Automatic Optimization of Software Transactional Memory Through Linear Regression and Decision Tree

Yang Xiao[1], Zhen Li[2], Ehsan Atoofian[1(✉)], and Ali Jannesari[2]

[1] Electrical Engineering Department, Lakehead University, Thunder Bay, Canada
{yxiao4, atoofian}@lakeheadu.ca
[2] Technical University of Darmstadt, Darmstadt, Germany
{li, jannesari}@cs.tu-darmstadt.de

Abstract. Software Transactional Memory (STM) is a promising paradigm that facilitates programming for shared memory multiprocessors. In STM, synchronization of accesses to the shared memory locations is fully handled by STM library and does not require any intervention by programmers. While STM eases parallel programming, it results in run-time overhead which increases execution time of certain applications. In this paper, we focus on overhead of STM and propose optimization techniques to enhance speed of STM applications. In particular, we focus on size of transaction, read-set, and write-set and show that execution time of applications significantly changes by varying these parameters. Optimizing these parameters manually is a time consuming process and requires significant labor work. We exploit Linear Regression (LR) and propose an optimization technique that decides on these parameters automatically. We further enhance this technique by using decision tree. The decision tree improves accuracy of predictions by selecting appropriate LR model for a given transaction. We evaluate our optimization techniques using a set of benchmarks from NAS and DiscoPoP benchmark suites. Our experimental results reveal that LR and decision tree together are able to improve performance of STM programs up to 54.8 %.

Keywords: Software Transactional Memory · Linear Regression · Decision tree · Performance

1 Introduction

Software Transactional Memory (STM) is becoming increasingly popular as a convenient way for writing parallel programs. STM provides an atomic construct, called transaction, which is used to protect shared memory locations from concurrent accesses by threads. Reads and writes to transactional data occur at a single instance of time. Intermediate transactional values are not visible to other transactions. STM executes transactions speculatively in parallel and monitor memory locations accessed by active transactions. If executing transactions do not conflict over shared memory locations, then they safely commit. However, in the event of conflict, only one transaction can

G. Wang et al. (Eds.): ICA3PP 2015, Part IV, LNCS 9531, pp. 61–73, 2015.
DOI: 10.1007/978-3-319-27140-8_5

proceed and the rest should abort and restart. Transactions log operations during the execution so that they can restore state of the running program if roll-back is needed.

STM eliminates many of the problems associated with locks and enables programmers to compose scalable applications safely. In an STM program, a programmer does not need to worry about priority inversion, deadlock, or live lock. This is in contrast to lock-based programming in which a programmer needs to deal with lock placement and synchronization bugs. In an STM program, the programmer only needs to reason locally about shared memory locations and mark sections of the program that should be executed concurrently. The underlying system guarantees correctness. In addition to ease of programming, STMs are speculative in nature. The benefit of speculative approach is that transactions do not need to wait for shared memory locations; instead, they can execute concurrently and modify disjoint memory locations safely, leading to performance gains.

In the last decade, there have been several implementations of STMs [2–4]. The emergence of new STM algorithms has not been slowed down in the recent years, and the support for transactional memory in new processors [6] is likely to increase the number of TM implementations. The performance of STMs depends on several factors such as lock acquisition time, granularity of conflict detection, the mapping of memory addresses to the lock table, etc. Some researchers have explored design space of STMs and proposed changing STM parameters during the run-time. For example, Marathe et al. [5] studies lock acquisition in STMs and showed that the time at which locks are acquired has drastic impact on scalability. While eager policy (encounter-time locking) reduces overhead, lazy policy (commit-time locking) provides better throughput for some multithreaded applications. Marathe et al. [5] proposed an adaptive technique which dynamically changes lock acquisition policy in run-time. The other example is granularity of conflict detection [4]. Felber et al. [4] showed that performance of STMs varies with granularity of conflict detection and non-optimum parameterization can slow down some programs by a factor of three. While the above techniques improve performance of STMs, all of them focus on execution of STM programs during the run-time. They do not provide any guidelines for programmers to write an efficient TM program in the first place.

The first step in writing an STM program is marking regions of a sequential code as transactions. In the next step, APIs such as TM_BEGIN() and TM_END() [2] which are provided by an STM library are inserted into the program to guarantee atomicity and correctness of transactions. The size of a transaction has significant impact on performance. If the transaction is too short, then the overhead of STM APIs exceeds performance gain of parallel execution and may lead to an STM program which is slower than sequential version of the program. On the other side, if the transaction is too large, then the cost of roll-back in applications with high abort rate may reduce speed-up in STM applications.

One way to find optimal transaction size is using try and error approach. A programmer can vary transaction size and finds out the optimal transaction size by running the program multiple times. This procedure is very time consuming and requires significant programming effort. To address this challenge, we propose two optimization techniques that automatically determine near optimal transaction size: the first technique exploits Linear Regression (LR) [8] to predict transaction size. LR receives

parameters of a non-optimized transaction such as transaction size, read-set size, and write-set size and predicts the optimum transaction size. While LR is simple to implement, its accuracy is low. Our second optimization technique exploits decision tree and enhances accuracy of predictions. The decision tree divides transactions into multiple groups and then uses a different LR model for each group. Using a set of benchmarks from NAS [9] and DiscoPoP [10], we show that decision tree and LR together increase accuracy of predictions significantly.

The rest of the paper is organized as follows. In Sect. 2, we explain the necessary background for our optimization techniques and discuss how LR and decision tree work. Section 3 explains the intuition behind our optimization techniques and evaluates sensitivity of STM programs to a few transactional parameters. Section 4 discusses our optimization techniques in details and reports experimental results. We review related work in Sect. 5. Finally, in Sect. 6, we offer concluding remarks.

2 Background

2.1 Linear Regression

Linear Regression (LR) is a mathematical equation which relates a response variable to a set of input parameters for a given design space [8]. LR is widely used to predict the response variable at an arbitrary point in the design space. Equation 1 shows a simple model for LR:

$$y = B_0 + \sum_{k=1}^{q} (B_k \times x_i) + \varepsilon \tag{1}$$

where y is response variable, x_i is input parameter, B_0 is the intercept of the fit with the y-axis, and ε is the error of LR model. B_i ($0<i$) is coefficient and represents the expected change in y per unit change in x_i. LR uses least square method to find the best-fitting curve to a set of test points. In this method, coefficients are calculated so that the sum of square of the errors for the test points (error of a test point is the distance of the point from the fitting curve) is minimized. While LR exploits a simple model for prediction, it shows excellent results in many applications and is able to predict the response variable with high accuracy. Examples of LR applications are prediction of stock market, oil price, and GDP [8]. Also, recently, Google used LR to predict revenue of a movie four weeks ahead of its release date [13].

Our goal in this paper is to accurately estimate transaction size so that execution time of STM applications is reduced. To do so, we explore static parameters that interact with performance. We achieve this by performing simulation-based experiments in which input parameters are varied before code compilation and the resulting transaction size is fitted as per Eq. 1. It is important to note that while parameters in Eq. 1 depend on STM library, the methodology that we use is general and can be applied to any STM implementation.

2.2 Decision Tree

Classification is the task of assigning objects to a set of predefined categories. Decision tree [17] is a popular approach for classification. Originally, decision tree was used in the field of statistics. However, soon it found to be effective in many other disciplines such as machine learning, image processing, etc. A decision tree classifies an input object through a set of functions organized in a hieratical manner and represented by a tree. A tree has three types of nodes: root, internal, and leaf [17]. An internal node splits the objects into two categories according to a test function. The inputs to the function are attributes of the object and the output of the function is a binary value: 0 or 1. A leaf represents a category. Objects are classified by navigating them from root down to the leaves, based on the output of the test functions along the path.

In this work, we use decision tree to classify transactions based on error of predicted transaction size. Objects in decision tree are transactions and attributes of the objects are read-set size, number of instructions between two consecutive transactions, etc. The decision tree predicts the error of transaction size to be predicted by LR. Section 4 discusses details of decision tree used in this study.

2.3 Benchmark Selection

To evaluate an STM system, researchers rely on a set of benchmarks. If the set of the benchmarks are selected from a specific field, then the outcome of the research is not reliable. To be able to extend the outcome of a research project to the real world applications, we need a set of comprehensive benchmarks that truly represent real world applications.

Asanovic et al. [18] proposed 13 Dwarfs as a guideline to develop benchmark suites for parallel applications. A dwarf is a high level abstraction which categorizes applications based on patterns of computation and communication. Asanovic et al. [18] showed that NAS benchmark suite [9] includes all those dwarfs and so in this work, we use NAS benchmark suite to evaluate our optimization techniques.

3 Motivation

In an STM program, transactions are implemented through APIs provided by an STM library. While STM does not require any changes in the architectural level, it may not result in significant speedup. This is mainly due to the overhead of STM APIs.

In this work, we use two Intel Xeon E5660 processors running at 2.8 GHz. Each processor has six cores and is capable of running up to 12 threads simultaneously. Each processor has a 12 MB shared L3 cache with 64B cache lines. Each core has a 32 KB instruction cache and a 32 KB data cache.

Figure 1a shows execution time of STM relative to sequential code for NAS benchmark suite. NAS benchmarks are originally developed using OpenMP library. We replaced critical sections in NAS with transactions. For each benchmark, the number of threads varies from two to 16. Bars more than one show speedup. On average, STM reduces performance by 43.8 %, 57.5 %, 59.1 %, and 81.7 %, when the

number of threads is 2, 4, 8, and 16, respectively. From Fig. 1a, we conclude that blindly using transactions in a parallel program may result in a program that is slower than its sequential version. To boost performance of STM programs, we need to reduce the overhead of APIs. There are two main approaches to optimize STMs: static and dynamic. In static approach, the STM program is changed during the code development or compilation whereas in dynamic approach, the system is optimized during the run-time and by hacking into STM library. While many research ideas have been proposed on the latter approach, the former one did not receive enough attention from researchers. In this section, we discuss three static parameters which impact performance of STM programs: size of transaction, write-set size, and read-set size.

Figure 1b shows the impact of size of transaction on performance of NAS benchmarks. We optimized each benchmark by varying the number of instructions in transactions manually and selecting the one which minimizes execution time. It is important to note that by changing the size of a transaction, we do not violate its atomicity. Figure 2 shows an example of a large transaction. The transaction is composed of three loops. We can decompose the outer loop into several smaller loops and assign each loop to a transaction. Similarly, when we combine smaller transactions to build a large transaction, we take into account the atomicity of transactions to make sure that we do not compromise correctness of transactions.

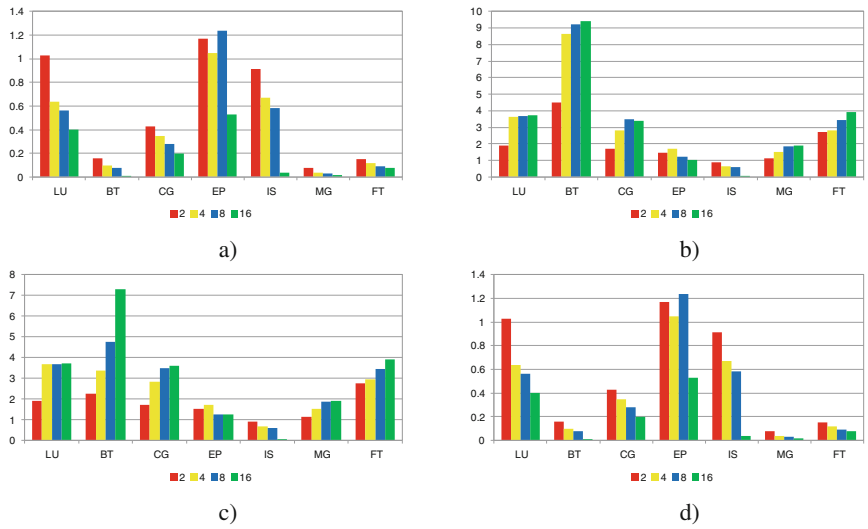

Fig. 1. (A) Speedup in baseline scheme. (B) Speedup when transaction size is optimized. (C) Speedup when write-set size is optimized. (D) Speedup when read-set size is optimized.

Speedup in a short transaction is limited since overhead of STM is high relative to the size of the transaction. A long transaction may increase abort rate as a large number of instructions in a transaction may increase the window during which transactions are identified as competitors. So, to boost performance of STM programs, we should merge

small transactions to reduce overhead of APIs. On the other side, in a large transaction, we should split the transaction into a number of small transactions to reduce abort rate and improve performance. Figure 1b shows that transaction size, indeed, has dramatic impact on performance. In BT, performance increases up to 9.3X when we change transaction size. On average, changing transaction size improves performance by 77.7 %, 88.4 %, 89.1 %, and 89.3 % when the number of threads is 2, 4, 8, and 16, respectively.

Figure 1c shows the impact of write-set on performance. STM uses a linked-list to record transactional write operations. When a transaction writes into a shared memory location, it inserts a new node to the linked-list. In commit, the transaction traverses the linked-list to acquire locks and update memory with new transactional data. If the transaction fails to acquire a lock, then it aborts and restarts. So, a transaction with a large write-set is more likely to abort. However, if we restrict transactions to have only small write-sets, then we need to split transactions into too many short transactions. This increases overhead of STM APIs relative to the performance gains of concurrent transactions and limits speedup. The optimum write-set size depends on pattern of memory accesses by transactions and varies from one benchmark to the other. We manually optimized performance of NAS benchmarks by changing write-set size. Figure 1c shows that by changing write-set size, performance of NAS benchmarks increases up to 7.3X.

Similar to write-set, a transaction uses a read-set to record memory locations that it reads. The read-set is implemented as a linked-list. In commit, the transaction traverses the read-set to verify that the read memory locations have not been written by other transactions. The optimum read-set size varies across the benchmarks. While a large read-set increases validation time, a small read-set increases the number of transactions and hurts performance. Figure 1d shows speedup in NAS benchmarks when read-set size is optimized. In Fig. 1d, we just changed read-only transactions in NAS benchmarks. Since there are a few number of read-only transactions in NAS benchmarks, the speedup is limited in Fig. 1d.

It is important to note that these three parameters are correlated. For example, when we split a transaction with 100 transactional writes into two transactions, each with 50 transactional writes, the size of the transaction is affected in addition to the write-set size.

4 The Prediction Model

4.1 Linear Regression

Size of transaction, write-set, and read-set are three factors that affect performance of STM applications. To optimize performance of STM programs, we build a linear regression model that predicts transaction size based on these three factors. The main reason that we decided to use transaction size as the predicted value by LR is that changing STM programs based on transaction size is straightforward. Quite often, it does not require any changes in the data structure of programs. For example, Fig. 2 shows a code snippet from BT benchmark. The loop iterations are independent and so

we can change transaction size by splitting the outer loop into a number of smaller loops and assigning each small loop to a transaction. On the other side, changing write-set and/or read-set of a transaction needs significant programming effort which complicates parallel programming. Hence, in all our experiments, we target transaction size for optimization. It is important to note that in some programs, it is not feasible to break down a large transaction because of dependency. For example, if the loop iterations in Fig. 2 are dependent, then we cannot break the outer loop.

While our optimization technique directly affects transaction size, it implicitly changes write-set and read-set sizes. For example in Fig. 2, if we split the k-loop into two equally sized loops, then write-set and read-set of each transaction is halved. So, by changing transaction size, we take into account the impact of all the three parameters on performance.

```
.....
TM_BEGIN();
for(k=1;…;k++)
    for(j=1;…;j++)
        rhs_t=(double)TM_SHARED_READ_F(rhs[k][j][i][0]);
        for(i=1;i<= grid_point[0]-2;i++){
            ui=rhs_t+dx1*tx1*(up1+us1-um1);
            .....
            rhs_t=ui*tx2-u[k][j][i][1];
            }
        TM_SHARED_WRITE_F(rhs[k][j][i][0],rhs_t);

TM_END();
....
```

Fig. 2. A code snippet from BT.

To train LR model, we use a set of benchmarks from NAS benchmark suite [9]. Table 1 shows the list of the benchmarks. The second column of the table shows the number of transaction per benchmark. We use 34 transactions for training of LR model. The rest are used for test. Also, we use DiscoPoP benchmark suite [10] to evaluate the impact of our optimization techniques on performance (Table 1).

We use SPSS [16] to find coefficients in our LR model. While our first LR model is simple and uses only three inputs, its accuracy is not acceptable. R_square which indicates how the data fit the regression model is only 45 %. According to this result, 55 % of data cannot be described by this model. We need to improve our LR model to reduce error rate of predicted transaction sizes.

To revise the LR model, we extend the inputs of the LR and include five more parameters: size of next transaction (SNT), number of assembly instructions between two consecutive transactions (NCT), write-set of the next transaction (WN), read-set of the next transaction (RN), and number of assembly instructions in a loop (TL). These five parameters are in addition to the original three parameters: size of transaction (ST), size of write-set (WS), and size of read-set (RS).

Table 1. NAS and DiscoPop benchmar suites.

Benchmark	Number of TXs
LU	6
BT	12
CG	4
EP	3
IS	6
MG	6
FT	9
Histo_serial	1
Ann_trainig	2
Mc_light	2
mandelbort	1

The first factor is called SNT. We explain why we use SNT as an input to LR model through an example. Assume that transaction A is followed by transaction B and transaction C is followed by transaction D. Transactions A, B, C, and D have 3000, 5000, 6000, and 11000 instructions, respectively, Assume that the optimum transaction size is 8000 instructions. We can combine transactions A and B and create a larger transaction with 8000 instructions. However, transactions C and D cannot be combined since the combined transaction has much more than 8000 instructions.

The second factor is called NCT. The number of instructions between two consecutive transactions affects the way we merge multiple small transactions into a large transaction. Assume that there are two transactions each with 3000 instructions. Similar to the previous example, assume that the optimum transaction size is 8000 instructions. If NCT is 2000 instructions, then the combined transaction results in optimum performance. However, if NCT is 10000, then we cannot combine the two transactions as the combined transaction is too large and hurts performance.

The third and fourth parameters are called WN and RN. Similar to SNT, write-set and read-set of the next transaction affect how we merge small transactions to build optimum transactions. So, to optimize transaction size, we need to consider WN and RN as well.

The fifth parameter is called TL. This parameter affects those transactions that are inside the body of a loop. If the total number of instructions in a loop is less than optimum transaction size, then we can move the whole loop in to a transaction. For transactions that are not inside a loop, we set this parameter to zero.

Equation 2 shows LR model using the 8 input parameters. We use SPSS [16] to calculate coefficients in Eq. 2. TS stands for transactions size.

$$TS = 5451 + 0.867 \times \mathrm{ST} + 0.015 \times \mathrm{RS} - 0.027 \times \mathrm{WS} - 0.832 \times \mathrm{TL} + 0.229 \times \mathrm{NCT} + 0.032 \times \mathrm{SNT} + 0.015 \times \mathrm{RN} - 0.026 \times \mathrm{WN} \quad (2)$$

Table 2 shows accuracy of predictions by LR. The test cases in Table 2 are transactions from NAS benchmarks. While accuracy is high in some of the benchmarks, i.e. test6, in most of the benchmarks, LR predictions result in significant error. The main reason for high error is that LR tries to draw a line to cover as many points as possible. If the points are scattered, then LR is unable to fit a line that covers all the points. This reduces accuracy of predictions.

Table 2. Predcition accuracy in LR

Test Cases	Original TX Size	Predicted TX Size	Optimum TX Size	Error (%)
test1	148258	10576.54	6739	-56.9%
test2	54736	6985	2488	-180.7%
test3	112816	9159	5128	-78.6%
test4	636460	27062	28930	6.5%
test5	204192	5343	6381	16.3%
test6	35122	34385	35122	2.1%

We need to revise the LR model to increase accuracy of predictions. Further investigation of LR model reveals that the error rate for transactions with large positive error is in the range of 6.5 %-16.3 %. On the other side, error rate of transactions with large negative error is in the range of 56 %-180.7 %. This motivates us to classify transactions into three categories: transactions with large negative error (class1), transactions with large positive error (class2), and transactions with small error (clas3). We use separate LR model for each class. This improves accuracy of predictions since the set of points within a class are well-organized and fitting a curve to the points results in less residual error. We use the same 8 input parameters for the three LR models: SNT, NCT, WN, RN, TL, ST, WS and RS. Equations 3-5 show the new LR models. TS1, TS2, and TS3 correspond to class1, class2, and class3, respectively.

$$TS_1 = 416 + 0.013 \times RS - 0.02 \times \mathrm{WS} - 0.043 \times \mathrm{TL} + 97.09 \times \mathrm{NCT} + 0.041\mathrm{SNT} + 0.012 \times \mathrm{RN} - 0.019 \times \mathrm{WN} \tag{3}$$

$$TS_2 = 7196 + 0.824 \times ST + 0.018 \times RS - 0.033 \times WS - 0.791 \times TL - 0.015 \times NCT + 0.03 \times SNT + 0.018 \times RN - 0.031 \times WN \tag{4}$$

$$TS_2 = 8142 + 0.799 \times ST + 0.024 \times RS - 0.039 \times WS - 0.765 \times TL - 0.026 \times NCT + 0.03 \times SNT + 0.023 \times RN - 0.035 \times WN \tag{5}$$

4.2 Decision Tree

To exploit the three LR models, we need to classify transactions into three categories: class1, class2, and class3. We use decision tree [17] for classification. C4.5 [15] is a popular decision tree algorithm and is able to classify objects with continuous attributes. We train the decision tree with already classified sample transactions. Each sample S_i consists of an 8-dimensional input vector (SNT, NCT, WN, RN, TL, ST, WS, and RS) as well as the class which the S_i belongs to. Through the training phase, the decision tree learns how to classify transactions. For test, we feed the decision tree an 8- dimensional vector and the decision tree predicts the class of the transaction corresponding to the vector.

4.3 Mixed Decision Tree and Linear Regressions Model

Our last optimization technique is a mixture of decision tree and linear regression. First, decision tree determine the class of a transaction. Then, we use one of the three LR models (Eqs. 3-5) to predict optimal transaction size.

We used the same method to test the mixed model: 34 transactions from NAS are used for training and the rest are used for test.

Table 3 shows error of predictions made by our mixed model. On average, error rate drops from 59 % to 2.8 %. The error rates in most of the test cases are very low. The largest error rate is 16 % in test5. This transaction has small read- and write-sets and the decision tree categorizes it in class 3. However, this transaction has a large positive error and should be categorized in class 2. We used a small set of transactions for training. However, if we include more transactions in training phase, then this abnormality may disappear.

Table 3. Prediction accuracy using mixed model

Name	Original TX Size	Predicted TX Size	Optimized TX Size	Error rate
test1	148258	6739	6739	0%
test2	54736	2488.45	2488	−0.02%
test3	112816	5129.87	5128	−0.04%
test4	636460	28930	28930	0%
test5	204192	5359.92	6381	16%
test6	35122	34859.46	35122	0.75%

To evaluate the impact of mixed model on execution time, we use benchmarks from DiscoPoP benchmark suite [10]. The last column in Table 4 shows speedup in optimized benchmarks. LR and decision tree together improve performance up to 54.8 %. On average, the performance is improved by 30.3 %.

Table 4. Speedup in DiscoPop [10] using mixed model

Name	Original TX Size	Predicted TX Size	Speedup
Histo_serial	320625	14186	47.0%
Mc_light	2125000	77310	28.4%
Mc_light	116250	11148	54.8%
Ann_trainig	72000	10614.15	18.6%
Ann_trainig	480000000	16328112	18.5%
mandelbort	78208	9776	14.9%

5 Related Work

Transactional memory was originally proposed by Herlihy and Moss [1]. Shavit and Touitou [7] were the first to introduce software implementation of transactional memory. Since then, many researchers offered new implementations for transactional memory or improved already existing implementations through optimizing different aspects of transactional memory.

Felber et al. [4] introduced TinySTM which is a time-based STM. The authors evaluated the impact of three parameters on performance: the number of locks, the hash function for lock table, and the size of hierarchical array in lock. The authors found that there is no one-size-fit-all value that works well across all applications. Even within an application, the optimum value of a parameter may change during the run-time. The authors proposed using hill-climbing strategy to adjust STM parameters. The dynamic optimization technique introduced in TinySTM can be combined with our static approach to improve performance of STMs further.

Wang et al. [14] proposed new techniques to optimize transactional memory in unmanaged programming languages such as C. Supporting transactions in an unmanaged language is much more challenging than managed code. For example, the lack of type safety in C forces programmers to implement validation in granularity of cache line rather than object. This makes optimization of STM overhead a challenging task. Wang et al. [14] proposed new constructs in C which allows a programmer to declare blocks that can be executed atomically. Furthermore, they exploited some compiler based optimization techniques such as inlining for fast paths, elimination of redundant barriers, and register check-pointing optimization to reduce overhead of STM. We use a different approach and focus on transaction, write-set, and read-set sizes to optimize STM applications.

DiscoPoP [10] is a tool that automatically finds parallelizable regions of a sequential code. DiscoPoP is able to identify parallelism between code regions with arbitrary granularity and does not require any predefined notion of language constructs. DiscoPoP identifies sections of the code in which data dependency does not exist. These sections are called Computational Units (CUs). Then, the tool builds a dependency graph using CUs. Nodes of the graph represent CUs and edges represent dependency between CUs. From the dependency graph, DiscoPoP determines potential parallelism available on varying levels of the code. The output of the DiscoPoP is a file that indicates which lines of the sequential code can be grouped as a task and run

concurrently with other tasks. We used the set of benchmarks introduced in DiscoPoP for evaluation of our mixed model. Our mixed model can be combined with DiscoPoP to convert sequential codes into highly optimized STM codes automatically.

Castro et al. [12] used machine learning for thread mapping in STMs. In thread mapping, executing threads are assigned to processing cores dynamically so that the latency associated to the memory hierarchy is reduced. To decide on thread mapping, status of transactions and also STM platform are monitored at specific intervals. At the end of each interval, thread mapping is adjusted based on a decision tree learning method (ID3) [15]. Our work is different as we use C4.5 for classification of transactions. C4.5 is an enhanced version of ID3 which supports continuous attributes. Also, we focus on transaction, write-set, and read-set sizes for optimization. On the other side, Castro et al. [12] focus on thread mapping.

Didona et al. [11] proposed self-tuning methodologies to dynamically adjust concurrency level in STMs. One of the key factors in STM programs is concurrency level. Too many threads in a program increase contentions over shred memory locations and hurt performance. On the other side, if concurrency level is too low, then exploited parallelism by STM programs will be limited. The optimum number of executing threads depends on many parameters including but not limited to pattern of addresses generated by transactions, OS scheduler, structure of memory hierarchy, etc. So, identifying the right level of concurrency in STMs is not a trivial task. Diego et al. [11] used a hill-climbing algorithm to explore concurrency level space in shared memory STMs. This optimization technique is a dynamic approach and can be combined with our static code optimization technique to improve performance of STM applications further.

6 Conclusion

In this paper, we presented an optimization technique that helps programmers to write efficient STM programs. We studied the impact of three parameters on STM performance and showed that STM applications are highly sensitive to the three parameters. Then, we exploited LR to predict transaction size based on the three parameters. Our first LR model was not accurate enough and it resulted in high error rate. We revised the LR model by extending its inputs from 3 to 8 parameters. Also, to improve accuracy of LR, we classified transactions into three groups: transactions with large positive errors, transactions with large negative errors, and transaction with low errors. We used decision tree to classify transactions automatically. Our mixed model reduces error rate from 59 % to 2.8 % on average. We also evaluated the mixed model using DiscoPoP benchmark suite. The mixed model improves performance of DiscoPoP up to 54.8 %.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA), pp. 289–300, May 1993

2. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Abadi, M., Harris, T., Mehrara, M.: Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, February 2009
4. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, February 2008
5. Marathe, V.J., Scherer III, W.N., Scott, M.L.: Adaptive software transactional memory. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 354–368. Springer, Heidelberg (2005)
6. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, March 2009
7. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual Symposium on Principles of Distributed Computing (PODC), pp. 204–213, August 1995
8. Goldberger, S.: Best linear unbiased prediction in the generalized linear regression model. J. Am. Stat. Assoc. **57**(298), 369–375 (1962)
9. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel Benchmarks. RNR Technical Report RNR-94-007, March 1994
10. Li, Z., Jannesari, A., Wolf, F.: Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI), Lyon, France, pp. 1004–1013, October 2013
11. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Identifying the optimal level of parallelism in transactional memory applications. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 233–247. Springer, Heidelberg (2013)
12. Castro, M., Góes, L.F.W., Fernandes, L.G., Méhaut, J.-F.: Dynamic thread mapping based on machine learning for transactional memory applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 465–476. Springer, Heidelberg (2012)
13. Google Inc. Quantifying Movie Magic with GoogleSearch. June 2013
14. Wang, C., Chen, W.Y., Wu, Y., et al.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: The Proceedings of the International Symposium on Code Generation and Optimization, pp. 34–48. IEEE Computer Society (2007)
15. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Francisco (1993)
16. SPSS Inc. 2007. SPSS 16.0 Command Syntax Reference. Chicago, IL: SPSS Inc
17. Quinlan, J.R.: Induction of decision tree. Mach. Learn. **1**(1), 81–106 (1986)
18. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Yelick, K. A.: The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)