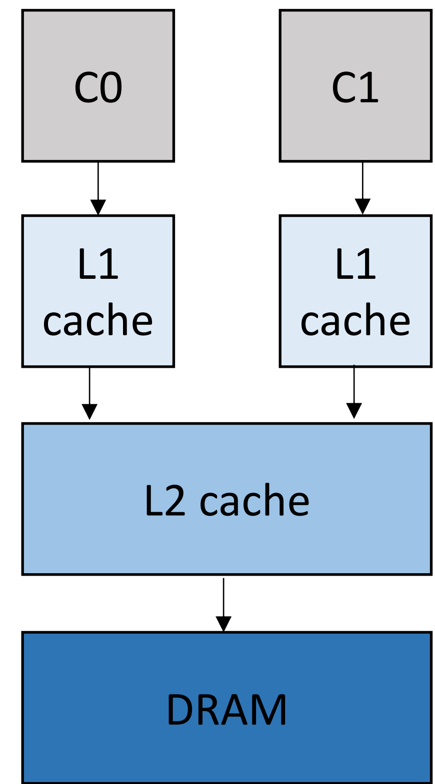
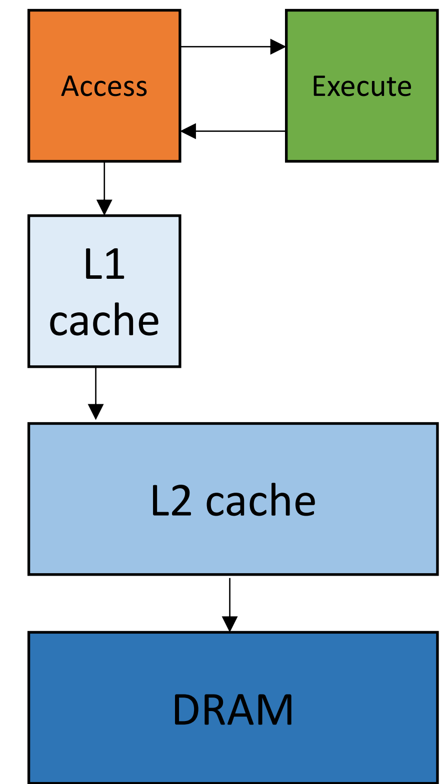
CSE211: Compiler Design

Dec. 3, 2020

- **Topic**: Continued DAE Performance Portability

- **Discussion questions**:
  - Thoughts on further optimizations for DAE systems?
  - How can we measure if a compiler optimization is worthwhile? On one machine? on many?





Announcements

- Midterm Graded; message me if you have any questions
- SETs are out. Please fill it out!
  - Personalized feedback is also welcome
- Projects set!
  - ½ day for each class next week
  - I will do an extended paper presentation for the rest of each class
- Final will be released Dec. 13 night and due Dec. 14 midnight AoE

Announcements

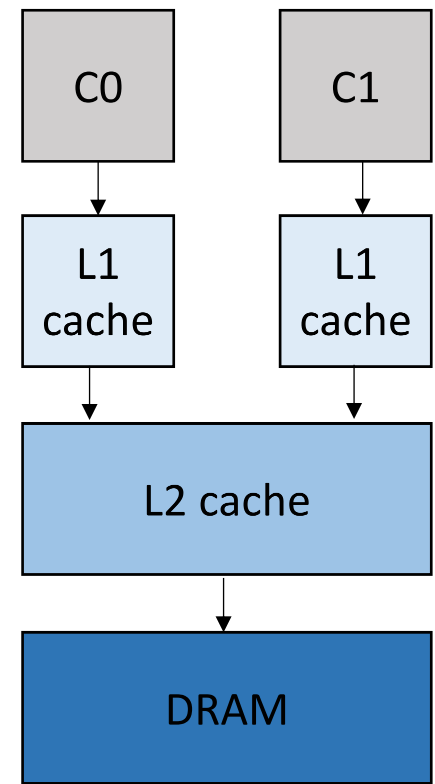
- Homework 4 will be uploaded by the end of today. Implementing parallel schedulers and simple DAE slicing.
  - Due day of the final: Dec. 14
- LSD Seminar on Friday!
  - Alexa is from the same research group as Dietrich (Adrian Sampson at Cornell)

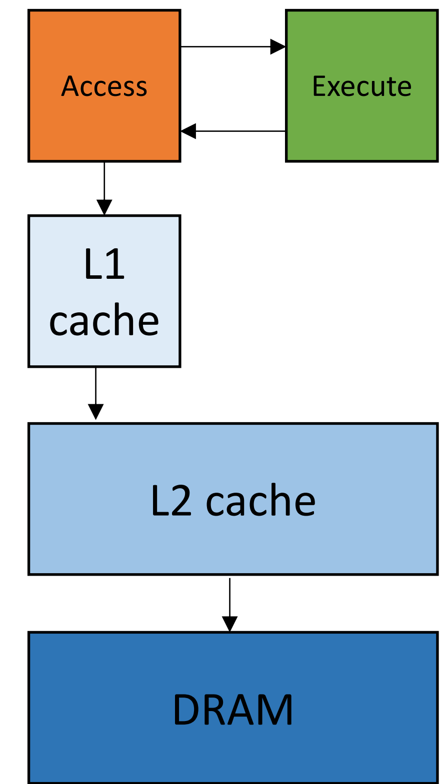
CSE211: Compiler Design

Dec. 3, 2020

- **Topic**: Continued DAE Performance Portability

- **Discussion questions**:
  - Thoughts on further optimizations for DAE systems?
  - How can we measure if a compiler optimization is worthwhile? On one machine? on many?



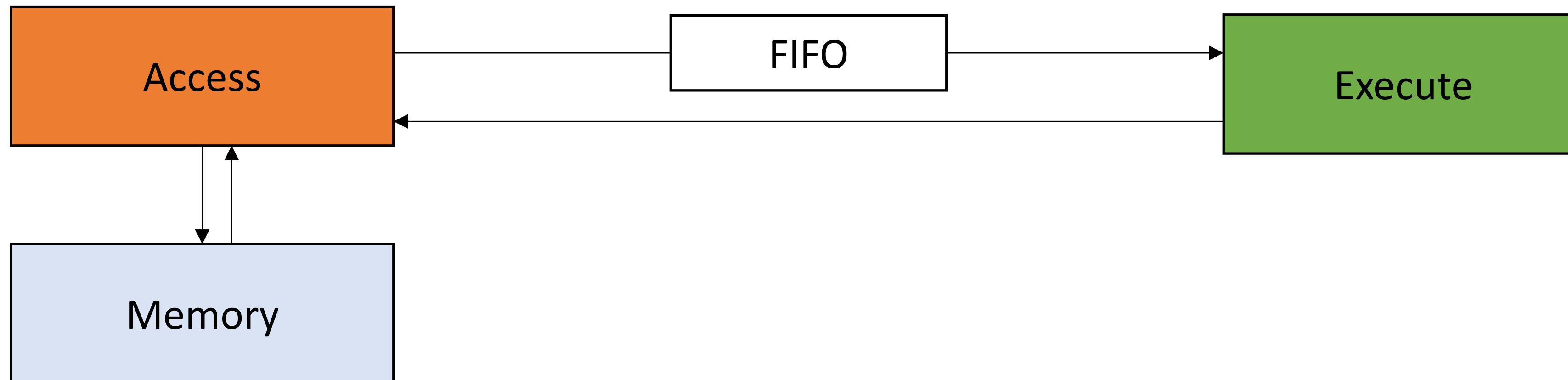


Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    FIFO_enqueue(r0);
    SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = FIFO_dequeue()
    float r1 = r0 * 3.14;
    SB_store_value(r1);
}
```

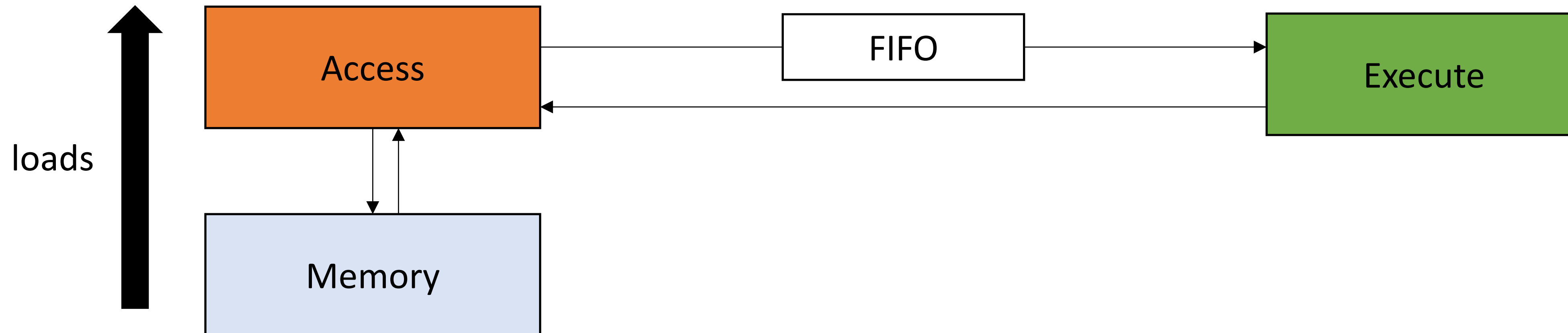


Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    FIFO_enqueue(r0);
    SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = FIFO_dequeue()
    float r1 = r0 * 3.14;
    SB_store_value(r1);
}
```

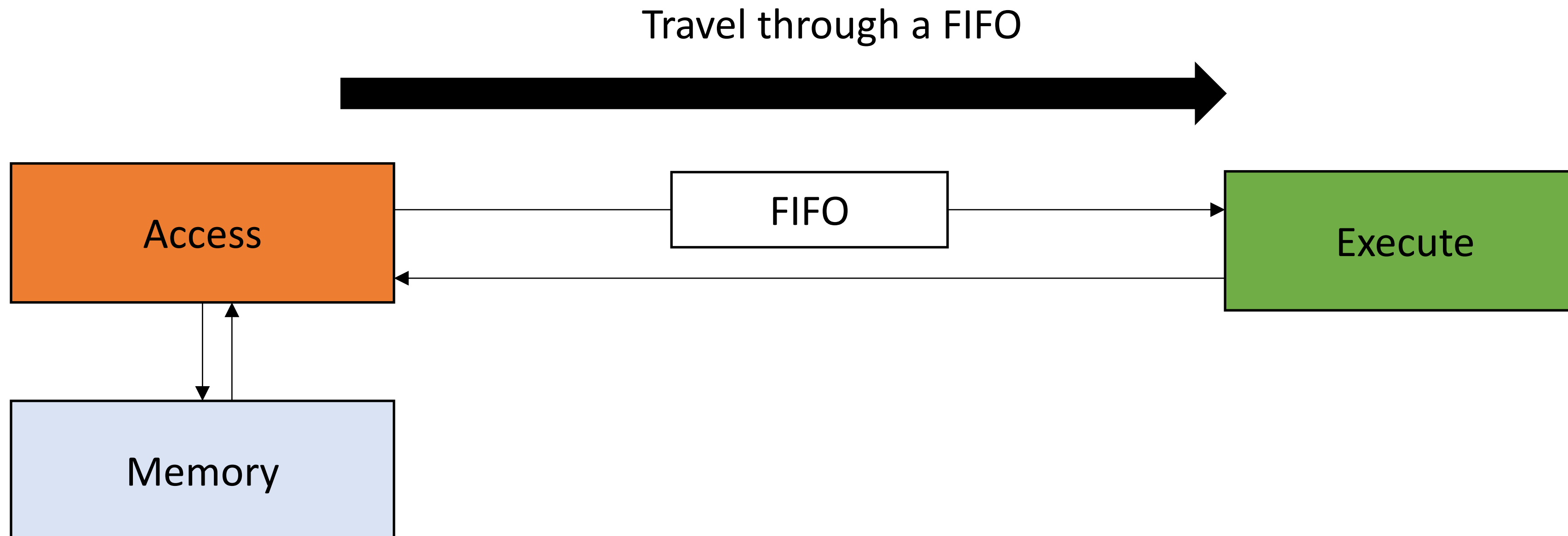


Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

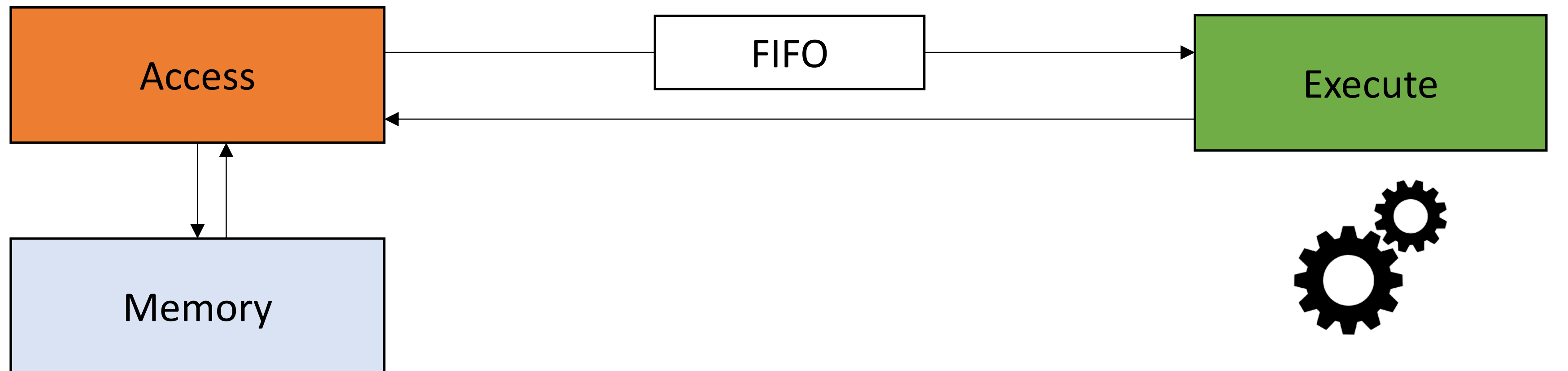


Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    FIFO_enqueue(r0);
    SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = FIFO_dequeue()
    float r1 = r0 * 3.14;
    SB_store_value(r1);
}
```



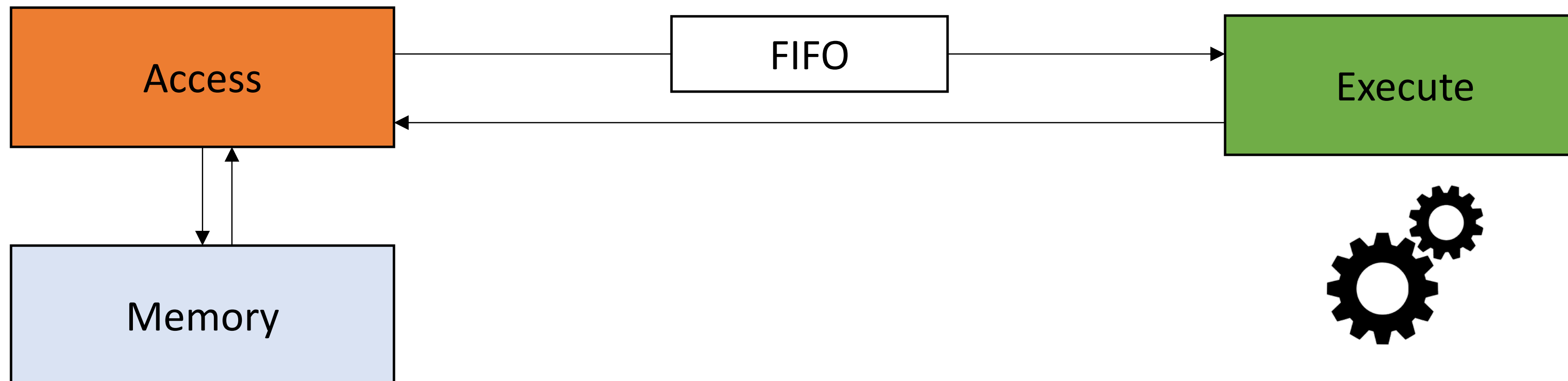
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

while computation is happening on
The execute, the access can "runahead"



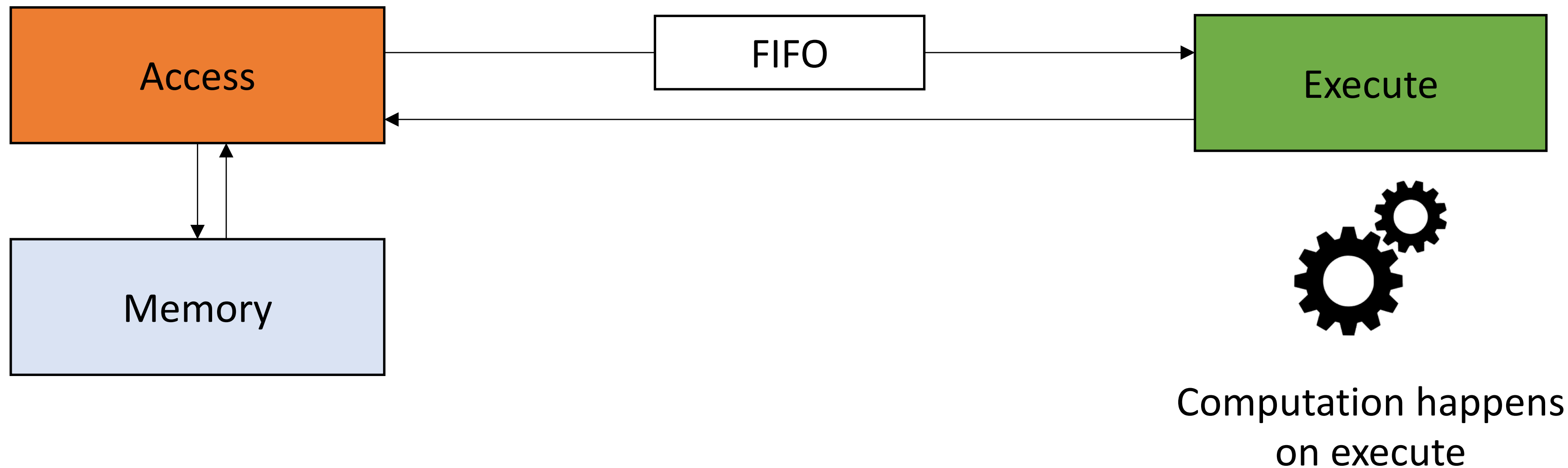
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

And fetch the value for the next iteration of the compute (i.e. the compute will have less latency).



Computation happens on execute

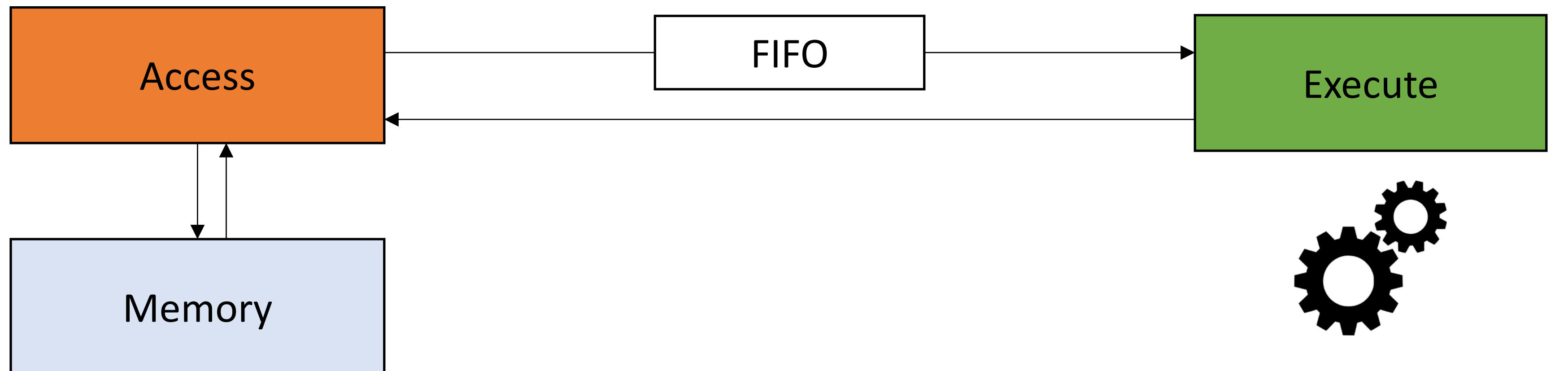
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

And fetch the value for the next iteration of the compute (i.e. the compute will have less latency).

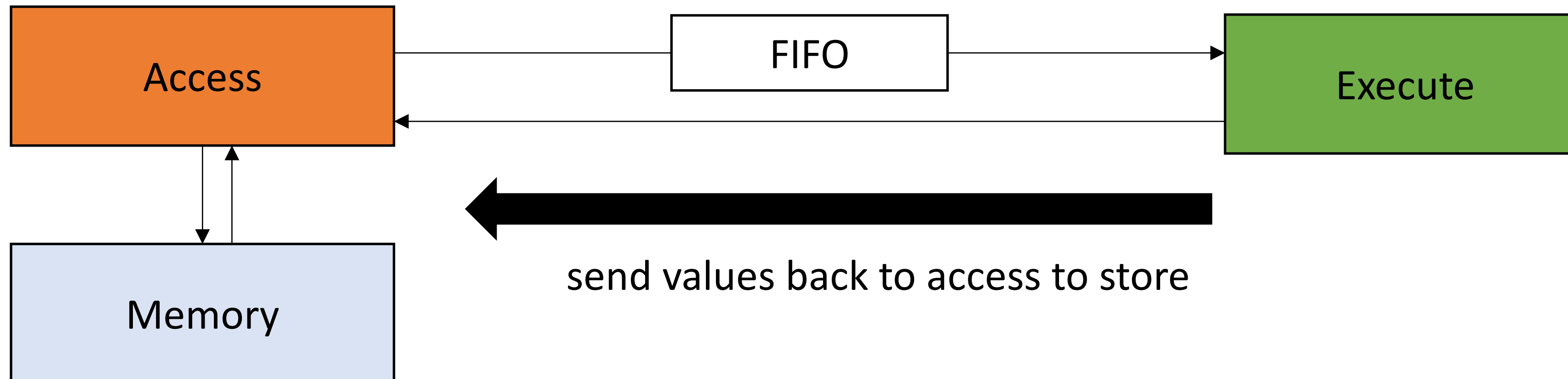


Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```



DAE key ideas

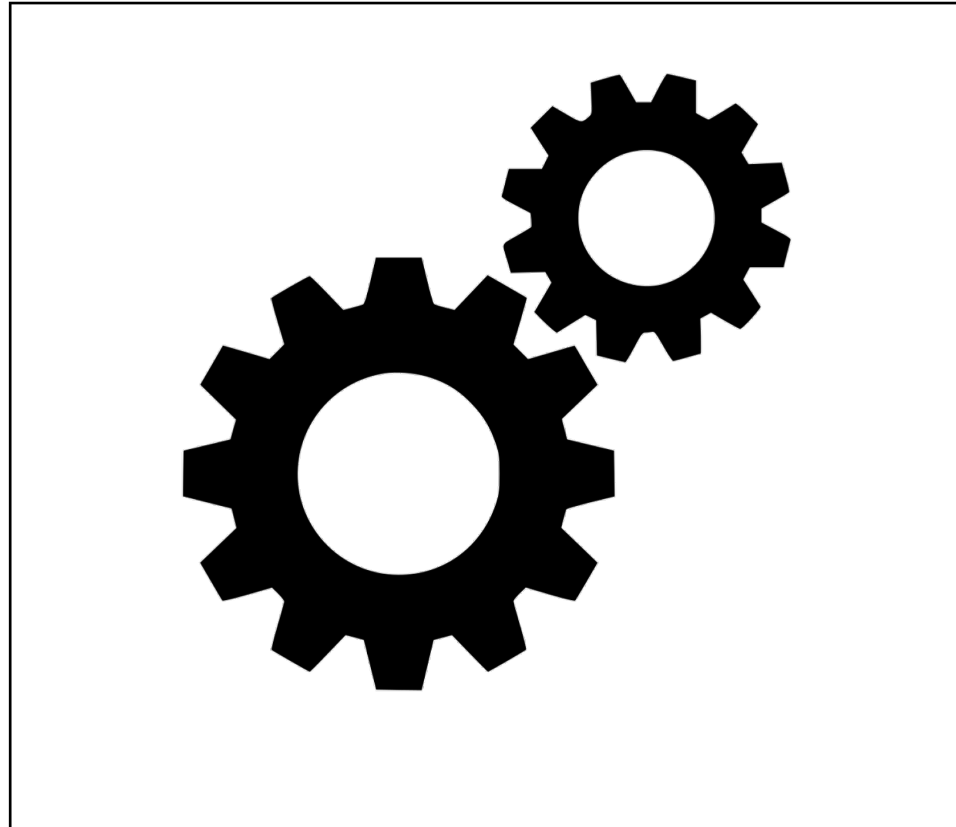
- Heterogeneous parallelism:
  - one slice does memory
  - the other does compute
- Explicit access to store buffer is key for access “runahead”
- Compiler copies and slices the program
- Good for overlapping memory and computation
- Pros/cons?

DeSC Terminal Loads: Key idea

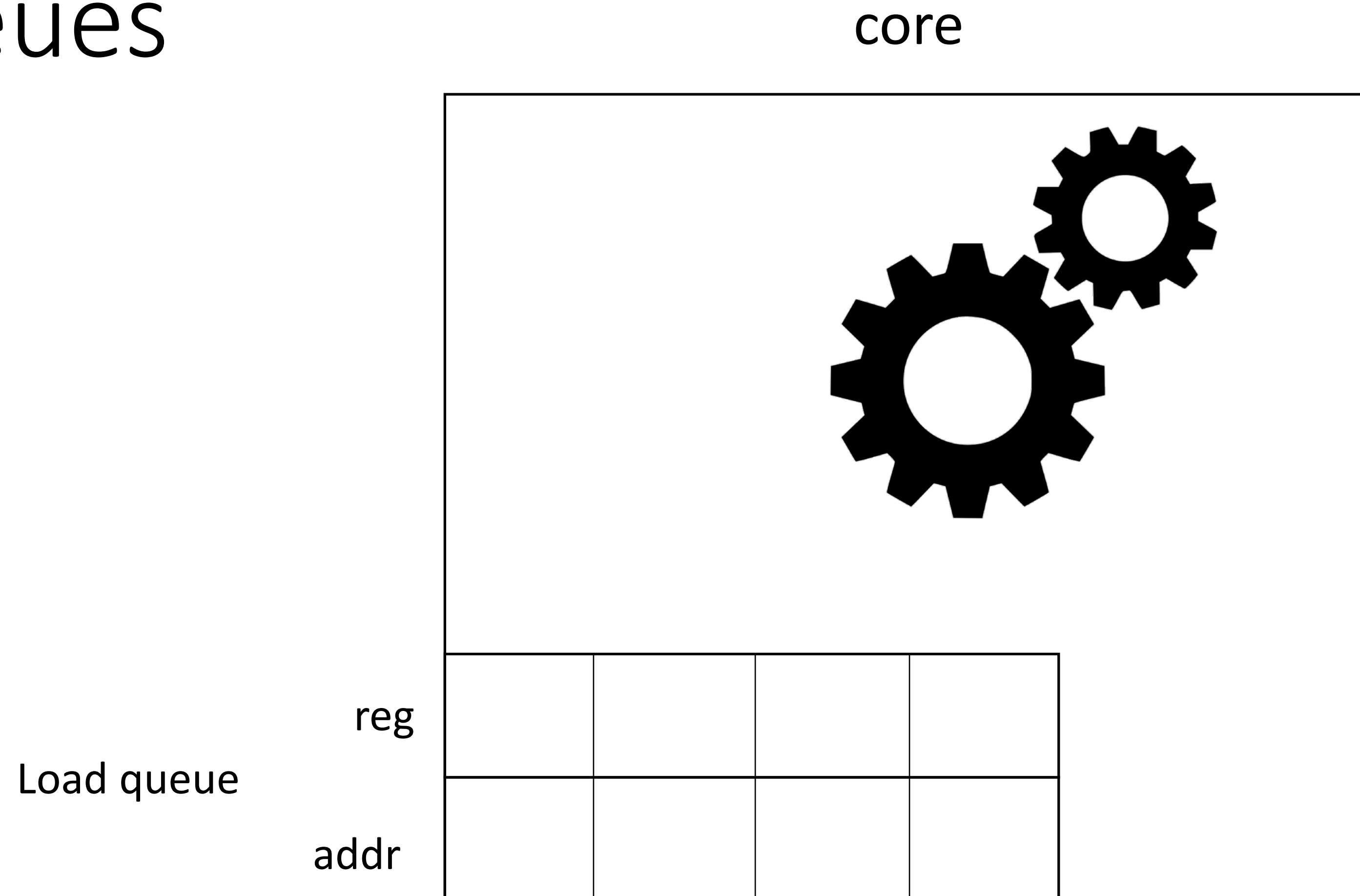
- 2015 variation of DAE
- Access could explicitly access a hardware component (SB) to perform stores asynchronously.
- Can we apply this to loads?

Load Queues

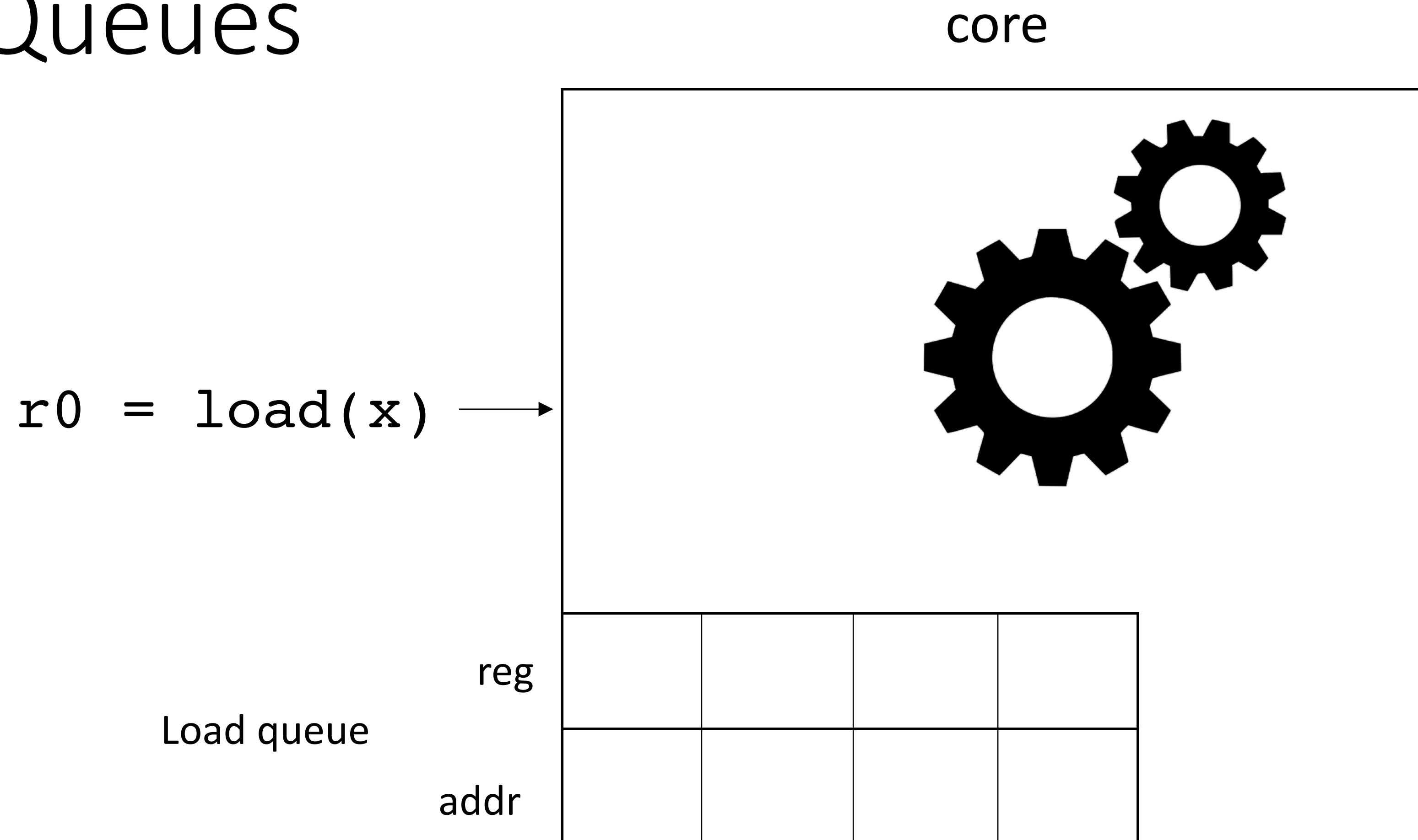
core



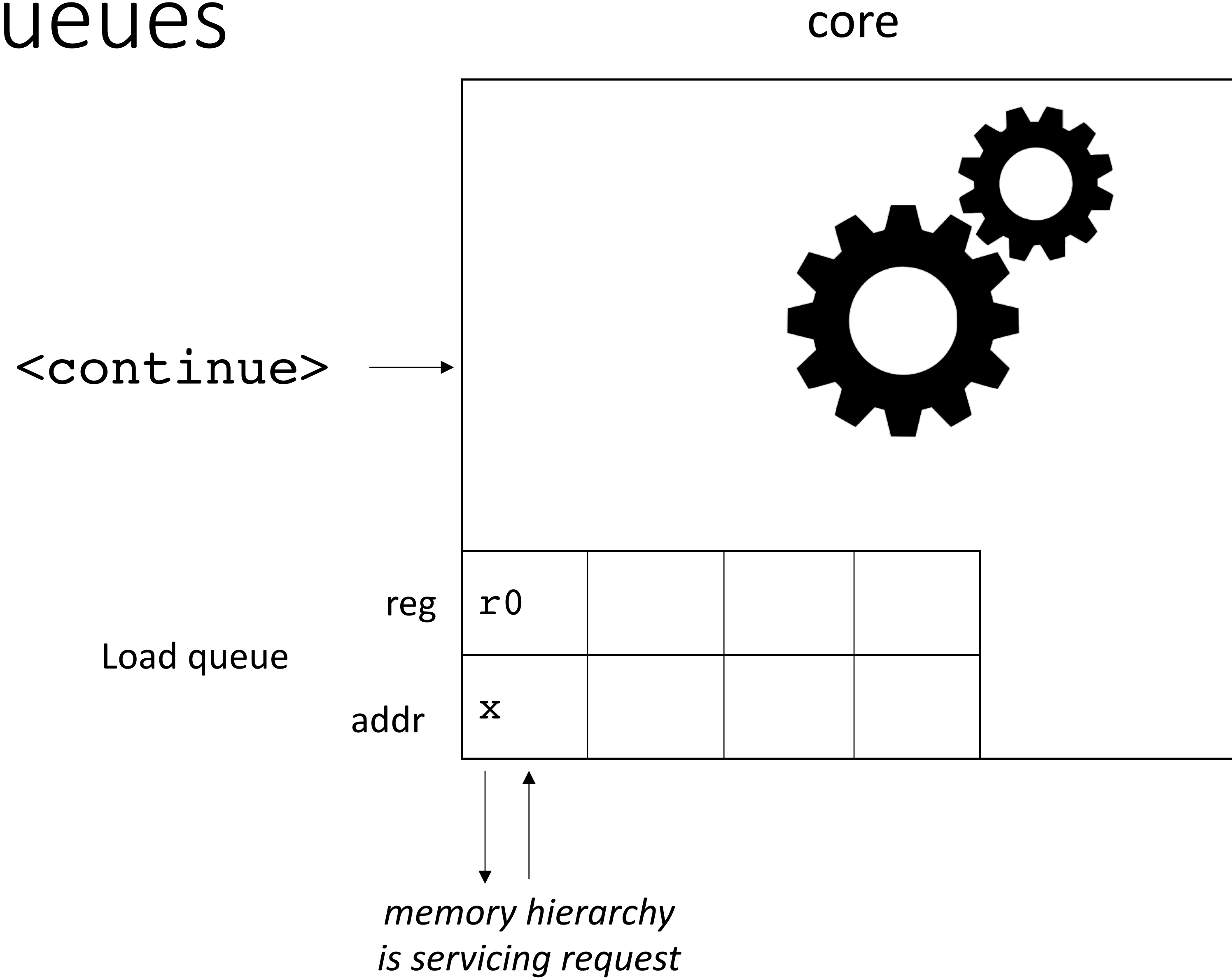
Load Queues



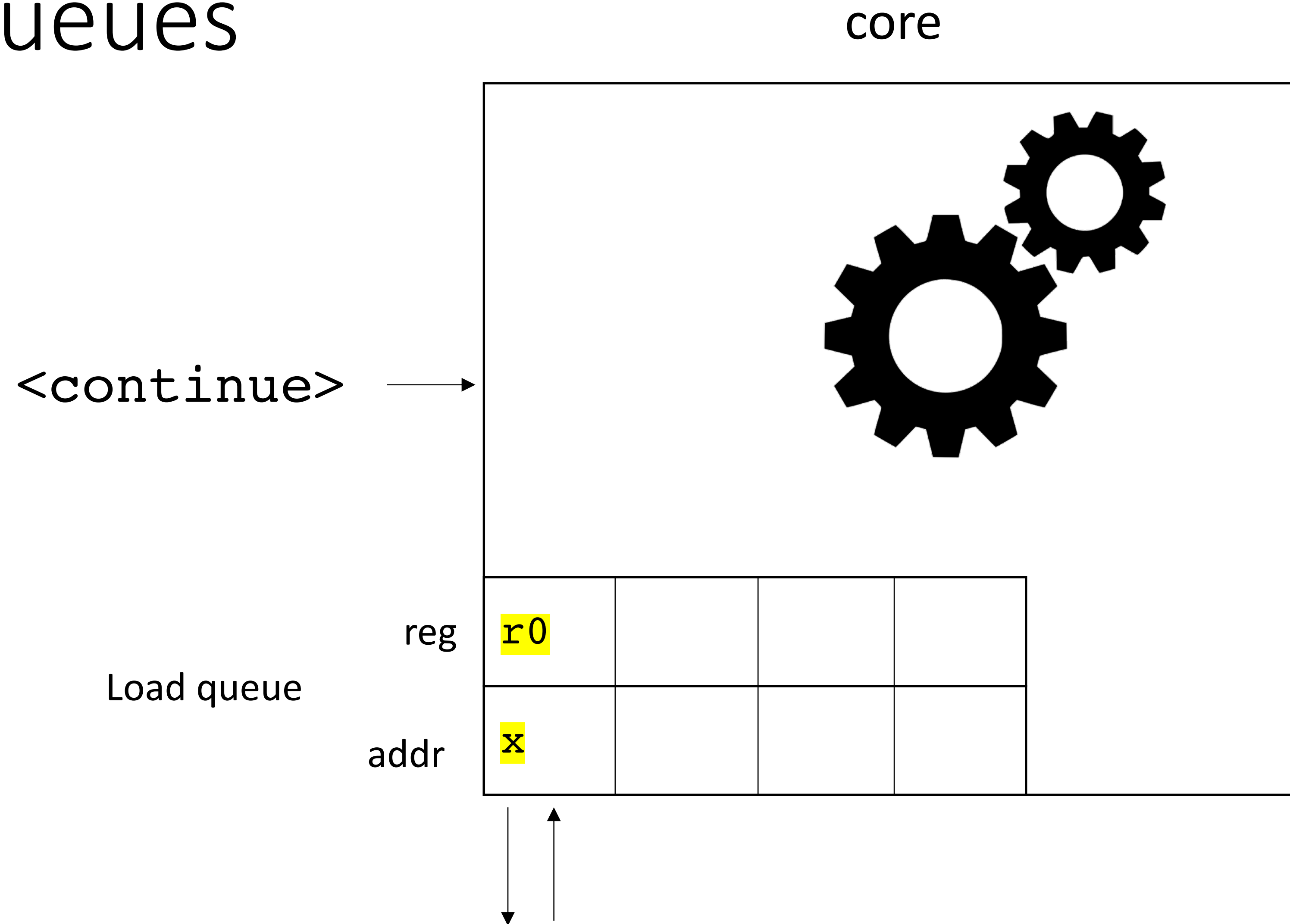
Load Queues



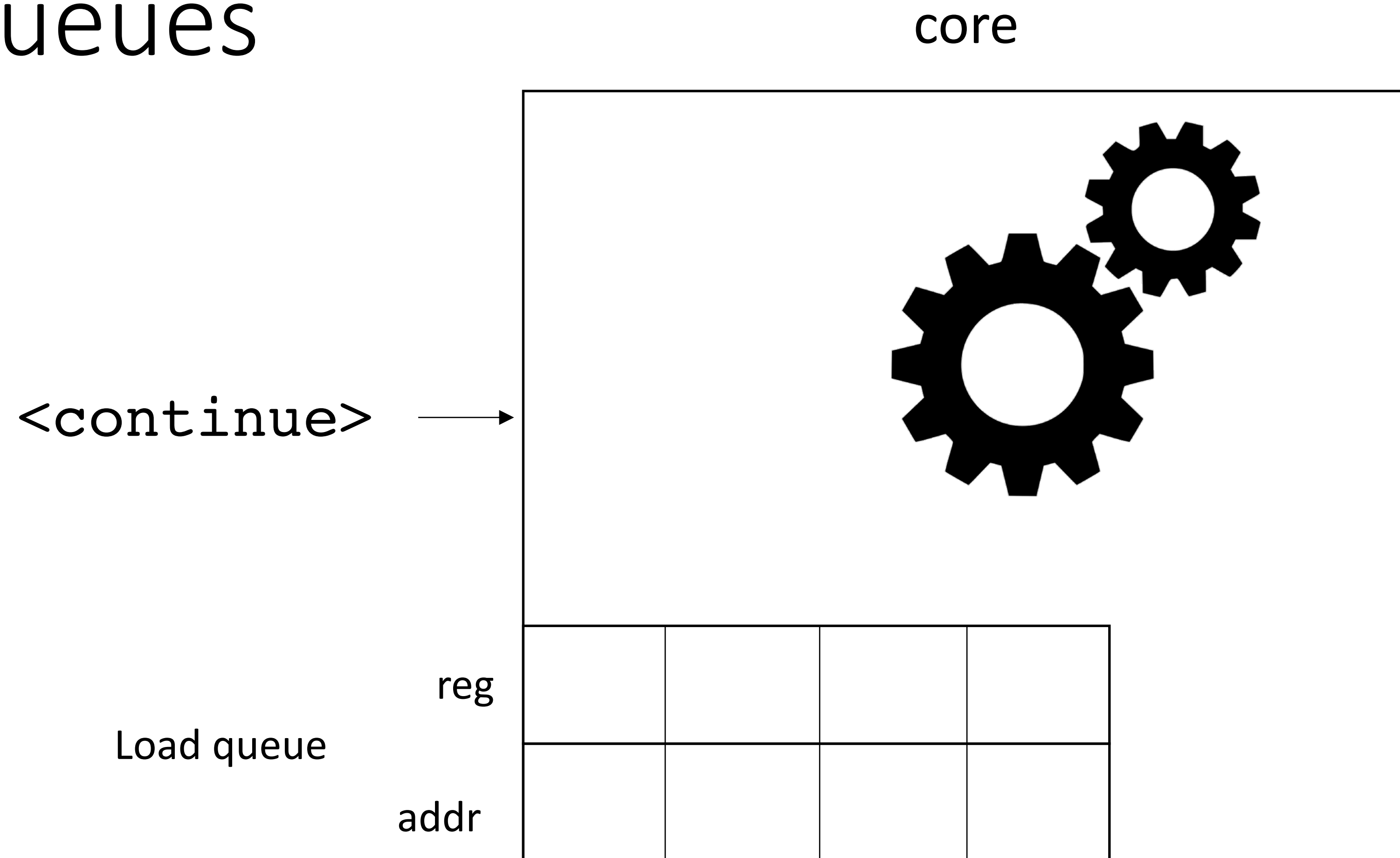
Load Queues



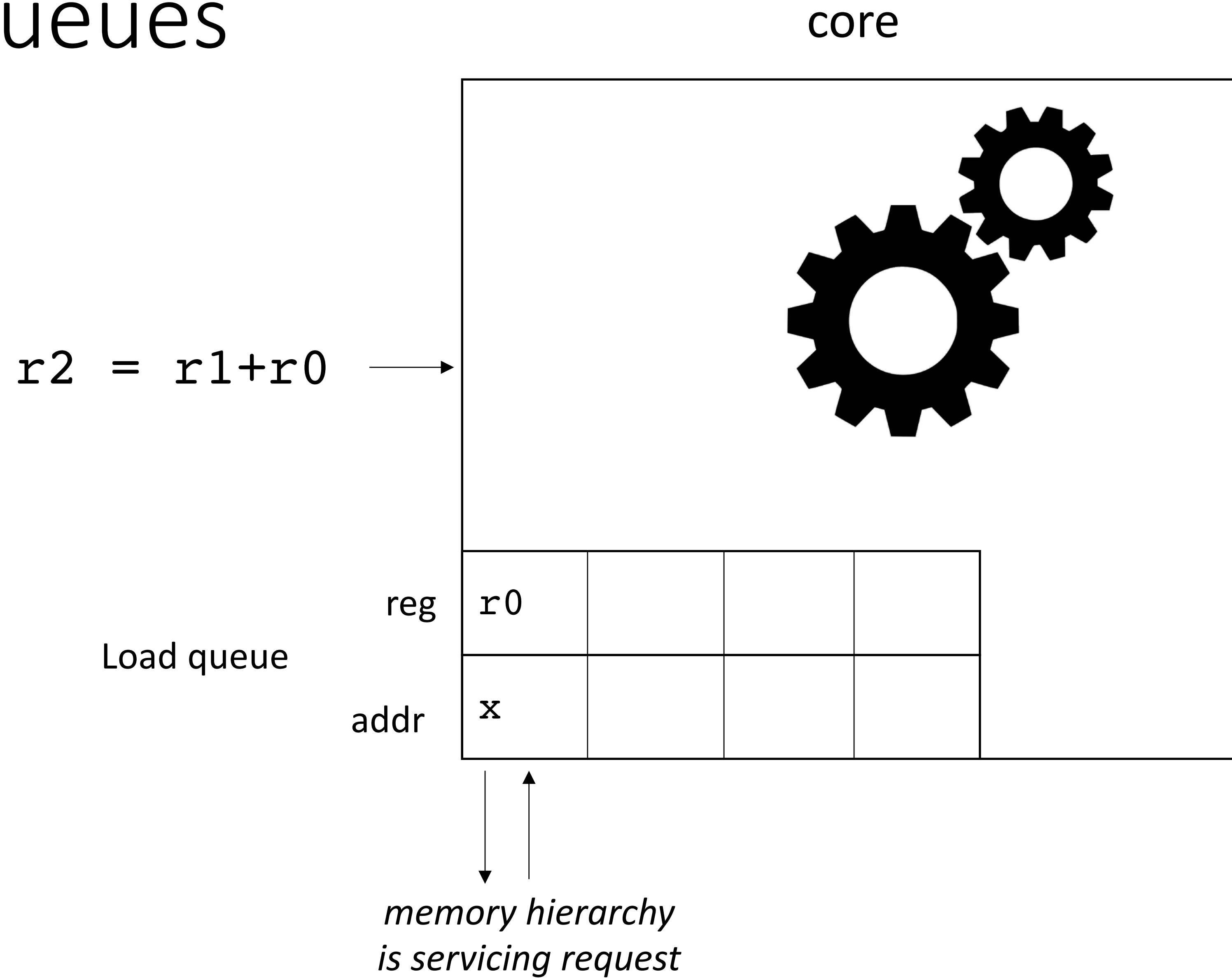
Load Queues



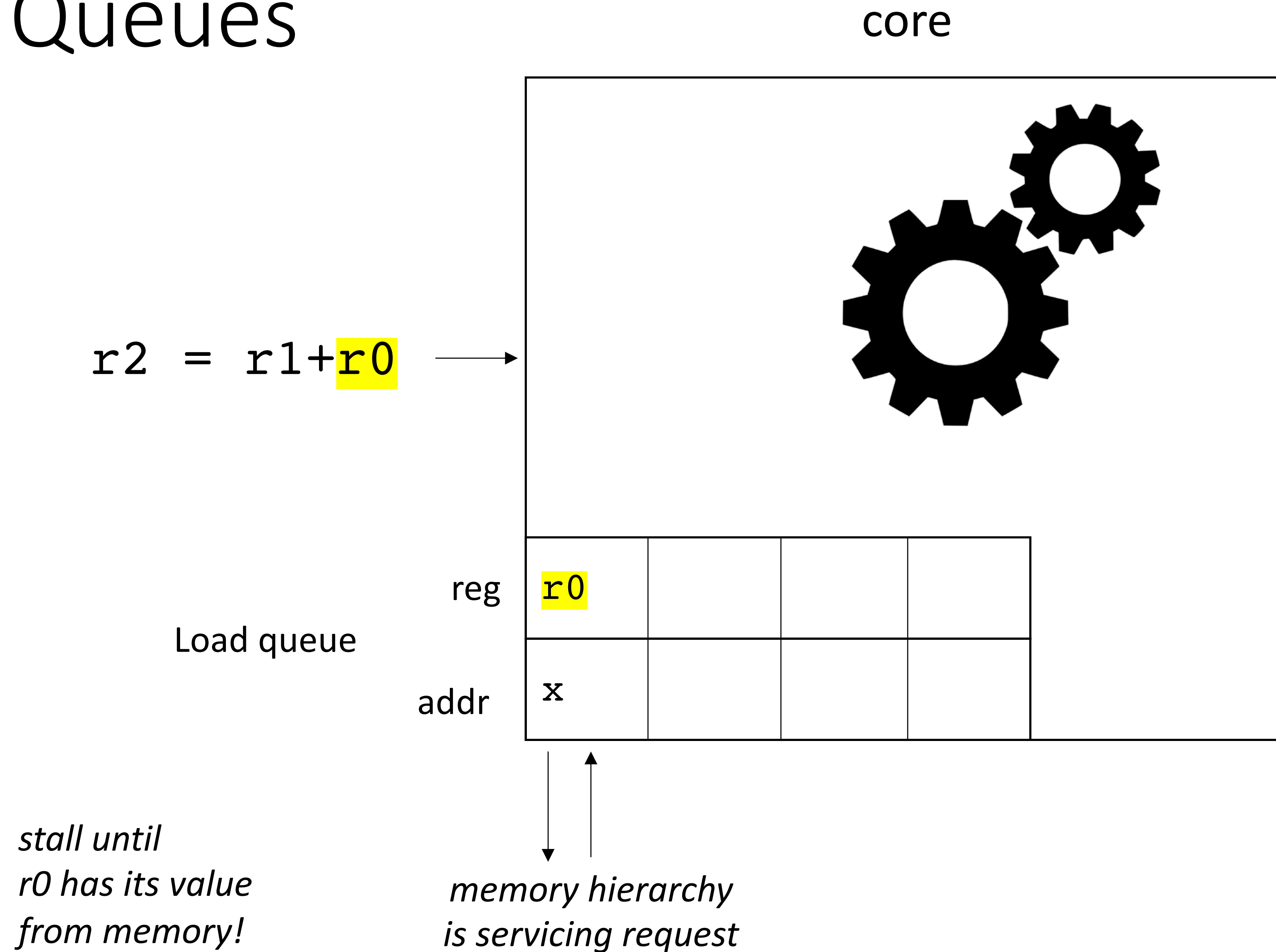
Load Queues



Load Queues



Load Queues



Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

The only use of r0 is to enqueue it to the Execute

LQ

reg	r0			
addr	b+0			

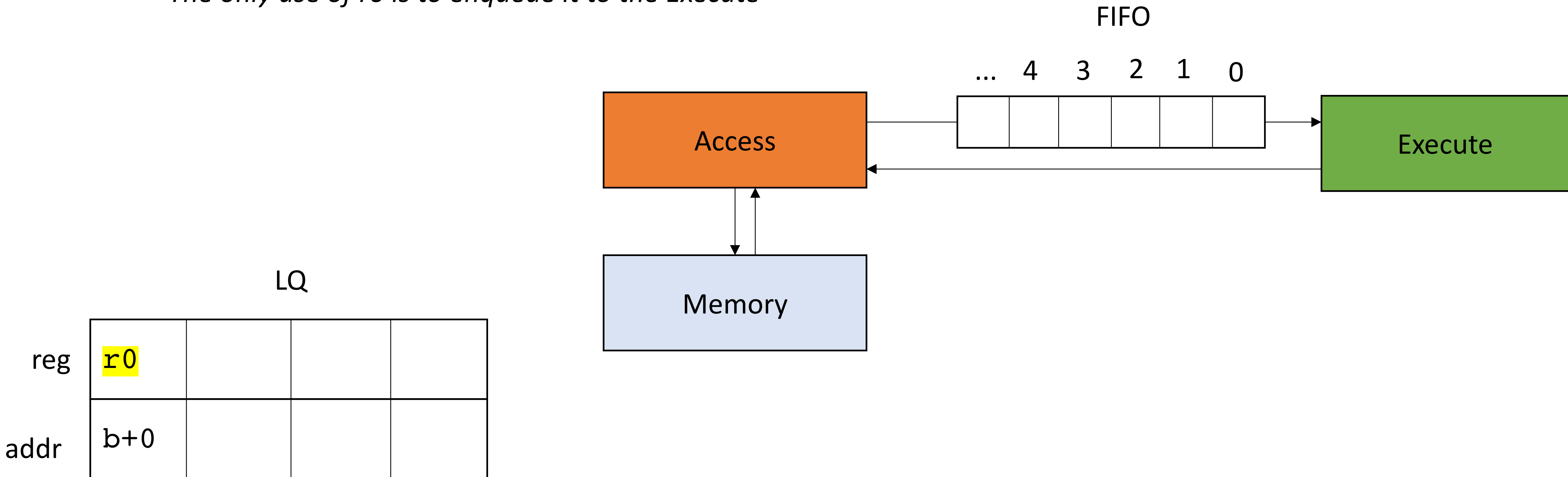
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    FIFO_enqueue(r0);
    SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = FIFO_dequeue()
    float r1 = r0 * 3.14;
    SB_store_value(r1);
}
```

The only use of r0 is to enqueue it to the Execute



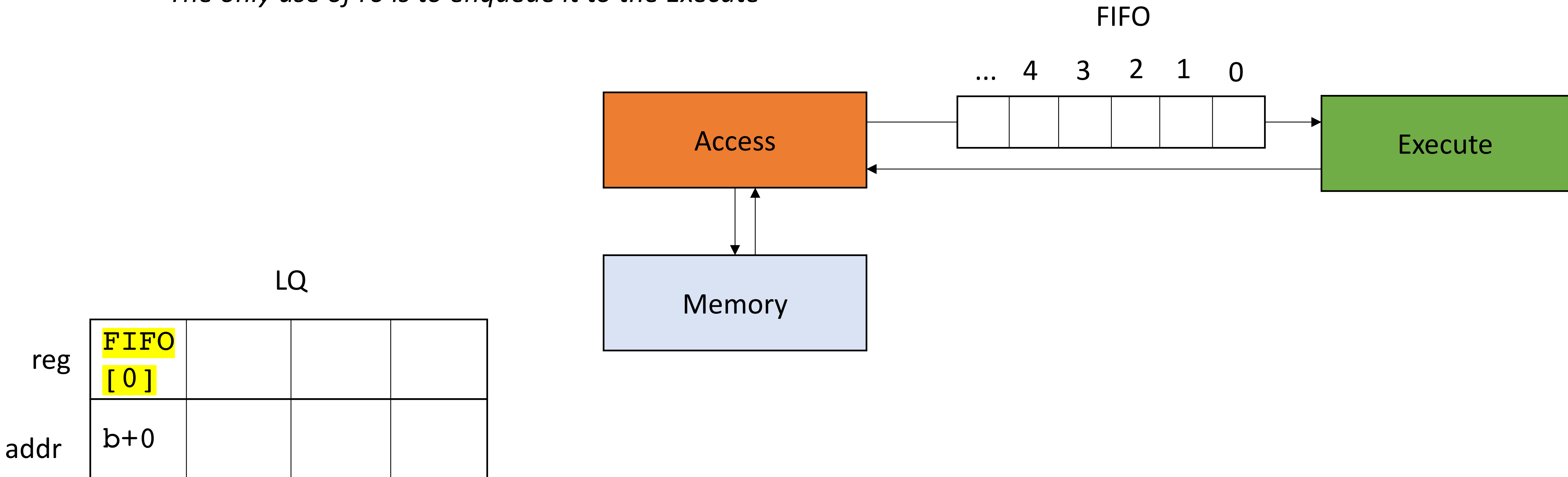
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

The only use of r0 is to enqueue it to the Execute



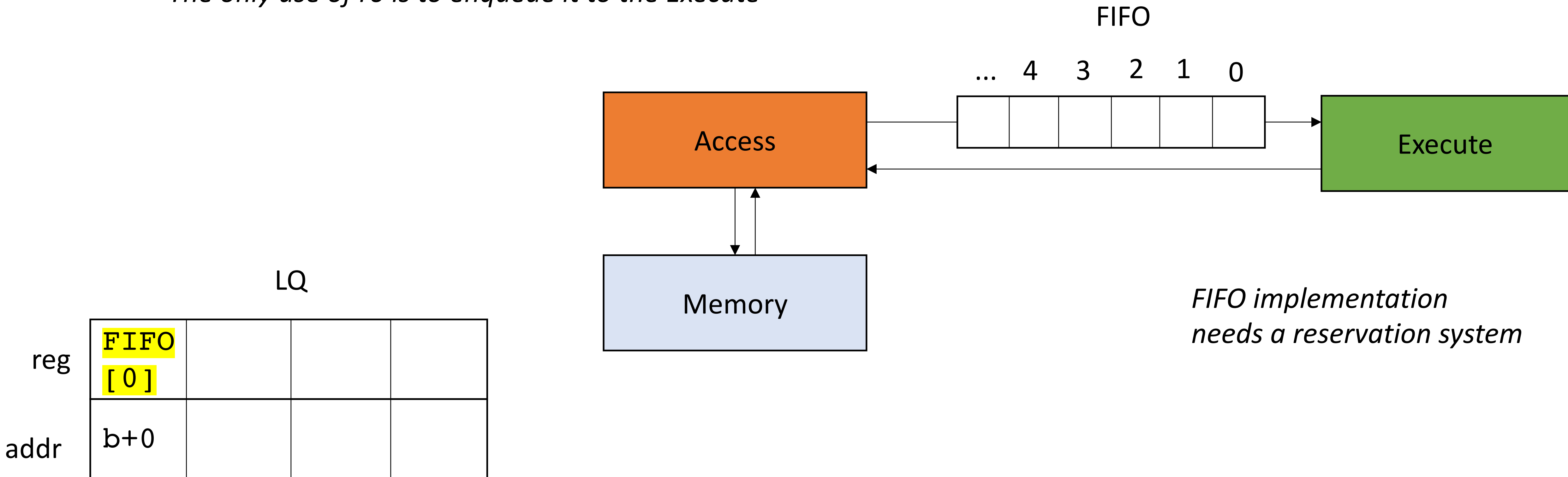
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

The only use of r0 is to enqueue it to the Execute



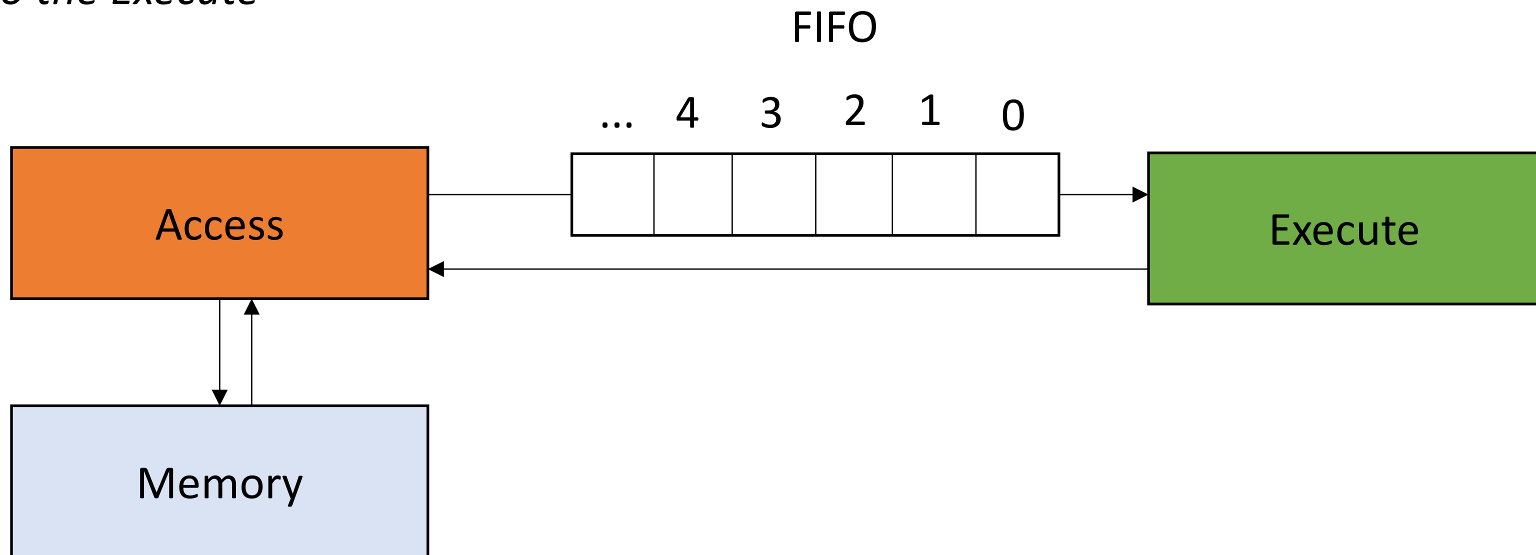
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

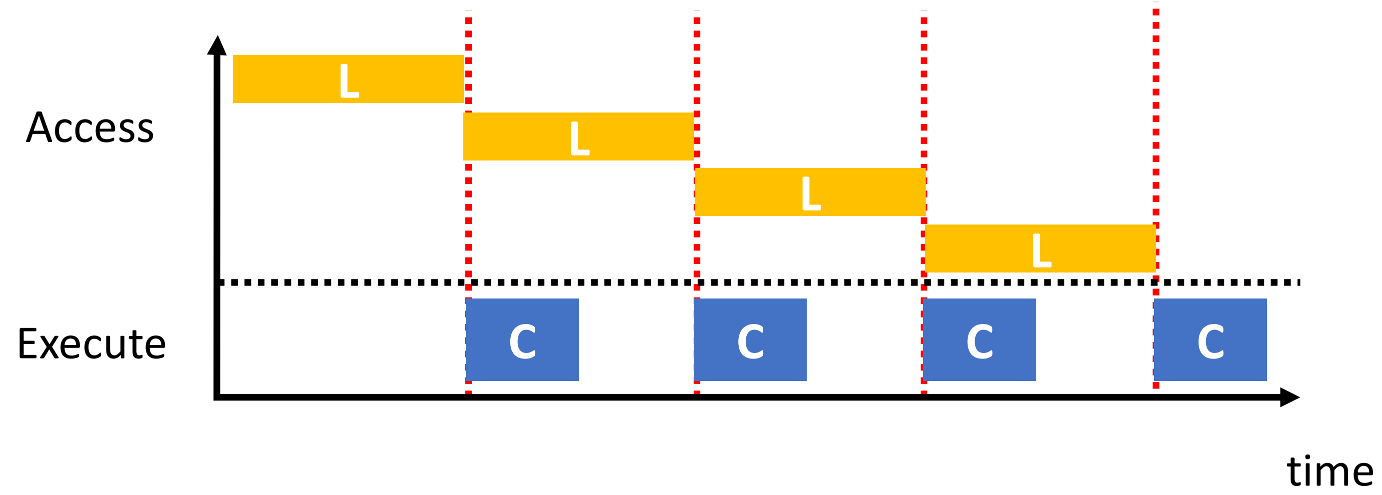
The only use of r0 is to enqueue it to the Execute



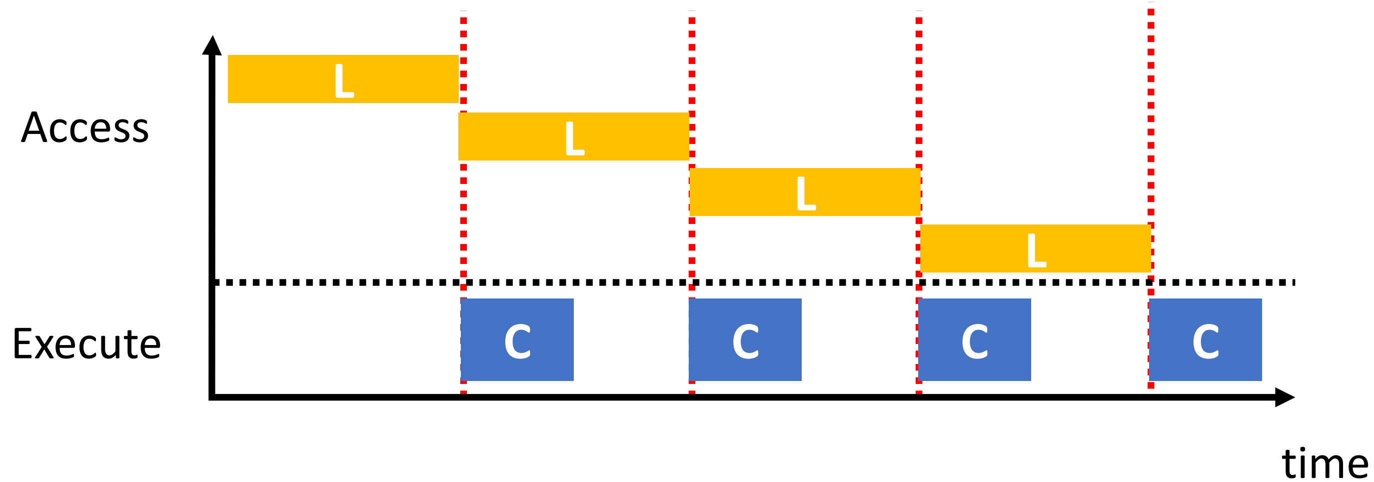
Many load requests in flight at the same time. Known as Memory Level Parallelism

LQ

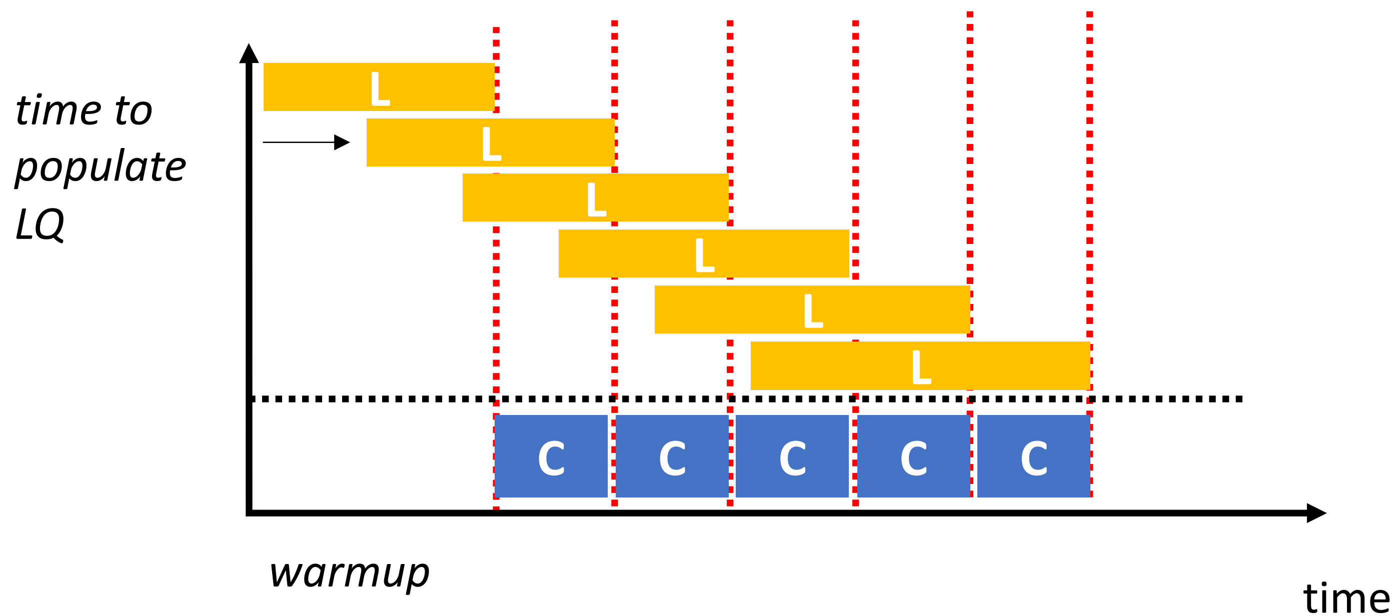
reg	FIFO[0]	FIFO[1]	FIFO[2]	FIFO[3]
addr	b+0	b+1	b+2	b+3


Memory bound applications without terminal loads
Access
Execute
L
L
L
L
C
C
C
C
time

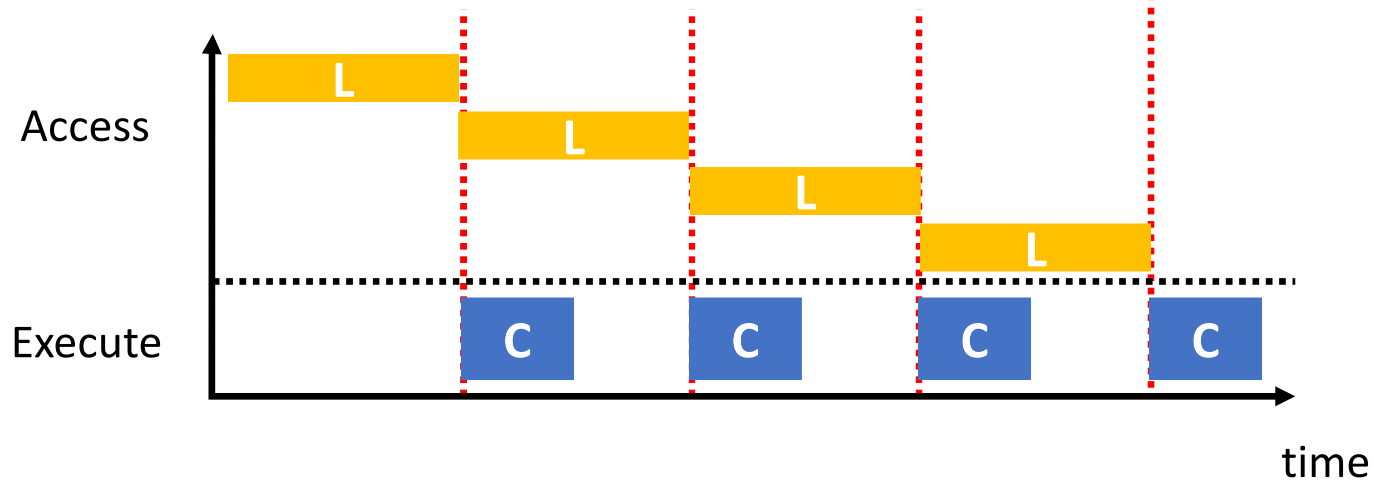
Memory bound applications without terminal loads



Memory bound applications **WITH** terminal loads



Memory bound applications without terminal loads



Memory bound applications **WITH** terminal loads

time to populate LQ

L L L L L L

C C C C C

warmup

time

new constraints?

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Access can "runahead" many loop iterations

How many iterations of runahead is optimal?

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Access can “runahead” many loop iterations

How many iterations of runahead is optimal?

Cache miss costs ~200 cycles in worst case.
A rough approximation is 200 iterations of runahead

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

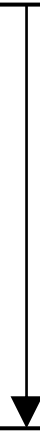
Access can “runahead” many loop iterations

How many iterations of runahead is optimal?

Cache miss costs ~200 cycles in worst case.
A rough approximation is 200 iterations of runahead

Compilers job

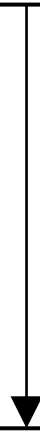
```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```



```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   FIFO_enqueue_addr(b+i);
   SB_store_addr(a + i);
}
```

Compilers job

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    FIFO_enqueue(r0);
    SB_store_addr(a + i);
}
```



```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
    FIFO_enqueue_addr(b+i);
    SB_store_addr(a + i);
}
```

For every load, check if the register is used anywhere except for the enqueue

- SSA?
- Non-SSA?

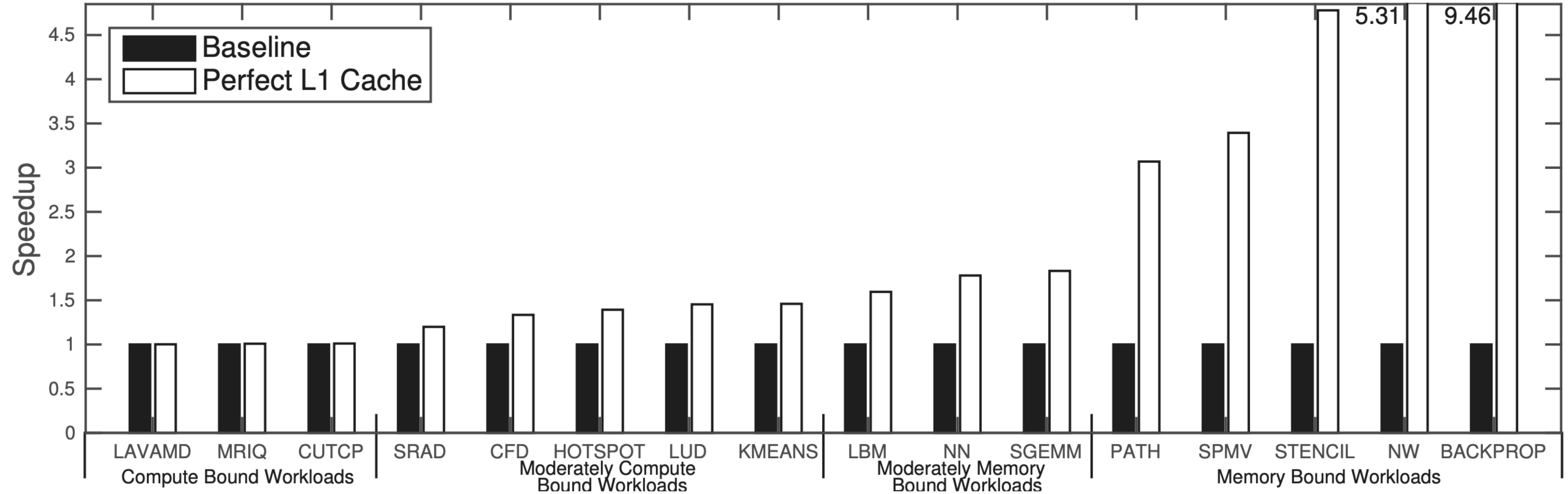
Performance bounds revisited

- A program p has execution time of $E(p)$. The time spent on compute (arithmetic) is $C(p)$. The time spent on memory latency is $M(p)$.
- For many scientific kernels we can approximate: $E(p) = C(p) + M(p)$
- Without terminal loads: $\max\big(C(p), M(p)\big)$best case is when $C(p) \sim = M(p)$, we get $2x$ performance increase
- With terminal loads: $C(p)$

Performance evaluation

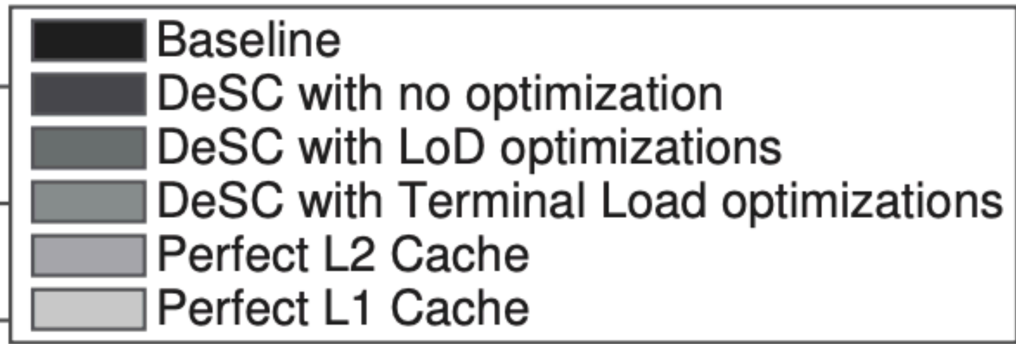
- Given a program p, we can approximate $C(p)$
  - Using profiles to obtain time spent on memory latency and subtracting it from total time
  - Using simulators with a “perfect cache” setting. Setting all cache/dram latencies to 0

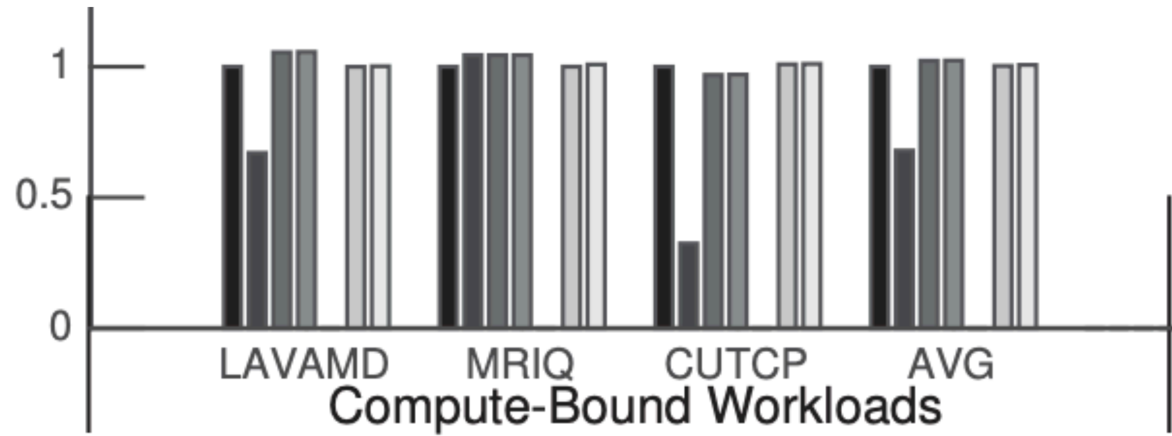
Characterizing applications



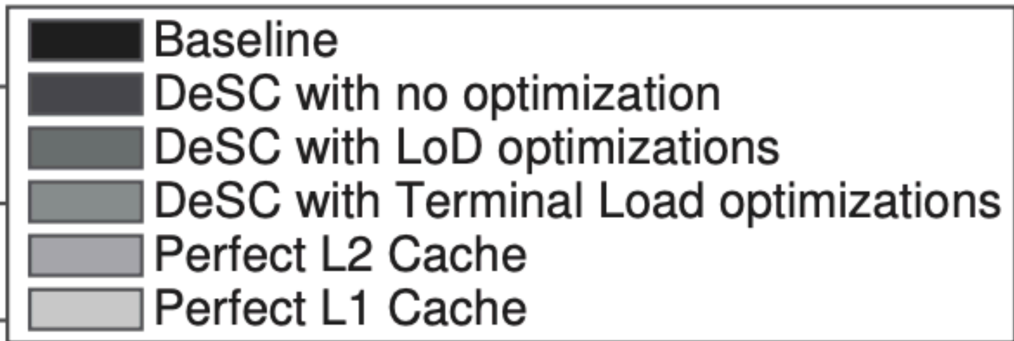
From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015

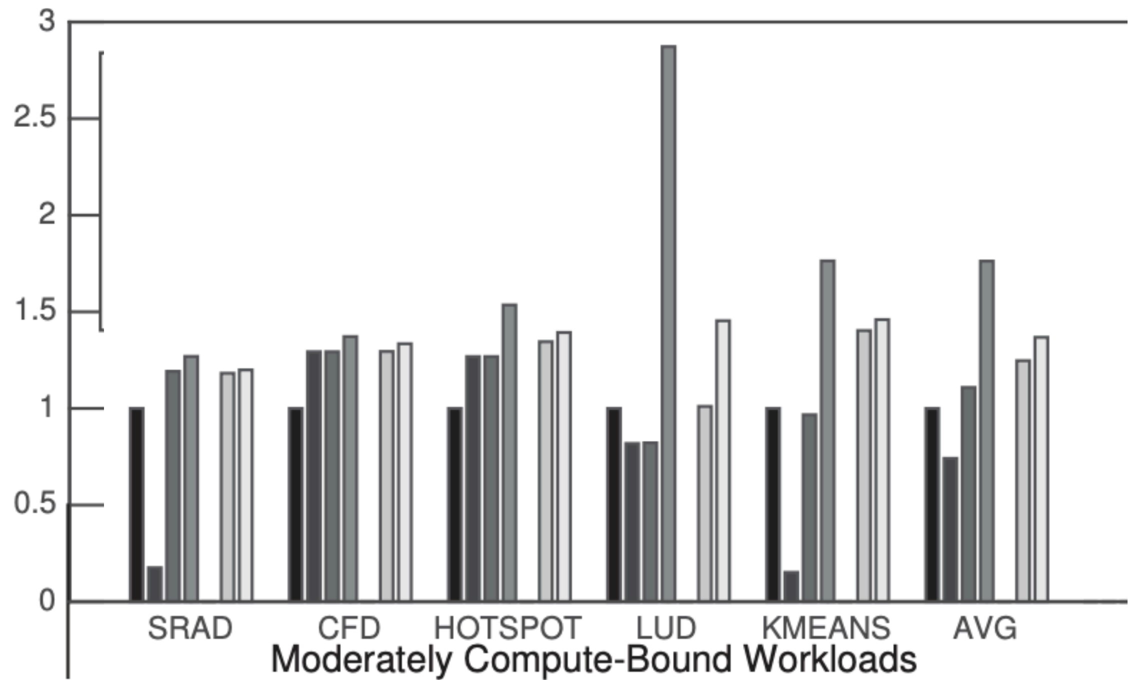
Parboil and Rodinia benchmark suites



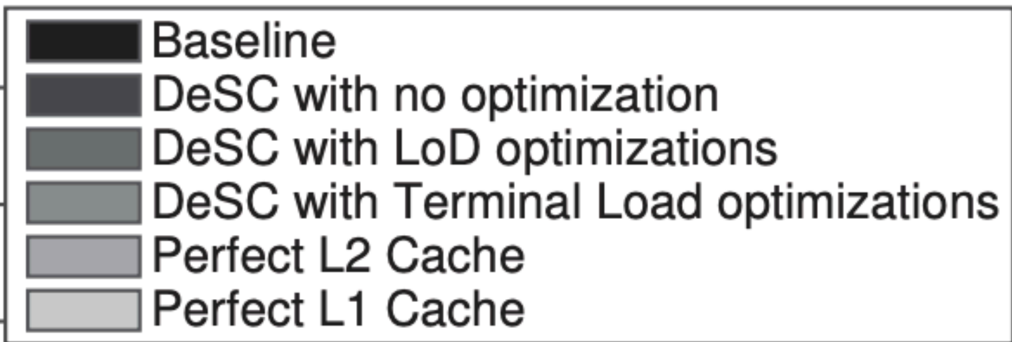


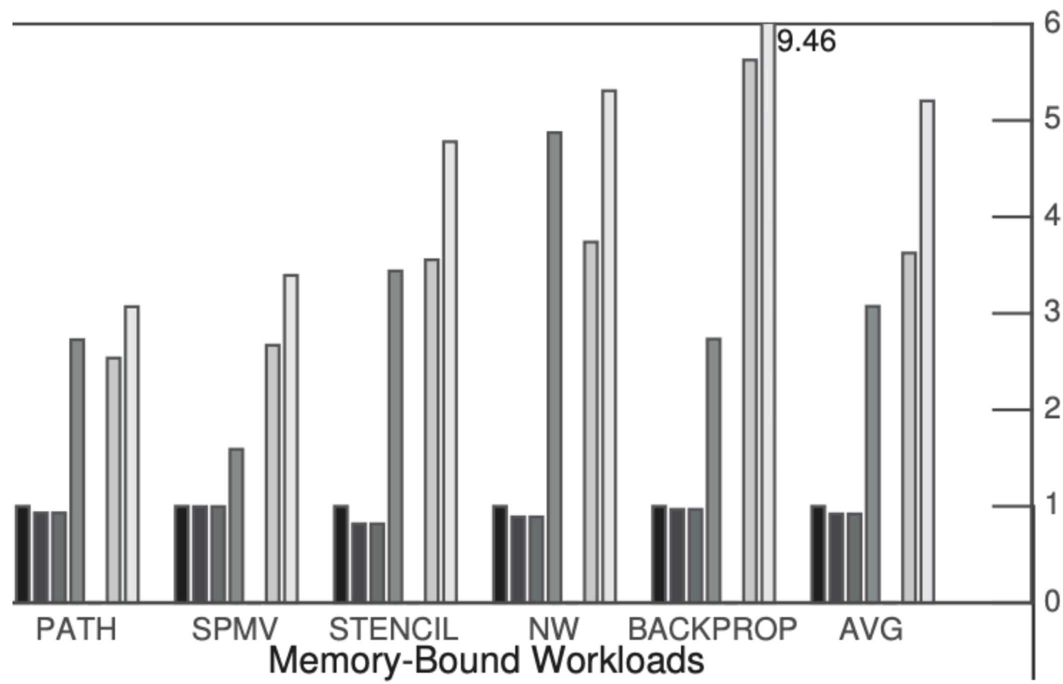
From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015





From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015





From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015

Further considerations in DAE systems

- DeSC paper discusses optimizations:
  - Access speculation on loads
  - Promoting address calculations
  - Store-to-load forwarding

- DeSC paper also discusses implementation details
  - FIFO implementation
  - SB implementation

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + index);
}
```

pointer chasing example

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + index);
}
```

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   FIFO_enqueue(index);
   float value = load(val_array + index);
   FIFO_enqueue(value);
   store_addr(val_array + index);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = FIFO_dequeue();
   float value = FIFO_dequeue();
   // value computation
   SB_store_value(value);
}
```

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + i);
}
```

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   FIFO_enqueue(index);
   float value = load(val_array + index);
   FIFO_enqueue(value);
   store_addr(val_array + index);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = FIFO_dequeue();
   float value = FIFO_dequeue();
   // value computation
   SB_store_value(value);
}
```

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + i);
}
```

cannot eliminate FIFO dequeues from execute alone!

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   FIFO_enqueue(index);
   float value = load(val_array + index);
   FIFO_enqueue(value);
   store_addr(val_array + index);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = FIFO_dequeue();
   float value = FIFO_dequeue();
   // value computation
   SB_store_value(value);
}
```

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + i);
}
```

cannot eliminate FIFO dequeues from execute alone!

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   FIFO_enqueue(index);
   float value = load(val_array + index);
   FIFO_enqueue(value);
   store_addr(val_array + index);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = FIFO_dequeue();
   float value = FIFO_dequeue();
   // value computation
   SB_store_value(value);
}
```

Eliminating unneeded data transfers

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   float value = load(val_array + index);
   // value computation
   store(val_array + i);
}
```

cannot eliminate FIFO dequeues from execute alone!

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = load(offsets + i);
   FIFO_enqueue(index);
   float value = load(val_array + index);
   FIFO_enqueue(value);
   store_addr(val_array + index);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   int index = FIFO_dequeue();
   float value = FIFO_dequeue();
   // value computation
   SB_store_value(value);
}
```

Memory conflicts

- So far we’ve assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

Memory conflicts

- So far we’ve assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

Memory conflicts

- So far we've assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

No because store buffer is ordered!

Memory conflicts

- So far we’ve assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

No because store buffer is ordered!

load-store conflict

```
r0 = load(x)
store(x,2)
```

Can r0 see a value from the future (e.g. 2?)

Memory conflicts

- So far we've assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

No because store buffer is ordered!

load-store conflict

```
r0 = load(x)
store(x,2)
```

Can r0 see a value from the future (e.g. 2?)

No because Access will finish the load before supplying the address to the store

Memory conflicts

- So far we’ve assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

No because store buffer is ordered!

load-store conflict

```
r0 = load(x)
store(x,2)
```

Can r0 see a value from the future (e.g. 2?)

No because Access will finish the load before supplying the address to the store

store-load conflict

```
store(x,2)
r0 = load(x)
```

Can r0 see a stale value? (i.e. r0 != 2?)

....

Memory conflicts

- So far we've assumed well-behaved store buffers, FIFOs, and queues
- Can memory conflicts be an issue when access runs ahead?

store-store conflict

```
store(x,1)
store(x,2)
```

Can the final value in x ever be 1?

No because store buffer is ordered!

load-store conflict

```
r0 = load(x)
store(x,2)
```

Can r0 see a value from the future (e.g. 2?)

No because Access will finish the load before supplying the address to the store

store-load conflict

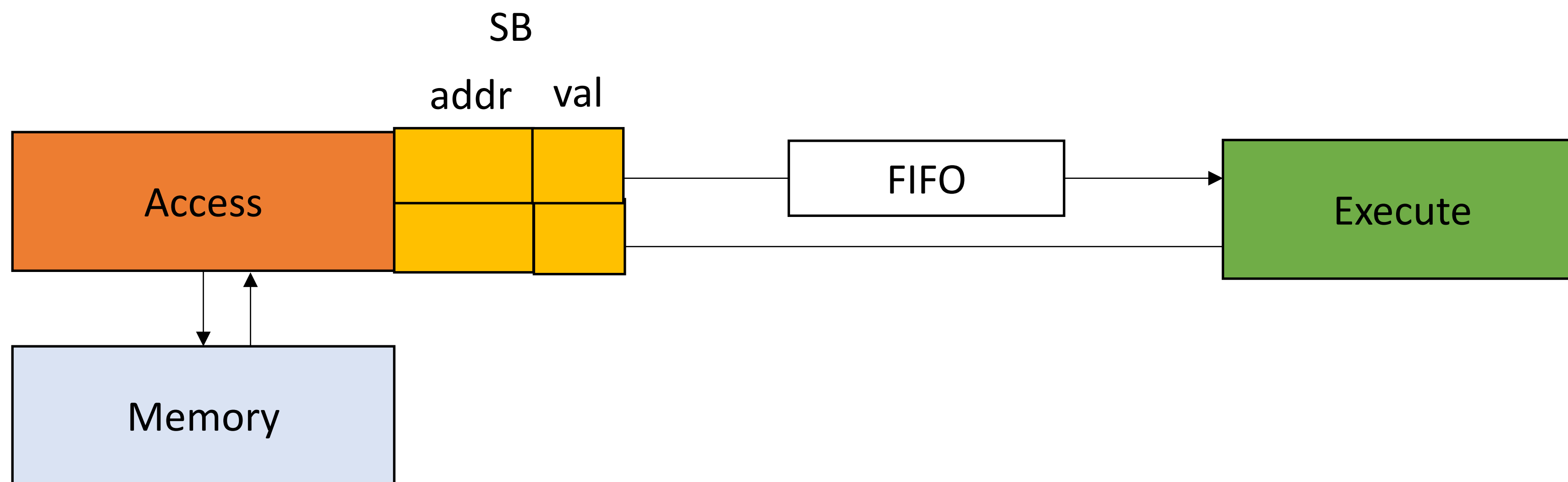
```
store(x,2)
r0 = load(x)
```

Can r0 see a stale value? (i.e. r0 != 2?)

Maybe!

Store-load conflict in DAE

```
store(x,2)
r0 = load(x)
```



Store-load conflict in DAE

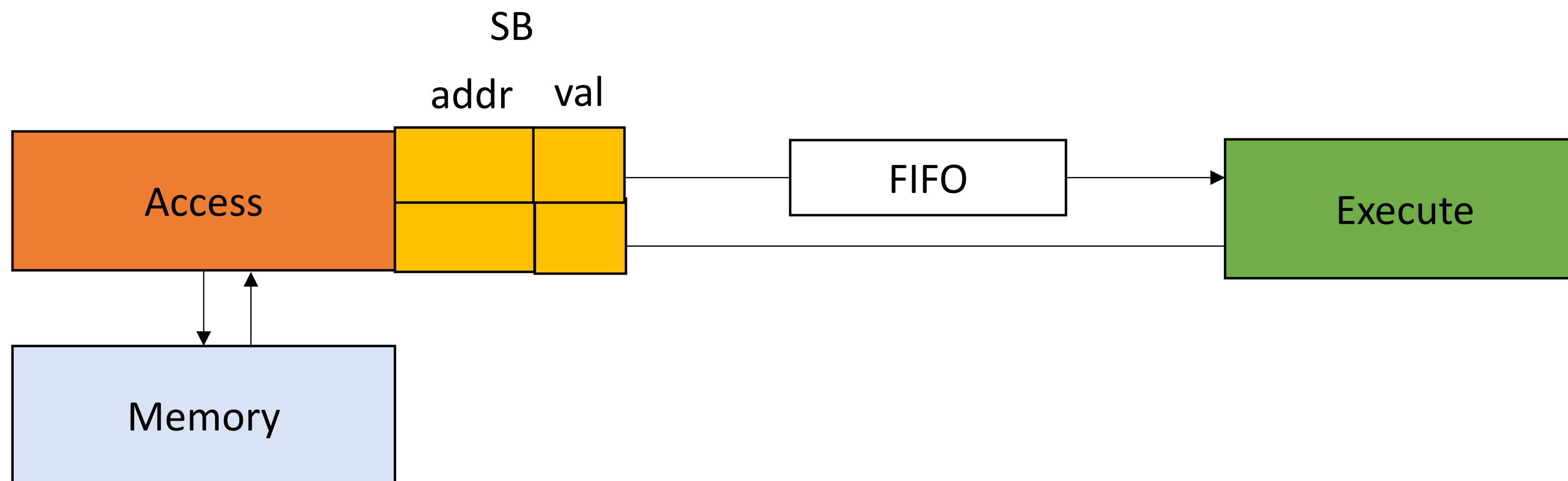
```
store(x,2)
r0 = load(x)
```

Access

```
store_addr(x);
int r0 = load(x);
FIFO_enqueue(r0);
```

Execute

```
SB_store_value(2);
r0 = FIFO_dequeue
```



Store-load conflict in DAE

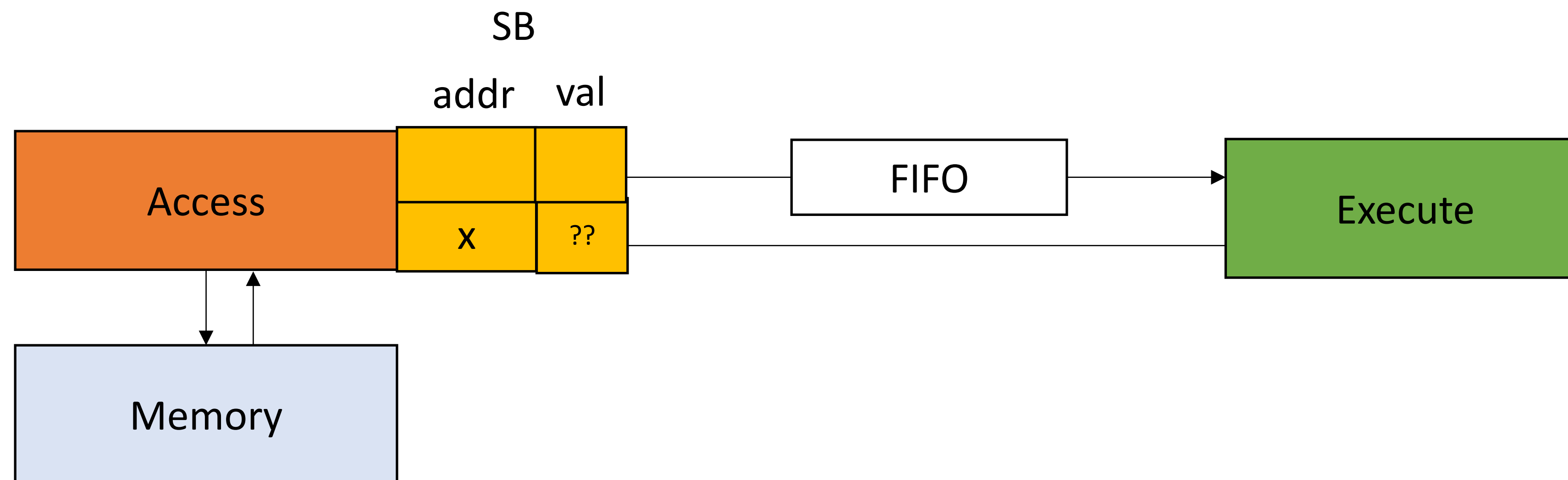
```
store(x,2)
r0 = load(x)
```

Access

```
store_addr(x);
int r0 = load(x);
FIFO_enqueue(r0);
```

Execute

```
SB_store_value(2);
r0 = FIFO_dequeue
```



Store-load conflict in DAE

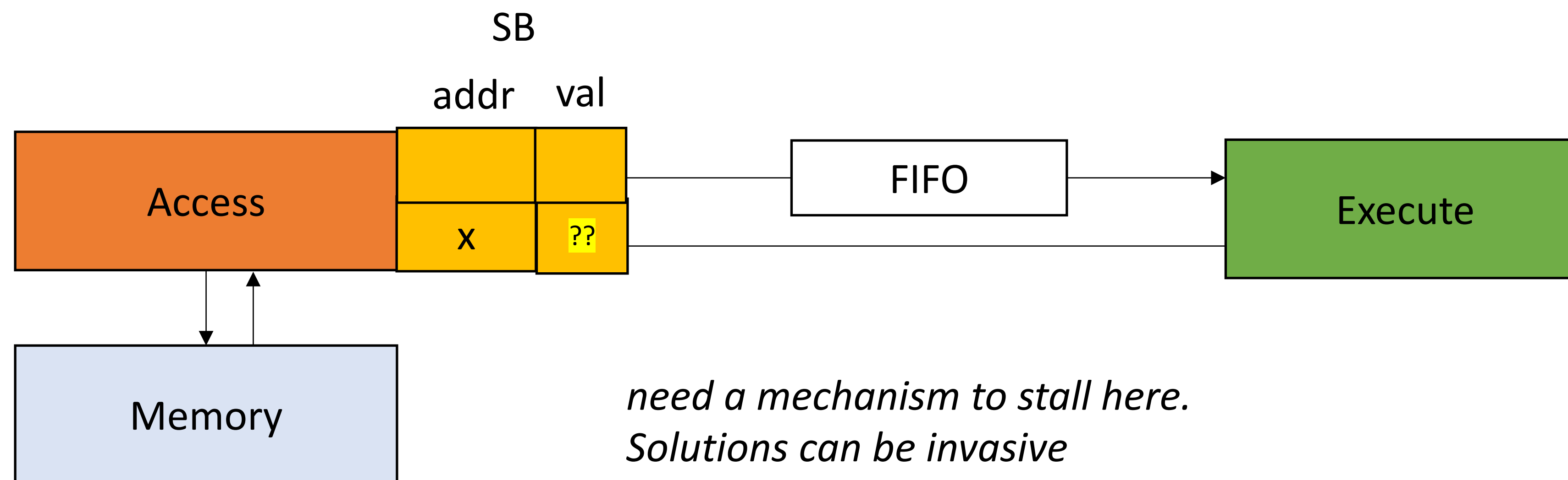
```
store(x,2)
r0 = load(x)
```

Access

```
store_addr(x);
int r0 = load(x);
FIFO_enqueue(r0);
```

Execute

```
SB_store_value(2);
r0 = FIFO_dequeue
```



Store-load conflict in DAE

```
store(x,2)
r0 = load(x)
```

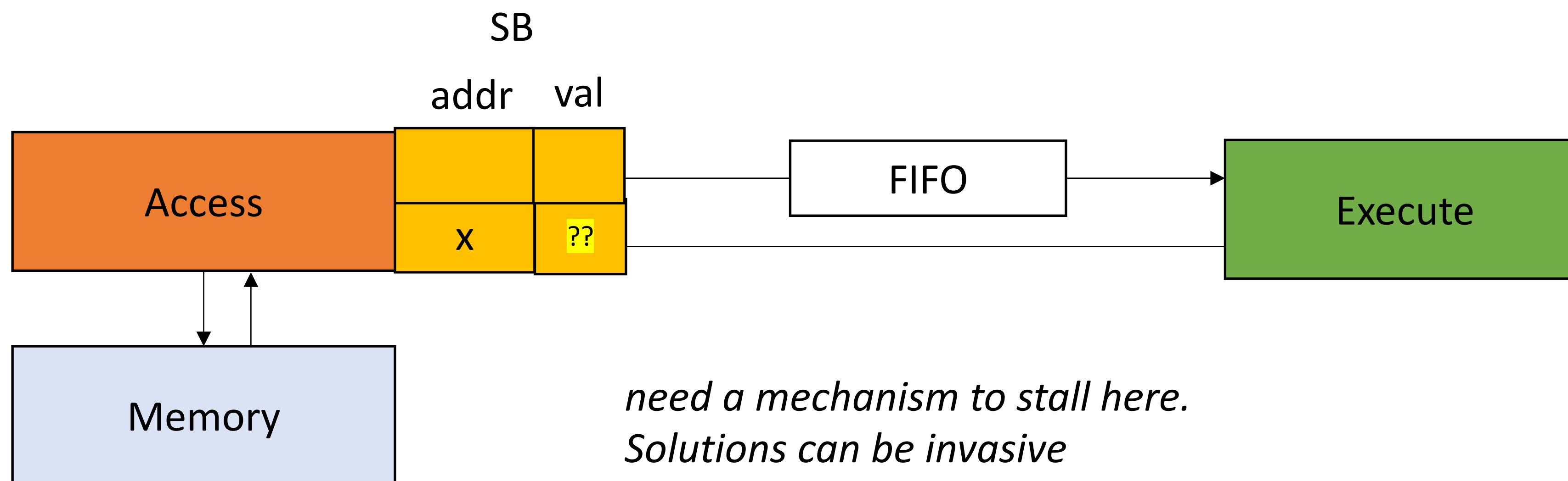
Access

```
store_addr(x);
while(unmatched(x));
int r0 = load(x);
FIFO_enqueue(r0);
```

Execute

```
SB_store_value(2);
r0 = FIFO_dequeue
```

One solution: compiler inserted checks



Store-load conflict in DAE

```
store(x,2)
r0 = load(x)
```

Access

```
store_addr(x);
while(unmatched(x));
int r0 = load(x);
FIFO_enqueue(r0);
```

Execute

```
SB_store_value(2);
r0 = FIFO_dequeue
```

We only need the check if there is potentially a store-load conflict!

SMT query for Store-load conflict

```
for (int i = 0; i < SIZE; i++) {
    a[write_index(i)] = a[read_index(i)];
}
```

for DOALL Loop:

0 < ix < SIZE
0 < iy < SIZE
ix != iy

check:
write_index[ix] != write_index[iy]
write_index[ix] != read_index[iy]

for DAE store-load conflicts:

0 < ia < SIZE
0 < ie < SIZE
ia >= ie

check:
write_index[ie] != read_index[ia]

SMT query for Store-load conflict

Example: Reduction:

```
for (int i = 0; i < SIZE; i++) {
    a[0] = a[0] + a[i];
}
```

Safe in DOALL or DAE?

for DOALL Loop:

0 < ix < SIZE
0 < iy < SIZE
ix != iy

check:
write_index[ix] != write_index[iy]
write_index[ix] != read_index[iy]

for DAE store-load conflicts:

0 < ia < SIZE
0 < ie < SIZE
ia != ie

check:
write_index[ie] != read_index[ia]

SMT query for Store-load conflict

Example: shift

```
for (int i = 0; i < SIZE - 1; i++) {
    a[i] = a[i+1];
}
```

Safe in DOALL or DAE?

for DOALL Loop:

0 < ix < SIZE
0 < iy < SIZE
ix != iy

check:
write_index[ix] != write_index[iy]
write_index[ix] != read_index[iy]

for DAE store-load conflicts:

0 < ia < SIZE
0 < ie < SIZE
ia != ie

check:
write_index[ie] != read_index[ia]

DAE Summary

- Small hardware modifications + compiler techniques create a simple architecture that is good at memory-bound applications.

- Lots of room for exploration in architecture specialization and compiler techniques!

- Keep your eyes open for DAE systems in the near future!

Compiler Optimization Policies

Big question

- When should optimizations be enabled or disabled?

Big question

- When should optimizations be enabled or disabled?
  - if optimization adds a large compile time
  - if optimization makes debugging harder
  - if optimization makes smaller binaries
  - if optimization is not well tested
  - if optimization is likely to provide a performance increase

What do modern compilers do?

- gcc?

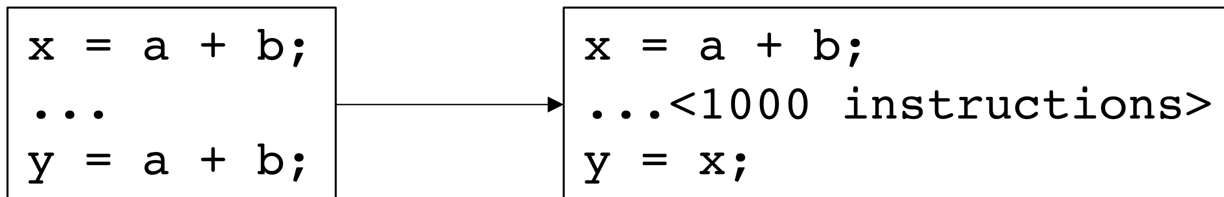
What do modern compilers do?

- gcc?
  - -O0, -O1, -O2
- See differences at:
  https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- different optimizations for different use cases
  - -Os, -Og, -Ofast

Making programs go faster

- All of the optimizations we’ve examined have had performance trade-offs

- Local value numbering?



what can go wrong?

Making programs go faster

- All of the optimizations we've examined have had performance trade-offs

- Local value numbering?

what can go wrong?

x might have gone to memory if there isn't enough registers. A memory access is more expensive than some arithmetic operations

Same issue for Pipelining and Super Scalar re-orderings!

Making programs go faster

- All of the optimizations we've examined have had performance trade-offs
- Loop unrolling?

Making programs go faster

- All of the optimizations we've examined have had performance trade-offs

- DOALL parallelism? DAE?

DOALL provides a 20x slowdown with insufficient work.
DAE slows down programs if they cannot be efficiently decoupled

Making programs go faster

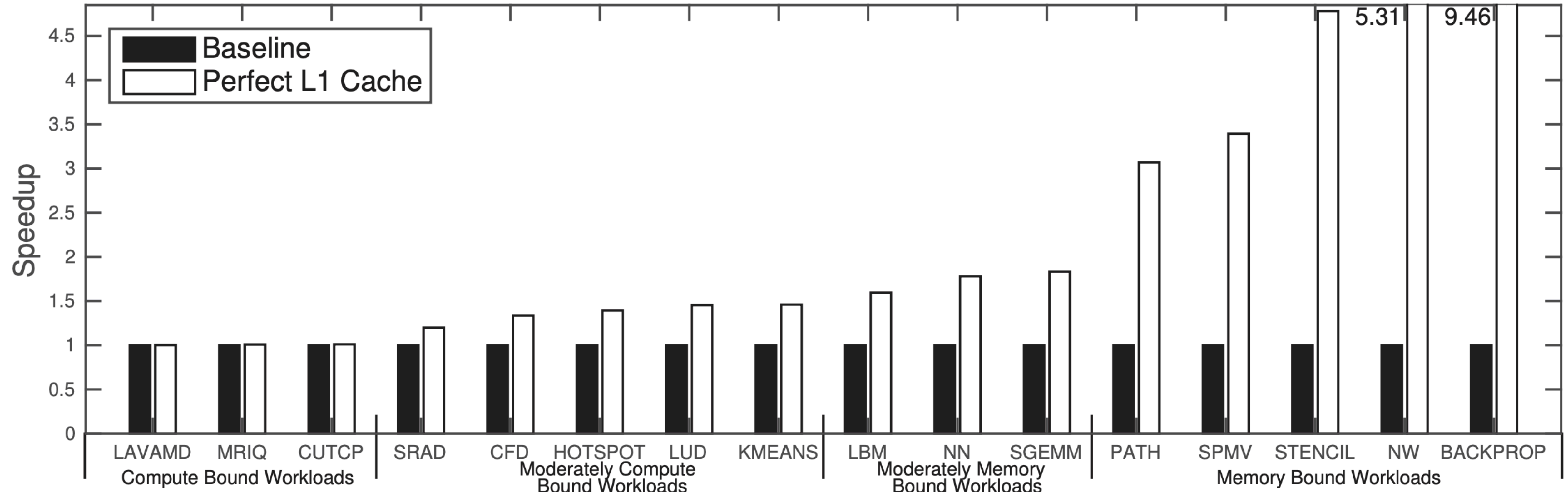
- Any optimizations always make code go faster?

Compilers are evaluated on benchmark suites

- Scientific computing
  - Rodinia, Parboil, Linpack
- Managed Languages:
  - Decapo (Java)
- Heterogeneous systems
  - SHOC
- GPU
  - Magma
- Graphs
  - GAPs

combination?
https://www.phoronix.com/scan.php?page=article&item=gcc-clang-2019&num=1

Good choice for DAE



From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015

Running benchmarks

- Just run it?
- Need to be careful...

Next week:

- Paper presentations
  - Half days for presentations
  - I’ll spend the other half doing my own paper presentations (DSL and performance portability)