An Evaluation of Intel's Restricted Transactional Memory for CPAs

Communicating Process Architectures 2013

Fred Barnes
School of Computing, University of Kent, Canterbury

F.R.M.Barnes@kent.ac.uk
http://www.cs.kent.ac.uk/~frmb/

PLAS
Programming Languages and Systems

University of Kent | Computing

Contents

- Intel's new instructions (TSX).
  - what we get and how to use it.
- Motivation.
- Transactions and transactional memory.

Intel's New Processor Extensions

- Intel's latest processor microarchitecture, **Haswell**, adds **Transactional Synchronization Extensions** (TSX).
  - **Hardware Lock Elision** (HLE).
  - **Restricted Transactional Memory** (RTM).
- **HLE** provides two new **instruction prefixes**.
  - intended for use with existing **exclusive lock** type code.
- **RTM** provides four new **instructions**.
  - a fairly powerful mechanism, but limited to the latest Intel CPUs (and not the 'K' variety yet).

Intel's New Processor Extensions

- Intel's latest processor microarchitecture, **Haswell**, adds **Transactional Synchronization Extensions** (TSX).
  - **Hardware Lock Elision** (HLE).
  - **Restricted Transactional Memory** (RTM).
- **HLE** provides two new **instruction prefixes**.
  - intended for use with existing **exclusive lock** type code.
- **RTM** provides four new **instructions**.
  - a fairly powerful mechanism, but limited to the latest Intel CPUs (and not the 'K' variety yet).

Intel's New Processor Extensions

- Intel's latest processor microarchitecture, **Haswell**, adds **Transactional Synchronization Extensions** (TSX).
  - **Hardware Lock Elision** (HLE).
  - **Restricted Transactional Memory** (RTM).
- **HLE** provides two new **instruction prefixes**.
  - intended for use with existing **exclusive lock** type code.
- **RTM** provides four new **instructions**.
  - a fairly powerful mechanism, but limited to the latest Intel CPUs (and not the 'K' variety yet).

Motivation

- For a long time (prior to Haswell) the amount of memory that could be **atomically** manipulated on **x86** was limited to a single word (32 or 64 bits).
  - the most complex being **compare-and-swap**.
  - other platforms provide things like **load-linked, store-conditional**.
- This has contributed to the development of:
  - entire classes of **non-blocking** **wait-free** and **lock-free** algorithms [1, 2].
  - programs (multi-threaded or interrupt-driven) need to modify **state** in a consistent way — e.g. chunks of linked data structures.
- Perhaps an argument that global linked data structures are **not** the best approach:
  - CPAs would advocate a process that encapsulates this state; other processes interact via channels (issues: contention, interrupts).
  - The ideal fix is possibly an educational one, but as long as people use sequential procedural languages on multicore, we have to live with it.

Motivation

- For a long time (prior to Haswell) the amount of memory that could be **atomically** manipulated on **x86** was limited to a single word (32 or 64 bits).
  - the most complex being **compare-and-swap**.
  - other platforms provide things like **load-linked, store-conditional**.
- This has contributed to the development of:
  - entire classes of **non-blocking wait-free** and **lock-free** algorithms [1, 2].
  - programs (multi-threaded or interrupt-driven) need to modify **state** in a consistent way — e.g. chunks of linked data structures.
- Perhaps an argument that global linked data structures are **not** the best approach:
  - CPAs would advocate a process that encapsulates this state; other processes interact via channels (issues: contention, interrupts).
  - The ideal fix is possibly an educational one, but as long as people use sequential procedural languages on multicore, we have to live with it.

Motivation

- For a long time (prior to Haswell) the amount of memory that could be **atomically** manipulated on **x86** was limited to a single word (32 or 64 bits).
  - the most complex being **compare-and-swap**.
  - other platforms provide things like **load-linked, store-conditional**.
- This has contributed to the development of:
  - entire classes of **non-blocking wait-free** and **lock-free** algorithms [1, 2].
  - programs (multi-threaded or interrupt-driven) need to modify **state** in a consistent way — e.g. chunks of linked data structures.
- Perhaps an argument that global linked data structures are **not** the best approach:
  - CPAs would advocate a process that encapsulates this state; other processes interact via channels (issues: contention, interrupts).
  - The ideal fix is possibly an educational one, but as long as people use sequential procedural languages on multicore, we have to live with it.

Transactions and Transactional Memory

- The concept of a **transaction** has been around for a long time.
  - probably since humans started interacting with each other.
  - but **databases** are where we see them most obviously.
- In the DB context, four principles [3]:
  - **atomicity**: seen to happen as a single thing.
  - **consistency**: preserve system invariants.
  - **isolation**: non-interfering in other transactions.
  - **durability**: be persistent once committed.
- For ourselves (system developers in general) most interested in **atomicity** and **consistency**.

Transactions and Transactional Memory

- The concept of a **transaction** has been around for a long time.
  - probably since humans started interacting with each other.
  - but **databases** are where we see them most obviously.
- In the DB context, four principles [3]:
  - **atomicity**: seen to happen as a single thing.
  - **consistency**: preserve system invariants.
  - **isolation**: non-interfering in other transactions.
  - **durability**: be persistent once committed.
- For ourselves (system developers in general) most interested in **atomicity** and **consistency**.

Transactions and Transactional Memory

- The concept of a **transaction** has been around for a long time.
  - probably since humans started interacting with each other.
  - but **databases** are where we see them most obviously.
- In the DB context, four principles [3]:
  - **atomicity**: seen to happen as a single thing.
  - **consistency**: preserve system invariants.
  - **isolation**: non-interfering in other transactions.
  - **durability**: be persistent once committed.
- For ourselves (system developers in general) most interested in **atomicity** and **consistency**.

Transactions and Transactional Memory

- **Transactional memory** ideas have been around for a while:
  - First described by **Herlihy and Moss** in **1993** [4].
  - Some specialised hardware support appeared: **IBM's BlueGene/Q** and **Sun Rock** processors.
- In the meantime, **software transactional memory** (STM) gained some momentum.
  - providing better programming abstractions to manipulate **shared memory** safely.
  - implementations in Haskell and (perhaps experimental) in Java.
- Issues with STM: performance guarantees..

Transactions and Transactional Memory

- **Transactional memory** ideas have been around for a while:
  - First described by **Herlihy and Moss** in **1993** [4].
  - Some specialised hardware support appeared: **IBM's BlueGene/Q** and **Sun Rock** processors.
- In the meantime, **software transactional memory** (STM) gained some momentum.
  - providing better programming abstractions to manipulate **shared memory** safely.
  - implementations in Haskell and (perhaps experimental) in Java.
- Issues with STM: performance guarantees..

Transactions and Transactional Memory

- **Transactional memory** ideas have been around for a while:
  - First described by **Herlihy and Moss** in **1993** [4].
  - Some specialised hardware support appeared: **IBM's BlueGene/Q** and **Sun Rock** processors.
- In the meantime, **software transactional memory** (STM) gained some momentum.
  - providing better programming abstractions to manipulate **shared memory** safely.
  - implementations in Haskell and (perhaps experimental) in Java.
- Issues with STM: performance guarantees..

Software Transactional Memory

- Illustration: (what the programmer wants to write)
  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Illustration: (what the programmer wants to write)

```
void add_to_list (list_t **lptr, list_t *itm)
{

    if (*lptr) {
        (*lptr)->prev = itm;
        itm->next = *lptr;
    }
    *lptr = itm;

}
```

  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Illustration: (what the programmer wants to write)

```
void add_to_list (list_t **lptr, list_t *itm)
{

    if (*lptr) {
        (*lptr)->prev = itm;
        itm->next = *lptr;
    }
    *lptr = itm;

}
```

  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Illustration: (what the programmer wants to write)

```
lock_t *list_lock = create_lock();

void add_to_list (list_t **lptr, list_t *itm)
{
  claim_lock (list_lock);
    if (*lptr) {
        (*lptr)->prev = itm;
        itm->next = *lptr;
    }
    *lptr = itm;
  release_lock (list_lock);
}
```

  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Illustration: (what the programmer wants to write)

```
lock_t *list_lock = create_lock();

void add_to_list (list_t **lptr, list_t *itm)
{
  claim_lock (list_lock);
    if (*lptr) {
        (*lptr)->prev = itm;
        itm->next = *lptr;
    }
    *lptr = itm;
  release_lock (list_lock);
}
```

  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Illustration: (what the programmer wants to write)

```
void add_to_list (list_t **lptr, list_t *itm)
{
  atomic {
    if (*lptr) {
        (*lptr)->prev = itm;
        itm->next = *lptr;
    }
    *lptr = itm;
  }
}
```

  - breaks horribly in an **unsafe** threaded environment.
  - solutions: add a lock (heavy or light).
- What we **really** want to say is: *do this atomically*.
  - which is what STM provides (in theory).

Software Transactional Memory

- Nice idea in theory, but **does not scale** well.
  - in Moss' "open nested transactions" (Java), some amount of effort to avoid infinite retry.
  - not a totally correct solution either (but close!).
- Can imagine how it works:
  1. make a record of any **shared memory** state that is read inside the '`atomic`' block.
  2. execute the transaction, putting writes in a buffer (not changing the global shared state).
  3. at the end of the transaction, check all the things we read in step 1; if the same (modulo changes in step 2) **commit** the changes en-masse (locking required), else **redo from start**.
- This sort of strategy cannot see $A \rightarrow B \rightarrow A$ changes.
  - could fix, but heading towards something which is significantly more expensive and inconvenient than the locks we were trying to avoid in the first place!

Software Transactional Memory

- Nice idea in theory, but **does not scale** well.
  - in Moss' "open nested transactions" (Java), some amount of effort to avoid infinite retry.
  - not a totally correct solution either (but close!).
- Can imagine how it works:
  1. make a record of any **shared memory** state that is read inside the '`atomic`' block.
  2. execute the transaction, putting writes in a buffer (not changing the global shared state).
  3. at the end of the transaction, check all the things we read in step 1; if the same (modulo changes in step 2) **commit** the changes en-masse (locking required), else **redo from start**.
- This sort of strategy cannot see $A \rightarrow B \rightarrow A$ changes.
  - could fix, but heading towards something which is significantly more expensive and inconvenient than the locks we were trying to avoid in the first place!

Software Transactional Memory

- Nice idea in theory, but **does not scale** well.
  - in Moss' "open nested transactions" (Java), some amount of effort to avoid infinite retry.
  - not a totally correct solution either (but close!).
- Can imagine how it works:
  1. make a record of any **shared memory** state that is read inside the '`atomic`' block.
  2. execute the transaction, putting writes in a buffer (not changing the global shared state).
  3. at the end of the transaction, check all the things we read in step 1; if the same (modulo changes in step 2) **commit** the changes en-masse (locking required), else **redo from start**.
- This sort of strategy cannot see $A \to B \to A$ changes.
  - could fix, but heading towards something which is significantly more expensive and inconvenient than the locks we were trying to avoid in the first place!

Technical Detail

- The RTM extensions provide four new instructions:
  - `XBEGIN`: initiates a transaction.
    - pointer to a **fallback handler** is given.
    - also valid with a transaction, permitting **nesting**.
  - `XEND`: completes a transaction, flushing changes to memory (if top).
  - `XABORT`: aborts the current transaction (with failure code for fallback handler).
  - `XTEST`: updates status register to allow conditional branch to test for "in transaction".

Restrictions

- Because this is **restricted** transaction memory, some limitations:
  - X87 FP and MMX instructions not supported, but SSE and AVX are (so not a huge issue, depending on generated code).
  - instructions that halt the processor (e.g. `PAUSE`, `WAIT`) not supported.
  - debugging instructions not supported (no breakpoints inside transactions).
  - an interrupt within a transaction will cause the transaction to abort, before the interrupt handler is run.
  - changes in privilege level (no kernel calls).
  - any exception (largely page-fault): all memory accessed during a transaction must be mapped.
- A lot of ways that a transaction can be aborted (including the obvious — another core accessing the memory).
  - nevertheless, a significant feature!

Restrictions

- Because this is **restricted** transaction memory, some limitations:
  - X87 FP and MMX instructions not supported, but SSE and AVX are (so not a huge issue, depending on generated code).
  - instructions that halt the processor (e.g. `PAUSE`, `WAIT`) not supported.
  - debugging instructions not supported (no breakpoints inside transactions).
  - an interrupt within a transaction will cause the transaction to abort, before the interrupt handler is run.
  - changes in privilege level (no kernel calls).
  - any exception (largely page-fault): all memory accessed during a transaction must be mapped.
- A lot of ways that a transaction can be aborted (including the obvious — another core accessing the memory).
  - nevertheless, a significant feature!

Nesting

- Transactions can be nested, but not in a clever way.
  - processor maintains a **transaction count**; '`XBEGIN`' increments, '`XEND`' decrements.
  - transaction only commited to memory when last '`XEND`' happens.
  - any conflict/etc. causes the outermost failure handler to be invoked.

Transaction Failures

- If a transaction is aborted (as defined earlier) all changes to the processor's state made within the transaction are discarded.
  - includes sources of **exceptions**: i.e. the exception handler is not invoked.
  - does not include **interrupts**, which are handled after the state has been discarded.
- Means we need to be *slightly* careful.
  - do not want a continuous cycle of try-read, page-fault, abort, try-read, page-fault, abort, ... (though unlikely)

Transaction Failures

- If a transaction is aborted (as defined earlier) all changes to the processor's state made within the transaction are discarded.
  - includes sources of **exceptions**: i.e. the exception handler is not invoked.
  - does not include **interrupts**, which are handled after the state has been discarded.
- Means we need to be *slightly* careful.
  - do not want a continuous cycle of try-read, page-fault, abort, try-read, page-fault, abort, ... (though unlikely)

Transaction Failures

- When aborted, the fallback (failure) handler is invoked, with EAX containing some flags to indicate what happened.
- As of July 2013, these were:
  - 0 (xabort): an `XABORT` instruction aborted the transaction, 8-bit code passed is available in EAX.
  - 1 (retry): the transaction might succeed if retried.
  - 2 (conflict): interference from another processor, core or hardware thread caused the abort.
  - 3 (overflow): overflow of buffers caused the abort.
  - 4 (debug): a debug breakpoint was encountered.
  - 5 (nested): transaction aborted within a nested transaction.

Transaction Failures

- When aborted, the fallback (failure) handler is invoked, with EAX containing some flags to indicate what happened.
- As of July 2013, these were:
  - 0 (xabort): an `XABORT` instruction aborted the transaction, 8-bit code passed is available in EAX.
  - 1 (retry): the transaction might succeed if retried.
  - 2 (conflict): interference from another processor, core or hardware thread caused the abort.
  - 3 (overflow): overflow of buffers caused the abort.
  - 4 (debug): a debug breakpoint was encountered.
  - 5 (nested): transaction aborted within a nested transaction.

Test Setup

- Run on a **Core i7-4770** processor at **3.4 GHz**.
  - **16 GiB** RAM at **1600 MHz** (9-9-9-24).
  - Ubuntu Linux 12.04.2 LTS with kernel 3.5.0-23-generic and stock GCC 4.6.3.
  - CPU **frequency scaling** and "turbo boost" disabled.
- RTM extensions accessed through inline assembler macros.
- First attempts to buy the new Haswell CPU failed: the 'K' (overclockable) version does not support RTM!

Test Setup

- Run on a **Core i7-4770** processor at **3.4 GHz**.
  - **16 GiB** RAM at **1600 MHz** (9-9-9-24).
  - Ubuntu Linux 12.04.2 LTS with kernel 3.5.0-23-generic and stock GCC 4.6.3.
  - CPU **frequency scaling** and "turbo boost" disabled.
- RTM extensions accessed through inline assembler macros.
- First attempts to buy the new Haswell CPU failed: the 'K' (overclockable) version does not support RTM!

Test Setup

- Run on a **Core i7-4770** processor at **3.4 GHz**.
  - **16 GiB** RAM at **1600 MHz** (9-9-9-24).
  - Ubuntu Linux 12.04.2 LTS with kernel 3.5.0-23-generic and stock GCC 4.6.3.
  - CPU **frequency scaling** and "turbo boost" disabled.
- RTM extensions accessed through inline assembler macros.
- First attempts to buy the new Haswell CPU failed: the 'K' (overclockable) version does not support RTM!

Test Operations

For testing, we define **four** different operations:

- **read**: load words from increasing memory locations.
- **write**: store words to increasing memory locations.
- **cas**: compare-and-swap words at increasing memory locations.
- **abortm**: store words to increasing memory locations, aborting the transaction after *m* words, and:
  abortn: setup to do *n* writes, but abort before doing any.
- Each of the above (minus abort) can operate in *untransaction* (**u**) or *transactional* (**x**) mode; word size either 32-bit or 64-bit.
- For each combination, test increasing numbers of words-per-operation (**operation size**).
- Tests done in a well-aligned memory region of 512 MiB, to minimise effect of L2 and L3 caches.
  - e.g. 16384 operations done when operation-size is 32 KiB.

Test Operations

For testing, we define **four** different operations:

- **read**: load words from increasing memory locations.
- **write**: store words to increasing memory locations.
- **cas**: compare-and-swap words at increasing memory locations.
- **abortm**: store words to increasing memory locations, aborting the transaction after *m* words, and:
  abortn: setup to do *n* writes, but abort before doing any.
- Each of the above (minus abort) can operate in *untransaction* (**u**) or *transactional* (**x**) mode; word size either 32-bit or 64-bit.
- For each combination, test increasing numbers of words-per-operation (**operation size**).
- Tests done in a well-aligned memory region of 512 MiB, to minimise effect of L2 and L3 caches.
  - e.g. 16384 operations done when operation-size is 32 KiB.

Test Operations

For testing, we define **four** different operations:

- **read**: load words from increasing memory locations.
- **write**: store words to increasing memory locations.
- **cas**: compare-and-swap words at increasing memory locations.
- **abortm**: store words to increasing memory locations, aborting the transaction after *m* words, and:
  abortn: setup to do *n* writes, but abort before doing any.
- Each of the above (minus abort) can operate in *untransaction* (**u**) or *transactional* (**x**) mode; word size either 32-bit or 64-bit.
- For each combination, test increasing numbers of words-per-operation (**operation size**).
- Tests done in a well-aligned memory region of 512 MiB, to minimise effect of L2 and L3 caches.
  - e.g. 16384 operations done when operation-size is 32 KiB.

Test Operations

For testing, we define **four** different operations:

- **read**: load words from increasing memory locations.
- **write**: store words to increasing memory locations.
- **cas**: compare-and-swap words at increasing memory locations.
- **abortm**: store words to increasing memory locations, aborting the transaction after *m* words, and:
  abortn: setup to do *n* writes, but abort before doing any.
- Each of the above (minus abort) can operate in *untransaction* (**u**) or *transactional* (**x**) mode; word size either 32-bit or 64-bit.
- For each combination, test increasing numbers of words-per-operation (**operation size**).
- Tests done in a well-aligned memory region of 512 MiB, to minimise effect of L2 and L3 caches.
  - e.g. 16384 operations done when operation-size is 32 KiB.

Zero Size Operation Cost

Operation	32-bit time	64-bit time	32-bit cycles	64-bit cycles
u_read	2.0ns	2.4ns	7	8
u_write	2.1ns	2.1ns	7	7
u_cas	2.0ns	2.0ns	7	7
x_read	15ns	15ns	50	50
x_write	15ns	15ns	50	50
x_cas	15ns	15ns	49	49
x_abortn	47ns	47ns	160	161
x_abortm	47ns	47ns	161	161

- Cost of invoking transaction mode appears to be **13ns** (47 cycles).
- Aborting is expensive: likely pipeline and cache flush.
  - infer: transactional operations are pipelined.

Zero Size Operation Cost

Operation	32-bit time	64-bit time	32-bit cycles	64-bit cycles
u_read	2.0ns	2.4ns	7	8
u_write	2.1ns	2.1ns	7	7
u_cas	2.0ns	2.0ns	7	7
x_read	15ns	15ns	50	50
x_write	15ns	15ns	50	50
x_cas	15ns	15ns	49	49
x_abortn	47ns	47ns	160	161
x_abortm	47ns	47ns	161	161

- Cost of invoking transaction mode appears to be **13ns** (47 cycles).
- Aborting is expensive: likely pipeline and cache flush.
  - infer: transactional operations are pipelined.

Cost of Small-Size Operations

Clock-cycle times for small numbers of operations (32-bit):

Operation	1 word	2	3	4	5	6	7	8
u_read	23	14	13	14	15	16	19	20
u_write	29	26	25	22	22	22	22	23
u_cas	39	59	74	93	113	132	151	170
x_read	60	61	62	62	65	66	67	69
x_write	62	59	59	59	59	59	60	60
x_cas	60	64	66	71	73	78	77	80
x_abortn	66	161	160	161	161	161	162	162
x_abortm	170	175	173	175	179	182	182	181

- Once a transaction reaches **2 words**, equivalent cost to a compare-and-swap.
  - beyond this, transactional operations more efficient than CAS (with the added benefit of overall atomicity).

Cost of Small-Size Operations

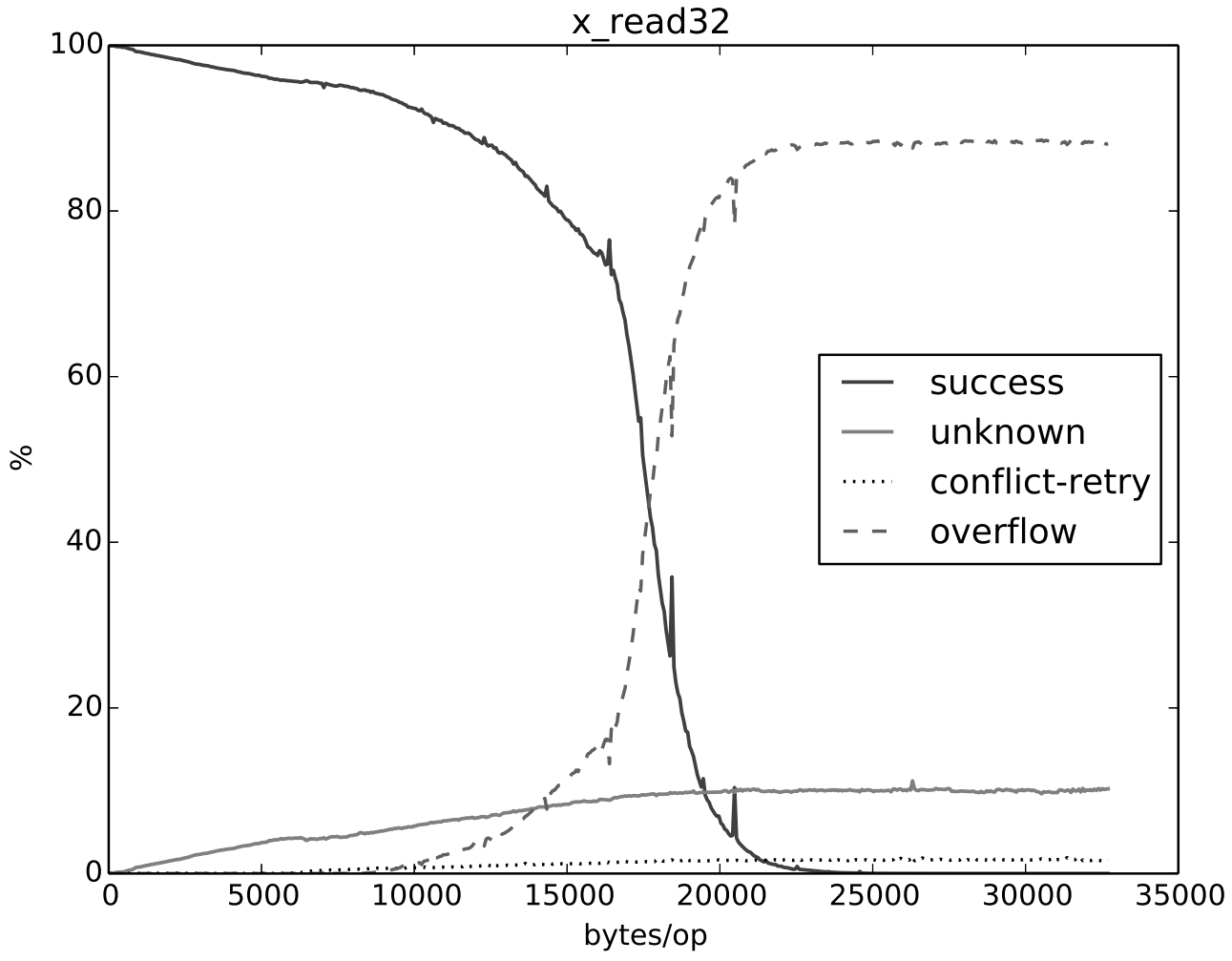
Clock-cycle times for small numbers of operations (32-bit):

Operation	1 word	2	3	4	5	6	7	8
u_read	23	14	13	14	15	16	19	20
u_write	29	26	25	22	22	22	22	23
u_cas	39	59	74	93	113	132	151	170
x_read	60	61	62	62	65	66	67	69
x_write	62	59	59	59	59	59	60	60
x_cas	60	64	66	71	73	78	77	80
x_abortn	66	161	160	161	161	161	162	162
x_abortm	170	175	173	175	179	182	182	181

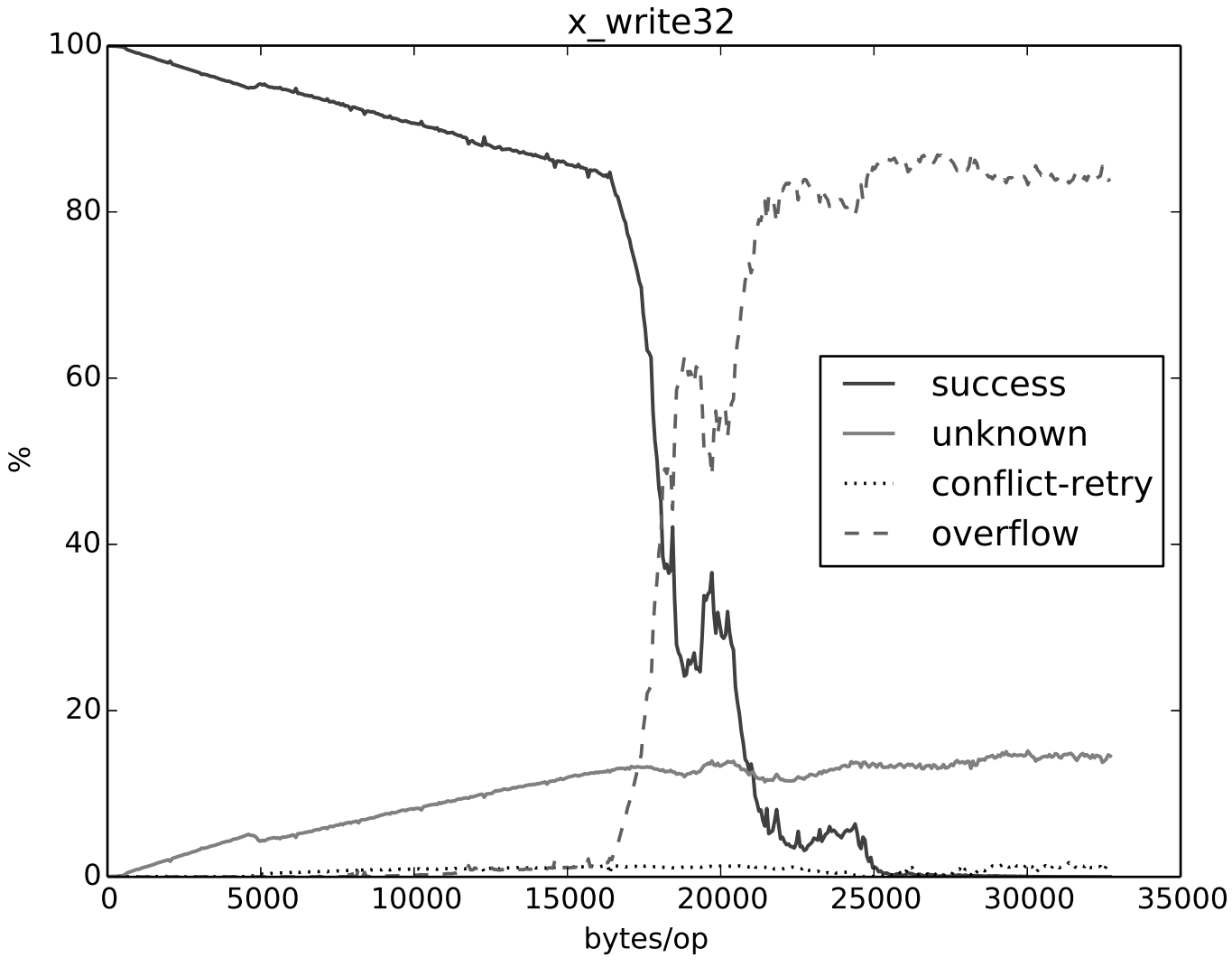
- Once a transaction reaches **2 words**, equivalent cost to a compare-and-swap.
  - beyond this, transactional operations more efficient than CAS (with the added benefit of overall atomicity).

Uncontended Transactions (32-bit read)

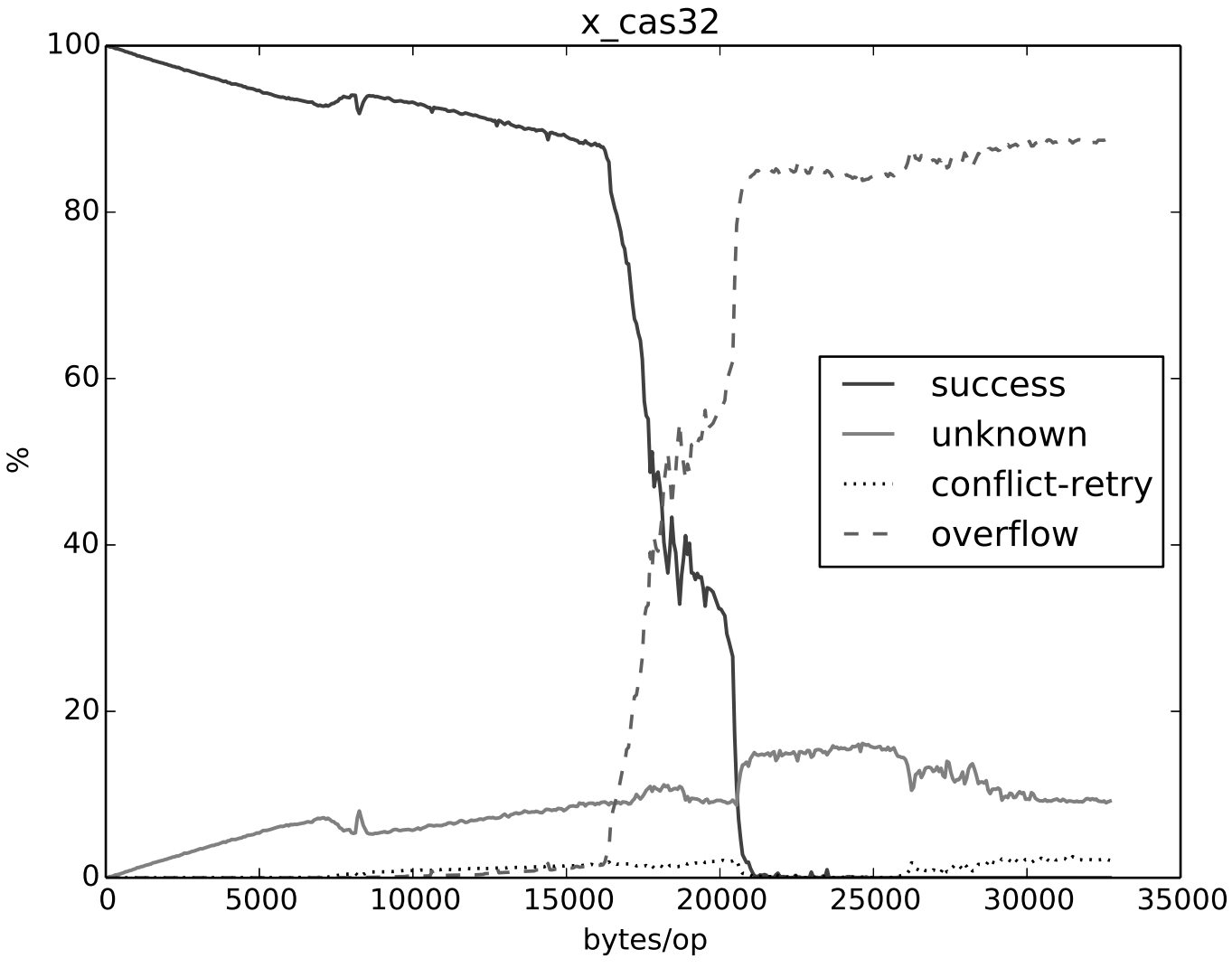
- To determine the maximum practical size of a transaction:



Uncontended Transactions (32-bit write)



Uncontended Transactions (32-bit CAS)



Uncontended Transactions: Observations

- Up to around 10 KiB, performance degrades from **success** to **unknown failure**.
  - likely due to OS context switching.
- From 16 KiB, pronounced **phase shift** where success rapidly gives way to **overflow** related aborts.
- Despite no shared memory contention, **conflict-retry** accounts for some of the failures.
  - either mis-reporting by the processor or caused by operations on another core (e.g. page-table manipulations).
- Largest **stable** transaction size is 16 KiB, with an 85% chance of success.
  - on our particular test setup and with no contention.
  - transaction buffer is probably the L1 data cache, also used to shadow modified registers.

Uncontended Transactions: Observations

- Up to around 10 KiB, performance degrades from **success** to **unknown failure**.
  - likely due to OS context switching.
- From 16 KiB, pronounced **phase shift** where success rapidly gives way to **overflow** related aborts.
- Despite no shared memory contention, **conflict-retry** accounts for some of the failures.
  - either mis-reporting by the processor or caused by operations on another core (e.g. page-table manipulations).
- Largest **stable** transaction size is 16 KiB, with an 85% chance of success.
  - on our particular test setup and with no contention.
  - transaction buffer is probably the L1 data cache, also used to shadow modified registers.

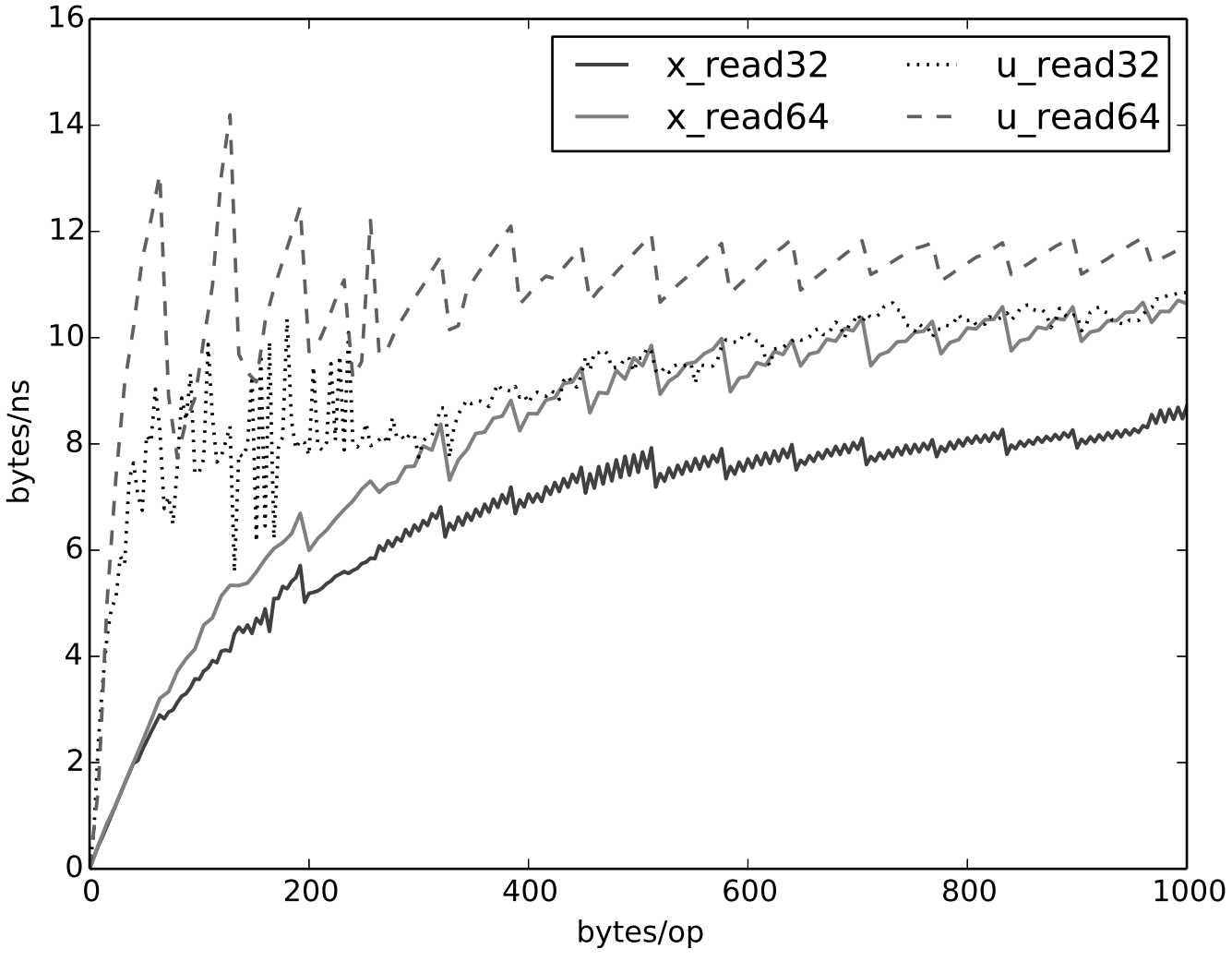
Uncontended Transactions: Observations

- Up to around 10 KiB, performance degrades from **success** to **unknown failure**.
  - likely due to OS context switching.
- From 16 KiB, pronounced **phase shift** where success rapidly gives way to **overflow** related aborts.
- Despite no shared memory contention, **conflict-retry** accounts for some of the failures.
  - either mis-reporting by the processor or caused by operations on another core (e.g. page-table manipulations).
- Largest **stable** transaction size is 16 KiB, with an 85% chance of success.
  - on our particular test setup and with no contention.
  - transaction buffer is probably the L1 data cache, also used to shadow modified registers.

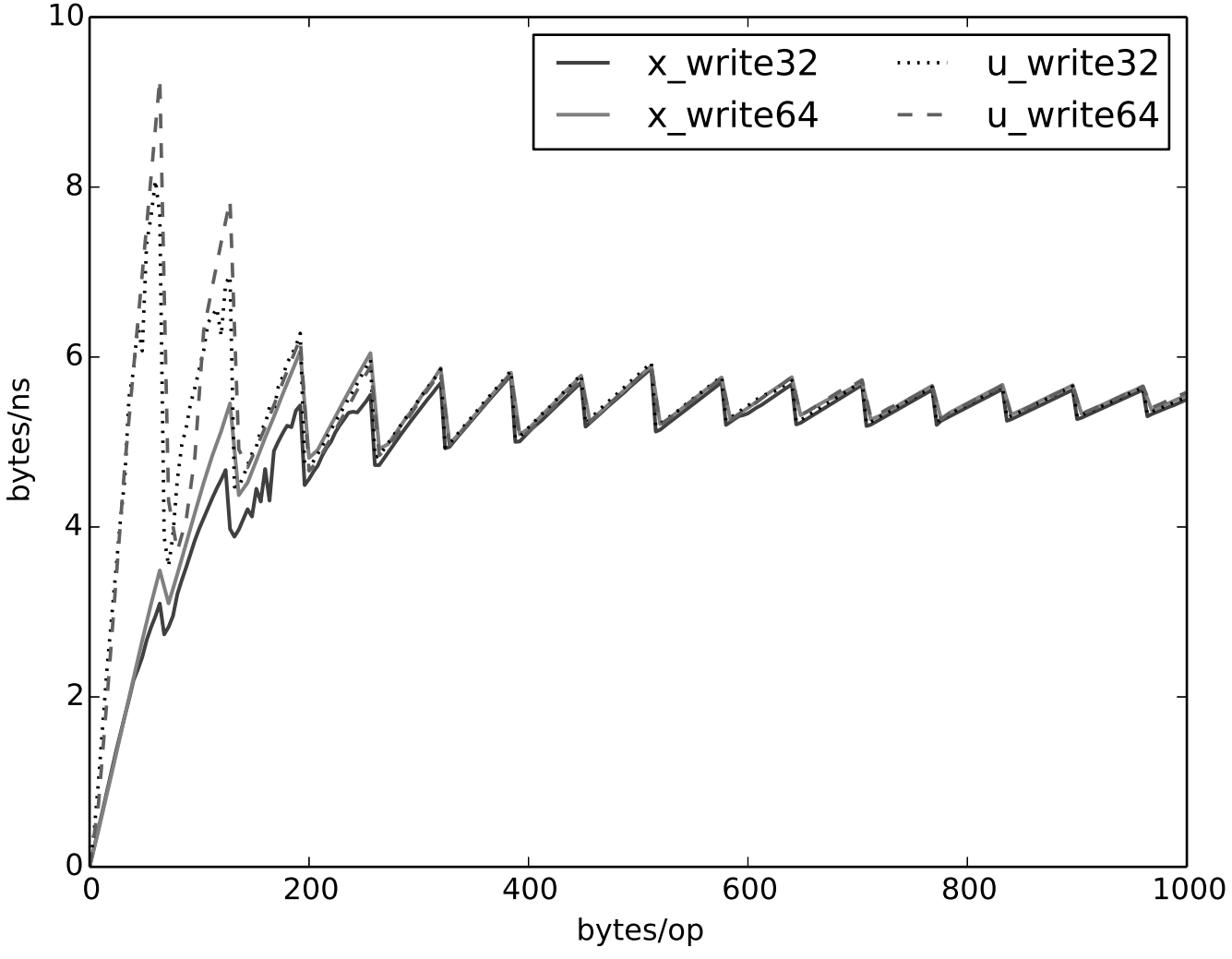
Uncontended Transactions: Observations

- Up to around 10 KiB, performance degrades from **success** to **unknown failure**.
  - likely due to OS context switching.
- From 16 KiB, pronounced **phase shift** where success rapidly gives way to **overflow** related aborts.
- Despite no shared memory contention, **conflict-retry** accounts for some of the failures.
  - either mis-reporting by the processor or caused by operations on another core (e.g. page-table manipulations).
- Largest **stable** transaction size is 16 KiB, with an 85% chance of success.
  - on our particular test setup and with no contention.
  - transaction buffer is probably the L1 data cache, also used to shadow modified registers.

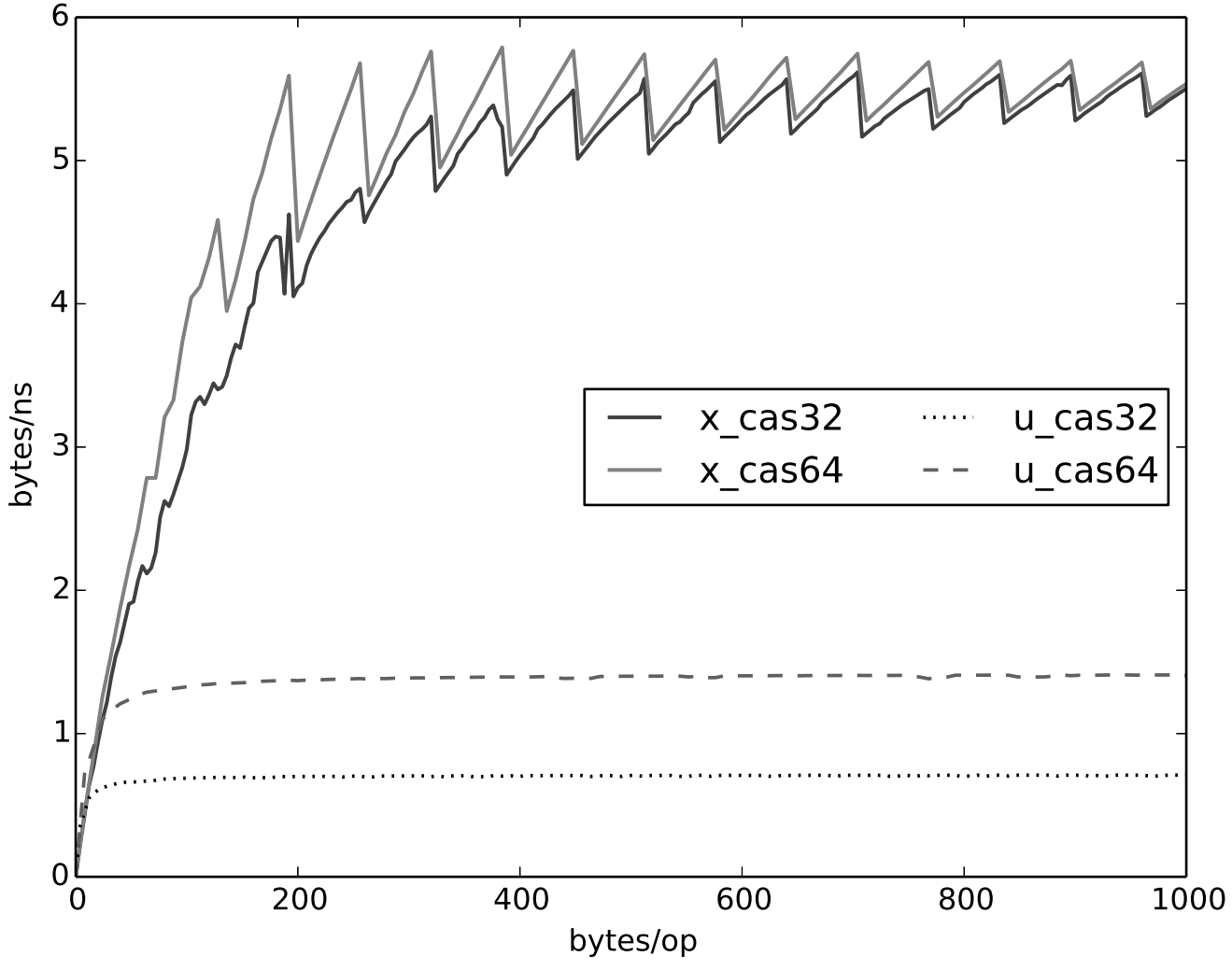
Transaction Performance (reading)



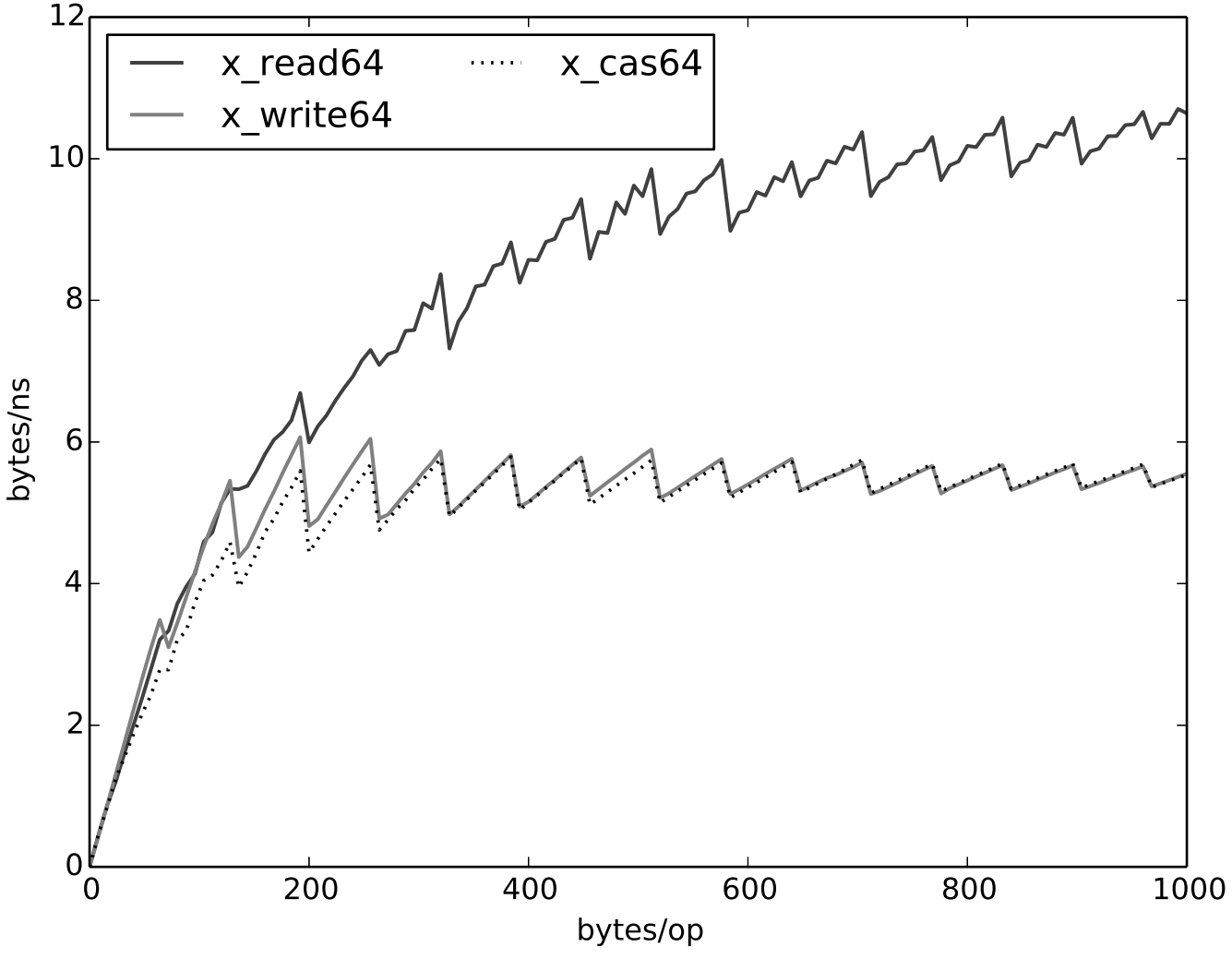
Transaction Performance (writing)



Transaction Performance (CASing)



Transaction Performance (comparison)



Transaction Performance: Observations

- The sawtooth pattern observed is at 64-byte intervals: **cache line**.
  - not unexpected, since the cost of 68-byte read is the same as a 128-byte read.
- Transactional reads reach 80% performance of plain reads.
  - suggests a fixed overhead for transactional reads.
  - not the case for **writes**, where the transaction cost is amortized early on (300 bytes).
- CAS is the most interesting:
  - use of the '`LOCK`' instruction prefix (as our non-transactional CAS does) has a significant overhead, compared with CAS in transactional mode.

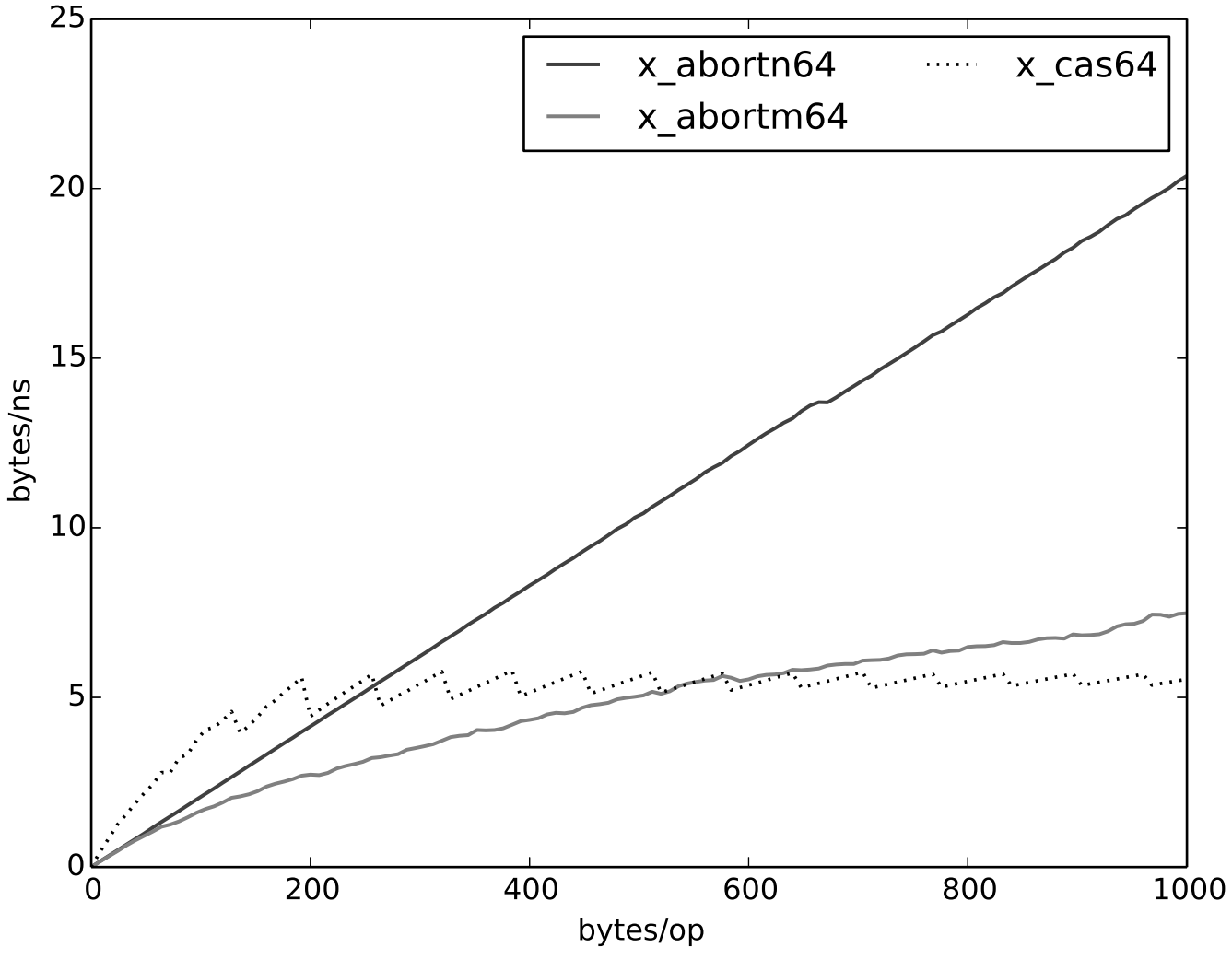
Transaction Performance: Observations

- The sawtooth pattern observed is at 64-byte intervals: **cache line**.
  - not unexpected, since the cost of 68-byte read is the same as a 128-byte read.
- Transactional reads reach 80% performance of plain reads.
  - suggests a fixed overhead for transactional reads.
  - not the case for **writes**, where the transaction cost is amortized early on (300 bytes).
- CAS is the most interesting:
  - use of the '`LOCK`' instruction prefix (as our non-transactional CAS does) has a significant overhead, compared with CAS in transactional mode.

Transaction Performance: Observations

- The sawtooth pattern observed is at 64-byte intervals: **cache line**.
  - not unexpected, since the cost of 68-byte read is the same as a 128-byte read.
- Transactional reads reach 80% performance of plain reads.
  - suggests a fixed overhead for transactional reads.
  - not the case for **writes**, where the transaction cost is amortized early on (300 bytes).
- CAS is the most interesting:
  - use of the '`LOCK`' instruction prefix (as our non-transactional CAS does) has a significant overhead, compared with CAS in transactional mode.

Transaction Aborting: Performance



Transaction Aborting: Observations

- **abortn**, which aborts before doing any writes, has the expected linear performance.
- **abortm**, which writes before aborting, is expensive.
  - surpasses the cost of transactional CAS at around 600 bytes.
- For small transactions, successful completion is significantly cheaper than unsuccessful completion.
  - likely a result of restoring **register state** after unsuccessful transactions, a cost not incurred by successful transactions.

Transaction Aborting: Observations

- **abortn**, which aborts before doing any writes, has the expected linear performance.
- **abortm**, which writes before aborting, is expensive.
  - surpasses the cost of transactional CAS at around 600 bytes.
- For small transactions, successful completion is significantly cheaper than unsuccessful completion.
  - likely a result of restoring **register state** after unsuccessful transactions, a cost not incurred by successful transactions.

Transaction Aborting: Observations

- **abortn**, which aborts before doing any writes, has the expected linear performance.
- **abortm**, which writes before aborting, is expensive.
  - surpasses the cost of transactional CAS at around 600 bytes.
- For small transactions, successful completion is significantly cheaper than unsuccessful completion.
  - likely a result of restoring **register state** after unsuccessful transactions, a cost not incurred by successful transactions.

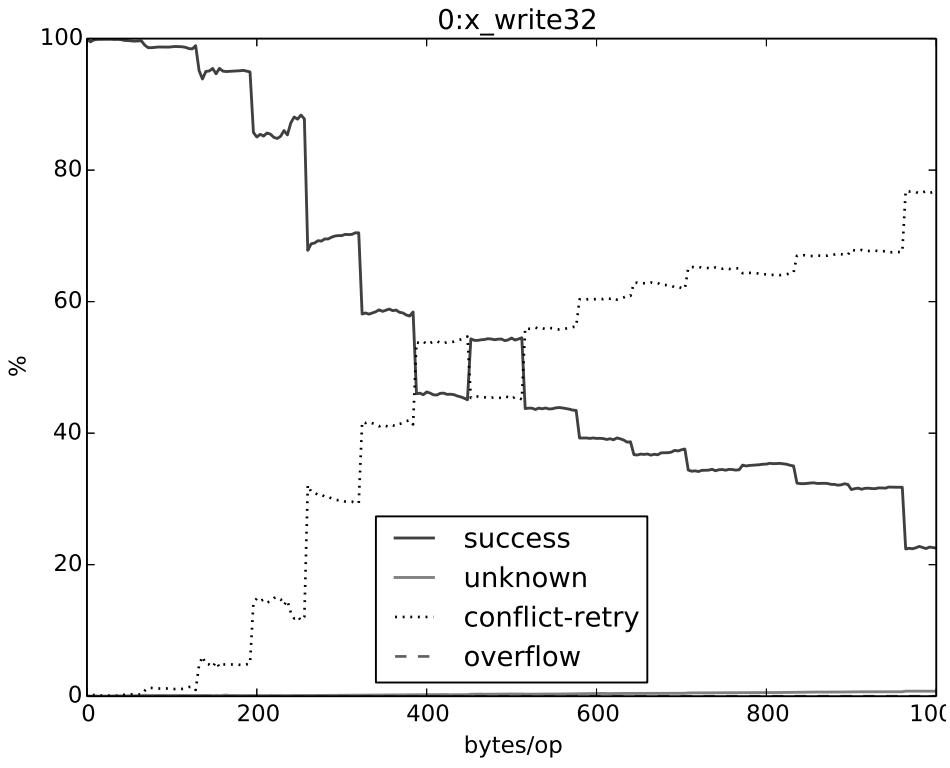
Transaction Contention

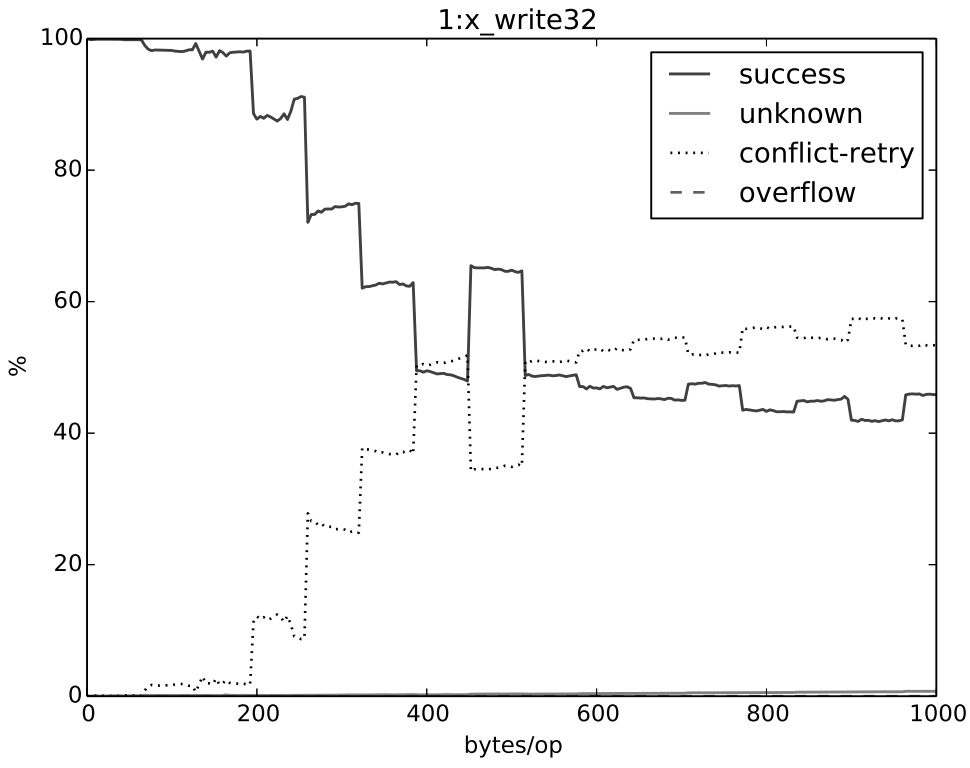
- As expected, contended reads (against reads) do not cause transaction aborts — good!
- In the next few slides, show the effect of **multiple threads** interacting via shared memory, with and without transactions.
  - more in the paper!

Transaction Contention

- As expected, contended reads (against reads) do not cause transaction aborts — good!
- In the next few slides, show the effect of **multiple threads** interacting via shared memory, with and without transactions.
  - more in the paper!

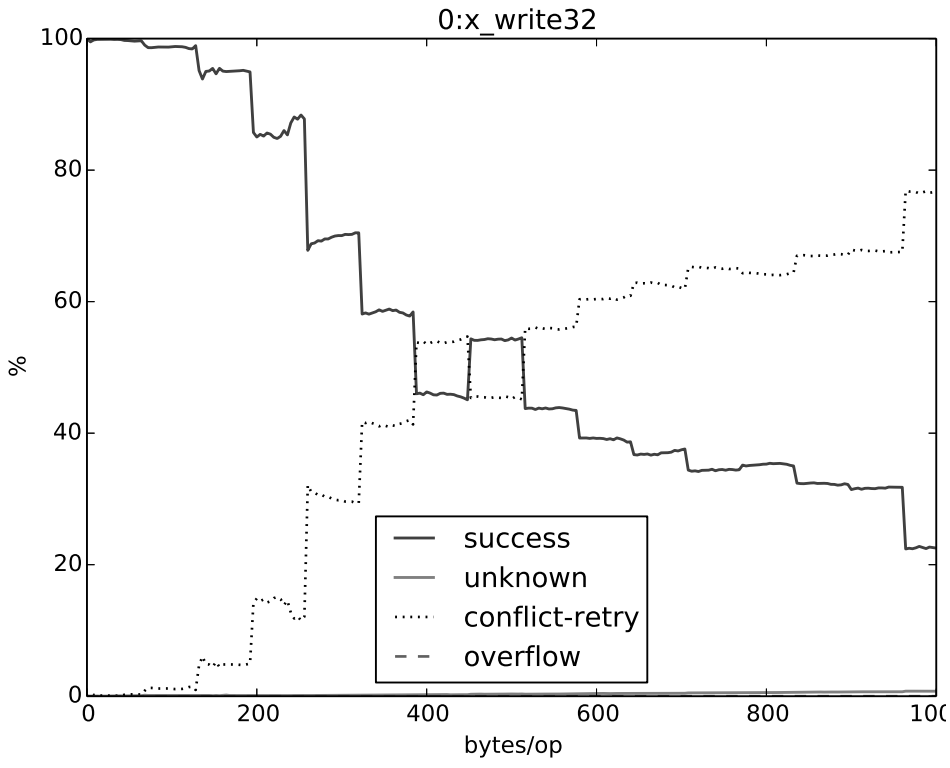
Transaction Contention: Competing Writes

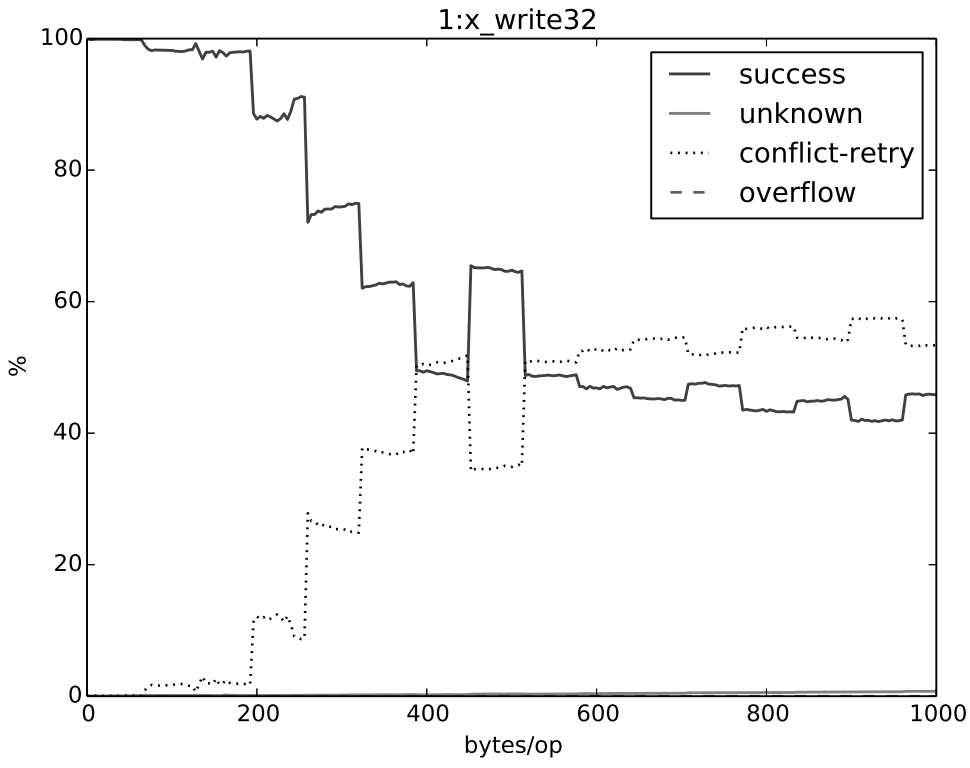




- No conflicts for **small** sizes (scheduling), but beyond 192 bytes (4 cache lines), performance degrades rapidly.
  - slight bias towards thread 1, possibly scheduling artefact.

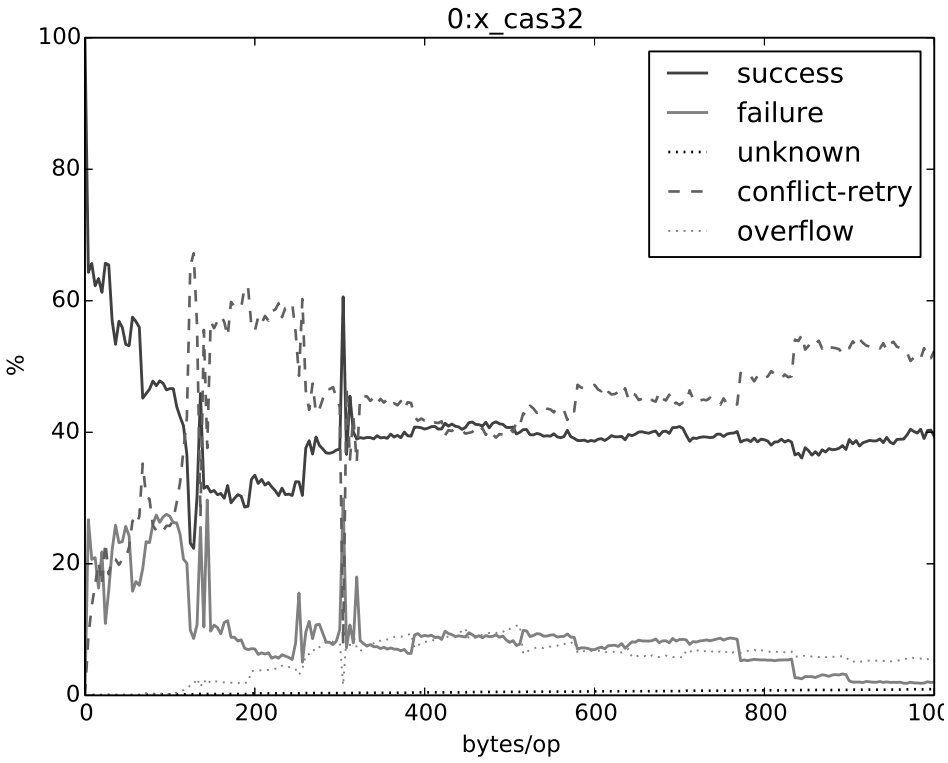
Transaction Contention: Competing Writes

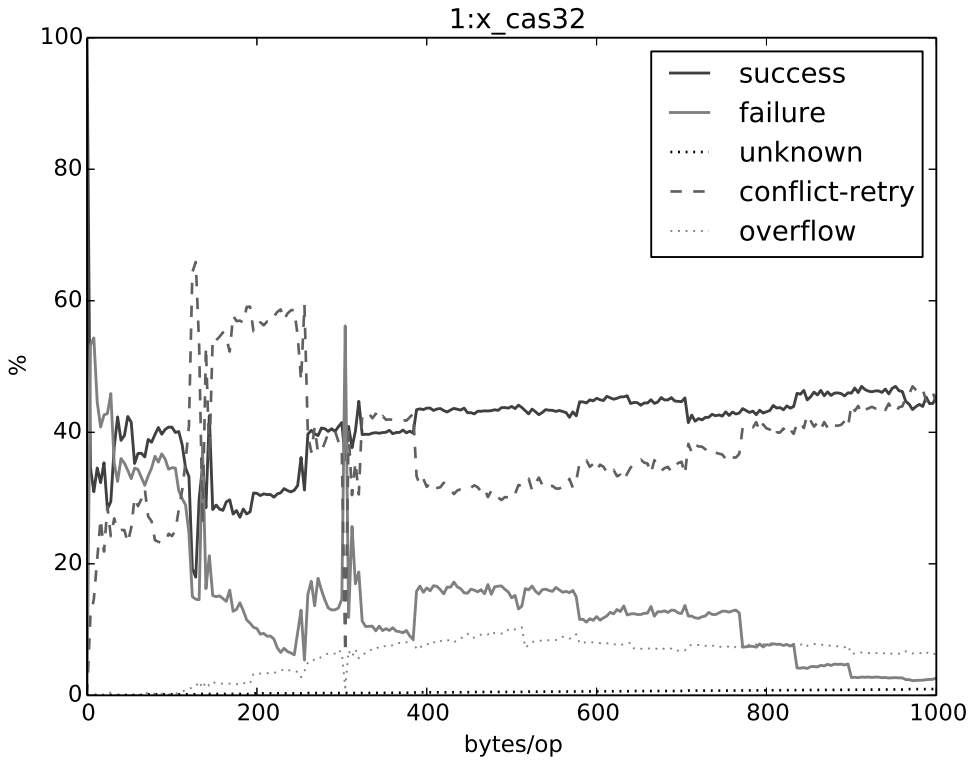




- No conflicts for **small** sizes (scheduling), but beyond 192 bytes (4 cache lines), performance degrades rapidly.
  - slight bias towards thread 1, possibly scheduling artefact.

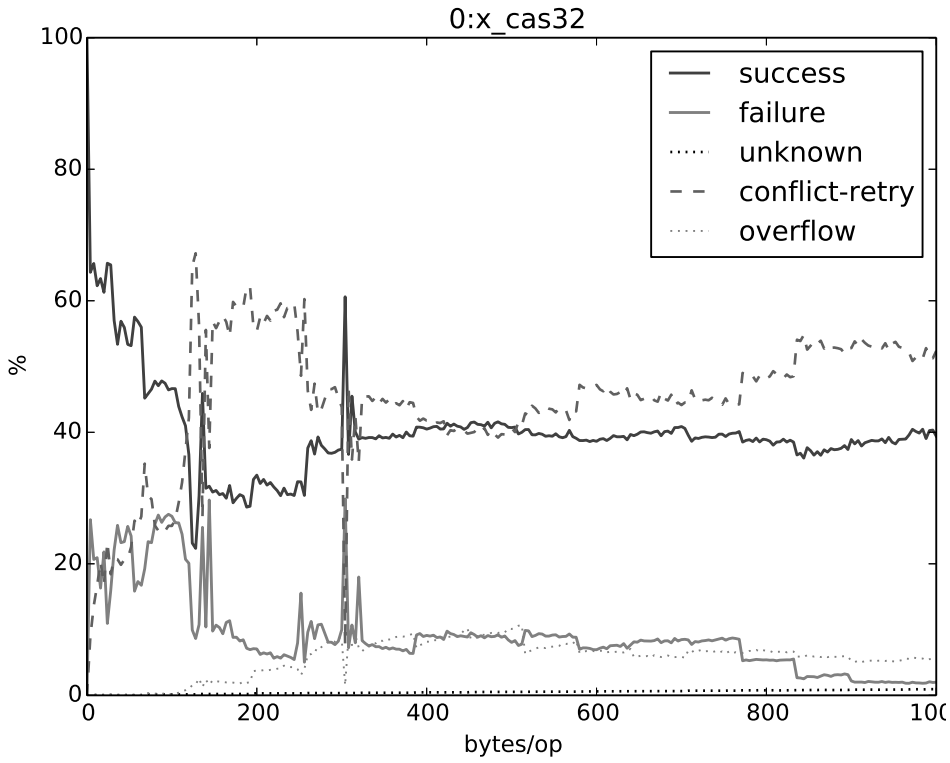
Transaction Contention: Competing CAS

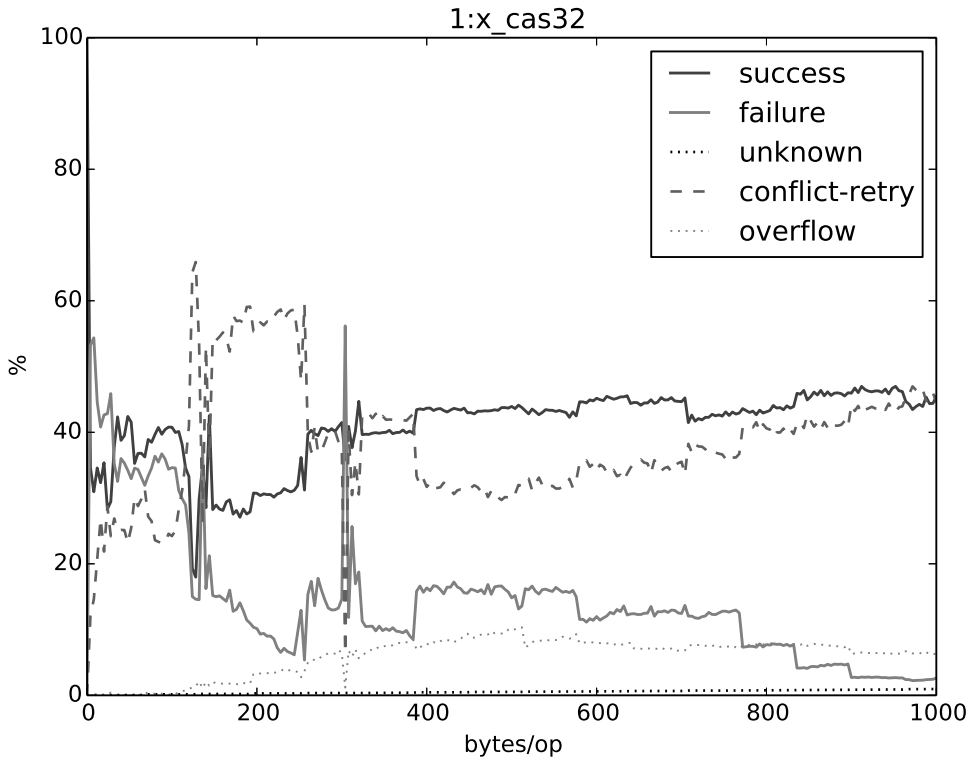




- The additional **failure** line is where the transaction succeeded, but the CAS failed.
  - because it got changed by the other thread already.
- More overflows than before.
  - either mis-reported, or something other than L1 cache-size involved.

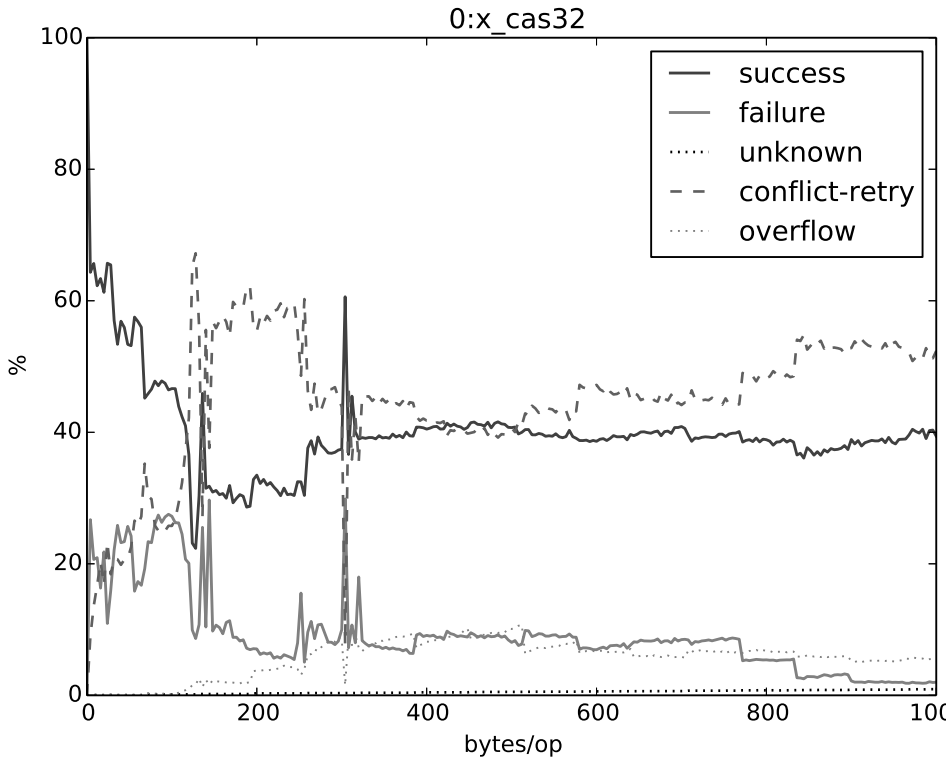
Transaction Contention: Competing CAS

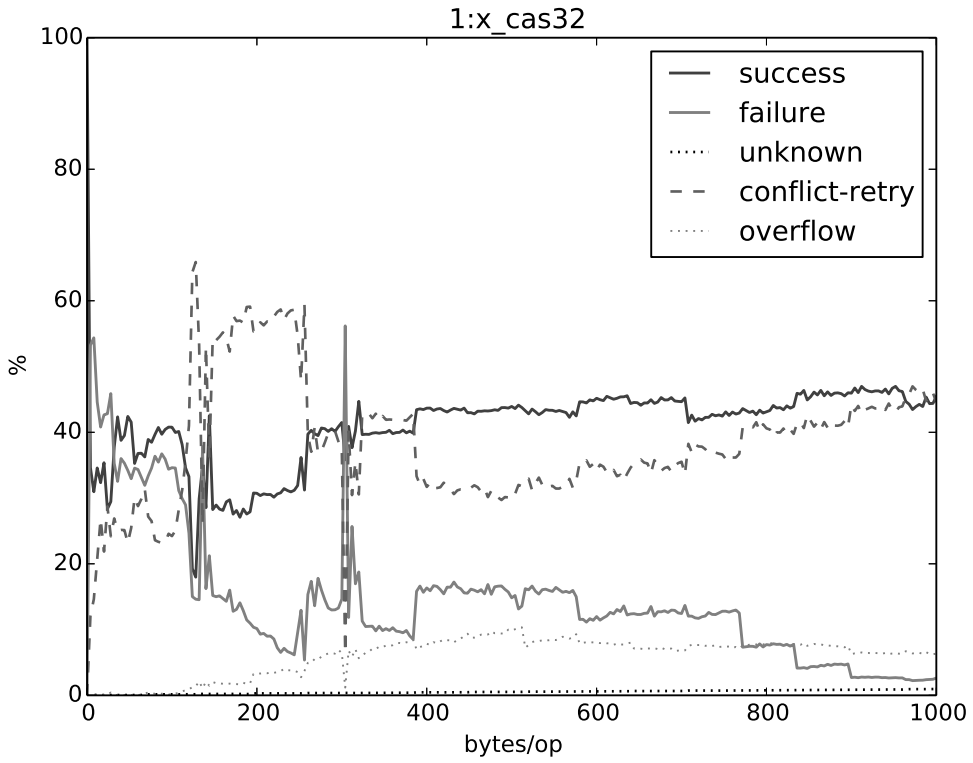




- The additional **failure** line is where the transaction succeeded, but the CAS failed.
  - because it got changed by the other thread already.
- More overflows than before.
  - either mis-reported, or something other than L1 cache-size involved.

Transaction Contention: Competing CAS

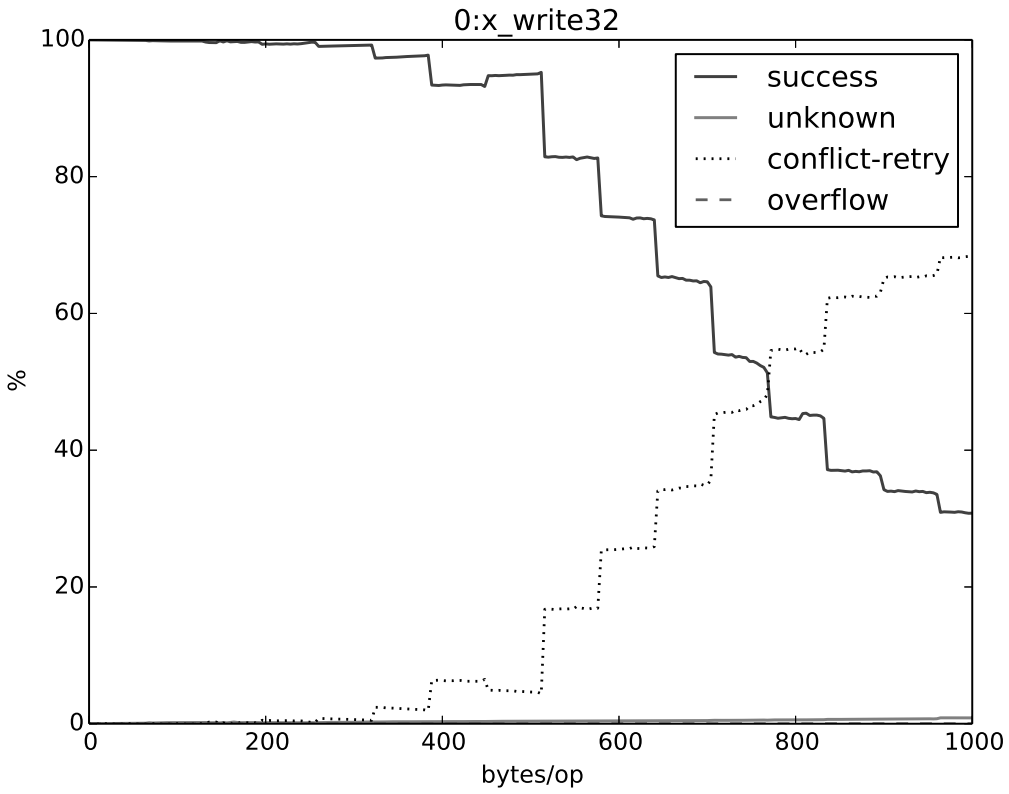




- The additional **failure** line is where the transaction succeeded, but the CAS failed.
  - because it got changed by the other thread already.
- More overflows than before.
  - either mis-reported, or something other than L1 cache-size involved.

Transaction Contention: vs. Non-Transactional

Transactional vs. non-transactional **write** (always succeeds).



- Shows less interference than two competing transactional writes.
  - area of interference from the non-transactional thread is at most 1 cache line.

Performance Summary

- **Setup** and **teardown** cost for a transaction is approx. 40 cycles.
  - for common operations such as CAS, easily amortized in 2-3 words of memory access.
- Transactions (realistically) can be up to **16 KiB** in size.
  - far from optimal here — for good performance, keep below **1 KiB**.
- No observable overhead on memory **writes**.
  - reads may incur up to 20% overhead.
- Transaction aborts are expensive — **150 cycles**, in addition to the overheads of the failed transaction.

Performance Summary

- **Setup** and **teardown** cost for a transaction is approx. 40 cycles.
  - for common operations such as CAS, easily amortized in 2-3 words of memory access.
- Transactions (realistically) can be up to **16 KiB** in size.
  - far from optimal here — for good performance, keep below **1 KiB**.
- No observable overhead on memory **writes**.
  - reads may incur up to 20% overhead.
- Transaction aborts are expensive — **150 cycles**, in addition to the overheads of the failed transaction.

Performance Summary

- **Setup** and **teardown** cost for a transaction is approx. 40 cycles.
  - for common operations such as CAS, easily amortized in 2-3 words of memory access.
- Transactions (realistically) can be up to **16 KiB** in size.
  - far from optimal here — for good performance, keep below **1 KiB**.
- No observable overhead on memory **writes**.
  - reads may incur up to 20% overhead.
- Transaction aborts are expensive — **150 cycles**, in addition to the overheads of the failed transaction.

Performance Summary

- **Setup** and **teardown** cost for a transaction is approx. 40 cycles.
  - for common operations such as CAS, easily amortized in 2-3 words of memory access.
- Transactions (realistically) can be up to **16 KiB** in size.
  - far from optimal here — for good performance, keep below **1 KiB**.
- No observable overhead on memory **writes**.
  - reads may incur up to 20% overhead.
- Transaction aborts are expensive — **150 cycles**, in addition to the overheads of the failed transaction.

Final Points

- RTM is not simply a drop-in replacement for CAS-based algorithms.
  - like non-blocking / lock-free algorithms, no guarantee of progress.
- Other uses include thread synchronisation, busy waiting, CCSP channel and scheduler algorithms, ...

Acknowledgements

- Carl Ritson did all the hard work – thanks Carl!
- The EPSRC funded MirrorGC project (EP/H026975/1).
- Faculty of Sciences research fund (for the hardware).
- Sources for the benchmarks are available:
  `https://github.com/perlfu/rtm-bench`


Questions?

References

[1] K. Fraser.
Practical lock-freedom.
PhD thesis, University of Cambridge, King's College, September 2003.

[2] M. Herlihy.
Wait-free synchronization.
ACM Trans. Program. Lang. Syst., 13(1):124–149, 1991.

[3] T. Haerder and A. Reuter.
Principles of transaction-oriented database recovery.
ACM Comput. Surv., 15(4):287–317, December 1983.

[4] M. Herlihy and J.E.B. Moss.
Transactional memory: architectural support for lock-free data structures.
SIGARCH Comput. Archit. News, 21(2):289–300, May 1993.