# Transactional Memory and the Birthday Paradox

Craig Zilles
University of Illinois at Urbana-Champaign
zilles@cs.uiuc.edu

Ravi Rajwar
Intel Corporation
ravi.rajwar@intel.com

**Abstract**

Transactional Memory (TM) has been proposed as an alternative implementation of mutual exclusion that avoids many of the drawbacks of locks (*e.g.*, deadlock, reliance on the programmer to associate shared data with locks, priority inversion, and failures of threads while holding locks). TM enables the programmer to denote atomic regions (transactions) that are executed optimistically; if a conflict is detected, the thread is rolled back to the beginning of the transaction. A Software Transactional Memory (STM) implements speculative execution (so that a roll back can be performed) and conflict detection by adding additional code to the application.

In an STM that tracks mutual exclusion at a word or cache-line granularity, an ownership table is used to keep track of which transactions currently have read and write permissions to which regions of memory. Memory addresses are mapped to ownership table entries by hashing the memory address. A frequently proposed design for ownership tables (used in previous STM and hybrid hardware/software TM proposals) is that of a *tagless* table, where read and write permissions are granted at the granularity of all addresses that map to a given ownership table entry. When two transactions have memory accesses that map to the same ownership table entry, the tagless nature of this organization requires the STM to conservatively force one transaction to abort if one or more of the aliasing accesses is a write.

Using address traces from a multithreaded program, we demonstrate that the frequency of these false conflicts grows superlinearly with both the TM data footprint and concurrency and that increasing the size of the ownership table results in only a sub-linear reduction in conflict rate. These somewhat surprising relationships have a theoretical foundation that is also responsible for the (naively) unintuitive statistical result generally referred to as the "Birthday Paradox": that in a collection of at least 23 randomly chosen people, the probability is more than 50% that at least two of them will have the same birthday. In layman's terms, two addresses are likely to map to the ownership table entry long before the table is full. We present an analytical model based on random population of an ownership table by concurrently executing transactions that correctly predicts the trends in measured data.

From this study, we conclude that tagless ownership tables are not a robust approach to supporting transactional memories. Even large tables ($\geq$ 64K entries) are only somewhat effective at mitigating false conflicts in the presence of modest sized transactions (*e.g.*, 20 cache blocks) and modest degrees of concurrency (*e.g.*, 4 simultaneous transactions). In contrast, tagged ownership tables, which record addresses and use chaining to handle aliasing, do not result in false conflicts and, when appropriately sized, only infrequently require chaining. This result is particularly important in the context of hybrid TMs, where the small transactions are likely handled in hardware, leaving only the large ones for the STM.

**Keywords**: Transactional Memory, Concurrency, Birthdays.

# 1 Introduction

Locks are the dominant primitive for implementing mutual exclusion, but they have a number of well documented problems [21]: they do not compose, they have a possibility for deadlock, they rely on programmer convention, and they represent a trade-off between simplicity of programming and concurrency. Transactional Memory (TM) [10] has been identified as a potential alternative to locks by providing an efficient implementation of *atomic blocks* [14], code regions that must (appear to) not be interleaved with other execution. Atomic blocks, or *transactions* as the recent literature calls them, simplify concurrent programming because, while the programmer must still identify *critical sections* (where shared state is not consistent), the programmer does not need to specify an explicit synchronization variable; that responsibility is performed by the TM system.

TM is implemented using mechanisms of: 1) optimistic execution of the transaction with the ability to rollback updates at any point, 2) detection of data conflicts (read-write or write-write) between transactions, and 3) atomic commit of a transaction. Transactional Memory was originally proposed as a hardware implementation [10]. Hardware implementations of transactional memory (HTMs) typically use the processor's data cache(s) to track which data an atomic block has read and to hold speculative data, use the existing cache coherence protocol to detect conflicts, and use "mass clear" operations to perform an in-cache commit atomically. While such proposals provide almost full-speed execution of transactions, their implementations are only straightforward for supporting transactions that fit within the processor's cache. Software implementations of transactional memory (STMs) [1, 5, 6, 7, 8, 9, 15, 18, 19, 20] use in-memory data structures (also referred to as metadata) to hold speculative data values, track ownership and detect conflicts, and commit using an atomic update to a single transaction state memory location associated with all of a transaction's writes. While such approaches can theoretically support transactions of arbitrary size (only limited by available) memory, they incur an overhead resulting from additional software operations and memory allocation.

To combine the performance of HTMs with the flexibility of STMs, numerous hybrid proposals have emerged where a hardware transactional memory is used for the common case where a transaction fits in the local caches and software support is invoked for cases where a transaction exceeds local buffering. This support may involve invoking hardware machinery to operate on software data structures [2, 16, 17] or falling back on to an STM implementation [4, 12].

Whenever a software data structure is used as part of a TM system (be it an STM or a form of hybrid TM), the metadata organization becomes central to the TM system. This metadata is, among other things, used to track ownership and detect conflicts. The metadata organization is closely related to the granularity at which ownership and conflicts are tracked. Object-based designs, generally found in object-oriented languages, track conflicts at the granularity of objects. The language allocates a field within each object (often by modifying the object layout) that is used by the STM for tracking readers and writers to that object. Word-based STMs track ownership at the granularity of fixed-sized chunks of memory, typically either individual words (*i.e.*, 64-bit on a 64-bit architecture) or whole cache lines (*e.g.*, 32 bytes). To avoid perturbing data layouts, ownership and conflict detection metadata information is stored separate from the program's data in a data structure called an *ownership table*.

While object-based STMs have some advantages—notably co-locating the ownership information with the object being accessed and preventing aliasing between objects in the ownership table—they are limited to code written in languages where the data layout is opaque to the programmer. In contrast, the word-based approach is broadly applicable. Even in object-oriented applications, word-based approaches will likely be chosen as the "greatest common divisor" in order to provide one platform that can support the native-code interfaces and language runtimes of object-oriented languages and support applications composed from pieces of multiple different languages with different object models. A word-based STM is also a natural choice for hybrid TM systems, since many of the HTM systems track conflicts at a cache line granularity. In addition, word-based STMs do not include a space overhead found in object-based STMs due to the inclusion of per-object memory allocated for the STM's implementation. As a result, this paper considers word-based STMs.

While a more complete description of an ownership table can be found in Section 2.1, this structure is generally implemented as a hash table where a given entry records ownership information relating to the addresses that hash to it. All of the word-based STMs [1, 6, 18, 19] and some hybrid TMs [4] that have been proposed have used a *tagless* implementation of the ownership table. In tagless implementations, if two distinct addresses map to the same entry in the table, they are treated as a data conflict *even though there isn't a true conflict*. While such a design is simple and eliminates the overhead of storing a tag, it results in false conflicts when otherwise independent data accesses alias in the ownership table.

This paper's contributions are as follows.

- We demonstrate using experimental (Section 2.2) and analytical (Section 3) models that false conflicts in a tagless design seriously impact actual concurrency and undermine the TM motivation itself. The rate of these alias-induced conflicts grows superlinearly with both concurrency and footprint of the transactions and the frequency of conflicts cannot be effectively mitigated by any reasonable scaling of the ownership table. Therefore, STM and hybrid TM implementations that aim to only reduce metadata management overhead by resorting to simple tagless ownership tables will end up limiting scalability and concurrency, a counter-intuitive result.
- We characterize the typical maximum size transactions an HTM in a hybrid TM implementation can sustain. We show that, even with a modest 32KB data cache, an HTM can frequently support transactions up to 30,000 instructions (Section 2.3), fairly large as compared to what is commonly believed. However, an implication of this is that if the ownership table is not properly designed, the concurrency for overflowed transactions would be 1.
- By validating our analytical model against the experimental data (Section 4), we demonstrate that the superlinear relationship between conflict rate and both transaction data footprint and concurrency can be explained in a straight-forward manner by building an analytical model of concurrent transactions updating an ownership table.
- The absence of actual TM workloads and benchmarks makes the design decisions behind TM implementations difficult. We show that using simple analytical models can help eliminate metadata designs that actually limit concurrency, even if they appear to be low overhead.

The obvious alternative to a tagless organization is one that maintains tags. In Section 5, we briefly overview a tagged organization of an ownership table that mitigates the perceived overheads of storing tags and supporting chaining in expected case of infrequent aliasing. Finally, we conclude in Section 6.

# 2 Metadata organization in a tagless ownership table

In this section, we describe a typical ownership table's structure and usage in its standard tagless formulation (Section 2.1). We then demonstrate that the rate of false conflicts resulting from this tagless organization grow proportionally to the square of the transaction's data footprint size, superlinearly with concurrency, and are only somewhat mitigated by increasing ownership table size (Section 2.2). Finally, we present data on the average maximum size of transactions that a hybrid TM can likely implement in HTM-mode, demonstrating that the hybrid TM's STM should be designed to support transactions with data footprints starting at 200 cache blocks (Section 2.3).

## 2.1 Organization of a tag-less ownership table

Word-based STMs use a centralized ownership table that permits the detection of conflicts between transactions. To detect conflicts, the ownership table tracks the read and write footprints of each transaction. Even STM implementations that do not visibly track readers would need to assign an ownership table entry for the read location to record version numbers. As conflicts occur when two transactions access a data item and at least one of the accesses is a write, we need to record the addresses of the data items accessed and whether the

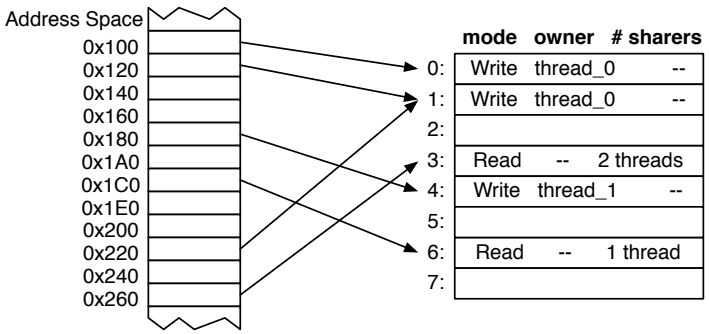


Figure 1: **A previously proposed organization for a tagless ownership table.** We have shown a mapping at the granularity of 32-byte cache blocks.

accesses to that address were read-only or if it included writes. In addition, each thread executing transactions maintains a (private) per-thread *log* that tracks the state of the transaction (*e.g.*, *active*, *committed*) and the transaction's footprint including speculative values for writes.

Figure 1 shows a proposed organization for an ownership table common to many STM proposals [1, 4, 6, 18, 19]. It consists of a table of entries, each with two fields. The first field is the type of entry: *Read*, *Write*, or *Free*. The second field holds either an *owner* that identifies the one thread writing the entry (for *Write*s) or a count the *number of sharers* for the entry (for *Read*s). Program data is mapped to ownership table entries by hashing the (virtual) address.

Because the ownership table will, in general, be much smaller than a program's data memory footprint, this mapping process results in multiple data items mapping to the same entry of the ownership table. For example, in Figure 1, cache blocks at both address 0x120 and 0x220 map to ownership table entry 1. Because the table is *tagless*—the particular address being accessed is not recorded in the ownership table—entry 1 provides `thread 0` exclusive access to both blocks 0x120 and 0x220. Tagless tables are chosen for their relative simplicity and space efficiency; no tag needs to be stored, nor tag comparison performed, and the hassles of maintaining a linked list for chaining is avoided.

However, this tagless design has the significant drawback of creating false conflicts. Without tags, any two accesses (involving a write) from distinct transactions must be conservatively considered a conflict. Due to the all-or-nothing nature of transactions, a single conflict forces a transaction to either abort or stall until the conflicting transaction commits. Either way, these false conflicts can reduce the achieved concurrency. In fact, in Damron *et al.*'s presented results, performance for their Berkeley DB lock subsystem benchmark actually *decreases* when scaling from 32 to 48 processors due to hash collisions in the ownership table [4]. Harris and Fraser observed aliasing even with microbenchmarks that accessed only 1 or 2 data items in a transaction [6].

## 2.2 Aliasing likelihood in an STM

In an STM, the ownership table is used for tracking typically all memory accesses in a transaction. In this section, we empirically study the relationship between concurrency and the likelihood of an aliasing-induced conflict. For this study, we perform experiments using synthetic transactions (using a real application), primarily due to the lack of available applications that use transactional memory. We collected address traces from a 4-processor (4-warehouse) execution of the SPECJBB2005 multithreaded benchmark. Using these traces, we populate an ownership table (with $N$ entries) using $C$ concurrent address streams until each stream has written to $W$ cache blocks. As we consume these traces, we remove any true conflicts so we can focus on the aliasing-induced conflicts found in real address streams.

We exhaustively evaluate the space spanned by $N = [1k, 4k, 16k, 64k, 256k]$, $C = [2, 3, 4]$ and $W = [5, 10, 20, 40, 80]$; for each data point, we run roughly 10,000 trace samples to compute a likelihood of an alias occurring before all traces complete $W$ writes.

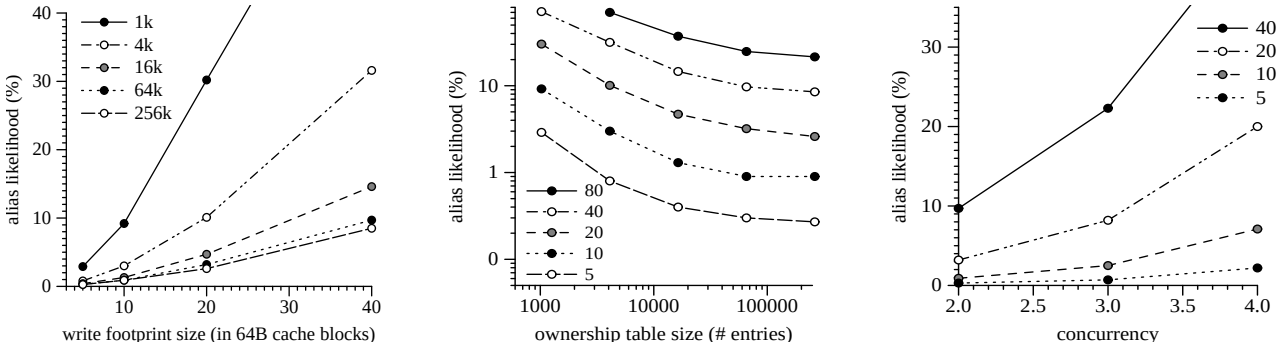


Figure 2: **Aliasing likelihood as a function of data footprint (a), ownership table size (b), and concurrency (c).** (a,b) data shown for concurrency $C = 2$ for data footprints $[5, 10, 20, 40, 80]$ and ownership table sizes $[1k, 4k, 16k, 64k, 256k]$ entries. (c) data shown for footprint sizes $[5, 10, 20, 40]$ and an ownership table of size $N = 64k$.

Figure 2(a) shows the likelihood of an alias-induced conflict as a function of the number of cache lines written. The relationship is clearly superlinear and, for configurations with modest conflict rates (*e.g.*, $<$ 20%) the conflict rate is almost proportional to the square of the size of the write footprint[1]. Similarly, Figure 2(b) shows the conflict rate as a function of the ownership table size. Initially, the rate is slightly less than inversely proportional (*i.e.*., a 4-fold increase in table size yields a 3-fold reduction in alias likelihood) but the effectiveness of further reductions diminishes as the alias likelihood approaches an asymptote at very large table sizes. Finally, Figure 2(c) plots the conflict rate as a function of concurrency. A strong superlinearity is evident here as well, with a concurrency of 4 having an almost 6-fold larger conflict rate than a concurrency of 2.

## 2.3 Characterizing overflows in a hybrid TM

In a hybrid TM system, the ownership table is typically used only when a transaction cannot fit in local hardware buffers. In such situations, the execution falls back to an implementation that uses a software-managed ownership table. Therefore, even for hybrid TMs, the organization of this table is critical. Because HTMs in a hybrid TM uniquely use the data itself for conflict checking (by using the coherence protocol), the HTMs do not suffer from false conflicts (except due to the second order effect of false sharing).

We characterize the average minimum size of a transaction that can be handled by a hybrid TM's STM. This data will be used in Section 3 to reason about the implications of our analytical results (an order-of-magnitude estimate is sufficient for our purposes). Because a hybrid TM primarily invokes its STM component for transactions that exceed its HTM's finite buffering, the average *minimum* size of a hybrid TM's STM transaction will correspond directly to the average *maximum* size of a hybrid TM's HTM transaction. Thus, we characterize the average maximum transaction footprint that fits in a conventional processor cache.

We consider a 32 KB 4-way set associative cache with 64-byte cache lines. This is representative of L1 data caches of contemporary microprocessor implementations. The data cache is used to track a transaction's read and write sets when executing in HTM mode.

The lack of available applications that use transactional memory forces us to use synthetic transactions from real applications. We collect data about memory behavior of existing sequential applications. This approach should not impact our findings because our goal is not to characterize the size of transactions in actual TM-parallelized application, but rather to characterize the transactions at the transition between a hybrid TM's HTM and STM. We expect that parallelizing sequential applications without completely re-writing them would only modestly change their memory access behavior. We extract traces synthetically representing

[1]Previous work on database transactions noted a similar quadratic relationship for true conflicts and the size of a transaction's data footprint [22].

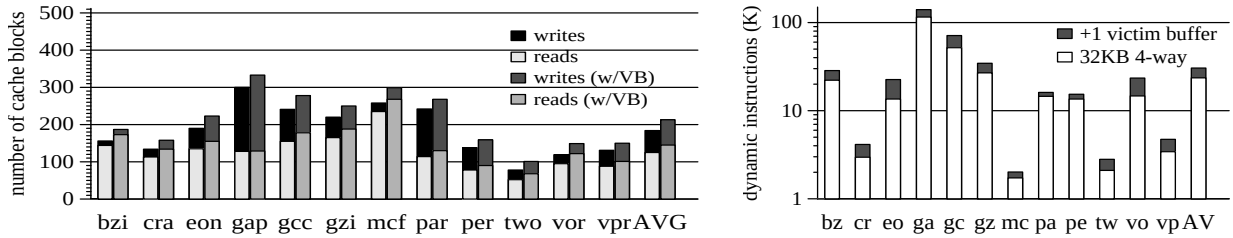


Figure 3: **Average maximum footprint size (a) and dynamic instruction count (b) for transactions overflowing a 4-way set associative 32KB cache with 64-byte blocks.** a) Typically overflow occurs when 2/5ths of the cache's 512 blocks have been touched. A single entry victim buffer results in a 16% increase in hardware supported transaction footprint size. b) Size (in thousands of dynamic instructions) of transactions at the time of overflow.

transactions from sequential applications and execute each trace on a cache simulator to identify the point at which an eviction of a data item touched by the trace occurs. This would correspond to the point where an HTM would overflow its resources.

We use the SPEC2000 integer benchmarks compiled for 64-bit Alpha ISA, with full compiler optimizations but without profile feedback. For each benchmark, we collected traces from each of the reference inputs, collecting at least 20 traces from at least two randomly selected checkpoints per benchmark. The data plotted is a simple arithmetic mean of the data collected for each benchmark.

We find, in general, that the HTM overflows long before its cache is full. As shown in Figure 3(a), overflow occurs on average when only 36% of the cache's 512 blocks are part of the footprint, with the region having executed over 23,000 dynamic instructions (Figure 3(b)). At the point of overflow, about one-third of the footprint is blocks that have been written by the transaction and the other two-thirds have only been read by it. The overflow occurs because of set conflicts: the transaction accesses a fifth block that maps to one of its 4-way set associative sets. Unsurprisingly, the impact of the limited associativity in these hot sets of the cache can be mitigated through the addition of victim buffers [11]. Even the addition of a single victim buffer provides a 16% increase in the utilization of the cache (from 36% to almost 42%) and an almost 30% increase in dynamic instruction count, suggesting victim buffers are a cost-effective approach for supporting larger transactions entirely in hardware.

While there is significant variability between the benchmarks, some general conclusions can be drawn. From these results, it is reasonable to expect that the STM component of a hybrid TM will typically be required to handle transactions consisting of (on the order of) a few hundred cache blocks and the ratio of read-only blocks to written blocks is around 2-to-1.

# 3 Analysis

In this section, we demonstrate that observed trends in the data presented in Section 2.2 can be explained through statistical analysis of the likelihood of collisions occurring from acquiring read and write ownership of random entries of a sparsely-populated ownership table. This effect strongly relates to the so-called "Birthday Paradox," which predicts that the likelihood of two people sharing a birthday is greater than 50% in an unintuitively small number of people (23). We compute closed formed solutions for the relationships between the conflict rate and transaction size, ownership table size, and concurrency by developing an analytical model. This model is developed in two steps: first we assume concurrency is two (Section 3.1) and then generalize to arbitrary concurrency (Section 3.2). In the section that follows, we validate these results (and the necessary simplifying assumptions that enable them) through statistical and trace-based simulations.

### 3.1 Result: conflict likelihood $\propto$ (transaction footprint)$^2$

Our first result is that the likelihood of an alias conflict grows quadratically with the transaction's footprint but is reduced only linearly with the inverse of the size of the ownership table (Equation 1). This means that to maintain the likelihood of a alias conflict below a given threshold, the ownership table must grow as quadratically with the size of the transaction footprints.

$$conflict\_likelihood \propto \frac{region\_footprint^2}{ownership\_table\_size} \tag{1}$$

For this analysis, we have simplified transaction execution in the following ways.

1. there are no true conflicts between transactions.
2. the cache blocks accessed by a transaction have addresses that are equally likely to map (using a hash function) to any of the $N$ ownership table entries. We discuss the implications of this simplification in Section 4.
3. ratio of cache blocks reads-to-writes is a constant, $\alpha$, such that $\alpha$ new cache block reads proceed every additional cache block written (*e.g.*, if $\alpha = 2$, a transaction consists of the repeating sequence [$read$ $read$ $write$]*).
4. the concurrent transactions begin at the same time and proceed in lock step, such that, at any given time, every transaction has read the same number of cache blocks ($R$) and written the same number of cache blocks ($W$).
5. the ownership table is sufficiently sparsely populated that aliasing within a transaction's footprint is negligible; that is, $R + W$ is a good approximation of the total footprint of a transaction that has read $R$ and written $W$ cache blocks. We validate this approximation below.
6. we are primarily concerned with ownership tables where the likelihood of aliasing is relatively small, making it permissible to consider each instance of potential aliasing as being independent from the others[2].

A transaction can only commit if it reaches its end without a conflict. While there are no true conflicts, false conflicts occur if addresses from two transactions map to the same entry of the ownership table and at least one of them is a write. If we first consider the case where only two transactions ($A$ and $B$) are concurrently executing, we find that:

1. when a read is performed to a new cache block, its likelihood of causing a conflict is proportional to the fraction of the ownership table containing write entries belonging to the other transaction: a W/N chance.
2. when a write is performed to a new cache block, its likelihood of causing a conflict is proportional to the fraction of the ownership table containing read *or* write entries belonging to the other transaction: a (R+W)/N chance.

Given that $R = \alpha W$, we can compute the likelihood that a conflict occurs every time transaction $A$ reads $\alpha$ new cache blocks and writes one new cache block, in terms of the current footprints of transaction $B$:

$$\Delta conflict\_likelihood(W_B) = \alpha\frac{(W_B - 1)}{N} + \frac{(\alpha + 1)W_B}{N} = \frac{(1 + 2\alpha)W_B - \alpha}{N} \tag{2}$$

where the first term accounts for the incremental likelihood of a conflict due to $\alpha$ new cache block reads (the $-1$ term of $W_B - 1$ indicates the reads are performed before the corresponding write); the second term accounts for $A$'s new cache block write.

[2]This assumption permits the use of a simpler-to-evaluate sum of probabilities formulation, rather than a product of probabilities, without significant loss of accuracy for the region of interest.

Since both transactions are executed in lock step ($W_A(t) = W_B(t) = W(t)$), there is the corresponding rate for transaction $B$. Thus, as we extend both transactions, the likelihood of a conflict is almost twice the likelihood computed in Equation 2; to compensate for double counting the likelihood of the $n_{th}$ write by $A$ conflicting with the $n_{th}$ write by $B$, we need to subtract out $1/N$. To compute the likelihood of a conflict for transactions of a given size, we can sum the incremental likelihoods for each step:

$$conflict\_likelihood(W) = \sum_{w=1}^{W} \frac{(2+4\alpha)w - 2\alpha - 1}{N} \tag{3}$$

$$conflict\_likelihood(W) = \frac{1}{N}((2+4\alpha)\frac{W}{2}(W+1) - (1+2\alpha)W) = \frac{(1+2\alpha)W^2}{N} \tag{4}$$

The practical implications of this relationship for a hybrid TM can be illustrated by performing a back of the envelope calculation with the empirical parameters found in Section 2.3 for the minimum size STM transactions ($W = 71$, $\alpha = 2$). Solving Equation 4 for $N$ indicates that achieving a commit probability above 50% requires an ownership table with more than 50,000 entries. A 95% commit probability requires a table with over a half million entries.

### 3.2 Result: conflict likelihood $\propto$ (concurrency)$^2$

In the subsection above, we considered the case where the concurrency was exactly two. In this subsection, we demonstrate that likelihood of a conflict grows quadratically with the concurrency (Equation 1), which, alternatively stated, requires the ownership table to grow as the square of the concurrency in order to prevent an increase in the likelihood of a conflict.

$$conflict\_likelihood \propto concurrency^2 \tag{5}$$

Our analysis that leads to this conclusion closely corresponds to the one above, except now with $C$ concurrently executing transactions, adding a cache block to a transaction's footprint potentially conflicts with any of the $C-1$ other transactions. Assuming again that the transactions are progressing in lock step ($W_A(t) = W_B(t) = ... = W_Z(t) = W(t)$), Equation 6 computes the likelihood that a conflict occurs every time a transaction reads $\alpha$ new cache blocks and writes one new cache block (corresponding to Equation 2). Again, we assume that the transactions cover disjoint subsets of the ownership table.

$$\Delta conflict\_likelihood(C, W) = \frac{(C-1)((1+2\alpha)W - \alpha)}{N} \tag{6}$$

With all $C$ transactions progressing concurrently, the likelihood of at least one conflict occurring is found by summing up the conflict likelihoods occurring with each cache block addition for each of the C transactions, as shown in Equation 7. As with Equation 3, we have compensated to prevent double counting. When reduced to Equation 8, we can see the quadratic dependence on the concurrency $C$ (as asserted by Equation 5). Furthermore, if we evaluate for $C = 2$ it reduces to equation Equation 4.

$$conflict\_likelihood(C, W) = \sum_{w=1}^{W} \frac{C(C-1)((1+2\alpha)w - \alpha) - \frac{C}{2}(C-1)}{N} \tag{7}$$

$$conflict\_likelihood(C, W) = \frac{C(C-1)(1+2\alpha)}{2N} \sum_{w=1}^{W} 2w - 1 = \frac{C(C-1)(1+2\alpha)W^2}{2N} \tag{8}$$

The practical implications of Equation 8, again using the empirical parameters found in Section 2.3, suggests that achieving a 95% commit probability with a concurrency of 8 requires a table with over 14 million entries.

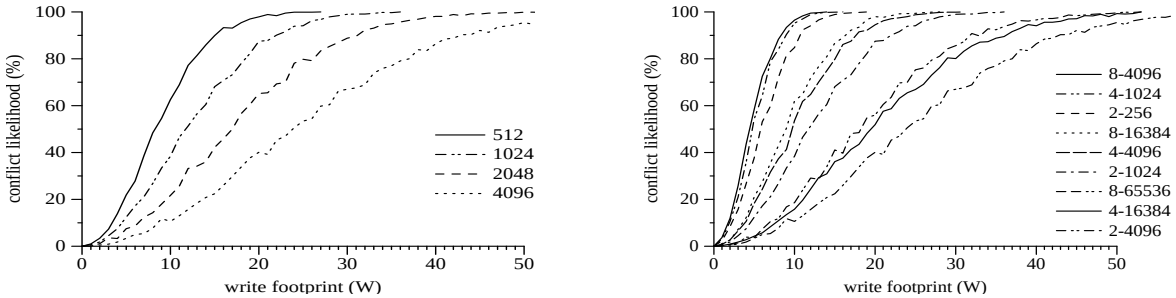


Figure 4: **Validation of the model through statistical simulation.** (a) conflict likelihood scales quadratically with the transaction footprint (x-axis) and inversely with the ownership table size (512-4096 entries). b) conflict likelihood asymptotically scales quadratically with the concurrency, but at low concurrency the linear term of $C(C-1)$ is non-negligible; data plotted for <concurrency, ownership table size> pairs.

# 4 Experimental validation

In this section, we support the conclusions of our analytical results with statistical simulations. In addition, with these simulations we can directly validate a number of assumptions incorporated into the analytical model. Specifically, we performed two sets of simulations: the first that strongly correspond to the analytical model and a second set where we measure transaction throughput in a closed system. In both sets of simulations, we simulate a set number of threads, each executing transactions consisting of a fixed number of cache blocks in the pattern of $\alpha$ reads followed by a single write. These cache blocks are assigned to random entries of the ownership table. We add to the footprints of each transactions in a round-robin manner.

In the first set of simulations, we begin execution of $C$ transactions at the same time and determine whether any conflicts occur before all transactions complete. By performing 1000 experiments for each data point we can compute conflict rates for each transaction footprint size and degree of concurrency. In Figure 4(a), we show the relationship between conflict rate and write footprint size for a concurrency of 2. The quadratic relationship between write footprint and conflict rate is evident and, at small footprints (*e.g.*, 8 writes), the inverse relationship on table size can be seen ($48\% \rightarrow 27\% \rightarrow 14\% \rightarrow 7.7\%$)

Figure 4(b) shows the asymptotically quadratic relationship between concurrency and conflict rate. In the graph there are three distinct clusters of three lines each, where the lines in each cluster represent a quadrupling of ownership table size for each doubling of concurrency. While the clusters are clear, there is some separation between the lines within the cluster, particularly between the $C = 2$ lines and the $C = 4$ and $C = 8$ lines. This separation occurs because conflict rate grows at $C(C-1)$ (Equation 8), which is only asymptotically a quadratic relationship. At lower $C$'s, the conflict rate grows faster than $C^2$, which is reflected in the simulation results.

These simulations allow us to validate the assumption from Section 3.1 that aliasing within a transaction (the number of independent cache blocks from the same transaction that map to the same ownership table entries) is negligible. This assumption is not made in the simulations and we observed the predicted relationships. Furthermore, by measuring the actual frequency that such aliasing occurs, we find that the aliasing rate is below 3% as long as the conflict rate is below 50%.

Our second set of experiments validate another simplification of the analytical model, by considering a closed system where $C$ "threads" attempt to complete as many (fixed-size) transactions in a given amount of time by executing them one after another; when no conflicts occur, our simulations complete 650 transactions. The start times of the threads are randomly staggered and, when conflicts occur, transactions are restarted. As a result, there are no particular relationships between the footprint sizes of the transactions at any given time, relaxing the model's assumption that all regions commence simultaneously.

In Figures 5(a,b) and 6, we plot the number of conflicts that occur as a function of write footprint, table size, and concurrency. As these graphs are plotted on a log-log scale the polynomial relationships between

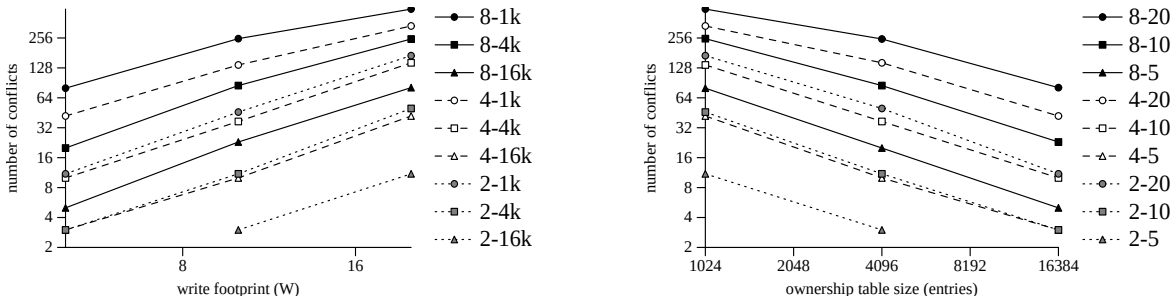


Figure 5: **Closed system simulations.** Data plotted for <concurrency, table size> and <concurrency, write footprint> pairs, respectively.

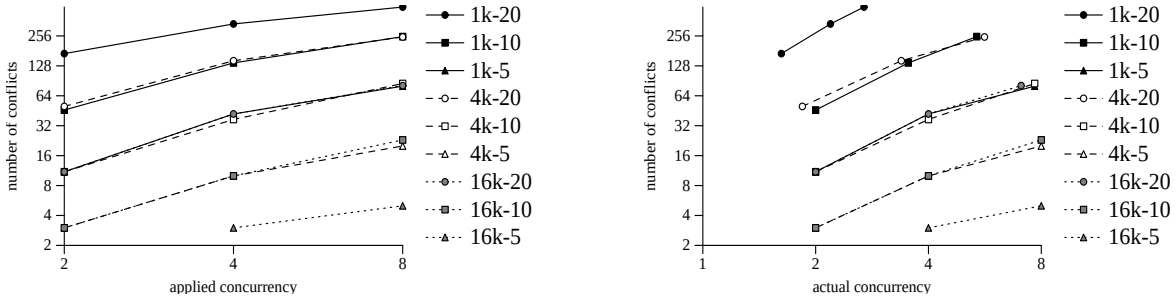


Figure 6: **Closed system simulations for applied and actual concurrency.** a) when the conflict rates are plotted against the applied concurrency (*i.e.*, number of threads) the lines begin to converge at high conflict rates; b) if we compensate for the reduction in concurrency and instead plot against actual concurrency, we recover the expected relationships between footprint, ownership table size and conflicts. Data plotted for <table size, write footprint> pairs.

these variables should appear as straight lines on the plots. With the exception of the concurrency plot (Figure 6(a)), where the linear term is non-negligible at low values of $C$ as described above, we observe the expected relationships in the data—straight lines of the expected slopes, with constant separation—for cases where the conflict rate is modest.

Where there are a high number of conflicts, however, the separation between the lines decreases, which was not predicted by the model. Further investigation revealed that this under-prediction of the conflict rate was due to the fact that when a transaction aborts, its entries are removed from the ownership table. When conflicts are infrequent, we measured the ownership table to have, on average, a number of entries filled corresponding to one-half the concurrency $C$ times the transaction footprint size, as is expected. When the conflict rate is high, the measured average can be quite lower—as much as 40% lower—because the effective concurrency has been reduced by the occurrence of these conflicts. If we compensate for this reduced concurrency (as shown in Figure 6(b)), we recover the expected relationships. This compensation also fixes the relationships in Figures 5(a,b), but there is not a meaningful way to plot the data.

The final unvalidated assumption in the model is that memory addresses that make up each transaction are identically distributed (*i.e.*, the $n$th cache block is equally likely to map to any entry of the ownership table, irrespective of where the previous $n-1$ were placed). Inspection of real memory traces suggest that this is not the case; in particular, it is not uncommon for a program's address trace to contain consecutive memory addresses, which through many hash functions map to consecutive entries of the ownership table). In spite of this, our models almost perfectly predict the relationships between alias likelihood and concurrency and write data footprint in Figures 2(a) and (c). In particular, the factor of six increase in conflict rate when increasing concurrency from 2 to 4 is exactly predicted by Equation 8's $C(C-1)$ term. Our model does not, however, predict the asymptotic behavior with increasing ownership table size seen in Figures 2(b). Understanding and modelling this behavior is part of our future work.

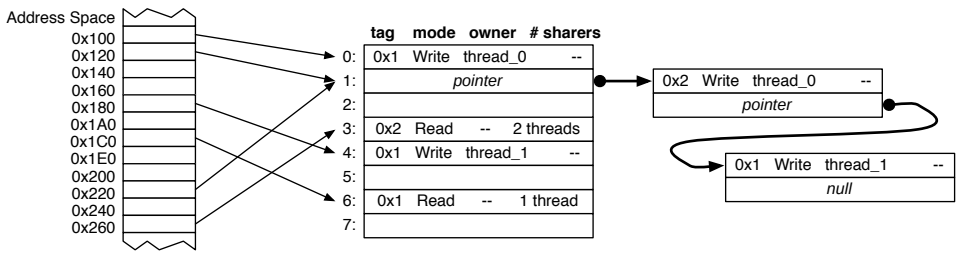


Figure 7: **A tagged ownership table implementation that uses chaining to handle aliases.** Note that the first level table stores either a ownership record *or* a pointer to a list of ownership records. As a result a level of indirection is added only for entries with aliases.

# 5 A tagged ownership organization for eliminating aliasing

To avoid the false conflicts associated with tagless tables, we must consider an ownership table organization that permits the storage of multiple distinct addresses that map to the same ownership table entry, and these entries must store tags so that they can be distinguished. One such structure is a chaining hash table, as shown in Figure 7. While one might presume that tags and chaining pointers might add significant space and execution overhead, this need not actually be the case. For tags, it is only necessary to store the bits not represented in the cache offset or the index into the ownership table (*e.g.*, for a 32-bit architecture using $2^6 =$ 64 byte cache blocks and a $2^{12} = 4096$ entry ownership table, only (32 - 6 - 12) = 14 bits of tag need to be stored). As a result, ownership table entries of the size of an architectural address (*e.g.*, 64-bit entries on a 64-bit architecture) should provide enough space for tag, mode, and sharers.

The overhead of chaining is potentially two-fold: first, the pointers that make up a linked list need to be stored in memory, potentially representing a 100% space overhead, and, second, at execution time these pointers need to be traversed, adding additional latency to access the ownership table. Both of these overheads can be mitigated in the common case by considering the entries in the ownership table as *either* ownership records *or* pointers to linked lists of ownership records. Such a design effectively exploits the fact that although some aliasing is bound to occur, the overwhelming majority of ownership table entries will store 0 or 1 ownership records. For these cases, there is no space overhead nor additional level of indirection to traverse; the only overhead will be an additional conditional branch based on which type of entry is present. With the vast majority of entries having no aliases, these branches should in practice be predictable, resulting in little overhead.

# 6 Conclusion

In this paper, we have demonstrated that a tagless ownership table results in an alias-induced conflict rate that approximately grows as the square of both the concurrency and the size of the transaction's data footprint. This relationship is the unavoidable result of aliasing being likely long before the ownership table being full. As such, tagless tables are not a robust approach. In particular, in the context of a hybrid TM, where the transactions that access the ownership table will be large (those that overflow the cache), a tagless organization will almost guarantee a maximum concurrency of 1 for overflowed transactions.

Finally, up to this point we have considered a weak isolation [3, 13] system where only threads that are actively in transactions must check for conflicts with the ownership table. In contrast, if we consider strong isolation, then even threads outside of isolation regions must perform ownership table look-ups to ensure they are not violating the isolation of a transaction. This additional concurrency makes the use of tagless ownership tables even more untenable.

## References


[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.

[2] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, February 2005.

[3] Colin Blundell, E Christopher Lewis, and Milo M.K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[4] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGOPS Oper. Syst. Rev.*, 40(5):336–346, 2006.

[5] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free , https://sourceforge.net/projects/libltx.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003.

[7] Tim Harris, Simon Marlowe, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[8] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Symposium on Principles of Distributed Computing (PODC)*, 2003.

[10] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[11] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[12] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2006. ACM Press.

[13] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, December 2006.

[14] David Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, pages 128–137, March 1977.

[15] Mark Moir. Transparent support for wait-free transactions. In *11th International Workshop on Distributed Algorithms*, pages 305–319. Sep 1997.

[16] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, February 2006.

[17] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[18] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[19] Nir Shavit, Dave Dice, and Ori Shalev. Transactional locking ii. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[20] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[21] Herb Sutter and James Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, September 2005.

[22] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998.