Predicting the Scalability of an STM

A Pragmatic Approach

Aleksandar Dragojević
EPFL, Switzerland
aleksandar.dragojevic@epfl.ch

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Abstract

Conducting a thorough performance evaluation of an STM is very time consuming. Depressingly, even with all this effort, and even with the same application, it can still be hard to predict the performance if the number of underlying threads on which the application needs to be deployed is different than those of the experiment. Basically, one might have to conduct an entire set of new experiments to get some understanding of the performance of the STM with the new number of threads.

We propose a pragmatic approach to contribute to changing this state of affairs. Using classical engineering approximation techniques, we extract from a set of STM performance measurements, analytical performance functions to model the scalability of the STM. We show, more specifically, that polynomial and rational functions provide good interpolations of STM performance: even with only a handful of measurements, the average error in most cases is around 1-2%. Further, we show that we can perform reasonably precise extrapolation using rational functions: basically, using measurements with up to m threads, we can predict the performance up to roughly $2m$ threads with a relatively low error (around 10% in best cases).

We discuss two possible applications of our approach: (1) statically deciding whether to use an STM for a given workload and a given number of threads, and (2) dynamically adjusting the number of threads that execute in parallel to match the optimal concurrency level of a given workload.

Keywords Software Transactional Memory, Performance, Scalability.

1. Overview

As its name indicates, Software Transactional Memory (STM) is built purely in software: part of its appeal is its independence from any specific hardware support. In principle, an STM can work on any hardware and for any size of transactions and data structures [3, 10, 12, 17, 21, 24, 26]. Yet, and not surprisingly, the performance, and actually the relevance, of the STM are highly sensitive to the target application, the workload, the underlying architecture, and the actual number of threads used for parallelizing the code.

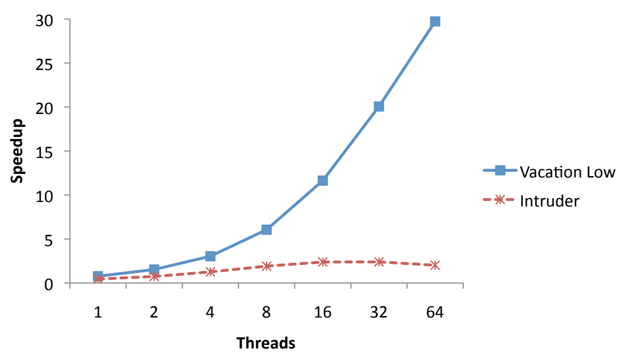
Figure 1(a) depicts the speedup of a parallel code that uses SwissTM [12] over sequential, non-instrumented code, for two different workloads from the STAMP benchmark suite [6]. The figure conveys the very fact that, while STM scales very well on the `vacation low` workload, outperforming sequential code by almost 30 times with 64 threads, its scalability is not nearly as good on `intruder`, where it outperforms sequential code by only 2 times with 64 threads. In fact, even the same application that uses STM can have highly varying performance depending on the workload configuration. Figure 1(b) illustrates this. The lower contention read-dominated STMBench7 [16] workload achieves a speedup factor of almost 12 with 64 threads, but the higher contention write-dominated workload of the same benchmark is faster than non-instrumented sequential code by less than 2 times.

In fact, two recent studies [8, 11] drew contradictory conclusions about the scalability of an STM even on the same subset of benchmarks. Not only the performance of different STM workloads can differ significantly, but it is also very difficult to predict how it will evolve should the number of available threads (cores) be increased. In general, experience shows that even if the scalability of an STM looks great for a range of thread values, performance might actually (slightly or significantly) drop after some point: basically, contention can simply become too high with too many threads. But knowing at which point this happens is hard without intensive experiments. For the workloads in Figure 1, the performance peaks at 22 threads for `intruder`, 32 for `STMBench7 write`, 52 for `STMBench7 read`, while for `vacation low` it keeps improving up to 64 threads.

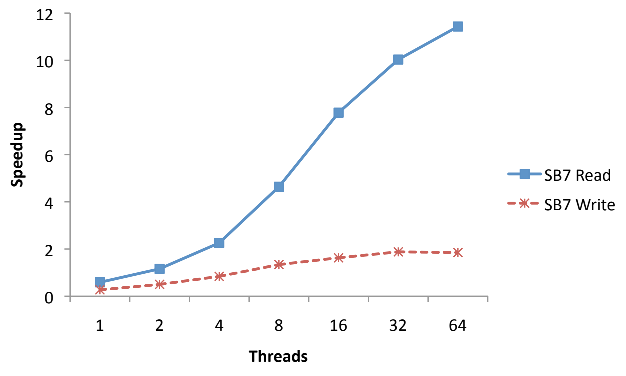
Clearly, this state of affairs might put potential adopters of STM in a difficult position: should they write their application with an STM in mind and expect the performance to speed up with a new, to be purchased, architecture with more cores? Or should they simply keep hacking their old lock-based techniques and forget about the STM?

Certain rules of thumb do exist. For example, if different application threads mostly access disjoint data, and if the majority of accesses are reads, resulting in lower contention, the application is probably a good candidate for parallelization with STM and scaling it to a larger number of threads would certainly reveal beneficial. Also, if only a small portion of the code is actually using transactions, the overheads of STM will not be significant and the application is likely to benefit from STM. However, these rules are somehow vague and not easily applicable in all cases.

Ideally, an automated tool would analyze the atomic blocks of the application and, based on their characteristics, would predict the scalability of the STM on that application. The analysis could partially be static, based on the source code, but would, most likely, also require analysis based on the profiling executions. Generating the code for the profiling executions is indeed feasible: (1) the programmer needs to identify atomic code blocks even if some other parallelization technique is used and (2) STM compilers [3, 13, 17, 28] automatically produce STM code eliminating the need for manual instrumentation. Nevertheless, atomic blocks have a wide variety of characteristics (e.g. duration, number and type of transactional accesses, etc.) and it is not completely clear which of these characteristics are relevant for predicting the performance. Furthermore, some of the characteristics are difficult to capture in a meaningful and concise way (e.g. transaction access sets). All of this makes the design of a tool for predicting scalability based on atomic block characteristics a daunting task.



(a) STAMP



(b) STMBench7

Figure 1. STM performance

This is where theory should, in principle, help. While inherently general, and not tied to any application, workload, or thread count, theoretical work on STM performance [5, 14, 15] however mainly deals with the *worst-case*. Worst cases are useful for defining lower bounds, but they are not very likely to occur in practice where the focus is the *common-case*. For example, it was shown in [14] that, in the worst case, the operation in a certain class of STMs that use invisible reads takes time linear in the number of objects in the system. Roughly speaking, this is because transactions in such STMs need to validate their read sets on every operation, in order to maintain consistency of the memory snapshot they observed. This lower bound also applies to STMs that use global counters in order to avoid some of the read-set validations. However, in the common-case, the use of a global counter will eliminate most read set validations anyway, significantly improving common-case performance over the worst-case.

Several approaches for modeling STM performance have been proposed [18–20, 25, 27]. None of these is completely general as they are used to either model one aspect of STM performance, such as algorithmic efficiency in [18–20] and transaction conflict behavior in [25], or are developed to answer specific questions, such as whether to use locking or STM for particular critical section in [27].

In this paper, we explore a much simpler and pragmatic approach to predicting the scalability of STM-based applications. We employ classical engineering approximation techniques [4] to predict the performance of a workload W with n threads, based on its performance with m_i $(1 \leq i \leq M)$ threads. Put differently, we construct an analytical *performance function* $performance = f(n)$, based on M performance measurements. Function f describes the characteristics of the application workload and the whole computing environment (STM, OS, hardware etc.). It takes as a parameter the number of threads n, and outputs a guess $f(n)$ of the expected performance. Interestingly, the approach does not require any knowledge of the workload or the system configuration when constructing f: no access to the source code of the application is required.

Basically, our approach to predicting performance consists of three steps:

1. **Profiling.** We measure the performance of the workload on a target STM and computer system with several thread counts.
2. **Approximation.** Using classical approximation techniques [4], and based on the performance measures collected during the profiling step, we construct a performance function *performance* = $f(n)$ supposed to capture all the characteristics of the workload, STM and the computer system.
3. **Prediction.** Finally, we use f to predict the performance with a different number of concurrent threads than we used during the profiling step.

We study several choices for the performance function f using ZunZun.com [2], MATLAB built-in polynomial approximations and rational approximation method from [7]. We evaluate the precision of the approximation using the STAMP benchmark suite [6] and a machine with UltraSPARC T2 CPU that supports 64 hardware threads. Our experiments reveal that both polynomial and rational functions can be used to provide good interpolations of STM performance. Using only 9 measurements (uniformly distributed across thread counts) we obtain the average error of only around 1-2% in most cases. Also, we show that reasonable extrapolations of STM performance can be performed using rational functions. Using measurements with up to m threads, in most cases we can predict the performance up to roughly $2m$ threads with a relatively low error (in best cases around 10%).

We illustrate two practical applications of our approach:

- Statically predicting the performance gain when adding additional threads (e.g. deciding whether to buy more CPUs in a cloud computing environment). Think of a system administrator who has to decide whether to assign (or buy) additional CPU cores to an already executing STM application. A performance function f can be constructed and used to speculate about the performance, should the additional CPU cores be assigned to the application.
- Dynamically adjusting the number of threads to achieve the optimal level of parallelism for a given workload. Remember that, on certain workloads, the performance degrades when adding threads after a certain threshold. We basically suggest a way to dynamically use the performance function in order to determine the optimal number of parallel threads.

In the rest of the paper, we first evaluate the suitability of different functions to interpolate STM performance (Section 2). Next, we evaluate the suitability of functions for extrapolating the performance when increasing the number of threads (Section 3). We then describe and evaluate an algorithm for dynamically adjusting the number of threads (Section 4). Finally, we conclude the paper by discussing the limitations of our approach (Section 5).

2. Interpolation

We discuss and evaluate below various possible function choices for interpolating STM performance.

The choice of the approximation function type is very important and can impact the error of the approximation significantly. The ideal function is one that is relatively simple, yet general, and can be applied to different systems and workloads. We considered several standard choices:

- Polynomial functions $f(n) = a_0 + a_1 n + ... + a_m n^m$.
- Rational functions $f(n) = \frac{a_0 + a_1 n + ... + a_p n^p}{b_0 + b_1 n + ... b_q n^q}$.
- Logarithmic functions e.g. $f(n) = ln(n)$, $f(n) = a + b \cdot ln(n) + c \cdot ln(n)^2 + d \cdot ln(n)^3$
- Exponential functions e.g. $f(n) = e^n$, $f(n) = ae^{\frac{b+cn}{d+en}} + f$
- Power functions e.g. $f(n) = n^a$, $f(n) = a(n-b)^c + d$

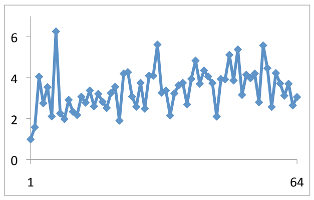
Polynomials can be used to approximate any continuous function on a closed interval to any degree of accuracy [23]. Rational functions have the same nice property, but they can model more diverse behaviors than polynomials. The major drawback of polynomials and rational functions is their not conveying much information about the workload behavior. On the other hand, logarithmic, exponential and power functions convey more information about the workload performance, and they could be used to approximate some of the workloads very well. Unfortunately, as the experiments below reveal, they could not be used for all the workloads equally well. Our experiments led us to use polynomial and rational functions as they, maybe unsurprisingly, proved to be the most general.

To perform the actual approximations, we used ZunZun.com [2], a web site for curve fitting with thousands of different functions. For polynomial approximations, we used polynomial functions constructed using MATLAB's `polyfit` function, which applies the method of least squares to the Vandermonde matrix whose elements are powers of n [1]. To construct rational approximations functions, we used the method of [7], which is easy to use in MATLAB, more flexible than ZunZun.com, and indeed produces good results.

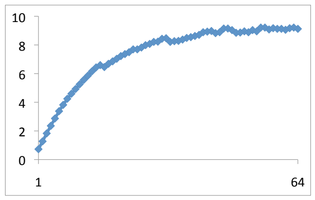
All our experiments were conducted on a Sun Microsystems UltraSPARC T2 CPU with 64 hardware threads. We used 10 workloads defined in STAMP [6] 0.9.10 distribution.[1] We decided to focus on STAMP because, while it might not be fully representative of all STM workloads, its performance functions are diverse and, in our experience, represent other workloads' (e.g. STMBench7 [16] and microbenchmarks) performance functions well. All the experiments were repeated several times in order to reduce variance in collected data.

We base our evaluation in this and the following sections on the performance measurements presented in Figure 2. The figure depicts the speedup of STM code over non-instrumented sequential code for varying thread counts. For all workloads we used, except for `bayes`, slightly changing the number of executing threads does not cause a significant change in the performance. `Bayes` implements a search algorithm that, for the same input, can execute a different number of transactions in different executions, which makes the execution time vary significantly. For this reason it is particularly hard to predict the performance of `bayes` well (as our evaluation confirms).

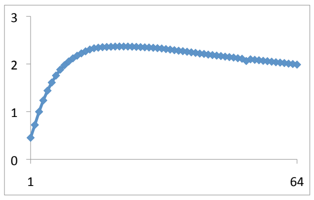
We first approximated the observed performance using all five function types, listed in Section 2, and using all measured data points (performance for all 64 thread counts) in order to derive the best precision of the approximation we can hope to obtain. The median and the first and third quartiles of relative error for each of the STAMP benchmarks are depicted in Figure 3.



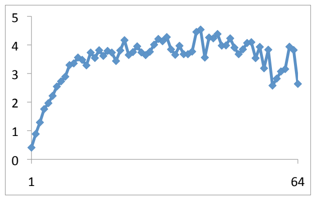
(a) Bayes



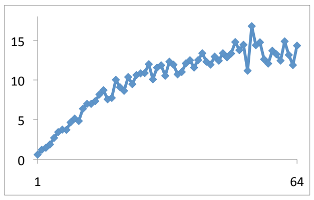
(b) Genome



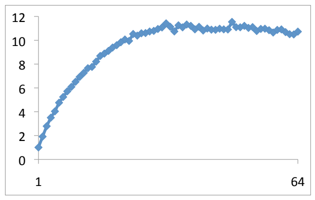
(c) Intruder



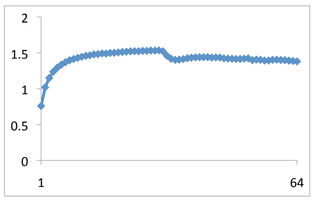
(d) Kmeans High



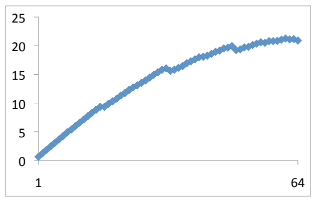
(e) Kmeans Low



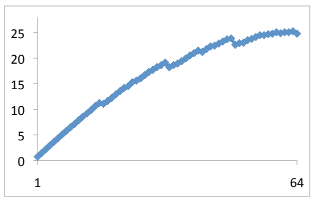
(f) Labyrinth



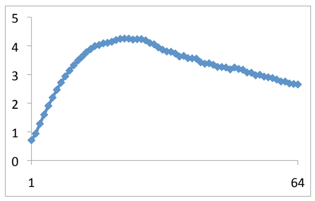
(g) SSCA2



(h) Vacation High



(i) Vacation Low



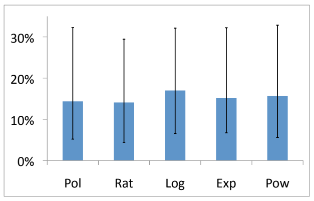
(j) Yada

Figure 2. Speedup over sequential code

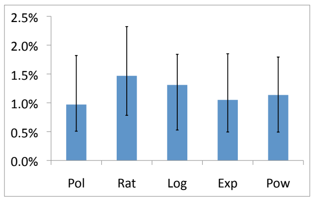
In the figure, we used the best fitting logarithmic, exponential and power functions as calculated by ZunZun.com. Different functions produced the best approximation on different benchmarks for the same function type. For example, the best power function approximation on `genome` is obtained by $f(n) = a(x-b)^c + d$, while the best power function approximation on `kmeans-low` is $f(n) = ab^{nc+d} + e$. This fact makes automatic approximation using logarithmic, exponential or power functions, difficult.

On the other hand, both polynomial and rational approximations use the same type of functions. We used polynomials of degree 6 and for rational approximations both numerator and denominator are of degree 3. Furthermore, the approximation error in most cases is the smallest with polynomial and rational functions. It is even more important that the small error is produced consistently, unlike with logarithmic, exponential and power functions. For ex-

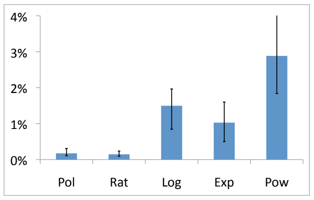
[1] We modified the STAMP benchmarks to support an arbitrary number of concurrent threads instead of only supporting thread counts that are a power of two, as in default STAMP distribution. The modified files are available from `http://lpd.epfl.ch/site/research/tmeval`.



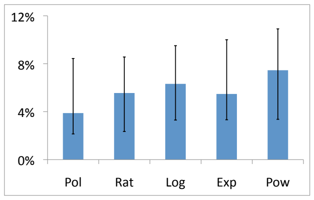
(a) Bayes



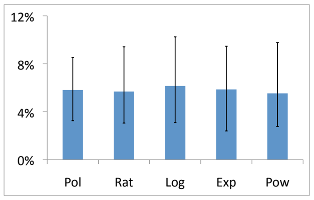
(b) Genome



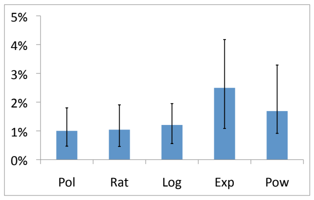
(c) Intruder



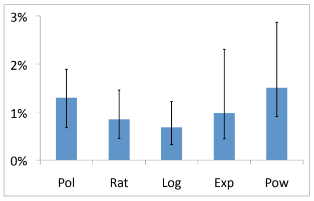
(d) Kmeans High



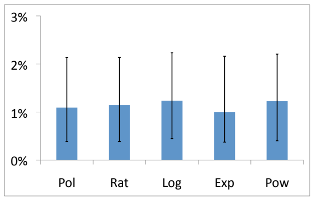
(e) Kmeans Low



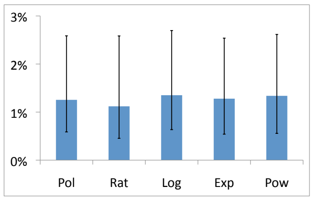
(f) Labyrinth



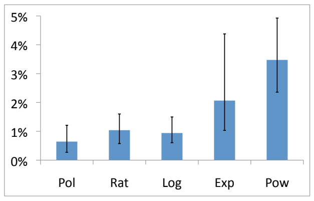
(g) SSCA2



(h) Vacation High

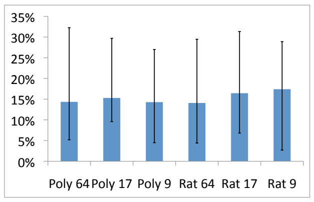


(i) Vacation Low

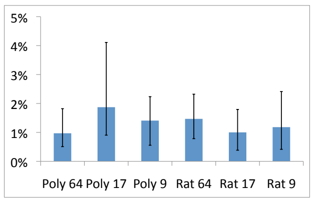


(j) Yada

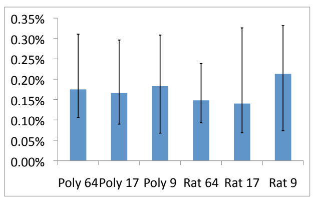
Figure 3. Approximation error with different function types



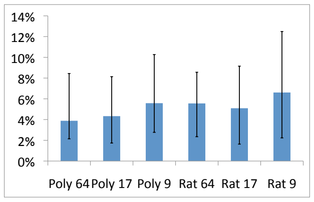
(a) Bayes



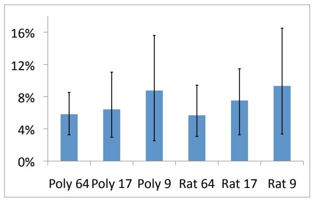
(b) Genome



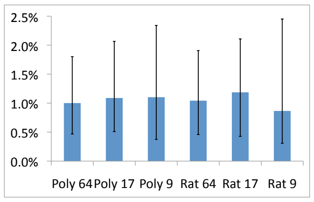
(c) Intruder



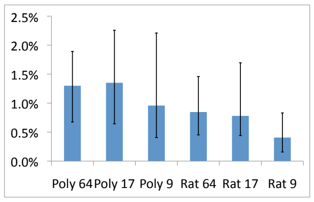
(d) Kmeans High



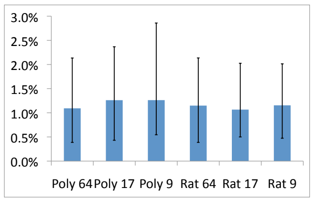
(e) Kmeans Low



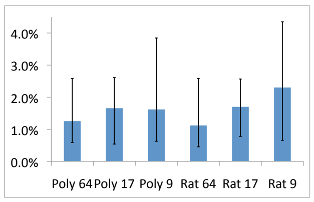
(f) Labyrinth



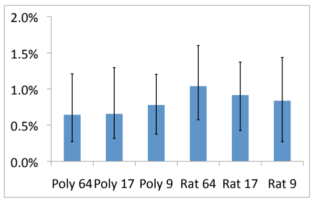
(g) SSCA2



(h) Vacation High



(i) Vacation Low



(j) Yada

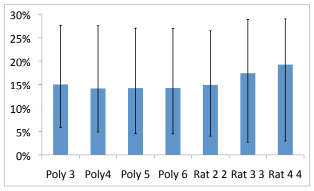
Figure 4. Interpolation error with different numbers of data points

ample, logarithmic approximation has the smallest error on `ssca2` workload, but it has the largest error on `bayes`, `kmeans low` and `vacation low`. It is also much worse than polynomial and rational approximations on `intruder` and `yada`.

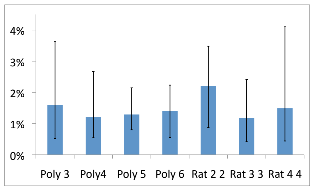
Clearly, polynomial and rational functions revealed to be the best suited for approximating STM performance, both because of the low approximation error, and uniform and simple approximation functions.

Next, we evaluated the approximation error with polynomial and rational functions with the subset of measurements to evaluate the quality of approximations with incomplete data. Here, we used 17 and 9 data points uniformly distributed across thread counts. The median and the first and the third quartile of relative error for each of the STAMP benchmarks are depicted in Figure 4. The figure shows that the error of approximation does not increase significantly even if only a relatively small subset of data points is used, as long as the data points cover the whole approximation range. The figure also shows that both rational and polynomial functions achieve relatively similar errors.

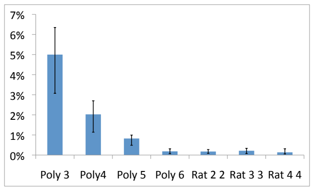
Finally, we present the approximation error with different types of polynomial and rational functions using 9 data points uniformly distributed across threads counts. Figure 5 depicts median and the first and the third quartile of relative error for each of the STAMP benchmarks with approximations using polynomial functions of degree 3, 4, 5 and 6 and rational functions with both numerator and denominator of degree 2, 3 and 4. The figure shows that the best polynomial approximations are achieved with the polynomials of degree 6. For this reason we used polynomial approximation of degree 6 in further experiments. The situation is not as clear with rational functions as approximations of degree 3 and 4 have similar errors. We decided to use rational approximations of degree 3 because they require less data points to construct.



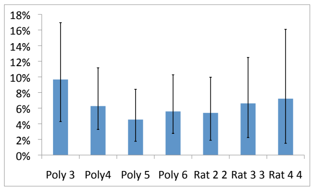
(a) Bayes



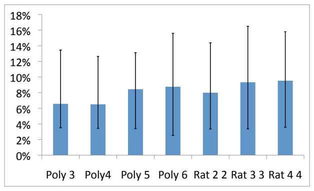
(b) Genome



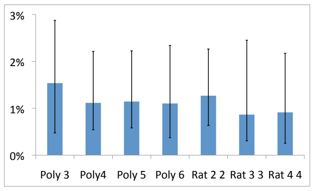
(c) Intruder



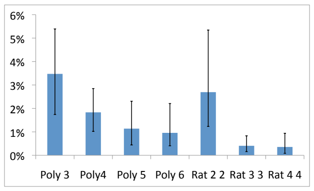
(d) Kmeans High



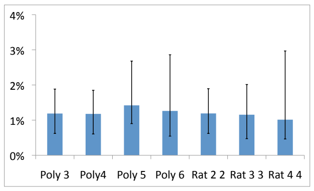
(e) Kmeans Low



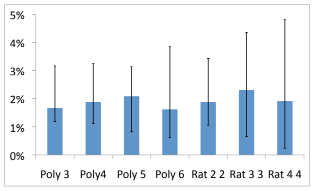
(f) Labyrinth



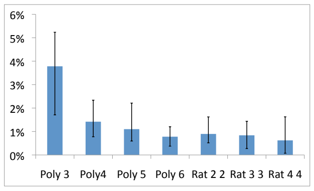
(g) SSCA2



(h) Vacation High



(i) Vacation Low



(j) Yada

Figure 5. Interpolation error with different types of polynomial and rational functions

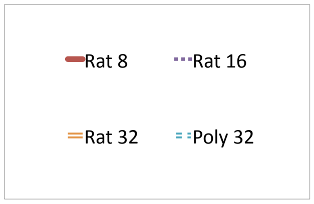
3. Extrapolation

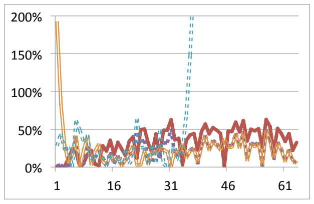
We describe and evaluate here the use of performance functions to extrapolate performance, i.e. predict performance for more than m threads, based on several measurement all with less than m threads. This is, arguably, the most interesting usage of performance functions.

We constructed rational and polynomial functions based on the measurements with 1 to 8, 1 to 16 and 1 to 32 threads. Figure 6 conveys the error of extrapolations for these cases.

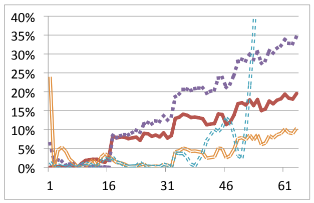
The figure shows that the polynomial functions are not very useful for extrapolation (which might not be very surprising) as the error increases rapidly after 32 threads.[2]

The figure also shows that the extrapolation of STM performance data can be performed using rational functions for many workloads. The only exception is `bayes` which has highly varying performance making performance prediction very hard. Extrapolation with rational functions achieves good results for `genome`, `intruder`, `labyrinth`, `ssca2`, `vacation-high` and `vacation-low` where at least one of the approximation functions has error less than 20% with 64 threads. Approximation using data for first m threads extrapolates data quite well for up to $2m$ threads in most workloads for *Rat 8* and *Rat 32*. For example, the

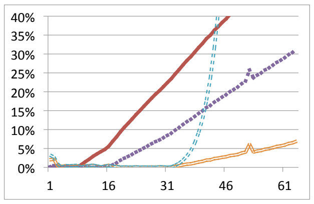




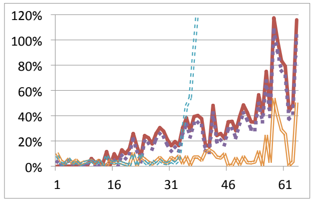
(a) Bayes



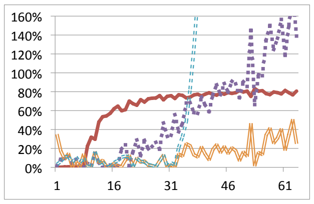
(b) Genome



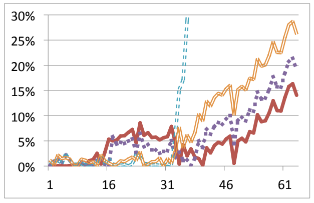
(c) Intruder



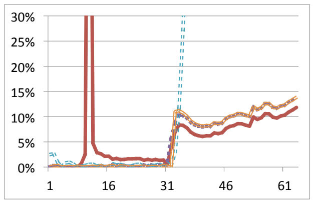
(d) Kmeans High



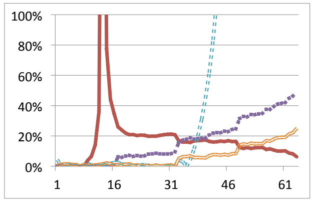
(e) Kmeans Low



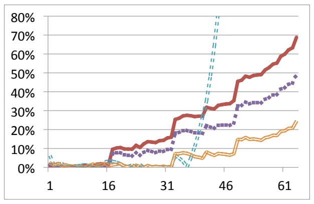
(f) Labyrinth



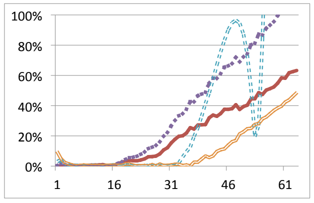
(g) SSCA2



(h) Vacation High



(i) Vacation Low



(j) Yada

Figure 6. Extrapolation error

[2] We observed the same trends for polynomial extrapolation based on 8 and 16 data points, but omit the data to avoid further cluttering the graphs.

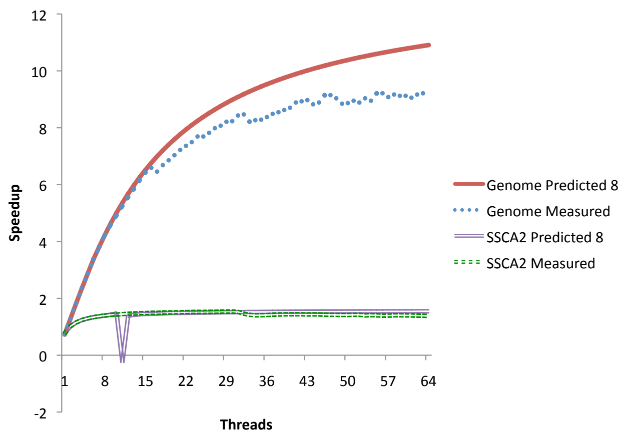


Figure 7. Measured and predicted performance (using *Rat 8*)

error is about 5% in `intruder` and about 10% in `genome`. Both `kmeans-high` and `kmeans-low` are quite difficult to predict due to high variance in measured performance, but the prediction error is still relatively low (less than 20% up to 23 threads with *Rat 8* and up to 50 threads with *Rat 32*).

It is interesting to note that the prediction error significantly changes at multiples of 16 threads for several benchmarks (e.g. `genome`, `ssca2`, `vacation-high` and `vacation-low`). This is because the CPU we used has 16 execution units, making the multiples of 16 a threshold for, effectively, changing characteristics of the hardware in some workloads. In several cases, *Rat 8* performs better than *Rat 16*. We believe that the main reason for this is the same. With *Rat 8* the performance function is not as much tailored to the characteristics of the machine in the first operating region as with *Rat 16*. With `labyrinth` and `ssca2` *Rat 8* is better than even *Rat 32*. The main reason for this is that performance curve for these two workloads significantly changes at 32 threads.

To better present the usability of performance extrapolation we show the measured and *Rat 8* predicted speedups of STM code over sequential, non-safe code for `genome` and `ssca2` benchmarks in Figure 7. The extrapolations in both cases are rather good (except for the dip at 11 threads for `ssca2`) and they could clearly be used to correctly predict that `genome` scales well while `ssca2` does not.

One important aspect of performance extrapolation is the ability to correctly predict the number of threads for which the performance is the highest. Table 1 depicts the thread counts with the highest performance (T_{max}) for measured and predicted performance and the performance impact that would result from using the predicted T_{max} ($s = \frac{Perf(T_{measured\_max}) - Perf(T_{predicted\_max})}{Perf(T_{measured\_max})} \cdot 100$). The table shows that increasing the number of points used for the extrapolation significantly improves accuracy of T_{max} prediction. Also, *Rat 32* predicts T_{max} quite accurately, with the average slowdown of 6.4%.

4. Optimal Concurrency

In this section, we describe and evaluate a simple scheme for using performance functions to dynamically adjust thread counts in order to achieve the best performance for a given workload.

A simple scheme for dynamically determining the optimal number of concurrently executing threads is sketched in Algorithm 1. The basic idea is to measure the performance of the workload with different number of threads during short time intervals (line 2) and use these measurements to construct the performance function

Algorithm 1: Matching optimal thread count

```
1  upon $period\_expired(job)$ do
2      $job.perf[job.thread\_count] \leftarrow job.perf$;
3      if $job.points < min\_count$ then
4          $job.thread\_count \leftarrow choose\_tc()$;
5          if $job.perf[job.thread\_count] = 0$ then
6              $job.points \leftarrow job.points + 1$
7      else
8          $perf\_func \leftarrow approximate(job.perf)$;
9          $tc \leftarrow max\_perf\_thread\_count(perf\_func)$;
10         $job.thread\_count \leftarrow tc$;
11 function $approximate(perf)$ do
12     if $size(perf) < 5$ then
13         return $approx\_poly(perf, size(perf) - 1)$;
14     else if $size(perf) \leq 6$ then
15         return $approx\_rat(perf, 2)$;
16     else
17         return $approx\_rat(perf, 3)$;
```

(line 8). During each interval, the workload is executed using the number of threads for which the predicted performance is the highest (line 10). If there are not enough measurements for approximation (line 3), the number of threads is chosen arbitrarily (line 4). The choice can be made randomly or according to some heuristic. Function `approximate` (line 11) takes at least 3 data points and uses a different approximation based on the number of measurements—polynomial with maximum degree if there are not enough measurements for rational and rational with maximum degree otherwise.

It is important to note that automatically scheduling the optimal number of threads is not straightforward, and would require some additional information from STM or other parts of the runtime. For example, thread T_1 could be running and waiting for thread T_2 to perform some task before continuing. If the thread scheduler temporarily suspends T_2 at this point it would actually degrade and not benefit the overall performance of the application.

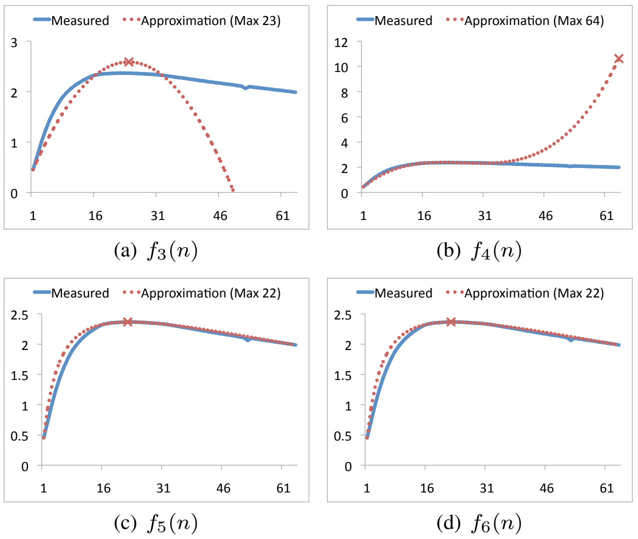
We implemented Algorithm 1 in MATLAB to evaluate its convergence speed and precision. Table 2 contains the convergence speed and error. The algorithm ends when the thread count to be used in the next time interval has already been used for approximation in the current time interval. The table contains results for different choices of starting measurements, which are given in column titles (e.g. 1-32-64 means that the starting measurements are performed with 1, 32 and 64 threads). We fix thread counts for starting measurements, in all but the first column, where one of the thread counts is chosen randomly. For this case, the table depicts the averages of five runs. The results are encouraging, as the algorithm converges in 6 to 8 steps on average no matter what the choice of starting points is. Its relative error is also quite low (6% to 8.5% on average). Table 2 also conveys the averages without including results for `bayes`, which suffers from a very high variance in collected data and is extremely difficult to predict. Without it, the average convergence speed stays about the same, but the average error is almost half of the average error with `bayes` included. Choosing 4 starting measurements instead of 3 (the last column), does not actually improve the speed or the precision of the algorithm. Overall, we obtain the lowest error when using 1, 16 and 32 threads for the starting measurements. It is interesting to note that the random choice of starting points also performs well, having the fastest con-

	Measured	Rat 8		Rat 16		Rat 32		Poly 32	
	T_{max}	T_{max}	s [%]	T_{max}	s [%]	T_{max}	s [%]	T_{max}	s [%]
Bayes	7	7	0	7	0	7	0	31	10
Genome	55	64	9.6	64	9.6	64	9.6	64	9.6
Intruder	22	64	16.1	64	16.1	22	0	64	16.1
Kmeans High	41	64	41.8	64	41.8	54	22	64	41.8
Kmeans Low	53	7	77.6	64	14.6	64	14.6	30	31.1
Labyrinth	48	64	7	64	7	64	7	64	7
SSCA2	30	64	10	64	10	33	7.6	29	0.2
Vacation High	61	13	63.7	64	1.7	64	1.7	34	25.5
Vacation Low	63	64	1.9	64	1.9	64	1.9	35	24.9
Yada	24	33	10.6	64	37.6	23	0	64	37.6
Avg			23.8		14		6.4		20.4

Table 1. Extrapolated maximum performance

	1-Random-64		1-32-64		1-16-32		1-8-24		1-8-24-64	
	Steps	Error [%]	Steps	Error [%]	Steps	Error [%]	Steps	Error [%]	Steps	Error [%]
Bayes	7.6	28	9	35.1	6	35.3	13	32.5	6	32.5
Genome	4.4	0.01	4	1	7	1	4	16.5	4	16.5
Intruder	3.2	2.1	12	0.1	6	0	6	0	6	0
Kmeans High	8	8	5	9	8	7.2	7	8.2	7	16.4
Kmeans Low	4.2	11.7	3	14.6	6	14	8	0	8	14.6
Labyrinth	6.8	3.7	8	1	10	2.9	13	3.6	8	2.9
SSCA2	13	1.8	11	2.9	11	0	9	0.2	9	0.2
Vacation High	3	2.7	5	0.7	9	0.7	6	0	4	1.7
Vacation Low	3.8	1.3	6	0.8	7	0	6	0	6	0.8
Yada	7.4	0.4	7	0.5	4	0	6	0	5	0
Avg	**6.1**	**6.1**	7	6.6	7.4	**6.1**	7.8	**6.1**	6.2	8.5
Avg no Bayes	**6**	3.6	6.8	3.4	7.6	**2.9**	7.2	3.2	6.2	5.9

Table 2. Algorithm 1 convergence speed and error



(a) $f_3(n)$ (b) $f_4(n)$ (c) $f_5(n)$ (d) $f_6(n)$

Figure 8. Execution of Algorithm 1 (1-16-32) for `intruder`

vergence (both with and without including `bayes` results) and also relatively low error in both cases.

From our results, it is clear that Algorithm 1 does not converge to the optimal thread count, but it comes very close in a few steps. A refined algorithm could continue the search in the vicinity of the thread count the Algorithm 1 converges to.

Figure 8 depicts a concrete execution of Algorithm 1 for `intruder` starting with measurements for 1, 16 and 32 threads. Each of the graphs depicts the measured speedup over sequential code (full line) and a single approximation (dotted line) which represents a single step in the algorithm. The cross marks the maximum value of the performance function in the current step. In this execution after 4 approximations the algorithm converges to the correct thread count of 22.

5. Discussion

We do not advertise our approach as the silver bullet to help make the right choice for the use or not of an STM in a given application and workload. The most important limitation of our work is that the performance function f strongly depends on both the workload and the computer system. Small changes in the workload or the system might introduce significant errors in the predictions. For example, even replacing two 4-core CPUs with a single 8-core CPU, or increasing the frequency at which a certain lexical transaction is executed might significantly change the performance function.

An ideal way of predicting the performance would use an analytical function of the form:

$$performance = f_g(c_1, ..., c_N, n) \qquad (1)$$

In such equation, c_i would denote the characteristics of various components of the computer system and the executed workload itself, whereas n would denote the number of threads executed in parallel. The system components and the corresponding characteristics would include the STM features (conflict detection, contention management, the mapping of memory locations to STM meta-data, etc.), the hardware (the number of CPUs, cores and hardware thread contexts, the characteristics of the cache and the memory hierarchy etc.), as well as the operating system and the system libraries (the thread scheduler, the memory allocator, etc.). The workload characteristics would include the size of the transactions, the ratio of transactional and non-transactional memory accesses, transaction access patterns etc.

Building such function f_g is very difficult, if not impossible, because the impact of c_i can be very different, depending on various factors. For example, the impact of the memory allocator concurrency control is significant only if there are many memory allocations and deallocations. If there are only a few, then there might be no difference between using a simple single-lock memory allocator and a fine-tuned parallel memory allocator. Furthermore, it might be difficult to concisely represent some of the important characteristics. One such example are transaction memory access patterns. It is not only relevant which memory locations are being accessed, but also at which point in the execution are they accessed, as it is not the same if some transaction writes to a heavily contended memory location near its beginning or near its end. Even if detailed information about memory access patterns of transactions is constructed, it needs to be represented in a simple and concise way so it can easily be used in f_g. On the other hand, it is not completely clear whether this information is needed at all—maybe some other characteristics capture enough information about the system and transaction access patterns are not necessary. Hence our pragmatic focus on the number of threads n.

We believe that our approach could also be useful for other types of concurrency schemes, e.g. lock-based techniques. It might not be as promising as with STM, because lock-based code is typically more difficult to write than the code that uses STM and, thus, the programs required for profiling runs are more difficult to obtain. Furthermore, overheads of STM are typically much higher, making the question of performance prediction more important. Also, we believe that our technique can be, rather straightforwardly, applied to upcoming Hardware [22] and Hybrid [9] Transactional Memory and that it will, most likely, be relevant for both.

In the future, we plan to improve approximations using the knowledge about the modeled system. For example, we know that the performance cannot be negative, and also, that it typically changes rather slowly. This could help us remove some local minimums and maximums in the predicted function. Also, using runtime information provided by the STM and the computer system could help us improve our approximations. In particular, it would be very interesting to focus on predicting the performance for machines with low number of hardware threads (desktop machines). In these cases, not many measurements can be collected and used for function fitting and the runtime information could be used instead to achieve better approximations.

Acknowledgments

This work is funded by the Velox FP7 European project and the Swiss National Science Foundation grant 200021-116745/1.

References

[1] Matlab documentation. `http://www.mathworks.com/access/helpdesk/help/techdoc/`.

[2] Zunzun.com online curve fitting and surface fitting. `http://www.zunzun.com`.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06*.

[4] N. I. Akhiezer. *Theory of Approximation*. 2004.

[5] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *OPODIS '09*, 2009.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*.

[7] B. F. Carlos. A full-newton approach to separable nonlinear least squares problems and its application to discrete least squares rational approximation. In *ETNA '09*, 2009.

[8] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11), 2008.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06*.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*.

[11] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a Research Toy. Technical report, 2009.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09*.

[13] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süsskraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *Transact '07*, August.

[14] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*.

[15] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL '09*, 2009.

[16] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys '07*.

[17] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06*.

[18] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53(8), 2009.

[19] A. Heindl and G. Pokam. An analytic model for optimistic stm with lazy locking. In *ASMTA '09*, 2009.

[20] A. Heindl, G. Pokam, and A.-R. Adl-Tabatabai. An analytic model of optimistic software transactional memory. In *ISPASS '09*, 2009.

[21] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03*.

[22] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[23] W. Karl. Uber die analytische darstellbarkeit sogenannter willkrlicher functionen einer reellen veränderlichen. 1885.

[24] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Transact '06*.

[25] D. E. Porter and E. Witchel. Modeling transactional memory workload performance. In *PPoPP '10*, 2010.

[26] T. Riegel, P. Felber, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08*.

[27] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT '09*, 2009.

[28] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. F. Spear, H. Y. Wang, and K. Wang. Compiler and runtime techniques for software transactional memory optimization. In *CCPE '09*, 2009.