TÉCNICO LISBOA

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

José Miguel Gonçalves Simões

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

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

Supervisor(s): Prof. João Barreto
Prof. Paolo Romano

Examination Committee

Chairperson: Prof.
Supervisor: Prof. João Barreto
Co-Supervisor: Prof. Paolo Romano
Members of the Committee: Dr.
Prof.

May 2016

But still try, for who knows what is possible.

Michael Faraday

Acknowledgments

I would like to thank my supervisors, Professors João Barreto and Paolo Romano, Diego Didona and Nuno Diegues for the invaluable support they provided.

I would also like to thank Duarte, for helping in having a proper work environment, and my family for their unconditional support.

I would also like to thank Luana, Diogo, Nereida, Janaína, Isa, Sofia, Liliana and Inês for the constant support throughout the duration of this work.

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 to create a tuning mechanism 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

Resumo

Memória Transacional é um paradigma proeminete da área de computação paralela, cada vez mais utilizado, que permite a paralelização de programas sequenciais de uma forma simples, e expõem grandes níveis de paralelismo. Existem muitas soluções diferentes no contexto da Memória Transaccional. Uma das categorias inclui os sistemas de Memória Transaccional com aninhamento paralelo, que permitem às transacções criar transacções filhas, efectivamente expondo paralelismo adicional de granularidade fina. Os sistemas de Memória Transaccional podem incluir mecanismos de ajuste, que regulam os parâmetros e dinâmica interna em resposta a medidas de desempenho do sistema, para melhorar o seu desempenho. No limite do nosso conhecimento, não foram explorados na pesquisa contemporânea mecanismos de ajuste para Memória Transaccional aninhada. Propomos nesta dissertação que é possível criar estes mecanismos para este tipo de sistema, e analisamos este problema utilizando um sistema de Memória Transaccional de ponta, a JVSTM. Desenvolvemos mecanismos para integrar rapidamente diferentes estratégias de ajuste na JVSTM, e estudamos os efeitos e melhorias, ou ausência das mesmas, de ajustar um sistema de Memória Transaccional aninhado. Os resultados obtidos oferecem perspectivas das diferentes estratégias de ajuste, bem como uma base sólida para realizar trabalho futuro.

Palavras Chave

Memória Transaccional, Memória Transaccional em Software, Ajuste, Aninhamento, Optimização, Ajustar Memória Transaccional Aninhada em Software

Contents

List of Figures

Abbreviations

TM Transactional Memory

STM Software Transactional Memory

HTM Hardware Transactional Memory

CD Conflict Detection

CM Contention Management

ML Machine Learning

GD Gradient Descent

LGD Linear Gradient Descent

FGD Full Gradient Descent

HS Hierarchical Scan

RRS Recursive Random Search

RS Random Sampling

1

Introduction

Contents

1.1 Context

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, and show a promise to dominate most platforms in the near future. 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]. The difficulties of parallel programming can even lead to parallel programs performing worse than their sequential version.

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 [14].

In the scope of our work, a distinction can be made in the parallel development methodologies: traditional lock-based synchronization and Transactional Memory [8]. 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 large 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 at the smallest granularity level. This approach avoids the inefficiency of the first one, but it is subject to pitfalls like deadlocks, live-locks, and data races. Furthermore, it adds complexity to the code and hinders its maintainability. In light of these considerations, none of these two approaches appears to be a suitable reference parallel programming paradigm, as they sacrifice either 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.

One such paradigm is Transactional Memory [8, 9], a promising model that abstracts parallel operations and synchronization and tries to provide performance gains and transparency when developing parallel software.

1.2 Motivation

1.2.1 Software Transactional Memory

Listing 1.1: STM Usage Example

```
1 public class TransactionalTask implements Thread{
2  public void start() {
3   Begin();
4   //do stuff
5   //this is a critical section
6   //do stuff
7   Commit();
8  }
9
10  public static void main(..){
11   //do stuff
12   ExecutorService executor = new Executor();
13   Thread thread1 = new TransactionalTask();
14   Thread thread2 = new TransactionalTask();
15   Executor.submit(thread1);
16   Executor.submit(Thread2);
17   //do stuff
18  }
19 }
```

Listing 1.1: An STM usage example. The `TransactionalTask` class executes a block of code transactionally, by explicitly beginning and committing a transaction, which runs in its own thread. The main section of the code instantiates two of these tasks and submits them to an executor, using a standard Java dialect.

One of the most influential models of fork-join parallel programming is Transactional Memory (TM)[8] . TM is a concurrency mechanism that takes advantage of the concept of transactions, widely used in database systems [9], 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 transactional control mechanisms such as begin, commit and abort to control the flow of these blocks (Listing 1.1). 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 [10]. 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 [11]. 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 [11] 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. 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. Our work applies only to STM, which we will cover in more detail in subsequent chapters.

The STM model involves the programmer in the parallelization process: Reasoning on the program semantics and adaptation to the transaction paradigm are required, albeit kept to a minimum, as STM systems strive to reduce development complexity [8]. STM systems achieve considerable speed-ups, as some data dependencies are resolved through the use of transactions. However, the most common approach to using STM systems is to adopt a coarse structure of transaction blocks, to avoid semantic complexity [12, 13]. Consequently, most parallel applications do not expose their full parallelism potential, as finer details in the parallel logic are left unexplored.

1.2.2 Nesting

Listing 1.2: STM Nesting Usage Example

```
1 public class NestedTask{
2  public static void main(..){
3   Transaction.begin();
4   try{
5     //transactional accesses
6    Transaction.begin();
7     //nested transactional accesses
8    Transaction.commit();
9     //transactional accesses
10   Transaction.commit();
11  }catch(CommitException ce){
12   Transaction.abort();
13  }
14 }
15 }
```

Listing 1.2: A nesting usage example. The main method starts a transaction which performs arbitrary transactional accesses. In the context of that transaction, it starts a nested transaction, which executes its own accesses and has its own (inner) begin and commit events.

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 (for example when only a small portion of the total application code can be parallelized using transactions), nesting allows transactions to spawn additional (inner) transactions

[1], enabling intra-transaction parallelism, and thus an improved overall level of concurrency and complexity.

Transactional Memory Nesting introduces its own set of problems, as TM systems now need to manage relationships between parent, child and sibling transactions. These must be taken into account when designing TM systems. There are different types of nesting models, and different approaches to solving these problems in each one. We will explore these in further sections.

1.2.3 Tuning

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 transactional 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 code blocks (e.g. libraries) without incurring major code changes, performance penalties or pitfalls like dead- and live-locks. This is achieved by means of nesting transactions. Nesting allows different TM code blocks to be easily and safely combined into a bigger single atomic block, where as locking approaches are usually difficult to interleave.

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 application's workload is crucial to obtain good performance [14–17]. Past and current research efforts focus on tuning a number of different aspects of a transactional systems. Some propose to tune the contention management policy, the mechanism that dictates the system behaviour in case of a conflict between transactions [18], while others focus on tuning the number of transactional threads available to enhance throughput [14]. Other parameters and mechanisms can be adjusted in this manner, but they are out of the scope of our study.

[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.

1.2.4 Tuning and Nesting

One of the most investigated problems in STM tuning is how to adjust, at runtime, the number of active transactional threads in order to maximize performance. However, several researchers [19–22] 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. 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.

We can define the search space of our problem as the domain in which the different parameters exist. There can be various interpretations for this search space, such as the cardinality of top-level transactional threads versus nested-threads. The key challenge in tuning the intra-transaction parallelism is the dimensionality of this search space. In fact, as opposed to traditional solutions that only have to tune the number of concurrently active transactional threads [23–26], tuning the intra-transaction parallelism also requires adjusting the number of nested transactions to be spawned. Inner parallelism adds an extra dimension to the parameter space, but if we assume that different top-level transactions may require different degrees of inner parallelism [27], additional complexity must be added to the concurrency control mechanism.

There are several nesting models and different types of tuning techniques. We overview each of these in subsequent chapters.

1.2.5 Outline

The main contribution of this dissertation is a tuning mechanism for JVSTM, a state-of-the-art Software Transactional Memory system. We develop such a tuning mechanism with minimal modifications of JVSTM's original code base, and provide different approaches to this problem.

Throughout this dissertation we expose and analyse the main challenges, the problem domain, and our proposed solution.

We test our work with two well-known benchmarks used in Transactional Memory research, STAMP Vacation and STMBench7. With the results obtained, we show that high contention configurations are an ideal case for tuning a transactional system, but mixed and low contention configurations are exceptionally hard to tune, with results worse than the baseline JVSTM, while high contention config-

urations show satisfactory performance gains.

2

Related Work

Contents

Parallel computing research has defined two prevalent paradigms for designing parallel applications for multi-core platforms: Task Parallelism and Data Parallelism.

Data Parallelism consists of splitting disjoint data sets over the available processors, such that each processor computes a given subset of the problem data, the outputs of which are subsequently joined. Data parallelism is not pervasive, as it requires very specific workloads that can be split into disjoint parts. This method applies only to a small set of problems, and is not viable to the generality of data structures and program architectures [4].

Task parallelism also envisions the division of work as parallel tasks running on different cores, but these tasks access shared data. This added liberty requires that data access be synchronized, to coordinate accesses between different threads and avoid data corruption. The traditional approach to this problem is the use of lock-based synchronization policies, where the programmer uses coarse- or fine-grained lock structures to serialize concurrent access to a given memory location. Fine-grained lock structures consistently yield better results and scalability [9], but are much more difficult to develop than coarse-grained ones, which confer better readability to the code and require less semantic reasoning on the program's logic. Transactional Memory is a model that follows Task Parallelism and tries to provide results on par with fine-grained locking, with the readability and simplicity of coarse-grained locking [9].

The contributions of this dissertation apply to the task parallelism paradigm. For the remainder of this work, we focus only on that paradigm, specifically on Software Transactional Memory. The following subsections overview the various types of Transactional Memory and some of its concrete details, nesting models and their properties, and the various types of tuning applied to TM, and how they influence the solution we propose.

2.1 Software Transactional Memory

Software Transactional Memory is a category of systems that provide Transactional Memory capabilities through software runtime support only.

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[8]. These aspects will be covered in the next subsection.

The main drawback of STM systems is the necessity of transaction boundaries and operations to

be instrumented, in order to keep track of their accesses to memory[9]. 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 and take additional time to execute.

2.1.1 Factors Affecting STM Performance

As discussed in the previous section, STM systems need to maintain state during a program's execution via additional structures embedded in the program code at compile time. These structures are at the heart of the runtime system, and perform a variety of functions, described followingly.

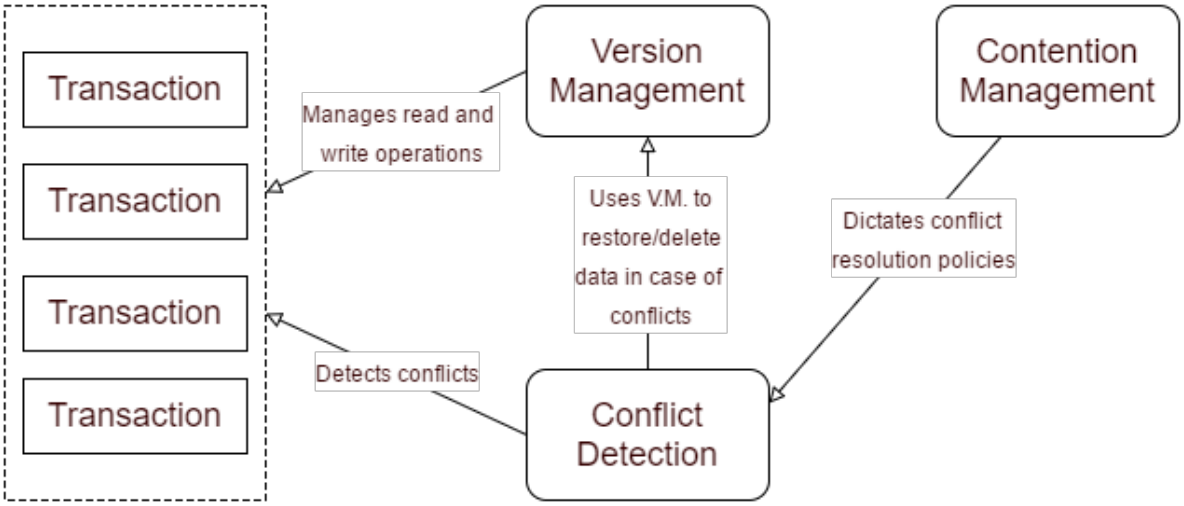

Figure 2.1: A conceptual diagram that illustrates the interplay between the different STM mechanisms. Version Management is used for reading and writing different versions of data, as well as restoring or deleting them when a transaction aborts. Conflicts are detected by the Conflict Detection mechanism. Contention Management dictates the action taken in case of a conflict between any two transactions.

2.1.1.A Conflict Detection

Conflict Detection (CD) (sometimes called *concurrency control* in the literature) is the means by which a transactional runtime system detects incompatibilities between two transactions. Many CD policies exist, most of which make heavy use of the versioning system, and can have different granularities and act at different moments during a transaction's lifetime.

Regarding granularity, two main classes can be defined: Object-based and word-based. Object-based CD consists of detecting conflicts in data structures, is mostly applied in object oriented languages, and integrates the notion of objects in its logic [28]. Word-based CD detects conflicts conflicts at the level of the processor architecture's word size [29]. These can be seen as coarse-grained and fine-grained, respectively.

Fine-grained approaches traditionally incur greater levels of overhead, mainly because they have to manage more data in the transaction logs, but at the same time may allow greater transaction throughput, as less conflicts are caused by accessing high-level data structures concurrently. Specifically, transactions may access objects concurrently but modify different elements, which does not cause a conflict. Conflicts caused by this type of access pattern are called *false conflicts*.

Regarding the time at which a conflict is detected during a transaction's lifetime, we can make two distinctions: Eager and lazy conflict detection. Eager CD checks for conflicts as soon as a transaction declares its intent to access data [29]. Lazy conflict detection occurs at commit time, when a transaction's logs are checked to assess if any conflicting data accesses happened [30]. As with other aspects of STM, choosing between these two methods also carries a trade-off: Eager CD causes overhead on every memory access, but prevents wasted work by detecting conflicts as soon as they happen. Lazy CD enhances concurrency, but induces wasted work in case of conflicts being frequent - as the logs are checked only at commit time, aborting transactions will discard most of the work they performed.

2.1.1.B Version management

In STM, version management is the mechanism used by the runtime system to keep different versions of data in case a transaction needs to abort, and the memory locations written by it need to be restored. There two main types of version management: Direct update and deferred update.

Direct update maintains an undo-log for each modified memory location, and transactions write their tentative values directly to memory. When a transaction aborts and its effects need to be rolled back, the undo-log is used to restore the data overwritten by the transaction [31].

Deferred Update systems maintain a redo-log (sometimes called write-set in the literature) for each transaction. Transactions write their tentative values to this redo-log, which is copied to the real memory locations when a transaction commits. Redo-logs become, for the duration of a transaction, a part of its memory snapshot, e.g. when a transaction tries to read a location it has previously written to, it consults its redo-log instead of the real memory location [32].

Direct update systems incur heavy overheads when transactions need to abort, as all written values need to be checked, and previous data restored [33]. On the other hand, deferred update version management have trivial abort mechanisms, but cause cumulative overhead when committing transactions, as their write-log must be checked and copied to the real memory locations.

2.1.1.C Contention Management

Contention Management (CM) is another factor that influences 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 [15], but have low performance on certain workload 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. Several recognized contention management policies exist [18], of which we can examine a few select examples:

- Passive - The simplest policy where the attacker transaction aborts itself and re-executes [29].
- Polite - The attacker transaction delays its progress for a fixed number of exponentially growing intervals before aborting the victim transaction. After each interval, the attacker checks if the victim has finished executing, if so the attacker proceeds without ever aborting the victim [34].
- Timestamp - The contention manager aborts any transaction that started executing after the victim transaction[18].
- Greedy - A timestamp is associated with a transaction when it first attempts to execute. A transaction aborts a conflicting transaction if the former has a younger timestamp than the latter, or if the latter is itself already waiting for another transaction. Unlike the previous approaches, this policy allows every transaction to commit within a bounded time, i.e. avoids starvation of transactions [32].

No policy performs universally best in all settings when designing a transactional system (Harris, Larus, & Rajwar 2010). The type of concurrency and workload profiles used should be considered when choosing a contention management policy. Some STM systems allow contention management to be parametrised ad hoc by the applications using them.

2.1.2 Nesting

Various STM nesting 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 [35]. 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, a 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. Finally, 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 [36].

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 [20]. 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 must now be taken into account when designing TM systems. There are multiple implementations of Nested TM.

PNSTM [19] 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 [22] is based on McRT-STM [37], 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.

JVSTM [20, 38] is a versioned Transactional Memory implementation which has been augmented by Diegues and Cachopo 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 trans-

action aborts and a roll-back 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.

2.1.3 Read-only Transactions and Opacity

Some STM systems favour read-only transactions in a way that allows them to never abort. These systems are said to support *invisible reads*, where reading transactions are not visible to concurrent writing transactions [29]. In order to preserve opacity, these systems have to implement additional mechanisms, in which reading transactions are responsible for detecting conflicts in their read operations. Two examples of these mechanisms are global clocks and multi-version:

- Global clock - The runtime transactional system maintains a global counter that is incremented every time a non-read-only transaction commits. Each transaction reads this value when it starts and uses it to define its position in the serial order of transaction history. This value represents the instant in which the transaction's memory snapshot is valid, and thus the transaction aborts if it reads an object whose version number is lower than its own version number.
- Multi-version - Some STM systems store multiple versions of each object [38], each one with its own timestamp. Additionally, each transaction maintains a timestamp window, during which its memory snapshot is guaranteed to be valid. As there are multiple versions of each object, it is likely that when a transaction tries to read one, there is a valid version available. Thus, if sufficient versions are available, it is guaranteed that read-only transactions will always commit. This aspect creates a trade-off between the amount of memory used by the versioning system and the throughput of read-only transactions. Usually some type of garbage collection must be performed, to free unneeded versions of data.

2.1.4 Programming Model

The majority of STM systems provide simple interfaces for programmers to use when parallelizing applications. Two of the most common constructs are annotations and begin/commit/abort instruc-

tions, used to wrap regions of code, and thus creating *atomic blocks*. Compilers and the runtime system further prepare these atomic blocks to be run transactionally.

The transactional runtime system guarantees the serial ordering of these atomic blocks, and further provides atomicity, consistency and isolation. The combination of these measures frees the programmer from the need to create low level threads and synchronization mechanisms, as atomic blocks synchronize implicitly with other atomic blocks that access the same data.

Additionally, these properties grant composability to atomic blocks. This means that one can combine a set of atomic blocks and the result will still be atomic. In contrast, lock-based approaches often require that either encapsulation be broken or internal concurrency control be exposed when composing different operations.

2.1.5 JVSTM

Several implementations of STM for the Java platform exist [39, 40]. JVSTM[20, 38] is a state of the art parallel nesting STM library that incorporates a sturdy algorithm to support efficient parallel nesting.

JVSTM This section covers the aspects of JVSTM most relevant to our work.

2.1.5.A API

JVSTM involves programmers in the parallelization effort by requiring them to explicitly use the library methods. The system has, however, a very simple API, where most applications need to use only two classes: jvstm.Transaction and jvstm.VBox. Listings 2.1, 2.2 and 2.3 show different use cases of the library provided by JVSTM. Listing 2.1 illustrates a simple use of top-level transactions, listing 2.2 provides a simple linear nesting example, and listing 2.3 reflects a parallel nested use case.

Listing 2.1: Use of top-level transactions example

```
1 public class Myclass{
2  VBox<Integer> i = new VBox<Integer>(); //transactional data
3
4  public static void main(..){
5   Transaction.begin();
6   try{
7    //transactional accesses to i
8    Transaction.commit();
9   }catch(CommitException ce){
10   Transaction.abort();
11  }
12 }
13 }
```

Listing 2.1: A simple JVSTM use case. A versioned box containing an integer ir created and accessed transactionally in the `main()` method, by explicitly controlling the start and finish of the transaction via the `Transaction` class' static methods.

JVSTM uses a multi-version versioning system to allow read-only transactions to always commit, never conflicting with other concurrent transactions, favouring read-dominated applications. The VBox (versioned box) class implements this concept, where each VBox instance represents a transactional object, and stores the versions of that object that have been committed over time by transactions. VBoxes have two main methods, *get* and *put*, which retrieve and update the VBox's value for the current transaction, respectively.

The Transaction class allows programmers to control the begin, commit, and abort operations of transactions explicitly. The *begin* method starts a new transaction, and sets it as the current transaction for the thread which calls it. *Commit* tries to commit the current transaction, and *abort* aborts it.

2.1.5.B Nesting transactions

Listing 2.2: Creation of a linear nested transaction example

```
1 public class Myclass{
2  public static void main(..){
3   Transaction.begin();
4   try{
5     //transactional accesses
6    Transaction.begin();
7     //transactional accesses
8    Transaction.commit();
9     //transactional accesses
10   Transaction.commit();
11  }catch(CommitException ce){
12  Transaction.abort();
13 }
14 }
15 }
```

Listing 2.2: A linear nesting usage example. An inner transaction is created (second `begin()` method call, line 6) to execute a portion of the work sequentially but in a logically distinct transaction.

JVSTM supports two types of nesting: Linear and Parallel. When using JVSTM in linear nesting mode, transactions are allowed to spawn any number of nested transactions, but at most one of them executes at once, i.e. sibling transactions execute sequentially in the parent transaction's thread. This can effectively be seen as a serialization of the inner transactions. As child transactions execute in the parent's thread, the parent transaction cannot perform any work while there are child transactions running.

Parallel nested transactions are represented by the jvstm.ParallelTask class. Each instance of this class will run the code to be executed, which is passed to it by the programmer. Additionally, this class implements the logic needed to run a nested transaction in a separate thread, freeing the parent transaction to continue its work while its children run in parallel. However, the parent transaction cannot commit until all of its children have committed too, and thus waits for them to commit even if it has no further operations to run. Listing 2.3 provides a usage example of these constructs.

Listing 2.3: STM Usage Example

```
1 public class Myclass{
2  public static void main(..){
3
4   Transaction.begin();
5   try{
6    //transactional accesses
7    List<ParallelTask<Integer>> tasks = new ArrayList<...>();
8    tasks.add(new ParallelTask<Integer>(){
9     @Override
10    public Integer execute() throws Throwable {
11     //transactional accesses
12    }
13   });
14   Transaction.manageNestedParallelTxs(tasks);
15   //transactional accesses
16   Transaction.commit();
17  }catch(CommitException ce){
18   Transaction.abort();
19  }
20 }
21 }
```

Listing 2.3: A parallel nesting usage example. An inner transaction (instance of `ParallelTask`) is created to execute a portion of the work in parallel. The method `Transaction.manageParallelNestedTxs` manages the execution of a group of transactions in separate threads.

2.1.5.C Versioning and Conflict Detection

Besides using multi-version, JVSTM also implements a type of global clock. Each top-level transaction keeps a version number (*nClock*) that it fetches from a global counter when it starts. This version number represents the data version of the modifications performed by the latest read-write transaction that successfully committed. Child transactions inherit this number from their parent, and

compute a map of counters of their own, named *ancVer*. This map is computed when the child transaction starts, by inheriting the parent's *ancVer* and adding to it the parent's *nClock*. This map thus associates an *nClock* value to each ancestor, and represents the versions of data that a transaction can read from its ancestors tentative values.

A VBox instance contains two lists of written values, one with the permanent values written by committed transactions, and another with tentative values written by running transactions. Transactions check their *ancVer* and *nClock* numbers against these values' versions when accessing them to ensure they are reading valid versions. Writing a value to a VBox inside a transaction is then accomplished by adding this value to the VBox's tentative values list, and acquiring ownership of that list (i.e. locking it for the current transaction). Transactions can only write values to a tentative value list if it is locked by itself or one of its ancestors. A conflict is caused if a transaction tries to write to a list that has been locked by a transaction outside of its hierarchy.

JVSTM uses optimistic concurrency control for top-level transactions and pessimistic concurrency control for nested transactions. Specifically, top-level transactions can continue executing even in case of conflicts, but nested transactions abort immediately when they perform conflicting write operations. JVSTM falls back to serialization in this case, and re-executes failed nested transactions sequentially in the context of the top-level transaction after all its descendant transactions have finished.

When a VBox is read inside a top-level transaction, a tentative value is return if the transaction or any of its descendants have written to it - the value written by the last committed child. When a parallel nested transaction reads a Vbox, a tentative value can come from a write access from itself, child transactions, or ancestor transactions. In the latter, the current transaction has to check its *nClock* and *ancVer* against the ancestor's version, lest it access an invalid version.

When there is no available value from the tentative list, a value is returned from the top-level transaction's local write-set. In case this write-set does not contain a value either, it is is fetched from the permanent list. The value returned may not have the latest version, but an version number that is equal or lower to the reading transaction's version number.

At commit time, VBoxes read by a transaction must have a version number equal or lower than the committing transaction. If this is not the case, another transaction has written values to that VBox while the committing transaction was running, and its read operations are no longer valid. Consequently, read-only transactions always commit, because they do not change the program state.

2.1.6 Conclusion

STM systems provide the abstractions needed to hide complex concurrency schemes from the programmer, allowing simple usage and additional benefits transparently, such as composability.

Compared to other approaches to easily parallelize sequential programs (such as thread-level speculation [41], message-passing parallel frameworks, etc.), STM systems generally expose more parallelism, remove the need of complex data dependencies and control flows, and are generally able to attain better performance.

Transactions are not a silver-bullet solution to parallelization, however. Programmers can still use transaction incorrectly, or use inappropriate levels of granularity: the problem of creating coarse-grained parallel regions to avoid complexity can still affect development, if a programmer chooses to expose the full parallelism inherent to an application. Additionally, Transactional Memory may simply be inadequate for certain workload profiles or parallel architectures.

Despite these considerations, STM provides a solid model to enhance parallelism, reduce the complexity of developing parallel systems, and expose greater levels of concurrency and performance.

2.2 Concurrency Degree Tuning

After focusing on the aspects that influence an STM system's performance, and broadly inspecting the logic behind JVSTM, different tuning techniques can be analysed with a better notion of the characteristics of the platform we use to solve the problem of tuning parallel-nested STMs. This section details these techniques, and how they might apply to our tuning mechanism for JVSTM.

2.2.1 Tuning Categories

Tuning techniques can be broadly divided in three main categories: White-box, black-box and gray-box. In this section, we cover these categories and offer insight on their main concepts and methodology. As our work focuses on black-box tuning, we inspect this category in greater detail.

2.2.1.A White Box

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 [42, 43]. 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 model is 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 (i.e., they yield static configurations). White Box models provide strong foundations for analysis and comparison of different Transactional Memory systems, in terms of their dynamics and behaviour.

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

2.2.1.B Black Box

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 [14, 44]. 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.

A – Offline Modelling 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 techniques that are commonly employed to develop offline black box models are Artificial Neural Networks (ANN), and Decision Trees (DT) [45].

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, among other factors. These inadequacies can be seen as the product of the training phase not being sufficiently accurate[14], as training times increase with the number of considered parameters and sample size and suitable, representative data may not be available. There is also the hindrance of training the mechanism in early deployment stages, which in certain applications may be unfeasable to perform.

Machine Learning (ML) solutions for 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 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. A number of different solutions exist in the literature [14, 23], which take varied approaches to tuning, from neural network algorithms that consume write- and read-set size, to selecting at runtime which features are desirable for analysis by the ML component.

B – Online Modelling 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 [25]. 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 Stochastic Gradient Descent[15], Simulated Annealing, Reinforcement 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.

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

In [26], a control feedback loop is exploited to quickly react to workload changes. F2C2 is built on top of TinySTM [46], 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 parallelism degree 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 detection component uses transaction throughput, defined as *tpi*, transactions per interval (or 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[1] 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.

2.2.1.C Gray Box

Gray Box Models are a type of solution that combines white-box and black-box approaches [44]. 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 current model of the target system, i.e. they require 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 system's dynamics.

Offline Machine Learning approaches usually require 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* [14] 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* [23] 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

[1] The authors avoid global synchronization in measurements to reduce overhead

unusable, particularly when the system's dynamics are too complex to model using White Box approaches.

- *Bootstrapping* [25] 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.

2.2.2 Conclusion

The analysis of the state of the art conducted in the previous sections highlights that there are no studies in the literature that have attempted to dynamically tune the concurrency degree in STMs that support parallel nesting. The goal of this 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 on-line 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. The solution we devise will also strive to minimize, and if possible completely avoid, changes to application code.

Having the contents of this chapter in mind, we can begin to explore 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.

3

Tuning the Concurrency Degree in Nested Software Transactional Memory

Contents

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. Additionally, we propose to implement a pluggable component within the JVSTM tuning, to enable experimenting with and testing different tuning approaches.

This section covers the various aspects of the development of JVSTM tuning.

Section 3.1 provides an overview of the challenges our solution faces, attempts to contextualize them in the JVSTM architecture and exposes the architecture we devised in this context, and how it integrates with JVSTM explicitly.

Section 3.2 covers the different tuning policies, i.e. tuning strategies, we used to explore the theme of tuning JVSTM.

3.1 Architecture

A system that tunes the concurrency degree of a nested STM system faces a number of challenges. These challenges can be specific to the STM implementation, or may apply to other TM systems in general.

This Section concisely defines the problems we face on our proposed solution, and elaborates over each one, clarifying its implications and presenting our decisions on how to overcome them.

3.1.1 Search Space

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* [1]. In the simplest case, we want to adjust the number of threads used for top level transactions and the threads used for nested transactions. An optimization problem has a

[1] 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.

search space, composed of the domains of its various dimensions. Thus, we define the dimensionality 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.

Another relevant interpretation of the dimensionality of our problem relies on defining one dimension for each level of the nesting tree. A transaction's level in the tree is its nesting depth. Thus, we would tune one top-level dimension and an unbound number of nested dimensions. A tuning system based on this interpretation would provide insights on how the nesting depth influences system performance. This is far beyond our initial objectives however, and is left for future work, after an exhaustive study on the simpler cases is complete. Thus, we choose to work with the two dimensions we described in the previous paragraph. Consequently all nested transactions fall in the nested dimension, and their nesting level is not considered when tuning the concurrency degree.

Regarding our approximation to an optimization problem, the goal is to optimize the output of the target function, whatever output measure we choose to represent system performance. As the input parameters are well-defined, further analysis can be performed. It is desirable for an STM system to take advantage of the number of cores available in a machine. In the absence of tuning mechanisms, thread count often fits the number of cores, as the literature suggests over-subscribing processor cores leads to decreasing performance [36]. STMs often use (static) configurations that match the number of threads they use with the processor count.

This influences the search space available to a tuning system. The number of nested or top-level threads cannot be less than one, naturally, but our tuning system should consequently avoid over-subscribing the processor cores. Thus, we can define limits to the number of threads that can be active at any given time. In JVSTM, each thread has access to a global thread pool, to which it submits nested transactions. This thread pool is global. The total number of transactional threads is defined as the product of the number of top-level threads and the number of nested threads. Whatever the configuration our tuning system applies to the system as a whole, the total number of threads cannot be greater than the available processor cores. Regarding our previous analogy, this can be seen as a constraint of the optimization function.

3.1.2 Tuning and Thread Management

Certain workloads and application profiles may achieve grater levels of performance with different numbers of transactional threads. Specifically, excessive contention in applications that perform many operations on the same sets of shared data can lead to high numbers of conflicts, and reduce performance. In such cases it is desirable to reduce the number of concurrent transactional threads. On

the other hand, if low levels of contention are detected, and unused processor cores are available, an STM system should try to increase the number of cores it uses in order to increase its performance. Considering the JVSTM architecture, this leads to the necessity of our tuning system to adjust the number of top-level or nested threads at runtime.

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. This characteristic leads to our first important design decision: To adjust the concurrency degree of JVSTM, we control the number of active threads by suspending/resuming the existing ones, not via thread creation and destruction.

This fact may lead to a pertinent hypothesis: The possibility of modifying JVSTM such that active transactions can be managed by regulating thread creation and destruction. Concretely, we would regulate the number of transactional threads by creating or destroying threads, instead of suspending and resuming them. We do not explore this aspect in our work. We ignore the motives behind this particular aspect of JVSTM's current architecture, and modifying it to such an extent is out of the scope of this dissertation. We mention this hypothesis because it could provide valuable insight when comparing the two approaches.

Affecting a transaction's execution without influencing its transactional thread means we must change the behaviour of a transaction when it begins. Specifically, we modified the code for each transaction class in JSVTM to signal our tuning component before starting. Additionally, if we are to keep an accurate measure of the active transaction count, we must also know when a transaction aborts or commits. We followed the same method: transactions signal the tuning system when they finish their work. For the purpose of managing threads, it is not relevant whether they aborted or committed. However, this becomes important when we desire to measure the system's performance.

JVSTM tuning must be aware of the system's performance when making decisions that will affect its concurrency degree. Several measurement types are used throughout the architecture. We choose committed transaction throughput (hereinafter referred to as *throughput*) as our unit of performance measurement. Throughput is an absolute measure that reflects the number of committed transactions, and by itself does not represent any time interval. In our solution, any throughput measurements shall refer to a certain time interval, after which a tuning decision is made, to allow for thread statistics to be collected (throughput measurements). We refer to this time interval simply as *interval* hereinafter.

For the tuning system's measurements we choose throughput because it is the simplest measurement that can accurately represent our main goal: to improve system performance. Other types of

measurements exist, such as the ratio between transaction starts and commits, but these are more convoluted and may be more suited to measure other properties (such as conflict rate, for Conflict Detection mechanisms).

The statistics collected by the tuning component are global. Each thread registers its begin, abort, and commit events, and an aggregation process occurs at the beginning of each interval, in which JVSTM tuning sums each thread's statistics into global values. To avoid synchronizing this access between the tuning component and each thread, which could affect a transaction's critical path and possibly slow its execution, all statistics are represented by volatile variables[2] with no synchronization, encapsulated in an instance of the `jvstm.tuning.ThreadStatistics` class. Each transactional thread maintains its own instance of this class - this is achieved via Java's `ThreadLocal` class, that guarantees each thread to have a single, isolated copy of a certain object. As each thread takes care of its own statistics, it needs not synchronize accesses to these members against other transactional threads. When the tuning mechanism aggregates data, it accesses the statistics without synchronization. This can lead to imprecise measurements, as data races may happen. This problem is not critical, however, given the sheer number of transactional events (begin, commit, or abort) that happen during an interval. As the decision-making mechanism runs complex algorithms, and their effects should last for a significant period, the intervals tend to be in the order of hundreds of milliseconds. We have not tested different time intervals extensively. Using different intervals could in principle affect the performance of the system, and our tuning system could even adapt its tuning interval in response to system behaviour. This analysis is left for future work.

When a transaction signals the tuning system, it declares its intent to start. A check is performed by the thread against the tuning mechanism's records, to determine if it should start immediately or wait before starting. The outcome of this test is determined by how many available slots exist for threads to start transactions. This property is itself determined by the decision taken by the tuning mechanism at each interval, and remains constant in each respective interval.

The tuning mechanism must keep a public record (i.e. accessible to the transactional threads) of how many transactions are allowed to run concurrently. The threads check these records when they start, and are served by the order in which they request permission to run. These records take the form of a semaphore in the current implementation. A semaphore provides a clean mechanism to force threads to wait, and to quickly adjust the concurrency degree of the system. Thus, the tuning mechanism has two semaphores, one for top-level threads and another for nested ones. The act of enforcing the tuning decision is materialized by modifying the respective semaphore's permits. We extended Java's `Semaphore` class in our implementation, in class `jvstm.tuning.AdjustableSemaphore`. The rationale for this extension is that Java's default semaphore class' method `reducePermits` is package-private. `reducePermits` is a method that reduces the number of permits a semaphore has without blocking.

[2] We use volatile variables to prevent the usage of cached values when aggregating each thread's statistics.

The only other way to decrease a semaphore's permit count is to use the `acquire` method, which by design blocks the calling threads until there is an available permit. Our tuning system must not block while managing the thread count, which is done by adjusting the number of permits, and hence we used this approach.

Increasing the number of permits of a semaphore is a trivial operation, achieved by calling the `release` method of a semaphore instance using the increment as an argument. The number of permits is thus modified, and if any threads were waiting they resume execution. When tuning decreases a semaphore's permits there are two possible paths: There are enough free permits to accommodate the change with no further complications, or, conversely, the resulting permit count will be smaller than the number of currently active threads. In the latter case, the effects of the configuration change may not be observed immediately. To reduce the number of active threads, we have two options: Wait for transactions to finish (either by committing or aborting), which will eventually bring their number down to the desired level, or select a number of threads to deactivate, i.e. abort their transactions, and re-schedule these transactions, and force the thread to wait. We have chosen the former in our implementation. Forcibly deactivating threads has been considered, and briefly studied, but was abandoned. The rationale for this decision is our attempt to modify JVSTM's current implementation to the smallest possible extent, with the aim of creating a self-contained tuning system. Creating a mechanism to re-schedule forcibly aborted threads in a way that recreates their original ordering would impose too many modifications on JVSTM's code. Additionally, we have no control on the ordering in which the semaphores distribute permits. If a thread were to be deactivated in this manner, we would have no control over the scheduling of this thread regarding the time at which it would acquire a permit. However, this approach constitutes a pertinent hypothesis to test in future work, as comparative tests would clarify which approach performs best.

The core of the tuning system is implemented in the `jvstm.tuning.Controller` class, which orchestrates the tuning system. The controller runs in a daemon thread, separate from the application and the JVSTM library. When JVSTM is first initialized, it instantiates the controller class, and starts a new thread for it to run (we call this the *controller thread*). A single instance of the controller is used by JVSTM. The controller thread is only active at the beginning of the tuning intervals, where it calls the tuning algorithms to calculate a target concurrency degree for JVSTM. These algorithms are contained in classes that extend the abstract class `jvstm.tuning.policy.TuningPolicy`. Subclasses of `TuningPolicy` (hereinafter referred to as *tuning policies*) represent a single, isolated tuning strategy that is pluggable into the tuning system, as discussed in this chapter's preamble (chapter 3). Section 3.2 dissects the algorithms used in the various policies we devised; the following section (3.1.3) covers the architecture of a tuning policy.

3.1.3 Tuning Policies

Tuning policies contain of the main logic used for generating dynamic configurations as a result of evaluating the system's performance. All policies must instantiate a `PointProvider` (class `jvstm.tuning.policy.PointProvider`), which implements a sampling algorithm. This algorithm is responsible for sampling points (whose coordinates represent a system configuration) from the search space (Section 3.1.1). As we wished to test different tuning policies we must allow flexibility in the algorithm choice, and thus the sampling algorithms are policy-specific: each policy uses a subclass of `PointProvider`, usually defined as an inner class, for modularity. The sampling algorithm may need information on the previous configurations and their performance to make an informed decision. As such, the `PointProvider` class maintains a record of all configurations used by JVSTM. As these records must be saved for future executions of the policy's tuning algorithm, we save them until the application ends its execution. The reason for this design is the production of a log containing all the steps taken by the tuning system, and their effects. The `jvstm.tuning.StatisticsCollector` class fetches this data during the Java VM finalization (via Java's standard `ShutdownHook`), and produces a log file. We use this log file extensively in our test. Further details are provided in section 4, including the overhead generated by maintaining the logs.

Tuning policies are also the containers of the global statistics (and are responsible for aggregating them from each transactional thread) and the semaphores which dictate the concurrency configuration. The motive behind this design is code clarity: Conceptually, the controller could host the global statistics and semaphores, but all interactions with them would require a call to the controller instance. Since the controller itself does not interact with these members, there is no reason to encapsulate them so.

Each tuning policy class must override the base methods `run`, `tryRunTransaction` and `finishTransaction` (Hereinafter we refer to these methods as the *signalling) methods*). Transactional methods invoke these methods in the controller instance, which in turn delegates the call to its tuning policy. This way, all interactions external to the tuning system are routed through the controller, whereas internal operations are allowed design freedom. The `run` method is called by the controller thread at each tuning interval, and is the method that effectively computes the new configuration. `run` uses the policy's `PointProvider` instance to sample a point in the search space when needed, or, otherwise its own logic to exploit a promising configuration. `tryRunTransaction` and `finishTransaction` try to acquire and release a permit from a semaphore, respectively.

The set of design aspects we analysed compose the broad architecture of our solution. The following section (3.2) overviews each tuning policy in detail, and attempt to clearly define its behaviour by providing an algorithmic perspective.

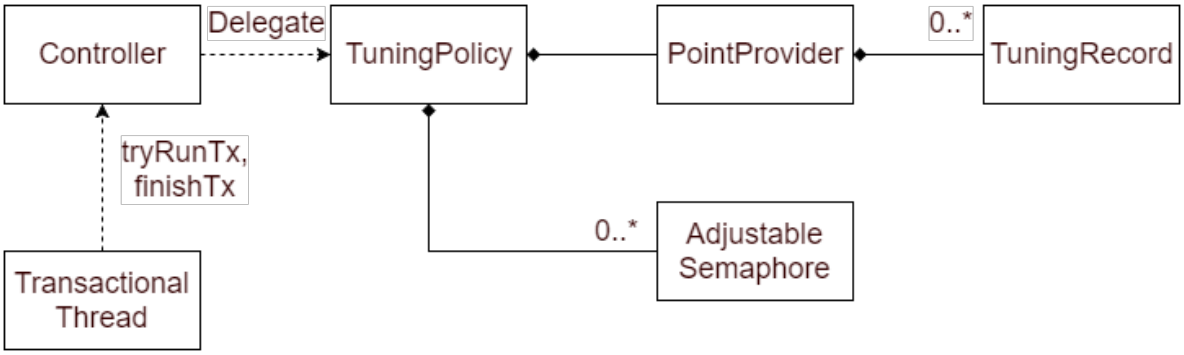

Figure 3.1: Architecture diagram. All transactional threads call the `Controller`'s methods when they begin (`tryRunTx`), commit, or abort (`finishTx`). The `Controller` delegates these calls to its current `TuningPolicy`. The policy instance contains a `PointProvider` for sampling and maintaing `TuningRecords`. After sampling a point, and running its own logic, the policy enforces the configuration represented by the point by adjusting the available permits of one or more `AdjustableSemaphores`. The policy's `tryRunTx` and `finishTx` methods respectively try to acquire and release a permit from a semaphore.

3.2 Algorithms

The centre of algorithmic complexity of JVSTM tuning lies in the tuning policies used by the controller. Each tuning policy subclass represents an isolated, independent algorithm to optimize the system's performance, i.e. produce a dynamic configuration by evaluating performance and previous configurations.

Policies are run by the controller instance, which centralizes all outside interactions with the tuning system. Their tuning logic is entirely independent of other tuning components, and is executed at the beginning of each tuning interval.

Section 3.1 offers an architectural overview of the system and the policies' role in it, whereas this section analyses each policy in an algorithmic perspective. Each policy is described in detail, and has an associated search pattern figure. The purpose of these figures is to convey a visual representation of a policy's search pattern, to complement the algorithmic description. The X axis represents top-level threads, the Y axis nested threads. The search space is bound so it does not oversubscribe the processor cores, that is, the total sum of transactional threads, top-level or otherwise, must not be greater than the processor count. Static local and global optima are highlighted. Grey lines represent the search path of the algorithm, and greyscale points represent configurations enforced by the tuning system. The darker the point, the more often it was chosen and enforced.

3.2.1 Explore and Exploit

An algorithm that tunes the concurrency degree of a parallel-nested STM can be regarded as an optimization algorithm: we make use of this subject's terminology to establish analogies and better describe the inner workings of our solution. One such aspect is the part of an algorithm that dictates whether it should explore points in the global search space of the current tuning context (bound by the machine's available cores), or exploit promising configurations that were previously found.

In the policies we created, this particular aspect is materialised as the simplest possible mechanism to switch between exploration and exploitation (hereinafter referred to as *switching*). Considering the nature of the problem, it is natural to focus on the sampling process first, and on switching later: A policy cannot correctly decide between exploration or exploitation if its sampling algorithm is immature and lacks robustness, as this decision should be made using information about the previously used points.

Following this reasoning, we use extremely simple switching mechanisms. Given a tuning policy with a sufficiently simple switching mechanism that ensures a basic measure of fairness [3], the results obtained from running JVSTM with this tuning policy should provide a general representation of the policy's behaviour. Thus, the research of efficient, and possibly complex, switching algorithms is left for future work, and we focus our work on strengthening the sampling algorithms of each policy. All the policies use simple switching mechanisms as described, with subtle variations.

3.2.2 Default Policy and Overhead Measurement

To compare the behaviour of executing JVSTM with and without tuning (*baseline*), we have created two special tuning policies.

`DefaultPolicy` is a policy that essentially does nothing. Statistic are not kept or collected, and its signalling and run methods are empty. This is equivalent of running JVSTM with no tuning, with an exception: Transactional threads will still invoke the controller's methods in their transactional events (begin, commit and abort). The the effect of these invocations is negligible, as no synchronization or complex operations are involved, are the methods have no code of their own.

The results of running JVSTM with this policy are used in our tests as the baseline results. The results of running JVSTM with this policy are a close approximation to the results of running an unmodified JVSTM. We use this policy as our baseline to simplify the execution of the experimental evaluation.

[3] We refer to fairness in such algorithms as the property that ensures that both exploration and exploitation should eventually have an opportunity to run. One such simple example is: a policy should have as many consecutive executions exploiting as it does exploring, once a promising point is found. This particular mechanism is used in the RRS policy (Section 3.2.6)

`FakeLinearGD` is a policy that performs all the steps we described so far, and uses a Gradient Descent algorithm (specifically, linear gradient descent, described in section 3.2.3.A) to sample the search space, but does not enforce the configurations it produces. This policy is used to measure the overhead of running the whole tuning process independently of the effects it exerts on JVSTM's baseline behaviour.

Performance comparisons are obtained by evaluating throughput measurements and/or execution time. Chapter 4 elaborates on these measurements and references these two policies in the context of analysing the experimental results we obtained.

3.2.3 Gradient Descent

Gradient Descent (GD) is a thoroughly studied category of optimization algorithms used for a variety of problems. Gradient Descent methods are based on the observation that if a multi-variable function is well-defined in a certain region, this function's value decreases fastest if one follows the direction of the function's negative gradient at a given point, i.e. if the function's derivative is not zero at that point, following the steepest slope guarantees the values taken by the function will decrease as fast as possible in the function's domain.

The GD algorithm variants are discrete due to the nature of our problem. The tuning configuration search space is composed of the 2D points inside the region bound the restrictions described in 3.1.1. Therefore, the sampling algorithms must extract points with non-fractional coordinates from this search space. In the context of sampling the search space we refer to tuning configurations as *points* where it is appropriate.

We use this GD algorithms in two policies. The following sections describe the particular variants used in each of these policies.

3.2.3.A Linear Gradient Descent

The Linear Gradient Descent (LGD) policy, embodied in the `LinearGradientDescent4`[4] class, implements the simplest variant of the GD algorithm. 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

[4] In the initial development phases we went through a number of iterations of this policy until we arrived at a stable architecture, yielding many of the insights presented in Section 3.1. `LinearGradientDescent4` is the fourth and final such iteration.

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.

This selection of the most promising point in a neighbouring region can be regarded as the exploitation phase of these category of algorithms. More complex strategies can be examined in future work. An example of one such strategy would be to record several of these scan phases and exploit (i.e. run a certain number of tuning intervals) the most promising point found. In the present case, switching to exploitation is triggered when no neighbouring point exhibits better performance: In this case, LGD maintains its current configuration.

LGD is the simplest of the GD policies, and attempts to measure the effect of applying such a simplistic approach to tuning.

Figure 3.2 provides a visual representation of the exploration path taken by an instance of this search algorithm.

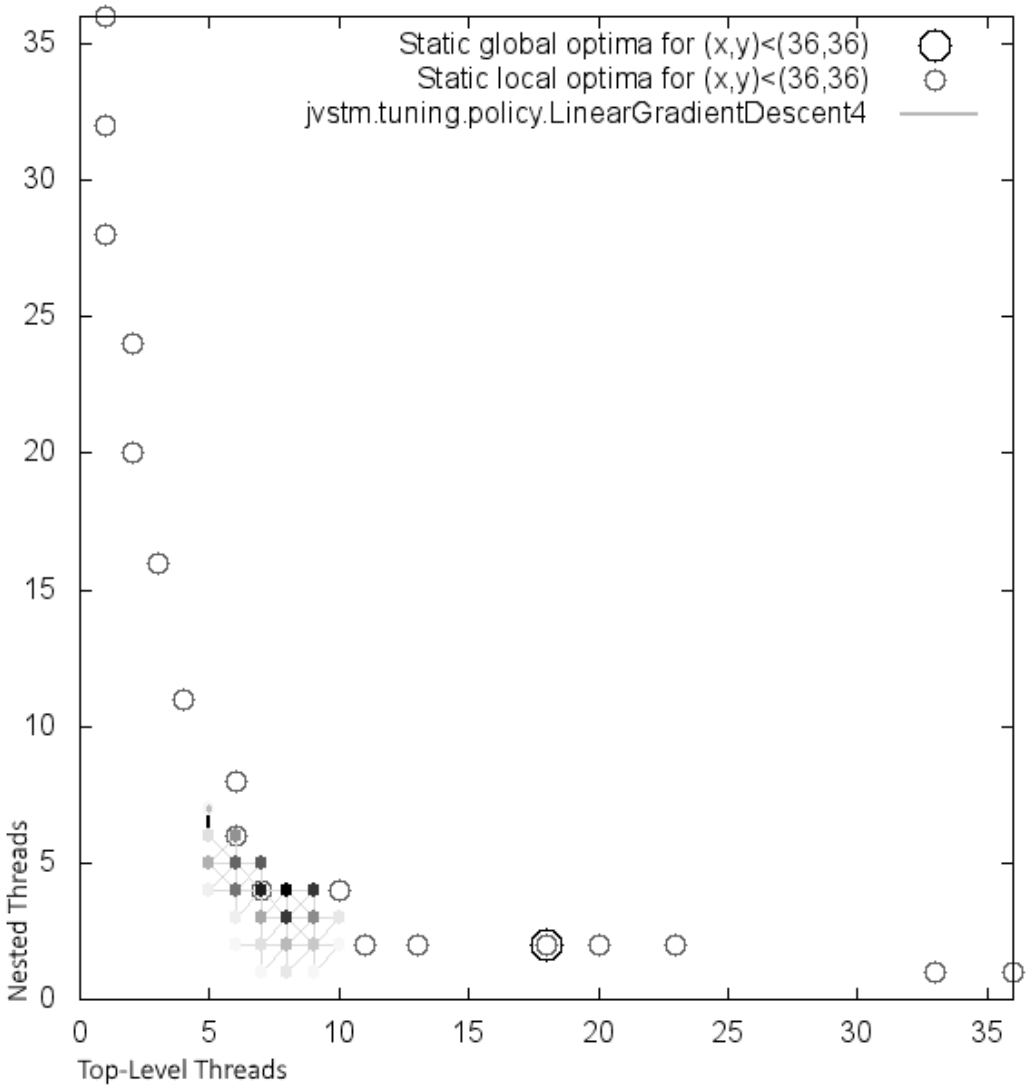

Figure 3.2: Linear GD search path and static optima. The X axis represents top-level threads, the Y axis nested threads. Grey lines represent the search path of the algorithm, and greyscale points represent configurations enforced by the tuning system. The darker the point, the more often it was chosen and enforced. In the LGD case, we can see the algorithm converging around a static local optimum with coordinates (7,4).

3.2.3.B Full Gradient Descent

Full Gradient Descent (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 LGD, with the exception of the sampling algorithm: Instead of extracting four points, FGD samples all the neighbouring points. In a 2D plane[5],

[5]For correctness, the sample is taken from the set of points with non-fractional coordinates that are contained in a two-dimensional plane, not the plane itself.

the cardinality of the set containing these points is 8.

This policy 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.

Figure 3.3 provides a visual representation of the exploration path taken by an instance of this search algorithm.

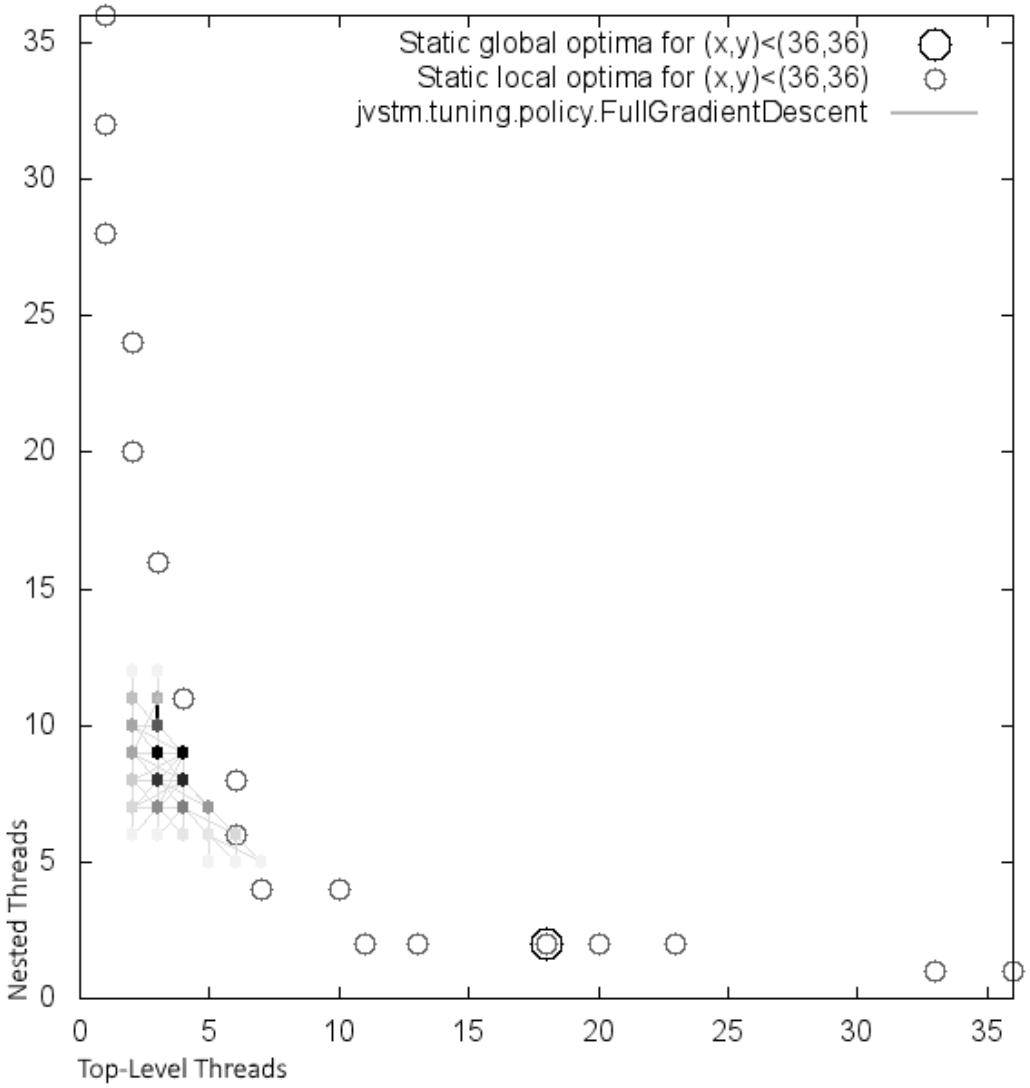

Figure 3.3: Full GD search path and static optima. The X axis represents top-level threads, the Y axis nested threads. Grey lines represent the search path of the algorithm, and greyscale points represent configurations enforced by the tuning system. The darker the point, the more often it was chosen and enforced. FGD does not converge to any static optima, despite scanning point neighbourhoods more exhaustively.

3.2.4 Hierarchical Scan

The Hierarchical Scan (HS) policy samples points by selecting them from *scanlines*. Given the nature of the search space, a scanline can be defined as a set of points with one fixed dimension and one free dimension, e.g. the set of points whose X (top-level) coordinate is the same for all points and each point's Y (nested) coordinate represents one of the possible values for y given x.

This policy explores the hypothesis that there may exist line-like regions in the search space that

yield better performance results than their neighbouring regions. Visually, this notion could be represented by a long lines of points, either vertical or horizontal, whose throughput value is greater than their neighbours, e.g. a line-like region of local optima.

The rationale behind the creation of this policy was to test if these types of regions existed in our search space. The literature suggests optimum points frequently cluster together[47]. With this policy we concluded that the hypothesis that these clusters are line-like does not hold, as discussed in Chapter 4.

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. Figure 3.4 provides a visual representation of the exploration path taken by an instance of this search algorithm.

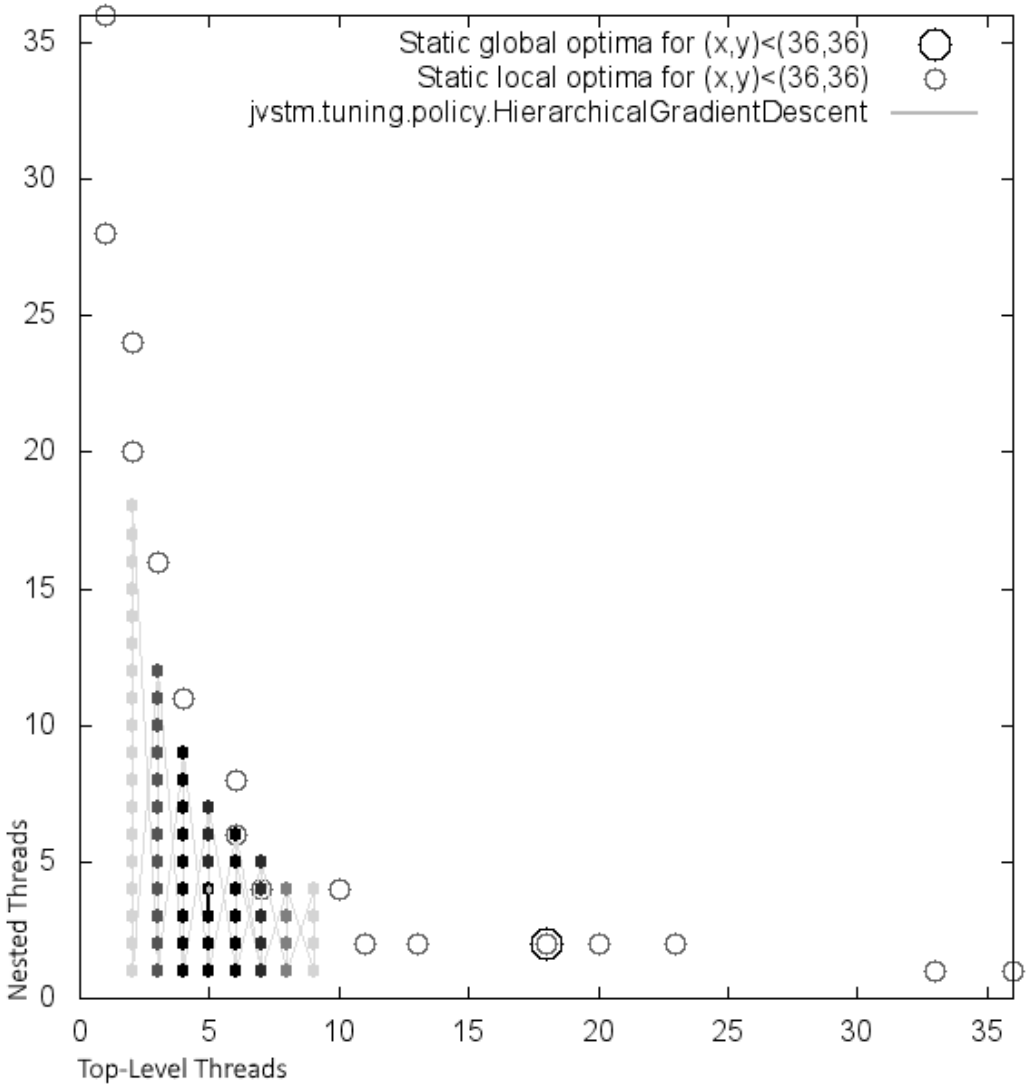

Figure 3.4: Hierarchical Scan search path and static optima. The X axis represents top-level threads, the Y axis nested threads. Grey lines represent the search path of the algorithm, and greyscale points represent configurations enforced by the tuning system. The darker the point, the more often it was chosen and enforced.

3.2.5 F2C2

The F2C2 policy follows the work developed by the authors of [26], where the authors apply of 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 original algorithm does not re-start its exploration phase. This assumes that the application's workload remains relatively stable during execution, and thus the region of interest found by the explore phase will be the optimal region to exploit through the application's lifetime. In the benchmarks used in our tests (discussed in section 4) this assumption does not hold. Workload shifts occur frequently and unpredictably. Additionally, we have to account for the greater dimensionality of our search space, when compared to the original algorithm. Thus, we devised a two-dimensional version of this algorithm, and a simple switching algorithm for our implementation of F2C2 (the switching algorithms follows the logic presented in section 3.2.1).

To explore our two-dimensional search space in the manner devised ty F2C2's authors, we fix one dimension and perform an exponential search step on the other. Afterwards, the originally fixed dimension is exponentially incremented, fixed again, an the process restarts. With this approach we sample points from the whole search space following the coarse-grained approach of F2C2.

F2C2 stops exploring after it performs the coarse-grained search over the whole search space. The search pattern fits our search space graciously, as shown in figure 3.5. This coarse-grained search phase highlights the most promising points for exploitation. F2C2 then starts exploiting one of these point (the most promising) by enforcing the configuration represented by the 2D point for as many tuning intervals as the exploration phase used. After this set time period, F2C2 re-starts the exploration phase, in the same manner as described above.

Figure 3.5 provides a visual representation of the exploration path taken by an instance of this search algorithm.

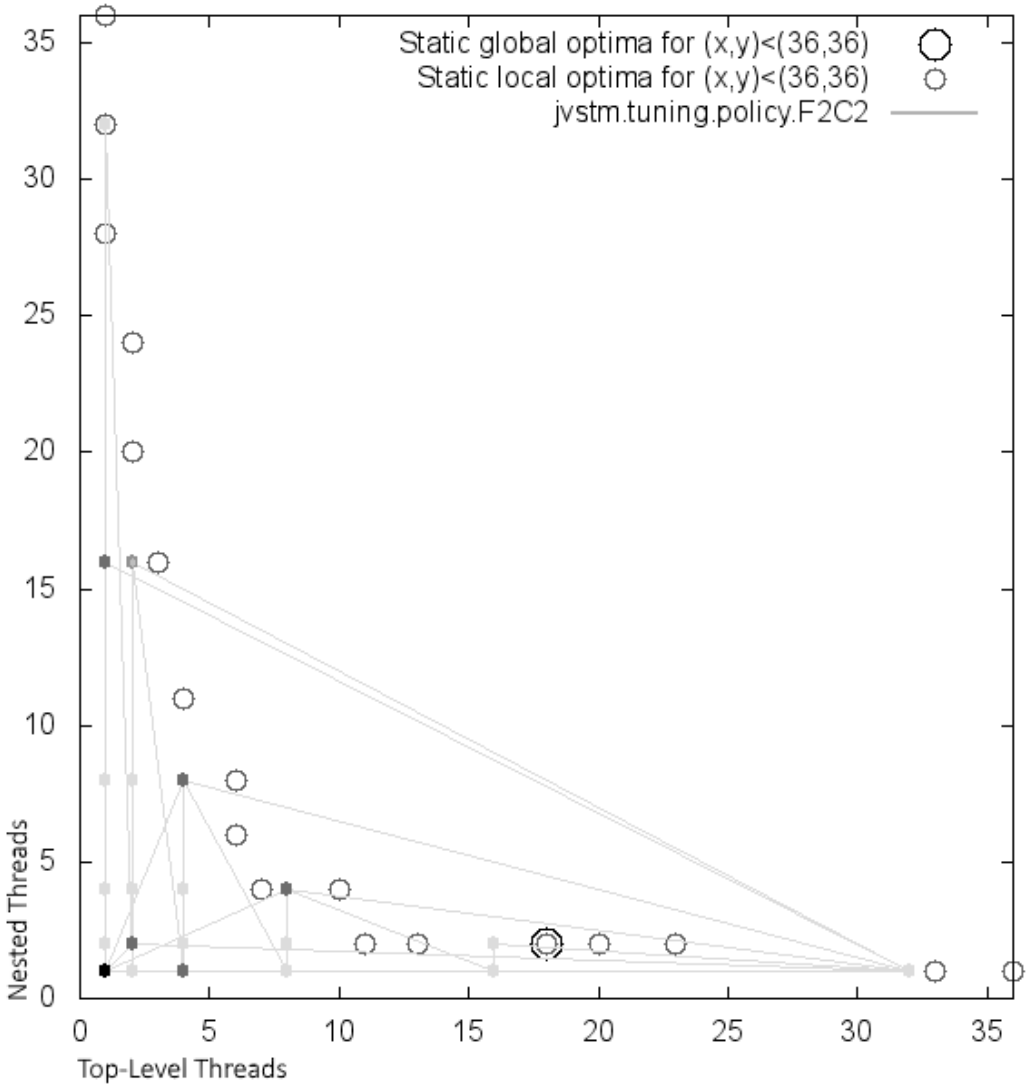

Figure 3.5: F2C2 search path and static optima. The X axis represents top-level threads, the Y axis nested threads. Grey lines represent the search path of the algorithm, and greyscale points represent configurations enforced by the tuning system. The darker the point, the more often it was chosen and enforced. The exponential increments in each coordinate can be seen as the long jumps in this visual representation.

3.2.6 Recursive Random Search

Recursive Random Search (RRS) [47] is a sampling 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.

With this algorithm we wished to test the adequation of Random Search to our problem. From an initial, uninformed point of view, this class of sampling algorithm did not seem appropriate, as our workloads vary frequently during execution, and the search space configuration (Section 4.2) is noisy. We wished to test this empirically, however, to validate or discard this possibility based on real data.

RRS relies on the assumption that promising points cluster around some optimum, either local or global. The authors provide a mathematical model of the probability of finding one of these clusters after a certain number of samples is taken. The model outputs this number, called `n`, which is the number of samples to be taken before a promising point can be identified. Two parameters must be configured, `p` and `r`. `p` represents the confidence in finding a promising point (i.e. close to an optimum) after `n` samples, and `r` represents the measure by which the search space is reduced after this find.

By carefully selecting parameters (`p` and `r`), one can obtain a precise number `n` of sampling steps to be taken such that any further sampled point with a performance measurement above the average performance measure of the first initial `n` points will be a point that is worth exploring, with confidence `p`. `p` reflects the confidence level one has that these partitions of the search space are likely to contain a global optimum. The authors suggest that `p` should be initialized to a high value (0.99 in their publication), and `r` should be considered carefully, as it regulates the switching between exploration and exploitation (0.1 in their publication).

`p` is dependant on the characteristics of the search space, specifically on the clustering of promising points. Should the optima be completely independent, and form no clusters, `p` should take a different value. We did not test different value ranges.

When the algorithm is exploring, after `n` points are sampled their respective measurements are averaged. Any subsequent point that has a measurement lower than this average triggers a resize or re-align operation.

Resizing the search space is accomplished by removing points from it until the size requirement is met, starting with those closest to its border. Re-aligning the search space consists of defining its centre at a certain point (e.g. a promising point found in exploration) and including neighbouring points until its size matches the required size. These two operations increase the resolution of the search phases

Exploitation is not well-defined is this algorithm. One can argue that decreasing the search space space size is an exploitation technique in itself. However, a mechanism that resembles exploitation is triggered when the exploration of the current search space has sampled a number of points greater than the previous iteration has (i.e. the exploration phase that happened before the search space was

modified) with no new candidate points. In this event, the search space is shrinked around the current best point.

When the search space size falls within a given threshold, exploration is re-started, using the original search space.

4

Experimental Results

Contents

In this chapter we evaluate the performance obtained by JVSTM's tuning system. Section 4.1 describes the experimental setup we used in our tests. Section 4.2 presents the results we obtained and discusses their implications.

4.1 Experimental Setup

Two benchmarks were used to evaluate JVSTM tuning's performance: STAMP Vacation and STMBench7. Sections 4.1.1 and 4.1.2 describe the configuration and usage of these benchmarks. Section 4.1.3 details our methods for data collection and analysis.

4.1.1 Vacation

Vacation belongs to the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite [48], 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.

During execution, clients threads perform sessions that interact with the database kept by the benchmark. These sessions perform tasks such as reservations, cancellations and updates. Each of these sessions is encapsulated in a coarse-grained transaction, to guarantee database consistency. Each operation performed in a session is contained in a nested transaction, and can run concurrently with other operations. Vacation emphasizes the elegance of using STM systems for parallelization, as parallelizing Vacation with lock-based schemes is far from trivial.

Vacation allows the contention level to be parametrized via controlling the percentage of objects in the database that can be accessed by operations. Lower percentages mean a more restrict section of the database will be accessed by transactions, thus increasing concurrent access and contention, while higher percentages induce less contention. In our tests we consider two configurations of this parameter: High-contention, which accesses 1% of the database, and Low-contention, which is allowed access to 90% of the database.

Vacation is initialised with a fixed number of transactions (tasks) to run, and its execution time varies. As direct throughput measurements are not trivial to gather in Vacation, we measure execution time

for each run. All algorithms go through an exploration-exploitation loop until the benchmark finishes execution, and therefore the configuration never stabilizes on a given value. As we mentioned in Chapter 3, we focus on sampling rather than switching between exploration and exploitation. Workloads are highly variable when compared to the static optima we found in preliminary analysis, that is, the static optima are not stable during execution.

4.1.2 STMBench7

STMBench7 is a benchmark for STM systems that tries to emulate a complex and realistic application [49].

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.

We provide results for three workload types: read-only, read-write, and write-only. In our tests we collect data for various configurations regarding the number of initial top-level and nested threads. All tests use a fixed length of 20 seconds. As in Vacation, the explore-exploit loop runs until the benchmark finishes. The results we obtained for STMBench7 were not averaged from a series of tests. Rather, they are come from a single test spanning all our pre-defined configurations and policies. Thus, care must be taken when interpreting them.

4.1.3 Data Collection

4.1.3.A Vacation

For the Vacation benchmark we collect a variety of measurements and represent them in plots to visually convey the information. These measurements are:

- Execution time - This measure represents the time it takes for the main segment of the algorithm to execute. 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.

- Overhead - A comparison between the execution times of running Vacation with the baseline JVSTM, JVSTM with statistics collection, JVSTM with tuning, and JVSTM with tuning and statistics collection. Each result is averaged from three executions.
- Exhaustive static configuration search - This measurement allows us to obtain a visual representation of the execution time of each possible static configuration. Although the benchmark's behaviour at runtime may not reflect the information these plots convey, they provide a broad notion of how much noise is present in the search space.

4.1.3.B STMBench7

This sections details the measurements we take from STMBench7's execution, and their relevance to our analysis.

- 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.
- Exhaustive static configuration search - This measurement mirrors the one we use in Vacation. It allows us to perform a visual analysis of the execution time of each possible static configuration.

4.1.4 Platform

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.

4.2 Results

This sections presents the results we obtained by running tests as described in section 4.1 for each benchmark. Each subsection discusses the implications these results.

4.2.1 Vacation

4.2.1.A Search Space overview

We mapped the execution time of all system configurations allowed by the number of cores in our test machine, for Vacation. We call these *exhaustive results*. Exhaustive results do not reflect the behaviour of the tuning system. Instead, they provided valuable insight of our search space in an initial analysis, as we built a notion of what to expect in terms of noise, i.e. the measure of how smooth our search space is in terms of each configuration's execution time.

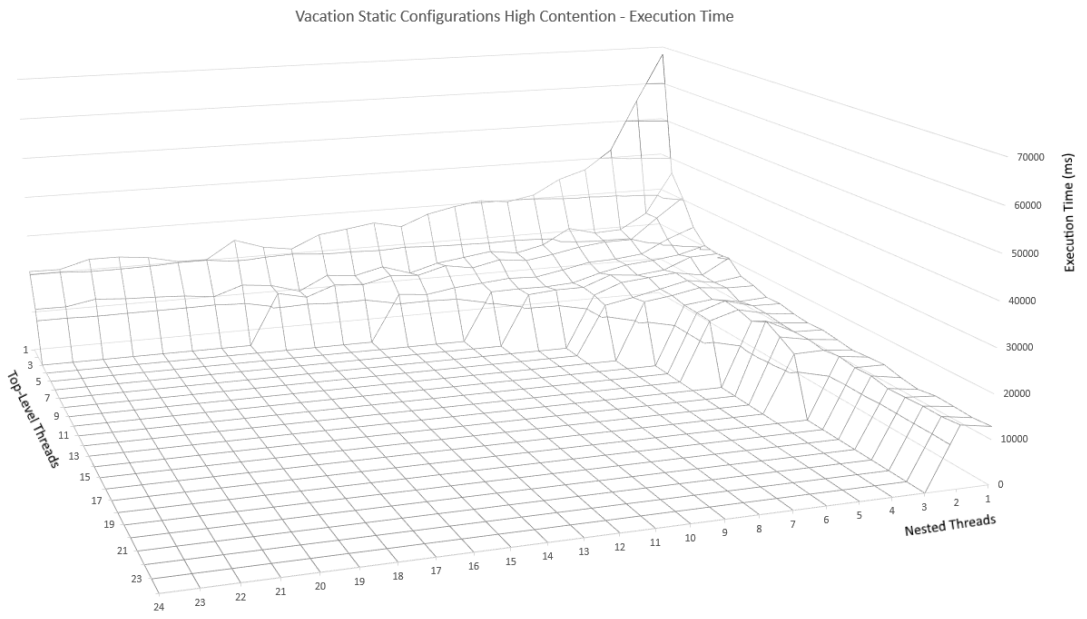

Figure 4.1: Vacation exhaustive test results for high contention. Horizontal axes represent top-level and nested thread count. The vertical axis represents execution time, in milliseconds. The flat portions of the graph where execution time is zero were not measured, as they represent configurations that would oversubscribe the processor cores. Tests were run using static configurations, i.e. configurations that remain unchanged for the whole execution.

Figure 4.2 details the search space mapping for Vacation running with a high contention configu-

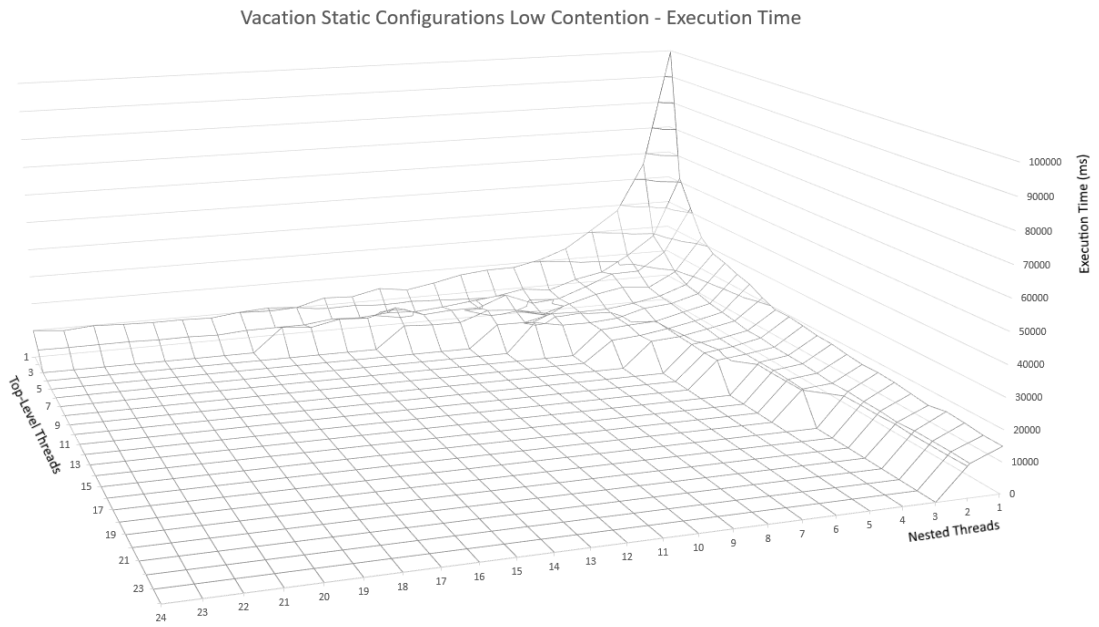

Figure 4.2: Vacation exhaustive test results for low contention. Horizontal axes represent top-level and nested thread count. The vertical axis represents execution time, in milliseconds. The flat portions of the graph where execution time is zero were not measured, as they represent configurations that would oversubscribe the processor cores. Tests were run using static configurations, i.e. configurations that remain unchanged for the whole execution.

ration, whereas figure 4.1 details the search space mapping for the execution of this benchmark with a low contention configuration.

These plots demonstrate a high level of noise once the number of top-level and nested threads increases, as the performance curves are not monotonic. This confirms our intuition that a tuning system must have a robust sampling algorithm, to avoid converging to a local optimum that may be far from the system's optimal configuration.

These 3D plots are not intended to provide precise information about each configuration's performance. Rather, they portray a general vision of performance variation between configurations. The same applies to STMBench7's exhaustive results, detailed in section 4.2.2.A.

4.2.1.B Overhead

To measure the overhead of the various components of our system, we separately measure the toll they take on execution time, running the Vacation benchmark.

Figure 4.3 compares the execution times of running Vacation with the baseline JVSTM (i.e. the

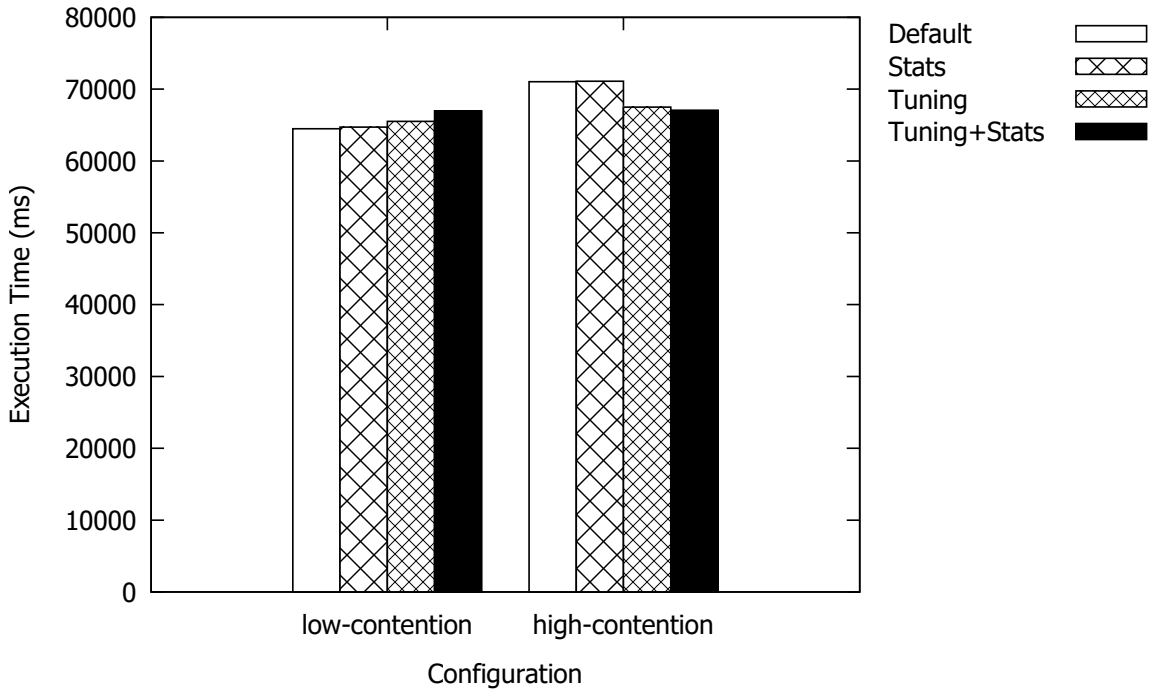

Figure 4.3: Vacation overhead test results.

Default tuning policy, represented as ”Default”), JVSTM with statistics collection (”stats”), JVSTM with tuning, and JVSTM with tuning and statistics collection. Each result is averaged from three executions. For the tuning times we used Linear Gradient Descent.

We conclude that the component parts of our system do not exert too much overhead on the natural execution of JVSTM, introducing average overhead levels of less than 1%. With low contention the system obtains worse performance than the baseline JVSTM. This result is expected, and discussed in detail in the following section.

With high contention values the execution time decreases. This is desirable, and expected, because with high contention the tuning mechanism the number of threads, which leads to JVSTM taking less time to execute.

4.2.1.C Execution Time Comparison

With Vacation, we obtain our main performance comparison results by measuring the benchmark’s execution time. We use a series of pre-defined configurations to represent different cases of nesting (or lack thereof).
Figure 4.4 details execution time comparison for Vacation with high contention, and figure 4.5 for low contention.

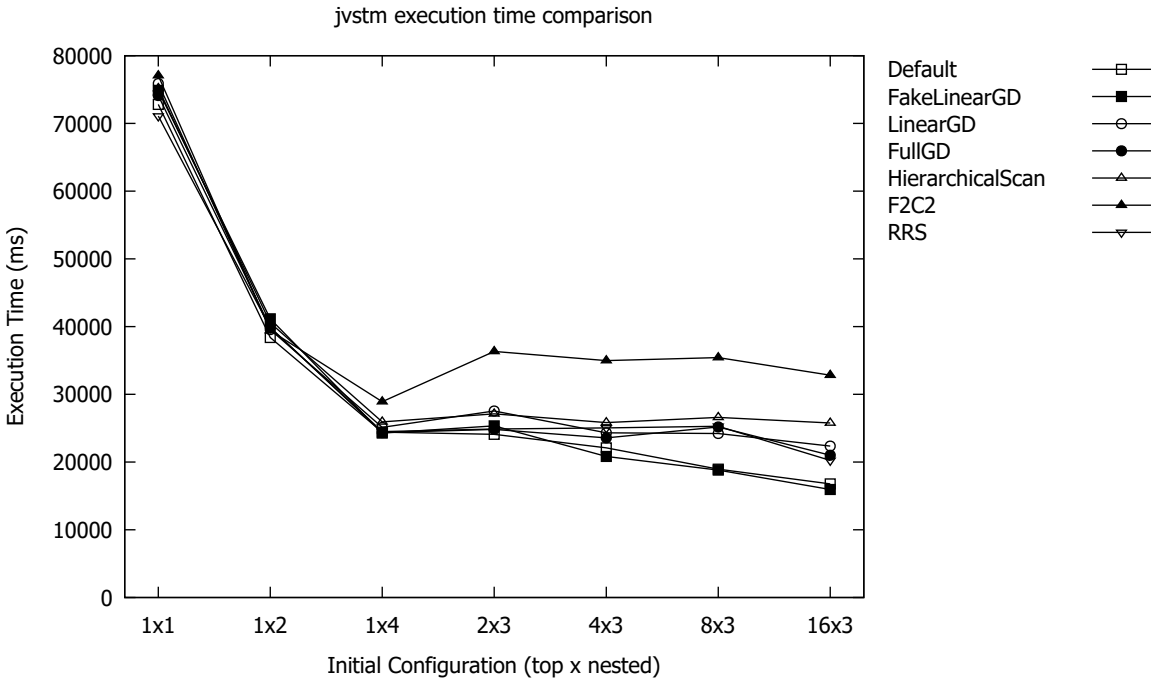

Figure 4.4: Vacation execution time test results for high contention. Lower is better. All policies perform worse than the baseline (*Default* policy, white squares), although Linear GD, Full GD and RRS have small performance losses. F2C2 performs especially bad in this scenario.

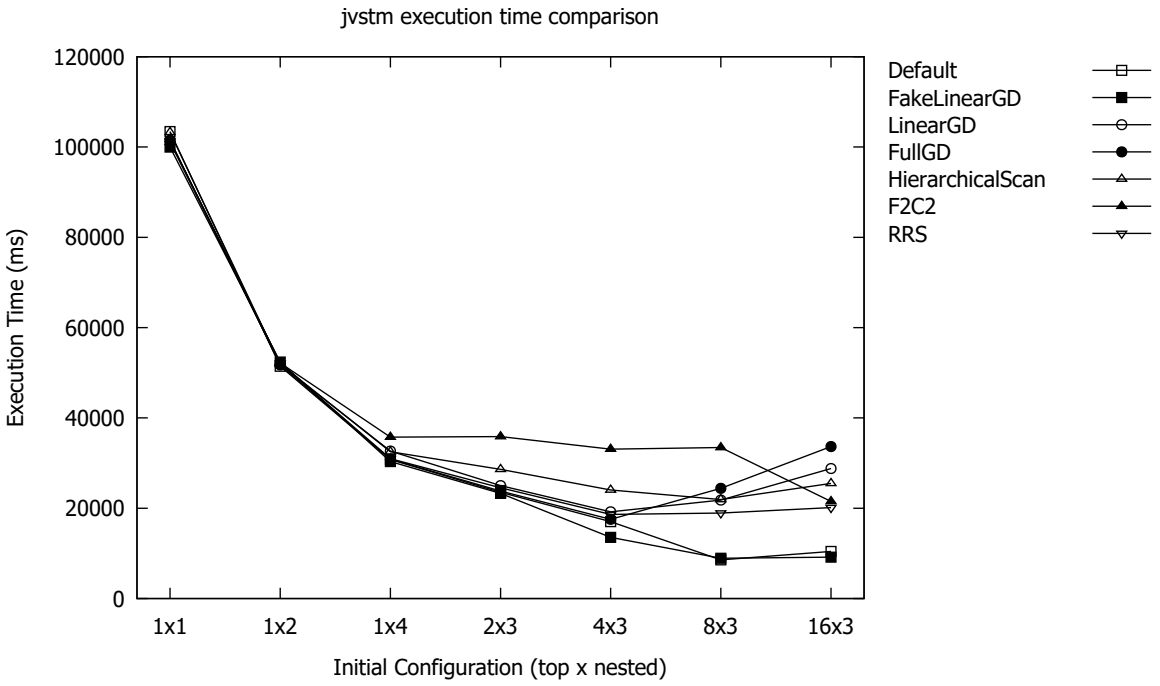

Figure 4.5: Vacation execution time test results for low contention. Lower is better. Hierarchical Scan and F2C2 perform generally bad, while the other policies keep up with the baseline, but degrade performance when the system's initial configuration has a higher numbers threads (8x3 and 16x3 configurations). RRS follows this pattern, but is the policy with lowest performance losses.

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. However, these policies manage to keep performance relatively close to the baseline.

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 using high contention. Random sampling is designed to cover significant portions of the search space quickly, which may lead to regions of interest being found quickly, after which the algorithm reduces the search space around them. Hierarchical Scan does not keep up with the baseline results, on par with F2C2.

4.2.2 STMBench7

4.2.2.A Search Space Overview

With STMBench7, as with Vacation, we mapped the whole search space to examine its shape. Figures 4.6 and 4.7 present this mapping for read-only and write-only configurations, respectively.

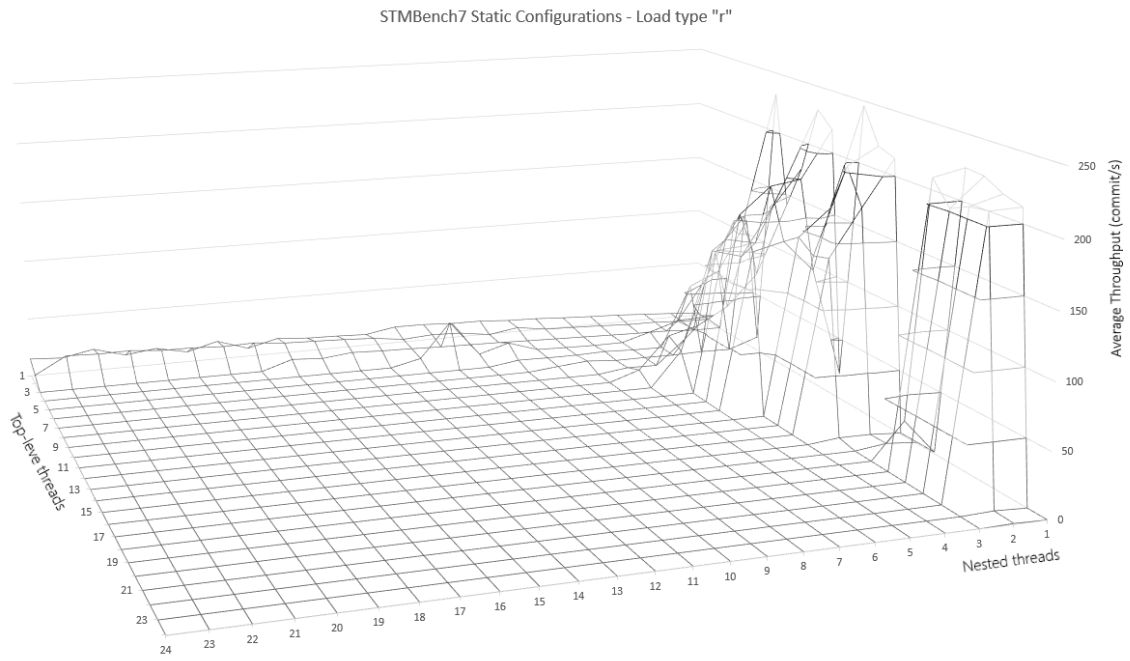

Figure 4.6: STMBench7 exhaustive test results - read-only workload. Horizontal axes represent top-level and nested thread count. The vertical axis represents average throughput, in commits/second. The flat portions of the graph where execution time is zero were not measured, as they represent configurations that would oversubscribe the processor cores. Tests were run using static configurations, i.e. configurations that remain unchanged for the whole execution.

Read-only configurations on STMBench7 favour a high level of top-level threads and low level of nesting. Because this is a read-only configuration, conflicts are rare, and long transactions can run unimpeded with infrequent aborts, which explains the performance gains.

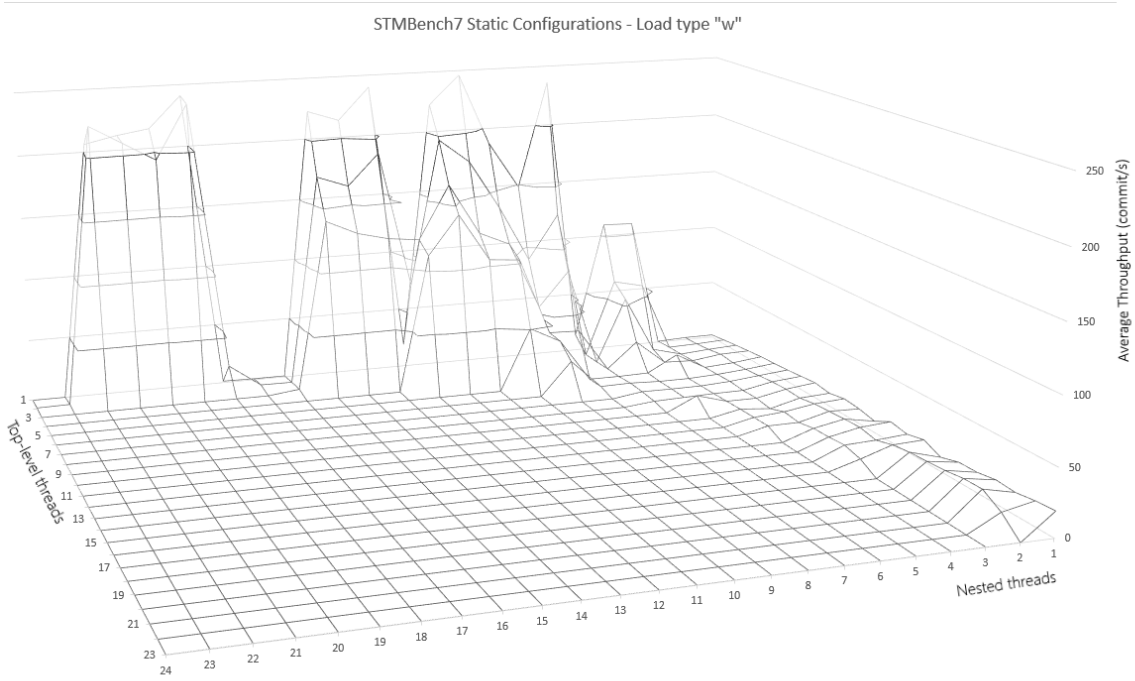

Figure 4.7: STMBench7 exhaustive test results - write-only workload. Horizontal axes represent top-level and nested thread count. The vertical axis represents average throughput, in commits/second. The flat portions of the graph where execution time is zero were not measured, as they represent configurations that would oversubscribe the processor cores. Tests were run using static configurations, i.e. configurations that remain unchanged for the whole execution.

Write-only configurations highly favour nesting. A high number of writing transactions will cause frequent conflicts and subsequent aborts. Thus, high levels of nesting, which cause the system to subdivide its operations into smaller transactions, cause these conflicts to occur in shorter operations, which can be quickly aborted and re-started, yielding better performance results.

4.2.2.B Execution Comparison

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 (relative to the baseline) 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 represents a halving of the baseline throughput.

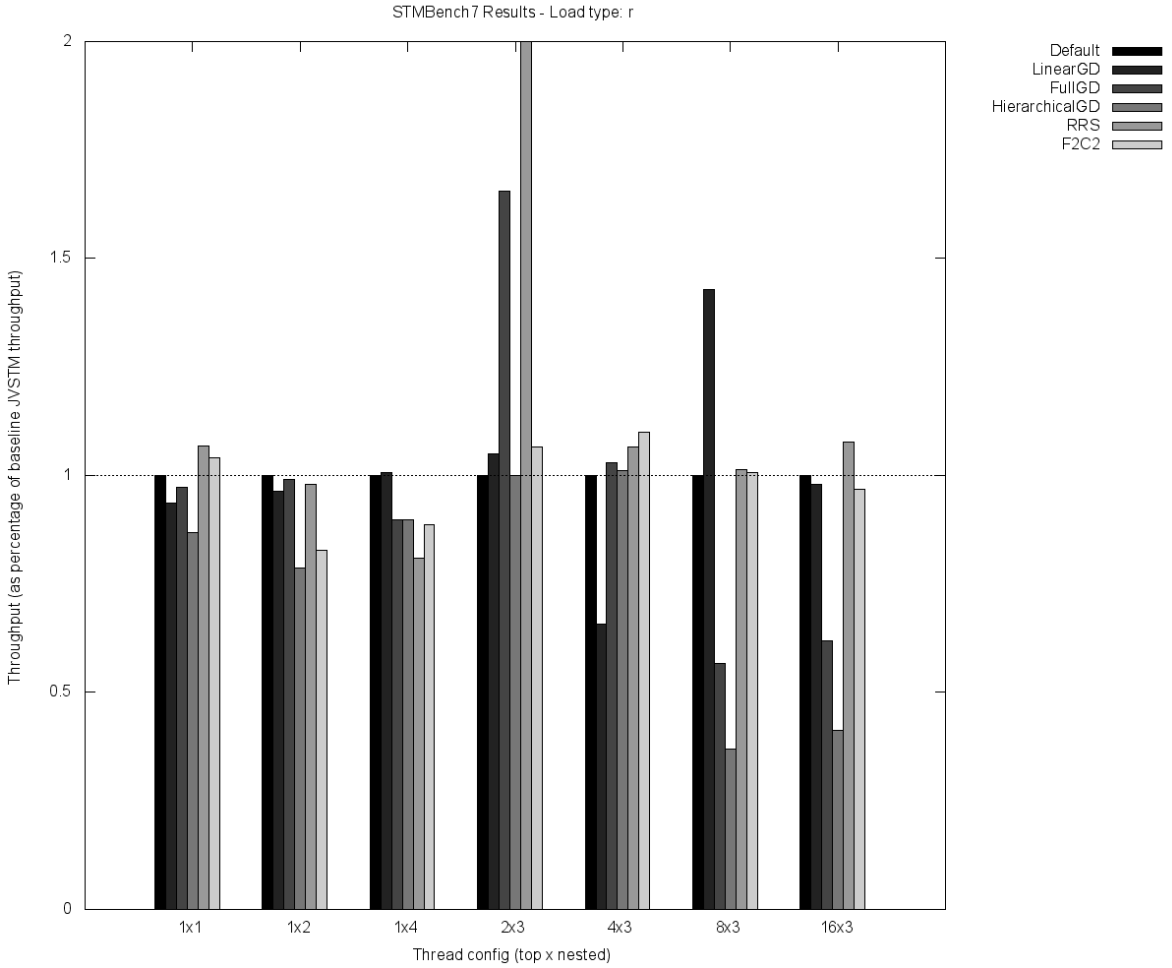

Figure 4.8: STMBench7 throughput test results - read-only workload.

Figure 4.8 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 are on par with the baseline for low thread starting configurations. They show performance peaks on the 2x3 configuration, which can be related to the search space plots presented before. The 2x3 configuration is not particularly efficient, but is close to neighbouring configurations that show extreme performance improvements. Therefore, we surmise that GD policies quickly locate these peaks when starting with a 2x3 configuration. With higher thread configurations performance decreases: as GD policies do not efficiently exploit promising points, they never stabilise around efficient configurations. As such, worse points are constantly enforced on the system.

Hierarchical Scan is generally worse than the baseline, especially with high thread configurations. At best, this policy remains on par with the baseline, on the 2x3 configuration.

RRS and F2C2 perform on par with the baseline. 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. One alternative hypothesis would be that RRS randomly sampled a promising point after the first round of exploration, and quickly converged on this point, and the area of interest around it. This is unlikely, however, because should that be the case, similar behaviour would be expected from other starting configurations, but is not observed.

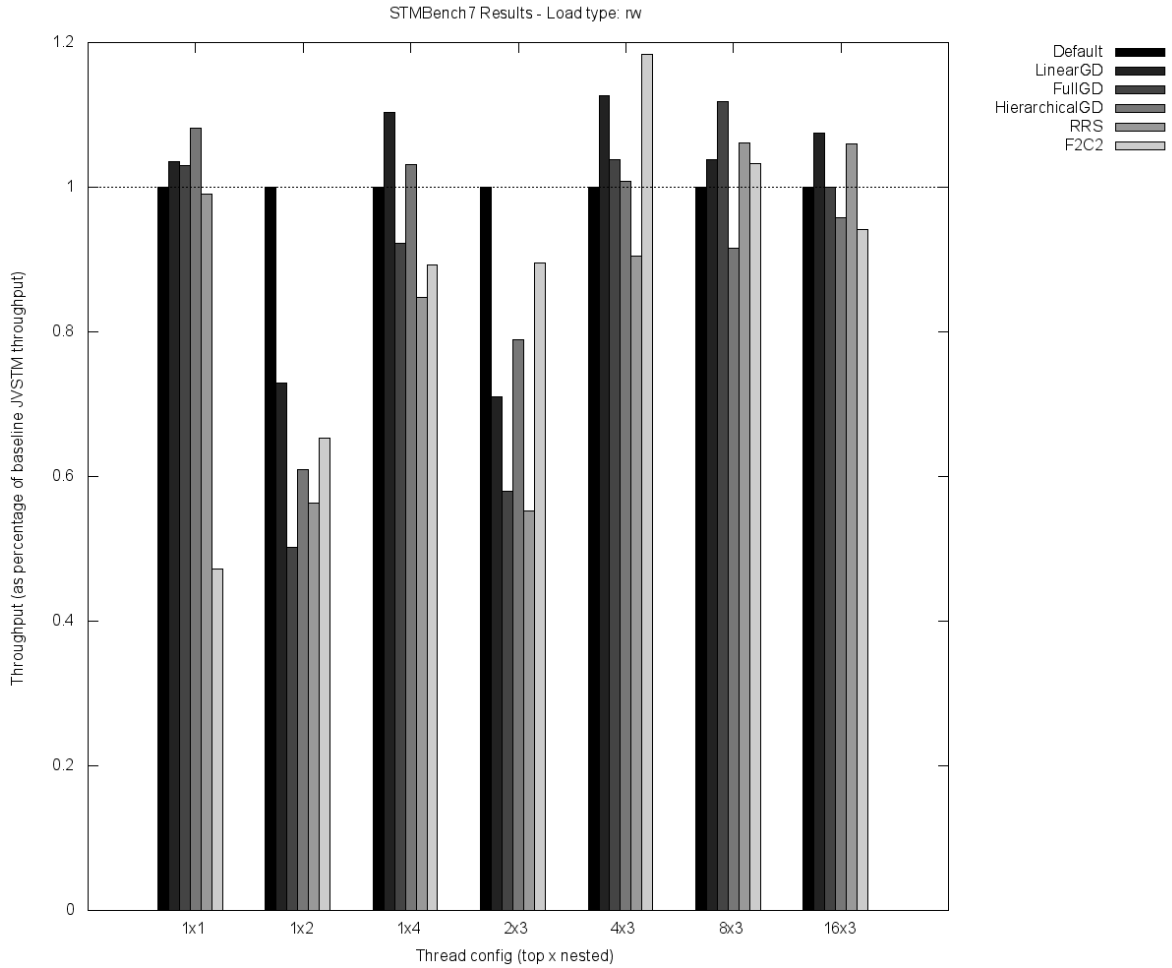

Figure 4.9: STMBench7 throughput test results - read-write workload.

Figure 4.9 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 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. Multiple local optima, frequent behaviour changes in conflict and commit rate, and other factors may contribute to this hypothesis. This workload type is where the tuning system obtains its worse results. Therefore, future work should focus on dealing with this particular case, and improving, or at least maintaining, performance levels when tuning is used on mixed workloads.

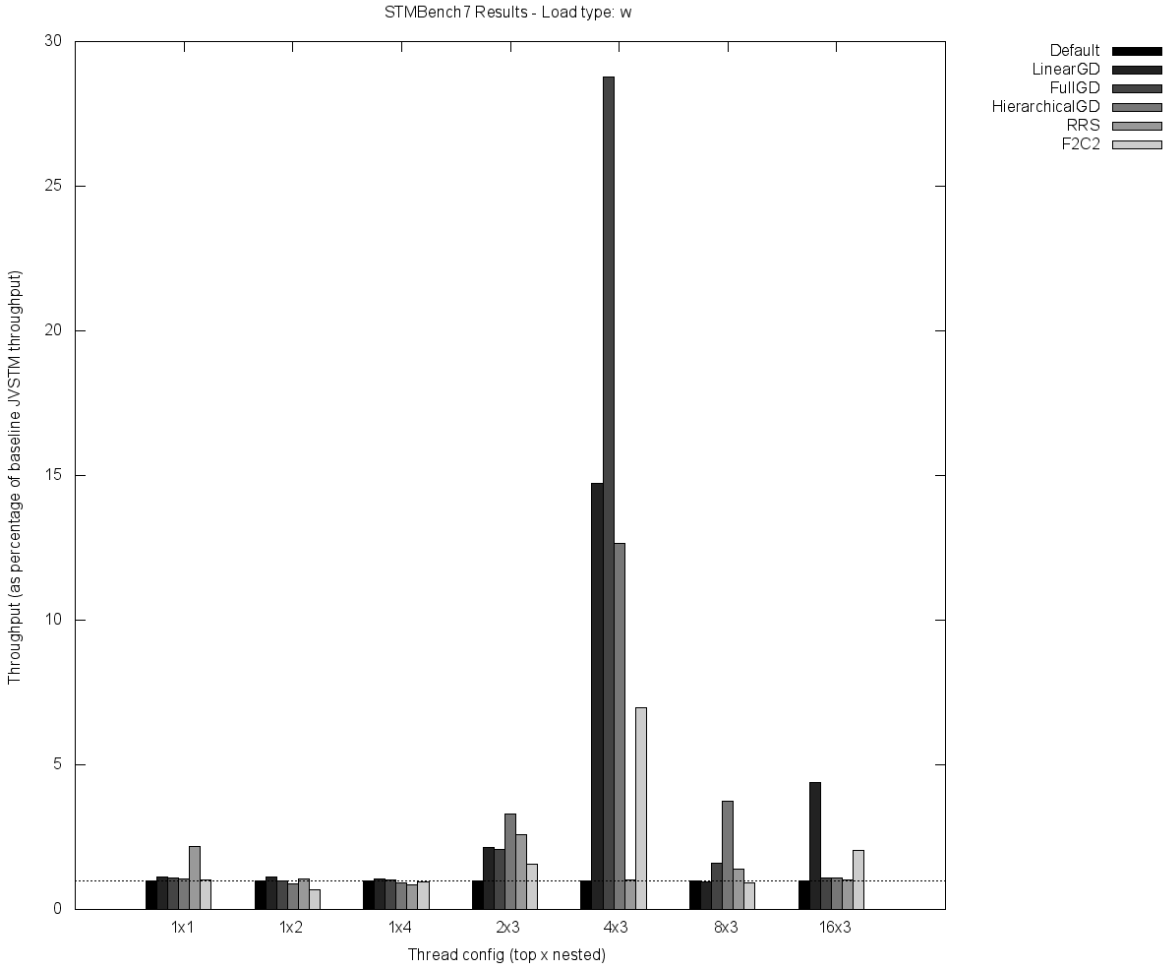

Figure 4.10: STMBench7 throughput test results - write-only workload.

Finally, figure 4.10 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 2x3 and 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 of wasted work, which the tuning system reduces when it finds a more suitable configuration. Thus, write-dominated workloads show the best improvements in performance. These results obtained in these tests should, however be observed with care. As results are not averaged from a series of runs, there may be statistical quirks, even if the observed behaviour is consistent for the same configurations across different polices.

5

Conclusions and Future Work

5. Conclusions and Future Work

Modern devices have increasing computational power and core count, which allow for the exploration of extremely complex applications that are unavailable with traditional single-core machines and sequential architectures. Financial and business computing, for example, gain competitive advantages in exploiting additional parallelism in their applications. Still, parallel programming is far from trivial, and the average programmer cannot implement such complex patterns in a manner that exposes the maximum available parallelism.

Software Transactional Memory is one of the most successful paradigms to abstract the inner complexity of modern-day parallelism, and effectively hides the burden of explicitly parallelizing an application with complex locking and synchronization schemes. STM is far from being an optimal solution however, as it is not globally appropriate for all application and workload profiles. As these can take many forms and respond differently to various transactional configurations, we propose in this dissertation to establish a framework with which one can study and test different types of tuning in STM systems, and to experiment with different tuning policies and measure their effects and results.

Tuning a nested STM system is no trivial task in itself. The architecture of the STM implementation must be considered, and a deep knowledge about its inner workings is required. However, the majority of challenges that the design of this system encompasses can be conceptualized and transported into a general STM environment, and can be useful even in the light of different architectures.

The core of the work developed for this dissertation was the creation of a tuning component in JSVTM, a state of the art STM system developed in the Java programming language. We showed that it is possible to introduce such a component without meddling excessively with JVSTM's code, thus creating a nearly self-contained package. Additionally, this component imposes no significant overhead on the system, as shown by our overhead tests. Specifically, we ran the whole tuning process, but discarded the configurations it produced, effectively exerting the natural overhead of the tuning component without any actual tuning taking place. The results were satisfying, as overhead is kept to a minimum. If the overhead levels were too high, they would entirely undermine the advantages of tuning the system.

Several unexplored or abandoned research paths were briefly overviewed, as we believe they may be worth studying to better understand the nature of tuning nested STM systems.

We used two benchmarks, Stamp (Vacation) and STMBench7, 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 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. Some specific cases show promising results. Tuning JVSTM with the SMTBench7 and a write-only (high contention) configuration yielded exceptional performance gains. This case is isolated, however, as other configurations and benchmarks did not achieve these gains. Despite this

fact, 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.

Our data collection focused on retrieving measurements of the system's throughput and execution times, commonly used in the literature, with the intent of providing a clear and accurate contribution to this area of research. Additionally, this allows our results and data collection tools to be reused in future work.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[2] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095418

[3] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[4] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2Nd Edition*, 2nd ed. Morgan and Claypool Publishers, 2010.

[5] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, ser. LADIS '08. New York, NY, USA: ACM, 2008.

[6] P. Rundberg and P. Stenström, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, vol. 3, 2001.

[7] A. Welc, S. Jagannathan, and A. Hosking, "Safe futures for java," *SIGPLAN Not.*, vol. 40, no. 10, pp. 439–453, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094845

[8] M. Herlihy and J. E. 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.

[9] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking concurrency," *Queue*, pp. 24–33, 2006.

[10] S. Lie and K. Asanovic, "Hardware support for unbounded transactional memory," Master's thesis, MIT, Tech. Rep., 2004.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," *SIGPLAN Not.*, vol. 41, no. 11, pp. 336–346, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168918.1168900

[12] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, "Atomic quake: Using transactional memory in an interactive multiplayer

game server," *SIGPLAN Not.*, vol. 44, no. 4, pp. 25–34, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504183

[13] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 187–207.

[14] D. Rughetti, P. Di Sanzo, A. Pellegrini, B. Ciciani, and F. Quaglia, "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*, ser. Lecture Notes in Computer Science, R. Guerraoui and P. Romano, Eds. Springer International Publishing, 2015, pp. 395–417. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-14720-8_18

[15] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Transactions on high-performance embedded architectures and compilers iii," P. Stenström, Ed. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Robust Adaptation to Available Parallelism in Transactional Memory Applications, pp. 236–255. [Online]. Available: http://dl.acm.org/citation.cfm?id=1980776.1980793

[16] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems*, V. Gramoli and R. Guerraoui, Eds. Springer Berlin Heidelberg, 2013, pp. 233–247.

[17] P. D. Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano, "On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking," *Performance Evaluation*, vol. 69, no. 5, pp. 187 – 205, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016653161100068X

[18] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," *SIGPLAN Not.*, vol. 44, no. 4, pp. 141–150, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504199

[19] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka, "Leveraging parallel nesting in transactional memory," *SIGPLAN Not.*, vol. 45, no. 5, pp. 91–100, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693466

[20] N. Diegues and J. Cachopo, *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Practical Parallel Nesting for Software Transactional Memory, pp. 149–163. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41527-2_11

[21] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345232

[22] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "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*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810528

[23] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory, pp. 475–486. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09873-9_40

[24] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 11:1–11:32, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2620001

[25] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/2668930.2688047

[26] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for stms," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 927–938.

[27] M. Ansari, "Weighted adaptive concurrency control for software transactional memory," *J. Supercomput.*, vol. 68, no. 3, pp. 1027–1047, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s11227-014-1138-5

[28] J. Sreeram, R. Cledat, T. Kumar, and S. Pande, "Rstm: A relaxed consistency software transactional memory for multicores," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 428–. [Online]. Available: http://dx.doi.org/10.1109/PACT.2007.62

[29] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 237–246. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345241

[30] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th International Conference on Distributed Computing*, ser. DISC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 194–208. [Online]. Available: http://dx.doi.org/10.1007/11864219_14

[31] R. Guerraoui and P. Romano, *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014. [Online]. Available: https://books.google.pt/books?id=X74iBgAAQBAJ

[32] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," *SIGPLAN Not.*, vol. 44, no. 6, pp. 155–165, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1543135.1542494

[33] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, Feb 2006, pp. 254–265.

[34] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '05. New York, NY, USA: ACM, 2005, pp. 240–248. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073861

[35] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy, "NePaLTM: Design and implementation of nested parallelism for transactional memory systems," in *ECOOP '09: Proc. 23rd European Conference on Object-Oriented Programming*, jun 2009, springer-Verlag Lecture Notes in Computer Science volume 5653.

[36] R. Filipe and J. a. Barreto, "Nested parallelism in transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, R. Guerraoui and P. Romano, Eds. Springer International Publishing, 2015, pp. 192–209.

[37] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "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*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 187–197. [Online]. Available: http://doi.acm.org/10.1145/1122971.1123001

[38] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172 – 185, 2006, special issue on synchronization and concurrency in object-oriented languages. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642306001171

[39] B. Hindman and D. Grossman, "Atomicity via source-to-source translation," in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC '06. New York, NY, USA: ACM, 2006, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178611

[40] G. Korl, N. Shavit, and P. Felber, "Noninvasive concurrency with java stm."

[41] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, “Unifying thread-level speculation and transactional memory,” in *Proceedings of the 13th International Middleware Conference*, ser. Middleware ’12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 187–207. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442639

[42] P. Di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano, “On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking,” *Perform. Eval.*, vol. 69, no. 5, pp. 187–205, May 2012. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2011.05.002

[43] A. Heindl and G. Pokam, “An analytic framework for performance modeling of software transactional memory,” *Comput. Netw.*, vol. 53, no. 8, pp. 1202–1214, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2009.02.006

[44] D. Didona, P. Romano, S. Peluso, and F. Quaglia, “Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids,” *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 11:1–11:32, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2620001

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

[46] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, “Time-based software transactional memory,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[47] T. Ye and S. Kalyanaraman, “A recursive random search algorithm for black-box optimization.”

[48] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford Transactional Applications for MultiProcessing,” in *IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[49] R. Guerraoui, M. Kapalka, and J. Vitek, “Stmbench7: A benchmark for software transactional memory,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273029