ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory*

Hugo Rito and João Cachopo

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{hugo.rito,joao.cachopo}@ist.utl.pt

Abstract. Software Transactional Memory (STM) is one promising abstraction to simplify the task of writing highly parallel applications. Nonetheless, in workloads lacking enough parallelism, STM's optimistic approach to concurrency control can adversely degrade performance as transactions abort and restart often.

In this paper, we describe a new scheduling-based solution to improve STM's performance in high-contention scenarios. Our Progressively Pessimistic Scheduler (ProPS) uses a fine-grained scheduling mechanism that controls the amount of concurrency in the system gradually as transactions abort and commit with success.

Experimental results with the STMBench7 benchmark and the STAMP benchmark suite showed that current coarse-grained, conservative transaction schedulers are not suitable for workloads with long transactions, whereas ProPS is up to 40% faster than all other scheduling alternatives.

Keywords: Performance, Software Transactional Memory, Transaction Conflict, Transaction Scheduling.

1 Introduction

Software Transactional Memory (STM) [11] turned into one of the most promising abstractions to bridge the gap between mainstream programmers and parallel programming. Unfortunately, the performance of STM-based applications may vary greatly, depending on the application's workload: Even though STMs exhibit very good performance for read-dominated workloads, the same cannot be said about highly contended workloads in which frequent transaction reexecutions place a significant stress on the system, hindering its performance [1,3,7].

Transactions reexecute whenever they conflict, which happens when the STM runtime speculatively executes two or more concurrent transactions that cannot both commit due to conflicting memory accesses.

A transaction scheduler [12,2,4] is an STM component that uses runtime information to predict conflicts and, thus, prevent transactions that are likely to

* This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

conflict from running concurrently. The assumption is that in workloads lacking inherent parallelism, executing a large number of transactions concurrently can degrade performance as transactions restart often. So, to limit the amount of restarts and the amount of wasted work, a transaction scheduler serializes conflicting transactions either at transaction begin or at transaction restart.

Unfortunately, most scheduling policies are too conservative as they over-serialize transactions—that is, two non-conflicting transactions are scheduled to execute one after the other when they could safely overlap.

In the next section, we discuss how STMs may benefit from transaction scheduling in high-contention workloads and we explain why coarse-grained and conservative scheduling policies, as those used by existing transaction schedulers, are unable to extract the latent parallelism of STM-based applications.

In this paper, we tackle the problem of efficient transaction scheduling and we make the following contributions:

– A new fine-grained progressively pessimistic scheduling policy (ProPS) for STM that collects information regarding the maximum concurrency level between pairs of atomic operations and, then, uses that information to gradually reduce concurrency as contention increases (Section 3).
– An overview of ProPS's implementation in the FlashbackSTM [10]. This fully decentralized implementation of our novel fine-grained scheduling policy has zero runtime overhead for read-only transactions (Section 4).
– A thorough evaluation of ProPS with both the STMBench7 benchmark [6] and the STAMP benchmark suite [8]. Results show that ProPS is up to 40% faster than ATS [12], CAR [2], and Shrink [4] (Section 5).

2 Why We Need Better Transaction Scheduling

The key observation behind transaction scheduling is that conflicts are dynamic, meaning that the order in which transactions execute influences the number of conflicts that occur. Moreover, in many STM-based programs, transactions execute independently of each other in a nondeterministic order. Hence, by changing the order in which transactions execute, a transaction scheduler may reduce the amount of wasted work in high-contention workloads and increase throughput.

In practice, transaction schedulers use serialization to order transactions with expected conflicts one after another, trading off concurrency between threads for less wasted work. To exemplify, consider the execution scenario 1 depicted at the top half of Figure 1. In this scenario, thread T_1 makes two calls to atomic operation $OP1$ while thread T_2 tries to execute atomic operation $OP2$ once.

Without a transaction scheduler, the concurrent execution of $OP1$ and $OP2$ has an adverse effect on performance because both atomic operations conflict and, thus, only Tx_1 (first) and Tx_3 (later) commit with success without conflicting. Transaction Tx_2, on the other hand, aborts and reexecutes twice before committing with success, which happens only when executing solo in the system.

With a transaction scheduler, after detecting the conflict between Tx_1 and Tx_2 (and to prevent Tx_2 from restarting again) the scheduler may force new

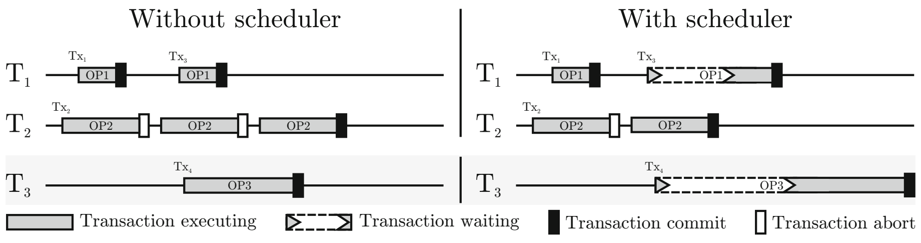

Fig. 1. Execution of operations $OP1$, $OP2$, and $OP3$ without a transaction scheduler and with a naive transaction scheduler by two concurrent threads (scenario 1) and three concurrent threads (scenario 2). Only $OP1/OP2$ conflict when executed concurrently.

transactions to serialize after Tx_2—that is, the scheduler delays Tx_3's start to after the successful commit of transaction Tx_2. With this decision, the scheduler reduces to half the number of transaction restarts, therefore reducing the execution time.

Ideally, the transaction scheduler is accurate enough to execute concurrently only transactions that will not conflict. Though possible in some particular cases, in general this is very hard to accomplish due to the dynamic nature of transactions and, thus, schedulers serialize transactions based on previous observations. Current scheduling solutions, however, are still too coarse-grained, too conservative, and, for those reasons, may serialize non-conflicting transactions.

Coarse-grained scheduling solutions [12] monitor the number of aborts to detect periods of high-contention, in which case they serialize all transactions. Such schedulers assume that, when contention is high, a transaction that aborts and restarts immediately has high probability to conflict again, leading to another transaction abort. Thus, to prevent conflict-prone transactions from conflicting again, the scheduler serializes all transactions that abort.

Despite their low overhead, these *all-or-nothing* approaches to scheduling have limited applicability because transactions are serialized not due to the transaction's expected behavior but because of the behavior of the system as a whole. To exemplify, consider scenario 2 of Figure 1 that extends scenario 1 with a third thread (T_3) executing a single atomic operation $OP3$.

With the transaction scheduler, the reexecution of transaction Tx_2 forces all subsequent transactions (Tx_3 and Tx_4, in this case) to serialize. Yet, as the execution without the scheduler shows, this coarse-grained scheduling policy is over-serializing transactions. Only Tx_2 and Tx_3 need to execute one after the other because only the pair $OP1/OP2$ conflict when executed concurrently. The pairs $OP1/OP3$ and $OP2/OP3$ do not conflict and may execute concurrently with performance benefits as we observe in the scenario without scheduling.

Our naive scheduling policy is an over-simplification of Yoo and Lee [12]'s Adaptive Transaction Scheduler (ATS). In ATS, each thread maintains a contention intensity (CI) value, which is decreased after each successful commit and increased after each abort, and threads serialize in a central queue whenever their CI value is above a predetermined threshold. ATS's scheduling policy is very simple and has nearly no overhead, but is too coarse-grained and unnecessarily reduces concurrency in high contention scenarios, as described before.

Conservative scheduling solutions [2], on the other hand, serialize transactions based on the fact that the atomic operations they execute conflicted with each other, at least once, in the past. By using per-transaction information, the scheduler attempts to predict more accurately how a particular transaction configuration will behave when executed again concurrently. Going back to the previous example, a conservative scheduler may learn that operations $OP1$ and $OP2$ conflict, in which case it will serialize all their future executions.

CAR-STM [2] is a conservative scheduling policy that maintains a per-core transaction queue and, when a transaction restarts, the dispatcher serializes the restarting transaction in the per-core queue containing the transactions with maximum probability of conflicting with it. Even though less conservative than ATS, with a large number of concurrent threads or under high contention, CAR-STM's per-core queues may constitute a performance bottleneck.

The problem with both scheduling policies is that they ignore the fact that transactions are dynamic—that is, a transaction's behavior may change as the state of the application also changes. This means that, for instance, operations $OP1$ and $OP2$ in our example may be able to execute concurrently in the future, if they access disjoint memory locations.

Recognizing this runtime property of transactions, the Shrink [4] scheduler uses the memory locations recently accessed by a thread to predict the read-set of future transactions executed by that thread. At transaction start, Shrink verifies whether any of the memory locations in the transaction's predicted read-set is being written by other concurrently executing transactions and, if that is the case, the starting transaction serializes by acquiring a global shared lock. However, it is unclear how the read-set of a transaction may help predict the read-set of a different transaction executing a distinct atomic operation, even considering the fact that both transactions are executed in succession by the same thread. Also, Shrink intercepts all read accesses to memory, adding a non-negligible overhead to the most common STM operation: the transactional read.

In summary, transaction schedulers' pessimistic approach to concurrency may reduce the number of conflicts between transactions but at the cost of reducing too much the parallelism in the application. The decision to serialize transactions that would execute without conflicting greatly hinders the throughput of the system and constitutes a fundamental obstacle to the effectiveness of scheduling. The challenge, then, is to develop a fine-grained, more optimistic transaction scheduler that is able to increase parallelism between transactions.

3 A Progressively Pessimistic Scheduling Policy

Although system-wide information may help describe the runtime behavior of the system as a whole, the transaction scheduler acts upon individual transactions and, for that reason, the scheduler needs fresh transaction-specific information to perform fine-grained scheduling decisions that minimize the number of transactions that are unnecessarily serialized.

To allow such fine-grained scheduling our new Progressively Pessimistic Scheduler (ProPS) maintains a *concurrency level* matrix (CL) between pairs of atomic operations—that is, for each atomic operation of type i and each atomic operation of type j, the value of CL_{ij} describes how many transactions executing atomic operations of type i may execute concurrently with one transaction executing atomic operation of type j.

In the beginning, all CL_{ij} values are equal to `MAX_THREADS`, which corresponds to the maximum number of concurrent threads in the systems (typically the number of processors in the machine), and ProPS uses CL values to adapt the amount of concurrency in the system: At transaction begin of atomic operation i, the scheduler calculates the minimum CL_{ij} value between the starting transaction and all other in-flight transactions. Atomic operations with a minimum CL value of `MAX_THREADS` proceed normally. Yet, as an operation's minimum CL value decreases, ProPS reduces the number of transactions executing that operation.

When a transaction of type i aborts due to a conflict with another transaction of type j, ProPS reduces the concurrency level between atomic operations of type j and i using equation 1 below, where k is a value in $[0, 1]$.

$$CL_{ji} = CL_{ji} \times k \tag{1}$$

By limiting the number of transactions of type j that may start concurrently with transactions of type i only, our new scheduling policy reduces the STM's level of optimism in a fine-grained way. Future transactions for different atomic operations are unaffected by this reduction and, thus, may proceed normally at transaction begin if their minimum CL value is equal to `MAX_THREADS`.

When a transaction of type i finally commits with success, for each operation of type j ProPS updates operation's i CL_{ij} values using equation 2 below, where α is a value in $[0, 1]$, and $numRestarts \geq 0$ corresponds to the number of times that the committing transaction restarted before this successful commit.

$$CL_{ij} = min(\texttt{MAX\_THREADS}, CL_{ij} + \texttt{MAX\_THREADS} \times \alpha \div (1 + numRestarts)) \tag{2}$$

Note that, by design, ProPS exponentially reduces concurrency as transactions conflict but increases concurrency only linearly at transaction commit. This design decision allows the scheduler to react very fast to periods of high contention, while, at the same time, to steadily revise its predictions as transactions start committing with success. Furthermore, at transaction commit, our scheduling policy uses the number of times the transaction aborted before committing with success to control how fast the scheduler restores concurrency, benefiting transactions that seldom conflict.

4 The ProPS Implementation

We implemented ProPS in the FlashbackSTM [10], a word-base, multi-version STM implemented as a pure Java library that extends the lock-free version of the JVSTM [5] with the concept of memo-transactions [9]. In the FlashbackSTM,

```
static double[][] CL; static TxInfo[] txs; TxInfo myInfo

upon tx.begin:
  myInfo.id = tx.id; myInfo.numRestarts = 0
  do
    cl = MAX_THREADS; enemies = 1; worstEnemy = nil
    for each inFlightTx in txs do
      if (CL[tx.id][inFlightTx.id] < cl)
    cl = CL[tx.id][inFlightTx.id]; enemies = 1; worstEnemy = inFlightTx
      else if (inFlightTx == worstEnemy)
    ++enemies
  while (cl ÷ enemies < 1)
  limitConcurrency(cl ÷ enemies)

upon tx.abort caused by enemyTx:
  myInfo.numRestarts++
  CL[enemyTx.id][myInfo.id] = CL[enemyTx.id][myInfo.id] * $k$

upon tx.commit:
  txs[myInfo.pos] = nil
  for each opId in atomicOperations do
    CL[myInfo.id][opId] = min(MAX_THREADS,
       CL[myInfo.id][opId] + MAX_THREADS × $\alpha$ ÷ (1 + myInfo.numRestarts))
```

Listing 1.1. The ProPS implementation. The scheduler is fully decentralized as each thread decides whether to wait or to begin immediately by itself.

reads are very fast, always consistent, and read-only transactions never conflict with other transactions. Read-write transactions, on the other hand, may conflict but only with other already committed read-write transactions.

To control the execution and the order in which read-write transactions commit, we changed the FlashbackSTM in two ways. First, we changed read-write transactions so that they report to the scheduler at transaction begin time, commit time, and abort time. Second, we changed the bytecode manipulator so that it assigns a unique identifier (`ID`) to each atomic operation.

Note that our modifications to the FlashbackSTM have zero runtime overhead for read-only transactions: Given that read-only transactions never conflict in the FlashbackSTM, they do not need to be scheduled and, thus, never report to the transaction scheduler as read-write transactions do. In Listing 1.1 we show the pseudocode of ProPS, which works in a fully decentralized way because each thread decides whether to wait or to begin immediately by itself.

ProPS stores per-thread information in a `TxInfo` object and system-wide information in a global `CL` matrix and in a global `txs` array. The thread-local `TxInfo` instance gathers information about the transaction currently in execution by the thread, such as the `ID` of the atomic operation, and the number of transaction restarts. On the other hand, the global `CL` matrix stores the concurrency level between pairs of atomic operations, as described in the previous section, whereas the global `txs` array contains all in-flight transaction currently in the system.

At begin time, the scheduler updates the thread's `TxInfo` instance with information regarding the new transaction (line 4) and uses the `CL` matrix to calculate the transaction's minimum `cl` value, depending on the operation's `ID` and the current system configuration (lines 5–12).

The `limitConcurrency` function (line 13) may delay the execution of a transaction because it forces the starting transaction to acquire a position in the `txs` array with a compare-and-swap (CAS) operation. When a transaction successfully acquires a given position in the `txs` array, it may begin its execution (otherwise, it will have to keep trying until it succeeds); when the transaction finishes, it releases its position in the `txs` array, as shown in line 20.

The size of the array corresponds to the maximum number of read-write transactions that the scheduler will allow to execute concurrently—in our current implementation, the size of the array corresponds to the number of cores in the machine—and the scheduler uses the minimum `cl` value of each starting transaction to control the number of positions in the array that may be used. With a `cl` value equal to `MAX_THREADS`, the scheduler behaves similarly to an optimistic scheduler. Lower `cl` values make ProPS progressively more pessimistic.

At transaction abort, the scheduler increments the number of restarts (line 16) and reduces the concurrency level (line 17). At commit time, a committing transaction increments its concurrency level with all atomic operations (lines 21–23).

It is worth mentioning that we made our implementation as lightweight as possible. For instance, accesses to the CL matrix are not explicitly synchronized and, thus, threads may read stale data. We argue, however, that adding random imprecisions to the scheduler is preferable than to pay the high cost of synchronization because, in this particular context, suboptimal scheduling decisions do not change the semantics of the programs, only their performance.

5 Experimental Results

To evaluate our approach, we used the STMBench7 benchmark [6] and the STAMP benchmark suite [8]. We ran these benchmarks using the FlashbackSTM either with no scheduler (shown as *Default*) or with one of the following schedulers: ProPS, ATS [12], CAR [2], and Shrink [4].

We configured ProPS with a k-value of 0.5, an α-value of 0.05. We tested with several values for these parameters and used the values that produced the best results. Due to space constraints, we do not show in this paper a sensitivity analysis for these parameters, but the results do not vary too much within a reasonable range for these values.

Neither one of the pessimistic schedulers used in our tests had an implementation for the FlashbackSTM, so we provided our own optimized implementation of each scheduling policy. To collect fair and comparable results, all four schedulers share the same FlashbackSTM code base and the same scheduling interface.

We ran our tests on a machine with four AMD Opteron 6168 processors, each with 12 cores, for a total number of 48 cores. All processors shared a Supermicro H8QG6 motherboard with 128Gb of RAM. The machine was running CentOS

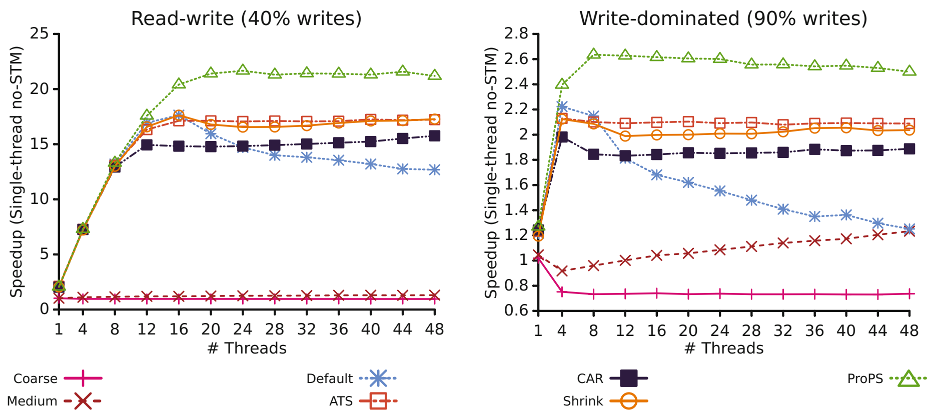

Fig. 2. Speedup of the STMBench7 benchmark with all long read-write traversals and all structural modifications disabled, for each of the two workloads

release 6.4 and Java SE version 1.7.0_21. We made 20 runs of each benchmark with 1 up to 48 threads in increments of 4 threads per test, and we removed the top 5 best and worst results, presenting only the average of the ten remaining values. The speedup results use as baseline the execution time of the benchmark running single-threaded without any STM instrumentation.

5.1 STMBench7 Benchmark: Short Transactions

The STMBench7 benchmark was designed to test STMs under high-contention scenarios, making it appropriate to understand how non-negligible concurrency among transactions that often results in reexecutions affects performance.

We measured the time it took for the benchmark to complete a fixed number of operations with all long read-write traversals and all structural modifications disabled in a read-write workload (40% writes out of 130000 total operations) and in a write-dominated workload (90% writes out of 60000 total operations).

In Figure 2, we present speedup results for the STMBench7 benchmark using both the FlashbackSTM with the various schedulers and two lock-based approaches: coarse-grained locks and medium-grained locks.

Although STM's indirect memory accesses add overhead, on both workloads with one thread the STM version of the benchmark is faster than the non-instrumented version of the benchmark. This happens because some operations execute repeated method calls. These methods, when executed inside Flashback-STM's memo-transactions, populate a per-transaction memo-cache with information about their runtime behavior. The STM then uses this information to identify repeated work that may be skipped, thus improving performance.

Comparing the results obtained with the various schedulers, we see that, regardless of the workload, ProPS outperforms all other approaches. The results for the read-write workload with the STM are specially good when compared to locks, because this workload benefits both from our less pessimistic approach to

Table 1. Percentage of aborts of the STMBench7 benchmark with all long read-write traversals and all structural modifications disabled, for both workloads with 48 threads

		Transaction scheduler			
Workload	Default	ATS	CAR	ProPS	Shrink
Read-write	59.41	9.34	8.71	15.96	30.49
Write-dominated	65.06	7.93	6.92	16.50	28.96

scheduling and from FlashbackSTM's read-only operations that have very low overhead and never conflict.

Overall we can conclude that, as the number of concurrent threads increases, conflicts become more frequent and, therefore, the benchmark starts to benefit from scheduling. The influence that conflicts have on performance is more evident on the write-dominated workload where the FlashbackSTM without scheduling achieves its peak speedup with 4 threads and then performance abruptly plunges to the point that, with 48 threads, the benchmark executes as fast as with 1 thread. With scheduling, on the other hand, the benchmark is able to maintain the performance stable as the number of threads increases.

Despite the drastic reduction in the abort rate (Table 1), none of the pessimistic schedulers' peak performance surpasses the peak performance of the Default scheduler. As the results with ATS and CAR clearly show, even on write-dominated, conflict-prone workloads a lower abort rate may not translate into better performance if the scheduler is too pessimistic and serializes transactions that could otherwise execute concurrently without conflicts.

Instead of serializing all transactions when contention is high as traditional pessimistic schedulers do, ProPS's progressively pessimistic scheduling policy gradually reduces concurrency when transactions start conflicting. Thus, even though the abort rate goes up to 16.50%, ProPS outperforms all other alternatives, showing that there is latent parallelism in the benchmark that is not explored by the pessimistic schedulers.

5.2 STMBench7 Benchmark: Mixed Transactional Workload

The previous results were obtained with two workloads that execute short transactions predominantly. Now, we explore how the various schedulers behave for a workload with very long transactions: For that, we use again the STMBench7 benchmark, but now with all long read-write traversals enabled.

For these tests, we changed the number of operations executed on each workload to 4000 operations on the read-write workload and to 2000 operations on the write-dominated workload. This change was necessary to maintain an average execution time of roughly 30 seconds with 48 threads. We present the speedup results in Figure 3.

As we can see, all pessimistic schedulers perform worse than the FlashbackSTM with no scheduler, a result somewhat surprising because the use of a scheduler

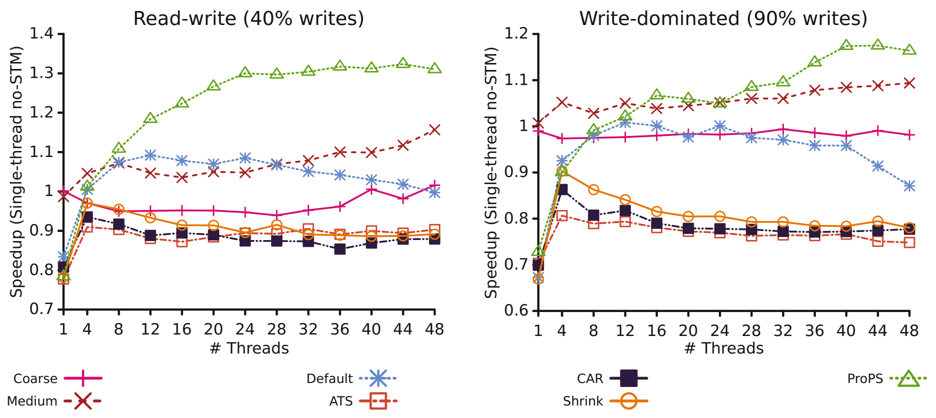

Fig. 3. Speedup results relative to a sequential execution of the STMBench7 benchmark with all structural modifications disabled, for each of the two workloads

should reduce the amount of wasted computation due to conflicting transactions, and STMBench7 is known for having a highly conflicting workload. Yet, despite its high abort rate, the Default approach extracts more parallelism from the benchmark with its optimistic approach, and, thus, it has better performance.

These results show that the performance issues caused by over-serialization are specially bad in applications that execute large numbers of threads in a mixed transactional workload where the size of transactions may vary greatly.

Furthermore, our results strongly indicate that the assumption behind most pessimistic scheduling policies—that in high contention workloads transactions that conflicted at least once in the past will always conflict with each other again in the future—is usually wrong and, for that reason, schedulers need to take into consideration the dynamic nature of transactions when deciding.

ProPS's more optimistic approach to concurrency, coupled with fine-grained information about the conflict probability between atomic operations, is able to make better scheduling decisions, extract more parallelism from the benchmark, and improve performance up to 35% in these two highly contented workloads.

Finally, despite the additional overhead imposed by the STM, the Flashback-STM with ProPS outperforms locks and scales better on both workloads. Even on the worst case scenario where 90% of transactions are read-write and may read up to 1 million memory locations, ProPS is able to extract the benchmark's latent parallelism and scale up to 40 threads. In this very demanding workload, medium-grained locks are only 10% faster with 48 threads than with 1 thread, whereas ProPS with 48 threads executes 70% faster than with 1 thread and surpasses the performance of medium-grained locks for 16 or more threads.

5.3 STAMP Benchmark Suite

STAMP has eight different applications but we limited our study to Genome and Vacation as these applications represent two important execution scenarios:

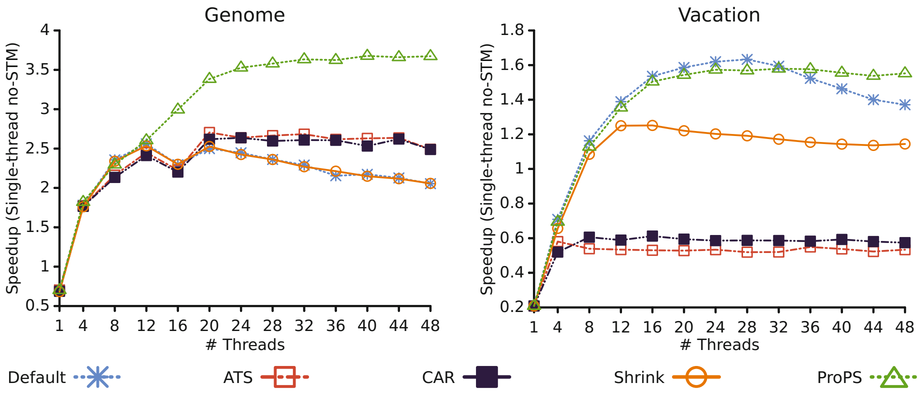

Fig. 4. Speedup results for the Genome and the Vacation applications

Genome executes millions of short transactions (98% of read-write transactions read less than 3 memory locations), whereas Vacation predominantly executes long transactions that perform up to 7226 transactional reads. Each application executed with the following parameters: For Genome, "-g 32768 -s 64 -n 66777216", and for Vacation, "-n 1800 -q 90 -u 90 -r 16384 -t 300000".

Figure 4 shows the speedup results for the various schedulers. Once again, ProPS consistently outperforms all other transaction schedulers.

Genome's results highlight the usefulness of our scheduler in an application that executes millions of micro transactions. ProPS is always as good or better than all other schedulers, improving performance up to 40%. Yet, ProPS does not scale past 24 threads and we believe that the cost of creating and terminating a high number of short lived transactions justifies this performance plateau.

The Vacation benchmark reinforces the idea that current pessimistic schedulers are not suitable for workloads with long transactions: Again, all pessimistic schedulers perform significantly worse than Default. Most transactions in this benchmark are long and, therefore, the decision to serialize any transaction that would be able to execute without conflicting greatly hinders the performance of the system. ATS, CAR, and Shrink use coarse-grained, conservative heuristics that fail to predict the behavior of each individual transaction and end up serializing almost all threads. ProPS, on the other hand, is the first transaction scheduler to perform well on these types of workloads.

6 Conclusions

In this paper we proposed ProPS, a new transaction scheduler for STM systems that gradually adapts the amount of concurrency in the application as transactions abort and commit. When compared to other scheduling policies, our new scheduling policy is fine-grained, because ProPS calculates CL_{ij} values for each pair of atomic operations i and j, and is *progressively pessimistic*, because rather than serializing all transactions when contention is high, ProPS gradually reduces concurrency as CL values decreases.

Experimental evaluation with the STMBench7 benchmark and the STAMP benchmark suite demonstrated the usefulness of our novel scheduling policy as ProPS was able to outperform and scale better than all other scheduling alternatives in a variety of workloads and applications. Unlike conservative solutions, our less pessimistic approach to scheduling performs well in workloads with long transactions and with a lot of latent parallelism.

References

1. Cascaval, C., Blundell, C., Michael, M., Cain, H., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue 6, 46–58 (2008)
2. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In: Proceedings of the 27th ACM Symposium on Principles of Distributed Computing, PODC 2008, pp. 125–134 (2008)
3. Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Commun. ACM 54, 70–77 (2011)
4. Dragojević, A., Guerraoui, R., Singh, A., Singh, V.: Preventing versus curing: Avoiding conflicts in transactional memories. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, pp. 7–16 (2009)
5. Fernandes, S., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 179–188. ACM (2011)
6. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 315–324 (2007)
7. McKenney, P., Michael, M., Triplett, J., Walpole, J.: Why the grass not be greener on the other side: A comparison of locking vs. transactional memory. SIGOPS Oper. Syst. Rev. 44, 93–101 (2010)
8. Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IEEE International Symposium on Workload Characterization, IISWC 2008, pp. 35–46. IEEE (2008)
9. Rito, H., Cachopo, J.: Memoization of methods using software transactional memory to track internal state dependencies. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ 2010(2010)
10. Rito, H., Cachopo, J.: FlashbackSTM: Improving STM performance by remembering the past. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 266–267. Springer, Heidelberg (2013)
11. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, pp. 204–213. ACM (1995)
12. Yoo, R., Lee, H.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 169–178. ACM (2008)