

```
A[i+2] = B[i+1] + k; /* S1 (i+1) */
A[i+1] = C[i+1] + m; /* S2 (i+1) */
```

```
A[i+3] = B[i+2] + k; /* S1 (i+2) */
A[i+2] = C[i+2] + m; /* S2 (i+2) */
```

Write-After-Write (WAW) Dependency forces iterations to be executed sequentially.

A WAW dependency is also known as an output dependency.
Parallelizing loops:

Name dependencies

- To extract parallelism from loops, iteration $i$ must be independent from iteration $j$

**Example F:**

```c
for (i=0; i < 100; i++) {
    A[i+1] = B[i] + k; /* S1 */
    A[i]   = C[i] + m; /* S2 */
}
```  

**NAME DEPENDENCY:** Name dependency are not true dependencies since they can be resolved by simply renaming the variable

**TRANSFORMATION**

```c
for (i=0; i < 100; i++) {
    A[i+1] = B[i] + k; /* S1 */
    X[i]   = C[i] + m; /* S2 */
}
```

**LOOP IS PARALLELIZABLE:** There are only intra dependencies

```c
A = X;
```
Parallelizing loops:

Checking for dependencies

- When the indexes are determined by an affine transforms, \((a \times i + b : a, b \text{ are constants }; c \times i + d : c, d \text{ are constants})\)
  \[
  A[a*i+b] = f(...);
  \]
  \[
  ... = g( A[c*i+d] , ... );
  \]

Define a dependency \textit{iff (if and only if)}:
\[
 a \times j + b = c \times k + d , \ \forall j,k : \text{ranges are in the loop limits}
\]

- The general solution for this problem is NP-complete
  - Some simple algorithms exist that check for dependencies, e.g., the greatest common divider (GCD) test
    - Checks if the GCD\((a,c)\) can divide \((d-b)\)
    - On failure indicates that no dependency exists; success does not guarantee that a dependency exists
  - For simple (specific) cases simple and precise algorithms exist
Parallelizing loops: Advanced example

```c
for (i=1; i<100; i++) {
    A[i] = B[i] + C[i+1] + D[i];  /* S1 */
    C[i-1] = 3 * sqrt(A[i]);       /* S2 */
    B[i+1] = D[i] + C[i]*2;        /* S3 */
    A[i] = D[i] + F[i];            /* S4 */
}
```

SAME CYCLE DEPENDENCIES

- Intra-cycle name dependency
- Intra-cycle true dependency

DIFERENT CYCLE DEPENDENCIES

- Inter-cycle name dependency
- Inter-cycle true dependency
Parallelizing loops: Advanced example

for (i=1; i<100; i++) {
    A[i]   = B[i] + C[i+1] + D[i];       /* S1 */
    C[i-1] = 3 * sqrt(A[i]);            /* S2 */
    B[i+1] = D[i] + C[i]*2;             /* S3 */
    A[i]   = D[i] + F[i];               /* S4 */
}

Anti-dependency that can be solved by renaming the output variable
Parallelizing loops:

Advanced example

```c
for (i=1; i<100; i++) {
    A[i] = B[i] + C[i+1] + D[i];  /* S1 */
    X[i-1] = 3 * sqrt(A[i]);       /* S2 */
    B[i+1] = D[i] + C[i]*2;       /* S3 */
    A[i] = D[i] + F[i];           /* S4 */
}
C=X;
```

- **True dependency that must be solved by instruction shifting and index changing**

```c
X[0] = 3*sqrt(A[1]);
```

### Prologue

```c
X[0] = 3*sqrt(A[1]);
```

- **S1**
- **S2**
- **S3**
- **S4**

### Epilogue

```c
B[100] = D[99] + C[99]*2
C=X;
```
Parallelizing loops:

Advanced example

for (i=1; i<100; i++) {
    A[i] = B[i] + C[i+1] + D[i];  /* S1 */
    X[1-1] = 3 * sqrt(A[i]);      /* S2 */
    B[i+1] = D[i] + C[i]*2;       /* S3 */
    A[i] = D[i] + F[i];           /* S4 */
}
C=X;

X[0] = 3*sqrt(aux);

for (i=1; i<99; i++) {
    B[i+1] = D[i] + C[i]*2;       /* S3 */
    aux = B[i+1] + C[i+2] + D[i+1]; /* S1 */
    X[i] = 3 * sqrt(aux);         /* S2 */
    A[i+1] = D[i+1] + F[i+1];     /* S4 */
}

B[100] = D[99] + C[99]*2
C=X;

To enable intra-cycle parallelization, resolve also the RAW dependency on A

Prologue

Epilogue
Parallelizing loops:
Advanced example

Instruction retiming can now take place to reduce the number of conflicts in the pipeline!

```
X[0] = 3*sqrt(aux);

for (i=1; i<99; i++) {
    B[i+1] = D[i] + C[i]*2;        /* S3 */
aux = B[i+1] + C[i+2] + D[i+1];/* S1 */
    X[i] = 3 * sqrt(aux);           /* S2 */
    A[i+1] = D[i+1] + F[i+1];       /* S4 */
}
B[100] = D[99] + C[99]*2
C=X;
```

Exercise: write the code in assembly and perform retiming (re-ordering) in order to reduce the number of stalls.
Parallelizing loops:

Software pipelining

- If there is parallelism at iteration level, it can be explored by executing instructions from different iterations of the same cycle.

- Software pipelining

In practice it is equivalent to:
- Loop unrolling
- Instruction retiming
- Loop rolling
Parallelizing loops:

Example of Software pipelining

// C Code
for (i=99;i>=0;i--)

; Assembly Code
Cont:  L.D    F0,0(R2)
       ADD.D  F2,F0,F1
       S.D    0(R2),F2
       DSUBI  R2,R2,#8
       BNE    R2,R1,Cont

; Loop unrolling
Cont:  L.D    F0,0(R2)
       ADD.D  F2,F0,F1
       S.D    0(R2),F2
       L.D    F0,-8(R2)
       ADD.D  F2,F0,F1
       S.D    -8(R2),F2
       L.D    F0,-16(R2)
       ADD.D  F2,F0,F1
       S.D    -16(R2),F2

...
Parallelizing loops:

Example of Software pipelining

Take one instruction from each iteration to form the loop core

Prologue

<table>
<thead>
<tr>
<th>L.D</th>
<th>F0,0(R2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</td>
<td>F2,F0,F1</td>
</tr>
<tr>
<td>L.D</td>
<td>F0,-8(R2)</td>
</tr>
</tbody>
</table>

Cont:

<table>
<thead>
<tr>
<th>S.D</th>
<th>0(R2),F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</td>
<td>F2,F0,F1</td>
</tr>
<tr>
<td>L.D</td>
<td>F0,-16(R2)</td>
</tr>
</tbody>
</table>

Loop Core

<table>
<thead>
<tr>
<th>DSUBI</th>
<th>R2,R2,#8</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>R2,R1,Cont</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

Epilogue

<table>
<thead>
<tr>
<th>S.D</th>
<th>0(R2),F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</td>
<td>F2,F0,F1</td>
</tr>
<tr>
<td>S.D</td>
<td>-8(R2),F2</td>
</tr>
</tbody>
</table>

; Loop unrolling

| Cont: L.D  | F0,0(R2) |
| ADD.D      | F2,F0,F1 |
| S.D        | 0(R2),F2 |
| L.D        | F0,-8(R2) |
| ADD.D      | F2,F0,F1 |
| S.D        | -8(R2),F2 |
| L.D        | F0,-16(R2) |
| ADD.D      | F2,F0,F1 |
| S.D        | -16(R2),F2 |

...
Parallelizing loops:

Example of Software pipelining

To guarantee correctness start executing the first and second loops on the prologue.

<table>
<thead>
<tr>
<th>Prologue</th>
<th>Cont:</th>
<th>Loop unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0 (R2)</td>
<td>L.D F0,0 (R2)</td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F1</td>
<td>ADD.D F2,F0,F1</td>
<td></td>
</tr>
<tr>
<td>L.D F0,-8 (R2)</td>
<td>S.D 0 (R2),F2</td>
<td></td>
</tr>
<tr>
<td>Cont: S.D 0 (R2),F2</td>
<td>S.D 0 (R2),F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F1</td>
<td>ADD.D F2,F0,F1</td>
<td></td>
</tr>
<tr>
<td>L.D F0,-16 (R2)</td>
<td>S.D -8 (R2),F2</td>
<td></td>
</tr>
<tr>
<td>DSUBI R2,R2,#8</td>
<td>L.D F0,-16 (R2)</td>
<td></td>
</tr>
<tr>
<td>BNE R2,R1,Cont</td>
<td>ADD.D F2,F0,F1</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>S.D -16 (R2),F2</td>
<td></td>
</tr>
<tr>
<td>Epilogue</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>S.D 0 (R2),F2</td>
<td>DSUBI R2,R2,#N</td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F1</td>
<td>BNE R2,R1,Cont</td>
<td></td>
</tr>
<tr>
<td>S.D -8 (R2),F2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallelizing loops: Example of Software pipelining

<table>
<thead>
<tr>
<th>Prologue</th>
<th>Loop Core</th>
<th>Epilogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R2)</td>
<td>S.D 0(R2),F2</td>
<td>S.D 0(R2),F2</td>
</tr>
<tr>
<td>ADD.D F2,F0,F1</td>
<td>ADD.D F2,F0,F1</td>
<td>ADD.D F2,F2,F1</td>
</tr>
<tr>
<td>L.D F0,-8(R2)</td>
<td>S.D -8(R2),F2</td>
<td>S.D -16(R2),F2</td>
</tr>
<tr>
<td>DSUBI R2,R2,#8</td>
<td>L.D F0,-8(R2)</td>
<td>L.D F0,-16(R2)</td>
</tr>
<tr>
<td>BNE R2,R1,Cont</td>
<td>ADD.D F2,F0,F1</td>
<td>ADD.D F2,F2,F1</td>
</tr>
<tr>
<td>NOP</td>
<td>S.D -8(R2),F2</td>
<td>S.D -16(R2),F2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSUBI R2,R2,#N</td>
</tr>
</tbody>
</table>

Cont: L.D F0,0(R2)  ADD.D F2,F0,F1  S.D 0(R2),F2

; Loop unrolling

On exit, finish executing the remaining iterations

BNE R2,R1,Cont
Parallelizing loops:

Example of Software pipelining

- None of the RAW dependencies in the loop generate hazards
- Anti-dependencies (WAR) do not generate hazards
- The loop core takes 5 cycles to execute
- Speed regarding simple intra-iteration instruction re-organization:
  \[ Speedup = \frac{6}{5} = 1.2 \]

20% faster
Parallelizing loops: Software pipelining vs Loop unrolling

**Loop unrolling:**
- Requires more memory to store the instructions
- Reduces the overhead of the loop taken by control instructions
- Does not guarantee maximum parallelism over time

**Software pipelining:**
- Requires less memory to store the instructions
- Reduces the amount of hazards and therefore maximizes the efficiency of the code
- Increases parallelism during the loop core
- Can have large prologues/epilogues

Best solutions are achieved by using the two methods together
Global scheduling techniques

- Efficient scheduling in loops with control instructions require global techniques:
  - Instruction shifting crossing control instructions
  - Objective: get the shortest sequence to the critical path, which corresponds to the longest sequence of dependent instructions

- Global scheduling techniques supported only in compilers are complex:
  - Use speculative processing which increases the amount of computation
  - Requires knowledge of the most frequent paths in the program execution to reduce the execution time

- Specific hardware help the compiler to solve these issues
Global scheduling techniques

LD F1,0 (R1)
LD F2,0 (R2)
DSUB F3,F1,F2
SD 0 (R1),F3
BNE R2,R0,ELSE

IF:
DADD F3,F6,F0
ADD R3,R4,R1
...
J ENDF

ELSE:
...

ENDIF:
...

Do something

Check condition

IF code

Else code

Join
Global scheduling techniques

Perform speculative execution:

- Do something
- Check condition
- IF code
- Undo if Execute Else
- Join

```
LD     F1, 0 (R1)
LD     F2, 0 (R2)
DSUB   F3, F1, F2
SD     0 (R1), F3
BNE    R2, R0, ELSE
IF:    DADD   F3, F6, F0
       ADD    R3, R4, R1
       ...
J      ENDIF
ELSE:  ...
       ...
ENDIF: ...
```
Global scheduling techniques

Perform speculative execution:

- Move the most frequent path to the normal flow
- The least frequent path must undo all speculative changes

Compiler:

- Must record all changes to the original program and undo all wrong speculations
- Cannot move instructions that might generate exceptions (e.g., LD/ST); otherwise artificial exceptions might be generated
Global scheduling techniques

Example of speculative execution

if (A==B)
    A = C;
else
    A = B;

Straight forward implementation

Global scheduling

Note that a load operation normally takes longer to execute; it is thus the best candidate to perform speculation
Conditional instructions or with predication

- Instruction is executed only upon a predicate or a condition
- If the condition/predicate is false, the instruction is discarded later on the pipeline
- Transforms control hazards in data hazards

Most architectures supply a reduced set of conditional instructions

- Pentium, PowerPC, MIPS, ...: CMOVZ Rx, Ry, Rz
- In ARM ISA all instructions are conditional
- IA-64 architecture supports predication of all instructions because it is an EPIC (Explicitly Parallel Instruction Computer)
- Many GPU architectures also support predication
Hardware support for global scheduling techniques

Example

```c
if (X<0) /* X em R1 */
    Y = -X; /* R2← Y */
else
    Y = X;
```

**Straight forward implementation**

```assembly
BLTZ R1,IF
ELSE:
    ADD R2,R0,R1
    J ENDIF
IF:
    SUB R2,R0,R1
ENDIF:
```

**Global scheduling**

```assembly
SUB R2,R0,R1
SLT R3,R0,R1 ; R3←1 if R1>0
MOVNZ R2,R1,R3 ; R2←R1 if R3≠0
```

**No control dependencies**

*Allows for normal instruction retiming and for the application of loop unrolling and/or software pipelining*

**With hardware support**

```assembly
SUB R2,R0,R1
BGEZ R1,ENDIF
ELSE
    ADD R2,R0,R1
ENDIF:
```
Problems that can only be solved with hardware support

- Exception, especially those leading to program termination
- Retiming (i.e., instruction shifting) of load/store instructions

Possible solutions:

a) Only perform retiming on instructions that do not generate exceptions

b) Cooperation between hardware and software to ignore exceptions in speculative instructions

c) Poison bits associated to the registers

When a register has a poison bit equal to one, it means that the stored value comes from an instruction that generated an exception
Hardware support for global scheduling techniques

Handling exceptions with poison bits

- Add a poison bit per register, indicating whether an exception was generated
- Add a speculative bit per instruction to indicate if the instruction was executed speculatively
- When a speculative instruction uses a source register with poison bit=1, propagate the poison bit to the destination register
- If a non-speculative instruction uses a register value with poison bit=1, generate an exception
- There are no speculative memory writes, thus writing a speculative register value to memory generates an exception
Hardware support for global scheduling techniques

Handling load/store instructions

- Dependencies in memory accesses are sometimes hard to be identified, specially with speculative execution.
- How can we switch the order between load and store instructions without changing the program?

IDEA:
- Move the load instruction and mark it as speculative (LD $\rightarrow$ sLD)
- Leave at the original load location a check (guardian) instruction

During execution
- When a speculative load is performed mark the load address on a table
- If a subsequent store changes the memory on the load address before reaching the check instruction, the speculation has failed; perform one of:
  - If only the load was speculated, update the loaded value
  - If other instructions were speculated, redo the operations
Dynamic techniques to extract parallelism
  ➢ Scoreboard
Exercises
Exercise 1

for (i=0; i<100; i++) {
    A[i] = B[i+1] + C[i+1]; /* S1 */
    B[i+1] = cos([A[i]]); /* S2 */
    A[i] = 2*A[i]; /* S3 */
}

- Assumindo que os arrays são distintos e não sobrepostos, apresente um grafo com todas as dependências existentes.

- Diga, justificando, se o código apresentado é paralelizável ao nível de ciclo, i.e., se é possível executar cada iteração deste ciclo independentemente e em paralelo. Em caso afirmativo, reescreva-o por forma a que a sua execução possa ser efectuada em paralelo.
**Exercise 1 — Solution**

A)  
```
for (i=0; i<100; i++) {
    A[i] = B[i+1] + C[i+1]; /* S1 */
    B[i+1] = cos([A[i]]);    /* S2 */
    A[i] = 2*A[i];           /* S3 */
}
```

B)  
```
for (i=0; i<100; i++) {
    A[i] = B[i+1] + C[i+1]; /* S1 */
    B[i+1] = cos([A[i]]);    /* S2 */
    A[i] = 2*A[i];           /* S3 */
}
```

**Iteration**  
- **Intra** (same i)  
- **Inter** (i different)

**Type of dependence**  
- **Verdadeira** (RAW)  
- **De Nome** (WAW / WAR)

**Directamente paralelizável sem necessitar de alterações**
Assumindo que os arrays são distintos e não sobrepostos, apresente um grafo com todas as dependências existentes.

Diga, justificando, se o código apresentado é paralelizável ao nível de ciclo, i.e., se é possível executar cada iteração deste ciclo independentemente e em paralelo. Em caso afirmativo, reescreva-o por forma a que a sua execução possa ser efectuada em paralelo.
Exercise 2 — Solution

A)

\[ X[101] = \alpha A[101] \]

\[
\text{for } (i=100; i>1; i--) \{
    A[i] = B[i] + C[i]; \quad /* S1 */
    X[i] = \alpha A[i]; \quad /* S2 */
    D[i] = B[i] + \beta; \quad /* S3 */
\}
\]


\[ D[1] = B[1] + \beta; \]

\[ C=X; \]

B)

\[ X[101] = \alpha A[101] \]

Paralelizável mediante alterações
Assumindo que os arrays são distintos e não sobrepostos, apresente um grafo com todas as dependências existentes.

Diga, justificando, se o código apresentado é paralelizável ao nível de ciclo, i.e., se é possível executar cada iteração deste ciclo independentemente e em paralelo. Em caso afirmativo, reescreva-o por forma a que a sua execução possa ser efectuada em paralelo.
Exercise 3 – Solution

A) 

B) 

for (i=100; i>0; i--) {
    C[i]   = A[i+1] * sqrt(B[i]);    /* S1 */
    B[i+1] = alpha*C[i+1];            /* S2 */
    A[i]   = B[i+2] + beta;           /* S3 */
}

O ciclo não é paralelizável, i.e., não é possível executar diferentes iterações do ciclo ao mesmo tempo!
Assumindo que os arrays são distintos e não sobrepostos, apresente um grafo com todas as dependências existentes.

Diga, justificando, se o código apresentado é paralelizável ao nível de ciclo, i.e., se é possível executar cada iteração deste ciclo independentemente e em paralelo. Em caso afirmativo, reescreva-o por forma a que a sua execução possa ser efectuada em paralelo.