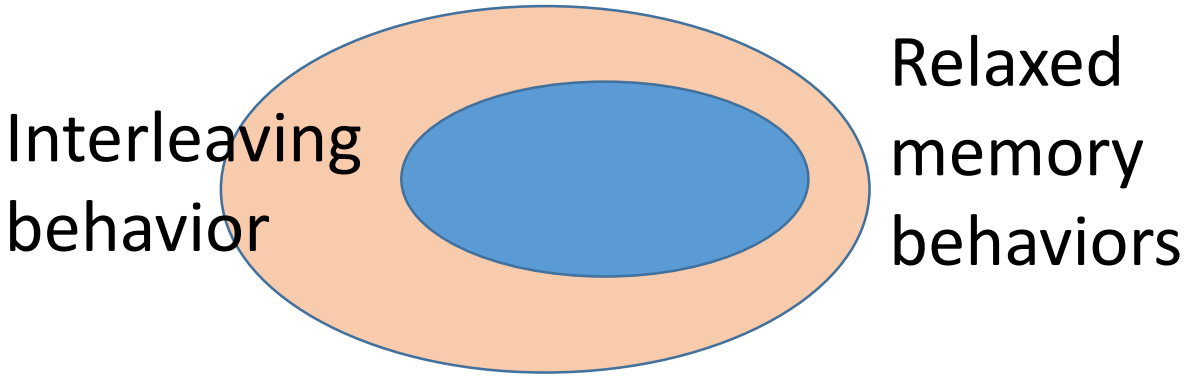
Recap



X=Y=0

X = 1;	Y = 1;
a = Y;	b = X;

a=0, b=0

Store Buffer (SB) Example

X=Y=0

X = 1; a = Y;	Y = 1; b = X;

a=0, b=0

Sequential consistency is too restrictive for this program behavior

It requires some relaxation to justify the program behavior

- Ordering of the instructions

Order relaxation

No order for Store – Load

X=Y=0

X = 1;	Y = 1;
a = Y;	b = X;

a=0, b=0

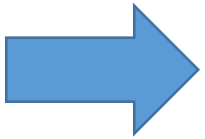
Order relaxation

No order for Store – Load

X=Y=0

X = 1;	Y = 1;
a = Y;	b = X;

a=0, b=0



X=Y=0

a = Y;	b = X;
X = 1;	Y = 1;

a=0, b=0

Order relaxation

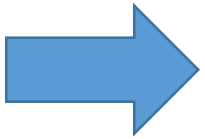
No order for Store – Load

X=Y=0

```
X = 1;
a = Y;
```

```
Y = 1;
b = X;
```

a=0, b=0



X=Y=0

```
a = Y; // 0
X = 1;
```

```
b = X; // 0
Y = 1;
```

a=0, b=0

Is it allowed by any relaxed memory model?

Total Store Order (TSO)

Relaxation for Store – Load

a;b	Store	Load
Store	N	Y
Load	N	N

Another Program: Message Passing (MP)

X=Y=0

```
X = 1;
Y = 1;
```

```
a = Y;
if(a == 1)
    b = X;
```

Is a=1, b=0 possible an allowed behavior?

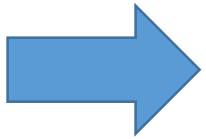
- SC ?
- TSO ?

Program: Message Passing (MP)

X=Y=0

```
X = 1;
Y = 1;

a = Y;
if(a==1)
    b = X;
```



X=Y=0

```
Y = 1;
X = 1;

a = Y; // 1
if(a==1)
    b = X;
```

a =1, b = 0

Requires Store-Store relaxation

Partial Store Order (PSO)

Relaxation for Store – Store

a;b	Store	Load
Store	Y	Y
Load	N	N

Yet Another Program: Load Buffering (LB)

X=Y=0

a = X;	a = Y;
Y = 1;	X= 1;

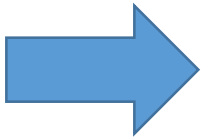
Is a=1, b=1 possible an allowed behavior?

- SC ?
- TSO ?
- PSO ?

Program: Message Passing (MP)

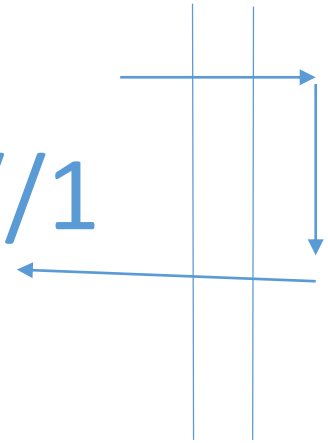
X=Y=0

a = X;	b = Y;
Y = 1;	X=1;



X=Y=0

Y = 1;	X = 1;
a = X; //1	b = X; // 1



a =1, b = 1

Requires Load-Store relaxation

Relaxed Memory Order (RMO)

Relaxation for Store – Store

a;b	Store	Load
Store	Y	Y
Load	Y	Y

Summary

ACC. Pair Relaxation → / Memory Model ↓	Store-Load	Store-Store	Load-Load	Load-Store
SC	N	N	N	N
TSO	Y	N	N	N
PSO	Y	Y	N	N
RMO	Y	Y	Y	Y

Recap: Question

Given a program and an outcome, is it a correct outcome?

Depends on

program execution

More Specifically …

Given a program and an outcome, is it a correct outcome?

Depends on

program execution

Depends on

Concurrency semantics/memory model

Memory Model

‘In computing, a **memory model** describes the interactions of threads through memory and their shared use of the data.’
~ wikipedia

e.g.

- SC
- TSO
- PSO
- RMO

Defining a Concurrency Semantics

- Transfomational
- Operational
- Axiomatic

Transformational Semantics

Step 1: Transform the program

Step 2: Consider all interleaving executions

Reordering & interleaving explains TSO, PSO, RMO

Note: Reordering requires dependency analysis

Dependencies

<table>
<tr><th colspan="3">Data-Dependence</th><th rowspan="2">Control-Dependence</th><th rowspan="2">Address-Dependence</th></tr>
<tr><td>Flow-Dependence</td><td>Output-Dependence</td><td>Anti-Dependence</td></tr>
</table>

Dependencies

Data-Dependence			**Control-Dependence**	**Address-Dependence**
Flow-Dependence	Output-Dependence	Anti-Dependence		
a = X; Y = a;				

Dependencies

Data-Dependence			**Control-Dependence**	**Address-Dependence**
Flow-Dependence	Output-Dependence	Anti-Dependence		
a = X; Y = a;	X = 1; X = 2;			

Dependencies

Data-Dependence			**Control-Dependence**	**Address-Dependence**
Flow-Dependence	Output-Dependence	Anti-Dependence		
a = X; Y = a;	X = 1; X = 2;	a= X; X = 1;		

Dependencies

Data-Dependence			**Control-Dependence**	**Address-Dependence**
Flow-Dependence	Output-Dependence	Anti-Dependence		

a = X;
Y = a;
	X = 1;
	X = 2;
		a= X;
		X = 1;
			a = X;
			if(a==1)
			Y = 1;	

Dependencies

Data-Dependence			**Control-Dependence**	**Address-Dependence**
Flow-Dependence	Output-Dependence	Anti-Dependence		

a = X;
Y = a;
	X = 1;
	X = 2;
		a= X;
		X = 1;
			a = X;
			if(a==1)
			Y = 1;
				a = X;
				b = Y[a];

True & False Dependence

True dependence: Cannot be eliminated by any transformation

```
a = X;
Y = a*5;
```

False dependence: Can be eliminated, depends upon transformations

```
a = X;
Y = a*0;
```

→

```
a = X;
Y = 0;
```

Limitations of Transformational Approach

1. Dependency analysis is undecidable in general

    * cannot always differentiate true & false dependencies

```
func(int m){
:
a = X;
Y = a*m;  // is m = 0 here?
}
```

Limitations of Transformational Approach

1. Dependency analysis is undecidable in general

    * cannot always differentiate true & false dependencies

2. Does not explain all behaviors of modern architectures

Limitations of Transformational Approach

Example:

```
                    X[0] = Y[1] = 0;
X[0] = 1;  ||  a = X[0];      // 1  ||  c = Y[0];      // 1  ||  Y[1] = 1;
           ||  b = Y[a*0]; // 0    ||  d = X[a*0]; // 0    ||
```

Allowed in PowerPC and ARMv7

* even without removing false dependencies

Limitations of Transformational Approach

```
                    X[0] = Y[1] = 0;
X[0] = 1;  ||  a = X[0];      // 1  ||  c = Y[0];      // 1  ||  Y[1] = 1;
               b = Y[a*0]; // 0        d = X[a*0]; // 0
```

Sequentialization:

$C_0 \mid\mid C_1 \text{ --> } C_0;C_1$

```
X[0] = 1;                  Y[1] = 1;
a = X[0];          ||      c = Y[0];
b = Y[a*0]; // 0           d = X[a*0]; // 0
```

Limitations of Transformational Approach

```
X[0] = 1;
a = X[0];
b = Y[a*0]; // 0
```

```
Y[1] = 1;
c = Y[0];
d = X[a*0]; // 0
```

constant propagation

```
X[0] = 1;
a = 1;
b = Y[a*0]; // 0
```

```
Y[1] = 1;
c = 1
d = X[a*0]; // 0
```

Limitations of Transformational Approach

Another Example: Allowed by ARMv8.

```
                X = Y = 0;

a = X; // 1   ||   Y = X;   ||   X = Y;
X = 1;
```

Limitations of Transformational Approach

1. Dependency analysis is undecidable in general

    * cannot always differentiate true & false dependencies

2. Does not explain all behaviors of modern architectures

3. Assumes correctness of transformations

    ** instead of proving them correct !*