Int J Parallel Prog (2017) 45:482–507
DOI 10.1007/s10766-016-0411-z

CrossMark

Restart Optimization for Transactional Memory with Lazy Conflict Detection

Miloš Cvetanović[1] · Zaharije Radivojević[1] · Veljko Milutinović[1]

Received: 26 August 2014 / Accepted: 15 March 2016 / Published online: 28 March 2016
© Springer Science+Business Media New York 2016

Abstract This paper presents an optimization algorithm for transactional memory with lazy conflict detection. The proposed optimization attempts to minimize the execution time of restarted transactions. Minimizing happens during restart, by avoiding the re-execution of a section of a transaction that is unaffected by the restart. The proposed optimization builds on previous research and differs in that it eliminates the need for the prediction of conflicting accesses and introduces incremental context saving. Moreover, the paper introduces analytical models for estimating the execution time of transactions, with and without the restart optimization, that are developed using the continuous-time model. A critical evaluation comparing analytical models with the simulation results is discussed in the paper.

Keywords Transactional memory · Restart optimization · Analytical modeling · Continuous-time modeling · Performance analysis

1 Introduction

The goal of transactional memory is to simplify multi-core concurrent programming [1]. A programmer demarcates a section of source code as a transaction while the transactional memory provides run-time support. This support includes atomicity, concurrency, and isolation.

Transactional memory achieves data consistency for transactions executing concurrently by providing conflict resolution. A conflict situation arises when two or more transactions access a shared data item, and at least one of them modifies it [2].

✉ Zaharije Radivojević
zaki@etf.bg.ac.rs

[1] School of Electrical Engineering, University of Belgrade, Belgrade, Serbia

Version management defines whether modifications perform directly to the shared data (eager version management), or are buffered as speculative writes (lazy version management). Speculative writes become visible in shared data at commit time after conflict resolution. The conflict detection strategy, depending on detection time, may also be eager or lazy. Eager conflict detection tends to prevent possible conflicts by introducing synchronization mechanisms that stall the offending transactions. Lazy conflict detection tends to react to consequences of the conflicts afterwards by aborting and restarting the offending transactions.

An implementation of transactional memory may be in software, hardware, or both. Hardware implementations provide low overhead by keeping transactions' internal states in cache memory [3–5], but suffer from possible resource limitations [6]. Software implementations offer the flexibility to run on existing hardware and use operating memory as an unlimited resource to keep transactions' internal states, but reduce performance [7,8]. Hybrid implementations attempt to find a compromise by bringing together low overhead and unlimited resources [9,10].

Transactional memory Coherence and Consistency (TCC) is an example of hardware implementation of transactional memory with lazy version management and lazy conflict detection [11]. The TCC executes transactions on multiple cores with separate level one (L1) caches that use level two (L2) cache as a shared memory. For each transaction, during execution, TCC buffers transactional writes locally as speculative writes. During the commit time of a transaction, TCC commits speculative writes of the transaction to shared memory atomically, and simultaneously broadcasts them to other transactions in the system. The other transactions snoop broadcasts to maintain cache coherence, while TCC aborts and restarts all transactions that accessed data modified by the committing transaction.

A transaction aborts by discarding all speculative writes and restarts by switching to the context corresponding to the transaction start. The main disadvantage of the abort and restart of a transaction is that all progress made in the transaction before a restart occurred is wasted. This paper presents an optimization that attempts to diminish the effects of a restart by reducing the time necessary for re-execution of the restarted transaction. Re-execution time of the restarted transaction can be reduced by exploiting the progress made before the restart occurred. Instead of switching to the start, the transaction switches to the last valid state. The last valid state corresponds to the context of the transaction just before the first access to the shared data that caused the restart. In order to reduced memory requirements for retaining all necessary contexts the proposed optimization saves the contexts incrementally.

In addition to proposing the restart optimization, this paper introduces analytical models for transactional memory with lazy conflict detection. The models cover transactional memories both with and without restart optimization. The analytical models study system performance independently from the time between consecutive memory accesses inside of a transaction by estimating the restart probability, execution time, relative performance gain, and spatial complexity. The model shows that introduction of the restart optimization, may bring the expected relative gain up to 0.33 for transactions that have been restarted at least once.

This paper has the following structure. Section 2 gives an overview of hardware transactional memory from the aspect of this paper. Section 3 describes principles of

the proposed optimization, and explains the usage of additional hardware resources and algorithms behind these principles with the TCC example. Section 4 presents the proposed analytical model for restart probability, execution time, relative performance gain, and spatial complexity of TCC with restart optimization. A critical evaluation comparing analytical models with simulation results is discussed in Sects. 5, and 6 reviews the conclusions of this research.

2 Related Work

Restart optimization for lazy version management can be achieved through various activities in execute, commit, and abort phase of a transaction. Execute phase encompass saving initial context, running transaction, and in some cases saving additional contexts for purpose of restart optimization. During commit phase speculative writes are broadcasted and conflicts are detected. Resolution of detected conflicts is conducted in abort phase where offended transactions are aborted and restarted by returning to the appropriate context. Most of work related to the restart optimization suggests either saving intermediate contexts during execute phase of a transaction or by optimizing commit/abort phase of a transaction as shown in Table 1 that gives an overview of solution space for improving lazy conflict detection for analyzed hardware transactional memories.

Table 1 Overview of solution space for improving lazy version management

Solution	Execute phase	Commit phase	Abort phase
Waliullah and Stenstrom [12]	Saving entire context when accessing predicted memory locations		Return to an appropriate context
Waliullah and Stenstrom [13]	Saving entire context when accessing predicted memory locations + removing conflicts		Return to an appropriate context
Ceze et al. [14]		Reduce commit time by using block signature	
Quislant et al. [15]		Reduce commit time by using asymmetric block signature	
Tomic et al. [16]	Keeping track of offending transactions		Fast return to the beginning of a transaction
Lupo et al. [17]	Storing speculative and non-speculative values in different cache levels		Fast return to the beginning of a transaction
Ros et al. [18]		Reduce commit time by reducing core-to-core traffic	

The research presented in [12] suggests saving entire context of a transaction during execute phase each time the transaction accesses to shared data for which there is a prediction that a conflict may occur. Prediction is created according to the history of previous executions of the transaction. The approach presented in this paper eliminates the need for the prediction of conflicting accesses by suggesting saving the contexts before any first access to shared data. The new problem that may arise is a number of contexts that need to be saved. In order to reduced memory requirements for retaining all necessary contexts the approach proposes saving the contexts incrementally.

Most of other related work is complementary to the approach presented in this paper and attempts to either remove conflicts when possible [13] or to optimize commit/abort phases of a transaction to improve the overall execution time of a set of transactions. A possible optimization of a commit/abort phase can occur by reducing the conflict detection time with the introduction of memory block signatures as in Bulk [14], or when dealing with asymmetry in transactional data sets [15], or by using early conflict detection combined with lazy conflict resolution [16]. Introduction of a new transactional memory coherency protocol, FasTM, which uses the L1 cache to save state before a transaction starts, may also optimize the abort phase [17]. In the case of many-core chip multiprocessors, a commit phase could be optimized by introducing a new cache coherence protocol (DiCo-CMP) aimed at reducing core-to-core traffic [18].

In addition to extending previous research regarding restart optimization, this paper introduces analytical models for transactional memory with lazy conflict detection. Other analytical frameworks for performance modeling of software transactional memory [19–24] are based on Markov chain models. The model presented in this paper observes a transaction execution as a continual process represented with a continuous-time model.

3 Restart Optimization

The main principle of the proposed optimization is avoiding re-execution of a section of a transaction that is not affected by a restart. The restarted transaction continues execution from the place where first access to the shared data that caused the restart occurred. Figure 1 gives an example of two transactions labeled T1 and T2 (Fig. 1a), and their schedules on TCC (Fig. 1b) and TCC with restart optimization (RO-TCC) implementations (Fig. 1c). Transaction T1 uses data A, B, C, and D, while transaction T2 uses only C. The example shows that after commit of transaction T2, transaction T1 aborts and restarts. After the restart, transaction T1 starts from the beginning in the case of TCC implementation. In the case of RO-TCC, transaction T1 does not re-execute the section not affected by the committing data C, and starts from the moment when the data C is first accessed. Execution time of a transaction that has no restarts is the same on both TCC and RO-TCC implementations.

Enabling the proposed restart optimization requires certain modifications within TCC. The modifications encompass changes in behavior of the existing TCC hardware and additional hardware resources. The essence of the algorithm behind additional hardware resources is that a processor state has to be saved before the first access to each shared cache line. If a conflict on a shared cache line causes a restart, then a

(a)

T1	T2
Start	Start
Read(A)	Read(C)
Read(B)	C = C + 1
A = A + B	Write(C)
Write(A)	Commit
Read(C)	
A = A + C	
Write(A)	
Commit	

(b)

Time	T1 on Core1	T2 on Core2
t_0	Start	Start
t_1	Read(A)	
t_2	Read(B)	
t_3	A = A + B	Read(C)
t_4	Write(A)	C = C + 1
t_5	Read(C)	
t_6	A = A + C	Write(C)
t_7	Write(A)	
t_8	Abort	Commit
t_9	Read(A)	
t_{10}	Read(B)	
t_{11}	A = A + B	
t_{12}	Write(A)	
t_{13}	Read(C)	
t_{14}	A = A + C	
t_{15}	Write(A)	
t_{16}	Commit	

(c)

Time	T1 on Core1	T2 on Core2
t_0	Start	Start
t_1	Read(A)	
t_2	Read(B)	
t_3	A = A + B	Read(C)
t_4	Write(A)	C = C + 1
t_5	Read(C)	
t_6	A = A + C	Write(C)
t_7	Write(A)	
t_8	Abort	Commit
t_9	Read(C)	
t_{10}	A = A + C	
t_{11}	Write(A)	
t_{12}	Commit	

Fig. 1 An example of **a** two transactions and their schedules on **b** TCC and **c** RO-TCC implementations

valid state of the processor needs to be restored. The valid state is defined as the state immediately before the first access to the cached line that caused the restart. The state consists of a core's context and the content of the corresponding L1 cache. The core's context is saved each time in its entirety, while the content of the L1 cache is saved incrementally by creating copies of appropriate cache lines. A copy of a cache line is created when an access for a write operation is performed, but only if a fetch of some other cache line is performed between the current access and previous access to the cache line. Such situations can be detected using a counter.

For supporting restart optimization, TCC hardware needs additional resources for storing multiple valid states during the entire lifetime of a transaction, as shown on Fig. 2. After each new cache line fetch, a core's context is added to the Context Buffer. Incremental storing of the content of the L1 cache can be implemented using the Version Cache for keeping track of speculatively modified cache lines. Moreover, identifying when a new version needs to be created can be done with a counter and it can be implemented with an additional line Id field in a tag of each cache line.

The algorithms explaining the behavior of RO-TCC hardware when performing read, write, and restart operations during transaction are shown on Fig. 3. The read operation in case when L1 hit occurs requires only reading data from appropriate L1 cache line, as shown on Fig. 3a. If L1 miss occurs then the Version Cache is checked. The Version Cache may contain multiple versions of the same cache line and therefore in the case of Version Cache hit the most recent version (the version with the highest Id) of the requested line is copied to L1 cache and then accessed. When the Version cache miss occurs, the counter i is incremented and the requested line is fetched from memory and stored in L1 cache with the line Id tag field set to the value of counter i. In the same time while fetching data from memory, the core's context is saved in the entry i of the Context Buffer.

The write operation, as shown on Fig. 3a, in case when L1 hit occurs checks whether the Id tag field of the requested line is equal or smaller (it cannot be higher) than the value of counter i. If equal (meaning that the requested line is the last fetched line or that it has been modified since the last line had been fetched) then the cache line is updated without modifying the Id tag field. If smaller (meaning that the requested line is neither the last fetched line nor it has been modified since the last line had been

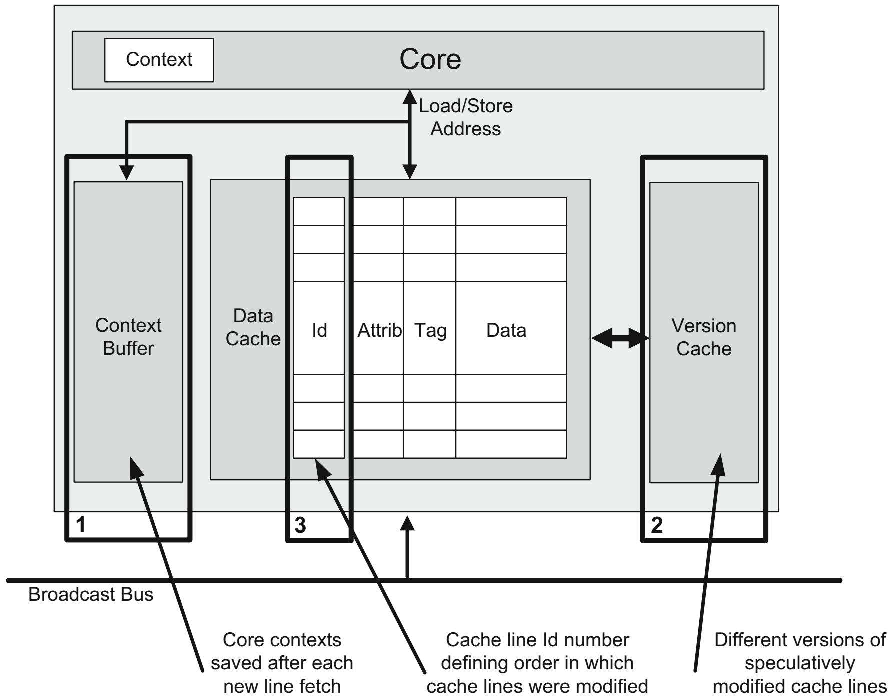

Fig. 2 The data cache organization for transactional memory with restart optimization

fetched) then the cache line is copied to the Version Cache, the Id tag field of the line is set to the value of counter i and the line is updated. If L1 miss occurs then the Version Cache is checked. In the case of Version Cache hit the most recent version (the version with the highest Id) of the requested line is copied to L1 cache and then the same steps are performed as when L1 hit occurs. When the Version cache miss occurs, the same steps are performed as when read operation Version Cache miss occurs with addition that the fetched line is updated.

The restart operation first determines the value of the Id tag field of the oldest line containing the offending address that caused the restart, as shown on Fig. 3b. The determined Id corresponds to the core's context created when the first access to the offending address occurred and therefore it needs to be restored. Moreover, all cache lines, from both L1 and Version Cache, with Id tag field greater or equal to the determined Id (created after the first access to the offending address) have to be invalidated. At the end, the counter i is set to determined Id.

4 Analytical Model

Instead of the explicit-modeling concurrent execution of many transactions, the analytical model presented in this paper characterizes the representative behavior of a single transaction and the influence that the rest of the concurrently executed transactions have on it. The impact of the other transactions is captured by computing the

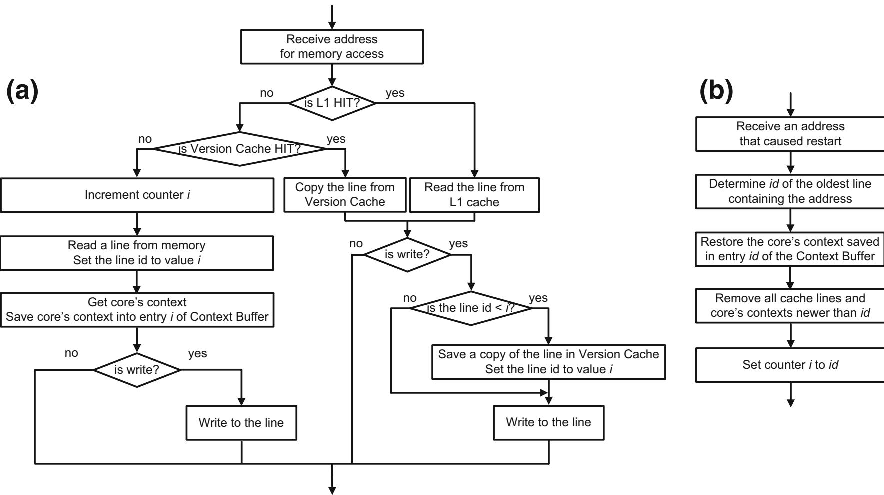

Fig. 3 Algorithms for performing **a** read, write, and **b** restart operations on RO-TCC implementation

appropriate parameters. The described approach assumes that all transactions have a similar probabilistic behavior.

The model relies on the following three assumptions:

1. Restart probability does not depend on where a transaction is, or whether the transaction has already been restarted.
2. Moments at which a transaction accesses memory locations for the first time are uniformly distributed within the transaction.
3. Moments at which a transaction accesses memory are uniformly distributed within the transaction (both first and subsequent access).

The model is defined with nine input parameters:

- The average transaction execution time L, in the absence of restarts.
- The average time between transactions V.
- Access-set size K (data accessed whether for read, write, or both).
- Write-set size K_w.
- Read-set size K_r.
- Number of subsequent read and write operations B (possibly repeatedly with the same data) that a transaction needs to successfully finish before it commits.
- Working-set size U, representing the total number of available transactional data.
- Probability that an access is a write operation P_w, or a read operation $P_r = 1 - P_w$.
- The number of cores N.

4.1 Transaction Restart Probability Model

The execution time of a set of transactions depends on how often they restart. The average number of times a transaction retries before it successfully commits could be

modeled with the appropriate restart probability R. This section develops the analytical model for calculating R using the input parameters and given assumptions.

Let us consider transaction A that accesses n different memory locations during the interval x. The interval x ranges from 0 to L and represents the time passed from the start of transaction A assuming that no restarts occurred. If in the same moment x some other transaction B with a write-set K_w commits, a conflict will not occur if all n memory locations fall into the set $U - K_w$. The probability $\mathrm{f_{nr}}(n)$ that transaction A will not detect a conflict is determined with the number of different ways that n can be chosen from the set $U - K_w$, divided by the number of different ways that n can be chosen from the entire set U:

$$\mathrm{f_{nr}(n)} = \frac{\binom{\mathrm{U-K_w}}{\mathrm{n}}}{\binom{\mathrm{U}}{\mathrm{n}}} = \frac{\mathrm{(U-n)! \cdot (U-K_w)!}}{\mathrm{U! \cdot (U-n-K_w)!}} \tag{1}$$

Probability that transaction A accessed to exactly n different memory locations during interval x can be modeled as a Poisson random variable $P(n)$ with distribution:

$$\mathrm{P(n)} = \mathrm{e}^{-\overline{\mathrm{n}}} \cdot \frac{(\overline{\mathrm{n}})^{\mathrm{n}}}{\mathrm{n!}}$$

Using assumption 2, the average number of different memory locations accessed during interval x is:

$$\overline{\mathrm{n}} = \mathrm{K} \cdot \frac{\mathrm{x}}{\mathrm{L}}$$

Let $\mathrm{f_{nr}}(x)$ be the probability that transaction A will not restart at a moment x at which transaction B commits. The probability $\mathrm{f_{nr}}(x)$ can be expressed using the probability that transaction A has accessed exactly n different memory locations $\mathrm{P}(n)$ and the probability that transaction A will not restart if it has accessed n memory locations $\mathrm{f_{nr}}(n)$. Taking into consideration all possible values for n, the expression for $\mathrm{f_{nr}}(x)$ is:

$$\mathrm{f_{nr}(x)} = \sum_{\mathrm{n}=0}^{+\infty} \mathrm{P(n) \cdot f_{nr}(n)} = \sum_{\mathrm{n}=0}^{+\infty} \mathrm{e}^{-\overline{\mathrm{n}}} \cdot \frac{(\overline{\mathrm{n}})^{\mathrm{n}}}{\mathrm{n!}} \cdot \frac{\mathrm{(U-n)! \cdot (U-K_w)!}}{\mathrm{U! \cdot (U-n-K_w)!}}$$

Let us consider the event where transaction A executes during interval x in which s other independent transactions commit in the moments $x_1, \ldots, x_\mathrm{s}$. The conditional probability that transaction A will not restart until moment x if it does not restart in any moment $x_1, \ldots, x_\mathrm{s}$ is:

$$\mathrm{P_{nrs}(x_1, x_2, \ldots, x_s | x) = f_{nr}(x_1) \cdot f_{nr}(x_2) \cdot \ldots \cdot f_{nr}(x_s)}$$

The mathematical expectation that transaction A will not restart until moment x is:

$$\begin{aligned} P_{nrs}(x) &= E(P_{nrs}(x_1, x_2, \ldots, x_s|x)) \\ &= \int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} g(x_1, x_2, \ldots, x_s|x) \cdot P_{nrs}(x_1, x_2, \ldots, x_s|x) dx_1 dx_2 \ldots dx_s \end{aligned}$$

where $g(x_1, \ldots, x_s|x)$ is the distribution density function of moments at which other s transactions commit. Following assumption 1 and assuming s independent transactions it can be stated that:

$$\begin{aligned} & g(x_1, x_2, \ldots, x_s|x) = g(x_1|x) \cdot g(x_2|x) \cdot \ldots \cdot g(x_s|x) \\ & g(x_i|x) \sim \mathrm{Unif}(0, x) \end{aligned}$$

Combining the last three expressions:

$$\begin{aligned} P_{nrs}(x) &= \left(\int_0^x \frac{1}{x} \cdot f_{nr}(x_1) dx_1 \right) \cdot \ldots \cdot \left(\int_0^x \frac{1}{x} \cdot f_{nr}(x_s) dx_s \right) \\ &= \left(\frac{1}{x} \cdot \int_0^x f_{nr}(x_1) dx_1 \right)^s = \left(\frac{F_{nr}(x)}{x} \right)^s \end{aligned}$$

where $F_{nr}(x)$ represents probability function that transaction A will not restart until moment x.

The probability that exactly s other transactions commit during interval x can be modeled as a Poisson random variable $P(s)$ with the distribution:

$$P(s) = e^{-\bar{s}} \cdot \frac{(\bar{s})^s}{s!}$$

where $\bar{s}$ is the average number of committed transactions during interval x. The value of the parameter $\bar{s}$ depends on the number of cores N, the average transaction execution time $E(T)$, and the average time between transactions V. During the time interval $E(T)+V$ all N concurrently executed transactions will commit. Among N concurrent transactions, besides transaction A, there are $N-1$ other concurrently executing transactions. The expression for the average number of committed transactions during interval x is:

$$\bar{s} = (N-1) \cdot \frac{x}{E(T)+V} = \frac{(N-1)}{\frac{E(T)}{L} + \frac{V}{L}} \cdot \frac{x}{L}$$

Let $P_{nr}(x)$ be the probability that transaction A will not restart until the moment x. The probability $P_{nr}(x)$ can be expressed using the probability $P_{nrs}(x)$ that transaction A will not restart during interval x until exactly s other transactions commit and the

probability $P(s)$ that exactly s other transactions commit without conflicting with transaction A. Taking into consideration all possible values for s, the expression for $P_{nr}(x)$ is:

$$P_{nr}(x) = \sum_{s=0}^{+\infty} P(s) \cdot P_{nrs}(x) = \sum_{s=0}^{+\infty} e^{-\bar{s}} \cdot \frac{(\bar{s})^s}{s!} \cdot \left(\frac{F_{nr}(x)}{x}\right)^s = e^{-\bar{s}} \cdot \sum_{s=0}^{+\infty} \frac{\left(\frac{\bar{s} \cdot F_{nr}(x)}{x}\right)^s}{s!}$$
$$= e^{-\bar{s}} \cdot e^{\frac{\bar{s} \cdot F_{nr}(x)}{x}}$$

The probability that transaction A will restart, R, is a complement of the event that transaction A will not restart during entire execution time, L, with probability $P_{nr}(L)$, which gives:

$$1 - R = P_{nr}(L)$$

furthermore:

$$R = 1 - e^{-\frac{(N-1)}{\frac{E(T)}{L} + \frac{V}{L}} \cdot \left(1 - \frac{F_{nr}(L)}{L}\right)} \tag{2}$$

4.2 Transaction Execution Time Model for TCC

This section develops the analytical model for calculating the expected transaction execution time, $E(T)$, for TCC implementation using the input parameters and the given assumptions. The expected execution time of a transaction can be calculated using:

$$E(T) = \sum_{i=0}^{+\infty} p_i \cdot \hat{T}_i \tag{3}$$

where p_i is a probability that the transaction restarts exactly i times, and $\hat{T}_i$ is the expected execution time of the transaction, if it restarts exactly i times.

Using assumption 1, the probability p_i that the transaction restarts exactly i times can be modeled with the geometric distribution with a parameter $1 - R$. The parameter $1 - R$ is the success probability in a sequence of Bernoulli trials and represents the probability that the transaction finishes without further restarts:

$$\begin{aligned} p_0 &= (1 - R) \\ p_1 &= R \cdot (1 - R) \\ &\dots \\ p_i &= R^i \cdot (1 - R) \end{aligned} \tag{4}$$

An expected execution time T_0 of a transaction that does not restart corresponds to the input parameter L:

$$T_0 = L$$

In the case of a transaction restart, the execution time includes the additional time spent during unsuccessful trials. When a transaction restarts exactly once at a location x_1 where the restart occurred corresponds to a random moment within the transaction and is in the interval from 0 to L. An expected execution time T_1 includes the time L needed for the successful completion of the transaction in the absence of restarts and the time interval x_1 spent during the unsuccessful trial:

$$T_1 = x_1 + L$$

For a transaction that restarts exactly i times during an expected execution time T_i includes the time L needed for the successful completion of the transaction in the absence of restarts and all i time intervals spent during unsuccessful trials:

$$T_i = x_1 + x_2 + \cdots + x_i + L = L + \sum_{j=1}^{i} x_j$$

All time intervals x_j correspond to restart moments within the transaction and are in the interval from 0 to L. The intervals x_j are random variables and according to the assumption 1 are mutually independent with uniform distribution. The expected execution time of the transaction, if it restarts exactly i times can be calculated:

$$\hat{T}_i = E(T_i) = L + \sum_{j=1}^{i} E(x_j)$$

$$x_j \sim \mathrm{Unif}(0, L)$$

$$E(x_j) = \int_0^L \frac{x_j}{L} \cdot dx_j = \frac{1}{L} \cdot \int_0^L x_j \cdot dx_j = \frac{L}{2}$$

$$E(T_i) = L + \sum_{j=1}^{i} \frac{L}{2} = L + i \cdot \frac{L}{2} = L \cdot \left(1 + \frac{i}{2}\right) \tag{5}$$

Combining Eqs. (3), (4), and (5) the expression for the expected transaction execution time E(T) is:

$$E(T) = \sum_{i=0}^{+\infty} R^i \cdot (1 - R) \cdot L \cdot \left(1 + \frac{i}{2}\right)$$

$$E(T) = \frac{2 - R}{2 \cdot (1 - R)} \cdot L \tag{6}$$

4.3 Transaction Execution Time Model for RO-TCC

This section develops the analytical model for calculating the expected transaction execution time, $E(T)$, for RO-TCC implementation using the input parameters and given assumptions. In both cases, TCC and RO-TCC, the probability p_i that the transaction restarts exactly i times is the same. In the case of RO-TCC, $E(T)$ differs in calculation of the expected execution time $\hat{T}_i$ of the transaction, if it restarts exactly i times.

As in TCC, the expected execution time T_0 of a transaction that does not restart corresponds to the input parameter L:

$$T_0 = L$$

In the case of a transaction restart, the execution time includes the additional time spent during the unsuccessful trials. When a transaction restarts exactly once, the location x_1 where the restart occurred corresponds to a random moment within the transaction and is in the interval from 0 to L. Contrary to the TCC, where the transaction restarts from the beginning, in the case of RO-TCC a location y_1, where the transaction restarts from is a random moment within the time interval of the unsuccessful trial and is in the interval from 0 to x_1. The expected execution time T_1 includes the time interval x_1 until a restart and the time L needed for the successful completion of the transaction in the absence of restarts diminished for y_1 from where the transaction continued with execution after the restart:

$$T_1 = x_1 + L - y_1 = x_1 - y_1 + L$$

In the case of a transaction restarting exactly twice, at a location x_1 where the first restart occurs and at a location y_1 from where the transaction continues with the execution, are determined in the same way as the case of the transaction restarting exactly once. After the first restart, the transaction does not execute a section of the transaction that comes before y_1 and therefore the location x_2 where the second restart occurs has to be after y_1. After the second restart, the location y_2 where the transaction restarts from is a random moment within the time interval from 0 to x_2.

$$T_2 = x_1 + x_2 - y_1 + L - y_2 = x_1 - y_1 + x_2 - y_2 + L$$

For a transaction that restarts exactly i times, the expected execution time T_i includes time L needed for the successful completion of the transaction in the absence of restarts and all i time intervals during the unsuccessful trials:

$$T_i = x_1 + x_2 + \cdots + x_i + L - y_1 - y_2 - \cdots - y_i$$

$$T_i = L + \sum_{j=1}^{i} (x_j - y_j)$$

All time intervals x_j and y_j are mutually dependent random variables and are modeled with uniform distributions on the defined interval. The expected execution time of the transaction, if it restarts exactly i times can be calculated:

$$\begin{aligned} &\hat{T}_i = E(T_i) = L + \sum_{j=1}^{i} (E(x_j) - E(y_j)) \\ &x_1 \sim \mathrm{Unif}(0, L) \\ &y_1 \sim \mathrm{Unif}(0, x_1) \\ &\ldots \\ &x_i \sim \mathrm{Unif}(y_{i-1}, L) \\ &y_i \sim \mathrm{Unif}(0, x_i) \end{aligned} \tag{7}$$

The mathematical expectations for the random variables x_i and y_i can be calculated as:

$$\begin{aligned} E(x_i) &= \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_1 dy_1 \ldots dx_i \\ E(y_i) &= \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_{y_{i-1}}^{L} \frac{1}{L - y_{i-1}} \int_0^{x_i} \frac{y_i}{x_i} dx_1 dy_1 \ldots dx_i dy_i \end{aligned} \tag{8}$$

Solving Eq. (8) for variable y_i shows a dependency between the mathematical expectations for x_i and y_i:

$$\begin{aligned} E(y_i) &= \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_{y_{i-1}}^{L} \frac{1}{L - y_{i-1}} \left(\int_0^{x_i} \frac{y_i}{x_i} dy_i \right) dx_1 dy_1 \ldots dx_i \\ E(y_i) &= \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_1 dy_1 \ldots dx_i \\ E(y_i) &= \frac{1}{2} \cdot E(x_i) \end{aligned} \tag{9}$$

Solving Eq. (8) for variable x_i shows a dependency between the mathematical expectations for x_i and y_{i-1}:

$$\begin{aligned} E(x_i) &= \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_i \right) dx_1 dy_1 \ldots dy_{i-1} \\ E(x_i) &= \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\frac{1}{L - y_{i-1}} \cdot \left(\frac{L^2}{2} - \frac{y_{i-1}^2}{2} \right) \right) dx_1 dy_1 \ldots dy_{i-1} \end{aligned}$$

$$E(x_i) = \frac{1}{2} \cdot L \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} dx_1 dy_1 \ldots dy_{i-1}$$
$$+ \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \cdots \int_0^{x_{i-1}} \frac{y_{i-1}}{x_{i-1}} dx_1 dy_1 \ldots dy_{i-1}$$
$$E(x_i) = \frac{1}{2} \cdot L + \frac{1}{2} \cdot E(y_{i-1}) \tag{10}$$

Combining Eqs. (9) and (10) results in a difference equation expressing the recurrence relation between the mathematical expectations for x_i and x_{i-1}:

$$E(x_i) = \frac{1}{2} \cdot L + \frac{1}{4} \cdot E(x_{i-1})$$

Solving the difference equation results in an expression for the mathematical expectation of random variable x_i:

$$E(x_i) = \frac{2}{3} \cdot \left(1 - \frac{1}{4^i}\right) \cdot L \tag{11}$$

According to Eq. (7), and using results from Eqs. (9) and (11) it can be calculated:

$$E(T_i) = \left(\frac{8}{9} + \frac{i}{3} + \frac{1}{9 \cdot 4^i}\right) \cdot L \tag{12}$$

Combining Eqs. (3), (4), and (12), the expression for the expected transaction execution time $E(T)$ on a RO-TCC system:

$$E(T) = \frac{(2 - R)^2}{(1 - R) \cdot (4 - R)} \cdot L \tag{13}$$

4.4 Model for Expected Relative Performance Gain

This section combines the results for the expected transaction execution time for TCC and RO-TCC with the aim to calculate the expected performance gain after the introduction of restart optimization. The calculation considers the execution time $E(T_R)$ of those transactions that restart at least once during the execution. The expected execution time, $E(T)$, of a transaction is equal to L in the case of the successful completion of the transaction in the absence of restarts, while in the opposite case, when the transaction restarts at least once, the execution time is equal to $E(T_R)$. The probability that the transaction restarts, R, is a complement of the event that the transaction does not restart during entire execution time with the probability $1 - R$. The relation between $E(T_R)$ and $E(T)$ can be expressed with:

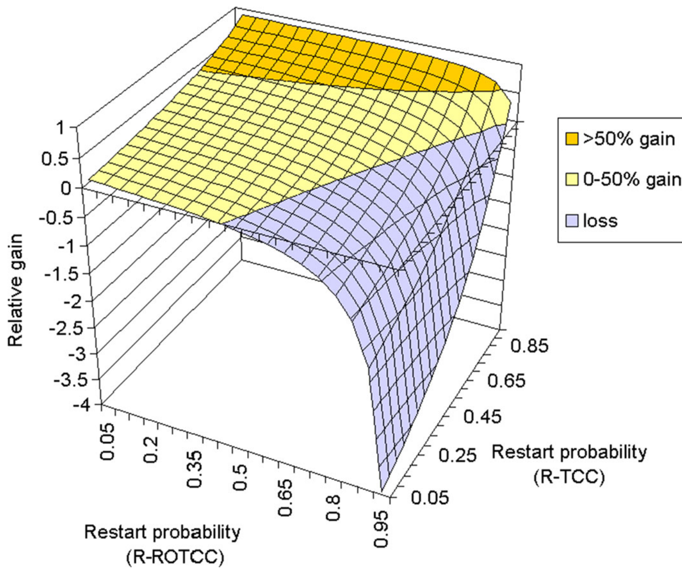

Fig. 4 The expected relative performance gain in a function of the restart probabilities

$$E(T) = (1 - R) \cdot L + R \cdot E(T_R)$$

that in the case of TCC can be combined with Eq. (6) to give:

$$E(T_{R-TCC}) = \frac{3 - 2 \cdot R_{TCC}}{2 \cdot (1 - R_{TCC})} \cdot L$$

or in the case of RO-TCC can be combined with Eq. (13) to give:

$$E(T_{R-ROTCC}) = \frac{5 - 5 \cdot R_{ROTCC} + R_{ROTCC}^2}{(1 - R_{ROTCC}) \cdot (4 - R_{ROTCC})} \cdot L$$

The expected relative performance gain is calculated as the relative difference of the expected execution times of those transactions that restart at least once during the execution for TCC and RO-TCC:

$$\delta_R = \frac{E(T_{R\text{-}TCC}) - E(T_{R\text{-}ROTCC})}{E(T_{R\text{-}TCC})}$$

A possible consequence of restart optimization is the difference of restart probabilities between TCC and RO-TCC. The difference comes from the fact that, after a restart in RO-TCC, the access buffer is not empty, and it can lead to new restarts occurring sooner, making the restart probability higher. However, after a restart in RO-TCC, a restarted transaction has less work to perform before it commits, leaving less time for new restarts to occur, making the restart probability lower. Based on the analytical model for the expected execution times $E(T_{R\text{-}TCC})$ and $E(T_{R\text{-}ROTCC})$, Fig. 4 shows

the expected relative performance gain δ_R for different values of restart probabilities R-TCC and R-ROTCC. Performance degradation could be expected in the case when the restart probability of RO-TCC is substantially higher than the restart probability of TCC.

Assuming that restart probabilities are approximately the same, an approximation that $R_{TCC} = R_{ROTCC} = R$ can be made. The approximation would enable easy assessment of the expected gain after the introduction of the proposed restart optimization into existing transactional memory systems with an experimentally determined restart probability. With the approximation, the expected performance gain δ_R is:

$$\delta_R = \frac{2 - R}{(3 - 2 \cdot R) \cdot (4 - R)} \tag{14}$$

The results for δ_R demonstrate that in cases when the restart probability remains the same after the introduction of restart optimization, the expected relative gain for transactions that have been restarted at least once is a monotonically increasing function with values in a range from 0.17 to 0.33 for R ranging from 0 to 1. Even though Eq. (14) does not demonstrate a strong sensitivity to small variations in R, Eqs. (6) and (13) imply that for higher values of restart probability even small variations lead to a large deviation in calculation of the expected transaction execution time and relative gain.

4.5 Spatial Complexity Model

Assessing restart optimization effectiveness depends on the complexity of the implementation. The complexity can be viewed from two aspects, temporal and spatial. Temporal complexity is defined as the time needed to store/restore the proper context, while spatial complexity is defined as the space required to store different versions of core's contexts and cache lines. The influence of time complexity can be disregarded if the time needed for storing/restoring core's context into/from Context Buffer is less than the time needed for fetching a cache line during a miss.

The spatial complexity is determined by Context Buffer size and Version Cache size. Context Buffer size depends on a number of core's contexts that need to be stored and is equal to a product of access-set size K and size of a core's context. Version Cache size depends on a number of versions created for speculatively modified cache lines. A new version of a cache line is created when a write to an older cache line occurs. The worst case for the spatial complexity is when each fetch of a new cache line is followed by write accesses to all cache lines that are already present in the cache, thus requiring new versions for each write. In the average case it can be expected that write accesses are done only to some cache lines. The probability P_{wr} for a write access to a cache line that is already present in the cache can be calculated using probability P_w that an access is a write operation. The write probability P_w can be expressed as a ratio between the number of write operations B_w, and the number of subsequent read and write operations that a transaction needs to successfully finish before it commits B:

$$P_w = \frac{B_w}{B} = \frac{B_{fw} + B_{nfw}}{B} = \frac{B'_{fw} + B''_{fw} + B_{nfw}}{B}$$

The number of write operations B_w can be expressed as the sum of write miss B_{fw} and write hit operations B_{nfw}. Write miss operations can be divided into B'_{fw} operations that write data that will never be read from within the current transaction (e.g., result data), and B''_{fw} operations that write data that may be read from or written to with the probability P_{wr} again within the current transaction (e.g., intermediate data). B_{fw} operations are divided into B'_{fw} and B''_{fw} operations because read-set K_r and write-set K_w can overlap ($0 \leq K_r \leq K$; $0 \leq K_w \leq K$; $K \leq K_r + K_w$):

$$\begin{aligned} B'_{fw} &= K - K_r \\ B''_{fw} &= P_{wr} \cdot (K_r + K_w - K) \\ B_{nfw} &= P_{wr} \cdot (B - K) \end{aligned}$$

Therefore the expression for the probability P_{wr} for a write access to a cache line that is already present in the cache is:

$$P_{wr} = \frac{P_w \cdot B - (K - K_r)}{B - 2 \cdot K + K_r + K_w}$$

The required space for holding different versions can be estimated by determining the number of new versions added during a period between two cache misses. The number of new versions can be calculated as the number of different cache lines that are written to during the period excluding the cases of writes to the data that caused the cache miss. Assuming that the cache contains n valid lines during the period, the number of new versions $m(n, w)$ added after w write operations can be calculated according to the Generalized Birthday problem:

$$m(n, w) = n \cdot \left(1 - \left(1 - \frac{1}{n}\right)^w\right) - \left(1 - \left(\frac{n-1}{n}\right)^w\right) = (n-1) \cdot \left(1 - \left(1 - \frac{1}{n}\right)^w\right)$$

The probability that the transaction has performed exactly w writes between two cache misses while the cache contains n valid cache lines can be modeled as the Poisson random variable $P(w)$ with the distribution:

$$P(w) = e^{-\overline{w}} \cdot \frac{(\overline{w})^w}{w!}$$

The average number of write operations $\overline{w}$ between two misses can be calculated using the probability P_{wr} for the write access to a cache line that is already present in the cache, the number of 'cache hit' accesses $B - K$, and the number of 'cache miss' accesses K. The expression for the average number of write operations is:

$$\overline{w} = P_{wr} \cdot \frac{B - K}{K}$$

Let $m(n)$ be the average number of new versions added between two cache misses in the case when the cache contains n valid cache lines. The number $m(n)$ is calculated by using the probability $P(w)$ that there is exactly w write operations between two misses and the average number of new versions added in the case when there are exactly w operations while the cache contains n valid lines $m(n, w)$:

$$\mathrm{m(n)} = \sum_{\mathrm{w}=0}^{+\infty} \mathrm{P(w)} \cdot \mathrm{m(n, w)}$$

The size can be expressed as the sum of all new versions that are added between the two misses:

$$\mathrm{m} = \sum_{\mathrm{n}=1}^{\mathrm{K}} \mathrm{m(n)}$$

The expression for calculating m can be simplified using the set of approximations:

$$\mathrm{m(n)} \approx \mathrm{m(n, \overline{w})} = (\mathrm{n} - 1) \cdot \left(1 - \left(1 - \frac{1}{\mathrm{n}}\right)^{\overline{\mathrm{w}}}\right) \approx (\mathrm{n} - 1) \cdot \left(1 - \left(1 - \frac{\overline{\mathrm{w}}}{\mathrm{n}}\right)\right)$$

$$\mathrm{m} \approx \frac{\mathrm{P_w} \cdot \mathrm{B} - (\mathrm{K} - \mathrm{K_r})}{\mathrm{B} - 2 \cdot \mathrm{K} + \mathrm{K_r} + \mathrm{K_w}} \cdot (\mathrm{B} - \mathrm{K}) \tag{15}$$

5 Results of Evaluation

The evaluation of the proposed RO-TCC and comparison with TCC were completed with a simulation based on the model described in Sect. 2, and the appropriate analytical models from Sect. 3. The simulation also served the purpose of verification of the proposed analytical models. In all experiments, the simulation parameters were set to values of a magnitude as they might occur in typical benchmark applications [25,26]. Table 2 shows the value ranges and default values for all simulation parameters. Each experiment varied only one simulation parameter, while other parameters were set to default values. This section presents experiments that characterize the essence of the proposed solution.

As opposed to the proposed analytical model, which models one transaction and its interaction with the rest of the system, the simulation model implements N transactions explicitly as concurrent processes. Each simulated transaction initially selects a sequence of B memory read and write accesses, with the write probability $\mathrm{P_w}$. The sequence is maintained throughout the lifetime of the transaction and even after a restart the transaction retries the identical sequence of operations. The memory accesses are uniformly distributed over the entire duration of the transaction L. The moments when a transaction accesses data for the first time is important from the aspect of RO-TCC. For the purposes of the simulation, the moments were uniformly distributed over the entire duration of the transaction. After a commit of a transaction, the core that executed the transaction starts a new transaction after the expiration of the period, V.

Table 2 Value ranges and default values for all simulation parameters

Parameter	Name	Typical range[a]	Default value[b]
Time outside of a transaction	V/L	0.03–32.3	0.1
Access set	K	20–800	600
Read set	K_r	10–800	400
Write set	K_w	10–800	300
Number of memory accesses	B	15–220,000	6000
Working set	U	10,000–3,000,000	40,000
Write probability	P_w	0.05–0.49	0.3
Number of cores	N	2–32	4

[a] Typical range in benchmark applications
[b] Default value in the experiments

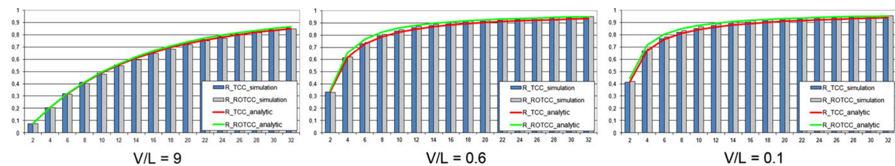

Fig. 5 Analytical and simulation results for restart probability depending on the number of cores for three different values of the parameter V/L

New transactions start until the number of restarts does not become sufficient for a confidence level of 95 % and a relative error of 5 %.

Beside a simulation based on the model described in Sect. 2, this section also presents an experiment that uses an architectural simulator. The architectural simulation enables characterizing the behavior of the proposed solution by executing the benchmark tests using configurable cache memory and servers a purpose of reevaluation.

5.1 Simulation Analysis of the Transaction Restart Probability Model

The analytical model shows that the transaction restart probability, R, depends on the number of parameters including the number of cores, the percentage of time a core spends in transactions, the size of working, access, write, and read sets. In order to validate the analytical model, the experiments were conducted, varying all parameters. The simulation shows that the predominant parameter for the restart probability calculation is the number of cores. Increasing the number of cores leads to greater congestion that results in an increased number of restarts. In the experiments, the number of cores ranged from 2 to 32 while all other parameters had default values as given in Table 2.

Figure 5 demonstrates how the restart probability for TCC and RO-TCC depends on the number of cores. The bars correspond to the simulation results, while the lines correspond to the analytical results. The figure shows that the analytical model accurately reflects the behavior of simulated systems with an average relative error of 1 %. The figure also shows that the restart probability enters saturation. In the case

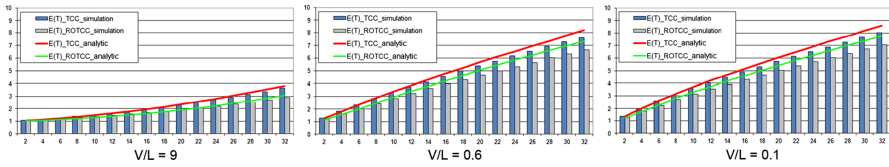

Fig. 6 Analytical and simulation results for normalized execution time depending on the number of cores for three different values of the parameter V/L

of parameters that correspond to the average load, the saturation occurs when the number of cores approaches 16. The difference between the simulation results for restart probability, R, obtained for TCC and RO-TCC indicates that in some cases the approximation $\mathrm{R_{TCC}} = \mathrm{R_{ROTCC}} = \mathrm{R}$ can be used.

5.2 Simulation Analysis of the Transaction Execution Time Model

The analytical expression obtained in Sect. 3 represents a dependency between expected transaction execution time $\mathrm{E}(T)$ and the transaction restart probability, R. Similar to the simulation analysis of the transaction restart probability, the predominant parameter is the number of cores. Figure 6 shows how the expected transaction execution time for TCC and RO-TCC depends on the number of cores. The transaction execution time overhead resulting from the transaction restarts has approximately linear dependency on the number of cores in the examined range (from 2 to 32). The analytical model accurately reflects the behavior of the simulated system with a relative error of less than 10 %.

5.3 Simulation Analysis of the Expected Performance Gain

Relative gain, δ, represents the expected performance gain after the introduction of restart optimization in TCC system. The relative gain is directly dependent on the expected transaction execution time. Section 4.2 demonstrates that the analytical model for $\mathrm{E}(T)$ has a relative error of up to 10 %, which results in calculating the relative gain with a relative error below 20 %. Calculating relative gain depends not only on effects of the superposition of the errors of two models (TCC and RO-TCC), but also on the fact that the transaction restart probability is not the same in these two models.

Figure 7 shows how the expected relative gain depends on the number of cores. Regardless of the differences between the experimental and analytical models, the figure shows that restart optimization brings a relative gain of around 10 %. A similar conclusion about the relative gain can also be derived from Fig. 8, which shows the dependency between the relative gain and the working set size.

5.4 Simulation Analysis of Spatial Complexity

Restart optimization depends on keeping track of multiple versions during transaction execution making space a critical resource for achieving the expected performance

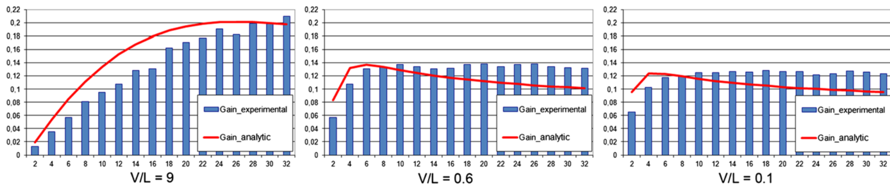

Fig. 7 Analytical and simulation results for relative gain depending on the number of cores for three different values of the parameter V/L

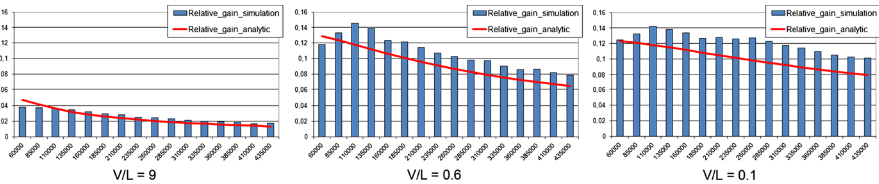

Fig. 8 Analytical and simulation results for the relative gain depending on the working set size for three different values of the parameter V/L

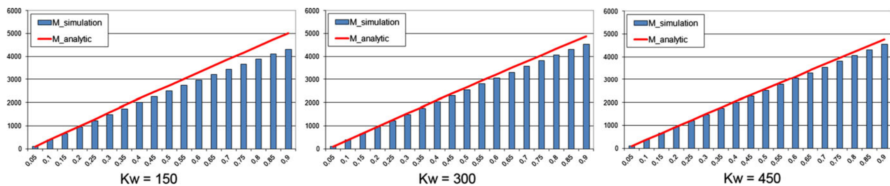

Fig. 9 Analytical and simulation results for the required space depending on the write probability for three different values of the parameter *Kw*

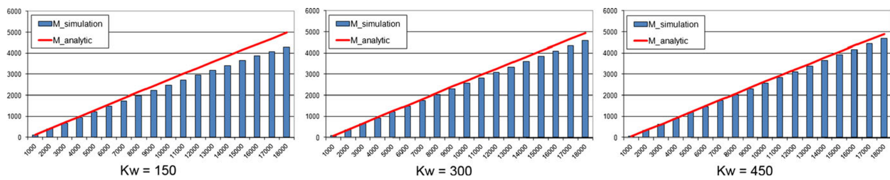

Fig. 10 Analytical and simulation results for the required space depending on the number of memory accesses for three different values of the parameter *Kw*

gain. The analytical model shows that the spatial complexity depends on the number of parameters including the number of memory accesses, write probability, and size of access, write, and read sets. Figures 9 and 10 show how the required space depends on the write probability and the number of memory accesses. Both figures show approximately linear dependency, which in the case of Fig. 9 can be explained by the fact that the access set size is almost equal to the sum of the write and read set sizes. The analytical model accurately reflects required space and has a relative error of less than 1 %.

Table 3 Configuration used for simulation with the architectural simulator

Parameter	Value
L1 data cache size	8 KB
L1 data cache block size	32-Byte blocks
L1 associativity	4-Way associative
Main memory miss penalty	96 cycles
Version cache hit time	1 cycle

5.5 Reevaluation with Architectural Simulation

The reevaluation, performed as a separate experiment, aims to revisit previous findings. In the contrast to the previous experiments, where the model simulator was used, the reevaluation is conducted by using an architectural simulator. The architectural simulator consists of the X86 processor and the configurable cache memory simulator. The X86 processor is part of the JPC simulator that simulates full PC-based system at the behavior level [27]. The JPC full system simulator includes the processor, motherboard, disk controller, graphics controller and user interface, but for the purpose integration with the configurable cache memory only the processor is used [28]. The configurable cache memory simulator operates at the register transfer level, provides statistical data, and logs complete memory access trace. For the purpose of the reevaluation the configurable cache memory simulator has been modified to support TCC and RO-TCC.

The architectural simulations were conducted using the STAMP benchmark with parameters set to default values. The benchmark programs were compiled in debug mode using MS Visual Studio 6.1. The simulations used the configuration of the architectural simulator given in Table 3. During simulations, the sizes of Context Buffer and Version Cache were unlimited in order to determine the maximal performance gain.

Figures 11 and 12 show the relative gain depending on the benchmark test and a number of cores used in simulation. The relative gain is calculated according to execution time spent in all transactions (Fig. 11) and execution time spent in transactions that restarted at least once (Fig. 12). Regardless of the differences between the model and architectural simulation, Fig. 12 shows that results are within expected ranges. The differences can be attributed to the fact that the transactions in benchmark tests may have different distributions than the one used in analytical model. The results obtained in case of ssca2 test show that the execution time for TCC and RO-TCC had negligible differences due to the fact that the test had low contention level and that all restarts required re-execution from the beginning of transactions. The experiment also showed that RO-TCC had more restarts than TCC, which confirms that a possible consequence of the restart optimization can be a higher restart probability. However, in case of Intruder test although the restart probability was almost double in case of RO-TCC, the restart optimization brought the relative gain of 29 % when observing all transactions, or 49 % when observing only transactions that restarted at least once.

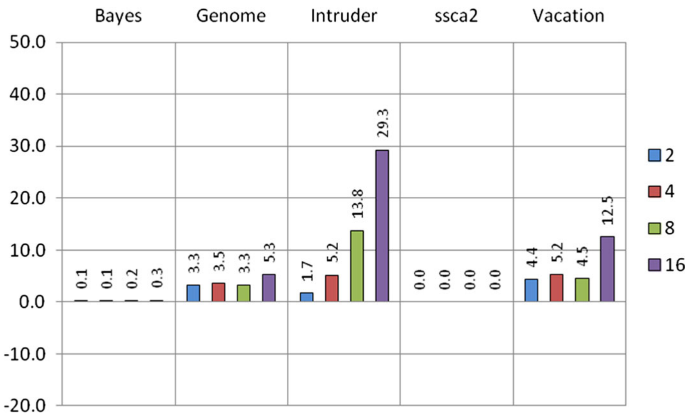

Fig. 11 Architectural simulation results for relative gain according to execution time spent in transactions depending on the benchmark test and a number of cores

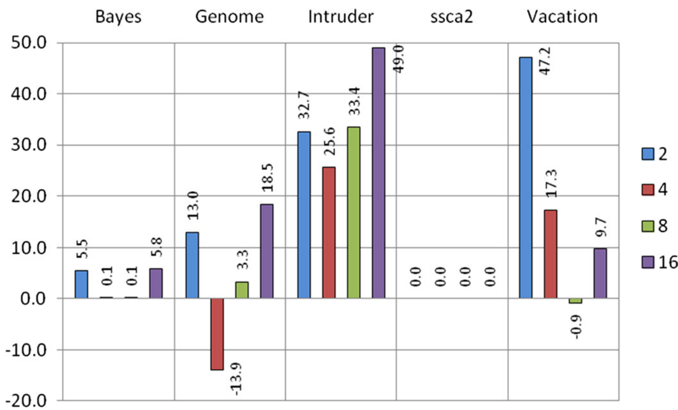

Fig. 12 Architectural simulation results for relative gain according to execution time spent in transactions that restarted at least once depending on the benchmark test and a number of cores

5.6 Threats to Validity

The correctness of the proposed analytical model depends on the justification of the adopted assumptions. For example, assuming that the locations within a transaction where the restart occurred and where the transaction restarts from are random moments with uniform distributions on the defined interval may not model real situations adequately and consequently limits the generalizability of the findings. Similarly, a workload chosen for evaluation purposes was based on the STAMP benchmark, which even though widely used, may not represent the proper choice for some applications. Moreover, the evaluation was based on the results gained with a model simulator, which does not produce a fine level of information, and therefore may represent a threat to the accuracy of the findings. Although the reevaluation with an architectural simulator confirmed that relative gains are in expected ranges, the usage of the

architectural simulator based on JPC that represents a single architecture may also be considered as a threat to validity of the findings.

6 Conclusion

The goal of transactional memory is to simplify multi-core concurrent programming. A programmer demarcates a transaction while the transactional memory provides conflict resolution for all concurrently executing transactions. This paper has introduced an optimization for transactional memory with lazy conflict detection. The proposed optimization builds on previous research and differs in that it eliminates the need for the prediction of conflicting accesses and introduces incremental context saving. The application of the proposed optimization on the TCC was analytically modeled using a continuous-time model.

The comparison of the analytical results with the results of simulations demonstrated both the accuracy and the capability of the proposed analytical model to reveal the trend behavior of transactions with and without the restart optimization as well as their relative performance. The simulation results indicate that the optimization does not have a significant influence on the value of the restart probability. However, at higher values of restart probability even small variations lead to large deviations in the calculation of the expected transaction execution time and relative gain. The results for the analytical model show that in cases when the restart probability remains the same after the introduction of restart optimization, the expected relative gain for transactions that have been restarted at least once is a monotonically increasing function with values ranging from 0.17 to 0.33. Even though the proposed optimization and its analytical model were developed to fulfill the specific requirements of the hardware transactional memory with lazy conflict detection, the generality of the presented approach allows its applicability to other types of transactional memory with lazy version management regardless of the conflict detection strategy.

Acknowledgements This work was supported by the Ministry of Education, Science, and Technological Development of the Republic of Serbia (III44009, TR32047). The authors gratefully acknowledge the support.

References

1. McDonald, A., Carlstrom, B.D., Chung, J., Minh, C.C., Chafi, H., Kozyrakis, C., Olukotun, K.: Transactional memory: the hardware–software interface. IEEE Micro **27**(1), 67–76 (2007)
2. Harris, T., Cristal, A., Unsal, O.S., Ayguade, E., Gagliardi, F., Smith, B., Valero, M.: Transactional memory: an overview. IEEE Micro **27**(3), 8–29 (2007)
3. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: log-based transactional memory. In: The Twelfth International Symposium on High-Performance Computer Architecture, pp. 254–265 (2006)
4. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. IEEE Micro **26**(1), 59–69 (2006)
5. Meunier, Q.L., Pétrot, F.: Lightweight transactional memory systems for NoCs based architectures: design, implementation and comparison of two policies. J. Parallel Distrib. Comput. **70**(10), 1024–1041 (2010)

6. Bobba, J., Moore, K., Volos, H., Yen, L., Hill, M.D., Swift, M., Wood, D.A.: Performance pathologies in hardware transactional memory. IEEE Micro **28**(1), 32–41 (2008)
7. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06), pp. 253–262 (2006)
8. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)
9. Vallejo, E., Sanyal, S., Harris, T., Vallejo, F., Beivide, R., Unsal, O., Cristal, A., Valero, M.: Hybrid transactional memory with pessimistic concurrency control. Int. J. Parallel Prog. **39**(3), 375–396 (2011)
10. Sonmez, N., Arcas, O., Pflucker, O., Unsal, O.S., Cristal, A., Hur, I., Singh, S., Valero, M.: TMbox: a flexible and reconfigurable 16-core hybrid transactional memory system. In: IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 146–153 (2011)
11. Hammond, L., Carlstrom, B.D., Wong, V., Chen, M., Kozyrakis, C., Olukotun, K.: Transactional coherence and consistency: simplifying parallel hardware and software. IEEE Micro **24**(6), 92–103 (2004)
12. Waliullah, M.M., Stenstrom, P.: Intermediate checkpointing with conflicting access prediction in transactional memory systems. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp. 1–11 (2008)
13. Waliullah, M.M., Stenstrom, P.: Removal of conflicts in hardware transactional memory systems. Int. J. Parallel Prog. **42**(1), 198–218 (2014)
14. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06), pp. 227–238 (2006)
15. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: Hardware signature designs to deal with asymmetry in transactional data sets. IEEE Trans. Parallel Distrib. Syst. **24**(3), 506–519 (2013)
16. Tomic, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., Unsal, O., Harris, T., Valero, M.: EazyHTM: eager-lazy hardware transactional memory. In: Proceedings of the 42nd International Symposium on Microarchitecture, pp. 145–155 (2009)
17. Lupon, M., Magklis, G., Gonzalez, A.: FASTM: a log-based hardware transactional memory with fast abort recovery. In: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09), pp. 293–302 (2009)
18. Ros, A., Acacio, M., Garcia, J.M.: A direct coherence protocol for many-core chip multiprocessors. IEEE Trans. Parallel Distrib. Syst. **21**(12), 1779–1792 (2010)
19. Heindl, A., Pokam, G.: An analytic framework for performance modeling of software transactional memory. Comput. Netw. **53**(8), 1202–1214 (2009)
20. Heindl, A., Pokam, G., Adl-Tabatabai, A.: An analytic model of optimistic software transactional memory. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), pp. 153–162 (2009)
21. Heindl, A., Pokam, G.: An analytic model for optimistic STM with lazy locking. In: Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications (ASMTA '09), pp. 339–353 (2009)
22. He, Z., Hong, B.: Modeling the run-time behavior of transactional memory. In: IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 307–315 (2010)
23. Sanzo, P., Di Ciciani, B., Palmieri, R., Quaglia, F., Romano, P.: On the analytical modeling of concurrency control algorithms for software transactional memories: the case of commit-time-locking. Perform. Eval. **69**(5), 187–205 (2012)
24. Poe, J., Chang-Burm, C., Tao, L.: Using analytical models to efficiently explore hardware transactional memory and multi-core co-design. In: The 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '08), pp. 159–166 (2008)
25. Minh, C.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 (2008)
26. Hughes, C., Poe, J., Qouneh, A., Tao, L.: On the (dis)similarity of transactional memory workloads. In: IEEE International Symposium on Workload Characterization (IISWC 2009), pp. 108–117 (2009)

27. Newman, R., Dennis, C.: JPC: an x86 PC emulator in pure Java. In: Spinellis, D., Gousios, G. (eds.) Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design, pp. 199–234. O'Reilly Media, Sebastopol (2009)
28. Radivojevic, Z., Cvetanovic, M.: Integration of the JPC simulator into the configurable cache memory simulator. In: Proceedings of the 54th ETRAN Conference (ETRAN LIV), pp. RT.4.10.1–RT.4.10.4 (2010)