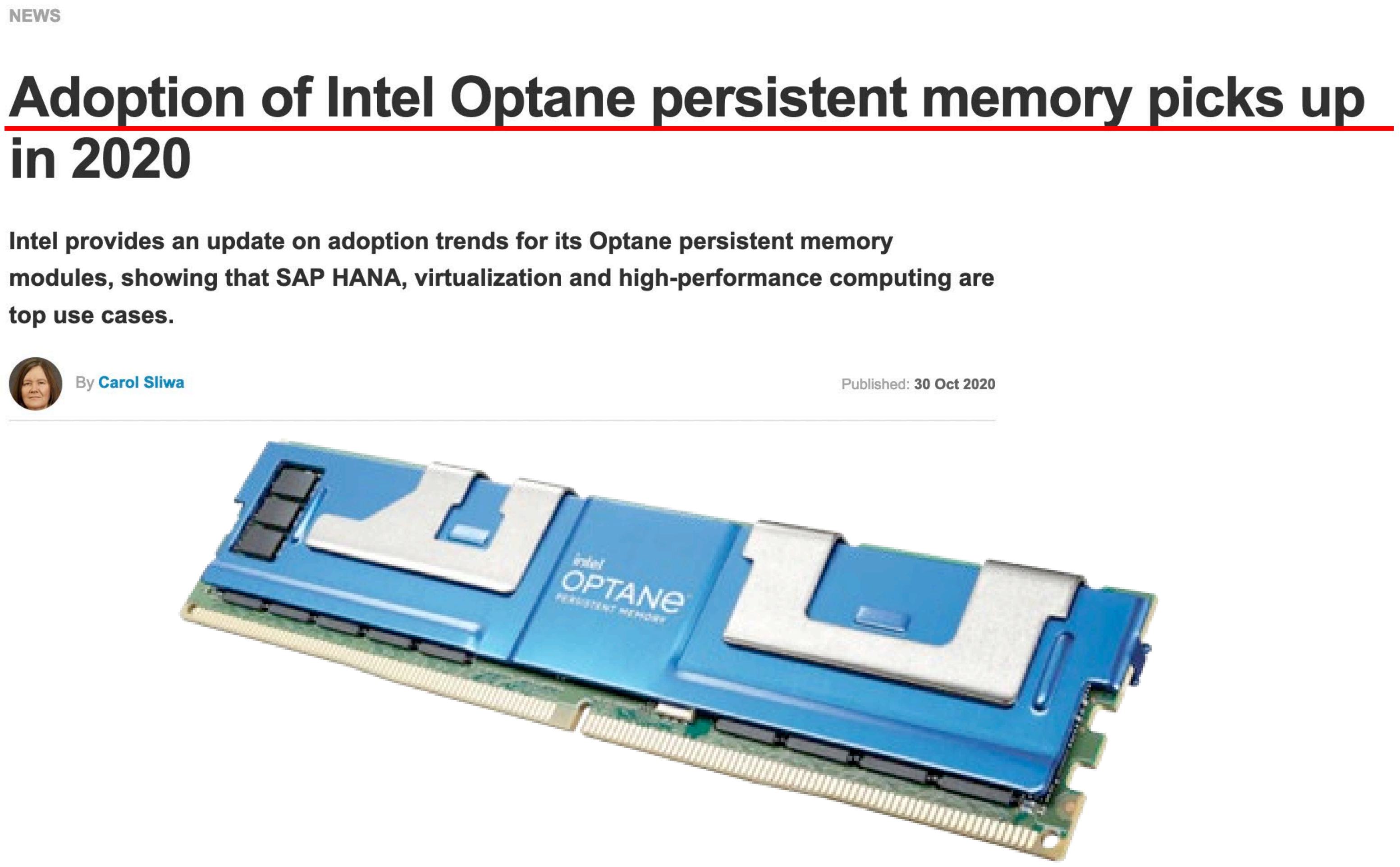
Understanding and Detecting Deep Memory Persistency Bugs in NVM Programs with DeepMC

Benjamin Reidys Jian Huang

UNIVERSITY OF ILLINOIS
URBANA-CHAMPAIGN

Non-Volatile Memory is a Promising Technology



NEWS

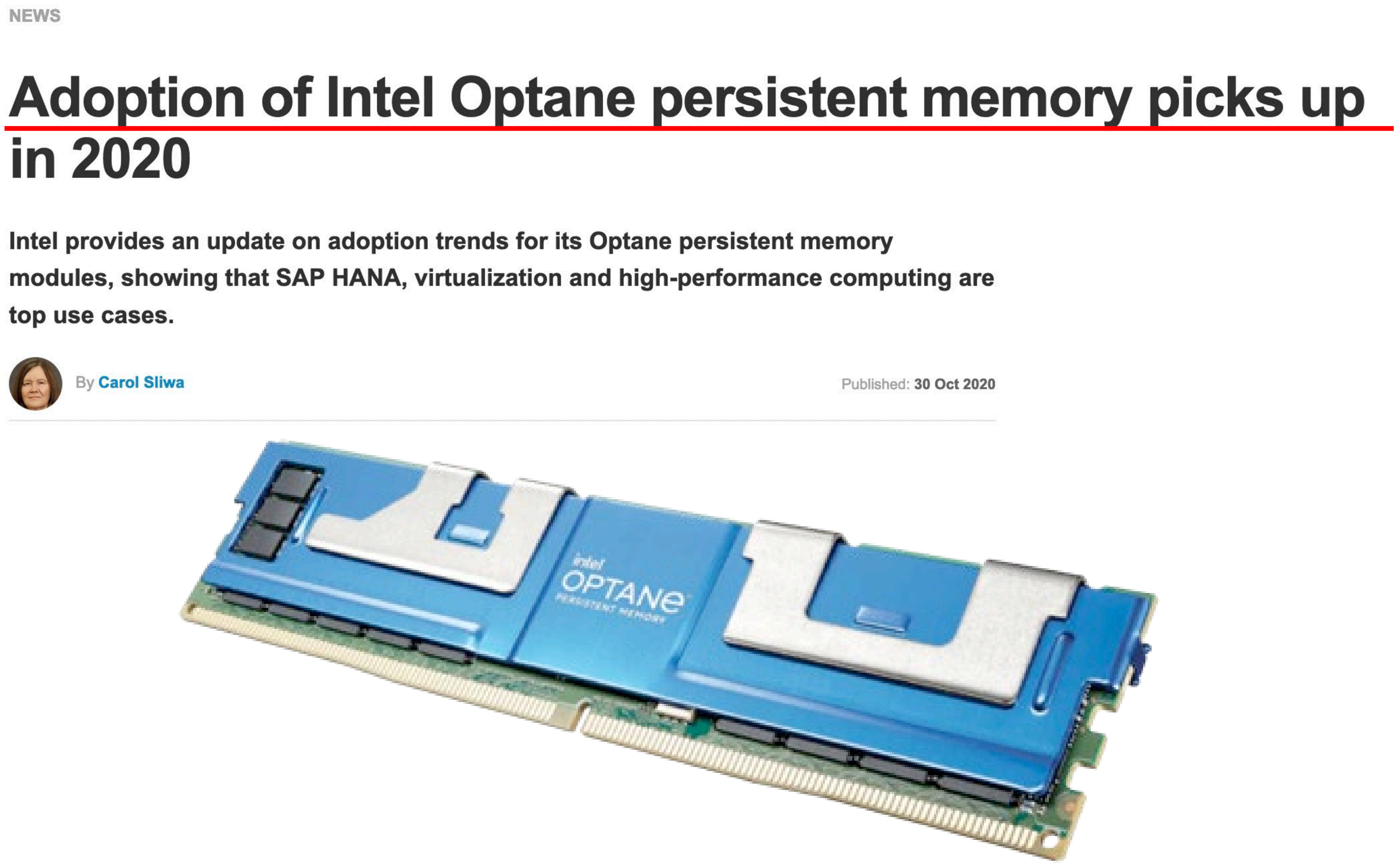
Adoption of Intel Optane persistent memory picks up in 2020

Intel provides an update on adoption trends for its Optane persistent memory modules, showing that SAP HANA, virtualization and high-performance computing are top use cases.

By **Carol Sliwa**

Published: **30 Oct 2020**

Non-Volatile Memory is a Promising Technology



NEWS

Adoption of Intel Optane persistent memory picks up in 2020

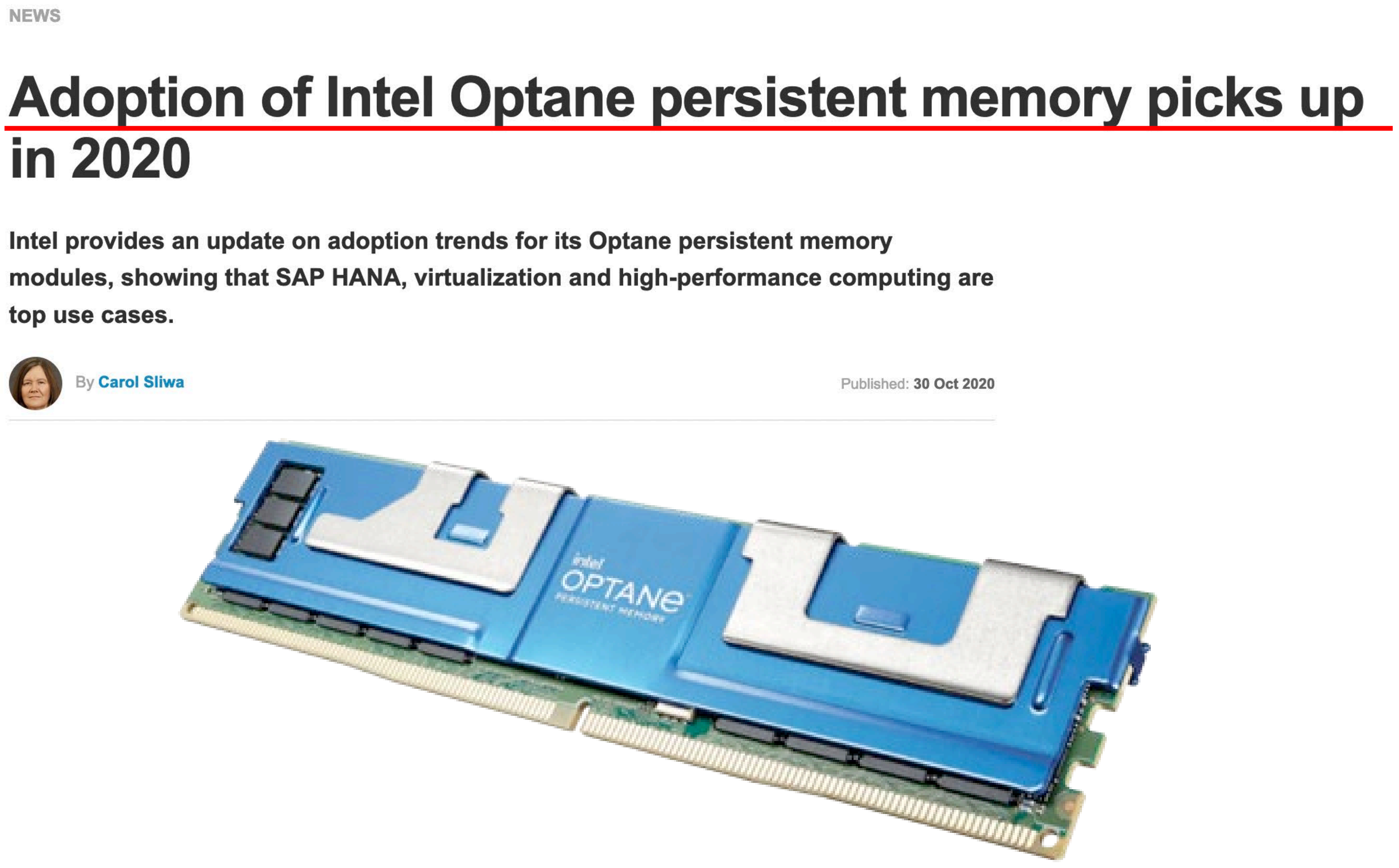
Intel provides an update on adoption trends for its Optane persistent memory modules, showing that SAP HANA, virtualization and high-performance computing are top use cases.

By **Carol Sliwa**

Published: **30 Oct 2020**

Data Durability

Non-Volatile Memory is a Promising Technology



NEWS

Adoption of Intel Optane persistent memory picks up in 2020

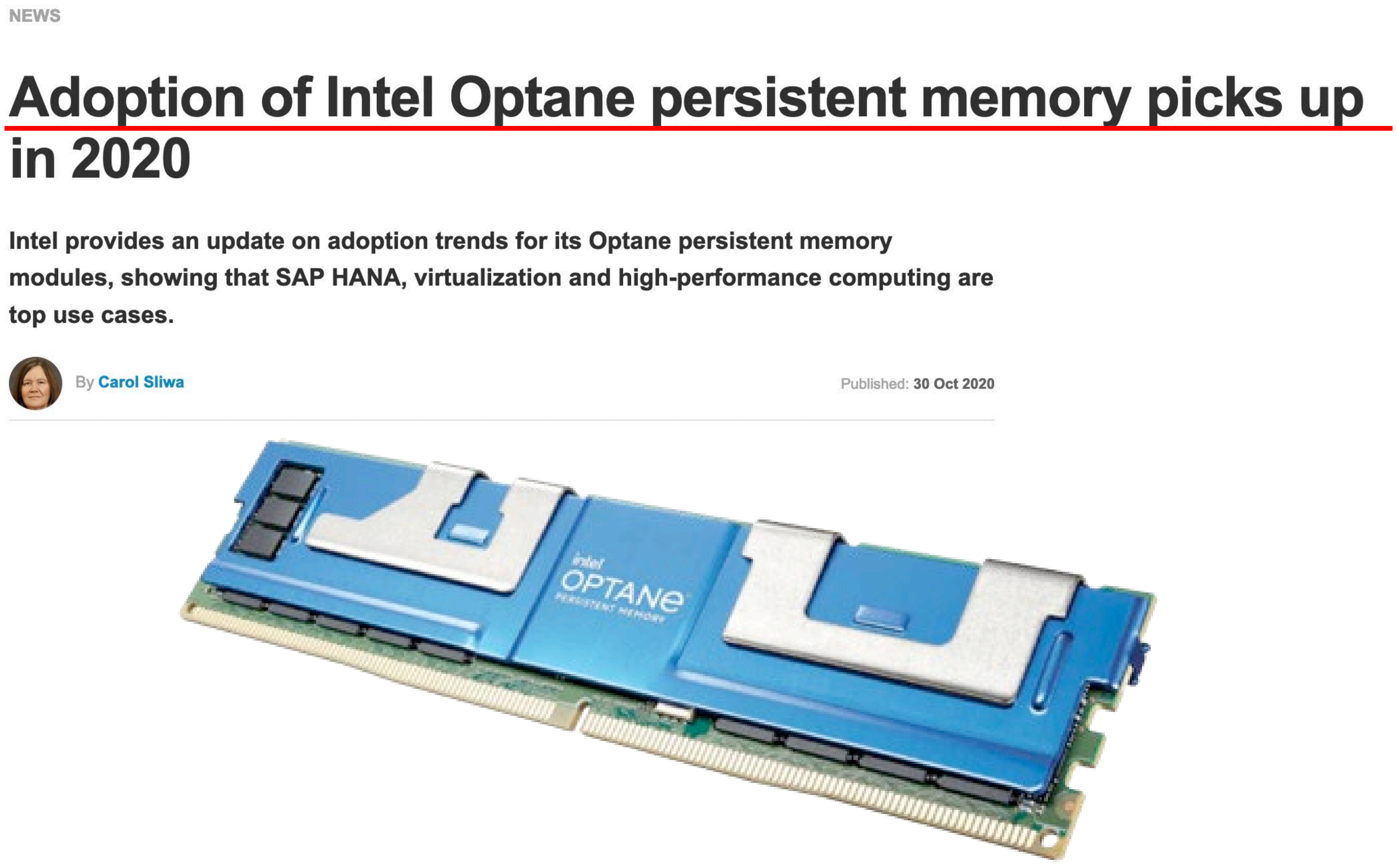
Intel provides an update on adoption trends for its Optane persistent memory modules, showing that SAP HANA, virtualization and high-performance computing are top use cases.

By **Carol Sliwa** Published: **30 Oct 2020**

Data Durability

High Performance

Non-Volatile Memory is a Promising Technology



NEWS

Adoption of Intel Optane persistent memory picks up in 2020

Intel provides an update on adoption trends for its Optane persistent memory modules, showing that SAP HANA, virtualization and high-performance computing are top use cases.

By **Carol Sliwa**

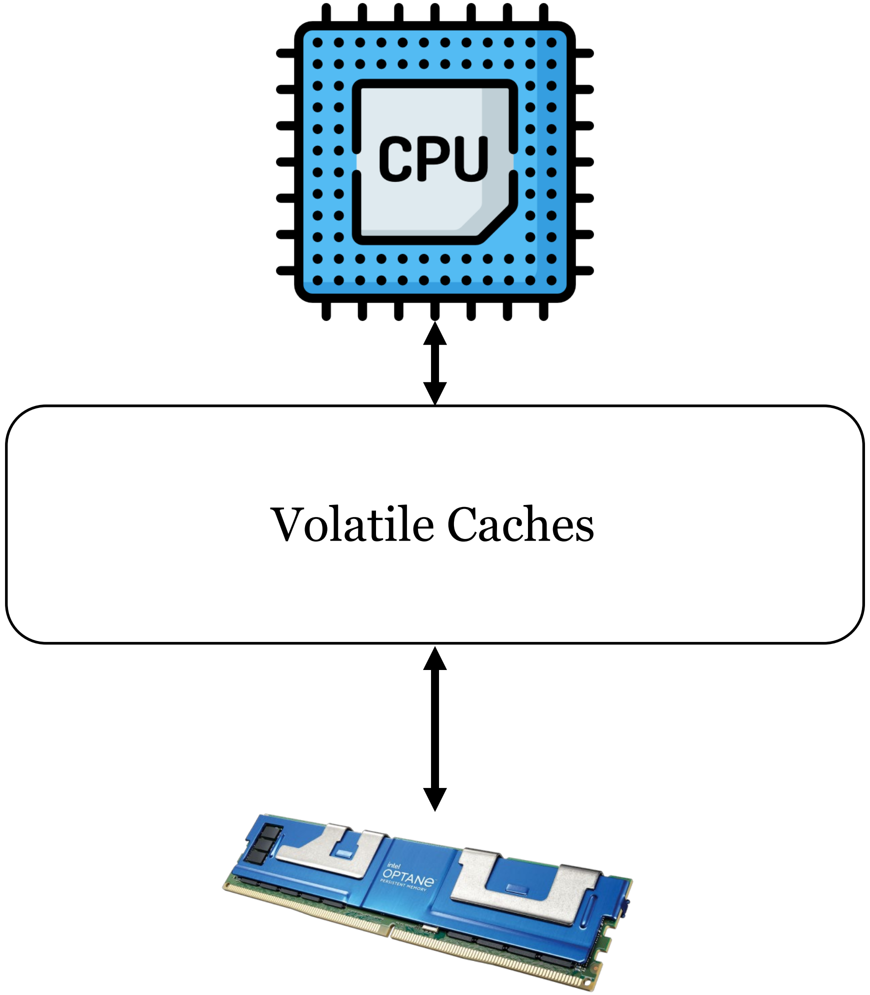
Published: **30 Oct 2020**

Data Durability

High Performance

Byte Addressability

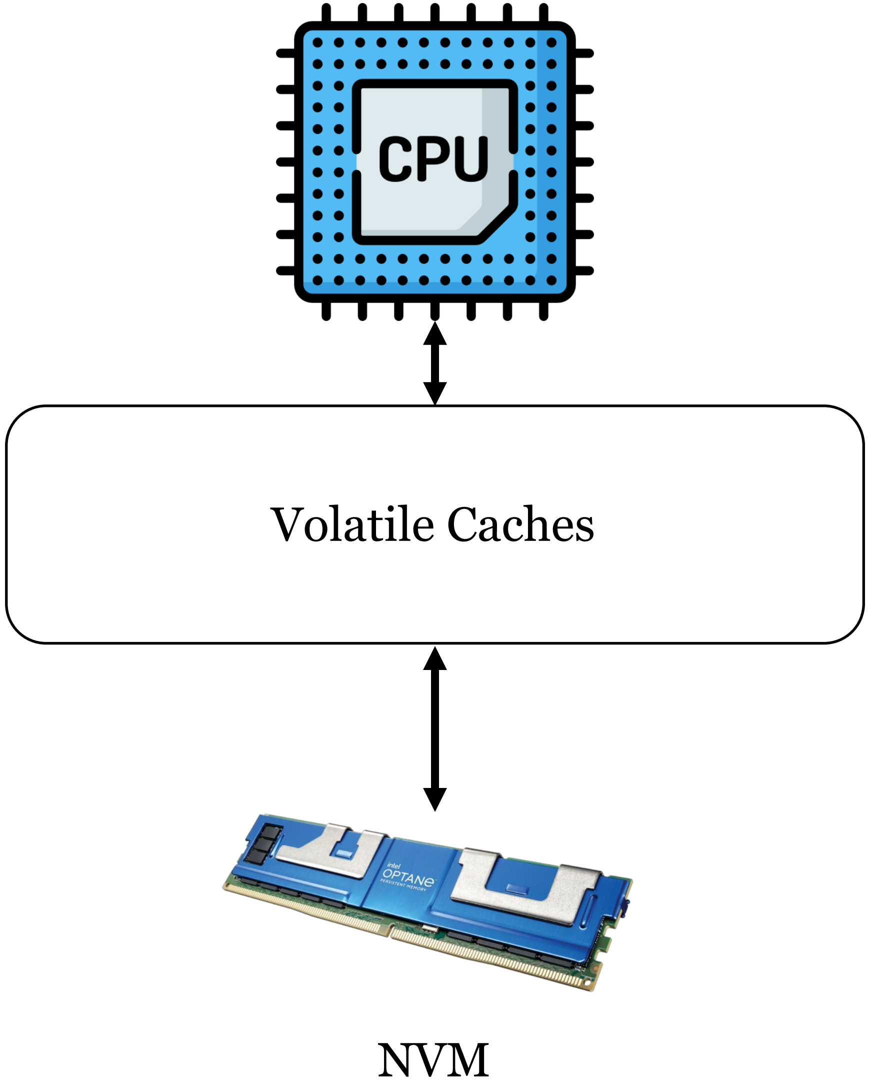
Using Non-Volatile Memory is Not Easy



Volatile Caches

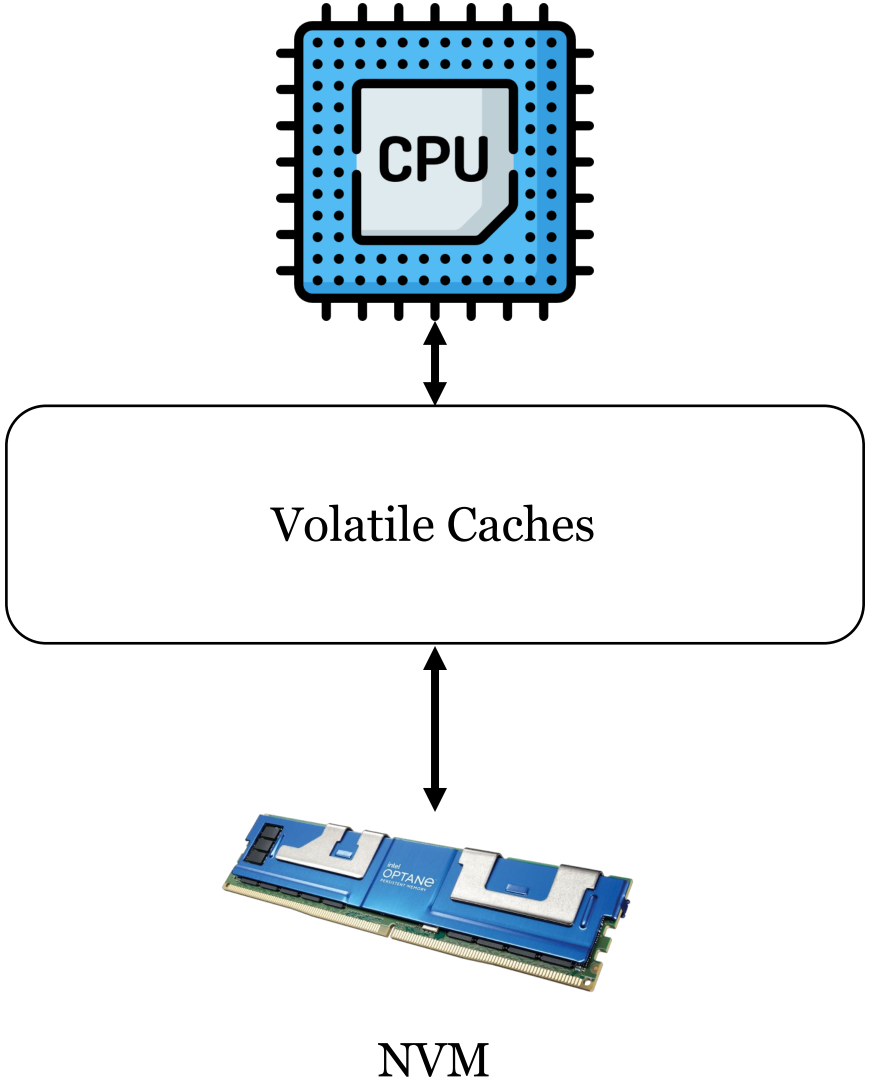
NVM

Using Non-Volatile Memory is Not Easy



Volatile Processor Caches

Using Non-Volatile Memory is Not Easy



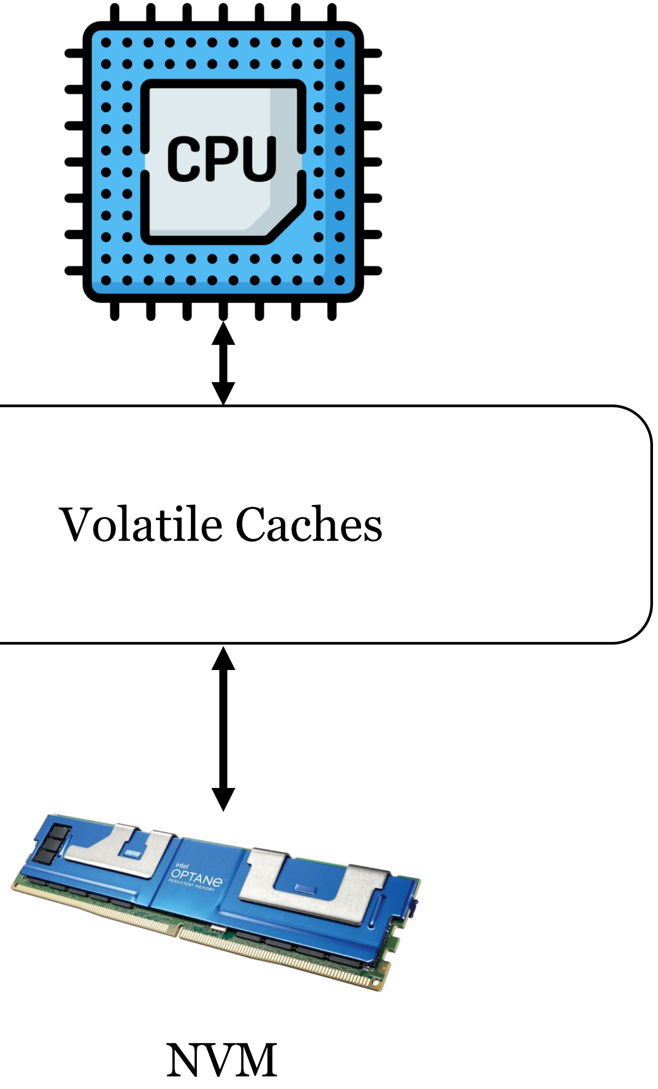
Volatile Caches

NVM

Volatile Processor Caches

Out-of-Order Execution

Using Non-Volatile Memory is Not Easy

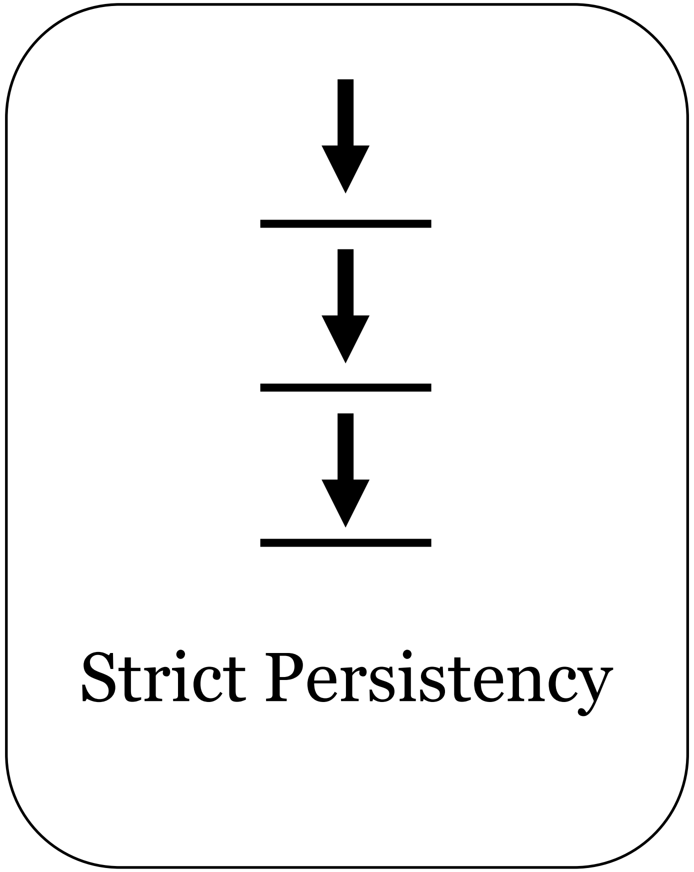


Volatile Processor Caches

Out-of-Order Execution

Persistence/Performance Tradeoff

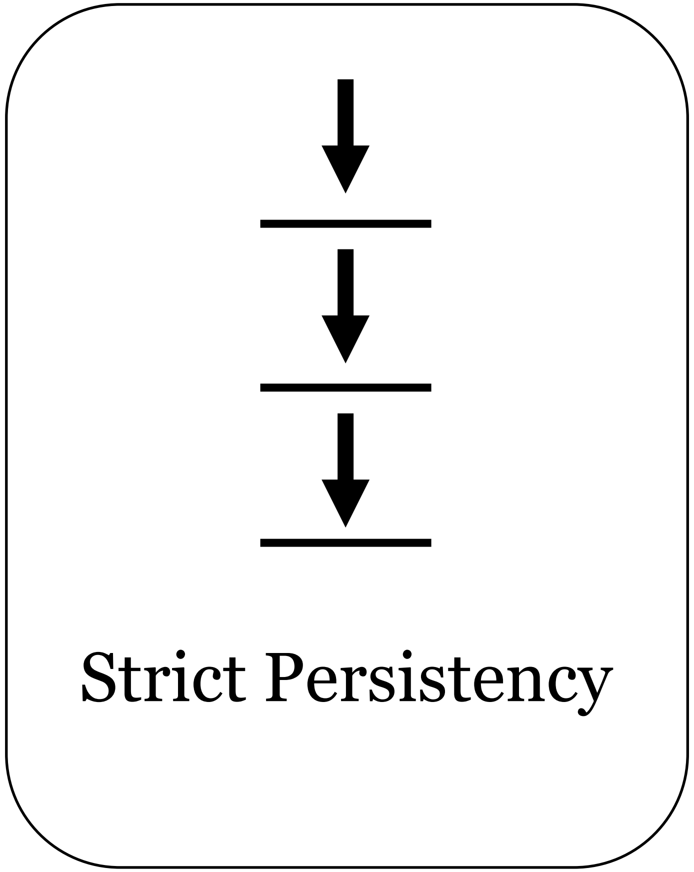
Persistency Models for Non-Volatile Memory



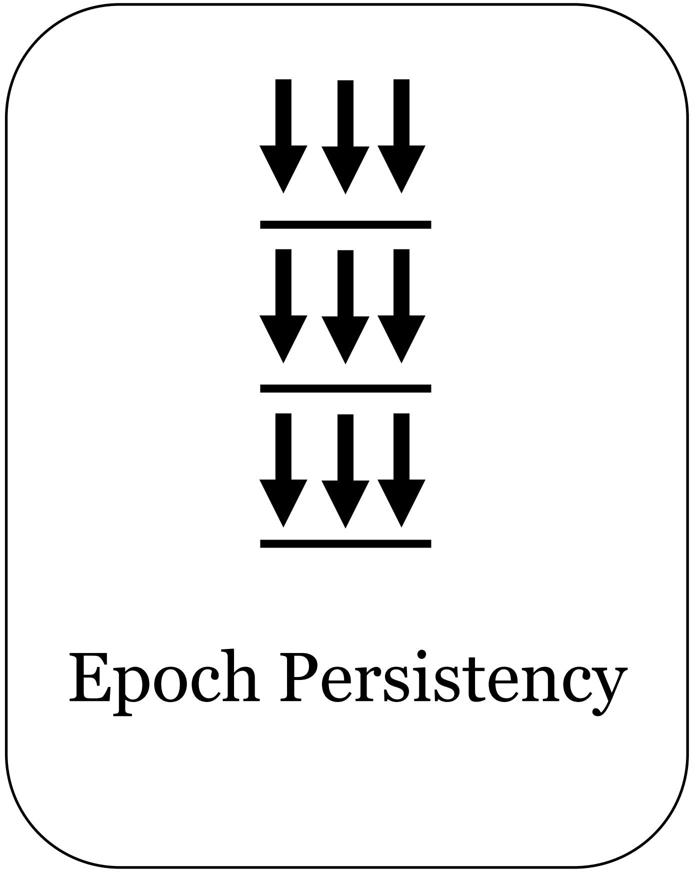
Strict Persistency

Writes must be persisted in program order!

Persistency Models for Non-Volatile Memory



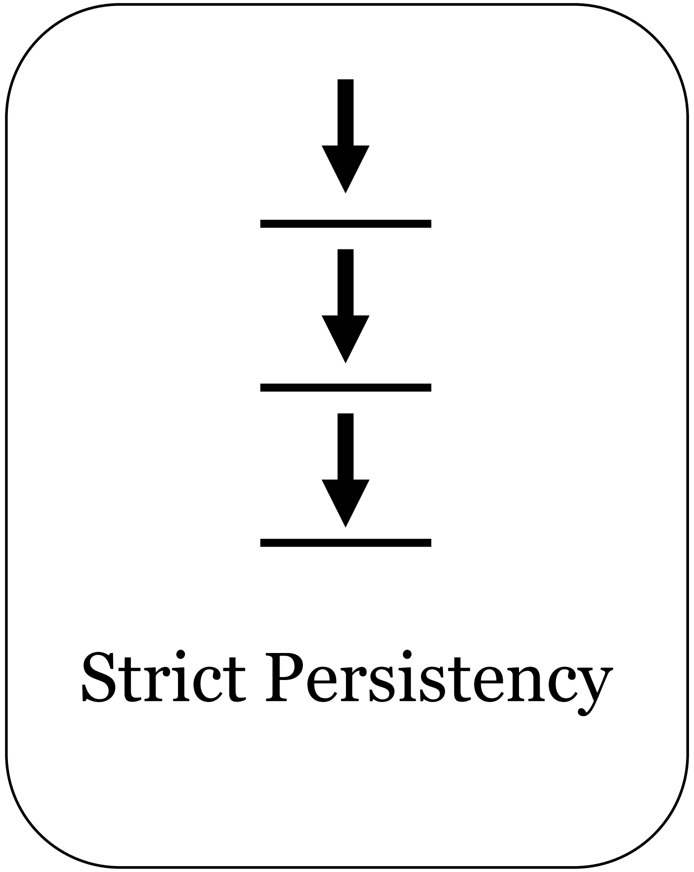
Strict Persistency



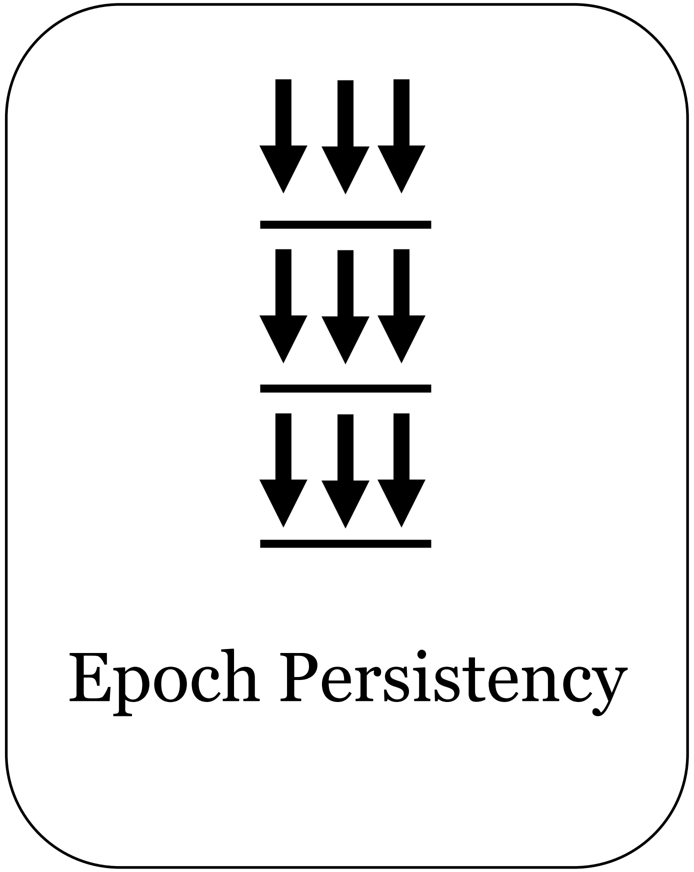
Epoch Persistency

Writes can be concurrent within an epoch!

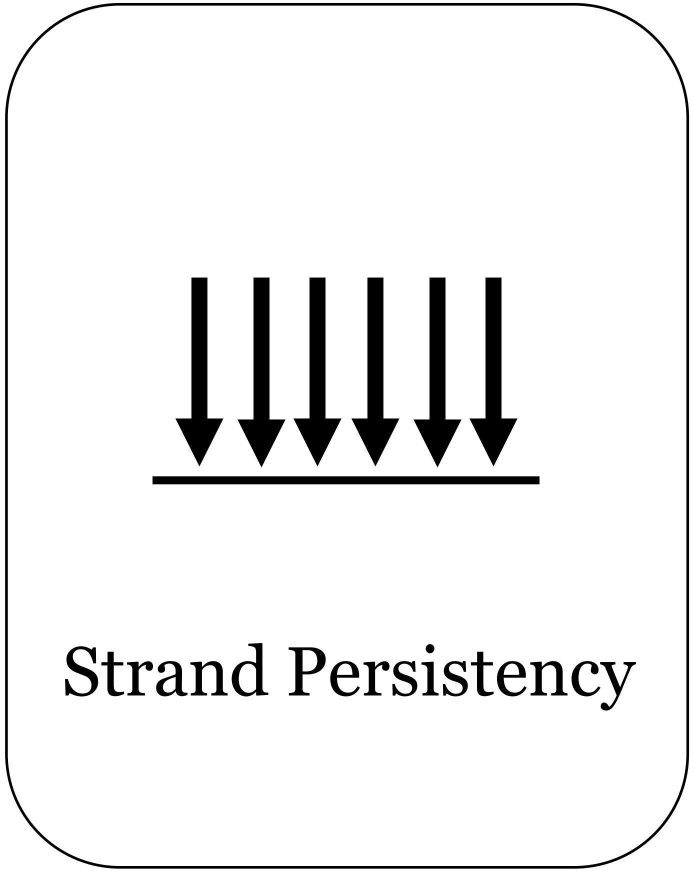
Persistency Models for Non-Volatile Memory



Strict Persistency



Epoch Persistency

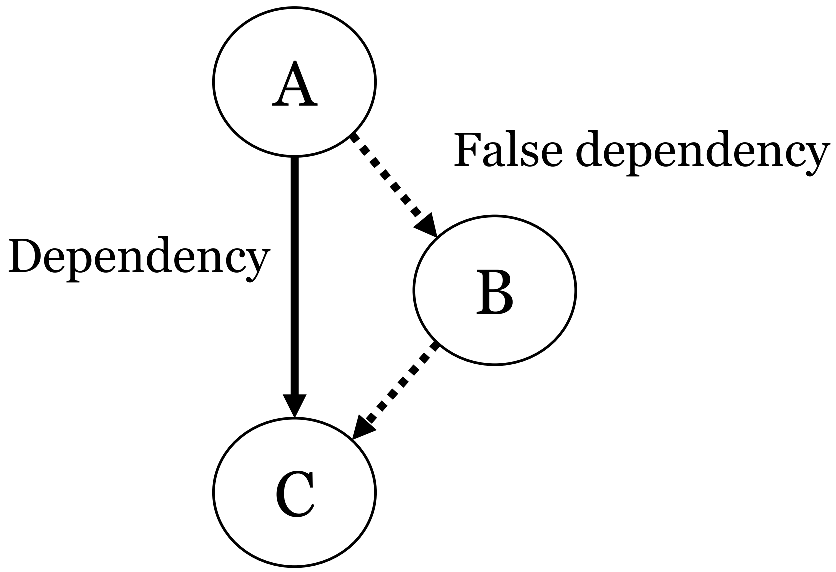


Strand Persistency

Writes can be concurrent within and across strands!

Persistency Models – Strict Persistency

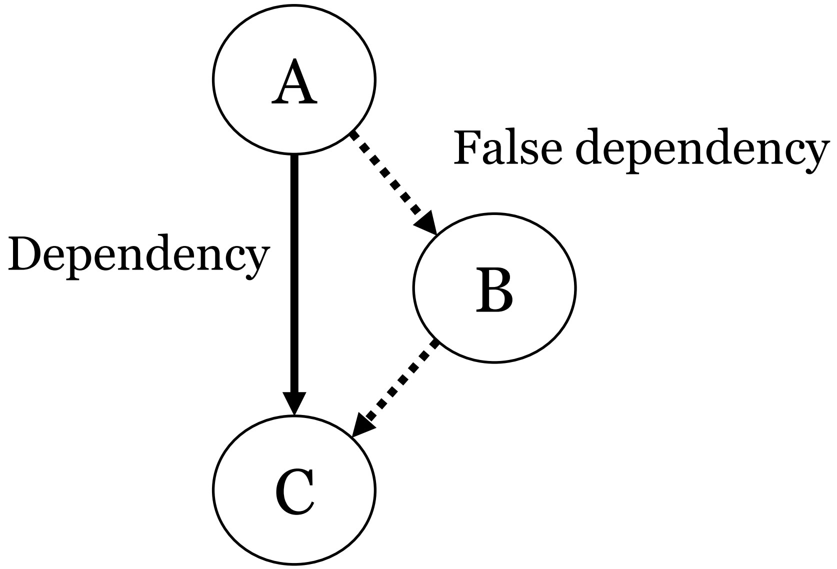
```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```



Writes must be persisted in program order!

Persistency Models – Strict Persistency

```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```

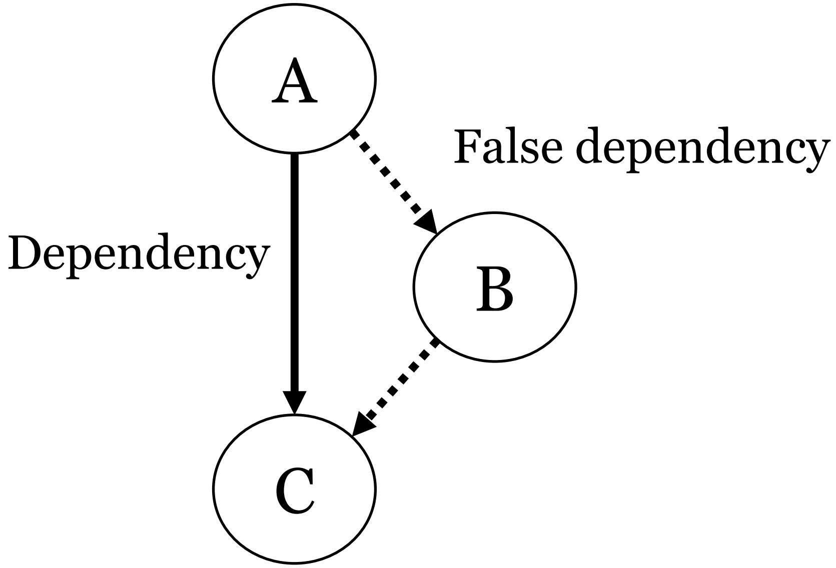


Writes must be persisted in program order!

Persistency Models – Strict Persistency

```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```

Insert barriers between operations!



Writes must be persisted in program order!

Persistency Models – Strict Persistency

```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```

Insert barriers between operations!

💪

Strong Persistency Guarantees

Writes must be persisted in program order!

Persistency Models – Strict Persistency

```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```

Insert barriers between operations!

Strong Persistency Guarantees

Easy to Use

Writes must be persisted in program order!

Persistency Models – Strict Persistency

```
A
clwb(A)
mfence
B
clwb(B)
mfence
C
```

Insert barriers between operations!

Strong Persistency Guarantees

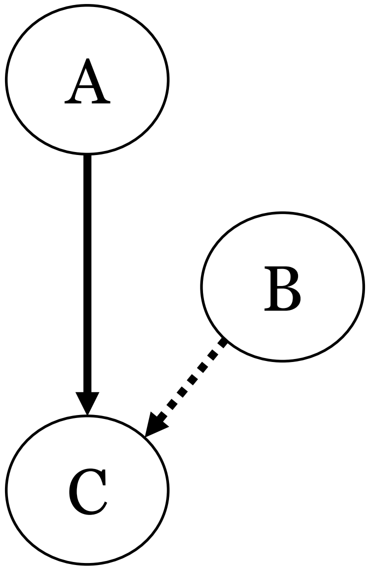
Easy to Use

Low Performance!

Writes must be persisted in program order!

Persistency Models – Epoch Persistency

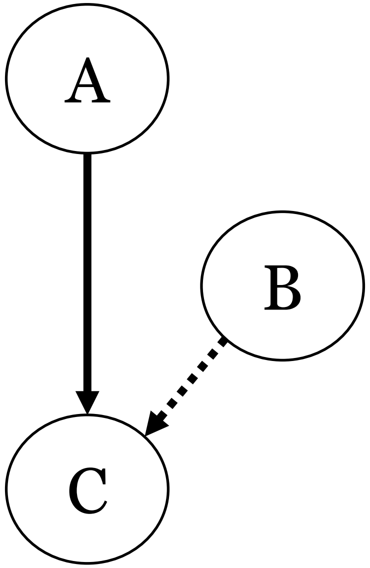
```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```



Writes can be concurrent within an epoch!

Persistency Models – Epoch Persistency

```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

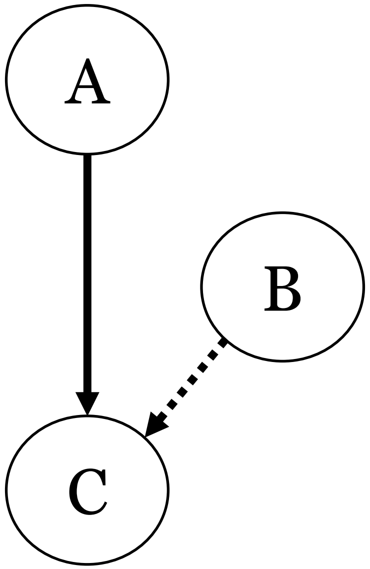


Writes can be concurrent within an epoch!

Persistency Models – Epoch Persistency

```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

Ordering enforced between epochs!



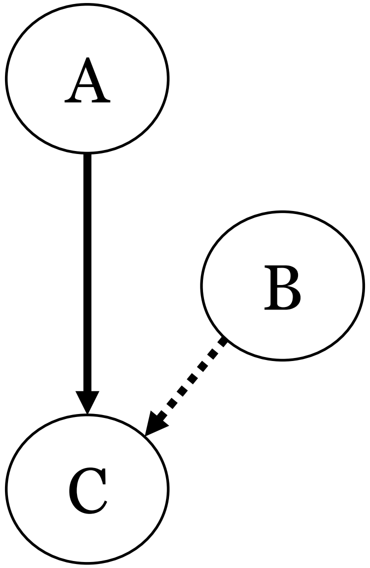
Writes can be concurrent within an epoch!

Persistency Models – Epoch Persistency

```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

Concurrency within an epoch!

Ordering enforced between epochs!



Writes can be concurrent within an epoch!

Persistency Models – Epoch Persistency

```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

Concurrency within an epoch!

Ordering enforced between epochs!

Relaxed Persistency Guarantees

Writes can be concurrent within an epoch!

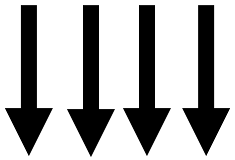
Persistency Models – Epoch Persistency

```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

Concurrency within an epoch!

Ordering enforced between epochs!

Relaxed Persistency Guarantees



Enables Higher Concurrency

Writes can be concurrent within an epoch!

Persistency Models – Epoch Persistency

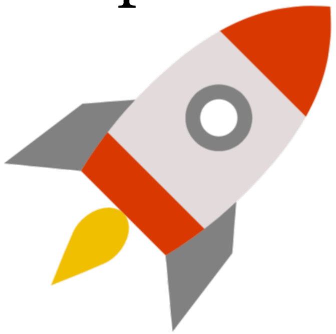
```
begin_epoch
A
B
end_epoch
begin_epoch
C
end_epoch
```

Concurrency within an epoch!

Ordering enforced between epochs!

Relaxed Persistency Guarantees

Enables Higher Concurrency

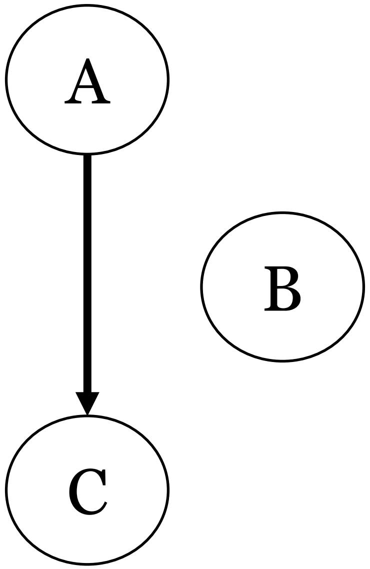


Improved Performance!

Writes can be concurrent within an epoch!

Persistency Models – Strand Persistency

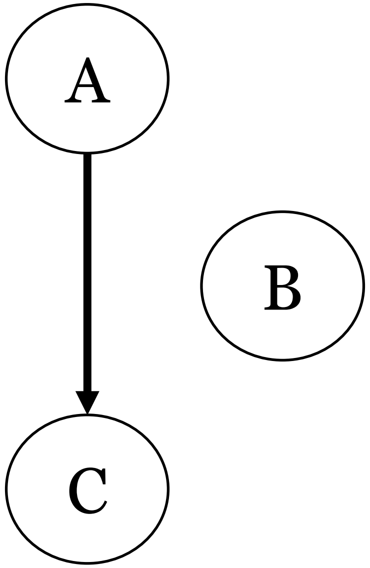
```
begin_strand
B
begin_strand
A
barrier
C
```



Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

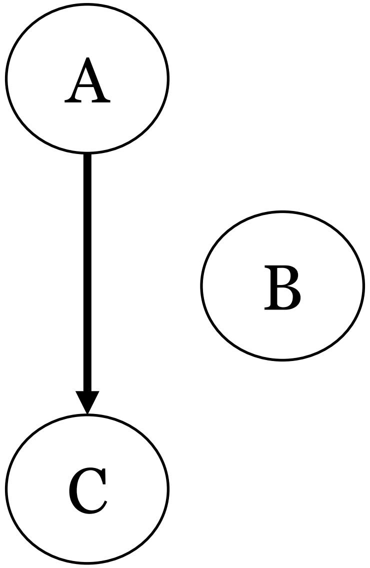
```
begin_strand
B
begin_strand
A
barrier
C
```



Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

```
begin_strand
B
begin_strand
A
barrier
C
```

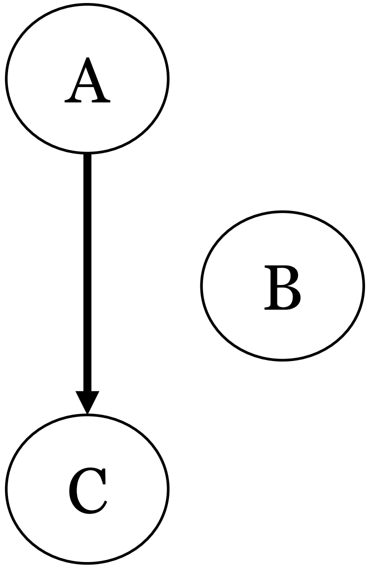


Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

```
begin_strand
B
begin_strand
A
barrier
C
```

Order within strand!



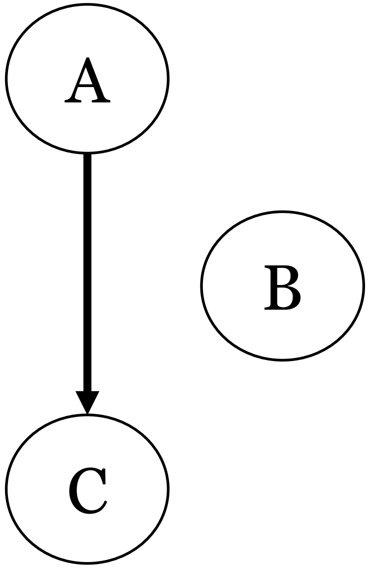
Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

```
begin_strand
B
begin_strand
A
barrier
C
```

No false dependencies!

Order within strand!



Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

```
begin_strand
B
begin_strand
A
barrier
C
```

No false dependencies!

Order within strand!

Highest Possible Performance

Writes can be reordered within and across strands!

Persistency Models – Strand Persistency

```
begin_strand
B
begin_strand
A
barrier
C
```

No false dependencies!

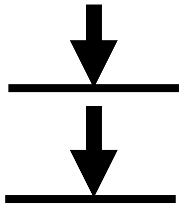
Order within strand!

Highest Possible Performance

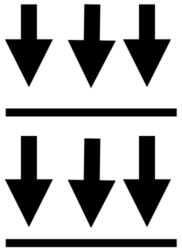
Difficult to program correctly

Writes can be reordered within and across strands!

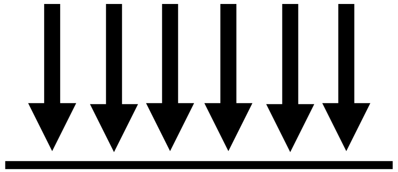
Implementing Persistency Models Properly is Challenging



Strict Persistency

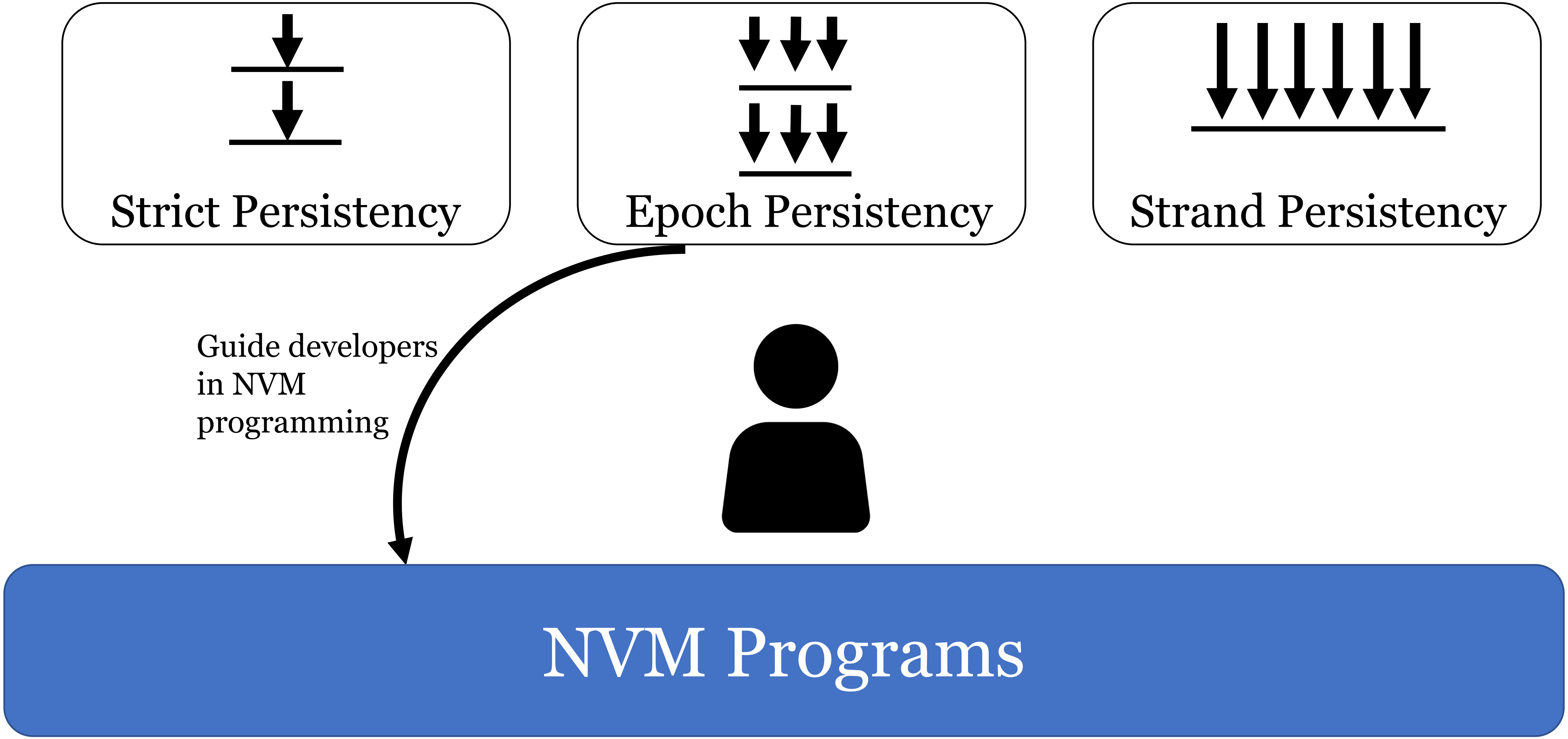


Epoch Persistency



Strand Persistency

Implementing Persistency Models Properly is Challenging



Strict Persistency

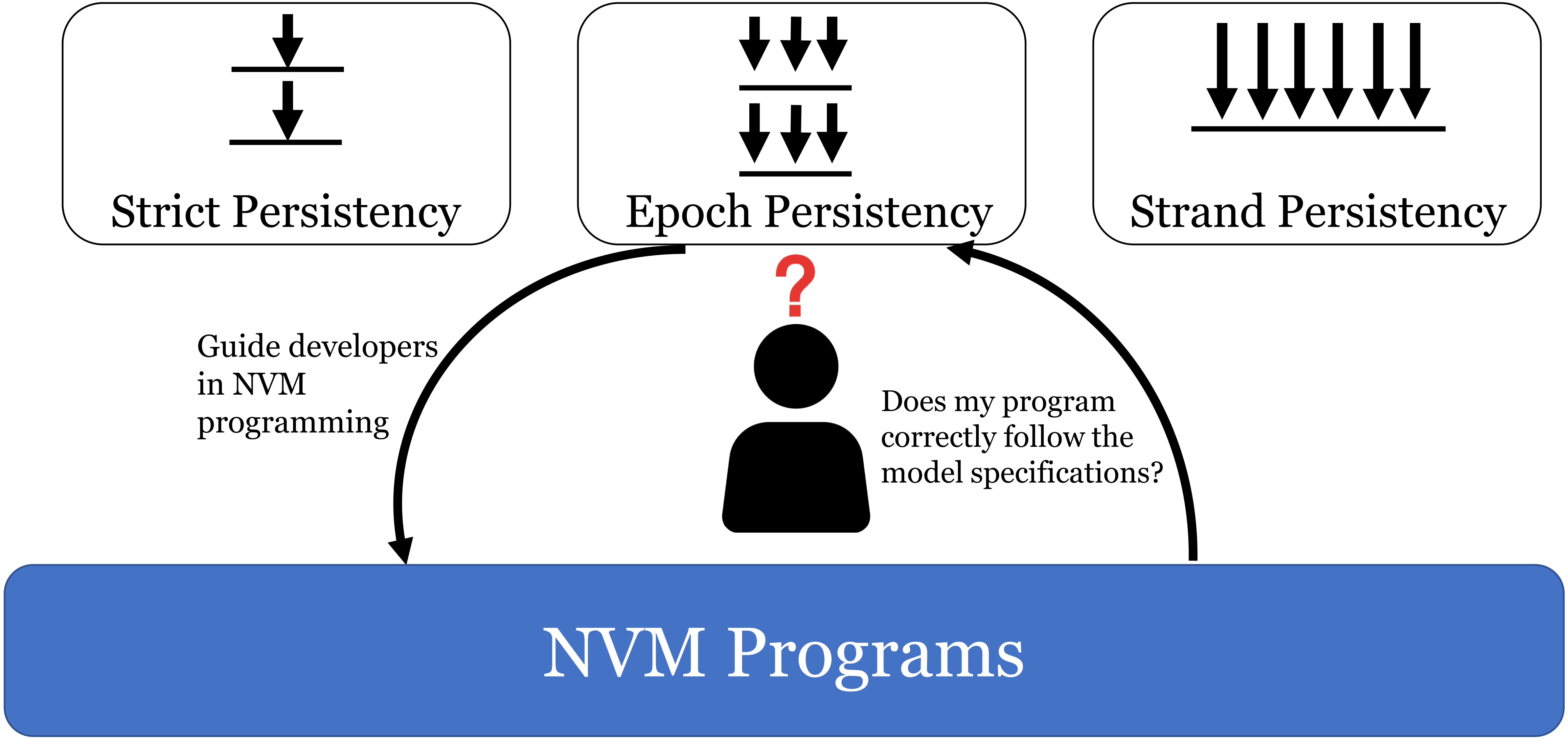
Epoch Persistency

Strand Persistency

Guide developers
in NVM
programming

NVM Programs

Implementing Persistency Models Properly is Challenging



Strict Persistency

Epoch Persistency

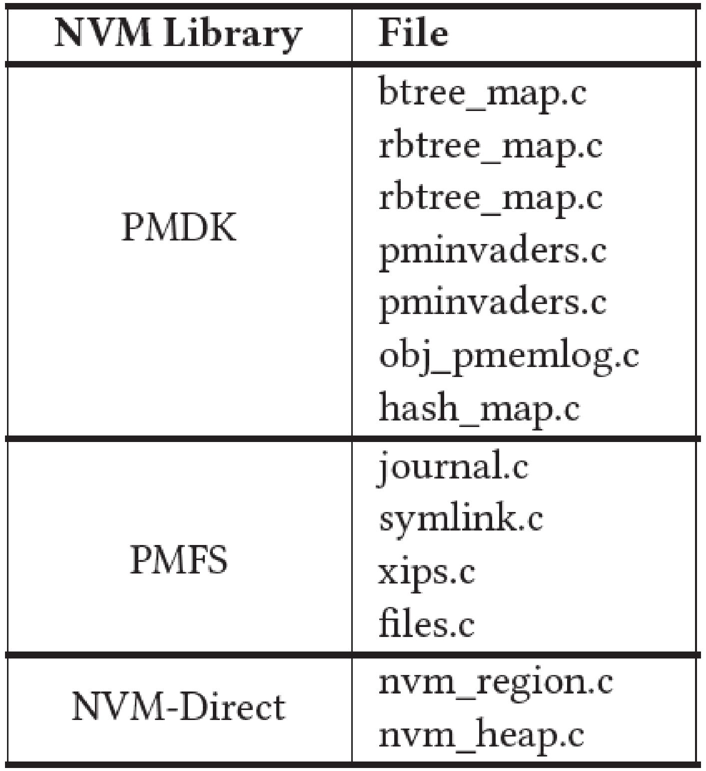
Strand Persistency

Guide developers in NVM programming

Does my program correctly follow the model specifications?

NVM Programs

Understanding Persistency Bugs in NVM Programs



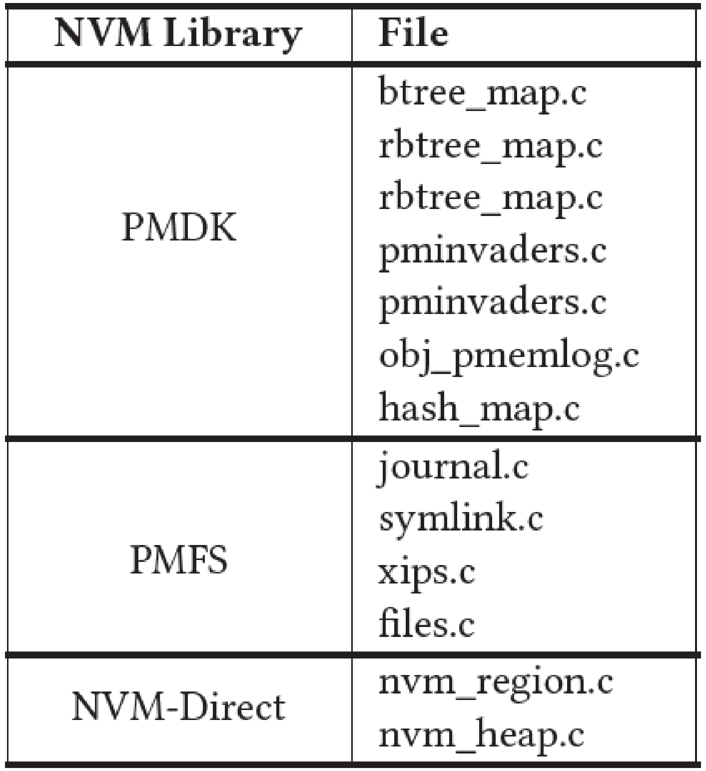
NVM Library	File
PMDK	btree_map.c rbtree_map.c rbtree_map.c pminvaders.c pminvaders.c obj_pmemlog.c hash_map.c
PMFS	journal.c symlink.c xips.c files.c
NVM-Direct	nvm_region.c nvm_heap.c

Understanding Persistency Bugs in NVM Programs

NVM Library	File
PMDK	btree_map.c
	rbtree_map.c
	rbtree_map.c
	pminvaders.c
	pminvaders.c
	obj_pmemlog.c
	hash_map.c
PMFS	journal.c
	symlink.c
	xips.c
	files.c
NVM-Direct	nvm_region.c
	nvm_heap.c

Select programs from open source framework PMDK, PMFS, and NVM-Direct

Understanding Persistency Bugs in NVM Programs

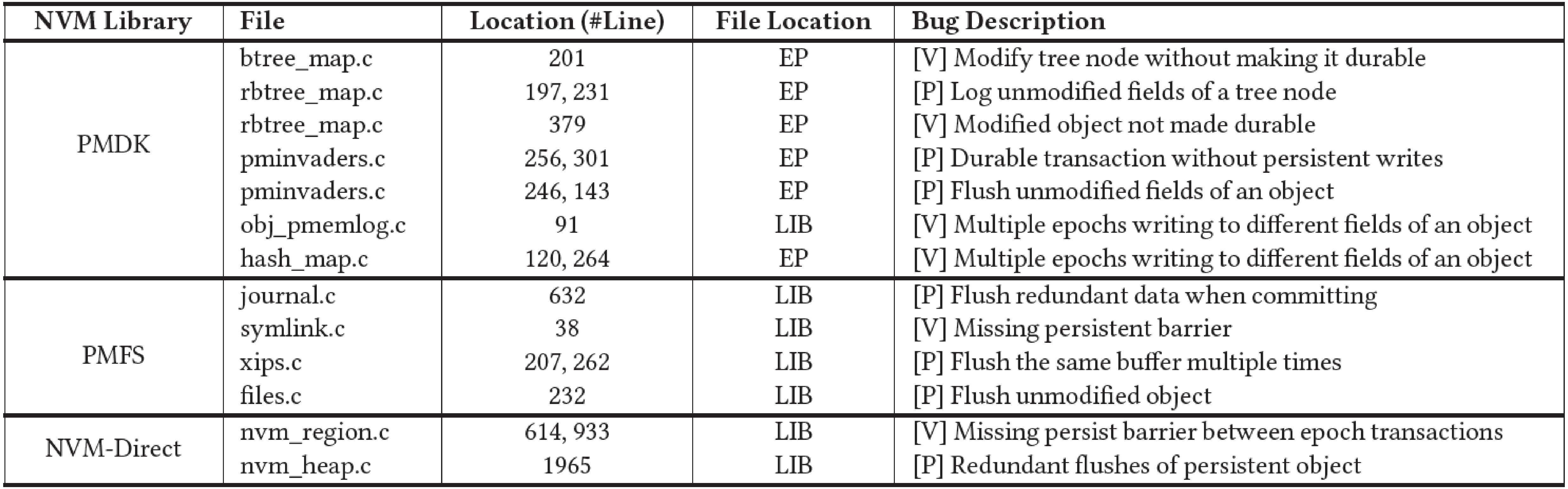


NVM Library	File
PMDK	btree_map.c
	rbtree_map.c
	rbtree_map.c
	pminvaders.c
	pminvaders.c
	obj_pmemlog.c
	hash_map.c
PMFS	journal.c
	symlink.c
	xips.c
	files.c
NVM-Direct	nvm_region.c
	nvm_heap.c

Select programs from open source framework PMDK, PMFS, and NVM-Direct

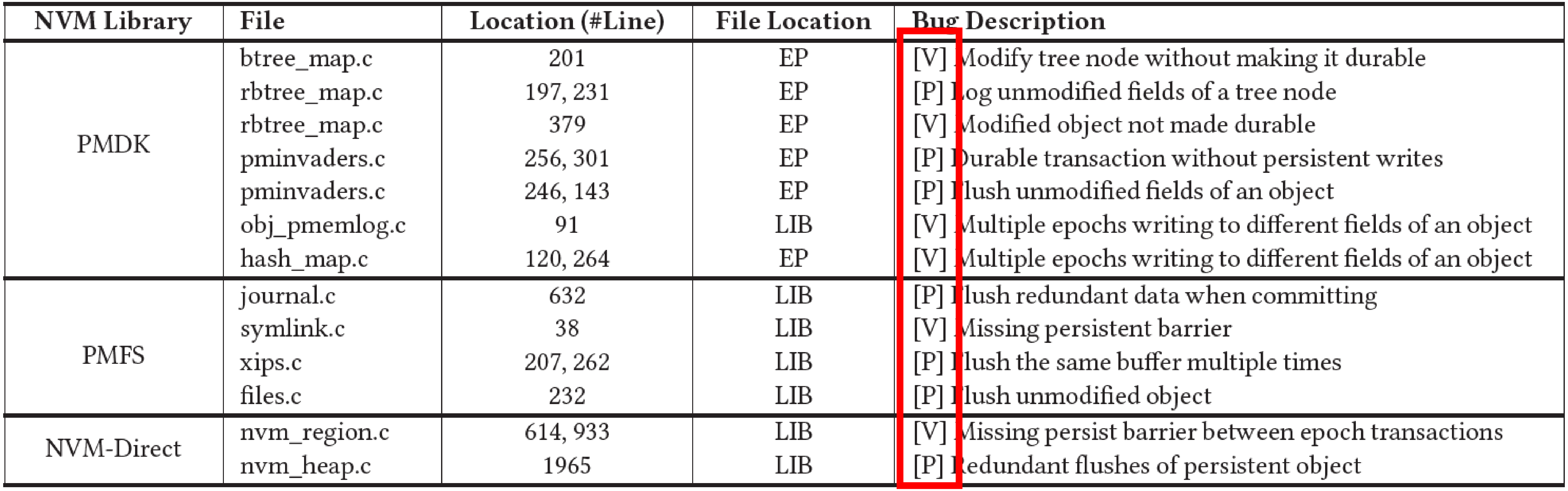
Manually study 19 representative persistency bugs

Understanding Persistency Bugs in NVM Programs



NVM Library	File	Location (#Line)	File Location	Bug Description
PMDK	btree_map.c	201	EP	[V] Modify tree node without making it durable
	rbtree_map.c	197, 231	EP	[P] Log unmodified fields of a tree node
	rbtree_map.c	379	EP	[V] Modified object not made durable
	pminvaders.c	256, 301	EP	[P] Durable transaction without persistent writes
	pminvaders.c	246, 143	EP	[P] Flush unmodified fields of an object
	obj_pmemlog.c	91	LIB	[V] Multiple epochs writing to different fields of an object
	hash_map.c	120, 264	EP	[V] Multiple epochs writing to different fields of an object
PMFS	journal.c	632	LIB	[P] Flush redundant data when committing
	symlink.c	38	LIB	[V] Missing persistent barrier
	xips.c	207, 262	LIB	[P] Flush the same buffer multiple times
	files.c	232	LIB	[P] Flush unmodified object
NVM-Direct	nvm_region.c	614, 933	LIB	[V] Missing persist barrier between epoch transactions
	nvm_heap.c	1965	LIB	[P] Redundant flushes of persistent object

Understanding Persistency Bugs in NVM Programs



NVM Library	File	Location (#Line)	File Location	Bug Description
PMDK	btree_map.c	201	EP	[V] Modify tree node without making it durable
	rbtree_map.c	197, 231	EP	[P] Log unmodified fields of a tree node
	rbtree_map.c	379	EP	[V] Modified object not made durable
	pminvaders.c	256, 301	EP	[P] Durable transaction without persistent writes
	pminvaders.c	246, 143	EP	[P] Flush unmodified fields of an object
	obj_pmemlog.c	91	LIB	[V] Multiple epochs writing to different fields of an object
	hash_map.c	120, 264	EP	[V] Multiple epochs writing to different fields of an object
PMFS	journal.c	632	LIB	[P] Flush redundant data when committing
	symlink.c	38	LIB	[V] Missing persistent barrier
	xips.c	207, 262	LIB	[P] Flush the same buffer multiple times
	files.c	232	LIB	[P] Flush unmodified object
NVM-Direct	nvm_region.c	614, 933	LIB	[V] Missing persist barrier between epoch transactions
	nvm_heap.c	1965	LIB	[P] Redundant flushes of persistent object

We analyze each bug and discover they fall into two categories:
Model Violations [V] or Performance Bugs [P].

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Persistency Model Violations: Semantic Mismatch

```
static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
    struct buckets *b = (struct buckets *) ptr;
    b->nbuckets = * ((size_t *) arg);
    pmemobj_memset_persist (pop, &b->bucket, 0,
                                   b->nbuckets * sizeof (b->bucket[0]));
    pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
    return 0;
}
```

hashmap from PMDK
using strict persistency

Persistency Model Violations: Semantic Mismatch

```
1  static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
2      struct buckets *b = (struct buckets *) ptr;
3      b->nbuckets = * ((size_t *) arg);
4      pmemobj_memset_persist (pop, &b->bucket, 0,
5                                   b->nbuckets * sizeof (b->bucket[0]));
6      pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
7      return 0;
8  }
```

nbuckets initialized on line 3

hashmap from PMDK
using strict persistency

Persistency Model Violations: Semantic Mismatch

```
1  static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
2      struct buckets *b = (struct buckets *) ptr;
3      b->nbuckets = * ((size_t *) arg);
4      pmemobj_memset_persist (pop, &b->bucket, 0,
5                              b->nbuckets * sizeof (b->bucket[0]));
6      pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
7      return 0;
8  }
```

nbuckets initialized on line 3

nbuckets is not persisted until line 6

hashmap from PMDK
using strict persistency

Persistency Model Violations: Semantic Mismatch

```
1  static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
2      struct buckets *b = (struct buckets *) ptr;
3      b->nbuckets = * ((size_t *) arg);
4      pmemobj_memset_persist (pop, &b->bucket, 0,
5                                            b->nbuckets * sizeof (b->bucket[0]));
6      pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
7      return 0;
8  }
```

nbuckets initialized on line 3

nbuckets is not persisted until line 6

hashmap from PMDK
using strict persistency

Strict persistency requires persists to occur in program order!

Persistency Model Violations: Semantic Mismatch

```
1  static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
2      struct buckets *b = (struct buckets *) ptr;
3      b->nbuckets = * ((size_t *) arg);
4      pmemobj_memset_persist (pop, &b->bucket, 0,
5                                 b->nbuckets * sizeof (b->bucket[0]));
6      pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
7      return 0;
8  }
```

nbuckets initialized on line 3

nbuckets is not persisted until line 6

hashmap from PMDK
using strict persistency

Crash between lines 4 and 6 results in inconsistency!

Persistency Model Violations: Unflushed/Unlogged Writes

```
static struct tree_map_node *
btree_map_create_split_node (struct tree_map_node *node,
                             struct tree_map_node _item *m) {
    .........

    node->items[c - 1] = EMPTY_ITEM;
    .........
    return 0;
} // This function is executed in a transaction.
```

btree_map from PMDK
using epoch persistency

Persistency Model Violations: Unflushed/Unlogged Writes

```
static struct tree_map_node *
btree_map_create_split_node (struct tree_map_node *node,
                             struct tree_map_node _item *m) {
    .........

    node->items[c - 1] = EMPTY_ITEM;
    .........
    return 0;
} // This function is executed in a transaction.
```

items is not logged in the transaction

btree_map from PMDK
using epoch persistency

Persistency Model Violations: Unflushed/Unlogged Writes

```
static struct tree_map_node *
btree_map_create_split_node (struct tree_map_node *node,
                             struct tree_map_node _item *m) {
    .........

    node->items[c - 1] = EMPTY_ITEM;
    .........
    return 0;
} // This function is executed in a transaction.
```

items is not logged in the transaction

btree_map from PMDK
using epoch persistency

Object is updated without logging and is not persisted!

Persistency Model Violations: Missing Persist Barrier

```
nvm_desc nvm_create_region (nvm_desc desc, const char* pathname,
const char *regionname, void *attach, size_t vspace, size_t pspace, mode_t mode) {
    .........
    nvm_flush (region, sizeof (*region));
    ...
    nvm_app_data *ad = nvm_get_app_data ();
    nvm_txbegin (desc);
    .........
    nvm_txend ();
    return desc;
}
```

nvm_create_region from NVM-Direct using strict persistency

Persistency Model Violations: Missing Persist Barrier

```
nvm_desc nvm_create_region (nvm_desc desc, const char* pathname,
const char *regionname, void *attach, size_t vspace, size_t pspace, mode_t mode) {
    .........
    nvm_flush (region, sizeof (*region));
    ...
    nvm_app_data *ad = nvm_get_app_data ();
    nvm_txbegin (desc);
    .........
    nvm_txend ();
    return desc;
}
```

No persist barrier to enforce ordering

nvm_create_region from NVM-Direct using strict persistency

Persistency Model Violations: Missing Persist Barrier

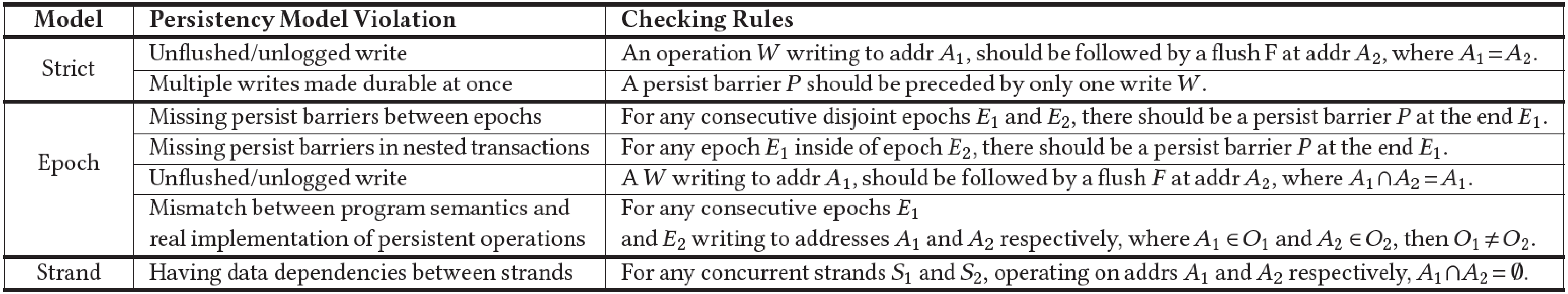
```
nvm_desc nvm_create_region (nvm_desc desc, const char* pathname,
const char *regionname, void *attach, size_t vspace, size_t pspace, mode_t mode) {
    .........
    nvm_flush (region, sizeof (*region));
    ...
    nvm_app_data *ad = nvm_get_app_data ();
    nvm_txbegin (desc);
    .........
    nvm_txend ();
    return desc;
}
```

No persist barrier to enforce ordering

nvm_create_region from NVM-Direct using strict persistency

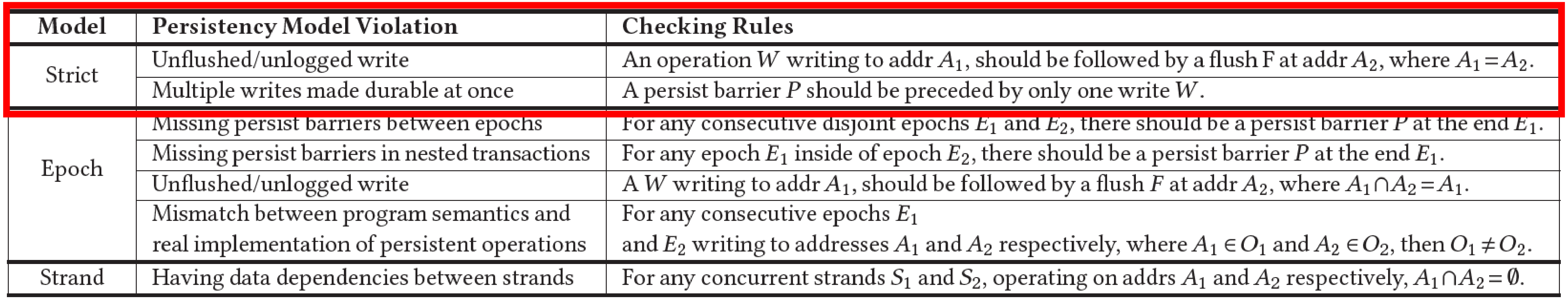
Object is flushed but ordering is not enforced with persist barrier

Persistency Model Violations: Checking Rules



Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

Persistency Model Violations: Checking Rules

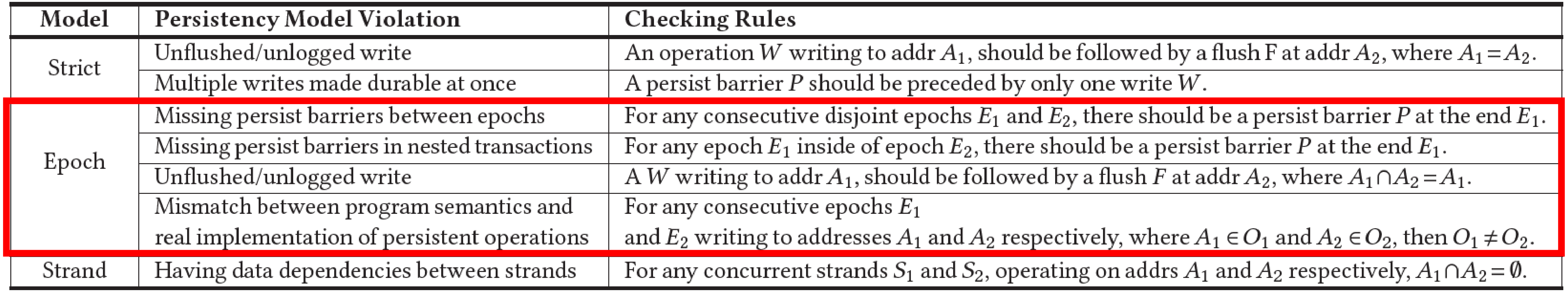


Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

Strict Persistency

- Every write is followed by a flush.
- Every flush is preceded by a single write.

Persistency Model Violations: Checking Rules



Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

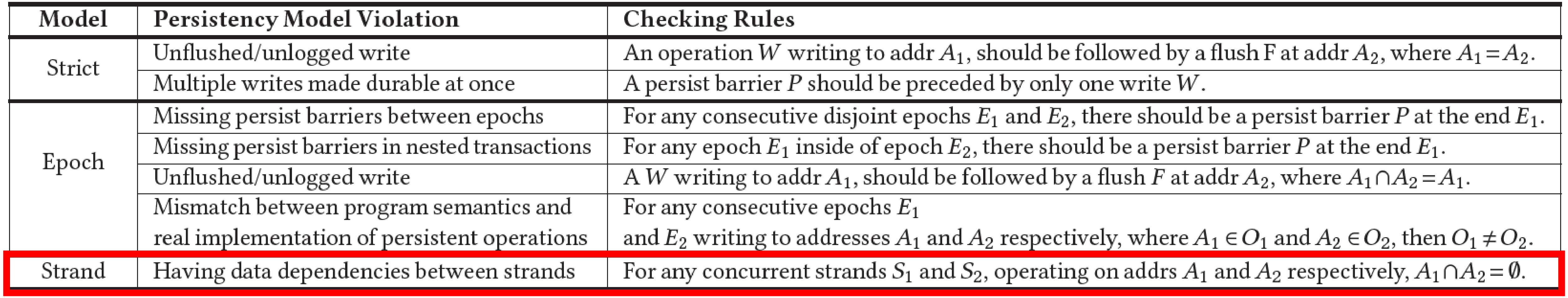
Strict Persistency

- Every write is followed by a flush.
- Every flush is preceded by a single write.

Epoch Persistency

- Every epoch is followed by a flush.
- Consecutive or nested epochs have barriers between them.

Persistency Model Violations: Checking Rules



Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

Strict Persistency

- Every write is followed by a flush.
- Every flush is preceded by a single write.

Epoch Persistency

- Every epoch is followed by a flush.
- Consecutive or nested epochs have barriers between them.

Strand Persistency

- Different strands should write to different addresses.

Persistency Model Violations: Checking Rules

Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

Strict Persistency

- Every write is followed by a flush.
- Every flush is preceded by a single write.

Epoch Persistency

- Every epoch is followed by a flush.
- Consecutive or nested epochs have barriers between them.

Strand Persistency

- Different strands should write to different addresses.

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Performance Bugs: Flushing Unmodified Data

```
static int pi_task_construct (PMEMobjpool *pop, void *ptr, void *arg) {
    struct pi_task *t = (struct pi_task *) ptr;
    struct pi_task_proto *p = (struct pi_task_proto *) arg;
    t->proto = *p;
    pmemobj_persist (pop, t, sizeof(*t));
    return 0;
}
```

pi_task_construct from PMDK

Performance Bugs: Flushing Unmodified Data

```
static int pi_task_construct (PMEMobjpool *pop, void *ptr, void *arg) {
    struct pi_task *t = (struct pi_task *) ptr;
    struct pi_task_proto *p = (struct pi_task_proto *) arg;
    t->proto = *p;
    pmemobj_persist (pop, t, sizeof(*t));
    return 0;
}
```

Persist entire object when only one field is modified.

pi_task_construct from PMDK

Performance Bugs: Flushing Unmodified Data

```
static int pi_task_construct (PMEMobjpool *pop, void *ptr, void *arg) {
    struct pi_task *t = (struct pi_task *) ptr;
    struct pi_task_proto *p = (struct pi_task_proto *) arg;
    t->proto = *p;
    pmemobj_persist (pop, t, sizeof(*t));
    return 0;
}
```

Persist entire object when only one field is modified.

pi_task_construct from PMDK

Flushing unmodified data hurts performance!

Performance Bugs: Redundant Write-Backs of Data

```
void nvm_free_callback (nvm_free_ctx *ctx) {
    .......
    nvm_free_blk (heap, nvb);
    nvm_flushl (nvb);
}

void nvm_free_blk (nvm_heap *heap, nvm_blk *nvb) {
    .......
    nvm_flushl (nvb);
}
```

nvm_free from NVM-Direct

Performance Bugs: Redundant Write-Backs of Data

```
void nvm_free_callback (nvm_free_ctx *ctx) {
    .......
    nvm_free_blk (heap, nvb);
    nvm_flushl (nvb);
}

void nvm_free_blk (nvm_heap *heap, nvm_blk *nvb) {
    .......
    nvm_flushl (nvb);
}
```

nvm_free from NVM-Direct

Performance Bugs: Redundant Write-Backs of Data

```
void nvm_free_callback (nvm_free_ctx *ctx) {
    .......
    nvm_free_blk (heap, nvb);
    nvm_flushl (nvb);
}

void nvm_free_blk (nvm_heap *heap, nvm_blk *nvb) {
    .......
    nvm_flushl (nvb);
}
```

Flushing twice in a row.

nvm_free from NVM-Direct

Performance Bugs: Redundant Write-Backs of Data

```
void nvm_free_callback (nvm_free_ctx *ctx) {
    .......
    nvm_free_blk (heap, nvb);
    nvm_flushl (nvb);
}

void nvm_free_blk (nvm_heap *heap, nvm_blk *nvb) {
    .......
    nvm_flushl (nvb);
}
```

Flushing twice in a row.

nvm_free from NVM-Direct

Redundant flushing does not affect correctness and hurts performance!

Performance Bugs: Transactions without Updates

```
static int timer_tick (uint32_t *timer) {
    int ret = *timer == 0 || ((*timer)--) == 0;
    pmemobj_persist (pop, timer, sizeof (*timer));
    return ret;
}
static void process_aliens (void) {
    .........
    if (timer_tick (&iter->timer)) {
        iter->timer = MAX_ALIEN_TIMER;
        iter->y++;
    }
    pmemobj_persist (pop, iter, sizeof (struct alien));
    .........
}
```

pm_invaders from PMDK examples

Performance Bugs: Transactions without Updates

```
static int timer_tick (uint32_t *timer) {
    int ret = *timer == 0 || ((*timer)--) == 0;
    pmemobj_persist (pop, timer, sizeof (*timer));
    return ret;
}
static void process_aliens (void) {
    .........
    if (timer_tick (&iter->timer)) {
        iter->timer = MAX_ALIEN_TIMER;
        iter->y++;
    }
    pmemobj_persist (pop, iter, sizeof (struct alien));
    .........
}
```

Persist alien object

pm_invaders from PMDK examples

Performance Bugs: Transactions without Updates

```
static int timer_tick (uint32_t *timer) {
    int ret = *timer == 0 || ((*timer)--) == 0;
    pmemobj_persist (pop, timer, sizeof (*timer));
    return ret;
}
static void process_aliens (void) {
    .........
    if (timer_tick (&iter->timer)) {
        iter->timer = MAX_ALIEN_TIMER;
        iter->y++;
    }
    pmemobj_persist (pop, iter, sizeof (struct alien));
    .........
}
```

Object is unmodified if condition is false!

Persist alien object

pm_invaders from PMDK examples

Performance Bugs: Transactions without Updates

```
static int timer_tick (uint32_t *timer) {
    int ret = *timer == 0 || ((*timer)--) == 0;
    pmemobj_persist (pop, timer, sizeof (*timer));
    return ret;
}
static void process_aliens (void) {
    .........
    if (timer_tick (&iter->timer)) {
        iter->timer = MAX_ALIEN_TIMER;
        iter->y++;
    }
    pmemobj_persist (pop, iter, sizeof (struct alien));
    .........
}
```

Object is unmodified if condition is false!

Persist alien object

pm_invaders from PMDK examples

Transactions without updates enforce unnecessary orderings!

Performance Bugs: Checking Rules

Flushing Unmodified Data

Redundant Write-Backs of Updated Data

Durable Transactions Without Updates

Performance Bugs: Checking Rules

Flushing Unmodified Data

Every flush should have a preceding write

Redundant Write-Backs of Updated Data

Durable Transactions Without Updates

Performance Bugs: Checking Rules

Flushing Unmodified Data → Every flush should have a preceding write

Redundant Write-Backs of Updated Data → Consecutive flushes should not flush the same address

Durable Transactions Without Updates

Performance Bugs: Checking Rules

Flushing Unmodified Data → Every flush should have a preceding write

Redundant Write-Backs of Updated Data → Consecutive flushes should not flush the same address

Durable Transactions Without Updates → Every transaction should contain at least one write

Classifying Persistency Bugs in NVM Programs

Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Classifying Persistency Bugs in NVM Programs

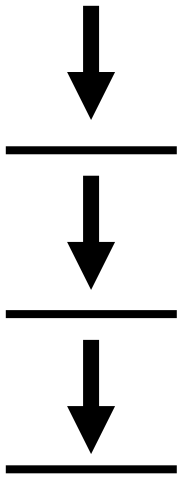
Persistency Model Violations

- Semantic Mismatch
- Unflushed/Unlogged Writes
- Missing Persist Barrier

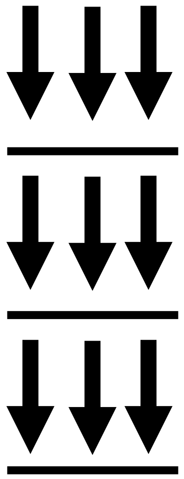
Performance Bugs

- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

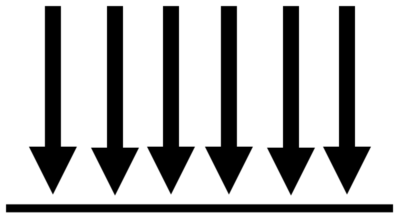
Persistency Models for Non-Volatile Memory



Strict Persistency



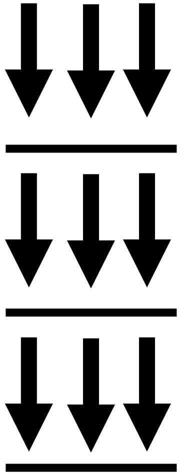
Epoch Persistency



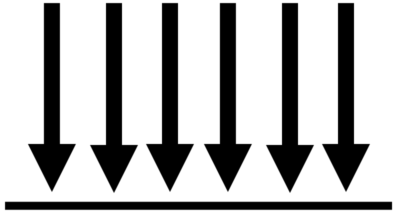
Strand Persistency

Persistency Models for Non-Volatile Memory

Static Analysis



Epoch Persistency



Strand Persistency

Strict Persistency rules can be checked statically!

Persistency Models for Non-Volatile Memory

Static Analysis

Dynamic Analysis

Detecting data races between strands or epochs requires dynamic analysis!

Persistency Models for Non-Volatile Memory

Static Analysis

Dynamic Analysis

The static and dynamic components combine to check **all** rules

Detecting Persistency Bugs in NVM Programs

- Semantic Mismatch

- Unflushed/Unlogged Writes
- Missing Persist Barrier
- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Detecting Persistency Bugs in NVM Programs

- Semantic Mismatch

- Unflushed/Unlogged Writes
- Missing Persist Barrier
- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

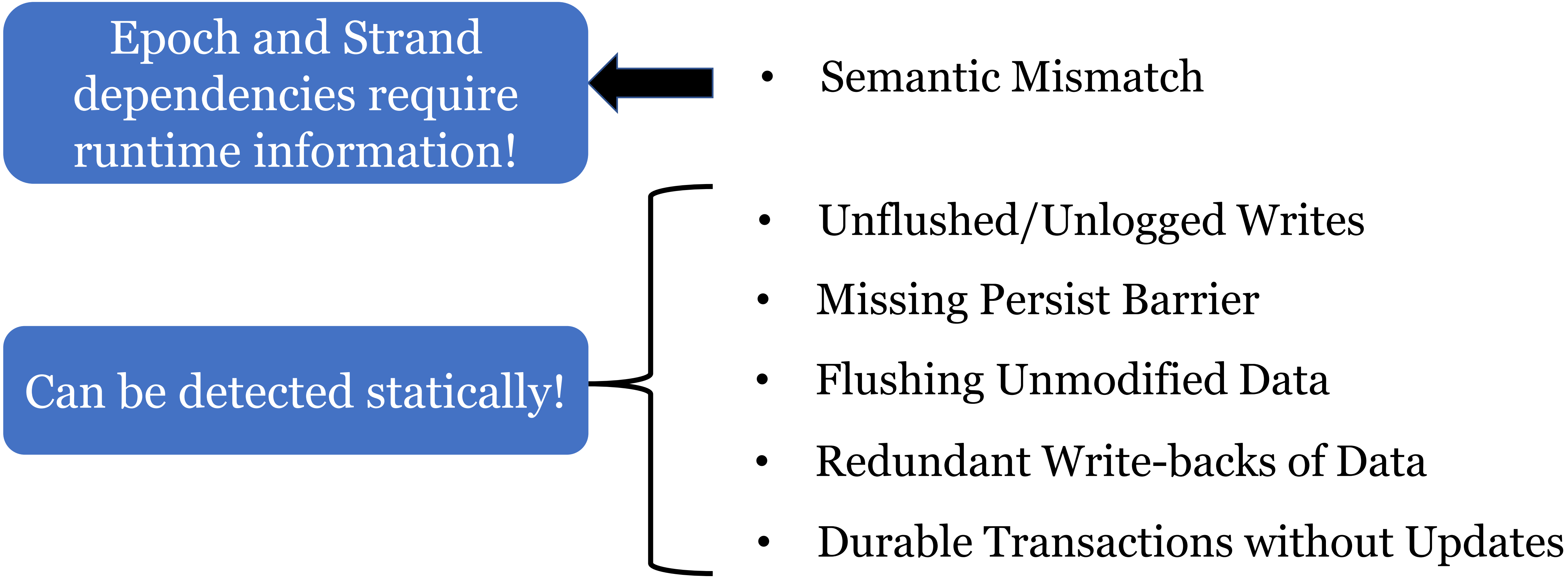
Detecting Persistency Bugs in NVM Programs

- Semantic Mismatch

Can be detected statically!

- Unflushed/Unlogged Writes
- Missing Persist Barrier
- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Detecting Persistency Bugs in NVM Programs



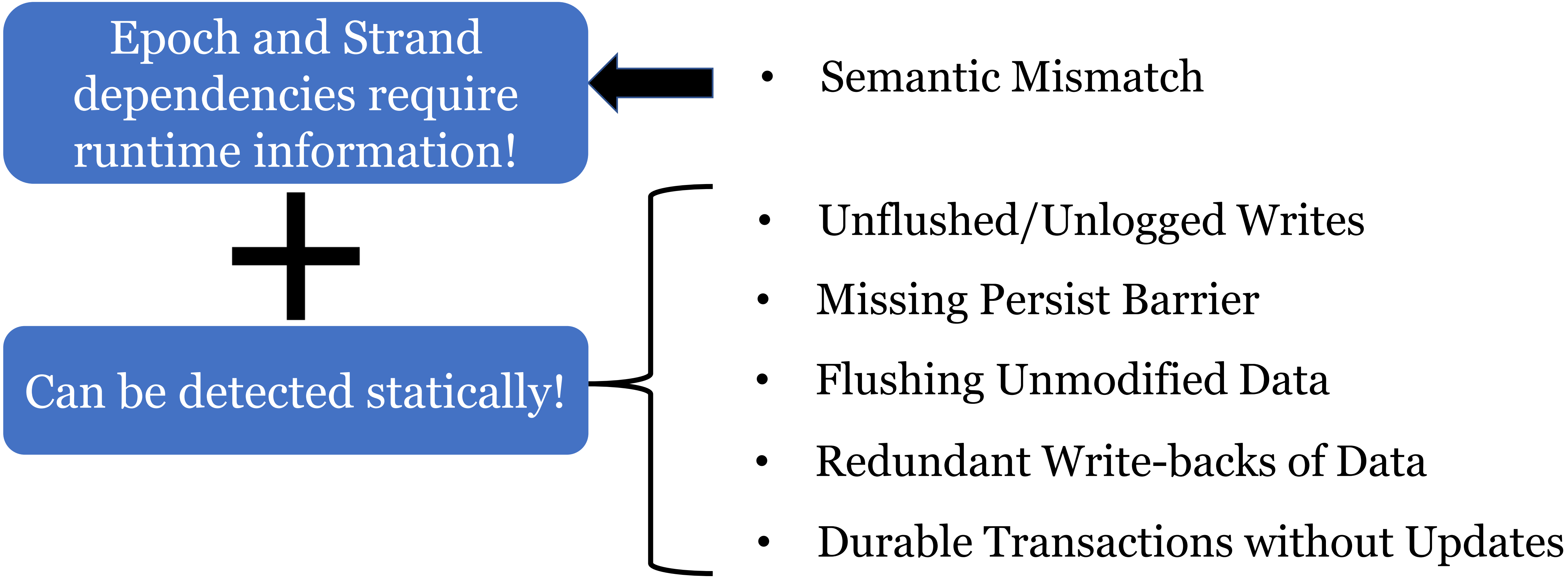
Epoch and Strand dependencies require runtime information!

- Semantic Mismatch

Can be detected statically!

- Unflushed/Unlogged Writes
- Missing Persist Barrier
- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

Detecting Persistency Bugs in NVM Programs



Epoch and Strand dependencies require runtime information!

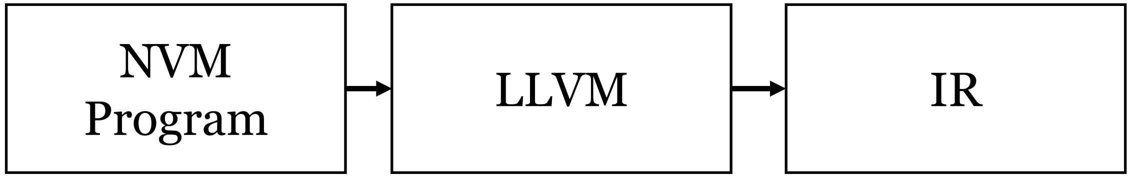
- Semantic Mismatch

Can be detected statically!

- Unflushed/Unlogged Writes
- Missing Persist Barrier
- Flushing Unmodified Data
- Redundant Write-backs of Data
- Durable Transactions without Updates

We introduce a static and dynamic component to check **all** rules!

Detecting Memory Persistency Bugs with DeepMC



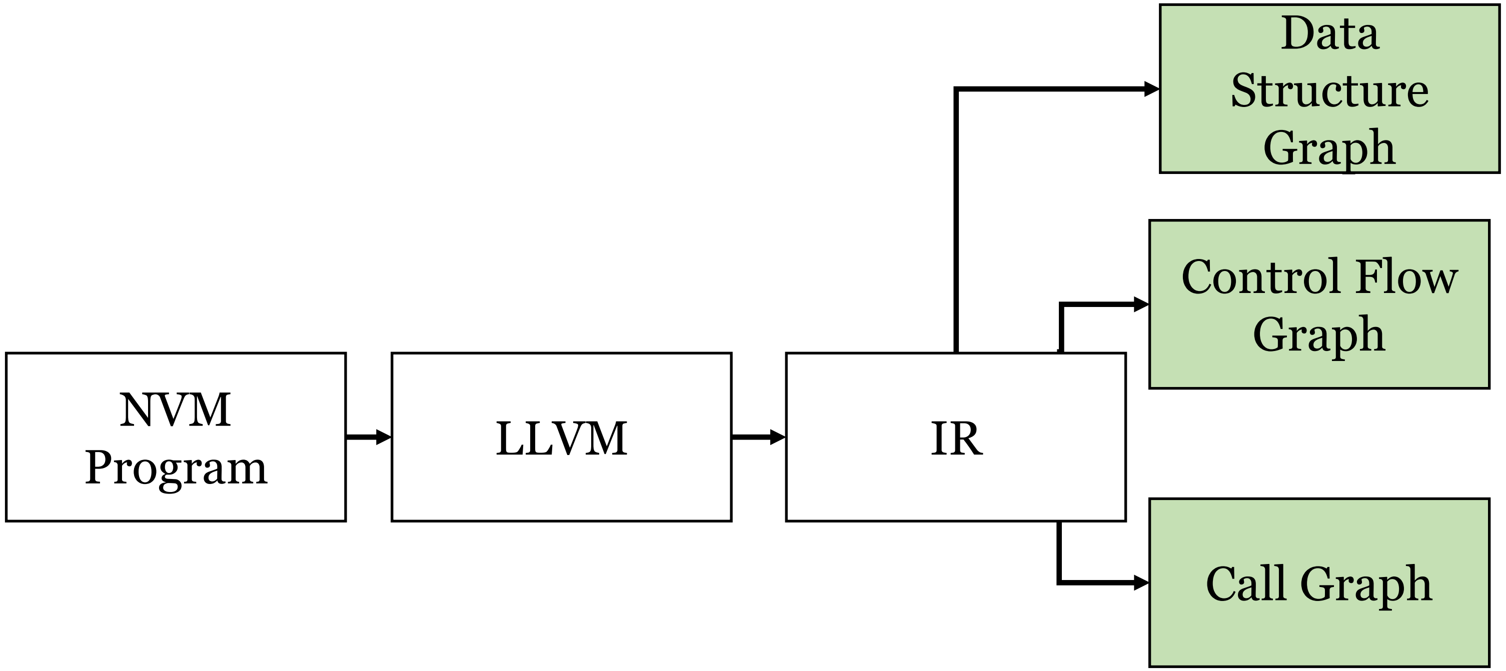
NVM Program

LLVM

IR

Compile the program into LLVM IR

Detecting Memory Persistency Bugs with DeepMC



Apply data structure analysis!

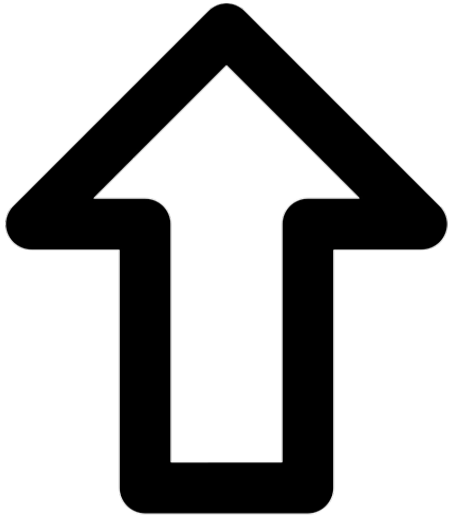
Adapting Data Structure Analysis to Persistent Objects

Phase 1:
Local Analysis

Create nodes for functions and new variables with edges for dependencies

Adapting Data Structure Analysis to Persistent Objects

Phase 1:
Local Analysis

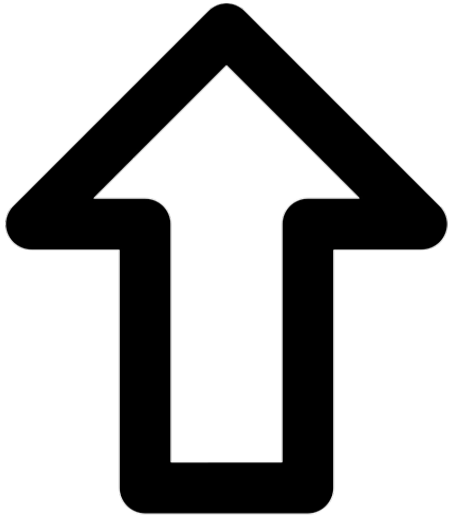


Phase 2:
Bottom-Up Analysis

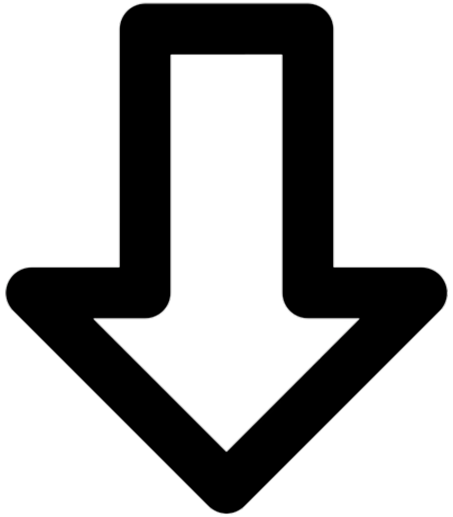
Resolve updates occurring in function calls with callee information

Adapting Data Structure Analysis to Persistent Objects

Phase 1:
Local Analysis



Phase 2:
Bottom-Up Analysis



Phase 3:
Top-Down Analysis

Include caller information to finalize the data structure graph

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct

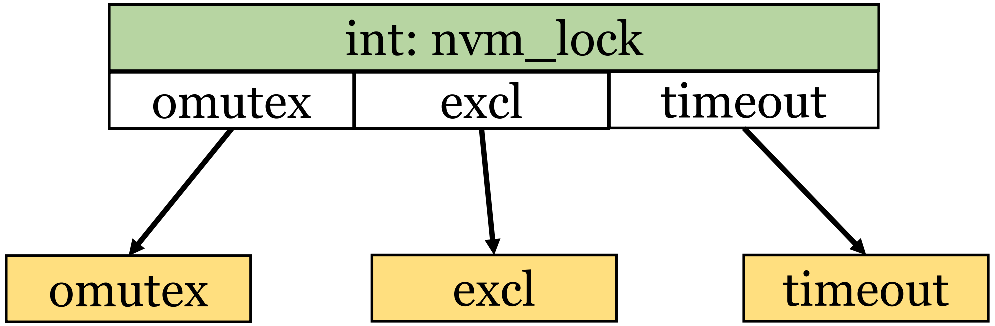
Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct

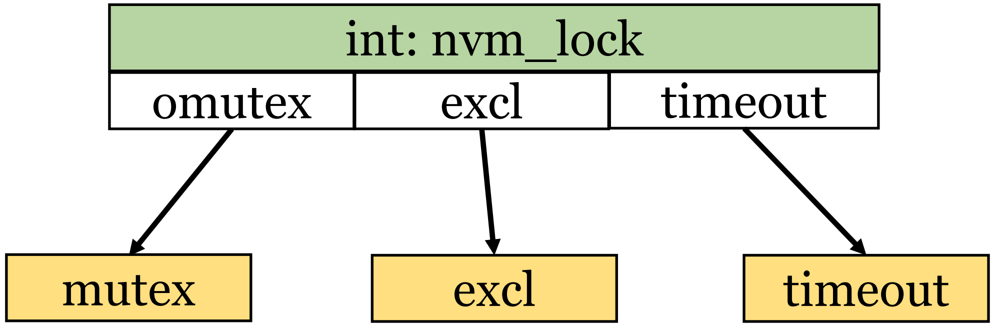


Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```



int: nvm_lock

omutex | excl | timeout

mutex

excl

timeout

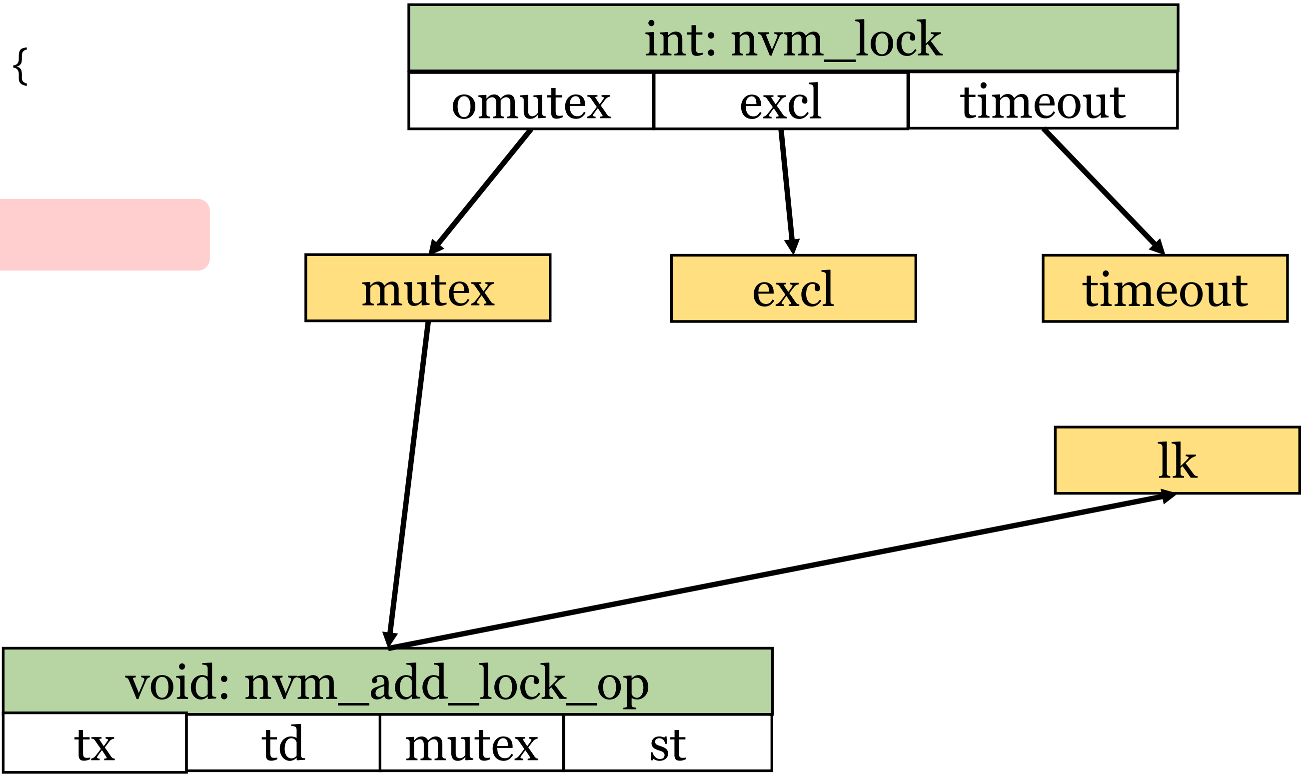
nvm_lock from NVM-Direct

Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```


int: nvm_lock
omutex
excl
timeout
mutex
excl
timeout
lk
void: nvm_add_lock_op
tx
td
mutex
st

nvm_lock from NVM-Direct

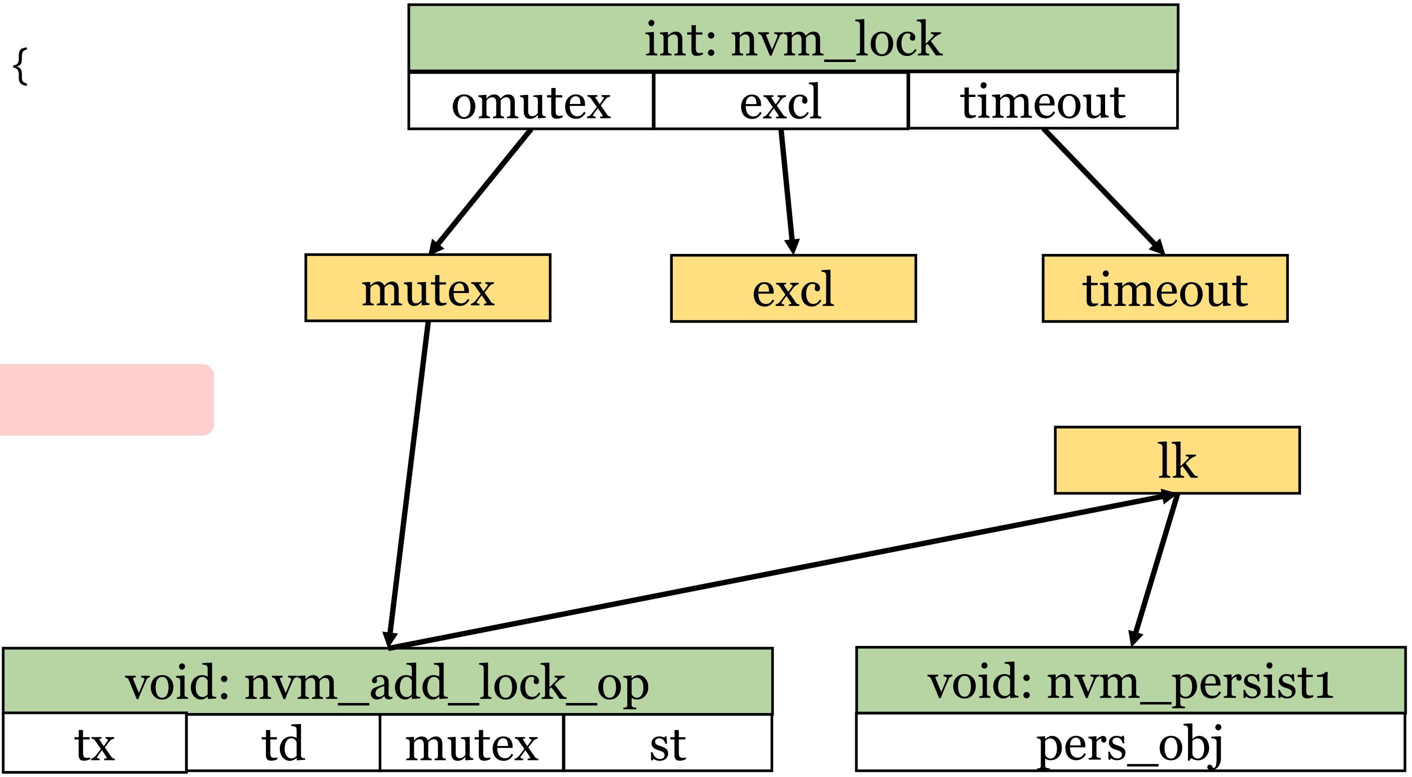
Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct

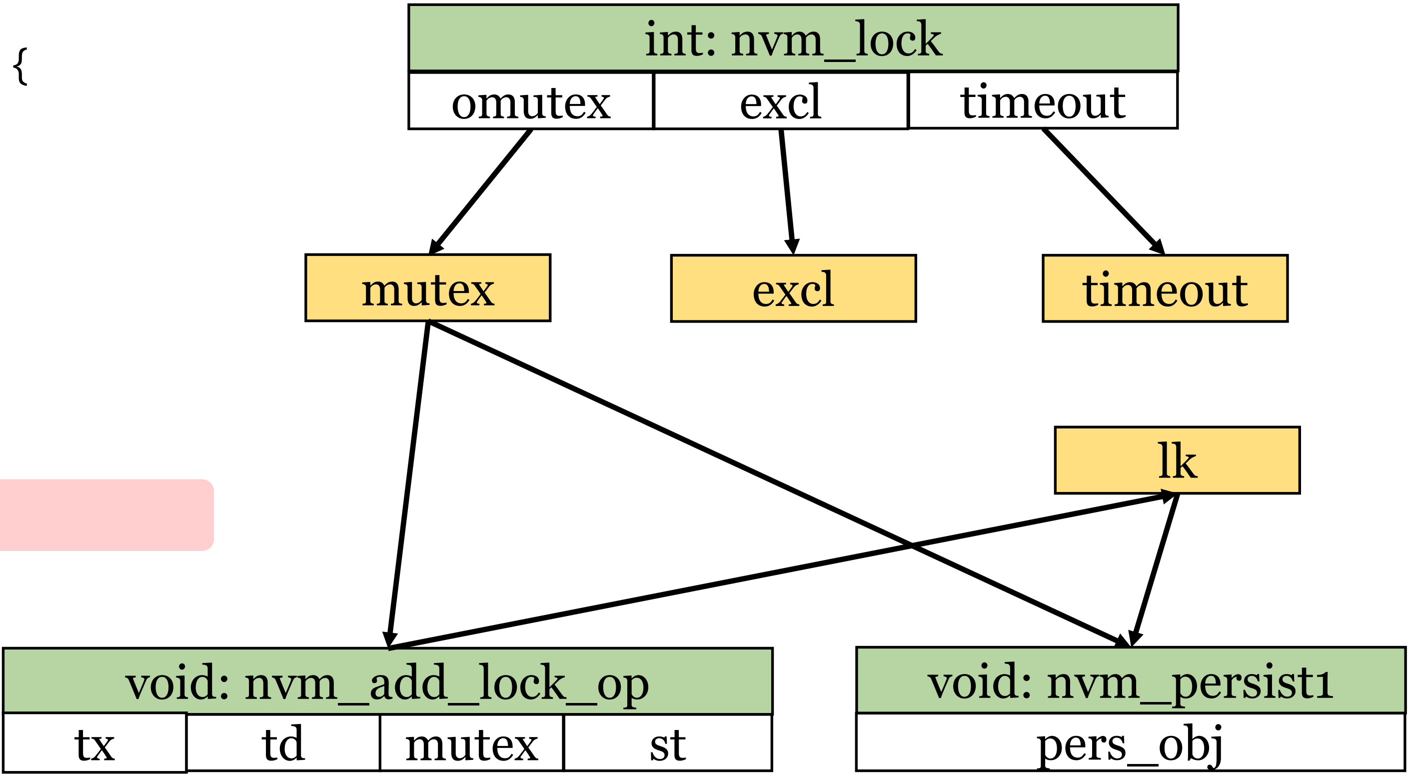


Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```



nvm_lock from NVM-Direct

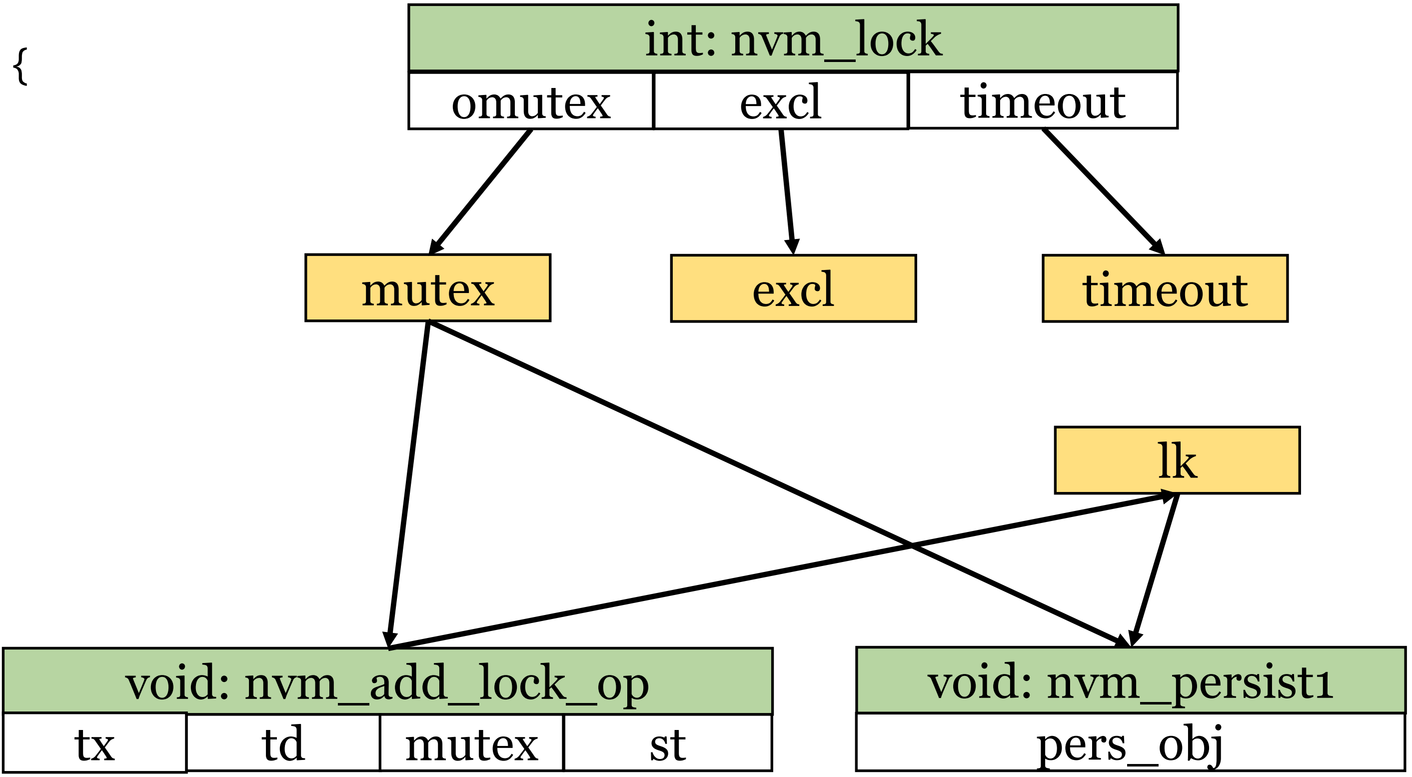
Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct



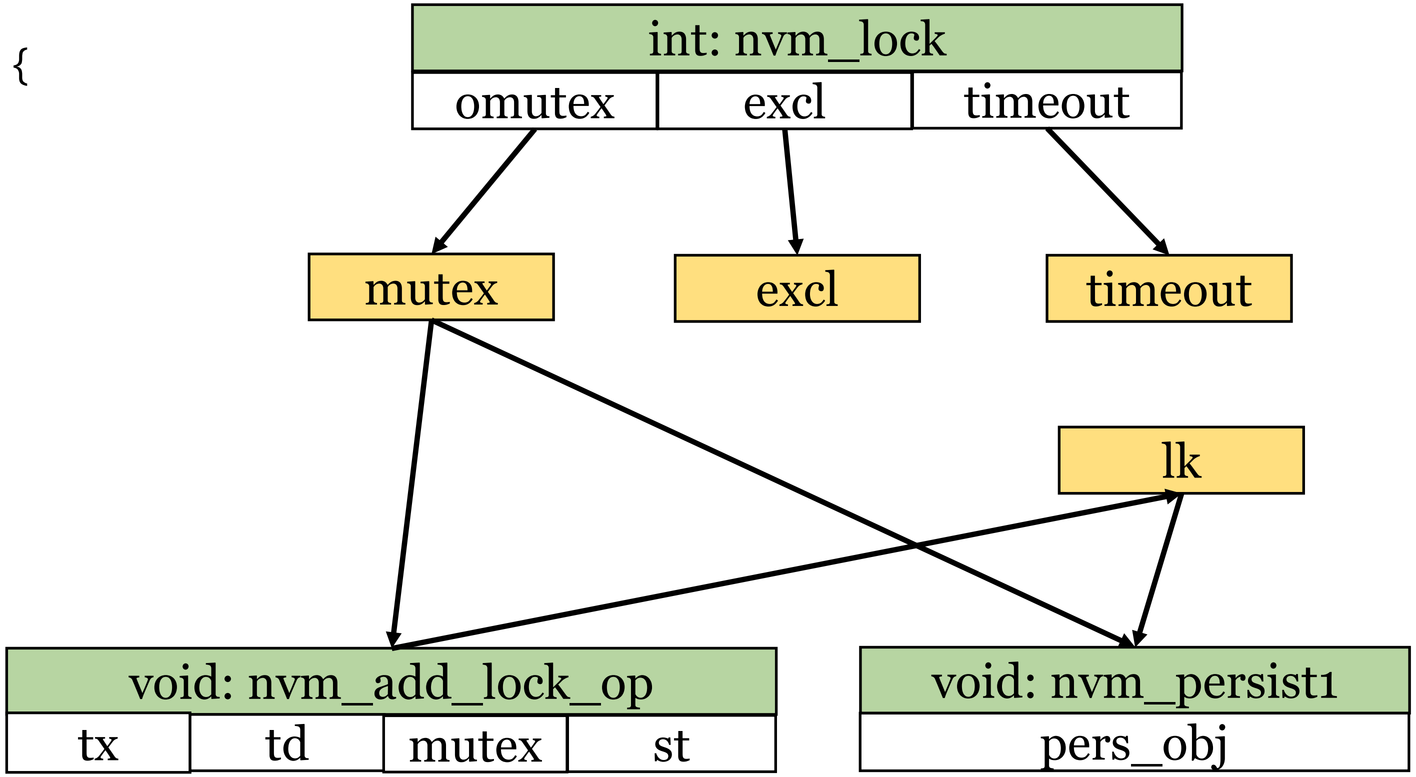
Data structure graph for nvm_lock

Phase 1: Local Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     . . .
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     . . .
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct

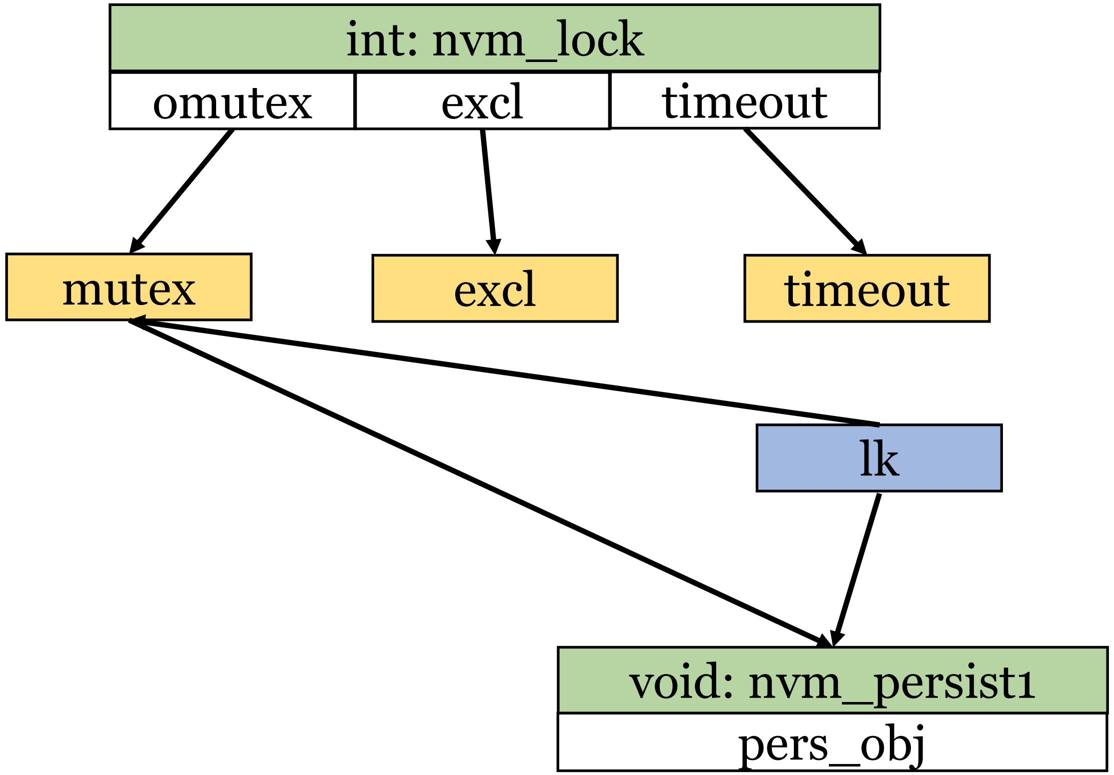


Data structure graph for nvm_lock

Phase 2: Bottom-Up Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```



nvm_lock from NVM-Direct

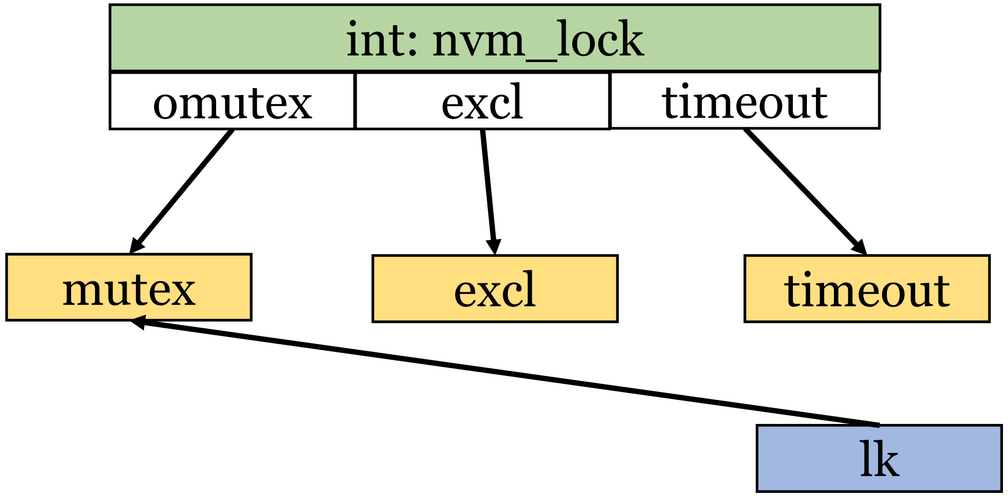
Data structure graph for nvm_lock

Phase 2: Bottom-Up Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
      nvm_amutex *mutex = (nvm_amutex*)omutex;
       ...
      nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
       ...
      lk->state = nvm_lock_acquire_s;
      nvm_persist1(&lk->state);
      mutex->owners--;
      nvm_persist1(&mutex->owners);
      if (mutex->level > lk->new_level)
          lk->new_level = mutex->level;
      lk->state = nvm_lock_held_s;
      nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct



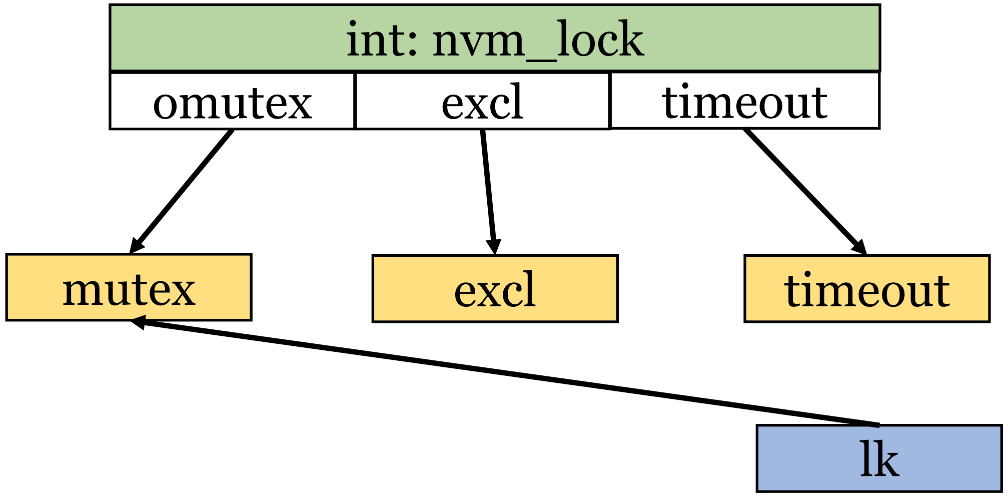
Data structure graph for nvm_lock

Phase 2: Bottom-Up Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct



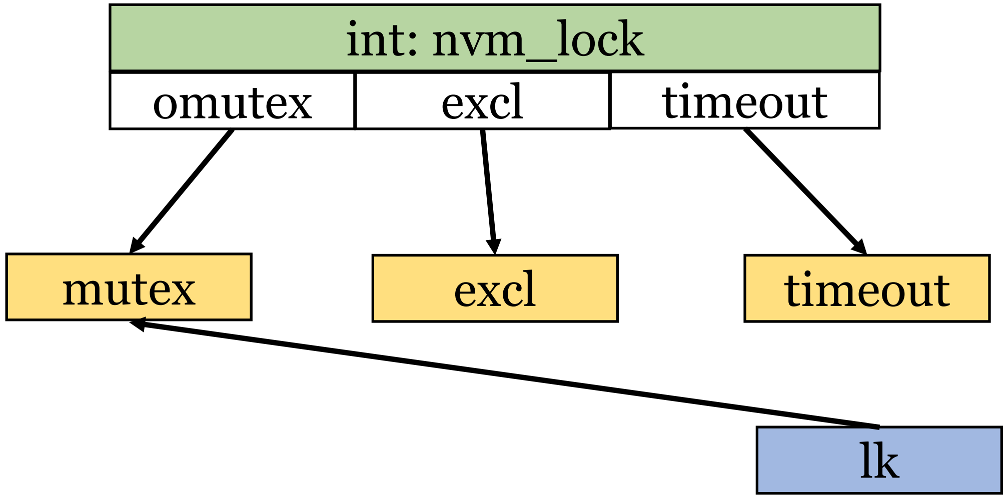
Data structure graph for nvm_lock

Phase 2: Bottom-Up Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

nvm_lock from NVM-Direct



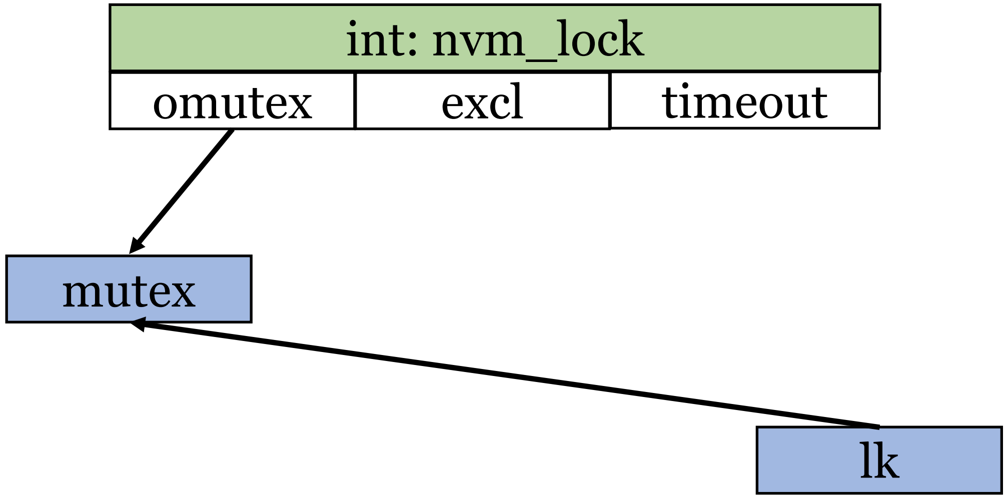
Data structure graph for nvm_lock

Phase 3: Top-Down Analysis

Adapting Data Structure Analysis to Persistent Objects

```
int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
    nvm_amutex *mutex = (nvm_amutex*)omutex;
     ...
    nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
     ...
    lk->state = nvm_lock_acquire_s;
    nvm_persist1(&lk->state);
    mutex->owners--;
    nvm_persist1(&mutex->owners);
    if (mutex->level > lk->new_level)
        lk->new_level = mutex->level;
    lk->state = nvm_lock_held_s;
    nvm_persist1(&lk->state);
}
```

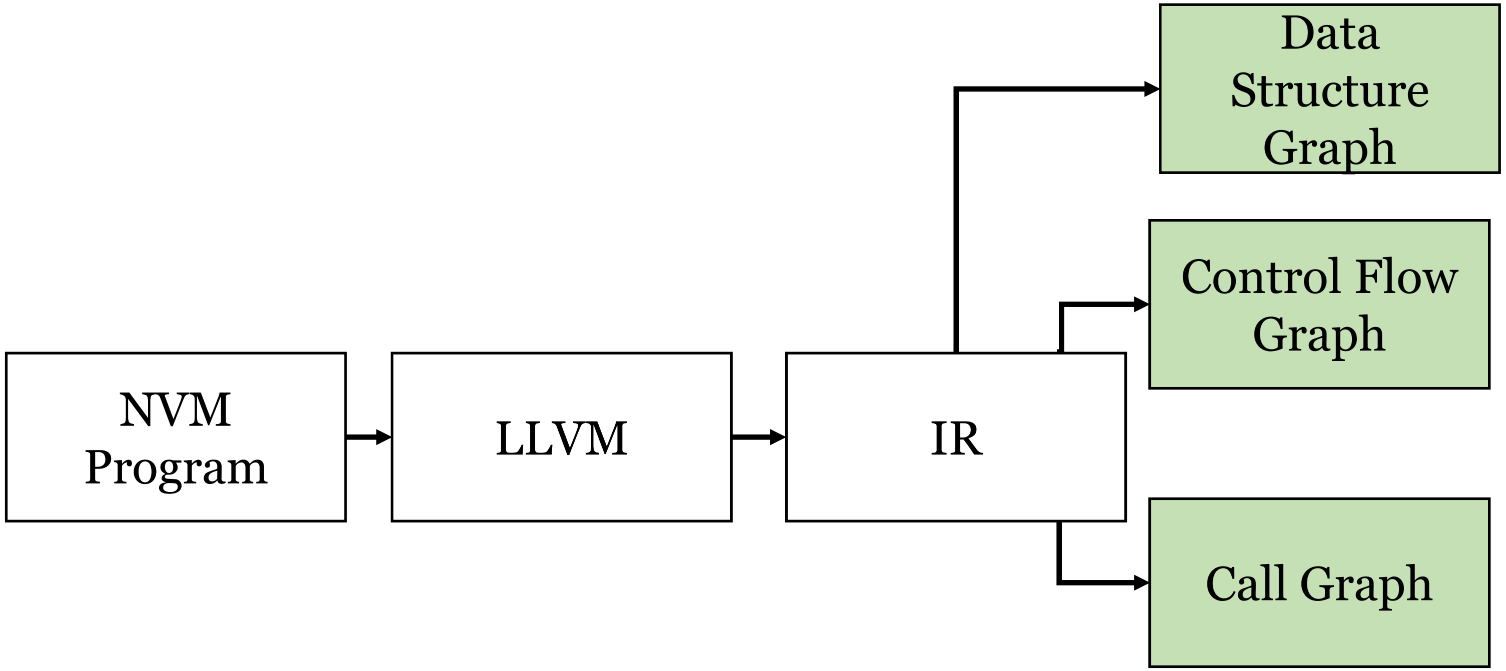
nvm_lock from NVM-Direct



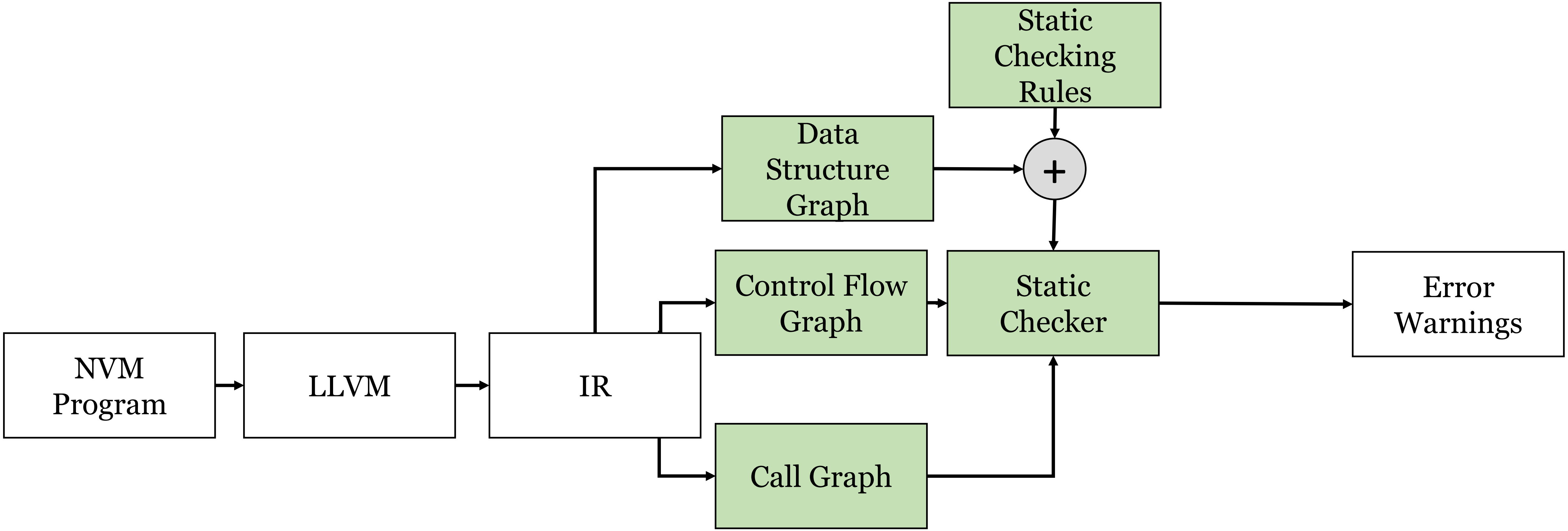
Data structure graph for nvm_lock

Phase 3: Top-Down Analysis

Detecting Memory Persistency Bugs with DeepMC



Detecting Memory Persistency Bugs with DeepMC



Combine with checking rules for static checking

Applying the Data Structure Graph to NVM Programs

Local Trace
in function A

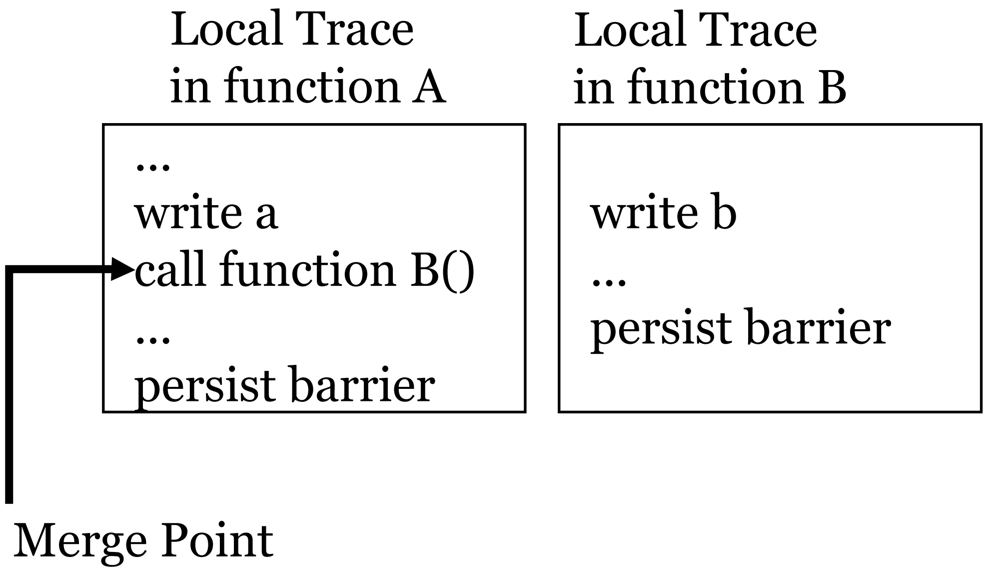
…
write a
call function B()
…
persist barrier

Local Trace
in function B

write b
…
persist barrier

Traverse control flow graph in depth-first order

Applying the Data Structure Graph to NVM Programs



Local Trace
in function A

```
...
write a
call function B()
...
persist barrier
```

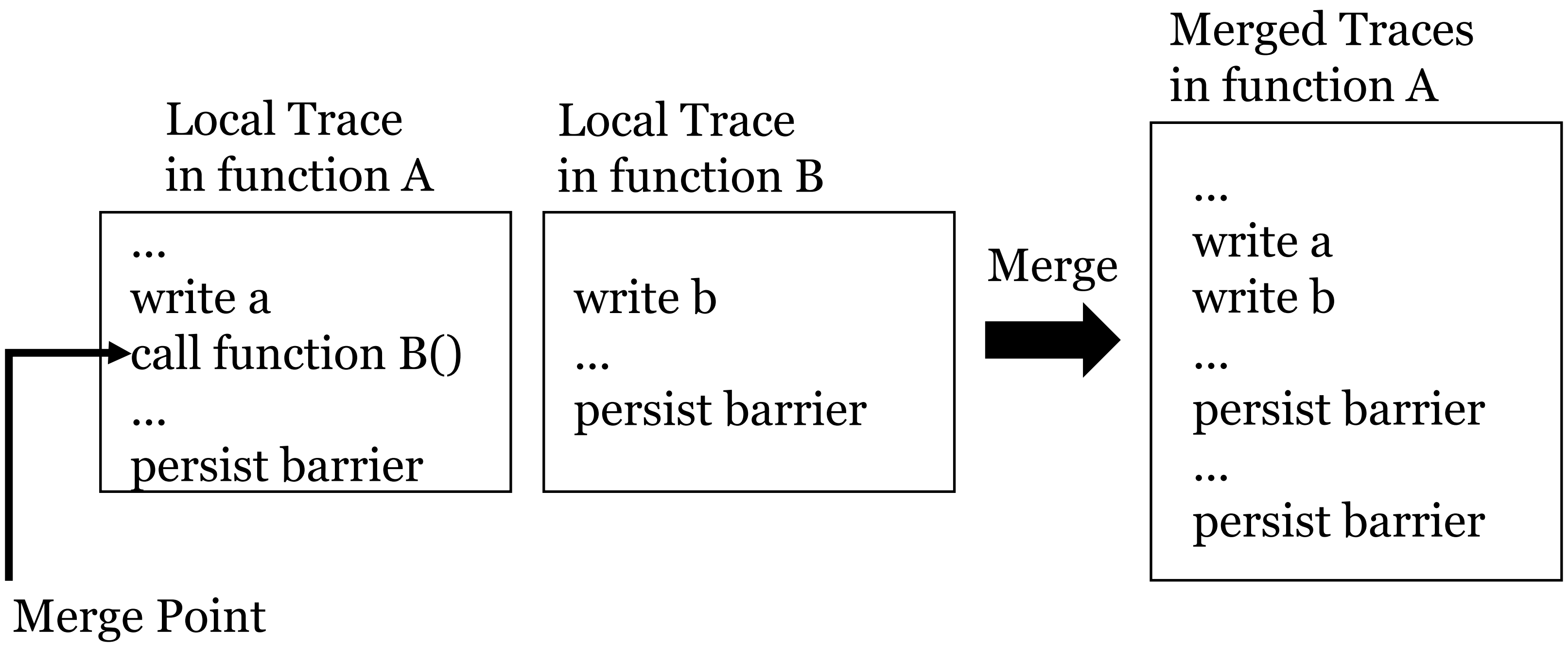
Local Trace
in function B

```
write b
...
persist barrier
```

Merge Point

Traverse control flow graph in depth-first order

Applying the Data Structure Graph to NVM Programs



Local Trace
in function A

...
write a
call function B()
...
persist barrier

Merge Point

Local Trace
in function B

write b
...
persist barrier

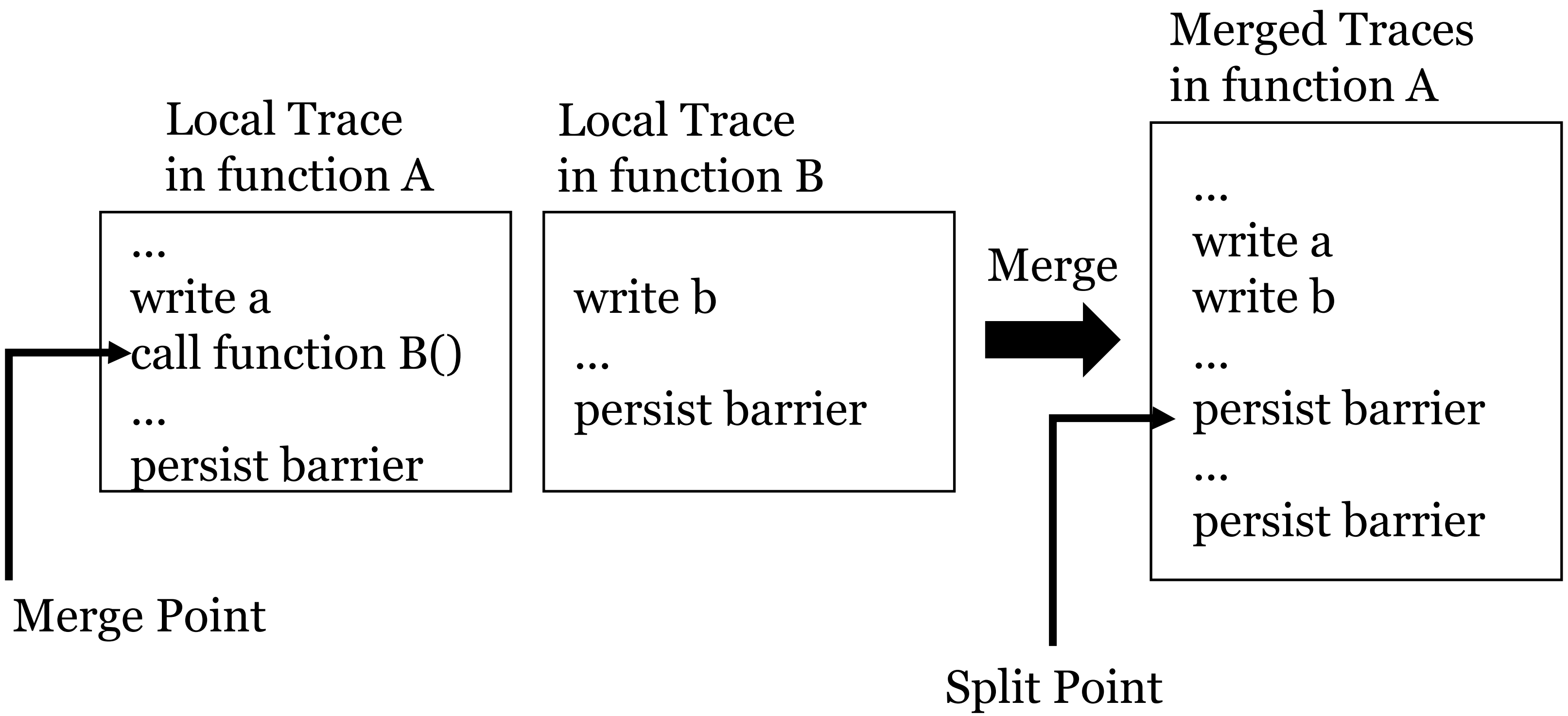
Merge

Merged Traces
in function A

...
write a
write b
...
persist barrier
...
persist barrier

Merge function calls into their call sites

Applying the Data Structure Graph to NVM Programs



Local Trace
in function A

...
write a
call function B()
...
persist barrier

Local Trace
in function B

write b
...
persist barrier

Merge

Merged Traces
in function A

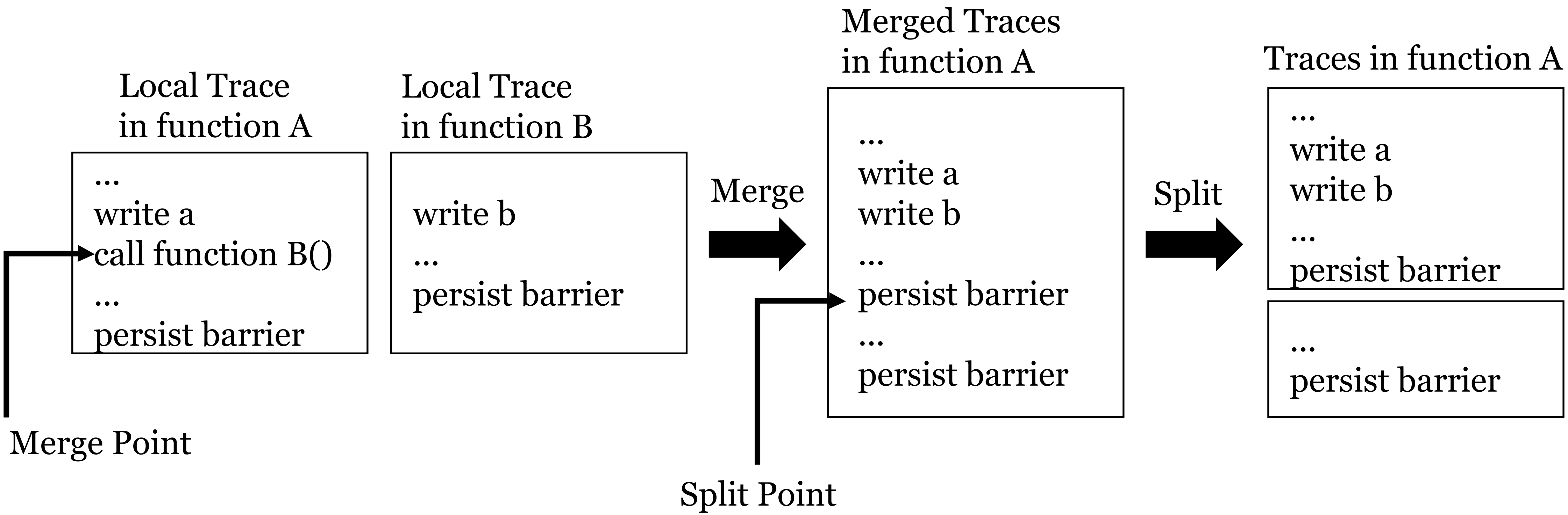
...
write a
write b
...
persist barrier
...
persist barrier

Merge Point

Split Point

Merge function calls into their call sites

Applying the Data Structure Graph to NVM Programs



Local Trace
in function A

…
write a
call function B()
…
persist barrier

Merge Point

Local Trace
in function B

write b
…
persist barrier

Merge

Merged Traces
in function A

…
write a
write b
…
persist barrier
…
persist barrier

Split Point

Split

Traces in function A

…
write a
write b
…
persist barrier

…
persist barrier

Split into smaller traces at persistent barriers

Applying our Checking Rules to Static Analysis

Op	Line	Obj

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
          ...
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
          ...
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
          ...
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
          ...
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state

Trace

Applying our Checking Rules to Static Analysis

```
1    int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2        nvm_amutex *mutex = (nvm_amutex*)omutex;
          …
3        nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
          …
4        lk->state = nvm_lock_acquire_s;
5        nvm_persist1(&lk->state);
6        mutex->owners--;
7        nvm_persist1(&mutex->owners);
8        if (mutex->level > lk->new_level)
9            lk->new_level = mutex->level;
10       lk->state = nvm_lock_held_s;
11       nvm_persist1(&lk->state);
12   }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
         ...
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
         ...
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
         ...
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
         ...
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners
Flush	7	owners
Fence	7	owners

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2         nvm_amutex *mutex = (nvm_amutex*)omutex;
           …
3         nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
           …
4         lk->state = nvm_lock_acquire_s;
5         nvm_persist1(&lk->state);
6         mutex->owners--;
7         nvm_persist1(&mutex->owners);
8         if (mutex->level > lk->new_level)
9              lk->new_level = mutex->level;
10        lk->state = nvm_lock_held_s;
11        nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners
Flush	7	owners
Fence	7	owners
Write	9	new_level

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
         …
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
         …
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners
Flush	7	owners
Fence	7	owners
Write	9	new_level
Write	10	state

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
          …
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
          …
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct

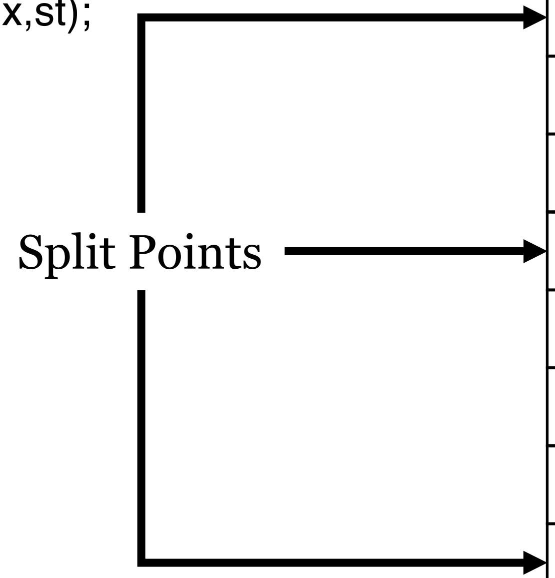
Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners
Flush	7	owners
Fence	7	owners
Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Trace

Applying our Checking Rules to Static Analysis

```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
         …
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
         …
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

nvm_lock from NVM-Direct


Split Points

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state
Write	6	owners
Flush	7	owners
Fence	7	owners
Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Trace

Applying our Checking Rules to Static Analysis

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Strict Persistency Checking Rules

Traces

Applying our Checking Rules to Static Analysis

Model Violations

Every write is followed by a flush

Every flush is preceded by a single write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✓
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✓
- Every flush is preceded by a single write ✓

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✓
- Every flush is preceded by a single write ✓

Performance Bugs

- Every flush should have a preceding write ✓
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✓
- Every flush is preceded by a single write ✓

Performance Bugs

- Every flush should have a preceding write ✓
- Flushes should flush different addresses ✓
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✓
- Every flush is preceded by a single write ✓

Performance Bugs

- Every flush should have a preceding write ✓
- Flushes should flush different addresses ✓
- Transactions must have at least one write ✓

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✗
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✗
- Every flush is preceded by a single write ✗

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush ✗
- Every flush is preceded by a single write ✗

Performance Bugs

- Every flush should have a preceding write ✓
- Flushes should flush different addresses ✓
- Transactions must have at least one write ✓

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Applying our Checking Rules to Static Analysis

Model Violations

- Every write is followed by a flush
- Every flush is preceded by a single write

Performance Bugs

- Every flush should have a preceding write
- Flushes should flush different addresses
- Transactions must have at least one write

Strict Persistency Checking Rules

Op	Line	Obj
Write	4	state
Flush	5	state
Fence	5	state

Write	6	owners
Flush	7	owners
Fence	7	owners

Write	9	new_level
Write	10	state
Flush	11	state
Fence	11	state

Traces

Dynamic Analysis for Epoch and Strand Persistency

Higher Possible Performance

Dynamic Analysis for Epoch and Strand Persistency

Higher Possible Performance

Read-after-Write Dependencies

Dynamic Analysis for Epoch and Strand Persistency

Higher Possible Performance

⚠ Read-after-Write Dependencies

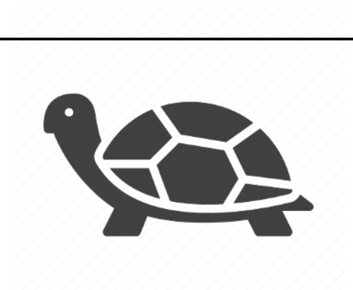
⚠ Write-after-Write Dependencies

Dynamic Analysis for Epoch and Strand Persistency

Dynamic Analysis

Dynamic Analysis for Epoch and Strand Persistency

Dynamic Analysis



High Overhead

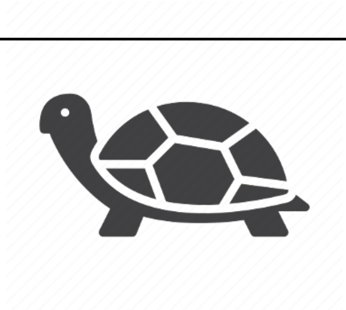
Dynamic Analysis for Epoch and Strand Persistency

Dynamic Analysis

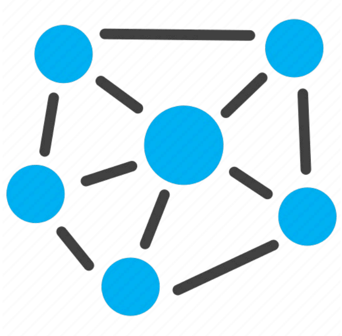
High Overhead

Dynamic Analysis for Epoch and Strand Persistency

Dynamic Analysis



High Overhead

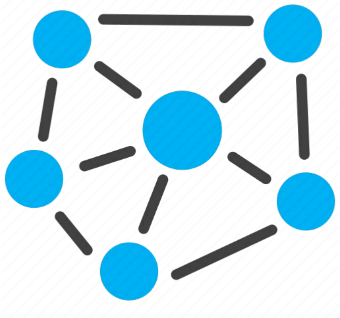


Use DSG to track only persistent objects!

Dynamic Analysis for Epoch and Strand Persistency

Dynamic Analysis

High Overhead



Use DSG to track only persistent objects!

Reuse existing library annotations!

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Checking for Epoch and Strand Violations

```
 Epoch 1            Epoch 2
begin_epoch;       begin_epoch;
x = a;             w = c;
y = b;             v = d;
barrier;           barrier;
z = x + y;         u = x + v*w;
end_epoch;         end_epoch;
```

Start tracking upon epoch annotations.

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment

u
v
w
x
y
z

Only include persistent object in the shadow segment

Checking for Epoch and Strand Violations

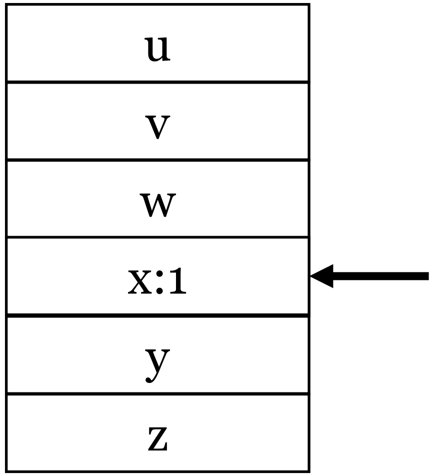
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



Checking for Epoch and Strand Violations

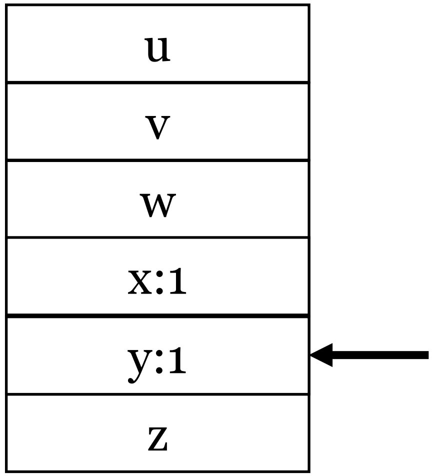
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



Checking for Epoch and Strand Violations

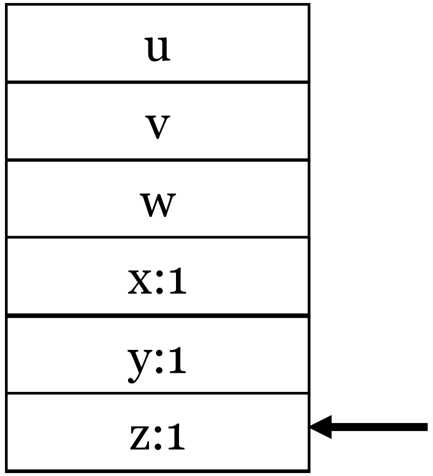
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



Checking for Epoch and Strand Violations

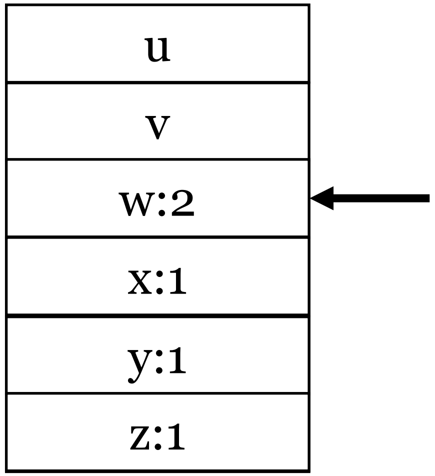
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



u
v
w:2
x:1
y:1
z:1

Checking for Epoch and Strand Violations

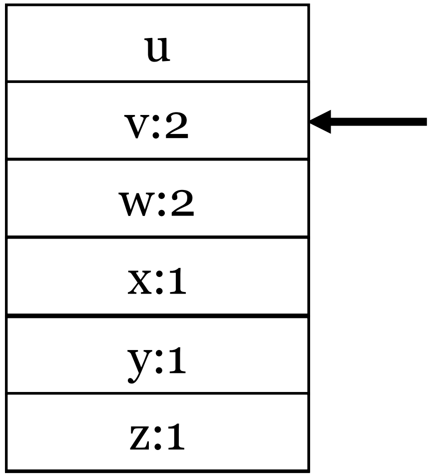
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



u
v:2
w:2
x:1
y:1
z:1

Checking for Epoch and Strand Violations

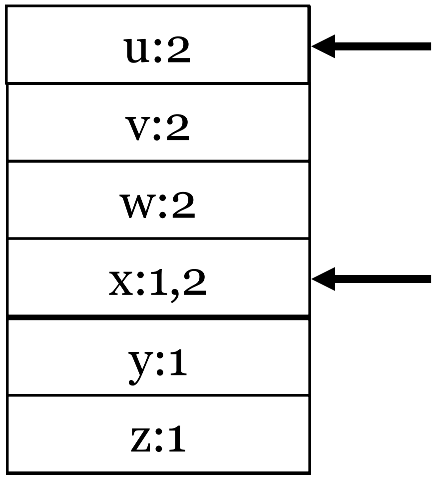
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment



u:2
v:2
w:2
x:1,2
y:1
z:1

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment

u:2
v:2
w:2
x:1,2
y:1
z:1

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment

u:2
v:2
w:2
x:1,2
y:1
z:1

End tracking with end of epochs

Checking for Epoch and Strand Violations

Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment

u:2
v:2
w:2
x:1,2
y:1
z:1

Checking for Epoch and Strand Violations

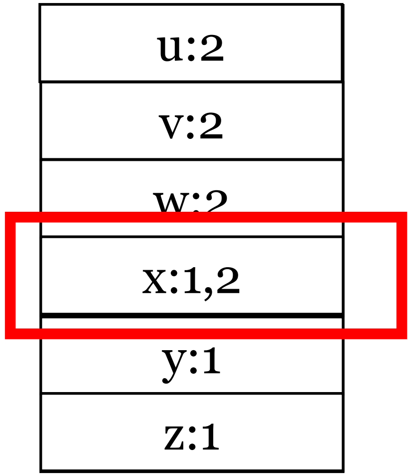
Epoch 1

```
begin_epoch;
x = a;
y = b;
barrier;
z = x + y;
end_epoch;
```

Epoch 2

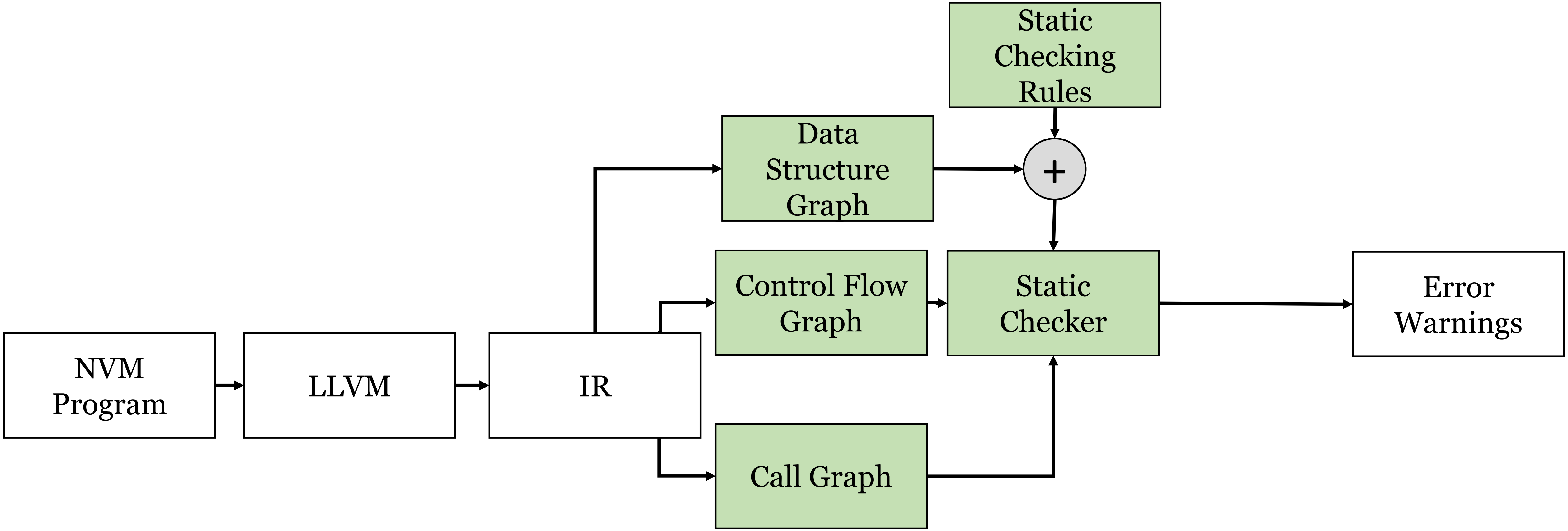
```
begin_epoch;
w = c;
v = d;
barrier;
u = x + v*w;
end_epoch;
```

Shadow Segment

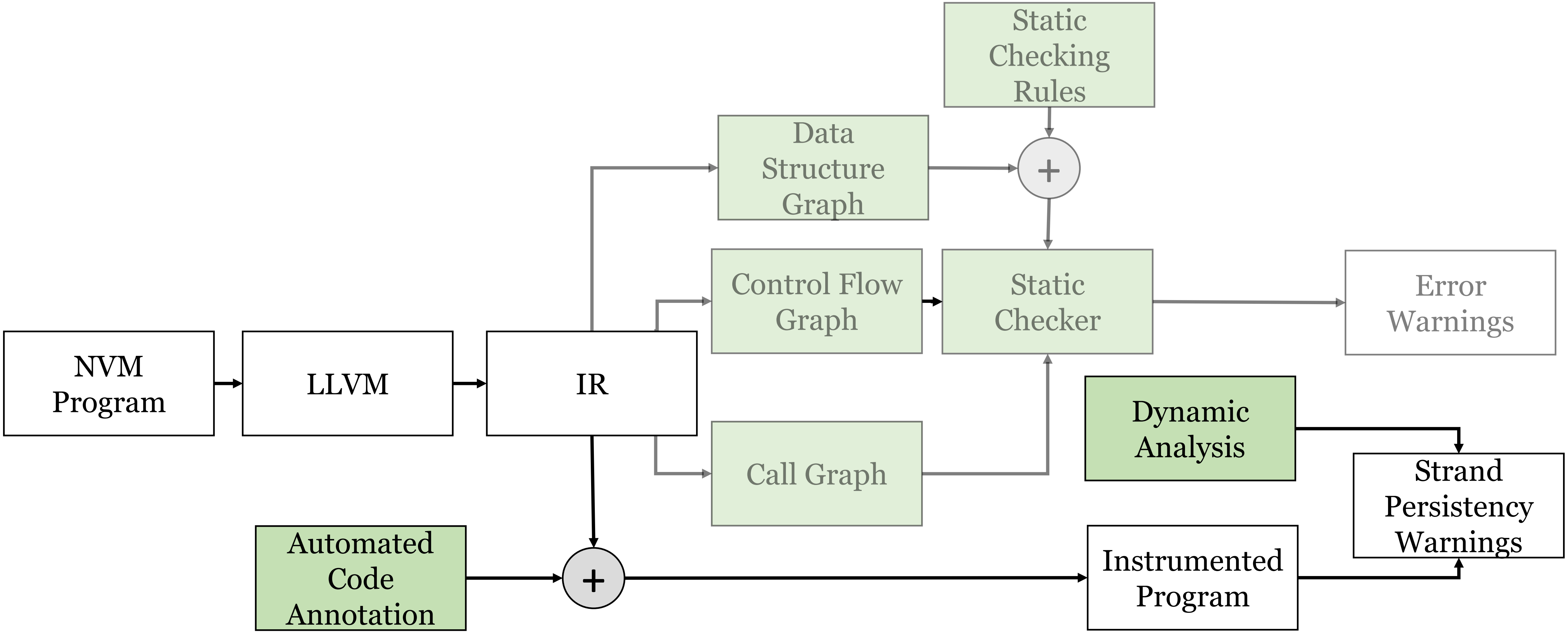

u:2
v:2
w:2
x:1,2
y:1
z:1

Accesses to x race and should be ordered!

Detecting Memory Persistency Bugs with DeepMC

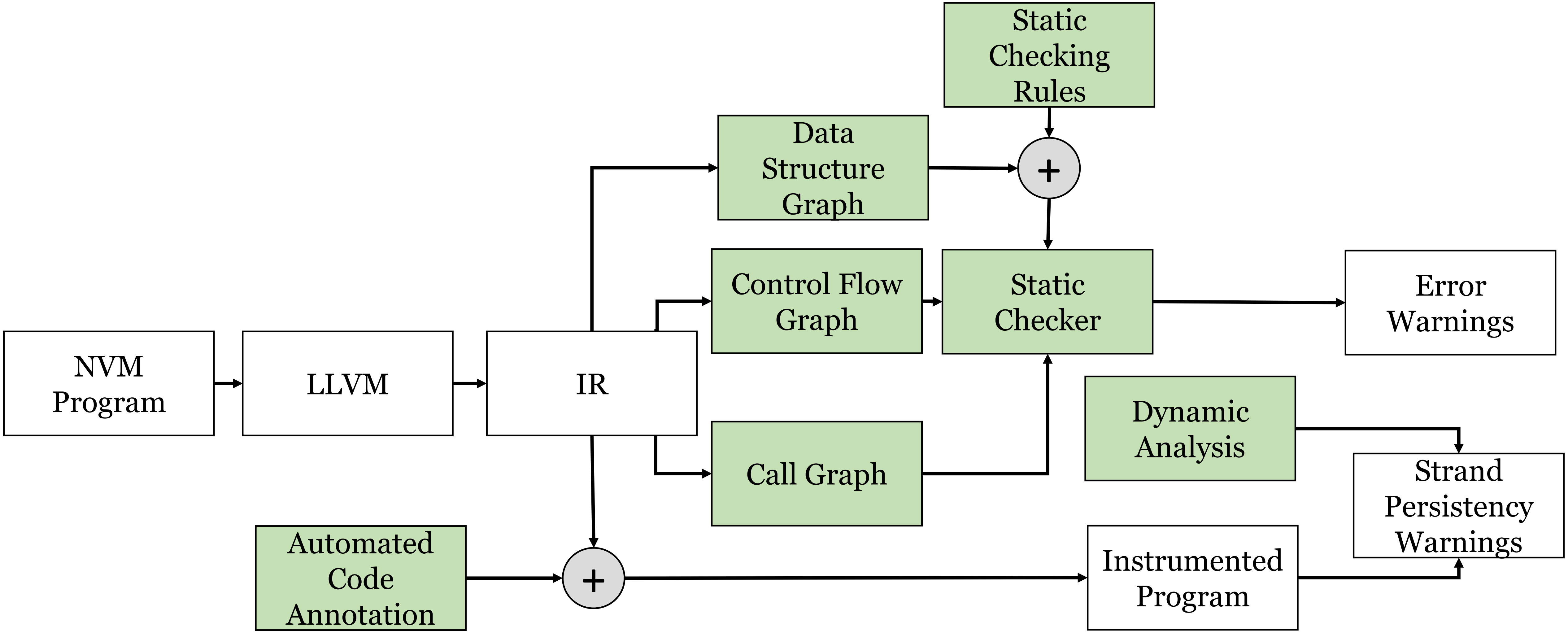


Detecting Memory Persistency Bugs with DeepMC



Dynamic component to catch strand persistency violations

Detecting Memory Persistency Bugs with DeepMC



DeepMC Implementation

DeepMC Implementation

Static Analysis

13k LoC on top of LLVM/Clang

Dynamic Analysis

450 LoC on top of ThreadSanitizer

DeepMC Implementation

Experimental Setup

Static Analysis

13k LoC on top of LLVM/Clang

Dynamic Analysis

450 LoC on top of ThreadSanitizer

Server

8 Intel Xeon(R), 3.3 GHz
16GB Main Memory
Ubuntu 18.04, Linux kernel 5.0
Clang/Clang++ 7.0.0, O3 optimization

Workloads

Memcached, Redis, Nstore
PMDK, PMFS, NVM-Direct, Mnemosyne

New Persistency Bugs

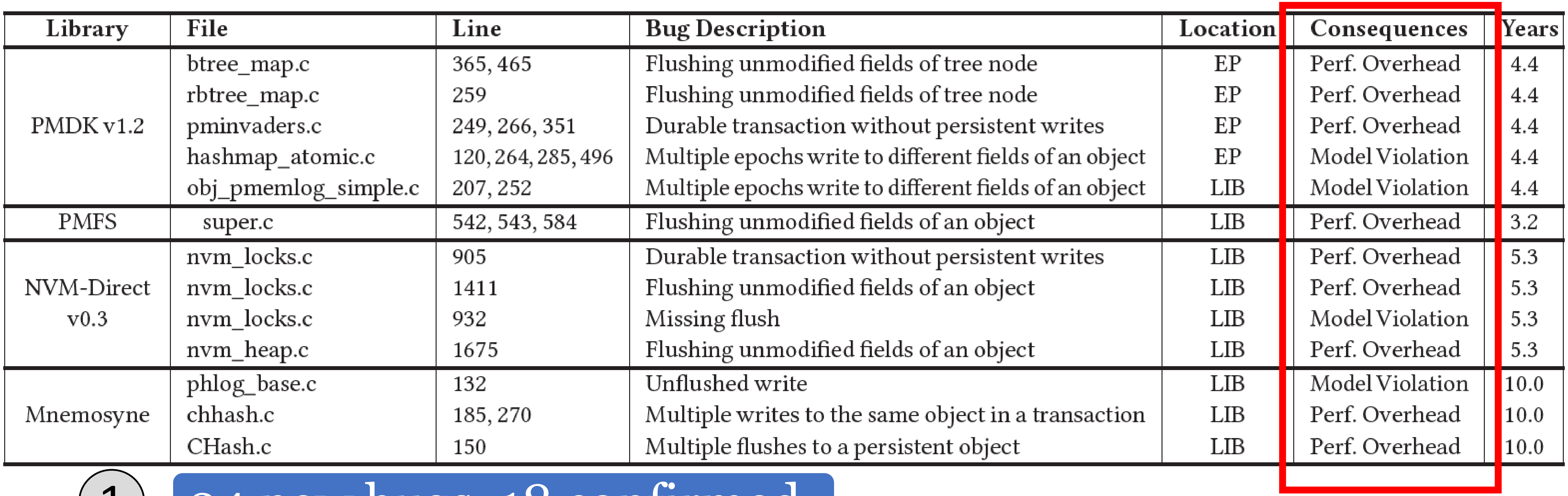
Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

New Persistency Bugs

Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

① 24 new bugs, 18 confirmed

New Persistency Bugs



Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

① 24 new bugs, 18 confirmed

② 8 model violations, 16 performance bugs

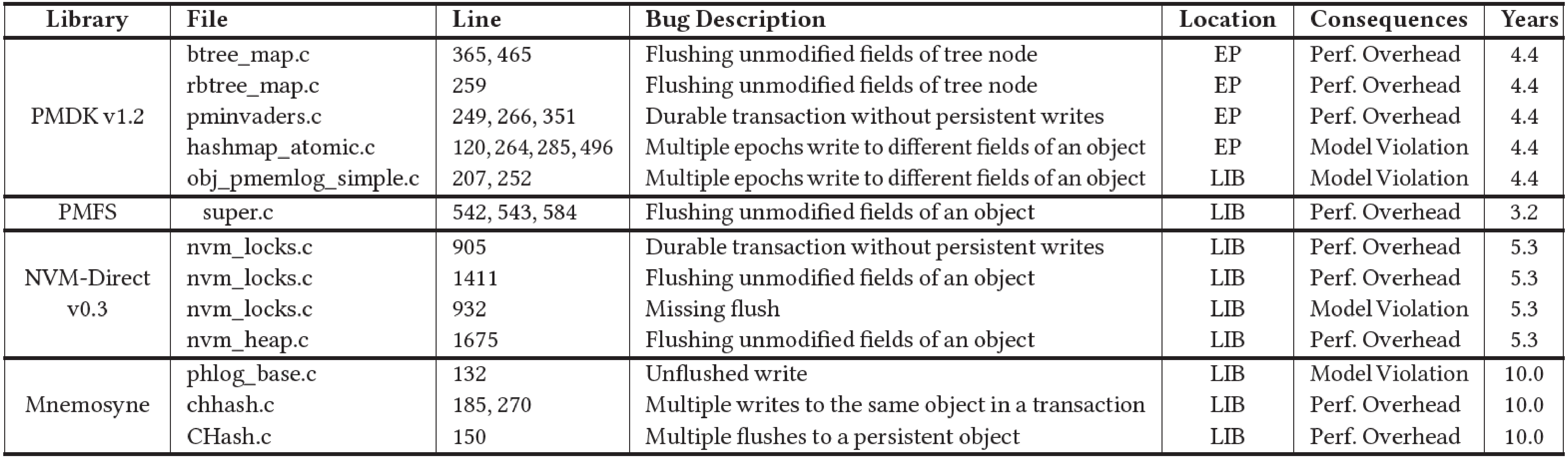
New Persistency Bugs

Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

① 24 new bugs, 18 confirmed

② 8 model violations, 16 performance bugs

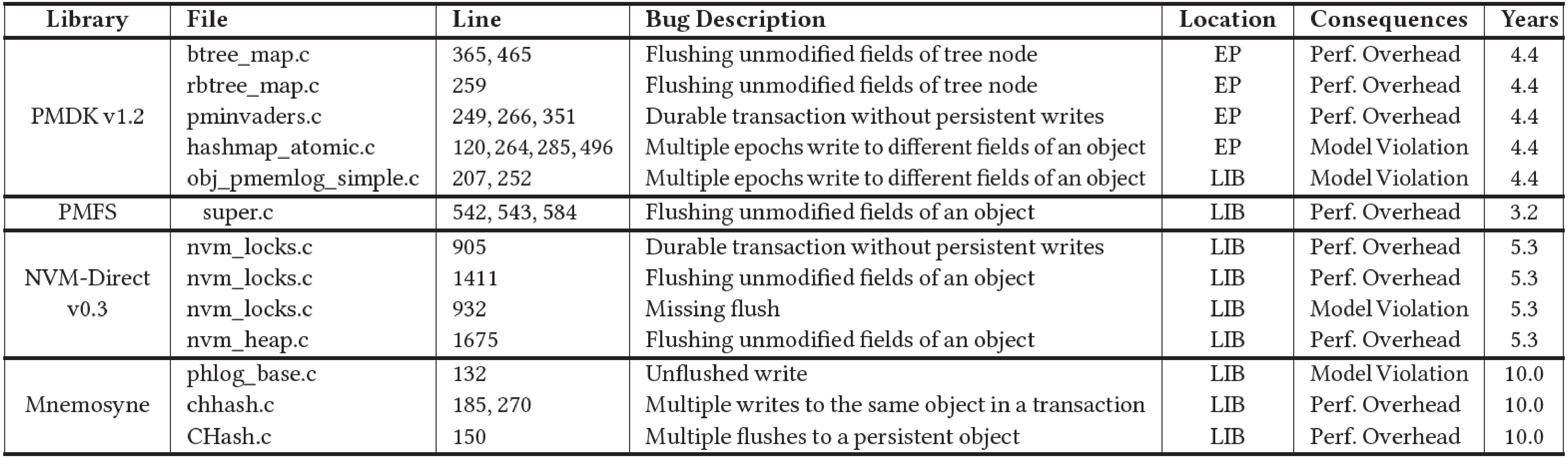
New Persistency Bugs



Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

1. 24 new bugs, 18 confirmed
2. 8 model violations, 16 performance bugs
3. 18 statically detected, 6 dynamically detected

New Persistency Bugs



Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

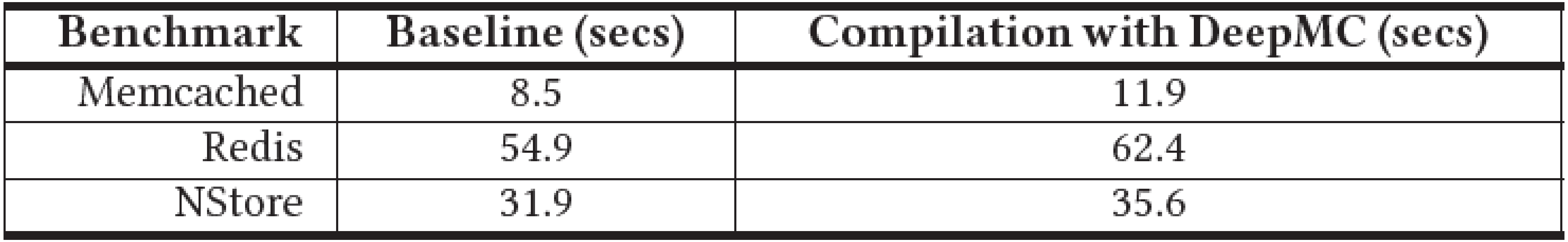
① 24 new bugs, 18 confirmed

② 8 model violations, 16 performance bugs

③ 18 statically detected, 6 dynamically detected

④ Common performance bug was flushing unmodified data!

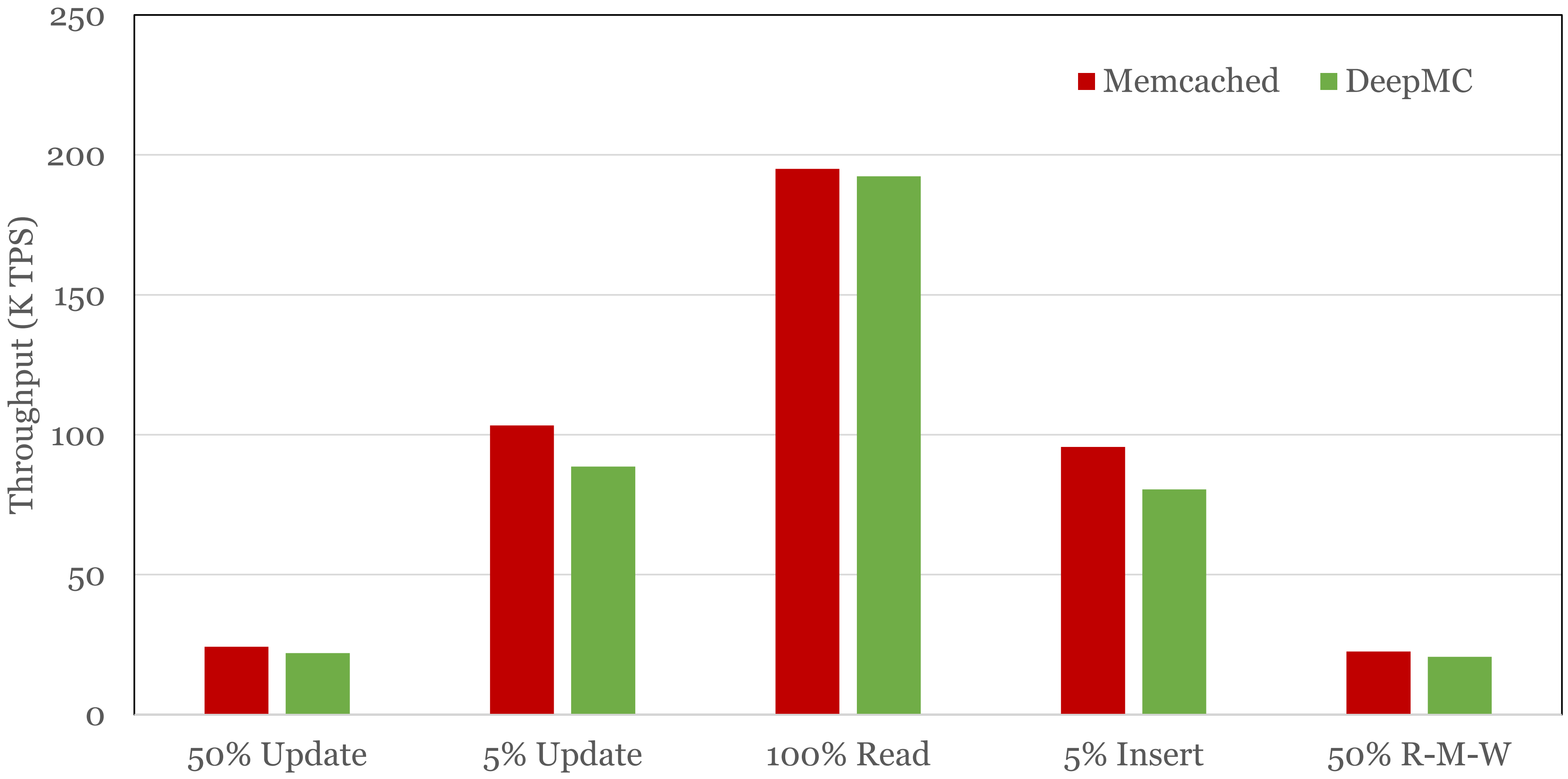
Impact of DeepMC on Performance



Benchmark	Baseline (secs)	Compilation with DeepMC (secs)
Memcached	8.5	11.9
Redis	54.9	62.4
NStore	31.9	35.6

Static analysis introduces minimal compilation overhead

Impact of DeepMC on Performance



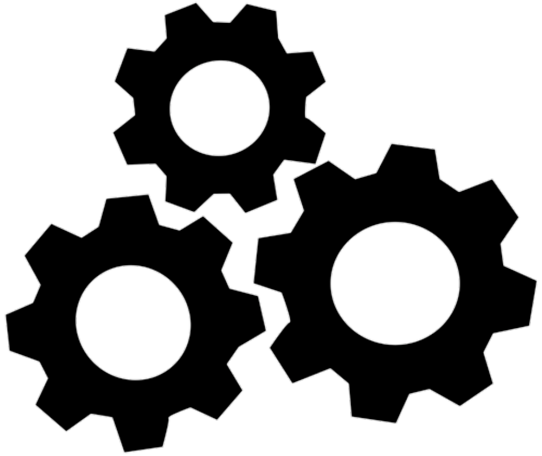
Dynamic analysis adds minimal performance overhead!

Limitations of DeepMC

Lack of dynamic context for DSA

Certain memory references cannot be resolved statically!

Limitations of DeepMC

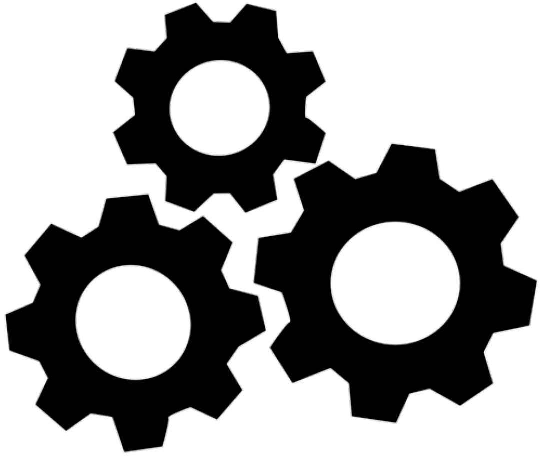


Lack of dynamic context for DSA

Approved violations of the model

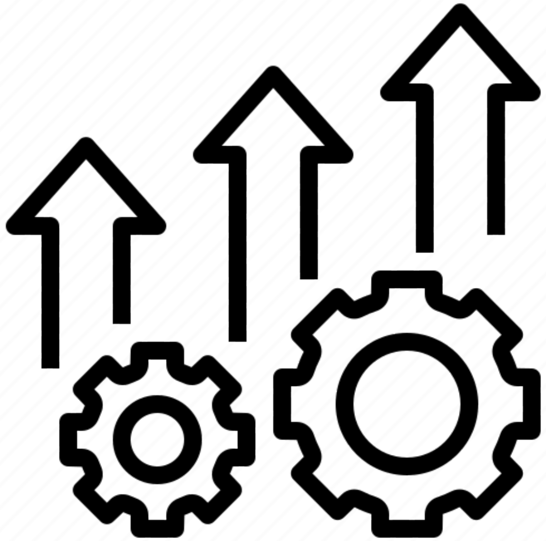
Programmers may violate the model intentionally for performance

Limitations of DeepMC



Lack of dynamic context for DSA

Approved violations of the model



Checking rules can be further enriched

Checking rules can be enriched as models are added and refined

DeepMC Summary

DeepMC Summary

Study Bugs in NVM Programs

DeepMC Summary

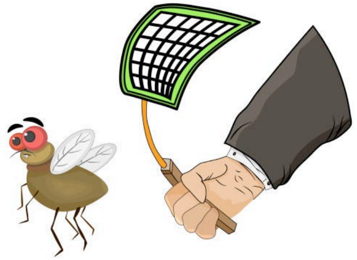
Study Bugs in NVM Programs

Develop Static and Dynamic Detection Tools

DeepMC Summary

Study Bugs in NVM Programs

Develop Static and Dynamic Detection Tools



Discover 24 new persistency bugs in NVM Programs

Thank You!

Benjamin Reidys, Jian Huang
breidys2@illinois.edu

Systems Platform Research Group

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN