Self-tuning the parallelism degree in Parallel-Nested Software Transactional Memory

José Simões
Instituto Superior Técnico,
jmgs@tecnico.ulisboa.pt

This work has been partially supported by project specSTM (PTDC/EIA-EIA/122785/2010)

Abstract. Transactional Memory is a promising parallel computing paradigm, increasingly used nowadays, that allows easy parallelization of sequential programs and can expose a great level of parallelism. Many different approaches exist. One such approach is parallel-nested Transactional Memory, where transactions are allowed to spawn their own child transactions, effectively exposing additional levels of fine-grained parallelism. Transactional Memory systems often have tuning mechanisms, which adjust parameters and internal dynamics according to system measurements, to increase performance. To the best of our knowledge, tuning nested Transactional Memory systems has not been explored in contemporary research. We propose modern tuning mechanisms can be created for parallel-nested Transactional Memory, and perform an analysis using a state of the art Transactional Memory system, JVSTM. We develop mechanisms to plug different tuning strategies into JVSTM, and evaluate their effects and improvements, or lack thereof, using a set of benchmarks designed for evaluating STM systems. The results we obtained offer insights on the different tuning strategies, as well as a framework with which future work can be developed.

Keywords: Transactional Memory, Software Transactional Memory, Tuning, Nesting, Nested, Optimization, Tuning Nested Software Transactional Memory

1 Introduction

Computational capacity has been on a steady increase since the early days of modern computing [1]. The vast computing power made available to modern applications can be attributed to the advances in processor technology [1, 2] and to the many results of a steady effort which envisions the creation and improvement of powerful and scalable parallel computing architectures. Multi-core processor technologies are becoming the norm in today’s hardware. Machines with four, eight and even hundreds of cores are commercially available today. Unfortunately, writing parallel applications that efficiently exploit the available parallelism is far from being a trivial task, as parallel programs are
harder to design, implement and debug than their equivalent sequential versions [3, 4]. Given these obstacles, the research community has constantly been trying to develop new models that allow non-trivial parallelization to be efficient and simple to implement [3–7]. Unfortunately, there is no silver-bullet approach to this problem, and several existing paradigms perform better or worse than their counterparts depending on the nature of the problem they are applied to [8]. One such paradigm is Transactional Memory [9, 10], a model that abstracts parallel operations and synchronization and tries to provide performance gains and transparency when developing parallel software. TM takes advantage of the concept of transactions, widely used in database systems [10], to manage concurrent access to an application’s shared memory. TM provides elegant abstractions to wrap groups of operations in a transaction, called atomic blocks in TM jargon, and provides simple instructions to control the flow of these blocks. A companion runtime system ensures that essential properties like atomicity and isolation are met [4].

There are three main categories of TM systems: Hardware Transactional Memory (HTM) supports the transactional constructs and the concurrency control mechanisms directly in the processor unit, thus removing the need for programming languages and runtime systems to implement them independently [11]. Directly opposite to HTM is Software Transactional Memory STM: STM strives to provide a transactional framework based entirely in software runtime support. STM has received more attention than HTM recently, given that it is a more portable solution, not bound to any hardware architecture or restricted by limitations of the underlying hardware [12]. In STM, atomic blocks are handled by the runtime system transparently, but the transactional logic and structures are kept in memory, which eliminates many problems of the hardware category. Finally, Hybrid Transactional Memory [12] allows for concurrent execution of transactions using HTM and STM. Hybrid TM tries to circumvent the limitations present in HTM by using cheaper hardware capabilities, resorting to more costly software transactions in case the hardware system cannot ensure smooth progress. Our work targets STM.

In Transactional Memory systems, nesting is the act of embedding a transaction within another transaction. Nesting models can be viewed as an extension of traditional Transactional Memory in which transactions are allowed to spawn further (inner) transactions [13].

In the cases where a programmer may be unable to expose enough parallelism to exploit the available hardware threads, nesting allows transactions to spawn additional (inner) transactions ¹, enabling intra-transaction parallelism, and thus an improved overall level of concurrency.

¹ In conceptual terms, spawning nested transactions effectively creates a concurrency tree. A transaction that spawns another is said be the latter’s parent, whereas the inner transaction is its child. Any two transaction are said to be siblings if they have a common ancestor in the transaction tree.
Transactional programming constructs provide the developer with a simple and transparent means for attaining high levels of concurrency without the need to craft complicated, fine-grained lock-based synchronization schemes. This is achieved by means of nesting transactions: nesting allows composing various atomic blocks a single, bigger, atomic block.

There are a number of subtle issues influencing TM’s actual efficiency. The literature provides plenty of evidence that indicates that properly tuning its internal parameters depending on the application’s workload is crucial to obtain good performance [14–17].

One of the most investigated problems in STM tuning is how to adjust, at runtime, the number of active transactions. Several researchers [18–21] have argued that in order to take full advantage of modern massively parallel architectures, it is often desirable to support intra-transaction parallelism. This has given rise to a research line aimed at designing efficient solutions for parallel nesting in TM. Concurrency degree tuning has, to the best of our knowledge, focused on tuning non-nested TM systems. With the introduction of nesting, the problem’s domain becomes two-dimensional, or greater, if we choose more complex tuning strategies (e.g. different concurrency levels for different top-level threads). Our work focuses on tuning the concurrency degree of Nested Transactional Memory Systems, i.e. adjusting the number of available threads a Nested TM system can use.

2 Related Work

In this section we overview the various classes of Transactional Memory systems, and different self-tuning solutions that have been proposed to improve their performance.

There are multiple implementations of Parallel-Nested TM [18, 21–23]. We base our work on JVSTM [22], a versioned Transactional Memory implementation which has been augmented by Diegues and Cachopo [19] to support parallel nesting. The original JVSTM design uses versioned boxes (VBox), a concept that represents transactional locations. Each VBox stores a history of the values committed to its memory location, which are used as an undo-log when a writing transaction aborts and a rollback is necessary. Transactions access VBoxes when executing and record these accesses in their respective read- and write-sets, which are used in validation. We develop our work using JVSTM’s current implementation.

The Performance of TM systems strongly depends on the proper tuning of its internal parameters against the workload that is currently exhibited by the application: concurrency level, conflict detection mechanisms and contention management policies play a key role in attaining high performance [8, 24–26].

Concurrency degree is the level of parallelism a transactional application can use. While there are different aspects of this feature, research usually focuses on either the number of active threads or the number of concurrent transactions.
Our work is focused on tuning the number of active threads at runtime.

Performance Modelling can be abstracted as an optimization problem. There is an objective function to optimize, its parameters are extracted from a search space that contains the application’s possible parameter values. Three main categories of performance modelling techniques exist:

– **White Box Models** - White Box approaches use available expertise on the internal dynamics of a system to model performance as a set of equations that map input parameters (e.g. workload characteristics and configuration parameters) to a target performance measure. With this technique, it is possible to create analytical models or simulators that require no training, or minimal profiling of the application, to predict performance.

– **Black Box Models** - Black Box Models overcome the need of knowing system internals by using various Machine Learning techniques and search algorithms to model its behaviour and identify its optimal configuration. There are two types of black-box models:
  
  • **Online models** - Online methodologies try to quickly adapt to the workload’s behaviour without relying on training-based models. Most proposals for online concurrency control algorithms refine some sort of Machine Learning algorithm or control loop.
  
  • **Offline models** - Offline methodologies are implementations of machine learning engines that try to produce an abstract model by collecting statistics during a training phase, usually an early deployment phase. The model is instantiated with this training data and is then used at runtime to produce an optimal configuration inferred from the application’s data samples[27].

– **Gray Box Models** - Gray Box Models are a type of solution that combines white-box and black-box approaches. This category of algorithms try to achieve the advantages of both, while mitigating the disadvantages. Most solutions use a combination of analytical models and machine learning systems.

The solution we propose to develop is an online black-box tuning mechanism. Ansari et al. [15] propose an adaptive concurrency regulation system which regulates available parallelism. Their system uses the ratio of committed transactions versus the total number of transactions (which they identify as Transaction Commit Rate, TCR) to fuel a Gradient Descent (GD) algorithm that adjusts the application’s thread count.

On a subsequent study, Ansari builds on the previous model with a novel approach: weighted concurrency control [24]. The main insight in this mechanism is that better results may be achieved by (de)activating specific threads, rather than blindly adjusting the concurrency level. In F2C2[25], a control feedback loop is exploited to quickly react to workload changes. F2C2 is built on top of TinySTM [28], a state-of-the-art Transactional Memory system. This system uses an exponential search phase that doubles the thread count at every sample interval while performance increases, and stops.
when a decrease is detected. After this initial phase, it performs a fine-grained search where the concurrency level is either increased or decreased by one unit at each iteration.

3 Tuning the Concurrency Degree in Nested Software Transactional Memory

The main contribution of this dissertation is a tuning component for JVSTM. This component is a self-tuning system used when JVSTM’s nesting capabilities are in effect. Hereinafter, we refer to this component as JVSTM tuning. As highlighted in previous sections, there is a relevant gap in the literature regarding the tuning of nested STM systems. Our main objective is precisely to fill this gap, and develop a mechanism that will dynamically tune the concurrency degree of JVSTM. This section covers the various aspects of the development of JVSTM tuning.

3.1 Architecture

Tuning a nested STM system can be loosely seen as an optimization problem: There is a function that outputs a measure of the system’s behaviour in terms of evaluating its inputs. The inputs of this function, in our specific case, are the number of threads JVSTM uses for its transactions. We call these transactional threads. In the simplest case, we want to adjust the number of threads used for top level transactions and the number of threads used for nested transactions. An optimization problem has a search space, composed of the domains of its various dimensions. Thus, we define the cardinality of our search space to be two, in this simplest case. Its dimensions are the number of top-level transactional threads and the number of nested transactional threads. Based on measurements of these two inputs, our goal is to optimize the output, which corresponds to system performance.

JVSTM uses a global thread pool for nested transactional threads, while top-level threads are created on-demand. However, top-level threads stay alive after they finish their first transaction, and are used to execute future transactions. Thus, transactional threads cannot be created and destroyed at will without undermining the natural behaviour of the whole system. We modified the code for each transaction class in JSVTM to signal our tuning component before starting. Additionally, our system must we are to keep an accurate measure of the active transaction count. To this end, transactions signal

\footnote{A distinction can be made regarding the types of threads used by an application. “Regular” threads, used by the application and orchestrated by it, and the transactional threads JVSTM uses. The latter are exclusively controlled by JVSTM, and lie solely in the context of JVSTM’s runtime transactional support, despite executing transactional tasks issued by the programmer. These are the threads we target in our solution.}
the tuning system when they finish their work, either by committing or aborting. A statistics collector keeps a record of these events, which are used for tunings itself and for log-keeping, which is our main data collection method. The measure we chose as our input parameter is committed transaction throughput. Throughput is an absolute measure that reflects the number of committed transactions, and by itself does not represent any time interval. Our tuning mechanism runs periodically, in periods we call tuning intervals. Throughput is recorded by the transactional threads during each interval, and are aggregate at its end to provide a global measurement of system performance. At the start of each interval this aggregated measure is saved in a log and a new measuring period starts.

Transactional threads record their lifetime events in the respective methods (begin, commit and abort). Additionally, these methods signal our tuning mechanism via a singleton instance of the class jvstm.tuning.Controller. This class is the central interface of our tuning system. The controller runs the main tuning mechanism, which we call tuning policy, and is embodied in subclasses of class jvstm.tuning.policy.TuningPolicy. Our system dictates the number of allowed threads via two semaphores (we use a variation of Java’s standard Semaphore class), one for top-level transactional threads and another for nested ones. The tuning policies directly control the number of permits available in these semaphores, which transactions must acquire to start and release to finish.

Thus, we accomplish a simple mechanism to control the system’s concurrency degree.

Tuning policies contain the main algorithms for analysing system performance and produce subsequent concurrency configurations to be applied. Various policies exist, which implement different tuning strategies.

### 3.2 Algorithms

Our work strives to provide insight on the matter of using different approaches to tune JVSTM, and thus we implement several algorithms in our tuning policies. This section covers these algorithms and their purpose.

Gradient Descent (GD) is a thoroughly studied category of optimization algorithms used for solving a variety of problems. Gradient Descent methods are based on the observation that the set of consecutive values a function takes decreases fastest if one follows the direction of the steepest slope in that function’s codomain. This applies to our problem specifically in the case of exploiting regions in the search space that contain a set of points whose coordinates (top-level x nested) cause performance to increase. There are two policies that use the DG approach:

- Linear Gradient Descent (LGD) - From the perspective of our 2D search space, LGD scans four points in the vicinity of the current point. Visually, these adjacent points can be observed be immediately above, below, to the left, and to the right of the current point. Each LGD execution sets one of these points as the current system configuration, and saves its results
at the end of the tuning interval. After measuring each point (the center point and the four neighbouring points), LGD chooses the point with the best throughput and sets it as the current point, after which it re-starts the process.

- Full Gradient Descent (FGD) - FGD is an extension of the previous policy which scans all the neighbouring points relative to the current point. All the aspects of this policy mirror the ones used in (acro) LGD, with the exception of the sampling algorithm: Instead of extracting four points, FGD samples all the neighbouring points (i.e. eight points).

FGD takes longer than LGD to scan the neighbourhood of each point in the search space, but has more information with which to make an informed decision to tune the system’s configuration. The additional time FGD takes to gather information results from a trade-off that affects all policies: The more points a policy explores, the longer the period of time it spends before discovering a promising point and starting exploitation.

The Hierarchical Scan (HS) policy samples points by selecting them from scanlines. We define scanlines as line-like regions in the search space. This policy explores the hypothesis that points in these regions yield better performance results than their neighbouring regions. Visually, this notion could be represented by a long line of points whose throughput value is greater than their neighbours.

Excluding its sampling algorithm, HS mirrors the patterns used in LGS and FGS. Specifically, after a scanline is sampled, the most promising point is selected and exploited for one tuning interval.

The F2C2 policy follows the work developed by the authors of [25], where the authors apply some of the TCP protocol’s mechanisms to tuning a non-nested STM system. There is an initial exploration phase, which performs a coarse-grained search over the search space, and a subsequent exploitation phase, or fine-grained search. The exploration process exponentially increases the number of allowed transactional threads (this is performed in each consecutive tuning interval) until one of these increments yields a throughput measurement that is lower than the previous measurement. This search pattern attempts to quickly converge on a broad region of interest in the search space, by avoiding exhaustive searches. The exploitation process imposes unitary fluctuations on the system’s configuration, and follows the steepest slope in a manner similar to GD. The objective of this phase is to comb over the region of interest in detail, and approximate the configuration to an optimal point. The F2C2 policy uses a variation of this search algorithm, adapted for two dimensions.

Recursive Random Sampling is an algorithm that tries to avoid inefficiency when performing Random Sampling (RS). As shown in the authors’ published work, RS is guaranteed to eventually converge to a global optimum. However, after a certain number of sampling steps, its efficiency drops exponentially, which often leads to long periods of time before convergence is achieved. The authors propose to avoid this inefficiency in the algorithm by reducing or re-aligning the search space size before this inefficient phase is reached. After a certain number of samples is taken from the search space, the algorithm either reduces its size or
re-aligns it (i.e. moves its centre to a point that yields good performance). The RRS tuning policy uses this algorithm to scan the search space and converge on optimum points.

4 Experimental Results

The simulations we ran intend to answer the question of whether the tuning system and the tuning policies we created yield performance gains when applied to JVSTM. This section describes our testing environment, details the results we obtained, and discusses their implications.

4.1 Experimental Setup

We ran the totality of our tests in a machine with 4 AMD Opteron 6168 processors, with 12 cores each, yielding 48 total processing cores, and 128 GB of RAM, running Linux and the Java Runtime Environment version 1.6.0_43. We use two benchmarks for testing our system: Vacation and STMBench7. Vacation belongs to the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite [29], a set of benchmarks for transactional memory systems that tries to provide realistic and diverse use cases for a transactional application. Vacation emulates a travel reservation system by maintaining a database implemented as a set of trees. This structure stores the identification of clients, as well as their reservations for varied travel items.

For Vacation we compare the execution time of the benchmark execution. By design, Vacation does not take initialization and finalization times into account, as the execution times of these processes are not relevant and could introduce bias in the results. This measurement is obtained directly from Vacation’s output. Each value presented is the average of three test runs.

STMBench7 is a benchmark for STM systems that tries to emulate a complex and realistic application [30].

The data structure used by STMBench7 consists of a set of graphs and indexes intended to be representative of complex, real-life applications. Several operations are supported to model a wide range of concurrency patterns and workload profiles. STMBench7 usage is simple to configure. Users may choose a workload type, number of top-level and nested threads and duration of the benchmark. Additionally, users can enable or disable structural modification operations and long graph traversals. These two aspects are not relevant to our work and are disabled throughout our tests.

In STMBench7 tests we measured only throughput. When STMBench7 finishes, it outputs a log with extensive information. We filter the average throughput and map its value on a plot for different tuning policies and starting configurations. We normalize this value, i.e. we divide all throughput measurements by the default policy’s measurement. This way, the plots convey a clear notion of speedup relative to the baseline JVSTM.
4.2 Results

This section presents the results we obtained by running tests for each benchmark, as described in the previous section. Each subsection discusses the implications these results.

Vacation

![Vacation Execution Time Comparison](image)

**Fig. 1.** Execution time results for Vacation with high contention. The X axis represents the system’s concurrency configuration, and the Y axis execution time. Lower is better.

![Vacation Execution Time Comparison](image)

**Fig. 2.** Execution time results for Vacation with low contention. The X axis represents the system’s concurrency configuration, and the Y axis execution time. Lower is better.
Figure 1 details execution time comparison for Vacation with high contention, and figure 2 for low contention. The results obtained show that there is no performance gain associated with tuning the system’s configuration for this benchmark. Tuning generally performs worse than the baseline JVSTM. This is slightly diluted with high contention configurations, and is expected, as in this configuration the system experiences many more conflicts than with low contention. This in turn leads to the performance losses being smaller with high contention configurations.

The Gradient Descent policies generally perform poorly. This is associated with the slow sampling method they use. In a search space with many local optima and high variation between configurations, these fail to quickly converge to a suitable configuration, and become stuck either in local optima or in long sampling phases that do not inspect a relevant portion of the search space. RRS is the policy with the smallest performance loss. This effect is associated with its efficient sampling algorithm, adapting to Vacation’s highly unstable workload behaviours.

STMBench7

This section details the performance results we obtained with each tuning policy, for different starting configurations of STMBench7, and different workload types. Contrary to Vacation, STMBench7 results are presented in terms of speedup obtained via measuring the system’s average throughput. STMBench7 is built with this specific measurement in mind, and the execution time is fixed for each execution. Thus, we cannot use the latter metric as a performance indicator. The plots display a horizontal line that represents unitary speedup, i.e. the performance of the default policy, which reflects JVSTM’s baseline performance. Speedup is a measure of how much better a given configuration performs relative to another, e.g. a speedup of two means the system yielded twice the average throughput of the baseline, whereas a speedup of 0.5 represent a halving of the baseline throughput.
Figure 3 presents the results for a read-only workload (meaning the system experiences low contention), with pre-set starting thread configurations that represent different levels of nesting. Gradient Descent policies generally perform worse than the baseline policy, except with high levels of concurrency (i.e. high numbers of both top-level and nested threads), where performance is similar to the baseline, and in one case (FullGD - 8x3, or 8 top-level threads and 3 nested threads) much better, with a speedup value of 2. We surmise that this particular thread configuration is close to the system’s optimal configuration, though which FullGD quickly converges. Hierarchical scan is on par with the baseline, except for the case of the 2x3 configuration (2 top-level threads and 3 nested threads). We surmise that this configuration is contained within one of the linear regions described in section 3.2. F2C2 performs on par with the baseline, as does RRS. RRS displays one spike in throughput, also in the (2x3) configuration. We believe this may be a statistical quirk, as RRS has no particular reason to favour a given starting condition, given that it randomly scans the whole search space.

Figure 4 presents the results for a mixed read-write workload, with pre-set starting thread configurations which represent different levels of nesting. Generally, no policy yields promising speedups in this type of workload, and at best remain on par with the baseline. Small speedup improvements are achieved in specific cases, but are not enough to justify the adequacy of the policy that yielded them to this workload type. We surmise that mixed workloads will
have high levels of noise in the search space performance levels, and are thus difficult to tune.

Fig. 4. STM Bench7 throughput test results - read-write workload.

Fig. 5. STM Bench7 throughput test results - write-only workload.
Finally, figure 5 presents the results for a write-dominated workload (the system experiences high levels of contention), with pre-set starting thread configurations which represent different levels of nesting. Almost all policies are able to at least remain on par with the baseline. Starting configurations with high level of nesting experience the highest levels of contention, and thus are ideal candidates for tuning. These show remarkable speedup when tuned, particularly in the (4x3) configuration, where Gradient Descent policies seem to perform well and exhibit high levels of speedup. This may be attributed to this starting configuration being close to a global optimum. These improvements in system performance are expected, as high-contention settings are the best candidates for our tuning system. The high level of conflicts implies that there is a great amount wasted work, which the tuning system reduces when it finds a more suitable configuration. Thus, write-dominated workloads show the best improvements in performance.

5 Conclusions and Future Work

The core of the work developed for this dissertation was the creation of a tuning component in JSVTM. We created several tuning policies and analyse their effects in this document. We used two benchmarks, Stamp (Vacation) and STM-Bench7, to assess the gains of the tuning component we created, with respect to each of its tuning policies. No single policy appears to be ideal, as the majority of them produce spikes in throughput on some specific configurations, but otherwise show neutral or negative effects on the system’s performance. This is an expected result, as this research had an exploratory nature. However, we believe the insights provided by our experimenting, and the data we collected are valuable contributions to current understanding and future work on this subject.

There is a vast number of algorithmic approaches that can be explored in future work to enhance the data we provide and lead nested STM tuning to be a pervasive performance enhancement technique on nested STM systems. The two key points we distinguish in a tuning policy are sampling and exploration/exploitation. Our results show that robust sampling techniques can converge on optima configurations faster than simple hill-climbing or gradient descent techniques. An equilibrium in exploration and exploitation phases is also critical, and different strategies for tuning this aspect are available in the literature.

References


