CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions

Sawan Singh[1], Josue Feliu[2], Manuel E. Acacio[1], Alexandra Jimborean[1], Alberto Ros[1]

[1] Universidad de Murcia
Murcia, Spain

UNIVERSIDAD DE
MURCIA

[2] Universitat Politècnica de València
Valencia, Spain

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Monday, October 23, 2023

32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)

Overview

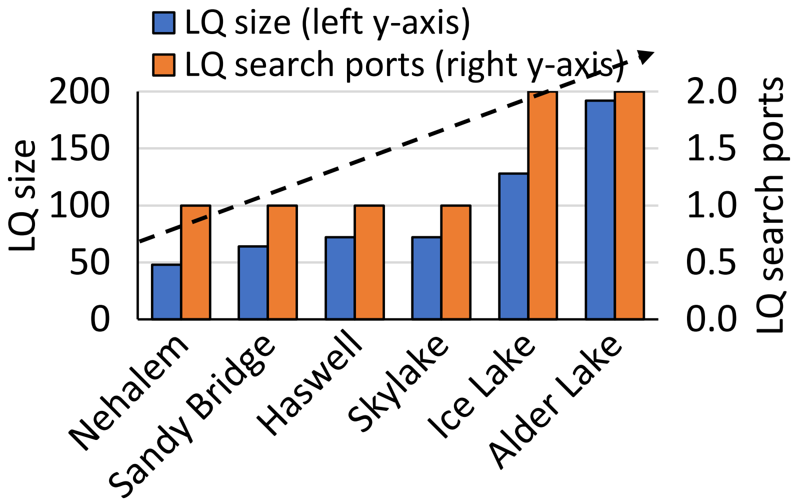
- Load queue (LQ) is one of the most critical structures in a processor

Overview

- Load queue (LQ) is one of the most critical structures in a processor
- LQ keeps all in-flight loads in order and supports priority searches

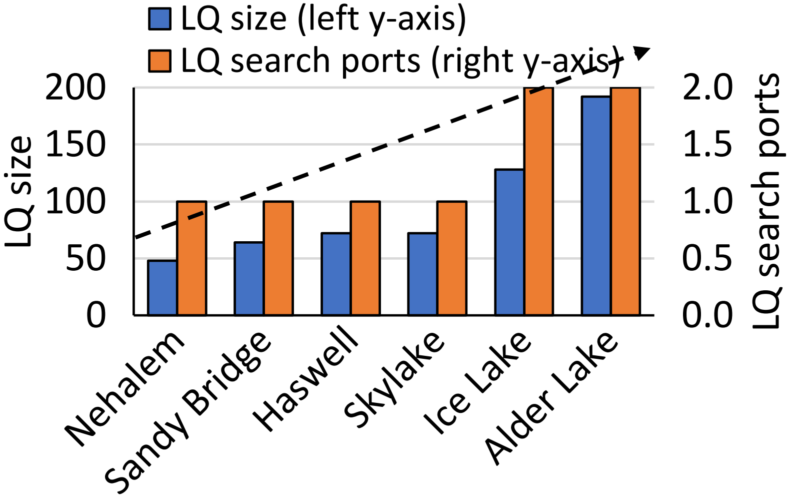
Overview

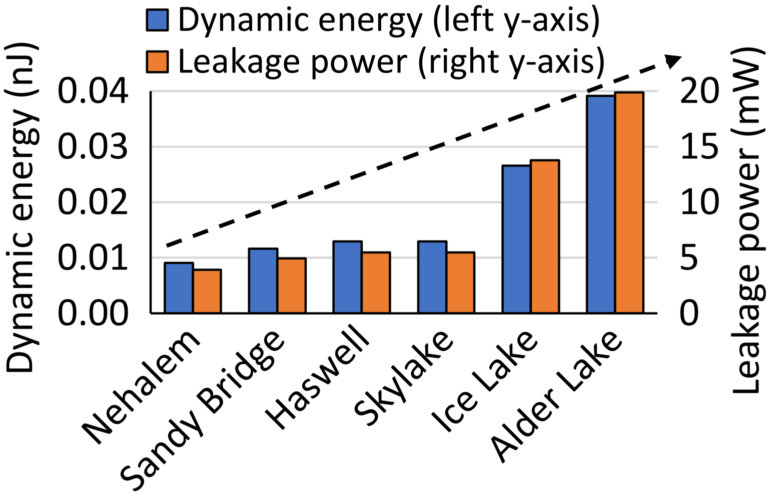
- Load queue (LQ) is one of the most critical structures in a processor
- LQ keeps all in-flight loads in order and supports priority searches
- LQ size has been increasing



Overview

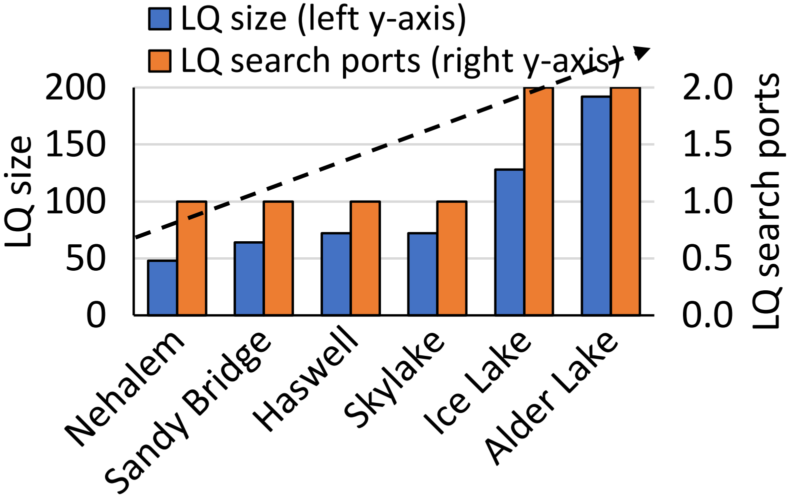
- ➢ Load queue (LQ) is one of the most critical structures in a processor
- ➢ LQ keeps all in-flight loads in order and supports priority searches
- ➢ LQ size has been increasing
- ➢ Energy consumption of the LQ is also growing

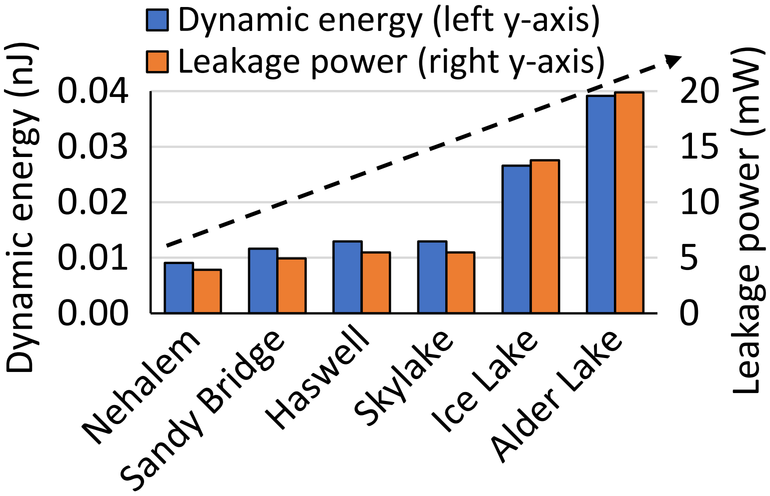




Overview

- Load queue (LQ) is one of the most critical structures in a processor
- LQ keeps all in-flight loads in order and supports priority searches
- LQ size has been increasing
- Energy consumption of the LQ is also growing
- Simultaneous multithreading (SMT) intensifies the pressure on LQ as it requires additional LQ searches





Overview

We propose CELLO

- A software-hardware co-design for SMT processors with TSO consistency model

Overview

We propose CELLO

- A software-hardware co-design for SMT processors with TSO consistency model
- The compiler detects memory operations in DRF regions

Overview

We propose CELLO

- A software-hardware co-design for SMT processors with TSO consistency model
- The compiler detects memory operations in DRF regions
- The hardware optimizes their execution by safely skipping the LQ searches without violating the TSO consistency model

Overview

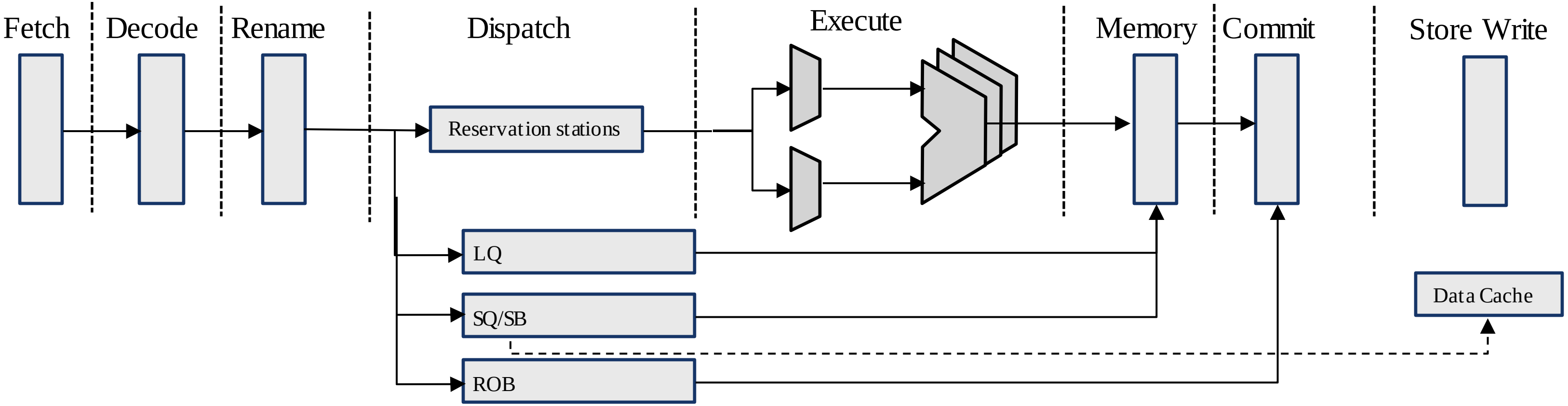
We propose CELLO

- A software-hardware co-design for SMT processors with TSO consistency model
- The compiler detects memory operations in DRF regions
- The hardware optimizes their execution by safely skipping the LQ searches without violating the TSO consistency model
- CELLO reduces LQ searches by half

Outline

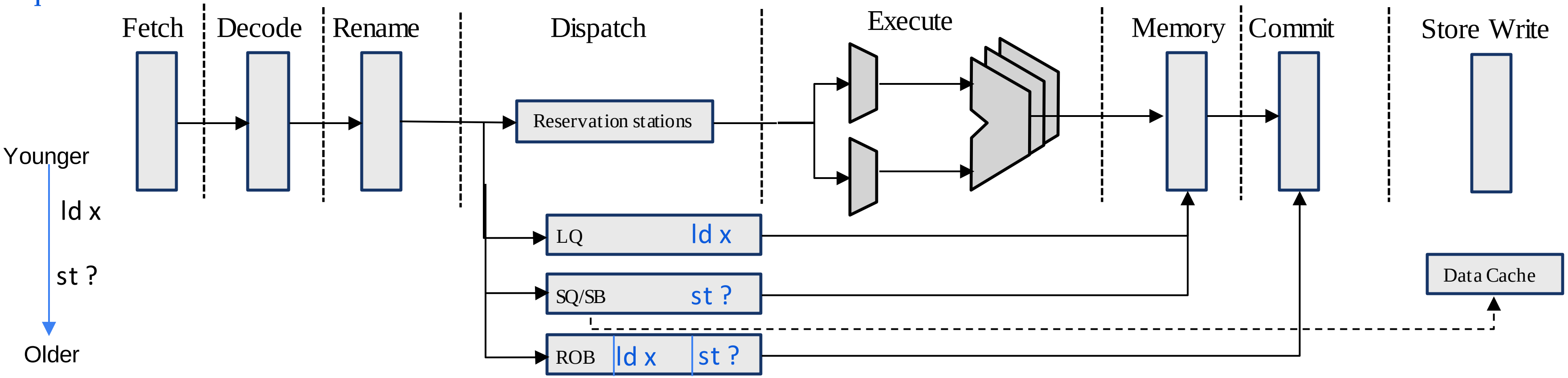
- Overview
- Background
- CELLO
- Evaluation
- Conclusion

Background [LQ searches in single thread]



Background [LQ searches in single thread]

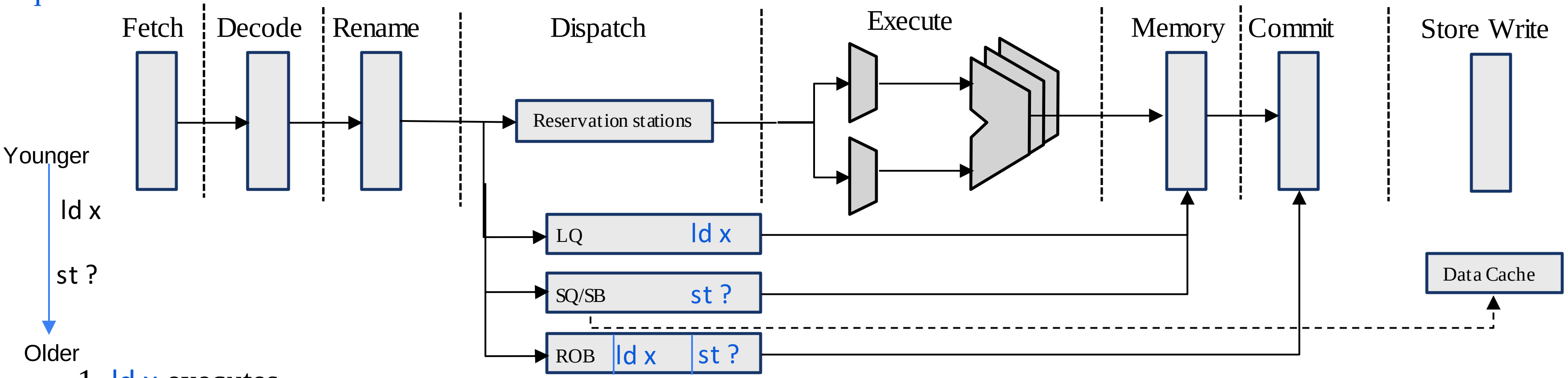
Sequential semantics



CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions @ PACT'23

Background [LQ searches in single thread]

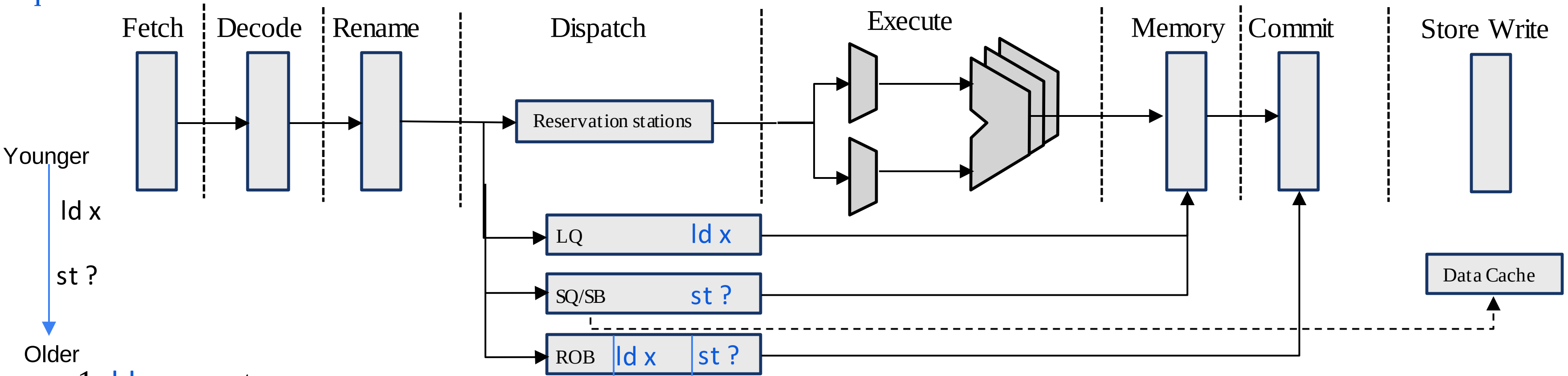
Sequential semantics



1. ld x executes
   - When loads execute the target address of older stores may be unknown

Background [LQ searches in single thread]

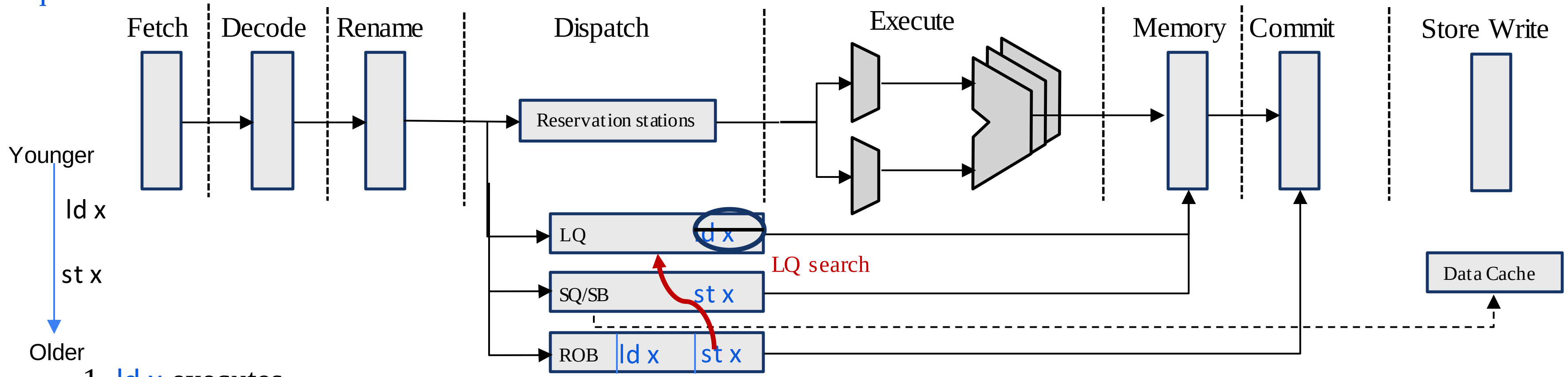
Sequential semantics



1. ld x executes
   - When loads execute the target address of older stores may be unknown
   - Loads executing in the presense of an older unresolved stores are dependency-speculative (D-speculative)

Background [LQ searches in single thread]

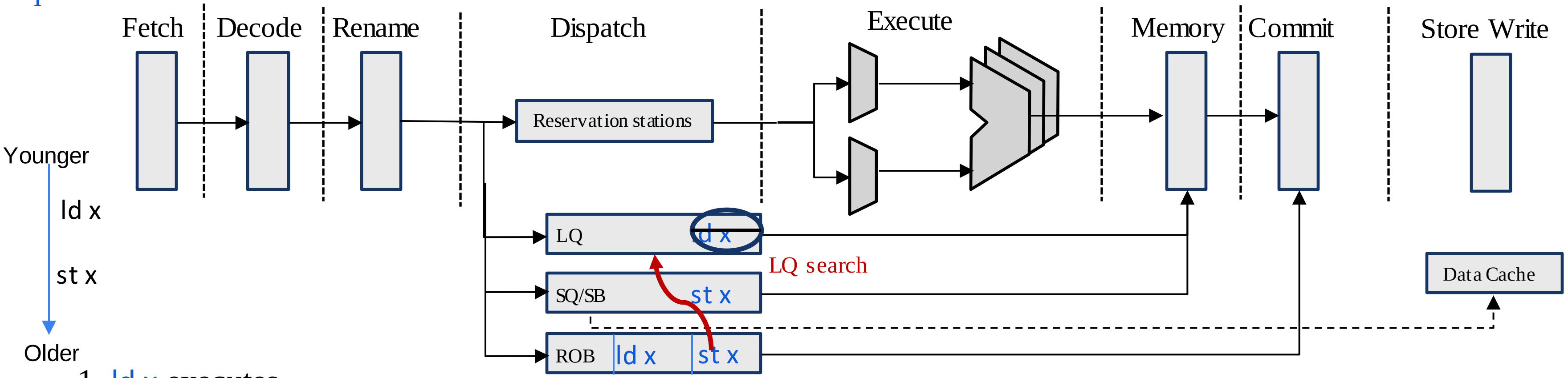
Sequential semantics



1. ld x executes
   - ➢ When loads execute the target address of older stores may be unknown
   - ➢ Loads executing in the presense of an older unresolved stores are dependency-speculative (D-speculative)
2. st resolves its address
   - ➢ Stores search the LQ to make their presence known to younger loads that might have executed D-speculatively

Background [LQ searches in single thread]

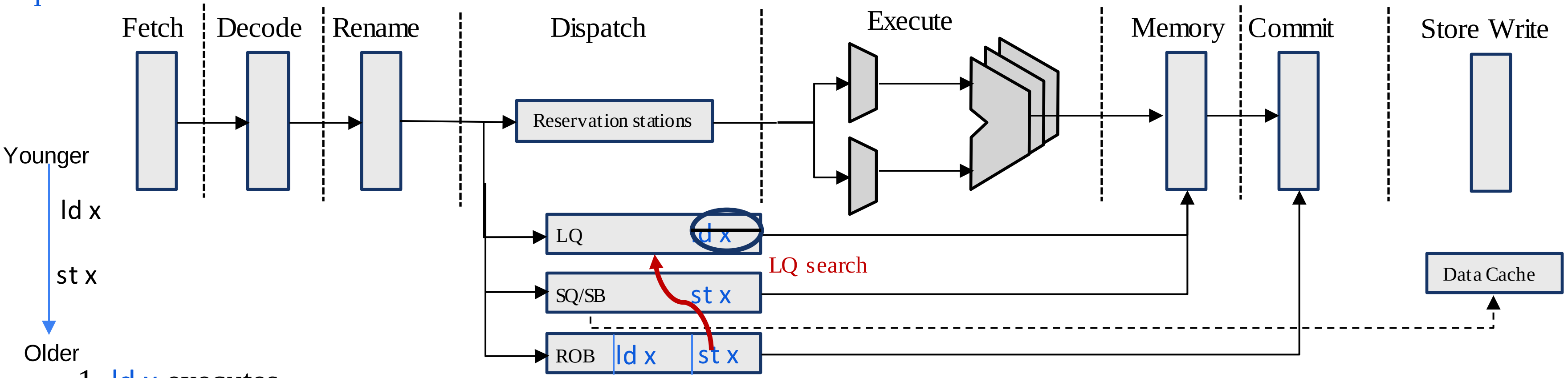
Sequential semantics



1. ld x executes
   - ➢ When loads execute the target address of older stores may be unknown
   - ➢ Loads executing in the presense of an older unresolved stores are dependency-speculative (D-speculative)
2. st resolves its address
   - ➢ Stores search the LQ to make their presence known to younger loads that might have executed D-speculatively
   - ➢ If found the load and the subsequent instructions are squashed and re-executed

Background [LQ searches in single thread]

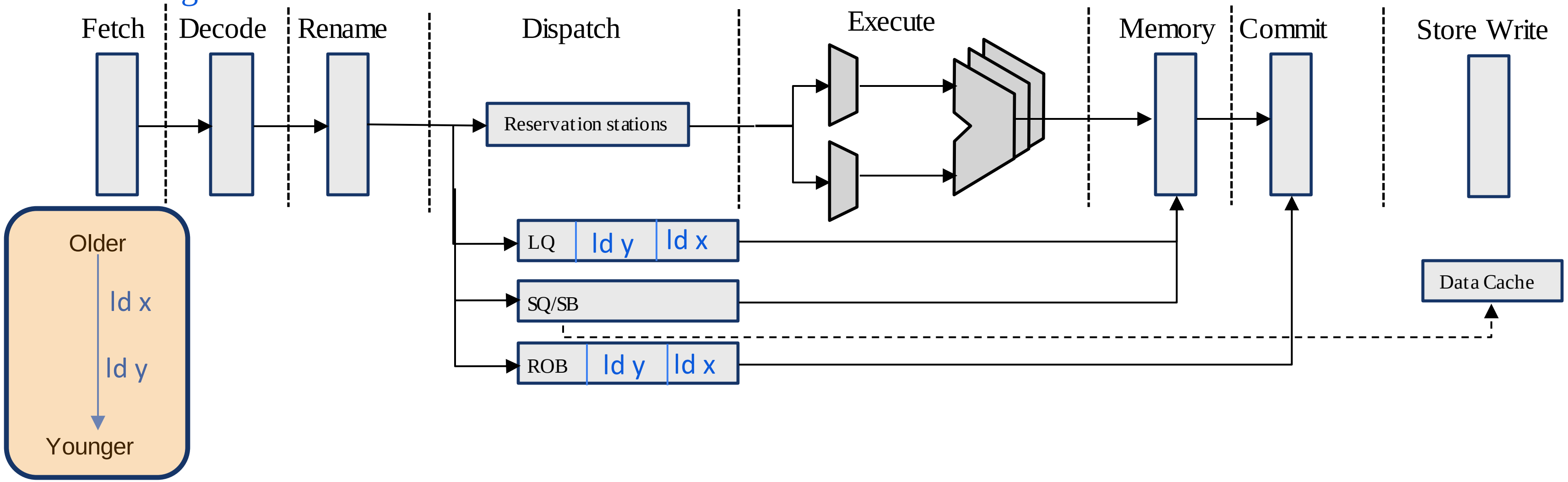
Sequential semantics



1. ld x executes
   - When loads execute the target address of older stores may be unknown
   - Loads executing in the presense of an older unresolved stores are dependency-speculative (D-speculative)
2. st resolves its address
   - Stores search the LQ to make their presence known to younger loads that might have executed D-speculatively
   - If found the load and the subsequent instructions are squashed and re-executed
   - LQ search by stores is 51% of total LQ searches

Background [LQ searches in single thread]

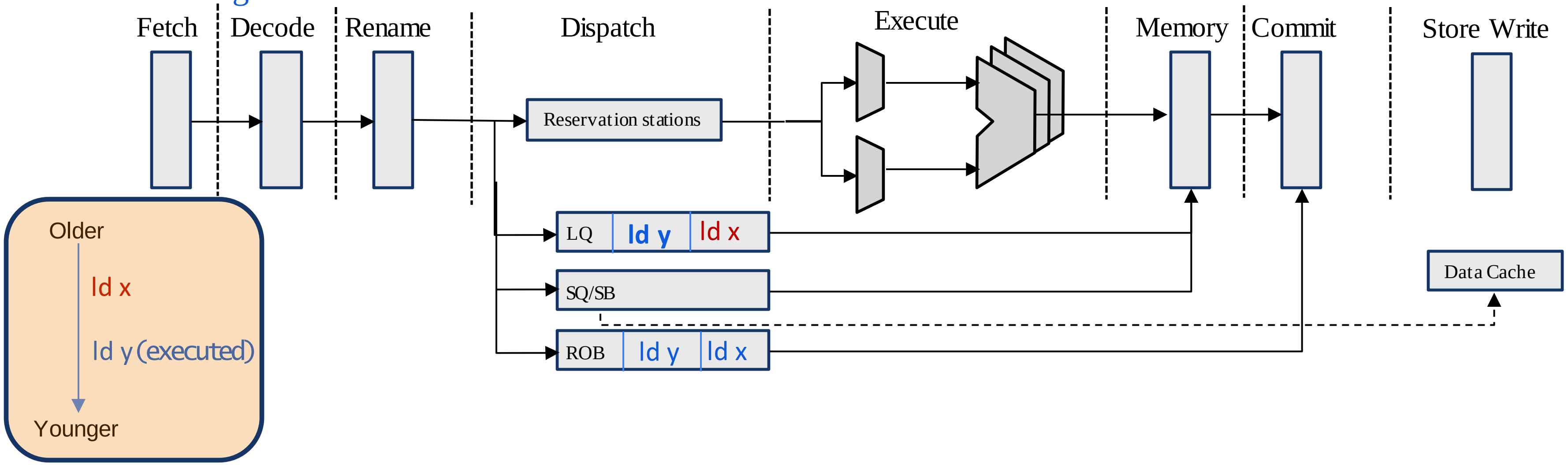
load-load ordering



- TSO respects load-load ordering

Background [LQ searches in single thread]

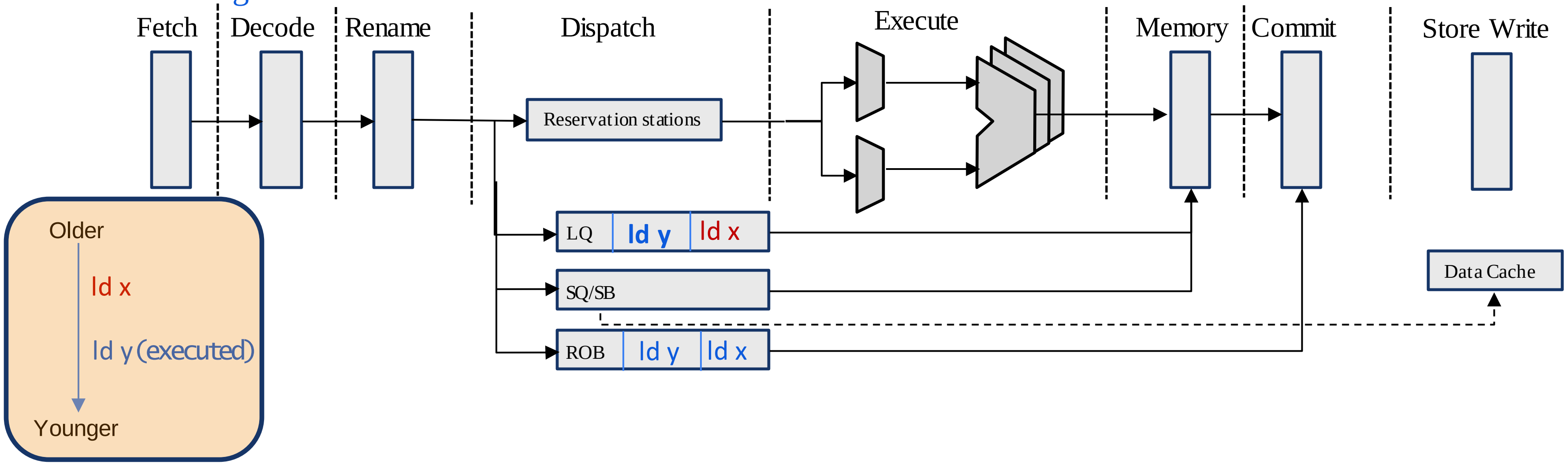
load-load ordering



- TSO respects load-load ordering
- The younger executed load becomes speculative when an older load has not yet performed

Background [LQ searches in single thread]

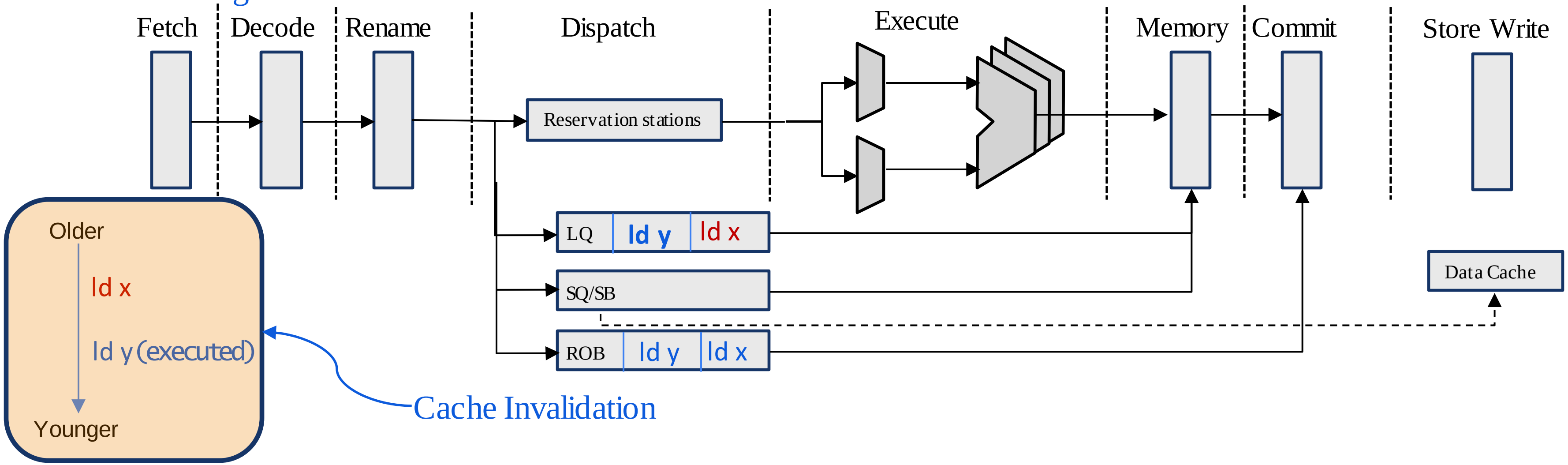
load-load ordering



- ➢ TSO respects load-load ordering
- ➢ The younger executed load becomes speculative when an older load has not yet performed
- ➢ These speculative loads are called memory-speculative (M-Speculative)

Background [LQ searches in single thread]

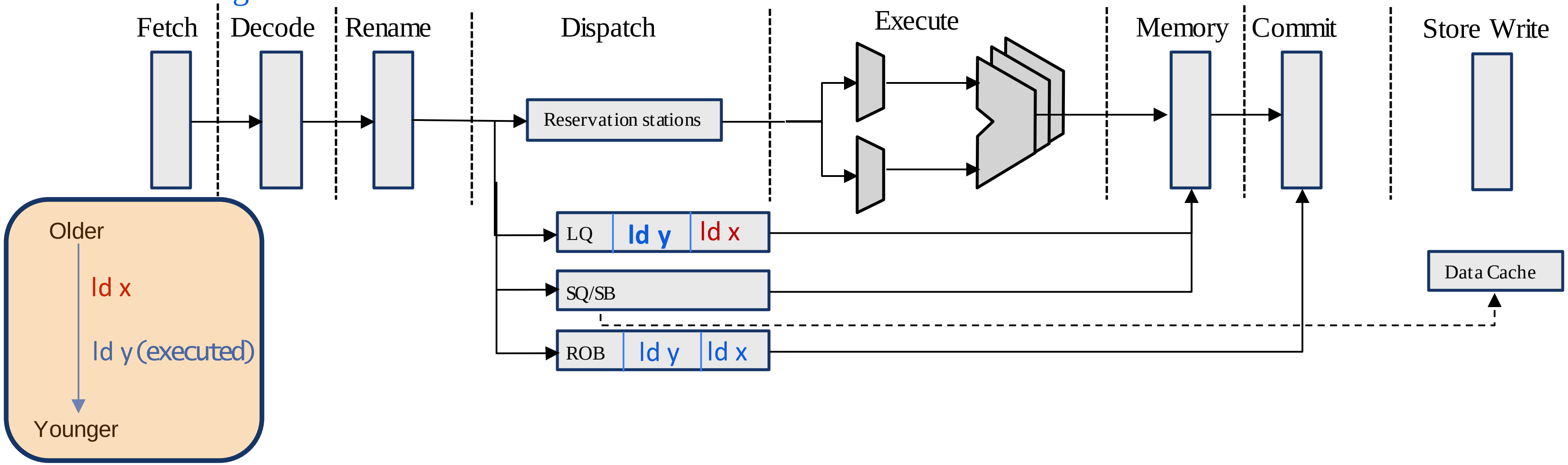
load-load ordering



- TSO respects load-load ordering
- The younger executed load becomes speculative when an older load has not yet performed
- These speculative loads are called memory-speculative (M-Speculative)
- Cache invalidations can expose speculative loads in another core

Background [LQ searches in single thread]

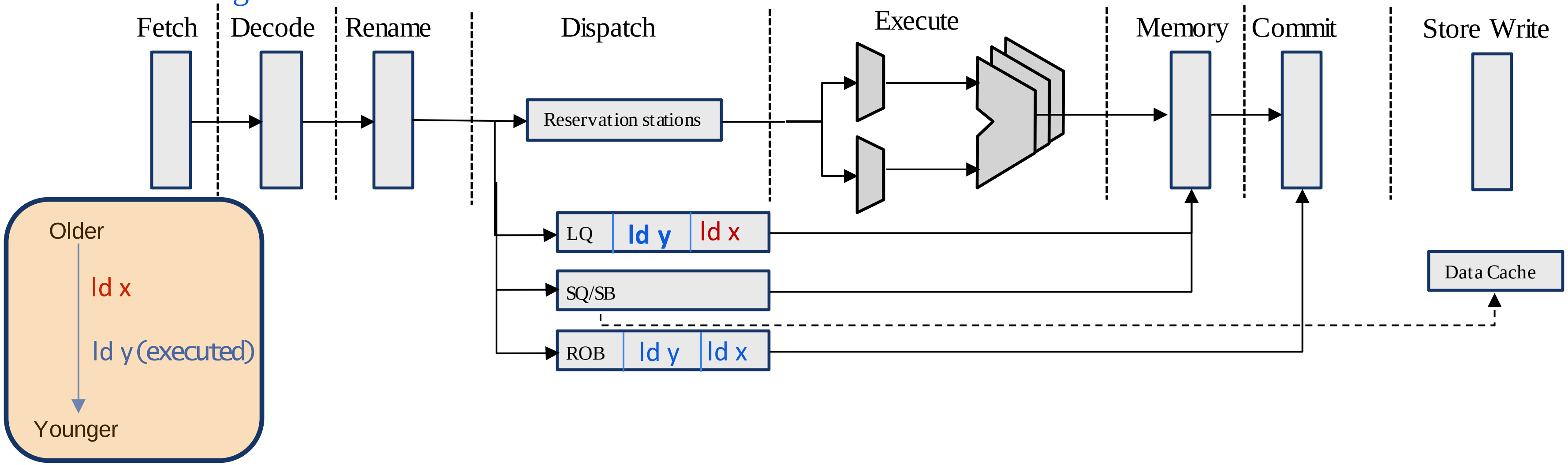
load-load ordering



- TSO respects load-load ordering
- The younger executed load becomes speculative when an older load has not yet performed
- These speculative loads are called memory-speculative (M-Speculative)
- Cache invalidations can expose speculative loads in another core
- Cache evictions are also treated as invalidations as once evicted from cache it no longer can receive an invalidation

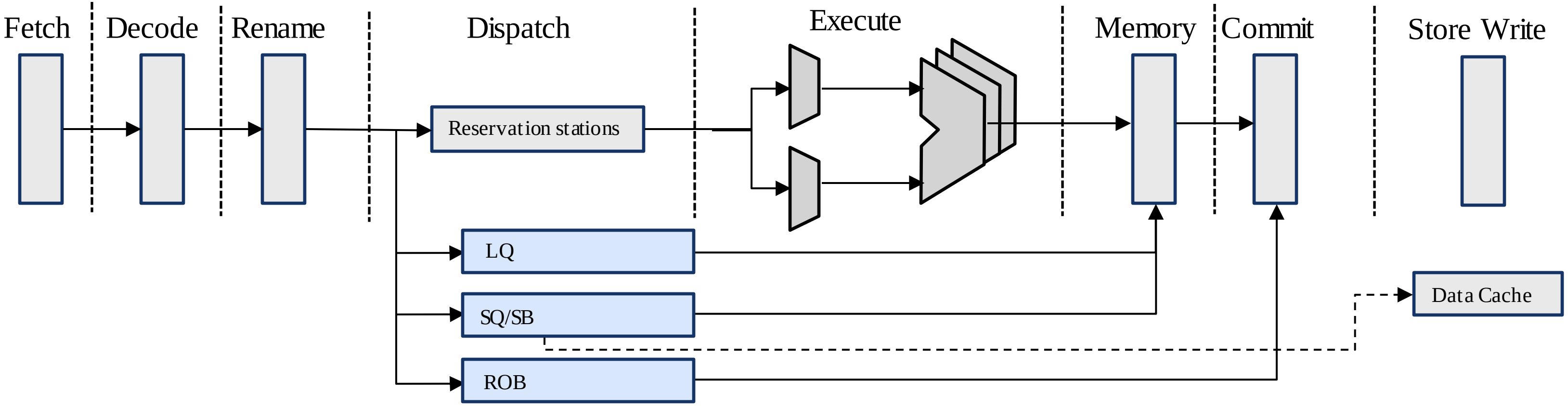
Background [LQ searches in single thread]

load-load ordering



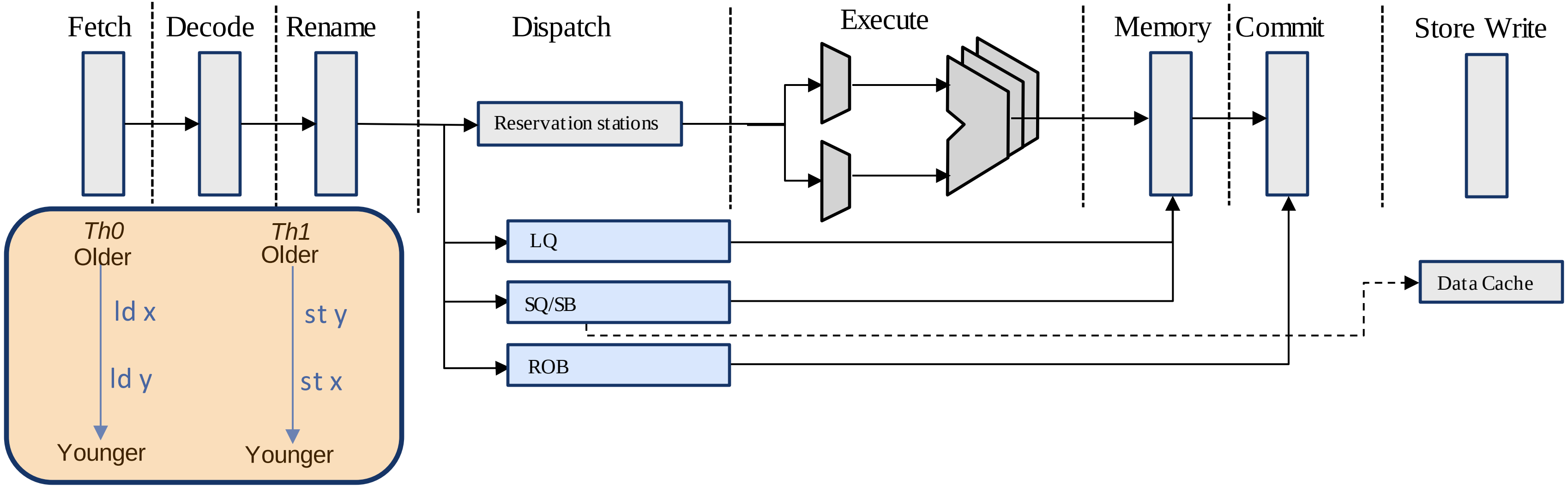
- ➢ TSO respects load-load ordering
- ➢ The younger executed load becomes speculative when an older load has not yet performed
- ➢ These speculative loads are called memory-speculative (M-Speculative)
- ➢ Cache invalidations can expose speculative loads in another core
- ➢ Cache evictions are also treated as invalidations as once evicted from cache it no longer can receive an invalidation
- ➢ The LQ is searched by cache invalidations and evictions, which is about 3% in evaluated benchmarks

Background [LQ searches in SMT]



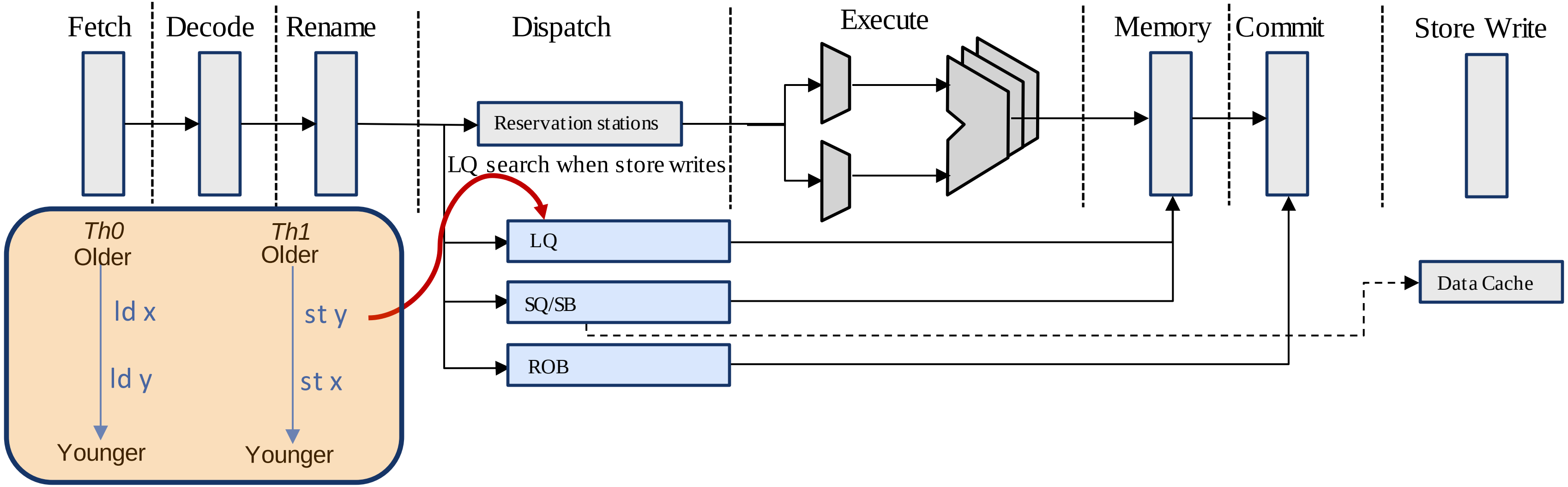
- ➢ Structures in blue are partitioned between SMT threads
- ➢ Multiple SMT threads can run in a single SMT core

Background [LQ searches in SMT]



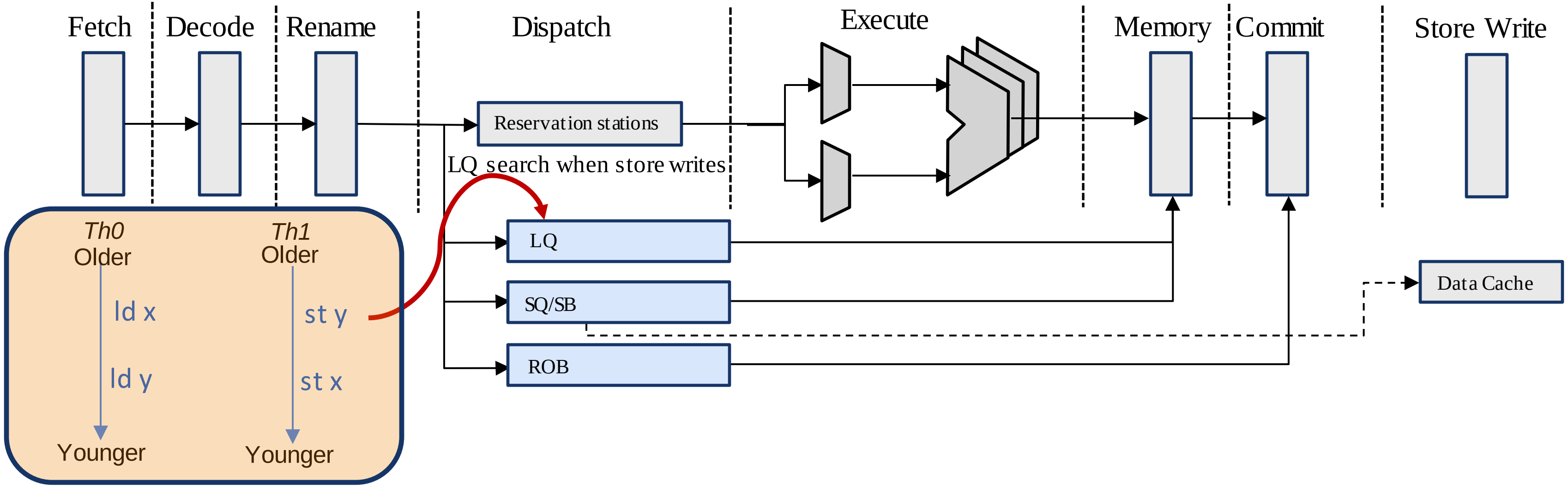
- No invalidations to check load-load ordering as now it executes in a single SMT core

Background [LQ searches in SMT]



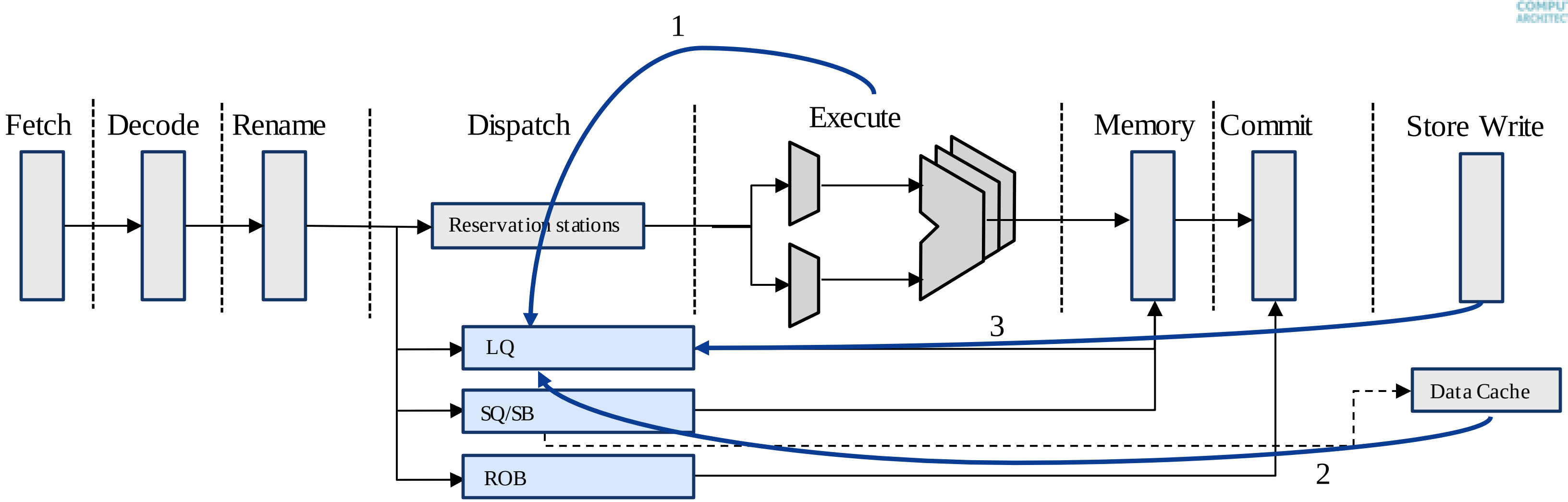
- ➢ No invalidations to check load-load ordering as now it executes in a single SMT core
- ➢ Stores search the LQ of other threads when writing to the cache

Background [LQ searches in SMT]



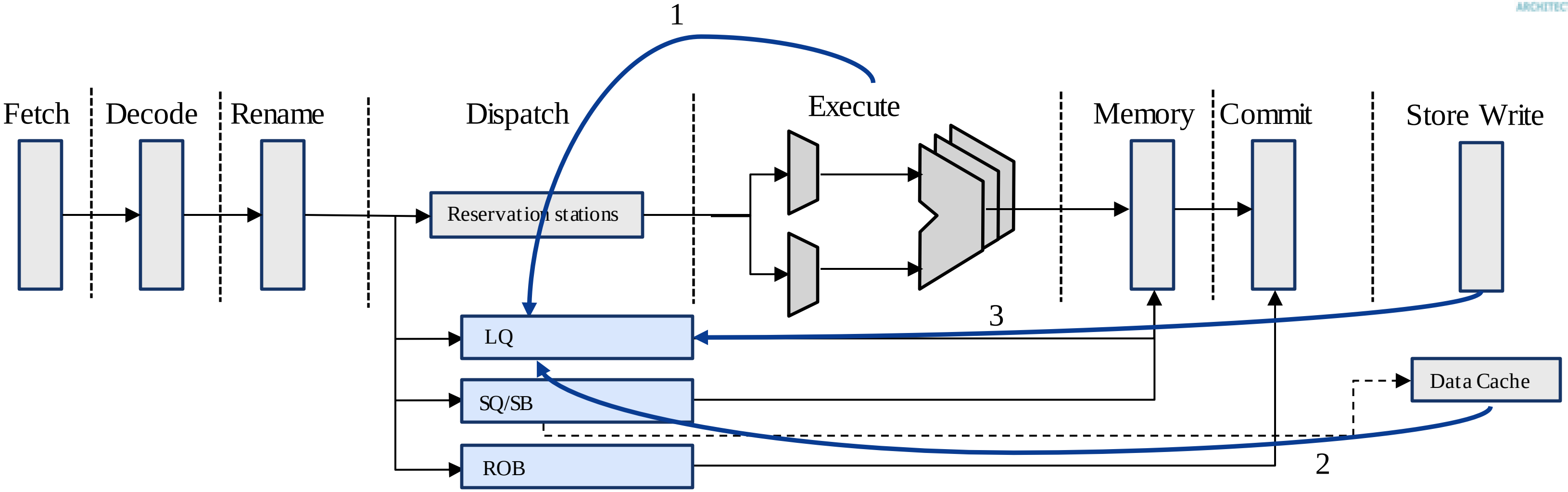
- No invalidations to check load-load ordering as now it executes in a single SMT core
- Stores search the LQ of other threads when writing to the cache
- The additional search required in SMT processor to maintain load-load ordering contribute to 46% of total LQ searches

Background [All LQ searches in SMT]



In the SMT processor, the LQ is searched at:-

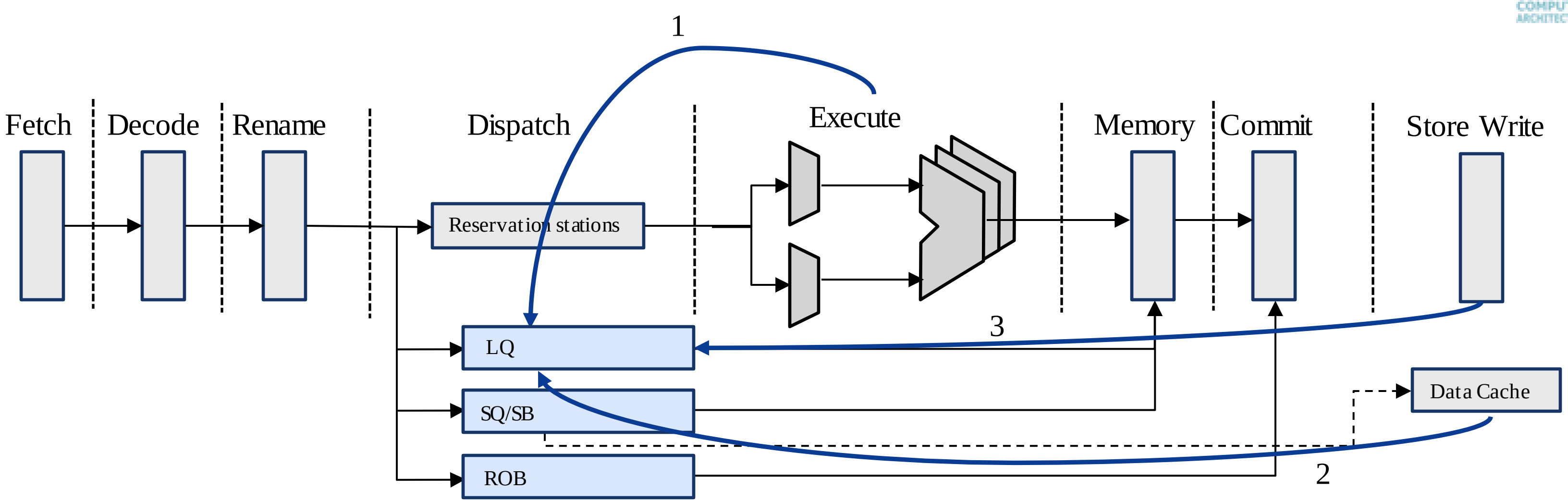
Background [All LQ searches in SMT]



In the SMT processor, the LQ is searched at:-

1. When the store resolves the address at execute stage (51%)

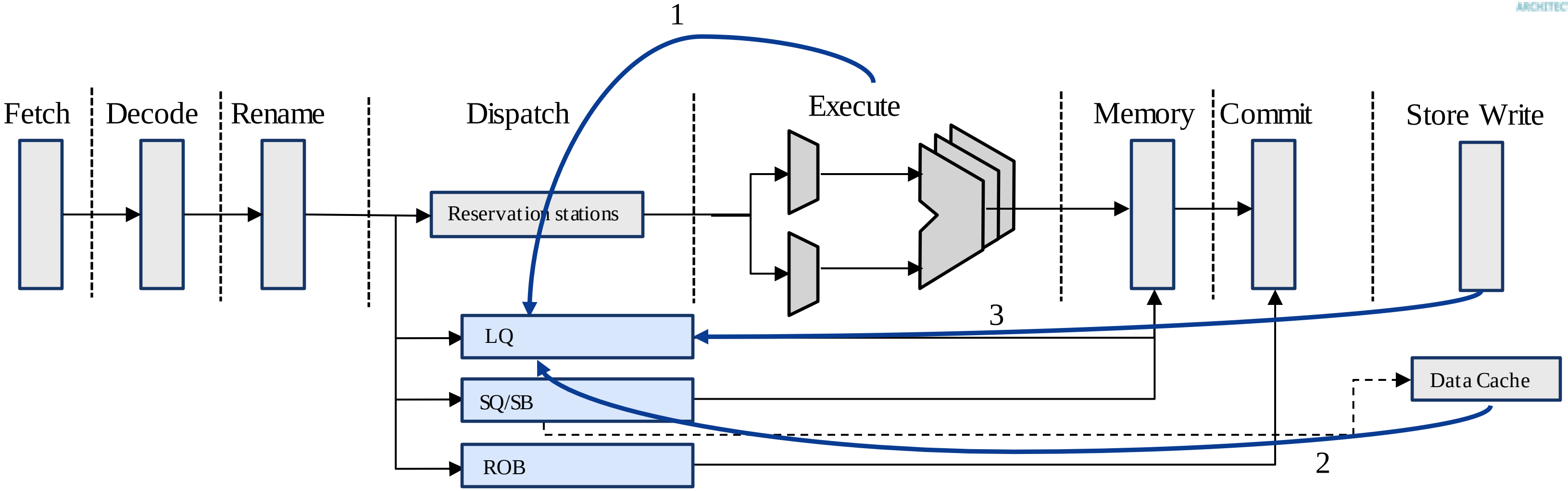
Background [All LQ searches in SMT]



In the SMT processor, the LQ is searched at:-

1. When the store resolves the address at execute stage (51%)
2. On cache invalidations and cache evictions (3%)

Background [All LQ searches in SMT]



In the SMT processor, the LQ is searched at:-

1. When the store resolves the address at execute stage (51%)
2. On cache invalidations and cache evictions (3%)
3. When stores write to cache (46%)

Outline

- Overview
- Background
- CELLO
- Evaluation
- Conclusion

CELLO [Design Overview]

- A software-hardware co-designed approach
- Leverages SC-for-DRF consistency model

CELLO [Design Overview]

- A software-hardware co-designed approach
- Leverages SC-for-DRF consistency model
- CELLO compiler classifies memory access within sync and DRF

CELLO [Design Overview]

- A software-hardware co-designed approach
- Leverages SC-for-DRF consistency model
- CELLO compiler classifies memory access within sync and DRF
- Compiler information is transmitted to the hardware by dedicated instruction

CELLO [Design Overview]

- A software-hardware co-designed approach
- Leverages SC-for-DRF consistency model
- CELLO compiler classifies memory access within sync and DRF
- Compiler information is transmitted to the hardware by dedicated instruction
- Based on the DRF information, CELLO,
  - Filters the LQ searches in the DRF region.
  - Facilitates early load exit from LQ.

CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i]  + 5;
}
```

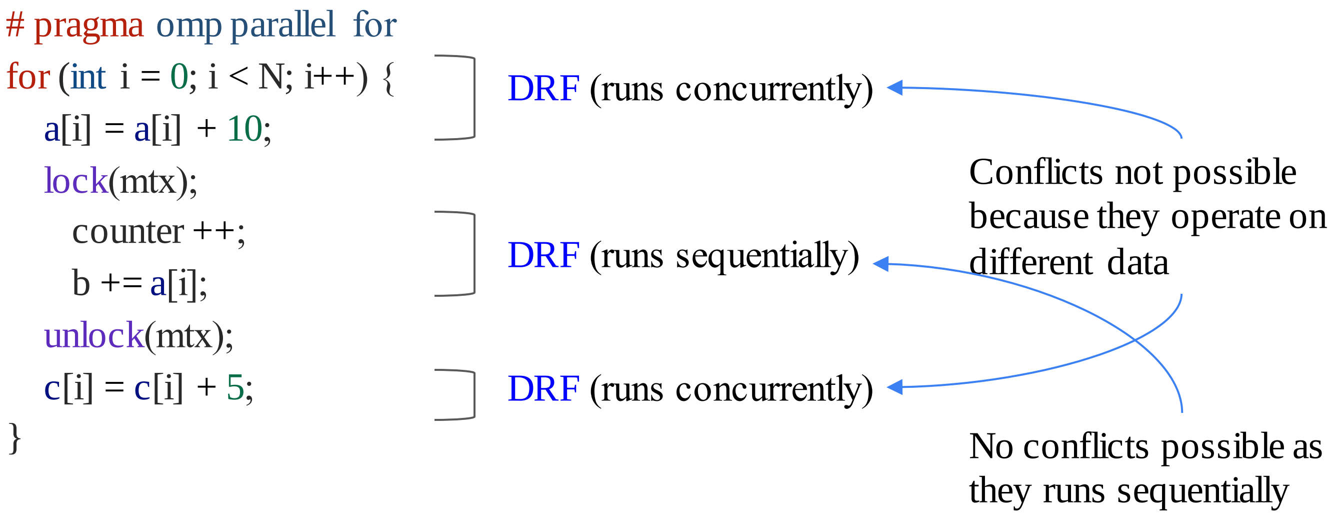
CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```

DRF (runs sequentially)

No conflicts possible as they runs sequentially

CELLO [Compiler]



```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
      counter ++;
      b += a[i];
    unlock(mtx);
    c[i] = c[i]  + 5;
}
```

DRF (runs concurrently)

DRF (runs sequentially)

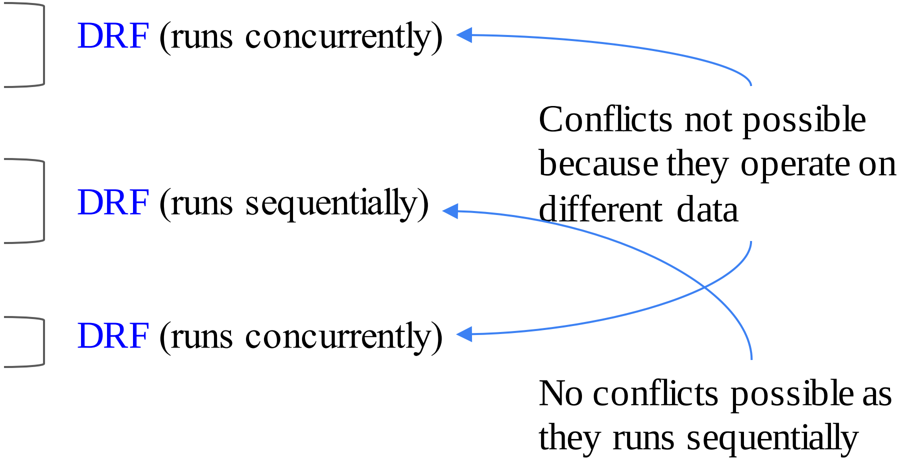
DRF (runs concurrently)

Conflicts not possible because they operate on different data

No conflicts possible as they runs sequentially

CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i]  + 5;
}
```



DRF (runs concurrently)

DRF (runs sequentially)

DRF (runs concurrently)

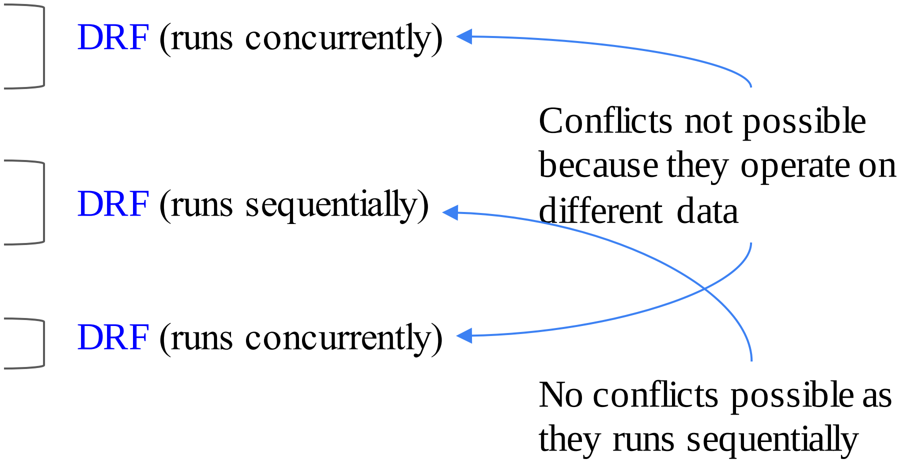
Conflicts not possible because they operate on different data

No conflicts possible as they runs sequentially

- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write

CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```



DRF (runs concurrently)

DRF (runs sequentially)

DRF (runs concurrently)

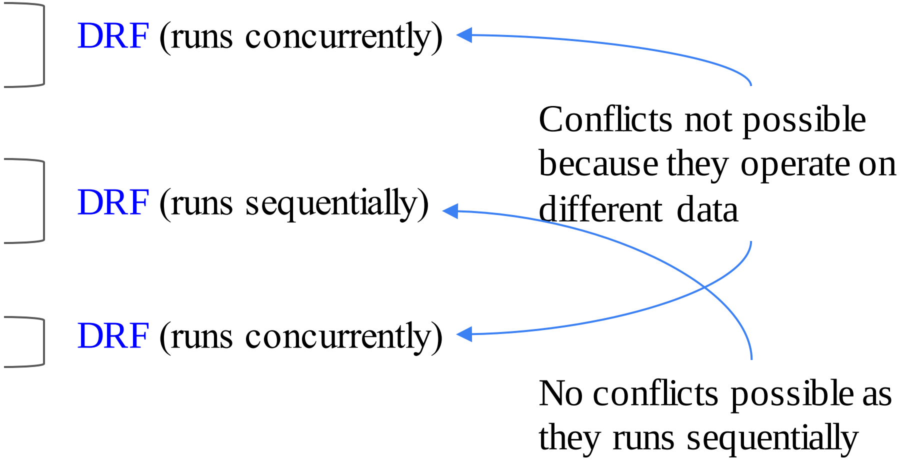
Conflicts not possible because they operate on different data

No conflicts possible as they runs sequentially

- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative

CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i]  + 5;
}
```



DRF (runs concurrently)

DRF (runs sequentially)

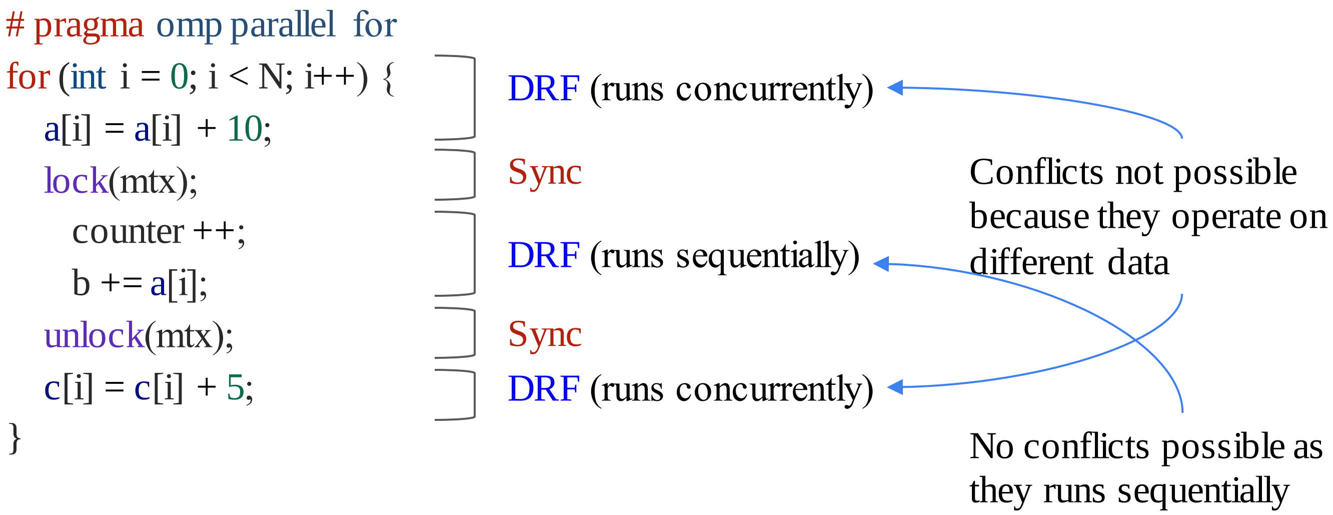
DRF (runs concurrently)

Conflicts not possible because they operate on different data

No conflicts possible as they runs sequentially

- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative
  2. No LQ search is required to maintain load-load ordering in a DRF region

CELLO [Compiler]



```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```

DRF (runs concurrently)

Sync

DRF (runs sequentially)

Sync

DRF (runs concurrently)

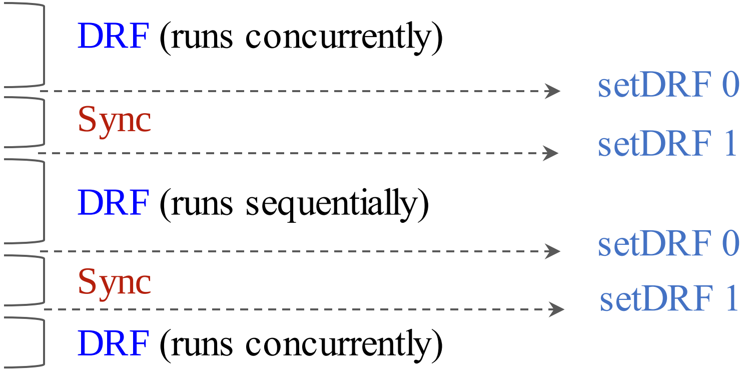
Conflicts not possible because they operate on different data

No conflicts possible as they runs sequentially

- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative
  2. No LQ search is required to maintain load-load ordering in a DRF region
- ➢ All non-DRF regions are sync regions and load-load ordering should be respected in TSO

CELLO [Compiler]

```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```



DRF (runs concurrently)

setDRF 0

Sync

setDRF 1

DRF (runs sequentially)

setDRF 0

Sync

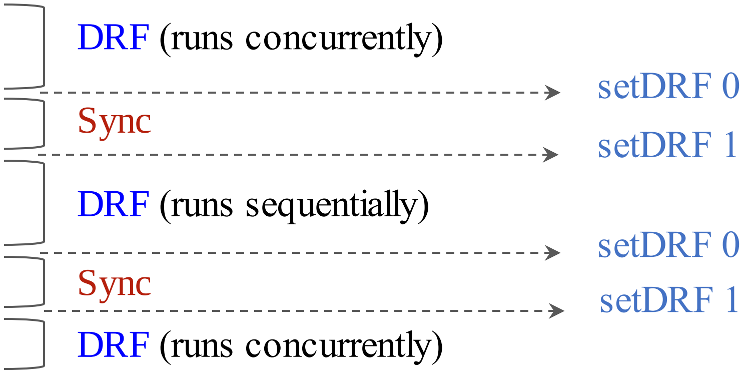
setDRF 1

DRF (runs concurrently)

- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative
  2. No LQ search is required to maintain load-load ordering in a DRF region
- ➢ All non-DRF regions are sync regions and load-load ordering should be respected in TSO
- ➢ CELLO delineates DRF and sync regions by setDRF instruction

CELLO [Compiler]

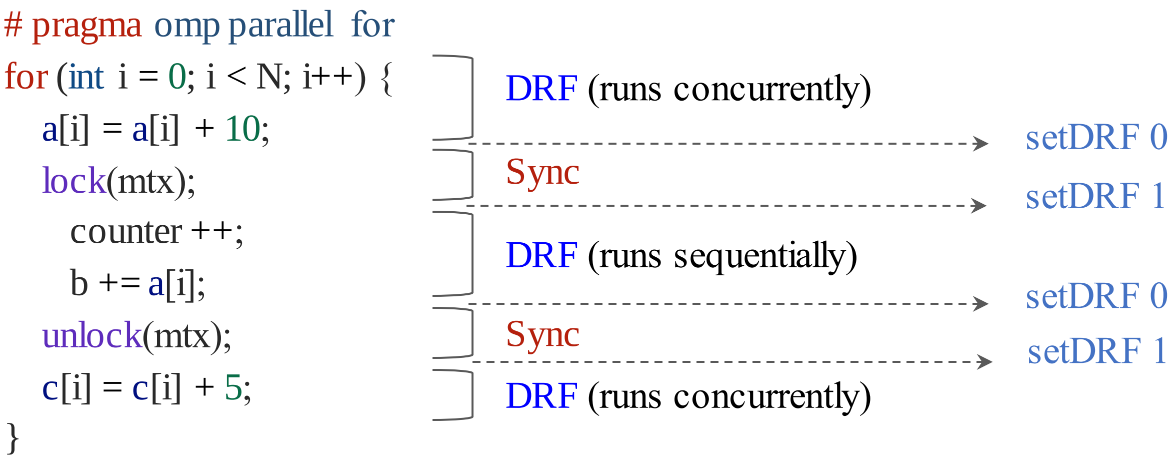
```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```



- ➢ In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative
  2. No LQ search is required to maintain load-load ordering in a DRF region
- ➢ All non-DRF regions are sync regions and load-load ordering should be respected in TSO
- ➢ CELLO delineates DRF and sync regions by setDRF instruction

  setDRF 1 : Start of DRF region

CELLO [Compiler]



```
# pragma omp parallel  for
for (int  i = 0; i < N; i++) {
    a[i] = a[i]  + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```

DRF (runs concurrently)

setDRF 0

Sync

setDRF 1

DRF (runs sequentially)

setDRF 0

Sync

setDRF 1

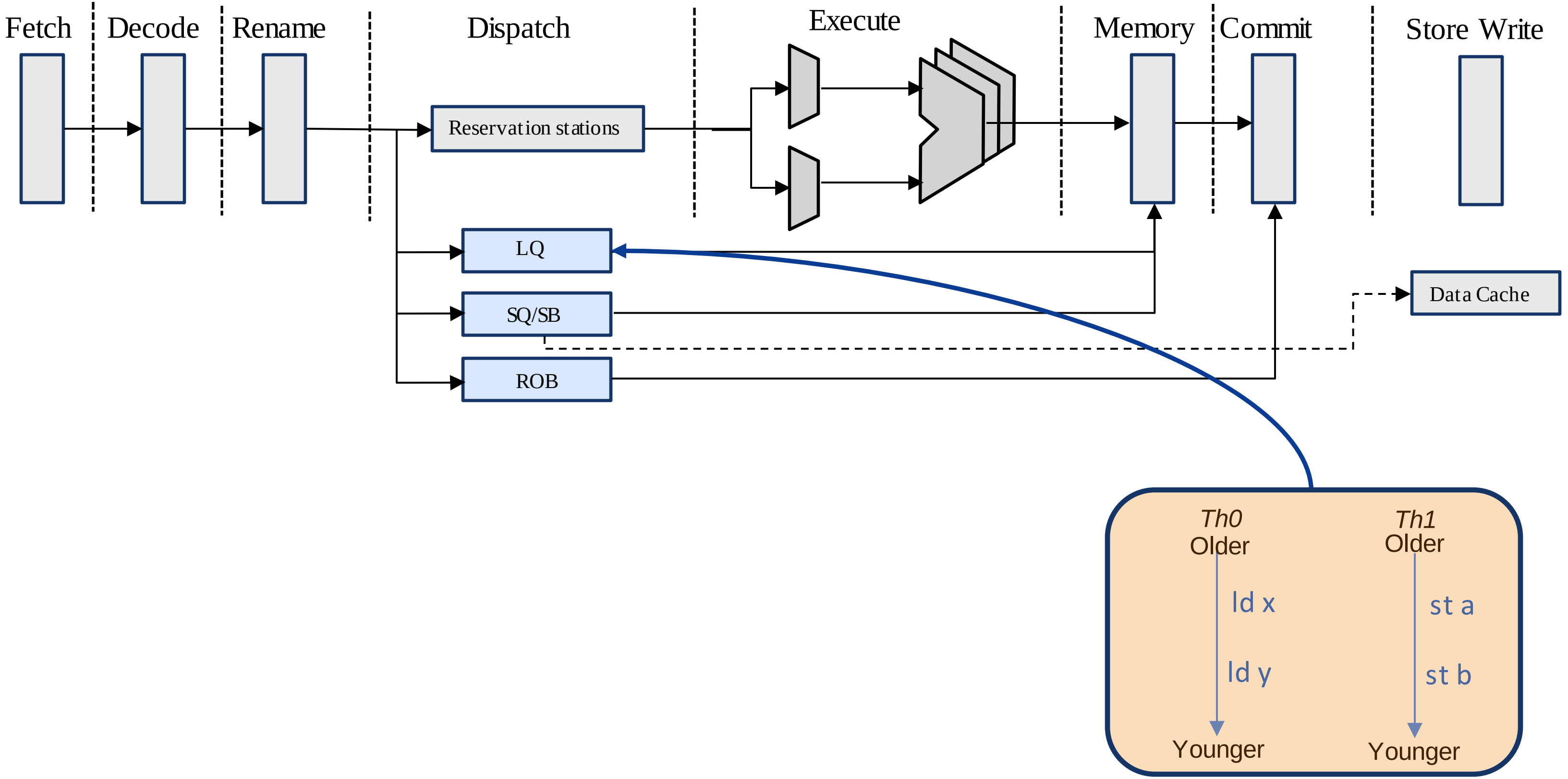
DRF (runs concurrently)

- In DRF regions no thread/core can perform concurrently to the same memory location if one of them is write
  1. Loads in DRF regions can perform OoO without breaking TSO guarantees, they are non M-Speculative
  2. No LQ search is required to maintain load-load ordering in a DRF region
- All non-DRF regions are sync regions and load-load ordering should be respected in TSO
- CELLO delineates DRF and sync regions by setDRF instruction

  setDRF 1 : Start of DRF region

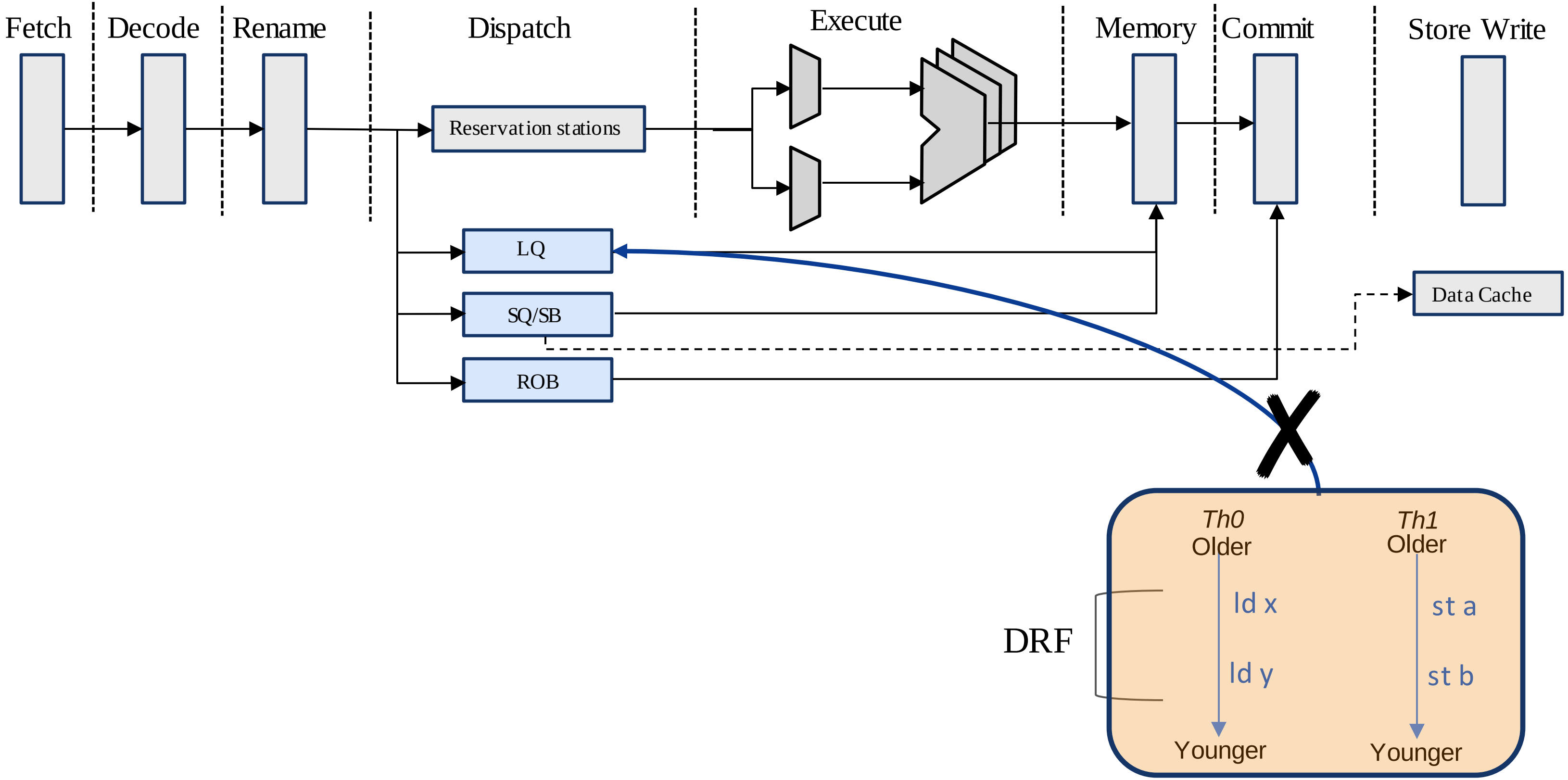
  setDRF 0 : End of DRF region

CELLO [Compiler]



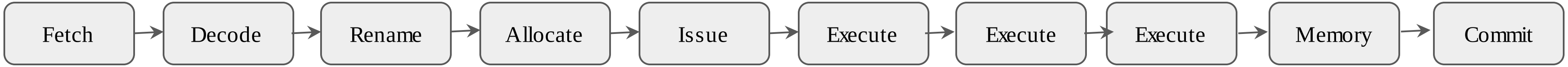
CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions @ PACT’23

CELLO [Compiler]

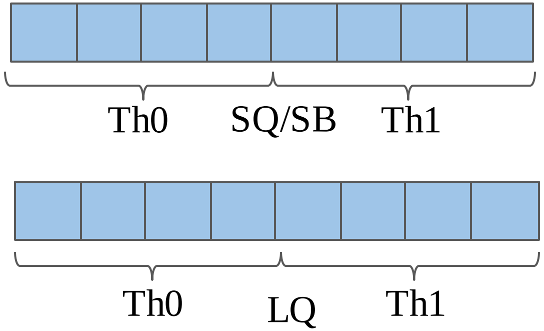


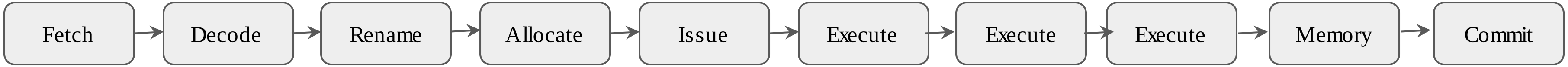
CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions @ PACT'23

CELLO [u-Arch design]

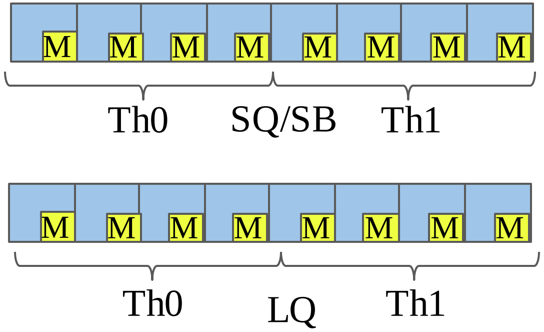


CELLO [u-Arch design]



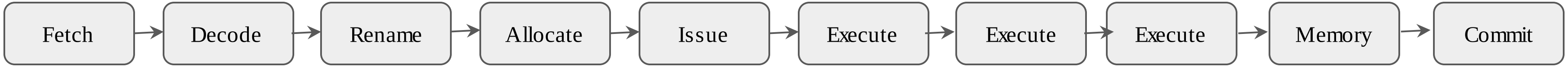


CELLO [u-Arch design]

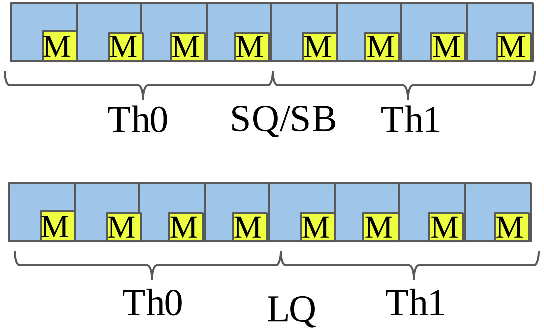


Mode bit (M)

- 1 bit per entry in SQ/SB and LQ
- Keeps DRF info per entry
- M = 1 : DRF entry
- M = 0 : Non-DRF (sync) entry

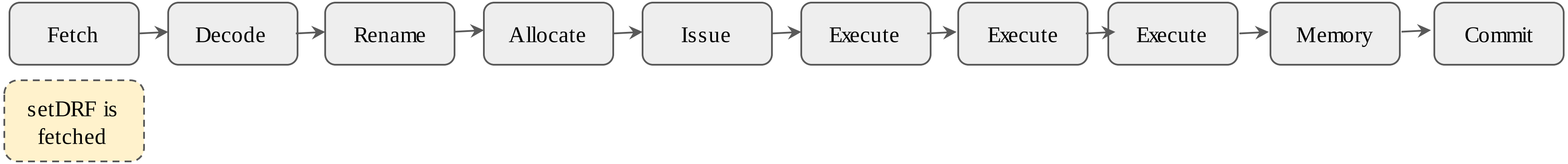


CELLO [u-Arch design]



Mode bit (M)

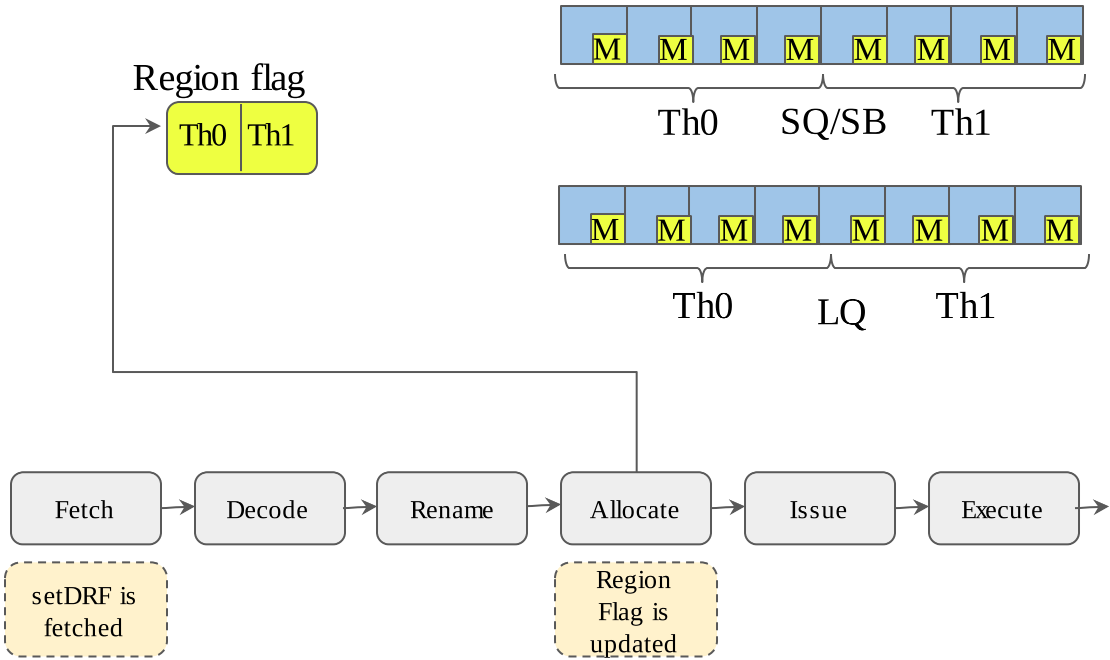
- ➢ 1 bit per entry in SQ/SB and LQ
- ➢ Keeps DRF info per entry
- ➢ M = 1 : DRF entry
- ➢ M = 0 : Non-DRF (sync) entry



setDRF is fetched

CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions @ PACT’23

CELLO [u-Arch design]



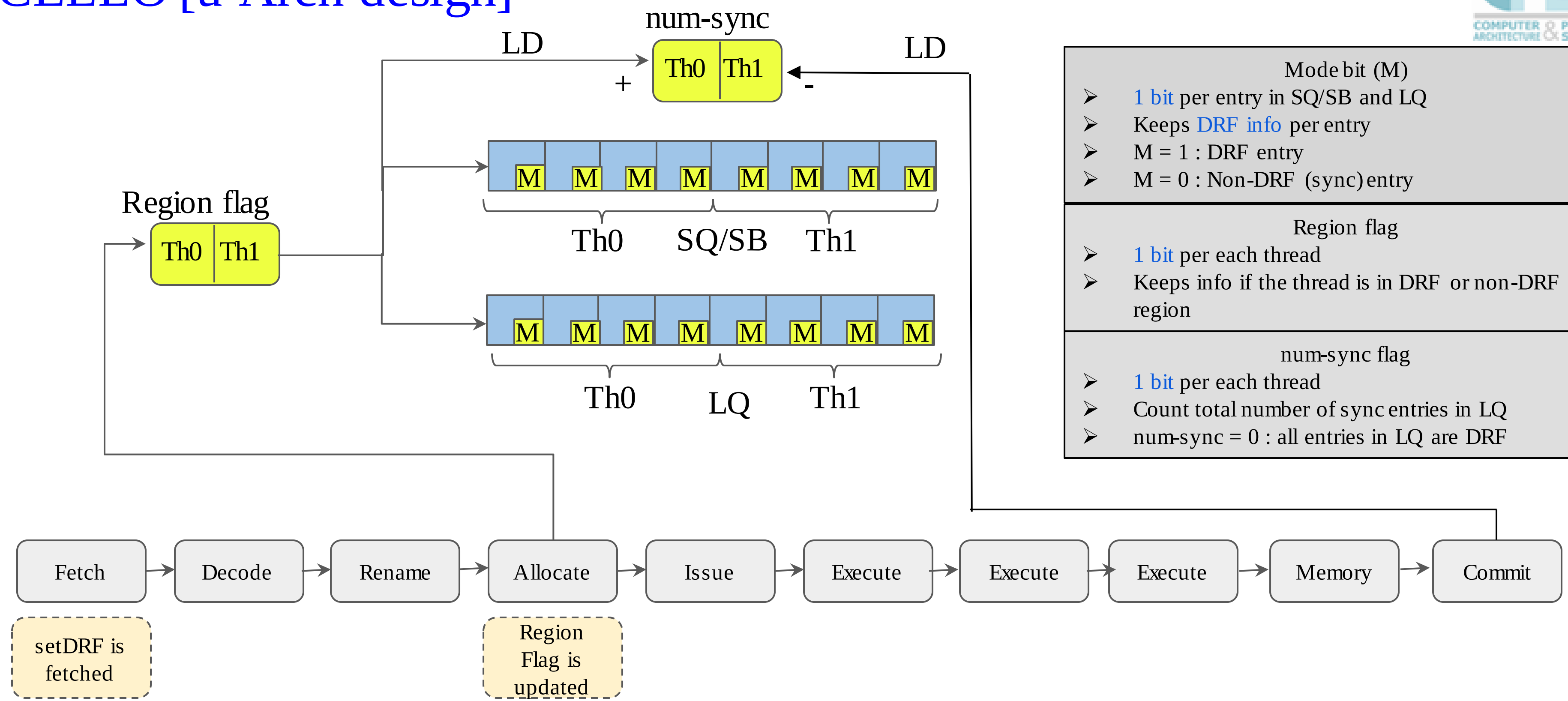
Mode bit (M)

- 1 bit per entry in SQ/SB and LQ
- Keeps DRF info per entry
- M = 1 : DRF entry
- M = 0 : Non-DRF (sync) entry

Region flag

- 1 bit per each thread
- Keeps info if the thread is in DRF or non-DRF region

CELLO [u-Arch design]



Mode bit (M)

- 1 bit per entry in SQ/SB and LQ
- Keeps DRF info per entry
- M = 1 : DRF entry
- M = 0 : Non-DRF (sync) entry

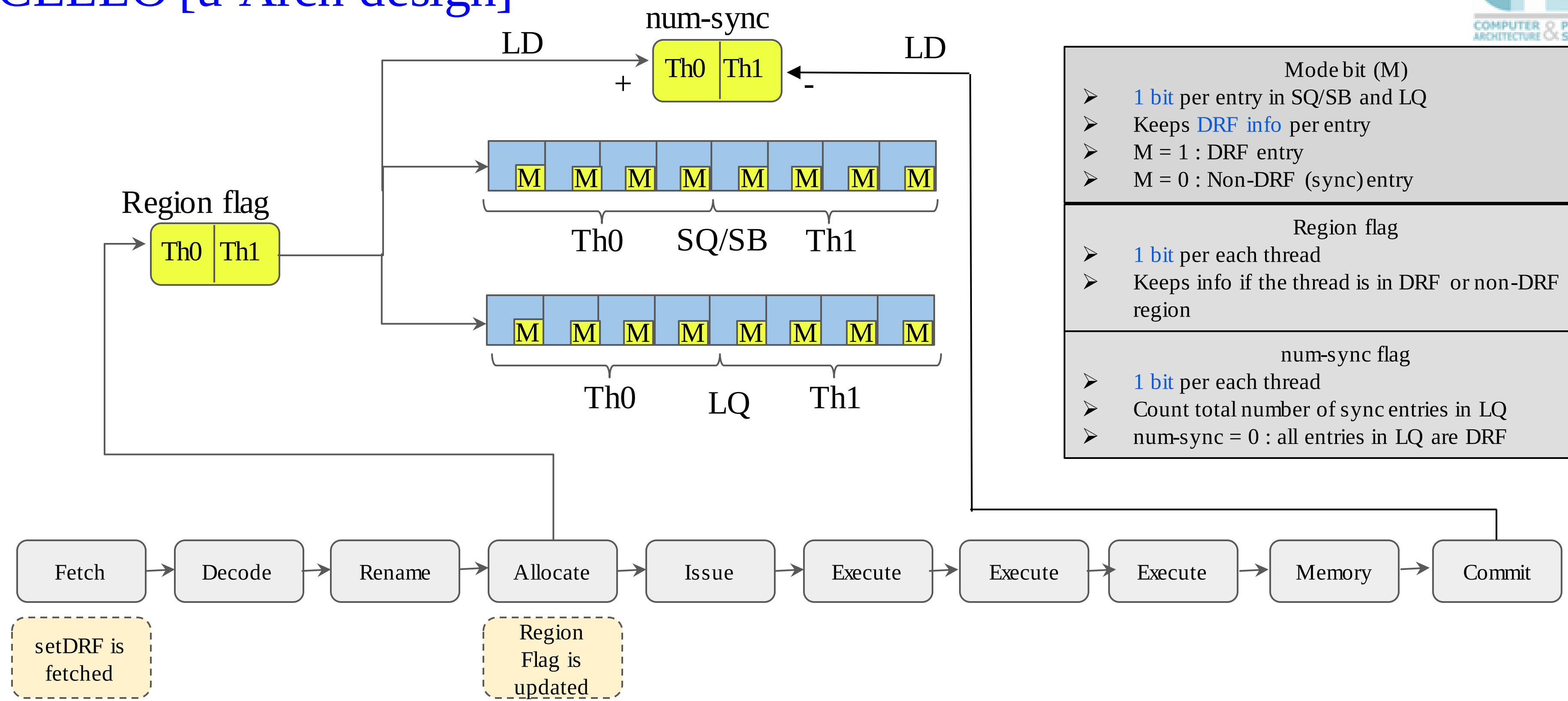
Region flag

- 1 bit per each thread
- Keeps info if the thread is in DRF or non-DRF region

num-sync flag

- 1 bit per each thread
- Count total number of sync entries in LQ
- num-sync = 0 : all entries in LQ are DRF

CELLO [u-Arch design]



Mode bit (M)

- 1 bit per entry in SQ/SB and LQ
- Keeps DRF info per entry
- M = 1 : DRF entry
- M = 0 : Non-DRF (sync) entry

Region flag

- 1 bit per each thread
- Keeps info if the thread is in DRF or non-DRF region

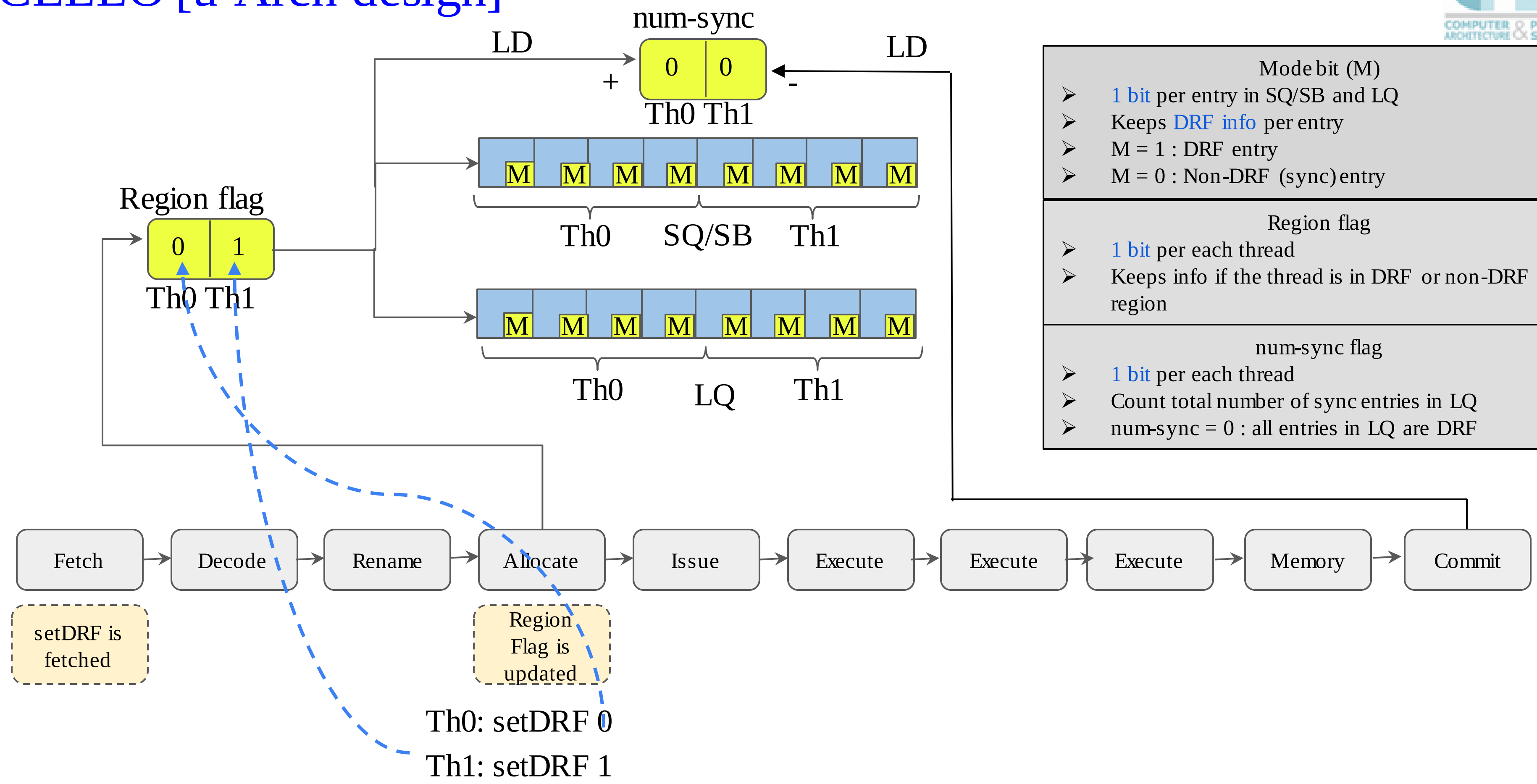
num-sync flag

- 1 bit per each thread
- Count total number of sync entries in LQ
- num-sync = 0 : all entries in LQ are DRF

Th0: setDRF 0

Th1: setDRF 1

CELLO [u-Arch design]



Mode bit (M)

- ➢ 1 bit per entry in SQ/SB and LQ
- ➢ Keeps DRF info per entry
- ➢ M = 1 : DRF entry
- ➢ M = 0 : Non-DRF (sync) entry

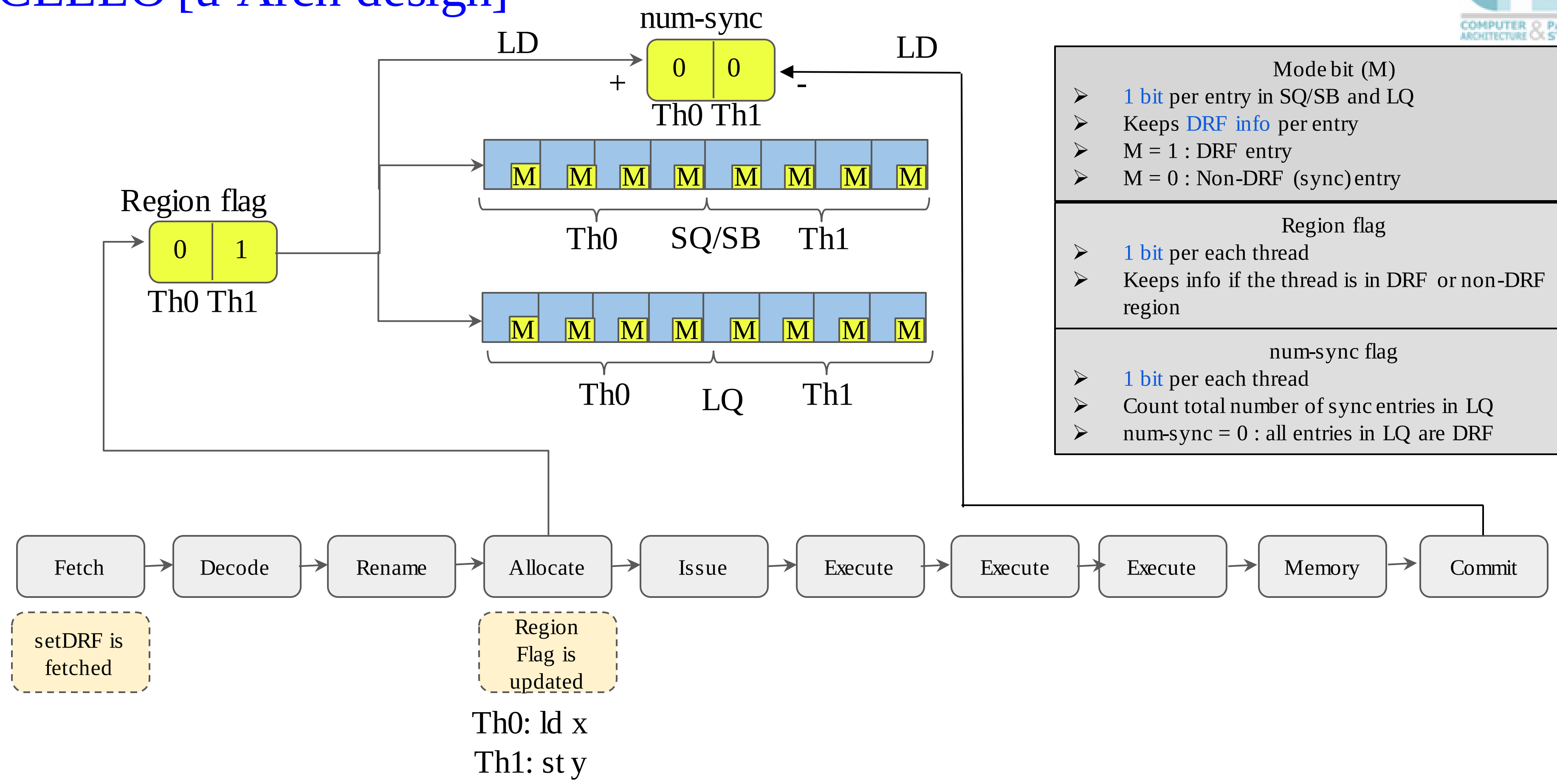
Region flag

- ➢ 1 bit per each thread
- ➢ Keeps info if the thread is in DRF or non-DRF region

num-sync flag

- ➢ 1 bit per each thread
- ➢ Count total number of sync entries in LQ
- ➢ num-sync = 0 : all entries in LQ are DRF

CELLO [u-Arch design]



Mode bit (M)

- ➢ 1 bit per entry in SQ/SB and LQ
- ➢ Keeps DRF info per entry
- ➢ M = 1 : DRF entry
- ➢ M = 0 : Non-DRF (sync) entry

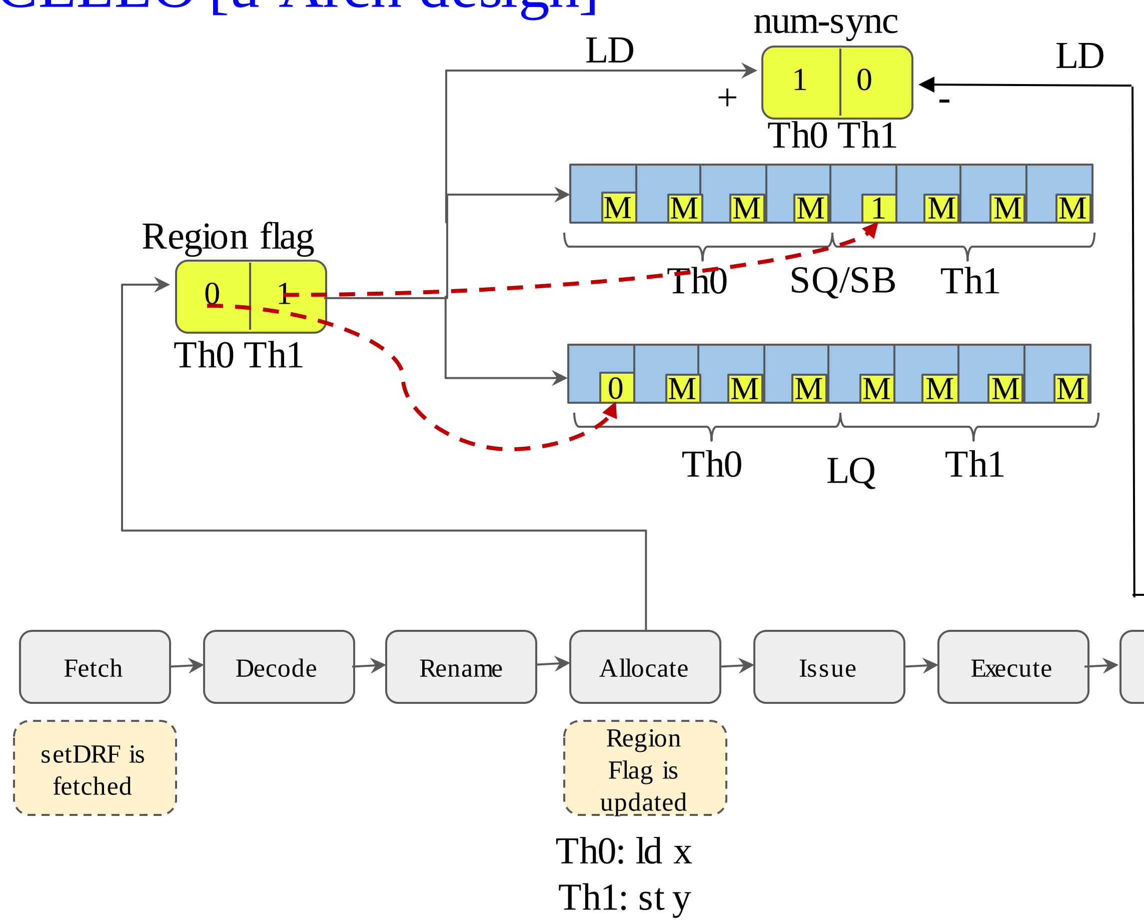
Region flag

- ➢ 1 bit per each thread
- ➢ Keeps info if the thread is in DRF or non-DRF region

num-sync flag

- ➢ 1 bit per each thread
- ➢ Count total number of sync entries in LQ
- ➢ num-sync = 0 : all entries in LQ are DRF

CELLO [u-Arch design]



Th0: ld x

Th1: st y

Mode bit (M)

- 1 bit per entry in SQ/SB and LQ
- Keeps DRF info per entry
- M = 1 : DRF entry
- M = 0 : Non-DRF (sync) entry

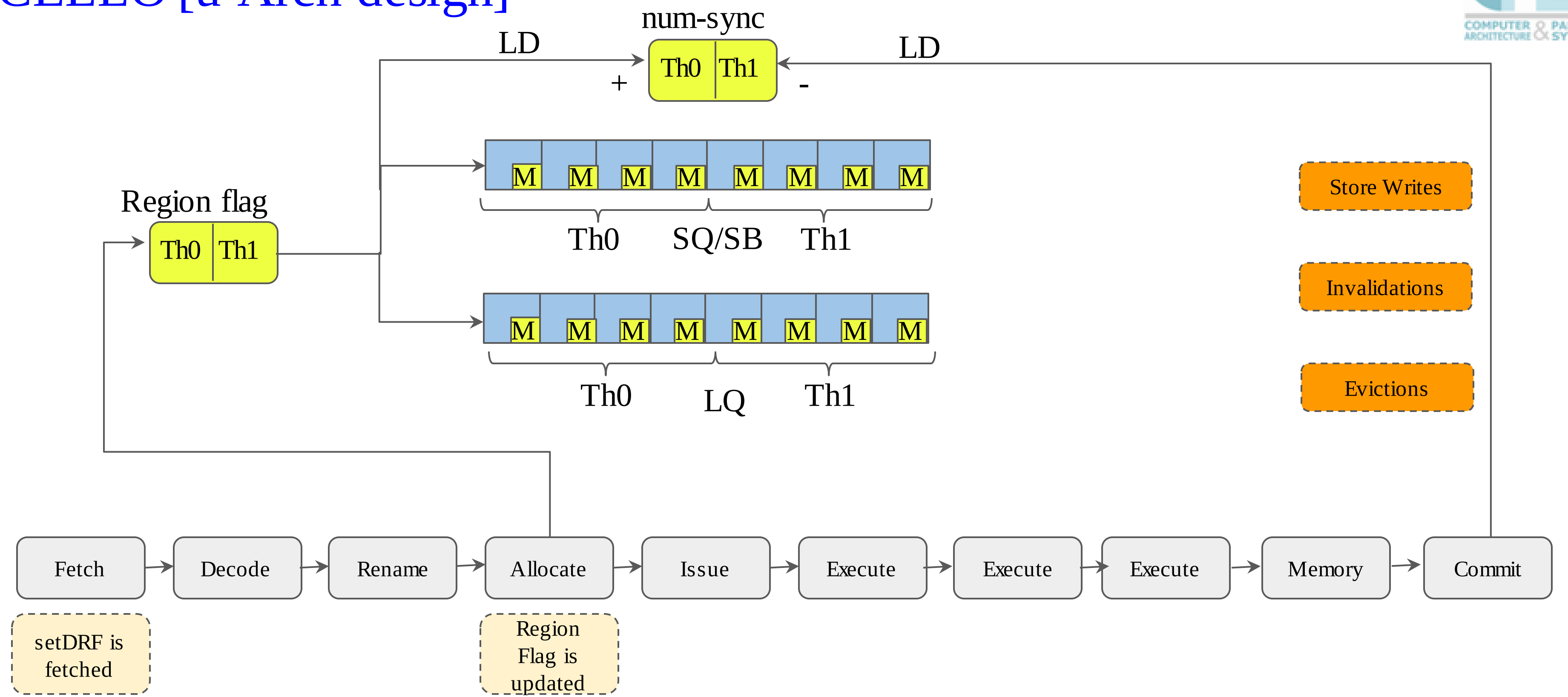
Region flag

- 1 bit per each thread
- Keeps info if the thread is in DRF or non-DRF region

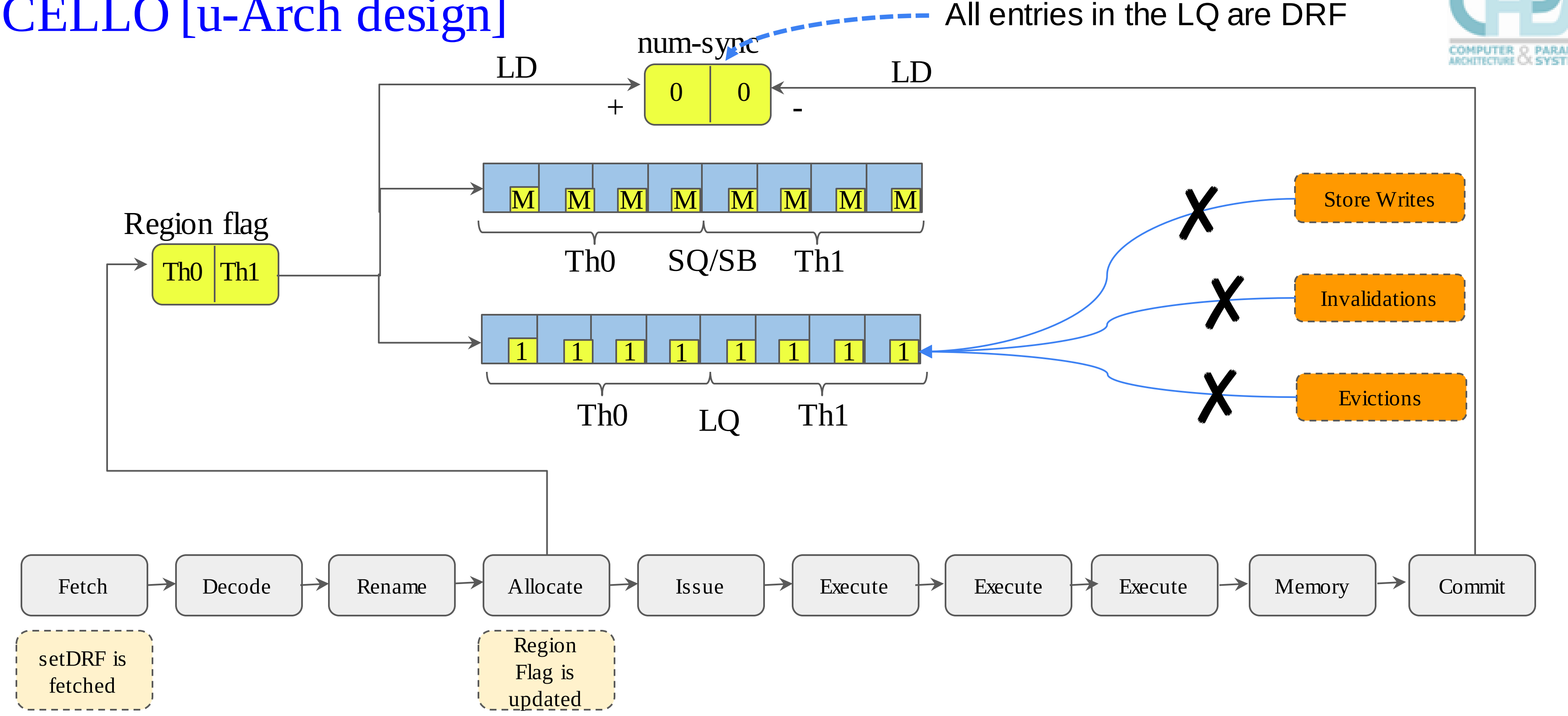
num-sync flag

- 1 bit per each thread
- Count total number of sync entries in LQ
- num-sync = 0 : all entries in LQ are DRF

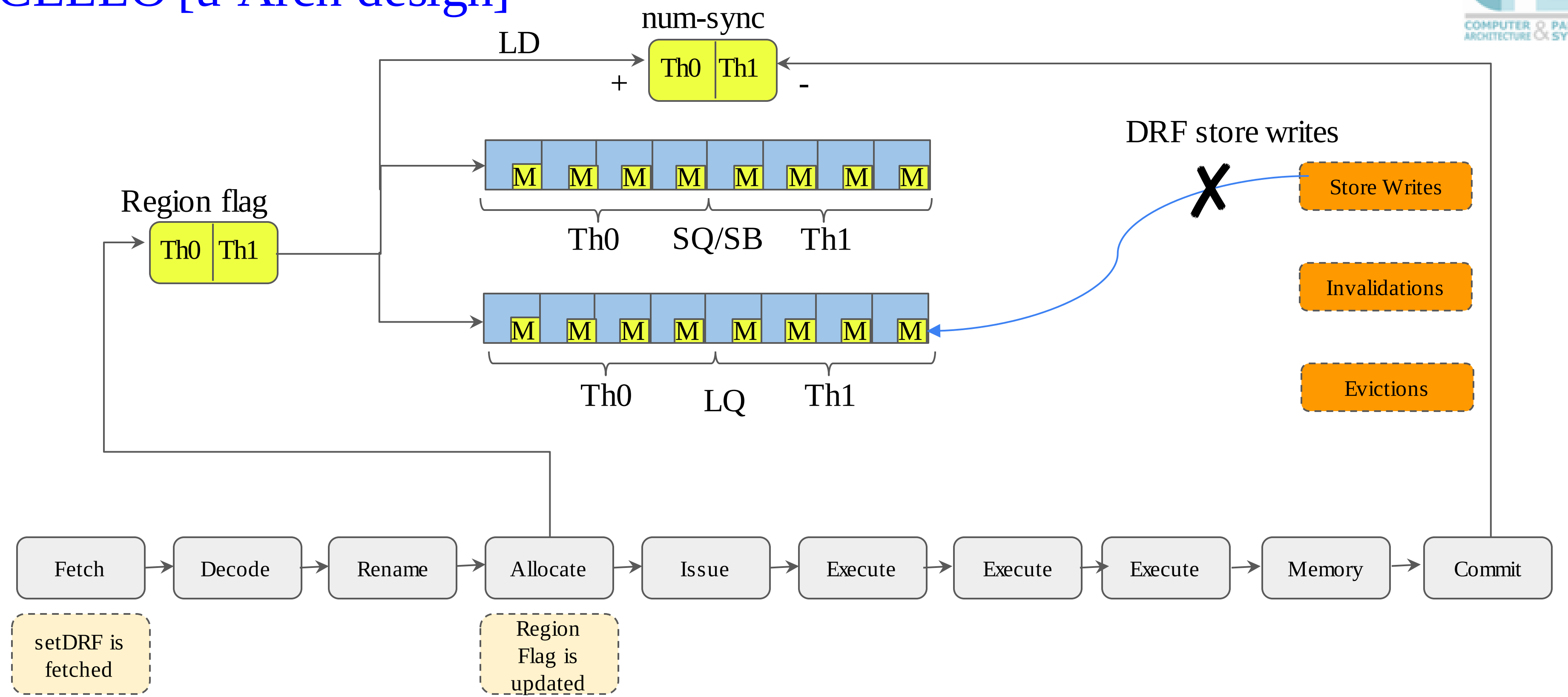
CELLO [u-Arch design]



CELLO [u-Arch design]

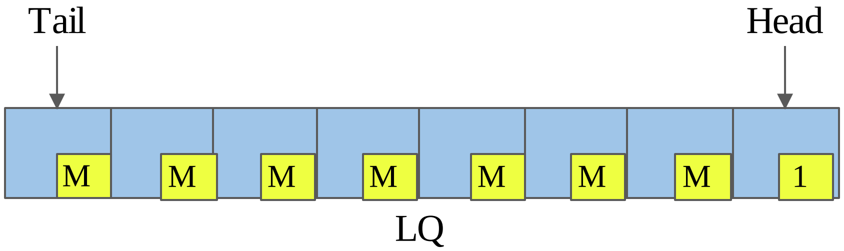


CELLO [u-Arch design]



CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions @ PACT'23

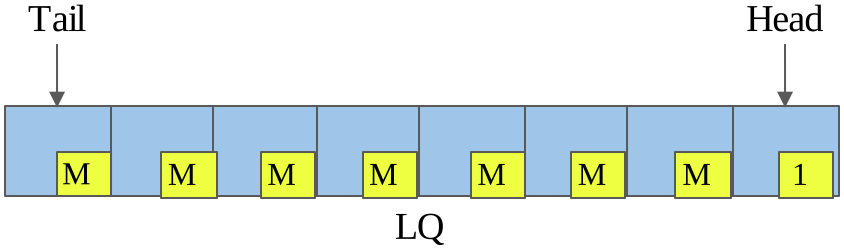
CELLO [u-Arch design, early removal of loads]



LQ head is safe to remove when

- LQ head becomes non M-Spec
- LQ head becomes non D-Spec

CELLO [u-Arch design, early removal of loads]



LQ head is safe to remove when

- ➢ LQ head becomes non M-Spec (DRF Loads are M-Speculative by default)
- ➢ LQ head becomes non D-Spec

CELLO [Recap]

- ➔ CELLO provides a simple design to filter M-spec LQ searches in SMT processors

- ➔ CELLO allows the DRF load to be removed early from the LQ head if all older stores have resolved the address and already searched the LQ

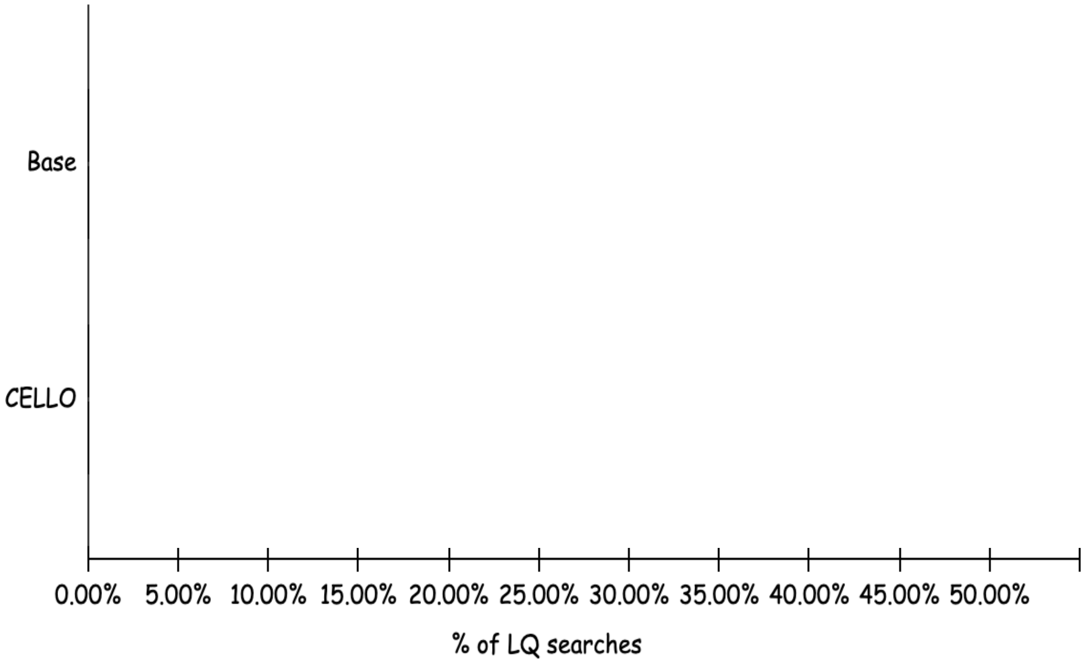
Outline

- Overview
- Background
- CELLO
- Evaluation
- Conclusion

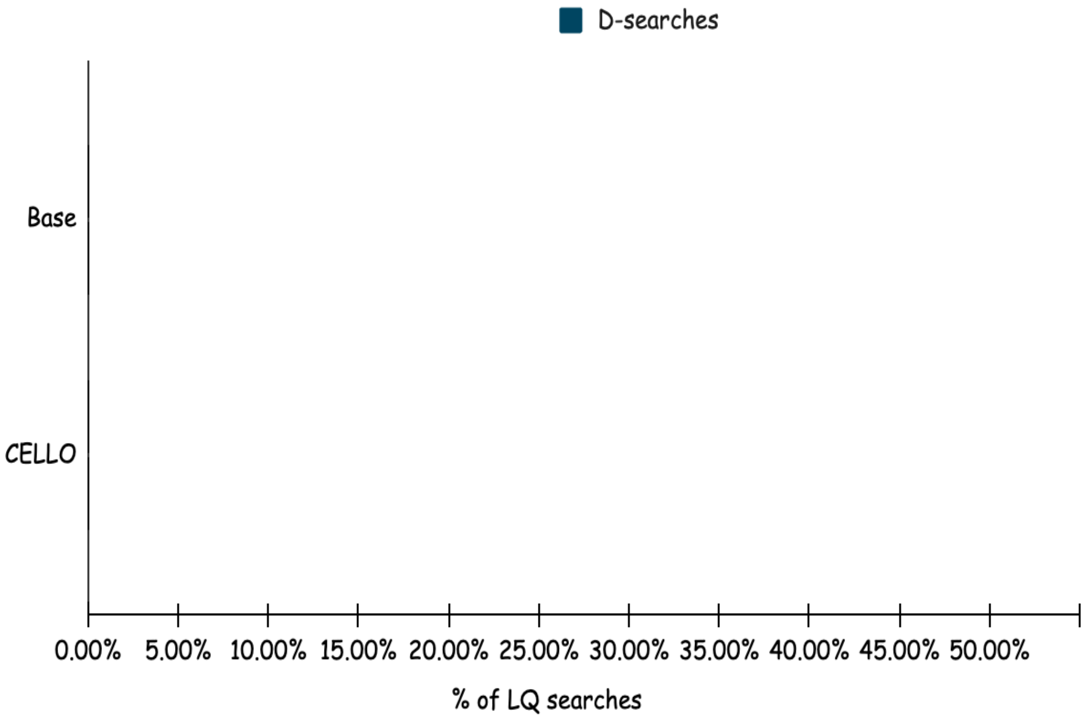
Evaluation

- ➔ Detailed In-house out-of-order SMT processor model
- ➔ Uses Sniper as front end and GEMS for memory model
- ➔ Standard invalidation-based directory protocol using GARNET
- ➔ TSO like consistency
- ➔ Intel Alder Lake micro-architecture
- ➔ CACTI-P is used to model energy consumption
- ➔ Splash-3, PARSEC 3.0, and six fine-grain synchronization-intensive applications are used as benchmarks

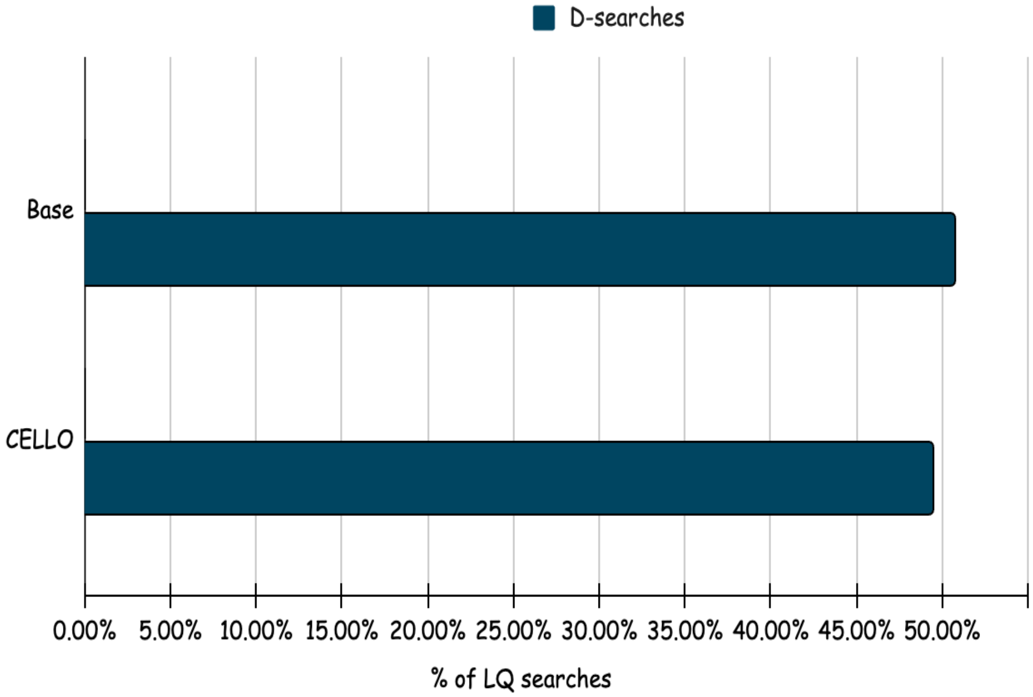
Evaluation [LQ Searches]



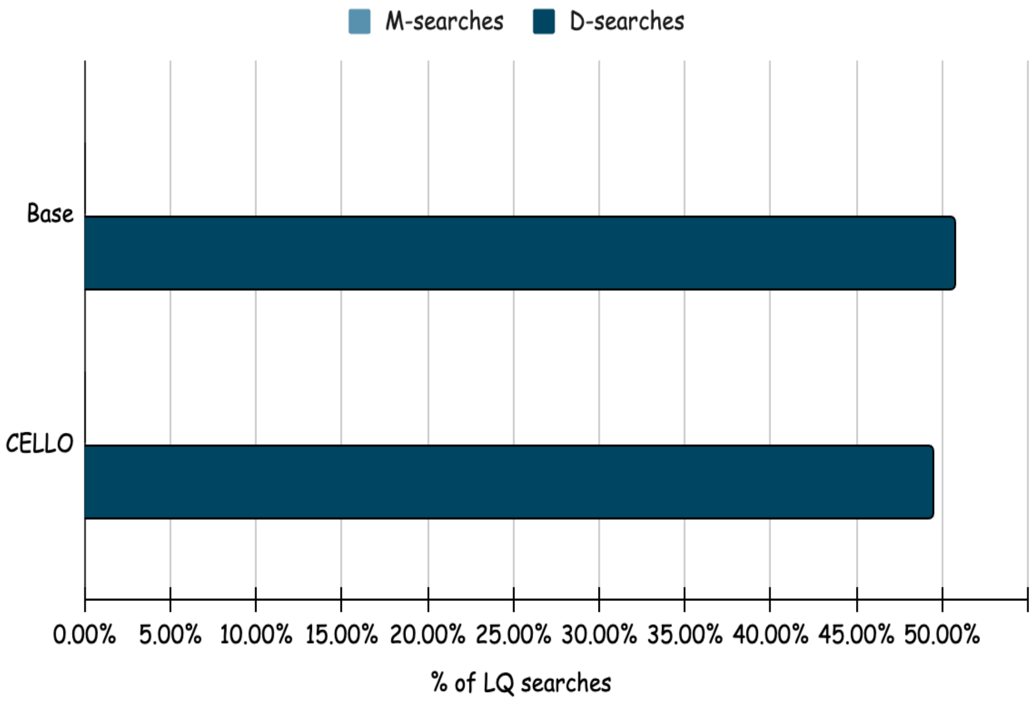
Evaluation [LQ Searches]



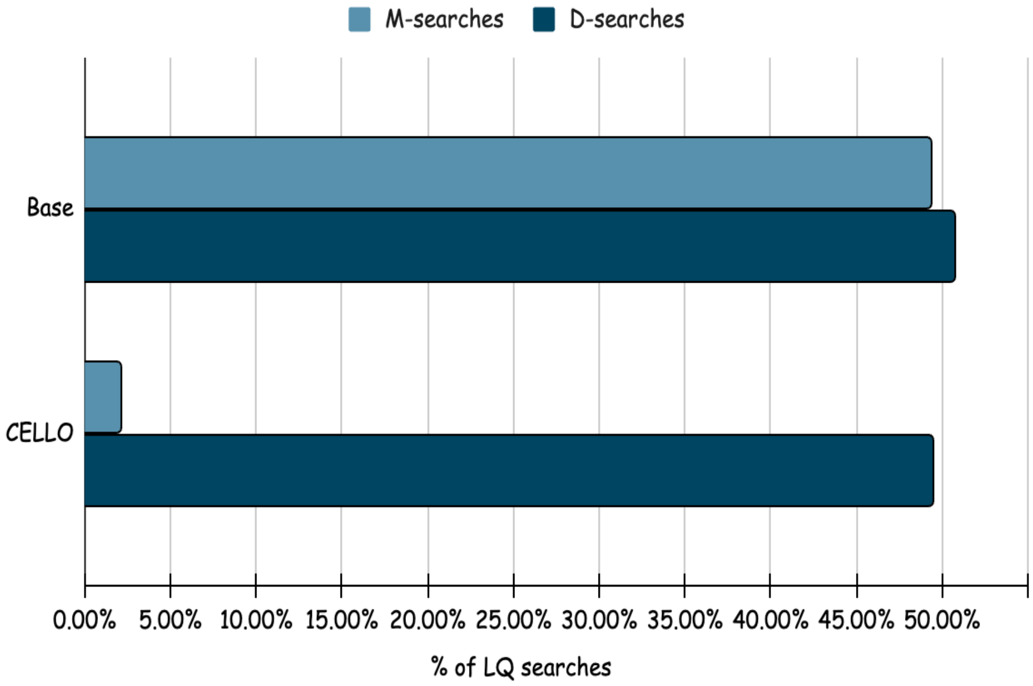
Evaluation [LQ Searches]



Evaluation [LQ Searches]

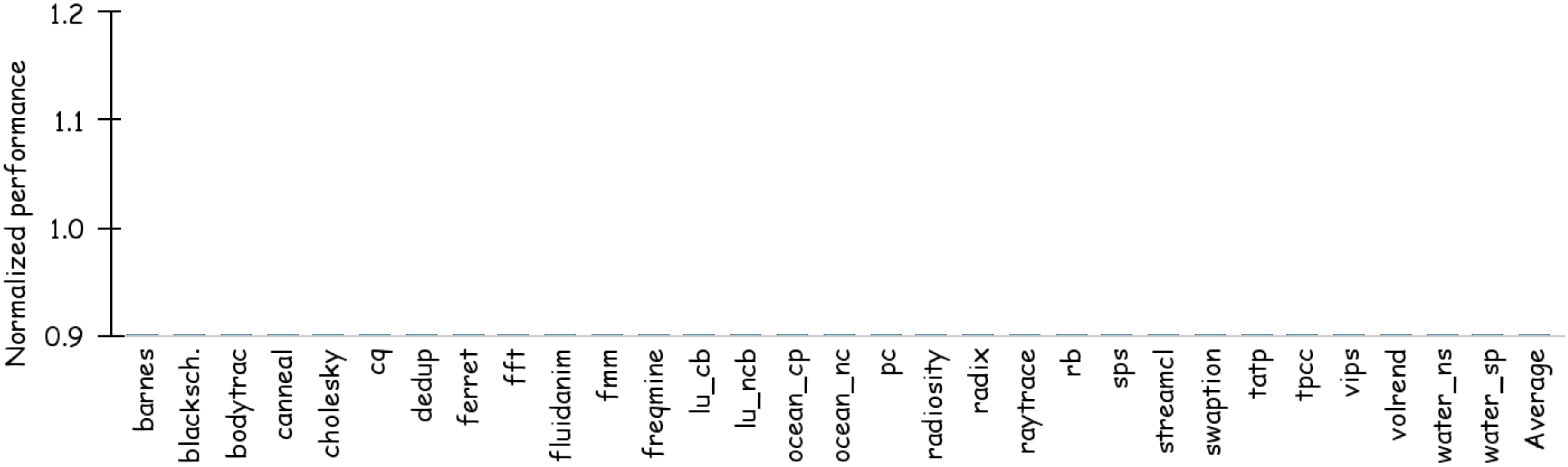


Evaluation [LQ Searches]

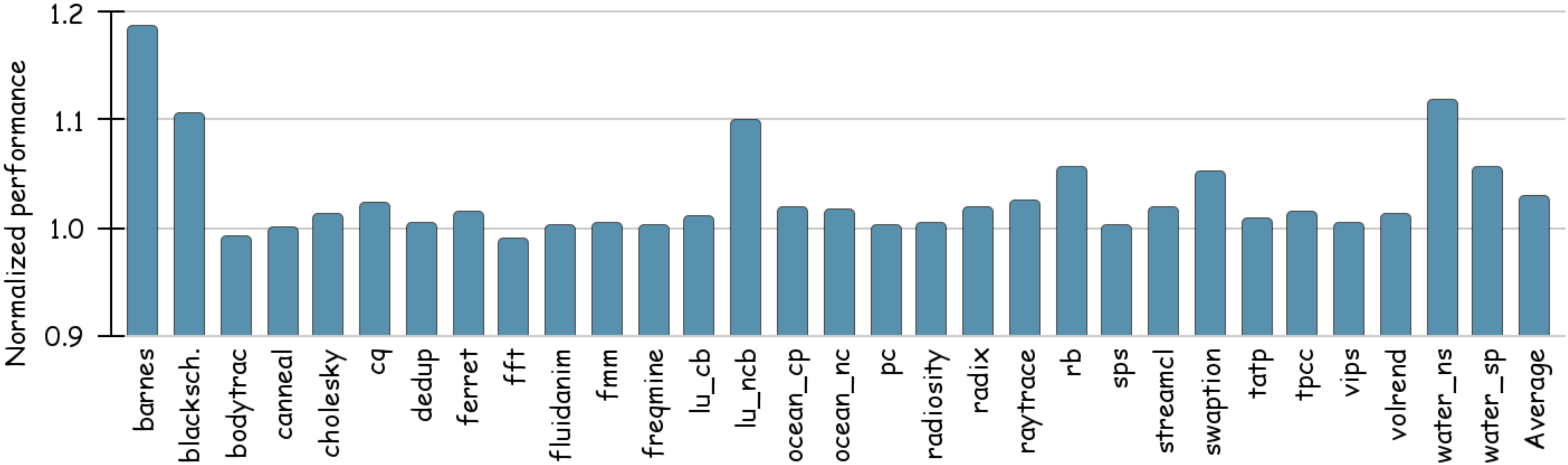


- ➔ M-speculative LQ searches are almost eliminated
- ➔ Overall, 47% of LQ searches are filtered by CELLO

Evaluation [Execution time]

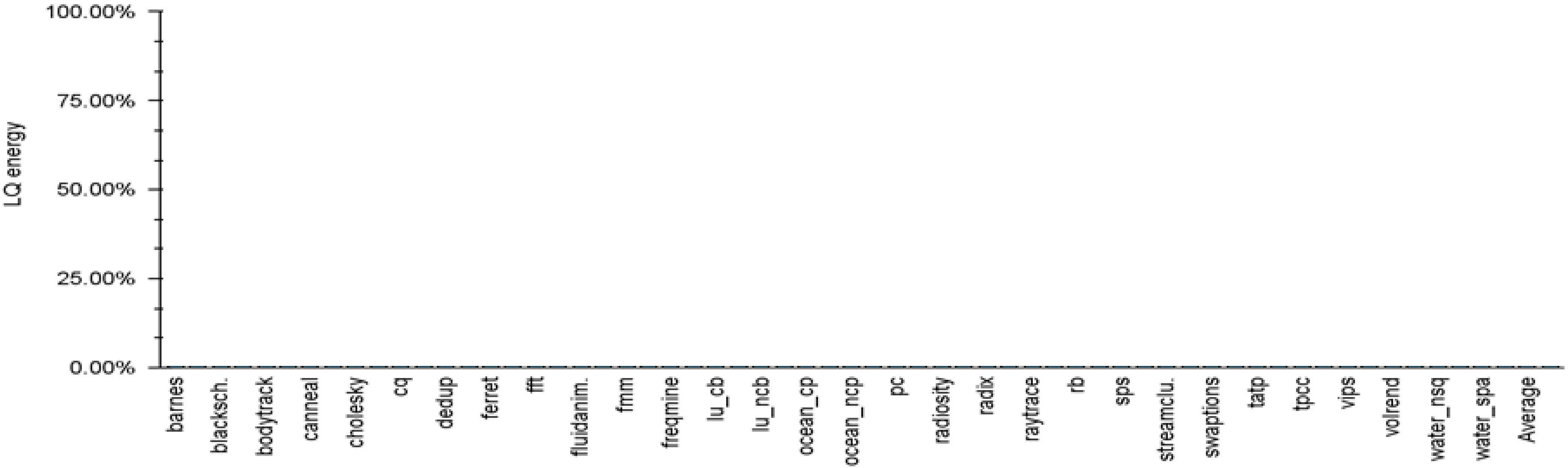


Evaluation [Execution time]



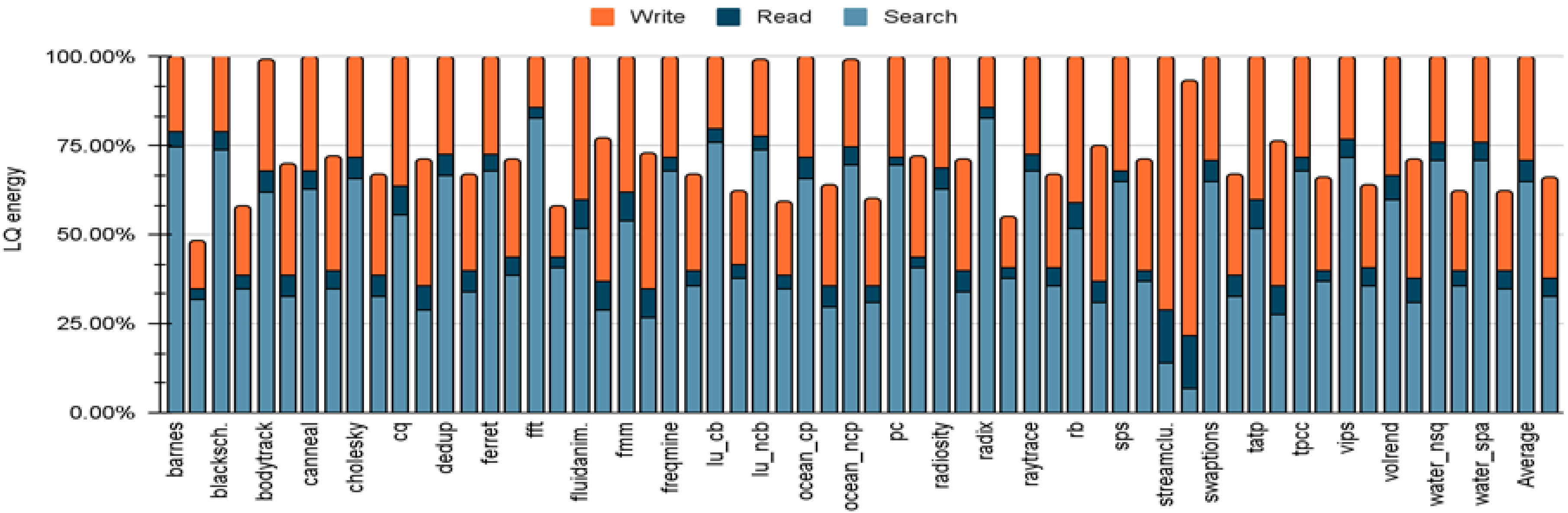
- ➔ LQ search filtering helps reduce the LQ search port contention
- ➔ Removing loads early helps in some applications
- ➔ CELLO provide a speed up of 2.8% on average

Evaluation [LQ energy]



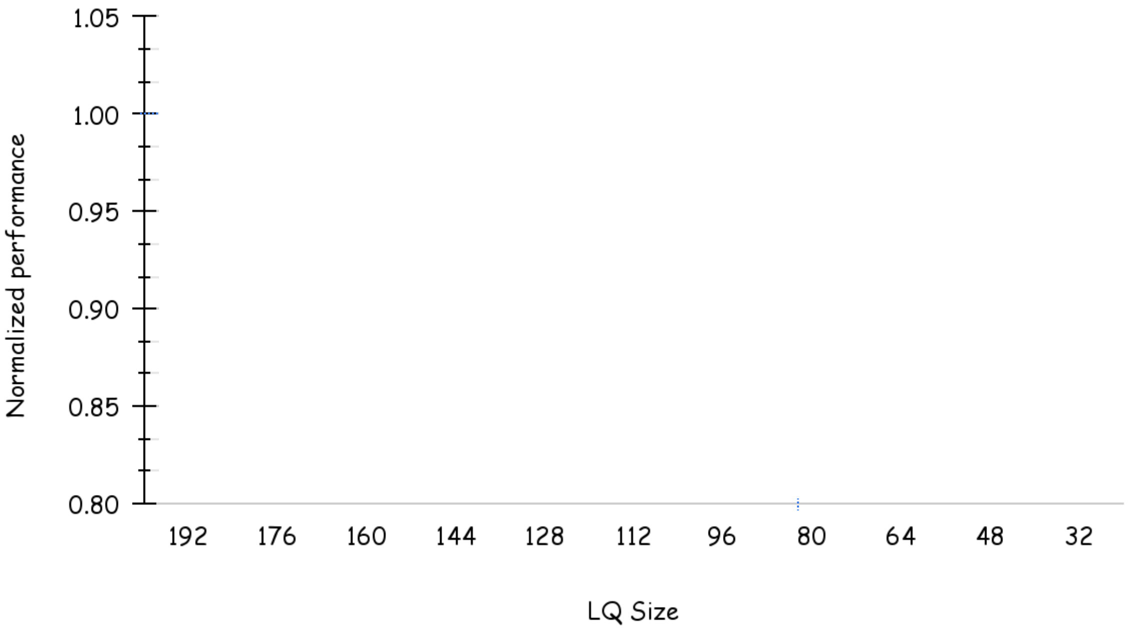
- ➔ Searches account for 65% of LQ energy consumption
- ➔ As CELLO filter most of the M-sepc search, the reduction in LQ energy expenditure is about 33%

Evaluation [LQ energy]

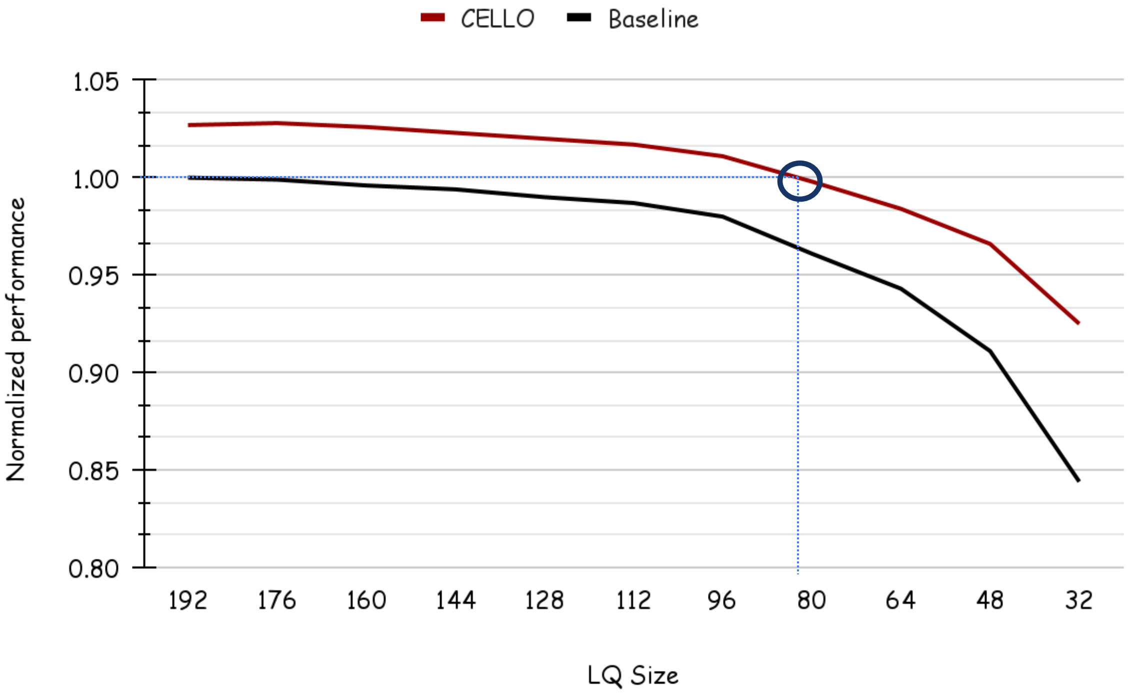


- ➔ Searches account for 65% of LQ energy consumption
- ➔ As CELLO filter most of the M-sepc search, the reduction in LQ energy expenditure is about 33%

Evaluation [Sensitivity analysis]



Evaluation [Sensitivity analysis]



Key observations:-

- ➔ Smaller LQ benefits from low energy consumption
- ➔ CELLO offers a design space with a smaller LQ size without compromising the performance when compared to the baseline without CELLO with 192 entries LQ
- ➔ CELLO managed to reduce the LQ size from 192 to 80 while providing the same performance

Conclusion

- ➔ The compiler can help optimize hardware

Conclusion

- ➔ The compiler can help optimize hardware
- ➔ SMT suffers from extensive LQ searches

Conclusion

➔ The compiler can help optimize hardware

➔ SMT suffers from extensive LQ searches

➔ CELLO can

1. Avoid LQ searches by 47%
2. Provide a speedup of 2.8% (up to 18.6%)
3. Reduce the LQ energy consumption by 33%

Conclusion

- ➔ The compiler can help optimize hardware
- ➔ SMT suffers from extensive LQ searches
- ➔ CELLO can

  1. Avoid LQ searches by 47%
  2. Provide a speedup of 2.8% (up to 18.6%)
  3. Reduce the LQ energy consumption by 33%
- ➔ CELLO provides an interesting design space by allowing to reduction the LQ size from 192 to 80 without any performance loss.

CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions

Sawan Singh, Josue Feliu, Manuel E. Acacio, Alexandra Jimborean, Alberto Ros

singh.sawan@um.es

Thank you for your attention!

Funded by the European Union

European Research Council
Established by the European Commission

MINISTERIO DE CIENCIA E INNOVACIÓN

Financiado por la Unión Europea
NextGenerationEU

Plan de Recuperación, Transformación y Resiliencia

AGENCIA ESTATAL DE INVESTIGACIÓN

ECHO, ERC Consolidator Grant (No 819134)

This presentation belong to the authors. No distribution is allowed without the authors' permission.

32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)

Backup Slides