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

José Simões

Instituto Superior Técnico,
jmgs@tecnico.ulisboa.pt

Abstract. In this document we present an overview of Transactional Memory models, as well as a comparison to traditional parallel frameworks. In this context, we follow with an analysis on the problem of self-tuning concurrency control in transactional memory, where different mechanisms are used to predict and adjust an application's concurrency level with respect to available parallelism. We are particularly interested in tuning nesting Transactional Memory systems, an aspect that has so far been left unexplored. We review a series of case studies on the various types of concurrency tuning in different transactional frameworks, and propose a new mechanism to tackle self-tuning in nesting Transactional Memory based on techniques used in the literature.

Keywords: Nested Transactional Memory, Parallelism Level, Concurrency Control, Self-Tuning

1 Introduction

Transactional Memory

Multicore processor technologies are becoming the norm in today's hardware, and show a promise to dominate most platforms in the near future. Machines with four, eight or 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. The typical approach to write parallel application relies on locks.

In the scope of this study, a distinction can be made in the parallel development methodologies: traditional lock-based synchronization and Transactional Memory[1]. Traditional lock-based synchronization frameworks in application development usually provide two alternatives. The first is a coarse-grained approach, in which a few locks are used to regulate access to huge portions of code. While very simple to implement, this solution is naturally prone to inefficiencies, as only a few very long critical sections can be executed in parallel. The second alternative is a fine-grained approach, which aims at extracting as much parallelism as possible from the application, by using locks to guard critical sections as small as possible. This approach avoids the inefficiency of the first one, but it is subject to pitfalls like deadlocks, livelocks, and data races. Furthermore, it adds complexity to the code and hinders its maintainability.

In light of these considerations, none of the two approaches appears to be a suitable reference parallel programming paradigm, as they either sacrifice efficiency in exploiting available computational power or code maintainability. For these reasons, many efforts have been made to devise a mechanism to facilitate parallel development and efficiency on these architectures.

Transactional Memory (TM) is a concept derived from long-standing practices in database development[2], which promises to simplify concurrent programming via simple constructs, and improve efficiency and scalability the way fine-grained locking does. TM borrows the construct of transaction from databases to be applied in parallel code blocks (hereafter referred to as *transactions* or *atomic blocks*) via the abstraction of atomic blocks: A transaction is either successful, in which case all modifications are committed to memory and their effects become globally visible, or fails, and the whole block aborts, and no modifications are ever observed out of its context. A transaction is said to abort when a conflict is detected. Conflicts in Transactional Memory can be due to a variety of causes, depending on each TM system's inner details, but the fundamental underlying source is the concurrent modification of the memory state, in such a way that isolation is no longer preserved between concurrent transactions (i.e. transactions no longer observe coherent memory values).

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. These constructs hide the details and mechanisms necessary to synchronize parallel execution, which is managed by the runtime support system. Not only does TM enhance code reliability and maintainability, but it also overcomes one of the fundamental issues of lock-based concurrency: The possibility of composing multiple parallel libraries without incurring major code changes and performance penalties or pitfalls like deadlocks/livelocks. This is achieved by means of nesting transactions, i.e., embedding a transaction within another transaction. Transaction nesting allows different TM code blocks to be safely combined into a bigger single atomic block. This capability can also be exploited to increase the parallelism of TM applications, by having multiple nested transactions, enclosed in the same parent transaction, to run in parallel (a.k.a. parallel nesting)[3].

Despite the fact that TM simplifies parallel programming to a large extent, there are still a number of subtle issues influencing its actual efficiency. The literature provides plenty of evidence that indicates that properly tuning its internal parameters depending on the applications workload is crucial to obtain good performance [4–7]. One of the most investigated problem is how to adjust at runtime the number of active transactional threads so as to maximize performance. However, several researchers 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.

The key challenge in tuning the intra-transaction parallelism is the dimensionality of the search space. In fact, as opposed to traditional solutions that only have to tune the number of concurrently active threads, tuning the intra-transaction parallelism also requires the tuning of the number of nested transactions to be spawned by each active top-level thread. Not only does inner parallelism add an extra dimension to the parameter space, but also, if we assume that different top-level transactions may require different degrees of inner parallelism [8], additional complexity must be added to the concurrency control mechanism to deal with this aspect.

This document presents a summary of the state of the art solutions in TM, focusing on the problem of self-tuning the parallelism degree, and serves as a knowledge base to investigate the design of self-tuning algorithms for regulating the degree of concurrency. Our main objective is to create one or more self-tuning algorithms for concurrency control in nested Transactional Memory. To this extent, we focus a mature nesting TM implementation which will serve as the baseline for further development.

Diegues and Cachopo,[9] developed a practical parallel nesting algorithm built on top of JVSTM [10], a Java STM implementation. Few TM implementations enable nesting, and this proposal introduces a sturdy algorithm that manages to overcome the additional challenges that nesting poses. The JVSTM implementation of Diegues and Cachopos algorithm will serve as baseline for our work.

The scope of this study is to survey existing techniques and methodologies of concurrency tuning to prepare further work. Our main goal is to design one or more algorithms to dynamically tune the concurrency level in multi-dimensional STM systems. The main challenge is to keep the overhead of our algorithm to a minimum, enabling near-optimal configurations to be found in usable time, without undermining the benefits of nested parallelism.

The remainder of this document is structured in the following manner:

Section 2 overviews different Transactional Memory models (Subsection 2.1), the base techniques and types of concurrency control (Subsection 2.3), and concrete solutions present in TM literature (Subsection 2.4).

Section 3 details our main goals, with a review of the main challenges and project planning.

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.

2.1 Transactional Memory

Transactional Memory is a vast area of research, in which a lot of different systems and algorithms have been developed. This section categorizes and explains the different aspects of these systems.

Hardware Transactional Memory

Hardware Transactional Memory is based on the idea of supporting the transaction construct and the concurrency control scheme directly in the processor unit[11]. Transaction logs and verification logic are handled by the hardware, usually taking advantage of the existing cache coherence protocol to validate transactions before committing them to memory. In HTM, critical sections execute concurrently unless the system detects that they need to be serialized to ensure correct execution. As HTM poses additional restrictions on how a transaction ca fail (such as insufficient onchip resources, context switches and system interrupts), HTM systems can be classified as *best-effort approaches*. Two fallback mechanisms are usually present to ensure forward progress when hardware transactions abort[12]: old-fashioned locking schemes that acquire a lock during the execution of an atomic block, and developer-defined software fallback paths, that can use custom code to deal with the conflict.

One useful case study is a research paper evaluating the Intel Transactional Synchronization Extensions [13]. Intel TSX is a hardware transactional support specification included in fourth-generation Haswell processors. TSX provides two mechanisms to achieve transactional capabilities: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE is a legacy compatible instruction set extension that tries to eliminate lock access in favor of transactions. The RTM mechanism, on the other hand, exploits a new, more flexible instruction set that explicitly enables the definition of transactional blocks and software fallback paths. TSX, as most other proposals, works on a best-effort basis. There are numerous architectural conditions that can cause a transaction to abort, such a data conflicts, buffering limits and instructions that always abort (e.g. system calls, context switches and I/O operations). These are at the root of HTM limitations. Several proposals have successfully dealt with these problems, but the developed solutions impose complex restrictions and complicated logic to be integrated in processor design, making it unlikely that they will be adopted in a near future [14]. An in-depth analysis of HTM systems lies outside the scope of this document, as our main focus is to provide an overview of the main TM technologies currently available. More information on HTM specifics can be found in [11, 14].

Software Transactional Memory

Software Transactional Memory is a category of systems that provide Transactional Memory capabilities through software runtime support only. In the last years STM has received more attention than HTM, given that it is a more

portable solution, not bound to any architecture or restricted by limitations of the underlying hardware. 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.

Most STMs keep a map of memory locations accessed within transactions. These structures can serve additional purposes, such as locking, ownership record, versioning, and log keeping[1].

The main drawback of STM systems is that they require transaction boundaries and operations to be instrumented, in order to keep track of their accesses to memory[2]. This leads to overhead, which may hinder performance and hamper the scalability of the application.

In comparison, STM systems can incur heavy instrumentation costs, which are avoided by HTM. Atomic code blocks are modified at compile time to account for transactional operations and log-keeping, as well as a plethora of details that are required by the runtime transactional support system.

Hybrid Transactional Memory

Hybrid Transactional Memory (HyTM) allows for concurrent execution of transactions using HTM and STM. HyTM tries to circumvent the limitations present in HTM by using cheaper hardware capabilities, resorting to more costly Sofware transactions in the fallback path to ensure progress[14]. As already mentioned, solutions for the buffering limits and context switch survival problems impose a series of restrictions on hardware design, and are thus unlikely to be adopted soon. HyTM exploits HTM, if available, with a combination of methods that fallback to STM if a transaction cannot be completed. To interleave HTM and STM, the code is instrumented at compile time to follow one of two paths for each atomic block, each one corresponding to a different TM type.

This category of implementations must define a contention management policy that decides when to resort to STM instead of restarting the transaction in hardware, as for the reasons described in the HTM section, there is no guarantee that a hardware transaction will ever succeed. Of course, to perform as a coherent whole, the HyTM system must be able to detect conflicts in hardware and software transactions simultaneously. This requirement is usually achieved by augmenting the hardware transactional structures[14], extending their visibility to the STM runtime so that the conflict detection system has access to both sides' information.

Hybrid TM shows a promising decoupling between hardware support and generic HTM usage, and can benefit from concurrency tuning approaches in both hardware and software transactional support. Although theoretically enticing, the mechanisms needed for conflict detection between the two underlying systems usually introduce large overheads [4].

Nesting in Transactional Memory

Parallel Nesting models can be viewed as an extension of traditional Transactional Memory in which transactions are allowed to spawn further (inner) transactions[3]. This section clarifies the challenges that come with its implementation, and explains some concepts needed to understand its challenges.

In some cases TM systems may be unable to expose the maximum available parallelism, for example when only a small portion of the total application code can be parallelized using transactions. Nesting allows these undersubscribed top-level transactions to spawn additional (inner) transactions.

Various models are defined in literature, which fall in two main categories: sequential and parallel nesting. Sequential nesting models in TM allow transactions to be nested, but serialize them implicitly [15]. This means that code logic in parallel blocks becomes clearer, but additional parallelism is left untapped. Briefly, three models of parallel nesting are defined in literature. In flat nesting, parent transaction sees all modifications to program state made by inner transactions, but an aborting child transaction also causes the parent to abort. Closed Nesting is similar but allows for a given nested transaction to abort without aborting its parent. Last, in open nesting any committed transaction's state remains globally visible, even if the parent transaction aborts. Although flexible, this model can introduce problems due to inconsistent program states[3].

Parallel nesting proves to be an exceptionally difficult problem in traditional lock-based concurrency models, as coherence in the interplay between locking mechanisms is very hard to achieve, and code logic tends to become extremely complex [9]. On the other hand, transactions provide a transparent and simple means of abstracting parallel regions without these complications, and therefore nesting becomes a viable possibility. Most nesting models face correctness challenges however, because ancestor-descendant relationships[1] must now be taken into account when designing TM systems.

There are multiple implementations of Nesting TM.

PNSTM [16] uses a global work-stealing queue to distribute tasks (transactions) between active threads, and each transactional location maintains a stack that registers the accesses performed by active transactions. PNSTM's transactions inherit their children's read- and write-sets, and only top-level transactions can commit values to memory. Inheritance is done in a lazy manner, to improve efficiency, but this implies that false conflicts can occur, as the logs are not cleared until the top-level transaction is completed.

NeSTM [17] is based on McRT-STM [18], a traditional blocking STM that uses eager conflict detection and undo logs for writes, at word granularity. Transactions lock memory locations at encounter-time, and the authors have extended McRT-TSTM's locks to provide additional fields and visibility, to manage ancestor-descendant relationships.

[1] 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.

JVSTM [10] is a versioned Transactional Memory implementation which has been augmented by Diegues and Cachopo [9] to support transactional 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. The extension proposed by Diegues and Cachopo augments this model with a clever and efficient design to allow nesting and manage ancestor-descendant relationships in nested transactions. VBoxes now store both committed and tentative values. Tentative writes are inherited by parent transactions when a child finishes, merging them into its write-set. Parent transactions successively inherit these tentative values until a top-level transaction commits. Transactions keep two counters, *nClock* and *ancVers*, which are accessed by their descendants and restrict the versions of a VBox they can read, to maintain temporal coherence between siblings and descendants.

2.2 Factors that affect performance in Transactional memory

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, and a number of studies exist that propose refinements and new designs in an attempt to reach satisfactory performance [8, 19–21].

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.

Contention management (CM) policies also influence TM's performance. Contention management determines the behaviour of transactions when a conflict is detected. Several CM techniques exist, again with different results on varying workloads. Some policies perform generally better [5], but have low performance on certain application profiles. There are a number of different policies based on the key ideas of assigning priority to transactions based on their state and backing off when a conflict is detected. Transaction threads can also be made to wait, in case of repeated aborts with the same underlying cause. Contention management is an important component in any TM system, and must be designed carefully.

Conflict detection schemes have been optimized in various ways when the TM's architecture allows it. Most designs try to minimize wasted work, via reducing the probability that a certain conflict will repeat, and quickly invalidating the correspondent transaction. As seen in some case studies, certain lazy conflict detection algorithms can cause inexistent conflicts, or worse, execute all code in a certain transaction only to have it abort when it's finished.

Nested TM involves a great deal of additional operations, given that sibling conflicts and parent-child relationships must be managed. Most models assume

that only top-level transactions can commit values globally, which implies that logs must be inherited by the parent when a nested transaction finishes. The way in which this inheritance is performed greatly influences the outcome, as conflicting transactions should be immediately aborted to prevent wasted work in top-level tasks. Additionally, most implementations provide a fast path for committing values in reading or writing transactions, given certain conditions that usually involve concurrent accesses being performed by transactions in one single tree branch (of the nested transaction tree). These conditions also allow commit operations to be performed quickly, usually when no inter-branch conflicts exist. Of course, these assumptions are specific to each implementation, and the vary widely between them.

2.3 Self-Tuning in Transactional Memory

This section details the main categories and techniques used in performance modelling for TM.

Dynamic configurations provides a great deal of adaptivity, as no fixed configuration works best for all application workloads. Dynamic configurations are produced via a tuning mechanism, embedded in the transactional system's runtime, that monitors some measure of the application's behaviour and adapts the configuration's parameters in real-time. This monitoring is done in sampling periods, because instantaneous fluctuations in performance are not enough to determine if there was a change of the workload's characteristics.

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. To ensure mathematical tractability this type of models are usually built around a set of simplifying assumptions on how the target system behaves, which introduces a weakness to scenarios where the underlying assumptions do not hold, e.g. specific areas of the configuration space. Additionally, analytical models are immutable (aside from re-evaluations of internal parameters), which prevents re-adaptation at runtime. White Box models provide strong foundations for analysis and comparison of different Transactional Memory systems, in terms of their dynamics and behaviour.

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 at its optimal configuration. Machine Learning techniques build a statistical performance model by observing the system behaviour under different configurations and workload profiles, and usually achieve high accuracy when predicting performance. Black Box models can be coarsely grouped in two classes: Online and offline.

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. These samples contain, respectively, a given system configuration and the resulting performance metric. This information is assumed to be representative of the parameter space, but in practise this assumption is too coarse. The search space of all the possible configurations is generally too large for a fully representative sample to be taken, and grows exponentially with the number of parameters (called features ML methodology) to be considered.

Two example techniques that are commonly employed to develop offline black box models are Artificial Neural Networks (ANN), and Decision Trees (DT)[22].

Offline techniques offer a great deal of accuracy when predicting near-optimal target configurations for transactional systems. They are, however, subject to bias and fitting problems, usually caused by inadequacies in the training data. These inadequacies can be seen as the product of the training phase not being sufficiently representative[4], as training times grow exponentially with the number of considered parameters and sample size. There is also the hindrance of training the mechanism in early deployment stages, which in certain applications may be unfeasable to perform.

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. Search algorithms that map well to concurrency control come in great variety, but usually must follow two conceptual steps: Exploration, or sampling, and exploitation[23]. Exploration, tries to broadly scan the parameter space to identify areas of interest, where there is a grater probability of obtaining better performance. This is also called sampling phase because well-known sampling approaches are used to obtain representative points for each parameter, which are then fed to the second mechanism. During the exploitation phase, on the other hand, the available knowledge is used to determine the optimal point of the parameter space. Examples of online learning algorithms, which are often applied to the problem of self-tuning the degree of concurrency include Gradient Descent, Simulated Annealing, Reinforced Learning algorithms, etc.

Systems that tune the concurrency degree must quickly react to workload changes. This leads to a need to quickly dispose of uninteresting solutions, as

well as clever exploration algorithms that swiftly focus on performance peaking values. These factors contribute to a general necessity of managing the jumps between exploration and exploitation in an elegant way. Sudden workload changes may require the system to ignore the current exploitation phase, or restart the exploration at any time.

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 key drawback of black-box models compared to white-box ones is that the former typically require observation of a large number of samples in order to derive a model of the target system, i.e. it requires long training phases. Analytical models, on the other hand, avoid the need for long training phases as they exploit a-priori knowledge on the internal systems dynamics.

Offline Machine Learning approaches usually take a heavy toll on application performance, due to requiring expensive profiling on the application's behaviour. On the contrary, analytical methods require few samples, but most do not account for workload changes during the application's lifetime, and are usually instantiated with values averaged from a whole execution. Gray Box models try to circumvent these restrictions by combining online Machine Learning techniques with White Box models.

Gray Box approaches can be broadly divided in three categories:

- *Parameter Fitting* [4] relies on fitting techniques to identify areas of a subset of input parameters whose direct measurement is undesirable. Fitting techniques aim to determine ranges of values that minimize the model's prediction errors over the training set.
- *Divide and Conquer* [24] techniques consist of building models for separate components of the target system, using AM or ML, which are then combined to obtain a prediction of the system as a whole. This approach is mainly used in scenarios where different sub-components have very different dynamics. Additionally, it allows the usage of ML techniques where AM would be unusable, particularly when the system's dynamics are too complex to model using White Box approaches.
- *Bootstrapping* [23] relies on building a synthetic training set for a ML predictor by using an AM component, avoiding expensive initial profiling and training phases. The ML component is re-trained over time as new data becomes available.

2.4 Self-tuning solutions for Transactional Memory

In this section we provide an overview of some of the solutions that fall in the previous categories. Each subsection details a light analysis on the solution, highlighting specific aspects that we find relevant to our work.

White-Box-Models

To the best of our knowledge pure White Box Models have not been used directly in self-tuning the degree of concurrency, but rather to support off-line what-if analysis studies and compare performance of alternative TM algorithms.

In [25] Heindl and Pokam introduce an analytical description of STM operations using a Discrete Markov Chain that represents a single transaction, as opposed to the whole system. Generalized transactional behaviour is captured by computing the parameters of the single-transaction model, which presumes all transactions have a similar probabilistic behaviour.

A state machine models each transaction's behaviour, where each state i represents a transaction that has performed i successful operations, that can be a read or write according to a given probability (the authors assume no particular access order, and therefore use a linear probability model for each access type). State transitions represent successful operations (with a given probability), while a transactional abort causes a restart in state 0. The model does not account for complex contention mechanics, and assumes an aborted transaction restarts immediately.

The authors underline that their work should adapt to different systems and serve as framework for parameter and performance analysis on Transactional Memory systems.

Di Sanzo et al. propose a different model [7] for commit-time locking STMs to overcome two weaknesses in the previous model. Their study adopts the assumption that threads alternate between transactional and non-transactional code, whereas Heindl and Pokam assume a continuous transactional workload. Further, in the first work, transactions are abstracted over time as a series of steps whose duration is unspecified, which prevents the forecasting of time-related performance metrics, such as response time or throughput.

This second model includes internal STM characteristics, and a heavy workload- and application- specific parametrization. These enhancements are accomplished with the use of continuous-time Markov Chains, aided by Queueing theory methods. The model is validated using simulation engines based on the STAMP [26] benchmark suite, where some different aspects of the system are evaluated, with satisfying results.

Black-Box Models

The Black-Box category of models has a number of distinctive proposals and implementations. Concurrency control algorithms based on Machine Learning techniques, Search algorithms, and a variety of other methods exist. Some proposals

even build mechanisms to dynamically switch some component or functionality of the system according to behaviour measurements.

In [27] a feedback-directed mechanism is proposed to switch the underlying implementation of atomic blocks at runtime, for specific conflicting transactions. The authors analyse different atomic block implementations in the Haskell programming language[2], and select two specific approaches (optimistic and pessimistic TM) for handling different types of behaviours when transactions conflict.

The main insight of this proposal is that repeating conflicts have a common cause in "hot" variables accessed by the application's various atomic blocks. In order to identify these hot variables the authors take advantage of Haskell's type system, and extend the runtime structures that represent transactional values (TVar) with a "blame" mechanism. Transactions that fail to commit have their written variables "blamed" for the conflict, and when a given variable exceeds a given threshold it is promoted to a "pessimistic transactional variable". These pessimistic TVars are dealt with internally by a pessimistic transactional scheme, which uses multi-reader single-writer (MRSW) locks to maintain serializability.

Overall results are encouraging, for comparisons with Haskell's native Transactional Memory mechanisms. This approach is, however, restricted to its environment, as many of the methods used are enabled by native structures in Haskell, which limits portability to other languages and transactional platforms.

Ansari et al. [5] 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. Four control models are presented, which range from a simple increase or decrease in thread count to a model that exponentially modifies this quantity, as well as the sampling intervals.

A fifth model, called *P-Only Transactional Concurrency Tuning* (PoCC) solves the previous models' problems by balancing the changes applied to the concurrency level, as the previous models proved too unstable.

Overall, Ansari et al. set a basis for concurrency control with their work, and achieve satisfying results over the baseline implementation.

On a subsequent study, Ansari builds on the previous model with a novel approach: weighted concurrency control [8]. 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.

The author shows that TCR rates between different threads of an application can vary widely, and develops a framework for (de)activating them according to specific measurements. This framework is built on top of the same system as the previous model, which uses a global thread pool, where each thread has a double-

[2] Haskell has native support for atomic blocks.

ended queue (deque) for work-stealing. This specific design is not required, but facilitates the implementation and testing of the system.

Four models are developed in this framework, which are permutations of methods for sorting and selecting threads for (de)activation. Overall, the more complex a model is, the greater the performance penalty associated.

In [19], 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. The authors distinguish between *scalability-limited* and *fully scalable* applications. In the former a system for regulating the degree parallelism must not only be able to quickly converge on optimal dynamic configurations at runtime. In the latter, the system must also minimize unnecessary overheads, where concurrency control is normally not needed or has very limited effects.

The proposal borrows some ideas from the TCP protocol's congestion control algorithm, namely:

- *Slow Start* is a mechanism that increases the search window when the algorithm starts. In F2C2 this concept maps to an exponential search phase that doubles the thread count at every sample interval while performance increases, and stops when a decrease is detected.
- *Congestion Avoidance* is a fine-grained search phase where the concurrency level is either increased or decreased by one at each iteration. This phase starts immediately after Slow Start.

The fine-grained search phase of F2C2 can be seen as a Gradient Descent algorithm applied to a Feedback Control Loop, whereas the coarse-grained phase can be thought of as a global sampling phase, where areas of interest are identified and a local search is then performed. F2C2 does not restart the coarse-grained search algorithm, and is thus prone to reacting slowly to abrupt workload changes. The fine-grained search phase is accomplished by imposing unitary fluctuations to the concurrency level, and measuring performance.

The detection component uses transaction throughput, defined as *tpi*, transactions per unit time, as a global measure of the system's performance, but does not average the total *tpi*. Instead, the authors avoid global synchronization in this measurement[3] by sampling a specific thread, chosen at random, and taking that thread's tpi as representative of the system's performance. As seen earlier, on Weighted Concurrency Control, this assumption may not hold for specific workloads or application designs.

Despite these pitfalls, F2C2 introduces clever mechanisms to exploit feedback control loops, which we will consider in our work, and obtains good results in experimental evaluation.

Machine Learning approaches to concurrency control are usually composed of three main components: a statistics collector, which takes measurements of the application performance at runtime; a machine learning module, which consumes

[3] The authors avoid global synchronization in measurements to avoid oerhead

measured values and produces a value for a target parameter; and a control algorithm, which actuates on the concurrency level based on the obtained target value and mediates the interactions between the statistics collector and ML module.

In [21], Rughetti et al. design a system that uses a Neural Network to characterize performance metrics at runtime and produce a corresponding concurrency level to apply. The Statistics Collector implemented in their work collects a set of parameters common to machine learning in concurrency control, namely average read- and write-set sizes, and execution time for transactional and non-transactional code blocks. In addition, two indexes are calculated based on estimated probability distributions, read-write affinity and write-write affinity, which represent an estimation of the probability of a given read or write operation to generate a conflict.

Comparing their model to the baseline TinySTM implementation yields increasing levels of performance with increasing maximum thread count, and slightly worse results with low maximum thread count.

In a more recent development, Rughetti et al. improve upon their earlier design with a key feature for machine-learning concurrency control: feature selection. The authors propose a mechanism to dynamically adjust the cardinality of the set of input data, following two key observations: (i) some feature values may show small variance during given time windows, and (ii) some features may be statistically correlated to other features during certain time intervals. In their particular case, where the target function describes wasted transaction time (w_{time}), the authors conclude that variations in this measurement do not depend on features with small variance, and that measurements of a single feature in a set of correlated features are representative of the whole set, given that certain conditions are met.

After experimental validation, the authors publish strong evidence that correlation between features does indeed occur in a substantial set of benchmarks, as well as reduced variance in certain values, namely read-write and write-write affinity. Additionally, tests are performed to assess the gains of shrinking feature sets. These tests demonstrate that a reduction of up to 90% can be obtained in worst-case scenarios, where thread count is low[4].

To restart the algorithm, when the current feature set becomes no longer representative of the system state, the control mechanism enlarges (or resets) the feature set when the ML predictions begin to decline when compared to the real result[5]. The authors choose to use the weighted root mean square error between predicted and measured values.

[4] Test results also demonstrate that overhead from sampling tends to scale down as the number of concurrent threads increase.

[5] As some features have been excluded, no recent values exist to verify if conditions have changed.

Rughetti et al. propose a Black Box approach to tuning the parallelism degree in HTM systems[29]. The main insight of the authors' work is that traditional parameters and measures used in concurrency tuning cannot be efficiently captured in HTM systems. Additionally, the authors propose a change in the feature set used, as the features used in previous STM concurrency tuning systems are not available in HTM. To this end Rughetti et al. develop a classification-based approach, relying on Machine Learning mechanisms that predict the optimal level of parallelism. The classification-based techniques reduces overhead when compared to regression engines. This work explores a novel feature in HTM tuning: The cause for transactional aborts. By integrating this factor in their model, the system achieves greater accuracy in predictions.

Gray-Box Models

As already mentioned in section 2.1, Gray-Box Models incorporate mechanisms from both white- and black-box models, with the goal of maximizing synergy between them to create performance predictions and tuning strategies with high extrapolating power and good accuracy.

Didona et al. [30] propose a mixed approach to address scaling of Distributed Transactional Memory (DTM) systems, which delegates the measurement of certain components of the system to different predictors. Analytic models are a well suited tool to model accesses and contention on Transactional Memory, but become extremely complex when network topologies and behaviour are considered. The authors of TAS (Transactional Auto-Scaler) solve this by creating a machine-learning component that deals with the dynamics of the network layer. Not only does this approach simplify the analytical model, but enhance portability, as the system is no longer designed for a specific network architecture. This approach has been used in other approaches in literature [31]

Traditional machine learning algorithms try to compute a performance metric for an unknown configuration given the current performance level and resource usage. TAS instead feeds the machine learning engine with an accurate estimate of the unknown configuration's resource usage, obtained by querying a white-box analytical model, eliminating a major regression step that would be expensive in distributed applications. The mechanism used is a decision-tree based regressor that computes linear models at each node instead of an element of the discrete parameter domain, and is updated at runtime with collected samples to account for scale changes in the system. TAS achieves impressive accuracy and provides a viable tool for elastic scaling of data grids, as well as QoS and cost-driven analysis and scaling policy creation.

A more recent work by Didona et al. [6] builds on top of TAS, with the aim of improving transaction throughput in the same category of systems distributed Transactional Memory systems. The authors begin by examining cen-

tralized STM systems and producing a mechanism similar to a feedback control loop, which explores neighbouring configurations in search of a local maximum. Benchmarking was performed with the STAMP suite and various synthetic micro-benchmarks, and results show that no applications with multiple maxima were found.

The core of this work lies in the exploration component, or "decision module", which incorporates TAS's logic with a "patcher", a Decision Tree-based mechanism that, given enough samples (and TAS's prediction) at runtime, can estimate corrective factors that minimize the prediction errors when contention reaches high levels (where TAS's accuracy starts to decline).

Didona et al. propose a further means [32] of combining AM with ML approaches. When addressing this challenge, the authors compare three different approaches:

- *K Nearest Neighbours*
  KNN is used in this context to evaluate the accuracy of the analytical model and various machine learning components, and can be used to select the best one at runtime, depending on the system behaviour and on the target workload/configuration;
- *Hybrid Boosting*
  HyBoost follows the intuition that a corrective function to predict the residual error from the analytical model may be easier to learn (with an ML algorithm) than the original target function. The final prediction is then produced by combining the AM output with the corrective factor obtained from the learning algorithm;
- *Probing*
  Probing is an approach that consists of using machine learning components exclusively on the regions of the search space where the analytical model does not achieve sufficient accuracy. More in detail, a classifier is used in order to determine in which subsets of the parameters the analytical model or a black-box model should be used. A second black box regressor is then trained exclusively with samples corresponding to regions in which the analytical model is known to achieve poor accuracy.

The system is tested in a total-order broadcast platform (TOB), and in a distributed transactional in-memory store, Infinispan. In TOB, HyBoost achieves disappointing results, and is outperformed by a pure ML baseline approach, while both Probing and KNN show encouraging improvements over the first two. On Infinispan the scenario is inverted, with KNN and Probing showing some improvement and HyBoost being extremely efficient. The low performance shown by HyBoost in TOB is due to the non-linearity of the AM's error distribution. Conversely, in Infinispan the corrective function is highly linear, which translates to high levels of speedup when using HyBoost. Although promising, these results show that there is no silver-bullet approach in gray-box models.

The technique of feeding ML models with data produced with AM is known as "Bootstrapping". While multiple approaches in literature use this method, some aspects are left unexplored in existing implementations. Didona and Romano [32] propose a novel research in which they algorithmically formalize two aspects of bootstrapping methods:

- The sample size of the AM's output that should be used to feed the ML model.
- The algorithms that fit best when updating the (initially fully) synthetic training set.

To this end, the authors use a ten-fold cross-validation approach when generating the initial knowledge base from the analytical model. This allows to assess the validity of the chosen set, which is selected if the average accuracy falls over a given threshold ϵ, and discarded otherwise, after which a new training set is tested and the algorithm repeated. Updating the training set is the core of the bootstrapping methodology, as it allows for the incremental refinement of the initial performance model.

Following the objectives of this study, training time is substantially decreased, while an accurate convergence is obtained with fewer samples at runtime.

3 Goals

Objectives

The analysis of the state of the art conducted in the previous section highlights that in the literature there are no solutions that have attempted to dynamically tune the concurrency degree in STMs that support parallel nesting. The goal of my dissertation is precisely to fill this relevant gap in the literature, by implementing a self-tuning mechanism capable of simultaneously adjusting not only the number of concurrent top-level transactions, but also the number of nested transactions active within each top-level transaction. Based on the pros and cons of previous self-tuning solutions for TM systems, we highly favour online black-box models, given that, if well designed they can overcome most of the alternative's shortcomings. Additionally, this class of concurrency tuning implementations avoids impractical offline training phases, which can be costly and time consuming. Our devised solution will also strive to minimize, and if possible avoid, changes to application code. A distinction can be made in the different aspects of parallelism we want to tune with our mechanism, given of the problem's dimensionality. Besides regulating top-level transactions, inner parallelism also serves as a configurable parameter. Additionally, there is the possibility of assigning different concurrency levels to different top-level transactions, as the natural parallelism level may not be uniform.

Challenges

In designing a concurrency control scheme for multi-dimensional TM systems, we face a few challenging problems, the greatest of which is the dimension of the search space. There are, of course, many well-understood algorithms in this domain, but our design must keep the search algorithm's overhead to a minimum, given the performance critical nature of our target system. Furthermore, the balance between exploration and exploitation phases is harder to achieve, given that local fluctuations in any dimension can lead to improved or reduced performance. Thus, whatever the direction our solution follows, two main features must be guaranteed:

- An efficient sampling algorithm that can quickly converge on areas of interest of a large search space
- The system must be able to rapidly switch between exploration and exploitation when workload characteristics change.

Planning

We plan to follow a few key steps in the early stages of our design. The first logical step would be the integration of mechanisms to support dynamic concurrency adaptation into JVSTM. This mechanism will allow the tuning of the parallelism degree of top-level and nested transactions. We will implement this at the JVSTM library level, to keep the system's transparency.

Next, we will proceed with the development of a simple tuning mechanism for JVSTM, to serve as a baseline performance measure when testing further work. This implementation will also be a fast prototype, to guide other development phases. After this initial prototyping, we will be ready to take the first measurements for future reference, after which the design and development of the main algorithm(s) can start.

Initial steps in the main research phase will focus on finding an efficient sampling algorithm that can converge on the search space in an efficient way. The work produced in this step will base itself on key features of other algorithms (e.g. Latin Hypercube Sampling[33]) that prove to tackle the dimensionality problem adequately. Meanwhile, a combination of techniques will be refined and tested in the context of JVSTM, such as Gradient Descent and Simulated Annealing, to perceive which approach fits best in nested parallelism. While testing these components we plan to use various measures to infer which metric is most accurate in reflecting the workload's characteristics. The metrics used largely influence the concurrency control system's behaviour. Different measures can be used, namely transaction commit rate, wasted transaction time, and abort rates. Given our dimensionality problem, we must carefully study these alternatives to determine which one fits best. Finally the system will need an algorithm to control the jumps between exploration and exploitation phases. UCB [34], RRS [35], and other proposals use clever conditions in the exploitation phase to restart exploration. We plan to test some of these methods and refine a trigger that can rapidly react to workload changes and restart exploration.

Further down the development process we intend to study the effects of shutting down inner parallelism entirely. We predict some workload profiles will perform better with top-level transactions only (measuring and results are needed). Switching off inner transactions requires that our main algorithm be capable of restarting this system component without too much overhead, as to provide performance benefits. If work re-distribution and thread creation and destruction undermine the benefits of having only top-level transactions, this aspect will not be considered in further work.

In summary, we propose to develop a novel concurrency control mechanism for multi-dimensional Transactional Memory systems, where the dimension of the search space poses an important challenge , and brings about a plethora of other details which we plan to face using knowledge gathered from other works in the literature.

3.1 Evaluation

The evaluation of the solution we obtain will be performed experimentally, across a number of components and with different algorithms. As we need to explore several techniques and try different approaches for each development step, tests will encompass all of these different aspects of our work. In detail, to evaluate our proposal we intend to proceed as follows:

- **Evaluation metrics**: the most relevant measure in respect to implementation quality is the execution time, when compared to a baseline measurement. We admit the possibility of using other support metrics, such as number of conflicts, transaction throughput, etc., if their usage reveals to be adequate.

- **Test cases**: There are a number of benchmark suites for transactional memory which try to reflect real-world workloads. We intend to use the STAMP suite, a collection of test cases frequently used in academic research.

- **Comparison with other systems**: Besides comparing the base system (JVSTM) running with static configurations to our solution's performance, our incremental development plan will allow for a rapid initial prototype to be used as baseline measure for comparison.

3.2 Roadmap

- January 9 - February 15: Integration of mechanisms into JVSTM to support the dynamic adaptation of the degree of concurrency of both top-level transactions and nested transactions. The mechanisms will be integrated at the JVSTM library level, so to achieve full transparency at the application level. The system shall also support the online monitoring of key performance indicators like throughput and abort rate, which will be used to feed the on-line tuning system.

- February 15 - March 31 : Development and evaluation of the first online tuning-system, which will apply gradient descent techniques and operate at the granularity of the single program, i.e. all top-level transactions will be allowed to spawn the same maximum number of top-level transactions.
- April 1 - May 31: Development and evaluation of a more sophisticated online tuning mechanism, which will operate at a finer granularity by allowing different top-level transactions to spawn a different number of nested transactions.
- May 31 - July 15: Investigate the possibility of achieving an even finer granularity, by allowing a top level transaction that spawns nested transactions more than once during its execution to use different degrees of concurrency.
- July 15 - August 31: Preparation of a paper for submission to a scientific conference
- September 1 - October 15: Preparation of the MsC dissertation

3.3 Conclusions

In this document I conducted a study of the state of the art on Transactional Memory, focussing on the problem of dynamically adjusting one of the key tuning knobs of this emerging class of systems, i.e. the degree of parallelism that TM systems should adopt. My analysis of the state of the art has highlighted a relevant gap in the existing literature, i.e. the lack of self-tuning mechanisms capable of adjusting the degree of parallelism in TM that support parallel nesting.

During then next phase of my dissertation I aim at addressing precisely this issue, and in this document I have already identified some of the key design choices and challenges that I will have to address. Further, I have laid out a roadmap for my future research activities.

References

[1] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 1993, pp. 289–300.

[2] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. "Unlocking Concurrency". In: *Queue* (2006), pp. 24–33.

[3] Ricardo Filipe and João Barreto. "Nested Parallelism in Transactional Memory". In: *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Ed. by Rachid Guerraoui and Paolo Romano. Springer International Publishing, 2015, pp. 192–209.

[4] Diego Rughetti et al. "Tuning the Level of Concurrency in Software Transactional Memory: An Overview of Recent Analytical, Machine Learning and Mixed Approaches". In: *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Ed. by Rachid Guerraoui and Paolo Romano. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 395–417.

[5] Mohammad Ansari et al. “Robust Adaptation to Available Parallelism in Transactional Memory Applications”. In: *Transactions on High-Performance Embedded Architectures and Compilers III*. Ed. by Per Stenstrm. Vol. 6590. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 236–255.

[6] Diego Didona et al. “Identifying the Optimal Level of Parallelism in Transactional Memory Applications”. In: *Networked Systems*. Ed. by Vincent Gramoli and Rachid Guerraoui. Springer Berlin Heidelberg, 2013, pp. 233–247.

[7] Pierangelo Di Sanzo et al. “On the analytical modeling of concurrency control algorithms for Software Transactional Memories: The case of Commit-Time-Locking”. In: *Performance Evaluation* 69.5 (2012), pp. 187 –205.

[8] Mohammad Ansari. “Weighted Adaptive Concurrency Control for Software Transactional Memory”. In: *J. Supercomput.* 68.3 (June 2014), pp. 1027–1047.

[9] Nuno Diegues and João Cachopo. “Practical Parallel Nesting for Software Transactional Memory”. In: *Distributed Computing*. Ed. by Yehuda Afek. Vol. 8205. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 149–163.

[10] Joao Cachopo and António Rito-Silva. “Versioned boxes as the basis for memory transactions”. In: *Science of Computer Programming* 63.2 (2006). Special issue on synchronization and concurrency in object-oriented languages, pp. 172 –185.

[11] Sean Lie. “Hardware Support for Unbounded Transactional Memory”. Massachusetts Institute of Technology. MA thesis. 2004.

[12] Chi Cao Minh et al. “An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 2007.

[13] Richard M. Yoo et al. “Performance Evaluation of Intel&Reg; Transactional Synchronization Extensions for High-performance Computing”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: ACM, 2013, 19:1–19:11.

[14] Peter Damron et al. “Hybrid Transactional Memory”. In: *SIGPLAN Not.* 41.11 (Oct. 2006), pp. 336–346.

[15] Haris Volos et al. “NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems”. In: *ECOOP ’09: Proc. 23rd European Conference on Object-Oriented Programming*. Springer-Verlag Lecture Notes in Computer Science volume 5653. 2009.

[16] João Barreto et al. “Leveraging Parallel Nesting in Transactional Memory”. In: *SIGPLAN Not.* 45.5 (Jan. 2010), pp. 91–100.

[17] Woongki Baek et al. “Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory”. In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. Thira, Santorini, Greece: ACM, 2010, pp. 253–262.

[18] Bratin Saha et al. "McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime". In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '06. New York, New York, USA: ACM, 2006, pp. 187–197.

[19] Kaushik Ravichandran and Santosh Pande. "F2C2-STM: Flux-Based Feedback-Driven Concurrency Control for STMs". In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 927–938.

[20] Diego Didona et al. "Identifying the Optimal Level of Parallelism in Transactional Memory Applications". English. In: *Networked Systems*. Ed. by Vincent Gramoli and Rachid Guerraoui. Vol. 7853. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 233–247.

[21] D. Rughetti et al. "Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. 2012, pp. 278–285.

[22] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[23] Diego Didona and Paolo Romano. "On Bootstrapping Machine Learning Performance Predictors via Analytical Models". In: *CoRR* abs/1410.5102 (2014).

[24] D. Rughetti et al. "Analytical/ML Mixed Approach for Concurrency Regulation in Software Transactional Memory". In: *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. 2014, pp. 81–91.

[25] Armin Heindl and Gilles Pokam. "An Analytic Framework for Performance Modeling of Software Transactional Memory". In: *Comput. Netw.* 53.8 (June 2009), pp. 1202–1214.

[26] Chi Cao Minh et al. "STAMP: Stanford Transactional Applications for Multi-Processing". In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. 2008, pp. 35–46.

[27] N. Sonmez et al. "Taking the heat off transactions: Dynamic selection of pessimistic concurrency control". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009, pp. 1–10.

[28] Pascal Felber et al. "Time-Based Software Transactional Memory". In: *IEEE Transactions on Parallel and Distributed Systems* 21.12 (2010), pp. 1793–1807.

[29] Diego Rughetti et al. "Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory". English. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Ins Dutra, and Vtor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 475–486.

[30] Diego Didona et al. “Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids”. In: *Proceedings of the 9th International Conference on Autonomic Computing*. ICAC ’12. San Jose, California, USA: ACM, 2012, pp. 125–134.

[31] Diego Didona and Paolo Romano. “Performance Modelling of Partially Replicated In-Memory Transactional Stores”. In: *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2014.

[32] D. Didona et al. “Combining Analytical Modeling and Machine-Learning to Enhance Robustness of Performance Prediction Models”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2015.

[33] Bowei Xi et al. “A Smart Hill-climbing Algorithm for Application Server Configuration”. In: *Proceedings of the 13th International Conference on World Wide Web*. WWW ’04. New York, NY, USA: ACM, 2004, pp. 287–296.

[34] Peter Auer, Nicol Cesa-Bianchi, and Paul Fischer. “Finite-time Analysis of the Multiarmed Bandit Problem”. English. In: *Machine Learning* 47.2-3 (2002), pp. 235–256.

[35] Tao Ye and Shivkumar Kalyanaraman. *A Recursive Random Search Algorithm for Black-box Optimization*.