Scheduling Weakly Consistent C Programs for Reconfigurable Hardware

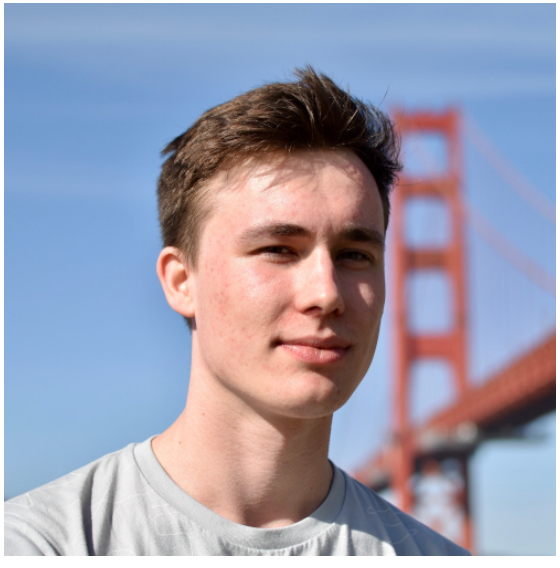
Nadesh Ramanathan, John Wickerson, George Constantinides.

Presented by
Akshay Gopalakrishnan

February 17, 2022

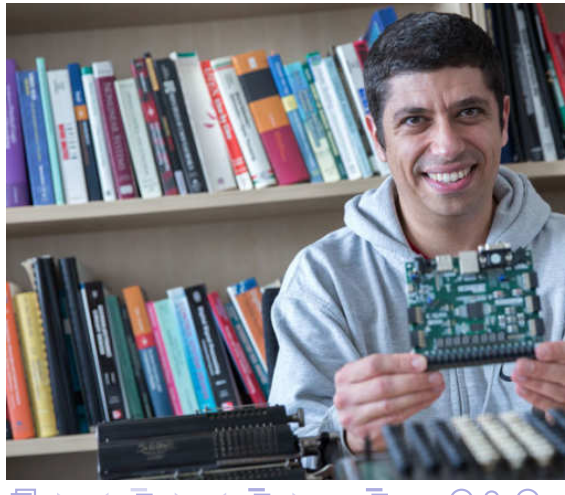
Who are they?

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



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

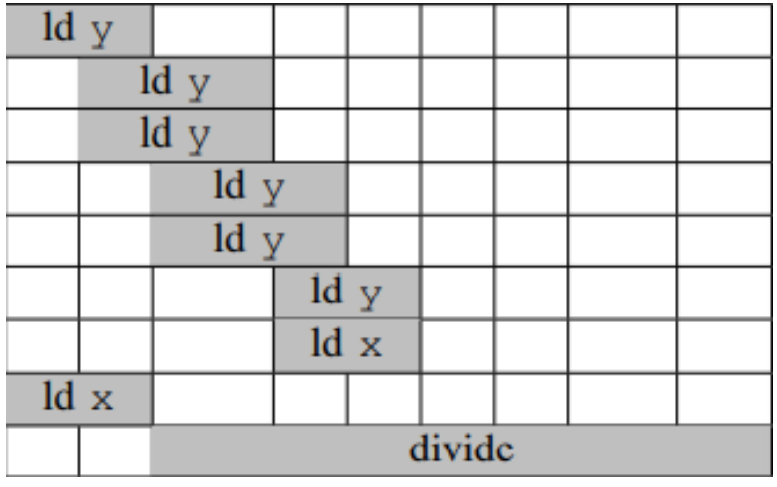
George Constantinides - Professor of Digital Computation, Imperial College London.



Scheduling Code block 1

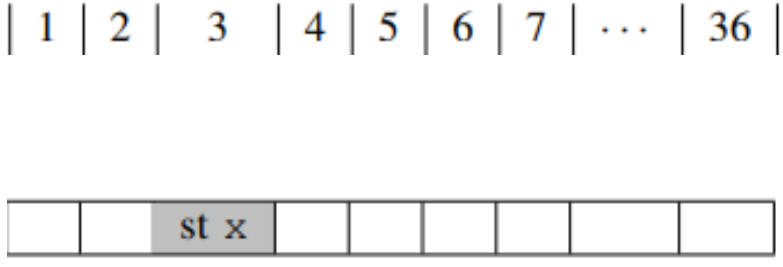
1	2	3	4	5	6	7	···	36

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



Scheduling Code block 2


```
x=1;
```

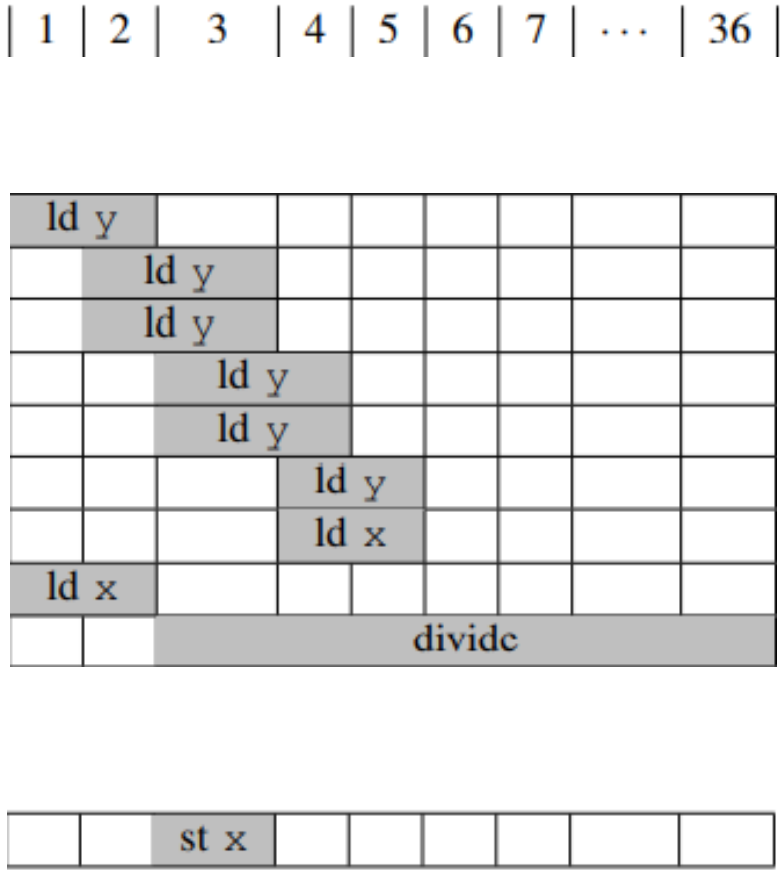


Concurrent scheduling of Block 1 and 2


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

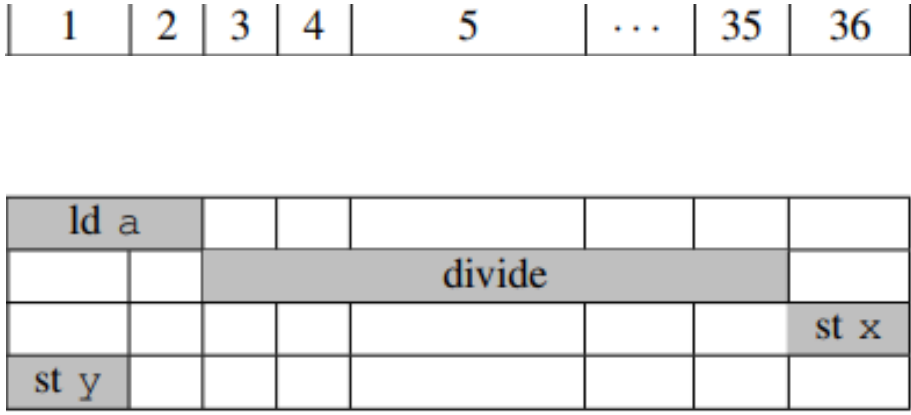
```
x=1;
```

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



Scheduling Code block 1


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



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					
				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;
```

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

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

Data Dependencies: non-aliasing

$$E_{\text{intra-iter}} = \{(v, v', 0) \mid sb(v, v') \wedge sloc(v, v') \wedge (v \in V_{\text{st}} \vee v' \in V_{\text{st}})\}$$

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

Adding WW—WR—RW—RR Dependancies

$$E_{\mathrm{sc}\nmid} = \{(v, v', 0) \mid sb(v, v') \wedge v \in V_{\mathrm{sc}}\}$$
$$E_{\mathrm{sc}\nmid} = \{(v, v', 0) \mid sb(v, v') \wedge v' \in V_{\mathrm{sc}}\}$$

```
int r0=0,r1=0,r2=0;
r0=y+y+y+y+y+y;          x=a/3;
r1=x;                    y=1;
r2=x/a;
```

Final Dependency Expression

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

Example

Without pipelining

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

With pipelining

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

Weakening: Adding Release-Acquire Dependencies

$$E_{\text{acq}\natural} = \{(v, v', 0) \mid sb(v, v') \wedge v \in V_{\text{acq}}\}$$
$$E_{\text{rel}\natural} = \{(v, v', 0) \mid sb(v, v') \wedge v' \in V_{\text{rel}}\}$$

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

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

Adding RR Dependency

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

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

Final Dependency Expression

$$E^{\text{pipe}}_{\text{mem,weak}} = E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \cup E_{\text{sc}\natural} \cup E_{\text{sc}\sharp} \cup E_{\text{acq}\natural} \cup E_{\text{rel}\sharp} \cup E_{\text{RAR}} \cup E_{\text{sc-inter-iter}} \cup E_{\text{acq-inter-iter}} \cup E_{\text{rel-inter-iter}} \cup E_{\text{RAR-inter-iter}}$$

Example

Without pipelining

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	

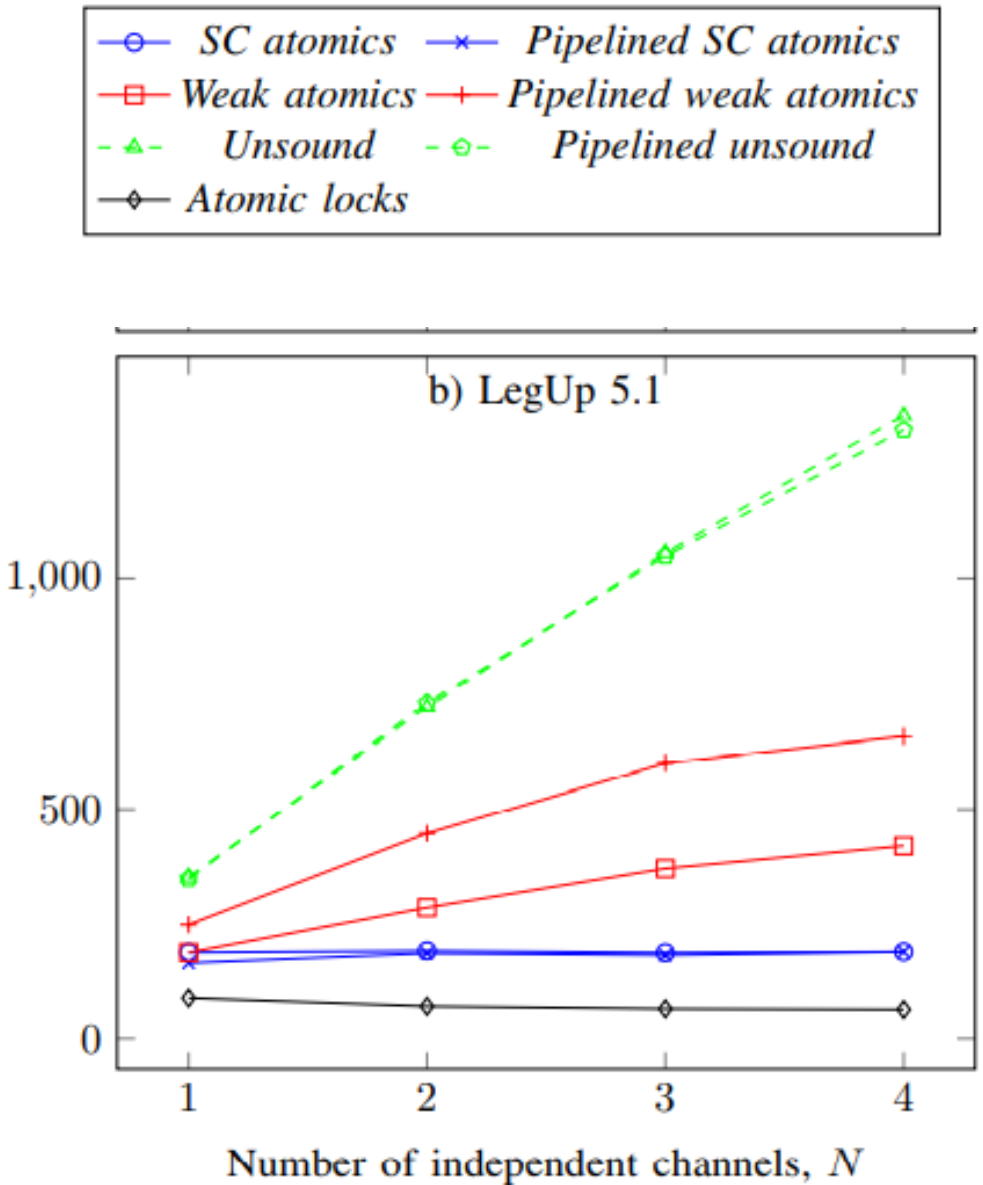
Message Passing Algorithm


```
atomic_int flag_1 = 0, ..., flag_N = 0;
int data_1 = 0, ..., data_N = 0;
```

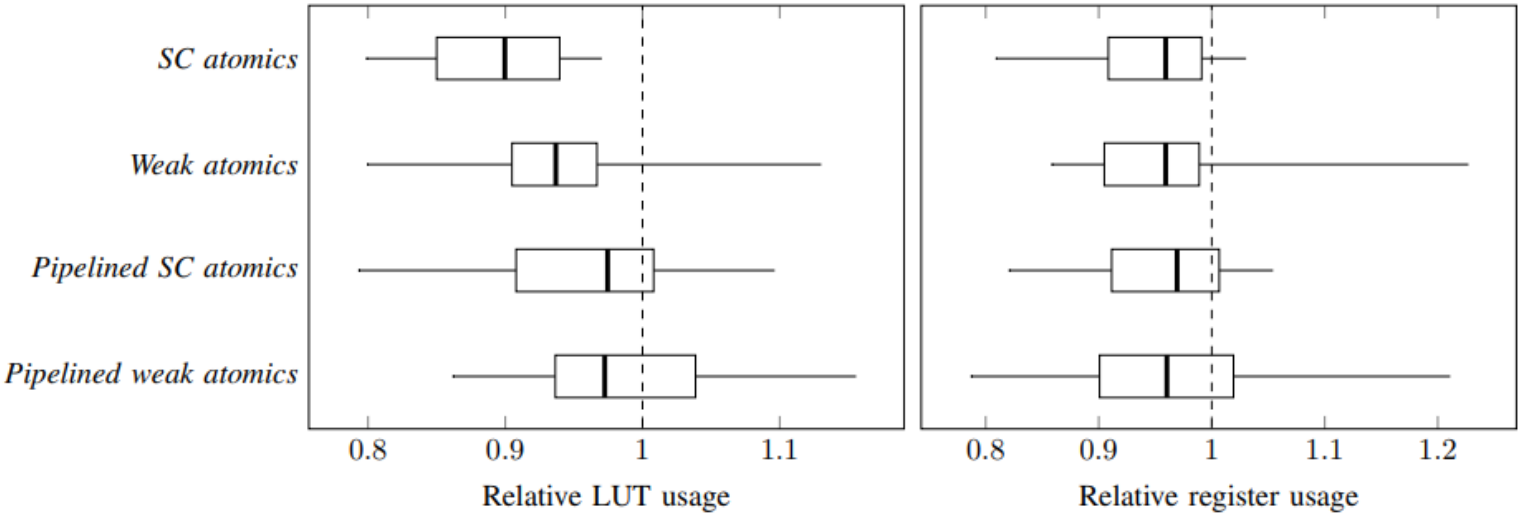
1.1 `for(i=0; i<ITER; i++) {`	2.1 `for(i=0; i<ITER; i++) {`
1.2 `if(ld(&flag_1,ACQ)==0){`	2.2 `if(ld(&flag_1,ACQ)==1){`
1.3 `data_1++;`	2.3 `data_1++;`
1.4 `st(&flag_1,1,REL);`	2.4 `st(&flag_1,0,REL);`
1.5 `}`	2.5 `}`
`...`	`...`
1.7 `if(ld(&flag_N,ACQ)==0){`	2.7 `if(ld(&flag_N,ACQ)==1){`
1.8 `data_N++;`	2.8 `data_N++;`
1.9 `st(&flag_N,1,REL);`	2.9 `st(&flag_N,0,REL);`
1.10 `}`	2.10 `}`
1.11 `}`	2.11 `}`

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

Impact of Modified Scheduling



Resource Usage



Positives

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

Pending Work

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

Thank you

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

Questions?