Implicit Privatization Using Private Transactions

Dave Dice
Sun Labs at Oracle, 1 Network Drive, Burlington, MA 01803-0903 USA
dave.dice@oracle.com

Alexander Matveev
Tel-Aviv University, Tel-Aviv 69978, Israel
matveeva@cs.tau.ac.il

Nir Shavit
Tel-Aviv University, Tel-Aviv 69978, Israel
shanir@cs.tau.ac.il

Abstract

In software transactional memory (STM) systems, it is useful to isolate a memory region accessed by one thread from all others, so that it can then operate on it "privately", that is, without the instrumentation overhead of inter-transactional synchronization. Allowing transactions to *implicitly* privatize memory is a source of major performance degradation in state-of-the-art STMs. The alternative, to explicitly declare and guarantee privacy only when needed, has been argued to be too tricky to be useful for general programming.

This paper proposes *private transactions*, a simple intermediate that combines the ease of use of implicit privatization, with the efficiency that can be obtained from explicitly knowing which regions are private.

We present a new scalable *quiescing algorithm* for implicit privatization using private transactions, applicable to virtually any STM algorithm, including the best performing TL2/LSA-style STMs. The new algorithm delivers virtually unhindered performance at all privatization levels when private transactions involve work, and even under the extreme case of empty private transactions, allows for a scalable "pay as you go" privatization overhead depending on the privatization level.

1. Introduction

One goal of transactional memory algorithms is to allow programmers to use transactions to simplify the parallelization of existing algorithms. A common and useful programming pattern is to isolate a memory segment accessed by some thread, with the intent of making it inaccessible to other threads. This "privatizes" the memory segment, allowing the owner access to it without having to use the costly transactional protocol (for example, a transaction could unlink a node from a transactionally maintained concurrent list in order to operate on it or to free the memory for reallocation.)

Today, many of the best performing lock-based software transactional memory (STM) algorithms [4–6] use a variation of the TL2/LSA [4, 11] style global-clock algorithm using invisible reads. When we say invisible reads, we mean that the STM does not know which, or even how many, readers are accessing a given memory location. Not having to track readers is the key to many efficient STMs, but also a problem if one wishes to allow privatization.

Allowing transactions to implicitly privatize memory is a source of major performance degradation in such STMs (One should note that STMs that use centralized data structures, such as RingSTM [14] or compiler assisted coarse grained locking schemes [10], can provide implicit privatization without the need for explicit visible readers). The alternative, to explicitly declare and guarantee privacy only when needed, has been argued to be too tricky to be useful for general programming.

Why is guaranteeing implicit privatization such a problem? Consider a transaction that has just privatized a memory segment. Even though the segment cannot be accessed by any other transaction (executing on the same or other processor), after the transaction commits, prior to the commit, latent transactional loads and stores might be pending. These latent loads and stores, executed by transactions that accessed the segment before it was isolated, can still read from and write into the shared memory segment that was intended to be isolated. This unexpected behavior is known as the "privatization problem." This results in unexpected changes to the contents of the isolated shared memory (which may have been reallocated and (although resident in the shared memory)) is intended to be outside of the transactionally shared data region. Other unexpected, generally asynchronous, behaviors can also occur.

For example, consider the scenario in Figure 1: an invisible read based transaction by a thread P removes a node form a linked list. Thus, once the transaction completes, the node will no longer be reachable to other threads and P will be able to operate on it privately. However, before P completes its transaction, another transaction Q reads the pointer, and is poised to read a value from the node. P has no way

TRANSACT '10 Date, City.

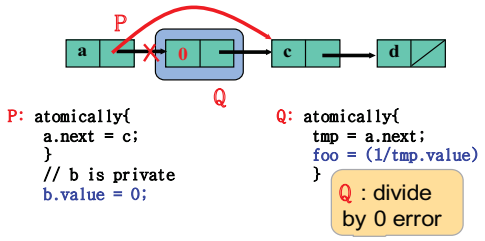

Figure 1. Privatization Pathology Example

of detecting Q. This is because Q reads the pointer invisibly, and will not see any trace of P touching the location in the node since P is operating on it non-transactionally. As a result, even though Q is doomed to fail (once it revalidates the locations it read, and detects the pointer has changed), in the interim it can perform illegal operations. This is an example of a privatization problem that one must overcome.

One solution to the privatization problem is to establish programming constraints against concurrent transactional and non-transactional access to the same set of shared memory locations. However, this is a solution we would like to avoid.

Another solution is to add privatization capabilities to a transactional memory. The transactional memory can employ either "explicit privatization," where the programmer explicitly designates regions passing out of transactional use to be quiesced, waiting for any pending transactional loads and stores to complete before the memory is allowed to be accessed non-transactionally. Programming explicit segment quiescence is complex and error prone. For example, it is insufficient for a transaction to explicitly privatize a segment from the transactionally shared data region before modifying that segment.

Alternately, transactions can employ "implicit privatization," where the STM automatically manages the memory accessibilty/lifecycle issues. A paper by Cheng et al. [15] describes an implicit privatization mechanism that quiesces threads instead of shared memory regions, potentially impacting overall scalability. As we said, the problem with implicit privatization techniques to date, is that they hinder the scalability of most of the best performing STMs.

The current state-of-the-art is thus that there exist several highly scalable STM algorithms that operate well without providing privatization, but do not scale well when implicit privatization capabilities are added (see TL2-IP algorithm in [3] and see [1, 8, 9, 13]). This is the situation we wish to rectify.

This paper proposes *private transactions*, a simple intermediate approach that combines the ease of use of implicit privatization, with the efficiency that can be obtained from explicitly knowing which regions are private. The idea behind private transactions is simple. The user will use implicit privatization to privatize memory segments just as before, but will declare, using a special keyword or keywords, which code segments he/she expects to be executed privately.

From the programmers point of view, a private transaction is thus a declaration of the code elements that are expected to be private: it is the programmers responsibility to make sure that the selected locations within the private transaction are indeed not accessible to successful transactions. It is the STMs responsibility to make sure that unsuccessful transactions do not violate the transactional barrier and access these privatized regions.

We believe private transactions will not add an extra burden to programmers beyond that of implicit privatization because the programmer must anyhow know what he expects to be private following the act of privatization! (This is definitely true for newly written code, and for legacy code in which the programmer is not performing guesswork while, say, replacing locks with transactions). Notice that there is no limitation on the code that can be called within a private transaction, and in particular one can call external libraries that know nothing about the transactions executing in the system.

What do we gain from the private transaction declaration? Our gain is twofold. We remain within the transactional model (i.e. the private transaction is a transaction, not a "barrier" whose semantics and rules of use with respect to other transactions are unclear), and we can algorithmically guarantee efficient execution of the privatized code (it will run at the pace of un-instrumented code), placing only a limited computational burden on regular non-private transactions. In other words, unlike with the standard model of implicit privatization, the privatization overhead using private transactions will have a pay-as-you-go nature: the less privatization, the less overhead.

An important contribution of our paper is a new scalable *quiescing algorithm* for implicit privatization using private transactions, applicable to virtually any STM algorithm, including the TL2/LSA-style STMs. We note that for those who do not wish to use private transactions programming model, this quiescing algorithm can still be used at the end of privatizing transactions to guarantee efficient implicit privatization.

To show the power of the new quiescing algorithm technique, we apply it to the latest version of the TL2 STM. We then compare our new TL2P algorithm, that is, TL2 with private transaction capability, to TL2-IP, that is, TL2 with a known implicit privatization mechanism based on shared reader counters.

As we show, the new algorithm is highly scalable. In a realistic situation in which private transactions include work (see Figure 5, it delivers virtually unhindered performance. In less realistic trying benchmark in which private transactions do not include work, it delivers great performance at low privatization levels, and unlike former techniques, as exemplified by TL2-IP, remains scalable (though not as effi-

cient as TL2) even with 100% privatization. We believe it can be applied to many existing STMs, allowing them to maintain scalability while providing low overhead privatization capabilities.

Interestingly, non-transactional data structures, such as those in the Java concurrency package, suffer from privatization issues. For example, a record removed from a red-black tree cannot be modified without worrying that some other thread is concurrently reading it. Our new private transaction mechanism offers a scalable way to add privatization to such structures.

2. Private Transactions

A *private transaction* is a transaction accessing memory locations that cannot be accessed by any other successful transaction.

The idea behind private transactions is simple. The programmer, using a regular transaction, privatizes certain sections of code by making them inaccessible to any thread that starts executing after the completion of this transaction. The programmer also declares, using the special private transaction keyword or keywords, which code segments he/she expects will be executed privately.

The private transaction is thus a declaration of the code elements that are expected to be private after regular successful transactions have privatized it. It is the STMs responsibility to make sure that unsuccessful transactions do not violate the regular transactional barrier.

Thus, in the classical linked list example, a programmer will use a regular transaction to privatize a linked list node, and then place all code accessing this node within the private transaction, knowing it is no longer accessible. If he/she correctly privatized using a regular transaction, the private transaction semantics will be guaranteed, and otherwise, as with any buggy program, all bets are off.

We believe private transactions will therefore not add an extra burden to programmers beyond that of implicit privatization because the programmer must anyhow know what he expects to be private following the act of privatization! Notice that there is no limitation on the code that can be called within a private transaction, and in particular one can call external libraries that know nothing about the transactions executing in the system.

3. Implementing Private Transactions

Our privatization technique can be added to any existing STM without changing it. The main idea, which we will call a *quiescing barrier*, is well known: track in a low overhead fashion when threads start transactions and when they end them. Using this tracking data, privatization can be provided on demand by waiting for all active transactions to complete before continuing with the execution of a private transaction. However, past attempts to make this type of algorithm scale failed because the mechanisms used to implement the quiescing barrier incurred too large an overhead, and this overhead was exacerbated by the requirement to privatize 100% of the transactions: there was no declaration of when it is actually required.

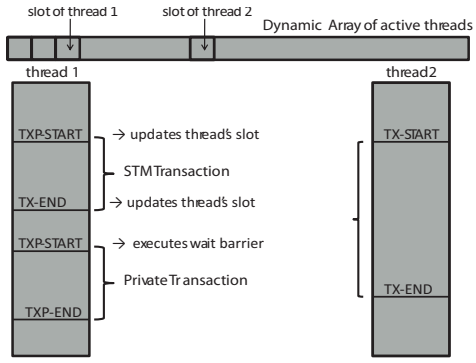

Figure 2. Two threads execute transactions. We see the dynamic array used to track quiescing information and bars tracking the execution phases of the two threads. Upon start and finish, the threads update the dynamic array slot associated with each one of them. When Thread 1 executes a private transaction, it will execute a wait barrier, waiting for Thread 2 because it detects that Thread 2 is in a middle of transaction execution.

Here we combine the use of private transactions with a very low overhead shared array to achieve a lightweight quiescing scheme.

The transactional tracking mechanism, depicted in Figure 2 is implemented as follows. The quiescing barrier needs to know about the threads that are transactionally active. We thus assign every thread a slot in an array which the barrier scans. For this to be efficient, we use an array proportional to the maximal concurrency level, and use a leasing mechanism [7] together with a dynamic resizing capability. We will explain shortly how this is done. A thread will indicate, in its array slot, if it is running an active transaction, and will add an increasing per-thread transaction number. The respective fields are the *IsRun* boolean flag indicating if the thread is executing a transaction, and *TxCount* is a byte size counter which is incremented upon every transaction start.

During a transaction's start:

1. **Map Thread Slot:** The thread id is mapped to an index inside the array that the barrier scans. An explanation will follow shortly.
2. **Increase Thread's Counter:** The *TxCount* counter is increased by one to indicate a new transaction has started.
3. **Update Run Status:** The transactional active status flag *IsRun* is set to TRUE.
4. **Execute a memory barrier:** A write-read memory barrier instruction is executed to ensure that the other threads see that current thread is active.

At the transaction's end:

1. **Update the Run Status:** The status flag *IsRun* is set to FALSE. (No need for a memory barrier).

As can be seen, the operations by any given thread amount, in the common case, to a couple of load and store instructions and a memory barrier.

In more detail, the quiescing barrier records the current run status of the transactionally active threads and waits for completion of each of them. It uses the following 4 array fields of *MAX_THREADS* length. Several arrays are used: an array *th_tx_counts[]* is used to hold the stored counters of the transactionally active threads, *th_ids[]* is used to hold the stored thread ids of the transactionally active threads, *th_checked[]* is used to indicate for which active threads the waiting condition needs to be tracked. Also, we maintain a global variable *CurNumberOfSlots* that holds the current number of assigned slots in the global slots array *th_slots[]* in which every assigned slot has a pointer to an associated *TxContext*.

Using these variables the quiescing barrier algorithm proceeds as follows:

1. **Store the Active Threads Status:** For every transactionally active thread with id *thread_id*, store the thread's *TxCount* and *thread_id* to the waiting thread's context fields *th_tx_counts[thread_id]* and the *th_ids[thread_id]*. Also set the waiting threads's *th_checked* at entry *thread_id* to FALSE, indicating that one needs to wait for this thread's progress.
2. **Wait For Progress:** For every tracked stored thread status which need to be checked for progress, check if the thread is still running and its *TxCount* counter is equal to the stored one. If so we need to wait, therefore start this step again. Return when all the threads we waited for made progress.

The number of threads in an application can be much higher (in the thousands) than the actual concurrency level at any given moment. This is typically determined by the number of hardware threads (in the tens). Therefore, we maintain an array of "leased" slots, proportional to the expected concurrency, to which the transactionally active threads are mapped. A *lease* [7] is a temporary ownership of a lock on a location, one that is either released by a thread or revoked if it is held beyond the specified duration. The allocated array itself can be much larger than the number of active threads, but we keep a counter indicating its actual active size at any point. The scanning thread, the one checking the barrier, need only traverse the array upto its actual active size.

When a thread starts a transaction, it checks if it owns its assigned array slot. If its does, then the thread continues as usual. Otherwise, the thread picks a slot in the array. If the hashed entry is free then the thread takes it, and otherwise it tries to steal the slot. The thread will succeed in stealing the slot only if the slot's lease time or timeout, from the last active run of the thread which owns the slot, has passed. If the timeout has not expired, than the thread tries to acquire another slot. If no slot can be acquired, the thread adds a new slot to the end of the array and assigns itself to it.

In the array, every thread's context *ctx* has a *is_slot_valid* boolean variable indicating if the thread's slot is valid (assigned and not stolen) and a *is_slot_steal_in_process* boolean variable indicating if some thread is trying to steal the current thread's slot.

The *Assert Thread Slot* works as follows:

1. **Check for a Steal:** If some other thread is in the process of stealing the current thread's slot then spin on *is_slot_steal_in_process*.
2. **Check Slot Validity:** Check that *is_slot_valid* is TRUE. If so, return to the caller. Otherwise, continue to the next step.
3. **Register a Slot:** Compute the *slot_number* of the thread by hashing it to its *thread_id*. For example, use *slot_number = thread_id mod number_of_cores*. If the slot with the computed number is free to try to acquire it using a Compare And Swap (CAS) to write into it a pointer to the thread's record *ctx*. If the slot is not free, or the CAS failed, then *Try To Steal Slot*. If the stealing failed, try to steal any other assigned slot. If all the stealing attempts failed, allocate a new slot in the dynamic array.

The *Try To Steal Slot* procedure checks for the slot timeout and if it has expired acquires the slot. *Try To Steal Slot* procedure works as follows:

1. **Check For a Timeout:** Check if the slot's owning thread timeout from last active transaction has expired. If not return failure.
2. **Check For a Steal:** Check if some other thread is already trying to steal that slot by looking at the *is_slot_steal_in_process* value of the slot's owning thread. If it is, return failure. Otherwise, try to CAS the *is_slot_steal_in_process* field to TRUE. If the CAS fails, return failure, and otherwise continue to the next step.
3. **Validate the Slot Status:** Check that the slot owner has not changed and check the timeout expiration again. If all checks are positive - continue to next step, and otherwise return failure.
4. **Steal the Slot:** Assign the slot's value to be a pointer to the stealing thread's context and set its *is_slot_valid* to TRUE. In addition, set the previous owning thread's *is_slot_valid* and *is_slot_steal_in_process* flag to FALSE.

Allocation of a new slot is done when all the steals failed. The procedure works as follows:

1. **Allocate a new Slot:** Increment the *CurNumberOfSlots* global limit variable using a CAS.

2. **Initialize the Allocated Slot:** Assign the slot's value to be a pointer to the thread's context and set its *is_slot_valid* to TRUE.

In order to garbage collect the expired slots, periodically execute a maintenance operation which checks for expired slots and frees them. This same operation can adjust the the CurNumberOfSlots according to the actual number of slots with unexpired leases.

The main purpose of this complex dynamic slot allocation algorithm is to avoid scanning an array proportional to the number of threads in the system, and instead scan only those which are transactionally active.

As we show, the complete mechanism is lightweight and delivers scalable performance.

The end result of this algorithm is the un-instrumented execution of privatized code, with no limitation on code that can be called within a private transaction: in particular one can call external libraries that know nothing about the transactions executing in the system.

4. Outline of correctness

For lack of space we do not discuss private transaction semantics and only briefly outline why our algorithm is correct. In a nutshell, each private transaction is preceded by a traversal through the privatization barrier, recording all active transactions. We assume all private transaction memory regions are not accessible to successful transactions. Thus, by waiting till all active transactions have completed, and given that private locations can no longer be reached by newly started transactions, privacy is guaranteed.

5. Empirical Performance Evaluation

Many of the scalable lock-based STM algorithms in the literature use a TL2 style locking and global clock based scheme, differing perhaps in details such as the order of lock acquisition and the abort and retry policies [4–6, 11, 12]. Most of these algorithms do not support privatization because of its high overhead. We will therefore provide an evaluation of our new privatization algorithm by adding it to the most efficient know version of the TL2 algorithm, one using a GV6 clock scheme citeTL2. We call this new version of TL2 supporting private transactions *TL2P*.

We would have loved to provide a comparison of TL2P to the STM of Saha et. al [10] that provides privatization via a global transactional quiescing mechanism, but unfortunately it is only available using the author's specific STM compiler framework which cannot be applied to our algorithm.

This section therefore presents a comparison of the vanilla TL2 algorithm with a GV6 counter, the TL2-IP algorithm that provides implicit privatization for TL2, and our new TL2P algorithm supporting implicit privatization with private transactions. The microbenchmarks include the (now standard) concurrent red-black tree structure and a randomized work-distribution benchmark.

The red-black tree was derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al [2]. We would have preferred to use the exact Fraser-Harris red-black tree but that code was written to to their specific transactional interface and could not readily be converted to a simple form.

The red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. The put operation installs a key-value pair. If the key is not present in the data structure put will put a new element describing the key-value pair. If the key is already present in the data structure put will simply insert the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls put, get and delete to operate on the underlying data structure. The harness allows for the proportion of put, get and delete operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range of 2K elements generates a small size tree while the range of 20K elements creates a large tree, implying a larger transaction size for the set operations. We report the aggregate number of successful transactions completed in the measurement interval, which in our case is 10 seconds.

In the random-array benchmark each worker thread loops, accessing random locations. The transaction length can be a constant or variable. While overly simplistic we believe our random access benchmark captures critical locality of reference properties found in actual programs. We report the aggregate number of successful transactions completed in the measurement interval, which in our case is 10 seconds.

For our experiments we used a 64-way Sun UltraSPARC® T2 multicore machine running Solaris™ 10. This is a machine with 8 cores that multiplex 8 hardware threads each.

In our benchmarks we "transactified" the data structures by hand: explicitly adding transactional load and store operators. Ultimately we believe that compilers should perform this transformation. We did so since our goal is to explore the mechanisms and performance of the underlying transactional infrastructure, not the language-level expression of "atomic." Our benchmarked algorithms included:

TL2 The transactional locking algorithm of [4] using the GV4 global clock algorithm that attempts to update the shared clock in every transaction, but only once: even if the CAS fails, it continues on to validate and commit. We use the latest version of TL2 which (through several code optimizations, as opposed to algorithmic changes) has about 25% better single threaded latency than the

version used in in [4]. This algorithm is representative of a class of high performance lock-based algorithms such as [6, 12, 16].

TL2-IP A version of TL2 with an added mechanism to provide implicit privatization. Our scheme, which we discovered independently in 2007 [3], was also discovered by Marathe et al. [8] who in turn attribute the idea to Detlefs et al. It works by using a simplistic GV1 global clock advanced with CAS [4] before the validation of the read-set. We also add a new *egress* global variable, whose value "chases" the clock in the manner of a ticket lock. We opted to use GV1 so we could leverage the global clock as the incoming side of a ticket lock. In the transactional load operator each thread keeps track of the most recent GV (global clock) value that it observed, and if it changed since the last load, we refresh the thread local value and revalidate the read-set. That introduces a validation cost that is in the worst case quadratic. These two changes – serializing egress from the commit – and revalidation are sufficient to give TL2 implicit privatization. These changes solve both halves of the implicit privatization problem, the 1st half being the window in commit where a thread has acquired write locks, validated its read-set, but some other transaction races past and writes to a location in the 1st thread's read-set, privatizing a region to which the 1st thread is about to write into. Serializing egress solves that problem. The 2nd half of the serialization problem is that one can end up with zombie reader transactions if a thread reads some variable and then accesses a region contingent or dependent on that variable, but some other thread stores into that variable, privatizing the region. Revalidating the read-set avoids that problem by forcing the 1st thread to discover the update and causing it to self-abort.

TL2P This is the same TL2 algorithm without any internal changes, to it to which we added the private transaction support mechanism.

5.1 Red-Black Tree Benchmark

In the red-black tree benchmark, we varied the fraction of transactions with privatization. In the top two graphs in Figure 3, private transactions involve no computation, stressing the quiescing mechanism. We can see that under these extreme circumstances, in all the cases, unlike TL2-IP, the TL2P scheme is scalable at all levels of privatization. This is quite surprising because one might think that as more threads run one needs to scan more entries in the dynamic array when performing the private transaction. But as can be seen d=from the graphs, it does not impose a significant overhead on the quiescence mechanism. The TL2P algorithm with 20% mutations pays a maximum penalty of 15% for 10% privatization case. 35% for 50% privatization and 50% for 100% privatization. With 4% of mutations (not shown in the graphs), perhaps a realistic level of mutation on a search structure, the maximum performance penalty in TL2P for 100% privatization, which is like implicit privatization, is no more than 25%. And if the privatization is only partial, say 10%, the penalty is just 11%! In general we can see that the TL2P algorithm with no private transaction usage is a little lower than TL2. It is because of the internal counters used for thread transactional tracking. They impose some minor overhead above the standard TL2.

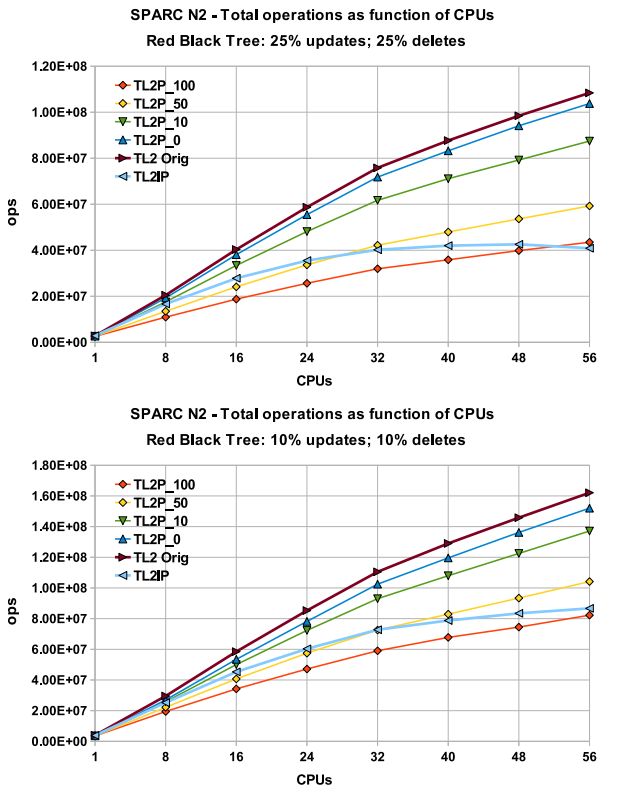

Figure 3. Throughput when private transactions do no work. A 2K sized Red-Black Tree on a 128 thread Niagara 2 with 25% puts and 25% deletes and 10% puts and 10% deletes for TL2, TL2-IP, and TL2P varying the percentage of private transactions: 100%, 50%, and 10%.

5.2 Random Array Benchmark

In the random-array benchmark we vary the privatization density and the transaction patterns. The goal is to estimate the penalty private transactions pay for different transaction lengths. For short transactions, we use 32 reads per reader and 16 read-modify-write operations per writer. We use 128 reads and 64 read-modify-write operations for the long transaction case. To mimic the heterogeneous case, the reader length is randomized between 1-128 reads, and the writer between 1-64 read-modify-write accesses.

In Figure 4 we can see that the performance in the short and long transactions benchmarks is nearly the same. In both

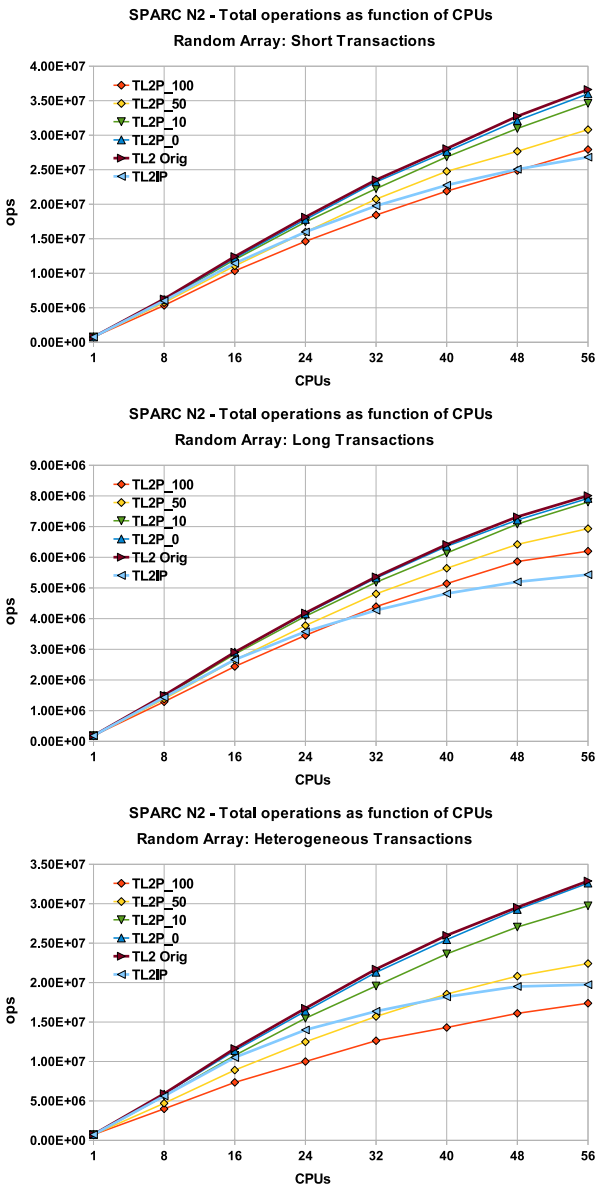

Figure 4. Throughput when private transactions do no work. A 4M sized Random-Array on a 128 thread Niagara 2 for, short transactions of 32 reads per reader and 16 read-modify-write operations per writer, long transactions of 128 reads per reader and 64 read-modify-write per writer and heterogeneous transactions with random [1-128] reads per reader and random [1-64] read-modify-write per writer

we see a 20% penalty for TL2P in the 100% privatization case, but the TL2IP performance is different in the long transaction case, caused by a heavier use of the global clock, which is affected by long transactions.

The heterogenous benchmark creates a higher penalty than the constant length transactions. That is because the private transaction barrier waits for all the active threads and the possibility that it will wait for the long one thread is higher as there are more threads in parallel. Therefore, the penalty for the quiesence is higher, but as the privatization level decreases to 50% and 10%, the performance improves. This is the case where the on demand privatization approach saves the situation and allows TL2P to achieve good results.

5.3 Realistic Private Transactions

The two graphs in Figure 5 depict the more realistic situation when private transactions involve computation. In this case it consists of a sequence of random reads approximately 10-15 times longer than the privatizing transaction. Because these are reads, TL2 can run them even though it has no privatization. Here you can see that TL2P (in blue) provides virtually the same performance as TL2 (in red) at all levels of privatization, confirming the potential of the private transaction quiescing technique. In contrast, TL2-IP (in orange) does not scale at any level.

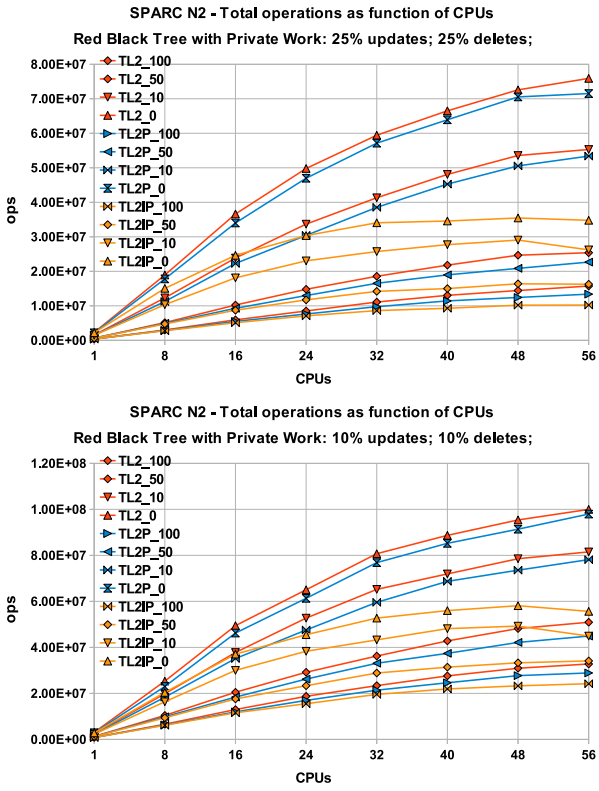

Figure 5. Throughput when private transactions do a large amount of private work. A 2K sized Red-Black Tree on a 128 thread Niagara.

In summary, we see that the simple quiescence privatization technique added to the TL2 STM, provides TL2 with privatization support which delivers great scalable performance under realistic conditions and takes advantage of partial privatization under full stress when there are empty private transactions.

6. Conclusion

We presented the first scalable approach for privatizing TL2/LSA style invisible-read-based STM algorithms. *Private transactions* offer a simple intermediate approach that combines the ease of use of implicit privatization, with the efficiency that can be obtained from explicitly knowing which regions are private. The result is a "pay as you go" cost for privatization, and a framework, private transactions, that will hopefully allow for further compiler and other optimizations that will make privatization a low cost addition technique not just for STMs but perhaps in general for concurrent data structures.

The *quiescing algorithm* at the basis of the private transaction methodology is of independent value as it can be used as a privatization barrier within STMs or in the context of other data structures.

7. Acknowledgements

This paper was supported in part by grants from the European Union under grant FP7-ICT-2007-1 (project VELOX), as well as grant 06/1344 from the Israeli Science Foundation, and a grant from Sun Microsystems.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.

[3] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Transact 2009*, Raleigh, North Carolina, USA, 2009.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[5] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542494.

[6] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345241.

[7] C. G. Gray and D. R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. Technical report, Stanford University, Stanford, CA, USA, 1990.

[8] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. *Parallel Processing, International Conference on*, 0:67–74, 2008. ISSN 0190-3918. doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2008.69.

[9] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 314–325, jun 2008.

[10] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic stm. In *Transact 2008 Workshop*, 2008.

[11] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[12] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. 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. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1123001.

[13] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proc. 12th International Conference on Principles of Distributed Systems*, dec 2008. Springer-Verlag Lecture Notes in Computer Science volume 5401.

[14] M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: scalable transactions with a single atomic instruction. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378583.

[15] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.4.

[16] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378584.