Scheduling Weakly Consistent C Programs for Reconfigurable Hardware

Nadesh Ramanathan, John Wickerson, George Constantinides.

Presented by
Akshay Gopalakrishnan

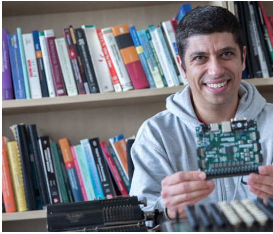
February 17, 2022

Who are they?

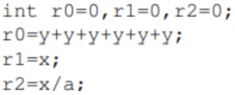
Yann Herklotz - PhD student supervised by John Wickerson, Imperial College London.

John Wickerson - Lecturer at Dept of Electrical and Electronic Engineering, Imperial College London.

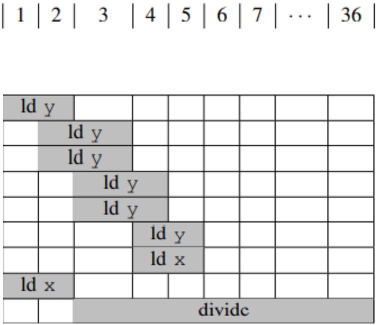
George Constantinides - Professor of Digital Computation, Imperial College London.



Scheduling Code block 1



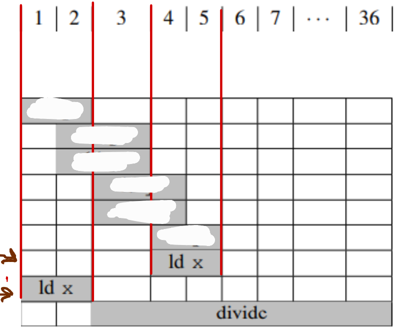
```
int r0=0,r1=0,r2=0;
r0=y+y+y+y+y+y;
r1=x;
r2=x/a;
```



1	2	3	4	5	6	7	···	36
ld y								
	ld y							
	ld y							
		ld y						
		ld y						
			ld y					
			ld x					
ld x								
		divide						

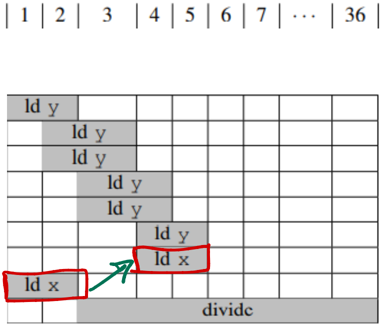
Scheduling Code block 1

```
int r0=0,r1=0,r2=0;
[redacted];
r1=x;
r2=x/a;
```



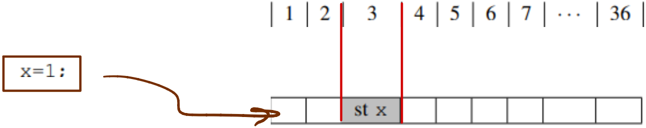
Scheduling Code block 1

```
int r0=0,r1=0,r2=0;
r0=y+y+y+y+y+y;
r1=x;
r2=x/a;
```



1	2	3	4	5	6	7	···	36
ld y								
	ld y							
	ld y							
		ld y						
		ld y						
			ld y					
			ld x					
ld x								
		dividc						

Scheduling Code block 2

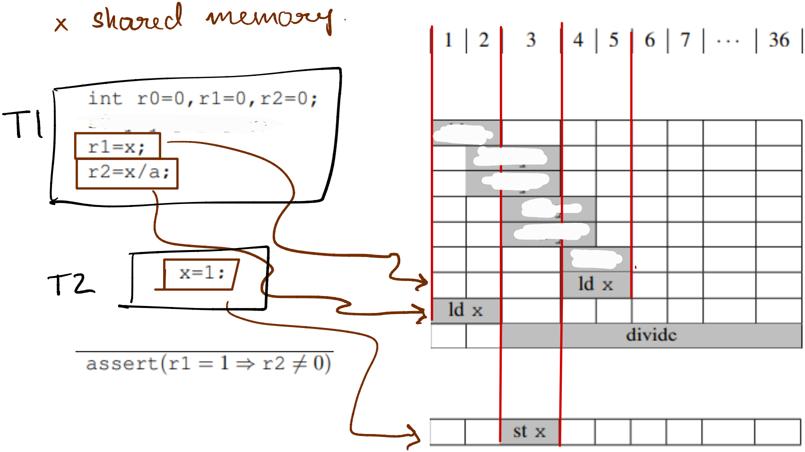


x=1;

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ··· | 36 |

st x

Concurrent scheduling of Block 1 and 2



x shared memory.

T1

```
int r0=0,r1=0,r2=0;

r1=x;
r2=x/a;
```

T2

```
x=1;
```

assert($r1 = 1 \Rightarrow r2 \neq 0$)

1	2	3	4	5	6	7	···	36
			ld x					
ld x								
		divide						
		st x						

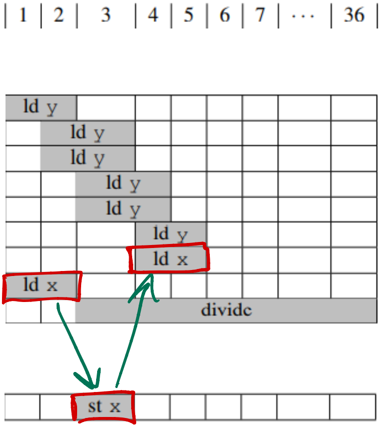
Concurrent scheduling of Block 1 and 2

```
int r0=0,r1=0,r2=0;
r0=y+y+y+y+y+y;
r1=x;
r2=x/a;
```

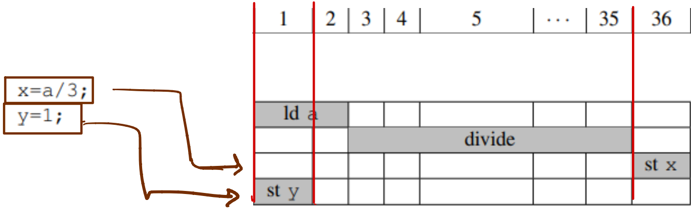
```
        x=1;
```

$$\boxed{\texttt{assert}(r1 = 1 \Rightarrow r2 \neq 0)}$$

↳ Fails.



Scheduling Code block 1

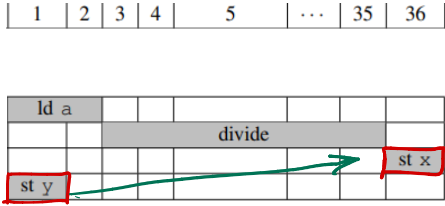


```
x=a/3;
y=1;
```

1	2	3	4	5	…	35	36
ld a							
		divide					
							st x
st y							

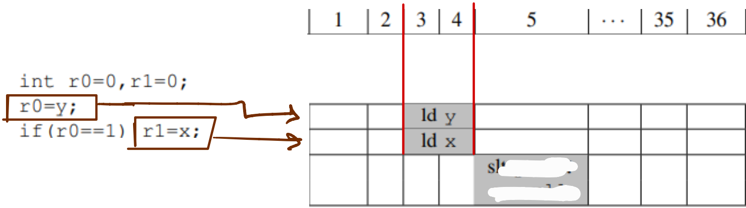
Scheduling Code block 1

```
x=a/3;
y=1;
```



1	2	3	4	5	⋯	35	36
ld a							
		divide					
							st x
st y							

Scheduling Code block 2

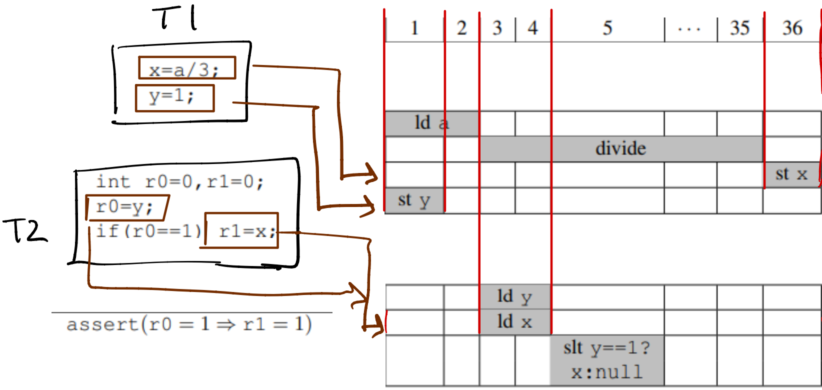


```
int r0=0,r1=0;
r0=y;
if(r0==1) r1=x;
```

1	2	3	4	5	...	35	36
		ld y					
		ld x					
				[redacted]			

Concurrent scheduling of Block 1 and 2

x, y shared memory.



```
x=a/3;
y=1;
```

```
int r0=0,r1=0;
r0=y;
if(r0==1) r1=x;
```

```
assert(r0 = 1 ⇒ r1 = 1)
```

1	2	3	4	5	...	35	36
ld a							
		divide					
							st x
st y							
		ld y					
		ld x					
				slt y==1? x:null			

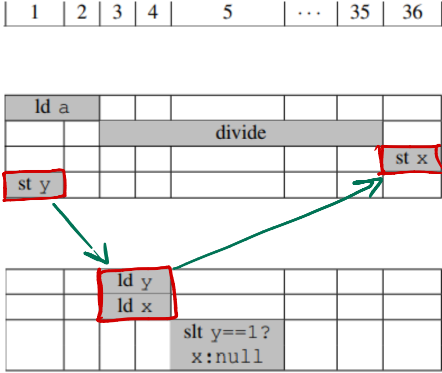
Concurrent scheduling of Block 1 and 2

```
x=a/3;
y=1;
```

```
int r0=0,r1=0;
r0=y;
if(r0==1) r1=x;
```

$$\boxed{\texttt{assert}(\texttt{r0} = 1 \Rightarrow \texttt{r1} = 1)}$$

↳ Fails.



1	2	3	4	5	···	35	36
ld a							
		divide					
							st x
st y							
		ld y					
		ld x					
				slt y==1? x:null			

Data Dependencies: non-aliasing

$$E_{\text{intra-iter}} = \{(v, v', \;) \mid \overset{\substack{\text{Syntax order} \\ \uparrow}}{\boxed{sb(v, v')}} \wedge \overset{\substack{\text{same location} \\ \uparrow}}{\boxed{sloc(v, v')}} \wedge (v \in \boxed{V_{\text{st}}} \vee v' \in \boxed{V_{\text{st}}})\}$$

$$E_{\text{inter-iter}} = \{(v, v', \underset{\substack{\downarrow \\ \text{Iteration distance.}}}{\boxed{1}}) \mid sloc(v, v') \wedge (v \in V_{\text{st}} \vee v' \in V_{\text{st}})\}.$$

Adding WW—WR—RW—RR Dependancies

$$E_{\mathrm{sc}\downarrow} = \{\boxed{(v, v'}, \;) \mid sb(v, v') \wedge \boxed{v \in V_{\mathrm{sc}}}\}$$

$$E_{\mathrm{sc}\uparrow} = \{\boxed{(v, v'}, \;) \mid sb(v, v') \wedge \boxed{v' \in V_{\mathrm{sc}}}\}$$

```
int r0=0,r1=0,r2=0;
r0=y+y+y+y+y+y;  sc
r1=x;            sc
r2=x/a;          sc
```

```
x=a/3;  sc
y=1;    sc
```

Final Dependency Expression

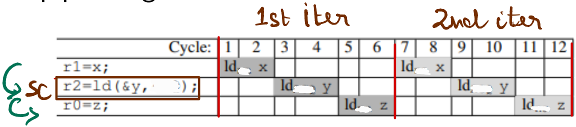
$$
\begin{aligned}
E^{\text{pipe}}_{\text{mem,SC}} = {} & E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \cup \\
& E_{\text{at}\downarrow} \cup E_{\text{at}\uparrow} \cup \boxed{E_{\text{at-inter-iter}}}
\end{aligned}
$$

↙

For loop pipelining.

Example

Without pipelining

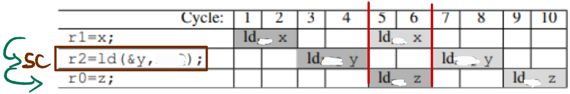


1st iter 2nd iter

SC

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12
r1=x;	ld	x					ld	x				
r2=ld(&y,);			ld	y					ld	y		
r0=z;					ld	z					ld	z

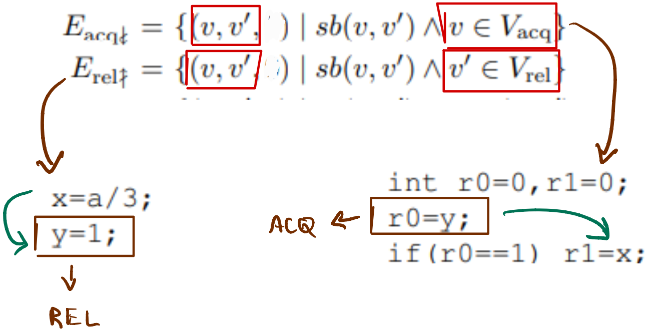
With pipelining



SC

Cycle:	1	2	3	4	5	6	7	8	9	10
r1=x;	ld	x			ld	x				
r2=ld(&y,);			ld	y			ld	y		
r0=z;					ld	z			ld	z

Weakening: Adding Release-Acquire Dependencies



$$E_{\mathrm{acq}\ddagger} = \{\boxed{(v, v',}\ ) \mid sb(v, v') \wedge \boxed{v \in V_{\mathrm{acq}}}\}$$

$$E_{\mathrm{rel}\ddagger} = \{\boxed{(v, v',}\ ) \mid sb(v, v') \wedge \boxed{v' \in V_{\mathrm{rel}}}\}$$

```
x=a/3;
y=1;
```

REL

```
int r0=0,r1=0;
r0=y;
if(r0==1) r1=x;
```

ACQ

Adding RR Dependency

$$V_{at} = V_{SC} \cup V_{REL} \cup V_{ACQ}$$

$$E_{\mathrm{RAR}} = \{(v, v', 0) \mid sb(v, v') \wedge \boxed{sloc(v, v')} \wedge \boxed{v \in V_{\mathrm{at}} \cap V_{\mathrm{ld}}} \wedge \boxed{v' \in V_{\mathrm{at}} \cap V_{\mathrm{ld}}}\}.$$

```
     int r0=0,r1=0,r2=0;
     r0=y+y+y+y+y+y;
AT   r1=x;
AT   r2=x/a;
```

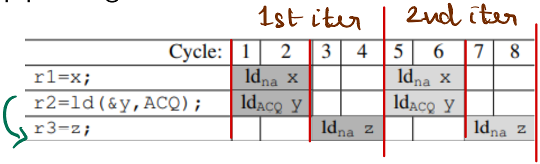
Final Dependency Expression

$$\begin{aligned} E^{\text{pipe}}_{\text{mem,weak}} = {} & E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \cup E_{\text{sc}\rotatebox{180}{$\ddagger$}} \cup E_{\text{sc}\ddagger} \cup \\ & E_{\text{acq}\rotatebox{180}{$\ddagger$}} \cup E_{\text{rel}\ddagger} \cup E_{\text{RAR}} \cup \\ & \boxed{\begin{aligned} & E_{\text{sc-inter-iter}} \cup E_{\text{acq-inter-iter}} \cup \\ & E_{\text{rel-inter-iter}} \cup E_{\text{RAR-inter-iter}} \end{aligned}} \end{aligned}$$

↳ loop pipelining.

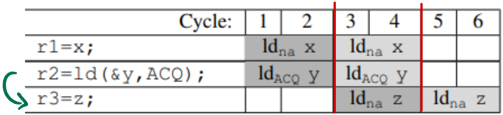
Example

Without pipelining


1st iter
2nd iter

Cycle:	1	2	3	4	5	6	7	8
r1=x;	ld$_{na}$ x				ld$_{na}$ x			
r2=ld(&y,ACQ);	ld$_{ACQ}$ y				ld$_{ACQ}$ y			
r3=z;			ld$_{na}$ z				ld$_{na}$ z	

With pipelining



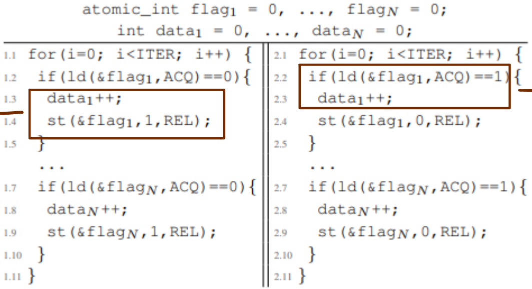
Cycle:	1	2	3	4	5	6
r1=x;	ld$_{na}$ x		ld$_{na}$ x			
r2=ld(&y,ACQ);	ld$_{ACQ}$ y		ld$_{ACQ}$ y			
r3=z;			ld$_{na}$ z		ld$_{na}$ z	

Message Passing Algorithm

$x = a/3$

$y = 1_{REL}$

N times



```
           atomic_int flag1 = 0, ..., flagN = 0;
              int data1 = 0, ..., dataN = 0;
for(i=0; i<ITER; i++) {        || for(i=0; i<ITER; i++) {
 if(ld(&flag1,ACQ)==0){        ||  if(ld(&flag1,ACQ)==1){
  data1++;                     ||   data1++;
  st(&flag1,1,REL);            ||   st(&flag1,0,REL);
 }                             ||  }
 ...                           ||  ...
 if(ld(&flagN,ACQ)==0){        ||  if(ld(&flagN,ACQ)==1){
  dataN++;                     ||   dataN++;
  st(&flagN,1,REL);            ||   st(&flagN,0,REL);
 }                             ||  }
}                              || }
```

Fig. 5. A two-threaded message-passing example with *acquire-release* semantics on N independent channels.

$r0 = Y_{ACQ}$

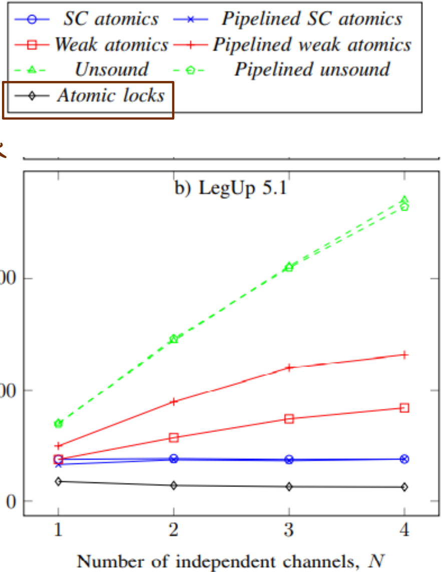
$r1 = X$

N times

Impact of Modified Scheduling

Use ←

lock/unlock



Scheduling Weakly Consistent C Programs for Reconfigurable Hardware

Presented by Akshay Gopalakrishnan

Motivating Examples

Example 1

Example 2

Problem

Dependency

SC Dependencies

New Scheduling

Rel-Acq Dependencies

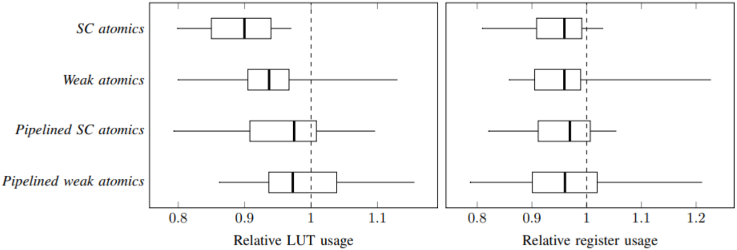
New Scheduling

Evaluation

MP Example

Pros/Cons

Resource Usage



Positives

- New scheduling for programs with atomics. (Ld|St)
- Do not require locks/notion of critical section for Hardware design.
- Efficient and correct scheduling

Pending Work

Compare-And-Swap

Test-And-Set

↑

- Does not support $\boxed{\text{Atomic Read-Modify-Write}}$ instructions.
- Addressing other potential transformations (eg: Elimination, Introduction, $\boxed{\text{Inlining,}}$ etc) for efficient scheduling.

↙

$$T1 \parallel T2 \rightsquigarrow T1; T2$$

Thank you

- John's Blog
- Previous Work
- Testing+Verification

Questions?