TÉCNICO LISBOA

A recommendation system approach to the tuning of Transactional Memory

André Alves Rogério Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Paolo Romano

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. Bruno Emanuel da Graça Martins

May 2017

To my parents, thank you for everything.

Acknowledgments

First and foremost I would like to thank my supervisor, Prof. Paolo Romano, for all the support in the realization of this dissertation. For all the hours spent discussing possible solutions and interpreting results, for the patience and the time that allowed me to write this dissertation.

Another person who helped a lot in doing this thesis was Diego Didona. Without you the dissertation would not have gone as smoothly, thank you very much. Thank you for all the time you spent explaining and debugging the code with me and for all the feedback throughout this dissertation. Thanks to you I got so familiar with ProteusTM code so fast and was able to do this dissertation.

To my parents, there is not enough words that can describe how thankful I am. Thank you for supporting me in everything in life and helping me achieve my goals and dreams.

Also, I want to thank all my friends that supported me through this challenge and university. I want to give a special thank you to Nuno Fangueiro, who has been with me through all of my academic experiences, whom I shared amazing experiences and projects throughout this six years. Another special thank you to Bernardo Rodrigues, Pedro Rosa and Miguel Costa, that stayed up with me all night in order to finish the dissertation.

Last but not least, I want to thank Sara Meneses. You have been one of my biggest pillars during my academic journey, believing in me every step of the way. Thank you for being part of my life, and being there when I most needed.

Abstract

Transactional Memory (TM) is a promising approach that simplifies parallel programming. However, in the broad spectrum of available TM implementations, there exists no one size fits all solution that can provide optimal performance across all possible workloads. This has motivated an intense research, over the last years, on the design of self-tuning solutions aimed at transparently adapting the choice and configuration of the TM run-time system.

This dissertation focuses on advancing a recent, state of the art solution in the area of TM self-tuning, called ProteusTM. ProteusTM builds on recommendation system and Bayesian analysis techniques in order to automate the tuning process of a TM system over a multi-dimensional configuration space — a unique feature in the literature on TM self-tuning.

In particular, this dissertation investigates two key research questions: i) how to extend ProteusTM to support sparse training sets, and ii) to what extent can the inclusion of workload characteristics (e.g., abort rate) enhance the accuracy achieved by ProteusTM's.

Keywords: parallel programming, transactional memory, self-tuning, ProteusTM, Machine Learning, recommender systems

Resumo

Memória Transacional (MT) é uma abordagem bastante promissora que simplifica drasticamente a computação paralela. Contudo, mesmo considerando o vasto espectro de implementações de MT disponíveis, não existe uma solução que seja claramente superior a todas as outras em diferentes cargas de trabalho. Ao longo dos últimos anos, este fato suscitou bastante interesse em investigar soluções auto-ajustáveis destinadas a adaptar de uma forma transparente a escolha e a configuração de um sistema de MT em execução.

O foco desta dissertação centra-se em melhorar uma solução recente, presente no estado da arte na área de MT auto-ajustável, nomeadamente ProteusTM. ProteusTM é uma solução baseada em sistemas de recomendação e em técnicas de análise bayesianas, com o objectivo de ajustar automaticamente um sistema de MT tendo em conta um espaço de configurações multidimensional - uma característica única na literatura de soluções auto-ajustáveis aplicadas a MT.

Em particular, esta dissertação investiga duas questões-chave: i) como estender a solução ProteusTM para suportar um conjunto de dados esparsos na fase de aprendizagem ii) até que ponto a inclusão de novas características referentes às cargas de trabalho (por exemplo, taxa de abortamento) aumenta a precisão alcançada pelo ProteusTM.

Palavras-chave: computação paralela, memória transacional, auto-ajustamento , ProteusTM, sistemas de recomendação

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multi-core architectures are nowadays ubiquitous, bringing parallel programming to the forefront of software development, with the objective of achieving better performance results. Hence, an important problem is how to make the processing capabilities made available by parallel multicore architectures easily accessible by the mass of programmer [1].

Parallel programming is very challenging. One of the most common problems is to synchronize multiple concurrent threads accessing shared resources. For many years lock-based systems were the standard approach for synchronization in concurrent applications. However, this approach has many known pitfalls. In fact, it is well known that coarse-grained locking is not scalable, whereas fine-grained locking is scalable, but suffers of subtle issues that can endanger both the correctness and actual performance of applications (e.g., deadlocks, live-locks).

Transactional Memory (TM) [1] is a promising approach to solve the problems of concurrent programming. TM offers a powerful abstraction to programmers, reducing the complexity of building parallel programming applications. With the TM abstraction programmers need only to specify *which* region of code to run atomically without worrying about *how* concurrent accesses should be synchronized (in contrast with locks) to guarantee correctness.

Over the years TM has gained a lot of attention [2, 3, 4], due to the maturing of the research and the release of the commercial processors providing hardware support for TM (Intel and IBM). Consequently numerous implementations were published, software-based, hardware-based and hybrid-approaches, providing evidence that TM is a promising solution.

However, in order to pave the ground for the adoption of TM as a mainstream paradigm for parallel programming, there is one crucial issue to address. Despite the numerous existing implementations, either in software or hardware, there exists no single solution that can provide optimal performance across all possible workloads [5, 6, 4]. Research has provided evidence that the choice of the best TM implementation is affected, in complex ways, by a number of workload characteristics [5, 4, 7]. In addition, performance can be affected by several factors, for instance programs inputs, phases of program execution, as well as the architectural aspects of the underlying hardware. Further, as heterogeneity of hardware/software architecture keeps on increasing, it appears quite unlikely that an “universal” solution

will be identified in the near future.

Due to the factors mentioned above, the programmer is left with the responsibility of determining the optimal TM application, which is not only a daunting task due to the vast TM space, may also be impossible to achieve using a single, static configuration (e.g., with time-varying workloads).

Self-Tuning reveals as an appealing solution to cope with heterogeneity in workloads and TM optimization, removing the burden of the programmer to identify the optimal configuration. Existing self-tuning techniques for TM systems [4, 5, 6] rely on modelling and forecasting techniques to optimize TM performance, , i.e., to identify the optimal configuration of one or more parameters controlling the behaviour of a TM algorithm based on the workload generated by some target application. Some example of black-box modelling techniques used in self-tuning systems for TM include Neural Networks [8], Decision Trees [9] and Collaborative Filtering [10].

Most of existing works using self-tuning to optimize TM performance aims to adapt dynamically either: (*i*) internal TM parameters, for instance number of retries when a transaction aborts [4]; (*ii*) the concurrency level of a TM, adapting the number of active threads [11, 12]; (*iii*) the choice of the TM algorithm to employ [6]. However, these solutions optimize the TM run-time along a single dimension, which is clearly unsatisfactory given the multitude of configuration choices/alternative implementations that exist.

To the best of my knowledge, the first and only multi-dimensional self-tuning solution for TM is *ProteusTM* [5], and it will be the main focus of this dissertation.

ProteusTM leverages on Collaborative Filtering and Baeysian Optimization [13] to identify the configuration that optimizes a given Key Performance Index (KPI) (e.g., throughput, execution time).

The goal of this dissertation is to investigate two research questions originated by ProteusTM:

1. One of the key techniques introduced by ProteusTM consists in an ad-hoc data normalization technique, called Rating Distillation. Rating Distillation addresses the problem of numerically manipulating KPIs of different applications, expressed in scales that broadly vary across different applications (e.g., throughput or execution time), in order to be used as input ratings for recommendation systems, based on, e.g., K Nearest Neighbours (KNN) [14] or Single-Value Decomposition (SVD) [15]), which assume that ratings are expressed in a uniform scale (e.g., a 5 star rating system).

   This normalization technique has been designed assuming the availability of full information regarding the performance of *every* application (included in the training set) across *every* possible of the available TM configurations.

   This is quite a relevant limitation, given that even in a small scale system the number of available TM configurations are on the order of hundreds and that the configuration space grow exponentially with its dimensionality (the well-known curse of dimensionality [16]).

   The first research question addressed by this dissertation deals precisely with this limitation of ProteusTM, proposing and systematically evaluating techniques aimed at supporting the use of sparse training information.

2. ProteusTM's training phase relies solely on the knowledge of the target KPI achieved by a work-

load when deployed over different set of configurations. In other words, it does not exploit any information regarding intrinsic workload characteristics (e.g., abort rate or transaction duration). Hence, a natural question, which is addressed in this dissertation, is to what extent can the accuracy of ProteusTM be improved by incorporating information on workload characteristics in its knowledge base.

We provide an answer to this question by considering different learners and normalization techniques.

This document is structured as follows. Chapter 2 provides a background on two main areas: TM and self-tuning. Chapter 3 describes ProteusTM in detail, and highlights the research directions we intend to study in this dissertation. Chapter 4 addresses the problem of how to support sparse training sets in ProteusTM. Chapter 5 studies the problem of incorporating workload information in ProteusTM's knowledge base. Finally, Chapter 6 concludes the dissertation by summarizing the results obtained in the previous chapters and discusses possible future work.

Chapter 2

Related Work

This chapter focus mainly on two topics: TM and Self-Tuning. Section 2.1 provides background on TM, by discussing the different trade-off's when designing a TM system. It includes insights on different types of TM systems, namely Software TM (STM), Hardware TM (HTM) and Hybrid TM (HyTM).

Section 2.2.2 reviews the state of the art on self-tuning of TM systems. Before presenting existing solutions, though, it first motivates the need for self-tuning in TM systems and provides background information on some key methodologies and Machine Learning (ML) algorithms that are often employed by this type of systems.

Concludes by detailing some existing solutions of self-tuning for TM.

2.1 Transactional Memory

TM is a promising concurrency-control mechanism that reduces the complexity of building parallel programming applications, relying on a powerful abstraction used for decades in the database community, namely *transactions* [1]. A transaction is atomic, in the sense that either all the various data accesses (read/write) of which it is composed are either executed as whole (commit), or none of them are (abort). Besides being atomic, a transaction also runs in isolation, which means that it appears to be executed as if it ran solo in the system, i.e., the effects of a transaction are not visible to the outside unless it commits (there were no conflicts). In practice, a TM system runs several transactions in parallel, and is up to the TM library to enforce isolation and atomicity via some concurrency-control algorithm.

TM reduces complexity by offering a powerful, yet intuitive, abstraction in which programmers need only to specify which code block they want to execute atomically, without worrying on how to guarantee atomicity [7, 1, 3]. In contrast, in other solutions like fine-grained locking, programmers have to specify how to synchronize concurrent accesses to shared data/memory.

As stated above, a TM system allows transactions to run in parallel and the problem of regulating concurrency is shifted from the programmers to the TM designers. The key mechanisms of TM design are data versioning and conflict detection. As transactions run in parallel we need to manage multiple versions of data, we can either use eager or lazy versioning. Eager versioning writes the new values

to memory immediately and saves the old values in an undo log. In case the transaction commits, no more actions are needed, if the transaction aborts we need to restore the old values from the undo log. With lazy versioning we store the new values in a buffer, and in case the transaction commits we have to write the new values to memory from the buffer. In contrast if it aborts no action is needed. This takes us to next key mechanism, conflict detection. A conflict occurs when two or more transactions access the same data and at least one issued a write operation. The conflict detection can be done while the transaction is running (pessimistic conflict detection) or during the commit phase (optimistic conflict detection).

The TM abstraction can be implemented in various ways, namely Software, Hardware and a combination thereof that is typically referred to as Hybrid TM. The following sections overview existing literature in each of the three areas above.

2.1.1 Software Transactional Memory

STM implements the TM abstraction entirely in software and unlike HTM (Section 2.3) needs software instrumentation of reads and writes memory accesses to trace conflicts between concurrent transactions. This instrumentation is typically inserted by a compiler, which transparently injects calls to allow the STM library to trace conflicts between concurrent transactions.

TM has attracted a lot of research activity and throughout the last decade STM systems have been extremely researched as a solution to replace lock based approaches. More recent studies also focus on STM as a fall-back path to HTM, when hardware resources are insufficient/incapable of successfully executing some transaction. Consequently these researches produced a variety of STM implementations (see [3, 2]), that differ in the following aspects [2]:

1. Conflict detection: eager/lazy/mixed, invisible readers/visible readers.
2. Buffering mechanism: redo log/undo log/cloning.
3. Meta-data organization: the data granularity either object-based/word-based/ownership records.
4. Contention management strategy, e.g., if a conflict is detected early either stall one of the transactions or abort one of them and retry later.
5. Validation Strategy: timestamps-based/lock-based.
6. Progress guarantees: non-blocking, livelock-free, starvation-free.

Most recent studies (e.g., [4, 7, 17]) affirm that there is no ”one-size-fits-all” solution that will outperform clearly all the other solutions, since TM performance is influenced by design decisions (e.g., examples above) and by the workloads they are executed on (e.g., read-intensive, write-intensive).

In the next subsections I will analyse and explain two different STM implementations, TinySTM [3], and Norec [2]. TinySTM [3] a word-based and time-based high-performance implementation, and Norec [2] optimizes STM performance at low thread counts.

2.1.1.1 TinySTM

TinySTM [3] is a lightweight word-based STM implementation, with a time-based design and a lock based implementation to protect shared memory locations. The authors of TinySTM opted to use an encounter-time locking (eager) because, as stated in the paper:

- Empirical observations indicate that an early detection of conflicts increases the transaction throughput, unlike in commit-time locking, that detects conflicts at commit time and cannot be resolved in other way than aborting some transaction.
- Encounter-time locking allowed the authors to efficiently handle read-after-writes.

Additionally two strategies for accesses of memory were implemented: *write-through* access and *write-back* access. With write-through access, transactions will write directly to memory and in case of abort need to revert their actions using an undo log. Write-back access, transactions write into a buffer and in case of commit write to memory.

TinySTM basic algorithm, like most word-based algorithms, relies on a shared array of locks, in which each lock covers part of the address space, based on an hash function. The locks serves as an ownership record, where the less significant bit of a lock marks if the lock is owned (0 or 1). Further, if the lock is not owned the remaining bits are used for the version number, that corresponds to the *commit timestamps* that last wrote to the memory location. In case the lock is owned, the algorithm varies depending on the strategy for accesses of memory: for write-through the last remaining bits are the address of the transaction that owns the lock; for write-back access the last remaining bits correspond to an entry in the write-set of the owner transaction. The version number is a shared counter that increments every time a transaction acquires the lock.

As mentioned it uses eager conflict detection, and to guarantee consistency uses the LSA [18] algorithm. LSA provides extendable timestamps, that avoid false positives. Basically if a transaction reads a value and it finds a more recent version, outside the validity range of its snapshot, it tries to construct a consistent snapshot by extending its own to the most current version by verifying if its read set is valid [18, 19].

To conclude TinySTM has proven to have good scalability and to have high performance on read intensive workloads. The authors also provide a dynamic tuning strategy for the granularity of the internal locks it uses, which further enhances the STM performance.

2.1.1.2 NOrec

NOrec is an ownership-record free STM known for the extremely low overhead at low thread counts, clean semantics and satisfactory scalability. This STM focuses on three key ideas:

1. Single global sequence lock based on TML [20];
2. An indexed write-set, as in previous work done by the authors [21];
3. Value based conflict detection.

NOrec uses a single commit writer protocol, where the single global lock is used to coordinate write transactions and for validation purposes. A single global sequence lock allows invisible readers, which means that read operations do not incur the overhead related with updating the lock. This type of protocol (used by TML) can become a source of overhead for writer intensive transactions, so NOrec unlike TML uses lazy conflict detection, in which transactions do not attempt to acquire the sequence lock until commit time (less time holding the lock). This design choice increases the time speculative writers/readers can run concurrently as well as the commit probability of read-only transactions. Using a lazy strategy the STM needs a redo log to buffer writes before writing them to memory (indexed write-set).

Another extension to avoid using a conservative implementation of reads and decreasing scalability is value-based validation. With value-based validation a transaction logs the location and the value of the *read operation*, to be used for later validation. Validation happens at lock acquisition or when a transaction reads a value and finds the global lock to have increased. In this case, it re-reads the addresses in the read-set and checks if the value is the same, or has a more recent value (invalid).

The lightweight design of NOrec makes it an appealing solution for low number of threads, as at high threads count the global lock risks becoming a bottleneck if there is a non-negligible probability of running update transactions. Also, due to its simplicity, it lends itself as a promising approach to be used as the fall-back STM for Hybrid TMs.

2.1.2 Hardware Transactional Memory

HTM is a *hardware* support for TM-based programming. HTM was the first approach to TM and lacked support at the time, actually the first proposed TM implementation [22] was back in early 90s, although the first implementations in real processors started to appear only in recent years. Indeed after the recent integration of HTM supports in processors by Intel and IBM and the maturing of TM, HTM attracted a lot of research.

HTM is potentially a very efficient solution, since it avoids the overheads of STM instrumentation. Currently available HTM implementations exploit the processor's cache coherency mechanism: they keep track of memory loads and stores in the cache or cache-like structures [23] for conflict detection. While this choice simplifies their design, it also imposes some relevant restrictions: since HTM are limited to their cache capacity, long running transactions can abort frequently, due to cache capacity exceeded. Even with its current limitations, HTM has exceptional performance on workloads characterized by small transactions, for example as a synchronization primitive for concurrent data structures.

One fundamental design in most of HTM implementations is its best-effort nature, which gives no guarantees on whether transactions will ever commit, even in the absence of conflicts [4, 23]. Due to this design choice, there is a need for a fall-back software solution mechanism, typically a lock or a STM [4, 2].

Although current HTM implementations do share some implementation approaches, there are some major differences in: cache detection granularity, cache geometry and abort reasons. Table 2.1 provides

Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	Power8
Conflict-detection granularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Transactional-load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
Kinds of abort reasons	-	14	6	11

Table 2.1: HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8 [23]

examples of different HTM implementations. And, just like mentioned when discussing STM systems in Section 2.1.1, most studies that analysed HTM's performance (e.g., [6, 7]) reached a consensus that there is no "one-size-fits-all" implementation. Different implementations work better in different workloads.

2.1.3 Hybrid Transactional Memory

HyTM uses both HTM and STM, in which normally, running with HTM is first attempted, falling back to STM to handle situations where the HTM could not execute the transaction successfully. This type of TM however, needs to be carefully implemented, so that both (HTM and STM) preserve correctness and transactional semantics when integrated.

Hybrid models require hardware and software transactions to co-exist, and most implementations monitor common memory location between transactions, to help in conflict detection. However, most mechanisms that are currently adopted to support this co-existence induce high overhead on the TM, which often leads to unsatisfactory performance [4, 24].

Some researchers believe that HyTM has the potential to overcome its current limitations [4, 24, 25] and achieve better results.

As stated in STM section, NOrec [2] authors believed that this STM could be used as a fall-back STM and they implemented the Hybrid NOrec that will be explained more in detail.

2.1.3.1 Hybrid NOrec

The goal of Hybrid NOrec [25] is to develop a novel HyTM algorithm that supports the execution of hardware and software transactions without the per-access instrumentation overhead on HTMs.

Hybrid NORec is probably one of the first proposed HyTMs, as well as one of the most popular. It has been designed to operate with different HTM implementations, including the ones provided by AMD's ASF [26], Sun's Rock [27], as well as IBM's P8 and Intel's TSX. Hybrid NOrec uses the NOrec STM as the fall-back path for HTM, appropriately adapted to ensure correct coupling with the HTM-based execution path. To extend NOrec's single-writer commit protocol, it was necessary to produce a design where HTM needs only to monitor one location.

NOrec original *global lock* is retained and used for synchronization between hardware transactions

and software transactions. In Hybrid NOrec the HTM reads the global lock right after the start of the HTM transaction, before any memory access is performed allowing the HTM to subscribe to STM — a technique also called eager lock subscription [28]. In addition they added a second lock, so that a HTM transaction can signal its commit to STM triggering validation on the last.

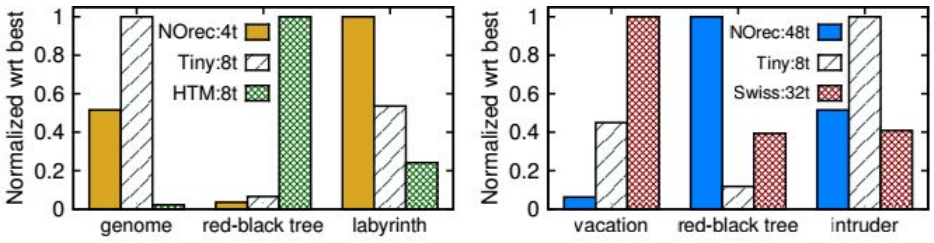
Software Transactions executes as normal (in NOrec), but at commit time tries to acquire both locks (2-Location algorithm [25]) and after write-back restores the locks. A hardware transaction starts by reading the global lock, and if it is not available it spins, guaranteeing that no HTM can commit when a software transactions write-back is being executed. Before commit, a hardware transaction reads and increments the second lock. The value-based validation of NOrec, enables short-running HTM to commit without forcing non-conflicting software transactions to abort.

Hybrid NOrec is a HyTM with low overhead on hardware transactions, that shows significant potential for Hybrid TM's and point out several challenges to faster reach that potential: HTM performance and STM scalability.

2.2 Self-Tuning for TM

Transactional Memory has several appealing characteristics, and, as mentioned, has attracted increasing attention in the literature, leading to the publication of a large number of works (e.g., [3, 2, 19, 23]). One of the key results from all the existing research is that, independently of the synchronization scheme adopted by different TMs, performance is strongly workload dependent and affected by complex factors, such as duration of transactions, level of data contention, ratio of reads/writes.

Existing literature in the area seems to have reached consensus on the fact that there is no "one-size-fit-all" solution that can provide optimal performance across all possible workloads. In fact the best TM for a given workload can become the worst for another one.



(a) Throughput/Joule on a single-chip 8-core CPU (Machine A in Table 2).

(b) Throughput on a multi-chip 48-core CPU (Machine B in Table 2).

Figure 2.1: Performance heterogeneity in TM applications[5]

Figure 2.1 provides experimental evidence of this fact. The study uses different architectures and metrics. The left graph reports the energy efficiency (Throughpust/Joule) of 3 different TM systems, using different TM implementations and internal configurations, when running 3 different applications. The right graph focuses instead on the throughput achievable by the same set of TM configurations when deployed on a different architecture and considering a different set of benchmarks. By the data

in the plots, we clearly see that, in different workloads, different TM implementations achieve the best performance results.

Self tuning represents a very appealing solution to tackle this issue. Generally speaking, self-tuning uses performance modelling and forecasting techniques to optimize, depending on the workload and TM, different parameters. There are two key design choices at the bases of a self-tuning system [29]:

1. When to trigger adaptation?
2. How to decide which adaptation should be triggered?

There are two main approaches on *when* to trigger the adaptation, *react* to workload changes (reactive) or *anticipate* them (proactive) [29]. Following this, to accurately apply one of this approaches, self tuning systems have to able to robustly distinguish workload changes from transient fluctuations that are not statistically meaningful, e.g., due to measurement noise in throughput.

Reactive schemes evaluate the needs for reconfiguration based on the current workload, unlike proactive schemes where the strategies attempt to anticipate the need for reconfiguration using predictive techniques. A reactive approach tracks the variation of the current workloads, allowing a promptly reaction to abrupt workload changes, but can have sub-optimal performance in transitory stages. In contrast, proactive schemes by predicting the workload needs of reconfiguration, it can adapt before the workload change reducing the time a sub-optimal configuration is used. The two schemes have complementary pros and cons, which has also motivated the design of hybrid schemes that use both approaches in combination.

When workload changes are detected or predicted, self tuning needs to decide *which* adaptation to trigger. Following the taxonomy presented by Didona et al. [29], the identification of optimal configuration can be performed by the mean of models, which can be classified into: white box, black box and grey box (hybrid of black and white) techniques.

- **White Box Modelling** [30], leverages on available knowledge of the internal dynamics of a system and/or application, and with it builds an analytical model to capture how the system's configurations and workloads parameters can affect performance. Analytical models do not normally need a training phase, in contrast with black box models. This approach normally relies on approximations and assumptions to derive a treatable model of system's performance, and thus achieving good accuracy can be challenging in certain scenarios. The main reason is the fact that inaccuracies are not amendable in analytical methods, since the system is characterised by means of equations and these are immutable.

- **Black Box Modelling** [11, 31], in contrast with white box modelling, does not require any knowledge about the system/application internal dynamics. This model relies on a training phase, during which, based on the observation of the system's actual behaviour under different configurations and subject to different workloads, infers a statistical performance model via different Machine Learning (ML) techniques. Normally using this approach one achieves great accuracy for scenarios close to the ones observed in the training set. Instead, accuracy of ML is typically poor in

regions of the parameters not sufficiently sampled during the training phase. The main drawback of this approach is that the number of configurations can grow exponentially with the number of input parameters of the model.

- **Grey Box Modelling** [12], employs both white and black box methodologies, creating an hybrid model that inherits the best features from both approaches: on the one hand, good accuracy for unseen configurations/workloads and minimal training time (white box), and, on the other hand, the robust and increasingly great accuracy by periodic training (black box).

One can yet sub-categorize these models: black and grey box models have training phases that can be performed, either *off-line* or *on-line*. Off-line learning is normally built by using Supervised Learning (SL) [8, 32], where the ML algorithm is trained over a training set. The off-line training set is assumed to be available to the learning algorithm, and it is up to this algorithm to output a model based on it. When confronted with a new value, not present in the training set, the model constructed during the training phase can be queried to predict which adaptation to use (e.g., using classification techniques [33, 34]).

On-line learning can use SL algorithms, updating a model by incrementally considering every new available data. This approach normally requires less computational power compared to off-line techniques, but can have a lower accuracy. Another way to approach on-line learning is by using Reinforcement Learning (RL) [35, 36] that uses concepts like *exploration* of untested actions for a new workload and *exploitation* of available and incomplete knowledge, to achieve the optimal configurations. RL techniques strive to maximize some notion of cumulative reward (e.g., throughput) when presented with a new state (e.g., workload) based on the data already gathered [29].

In the remainder of this section we will address in more detail some of the machine learning algorithms presented. Firstly, a brief overview about: SL and RL. Next, we explain with some detail Collaborative Filtering (CF) and different approaches to tackle this problem as well as present different self-tuning solution for TM.

2.2.1 Main Machine Learning Techniques for self-tuning of TM

We overview the main ML techniques that are relevant for this thesis, either because they have already been used in the literature on TM self-tuning or because we plan to exploit it in this dissertation.

2.2.1.1 Supervised Learning

SL [8, 32] is very commonly used in black-box methodologies, where the algorithm is used to update a model incrementally with arrival of new data. In more detail, the training algorithm objective is to infer a function, also called model (ϕ), over a training set where X is the input space and Y the output space:

$$\phi : X \rightarrow Y \tag{2.1}$$

Note that to build this model, the training algorithm is provided with training examples which are normally constituted by pairs $\{(x_1, y_1), ..., (x_N, y_N)\}$ of labelled features, where for any input $x_i \in X$ the

output $y_i = f(x_i) : y_i \in Y$ is known.

Finally the model ϕ, which is an approximation of the function $f(x)$, optimally when it receives a new feature x_i possibly not present in the training set, determines correctly the output $\hat{y} \in Y$.

Regarding SL we can distinguish between two type of problems, *classification* and *regression*, differing in the way they map the input and output. For classification the co-domain Y is a discrete set normally called classes or continuous space, while in regression the co-domain Y is a continuous set.

2.2.1.2 Reinforcement Learning

RL takes a different approach: unlike in SL, where all the available knowledge is assumed to be made available to the learner upfront, RL techniques aim at achieving some target goal by learning through several iterations of trial-and-error in the system environment. There are three fundamental parts of a RL problem: the environment, the reinforcement function, and the value function [35].

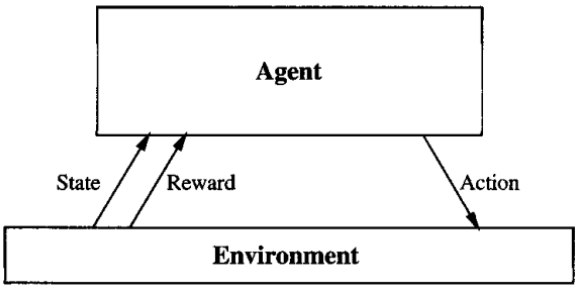


Figure 2.2: Example of RL interactions [9]

Figure 2.2 describes a general approach and set of interactions that the RL algorithm uses to build its model. In standard RL the *agent* is connected to its *environment*. The environment has to be at least partially observable by the RL systems, so that the agent can observe the *states* of the environment, either with sensors or symbolic descriptions [35]. An agent interacts with its environment through *actions*, causing an alteration in some state of the environment.

The ”goal” of RL is defined through a *reinforcement function/reward function* that normally assigns a numerical value - immediate payoff - to each distinct action the agent may take from each distinct state. In other words, it maps states and actions to reinforcements (normally called reward in the form of a scalar value). Summarizing, the agent learns to perform actions that will maximize the sum of the reinforcement received since the initial state to the last.

Until this point we described the interaction of an agent with its environment as a sequence of actions, receiving a reward of realizing that action in a determined state. The value function tackles the problem of distinguishing between ”good” actions. Associated with the value function comes two terms: policy and value. A policy is what determines what choices to take in different states in order to maximize the reward function. The value of a state is determined by the sum of the reinforcements received in the current state to a terminal state. In summary, the value function is a mapping from state to state.

One of the major problems in RL is that certain actions cannot be represented as good actions or bad actions. One of the most-used techniques to solve this problem is *dynamic programming* [37].

To finalize this topic, which only overviews how a standard model of RL is built, it is important to mention that several algorithms have to make a trade-off between exploration and exploitation. Exploration refers to the act of performing new actions in a state in order to learn its reward, while exploitation focus on the knowledge already gathered with high reward to make a decision [9].

2.2.1.3 Collaborative Filtering

Everyday we are inundated with choices and decisions such as: ”Where will I lunch today?”, ”What movies/series should I watch?”, ”What countries to visit?”. The size of the available options nowadays is enormous, so people normally rely on recommendations of other people or known web sites to make simple choices that we are confronted with daily.

There has been increasingly more investigation on how to automatically give recommendations to people, one of the most popular examples is Amazon which has been giving automatic recommendations to its users since the late 1990s [14] or the Netflix Prize [38]. This type of systems are called Recommendation Systems (RS), systems that seek to predict the rating a user would give to an item, and exploit these recommendations to recommend items of interest to users [5].

One of the most popular techniques in the recommendation systems literature is CF algorithms (e.g., [34, 33, 14]).

One of the fundamental assumptions of CF is that if user X and Y rate the same items similarly, they will rate or recommend other items similarly [15]. CF systems uses a database of preferences, i.e., the information domain consists of a list of users that have expressed their preferences for various items. This preferences are normally called ratings and the interactions can be represented in the form of a triple (User, Item, Rating). Ratings can take a lot of forms, one very recent example of *explicit information* is Netflix which used integer-valued rating scaled (0 - 5 stars), but have recently updated to a system of like and dislikes similar to the one used by Facebook in photos. Another form is *implicit information* which are inferred by the systems such as purchases performed online by users or clicks.

The set of all the triples constitutes an Utility Matrix (UM), the most usual representation of a data set in CF, where normally we represent the user in the rows of the matrix, the various items in the columns and each cell is the rating of each user for the respective item. To better visualize the problem, bellow we present Table 2.2 a small example of an UM in a movie recommendation scenario:

	Lord of the Rings	Batman begins	Titanic	The Hobbit
Alice	4	?	3	5
Bob	?	5	4	?
Jeff	5	4	2	?

Table 2.2: Small sample of an utility matrix, using a 0 to 5 star scale.

This small example is constituted by 3 users and 4 movies, where the symbol $?$ represents unknown values, i.e., the user has not yet made a recommendation for that movie. When using CF techniques the main focus is to *predict* the ratings a user would give to an item, take the sample matrix in Table 2.2

again as an example, the task would be to fill the missing values in the matrix. After the prediction, we *recommend* the best n-items based on a list of predictions for the user.

CF systems have many challenges and need to possess certain characteristics, since they are applied in challenging scenarios. This type of systems must be able to [15]:

- Deal with highly sparse data.
- Scale with respects to the increasing number of users and items.
- Give satisfactory recommendations in a short time.

There are several collaborative filtering methods that try to tackle this challenges [15, 33, 14], we will address some approaches which are used by the main focus of this thesis, ProteusTM [5]. In the following we will overview some of these approaches, as they are closely related with the work carried out in this dissertation.

K Nearest Neighbours

KNN is considered one of the most intuitive among the several CF algorithms, it tries to predict the user preferences based on users that share similar interests. One of the most critical steps of this type of algorithm is the *similitarity function*, $sim(u, v)$, that identifies similarities between users or items [14]. The input variables of a similarity function are two vectors, u and v, both with n elements. In order to compute $sim(u, v)$, only the shared elements, $S = \{i \in u \cap v\}$, by both vectors are considered. In Table 2.2 if vector u and v correspond to the rows of the users Alice and Jeff respectively, the values of i would be Lord of the Rings and Titanic.

Some of the most popular similarity functions are: Pearson Correlation and Cosine. The Pearson Correlation function measures the extent to which two variables relate with each other:

$$sim(u,v) = \frac{\sum_{i \in S} (r_{u,i} - \overline{r_u})(r_{v,i} - \overline{r_v})}{\sqrt{\sum_{i \in S} (r_{u,i} - \overline{r_u})^2} \sqrt{\sum_{i \in S} (v_{u,i} - \overline{v_u})^2}}, \tag{2.2}$$

where i refers to the identifier of an element present in both vectors, $\overline{r_u}$ is the average rating of the co-rated items in vector u and $r_{u,i}$ a known rating for user u and item i.

The Cosine based similarity can be measured by computing the cosine distance of the two vectors, obtained by computing the scalar product of the two vectors and normalize by the product of the their norm:

$$sim(u,v) = \frac{\sum_{i \in S} (r_{u,i}\, r_{v,i})}{\sqrt{\sum_{i \in S} (r_{u,i})^2} \sqrt{\sum_{i \in S} (v_{u,i})^2}}, \tag{2.3}$$

As mentioned to compute the similarity function KNN can either search based on the user (user-based KNN) or on items (item-based KNN). Denote a domain of a set of U users and a set of I items.

User-based KNN first computes the neighbourhood $N \subset U$ composed of K elements which are the most similar users to user u (active user). After identifying the set of neighbours N, all that is left is to generate the rating predictions for user u and item i, which are normally obtained by computing the

weighted average of the neighbouring user's ratings i using similarity as the weights. Denote $p_{u,i}$ as the value of the prediction:

$$p_{u,i} = \overline{r_{u,i}} + \frac{\sum_{u' \in N} sim(u,u')(r'_{u,i} - \overline{r'_{u,i}})}{\sum_{u' \in N} sim(u,u')}, \tag{2.4}$$

Item-based KNN contrary to the user-based approach instead of using the users rating for the prediction it uses the similarities between rating patterns of items. Similar to the user-based KNN this approach also starts by finding a set of K neighbours, but now it searches for the K rating vectors $C \subset I$ which are similar to the target item of the prediction. After gathering subset C, the prediction is computed:

$$p_{u,i} = \frac{\sum_{j \in C} sim(i,j)(r_{u,j})}{\sum_{j \in C} sim(i,j)}, \tag{2.5}$$

Matrix Factorization

Matrix Factorization (MF) is another method that can be used to tackle the CF challenges. Matrix Factorization techniques characterize both users and items by vectors of factors inferred from rating patterns, that are hidden in the UM. The formalization of this technique is done by mapping both users and items to a joint latent vector space of dimensionality f [10], where the user-item interactions are modelled as inner products in that space. Intuitively matrix factorization tries to find a latent factor that determines how a user rates an item, e.g., in a movie scenario the possible discovered factors might measure if the movie is a drama, comedy, horror or even how much a user likes comedy movies or certain actors/actresses.

Matrix factorization associates each item i with a vector $q_i \in \mathbb{R}^f$ and the each user u with a vector $p_u \in \mathbb{R}^f$. The elements vector q_i represents how much item i possesses those factors and p_u measures the interests of a user in items on the corresponding factor. Given this formalization the approximation/prediction, $\hat{r}_{u,i}$ is given by:

$$\hat{r}_{u,i} = q_i^T p_u \tag{2.6}$$

Computing the latent factors that map q_i and p_u, is one of the biggest challenges in MF.

Single-Value Decompostion (SVD) is one of the most renowned methods to identify and extract latent factors. MF tries to computes an approximation R similar to the original UM $M^{m \times n}$, with number of rows (users) $m = |U|$ and $n = |I|$ columns (items):

$$R = Q^T P \approx M \tag{2.7}$$

Note that we would like to discover f latent features, so matrix P is a $m \times f$ matrix and Q is a $n \times f$ matrix. In this way, the rows of matrix P represents the users interest in each of the f features. Similarly, each row of matrix Q would represent the items relevance for each feature.

In practice factoring the UM only using SVD raises difficulties since this technique is undefined when the matrix is sparse or incomplete, which is often the case of matrix M. So in order to compute matrix

R based only on the known ratings, we need to use an auxiliary method that fills in the missing values of the matrix.

One of the most popular methods is the Stochastic Gradient Descent (SGD) [39], which reaches R by iteratively adjusting the values of elements in P and Q so as to minimize the square fitting error of R in respect to M. With the computation of R we can estimate the ratings using Equation 2.6.

2.2.1.4 LibFM: Factorization Machine

LibFM [40] is a software implementation for Factorization Machines (FMs) that features different learning algorithms such as SGD, Alternating least-Squares (ALS) and Markov Chain Monte Carlo (MCMC). FMs is a generic approach since it is capable of mimicking most of the existing factorization models (e.g. SVD) by feature engineering. This new approach is capable of supporting any real-value based feature vector, while able to give accurate prediction in sparse scenarios, since it models all variable interactions with factorized parametrization.

As mentioned, LibFM uses a different representation - with respect to the ones presented until now - of the data set by means of features vectors, also know as feature engineering. Feature Engineering is a well known pre-processing step in ML literature [41] that provides the ability to support any real-value based data set, i.e., FMs are not limited to use uniform ratings, as for the case of SVD. In order to illustrate how FMs work, let us start by illustrating how one can encode the information in a conventional UM using a FM-based approach. Denote a matrix $M \in \mathbb{R}^{u \times p}$, whose i-th row, noted x_i, encodes a feature vector compose by p real-valued variables, and is associated with a target prediction, note y_i. Briefly, a data set would be described as a set of tuples(x, y), as we can see in Figure 2.3.

Figure 2.3: Example for representing a recommender problem with real valued feature vectors $\mathbf{x}$. Every row represents a feature vector $\mathbf{x_i}$ with its corresponding target $\mathbf{y_i}$. For easier interpretation, the features are grouped into indicators for the active user (blue), active item (red), other movies rated by the same user (orange), the time in months (green), and the last movie rated (brown) [41, 40].

Converting a typical UM into a feature vector space we could consider only the information included in the User, Movie subsets (blue and red) of Fig. 2.3, as well as the target y. Denote U as the set of users, and I the items. The blue set (User) encodes a total of $|U|$ different users, each encoded via a unique binary identifier. Similarly the red set (Movie), encodes $|I|$ different items, each associated with a unique identifier. The target y corresponds to the rating given by user $u \in U$ to item $i \in I$. Assume

as well the triple notation referenced in Section 2.2.1.3., an example of translation based on Figure 2.3, focusing only in the blue and red indicator , would be: x_1 = (UserA, TI, 5); x_2 = (UserA, NH, 3); x_2 = (UserA, SW, 1); x_4 = (UserB, SW , 4); (...).

One of the reasons that make libFM relevant for this thesis is the fact that it supports implicit indicators in different scales such as other movies rating (yellow indicator), or even the time passed since the rating of the item (green indicator).

The application of FMs for prediction tasks starts by building a model that captures all the nested interactions up to order d between the p input variables in x. The factorization model for $d = 2$ (value used in the software tool) is:

$$\hat{y}(x) = w_0 + \sum_{j=1}^{p} w_j x_j + \sum_{j=1}^{p} \sum_{j'=j+1}^{p} x_j x_{j'} \sum_{f=1}^{k} v_{j,f}\, v_{j',f} \tag{2.8}$$

where k is an hyper-parameter that defines the dimensionality of the factorization and the model parameters to be estimated are:

$$w_0 \in \mathbb{R}, w_j \in \mathbb{R}^p, V = \{v_{1,1}, ..., v_{p,k}\} \in \mathbb{R}^{p \times k} \tag{2.9}$$

Considering equation 2.8 [40]:

- w_0 is the global bias
- w_j represents the strength of the j-th variable
- the two nested sums represent all the pairwise interactions of the input variables, $x_j x_{j'}$.
- $\sum_{f=1}^{k} v_{j,f}\, v_{j',f}$ is the factorized parametrization that allows this approach to estimate reliable model parameters even with sparse data, by attributing a low rank to the pairwise interactions.

To compute the model and learn the model parameters libFM uses the three learning algorithms mentioned above: SGD, ALS and MCMC. Concluding FMs most appealing characteristics are:

- Allow a reliable estimation of the model parameters under sparse data, with parametric factorization.
- Linear complexity, by factorizing the pairwise interactions.
- FMs is a general predictor that works with any real-valued feature vector, that supports multiple input variables.

2.2.2 Review of existing solutions for self-tuning of TM systems

In the next sub-sections I will present some insights on some Self-Tuning solutions for TM systems, based on different methodologies. Recent work using self-tuning to optimize TM performance has focused on the problem of dynamically adapting: (*i*) internal TM parameters, for instance number of retries when a transaction aborts [4]; (*ii*) optimize the concurrency of a TM, adapting the number of active threads [11]; (*iii*) switch the concurrency control algorithm used by the TM [6].

In the following I will overview some recent works in this area.

2.2.2.1 Tuner

Tuner [4] is a self-tuning solution for Intel TSX (HTM), which automatically tunes TSX software fall-back path. Tuner optimizes Intel TSX performance in presence of heterogeneous workloads, by using RL techniques to dynamically configure a relevant *TM parameter* without any a-priori knowledge of the application. Specifically, the HTM parameter optimized by Tuner is the maximum number of attempts for a transaction to be executed in hardware.

Some papers [42, 43] report that setting the number of attempts to 5 gives an all round solution, but this is a sub-optimal solution since it does not consider workload heterogeneity. Tuner focus on two problems:

- How many times should a transaction retry/attempt in hardware?
- How to optimize retries that the hardware provides information on the nature (capacity vs conflict induced) of a transaction abort error code?

Tuner adopts an on-line approach since it fits better with irregular applications, unlike off-line approaches, which spare from the cost of gathering an initial training set representative of the target architecture and application to be used. Tuner uses a combination of algorithms borrowed from the RL literature, namely Upper Confidence Bounds (UCB) [44] for adapting the capacity abort management and Gradient Descendent Exploration (GRAD) for adapting the number of attempts for a transaction.

The UCB algorithm is an efficient solution for the bandit problem, a well-known problem in the RL literature, in which an agent is faced with a bandit (slot machine) with k arms each one with a unknown reward distribution. The agent iteratively plays one arm, gets the associated reward and tries to maximize the average reward. UCB creates an over-estimation of each possible decision and lowers this estimation for each sample drawn.

UCB was integrated with TSX, by encapsulating each atomic block with an UCB instance containing a slot machine, this way UCB will try to maximize the reward for each arm, where an arm represents actions when faced with a capacity abort (give up, half the attempts, decrease only one attempt). Gradient Descent Exploration is similar to hill climbing techniques, and since it scales better than UCB for large search spaces it was chosen - as the technique to optimize the overall number of attempts.

The final algorithm, uses both techniques at the same time and consequently there are scenarios when both techniques will not converge to the same result, resulting in a ”ping-pong” effect, so the authors decided to implement a hierarchy between the techniques, where UCB can force GRAD to explore in the direction predicted by UCB to be more promising.

Tuner was integrated in GCC compilers, achieving total transparency for the programmer. The author’s evaluation shows that Tuner shows consistent gains to static solutions, using RL techniques.

2.2.2.2 SAC-STM

Self-Adjusting Concurrency STM (SAC-STM) [11] as the name suggests tunes the *concurrency level* for an application, i.e., controls and optimizes the level of parallelism. The problem with tuning concurrency level is the trade-off between more parallelism and data conflict, since increasing the number of threads can possibly increase the conflict ratio of transactions (more aborts/retries), harming performance.

SAC-STM is a black box approach, that relies on TinySTM [3] (Section 2.1.1.1) for the STM layer without the dynamic adaptation already included, and uses Neural Networks [9] (NN) to predict the optimum level of parallelism. It is composed by:

- A Statistics Collector(SC);
- A Neural Network (NN);
- A Control algorithm (CA);

In a periodic sampling interval the *Statistics Collector* estimates statistical parameters: read/write-set sizes (rs_{size}/ws_{size}), the average execution time for commited transactions (t_{time}) and the average time of non-transactional code blocks (ntc_{time}). In addition, it estimates the probability of an object read by a transaction is also written by other concurrent transaction (rw_{aff}), and the probability of a object written by a transaction being written by a concurrent one(ww_{aff}) [11].

The Controller receives the statistical sampling and exploits Neural Networks to predict, the wasted time that will characterize the application execution in a near future ($wtime$) using *n* threads.

Neural Network is a ML technique, composed by processing elements that compute a function through approximations and exploitation of a training set. In this case, it calculates the same statistics of the SC based on a training set, so that it can infer a function between all the statistical data, the number of threads and the $wtime$.

After NN predicts the values of $wtime$ for all possible number of threads, the Controller chooses the number of threads *m* that should be active, where *m* is equal to the value of *i* that maximizes the function:

$$\frac{i}{w_{time,i} + t_{time,i} + ntc_{time}} \tag{2.10}$$

Note that *i* represents the active number of threads, and the denominator of the equation represents the *predicted* average execution time between the commit operations of two consecutive transactions along a given thread with *i* active threads [11]. Lastly the CA configures the application to use *m* number of threads.

SAC-STM shows that using self-tuning the concurrency level in the context of TM can accomplish promising results. Using a learning-based solution to address the problem of adapting the number of concurrent threads running in a TM can achieve almost optimal results. Actually, the effectiveness of the methodology at the basis of SAC-STM has also been confirmed, more recently, in the context of HTM [45].

2.2.2.3 Automatic Performance Tuning

The work by Qingping, et al. [6] highlights the fact that there are numerous different STM implementations, with none being able to clearly outperform another, which poses as an obstacle to the acceptance of transactional memory. The different implementations provide good performance for certain workloads and architectures, so this paper introduces methods that dynamically select the best STM algorithm based on static analysis and dynamic profiling.

The proposed solution is a framework that allows to dynamically pick the best STM algorithm, using an off-line training phase and a dynamic (run-time) profiling with the help of a custom STM implementation, ProfileTM.

The off-line training phase, firstly executes a set of micro-benchmarks, and it then uses an ad-hoc built STM, called ProfileTM, to gather some dynamic characteristics and perform static analysis. All the data is then given to the ML training policy which produces the adaptivity policy.

This work uses four triggers to activate the adaptivity framework:

1. Number of consecutive aborts exceeds a defined threshold;
2. Long delays when attempting to begin a transaction;
3. Thread creation and destruction;
4. Totals commits below the threshold.

On every trigger, the system blocks the start of new transactions and waits for the active transactions to either abort or commit. Afterwards switches to ProfileTM and runs N transactions, one at the time, updating the dynamic profile. The system then outputs the new algorithm based on the adaptivity policies and the profile.

One interesting feature of this frameworks - is how the authors handle repeated recommendations when the abort ratio is high. Basically, there are some workloads that behave the best in algorithms that admits frequent aborts (multiple consecutive aborts, trigger 1), and changing the algorithm blindly can hinder performance. To cope with this problem, the framework saves the total number of commits and aborts, and upon the next trigger if the same recommendation/algorithm is given, the protocol only accepts the recommendation if there was forward progress [6] (e.g., more commits), and doubles the abort threshold.

It is important to refer that the adaptivity policies can have three approaches: (*i*) expert policies, (*ii*) completely automate ML system or (*iii*) collaborations between programmers and ML. Expert Policies are written by the programmer to satisfy arbitrary requirements, while ML-based policies create a policy based on ML techniques, such as NN.

To conclude the insight of this paper, it is important to refer that this was the first ML-based adaptivity system for synchronizing parallel programs. The best performance was reached by using expert knowledge and machine learning. It shows that the combination of performance, maintainability and flexibility in ML-systems make them an appealing approach [6].

This review highlights the fact that Self-Tuning over the years has emerged as an attractive technique to improve performance in Transactional Memory systems. Instead of producing more and more TM implementations, self-tuning solutions achieve performance close to the optimal configuration, and gain when compared to static implementations. Among the presented solutions, none provides a comprehensive solution capable of optimizing a TM system across the multiple configuration dimensions that such systems support.

Most of the existing solutions, e.g., [6, 4, 11], only work in one-dimension, which is summarized in Table 2.3. This suggests that self-tuning should follow a holistic approach, seeking a global optimization of the various TM parameters.

System	Model	Learning	TM parameters	Concurrency	TM Back-End
TinySTM	white-box	off-line	$\surd$		
Tuner	black-box	on-line	$\surd$		
SAC-STM	black-box	off-line		$\surd$	
Qingping, et al. [6]	black-box	hybrid			$\surd$

Table 2.3: Summary of the presented self-tuning systems

This relevant limitation is tacked and overcome by a recent solution, called ProteusTM [5], which we are going to describe in detail in the next chapter and whose extension will focus the work of this dissertation.

Chapter 3

Study on ProteusTM

In this chapter, we present an in depth review of ProteusTM, a self-tuning system for TM, which, already mentioned in Chapter 1, represents the main focus of this dissertation. Firstly we present an overview of ProteusTM as well as its architecture and main components. At the end, we present some of its key limitations, that will be addressed later in the dissertation.

3.1 ProteusTM

ProteusTM [5] is the only self tuning solution for TM systems that supports a multi-dimensional optimization scheme, in contrast to all the solutions until this point, which were unidimensional. ProteusTM is a state of the art solution that keeps the abstraction and simplicity of TM, while also tuning the TM implementation according to workloads.

At its core ProteusTM has two main components *PolyTM* and *RecTM* (see Figure 3.1). PolyTM consists of a Polymorphic TM library, with several TM implementations (HTM, STM, HyTM) and allows the reconfiguration of TM along the following dimensions:

1. Switch between different TM implementations (TinySTM to NOrec for example);
2. Reconfigure internal parameters of a TM;
3. Adapt the number of threads concurrently generating transactions.

RecTM is responsible for identifying the best configuration of PolyTM for the current workload. It tackles this challenge by using techniques developed in the *recommendation* system's literature.

Briefly, ProteusTM applies CF to the problem of identifying the best TM configuration that maximizes the KPI for some workload. We now present the architecture of ProteusTM in Figure 3.1 and overview its internal components which enables the self tuning capabilities of this solution.

As explained above PolyTM is a polymorphic TM library that contains several TM implementations, allowing the reconfiguration on a multi-dimensional level on TM internal parameters.

In Figure 3.1 note that RecTM, the component responsible for predicting the best TM configuration, is composed by 3 sub-modules [5]:

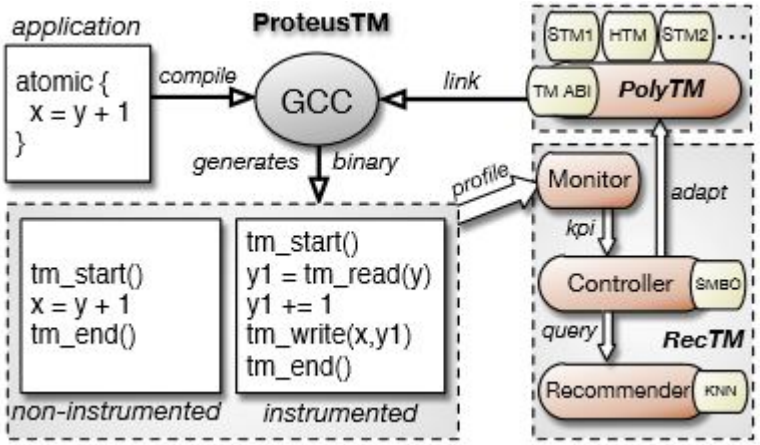


Figure 3.1: Architecture of ProteusTM system[5]

1. **Recommender** the recommendation system that acts as the predictor and supports different CF algorithms.

2. **Controller** selects the configuration to be used and triggers the adaptation in PolyTM, by querying the Recommender with the values obtained from the Monitor.

3. **Monitor** is responsible for gathering the target KPI to give feedback to the Controller about the quality of the current configuration. Also, it detects changes in the workloads with the objective of triggering a new optimization phase on the Controller.

The focus of this thesis is on one of the main components: RecTM which will be explained more in detail next.

3.1.1 RecTM

RecTM is the component responsible of finding the best TM configuration in order to optimize PolyTM, developed in Java. As mentioned it casts this challenge to a recommendation problem, and uses CF techniques to maximize a certain KPI.

In Section 2.2.1.3 we explained some challenges when using CF techniques and some advantages of relying on this type of algorithms. A brief reminder, CF receives as input a sparse matrix (UM), and tries to estimate the missing values of the matrix to give accurate recommendations.

In ProteusTM the UM associates each row to a different workload, and each column to a different TM configuration. Exploiting CF-based recommendation systems for the self-tuning of TM raises a non-trivial problem, which is discussed next.

3.1.1.1 The Rating Heterogeneity Problem

CF algorithms assume the use of homogeneous rating scales, e.g., from 0 to 5 stars. Conversely, typical metrics used to guide the optimization process of a TM (such as throughput, execution time, abort rate) can span across very heterogeneous scales, depending on the applications characteristics

(e.g., long vs short transactions) and hardware characteristics (e.g., CPU clock speed). So, an optimal configuration for a given TM workload may yield, for instance, a throughput of $1e6$ txs/sec, whereas, for a different workload, the corresponding optimal configuration may yield a throughput of $1e3$ txs/sec (e.g., if transactions have a much longer duration in this second workload). In contrast, typical CF algorithms assume that the optimal ¡¡ for different "users" will have very close ratings (e.g., 0-5 stars).

A solution to the problem of heterogeneity is to normalize the whole UM. Normalization in this scenarios is not trivial since the maximum or minimum for KPI value of specific configurations for an unknown workload is clearly unknown — else the optimization problem would be trivial. Ideally, an efficient normalization should be able to transform the entries so that similarities can be mined and enable the use of CF techniques.

To further motivate the use of normalization we present a study on the performance of ProteusTM predictor capabilities, using different methods of normalization (depicted in Figure 3.3). The different types of normalization used were:

- NONE: No normalization, the CF algorithm is applied in the raw UM;
- MAX: normalization with respect to the max in the training set;
- WRT-BEST/IDEAL: an ideal normalization technique that assumes to know a priori the absolute value of the KPI in the optimal configuration for each workload (also the one being queried). The KPI achieved in the optimal configuration is then used as normalization factor, which ensures that the ratings are expressed using a single, uniform scale between [0,1] where 1 corresponds to the rating of the optimal configuration (assuming a maximization problem).
- Rating Distillation: a normalization procedure proposed in the ProteusTM work [5] and described more in detail in the following subsection.

The performance of ProteusTM will be evaluated using two accuracy metrics:

- Mean Average Percentage Error (MAPE) defined as: $\sum_{\langle u,i\rangle \in S} |r_{u,i} - \hat{r_{u,i}}|/r_{u,i}$. Where $r_{u,i}$ represents the real value of the target KPI for workload u when running i as configuration, $\hat{r_{u,i}}$ the corresponding prediction of the Recommender, and S the set of testing $\langle u, i\rangle$.
- Mean Distance From Optimum (MDFO) defined as: $\sum_{\langle u,i\rangle \in S} |r_{u,i^*_u} - r_{u,\hat{i^*_u}}|/r_{u,i^*_u}$. Where i^*_u is the optimal configuration for workload u, and $\hat{i^*_u}$ the best configuration found be the Recommender.

The MAPE reflects how well the CF learner predicts performance for an application, while MDFO captures the quality of the final recommendations.

In order to build the experimental test-bed used in all the experiments of this dissertation, ProteusTM was deployed in two machines with different characteristics with a wide variety of TM applications, and tuning parameters depicted in Figure 3.2. In these two machines, over 300 workloads were deployed which are representative of heterogeneous applications such as (summarized in table 3.1): STAMP [46], Data Structures, STMBench7 [47], TPC-C and Memcached [48]. The experience data set was built by

Machine ID	TM Backend	# threads	HTM Abort Budget	HTM Capacity Abort Policy
Machine A	STMs and TSX [67]	1,2,3,4, 5,6,7,8	1,2,4, 8,16,20	Set budget to 0; decrease budget by 1; halve budget
Machine B	STMs	1,2,4,6, 8,16,32,48	N/A	N/A

Figure 3.2: Parameters tuned by ProteusTM. STMs are TinySTM [3], SwissTM , NORec [2] and TL2

Benchmarks	Description
STAMP [46]	Suite of 8 heterogeneous benchmarks with a variety of workloads (genomics,graphs,databases).
Data Structures	Concurrent Red-Black Tree, Skip-List, Linked-List and Hash-Map with workloads varying contention and update ratio.
STMBench7 [47]	Based on OO7 [49] with many heterogeneous transactions over a large and complex graph of objects.
TPC-C	OLTP workload with in-memory storage adapted to use one atomic block encompassing each transaction.
Memcached [48]	Caching service with many short transactions that are used to read and update the cache coherently.

Table 3.1: TM applications. These 15 benchmarks span a wide variety of workloads and characteristics [5].

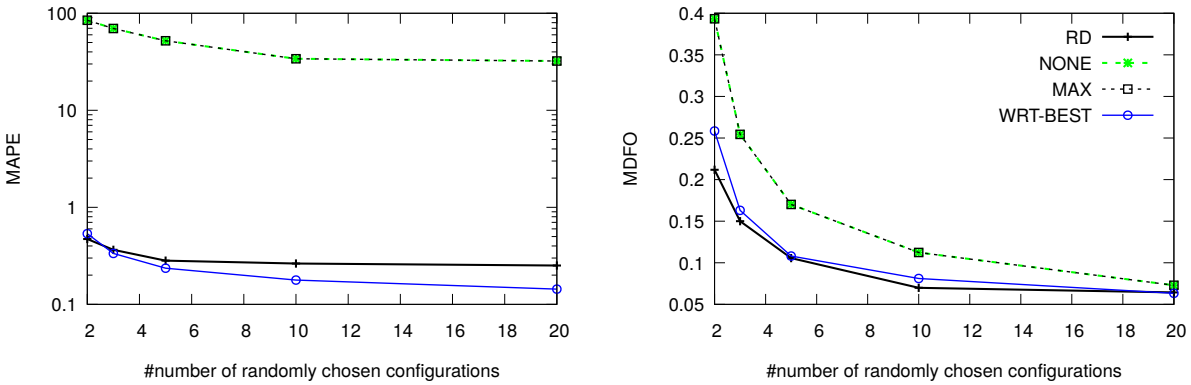
collecting over 5 runs, the KPI (e.g., throughput and execution time) in a real-time trace driven evaluation of over 300 workloads and 160 TM configurations.

The KPI used for this study was the execution time. A brief reminder the training set - also called UM - will have as rows the workloads and as columns the TM configurations.

The evaluation's learner was KNN with cosine similarity, trained with a random sub-set of the original data set split into 30% of training set and the remaining 70% the corresponding test set. The results presented are an average of 10 runs, for 10 different sub-sets of the original data set and for a different number of configurations with known performance for a given workload (chosen at random). The training set is used to instantiate the predictive model and where the normalization techniques are applied. The test-set has no intersection with the training set and provides each workload to ProteusTM, so that this can predict the values not present in the sampling. To simulate sampling for the performance of an workload for a given configuration, we use the corresponding value from the test-set and add this value to the UM of the Recommender.

To ensure fairness, the study was conducted with the same training sets and the same initial configurations were provided for the different normalizations. Figure 3.3a represents how far in average are the predictions from the actual rating, while Figure 3.3b quantifies the quality of the final recommendation, i.e., how far is the recommended configuration from the actual optimum.

Considering different normalizations and Figure 3.3 we reinforce the statement that the predictive accuracy of CF algorithms is strongly affected by the choice of the normalization procedure. We can verify that applying no normalization or using the normalization with respect to the max, the predictor performs very poorly in terms of overall predictions (Fig. 3.3a) and when finding the optimal configuration (Fig. 3.3b). While using an ideal solution or Rating Distilation (RD) the recommendations are really close



(a) MAPE (b) MDFO

Figure 3.3: KNN Cosine with 2 neighbours. The graphs were obtained by running ProteusTM with different training sets, in which we provided different initial configurations for the sampling of a new workload

to the optimum configuration and to the overall absolute values of the predictions.

3.1.1.2 Normalization in the Recommender

An ideal normalization cannot be used in ProteusTM since it would require knowing a priori the KPI of the optimal configuration for an unknown workload — where the optimization problem being targeted consists in fact in identifying which is the best performing configuration. Also, as we have seen in the above study, the use of a static normalization solution (e.g., based on normalizing by the maximum KPI measured across all possible workloads) provides strongly suboptimal results, since the absolute KPI values achieved by different workloads are distributed over very heterogeneous scales whose min/max values can span several orders of magnitude.

RD, the normalization techniques proposed in the ProteusTM's work [5], aims at approximating the ideal solution described in the previous sub section by ensuring that for any workload w:

(i) of two configurations c_i and c_j, namely kpi_w, c_i and kpi_w, cj, the ratio is preserved in the rating space, i.e., $\frac{kpi_{w,c_i}}{kpi_{w,c_j}} = \frac{r_{w,c_i}}{r_{w,c_j}}$ where r_w, ci and r_c, c_j represent the ratings attributed, respectively, to configurations ci and cj for workload w.

(ii) the ratings of the various configurations of a workload w are distributed in the range [0, M_w] so as to minimize the the index of dispersion of M_w : $D(M_w) = \frac{var(M_w)}{mean(M_w)}$

Property (i) ensures the distance between two configurations is correctly encoded when the ratings are normalized, while property (ii) by minimizing the index of dispersion of the vector M_w attempts to align the scales used to express the ratings for the various workloads to a similar upper bound, which is as homogeneous as possible across different workloads.

Rating Distillation (RD) algorithm used by the Recommender is depicted in Algorithm 1. RD is a normalization technique that assumes the availability, for each workload included in the training set, of the KPI achieved by using *all* the available TM configuration $C_i \in C_1, \ldots, C_K$. In other words, RD assumes a dense UM in the training set, except of course, for the workload being queried for which

Algorithm 1 Rating Distillation algorithm [5].

1: **for** $C_i \in C_1 \dots C_K$ **do**
2: Normalize Matrix KPI w.r.t. C_i
3: Collect the vector M_w with the max values per row
4: Compute $mean_i(M_w)$ and $var_i(M_w)$
5: **end for**
6: Return C* $= argmin_{i \in 1 \dots M}\ var_i(M_w)/mean_i(M_w)$

only a small subset of configurations is assumed to have been sampled. It starts by normalizing the dense UM with regards to a configuration C_i (line 2), and collects the maximum performance of each workload , building the vector M_w that contains the maximum values on a per workload basis (line 3). Bellow we have an graphical example of the first iteration of this algorithm (Figure 3.4), for C_1 where $r_{w,i} = kpi_{w,i}/kpi_{w,1}$:

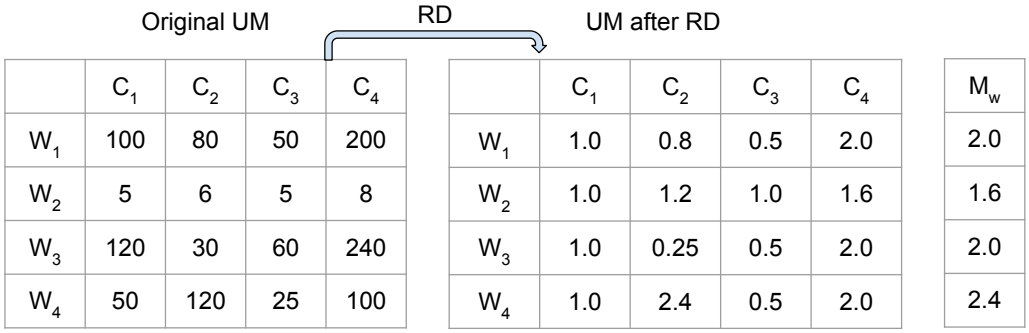


Figure 3.4: First iteration on a matrix 4×4 of the algorithm RD, i.e., normalize the matrix with respect to configuration C_1 [5]

Repeating this method for all the configuration space C, one can obtain, for each configuration in C, a different M_w vector. Rating distillation selects as pivot column C^* for the normalization, the one whose corresponding M_w has the smallest index of dispersion. Once C^* has been identified, the UM matrix, including the sparse row containing the workload being predicted is normalized by dividing the KPIs in each workload (i.e., row) by the KPI achieved by that same workload in configuration (i.e., column) C^*. In other words, the UM matrix, after normalizing with RD, expresses as rating the normalized performance achieved by the TM system with respect to the the pivot configuration C^*.

3.1.1.3 RecTM Workflow

After addressing how the recommender tackles heterogeneity, the next step is to understand how does RecTM optimize PolyTM (see Algorithm 2).

RecTM employs a black-box approach that relies on on-line and off-line training. Firstly it does off-line performance profiling of an initial training set, in which it explores a mix of workloads on the full spectrum of the available TM configurations. Next, it applies the rating distillation to obtain homogeneous ratings for each workload, resulting in the initial/training dense UM. Based on the training UM, the sub-module *Recommender*, selects and configures the CF algorithm (choosing between KNN and SVD) to use at run-time. Now that the off-line configurations ended, the system is ready to receive new workloads.

Algorithm 2 RecTM work-flow [5]

1: Off-line performance profiling of an initial training set of applications
2: Rating distillation and construction of the Utility Matrix.
3: Selection of CF algorithm and setting of its hyper-parameters.
4: Upon the arrival of a new workload:
5: Sample the workload on a small set of initial configurations.
6: Recommend the optimal configuration).

At the arrival of a new workload, the Controller drives the on-line profiling using Sequential Model-based Bayesian Optimization (SMBO). This technique samples the new workload in a small number of initial configurations, and tries to fit a probabilistic model. Then, it identifies the next point (TM configuration) to be sampled based on the expected gain computed on the basis of the CF-based recommendations (Expected Improvement [50]). This process of exploration stops when the values for Expect Improvement (EI) either:

- Decreases twice in a row;
- for the k-th exploration was marginal, i.e., below a certain range with respect to the current best KPI;
- the achieved improvement did not exceed some threshold related to the current best KPI;

Based on the selected CF algorithm, it recommends the optimal configuration for the new workload. The implementation of CF algorithms in ProteusTM is accomplished by using Mahout [51], a ML framework containing several CF algorithms including the ones used by RecTM, namely KNN and SVD.

The *Monitor* is responsible for collecting the target KPI at the arrival of new workloads and feed it to the *Controller*, so that the later can realize the on-line profiling. The Monitor is also responsible for the detection of workload changes and for triggering the optimization process.

3.2 ProteusTM Limitations

ProteusTM is the first solution to tackle the problem of optimizing TM in a multi-dimensional scheme, unlike existing solutions in the literature on self-tuning of TM (See Section 2.2.2). However, it is not a solution exempt of limitations, which we will analyse in the remainder of this section.

As stated, Collaborative Filtering techniques in ProteusTM, bases its estimations assuming the existence of a fully populated *UM*. This is only feasible if the configuration space is small, e.g., hundreds configurations and workloads. However, considering the many parameters that can affect TM performance (including not only the internal TM parameters, but also the choice of the TM algorithm and the architectural characteristics of the underlying hardware platform), there exist a great variety of possible configurations (columns of the UM). In the light of these considerations, if one wanted to include additional dimensions/parameters (e.g., thread mapping, different hardware architectures, etc.) to the available search space, it would make the use of ProteusTM's RD technique impractical or even prohibitively onerous/time consuming.

Since it is only feasible to use an initial dense matrix if the configuration space is small, ProteusTM suffers from a scalability problem. To improve the scalability of ProteusTM, one possible solution is to redesign it in order to use an initial sparse UM. In fact, CF techniques are known to achieve good results using *sparse* UM [34, 33], provided that they are fed with ratings expressed in uniform scales. A tightly intertwined problem is to identify normalization techniques alternative to rating distillation that do not assume the availability of a dense UM.

The *first part* of this thesis will target precisely this problem.

A common technique adopted in the CF literature to cope with sparse UM matrix is to incorporate in the knowledge base not solely the explicit users ratings of the various items, but also a characterization of the user's profile [10].

This class of approaches motivates the research direction that this dissertation investigates: to what extent can the availability of information on the workload characteristics (e.g., transaction duration or abort rate) enhance the accuracy of RecTM, which, we recall, relies solely on KPI information. The key idea here is to extend the UM used in ProteusTM to incorporate workload information, i.e., extending the UM to include columns containing workload characteristics, information traceable at run-time. Figure 3.5 exemplifies this approach, where we extended the original UM represent by the columns $C_{1..6}$ with additional workload information WI. Let us assume one wants to predict the value for $kpi_{1,4}$, if we only consider the original UM (ignore WI) identifying the best similarity is not trivial since it shares one rating with two other workloads (w_3, w_4) for column C_4. However, if we consider the new additional workload information it becomes clearer to which of the workloads is the most similar, with respect to w_1 . The workload w_3 if we consider this new information is likely to share the same trend as w_1 for configuration C_4.

UM with workload information

	C_1	C_2	C_3	C_4	C_5	C_6	WI_1	WI_2
W_1	100		50				0.5	127
W_2	5				8		0.3	346
W_3		100	50	10			0.5	127
W_4		120		200			0.78	48
W_5		6			8		0.3	210
W_6			50	200		25	0.78	58

Figure 3.5: User-Item UM (straight line) and the UM with additional information (crossed line)

In summary with this thesis we intend to answer the following research questions:

1. How can ProteusTM be extended to exploit a sparse training set/UM?
2. Would incorporating workload information improve ProteusTM accuracy?

Chapter 4

ProteusTM Extension: Sparsity

The previous chapter presented in some detail ProteusTM, a self-tuning solution for TM. Despite being the only solution that uses self-tuning in multiple dimensions, it is not exempt of faults or limitations.

As highlighted the reliance on an initial dense matrix is a relevant drawback that can severely hinder the scalability of ProteusTM. This chapter will address this limitation of ProteusTM by investigating the problem of how to support sparse UM matrices in the ProteusTM framework.

As already mentioned, the base CF algorithms (SVD and KNN) employed by ProteusTM are already equipped to cope with sparse UM matrices. The key problem to address, though, is how to adapt the normalization procedure that generates the UM that is then fed to the CF algorithms. As already discussed in Section 3.1.1, on the one hand, the RD algorithm achieves very good results, but assumes a dense KPI matrix; on the other hand, simple normalization approaches (e.g., dividing by the max value in the training set), which would be straightforwardly usable with sparse UM lead to very unsatisfactory accuracy of the CF-based learners.

The key problem with using RD in presence of sparse matrices is that it requires to normalize each row with regards to the corresponding KPI value of each column. With a sparse matrix, we cannot normalize any row according to any column, as that row/workload may not store a KPI value for some specific column/configuration. Figure 4.1 describes a scenario of this problem, using the first iteration of RD.

In Figure 4.1, for the row w_1 since the rating for column C_1, $kpi_{1,1} = 100$ is present, the algorithm is successful: the formula for normalization $r_{w,i} = kpi_{w,i}/kpi_{w,1}$ works. In contrary, for w_2 the rating $kpi_{2,1} =?$ is missing and the denominator of formula is invalid, which also invalidates the use of this algorithm in sparse scenarios.

In the following we propose and evaluate the use of two normalization schemes designed to cope with sparse matrices:

- the Box-Cox transformation, a well known data transformation technique [52] that has been already successfully used in other (non-TM related) optimization problems [53] and that aims to generate data that follows normal distributions (See Section 4.1).

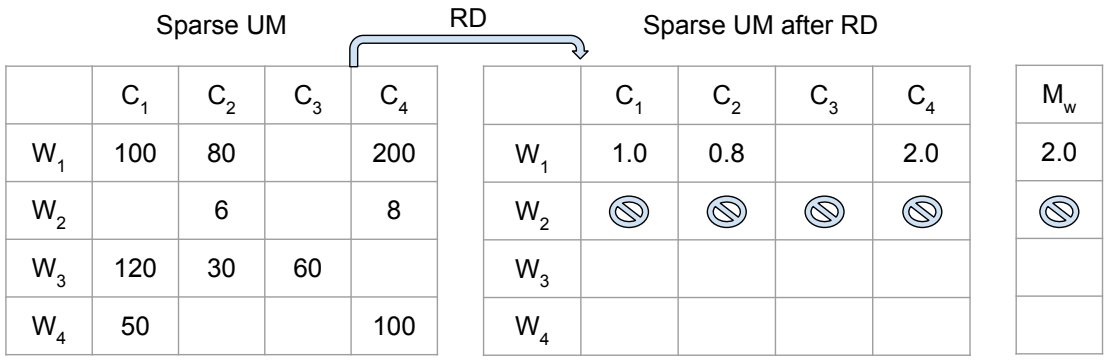


Figure 4.1: Example of RD first iteration with sparsity, where the algorithm fails since the KPI for column C_1 is not present in all rows.

- Sparse Rating Distilation (SRD), namely a variant of the RD procedure, that uses a normalization technique similar in spirit to RD but adapted to cope with sparse matrices. (See Section 4.2)

4.1 Box-Cox

Box-Cox is a parametric power data transformation, proposed in 1964, used in order to reduce anomalies such as non-normality in data [52].

The work in [53] used this transformation in order to stabilize data variance and make the data more normal distribution-like to fit their matrix factorization technique. Similarly in our own context, Box-Cox will be used to normalize the original UM.

Box-Cox is defined as follows, when the original data set $S \in R^+$:

$$boxcox(y) = \begin{cases} \frac{(y^\lambda - 1)}{\lambda}, & \text{if } \lambda \neq 0 \\ log(y), & \text{if } \lambda = 0 \end{cases} \tag{4.1}$$

where the parameter λ controls the extent of the transformation. The estimation of the parameter λ is one of the challenges for the application of this technique and it does not exist a standard approach to the problem of finding the best value for the parameter λ [52]. Next we present a different solution to the problem of tuning parameter λ based on the Kolmogorov-Smirnov test.

4.1.1 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov Test (KS-Test) is a non-parametric test that tries to determine if two data sets differ significantly from one another. If the test is concerned with the agreement between the distribution of a set of sample values and a theoretical distribution we call it a ”test of goodness of fit” (One sample KS-Test). Otherwise, it can be used to compare two different samples (Two sample KS-Test).

Suppose that a population is thought to have some specified cumulative frequency distribution function, say $F_0(x)$, i.e, for any specified value of x, the value of $F_0(x)$ is the fraction of observations in

the population having measurements less than or equal to x. If $F_0(x)$ is the population cumulative distribution, and $SN(x)$ the observed cumulative step-function of a sample then the Kolmogorov-Smirnov statistic is [54]:

$$Dn = max|F_0(x) - SN(x)| \tag{4.2}$$

This statistic quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. If the sample $SN(x)$, comes from a distribution close to $F_0(x)$ then the value of Dn will converge to 0.

We will use the Kolmogorov-Smirnov statistic in order to chose which is the best alpha for a training set in an automatic way.

4.1.2 Box-Cox and KS-Test

The algorithm that uses KS-Test to tune the parameter λ of Box-Cox will be presented in this section. The main objective of the algorithm is to find a value for λ that better normalizes a random training set, which is the one that minimizes the Kolmogorov-Smirnov statistic. We will use the one-sample KS-Test which will compare the empirical cumulative distribution of the matrix UM against a Normal Distribution with the same average and standard deviation. The closer the Dn is to zero, the better is the value for λ, i.e., the transformation makes the original training set closer to a normal distribution.

Before explaining the algorithm it is relevant to mention that, the Box-Cox algorithm standard values for λ are normally in between the interval [-2,2] [55].

In order to use KS-Test together with Box-Cox this was the developed algorithm:

Algorithm 3 Box-Cox function in ProteusTM

1: **for** $\lambda \in [-2; 2]$; with step 0.1 **do**
2: Normalize Matrix KPI using box-cox with λ (equation 4.1).
3: Gather the new average (avg) and standard deviation (stdev).
4: Transform the resulting matrix into a vector v organized in an ascending order of its elements.
5: Apply KS-Test:
6: Build the reference NormalDistribution(avg, stdev), F_0.
7: Build the Empirical Cumulative Distribution (ECD) of v, $SN = ECD(v)$
8: Compute Dn for the current λ(§ 4.1.1 equation 4.2)
9: **end for**
10: **return** the value λ^* that achieved minimum Dn

The first step to implement this algorithm is to apply the Box-Cox normalization (Equation 4.1) on the UM, in an interval starting from -2 to 2, increasing by a fixed step (which we set to 0.1) per iteration (line 1-2). In order to use the KS-Test we need the reference Normal Distribution and the Empirical Cumulative Distribution (ECD) of our sample (UM). To build the reference Normal Distribution, we first calculate the average and the standard deviation of the elements present in the resulting matrix, and using the Apache Commons Library [1] we create the reference *NormalDistribution(average, standard*

[1] http://commons.apache.org/

deviation). Next, we transform the normalized UM into a vector, of size *n*, organized in ascending order and calculate its ECD, defined as follows for *n* observations X_i:

$$SN(x) = \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty,x]}(X_i) \tag{4.3}$$

where $I_{[-\infty,x]}(X_i)$ is the indicator function, equal to 1 if $X_i \leq x$ and equal to 0 otherwise.

With these two distributions we can calculate the Dn (line 8), repeating these steps until we reach the end of loop. Finally, the returned λ^* is the one that achieves the smallest Dn value (line 10) amongst the several iterations. From some examples of this method [54, 52], the values for λ had deeper granularity, normally numbers in the thousandth.

We created a new function that modifies the interval (used in line 1) at run-time, "zooming in" the chosen lambda, i.e., creates a new interval around the chosen λ^*, $[\lambda^* - 0.1, \lambda^* + 0, 1]$, to verify deeper granularities and invokes Algorithm 3 once again with this new interval: **for** $\lambda \in [\lambda^* - 0.1, \lambda^* + 0, 1]$; with step 0,001 **do**.

To demonstrate the capabilities of the developed algorithm, we submit this technique to the most heterogeneous data set available (UM), obtained using the same methodology presented in the previous study in which the KPI is the throughout (number of committed transactions per second). The data-set is very heterogeneous as it can be seen by the plot below and definitely not normally distributed before applying the box-cox transformation.

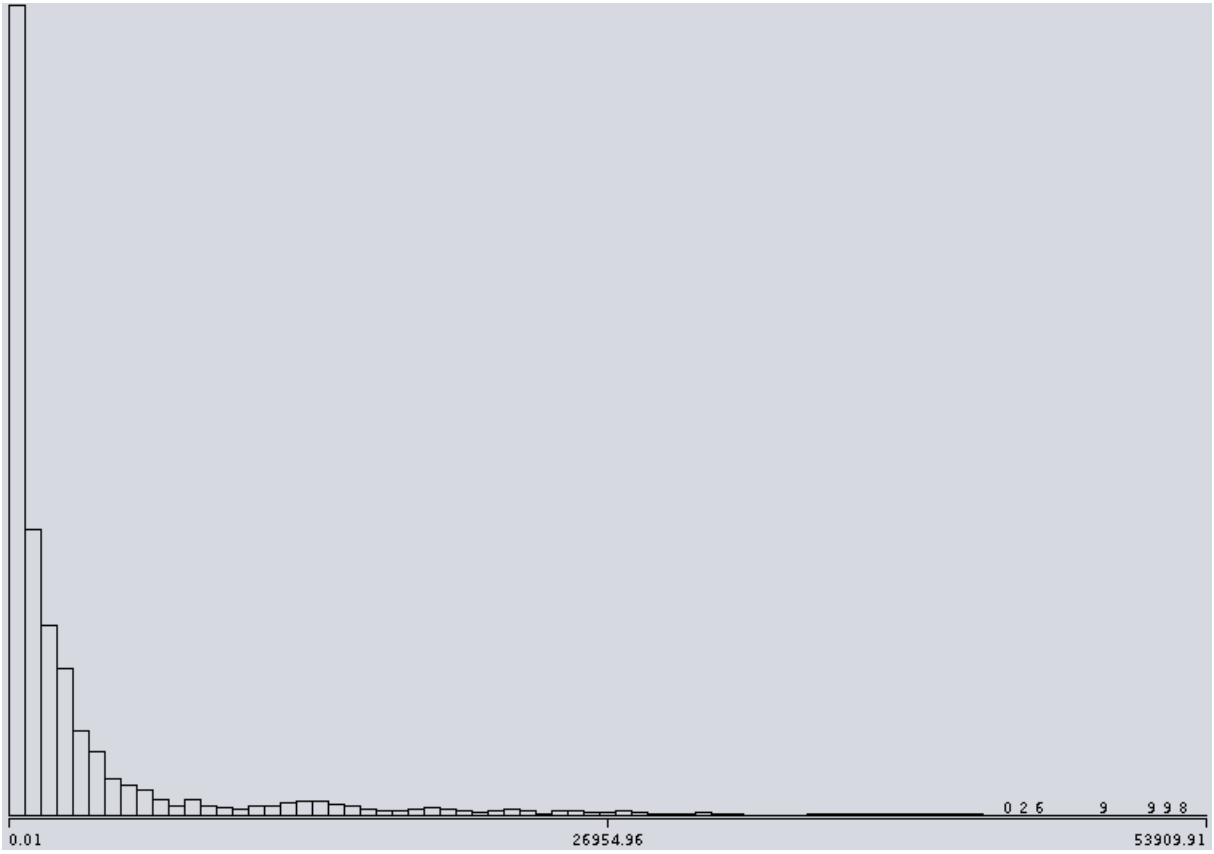


Figure 4.2: Heterogeneous data set with throughput values in the interval [0,01; 53909,906]

We use Weka [56], a workbench with several machine learning algorithms and data pre-processing tools, to automatically build an histogram to verify if the algorithm transforms the original data into a normal shape. Figure 4.2 is the training set with no transformation whatsoever, where the x-axis represents the different throughput values and the y-axis the frequency with which a given KPI (on the x-axis) appears in the original data-set. The buckets are automatically chosen by Weka, to better fit all the data in the histogram, choosing very high buckets size, proportional to the interval in which the data set is distributed.

We will now apply Algorithm 3 with the zoom-in function to this data set, generating Figure 4.3.

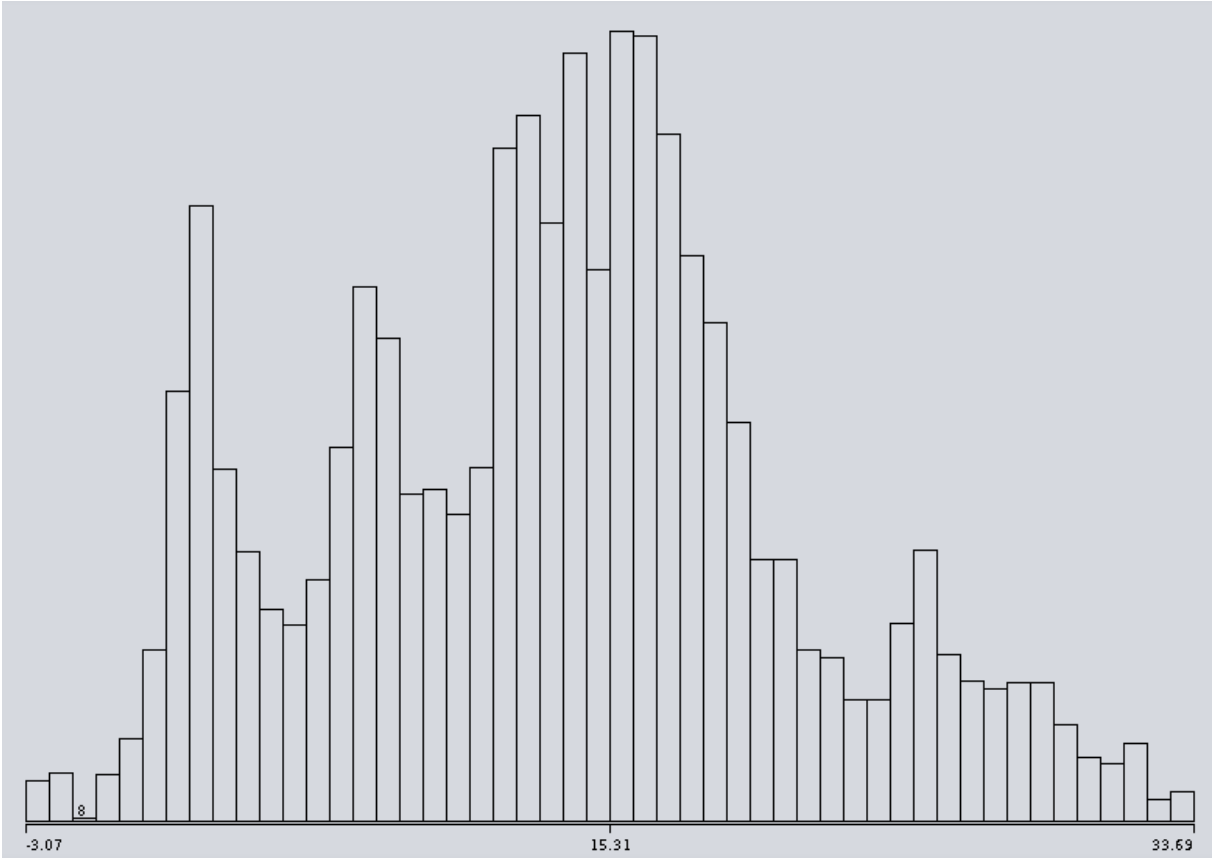


Figure 4.3: Algorithm 3 and zoom-in function applied in the data set depicted in Figure 4.2

Even though there are some bumps in the transformed data presented in Figure 4.3, we can easily conclude that using this normalization technique, successfully transformed the original data set into a data set that approximates much better a normal distribution. The mean for the normalized data is 13,675, which is rather close to the ”belly” shape of the figure, strengthening the claim that the data is closer to a normal distribution. Although the data looks close to a normal distribution, we cannot take any conclusions regarding the impact in the accuracy of CF algorithms, a study that will be presented later in this chapter.

4.2 Sparse Rating Distilation

RD obtained results really close and sometimes better than the ”ideal normalization”, motivating the redesign of this algorithm to work in sparse scenarios. Briefly summarizing the problem of RD is the per column normalization for each row, where in a sparse scenario not every row of the matrix is obliged to have the chosen column invalidating the use of this algorithm. In the remaining of this section we will use the terminology presented in Algorithm 1 (Section 3.1.2).

Algorithm 4 Sparse Rating Distillation algorithm.

1: p = sparsity percentage
2: **for** $C_i \in C_1 \dots C_K$ **do**
3: Build Sub-Matrix $m \times C_K$, with rows where $kpi_{w,i} \neq null$
4: Normalize Sub-Matrix KPI w.r.t. C_i
5: Collect the vector M_w with the max values per row
6: Compute $mean_i(M_w)$ and $var_i(M_w)$
7: **end for**
8: C* = ($argmin_{i \in 1 \dots M \wedge |m_i| > |w| \times (1-p)}$ $var_i(M_w)/mean_i(M_w)$)
9: **if** C* = $null$ **then**
10: C* = ($argmin_{i \in 1 \dots M}$ $var_i(M_w)/mean_i(M_w)$)
11: **end if**
12: Return C*

SRD is our modified version of the RD that works very intuitively, and adds a few steps to the original RD approach (See Algorithm 4). The first modification starts by building a sub-matrix of the UM of size $m \times C_K$, where only the rows that store some KPI value for column C_i are present, i.e, rows missing the rating for column C_i are removed. Intuitively, the formula $r_{w,i} = kpi_{w,i}/kpi_{w,i}$ will never fail since we guarantee that only workloads with C_i are present. If we recall Figure 4.1, the application of this new algorithm would build a new sub-matrix, excluding w_2 since it is missing the rating for the column being tested, C_1.

The first modification did not account for the fact that by building a new sub-matrix with respect to a configuration C_i, we could be reducing significantly the size of the original matrix. This may lead to computing the index of dispersion on an excessively small array M_w, leading to misleading results with no statistical significance. To mitigate this problem, we extended the condition which chooses configuration C^* that minimizes the index of dispersion of M_w (first condition of line 8) to also consider a new threshold.

This threshold imposes a new condition on how to chose the best configuration of RD. Consider p the percentage of sparsity in the UM, m the workloads selected to be part of the sub-matrix and w the number of workloads in the original matrix. The threshold is the following: $|m| >= |w| * (1.0 - p)$, i.e., the number of rows of the sub-matrix cannot be less than $(1.0 - p)$ of the original matrix.

The new algorithm chooses the first configuration C^* that minimizes the index of dispersion M_w and respects the threshold (line 8); for extremes cases if there is no configuration C_i that respects both conditions we use the return statement of the original algorithm (line 9-12).

4.3 Evaluation

This section is devoted to presenting a study of the quantification of the accuracy of ProteusTM with the introduction of sparsity, as well as compare the two normalization techniques: Box-Cox (BC) and SRD.

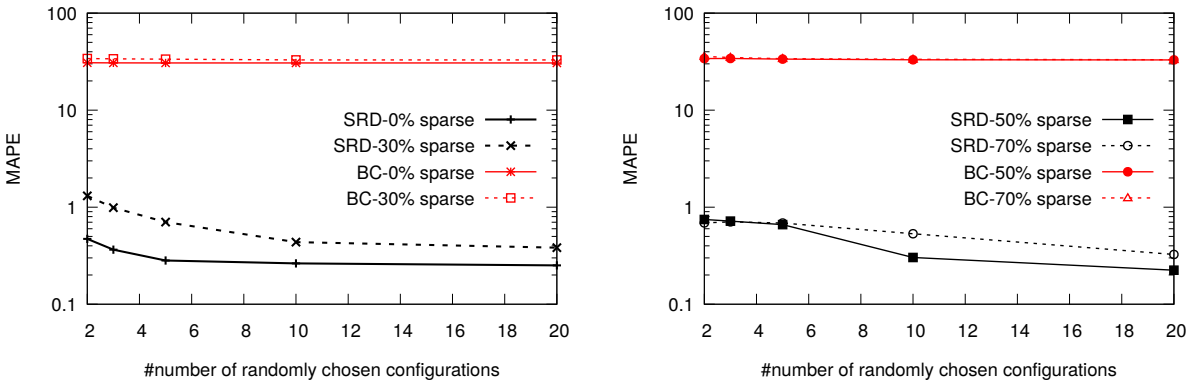
The evaluation metrics and the experimental test bed are the same as the ones detailed in section 3.1.1, with execution time as the KPI. The study will conducted using two different learners: KNN and SVD. With this evaluation we tackle the first research question presented.

4.3.1 KNN

To ensure fairness when comparing the different normalization techniques, we used the same training sets and the same initial configurations. The results presented in this section are the average of 8 different randomly generated 30% training and 70% test sets derived from the original data-set. For each training set we ran 10 iterations of randomly chosen initial configurations.

We chose a fixed configuration for KNN in order to obtain comparable results without any additional nuances. The initial configuration for KNN was: User-based cosine similarity with 2 neighbours.

The remainder of this section will focus on the comparison of the two normalization techniques and the behaviour of ProteusTM with sparsity.



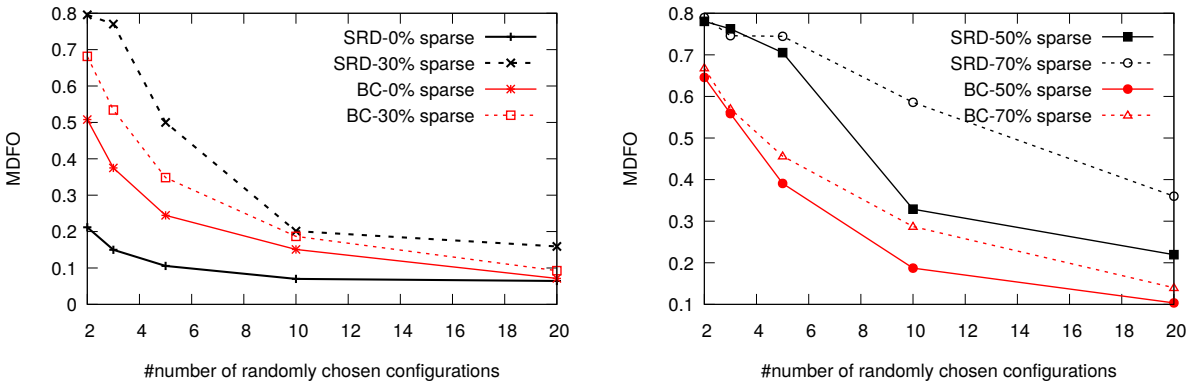
(a) MAPE for 0% and 30% sparsity

(b) MAPE for 50% and 70% sparsity

Figure 4.4: KNN - MAPE for different sparsity level of a 30% training set.

Figure 4.4 (a) and 4.4 (b) shows that BC performs rather poorly in terms of overall predictions, since it has an almost constant MAPE for any number of initial configurations. In contrast, SRD attains a better MAPE. The quality of predictions in SRD increases when adding additional sampled workloads and naturally decreases as the sparsity increases. However even at 70% sparsity, the MAPE is in average only 40% worse than the MAPE achieved with a dense matrix.

One interesting result of this plots, is the correlation between the MAPE and MDFO. Figure 4.5 (a) and (b) is the study using the metric MDFO for different sparsity levels. An overview of this plots is: increasing sparsity, in general decreases the accuracy of the predictor with any of the presented normalization techniques.



(a) MDFO for 0% and 30% sparsity

(b) MDFO for 50% and 70% sparsity

Figure 4.5: KNN - MDFO for different sparsity level of a 30% training set.

Note that for 0% sparsity SRD always beats BC, as we would expect looking at the MAPE. The results changes when sparsity is introduced. Although, the MAPE results for BC are really poor, it predicts with good accuracy the optimal configurations (Figure 4.5). BC with little knowledge about the workloads (2,3 initial configurations) obtains an MDFO around the 50%-70%, while SRD obtains values firmly around the 80%. Regarding the best results, when we sample 20 initial configurations BC reaches an MDFO around 10% while SRD considering only sparsity > 0 oscillates between 20%-48%.

In general, if the target problem consists in identifying the optimal configuration (and not predicting its corresponding KPI value), BC outperforms significantly SRD in presence of sparse training sets. Relative to the MAPE, although BC is clearly worse then SRD, none of the of the two considered approaches obtains as good results as for when the matrix has no sparsity.

Due to the discrepancy of the results between the MAPE and MDFO for BC, we conducted another study for the values of the KPI when using this normalization technique, which aims at shed lights on why BC is good when finding the optimum, but bad in the overall predictions. To this end, in Fig. 4.6, we show the real (red columns) vs predicted (blue columns) KPI values using BC for a given configuration, after having sorted the configurations on the x-axis from best (smallest execution time) to worse (largest execution time) for 0% sparsity .

Notice that in the left most part of the graph the predicted values with the best rankings/predictions are not really that far away from the real in terms of absolute value. Although, only the predictions are organized in ascending order, the real values with some exceptions seem to follow the same trend. Now, if we compare the right part of the graph the difference between the predicted and the real values becomes really accentuated. Provided Figure 4.5 and 4.6 we can conclude that using BC to normalize the original data it sets a scale that allows the identification with good accuracy of the optimal configurations, but is not so good in terms of absolute values of the predictions.

We provide another study with a 70% training set and 30% test set (Figure 4.7 and 4.8), to further back our claims. The same trend is noticeable from analysing this plots, for 0% sparsity SRD seems to achieve better results overall in terms of average prediction and finding the optimal configuration. However, introducing sparsity once again BC achieves better results when finding the optimal configuration.

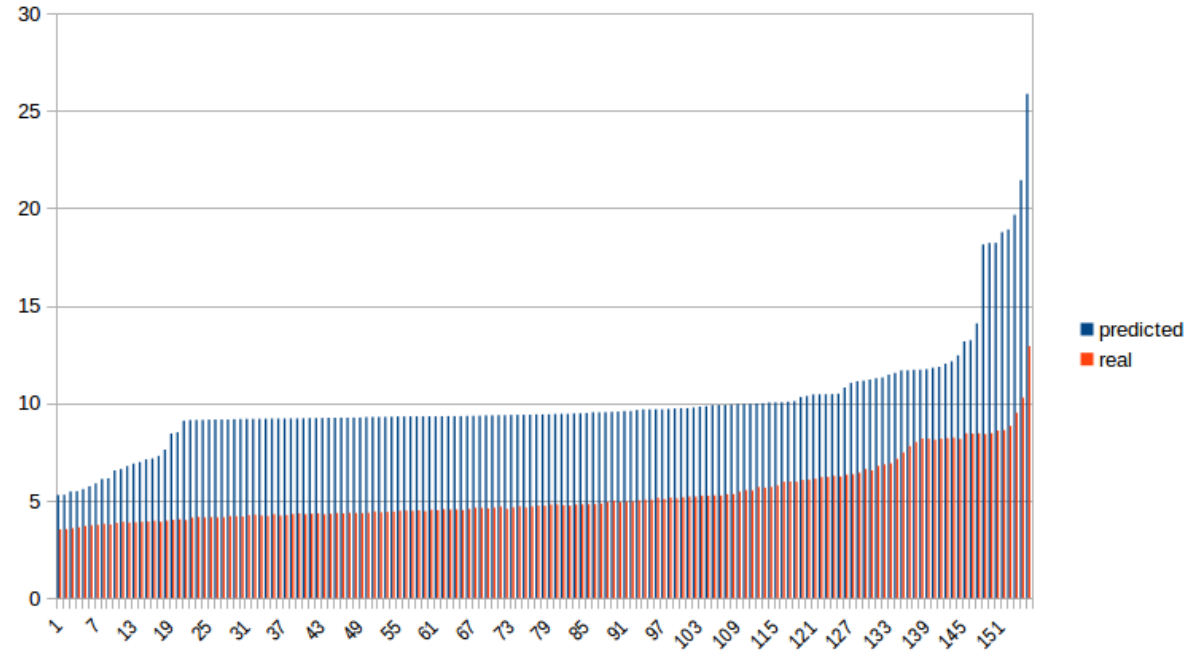
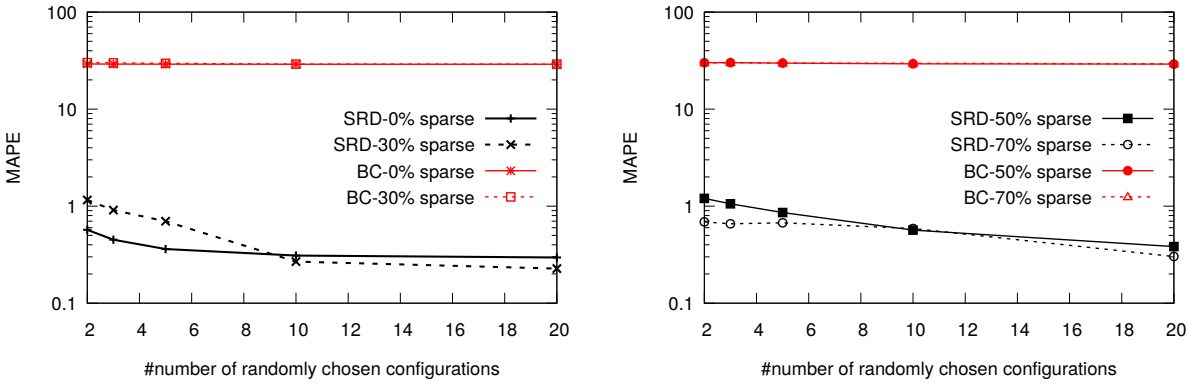


Figure 4.6: Predicted Ratings vs Real Rating, organized from the best predicted (lowest rating) to the worst, and the corresponding real ratings. The x-axis does not represent the id of a configuration but its ranking.



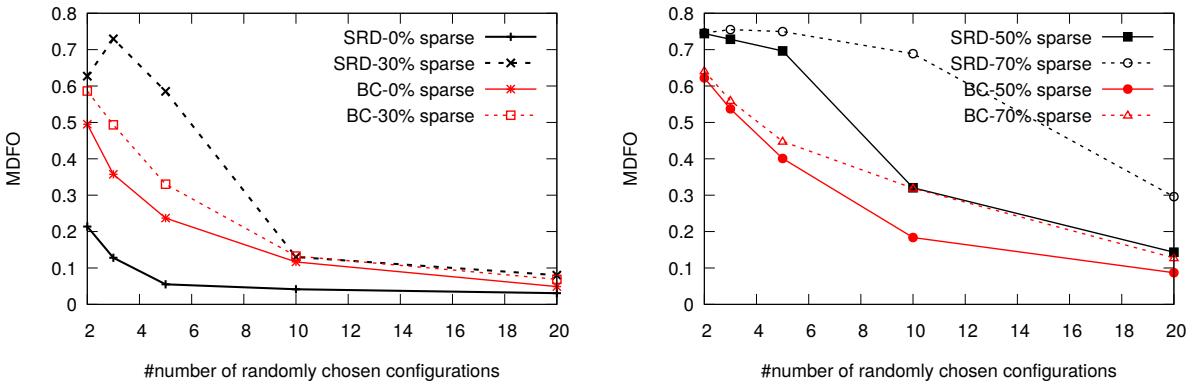
(a) MAPE for 0% and 30% sparsity

(b) MAPE for 50% and 70% sparsity

Figure 4.7: KNN - MAPE for different sparsity level of a 70% training set.

A relevant observation to make is that with 70% training set the predictor improve its accuracy in comparison with a 30% training set. Considering the 0% sparsity scenario, SRD now achieves and MDFO of 3% from the optimum comparing to 6% in the 30% training set. This trend seems to follow for all the different sparsity levels. This is not unusual when we provide a bigger training set the more accurate becomes the model since it becomes richer in the parameters space [29].

Following these statements, we present a intuitive reason for why SRD achieves worse accuracy performance. Figure 4.9 depicts a study on how many rows from the original training set were dropped, for different training sets percentages (lines) and for different sparsity levels. One can easily conclude that the percentage of rows dropped is highly correlated with the sparsity level, and this can explain why



(a) MDFO for 0% and 30% sparsity

(b) MDFO for 50% and 70% sparsity

Figure 4.8: KNN - MDFO for different sparsity level of a 70% training set.

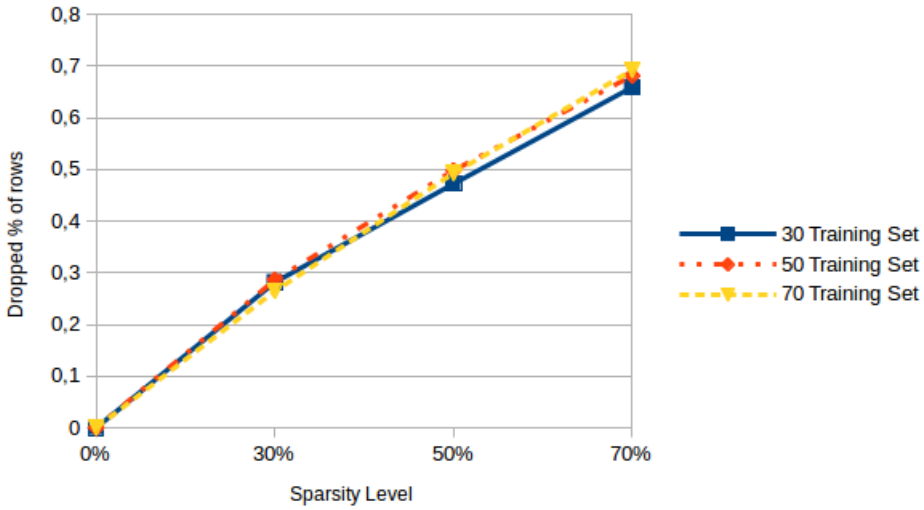


Figure 4.9: The graph represents how much % of the training set was dropped for different sparsity levels and training sets

SRD starts under performing when sparsity is introduced. While BC which is a black model approach uses the total number of rows/workloads, SRD drops a percentage p close to the sparsity present in the matrix.

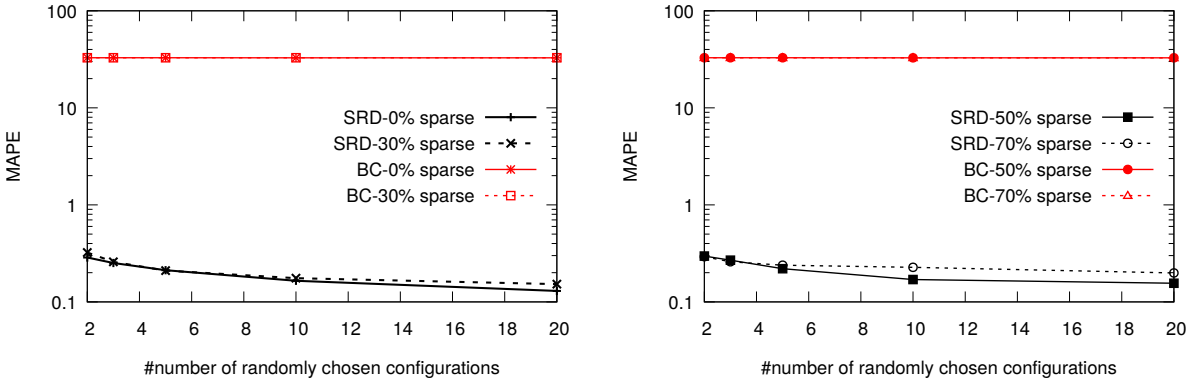
This is in fact rather intuitive, given that setting the sparsity level to $v \in [0, 1]$ coincides with setting to v the probability to include any cell in the UM. This probability coincides with the probability for SRD to drop a row of the original training set, since this happens whenever the cell associated with the target column consider by SRD is absent. Such probability clearly coincides with v.

4.3.2 SVD

SVD is the second learner we will use to evaluate the performance of ProteusTM and further validate the conclusions. SVD is significantly slower when compared to KNN (6$\times$ slower), so we had to reduce the set of considered test scenarios. Maintaining the number and the percentage of the (30%) training and (70%) test sets, we alter the number of iterations to two for the randomly chosen configurations

instead of using ten as in KNN.

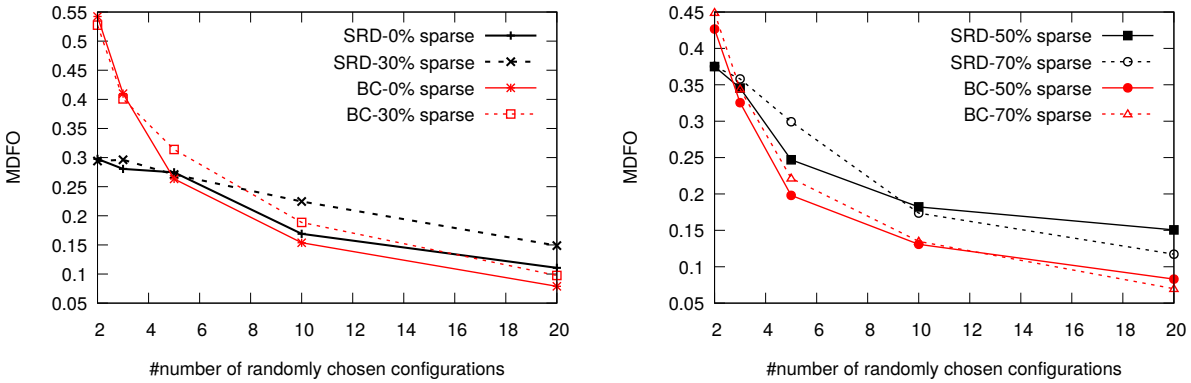
We will now present in the same fashion as in the previous section plots for the MAPE and the MDFO.



(a) MAPE for 0% and 30% sparsity

(b) MAPE for 50% and 70% sparsity

Figure 4.10: SVD - MAPE for different sparsity level of a 30% training set (logarithmic scale).



(a) MDFO for 0% and 30% sparsity

(b) MDFO for 50% and 70% sparsity

Figure 4.11: SVD - MDFO for different sparsity level of a 30% training set.

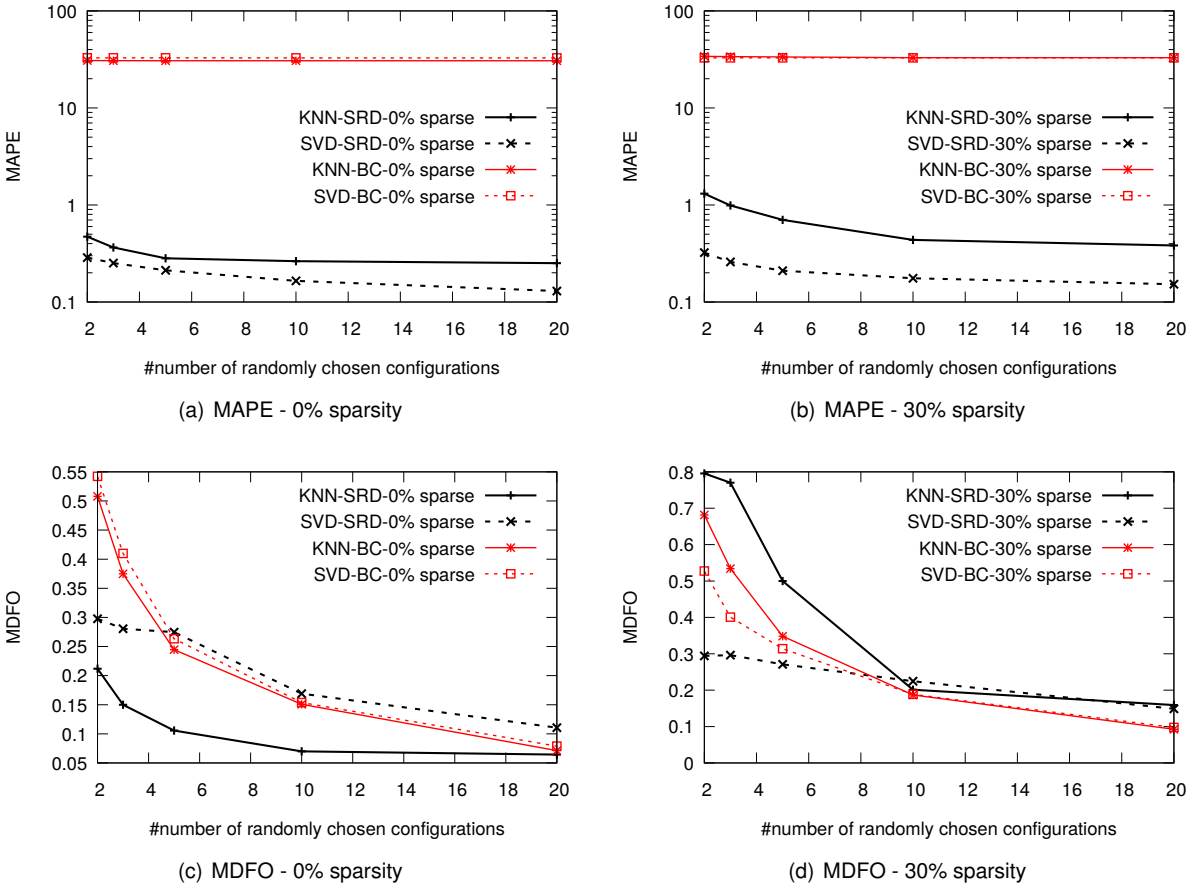
Regarding the MAPE (see Figure 4.10) for BC the results and conclusions are similar to the ones seen with KNN, i.e., constant and high values. SRD even with sparsity obtain good results in terms of MAPE, i.e., even in the worst case the predictions are in average 40% far from the actual ratings.

In the presence of sparsity and using SVD the MDFO results look promising, especially for SRD which is now competitive regarding BC MDFO results. SRD in small sparsity percentages (0% sparsity and 30%), for a small number of sampled configurations (initial configurations < 5) achieves very good results that are 30% from the optimal configuration. In contrast BC seems to have a cold star problem, i.e., poor results for a small number of sample configurations. If we increase the number of sampled configurations BC improves significantly, in average 40% if we consider the best scenario (20 initial configurations).

BC achieves better performance overall, i.e., for more than five sampled configuration for all levels of sparsity obtains better MDFO results, but SRD is only 5% - 10% from BC MDFO results.

4.3.3 KNN vs SVD

We also present a study comparing KNN and SVD (Figures 4.12 and 4.13). We display the plots in a different manner to have a better perception when comparing both learners. The same data is utilized to build this plots. In addition, we divided the plots for each different sparsity level where we compare both the normalization techniques with the different learners, i.e., KNN-SRD-0% refers to KNN using SRD for 0% sparsity, whereas SVD-BC-0% uses SVD with BC for 0% sparsity.



(a) MAPE - 0% sparsity

(b) MAPE - 30% sparsity

(c) MDFO - 0% sparsity

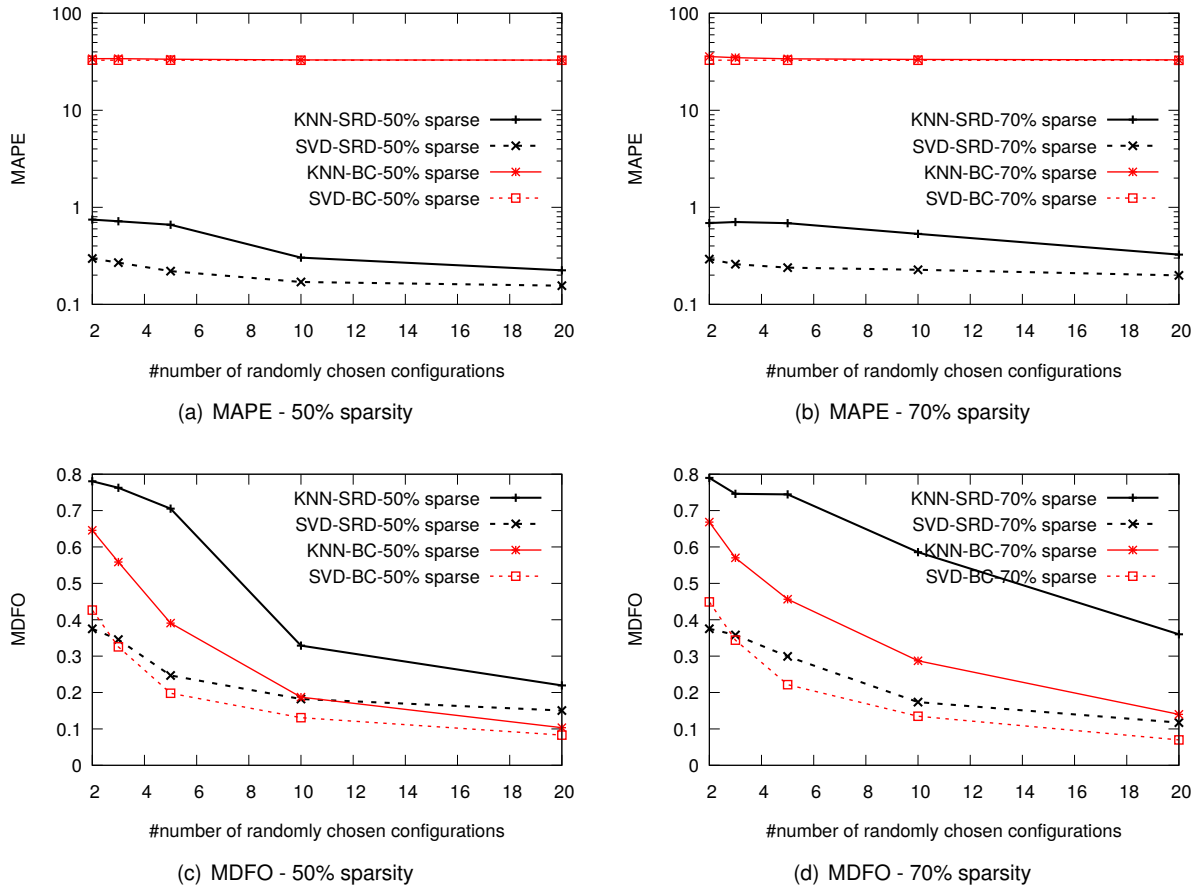
(d) MDFO - 30% sparsity

Figure 4.12: Impact of adding workload information to the original UM

An easily noticeable trait, is that BC results in terms of MAPE seems to follow the same trend for both learns and different sparsity levels: high and constant. In contrast, SRD in terms of MAPE is better when we use SVD even when we add sparsity to the training set. With SVD in terms of MAPE increasing sparsity does not significantly worsens the overall predictions performance. However, this trait is not shared with KNN, which is highly affected by the introduction of sparsity.

KNN achieves its best results when there is no sparsity, with both normalization techniques. It actually achieves better performance in terms of MDFO for no sparsity, compared to SVD. However, if we compare SRD for both learners when we introduce sparsity in the worst case scenario SRD reaches a MDFO of 38%, less than half of the one seen in KNN (80%).

BC achieves good MDFO results for either the learners. Nevertheless, if we compare the results for for 0% and 30% of sparsity the difference is almost negligible for both learners. When we reach



(a) MAPE - 50% sparsity

(b) MAPE - 70% sparsity

(c) MDFO - 50% sparsity

(d) MDFO - 70% sparsity

Figure 4.13: Impact of adding workload information to the original UM

higher level sparsity levels the difference becomes clearer, in which SVD achieves better results for both normalization techniques.

4.3.4 Results Overview

In conclusion, BC is a very effective technique if the problem is finding the optimum values, as we have shown using two different CF techniques. In contrast, if the problem is predicting the absolute values for a certain workload BC has a very poor performance. When looking at the MDFO metric, RD generally outperforms BC in absence of sparsity (especially with KNN). When sparsity is introduced, though, BC achieves significantly higher accuracy levels than SRD. This is due to the fact that SRD suffers of a key drawback: it forces to discard full rows (which may convey valuable information for the learner) if they lack information on a candidate “pivot” configuration.

Regarding the different learners KNN achieves better results with no sparsity, but is outperformed by SVD in terms of MAPE and MDFO when sparsity is introduced.

Chapter 5

ProteusTM Extension: Workload Characteristics

Recommendation Systems (RS) have attracted a lot of research in the past two decades and consequently several algorithms that generate recommendations were developed. In general, the research consolidated the importance of the regular UM (User-Item mapping) used by several algorithms, mainly CF techniques. Although, using UM alone obtains accurate recommendations [14, 57], most recent research propose new recommendation scenarios that go beyond the current UM, by incorporating additional information regarding the profile of the user (e.g., age and sex), of the items (e.g., genre of a movie) and/or context in which users interact (e.g., click or purchase history) [57, 40, 41].

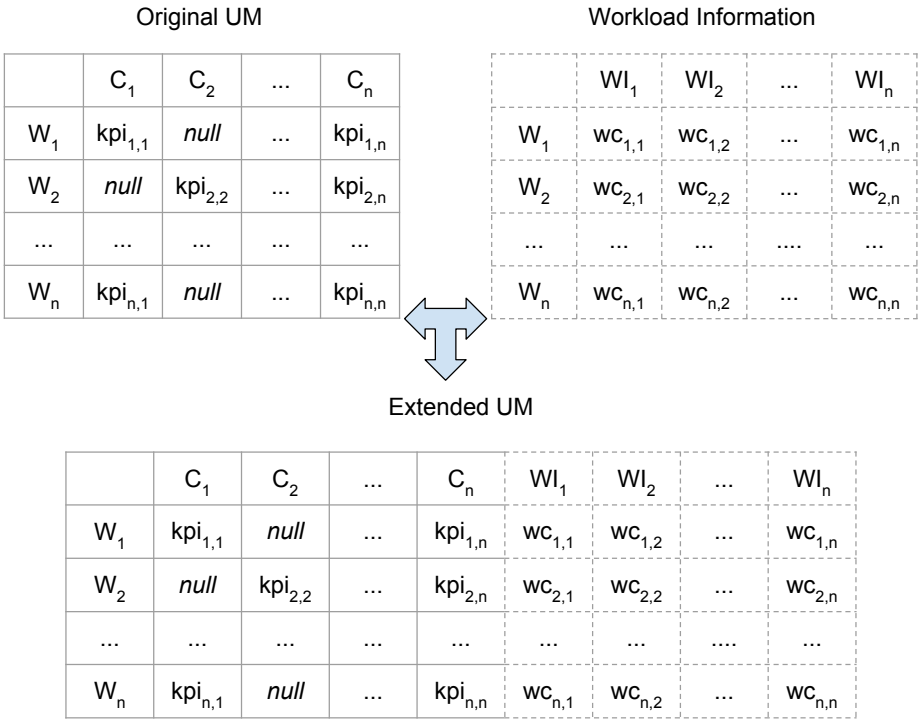
In this chapter we present a study on the impact of adding additional information to the original UM of ProteusTM. More precisely, this study assess the idea of including in the UM, besides the KPIs achieved by the various workloads across the various configurations (which corresponds to the user's ratings for the various items) also different workload characteristics, including average aborts, maximum retries, writes duration.

5.1 Integrating workload characteristics in the UM

ProteusTM UM was designed considering only the CF User-Item matrix, so in order to produce this study we extend the the code to be able to handle the new UM. It is relevant to mention that the additional information will be side user-contributed information, since it will describe the workloads in different workload characteristics.

In order to extend ProteusTM we modify the current implementation that builds the UM. ProteusTM was redesigned to support the introduction of additional information, depicted in Figure 5.1. Starts from from two different data sets one containing the normal User-Item UM (Figure 5.1 Original UM) and another data set of characteristics for the corresponding workload (Figure 5.1 Workload Information). The objective is to merge this two data sets per rows creating the Extended UM (EUM), and give the ability to ProteusTM to know which are the indexes containing the TM configurations and the workload

characteristics.



	C_1	C_2	...	C_n
W_1	$kpi_{1,1}$	*null*	...	$kpi_{1,n}$
W_2	*null*	$kpi_{2,2}$	...	$kpi_{2,n}$
...	...	...	...	...
W_n	$kpi_{n,1}$	*null*	...	$kpi_{n,n}$

	WI_1	WI_2	...	WI_n
W_1	$wc_{1,1}$	$wc_{1,2}$	...	$wc_{1,n}$
W_2	$wc_{2,1}$	$wc_{2,2}$	...	$wc_{2,n}$
...	...	...	....	...
W_n	$wc_{n,1}$	$wc_{n,2}$	...	$wc_{n,n}$

	C_1	C_2	...	C_n	WI_1	WI_2	...	WI_n
W_1	$kpi_{1,1}$	*null*	...	$kpi_{1,n}$	$wc_{1,1}$	$wc_{1,2}$	...	$wc_{1,n}$
W_2	*null*	$kpi_{2,2}$	...	$kpi_{2,n}$	$wc_{2,1}$	$wc_{2,2}$	...	$wc_{2,n}$
...	...	...	...	...	...	...	....	...
W_n	$kpi_{n,1}$	*null*	...	$kpi_{n,n}$	$wc_{n,1}$	$wc_{n,2}$	...	$wc_{n,n}$

Figure 5.1: Graphical example of the merge of original UM with a user-based matrix with workload information/characteristics

A key conceptual problem at the basis of this approach is that it leads to blend in the same row of the UM different types of information, which can be expressed using completely heterogeneous scales. In fact, while the values stored in the original UM refer to the same KPI/metric (although expressed workload-dependant scales), the EUM contains numerical values associated with very diverse domains (e.g., abort rate, transaction duration and throughput).

Hence, a relevant problem to address in order to jointly use KPIs and workload characteristics is how to ensure that the information encoded in the EUM can be meaningfully interpreted by a RS.

We will consider two possible approaches to tackling this problem:

- using different normalization schemes, including i) using the same normalization technique for the whole EUM matrix and ii) normalizing the original UM and the workload info matrix using different normalization techniques. Then, feed the resulting matrix to the CF-based algorithms used by ProteusTM, e.g. SVD or KNN.

- using LibFM, a RS software tool based on FMs designed to support features comprising values belonging to different domains and expressed using heterogeneous scales.

5.2 Integration of LibFM

Following the extension of the UM, ProteusTM can now consider new learners designed to support data beyond the User-Item UM. As mentioned, several algorithms were designed to consider an UM with additional information, so we integrate a new leaner to the Recommender a FMs software tool, namely LibFM [40].

The integration of LibFM software tool [1] with ProteusTM was not direct, first some problems had to be addressed. The first problem was the language compatibility, because LibFM was developed in C++ while RecTM as mentioned was developed in Java. In a similar way to Mahout [51], LibFM is integrated as an external library used by the Recommender to compute the recommendations. The other challenge in the integration, is the data model used by both solutions. LibFM does not use the typical UM instead uses feature vectors (explained Section 2.2.1.4).

Due to good architecture of ProteusTM we create a Java wrapper for LibFM, which is transparently added to the recommender as another choice besides the CF techniques. Further, the Java wrapper addresses the two problems mentioned above, when it receives the UM created by ProteusTM it translates it to a feature vector space. And through the Java Runtime [2] we invoke the libFM library with the corresponding hyper-parameters and the new data model described through means of feature vectors. The hyper-parameters have to be decided a-priori in a configuration file which is used by to set the respective values in the wrapper.

5.3 Evaluation

The objective of this section is to answer this question: how would this additional information affect the predictive capabilities of ProteusTM in a sparse matrix?

This evaluation will use the same metrics and experimental test-bed presented in Section 3.1.1. Further the different workload characteristics were gathered in the same way as the previous data sets, considering the same workloads. Table 5.1 contains the different features, derived from the features of the ProfileTM [6] (mentioned in Section 2.2.2.3) and a brief description.

This evaluation starts by studying the impact on one of the CF techniques, namely KNN. The study on SVD was not conducted due to time constraints since some of the tests using this technique take a lot of time. Afterwards, we present the baseline for the new learner introduced in ProteusTM, libFM and the study with the different features.

As explained and motivated throughout this dissertation CF algorithms have better performance if the KPI is in a homogeneous scale otherwise it clouds the predictor capabilities. The intuitive belief is that adding workload characteristics blindly in a dense matrix it should worsen the predictions since we are adding new columns to the matrix in a completely different scale. But in the presence of sparsity will this new additional information help find better similarity's between workloads?

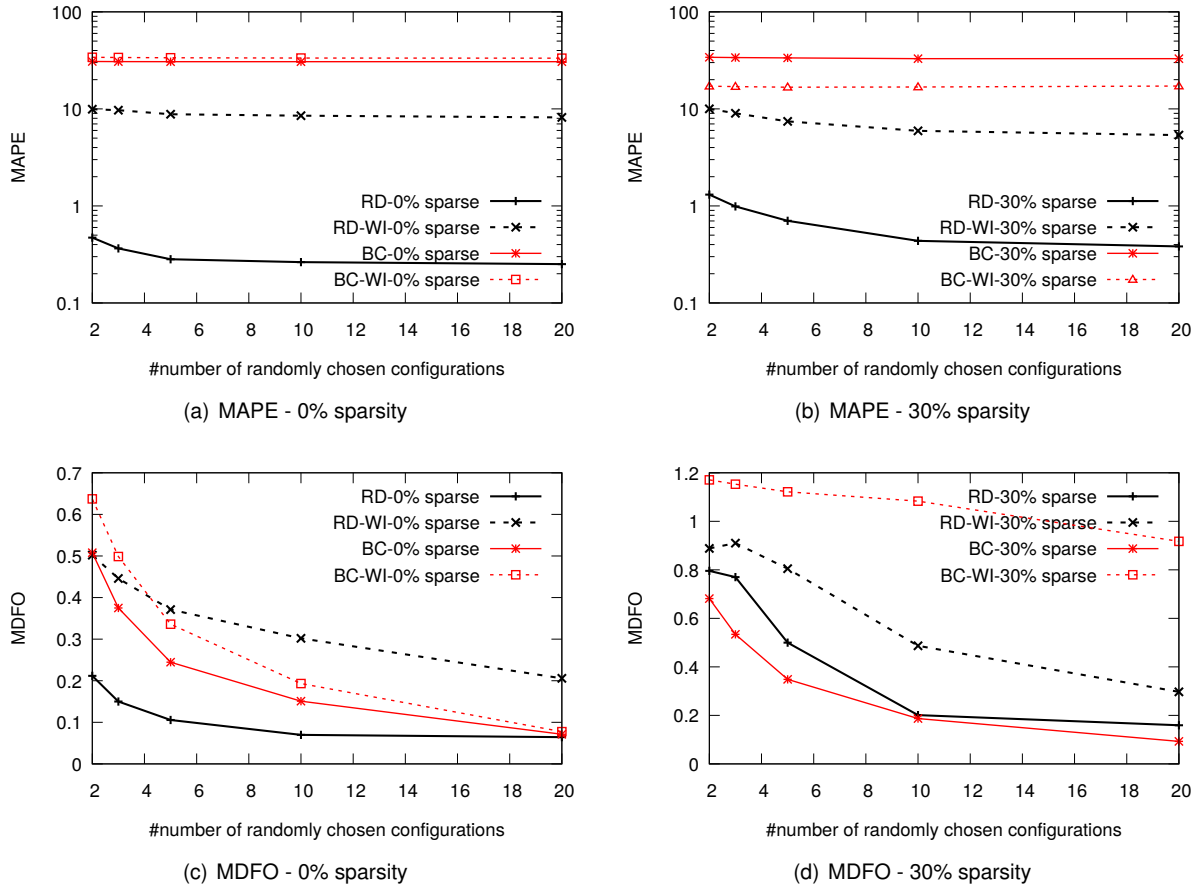
[1] https://github.com/srendle/libfm
[2] https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html

Features	Description
average aborts	average number of times a transaction of aborts
max retries	maximum number of times a TM retried a transaction.
writeXactWriteset_items	distinct items written by a transaction
writeXactReadset_items	distinct items read by a write transaction
ROXactReadset_items	distinct item read by a read only transaction
read_duration	average duration of a read-only transaction
write_duration	average duration of a write transaction
rbfw	number of read operations performed before the first write
tot_wr	total number of writes
tot_rd	total number of reads
wr_per_xact	number of writes per transaction
avg_reads_per_write	average number of reads reads per write
avg_reads_per_ro	average number of reads done by a write/read-only xacts

Table 5.1: Workload Characteristics/Features gathered [5].

5.3.1 KNN with extended UM

In the same fashion as in the previous evaluation, we guarantee fairness by using the same training set and initial configurations, when performing this study on the accuracy of ProteusTM using KNN and the extended UM. We will use the same 30% training sets and 70% test sets.



(a) MAPE - 0% sparsity

(b) MAPE - 30% sparsity

(c) MDFO - 0% sparsity

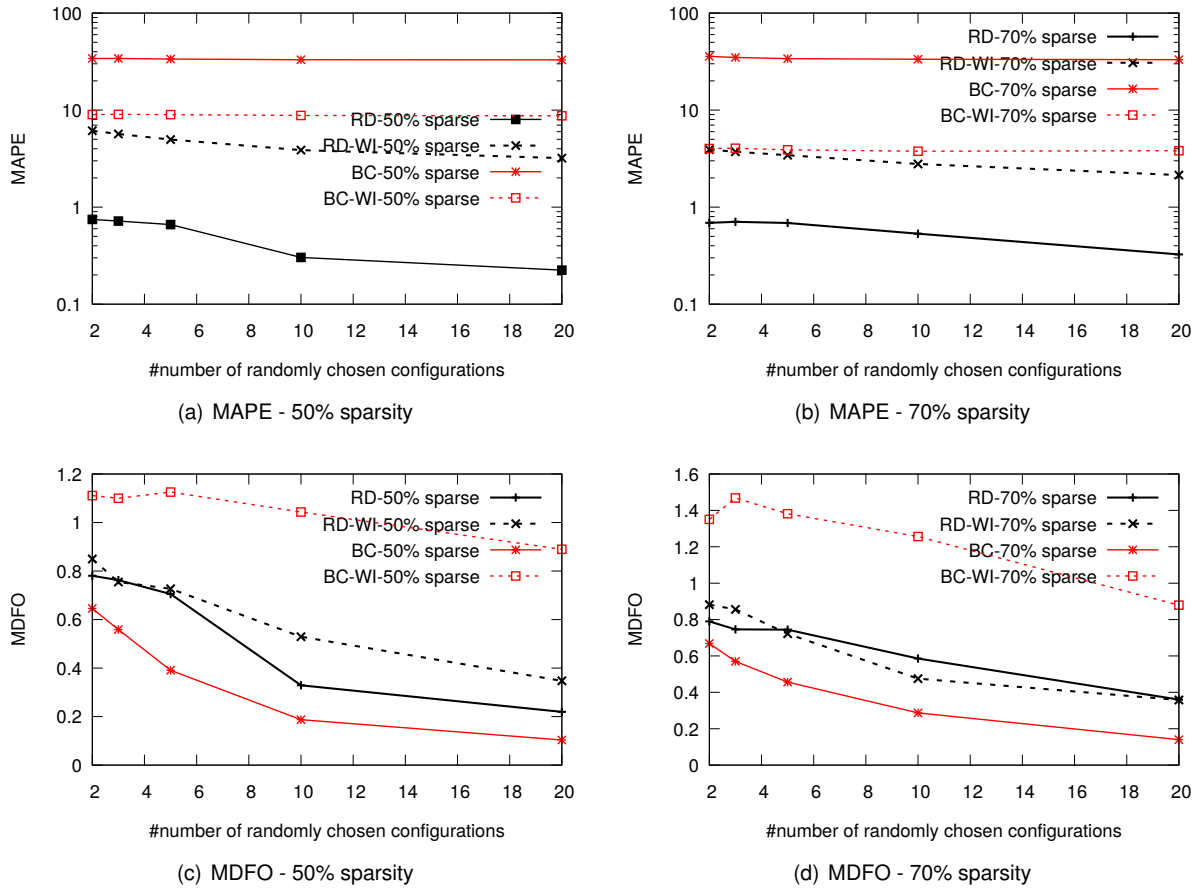
(d) MDFO - 30% sparsity

Figure 5.2: Impact of adding workload information to the original UM for sparsity levels 0% and 30%

The experimented was conducted utilizing the two implementations variants, mentioned in Section

5.1.1. We first analysed the impact of the more direct approach, applying the normalization techniques in the UM and then merge the workload information data set. Secondly, we apply the BC normalization on the full EUM to verify the results. The keys SRD or BC correspond to the the typical UM and the keys SRD-WI or BC-WI are used with regards to the extended UM.

This first approach, Figure 5.2 and 5.3, behaves as expected for 0% sparsity: adding workload information in a dense matrix does not improve the performance of KNN. Focusing on Figure 5.2 if we look closely at the results, we verify that for SRD there seems to be a significant increase on the MAPE and respectively on the MDFO as well, a behaviour that is consistent with the other sparsity levels. The only exception is with 70% sparsity, where both accuracy are very similar and sometimes adding workload information was beneficial (5 and 10 initial configurations).



(a) MAPE - 50% sparsity

(b) MAPE - 70% sparsity

(c) MDFO - 50% sparsity

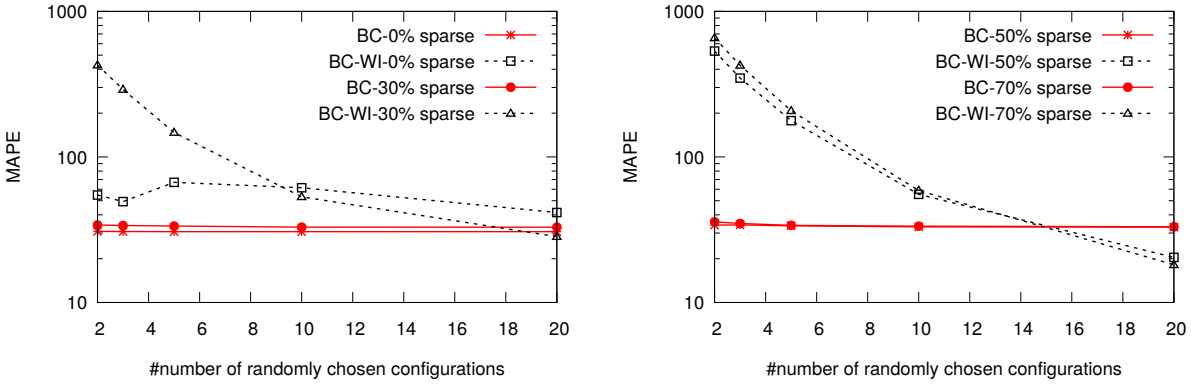
(d) MDFO - 70% sparsity

Figure 5.3: Impact of adding workload information to the original UM for sparsity levels of 50% and 70%.

Regarding BC the MAPE values decrease with additional information, because with the addition of sparsity the scale of the workload information starts to be more imposing, the same as in SRD. This decrease on the MAPE negatively impacts the MDFO, the prediction for the optimal worsens when adding workload information. BC which normally achieves good performance when predicting the optimal configuration for a workload (MDFO), with new information deteriorates in more than 100 times.

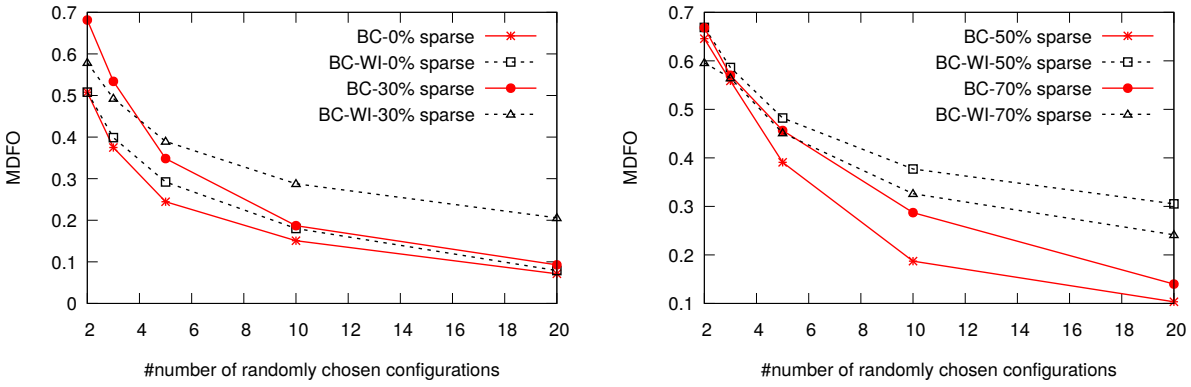
In this first experiment the accuracy's decay is more accentuated with BC than with SRD, but in general KNN accuracy did not improve when adding workload information. Analogous conclusions are attained if one considers an alternative normalization approach, which simply applies BC to the EUM,

avoiding to first normalize the original UM. The corresponding results are shown in Figure 5.4. As we can see, the inclusion of WI, also in this case leads to a significant degradation of the learner's accuracy, both in terms of MAPE and MDFO.



(a) MAPE - 0% and 30% sparsity

(b) MAPE - 50% and 70% sparsity



(c) MDFO - 0% and 30% sparsity

(d) MDFO - 50% and 70% sparsity

Figure 5.4: Impact of normalizing the extended UM with BC.

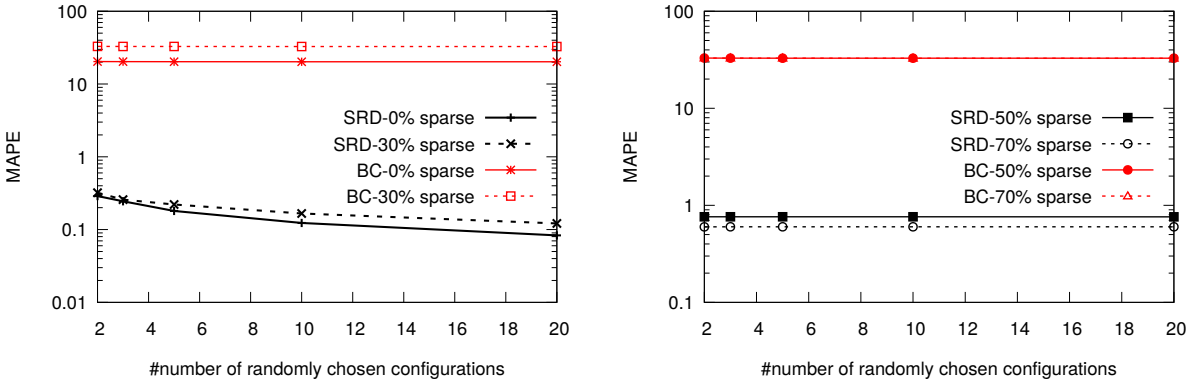
Concluding the study with KNN, the results suggest that overall using the additional information is not improving the predictor performance for either normalization technique.

5.3.2 LibFM

LibFM is the new leaner introduced in ProteusTM, with the ability to handle additional information in the UM. Since this is the first evaluation using LibFM we first provide the baseline,i.e., typical UM with no features and using a 30% training set:

Figure 5.5 shows something different when compared to the other results. While for 0% and 30% sparsity the MAPE values are similar to the ones seen, surprisingly for 50% and 70% sparsity they become constant, an unusual behaviour comparing to the previous learners. And this anomaly, seems to be reflected in the MDFO, for percentages 50% and 70% the normalization is confusing the predictor. Even when the predictor has more knowledge about the workload the accuracy decreases (e.g., Figure 5.6(b) 5 to 10 initial configurations).

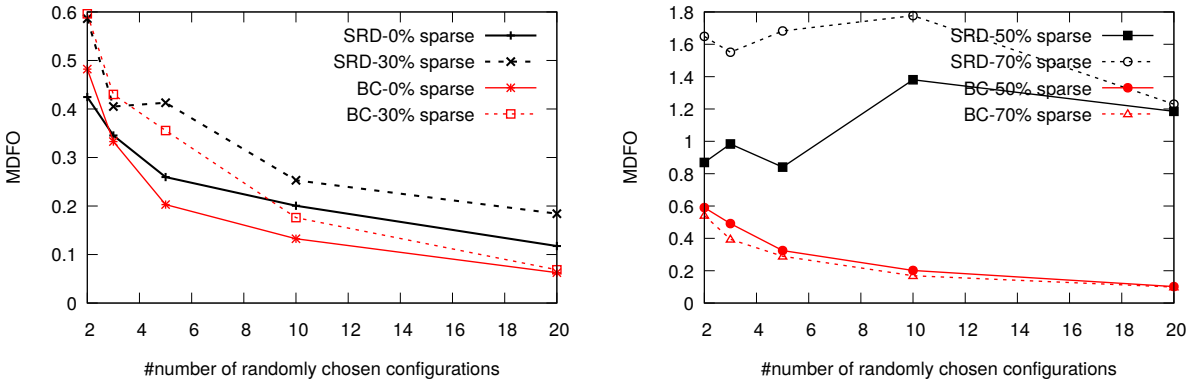
Regarding BC the MAPE shows the same trends as seen before, and this is true for the MDFO as



(a) MAPE for 0% and 30% sparsity

(b) MAPE for 50% and 70% sparsity

Figure 5.5: LibFM - MAPE for different sparsity level of a 30% training set.



(a) MDFO for 0% and 30% sparsity

(b) MDFO for 50% and 70% sparsity

Figure 5.6: LibFM - MDFO for different sparsity level of a 30% training set.

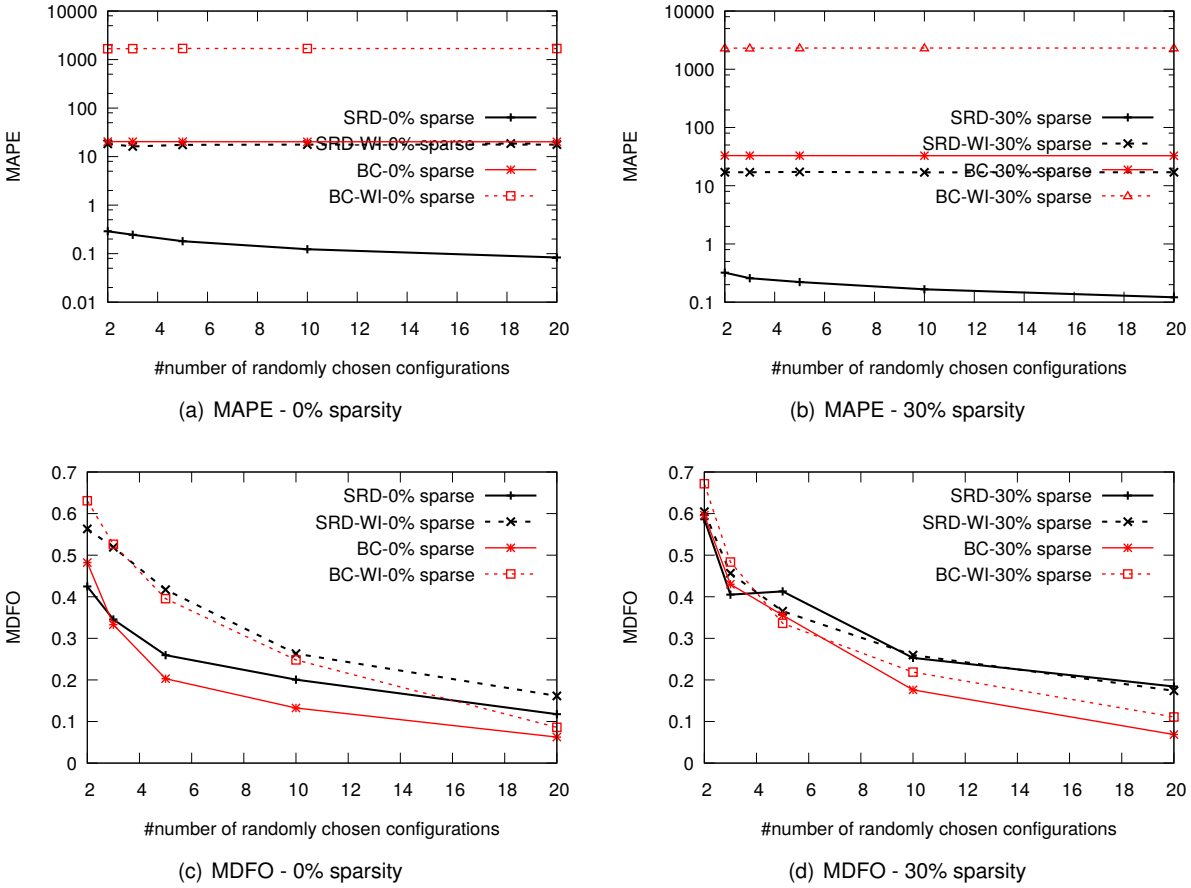
well. In addition, it obtains better MDFO then SRD for all sparsity levels.

To conclude the baseline analysis, this results only strengthens the conclusions attained from the previous evaluation: BC obtains very good accuracy when the problem is identifying the optimal configuration, but in average the predictions are really far from its real absolute value. SRD with regards to LibFM seems to confuse the predictor when the sparsity level is high.

Proceeding with the study, we will now evaluate the impact of using the EUM and this learner. The extended UM is composed by the same 30% training set and the respective thirteen features in Table 5.1. LibFM suffered some problems with the new features, since they were in an extremely heterogeneous scale, it predicted the same value over and over again independent of the test workload. To mitigate the heterogeneous scale in which the characteristics were gathered the first step was the normalization of all the columns between [0, 1], which was achieved by taking a per column normalization. Firstly we find the maximum value $max(C_i)$, and for column i divide each element of the column by the found max value. Thus, introducing a sense of scale on per column basis.

The plots presentation will change compared to the ones presented until now. For each plot we will have a comparison of the baseline of LibFM (Figures 5.5 and 5.6), which only considers the typical UM with either normalization technique (BC or SRD), against the results obtained with the EUM for the same

sparsity level. Once again, the keys SRD or BC correspond to the the typical UM and the keys SRD-WI or BC-WI are used with regards to the extended UM.



(a) MAPE - 0% sparsity

(b) MAPE - 30% sparsity

(c) MDFO - 0% sparsity

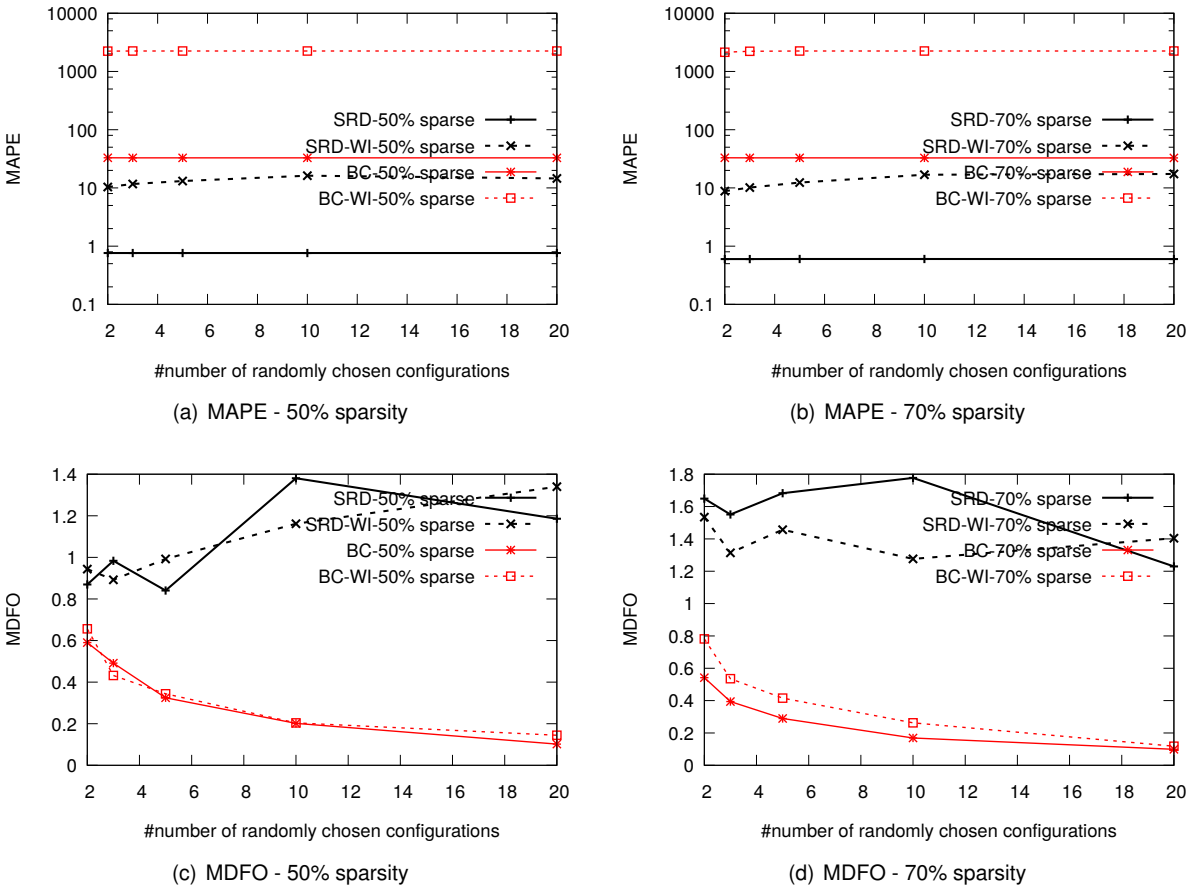
(d) MDFO - 30% sparsity

Figure 5.7: Comparison in the predictors accuracy using the UM and the extended UM (WI) for 0% and 30% sparsity

Figure 5.6 reports the MAPE and MDFO values achieved when considering sparsity values equal to 0% and 30%. For these sparsity levels, the use of workload information does not help; on the contrary, using this additional information even when using libFM is actually confusing the predictor. MAPE for SRD becomes around 100$\times$ worse for both sparsity levels, and BC obtains even worse results. Also considering the MDFO, we observe that better results are achieved, on average, when workload information is not used.

Increasing the sparsity level (See Figure 5.8), does not show improvement regarding the accuracy of LibFM. Overall, SRD normalization seems to be confusing the predictor, which, we argue, may be due to the loss of information imposed by this normalization technique (see Section 4.3.1) when the sparsity level grows. While BC using the typical UM in average always achieves better results then with the extended UM.

Overall, the results presented so far suggest a negative answer to the question: does the joint use of workload information help in increasing the prediction accuracy? Clearly, however, such a negative answer hold only for the data set available for this study, and for the considered approaches to data fusion and normalization.



(a) MAPE - 50% sparsity

(b) MAPE - 70% sparsity

(c) MDFO - 50% sparsity

(d) MDFO - 70% sparsity

Figure 5.8: Comparison in the predictors accuracy using the UM and the extended UM (WI) for 50% and 70% sparsity

This observation led us to consider an additional data pre-processing step: applying a preliminary feature selection phase on the workload information, in order to filter out redundant or non-informative metrics. Next we briefly explain what is feature selection and how we utilize the technique to chose the best sub-set of additional information/features.

5.3.3 Feature Selection

Feature Selection (FS) is the process of choosing a subset of features of an initial data-set in order to the ML method performance [58]. FS is in fact one of the most frequently used techniques in data preprocessing, known for reducing the number of features, removing irrelevant, redundant or noisy data [59].

FS can be divided in three models: filter models, wrapper models and hybrid models [59]. Filter models rely on general characteristics of the data and are independent of the learning algorithm. In contrast, wrapper models optimize the subset of features based on a predetermined learning algorithm and uses its performance as the evaluation criteria. The hybrid model exploits the evaluation criteria of both approaches in different search stages.

We will focus on the filter models in order to identify the best sub-set of features for our data-set. We carefully analyse the features data set and calculate the coefficient of correlation between each

pair of features. Based on the coefficient of correlation, we identify the pairs that are highly correlated and eliminate one of them. We calculate the coefficient correlation, using the Pearson product-moment correlation coefficient (PCC) (also denoted by r) between every pair of features X and Y:

$$r_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{5.1}$$

where cov is the covariance and σ the standard deviation. PCC can give results between [-1,1] where the interval values mean that there is a high correlation, and 0 that there is no linear correlation. With this first preprocessing step, we identified two pairs of features that were highly correlated ($r = 1$) choosing only one feature of each pair, reducing the number of features to eleven (Appendix A).

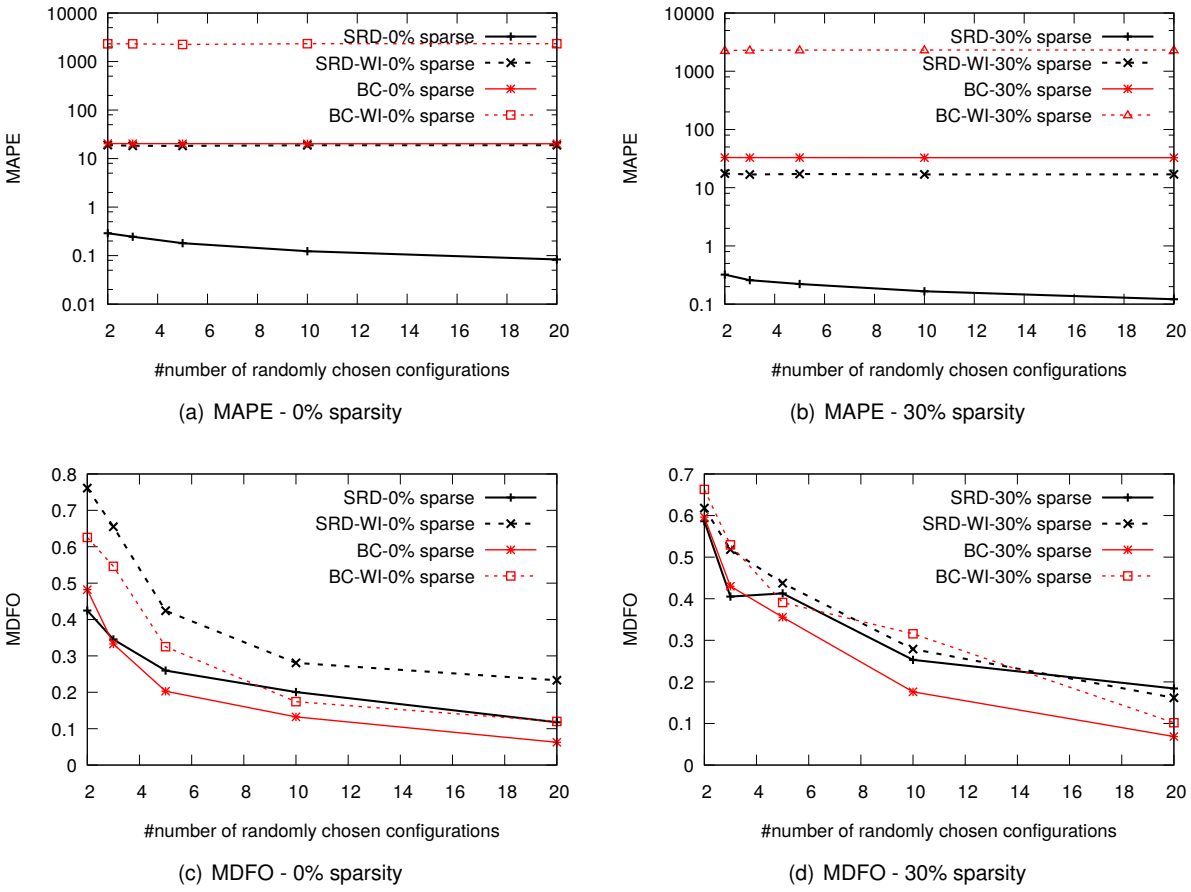
Following this small reduction, we utilize a well known machine learning workbench already mentioned in this dissertation, Weka [56] to apply a filter model to the features, namely Information Gain (IG). IG will be utilized in order to output a feature ranking, in which will be suggesting the most informative features for our training set. Table 5.2 shows the ranking of the features organized from best ranking to worst, with only eleven features:

Features	IG
write_duration	8,0779
average aborts	7,783
max retries	6,9948
writeXactWriteset_items	6,5959
read_duration	6,5434
avg_reads_per_write	6,3687
writeXactReadset_items	6,3687
rbfw	6,3687
wr_per_xact	6,3398
ROXactReadset_items	5,3658
avg_reads_per_row	5,3658

Table 5.2: Features ranking using IG.

In the following we report the MDFO and MAPE achieved when using 11 (Figures 5.9 and 5.10) and 5 (Figures 5.11, 5.12 and 5.13) best ranked features according to the FS process. Results related to the use of the 2 best ranked configurations show analogous trends and can be found in Appendix b.

Unfortunately, also after applying the FS process, the results continue to lead to the same conclusion: at least for the considered information fusion approaches and datasets, the use of additional workload information degrades, rather than enhancing, the accuracy of typical RS algorithms, in particular KNN and LibFM.



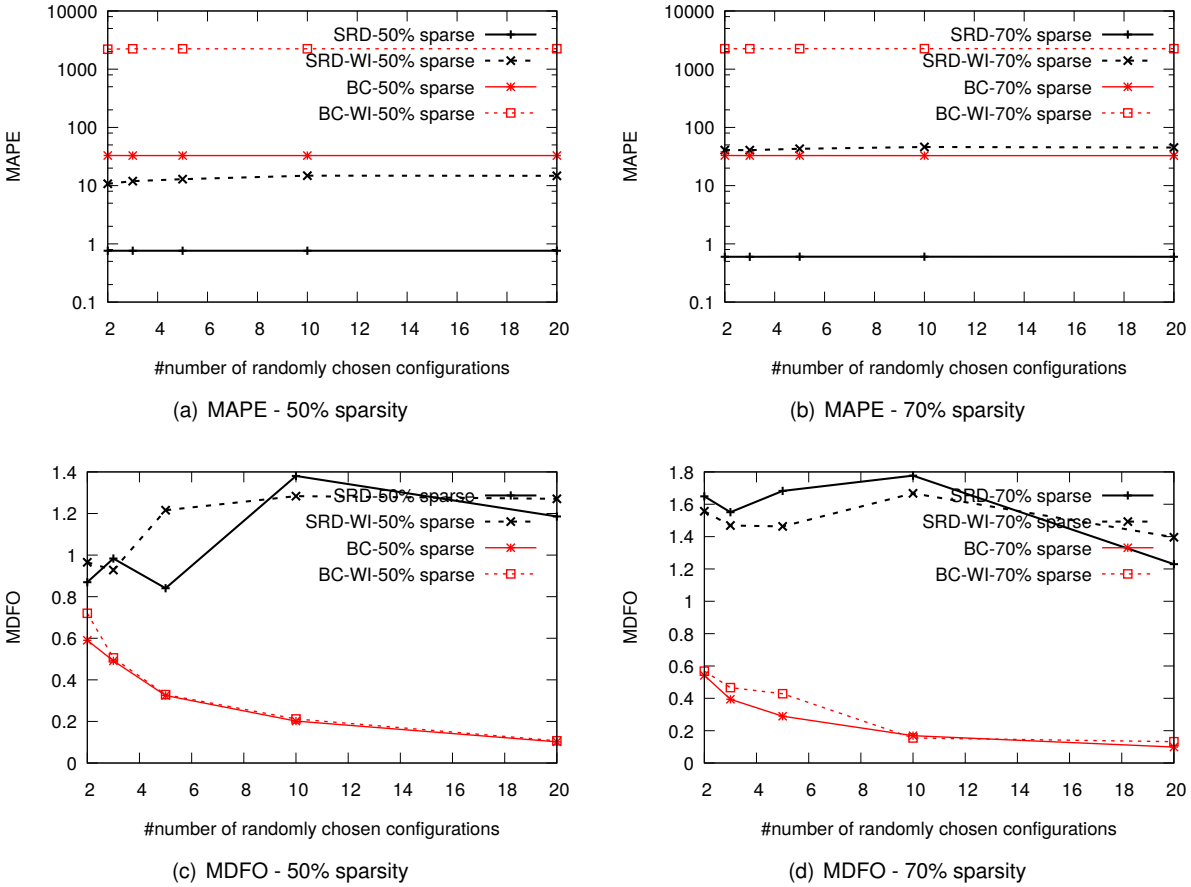
(a) MAPE - 0% sparsity

(b) MAPE - 30% sparsity

(c) MDFO - 0% sparsity

(d) MDFO - 30% sparsity

Figure 5.9: Comparison in the predictors accuracy using the UM and the extended UM with eleven features (WI) for 0% and 30% sparsity



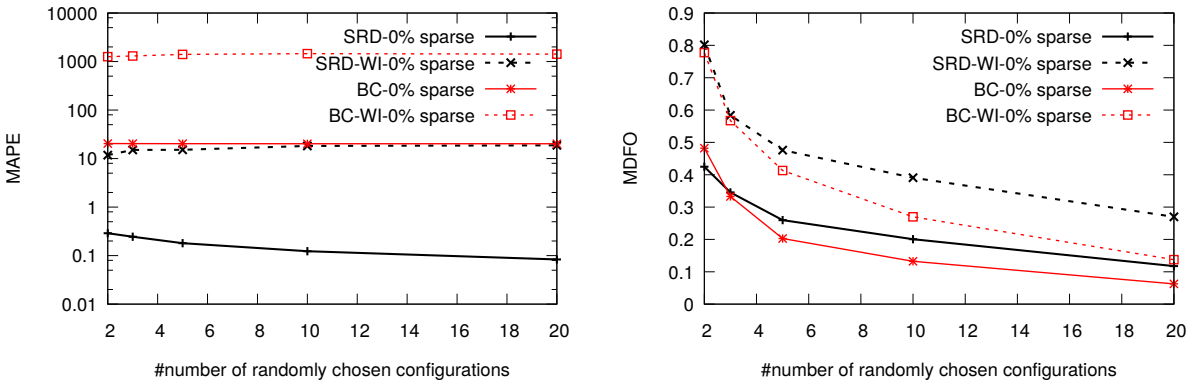
(a) MAPE - 50% sparsity

(b) MAPE - 70% sparsity

(c) MDFO - 50% sparsity

(d) MDFO - 70% sparsity

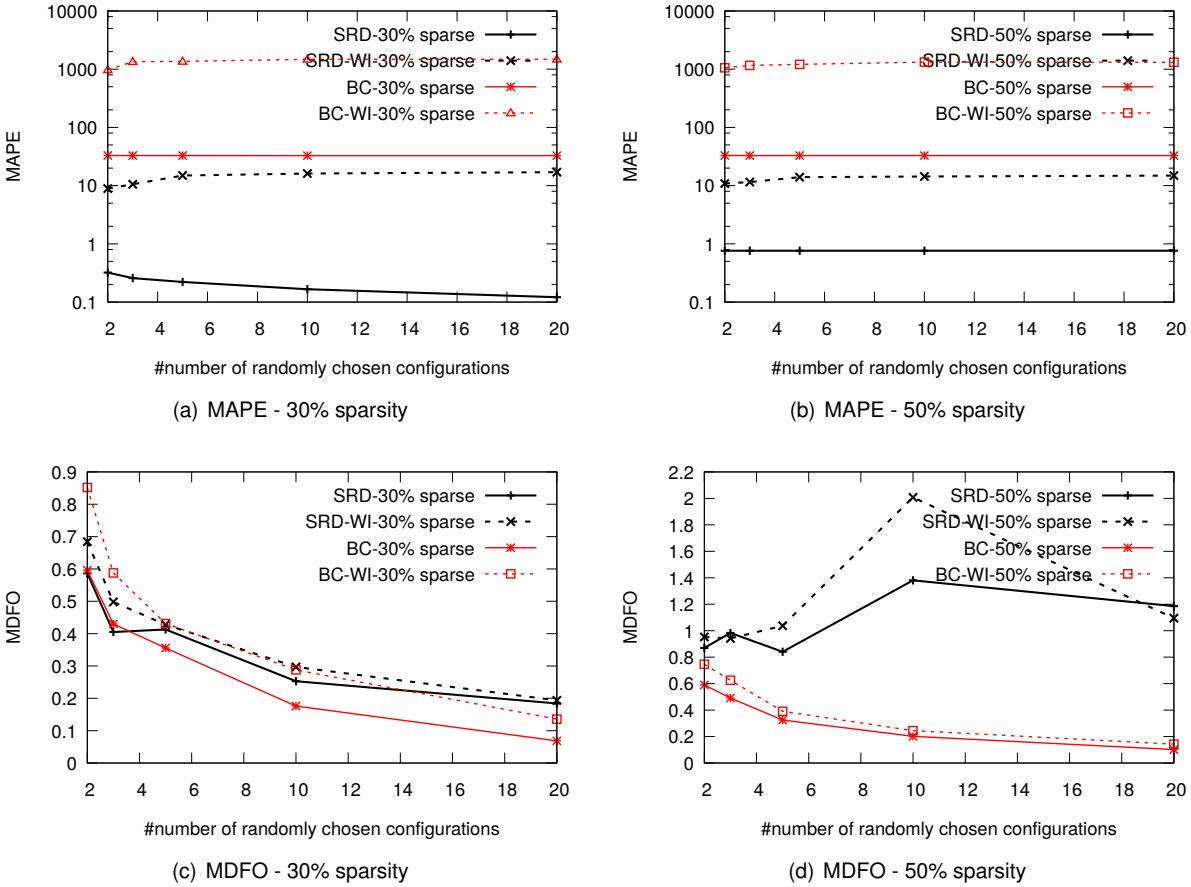
Figure 5.10: Comparison in the predictors accuracy using the UM and the extended UM with eleven features (WI) for 50% and 70% sparsity



(a) MAPE - 0% sparsity

(b) MDFO - 0% sparsity

Figure 5.11: Comparison in the predictors accuracy using the UM and the extended UM with five features (WI) for 0% sparsity



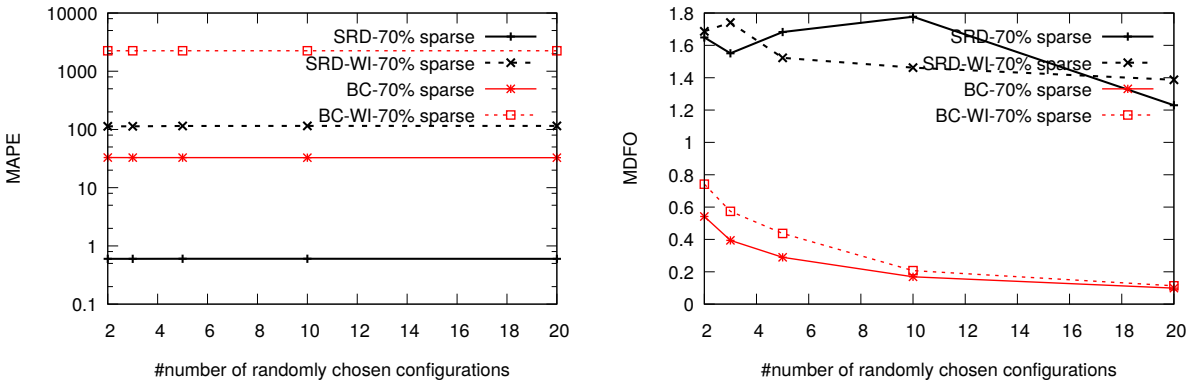
(a) MAPE - 30% sparsity

(b) MAPE - 50% sparsity

(c) MDFO - 30% sparsity

(d) MDFO - 50% sparsity

Figure 5.12: Comparison in the predictors accuracy using the UM and the extended UM with five features (WI) for 30% and 50% sparsity



(a) MAPE - 70% sparsity

(b) MDFO - 70% sparsity

Figure 5.13: Comparison in the predictors accuracy using the UM and the extended UM with five features (WI) for 70% sparsity

Chapter 6

Conclusions

TM is an emerging approach for parallel programming. Unfortunately, the performance of existing TM implementations is known to be very sensitive to the characteristics of application's workloads. Given the large number of existing TM implementations and of their configuration space, several self-tuning techniques have been proposed in the literature for automating the identification of the optimal configuration of a TM run-time system.

This dissertation builds on a recent self-tuning system for TM, called ProteusTM, which has a unique feature in the literature: it is the only self-tuning solution for TM systems that supports dynamic optimization across a *multi-dimensional* configuration space.

In particular, this dissertation investigates two key research questions : i) how to extend ProteusTM to support sparse training sets, and ii) to what extent can the inclusion of workload characteristics (e.g., abort rate) enhance the accuracy achieved by ProteusTM's. We answered the first question by proposing and evaluating the use of two alternative normalization techniques, based on the Box-Cox data transformation and on a novel technique, which we called Sparse Rating Distillation (SRD). We extensively study the predictive capabilities of ProteusTM using both normalization algorithms in various sparsity levels, over a data set with over 300 workloads and 160 TM configurations.

The results highlight that BC is the best technique for ProteusTM, when sparse training sets are used. BC even in very sparse scenarios can achieve results close to the optimum (around 10% from optimal) with both learners. In comparison, with sparsity SRD does not perform as well as BC due to the fact that it forces a drop in the percentage of the training set equal, on average, to the percent of sparsity of the whole data set.

As for the second question, in contrary to our initial expectations, our results suggest that the inclusion of workload information clouds, in a remarkably consistent way, the predictor's accuracy for all the considered data fusion, normalization and learning techniques. Although sometimes adding new features to the UM achieves the same performance as with the typical UM, most of the times the learners under performs.

6.1 Future Work

In this section we present possible future work, in order to extend and improve ProteusTM.

A first direction would be to apply a different approach to integrate the workload information. Instead of extending the UM, the idea would be to leverage on two different learners to predict,i.e., one learner would use the UM and the other learner the workload information. Next, the uncertainty in the prediction of the two learners could be estimated using, e.g., techniques based on bagging [60]. The learner's uncertainty could then be used as decision criterion of a voting scheme aimed at reconciling the outputs produced by the two learners.

Another possible future direction as to do with improving ProteusTM applicability, facing a more business related problem. ProteusTM online sampling does not consider the cost of exploring different configurations. Thus, the challenge here is to consider the trade-off between exploring the proposed configuration because it has high chances of being the better choice, or choosing another configuration with lower EI taking in consideration time and economical cost required to explore a configuration during the on-line learning phase. Hence, the solution should schedule which configurations to explore keeping into account: the past previous exploration (i.e., the currently available hardware platforms) as well as looking ahead to the most likely future explorations, so to plan which would be the most cost effective exploration strategy.

Bibliography

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2006.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[3] 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*, pages 237–246. ACM, 2008.

[4] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, 2014.

[5] D. Didona, N. Diegues, R. Guerraoui, A.-M. Kermarrec, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. In *Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[6] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):54, 2012.

[7] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.

[8] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[9] T. M. Mitchell. Machine learning. *Machine Learning*, 1997.

[10] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[11] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 278–285. IEEE, 2012.

[12] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, 97(9):939–959, 2015.

[13] E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[14] M. D. Ekstrand, J. T. Riedl, J. A. Konstan, et al. Collaborative filtering recommender systems. *Foundations and Trends® in Human–Computer Interaction*, 4(2):81–173, 2011.

[15] X. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009:4, 2009.

[16] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 160–164. ACM, 1994.

[17] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[18] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010.

[19] G. Kestor, R. Gioiosa, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 221–231. IEEE, 2011.

[20] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.

[21] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *ACM Sigplan Notices*, volume 44, pages 141–150. ACM, 2009.

[22] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[23] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 144–157. ACM, 2015.

[24] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[25] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGARCH Computer Architecture News*, 39(1):39–52, 2011.

[26] I. Advanced Micro Devices. Advanced synchronization facility: Proposed architectural specification, 2009. URL http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/09/45432-%ASF_Spec_2.1.pdf.

[27] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE micro*, (2):6–16, 2009.

[28] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

[29] M. Couceiro, D. Didona, L. Rodrigues, and P. Romano. Self-tuning in distributed transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 418–448. Springer, 2015.

[30] 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. *Performance Evaluation*, 69(5):187–205, 2012.

[31] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10. IEEE, 2011.

[32] T. M. Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[33] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[34] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[35] M. E. Harmon and S. S. Harmon. Reinforcement learning: A tutorial. *WL/AAFC, WPAFB Ohio*, 45433, 1996.

[36] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[37] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA, 1995.

[38] J. Bennett, S. Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA, 2007.

[39] Y. Anzai. *Pattern recognition and machine learning*. Elsevier, 2012.

[40] S. Rendle. Factorization machines with libfm. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(3):57, 2012.

[41] S. Rendle. Factorization machines. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 995–1000. IEEE, 2010.

[42] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 476–487. IEEE, 2014.

[43] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE, 2013.

[44] R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.

[45] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing*, pages 475–486. Springer, 2014.

[46] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[47] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. Technical report, 2006.

[48] A. G. Holla and M. Herlihy. Lock elision for memcached: Power and performance analysis on an embedded platform. *Computer Science Department, Brown University*, pages 1–9, 2013.

[49] M. J. Carey, D. J. DeWitt, and J. F. Naughton. *The 007 benchmark*, volume 22. ACM, 1993.

[50] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[51] S. Owen, R. Anil, T. Dunning, and E. Friedman. Mahout in action. greenwich, ct, 2011.

[52] R. Sakia. The box-cox transformation technique: a review. *The statistician*, pages 169–178, 1992.

[53] J. Zhu, P. He, Z. Zheng, and M. R. Lyu. Towards online, accurate, and scalable qos prediction for runtime service adaptation. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 318–327. IEEE, 2014.

[54] F. J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[55] J. Hintze. Ncss statistical software. *NCSS, Kaysville, UT*, 1998.

[56] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[57] Y. Shi, M. Larson, and A. Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)*, 47(1):3, 2014.

[58] A. M. De Silva and P. H. Leong. *Grammar-based feature generation for time-series prediction*. Springer, 2015.

[59] H. Liu and L. Yu. Toward integrating feature selection algorithms for classification and clustering. *IEEE Transactions on knowledge and data engineering*, 17(4):491–502, 2005.

[60] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

Appendix A

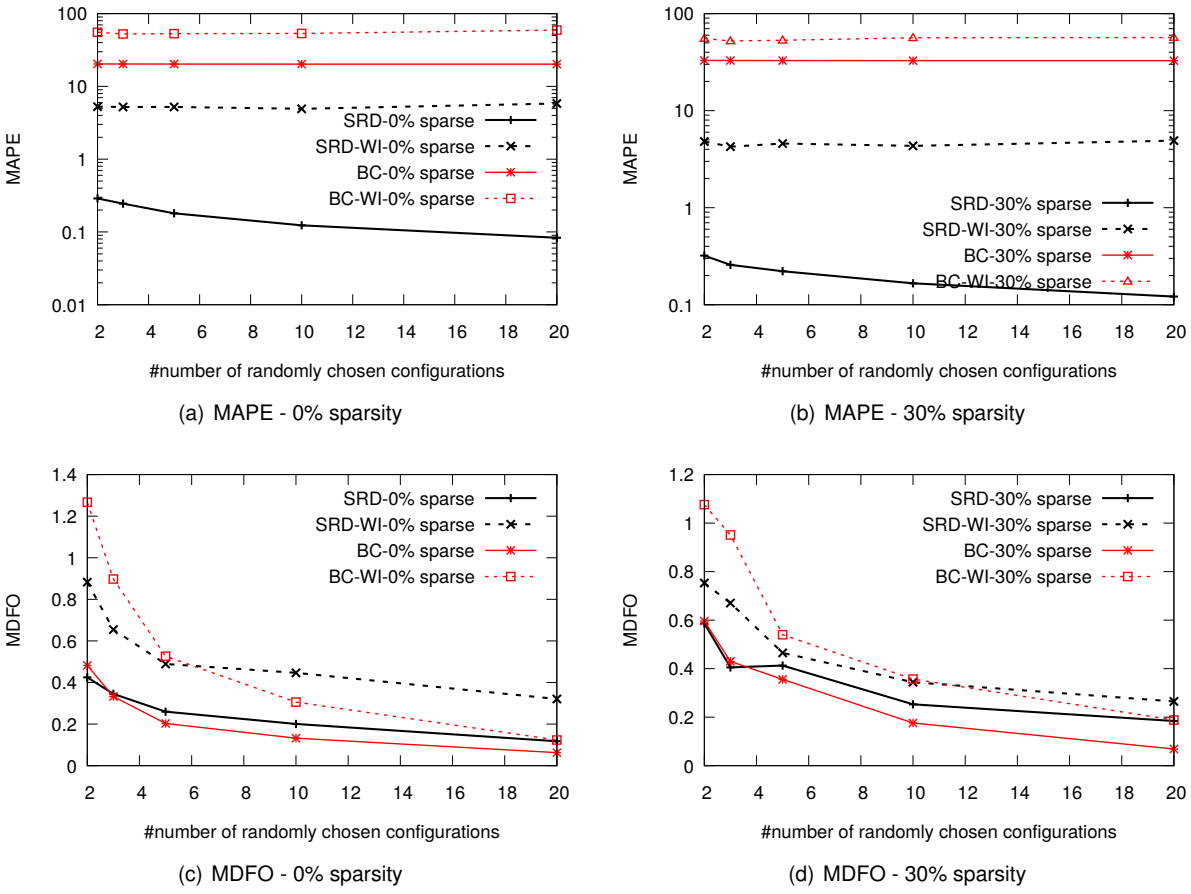
Coefficient of Correlation for the different features

	avg_aborts	max_retries	writeXactWriteset_items	writeXactReadset_items	ROXactReadset_items	ro_duration	wr_duration	rbfw	tot_wr	tot_rd	wr_per_xact	avg_reads_per_wr	avg_reads_per_ro
average aborts	1	0,4935054691	0,2088994105	0,6274372641	-0,0868810942	-0,0597977411	0,0204435431	-0,0100776247	0,2088994105	-0,0100776247	0,2304088897	0,2116014581	-0,0154213755
max_retries	0,4935054691	1	0,6775205824	0,7161489813	-0,1985420783	0,1830880201	0,0572562645	-0,0235132662	0,6775205824	-0,0235132662	0,5686169705	0,5872124033	0,0656516312
writeXactWriteset_items	0,2088994105	0,6775205824	1	0,5926277883	-0,1311006347	0,1184660431	0,0493537229	-0,0154053287	1	-0,0154053287	0,8689879598	0,6084119361	0,1815641199
writeXactReadset_items	0,6274372641	0,7161489813	0,5926277883	1	-0,1447293689	0,0135709328	0,0429217442	-0,0153023258	0,5926277883	-0,0153023258	0,5705506801	0,438711 0868	0,2044454848
ROXactReadset_items	-0,0868810942	-0,1985420783	-0,1311006347	-0,1447293689	1	0,2714623781	-0,0284099466	-0,0082984924	-0,1311006347	-0,0082984924	-0,1232804399	-0,1047111269	-0,0251214042
ro_duration	-0,0597977411	0,1830880201	0,1184660431	0,0135709328	0,2714623781	1	0,0371413441	0,0342965091	0,1184660431	0,0342965091	0,1094239496	0,1258558348	0,0504878978
wr_duration	0,0204435431	0,0572562645	0,0493537229	0,0429217442	-0,0284099466	0,0371413441	1	-0,0134211153	0,0493537229	-0,0134211153	0,0443730677	0,0404324477	0,0178471946
rbfw	-0,0100776247	-0,0235132662	-0,0154053287	-0,0153023258	-0,0082984924	0,0342965091	-0,0134211153	1	-0,0154053287	1	-0,0144777077	-0,0113414192	-0,0045433276
tot_wr	0,2088994105	0,6775205824	1	0,5926277883	-0,1311006347	0,1184660431	0,0493537229	-0,0154053287	1	-0,0154053287	0,8689879598	0,6084119361	0,1815641199
tot_rd	-0,0100776247	-0,0235132662	-0,0154053287	-0,0153023258	-0,0082984924	0,0342965091	-0,0134211153	1	-0,0154053287	1	-0,0144777077	-0,0113414192	-0,0045433276
wr_per_xact	0,2304088897	0,5686169705	0,8689879598	0,5705506801	-0,1232804399	0,1094239496	0,0443730677	-0,0144777077	0,8689879598	-0,0144777077	1	0,5385643912	0,1971129256
avg_reads_per_wr	0,2116014581	0,5872124033	0,6084119361	0,438711 0868	-0,1047111269	0,1258558348	0,0404324477	-0,0113414192	0,6084119361	-0,0113414192	0,5385643912	1	0,0199193499
avg_reads_per_ro	-0,0154213755	0,0656516312	0,1815641199	0,2044454848	-0,0251214042	0,0504878978	0,0178471946	-0,0045433276	0,1815641199	-0,0045433276	0,1971129256	0,0199193499	1

Figure A.1: Coefficient of Correlation for each pair of features.

Appendix B

LibFM with two workload characteristics



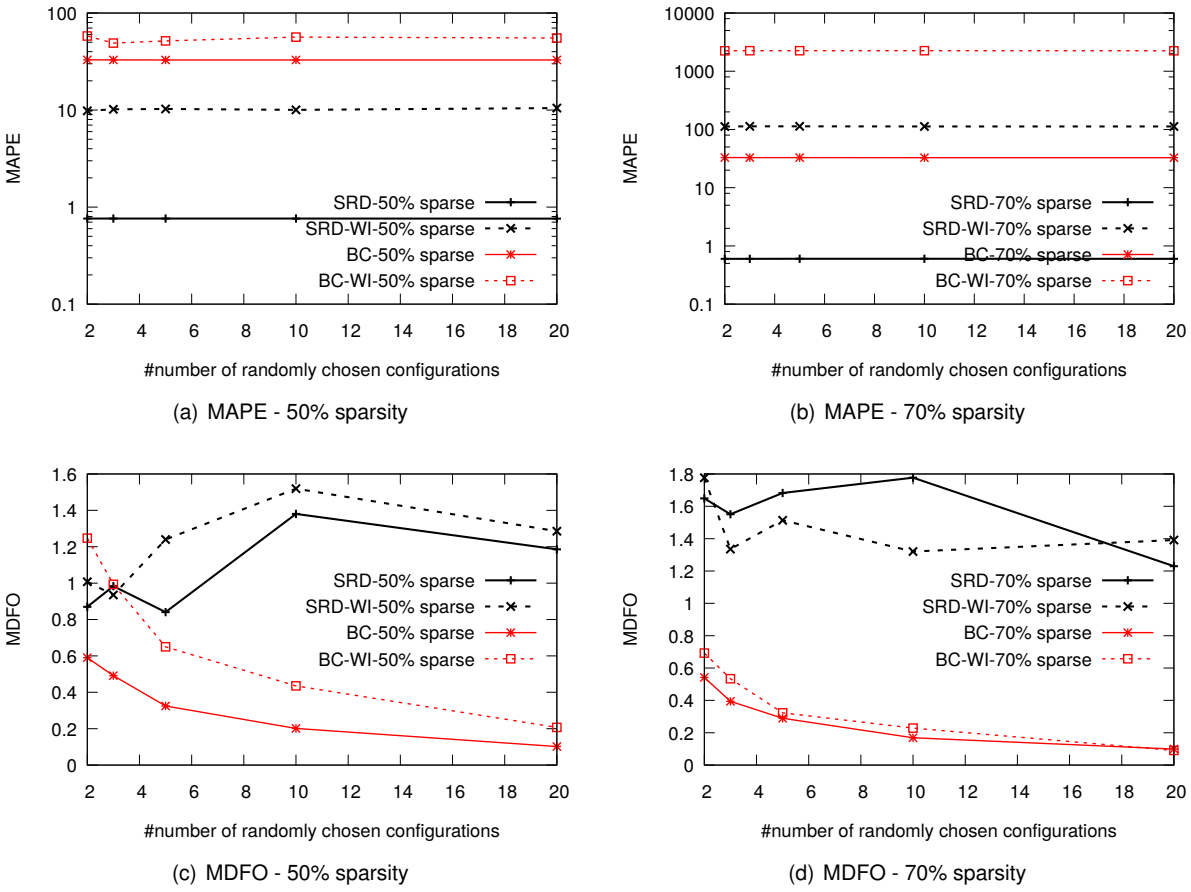
(a) MAPE - 0% sparsity

(b) MAPE - 30% sparsity

(c) MDFO - 0% sparsity

(d) MDFO - 30% sparsity

Figure B.1: Comparison in the predictors accuracy using the UM and the extended UM (WI) with 2 WI for 0% and 30% sparsity



(a) MAPE - 50% sparsity

(b) MAPE - 70% sparsity

(c) MDFO - 50% sparsity

(d) MDFO - 70% sparsity

Figure B.2: Comparison in the predictors accuracy using the UM and the extended UM (WI) with 2 WI for 50% and 70% sparsity