Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

An Evaluation of Intel's Restricted Transactional Memory for CPAs

Carl G. RITSON and Frederick R.M. BARNES
School of Computing, University of Kent, UK
{c.g.ritson, f.r.m.barnes}@kent.ac.uk

Abstract. With the release of their latest processor microarchitecture, codenamed Haswell, Intel added new Transactional Synchronization Extensions (TSX) to their processors' instruction set. These extensions include support for Restricted Transactional Memory (RTM), a programming model in which arbitrary sized units of memory can be read and written in an atomic manner. This paper describes the low-level RTM programming model, benchmarks the performance of its instructions and speculates on how it may be used to implement and enhance Communicating Process Architectures.

Keywords. Transactional Synchronization Extensions, TSX, Restricted Transactional Memory, RTM, transactional memory, performance

Introduction

With the release of their latest processor microarchitecture, codenamed Haswell, Intel added new *Transactional Synchronization Extensions* (TSX) to their processors' instruction set [1,2]. These extensions provide *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM). RTM allows the atomic manipulation of arbitrary size units of memory. Without RTM the largest unit of memory that can be atomically manipulated on x86 architectures is a single memory word, 32 or 64 bits, using for example a *compare-and-swap* (CAS) operation [1]. This restriction has necessitated the development of entire classes of non-blocking, lock-free and wait-free algorithms [3,4] along with complex architectural memory models [5,6]. Programmers using a *Communicating Process Architecture* (CPA) such as that provided by occam-π [7], Communicating Scala Objects [8] or ProcessJ [9], are freed from considering the details of atomic synchronisation on shared memory systems. However, the developers of high-performance CPAs for modern multi-core and many-core *must* concern themselves with these details.

Transactions are commonly found in database systems. These systems are, at least conceptually, providing a large shared mutable memory [10]. In a database system transactions provide: atomicity, consistency, isolation and durability [11]. In practice this means that all operations in the same transaction should be enacted together or at-least seen to be done so (atomicity), preserve system invariants (consistency), not interfere with other transactions in non-deterministic ways (isolation) and be persistent once committed (durability). Most relevant to the developers of operating systems and language run-time systems are atomicity and isolation, whereas consistency and durability are more higher level concerns. Transactions have the potential to vastly simplify the implementation of lock-free and non-blocking algorithms which required for efficient communicating process architectures [12,13].

[1] Compare-and-swap updates a memory location M with a new value Y iff its current value matches another value X, reporting the success or failure of the update.

The concept of using transactions for system memory was first described by Herlihy and Moss in 1993 [12]. Although hardware support for transactional memory has been implemented in specialised processors such as IBM's BlueGene/Q [14] and the cancelled Sun Rock processor [15], it has taken 20 years for it to become available in a commodity processor. In the intervening timespan much work has been done on *Software Transactional Memory* (STM) [16], where transactional memory is provided by a language run-time system rather than dedicated hardware. The literature has focused on how memory transactions (using STM) can be exposed to programmers; however, as CPAs do not typically expose shared memory this work is not relevant. Coupled with this, the lack of hardware support and the poor performance of STM [17], transactional memory has not seen significant uptake in the implementation of CPAs. One notable exception is Brown's *Communicating Haskell Processes* which uses STM to implement multiway choice and synchronisation [18].

This paper focusses on what Intel's hardware RTM offers the designer or implementor of a *run-time system* (RTS) or *virtual machines* (VM) for a *communicating process architecture* (CPA). Content is broadly divided into three sections: section 1 describes the instruction set and associated *programming model* for RTM; section 2 evaluates the *performance* of RTM on a recently released Core i7-4770 processor; and section 3 looks at the potential *applications* of RTM in the implementation of CPAs. Conclusions and future work are reviewed in section 4.

1. Programming Model

This section briefly describes the instruction set and programming model associated with Intel's RTM facilities.

At an instruction level RTM is simple to use. A transaction is initiated with an `XBEGIN` instruction. Computation proceeds normally; memory can be read and written with common instructions and other activities such as branching and arithmetic may also be used. At the end of the transaction an `XEND` instruction commits any changes to memory.

During the transaction read and write sets are constructed. These sets have cache line granularity and are based on the memory addresses read or written by the transaction's body. If these sets conflict with memory being read or written by other hardware threads then the transaction is aborted. If a transaction is aborted, all changes to memory and registers are discarded and execution jumps to a *fallback handler* supplied to the initial `XBEGIN` instruction. Before invoking the fallback handler, status flags are set in the processor's `EAX` register. These flags allow the fallback handler to determine what caused the transaction to abort.

It is worth clarifying that read and write sets are specific to a given hardware thread. We use the term hardware thread to describe the smallest unit of parallel execution in a computer system. In Intel's Haswell architecture a computer may have multiple processors, each of which has multiple cores, each of which has multiple hardware threads. Hardware threads on the same core share components such as arithmetic units and cache, but have their own registers and instruction pointer.

1.1. Instructions

There are only four machine instructions used to access RTM functionality.

- `XBEGIN` initiates a transaction. An instruction pointer (relative to the current instruction pointer) to a fallback handler is passed. `XBEGIN` is also valid within a transaction, permitting *nesting* (see 1.3).
- `XEND` completes a transaction, flushing changes to memory, assuming this is not a nested transaction (see 1.3). Following this instruction it can be assumed that changes are visible to other hardware threads.

- `XABORT` aborts the current transaction. It takes an 8-bit failure code which is accessible to the fallback handler.
- `XTEST` updates the processors comparator flags such that a subsequent branching instruction can determine if the processor is executing within a transaction or not.

1.2. Restrictions

The term *restricted* in restricted transactional memory refers to the fact that not all processor instructions and functions may be used within a transaction. If a restricted operation is attempted then the transaction is aborted and the fallback handler invoked. Essentially any instruction that causes changes to the processor or system state that cannot (trivially) be reverted induces a transaction to abort.

There are several restrictions worth noting:

- X87 floating-point and *multi-media extensions* (MMX) are not supported; however, this is a not a significant concern as *streaming SIMD extensions* (SSE) and *advanced vector extensions* (AVX) are supported permitting floating-point and vectorised computation.
- Instructions that halt the processors execution such as `PAUSE` and `MWAIT` are not supported.
- Debugging mechanisms are not supported, while not verified in this work essentially breakpoints cannot be placed within a transaction.
- An interrupt within a transaction will cause the transaction to abort before invoking the interrupt handler.
- Changes in privilege level will cause a transaction to abort and disregard the source of the invocation. This will prevent calls into the operating system kernel.
- Any exception will cause a transaction to abort and disregard the source of the exception. This includes memory addressing exceptions and page faults, meaning all memory accessed during a transaction must be mapped.

1.3. Nesting

Nesting of transactions is supported. Each invocation of the `XBEGIN` instruction increases a *nesting count* and `XEND` decrements the same count. The transaction is only completed and committed to memory when the nesting count is reduced to zero. However, any conflict causing the transaction to abort will invoke the outermost fallback handler. Essentially nested transactions are simply folded into the outermost transaction.

1.4. Failures

When a transaction is aborted and fails all changes to the processor state made within the transaction are discarded. This includes any exceptional conditions raised, such as a page faults. Thus if an exception is triggered within a transaction, the transaction will abort and the exception handler *will not* be invoked. A special case is interrupts which cause the transaction to abort, but still invoke the interrupt handler once the transaction state has been discarded.

The cause of a transaction's failure is reported to the fallback handler by setting flags in the `EAX` register. At the time of writing six flags are defined:

0 *XABORT*: An `XABORT` instruction aborted the transaction, in which the `EAX` register also carries the 8-bit code pass to the `XABORT` instruction.
1 *Retry*: The transaction might succeed if retried.
2 *Conflict*: Interference from another processor, core or hardware thread caused the transaction to abort.

3 *Overflow*: Overflow of buffers caused the transaction to abort.
4 *Debug*: A debug break point was encountered causing an abort.
5 *Nested*: The transaction was aborted within a nested transaction.

2. Performance

This section evaluates the performance of RTM on generally available hardware. The goal is to provide data which can inform the design of language run-time systems and CPA implementations.

When implementing a language run-time it is important to minimise run-time cost in terms of space and time complexity. However, without meaningful data it is not possible to make reasoned choices. For example, when implementing for performance it would not make sense to use a memory transaction if the same operation could be implemented with three CAS instructions if the overhead of a transaction is ten times that of a CAS.

With respect to Intel's RTM the critical questions to answer are:

- What is the cost of setting up a transaction? (see 2.3)
- What is the cost of committing a transaction? (see 2.3)
- Is there an overhead on operations within a transaction? (see 2.3 and 2.5)
- How big can a transaction reasonably be? (see 2.4)
- What is the cost of failed transactions and conflicts between transactions? (see 2.3 and 2.6)

2.1. Test Setup

All tests and benchmarks were performed using a system with a single Core i7-4770 processor running at 3.4GHz, with 16GiB of RAM running at 1600MHz and CAS timing of 9-9-9-24. Stock Ubuntu Linux 12.04.2 LTS was used with kernel 3.5.0-23-generic. No specific compiler support for RTM was employed, instead assembly macros were used in the standard GCC 4.6.3 installation.

The processor *Turbo Boost* was disabled in the system's BIOS to allow for consistent results regardless of hardware temperature. Additionally, CPU frequency scaling was disabled by modifying the `scaling_min_freq` parameter in the `sysfs` entry to match the `scaling_max_freq`. While executing POSIX threads were bound to specific hardware threads (or processor cores) using the `sched_setaffinity` API. The scheduling quantum was not adjusted.

2.2. Test Strategy

We define four different test operations:

- *read*: load words from increasing memory locations.
- *write*: store words to increasing memory locations.
- *cas*: compare-and-swap words at increasing memory locations.
- *abort*: store words to increasing memory locations, but abort the transaction after n words.

Each operation (with the exception of *abort*) can operate in either untransactional (u) or transactional (x) *mode*. Additionally the *word size* for these operations can be either 32-bits or 64-bits. For each combination of operation, mode and word size we test increasing numbers of words per operation, or *operation size*.

At each operation size we recorded the amount of time (and clock cycles) required to complete a number of operations (between 16384 and 131072), in order to derive the mean

speed in bytes per nanosecond. Within a test run each operation uses a distinct memory area (aligned to a cache line boundary) to minimise the effect of L2 and L3 processor caches. The total size of the memory area is 512MiB, hence 16384 operations are performed when the total operation size is 32KiB.

We also recorded the success or failure of operations allowing calculation of failure rates. We do not retry failed transactions, but simply move on to the next operation. Failed transactions are disregarded when computing performance.

Untransactional reads and writes will always succeed; however, untransactional compare-and-swap (using `LOCK` prefix) can fail if multiple threads are using the same memory. Transactional aborts will always fail, although the failure reason may not be the abort if a transaction was interrupted before the `XABORT` instruction.

For abort operations we define two variants:

- *abortn* which sets up to do an operation of a given size, but aborts before performing any memory access,
- *abortm* which performs all memory access before aborting the transaction.

Both have the same compiled structure and use a counter to trigger the transaction abort. This is intended to minimise pipelining effects in the processor.

The source code for our benchmark is available on GitHub: `https://github.com/perlfu/rtm-bench`.

2.3. Base Transaction Cost

To determine the base cost of transactions we examine the time taken to complete untransactional and transactional operations with a size of zero. The overhead of our instrumentation and test framework will be the same between untransactional and transactional versions allowing us to determine the cost of invoking transactional mode on the processor.

Table 1. Cost of zero size test operations.

Operation	Time 32-bit	Time 64-bit	Cycles 32-bit	Cycles 64-bit
u_read	2.0ns	2.4ns	7.0	8.0
u_write	2.1ns	2.1ns	7.0	7.0
u_cas	2.0ns	2.0ns	7.0	7.0
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

With respect to table 1 the base cost of invoking transactional mode appears to be 13ns or 43 clock cycles. We also observe that triggering a transactional abort is relatively expensive. If we assume this is because a pipeline and cache flush occurs then we can infer that successful transactions are effectively pipelined.

Before plotting trends it is desirable to closely review the cost of small numbers of operations. Table 2 and table 3 show these costs for 32-bit and 64-bit operations respectively. The critical observation is that once a transaction reaches two words in size it has equivalent cost to compare-and-swap operations on the same number of words. Beyond this (very low) tipping point a transaction is more efficient than compare-and-swap with the benefit of overall atomicity.

Table 2. Clock cycles for operations on small numbers of words for 32-bit mode.

Operation	1 word	2 words	3 words	4 words	5 words	6 words	7 words	8 words
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

Table 3. Clock cycles for operations on small numbers of words for 64-bit mode.

Operation	1 word	2 words	3 words	4 words	5 words	6 words	7 words	8 words
u_read	18	11	11	12	14	16	16	17
u_write	37	27	24	23	22	22	22	22
u_cas	37	58	72	92	110	129	148	167
x_read	60	62	67	67	67	67	67	67
x_write	63	62	61	61	60	60	60	61
x_cas	60	64	64	71	72	78	75	80
x_abortn	66	162	162	162	162	163	163	163
x_abortm	170	175	173	175	179	182	182	181

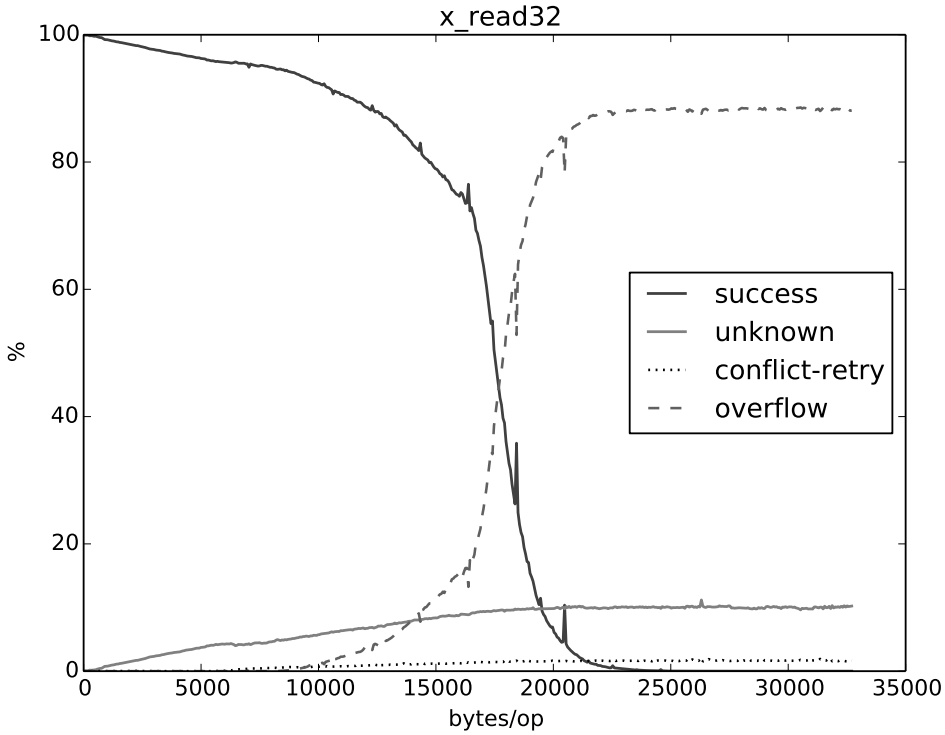

Figure 1. Success and failure rates for *read* transactions of varying sizes.

2.4. Transaction Size and Failures

We examine transaction completion rates at different sizes to determine both optimal and maximal transaction size. Figure 1 shows transaction failure and success rates for increasing operation (and transaction) sizes for 32-bit *read* operations. Figures 2 and 3 show the same information for *write* and *cas* operations respectively.

Up to 10KiB transactions performance degrades from success to unknown failure in an

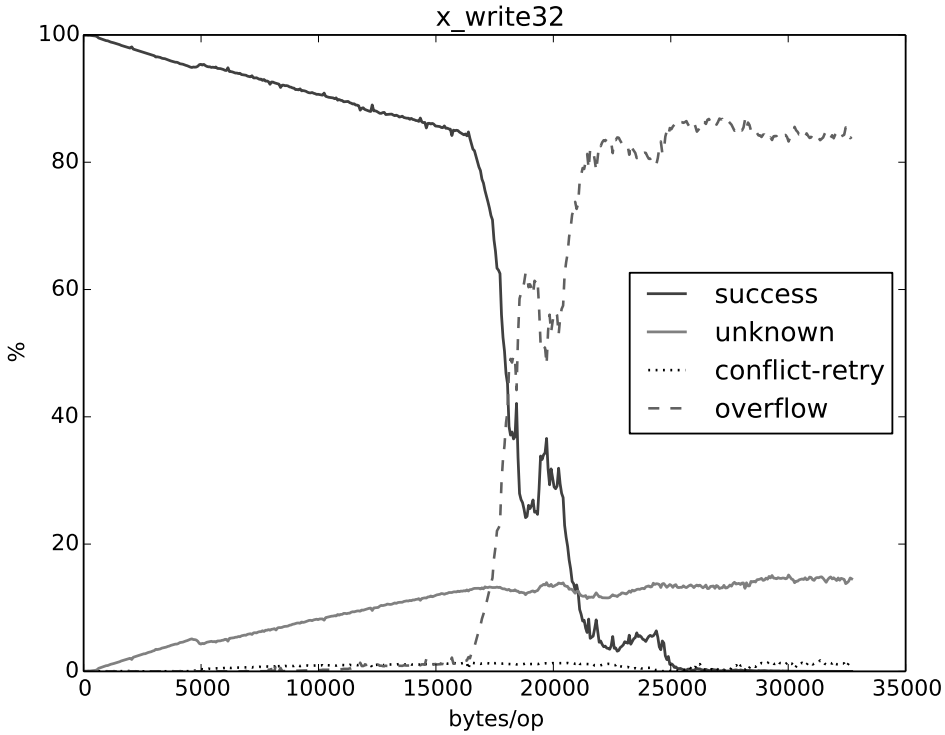

Figure 2. Success and failure rates for *write* transactions of varying sizes.

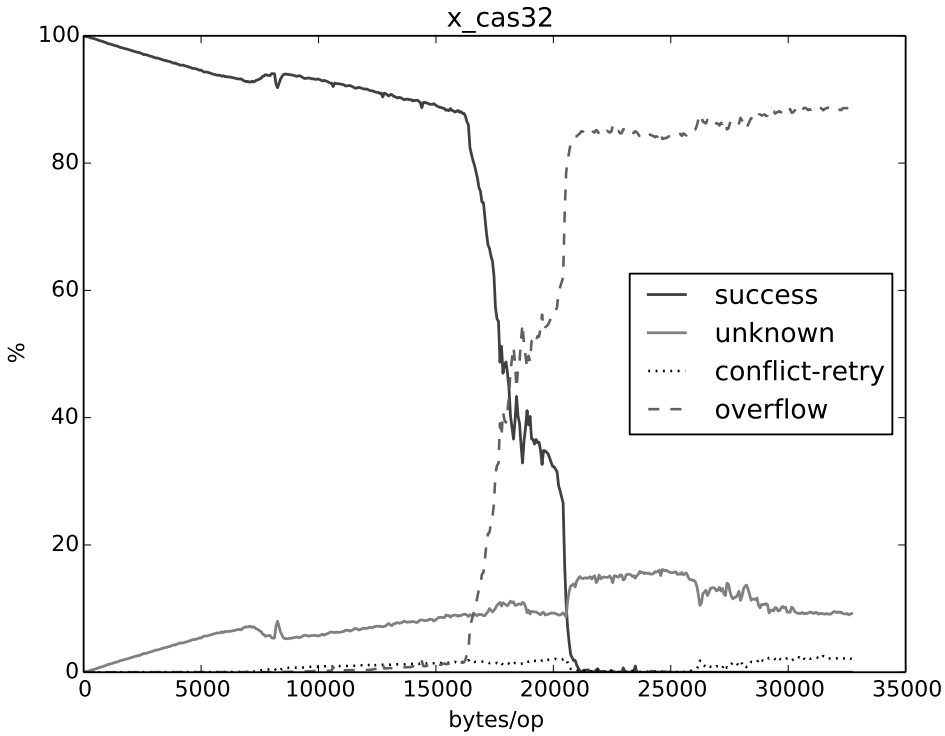

Figure 3. Success and failure rates for *cas* transactions of varying sizes.

almost linear manner. These unknown failures indicate that no transactional abort cause was indicated by the processor, which we believe relates to operating system context switches. This behaviour would be consistent with increased transaction length resulting in increased probability of an operating system interrupt. We could have adjusted the operating system scheduling quantum, but chose not to do so in order provide results representative of normal operating conditions.

There is a pronounced phase shift at 16KiB where transaction success rapidly gives way to overflow related transaction aborts. It is also worth noting that there is a small, but

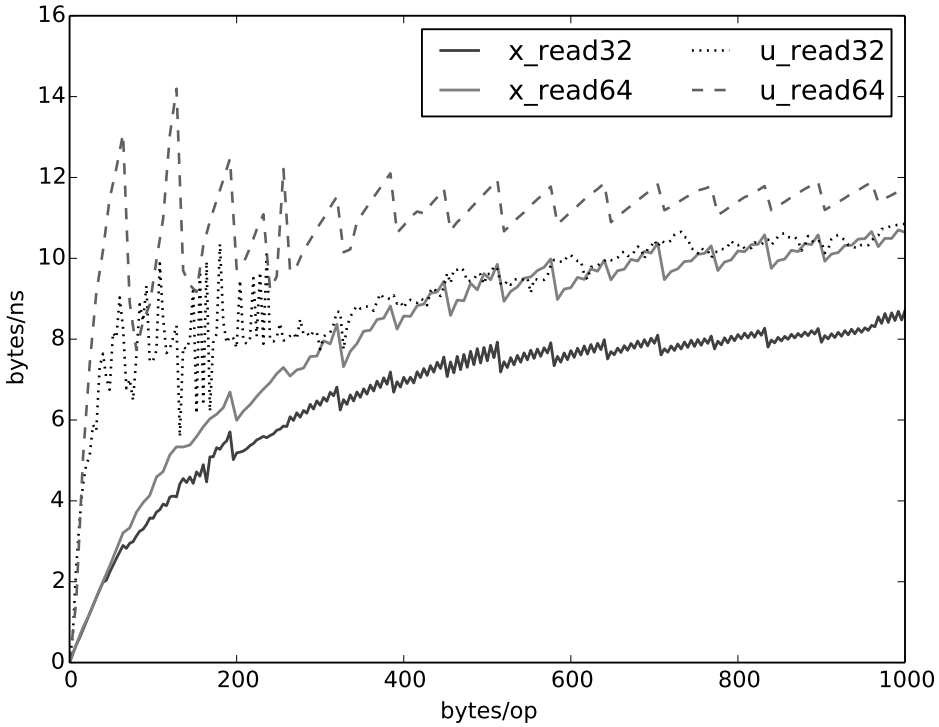

Figure 4. Performance of transactional reads compared with untransactional reads.

observable, set of failures related to memory conflicts despite the fact that no shared memory has been used or accessed. We assume this is either incorrect reporting by the processor or operating system operations (such as page table manipulations) occuring on another core.

It is our observation that the largest stable transaction size is 16KiB, where there is a 85% chance of success (on our test setup and without contention). We speculate that the transaction buffer is the processors L1 data cache; however, parts of this are also used to shadow any modified registers reducing the available buffer size. We also observe that no transactions complete once the transaction size exceeds 26KiB.

2.5. Transaction Performance

Figure 4 shows the performance (in bytes per nanosecond) of memory reads occuring with increasing transaction sizes. At small sizes we observe transactional reads to have significantly lower performance than untransactional reads; however, at larger transaction sizes transactional reads reach 80% of the performance of untransactional reads. A sawtooth pattern occurs in these results (and others) at a frequency of 64 bytes, which corresponds to the size of a cache line. With respect to the data, optimal performance is achieved when the operation is a multiple of the cache line size. We infer that this occurs because the processor loads a cache line of data at once, thus an operation loading 68 bytes of data will have the same cache fill overhead as a 128-byte operation.

Figure 5 shows the performance of memory writes, transactional versus untransactional. Unlike memory reads we observe rapid convergence of transactional and untransactional write performance. At operation sizes 256 bytes and larger the overheads of transaction setup and teardown have been completely amortized. We propose that this suggests that for reads there is a constant overhead per-read when operating within a transaction, but there is not for writes.

Figure 6 compares the performance of transactional compare-and-swap operations with untransactional compare-and-swap operations synchronised using the `LOCK` prefix instruction. This locks the memory bus, making changes to memory by a following (single) instruction atomic. In practice modern processors do not lock the memory bus, but use cache coherence

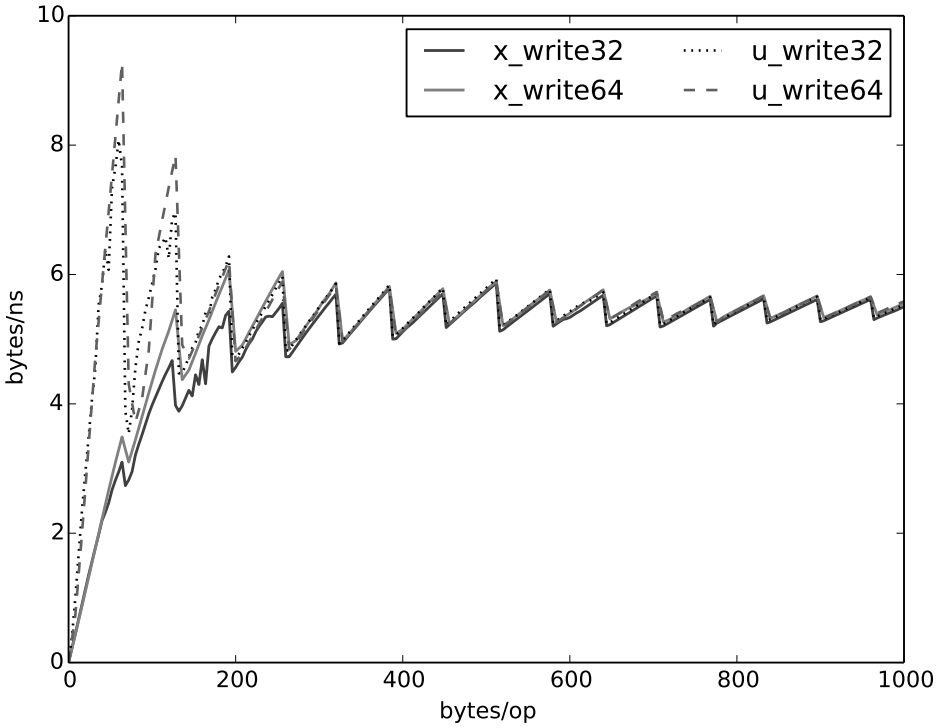

Figure 5. Performance of transactional writes compared with untransactional writes.

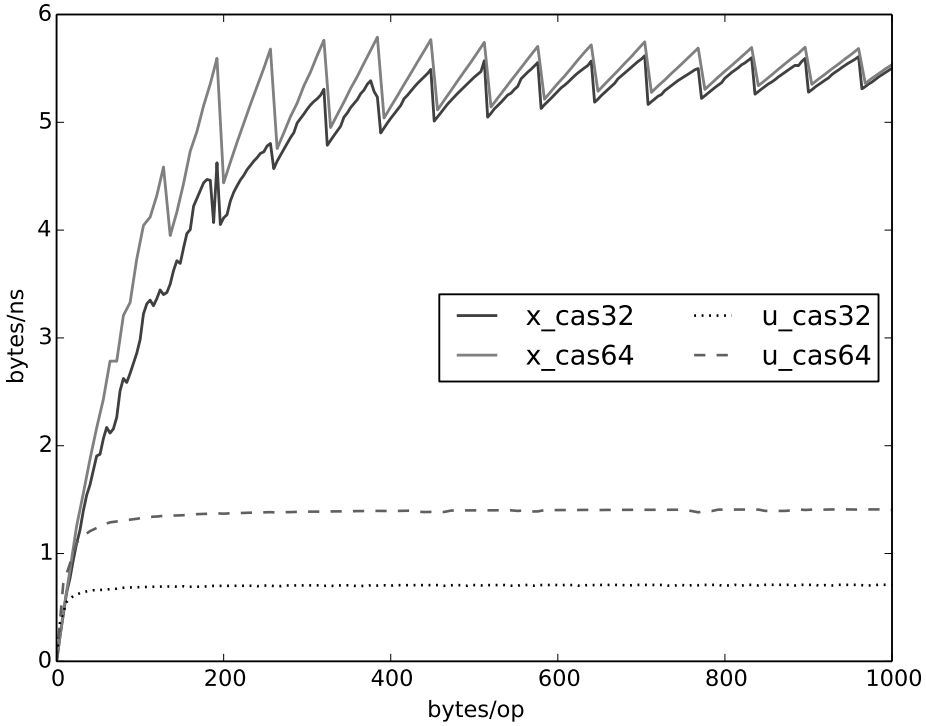

Figure 6. Performance of transactional *cas* compared with untransactional *cas* (synchronised with `LOCK` prefix instructions).

mechanisms to achieve the same effect. We observe that transactional compare-and-swap quickly surpasses the (constant) performance of its untransactional equivalent, achieving approximately five times the performance.

Figure 7 compares the performance of different kinds of transactional operations. What can be observed is that the cost of transactional write and compare-and-swap operations converge once they reach 256 bytes in size.

Finally, Figure 8 illustrates the cost of explicitly aborting transactions. The *abortn* op-

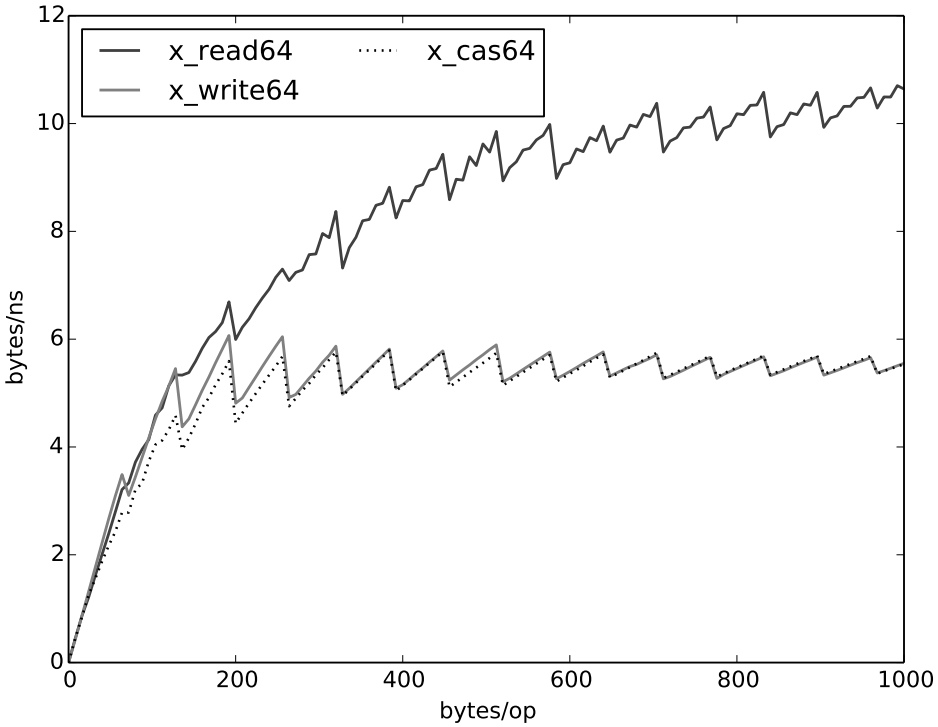

Figure 7. Performance of three different types of transactional operations: 64-bit read, 64-bit write and 64-bit *cas*.

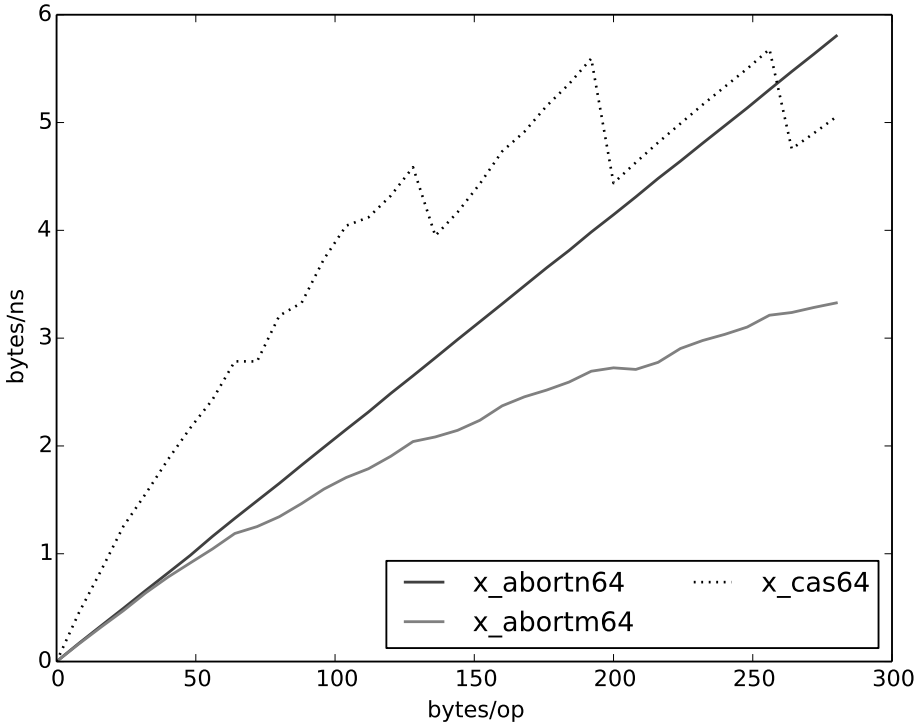

Figure 8. Performance/cost of explicitly aborting a transaction, 0 to 300 bytes in size.

eration aborts the transaction without explicitly modifying any data, thus shows a linear ($y = mx$) relationship as we would expect. Performance of the *abortm* operation which writes all memory words of the operation size before aborting the transaction has significantly worse performance than compare-and-swap operations which equally write all memory words before completing. That said, there are signs of convergence, and looking at larger operation sizes (Figure 9), *abortm* performance surpasses compare-and-swap performance around 600 bytes. From these results we can conclude that for small transactions successful completion

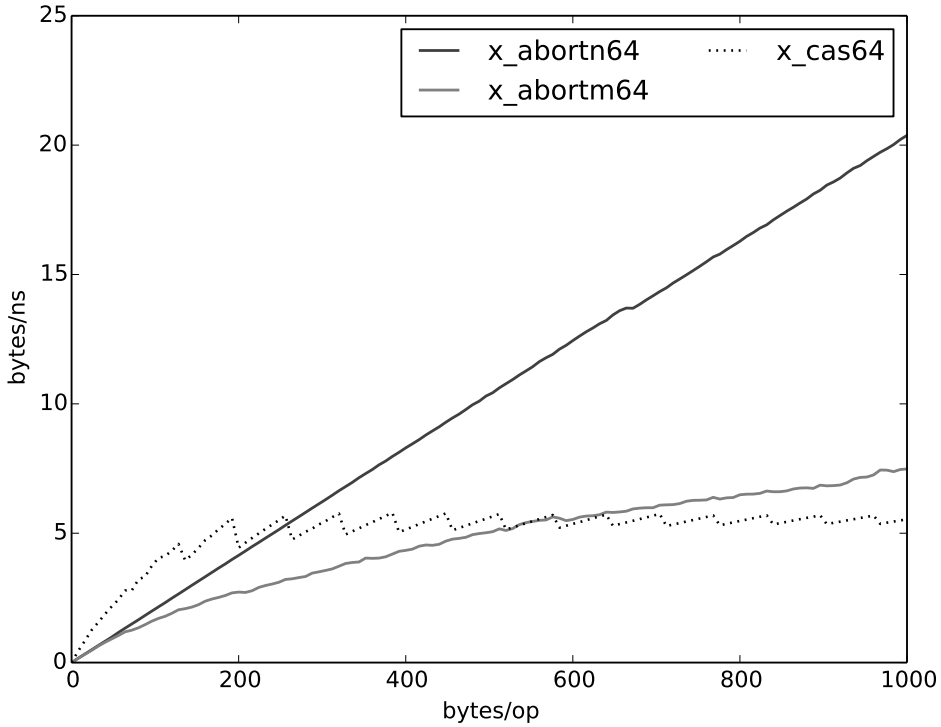

Figure 9. Performance/cost of explicitly aborting a transaction, 0 to 1000 bytes in size.

is significantly cheaper than unsuccessful completion. We infer that this cost arises from the restoration of register state within the processor, a cost not incurred by successful transactions which can simply discard old state data.

2.6. Transaction Contention

We performed a very preliminary evaluation of the interactions between concurrent transactions by different cores on a shared area of memory. Results from these evaluations may be less accurate than those presented so far as threads may progress at different rates artificially reducing contention. That said, for larger operation sizes these results should still be indicative of real-world behaviour.

Initially we tested the effect of two threads both reading from the shared memory. We observed no change in the completion rate of either thread. This is what we would expect and shows that shared read sets do not cause transactions to abort.

Figure 10 shows the results of two threads (0 and 1) performing competing write transactions on the same memory buffer. For smaller operation sizes conflicts do not arise, but beyond 192 bytes (four cache lines) the success rate rapidly degrades in both threads. There appears to be a slight bias toward thread 1 succeeding, but this is probably an artifact of synchronisation within the test harness. The stepped changes in success rates reinforce the notion that read and write sets of transactions are of cache line granularity [1].

Figure 11 shows the results of two threads (0 and 1) performing competing compare-and-swap transactions on the same memory buffer. Here *failure* indicates that the transaction completed successfully, but that the memory had already been compare-and-swapped by the other thread. The interesting observation in this test is that almost 10% of transactions fail with transactional buffer overflows despite the relatively small size of transactions. This suggests that either the processor is misreporting failures as overflows or there is another factor aside from L1 cache size which is involved when multiple cores are performing transactions.

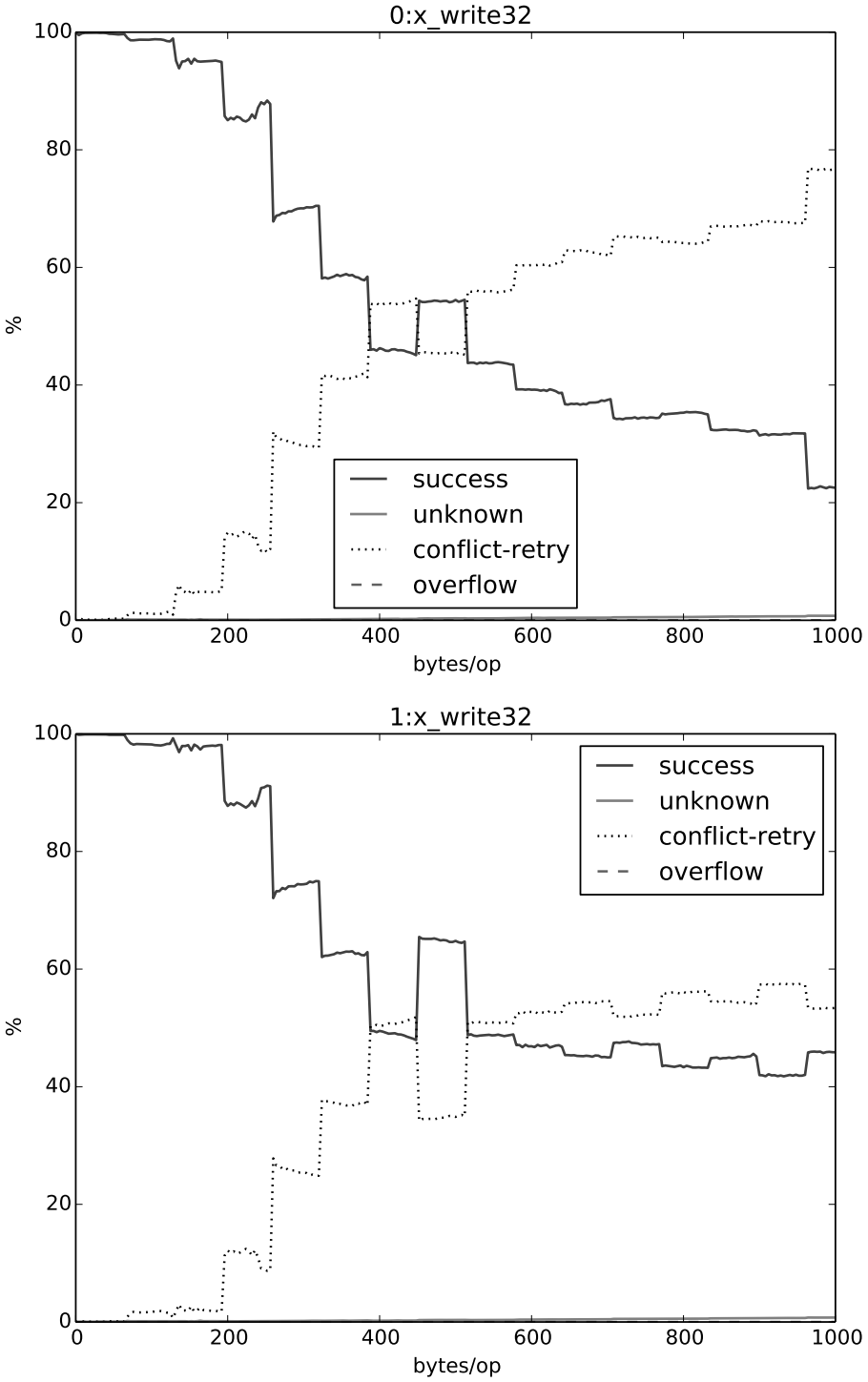

Figure 10. Success and failure rates for competing *write* transactions of varying sizes.

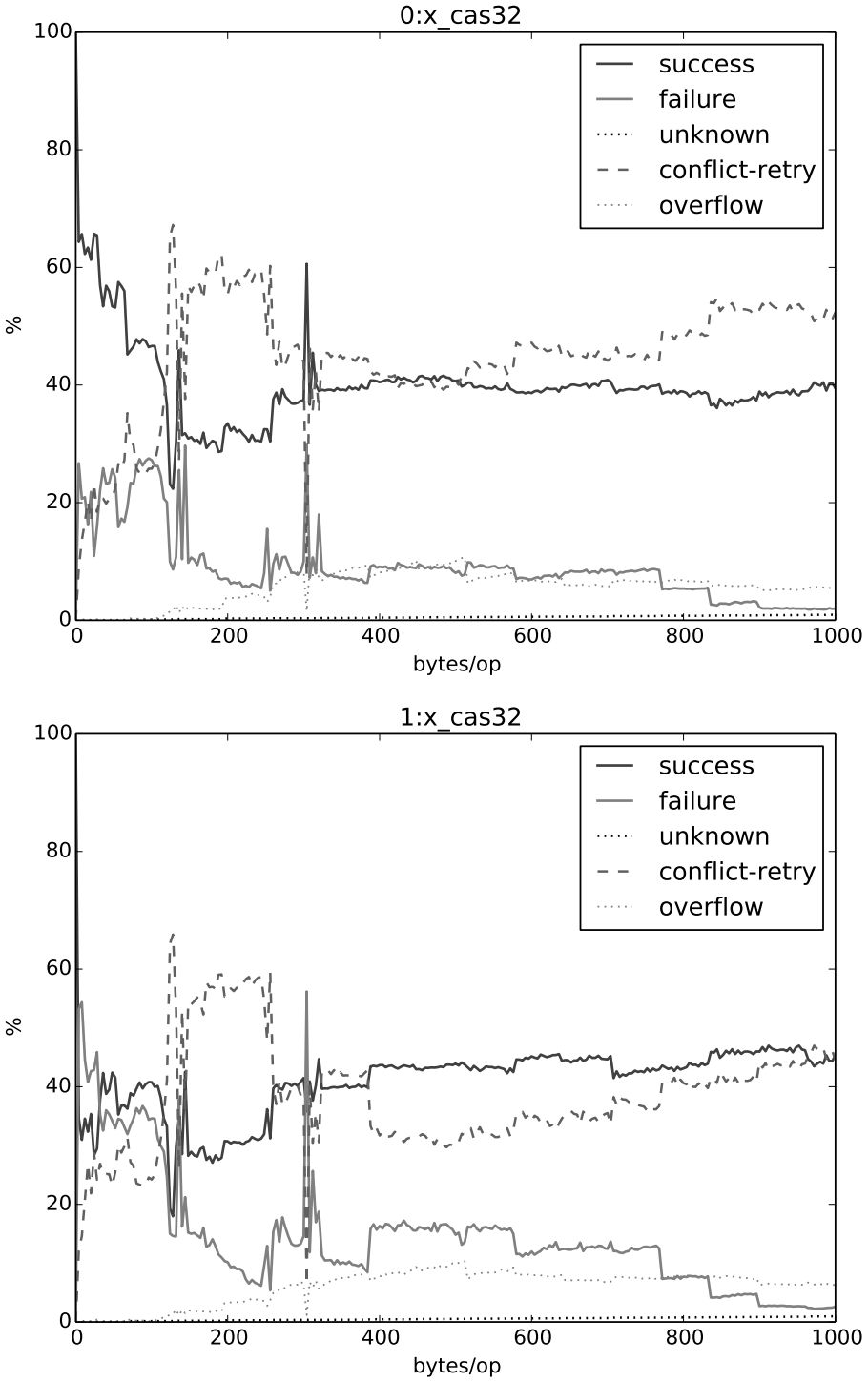

Figure 11. Success and failure rates for competing *cas* transactions of varying sizes.

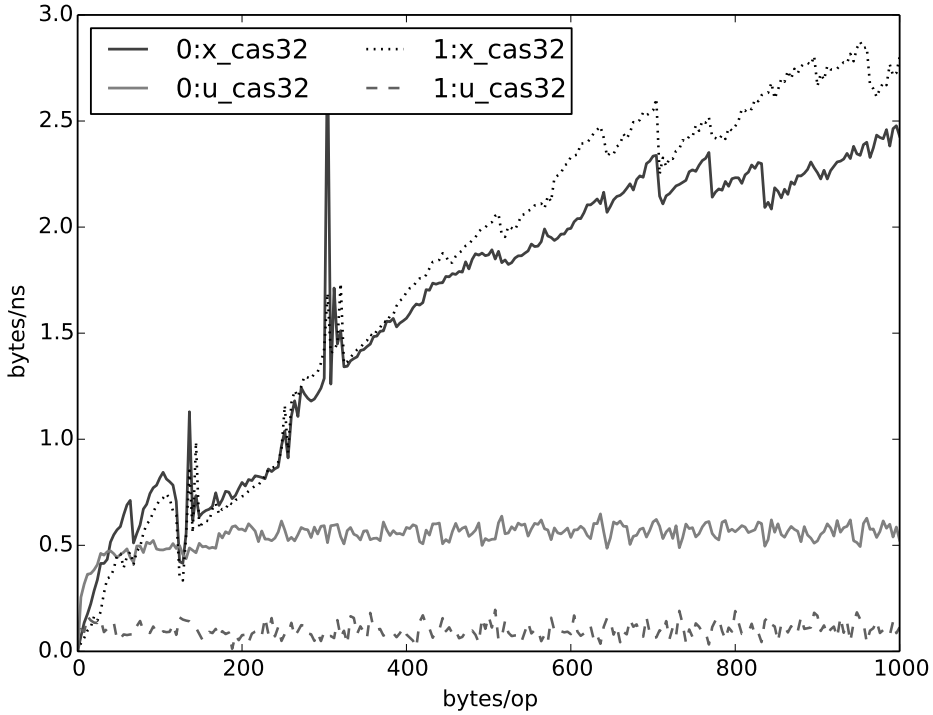

Figure 12. Performance of competing transactional and untransactional *cas* operations.

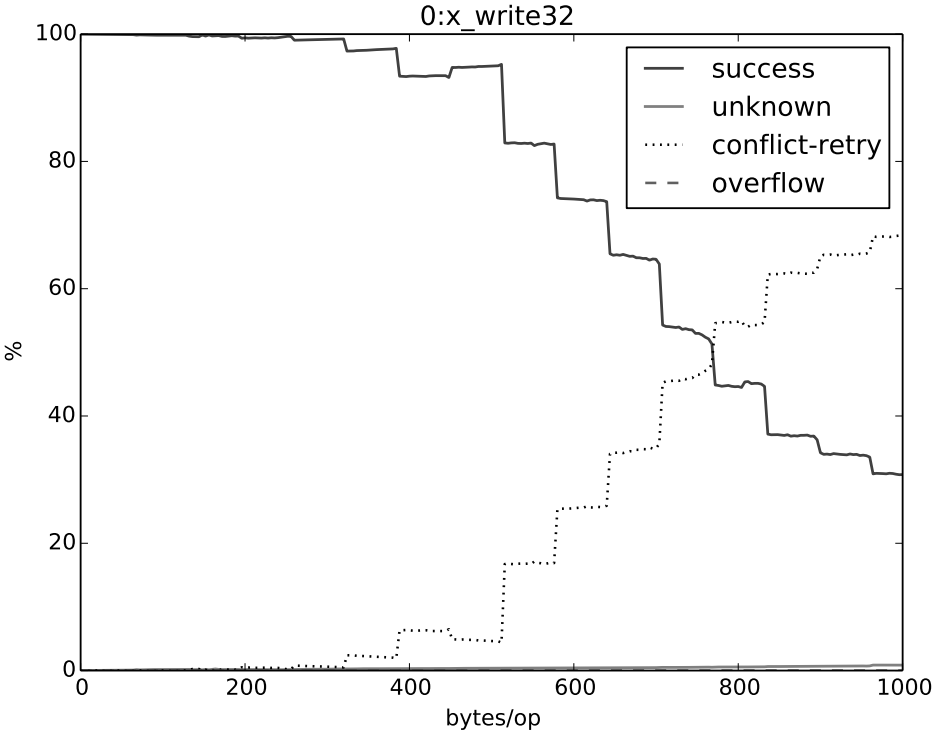

Figure 13. Success and failure rates for transactional *write* operations competing against untransactional *write* operations.

Figure 12 illustrates the performance of these competing threads and compares them to equivalent competing untransactional threads. The performance of untransactional compare-and-swap operations is essentially constant, whereas transactional compare-and-swap operations have better performance with larger operation sizes.

Figure 13 shows the competition of transactional memory writes with untransactional memory writes. As untransactional writes always succeed, success and failure is only shown for the transactional thread. The results show a lower level of interference than competing

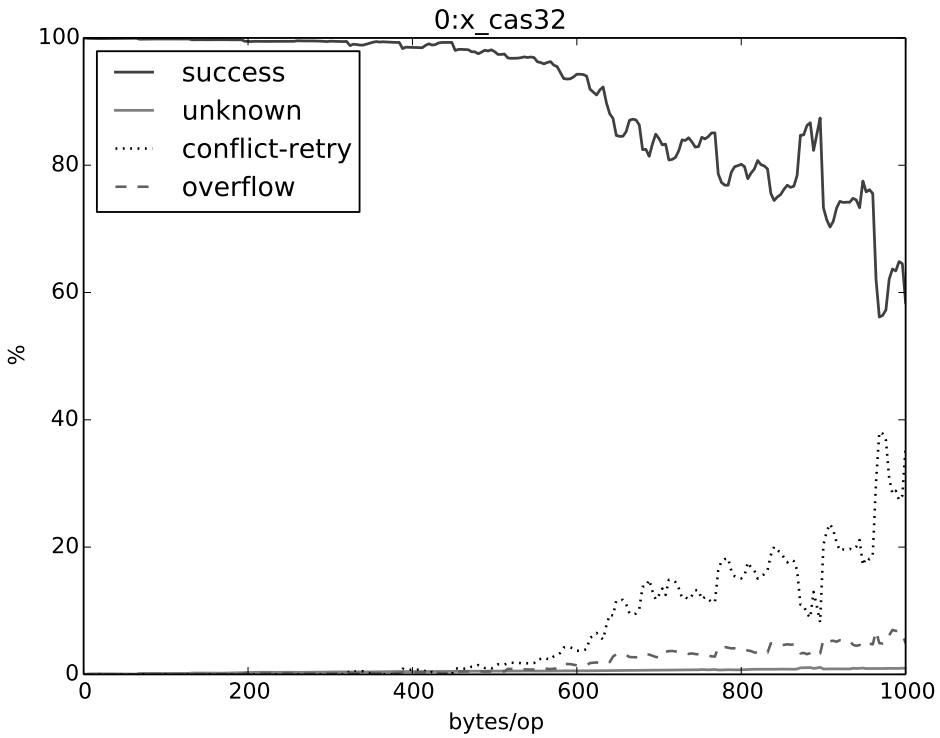

Figure 14. Success and failure rates for transactional *cas* operations competing against untransactional *cas* operations.

transactional memory writes. Presumably this is because the area of interference from the untransactional thread is always limited to the size of a cache line.

Finally, Figure 14 explores the competition of transactional compare-and-swap operations with untransactional memory reads. As with the previous example success rates are only shown for the transactional thread. This figure suggests that reads can interfere with active transactions. As read performance is greater than write performance it is possible for the reading thread to race ahead, thus the true extent of interfere may in practice be greater than that indicated in the figure. However, Figure 15 shows the performance of the same threads and suggests that the memory bus is saturated (convergence) with operations of 400 bytes and greater. Thus interference indicators for 400 to 1000 bytes should be broadly accurate.

2.7. Summary

The setup and teardown cost of a transaction is approximately 40 processor clock cycles. For common atomic operations such as compare-and-swap this cost can be amortized within approximately two to three words of memory access.

Transactions can realistically be up to 16KiB in size, but are probably far from optimal at this maximum size. For optimal performance transactions should probably be less than 1KiB.

There is no observable overhead on memory writes within a transaction, although there may be upto 20% overhead on memory reads. Thus the larger the transaction, the more effectively the transaction setup costs are amortized, at increased risk of failure.

Under contention, transaction failure (through conflict) becomes common when more than three or four cache lines are involved in the transaction. With high contention performance is equivalent to that of existing atomic operations such as compare-and-swap. There appears to be a degree of bias toward particular processor cores; however, this requires further investigation.

Transaction aborts are expensive, costing at least 150 processor clock cycles. This cost is in addition to the operations performed within the transaction. We infer that this cost or a

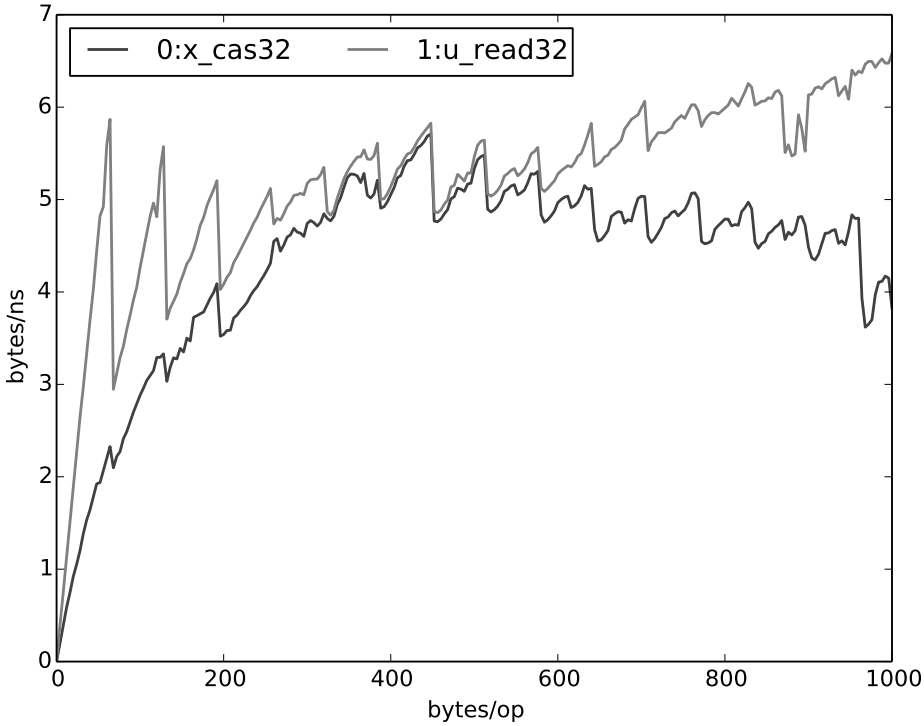

Figure 15. Performance of competing transaction *cas* and untransactional *read* operations.

higher one is also paid when a transaction is aborted due to interference. Where worst-case performance matters, for example within wait-free algorithms, traditional atomic operations (untransactional compare-and-swap) may be more appropriate.

3. Applications

This section describes how RTM can potentially be applied to the implementation of CPAs and their associated run-time systems. Subsections describe either novel techniques made possible using RTM or potential strategies for applying RTM to existing problems in the implementation of CPAs.

3.1. Fallback

A critical observation based on the performance details in section 2, is that a restricted transactional memory is not simply a CAS replacement. Replacing an algorithm which deterministically requires only one or two atomic operations with a transactional variant will likely reduce performance. Additionally, an existing algorithm embedded in a transaction is not guaranteed to make progress, as a transaction may encounter a conflict even when repeated if contention is present. This concern is common to lock-free and non-blocking algorithms [19] and where progress is to be guaranteed a fallback algorithm, perhaps with a lock, must be integrated. This fallback must also interact correctly with operating transactional code.

Fallback from transactional code to locked code can be achieved in a number of ways. Any read during a transaction adds the relevant cache line to the transaction's read set. Thus at the beginning of a transaction a lock word can be tested adding it to the read set. If non-transactional code takes this lock (changing the lock word) then the transaction will be aborted due to conflict.

Alternatively the transactional code could write the lock word at the beginning of the transaction. This will conflict with any other processor accessing the same lock word simultaneously regardless of other memory access within the transaction. Non-transactional code

will still be able to override transactions by setting the lock word and aborting any running transaction, giving non-transactional code priority over transactional code. This is essentially the approach used by Intel's *Hardware Lock Elision* (HLE).

3.2. Conflict Synchronisation

The fallback mechanism detailed in section 3.1 highlights the use of transactional conflict as a form of thread interlock. The same mechanism can be used to implement synchronisation between a number of active threads.

One thread reads from a shared memory word and then spins performing no-op instructions. Other threads can then release the spinning thread by simply writing the shared memory word. Multiple threads could spin on the same word and be released simultaneously.

This mechanism has a great deal of similarity with a classic spin-lock; however, the spinning code is not required to read any memory. Whether the absence of memory operations makes a difference to performance needs to be established through experimentation, but we expect that there would be no observable difference due to existing caching mechanisms. The significant difference with RTM is that multiple memory locations can be monitored simultaneously with only marginally increased setup cost. Furthermore, using gather operations introduced with advanced vector extensions (AVX), also part of the Haswell microarchitecture, upto eight separate cache lines could be loaded simultaneously. This means that synchronisation on upto 512 bytes of memory could be implemented in just five machine instructions.

3.3. Context Switch Detection

As illustrated by the failure rates in section 2.4, operating system interference causes transactional abort. These aborts are detectable by the absence of error flags being set. In turn this means that user code can detect operating system context switches.

This effect could be applied in the implementation of timing critical code. It is worth noting that instructions which read the CPU cycle counter, e.g. `RDTSC`, are valid within a transaction. Thus transactions could be used to ensure accurate micro-benchmarking of instruction sequences. This may be useful during run-time system initialisation.

Broadly speaking operating system interference represents the same level of risk to run-time algorithms as data races with other processors. For example, operating system interference and scheduling is a key factor in why spin-locks are not appropriate for use in user code. Any user level locking primitive must account for operating system scheduling. Hence for algorithms with short and relatively deterministic execution times, RTM could greatly simplify implementation.

3.4. Exception Handling

Hardware exceptions which occur within a transaction are discarded once the transaction has been rolled back. This applies equally to events normally mediated by the operating system such as illegal memory accesses or instructions sequence. For example, if a *null* pointer is accessed within an transaction the transaction will abort, but the operating system will not be invoked. This allows very localised detection of exceptional conditions and mitigates the need for (program global) operating system signal handlers. Additionally, operating system signal propagation requires one or two orders of magnitude more time than a transactional abort.

3.5. Scheduling

Wait-free work-stealing algorithms such as those in the occam-π run-time system have demonstrated efficient and scalable scheduling of CPA workloads [20,13] and have also been used effectively with data parallel computations [21,22]. These algorithms are typically biased toward a work queue's owner, making stealing of work a costly operation. This cost is justified as the processor stealing work is otherwise idle. To ensure wait-freedom (bounded completion) work queues must also be of bounded size. It is also desirable to keep this bound low as it is directly related to maximum number of atomic operations the algorithm must perform.

RTM could be used to relax queue size constraints in common use cases and code paths. Existing operations which are efficiently implemented using compare-and-swap remain unchanged; however, operations which potentially traverse the queue, such as work-stealing, are performed transactionally. This maintains the existing bias as untransactional (local) operations will have priority.

As an extension of the above, a cleanliness (or bias) flag could be explicitly recorded within the work queue. When work is stolen from the queue this flag is updated at the beginning of a transaction to indicate the queue has been modified. The queue owner also tests the cleanliness flag when updating the queue in such a way that transactions conflict and abort. We speculated that using some form of this scheme it may be possible to implement owner updates without atomic operations, assuming all work-stealing is done transactionally.

As an optimisation, work-stealing schedulers often spin looking for work, before falling back to a use of condition variables or operating system primitives waiting for work. Using the method detailed in section 3.2 a transaction could instead be used to monitor work queues. This transaction can be constructed while searching for work to steal. If a monitored work queue is modified then the transaction will abort and the scheduler can again search for work.

3.6. Channel Communication

Previous work has demonstrated how channel communication in CPAs can be implemented efficiently using a single atomic operation per communication [13,23]. Therefore the more heavy-weight RTM does not provide an immediate benefit to implementation of point-to-point communication. However, there are often operations closely related to channel communication which may benefit from transactions or from being encompassed in a transaction that subsumes the communication.

In occam-π a shared channel has its reference copied when communicated. To support this behaviour each shared channel (or channel bundle) has an associated reference count. This reference count must be incremented when the shared channel is communicated. In principle this update could be made into a transaction along with the update of the channel state. While this would not represent any improvement in overhead, as only two atomic operations are involved, it could be of significant benefit when applied to more complex memory and object models than those used for occam-π.

3.7. Barriers

In order to support very large numbers of processes, priority and various forms of processor affinity, barriers within the occam-π run-time have a complex implementation and data structure [24]. We speculate that operations on these structures or the structures themselves could be simplified by use of a transactional model. Such changes would however need to efficiently handle transactional failure and guarantee progress as contention on barriers is common in many agent-based simulations.

3.8. Choice

Choice or alternation is the area of CPAs in which RTM may have the most impact. Here we briefly mention a few possible implementation starting points.

Brown has previously demonstrated the use of general and unrestricted *software transactional memory* (STM) to implement generalised choice over events [18]. This would be one possible route to use of RTM, although allowances would need to be made for the limited size of transactions.

For implementing choice over barriers an oracle mechanism similar to that proposed by Welch [25] could be adapted for transactional use. An oracle data structure common to multiple threads would be transactionally manipulated. This strategy could be particularly effective when compared to a lock-based implementation of the same data structure. This implementation might also be simpler than generalised alternation as proposed by Lowe [26].

The existing wait-free implementation of alternation for occam-π by Ritson, Sampson and Barnes [13] could be placed within a transaction. Consideration would need to be given to the relationship between the number of channels in the alternation and maximum transaction size. Additionally, the occam-π compiler does not guarantee that channels will not share cache lines and thus false sharing of channel cache lines could cause excessive transaction failures. If channels did occupy separate cache lines then in principle the transaction could only conflict as many times as there are channels involved, as each conflict represents a process committing to communication.

An alternative to the above strategy for occam-π would be to use a pre-enabling step as proposed by Barnes [27]. A transaction is initiated and channels are tested to see if there are waiting processes. This is done in priority order such that higher priority channels are added to the read set of the transaction first. If a channel is ready then communication is initiated with that channel, this can complete within the transaction. If no channels are ready then standard enabling and alternation wait is initiated using existing algorithms. The effect of this is that a channel being made ready by another process during the transaction will cause the transaction to abort and be restarted. This accelerates alternation over sets of channels where processes are already waiting. A lengthy enabling and disabling sequence is avoided and this method is multi-processor safe unlike Barnes original method.

4. Conclusions and Future Work

In this preliminary exploration we have establish baseline performance indicators for Intel's *Restricted Transactional Memory* (RTM) extensions as implemented on their Haswell microarchitecture. These performance indicators suggest that RTM is well suited for use on both relatively small (three memory words) and relatively large (upto 16KiB) transactions. RTM is available in the majority of 4th generation Core i7 and Core i5 processors. Thus will likely be widely available in laptop, desktop and server computers within two to three years.

Going forward further experimental work is required to explore the application of RTM to existing and emerging CPA run-time systems and languages. Additionally there are number of RTM characteristics which require further exploration:

- *Contention performance*: our results only evaluated a very small class of contention scenarios. A more intensive assessment is required. In particular, the observed bias to particular processor cores should be further explored.
- *Failure semantics*: our results (section 2.4) suggest that either the processor may not always accurately report the cause of a transactional abort, or that that there are other factors in addition to L1 cache size which affect the maximum size of a transaction.

- *Atomic operations within transactions*: instructions prefixed with `LOCK` are valid within a transaction, but we have not assessed their performance. It may be that processor implementations ignore the `LOCK` prefix, safely accelerating existing algorithms.
- *Nested transactions*: we have not tested nested transactions or established the nesting limit supported by the Haswell micro-architecture.
- *Performance counters*: our measurements do not include data from processor performance counters. These could be used to provide enhanced data on transactional failure causes and the impact of transactions on L1 and L2 caches.
- *Scheduler quantum*: an obvious experiment would be to change the operating system scheduling quantum and observe its impact on unknown transactional failures.
- *Transactional spin-locks*: our conflict based synchronisation mechanism as proposed in section 3.2 should be benchmarked to establish its applicability to real-world code.

Acknowledgements

This work was made possible by support from the EPSRC-funded MirrorGC project (EP/H026975/1) [2], and the University of Kent's Faculty of Sciences' research fund.

References

[1] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2013.
[2] Intel. *Intel® Architecture Instruction Set Extensions Programming Reference*, February 2012.
[3] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, King's College, September 2003.
[4] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
[5] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009.
[6] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M.M.K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Computer Aided Verification, 24th International Conference, CAV 2012*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, July 2012.
[7] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.
[8] B. Sufrin. Communicating Scala Objects. In *Communicating Process Architectures 2008*, pages 35–54, sep 2008.
[9] J.B. Pedersen and B. Kauke. Resumable Java Bytecode - Process Mobility for ProcessJ targeting the JVM. In *Communicating Process Architectures 2009*, pages 159–172, November 2009.
[10] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.
[11] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.
[12] 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.
[13] C.G. Ritson, A.T. Sampson, and F.R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727 – 740, 2012.
[14] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, March 2012.
[15] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGPLAN Not.*, 44(3):157–168, March 2009.

[2] `http://www.cs.kent.ac.uk/projects/gc/mirrorgc/`

[16] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[17] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006.

[18] N.C.C. Brown. *Communicating Haskell Processes*. PhD thesis, University of Kent at Canterbury, 2011.

[19] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[20] C.G. Ritson, A.T. Sampson, and F.R.M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[21] R.D. Blumofe, C.F. Jöerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[22] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 536–545, September 2008.

[23] K. Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent, December 1998.

[24] C.G. Ritson. *Scalable Support for Process-Oriented Programming*. PhD thesis, University of Kent, July 2013.

[25] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B.H.C. Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, March 2010.

[26] G. Lowe. Implementing Generalised Alt. In *Communicating Process Architectures 2011*, pages 1–34, June 2011.

[27] F.R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.