Towards Memory-Optimal Schedules for SDF

Mitchell Jones

Department of Computer Science
University of Illinois at Urbana-Champaign

Julian Mestre

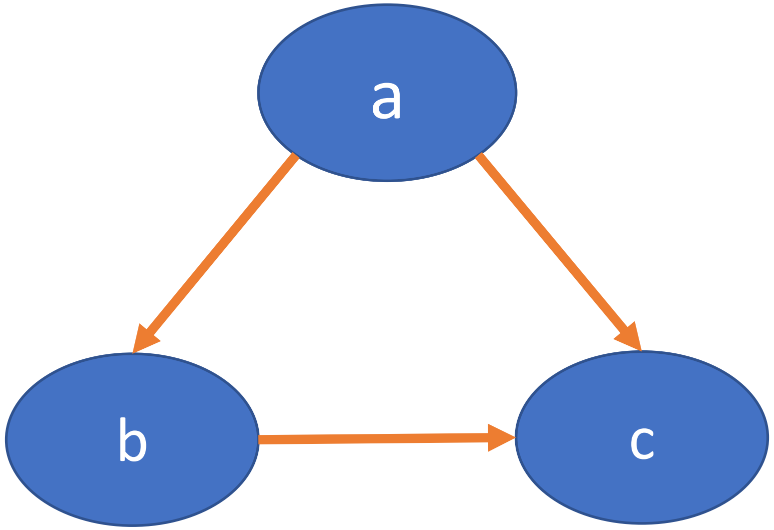
School of Information Technologies
The University of Sydney

Bernhard Scholz

School of Information Technologies
The University of Sydney

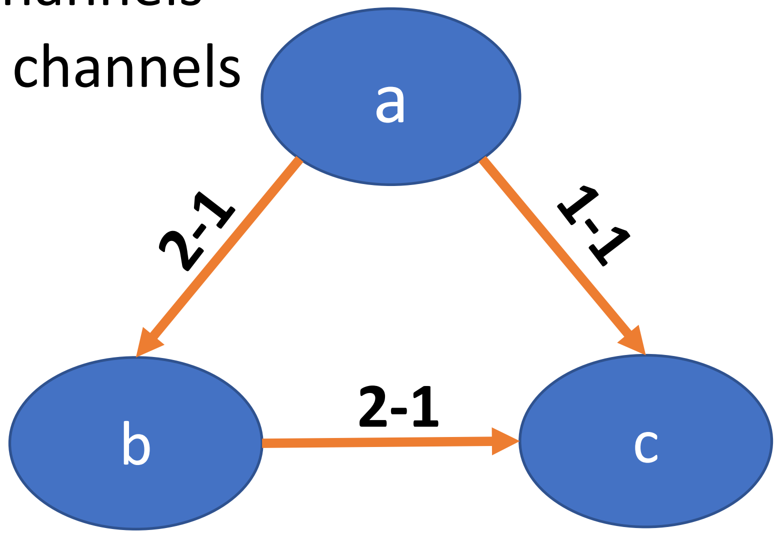
Data-Flow Computation

- Stream programming paradigm based on Kahn's processing model
- Processes large unbounded regular sequences of data forever
- Applications
  - Digital signal processing, audio, video, graphics, networking, and for big data
- Computations
  - Actors communicate via data channels only
  - Data-channel connects *producer* with *consumer*
  - Tokens are send and received on data channels
  - Actor invocations (*aka. firing*) require coordination
    - Otherwise starvation of actors or memory depletion



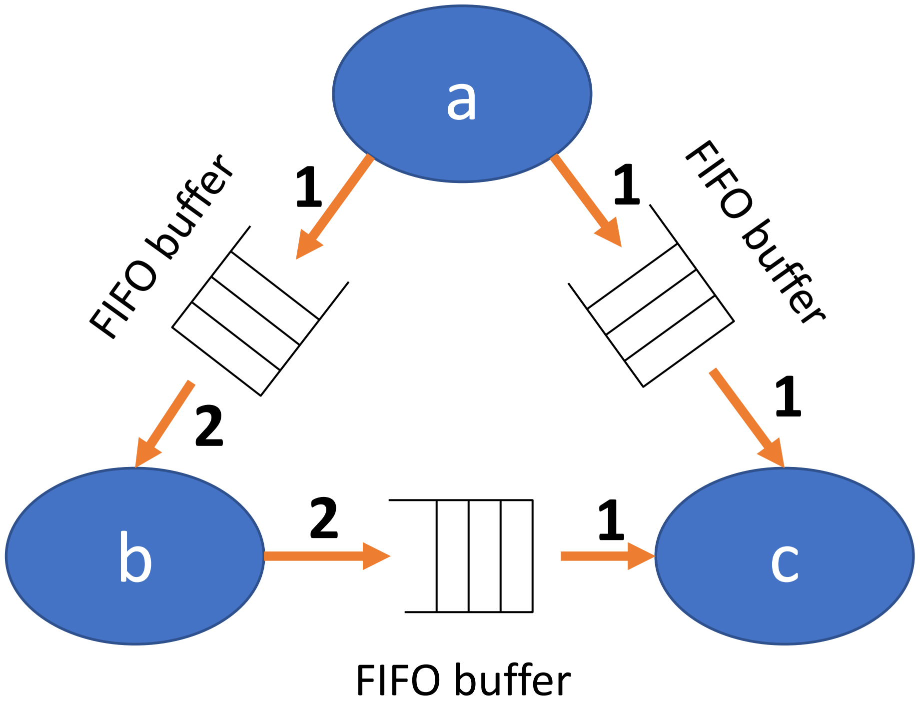
Synchronous Data-Flow (SDF)

- Restricted Data-Flow Computational model
- Permits static scheduling / run in steady state
  - Pre-computes sequences of actor invocations (firings)
- Per actor firing data rates are fixed:
  - Actor produces a fixed number of tokens on outgoing channels
  - Actor consumes a fixed number of tokens from ingoing channels
- Data-channels implemented as FIFO-buffers
- ***Research Questions***:
  - *What are the sizes of the FIFO buffers?*



Data-Channels as FIFO Buffers

- Data channels implemented as FIFO buffers
  - Tokens from producer stored in buffer
  - Consumer can retrieve them
- FIFO buffers require memory
  - Scarce resource
  - Cache effects
  - Small memory for embedded systems
- Different FIFO implementations
  - Static allocation in memory
  - Dynamic allocation
  - In hardware



Static FIFO Buffers for Data Channels

- FiFO buffer for a single data-channel between actor u and actor v
- Memory is ***not*** shared between FIFO buffers; size is fixed.

```
token queue_uv[SZ_uv]; // memory for data channel
int head_uv = 0, tail_uv = 0;

void produce_uv(token t) {
   queue_uv[tail_uv++] = t;
   tail_uv %= SZ_uv;}

token consume_uv() {
   token t=queue_uv[head_uv++];
   head_uv %= SZ_uv;
   return t; }
```

- ***Minimize the sum over all buffer sizes `SZ_uv`?***

Dynamic FIFO Buffers for Data Channels

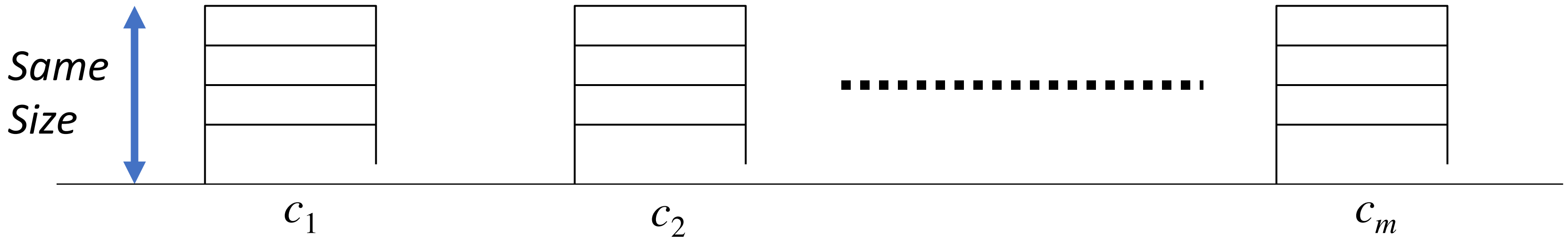
- FiFO buffer for a single data-channel between actor u and actor v
- Memory for tokens is shared between FIFO buffers; size is variable.

```
tokenQ head_uv = NULL, tail_uv = NULL;
void produce_uv(token t) {
  tokenQ p = alloc(t);
  if (head_uv == NULL) head_uv = tail_uv = p;
  else tail_uv->next = p;
}
token consume_uv() {
  token t = head_uv->data;
  tokenQ p = head_uv;
  head_uv = head_uv->next; free(p);
  if (head_uv == NULL) tail_uv = NULL;
  return t; }
```

- ***Minimize heap memory used by alloc/free!***

Hardware Implementation for Data Channels

- Fixed-sized FIFO buffer for all data-channels



- May be relevant for hardware/FPGA implementations for SDF
- Minimize the total memory for all fixed sized FIFO buffers

Research Result for Minimizing FIFO Buffers

- Our research results for minimizing sizes of FIFO buffers

Static FIFOs	Dynamic FIFOs	Hardware FIFOs
Minimal in ***P***	Minimal in ***NP***	Minimal in ***P***

- Main Idea for Static FIFOs and HW FIFOs
  - Exploit properties of steady state scheduling theory
  - Online algorithm based on priority queues
    - Space complexity $O(n)$; run-time complexity $O(\log n)$ per actor invocation
  - Priority queue contains a single element for each actor

Steady-State Schedule

- Static periodic finite schedule for well-formed SDF programs
- Fill-state of buffers is the same before and after executing schedule
  - Also known as steady-state
- Static periodic finite schedule is executable ad-infinitum
- Basic Balance Equations for Steady State Schedule:

$$
\begin{aligned}
&p(u,v)\cdot r(u) = c(u,v)\cdot r(v) && \forall (u,v)\in E\\
&r(u)\geq 0 && \forall u \in V
\end{aligned}
$$

- Production rate $p(u,v)$; consumption rate $c(u,v)$; repetitions $r(u)$
- Null-space of topological matrix

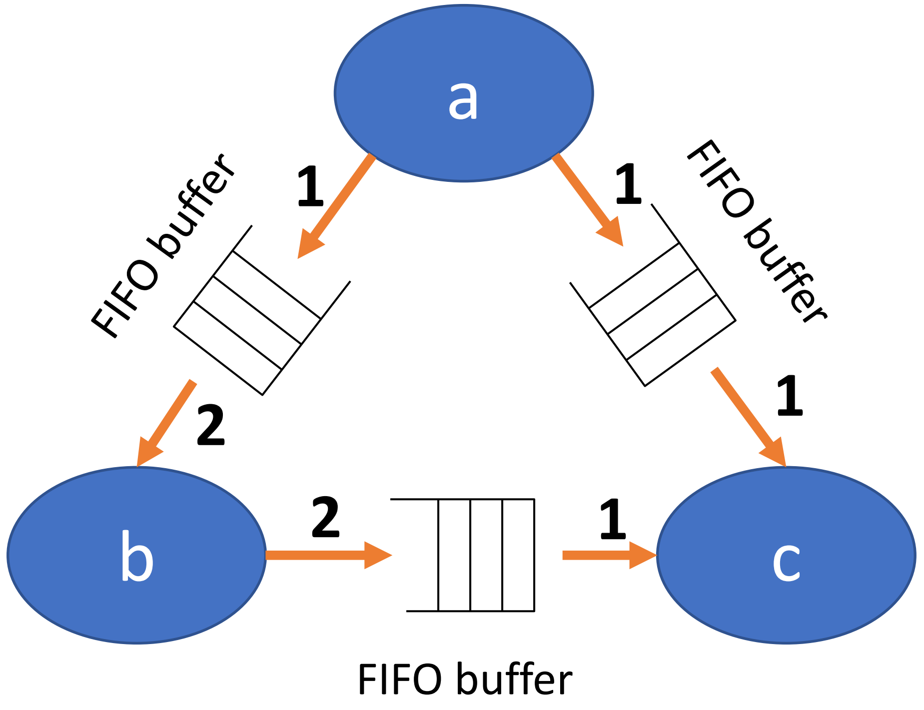
Example: Steady-State Schedule

- Balance Equations

$$
\begin{aligned}
1 \cdot r(a) &= 2 \cdot r(b) \\
1 \cdot r(a) &= 1 \cdot r(c) \\
2 \cdot r(b) &= 1 \cdot r(c)
\end{aligned}
$$

- Solution:

$r(a) = 2$
$r(b) = 1$
$r(c) = 2$



Steady-State Schedules

- Repetition vector dictates the number of occurrences & length of schedule
- Exponential number of schedules |S| for steady-state

$$L = \sum_{u \in V} r(u)$$

$$|S| = \frac{L!}{\prod_{u \in V} r(u)!}$$

- ***Which schedule is beneficial for minimizing memory?***
  - Exhaustive search intractable
  - *State-of-the-art:* finds an ad-hoc solution

Optimal Schedule

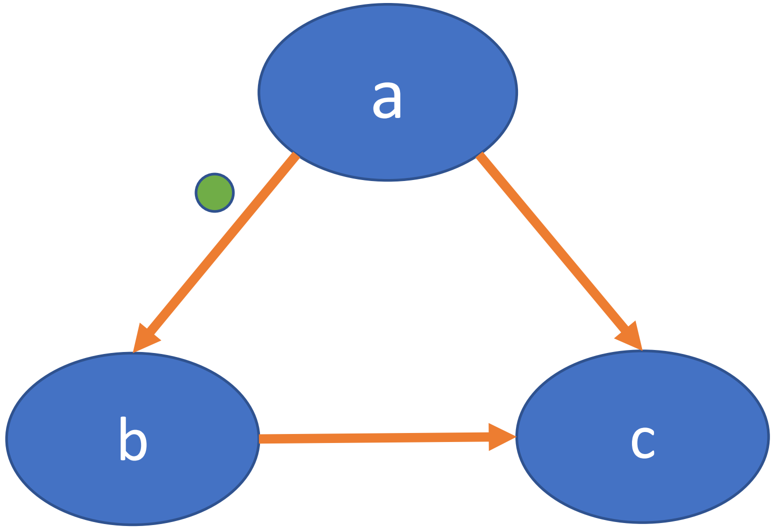
- Schedule

a	b	c	a	c

Maximum Tokens: 1

Max fill-state:

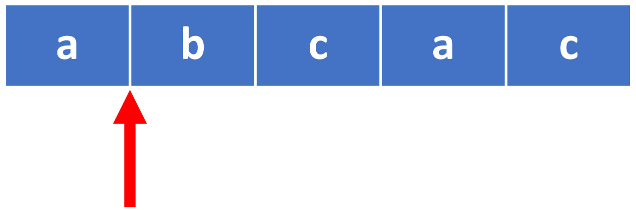
- (a,b)➔ 1
- (a,c)➔ 0
- (b,c)➔ 0



- Assume initial token on channel (a,b)

Optimal Schedule

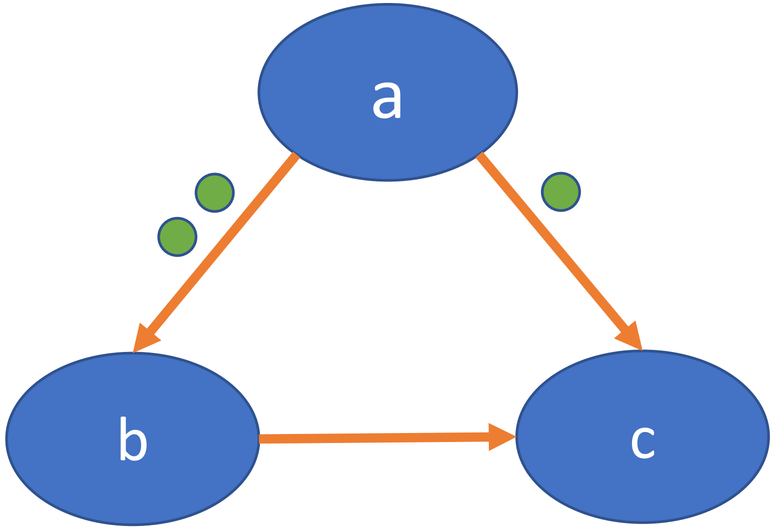
- Schedule



Maximum Tokens: 3

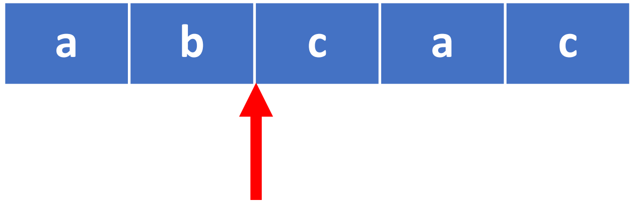
Max fill-state:

- (a,b)→ 2
- (a,c)→ 1
- (b,c)→ 0



Optimal Schedule

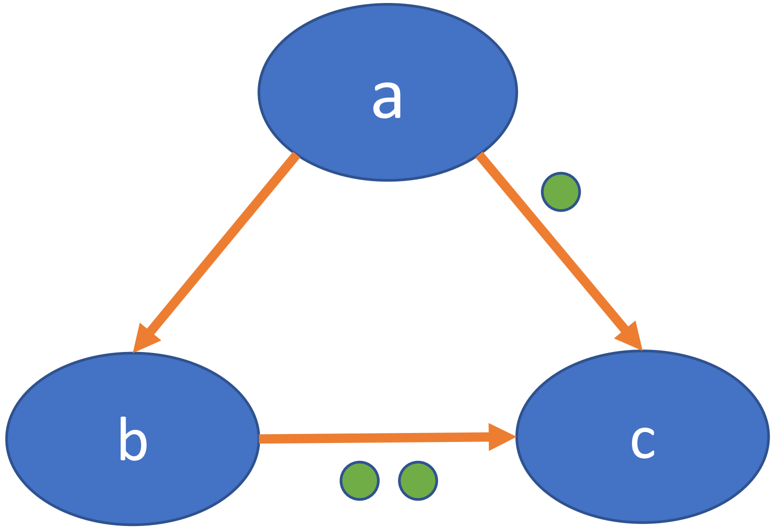
- Schedule



Maximum Tokens: 3

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 1
- (b,c)➔ 2



Optimal Schedule

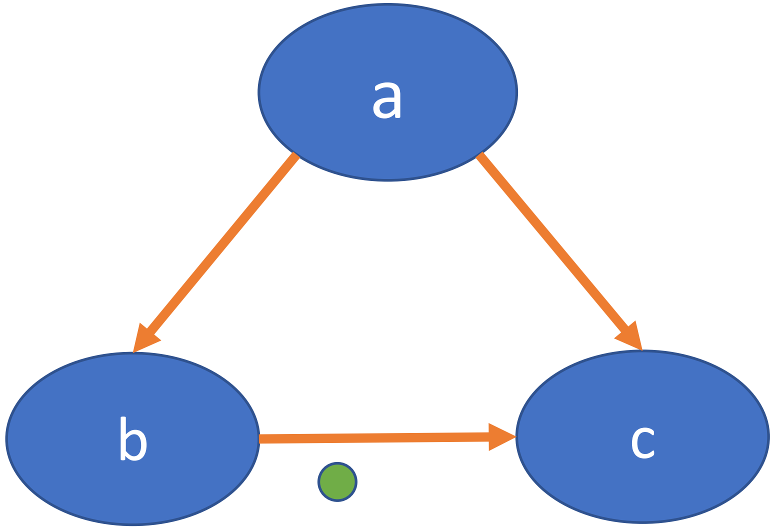
- Schedule

a	b	c	a	c

Maximum Tokens: 3

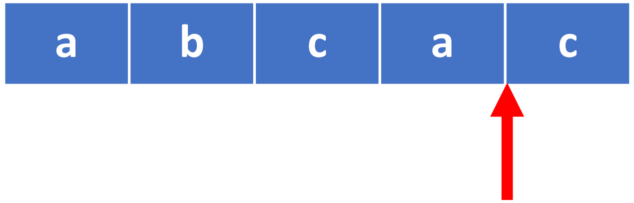
Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 1
- (b,c)➔ 2



Optimal Schedule

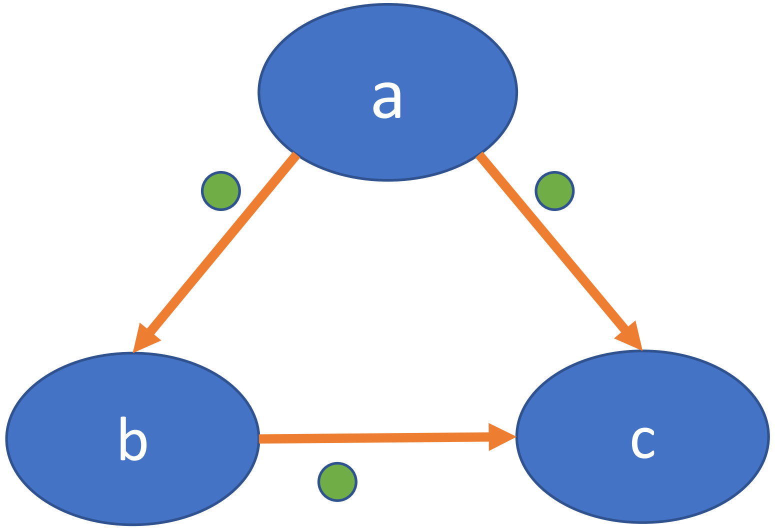
- Schedule



Maximum Tokens: 3

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 1
- (b,c)➔ 2



Optimal Schedule

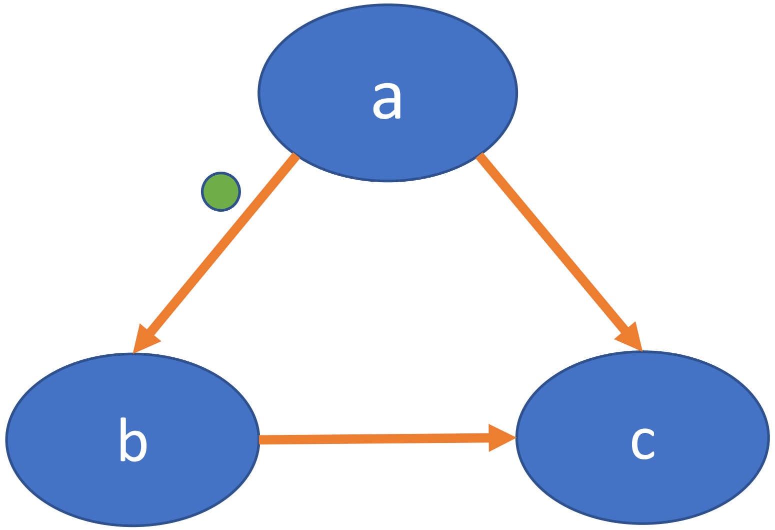
- Schedule

a	b	c	a	c

Maximum Tokens: 3

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 1
- (b,c)➔ 2



Greedy Algorithm

- Greedy algorithm developed by Battacharya, Murthy, and Lee
  - An actor is fireable if incoming edges have enough tokens to fire actor
  - An actor is deferrable if it is fireable and the consumers attached to its out-going edges are fireable
- Favor fireable and non-deferrable nodes to minimize buffer sizes
- If non-deferrable nodes do not exist, search for beneficial fireable node

Greedy Algorithm

Algorithm 1 GREEDY$((V, E, p, c), t)$

$L \leftarrow \sum_{u \in V} r(u)$
let F be the set of fireable actors in V using fill-state t
let D be the set of deferrable actors in F
for $i = 1$ to L **do**
    if $F \setminus D \neq \varnothing$ **then**
        $u \leftarrow$ an actor from $F \setminus D$
    else
        $u \leftarrow$ an actor in F that increases total number of tokens the least
    add u to the schedule s
    $r(u) \leftarrow r(u) - 1$
    invoke actor u
    update F and D // An actor u is not fireable if $r(u) < 1$.
return s

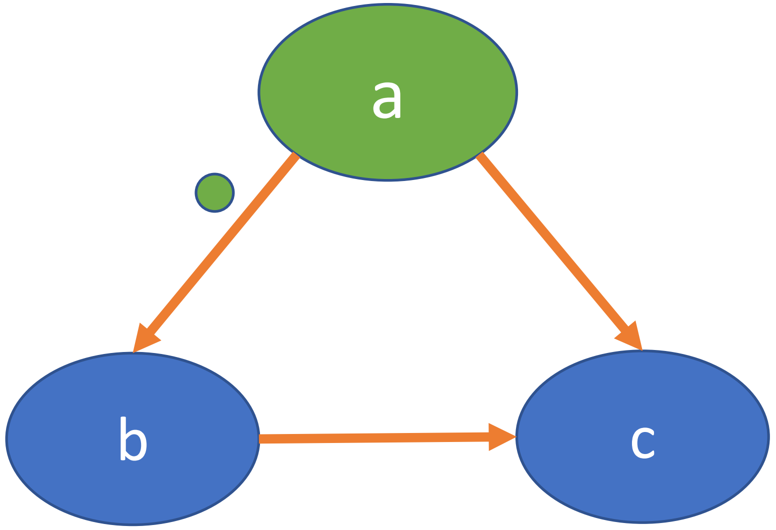
Greedy Schedule

- Schedule

Maximum Tokens: 1

Max fill-state:

- (a,b)➔ 1
- (a,c)➔ 0
- (b,c)➔ 0



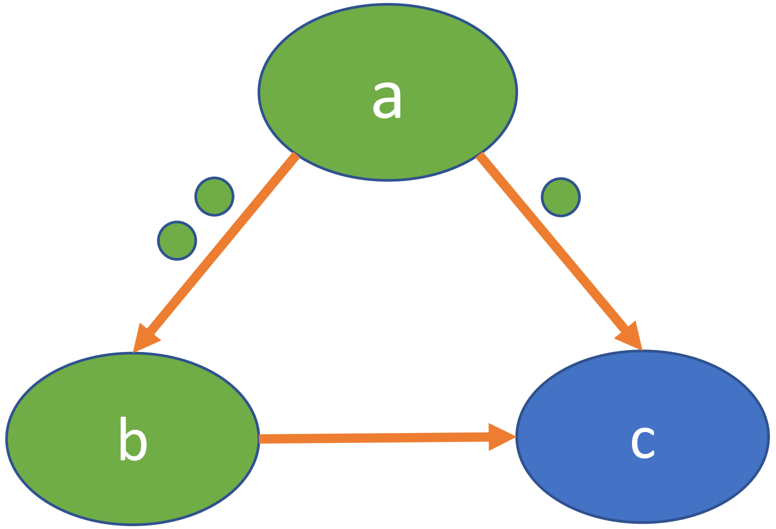
Greedy Schedule

- Schedule

Maximum Tokens: 3

Max fill-state:

- (a,b)→ 2
- (a,c)→ 1
- (b,c)→ 0



Greedy Schedule

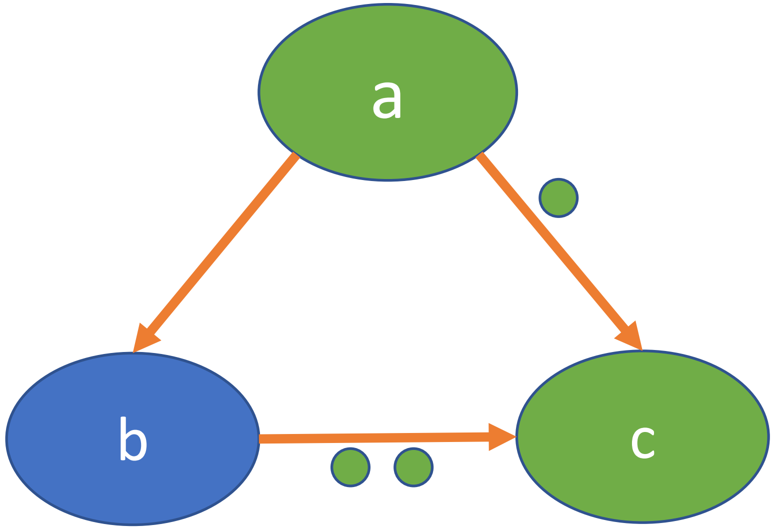
- Schedule

a	b	a	c	c

Maximum Tokens: 3

Max fill-state:

- (a,b)→ 2
- (a,c)→ 1
- (b,c)→ 2



Greedy Schedule

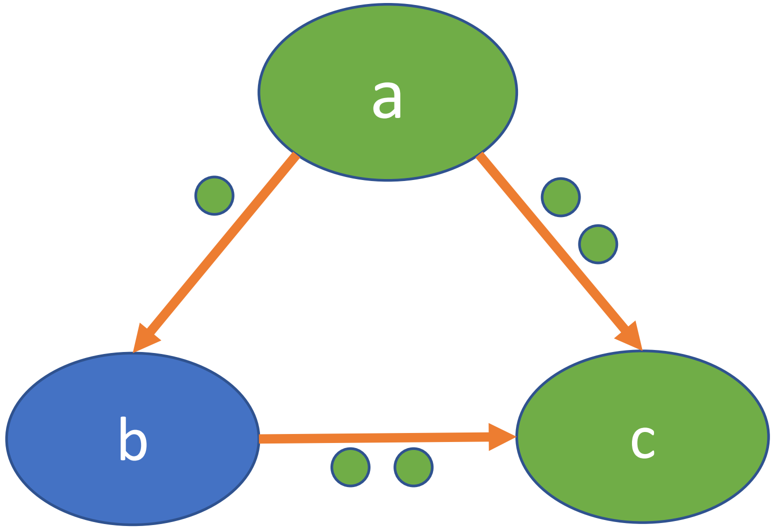
- Schedule

a	b	a	c	c

Maximum Tokens: 5

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 2
- (b,c)➔ 2



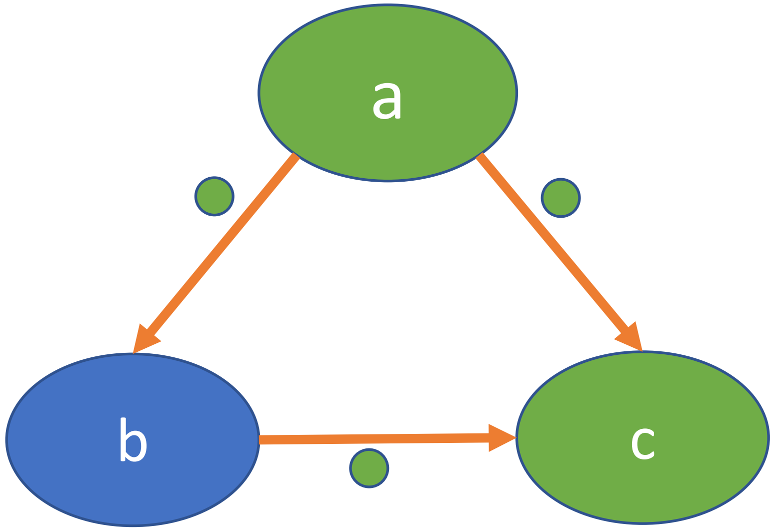
Greedy Schedule

- Schedule

Maximum Tokens: 5

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 2
- (b,c)➔ 2



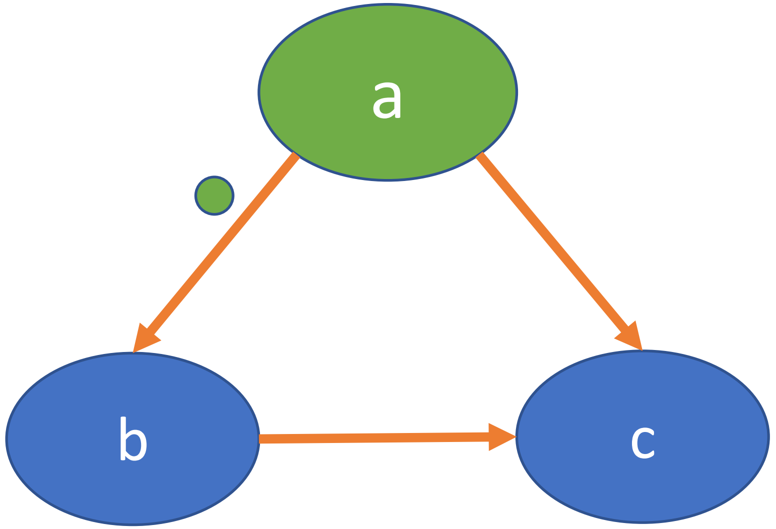
Greedy Schedule

- Schedule

Maximum Tokens: 3

Max fill-state:

- (a,b)➔ 2
- (a,c)➔ 2
- (b,c)➔ 2



Fill-State of Channels

- Define a fill-state function for channels (u,v)
- Tracks fill-state of data-channel for a schedule

$$f_s^{i+1}(u,v) = \begin{cases} f_s^i(u,v) + p(u,v), & \text{if } u = s(i+1), \\ f_s^i(u,v) - c(u,v), & \text{if } v = s(i+1), \\ f_s^i(u,v), & \text{otherwise.} \end{cases}$$

where s is the schedule, i is the step in the schedule.

- Initial fill-state is known as delay.

Problem Definitions with Fill-States

- Minimize Buffers

$$\textbf{(P1)} \quad \min_{(s,t)} \max_{0 \le I \le L} \max_{(u,v) \in E} f_s^I(u,v)$$

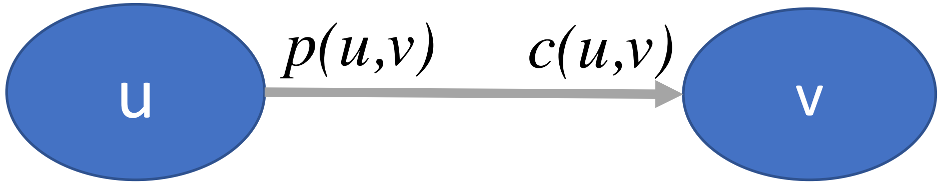
$$\textbf{(P2)} \quad \min_{(s,t)} \sum_{(u,v) \in E} \max_{0 \le I \le L} f_s^I(u,v)$$

$$\textbf{(P3)} \quad \min_{(s,t)} \max_{0 \le I \le L} \sum_{(u,v) \in E} f_s^I(u,v)$$

- (**P1**) <=> HW buffers; (**P2**) <=> static buffers; (**P3**) <=> dynamic buffers

Lower-Bound on Fill-State

- Assume production rate p, consumption rate c



- Lower-Bound for FIFO buffer size
  - $LB(u,v) = p(u,v) + c(u,v) + gcd(p(u,v), c(u,v))$
  - $\forall i$: $LB(u,v) \leq f^i(u,v)$
- Follows Euclid's argument (cf. Lemma 1)
- Lower-bound is maintainable as upper-bound (cf. Lemma 2)

Canonical Algorithm

- (**P1**) and (**P2**) permit an optimal algorithm in P
  - Requires an arbitrary total order for actors π
  - Compute initial fill-state for data-channels
  - Each actor has a priority x and is added to a priority queue
    - Initial priority of an actor is 0 and added to queue
    - Ties are broken with total order
    - New priority is $x' = x + [1/r(u)]$ where x is old priority of an actor
  - Use a priority queue in size of the number of actors
  - Break ties with a fixed order
- (**P3**) is a NP hard problem
  - Reduction uses the Minimum Feedback-Arc-set (FAS) Problem
  - See Appendix of paper

Canonical Algorithm

Algorithm 2 $\textsc{canonical}((V, E, p, c), \pi)$

for $(u, v) \in E$ **do**

  $t_\pi(u, v) \leftarrow \begin{cases} c(u, v) - \gcd(p(u, v), c(u, v)) & \text{if } \pi(u) < \pi(v) \\ c(u, v) & \text{if } \pi(v) < \pi(u) \end{cases}$

let Q be an empty priority queue

for $u \in V$ **do**

  `insert` u with priority 0 into Q

while true **do**

  $(u, x) \leftarrow$ `delete-min`(Q) // break ties using the π order

  execute actor u

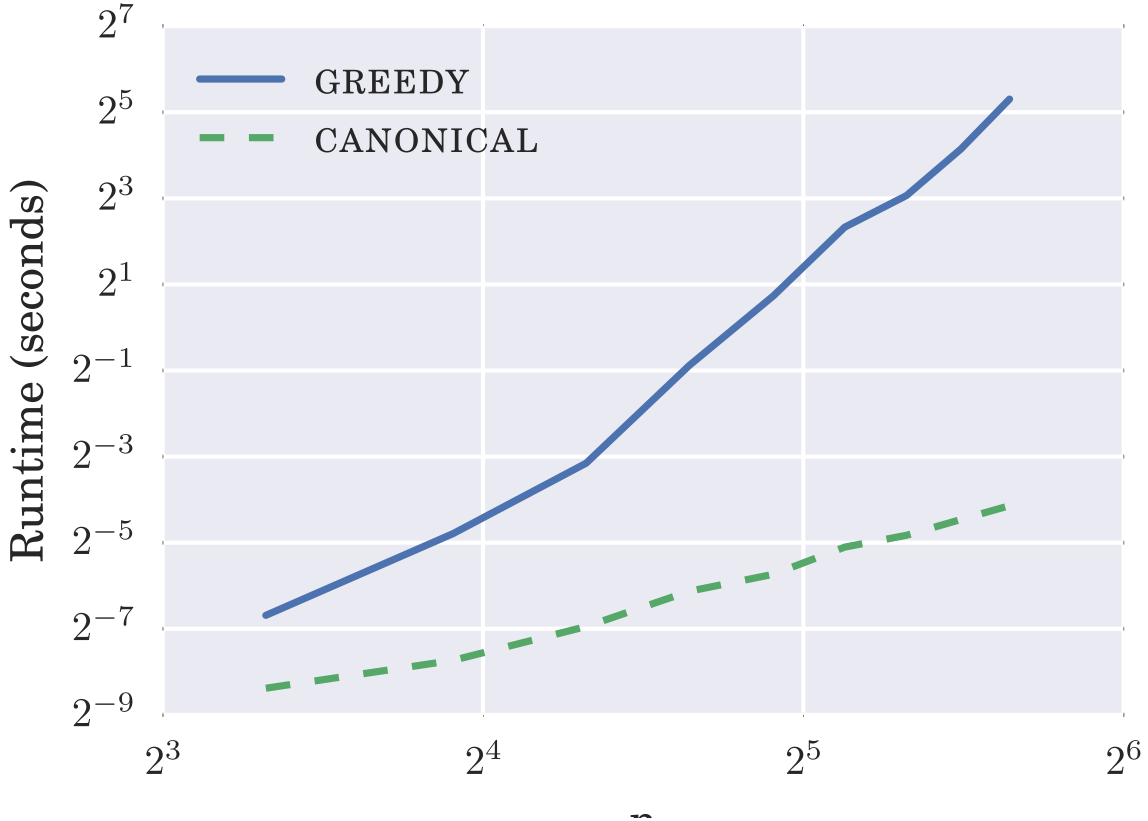
  `insert` u with priority $x + \frac{1}{r(u)}$ into Q

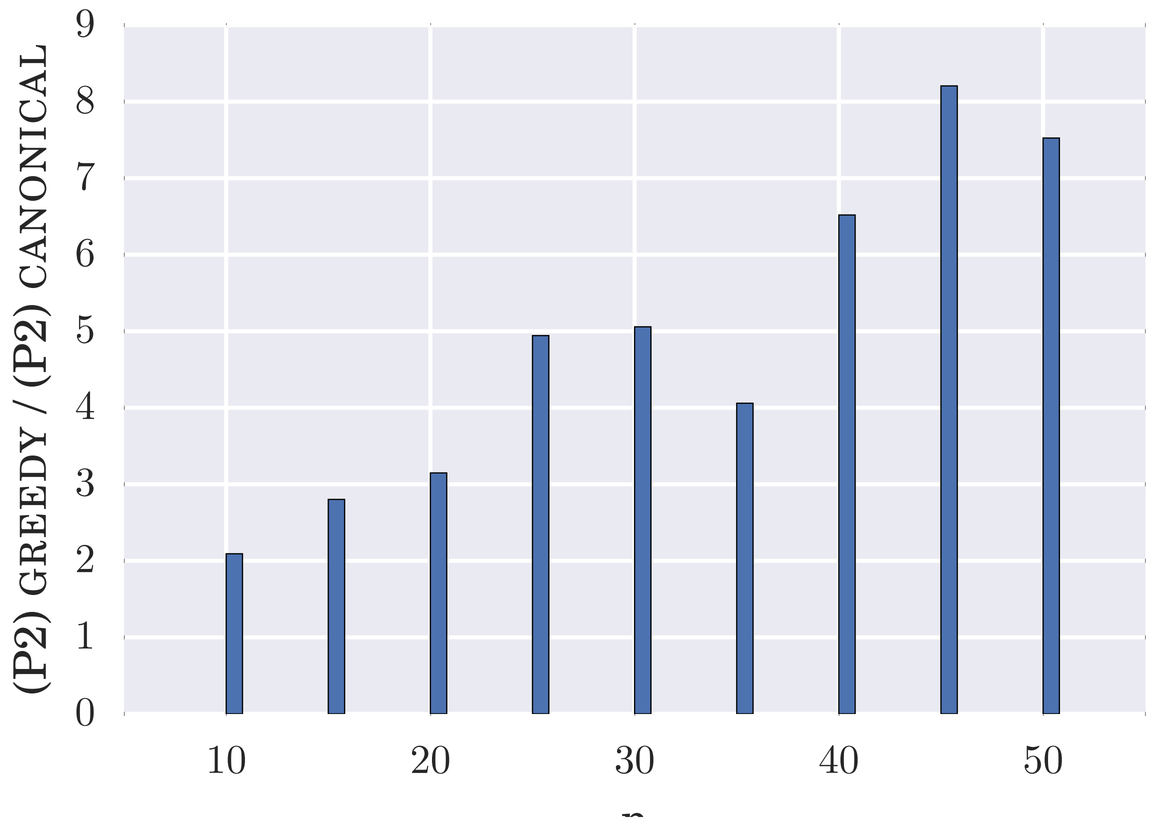
Experiments: Quality & Runtime

Instance	CANONICAL			GREEDY	
$\|V\|$	**(P1)**	**(P2)**	time (s)	**(P2)**	time (s)
10	20	586	0.0030	1226	0.0097
15	28	1048	0.0047	2937	0.0362
20	32	2483	0.0081	7818	0.1124
25	48	6131	0.0143	30306	0.5421
30	60	9486	0.0188	47979	1.6658
35	70	16782	0.0291	68126	5.0272
40	80	22927	0.0352	149469	8.3609
45	84	29781	0.0454	244380	17.6809
50	100	46203	0.0567	347676	39.5296

- Greedy
  - computation of fireable / deferrable takes too long.
  - uses more memory

Experiments





- Canonical: faster and better performance for random instances

Conclusion

- Minimizing buffer sizes for SDF programs is important
  - Impact on data caches
  - large data to process
  - increases with complex SDFs
- Three notions of memory optimality
  - Static FIFO buffers (in P)
  - Dynamic FIFO buffers (in NP)
  - Hardware FIFO buffers (in P)
- Introduce Canonical Algorithm using a Priority Queue
  - Show that lower-bound is maintainable as upper-bound
  - Faster than state-of-the-art algorithm
  - Optimality guarantees
  - Space complexity $O(n)$; run-time complexity $O(\log n)$ per actor invocation
- Provided some preliminary experimental results