Caches, Transactions and Memories

Models, Coherence and Consistency

YUNYUN ZHU
Computers have brought us inestimable convenience in recent years. We have become dependent on them and more sensitive to their performance. During the past decades, we have been trying to improve program efficiency. The invention of multi-core systems is regarded as the new era of boosting performance of computer programs. When we focus on improving program efficiency, we also need to pay attention to program correctness. In some specific areas, errors, aka bugs, of programs can cause disastrous consequences. The dominant approach to bug detection is testing, which is conducted by executing a program against test cases generated based on scenarios. A bug is found when the output of the program does not match the expected output defined in the test case. One main drawback of testing is that it only shows the presence of bugs. An alternative approach is formal verification, which is a method that can exhaustively analyze the program executions and therefore show the absence of bugs. This thesis focuses on one of the main areas of formal verification - model checking. Model checking analyzes a mathematical model extracted from a program and automatically checks if it satisfies the desired properties.

In this thesis, we first consider verifying safety and liveness properties for transactional memories. In particular, we consider the FlexTM hybrid transactional memory. We build a formal model of FlexTM, and apply a small model theorem that restricts the number of threads and variables in the model. This allows us to reduce the problem of verifying safety and liveness properties of FlexTM to checking language inclusion between the automata of FlexTM and a reference transactional memory. Second, we present a method for automatic verification of cache coherence protocols in the presence of transactional memories. We build a formal model containing a filter that represents the conflict resolution strategies of the transactional memory. We also apply a small model theorem which limits the number of cache lines of the protocol. To check cache coherence, we extend a backward reachability algorithm for infinite state systems, by removing the traces not allowed by the filter. Using this technique, we verify two cache protocols under different transactional memories respectively and conclude that they both maintain coherence. Finally, we consider verification of safety properties of programs running over Self-Invalidate and Self-Downgrade cache coherence protocols. To that end, we define a formal model which captures the weak memory model induced by such protocols. We design an algorithm for inserting a set of optimal fences in the program, which guarantees the safety property while still maintaining the efficiency of a maximal degree.

Keywords: cache coherence protocol, transactional memory, weak memory model, model checking, parameterized system

Yunyun Zhu, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Yunyun Zhu 2018

ISSN 1651-6214
urn:nbn:se:uu:diva-347787 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-347787)
"Learning without thought is labour lost; thought without learning is perilous."
— Confucius
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I  Verifying safety and liveness for the FlexTM hybrid transactional memory —

II Verification of Cache Coherence Protocols wrt. Trace Filters —
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Zeinab Ganjiei, Ahmed Rezine, Yunyun Zhu.

III Verification of Cache Coherence Protocols wrt. Trace Filters —
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Stefanos Kaxiras, Carl Leonardsson, Alberto Ros, Yunyun Zhu.

Extended version of:
Fencing Programs with Self-Invalidation and Self-Downgrade —
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Stefanos Kaxiras, Carl Leonardsson, Alberto Ros, Yunyun Zhu.

Reprints were made with permission from the publishers.
Acknowledgements

First and foremost, I would like to thank my supervisor Parosh Abdulla. It is impossible to put into words how grateful I am for your guidance. You have opened the door to the world of academic research for me. Without your persistent patience of explaining questions to me and discussing all details with me, I couldn’t have done the thesis. You have always been encouraging and have made me believe that I might be doing a better job than I thought. Besides, the discussions of all these work-unrelated topics with you, from low-carbohydrate diets to witch-hunts in early modern Europe, have been inspiring.

I would also like to thank my co-supervisors, Mohamed Faouzi Atig and Ahmed Rezine. Faouzi, thank you for always having put efforts in answering my theoretical and practical questions. Your hard work and thorough thoughts on solving problems have been more than impressive. Ahmed, I have learned how to write a scientific paper in our cooperation of my first publication. Thank you so much for all the remote discussions via Skype when you were in office, at home with your kids running around you or on the train. A thank you goes to my third co-supervisor Bengt Jonsson as well, for having been supportive during my PhD. I appreciate the help from all of you.

It has been a great joy working with my co-authors Sandhya Dwarkadas, Arrvindh Shriraman, Zeinab Ganjiei, Stefanos Kaxiras, Carl Leonardsson and Alberto Ros. Thank you for your collaboration and interesting ideas. A special thank you to Lukáš Holík, who has been extremely helpful with discussions and relevant techniques.

My gratitude also goes to the earlier and current members (in alphabetical order) of the Verification group in which I have been a member: Phi Diep Bui, Jonathan Cederberg, Yu-fang Chen, Aiswarya Cyriac, Frédéric Haziza, Lukáš Holík, Lisa Kaati, Carl Leonardsson, Tuan Phong Ngo, Ahmed Rezine, Othmane Rezine, Amendra Shrestha, Jari Stenman and Cong Quy Trinh. The unique Friday lunches with all of you are unforgettable.

My PhD life would not have been so much fun without my colleagues and friends. I would like to thank Xiaoyue Pan for the endless chit-chats about anything we can think of; Edith Ngai for the company in different sorts of Swedish classes and gym sessions; Sofia Cassel, Raabjerg and Andreína Francisco for the burrito-eating-board-game weekends (which I mostly enjoy remotely); Aleksandar Zeljić for the trips we have experienced together; Philipp Rümmer for letting me play with his camera during some group trips; Nan Guan, Mingsong Lv, Chuanwen Li, Cheng Xu and a lot of Chinese friends here for the parties when I am far away from my homeland; and Ricardo
Alves and Germán Ceballos for attending the teacher training course together. I would like to thank Wang Yi and Mats Daniels for the nice greetings and support during these years; Ulrika Andersson, Ulrika Jareslund and Elisabeth Linqvist (and a lot of other administrative staff) for being helpful whenever I have administrative questions.

A big thanks goes to Anders Berglund who recruited me as an exchange student at the department in 2005, where my horizon was widely broadened and I decided to stay in Uppsala for a much longer time; and Roland Bol who was one of my references when I applied for the PhD, and was always so helpful when I took care of the mails from potential Master students as a part of my teaching duty. Thanks a lot to both of you!

I would like to especially thank my Swedish teacher, Lillemor Aronsson, from whom I have learned a lot of Swedish culture as well as the language during the past years. On top of everything else, Lillemor has helped me with the Swedish summary of this thesis, word by word. Tack så mycket, Lillemor! Many thanks to Sofia Cassel for writing the technical parts and fixing numerous typos in the Swedish summary for me; Carl Leonardsson, Xiaoyue Pan, Palle Raabjerg and Othmane Rezine for proofreading my thesis chapters; and Ricardo Alves and Alberto Ros for explaining the technical details which are not in my research area.

I would also like to thank Wong Kar Wai, whose movies have companied me during the darkest nights in this Scandinavian country; and Jimmy Liao, whose illustrations have provided me with a continuous inspiration of designing PhD thesis covers for myself and my friends - although they might never see this acknowledgement.

Tons of thanks to my parents Xiyu Zhu and Changfen Hu, for their unconditional love and support. They have taught me that one should pursue what one wants in one’s life. I would like to thank my good friends Yan Liu, Mei Chen, Shuyi Yang, Yunjing Mou, Yaxian Tong, Jiahui Mei, Qianjia Zhou, Fanzhen Kong and Jian Huang, who have made my every single visit to China so exciting.

Finally, my profound gratitude goes to Ivan Christoff, who has been a great friend since I started my MSc Program here in Uppsala. His kindness to every student has impressed me as his teaching assistant. He tried his best to convince me to apply for the PhD and has constantly encouraged me whenever I felt frustrated. Thank you, Ivan. I wish you could see this.

This thesis is funded by Uppsala Programming for Multicore Architectures Research Center (UPMARC).
Datorer har gett oss aktningsvärd bekvämlighet de senaste åren. Vi skriver våra arbetsrapporter med ett textredigeringsprogram; vi bokar biljetter och hotell från onlinebyråer för våra resor; eller vi kontakta familj och vänner via videochattjänster när vi reser. Å andra sidan har vi blivit mer känsliga för programprestanda. Vi kan inte tolerera att vänta några minuter när vi försöker öppna en textfil, eller vi förväntar oss inte att en vägnavigationsenhet i bilen visar oss försonad information när vi har kört förbi platsen.


När vi fokuserar på att förbättra program effektivitet, måste vi också uppmärksamma programkorrekthet. På vissa specifika områden kan fel, även kända som buggar, i program orsaka katastrofala konsekvenser. Vi har till exempel inte råd med en krockkudde inte utlöses i en bilkollision eller att syrgasförsörjningen upphör utan varning i en intensivvårdsnhet.


En alternativ approach är formell verifiering, vilket är en metod som kan grundligt analysera programexekveringen och därför visar frånvaron av fel. Mer exakt visar det att ett program är korrekt i förhållande till önskade egenskaper, baserat på en modell som extraheras från programmet. Det finns två huvudområden i formell verifiering, nämligen teoremprovning och modelknotroll.

I fråga om teoremprovning visar experter i matematisk logiskt att modellen av ett program uppfyller en given egenskap via logisk resonemang med hjälp av några andra program, som kallas teoremprövare. Approchen kan vara tidskrävande på grund av det testade programmets komplexitet. Dessutom behöver man förmodligen ibland interagera med teoremprövare för att ge extra information när det behövs. På grund av detta är teoremprovningen begränsad till att användas för vissa speciella system som flygrum eller försvarsindustrin m.m.


En begränsningsfaktor för modellkontroll är tillstånds-utrymmesexplosion, vilket innebär att antalet tillstånd växer exponentiellt med vissa parametrar i det verifierade systemet och det blir till slut för stort för modellkontrollen att hantera. Ibland kan tillstånds-utrymmet till och med bli oändligt. Då skulle verifieringsproceduren aldrig upphöra eftersom det är omöjligt att ens räkna upp alla möjliga beteenden hos en oändlig modell.


I denna avhandling, beaktar vi först att verifiera säkerhet och liveness egenskaper för transaktionella minnen. I synnerhet granskar vi FlexTM hybridtransaktionsminnet. Vi bygger en formell modell av FlexTM, och tillämpar en liten modelltioeorm som begränsar antalet trädar och variabler i modellen. Detta gör att vi kan minska problemet med att verifiera säkerhet och liveness egenskaper hos FlexTM till att kontrollera språk-inkludering mellan automaten i FlexTM och ett referenstransaktionsminne. För det andra presenterar vi en metod för automatisk verifiering av protokoll för cache-koherens i när-
Contents

1 Introduction ................................................................. 15
  1.1 Multi-core Systems .................................................. 17
    1.1.1 Single-core Systems ........................................ 17
    1.1.2 Hardware Problems ......................................... 18
    1.1.3 Multi-core System Organizations .............................. 18
  1.2 Potential Problems ................................................ 20
  1.3 Challenges in Model Checking ..................................... 21

2 Cache Coherence .......................................................... 23
  2.1 Definition of Cache Coherence .................................... 23
  2.2 An Overview of Cache Coherence Protocols ....................... 26
    2.2.1 States and Events .......................................... 27
    2.2.2 Snooping and Directory .................................... 28
  2.3 The MESI Cache Coherence Protocol ............................... 29

3 Transactional Memory .................................................... 34
  3.1 Motivation ................................................................ 34
  3.2 Transactions and the Properties of a Transaction ............... 35
  3.3 More Details of TMs .................................................. 36
    3.3.1 TM Designs ....................................................... 36
    3.3.2 TM Implementation ............................................ 37
    3.3.3 The FlexTM System ............................................ 38
    3.3.4 The TMESI Protocol ........................................... 40
  3.4 TM Safety and Liveness Properties .................................. 45
    3.4.1 Safety Properties .............................................. 45
    3.4.2 Liveness Properties .......................................... 51

4 Memory Consistency Models .............................................. 53
  4.1 What and Why ........................................................ 53
  4.2 Sequential Consistency .............................................. 54
  4.3 Relaxed Memory Models .............................................. 56
    4.3.1 Examples of Relaxed Memory Models ......................... 56
    4.3.2 The SISD Memory Model ...................................... 59
  4.4 Fences ................................................................ 64
5 Contributions ........................................................................................................................................ 68
5.1 Paper I: Verifying Safety and Liveness for the FlexTM Hybrid Transactional Memory [7] .................................................. 68
5.3 Paper III: Mending Fences with Self-Invalidation and Self-Downgrade [4] ........................................................................... 78

6 Conclusion and Future Work ................................................................................................................. 81

References .................................................................................................................................................. 83
1. Introduction

We have to admit that we can hardly live without software nowadays. We write our working reports with a text editing program; we choose hotels from an online platform after reading the comments left there; we contact our family via instant message and video chat services when we travel. Our life has been made easier because of the convenience brought by various software programs. On the other hand (unfortunately), this also means that we have been increasingly dependent on software programs and become sensitive to the efficiency of the programs. We cannot tolerate waiting for ten minutes to save an edited text file, or we do not expect a road navigation device in the car to show us which direction to turn after we have driven past the spot over one kilometer. In other words, we, as users, have performance requirements for software programs.

Computer scientists have been working on the improvement of program performance almost ever since the first modern computer appeared in the world. When integrated circuits replaced transistors, which had replaced vacuum tubes earlier, computers experienced a significant increase in speed. Gordon E. Moore, the co-founder of Intel, predicted that the number of components per integrated circuit would be doubled every year [35], which later on was quoted as 18 months after Intel executive David House predicted the period [28], and it was coined as Moore’s law now. Today a central processing unit (CPU) is still contained on an integrated circuit, but the invention of multi-core systems is regarded as the end of Moore’s Law and the start of a new era of boosting the performance and the efficiency of programs. [48, 49]

Intuitively, a multi-core system has two or more processing units on a single computing component. The processing units, or the cores, execute program instructions concurrently. With multiple cores working at the same time, the speed of program execution is expected to increase since the working tasks are carried out by several cores instead of one. More details of multi-core systems will be introduced in section 1.1.

While the performance of programs is improved, there is another aspect which should attract even more attention, i.e., correctness. More precisely, programs should ensure that they behave as they are expected. All of us have encountered program errors in our life. For instance, an online video clip stops half-way so we have to reload the clip; or a text editing program may not respond when we try to save the file, and we finally end up losing the newly-added content of the file by rebooting the computer. Such errors, or bugs, are not fatal, although they bring us inconvenience. There are some other bugs
which can cause disastrous consequences. We cannot afford that an airbag does not eject in a car collision, or the oxygen supply stops in an intensive care unit when an electrocardiograph machine makes an alarm sound.

The dominant approach to bug detection in software engineering is testing, which is conducted in the runtime environment. During testing, the program is executed against test cases which are generated based on scenarios. A scenario is a hypothetical story about how the user will use the program to achieve a goal. A test case contains inputs, expected outputs and the steps of finishing a part of a scenario. A bug is detected whenever an unexpected output is met. Testing is efficient for discovering bugs. However, it does not ensure that all the executions of the program are covered, which means there can be errors still existing in the program. This cannot be tolerated when life-critical systems are concerned, such as the examples illustrated earlier. Furthermore, in a development life cycle, the earlier a bug is detected, the lower the cost of fixing the problem is. Ideally, all bugs should be discovered as early as possible.

Unlike testing, the formal verification methods are exhaustive. This means that they can show the absence of bugs while testing only shows their presence [13]. Meanwhile, formal verification is conducted at an early stage of the program development. This makes the cost of discovering and fixing bugs cheaper compared with testing that usually happens at a later stage. Formal verification is a rigorous technique which proves the correctness of a program with respect to its desired properties in a mathematical way. It needs to be performed on a mathematical model extracted from a program or a system. We say that the program (system) is correct if its desired properties are satisfied by the model. There are two main areas of formal verification - theorem proving and model checking.

In the case of theorem proving, aka deductive verification, experts in mathematical logic prove that the model of the program satisfies a given property using logical reasoning with the help of some computer programs, which are called theorem provers. Such an approach can be time-consuming due to the complexity of the verified program (system). In addition, one probably needs to interact with the theorem provers at times to provide extra information when necessary. Because of these reasons, theorem proving is limited to be used for some special systems such as aerospace or defense etc.

On the other hand, model checking provides tools that can automatically prove if a model of the program satisfies some desired property. There are two possible outcomes: the tool will either return a "yes" if the property is satisfied by the model; or a "no" and possibly a counterexample if the property is not satisfied. The counterexample is helpful in finding out where the defect is. This thesis focuses on model checking and its application of multi-core systems.
1.1 Multi-core Systems

Imagine in a big mansion, a worker takes care of all the chores - cooking, laundry, cleaning and so on - to ensure that the mansion functions normally. The worker tries to work as fast as possible to accomplish all the tasks, but since there are limits to a single person’s ability, when the workload is heavy, the worker might not be able to finish the chores in time and the mansion would probably not function well (say dinner is not served in time). Suppose now we clone several workers to share the chores with the original one, obviously, the working tasks will be accomplished faster because of the increased number of workers. We can regard the mansion with only one worker as a single-core system, and the mansion with several workers as a multi-core system.

1.1.1 Single-core Systems

As the name implies, in a single-core system there is only one central processing unit (CPU), or simply processor, on an integrated circuit die, or chip. A processor performs computational tasks and can only work with the data in the main memory, which stores data and programs. Two typical operations that a processor usually executes involving the memory are 1) read, which retrieves data from the memory and brings them to the processor and 2) write, which stores data in a memory location from the processor. Sometimes, we also refer to read as load and refer to write as store. A processor fetches the instructions and the data from the memory, performs the tasks according to the instructions and stores the result back to the memory after completing the operation.

One of the challenges for performance improvement is to increase the number of the instructions executed by the processor per unit time. A technique called pipelining is adopted for this purpose. Pipelining separates an instruction into several steps, each of which is executed by a different part of the processor so that multiple instructions can be performed simultaneously (at different steps) like in an assembly line. A step in a pipeline is called a stage.

Another challenge is that the access to the memory is relatively slow, therefore it is important to reduce the frequency of visiting the memory and thus improving the computing performance. To balance the high processor speed and the low memory access speed, there is usually one or multiple levels of caches between the memory and the processor. A cache is a smaller and faster memory which contains recently fetched data from the memory and therefore can speed up the future access to the same data. When data are transferred between the memory and caches, they are transferred in blocks of fixed-length. We call such a block of data in a memory as a memory block. After a memory block is copied into a cache, we call it a cache line. Each cache consists of a number of cache lines. Unless otherwise stated, we assume the issues that a cache is over-sized with cache lines are handled by the software of the system and they are out of the scope of this thesis.
1.1.2 Hardware Problems

As mentioned before, computer scientists have been working on improving the performance of computers during the past decades. One of the traditional methods is to increase the hardware speed of the processor. However, the main problems with this technique are 1) the power density increases with the processor speed, which makes it hard to dissipate the generated heat [12, 21] and will consequently cause some physical problems of the computer components; 2) the speed of data transformation between the processor and the main memory is low despite the high processor speed, which worsens the overall performance of the program execution.

Due to these problems, researchers have also spent some concentration on architectural approaches. One of the approaches is to enlarge the cache capacity. As the intermediate storage device between the processor and a slower memory, a cache enables a faster future data access. With a larger capacity of cache, programs are supposed to run faster. Nowadays there are usually two to three level of caches in a computer. The closer the level of the cache is to the processor, the faster it is. Usually, the processor issues instructions such as read (load) or write (store). If the required data is in the first level of (L1) cache, then the execution of the instruction continues; otherwise, the data is loaded from the lower levels of caches or the main memory to the L1 cache first.

Another way is to increase the parallelism of instruction execution in the processor. Typical techniques can be increasing the stages of pipelines so that the processor can work on more instructions at the same time; or superscalar which enables multiple pipelines within the processor so that instructions can be executed in parallel in the parallel pipelines. However, both methods were reaching a point of diminishing returns by the mid to late 90s. [47]

1.1.3 Multi-core System Organizations

Instead of increasing the complexity within a processor, researchers have now focused on a new direction, i.e., putting multiple processors on the same chip with a shared cache, which is called multi-core. Researches show that the increase of performance is roughly proportional to the square root of the increase of complexity in a processor [12], while the performance of an N-core processor can ideally be N times of the performance of a single-core processor. Furthermore, the power consumption of a larger cache on a chip is much less than that of a more complex processor [47].

There are some variations of the organizations for multi-core systems. The L1 cache is almost always the local cache for each processor, but the level 2 (L2) and level 3 (L3) caches can be on/off the chip and shared by one or more processors. Figure 1.1 shows some different versions of multi-core system organizations. [47]
In this thesis, when we build models (which we will describe with more details later), we assume every processor has an identical local L1 cache, and the other caches (e.g., L2 and L3 caches) are regarded as the last level cache (LLC) or the main memory shared by all the processors, as shown in Figure 1.2. This is because that multiple levels of caches are mainly designed to provide better performance, but the shared caches function the same as the main memory logically. In most cases, when a processor executes a write in-
struction, the change of the relevant data happens in the local cache and the other processors are not aware of it.

1.2 Potential Problems

Now let us go back to the mansion example. Suppose all the chores to be done are listed on a notification board. The workers check the board, find out what tasks are assigned to them, write down the tasks in their booklets and perform them. Each time they finish a task, they report it to the butler who will update the notification board accordingly. Assume workers A, B, and C get the task "putting roses into the vases in all the rooms"; worker D gets the task "cutting 90 blooming flowers in the garden". When worker D finishes the task and tells the butler that the job is done, the butler realizes that there are only lilies cut - roses are not blooming yet. (Let us forget why the butler would assume that roses are blooming without having it checked for the time being). So the task for workers A, B and C on the notification board becomes "putting lilies into the vases in all the rooms". Suppose for some reason worker B checks the notification board again, sees the new task and writes it down in the booklet. But the information that workers A and C keep in their booklets is still old, or stale. If they continue to execute the task, no rose would be found. Similarly, in a multi-core system, if a processor updates a datum in the main memory (in our example worker D reports to the butler then the task on the notification board is changed), while some processors are not aware of the change and thus keep the stale datum in the L1 caches (workers A and C still hold the old task in the booklets), we say the situation is incoherent. A multi-core system is expected to be coherent, or keep the coherence.

Another problem can be that: workers A, B and C get the task of putting roses in all the rooms and write it down in their booklets. The butler later realizes that some tasks need to be changed - including the task of putting roses in the vases - and tells all the workers through walkie-talkie that they should check the notification board again. Worker A is right beside the board and checks it immediately, finding out that the task is still putting roses in the vases. So worker A does not change the information which has been written in the booklet. After a little while, the butler comes to the notification board and changes the tasks. Although the working tasks are updated, since worker A checks the board earlier than the butler updates the information on it, worker A still does not get the right information and consequently will not find any roses. If we look at a multi-core system, it can happen that a processor first loads a datum from the main memory in the L1 cache (worker A writes down the task in a booklet). Later on, the processor needs to read the datum. Since the datum is not changed in the memory yet, the processor does not need to load it and reads it from the L1 cache (worker A does not change the task in the booklet). Afterwards the datum is updated in the memory, which is supposed
to be read by the processor (the task on the notification board is changed after worker A checks it). When the updates of the data are not observed in the order that is intended to be, we call it as inconsistency. Multi-core systems are supposed to be consistent.

These are two typical problems that can happen in multi-core systems. We will introduce more details about coherence and consistency later in section 2.1 and section 4.1. As we can see, these problems would not occur in a single-core system because 1) there is only one L1 cache, thus it is impossible that some L1 caches have an updated copy of data while some still keep a stale copy; 2) instructions are issued one at a time, so it cannot happen that a read instruction does not get the datum which is just updated. In a multi-core system, however, multiple processors work concurrently, each with an L1 cache and there are many possible interleavings of instructions. If the system cannot ensure the coherence and consistency, it might lead to erroneous execution. This brings up the necessity of verification of multi-core systems.

1.3 Challenges in Model Checking

As mentioned before, a model checking tool takes as input the model which describes the system to be tested and verifies if the system satisfies the desired properties. The model is usually a transition system which captures the behaviors of the system. A transition system consists of a set of states (or configurations) together with a set of transitions between the states. A configuration contains the information of different components in the system, e.g., the states of the processes and the values of the local and the shared variables etc. A transition connects two states and shows the relation between them - the system moves from one configuration to another through the transition. A typical transition can be a read or write operation on a variable. Given a transition system, the model checking tool, or the model checker will systematically examine all the possible behaviors of the system. For instance, to check safety properties, all the configurations will be exhaustively checked by the tool.

A limiting factor in the application of model checking is state-space explosion, which means that the number of states grows exponentially with some parameters of the verified system and it finally becomes too large for the model checker to handle. To make matters worse, sometimes the state space can get infinite. A model checker cannot naively enumerate all the possible behaviors of an infinite model - the verification procedure would never terminate then.

There are several reasons why a system may have an infinite state space. One of them is that there can be an arbitrary number of components in a system. For instance, the processes in a multi-core system can be arbitrarily many. This means that the states of the model based on the multi-core system can also be infinite. One way to handle such state spaces is parameterized verification. We first need to build a parameterized model, in which the number of some
components (such as processes) is set as a parameter. An instance of a parameterized system with a fixed number of processes yields a transition system. Parameterized verification is to verify the correctness of the system regardless of the number of components. In other words, it is to verify an infinite set of transition system instances, each corresponding to a fixed number of components. Parameterized systems can also be extended to multiple dimensions, in the sense that there are several parameters where each parameter represents a different kind of components, such as processes or variables. Some of the works in this thesis are based on parameterized verification.
2. Cache Coherence

In this chapter, we will first look at a real-life example which explains the motivation for maintaining cache coherence in a multi-core system. Then we will consider the formal definition of cache coherence followed by an overview of cache coherence protocols, including snooping coherence protocols and directory coherence protocols. At the end of the chapter, we will explain the well-known MESI cache coherence protocol in detail.

2.1 Definition of Cache Coherence

Roughly speaking, cache coherence aims to make the caches in a multi-core system invisible to software and behave as if the system is single-core. This means that all the processors should read the same (latest) value in the local L1 caches. Situations such as a processor accessing a stale value, so-called incoherence, should never happen. The reason why incoherence can occur is that there may be more than one processor which accesses the data in the memory. We have seen a metaphorical example illustrated in section 1.1. Figure 2.1 illustrates an example of incoherence which takes place in a (simplified) multi-core system. In the example, we can see that processor $P_2$ holds the old value of $x$ which may be returned afterwards and consequently cause the program to produce wrong results.

Incoherence can be prevented by using cache coherence protocols (or just cache protocols or coherence protocols for simplicity). A cache coherence protocol defines the rules following which processors should behave. In the example in Figure 2.1, when $P_1$ changes the value of $x$, the protocol should ensure that, say, $P_2$ is notified that the copy of $x$ it holds will get stale and thus takes action (e.g., invalidating the copy of $x$ in its L1 cache). Before we go into the details of cache coherence protocols, we introduce the formal definition of cache coherence.

There are different definitions of cache coherence. For instance, in [19, 20], there are two conditions that must be satisfied for the system to be considered as coherent: 1) every write must be eventually visible to all the processors, and 2) every processor observes the writes to the same memory block in the same order. Another version is [34] which uses the notion of tokens to define

---

1In reality, a processor usually loads a whole memory block which contains more than one variables. But here let us assume that there is only one variable per block, which is enough for explaining the problem without complicating the discussion.
At the beginning, the value of $x$ is 5 in the memory.

Then processors $P_1$ and $P_2$ load $x$ from the memory in the local L1 caches respectively.

Later on processor $P_1$ changes the value of $x$, and now $x$ is equal to 6 in its L1 cache.

However, $P_2$ is not aware of the change and therefore keeps the stale value of $x$ in the L1 cache, which leads to incoherence.

*Figure 2.1. An example of cache incoherence*
cache coherence. The definition states that there are at most \( n \) tokens for each memory block, where \( n \) is equal to the number of the processors in the system. A processor is allowed to write the block when it holds all the tokens, and to read the block when it holds one or more tokens. Since there are only \( n \) tokens, it can never happen that when one processor writes a memory block, another processor also reads or writes the same block.

The definition that we will consider in this thesis is from [44]. Before we look at the formal definition, we need to introduce the notions of logical time and epoch first.

Logical time, or logical clock, is defined by Leslie Lamport in [30]. Intuitively, for two events \( A \) and \( B \), event \( A \) happens before event \( B \) in logical time if any of the following three conditions is fulfilled: 1) event \( A \) occurs before \( B \) in the same processor, 2) event \( A \) is the sending of a message and event \( B \) is the receiving of the message, or 3) event \( A \) happens before event \( C \) and event \( C \) happens before event \( B \). If events \( A \) and \( B \) are not related with the happens-before-relation, then the order between them can be either way.

The lifetime of each memory block is considered to consist of a sequence of epochs, where an epoch is an interval of time during which either 1) a processor can write (and read) the block, or 2) one or more processors can read the block. In other words, when the permission that a processor can operate on the memory block changes, the current epoch ends and a new epoch starts. Figure 2.2 illustrates an example of epochs of a memory block. In the example, processor \( P_1 \) can read and write the memory block during epoch \( a \). When \( P_2 \) is allowed to read and write the block, epoch \( a \) finishes and epoch \( b \) starts. Then \( P_2, P_5 \) and \( P_6 \) are allowed to read the block during epoch \( c \), and so on.

![Figure 2.2. An example of epochs of a memory block](image)

The definition of cache coherence is the following. Formally, there are two invariants of cache coherence:

- First, the single-writer-multiple-reader (SWMR) invariant. More precisely, it defines that at any (logical) time for any memory block, there is either
  - a single processor which may write (and read) the block, or
  - one or more processors which may read it.

- Second, the data value invariant. This means that the value of a memory block at the beginning of an epoch is the same as the value at the end of its last read-write epoch.

To maintain these two invariants, we need a cache coherence protocol. In next section we will explain how such protocols work in general.
2.2 An Overview of Cache Coherence Protocols

As mentioned above, a cache coherence protocol specifies how the processors and the memory should interact to guarantee coherence. More precisely, in a cache coherence protocol (see Fig 2.3, the processors and the memory communicate with each other through an interconnection network to maintain the SWMR and the data value invariants, for every memory block and all the time. Typically, for instance, if a processor issues a load (read) or a store (write) instruction but fails to find the data that it is about to operate on, which is a so-called cache miss, the processor will send a request for the data through the interconnection network. Either the memory or some other processor, depending on the cache coherence protocol, will respond to the request with the data. Sometimes, after a processor changes the data in its L1 cache, it will notify other processors which hold the same data about the change in some way. The memory sends responses and forwards requests when necessary. Every cache protocol has its own rules about how the messages should be exchanged and processed.

![Figure 2.3. An overview of the components in a cache coherence protocol](image)

In a cache coherence protocol, we assume that the behaviors of every processor and the memory are captured by a set of finite state machines (FSMs) which are identical and independent. Each FSM represents a memory block or a cache line, and the states of the FSM indicate the conditions of the block (line) in the memory (processor). When a processor receives a message, e.g., a request for a cache line $A$, it processes the message (say responding the request with the data) and may correspondingly change the state of the FSM representing the line $A$ (or the state of the line $A$ for simplicity). The states of each cache line and the transitions between the states can vary in different cache coherence protocols. We assume that all the L1 caches contain the same set of cache lines, namely the same set of FSMs. Below we will introduce a simple example for an intuitive understanding of cache coherence protocols,
followed by the introduction of two main categories of cache coherence protocols. In section 2.3, we will look at the concrete example of the well-known MESI protocol.

2.2.1 States and Events

We illustrate a simple cache coherence protocol in Table 2.1 [44, 45], where the rows indicate the states of a cache line and the columns are the relevant events that can happen. Each state represents a different condition of a cache line in the L1 cache belonging to some processor. The conditions include that 1) the line is not accessible by the processor (state N); 2) the line can only be read by the processor (state R); and 3) the line can be both read and written by the processor (state W). An event can either be an operation issued by the local processor (load/store), or a request from another processor (e.g., request for read-only permission). The entries of the table are the transitions between the states. They are in the form of Action/State, where Action means the action taken by the processor when an event happens to the cache line, and State is the next state that the cache line will move to. For example, the entry "Request for read-only permission/R" indicates that when a processor tries to load a cache line which is not available in its local L1 cache, the processor will issue a request which asks for the read-only permission of the line and the state of the cache line in the L1 cache of the processor will change to read-only (R). If the action is not mentioned in an entry, then it means that no action is taken upon the event; the absence of the state means that the cache line does not move to any other state, and blank entries indicate that the request is ignored by the processor.

The states in Table 2.1 are called stable states. Typically, in the implementation of a cache coherence protocol, there are also transient states which occur in the transition between two stable states. For instance, a processor cannot access a cache line when it is in state N. Then the processor sends a read-only request for the line and gets the response. Now the cache line state is R. A transient state of this line arises when the processor is waiting for the response, i.e., between states N and R. In this thesis, unless explicitly mentioned, we assume that the events of sending a request, receiving the request, responding with the data and changing the relevant cache line state happen atomically. This means we only need to focus on stable states. Besides cache lines, memory blocks are also marked with states. The states of a memory block indicate the states of the corresponding lines in the L1 caches. A typical example is that if a cache line is inaccessible in all the L1 caches, namely it is in state N, then the state of this block in the memory is N as well. Memory block states are also out of the scope of this thesis.
<table>
<thead>
<tr>
<th>States</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Request for read-only permission/R</td>
</tr>
<tr>
<td>Store</td>
<td>Request for read-write permission/W</td>
</tr>
<tr>
<td>Read-only (R)</td>
<td>Read data</td>
</tr>
<tr>
<td></td>
<td>Request for read-write permission/W</td>
</tr>
<tr>
<td>Read-write (W)</td>
<td>Read data</td>
</tr>
<tr>
<td></td>
<td>Write data</td>
</tr>
<tr>
<td></td>
<td>Send data to requestor and memory/R</td>
</tr>
<tr>
<td></td>
<td>Send data to requestor/N</td>
</tr>
</tbody>
</table>

Table 2.1. General behaviors of a processor wrt. a cache line

2.2.2 Snooping and Directory

There are two main categories of cache coherence protocols: snooping and directory:

- **Snooping protocol**: In a snooping protocol, all the processors are made aware of the notifications sent on the interconnection network. They "snoop" on the network and react following the protocol rules. A notification can be a request for a cache line or the message that the value of a cache line is changed by some processor etc. When a processor is about to write a cache line, for instance, it first sends a notification to the network, forcing the other processors holding the line to invalidate it and then performs the write operation.

- **Directory protocol**: In a directory protocol, the main memory maintains a directory that records the information of every cache line, such as which processors have the read-only or read-write permission of this line etc. Directory protocols make all the requests go through the memory. For example, when a processor needs to access a cache line, it sends the request to the memory. The memory checks the directory and decides which processor should respond to the request. Then the request is forwarded by the memory to that processor, which responds to the requestor with the data. If there is no processor holding the data, the memory itself responds the request by sending the data. Another typical example is that a processor needs to write a line. In such a case, it first notifies the memory, which sends messages to all the processors holding the same copy of the line and forces them to invalidate the copy. The re-
questing processor will not be allowed to write the line until it receives the acknowledgements from all the processors that hold the copy.

Both kinds of protocols have trade-offs, and it is hard to conclude which one is superior to the other without considering the context. Snooping protocols do not scale to a big number of processors due to broadcasting, while directory protocols may take more time to process requests (requests sent between processor, memory and processor instead of between processors. The MESI protocol that we will introduce in next section can be implemented in both ways. [44]

2.3 The MESI Cache Coherence Protocol

We introduce the well-known MESI protocol [37] in this section. The name MESI stands for the four states that can mark a cache line, namely Modified, Exclusive, Shared, and Invalid. The detailed meaning of each state is as follows.

- **Modified**: the cache line is only available in the current L1 cache and is dirty, which means that it has been modified by the processor holding it, but not updated to the memory yet. In this way the value of the corresponding block in the main memory is stale.
- **Exclusive**: the cache line is only available in the current L1 cache and it is clean, which means that the value of it is the same as the value of the corresponding block in the main memory.
- **Shared**: the cache line is available in the current L1 cache, and may also be available in other L1 cache(s). The line is clean in these caches.
- **Invalid**: the cache line is invalid, which means that it is unavailable in the current L1 cache.

Fig 2.4 illustrates the transitions between the states of a cache line held by some processor. In the figure, each transition is labeled with operations and/or messages, where the black texts are the operations and the messages issued by the processor, and the red texts are the messages sent to the processor. Some transitions are of the form $A/B$, where $A$ is the operation issued by the processor or the message sent to it, and $B$ is the message or the operation issued by the processor upon $A$. For example, the transition from $M$ to $I$ in the figure is labeled with GetM/Flush, which means that the processor receives a GetM message, flushes the data (i.e., updates the data into the memory) and moves the cache line to the state $I$.

Before we go into the details of the transitions in the MESI protocol, we need to introduce two messages first:

- **GetS**: stands for GetShared, which corresponds to requesting the read-only permission of the cache line.
- **GetM**: stands for GetModified, which corresponds to requesting the read-write permission of the cache line.
Figure 2.4. Transitions between the states in the MESI cache coherence protocol

We introduce the transitions following the description in [47]. A processor can read or write data. When a processor is about to operate on some data, there are two cases: 1) cache hit, which means that the data can be found by the processor in the L1 cache, or 2) cache miss, namely the processor fails to find the data in the L1 cache. Therefore, combined with read and write operations, there are four possible cases when a processor operates on a cache line, i.e., read hit, read miss, write hit and write miss. We classify the transitions in Fig 2.4 into these four cases as below. We use the form $A \xrightarrow{t} B$ to denote a transition, where $A$ and $B$ are the source and target states respectively, and $t$ is the transition label.

- **Read hit**: a read hit happens when the processor manages to access the cache line and reads the data. The cache state does not change upon a read hit. The transitions corresponding to a read hit are: $M \xrightarrow{\text{Load}} M$, $E \xrightarrow{\text{Load}} E$ and $S \xrightarrow{\text{Load}} S$.

- **Read miss**: a read miss can only happen when the cache line is in the state of Invalid. The processor needs to issue a GetS request for the data. There are different transitions corresponding to a read miss from
the *Invalid* state. The transitions depend on the conditions of the other processors. More precisely:

1. None of the other processors holds a copy of the requested cache line. The processor receives the data, reads the data and moves the cache line state from *Invalid* to *Exclusive*, since the processor is the only one that has the line in its L1 cache. The corresponding transition is $I \xrightarrow{Load/GetS} E$.

2. There is one processor holding the cache line which is in the state *Exclusive*. In this case, this processor will move the cache line state from *Exclusive* to *Shared* ($E \xrightarrow{GetS} S$), since now there will be two processors sharing the cache line. Meanwhile, the requesting processor reads the data and moves the cache line state from *Invalid* to *Shared* ($I \xrightarrow{Load/GetS} S$).

3. There are one or more processors holding the cache line in the *Shared* state. The requesting processor receives the data, reads the data and moves the cache line state from *Invalid* to *Shared* as above ($I \xrightarrow{Load/GetS} S$). The processors which already hold the data do not change the cache line state, since it is already *Shared*.

4. There is a processor holding the cache line in the *Modified* state. Upon receiving a GetS request, the processor flushes the changed value to the memory first. Then it moves the cache line state from *Modified* to *Shared*. This means that the processor only has the read permission of the cache line now, since it is sharing the line with another processor. The corresponding transition in the figure is $M \xrightarrow{GetS/Flush} S$. The requesting processor, same as above, reads the line and moves the cache line state from *Invalid* to *Shared* ($I \xrightarrow{Load/GetS} S$).

- **Write hit**: a write hit happens when the cache line state is not *Invalid* in the L1 cache. We need to consider different states of the line:
  
  1. The cache line is in the state of *Modified*. Then the processor simply writes the data without changing the cache line state, since it already has the write permission of the line ($M \xrightarrow{store} M$).
  
  2. The cache line is in the state of *Exclusive*. The processor writes the value and changes the state of the line from *Exclusive* to *Modified* ($E \xrightarrow{store} M$), meaning that it has the read-write permission of the cache line now.

  3. The cache line is in the state of *Shared*. The processor needs to send a GetM request first. The other processors which also have the cache line in the *Shared* state change the state from *Shared* to *Invalid* upon the GetM message ($S \xrightarrow{GetM} I$), since the data in their L1 caches will be stale. The processor which sends the message
changes the data and moves the cache line state from \textit{Shared} to \textit{Modified} ($S \xrightarrow{\text{Store/GetM}} M$).

- **Write miss:** a write miss happens when the cache line state is \textit{Invalid} in the L1 cache. The processor needs to send a GetM request for the read-write permission of the data. It then changes the data and moves the cache line state from \textit{Invalid} to \textit{Modified} ($I \xrightarrow{\text{Store/GetM}} M$) after it receives the response. There are four possible cases that we need to consider:
  1. None of the other processors has the cache line in the L1 cache. Then the requestor gets the data from the memory and does not need to invalidate any copy of the line.
  2. There is a processor holding the cache line in the \textit{Exclusive} state. Then it needs to move the state of the cache line to \textit{Invalid} ($E \xrightarrow{\text{GetM}} I$), since the data will be stale.
  3. There is one or more processors holding the cache line in the \textit{Shared} state. Similar to the case above, the processor(s) move the state from \textit{Shared} to \textit{Invalid} ($S \xrightarrow{\text{GetM}} I$).
  4. There is a processor having the cache line in the \textit{Modified} state. The processor first needs to update the value of the data to the memory and then changes the cache line state from \textit{Modified} to \textit{Invalid} ($M \xrightarrow{\text{GetM/Flush}} I$).

As we know, in a snooping protocol, messages are sent to all the processors and the memory, and the processors react according to the messages; while in a directory protocol messages are always sent to the memory, which decides whether itself or some processor should respond to the request. The MESI protocol can be implemented in both ways. The transitions in Fig 2.4 are independent of how actually the MESI protocol is implemented. For example, when a processor receives a GetS message for a cache line in state \textit{Exclusive} in its L1 cache, it will change the state to \textit{Shared}, no matter whether the request is sent from the memory or from another processor. In this thesis, when we talk about cache coherence protocols, we only focus on the transitions between the states without exploring the details of how the messages are processed, either by going through the memory first or by broadcasting it to all the processors.

Recall that cache coherence consists of two invariants: 1) SWMR, which means that at any time only one processor can write (and read) a memory block, or one or more processors can read the block; 2) the data value invariant, i.e., the value of a memory block at the beginning of an epoch is the same as the value at the end of its last read-write epoch. In the MESI protocol, the definition of the states ensures that a cache line can only be written by one processor at a time, since if a line is \textit{Modified} in one L1 cache, it must be \textit{Invalid} in all the other L1 caches. Also, a cache line can be in the \textit{Exclusive} state in one L1 cache or in the \textit{Shared} state in one or more L1 caches with all
the other L1 caches having the line in the *Invalid* state. This means that the protocol allows one or more processors to read a cache line at the same time.

The data value invariant is ensured by the transitions of $M \xrightarrow{\text{GetS/Flush}} S$ and $I \xrightarrow{\text{Load/GetS}} S$ (resp. $M \xrightarrow{\text{GetM/Flush}} I$ and $I \xrightarrow{\text{Store/GetM}} M$), since they guarantee that when a read-only (resp. read-write) epoch starts, the previous read-write epoch ends. Meanwhile, the value changed during the previous read-write epoch is obtained by flushing it into the memory. Now we recall the example in Fig 2.1. In the MESI protocol, the cache line states of the data in $P_1$ and $P_2$ will be *Shared* after they both load the data. When $P_1$ changes the data, it will send a GetM request. When $P_2$ sees the message, it will invalidate the data in its local cache ($S \xrightarrow{\text{GetM}} I$). The cache line state in $P_1$ will then become *Modified* ($I \xrightarrow{\text{Store/GetM}} M$). In this way, the incoherent situation that $P_1$ changes the line while $P_2$ holds the stale data will never happen in MESI.
3. Transactional Memory

In this chapter, we will introduce the notion of transactional memory (TM). The aim of TM is to simplify the programming in multi-core systems by atomically executing a sequence of instructions. We will first introduce the motivation for TM followed by the definition and the properties of a transaction. Then we will describe the design choices of TMs in different aspects, including conflict detection, conflict resolution, and version management etc. Following this, we will introduce different categories of TM, including hardware TM, software TM and hybrid TM. After that, we will have a detailed description of a real-life TM system, FlexTM, and TMESI which is the cache coherence protocol in the FlexTM system. Finally, at the end of this chapter, we consider the safety properties such as strict serializability and abort consistency, and the liveness properties such as livelock freedom and obstruction freedom, that a TM is expected to satisfy.

3.1 Motivation

In multi-core systems, multiple programs are running concurrently, where each of them is executed on a processor. Here we refer to such a program in execution as a process. An important issue of multi-core systems is to ensure the mutual exclusion property [17], which requires that the shared resource can be accessed by only one process at the same time. Typically, when a process accesses the resource, it is in a critical section, which is a specific program part where the process can access the shared resource. A simple way to guarantee mutual exclusion is using locks to control when a process is allowed to access the resource. More precisely, a process needs to require a lock to enter a critical section, and it releases the lock after it leaves the critical section. A process holds the lock when it is in a critical section, and in such a manner it blocks the resource. The other processes have to wait for their turn to get the lock. However, using locks may cause some problems [27]. A typical example is priority inversion, which happens when a higher-priority task has to wait for a (or more) lower-priority task(s) blocking the shared resource that it needs, possibly for an unpredictably long time. Another potential problem is deadlock, which means that there is a set of processes, each of which holds some resource and waits for the resource held by another process to finish some task. This results in the situation that all the processes are suspended from functioning.
In 1993, transactional memory (TM) was proposed as an alternative to locks [27]. TMs simplify the programming in multi-core systems by executing a group of instructions atomically. The name of transactional memory comes from a concept of database system programming, since the transactions in TM satisfy some properties as the transactions in databases do. As described in [27], TMs do not need to use locks and thus can avoid the problems mentioned above. Today there are TMs retaining locks. Some examples can be TL2 [16], McRT-STM [39], or JudoSTM [36]. But TMs still address the problems such as simplifying the programming in multi-core systems and making traditional lock-based mechanisms more concurrent [25]. In next section, we will introduce more details about transactions and the properties that a transaction should satisfy.

3.2 Transactions and the Properties of a Transaction

A transaction is a finite set of instructions, such as load or store, executed by a process or a thread. A thread is a component of a process which consists of a sequence of instructions and can co-exist with other threads in the same process. We assume that all processes are single-threaded and we say that a transaction is executed by a process or a thread for the same meaning logically. Observing from outside, the set of instructions appear inseparable and executed all at once. More precisely, a transaction satisfies the following properties [24]:

- **Atomicity**: requires that a transaction either commits, which means that all the instructions in the transaction have been successfully executed and other processors can observe the changes made by the transaction immediately after the commit; or aborts, which means that all the tentative changes made by the transaction are restored and it appears that the transaction has not made any changes.
- **Isolation**: requires that the instructions are not visible by other processes during the execution of the transaction. In other words, the execution of a transaction can never be interfered by the transactions running on other processors (except that a transaction can be aborted half-way, which we will discuss in details in section 3.3.1).

Another relevant property is consistency [24]. However, the definition of consistency depends on the semantics of the system, and it is meaningless to discuss it without considering the context. In this thesis, we only consider the properties of atomicity and isolation, which is the same as [27] where TM was formally suggested for the first time.
3.3 More Details of TMs

In general, a set of transactions are running concurrently in a TM system. We assume that one process executes only one transaction at the same time. A transaction may access or change the shared memory locations. During the execution, a conflict happens when both 1) a transaction writes to a memory location, and 2) one or more other transactions read from or write to the same location [24]. A transaction can commit if it has not encountered any conflicts, or the conflicts cannot stop it from committing. Otherwise the transaction aborts and will retry later on. It is the design of the TM system that decides if a transaction can commit upon a conflict. In this section, we will introduce different options of TM designs and the categories of TM implementations, together with the illustration of a TM system FlexTM [43] and the corresponding cache coherence protocol.

3.3.1 TM Designs

A TM design specifies, for instance, when a conflict should be detected, at which granularity a conflict will occur, how conflicts should be resolved and which transaction should commit or abort when it conflicts with other transactions etc. There are many alternatives of TM designs. We introduce the main designs in the following aspects based on [23] and [24].

- **Conflict detection** identifies when a conflict happens.
  - The tentative (or eager) detection identifies conflicts during the execution of transactions. For instance, if a transaction $t_1$ reads from a memory location and later on another transaction $t_2$ writes to the same location before $t_1$ commits, then the conflict is detected when $t_2$ writes to the location.
  - The committed (or lazy) detection identifies conflicts between a running transaction and a transaction that is about to commit. Typically, the committing transaction will check 1) which running transactions have read from or written to the memory locations that it has written to, and 2) which transactions have written to the locations that it has read from. Conflicts will be detected at that time.

- **Conflict resolution** decides which transaction should abort regarding the detected conflicts.
  - The pessimistic (or eager) approach resolves a conflict once it is detected. In other words, during the execution of the transactions, when a conflict is eagerly detected, one of the involved transactions will abort.
  - The optimistic (or lazy) version waits until one of the conflicting transactions is about to commit, and then decides which transaction should abort, though the conflicts can be detected earlier (i.e., during the execution of the transaction).
• **Conflict granularity** specifies at which level a conflict is detected. It can be a cache line (e.g., both transactions write to the same cache line), an object, or a part of a data-structure etc. Note that when the granularity is a cache line, two transactions may conflict even if they do not access or change the same variable, since there can be more than one variable in a cache line.

• **Version management** specifies how the tentative changes of a transaction are managed. There are two main approaches:
  - The **direct (or eager)** version management: a transaction keeps the older data in an *undo-log* and changes the memory locations directly. If the transaction aborts, the previously-changed memory locations will be restored with the older values in the log.
  - The **deferred (or lazy)** version management: a transaction keeps the tentative changes in a *redo-log* and updates the memory with the log at commit time. It will abandon the redo-log if the transaction aborts.

Every TM design option has both advantages and disadvantages. It is difficult to conclude which option is better-performed without taking the contextual constraints into account. For example, lazy conflict resolutions are more suitable for the programs with frequent conflicts while eager resolutions are used more if conflicts do not occur so often. The FlexTM system that we will introduce in next section adopts lazy version management and eager conflict detection, but enables both eager and lazy conflict resolution.

### 3.3.2 TM Implementation

There are three main categories of TM implementation: *hardware TM (HTM)*, *software TM (STM)* and *hybrid TM (HybridTM)*.

The first HTM was proposed in [27]. Most HTMs are implemented based on the modification of some existing cache coherence protocol, such as MESI or MOESI, to match the TM context. Usually, there are two kinds of cache line states in the extended protocols. One is the standard states in the traditional protocol; the other is the transactional states reflecting the fact that the cache line is being operated by a transaction. Typically, when a processor issues a transactional instruction, the relevant cache line will change to a transactional state. When the transaction ends (either commits or aborts), the cache line state will change back to one of the standard states. We give an introduction of a typical example, namely TMESI, which is an extension of the MESI protocol, in section 3.3.4.

STM was first proposed in [42]. As the name implies, STMs are developed with software techniques to support transactional operations. Compared with HTMs, STM users have full-freedom in implementing different strategies (of conflict detection and resolution etc) without any cost of changing the
hardware; while HTMs lack choices of strategies as they are already designed in the hardware. Furthermore, the transaction sizes in HTMs are limited due to the fixed size of the L1 caches, while STMs do not have such limitations. However, STMs have worse performance than HTMs due to the overheads, for instance, of keeping logs which record tentatively read and written values in transactions.

HybridTMs combine the two approaches and try to avoid the disadvantages of either one. There are various kinds of HybridTMs. For example, some can switch between software and hardware modes [33]; some support HTM executions but the transactions are executed in software when the hardware resource is exceeded [15, 29]; and some have hardware which accelerates the performance of STM [14, 40]. In the FlexTM system, hardware detects and records conflicts, while software chooses the mode (lazy or eager) of conflict resolution.

3.3.3 The FlexTM System

In order to describe the main idea behind a TM, we introduce a real-life TM system, namely FlexTM. We focus on the details of how FlexTM detects and resolve conflicts, and ignore issues such as the implementation details that are out of the scope of the thesis.

FlexTM enables both transactional and conventional instructions at the same time. A conventional instruction is an instruction such as load or store issued by a processor without the context of transactions, and a transactional instruction is an instruction from a transaction. To start with, we consider the context only with transactional instructions involved for simplicity. We will describe how FlexTM manages conventional instructions at the end of this section.

In FlexTM, each processor keeps:

- A read signature ($R_{\text{sig}}$), which records the cache lines read by the transaction that is currently running in this processor.
- A write signature ($W_{\text{sig}}$), which records the cache lines written by the transaction that is currently running in this processor.

Furthermore, each processor keeps three Conflict Summary Tables (CSTs) which record the conflicts between the local transaction and the other transactions. They are:

- **Read-Write conflict table (R-W)**, which records the transactions whose $W_{\text{sig}}$ has non-empty intersection with the $R_{\text{sig}}$ of the local transaction. In other words, the transactions in the R-W table write to the same cache line as the local transaction reads from.
- **Write-Read conflict table (W-R)**, which records the transactions whose $R_{\text{sig}}$ has non-empty intersection with the $W_{\text{sig}}$ of the local transaction.
• Write-Write conflict table (W-W), which records the transactions whose \(W_{\text{sig}}\) has non-empty intersection with the \(W_{\text{sig}}\) of the local transaction.

During the execution of a transaction, when it issues a load or a store instruction, it also updates its \(R_{\text{sig}}\) or \(W_{\text{sig}}\) accordingly. FlexTM adopts eager conflict detection which identifies conflicts during the execution of transactions. More precisely, in FlexTM, when a transaction issues an instruction, the other running transactions will be notified about it. If there are conflicts arising, the involved transactions will send messages about the conflicts to the one that issues the instruction. On receiving the messages, the transaction issuing the instruction will update the CSTs based on the content of the messages. Meanwhile, when a transaction receives the notification about the instruction issued by another transaction, it checks its own \(R_{\text{sig}}\) and \(W_{\text{sig}}\), sends a message and updates its CSTs if there is a conflict. We will discuss more details about the messages concerning the conflicts in section 3.3.4 where we introduce the TMESI protocol.

FlexTM detects conflict on the level of cache lines. Since there can be more than one variable in each cache line, two transactions can conflict even if they process different variables. For instance, transaction \(t_1\) writes to variable \(x\) and transaction \(t_2\) writes to \(y\). If \(x\) and \(y\) are in the same cache line, then \(t_1\) and \(t_2\) conflict although they write to different variables.

FlexTM allows both lazy and eager conflict resolution. In the lazy mode, conflicts are not resolved until the committing time. Suppose \(t\) is a transaction in FlexTM. When \(t\) is about to commit, it traverses its W-R and W-W conflict tables and aborts all the transactions that are logged in the two tables, so that these transactions would not continue executing with the stale data after \(t\) has committed with the new data. For all the transactions that are in the R-W conflict table of \(t\), \(t\) needs to remove itself from the W-R tables of these transactions. For example, if transaction \(t'\) is in the R-W table of \(t\), then \(t\) needs to delete itself from the W-R table of \(t'\). In this way, when \(t'\) commits afterwards it would not need to abort \(t\) in vain, as \(t\) has already committed.

When \(t\) aborts, it empties all its CSTs and, in the same way as when it commits, removes itself from the W-R tables of the transactions that are logged in the R-W table of it.

In the eager mode, when a conflict is detected, it is the FlexTM system that decides which transaction should abort. When a transaction commits in the eager mode, its CSTs should be empty since all the conflicting transactions have already aborted.

When a transaction finishes, either commits or aborts, and either in the lazy mode or the eager mode, its \(W_{\text{sig}}\) and \(R_{\text{sig}}\) are emptied.

As we have mentioned before, FlexTM supports both transactional and conventional instructions. If a conventional instruction such as load or store appears within a transaction, it is treated as a transactional instruction. If a conventional store instruction tries to write to the cache line that is in the \(W_{\text{sig}}\) or the \(R_{\text{sig}}\) of a transaction, then the transaction will abort and retry after the store
instruction finishes. If an ordinary load instruction hits the $W_{sig}$ of a transaction, then from outside it appears that the load instruction happens before that transaction, since the tentative changes by the transaction are still invisible at the moment.

3.3.4 The TMESI Protocol

In this section, we introduce the TMESI protocol which is the cache coherence protocol in the FlexTM system. TMESI is extended based on the MESI protocol. Compared with MESI, TMESI has two more states, two more instructions, one more message and several more transitions because of the additional instructions and states. We describe them in detail as below.

The two extra states in TMESI are:

- **TMI**, which indicates that the cache line is written by a transaction. In other words, the cache line is in the $W_{sig}$ of the transaction. If a processor has a cache line in the TMI state, then all the other processors may have the same line in the state of TMI, TI or I.

- **TI**, which means that 1) the cache line is in the $R_{sig}$ of the transaction; and 2) when the transaction reads the cache line, some other transaction has the same line in the TMI state. If a processor has a local cache line in the TI state, then the other processors hold the same line in the state of either
  - TMI, TI or I; or
  - TI, S or I.

We will explain how cache lines move to the states mentioned above with more details later in this section.

Fig 3.1 illustrates the transitions between the six states of the TMESI protocol. The transition labels are in the same format as the ones in MESI (Fig 2.4). I.e, A/B indicates operation/message; the black texts are the operations and the messages issued by a processor, and the red texts are the messages sent to a processor. The bold texts are the new messages and operations in TMESI compared with the MESI protocol. We have also made the transitions arrows bold as emphasis.

There are two additional instructions in TMESI compared with MESI:

- **TLoad**, which stands for transactional load, namely a load instruction issued from a transaction

- **TStore**, which stands for transactional store, namely a store instruction issued from a transaction

There is also one more message in the TMESI protocol, TGetM, which stands for Transactional GetModified. When a processor issues a TStore instruction and encounters a write miss, it sends out a TGetM message to request for the read-write permission of the cache line. A processor sends out a GetS message to request for the read permission of the cache line when it issues a
**Figure 3.1. Transitions between the states in the TMESI cache coherence protocol**

*TLoad* or a Load instruction and encounters a read miss, which is the same as in the MESI protocol. If the requested cache line is in the $W_{\text{sig}}$ of some other transaction, the processor will receive the data together with a *Threatened* message. An example is the transition $I \xrightarrow{\text{Load/GetS}(T)} I$, where $T$ stands for the *Threatened* message that the requestor receives. We will introduce more details about *Threatened* messages below.

The TMESI protocol keeps all the transitions in MESI. Besides, there are also transitions with transactional instructions involved. In a similar manner as section 2.3, we introduce these transitions in four classifications, namely read hit, read miss, write hit and write miss.

- **Read hit**: which means when a transaction issues a *TLoad* instruction, it is able to access the cache line in the local L1 cache. The cache line state does not need to change upon a read hit. The corresponding tran-
sitions between the standard states are similar to the ones in the MESI protocol, i.e., \( M \xrightarrow{TLoad} M, \ E \xrightarrow{TLoad} E \) and \( S \xrightarrow{TLoad} S \). Furthermore, the TMESI protocol also has a transition with a transactional state involved: \( TMI \xrightarrow{TLoad} TMI \).

- **Read miss:** namely a transaction tries to read a cache line which is in the *Invalid* state in the local L1 cache. There are two possible cases, depending on whether the line is in the *TMI* state held by some other processor.

  - There are no other processors holding the same line in the *TMI* state. In this case, the transitions upon a read miss in the TMESI protocol are the same as the ones in MESI:
    
    * \( I \xrightarrow{TLoad/GetS} E \) if none of the other processors holds a copy of the requested line.
    
    * \( I \xrightarrow{TLoad/GetS} S \) if there is one processor having the cache line in the *Exclusive* state, which moves the line to the *Shared* state after receiving the GetS message from the requesting processor (\( E \xrightarrow{GetS} S \)).
    
    * \( I \xrightarrow{TLoad/GetS} S \) if there are one or more other processors holding the line in the *Shared* state. The processors do not change the cache line state in this case.
    
    * \( I \xrightarrow{TLoad/GetS} S \) if there is a processor holding the line in the *Modified* state. The processor changes the cache line state to *Shared* after flushing the data in the main memory upon the GetS message (\( M \xrightarrow{GetS/Flush} S \)).

  - There is one or more other processors holding the cache line in the *TMI* state, which means that the requested line is in the *Wsig* of the transactions running on these processors. The requesting processor will receive the data as well as a threatened message, indicating that the same line is being written by some other transactions. After getting the reply, the requestor will move the cache line state from *Invalid* to *TI* (\( I \xrightarrow{TLoad/GetS(T)} TI \)), and the line has to move back to the *Invalid* state when the transaction ends. The cache lines in the *TMI* state do not need to change the state since they are in transactions which are isolated from outside before the transactions commit or abort.

Besides what we have introduced above, there is one more transition wrt. read miss: \( I \xrightarrow{Load/GetS(T)} I \). It happens when a processor issues a conventional load instruction and encounters a read miss. Meanwhile some other processors have the cache line in the *TMI* state. So the requestor receives the data and a Threatened message, but still stays in the *Invalid* state, since the cache line in the *TMI* state may move to the *Modified*
state later on. This transition does not have transactional instructions involved, but it happens in TMESI because of transactions. This is why we also introduce it here.

- **Write hit:** namely when the transaction issues a `TStore` instruction, it manages to access the cache line in the local L1 cache. There are two possibilities, depending on whether the hit cache line is in a standard state or in a transactional state.

  - If the line is in one of the standard states, then the transitions are similar to the write hit transitions in the MESI protocol, except that the cache line state becomes `TMI` instead of `Modified`:
    - If the line is in the *Modified* state, then the transition is `M \xrightarrow{TStore} TMI`.
    - If the line is in the *Exclusive* state, meaning that the cache line is now written by a transaction, then the transition is `E \xrightarrow{TStore} TMI`.
    - If the cache line is in the *Shared* state, similar to the procedure in the MESI protocol, it first sends a `TGetM` message. The processors holding the same cache line in the *Shared* state will invalidate the line when they receive the message (`S \xrightarrow{TGetM} I`), since the data in their L1 caches will be stale. The processor which sends the message moves the cache line state to `TMI` (`S \xrightarrow{TStore/TGetM} TMI`).

  - If the cache line is in a transactional state, and
    - If it is in the *TMI* state, then the state does not change since it is already regarded as a written line in the transaction (`TMI \xrightarrow{TStore} TMI`).
    - If it is in the *TI* state, then the state changes to the `TMI` state after it sends a `TGetM` message (`TI \xrightarrow{TStore/TGetM} TMI`). If there are processors holding the same line in the *Shared* state, they should invalidate the line upon the `TGetM` message since the data will be stale (`S \xrightarrow{TGetM} I`).

- **Write miss:** namely when a transaction tries to write a cache line which is in the *Invalid* state in the L1 cache. Similar to the MESI protocol, the requestor needs to send a `TGetM` message for the data and moves the state of the cache line to `TMI` after receiving the response (`I \xrightarrow{TStore/TGetM} TMI`). There are two possibilities depending on the state of the available line(s).

  - The cache line is unavailable or available in a standard state. The scenarios are similar to write miss in the MESI protocol:
    - If none of the other processors has the line in the local L1 cache. Then the requestor simply gets the data from the memory.
* If a processor has the line in the Exclusive state, it needs to invalidate the state upon the message ($E \xrightarrow{TGetM} I$).
* If one or more processors have the line in the Shared state, all these processors need to invalidate the cache line like above when they receive the message ($S \xrightarrow{TGetM} I$).
* If a processor holds the line in the Modified state, then it needs to update the value of the line to the memory first, and then invalidates the cache line ($M \xrightarrow{TGetM/Flush} I$)

For the transactional states, there are two possible cases:
* All the other processors have the cache line in the state of $TMI$, $TI$ or $I$. Then these processors do not change the state of the line, as the operations to the data within a transaction is still invisible and isolated.
* All the other processors have the cache line in the state of $TI$, $S$ or $I$. Then the processors holding the line in the Shared state will invalidate the cache line upon the TGetM message ($S \xrightarrow{TGetM} I$).

Note that the processors in the FlexTM system can hold a cache line in the state of $TI$, Shared or Invalid at the same time, following the scenario below: at first the processors have the cache line in the state of $TMI$, $TI$ or Invalid. Then the transactions which have the $TMI$ line abort and the cache line state becomes Invalid. Later on, another processor which has the cache line in the Invalid state issues a load instruction for the line. The cache line state of the requestor will move to Shared in this case ($I \xrightarrow{Load/GetS} S$), since the cache lines in the $TI$ state will move to Invalid no matter whether the transactions abort or commit. The coherence of the caches is kept anyway.

In the figure, we can see that there are also two dashed arrows which represent the transitions of commit and abort respectively. The arrows are between the two rectangles outside the states instead of between the states. This is because that when a transaction commits or aborts, the change of the cache line states happens on a batch of cache lines, instead of on one line per time, like a conventional operation in a cache coherence protocol. However, the state transitions wrt. commits and aborts of each cache line still hold, and we treat them like all the other transitions:

- $TMI \xrightarrow{Commit} M$
- $TI \xrightarrow{Commit} I$
- $TMI \xrightarrow{Abort} I$
- $TI \xrightarrow{Abort} I$

Furthermore, there is a transition $TMI \xrightarrow{GetM} TMI$, indicating that a cache line in the $TMI$ state does not move to other states upon a GetM message, which is sent by a conventional store instruction when a write miss happens. As we have mentioned before, when a conventional store instruction tries to
write a cache line that is already in the $W_{\text{sig}}$ of a transaction, the transaction has to abort. In the implementation of FlexTM, the transaction is aborted by a triggered handler on the thread where the transaction is running. This means that the transition from $TMI$ to $Invalid$ of the requested cache line does not happen immediately after the processor receives the GetM message. This is why the cache line still stays in the $TMI$ state upon a GetM message. The implementation details are beyond the scope of this thesis. What we need to focus on is that the line moves from the $TMI$ state to $Invalid$ when the transaction aborts ($TMI \xrightarrow{Abort} I$).

3.4 TM Safety and Liveness Properties

In this section we introduce safety and liveness properties for TMs. Generally speaking, safety properties ensure that nothing bad would happen to the TM, and liveness properties ensure that the TM will finally make progress during the execution. We will first introduce two safety properties, namely strict serializability and abort consistency, and then two liveness properties, namely obstruction freedom [26] and livelock freedom [10], in the same manner as in [22]. We will mainly focus on the safety properties in this thesis.

3.4.1 Safety Properties

Before we look at the formal definition of the safety properties, we need to introduce how a sequence of statements is generated in a TM system. A statement indicates the instruction, the relevant variable and the thread which has issued the instruction. An example can be $((\text{store}, a), t_1)$. As we can see in Fig 3.2 and Fig 3.3, the execution of a TM program is conducted on $n$ threads, where each thread has one transaction running on it at a time. The scheduler decides which thread to execute the next instruction, and the TM algorithm decides whether 1) the instruction can be executed; or 2) the transaction issuing the instruction has to abort. Observing from outside the system, as the program runs, it generates a sequence. A sequence consists of the accepted statements (namely the accepted instructions, the relevant variables and threads), together with the abort statements, in the order that they are executed by the TM.

As shown in Fig 3.2 and Fig 3.3, at first thread $t_1$ is chosen by the scheduler and $t_1$ issues a store instruction on variable $a$, which is accepted by the TM algorithm. The statement $((\text{store}, a), t_1)$ now appears at the end of the generated sequence. The scheduler next chooses thread $t_2$, which issues a store instruction on variable $b$. The instruction is also accepted by the algorithm. The statement $((\text{load}, b), t_2)$ is appended to the end of the sequence. The execution continues and the statements of $((\text{load}, c), t_n), ((\text{load} a), t_2)$ and $((\text{load} b), t_1)$ are appended to the sequence one at a time. Let us call the transaction running on thread $t_1$ as transaction $x$, and the one on $t_2$ as transaction $y$. At this time,
A TM system with \( n \) threads running concurrently. Each thread executes one transaction at a time. The scheduler selects the next thread to execute. The selected thread issues an instruction, after which the TM algorithm decides whether to accept or reject it.

The scheduler chooses thread \( t_1 \), which issues an instruction (store, \( a \)). The TM algorithm accepts the instruction.

The scheduler now selects thread \( t_2 \), which issues an instruction (store, \( b \)). The TM algorithm also accepts the instruction.

*Figure 3.2. Generated sequence*
The execution continues. Threads $t_n$, $t_2$ and $t_1$ have issued the instructions (load, $c$), (load, $a$) and (load, $b$) respectively. Now thread $t_1$ tries to commit the transaction running on it. The transaction on $t_2$ needs to abort before $t_1$ can commit its transaction, so $t_1$ needs to wait for the response from the TM algorithm.

Thread $t_2$ aborts the transaction running on it.

Now thread $t_1$ can commit the transaction running on it successfully.
transaction \( x \) tries to commit. In this particular scenario, we assume here that all the conflicting transactions have to abort before a transaction can commit successfully. If transaction \( x \) needs to commit, then transaction \( y \) has to abort, since it both 1) writes to the variable that transaction \( x \) reads (the variable \( b \)), and 2) reads the variable that transaction \( x \) writes (the variable \( a \)). After \( y \) aborts, \( x \) commits successfully. The statements (abort, \( t_2 \)) and (commit, \( t_1 \)) are appended to the end of the sequence, and the execution continues.

Below we introduce the notions of *strict serializability* and *abort consistency* with figures. Note that in some of the figures, for simplicity, we use \( st \) to denote store, \( ld \) to denote load, \( com \) to denote commit and \( abt \) to denote abort.

A *word* is a finite sequence of statements. Given a word \( w \) and a thread \( t \), we say that a subsequence of \( w \) is the projection of \( w \) on \( t \) if it consists of all the statements from \( t \).

If a transaction \( x \) commits successfully, we say that \( x \) is committing. For a word \( w \), we use \( \text{com}(w) \) to denote the subsequence of \( w \) which consists of all the statements from all the committing transactions in \( w \). (See the example in Fig 3.4.)

For a word \( w \) and two transactions \( x \) and \( y \) in \( w \), we say that \( x \) precedes \( y \) in \( w \) if the last instruction in \( x \) appears before the first instruction in \( y \). We also say that a word \( w \) is *sequential* if for every two transactions \( x \) and \( y \) in \( w \), either \( x \) precedes \( y \) or \( y \) precedes \( x \).
Given a thread $t$, a transaction $x$ on $t$ and a variable $v$, we say that a statement $s = ((\text{load}, v), t)$ is a **global read** of $v$ in transaction $x$, if there is no statement which writes to $v$ and appears before $s$ in $x$. (See the example in Fig 3.5.)

**Figure 3.6.** Conflict between global read and commit

Consider a word $w$ together with two statements $s_1$ and $s_2$, which are from transactions $x$ and $y$ respectively. We say that $s_1$ and $s_2$ conflict in $w$ if either

- in transaction $x$, $s_1$ is a global read of some variable, transaction $y$ writes to the same variable and $s_2$ is a commit (Fig. 3.6); or

The conflict here has the same meaning as the conflict mentioned in Sec 3.3, although they are not defined in the exactly same way.

---

1The conflict here has the same meaning as the conflict mentioned in Sec 3.3, although they are not defined in the exactly same way.
transactions \( x \) and \( y \) both write to the same variable, and statements \( s_1 \) and \( s_2 \) are both commits (Fig. 3.7).

For two words \( w \) and \( w' \), we say that \( w \) is strictly equivalent to \( w' \) if the following requirements are satisfied:

- For every thread \( t \), the projection of \( w \) on \( t \) is the same as the projection of \( w' \) on \( t \); (Fig. 3.8)
- For every two statements \( s_i \) and \( s_j \) in \( w \), if \( s_i \) and \( s_j \) conflict, and \( s_i \) appears before \( s_j \) in \( w \), then \( s_i \) also appears before \( s_j \) in \( w' \); (Fig. 3.9)
- For every two transactions \( x \) and \( y \) in \( w \), if \( x \) precedes \( y \) in \( w \), then \( y \) does not precede \( x \) in \( w' \). (Fig. 3.10)

\[
\begin{align*}
\text{w:} & \quad ((ld, a), t_1) ((st, a), t_2) ((st, b), t_2) ((st, b), t_3) (abt, t_1) (com, t_1) (ld, c), t_3) (ld, a), t_1) (abt, t_1) \\
\text{w':} & \quad ((ld, a), t_1) ((st, a), t_2) ((st, b), t_3) ((st, b), t_3) ((ld, c), t_3) (abt, t_1) (com, t_2) (com, t_2) (com, t_2)
\end{align*}
\]

**Figure 3.8.** Projections of \( w \) and \( w' \) on all the threads

\[
\begin{align*}
\text{w:} & \quad ((ld, a), t_1) ((st, a), t_2) ((st, b), t_2) ((st, b), t_3) (abt, t_1) (com, t_1) (ld, c), t_3) (ld, a), t_1) (abt, t_1) \\
\text{w':} & \quad ((ld, a), t_1) ((st, a), t_2) ((st, b), t_3) ((st, b), t_3) ((ld, c), t_3) (abt, t_1) (com, t_2) (com, t_2) (com, t_2)
\end{align*}
\]

**Figure 3.9.** Conflict statements in \( w \) and \( w' \)

A word \( w \) is strictly serializable if there is a sequential word \( w' \) which is strictly equivalent to \( com(w) \). A word \( w \) is abort consistent if there is a sequential word \( w' \) which is strictly equivalent to \( w \). (Fig. 3.10)

Intuitively, strict serializability preserves

- the order of the conflicting statements in the committing transactions;
- and the order of the non-overlapping transactions.

Furthermore, abort consistency ensures that even aborting transactions also read consistent values, since inconsistent

We say that a TM ensures strict serializability (abort consistency) if, for every word \( w \) generated by all the programs with an arbitrary number of threads and variables running on the TM, \( w \) is strictly serializable (abort consistent).
Figure 3.10. Transactions preceding orders in \( w \) and \( w' \)

Note that here conflict is defined under deferred version management. According to [22], the methodology can also be used in the context of direct version management with the definition of conflicts changed.

### 3.4.2 Liveness Properties

Intuitively, *obstruction freedom* requires that a transaction is guaranteed to make progress if all the other transactions are suspended. In other words, if a thread executes instructions in isolation and it executes an infinite number of aborts, then it executes an infinite number of commits. Formally, an infinite word \( w \) is obstruction-free if

\[
\forall t \in T (\Box (\text{abort}, t) \rightarrow \Box ((\text{commit}, t) \lor \bigvee_{c \in C, u \in T \setminus \{t\}} (c, u)))
\]

where \( T \) is the set of threads and \( C \) is the set of instructions (read, write, and commit).

*Livelock freedom*, on the other hand, requires that there is always progress in any infinite word. More precisely, it requires that in every infinite word, either 1) there are an infinite number of commits; or 2) there is a thread which executes a finite number of aborts and an infinite number of instructions. Formally, a word is livelock-free if

\[
\Box (\forall t \in T (\text{commit}, t)) \lor \bigvee_{t \in T} (\Box (\forall c \in C (c, t)) \land \Box \neg (\text{abort}, t)))
\]

where \( T \) is the set of threads and \( C \) is the set of instructions (read, write, and commit).
We say that a TM ensures obstruction-freedom (resp. livelock-freedom), if for every word \( w \) generated by all the programs with an arbitrary number of threads and variables running on the TM, \( w \) is obstruction-free (resp. livelock-free).
4. Memory Consistency Models

In this chapter, we will introduce memory consistency models. We will start with the definition of memory consistency models (or just memory models or consistency models for simplicity), and explain the reason why we need such models, with a simple example. Then we will introduce Sequential Consistency (SC), a classical consistency model. Following this, we will consider some relaxed memory models. A relaxed memory model allows the reordering of instructions to some extent, in order to improve performance. We will introduce some well-known relaxed memory models such as TSO, PSO and Power. Following that we will introduce the SISD model which allows the reordering of instructions because of the adopted cache coherence protocol. We will also discuss the potential problems of relaxed models. Finally, we will introduce fences which limit the reordering of instructions and hence can be used to prevent the problems caused by relaxed models.

4.1 What and Why

The execution of a multi-core program should behave in a way such that when values are read from memory by some instruction, they are the same values as the programmer expects to be read. A memory consistency model is a set of rules which define the order in which all the load and the store instructions appear to execute to the programmer. More precisely, a memory consistency model specifies when a written value from some processor is returned by the read of some other processor. As the program is run on multiple processors, there can be more than one execution order that the instructions may follow. In other words, the program can have different outputs for the same input, which means that multiple behaviors of a multi-core program are usually allowed.

To illustrate why we need memory consistency models, we use a simple example in Fig 4.1. The figure shows the classical Dekker’s algorithm. Intuitively, the possible outcomes of the program can be:

1. ($r_1 = 0, r_2 = 1$), with the execution of $L_1, L_2, L_3, \text{ and } L_4$;
2. ($r_1 = 1, r_2 = 0$), with the execution of $L_3, L_4, L_1, \text{ and } L_2$;
3. ($r_1 = 1, r_2 = 1$), with the execution of $L_1, L_3, L_2, \text{ and } L_4$. There are several executions which lead to this outcome. Here we only illustrate one of them for simplicity.

However, on most modern architectures, the outcome of ($r_1 = 0, r_2 = 0$) is also possible due to some hardware optimizations that cause the reordering
of the instructions under the semantics. This additional behavior may cause bugs in case the programmer is not aware of it. Therefore, memory consistency models are necessary since we need to define 1) what behaviors of a system are allowed; and 2) what optimizations a system can use.

![Figure 4.1. Dekker’s algorithm.](image)

### 4.2 Sequential Consistency

One of the most classical and intuitive memory consistency model is sequential consistency (SC), which was first formally defined in [31]. According to the definition in [31], a multi-core system is sequentially consistent if

1. "the result of any execution is the same as if the operations of all processors were executed in some sequential order"; and
2. "the operations of each individual processor appear in this sequence in the order specified by its program"

![Figure 4.2. SC with switch](image)

As mentioned in the definition above, the order of operations, as specified by the program is program order. Here we use the terms instruction and operation interchangeably. The former item in the definition means that SC requires that the execution of the program should follow some sequential order of the
instructions from all the processors; and the latter means that the execution should also maintain the program order of the instructions from every single processor.

Intuitively, from a programmer’s perspective, one can imagine that there is a switch between the processors and the main memory as shown in Fig 4.2. Each processor issues instructions following the program. The switch randomly chooses a processor, one at a time, and lets that processor execute the load or the store instruction atomically. The switch repeats the procedure as long as there are operations to be issued by the processors.

![Diagram](image)

**Figure 4.3.** Different processors observing the same ordering of memory accesses

Note that in the SC model, operations from different processors can interleave, but all the processors must see the instructions accessing the main memory in the same order. For example, in Fig 4.3, $P_0$ and $P_1$ write 1 and 2 to the variable $x$ respectively. The two writes can be ordered in either way, but when $P_2$ and $P_3$ read the value of $x$ twice, they should observe the changes to $x$ in the same order. I.e., either 1) both $P_2$ and $P_3$ first read the value 1 of $x$ and then the value 2; or 2) $P_2$ and $P_3$ first read the value 2 of $x$ and then the value 1.

A multi-core system which satisfies SC respects the program order. However, such systems restrict the order of the instructions in each processor, and thus restrict the implementation of optimization in hardware. In the next sec-
tion, we will introduce some memory models which allow the reordering of instructions to improve performance.

4.3 Relaxed Memory Models

A memory model which relaxes the ordering requirements is a relaxed memory model. By allowing reordering of the program instructions, we can improve the performance of the system. The reordering can be caused by a wide variety of hardware optimizations. One of them is store/load buffering in hardware. A typical example is Total Store Order (TSO) [46]. In the TSO memory model, store buffers are added to hide store latency. When a store instruction is in a buffer, a later load instruction of the same variable from the same processor can appear to be executed earlier than the store. Compilers can also reorder instructions such as C11 [11]. Sometimes, a cache coherence protocol does not ensure strict coherence and allows the reordering of instructions, like SISD [38]. In this section, we will introduce the relaxed memory models of TSO and PSO which support store buffering; the Power model [41] which allows different processors to independently observe different orderings of the same memory accesses; and SISD which follows the semantics of the SISD cache coherence protocols. Some of the content in this section is from the published work [3], which is appended at the end of this thesis.

4.3.1 Examples of Relaxed Memory Models

Total Store Order (TSO)

As illustrated in Fig 4.4, each processor has a store buffer in the TSO model. When a processor executes a write instruction, the instruction is appended to the end of the buffer of the processor. At any point of the execution, the instruction at the head of the buffer may nondeterministically be removed and applied to update the memory. When a processor reads the value of a variable, it fetches the value from the most recent write instruction on the variable in its buffer. If such a write instruction is missing, then the value is fetched from the memory.

![Figure 4.4. TSO](image)
The TSO model is weaker than SC. We can look back at Dekker’s algorithm illustrated in Fig 4.1. Under the SC semantics, the program does not have any executions satisfying the assertion, since neither the instructions of $P_0$, nor the ones of $P_1$ are reordered. When $r_1 := y$ is executed by $P_0$, $x := 1$ must have been executed and the value 1 of $x$ is updated to the memory. If $r_1 = 0$ holds, then $P_0$ sees the value of $y$ as 0 in the memory, which means that $y := 1$ is not executed by $P_1$ yet. When $P_1$ executes $y := 1$ and $r_2 := x$, the value of $x$ in the memory is already 1 and thus 1 is assigned to $r_2$.

On the other hand, a program execution under the TSO semantics can satisfy the assertion as follows. First, $P_0$ and $P_1$ execute the instructions $x := 1$ and $y := 1$ respectively, and put them in the store buffers. Then, $P_0$ executes the instruction $r_1 := y$. Since the new value of $y$ changed by $P_1$ has not been updated in the memory, $P_0$ reads the value of $y$ as 0 and assigns 0 to $r_1$. For the same reason, $P_1$ sees the value of $x$ as 0 and assigns 0 to $r_2$.

**Partial Store Order (PSO)**

While in the semantics of TSO, each processor is equipped with one store buffer, in the PSO memory model, for each processor, there is a set of store buffers where each of them is used for one distinct variable (Fig 4.5).

![Figure 4.5. PSO](image)

In fact, the PSO model is even weaker than TSO, which means that it is also weaker than SC. Consider the program $MP$ in Fig 4.6 for example. The program executions under the semantics of TSO cannot satisfy the assertion. The reason is that the two write instructions performed by $P_0$ will reach the memory in the same order as they are performed, i.e, $x := 1$ and then $y := 1$. Furthermore, the two read instructions performed by $P_1$ are not reordered according to the TSO semantics. Therefore, after $r_1 := y$ is performed, if $r_1 = 1$ holds, then $P_1$ will also see that the value of $x$ is equal to 1 when it performs the assignment $r_2 := x$. 

57
Initially: $x=0$, $y=0$

<table>
<thead>
<tr>
<th>process P0</th>
<th>process P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>registers $r1$ $r2$</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>L1: $x := 1$;</td>
<td>L3: $r1 := y$;</td>
</tr>
<tr>
<td>L2: $y := 1$;</td>
<td>L4: $r2 := x$;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Assertion: $r1=1$, $r2=0$

Figure 4.6. Program MP.

However, there are program executions that satisfy the assertion under the semantics of PSO. More precisely, $P_0$ executes the instructions $x := 1$ and $y := 1$ and puts them in the store buffers for $x$ and $y$ respectively. Later on $y := 1$ is updated to the memory. Now $P_1$ executes $r1 := y$, sees the value 1 of $y$ in the memory and assigns 1 to $r_1$. Since $x := 1$ is not updated to the memory yet, $P_1$ sees value 0 of $x$ and assigns it to $r_2$.

The Power Model

Unlike TSO or PSO, the IBM Power model [41] does not have store buffers. Actually, most instructions of a program under the Power semantics do not have to follow the program order. Instead, there are some other constraints which define the order between the instructions. The full description of the Power semantics is very complicated and beyond the scope of this thesis. Here we will introduce the Power model on a high level for the sake of comparing with other memory consistency models.

In the Power model, after a processor $P$ writes to a variable, the write instruction may be *committed*. Right after the commitment, $P$ will *propagate* the instruction to itself. Following this, the processor is allowed to nondeterministically propagate the instruction to some set of other processors. We use the term *local write* of $x$ in processor $P$ to mean a write to $x$ performed by $P$ before it propagates the write to itself; and we use the term *propagated write* of $x$ to $P'$ to mean a write to $x$ performed by processor $P$ propagated to $P'$. Note that $P'$ can also be $P$.

When a processor $P$ needs to read the value of variable $x$, it gets the value from the most recent local write. If there is no local write of $x$, then it gets the value from the most recent propagated write. If there is neither a local write nor a propagated write of $x$ to $P$, then the read gets the initial value of $x$.

If a processor writes to the same variable more than once, then the write instructions should follow the program order. For instance, if processor $P$ first writes value 1 to variable $x$ and then writes 2 to it, in program order, then $P$ should commit the instruction $x := 1$ first and then $x := 2$. Furthermore, the processors should observe the same order as the program order in which the
writes to some variable is propagated. A typical example can be, instructions $x := 1$ and $x := 2$ are propagated to processor $P_1$ in the order of first $x := 1$ and then $x := 2$. If the two instructions are also propagated to another processor $P_2$, then $P_2$ should first see the propagation of $x := 1$ and then $x := 2$.

Furthermore, if there is a dependency between two instructions, the instructions cannot be reordered either. An example of dependency between two instructions $l_1$ and $l_2$ in the same processor is the following: $l_1$ is a read instruction $s r := x$, and $l_2$ is a write instruction with $s r$ involved: $y := 2 * s r$. Since $l_2$ uses the register whose value is defined by $l_1$, $l_2$ must be committed after $l_1$.

| Initially: x=0, y=0 |
|---|---|---|---|
| process P0 registers begin L1: $x := 1$; end |
| process P1 registers $r_1$ $r_2$ begin L2: $r_1 := x$; L3: $r_2 := y$; end |
| process P2 registers begin L4: $y := 1$; end |
| process P3 registers $r_3$ $r_4$ begin L5: $r_3 := y$; L6: $r_4 := x$; end |

Assertion: $r_1=1$, $r_2=0$, $r_3=1$, $r_4=0

Figure 4.7. Program IRIW.

We describe why the Power model is weaker than PSO, and consequently weaker than TSO and SC. We consider the program IRIW in Fig 4.7. The program executions under the PSO semantics cannot satisfy the assertion. In PSO, if a processor can read the new value of a variable updated by another processor, it means that the value has been updated to the memory. Then all the processors will be able to see the new value. In this way, when $s r_1 := x$ is performed by $P_1$ and $s r_3 := y$ is performed by $P_3$, if both $s r_1 = 1$ and $s r_3 = 1$ hold, then instructions $x := 1$ and $y := 1$ must have been updated to the memory. As there is no store instruction in $P_1$ or $P_3$, the instructions in $P_1$ and $P_3$ are not reordered. When $P_1$ executes $s r_2 := y$ or $P_3$ executes $s r_4 := x$, at least one of them will see the value 1 of the associated variable.

On the other hand, there are program executions under the Power semantics that satisfy the assertion. Consider the following scenario: first $P_0$ executes $x := 1$, propagates the write to itself and then to $P_1$. Then $P_2$ executes $y := 1$ and propagates it to itself and $P_3$. Since $P_1$ does not get the propagation of the new value of $y$, it assigns 1 to $s r_1$ and 0 to $s r_2$. For the same reason, we can get $s r_3 = 1$ and $s r_4 = 0$.

4.3.2 The SISD Memory Model
The SISD model is modeling the behaviors of a cache coherence protocol with Self-Invalidation (SI) [32] and Self-Downgrade (SD) [38]. As opposed to
in the MESI protocol, in the SISD protocol, a processor does not invalidate a cache line from its L1 cache when another processor writes to the line. Instead, the old line can be kept until a synchronization operation is carried out by the processor. During the synchronization the line is invalidated, which is called self-invalidate. Similarly, unlike the MESI protocol which makes a processor update a written line to the memory when another processor needs to read the changed line, in the SISD protocol, a processor does not need to update the written line to the memory until a synchronization comes. Then the line is written back to the memory, which is called self-downgrade. Since the invalidation and the downgrade only forced during synchronizations, the SISD protocol does not need a directory to track the sharers or the last writer of a cache line. We call the memory consistency model based on the SISD protocol the SISD memory model for simplicity.

We formalize the SISD memory model semantics based on the protocol in [4]. When a processor needs to read a cache line, it gets the line from the L1 cache. If there is no such a line, then it fetches the cache line from the last level cache (LLC), which is logically shared among all the cores. When a processor writes to a line, it might not update the changed value to the LLC immediately. Instead, it may update the new value to the LLC nondeterministically some time after the write happens. A processor can also fetch a line from the LLC nondeterministically. A clean line, which means that the value of the cache line is the same as the value of the corresponding line in the LLC, can be evicted from the L1 cache nondeterministically as well. Since a processor can fetch a cache line at any time, and the written line is not updated to the LLC immediately after it is changed, a program under the semantics of the SISD model cannot guarantee that the processors read the latest written values. In other words, the instructions may be reordered and the SISD model is a weak memory model.

In the next few paragraphs, we will proceed to compare SISD with other memory models.

Comparing with SC
The SISD model is weaker than SC. We first show that SC is not weaker than SISD by showing that any execution of the program under SC may be simulated under SISD following this procedure: 1) right after each write instruction, the assigned value of the variable is updated to the LLC and the variable is invalidated from the local cache immediately; 2) right before each read instruction, the value of the variable is fetched from the LLC and after reading the value, the variable is immediately invalidated from the local cache. Now we consider the Dekker’s algorithm illustrated in Fig 4.1. We know that the assertion is not satisfied by any execution under SC. Furthermore, there are executions of the program under the semantics of SISD that can satisfy the assertion, which means that SISD is weaker than SC. For example, processor $P_0$ and $P_1$ fetch the value 0 for variables $y$ and $x$ respectively, and then assign
1 to $x$ and $y$ respectively. When $P_0$ reads the value of $y$, it gets 0 from its L1 cache and assigns it to $r_1$. For the same reason, we can get $r_2 = 0$.

**Comparing with TSO**

The SISD model and TSO are not comparable. First, we show that TSO is not weaker than SISD. Consider the program $MP$ in Fig 4.6. Under the TSO semantics, the program $MP$ does not have any executions that satisfy the assertion. The reason is that the two write instructions performed by $P_0$ will reach the memory in the same order as they occur in the program, i.e., $x := 1$ and then $y := 1$. Furthermore, the two read instructions performed by $P_1$ are not re-ordered according to the TSO semantics. Therefore, if the read $r_1 := y$ sees the value 1, then $x := 1$ is already updated in the memory. The read $r_2 := x$ performed by $P_1$ will see the value 1 as well. In contrast, the program MP has the following scenario under SISD that satisfies the assertion. First, the processor $P_0$ assigns 1 to both $x$ and $y$, but only updates the value of $y$ to the LLC. Next, $P_1$ fetches the values of $x$ and $y$ (0 and 1 respectively) from the LLC, and assigns them to the registers using the instructions $r_1 := y$ and $r_2 := x$, which means the assertion will be satisfied.

The program $ReadSeq$ in Fig 4.8 shows that SISD is not weaker than TSO. Under TSO, the program exhibits an execution that satisfies the assertion as follows. First, $P_0$ executes the instructions $x := 1, x := 2, x := 3,$ and $x := 4$, one by one, and puts the instructions in its buffer. Then, $P_1$ executes the instructions $y := 1, y := 2, y := 3,$ and $y := 4$, one by one, again putting the instructions in its buffer. Now, the memory is updated with $x := 1$ after which $P_1$ executes $r_5 := x$, thus assigning 1 to $r_5$. Following this, the memory is

---

**Figure 4.8. Program ReadSeq.**

![Figure 4.8. Program ReadSeq.](image)
updated with $x := 2$ after which $P_1$ executes $\$r_6 := x$, thus assigning 2 to $\$r_6$. Finally 3 and 4 are assigned to $\$r_7$ and $\$r_8$ respectively in similar manners. A similar sequence of operations is performed assigning 1, 2, 3 and 4 to $\$r_1$, $\$r_2$, $\$r_3$, and $\$r_4$ respectively.

However, under the SISD semantics, the ReadSeq program does not have any executions that satisfy the assertion. More specifically, since the processors do not have any store buffers, at most three different values of a variable can be kept in this example. I.e., one in the local cache of the processor which executes the write operation of the variable, one in the local cache of the processor which executes the read operation of the variable, and one in the LLC. When there are four or more values assigned to a variable in a similar manner as in ReadSeq, the assertion that each processor reads all the values of the variable in the same order as it is written by the other processor cannot be satisfied.

Comparison with PSO

The S1SD model and PSO are not comparable. To illustrate the difference between SISD and PSO, we consider the program $\text{WRC}$ in Fig. 4.9. Under the PSO semantics, the program $\text{WRC}$ does not have any executions that satisfy the assertion. More precisely, when $\$r_1 := x$ is executed by $P_1$, if $\$r_1 = 1$ holds, then the write instruction $x := 1$ by $P_0$ must have reached the memory before the write operation $y := 1$ has been performed by $P_1$. Furthermore, if $\$r_2 = 1$ holds then the write operation $y := 1$ must have reached the memory before the instruction $\$r_3 := x$ has been performed by $P_2$. Since read instructions are not re-ordered in PSO and since $x := 1$ reaches the memory before $y := 1$, it follows that the value of $x$ is equal to 1 in the memory when $\$r_3 := x$ is performed by $P_2$, and hence $\$r_3 = 1$ at the end of the execution.

Initially: $x=0$, $y=0$

<table>
<thead>
<tr>
<th>process $P_0$</th>
<th>process $P_1$</th>
<th>process $P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers $$r_1$</td>
<td>registers $$r_1$</td>
<td>registers $$r_2$, $$r_3$</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>$L_1$: $x := 1$;</td>
<td>$L_2$: $$r_1 := x$;</td>
<td>$L_4$: $$r_2 := y$;</td>
</tr>
<tr>
<td>end</td>
<td>$L_3$: $y := 1$;</td>
<td>$L_5$: $$r_3 := x$;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Assertion: $\$r_1 = 1$, $\$r_2 = 1$, $\$r_3 = 0$

Figure 4.9. Program $\text{WRC}$.
variable $y$ and updates the value to the LLC. Processor $P_2$ fetches the value of $y$ from the LLC and then executes the instruction $r_2 := y$ which means that $r_2 = 1$. Finally, $P_2$ executes the instruction $r_3 := x$ without fetching the new value from the LLC and we get $r_3 = 0$.

As PSO is weaker than TSO, we can use the example in Fig 4.8 again to show that the ReadSeq program also has an execution under PSO satisfying the assertion, which means that SISD is not weaker than PSO.

Comparison with Power
The SISD model and Power are not comparable. There are executions which are allowed by SISD but not by Power. Intuitively this is because in the Power model address dependencies enforce order between memory accesses while in the SISD model they do not. We show this with the program PwrEg in Fig 4.10. Note that in the figure, the instruction sync labelled with L2 is a fence instruction. It means that the instructions before and after sync cannot be reordered across it. The instruction fence labelled with L2 in Fig 4.11 is similar to sync. We will introduce more details of such instructions in the next section. Besides, the instruction labelled with L5 in Fig 4.10 is an expression where $\&x$ indicates the value of $x$ fetched from its address.

The program does not have any executions that satisfy the assertion under Power, since cycles of the form write-(sync)$\rightarrow$write-(read from)$\rightarrow$read-(address dependency)$\rightarrow$read-(from read)$\rightarrow$ are not allowed by Power. The sync instruction maintains the order between the two write instructions and the address dependency maintains the order between the two read instructions, which make the cycle impossible.

<table>
<thead>
<tr>
<th>Initially: $x=0$, $y=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>process P0 registers</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>L1: $x := 1;$</td>
</tr>
<tr>
<td>L2: sync;</td>
</tr>
<tr>
<td>L3: $y := 1;$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>process P1 registers $r1$ $r2$</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>L4: $r1 := y;$</td>
</tr>
<tr>
<td>L5: $r2 := [&amp;x + 0*&amp;r1];$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Assertion: $r1=1$, $r2=0$</td>
</tr>
</tbody>
</table>

*Figure 4.10. PwrEg*

However, under SISD the program SisdEg in Fig 4.11 has an execution that satisfies the assertion. Processor $P_1$ fetches the initial value 0 of variable $x$. Process $P_0$ assigns 1 to $x$, updates the value to the LLC, assigns 1 to $y$, and updates the value to the LLC again. Process $P_1$ fetches the value 1 of $y$ from the LLC and then executes the instruction $r_1 := y$. Therefore, the value of $r_1$ is equal to 1. Finally, $P_1$ executes the instruction $r_2 := x$ and we get $r_2 = 0$. 
4.4 Fences

As we have seen before, the reordering of the program instructions in a relaxed memory model may cause expected behaviors. A typical example is Dekker’s algorithm (Fig 4.1) under the semantics of TSO, where the processors read the value 0 of $x$ and $y$. Therefore, sometimes we need to enforce the order between some instructions which are allowed to be relaxed by the memory model. In order to enforce such orderings, we use fence instructions.

Fence instructions from different memory models may have different behaviors, but they all have the common effect which of restricting the order in which the instructions access the memory. For example, in TSO, a fence instruction forces that the execution of a processor blocks until the store buffer in the processor is emptied. As a result, a fence instruction ensures that, in this processor, all instructions preceding the fence in program order access the memory before any instructions succeeding the fence in program order. In fact, some memory models also provide different types of fence instructions. For example, there are two fence instructions, namely full fence and sfence in PSO. A full fence instruction is the same as a fence instruction under TSO, which means that all the instructions before the fence must take effect before all instruction which are after it. On the other hand, an sfence instruction under PSO is modeled as follows according to [5]: when an sfence is executed, it is appended at the end of each store buffer in the processor. The stores queued after the sfence in the buffer cannot be removed before sfence. When sfence is at the head of all store buffers of the processor, it can be removed simultaneously from all of them. An sfence requires that the store instructions before sfence in program order of the same processor must update the variables in the memory earlier than any store instructions after the sfence in program order.
We first take a look at a simple example that shows how to use fence instructions, and then introduce the fence instructions in the SISD memory model.

Initially: \( x=0, y=0 \)

```
process P0
  registers $r1
  begin
    L1: x := 1;
    L2: fence;
    L3: $r1 := y;
  end

process P1
  registers $r2
  begin
    L4: y := 1;
    L5: fence;
    L6: $r2 := x;
  end
```

Assertion: \( $r1=0, $r2=0 \)

Figure 4.12. Dekker’s algorithm with fences inserted.

Consider Dekker’s algorithm in Fig 4.1 under TSO, if we add a fence instruction between L1 and L2 in \( P_0 \), and between L3 and L4 in \( P_1 \) respectively as we can see in Fig 4.12, any execution satisfying the assertion is prevented. With a fence instruction inserted after L1, the store buffer of \( P_0 \) is emptied and the value 1 assigned to \( x \) is updated to the memory before \( P_0 \) reads the value of \( y \) (L3). For the same reason, the value of \( y \) in the memory is updated before \( P_1 \) reads the value of \( x \). In this case, it is impossible that the processors will read the value 0 for both \( x \) and \( y \).

In SISD, there are three kinds of fence instructions, namely

- **store-store fence (ssfence)**, which requires that all the dirty values in the L1 cache of the processor are flushed before the ssfence is executed. This ensures that all the stores before the ssfence instruction (in program order) take effect before any stores after ssfence in the processor, which means that the store instructions before the ssfence cannot be reordered after it, and vice versa.

- **load-load fence (llfence)**, which requires that there is no clean value in the L1 cache of the processor when llfence is executed. This enforces that all the load instructions before llfence in program order are executed before any loads after the llfence instruction of the same processor.

- **full fence**, which is the strongest fence instruction in SISD. It requires that the L1 cache is emptied when the fence instruction is executed. The full fence instruction ensures that all the instructions before fence in program order should take effect before any instructions after it in the same processor.

The reason that there are more than one kind of fences in SISD or other memory models is that, the fence instructions have different costs and can be used under different contexts accordingly. To ensure the program correctness, we do not always need to use full fences. Sometimes fences with lower costs can also prevent undesired reordering. By using lighter fences, we can save
some cost of executing fences and thus avoid performance regression to some extent.

| Initially: x=0, y=0
| process P0 registers begin L1: x := 1; end |
| process P1 registers $r1 begin L2: $r1 := x; L3: y := 1; end |
| process P2 registers $r2 $r3 begin L4: $r2 := y; L5: llfence; L6: $r3 := x; end |
| Assertion: $r1=1, $r2=1, $r3=0 |

Figure 4.13. Program WRC with fence inserted.

We look at the WRC program illustrated in Fig 4.9 again. If a full fence is inserted between instructions of L4 and L5 in processor \( P_2 \) under SISD, then the local cache of \( P_2 \) is emptied before the fence. When \( P_2 \) tries to assign the value of \( x \) to \( $r_3 \), it has to fetch the updated value of \( x \) from LLC. The assertion cannot be satisfied in this way. However, we only need to insert an llfence instead of a full fence after L4 (Fig 4.13), since what we need is to force \( P_2 \) to read the value of \( x \) from LLC. With an llfence inserted, the program still cannot satisfy the assertion and yet saves some cost of executing a full fence.

When we insert fences into a program, we need to make sure that 1) we insert enough fences so that all the undesired behaviors of the program under the current semantics can be avoided, and 2) we do not insert too many (or too heavy) fences which may result in a worse performance and make adopting relaxed memory models meaningless. As manual fence insertion is time-consuming and error-prone due to the complicated behaviors of multicore programs, we would like to have an automatic fence insertion algorithm.

Figure 4.14. Fence insertion algorithm
Fig 4.14 illustrates the main procedure of a fence insertion algorithm. We can see that the algorithm takes a program as input, and checks if some bad states can be reached (or in other words, if there is an execution of the program that satisfies the undesired assertions under the semantics of the memory model). If none of the bad states is reachable, then the program is correct. Otherwise we will get a counter example which reaches a bad state. The algorithm then analyzes the counter example, identifies the reordering(s) in the example, inserts fence(s) to prevent the example from occurring and checks if the bad states are reachable again. If they are still reachable, then the algorithm repeats the procedure until either 1) none of the bad states is reachable, which means that the program is now correct with the fences inserted. The algorithm returns the set of fences which should be inserted to prevent the program from reaching the bad states; or 2) a counter example with no reordering has been found, which means that the program is incorrect even under SC, and it is impossible to correct the program by just inserting fences. The algorithm can be extended to provide all the optimal sets of fences. We explain the details in [4]. We use reachability analysis to check whether a bad state can be reached, and use trace analysis to find out the reorderings in a program.
5. Contributions

In this chapter, we introduce the contributions of each published paper in detail. We consider the analysis of the models based on transactional memories, cache coherence protocols in the presence of transactional memories, and weak memory models induced by some cache coherence protocols.

5.1 Paper I: Verifying Safety and Liveness for the FlexTM Hybrid Transactional Memory [7]

Our main contributions of this paper are as follows:

1. We propose a parameterized system model of the hybrid transactional memory FlexTM. We show that the obtained model satisfies four properties which allow us to apply a small model theorem. The theorem ensures that it is enough to verify strict serializability and abort consistency of models with two threads and two variables.

2. We build the model in the form of automata, considering both the lazy and the eager conflict resolutions. The automata contain tens of thousands of states. We also build automata of two reference models which characterize the exact descriptions for strict serializability and abort consistency.

3. We prove strict serializability and abort consistency of FlexTM by means of language inclusion between the automata representing FlexTM and the reference automata. We use an efficient tool which checks language inclusion for automata. To our knowledge, this is the first time of automatically and formally verifying a hybrid transactional memory.

4. We show that the model of FlexTM satisfies two additional properties allowing us to apply another small model theorem, which ensures that it is enough to verify obstruction-freedom of models with two threads and one variable.

5. We analyze the obtained model of FlexTM, establish obstruction-freedom and show that FlexTM does not satisfy livelock freedom.

Contribution 1

We build a model of the hybrid transactional memory FlexTM that is parameterized in multiple dimensions, namely 1) the size of concurrent transactions; 2) the number of threads; and 3) the number of shared variables. As we need
to verify an infinite family of transition systems, we should try to reduce the problem and make the verification simpler. There are several ways to do so, one of them is to apply a small model theorem. A small model theorem ensures that, if a transition system with a fixed number of components of some parameterized system model satisfies some properties, then any transition system with an arbitrary number of components of that model satisfies the properties. We apply the small model theorem proposed in [22]. In order to apply the theorem, a transactional memory \( M \) needs to satisfy four properties. In such a case, if \( M \) ensures strict serializability and abort consistency for all programs with two threads and two variables, then it ensures the same properties for all programs with an arbitrary number of threads and variables.

We show that FlexTM ensures the four properties. Let \( M \) be a transactional memory, \( p \) be a program with \( n \) threads and \( k \) variables and \( w \) be a finite prefix of a word in \( p \) of \( M \).

- **Property 1: symmetry in threads.** The property states that if 1) there is no aborting transaction in \( w \); and 2) there are two threads \( t_1 \) and \( t_2 \) such that for all the committing transactions \( x \) in \( t_1 \) and \( y \) in \( t_2 \), either \( x \) precedes \( y \) or \( y \) precedes \( x \), then the word generated by moving all the transactions in thread \( t_2 \) to \( t_1 \) is a finite prefix of a word in some program of \( M \) with \((n - 1)\) threads and \( k \) variables.
  
  The property holds in FlexTM. Since there is no aborting transaction in \( w \), there is at most one pending transaction at the end of threads \( t_1 \) and \( t_2 \) respectively.
  
  - If there is no pending transaction from \( t_1 \) or \( t_2 \), then moving the transactions in \( t_2 \) to \( t_1 \) would not cause any overlapping. As the committing transactions are not overlapping, we can regard that all the transactions are from one thread, since the transactional memory is not aware which threads execute transactions if they are sequential.
  
  - If there is one pending transaction from \( t_1 \) or \( t_2 \), then we can place it after the last committing transaction while moving the transactions in \( t_2 \) to \( t_1 \). It is reasonable even if the pending transaction commits later on, since the last committing transaction has already taken effect. Same as above, moving the transactions would not cause any overlapping, so we can regard that all the sequential transactions are from one thread.
  
  - If there is a pending transaction in both \( t_1 \) and \( t_2 \), then moving the transactions will cause the two pending transactions to overlap. But since they are not committed, they can be merged and regarded as the instructions of a new program.

- **Property 2: transaction projection.** The property states that if \( w' \) is a subsequence of \( w \), and it is the projection on a subset of the set of the committing and the pending transactions in \( w \), then \( w' \) can be produced by another program running on \( M \).
As we know, in FlexTM, a transaction can influence other transactions only by aborting them. An aborting or pending transaction cannot abort any transactions since it has not committed yet. Therefore, removing all the aborting and some pending transactions would not influence any transactions. Removing a committing transaction may allow some transactions to commit which should have been aborted. But since all the aborting transactions are removed, removing some committing transactions would not influence any other transaction either.

- **Property 3: variable projection.** The property states that if there is no aborting transaction in \( w \), then the subsequence of \( w \) gained by removing all the read and write statements (see 3.4.1) of some of the variables in \( w \), is a word in some program of \( M \). The property holds in FlexTM, since removing the statements of some variables would not cause new conflicts or abort more transactions.

- **Property 4: monotonicity property.** The property states that if
  - \( w = w_1 \cdot s \) where \( w_1 \) is a strictly serializable (abort consistent) word,
  - \( s \) is a statement of the only pending transaction in \( w \), and
  - \( s \) is not an aborting statement,
then there exists a word \( w_2 \) such that
  - \( w_2 \) is strictly equivalent (see 3.4.1) to \( w_1 \),
  - \( \text{com}(w_2) \) is sequential, and
  - \( w_2 \cdot s \) is a finite prefix of a word of some program in \( M \).

The property holds in FlexTM, because if FlexTM produces a word, then the more sequential version of the word can also be produced by the system, since it does not change any conflicts or transactions.

**Contribution 2**

Since FlexTM satisfies the four properties of the small model theorem, we can restrict the numbers of both variables and threads in the parameterized system model of FlexTM to two. The size of transactions still remains arbitrary. This can be handled by building an automaton which captures all possible behaviors of FlexTM regardless of the program.

We use the two reference transactional memories called RSS and RO proposed in [22]. The RSS transactional memory can generate all strictly serializable words, while RO can generate all abort consistent words. We build two automata of RSS and RO that exactly characterize the set of strictly serializable words and the set of abort consistent words respectively.

We build six finite state automata in all.

- Four of them capture the behaviors of FlexTM where
  - two are in the lazy conflict detection mode; and
  - two are in the eager conflict detection mode.

For each mode, one of the automata is modeled with two cache lines where each of them contains a single variable, while the other automaton is modeled with only one cache line which contains two variables.
• One is the reference automaton based on RSS, which captures most general descriptions for strict serializability.
• One is the reference automaton based on RO, which captures most general descriptions for consistency respectively.

Each automaton involves two threads and two variables. A state of the automaton is a configuration of the system, i.e., the states of the two threads. A transition is a statement, such as a read of variable $v_1$ from thread $t_1$. As the numbers of the threads, the variables and the possible operations on a variable are all finite, the automaton is also finite.

Contribution 3
We check language inclusion between the four models and the two reference models with the efficient tool VATA Library. The result shows that all the four models capturing the FlexTM behaviors are included in the two reference models, which means that the transition system of FlexTM with two threads and two variables ensures strict serializability and abort consistency. By applying the small model theorem, we can infer that FlexTM ensures these two safety properties.

Contribution 4
Besides the small model theorem applied for the safety properties, there is another (small model) theorem which can be applied to verify obstruction-freedom [22]. The theorem states that if a transactional memory $M$ ensures obstruction-freedom for all programs with two threads and one variable, then it ensures the property for all programs with an arbitrary number of threads and variables. In order to apply the theorem, $M$ needs to satisfy two additional properties.

We show that FlexTM ensures these two properties.

Let $M$ be a transactional memory, $p$ be a program running on $M$ and $w = w_1 \cdot w_2$ be an infinite word generated by $p$, such that $w$ satisfies the conditions as follows:

• no pending transactions from $w_1$ has a statement in $w_2$
• all statements in $w_2$ belong to the same thread
• there is no commit in $w_2$

The two properties are as follows:

• Property 5: transaction projection
  1. If word $w'_1$ is the projection on the non-aborting transactions in $w_1$, then $w' = w'_1 \cdot w_2$ is a word generated by some other program running on $M$, and
  2. if
     – there is no aborting transaction in $w_1$,
     – the read and the write instructions in $w_2$ only operate on one variable, and
– $w''_j$ is obtained by projecting $w_j$ on some thread $t$ where $t$ has statements in $w_j$
then the word $w' = w''_j \cdot w_2$ is a word generated by $p$.

The property holds in FlexTM. For the first condition, projecting away the aborting transactions would not influence any other transaction, since the only way that one transaction can affect another is by aborting it. As for the second condition, since there is no aborting transaction in $w_j$, projecting away the transactions on other threads would not remove the conflicts that may cause some transactions to be aborted. This means that the word $w'$ can exist.

- **Property 6: variable projection**
  1. if $w'_2$ is obtained by projecting on some variable accessed in $w_2$, then $w_j \cdot w'_2$ is a word generated by some other program running on $M$; and
  2. if
     - there is no aborting transaction in $w_j$, and
     - $w'_j$ is obtained by projecting $w_j$ on the variables which are accessed both in $w_j$ and $w_2$,
then $w'_j \cdot w_2$ is a word generated by some other program running on $M$.

The property holds in FlexTM. For the first condition, as mentioned before, projecting away some variables would not cause new conflicts which may result in aborting transactions. All the statements in $w_2$ are from the same thread and they do not interleave with the statements in $w_j$ (since there is no pending transaction from $w_j$ that has a statement in $w_2$), so the transactions in $w_2$ would not be affected when some variables are projected away. For the second condition, since there is no aborting transaction in $w_j$, projecting away some variables in $w_j$ would not remove the conflicts that may cause some transactions to be aborted. This means a word like $w'_j \cdot w_2$ can exist.

---

**Contribution 5**

For a transition system, a loop is a finite sequence of transitions if it starts and ends in the same configuration. According to [22], verifying obstruction-freedom of a transactional memory amounts to checking the absence of loops where 1) all the statements are from the same thread, 2) there is an abort, and 3) there is no commit in the loop; and verifying livelock-freedom of a transactional memory is to check the absence of loops where there is no commit, while all the threads have at least one abort.

We establish that FlexTM is obstruction-free by analyzing the generated models, and refute livelock-freedom by finding loops which satisfy the conditions mentioned above. An example of such loops can be $((\text{write},v,t_1),(\text{write},v,t_2),(\text{abort},t_1),(\text{abort},t_2))$, where the transition
\((write, v, t_1)\) starts from the same configuration that the transition \((abort, t_2)\) goes to.

**Summary**

In conclusion for this section, our contributions in the paper are as follows:

1. We propose a parameterized system model of FlexTM, and show that it satisfies four properties which allow us to apply a small model theorem for verifying safety properties.
2. We build four automata based on FlexTM, considering the lazy and the eager conflict strategies, and the two automata of the reference transactional memories for strict serializability and abort consistency.
3. We prove strict serializability and abort consistency of FlexTM by means of language inclusion with a tool.
4. We show that the model of FlexTM satisfies two additional properties, which allow us to apply another small model theorem for the verification of obstruction-freedom.
5. We analyze the model of FlexTM, establish obstruction-freedom and refute livelock freedom.

### 5.2 Paper II: Verification of Cache Coherence Protocols wrt. Trace Filters [2]

In the paper introduced in the previous section, we verify that the words (i.e., traces) produced by the programs running on the FlexTM transactional memory ensure some safety properties, following the conflict resolutions defined in FlexTM. However, safety properties like strict serializability or abort consistency do not specify which value a read instruction should read. If a transactional memory only ensures safety properties, it can happen that the program running on it encounters a cache incoherent state. For example, two processors may end up containing the same cache line in the state of \(M\) at the same time. Such incoherent states should be avoided, since it is not clear which cache line should be loaded when a processor needs to read a value from the line. When two possibly different versions coexist, it is possible that the processor reads the older value instead of the latest one, which may cause unexpected consequences. In this paper, we verify that cache coherence in a transactional memory is maintained when the conflict resolutions are obeyed.

The main contributions in this paper are as follows:

1. We propose a parameterized system model for cache coherence protocols running under transactional memories. The conflict resolution mechanism of the relevant transactional memory is defined as a *filter* which restricts the set of possible executable traces.
2. We show that the obtained model satisfies two properties which allow us to apply a small model theorem. The theorem implies that it is enough
to verify cache coherence with a limited number of variables (i.e., cache lines).

3. In order to use the backward reachability algorithm [1] for verifying the cache coherence protocols, we first need to apply the technique of monotonic abstraction [6] to make the protocols monotonic wrt. some preorder. Our third contribution is that we extend the framework of monotonic abstraction by removing the traces which are not allowed by the conflict resolution strategies.

4. We run the tool ZAAMA [18] which implements the algorithm to verify two cache coherence protocols, namely the TMESI protocol in the transactional memory FlexTM, and UTCP in DynTM. The results show that both protocols maintain cache coherence in the presence of transactional memories.

Contribution 1
We assume that in a transactional memory, each processor has a local cache where all the caches contain the same set of cache lines. A cache line changes its state following the rules of the cache coherence protocol. We build a model for cache protocols in transactional memories that is parameterized in multiple dimensions, namely 1) the number of cache lines, 2) the number of caches, and 3) the size of transactions. Recall that a parameterized model induces an infinite set of transition systems, one for each possible size of the system. A configuration of a transition system in our case represents the states of all cache lines in every cache. A transition is a statement, such as a read or a write to some cache line in some cache, or a commit or an abort of some transaction. The configurations change the states according to the cache protocol rules.

One important feature of the cache protocol model under transactional memories is that there is a filter which restricts the set of possible executable traces. We know that every transactional memory has its own conflict resolution policies. For instance, as we have mentioned in Sec 3.3.3, FlexTM has lazy and eager policies. A lazy policy allows conflicts to exist until one of the conflicting transactions is about to commit, while an eager policy forces one of the transactions to abort once a conflict is detected. We capture the conflict resolution mechanism with filters, which are actually a set of simple patterns where each pattern is the minimal requirement for a trace to violate the conflict policy. For example, in the lazy mode of FlexTM, traces where there are two committing transactions which have write-write conflict(s) with each other will be forbidden. An example of a simplest forbidden trace, namely a pattern, can be (write, x, t1), (write, x, t2), (commit, t1), (commit, t2). For a trace σ and a pattern π, we say that σ matches π if we can get π by removing a set of read or write statements from σ, modulo renaming of the transactions and the variables. We also say that σ is valid wrt. the filter if σ does not match π for any π in the filter, which means that σ is allowed by the conflict resolution mechanism.
Our goal is to check if there is any valid trace that can reach a bad configuration. Intuitively, a bad configuration is a configuration which contains incoherent states wrt. the cache coherence protocol. In order to identify such a configuration, we first need to introduce an ordering between two configurations.

For two configurations $c_1$ and $c_2$, we say that $c_2$ is larger than $c_1$ if

- the number of the caches in $c_2$ is larger than or equal to the number in $c_1$;
- the number of the cache lines in each cache in $c_2$ is larger than or equal to the number in $c_1$; and
- after removing
  - a subset of cache lines in each cache in $c_2$; and
  - a subset of caches in $c_2$;
  we get a new configuration $c'_2$ where the states of all cache lines in $c'_2$ are exactly the same as the states of the cache lines in $c_1$, modulo renaming of the caches and cache lines.

We can characterize a set of bad configurations using the minimal number of cache lines. Any configuration that is larger than a bad configuration with the smallest number of cache lines is also a bad configuration.

Given a cache coherence protocol together with the filter, our task amounts to checking if there exists a trace such that

- it is produced by the transition systems based on the cache coherence protocol, over arbitrary numbers of caches and cache lines;
- it is valid wrt. the filter of the transactional memory; and
- it can reach a bad configuration.

I.e., if there is a bad configuration reachable via a valid trace.

**Contribution 2**

We propose a small model theorem that allows us to restrict the problem to checking the reachability of a bad configuration with a limited number of cache lines. To apply the small model theorem, we show that the protocols satisfy two properties:

- Property 1: **closure property of the cache protocol.** Given two configurations $c_1$ and $c_2$, where $c_1$ is obtained by removing a subset of cache lines in $c_2$ (modulo renaming of the caches and cache lines). If $c_2$ is reachable via a trace $\sigma$, then $c_1$ is also reachable via the trace gained by removing the read/write instructions that operate on the set of the removed cache lines in $\sigma$.

  This property holds in our model. The only possible way of changing a cache line state is by either 1) a read or a write instruction to this cache line, from the local or other transaction(s), or 2) a commit or an abort of the local transaction. When the read/write instructions of some cache lines are removed, the remaining read/write instructions are not changed, so the states of the rest of the cache lines will be the same.
as before. The transactions in the new trace can also abort or commit as before, since there are no new conflicts generated. Besides, if there is any transaction which happens to have all the read/write instructions removed, the commit or abort instruction for an empty transaction is also accepted by our model. This means that the smaller configuration can be reached through the new trace.

• Property 2: closure property of the filter. If a trace $\sigma$ is valid wrt. a filter $F$, then the trace $\sigma'$ is also valid wrt. $F$ where $\sigma'$ is obtained by removing some read or write statements.

This property holds in our model. A trace matches a pattern if it is the same as the pattern after removing some read or write statements from it. If $\sigma$ does not match any pattern in $F$, then whatever read/write statements are removed from it, $\sigma'$ would not match any pattern either. This means that $\sigma'$ is valid wrt. $F$.

Given two configurations $c$ and $c_{bad}$, where $c$ is larger than $c_{bad}$ and $c$ is reachable via some valid trace $\sigma$ wrt. $F$, we construct a configuration $c'$ by removing some cache lines in $c$ such that $c'$ is larger than $c_{bad}$, and has the same number of cache lines as $c_{bad}$. From the two properties, we get that

• $c'$ is also reachable via some trace $\sigma'$ which is obtained by removing the read/write instructions which operate on the set of removed cache lines in $c$ (from Property 1), and

• the trace $\sigma'$ is also valid wrt. $F$ (from Property 2).

A bad configuration can be characterized with only one cache line. An example would be a configuration that contains two caches where there is one line in each cache, and the state of both lines is $M$. In such a case, our verification problem can be reduced to checking whether there exists a bad configuration with one cache line which is reachable via a valid trace wrt. the filter.

**Contribution 3**

Although we have reduced the number of cache lines to one, the number of caches and the size of transactions are still arbitrary. This can be handled by applying a backward reachability algorithm [1] to the transition systems and checking if the systems can reach a bad configuration. Generally speaking, the algorithm starts from the set of bad configurations, and searches backwards for a trace to the initial configuration, following the transition rules. In each iteration, it calculates the set of configurations from which any member of the set of configurations in the current iteration is reachable, following one transition rule. The algorithm terminates when either 1) the initial configuration is found during the backward analysis, which means that a bad configuration is reachable from the initial one, or 2) no more new configurations are found, which means that none of the bad configurations is reachable.

According to [1], there should be a preorder $\preceq$ on the set of configurations such that it satisfies the conditions as follows:
The transition relation is **monotonic** wrt. $\preceq$. More precisely, given three configurations $c_1$, $c_2$ and $c_3$, if $c_1 \preceq c_2$ and $c_1$ can make a transition to $c_3$, then $c_2$ can also make a transition to some configuration $c_4$ where $c_3 \preceq c_4$.

The preorder $\preceq$ is a **well-quasi ordering**, which means that for any infinite sequence $c_0, c_1, c_2, \cdots$, there are $i$ and $j$ where $i < j$ and $c_i \preceq c_j$.

The set of bad configurations is **upward closed** wrt. $\preceq$. More precisely, if $c_1$ is in the set and $c_1 \preceq c_2$, then $c_2$ is also in this set.

The preorder that we use on the set of the configurations, is the one from which we obtain a smaller configuration by removing a set of caches from a larger configuration. However, the transition systems induced by our parameterized model are not monotonic wrt. the preorder. This is due to the **universal conditions** of the transition rules based on the cache protocol. For instance, the TMESI cache protocol in FlexTM defines the rule of $I \rightarrow E$ as follows: upon a read instruction, a cache line in state $I$ can move to $E$ only if there is no other caches containing the same cache line in the state of $S, E, M, TI$ or $TMI$. In our transition systems for TMESI, given three configurations $c_1$, $c_2$ and $c_3$ such that

- $c_1$: there are two caches where each contains one cache line in state $I$,
- $c_2$: there are three caches where two of them contain a line in $I$ and the third one contains a line in $TMI$,
- $c_3$: there are two caches where one of them contains a line in state $I$ and the other contains a line in $E$.

We can get that $c_2$ is larger than $c_1$ since $c_1$ can be obtained by removing a cache with the line in the state of $I$ from $c_2$. Besides, $c_1$ can move to $c_3$ following the rule of $I \rightarrow E$. But there is no configuration $c_4$ such that $c_4$ is larger than $c_3$ and $c_2$ can move to $c_4$. The rule is not enabled on $c_2$ since it contains a cache line in state $TMI$. This means that monotonicity is violated.

To overcome the problem, we apply **monotonic abstraction** [6] which forces monotonicity by removing the caches in the configuration that violate the universal condition. In our example, the cache in $c_2$ which contains a cache line in state $TMI$ will be removed, so the rule can be applied to $c_2$. Monotonic abstraction allows more traces in the transition systems, which means that it generates an over-approximation of the transition system. If the abstract system cannot reach a bad configuration, then it implies that the original systems cannot reach a bad configuration either. Through monotonic abstraction, we retrieve monotonicity of the transition system, and we can therefore apply the backward reachability algorithm.

Our third contribution is that we extend the algorithm by taking the filters into account. More precisely, during the backward analysis, besides configurations, we also need to keep track of the traces. We need to remove the configurations that are backwards generated by the traces violating the conflict resolutions of the transactional memory, so that all the configurations are generated by valid traces wrt. the filter.
**Contribution 4**

We use the tool ZAAMA [18] which implements monotonic abstraction to check 1) the TMESI cache coherence protocol in the FlexTM transactional memory, and 2) the UTCP protocol in DynTM. Our results show that TMESI (resp. UTCP) does not violate coherence with lazy or eager FlexTM (resp. lazy or eager DynTM). However, if we only consider strict serializability of the traces, both TMESI and UTCP become incoherent.

**Summary**

In conclusion, our contributions in this paper are as follows:

1. We propose a parameterized system model for cache coherence protocols in transactional memories, where the relevant conflict resolution strategies are defined as filters to restrict executable traces.
2. We show that the obtained model satisfies two properties which allow us to apply a small model theorem for verifying cache coherence.
3. We apply the backwards reachability algorithm with monotonic abstraction to verify our model, and extend the algorithm by taking filters into account.
4. We run a tool which implements the algorithm and show that the cache protocols TMESI in the FlexTM transactional memory, and UTCP in DynTM are coherent.

**5.3 Paper III: Mending Fences with Self-Invalidation and Self-Downgrade [4]**

Recall that a cache protocol which does not ensure strict coherence can induce a weak memory model (Sec 4.3). A weak memory model allows reordering of program instructions and may cause undesired behaviors for the programmer. An example of such a model is SISD. In this paper, we formalize the SISD memory model and design an algorithm which inserts an optimal set of fences to ensure the correctness of the program running under SISD wrt. a given safety property.

Our contributions in this paper are as follows:

1. We formalize a model which captures the semantics of programs running under the SISD weak memory model. Following the semantics, we consider a set of fences with different costs which can be inserted into the programs to prevent instruction reordering.
2. We design an algorithm which inserts an optimal set of fences into the program under SISD to satisfy a given safety property. The algorithm analyzes the reachability of a set of bad configurations, and may provide a counter-example, which is used by a procedure that inserts fences to ensure the correctness of the program.
3. We compare the SISD memory model with other well-known memory models, and find that it is weaker than SC while it is not comparable with TSO, PSO or POWER.

**Contribution 1**
We formalize the semantics of programs running under SISD as a transition system. A configuration is the state of the system and consists of two parts, namely
- The local part, which defines the local states of the processes. For each process, the local part consists of:
  - the next instruction that will be executed by the process;
  - the values stored in the registers of the process; and
  - the variables (namely memory locations) that are currently cached in the L1 cache, together with their status, i.e., invalid, clean or dirty, and the current value of the variable if it is not invalid.
- The shared part, which defines the value of each variable in the LLC.

A transition can either 1) be performed by a given process when it executes an instruction, such as a read or a write of some variable; or 2) occur due to a system event, which can be:
- Fetch, that occurs when a process does not have the cache line in its L1 cache and fetches it from the LLC.
- Evict, that occurs when the cache line is clean in the L1 cache and is removed from L1.
- Write-LLC, that occurs that when a cache line is dirty in the L1 cache of some process, and the value of it is updated to the LLC.

A safety property states that some set of bad configurations will not occur during any execution of the program, where the bad configurations are defined by the programmer. The execution of the program can be regarded as a run, which consists of a sequence of transitions. Checking the safety property of the program amounts to checking whether there is a run leading to any bad configuration.

Since the programs running under SISD allow reordering of instructions which may cause undesired behaviors, we use fences to prevent the reordering. We consider three types of fences following the program semantics, namely
- **Load-load fence (llfence)**, which invalidates clean lines in the L1 cache, and thus limits the reordering between two read instructions;
- **Store-store fence (ssfence)**, which invalidates dirty lines in the L1 cache, and thus limits the reordering between two write instructions;
- **Full fence**, which empties the whole L1 cache, and thus limits instruction reordering before and after the full fence.

The llfence and ssfence cost less than a full fence. With these three types of fences, we have more options when we perform fence insertion, and can reduce the performance impact when possible.
Contribution 2
Recall that when we insert fences into a program (Sec 4.4), we need to make sure that 1) we insert sufficient fences so that there will not be any undesired behaviors caused by the program, and 2) we insert an optimal set of fences so that the performance of the program will be affected to the lowest extent. As manual fence insertion is time-consuming and error-prone because of the complicated behaviors of programs under SISD, we design an automatic fence insertion algorithm which we have introduced in Sec 4.4.

The algorithm is counter-example guided, which means that it checks the reachability of a set of bad configurations for a given program. If such a configuration is reachable, then the algorithm provides a counter-example, namely a run which leads to the bad configuration. A fence insertion procedure will then analyze the run and provide a required set which contains all the fences that can prevent the run from occurring. Based on the required set of fences, the algorithm chooses the optimal set of fences, inserts them in the program, and checks the reachability again. The algorithm terminates when 1) no bad configuration is reachable, which means that the program is correct with the current set of fences inserted, or 2) a counter-example with no reordering is found, which means the program is incorrect even under SC. The optimality is calculated from the number of the occurrences of fences and the cost of different fences.

Contribution 3
We compare the SISD model with some well-known memory models, and use the set of litmus tests to illustrate the difference. In particular, we find that the SISD model is weaker than SC and not comparable with TSO, PSO or POWER. We also find that the ISA2 and IRIW programs have runs under SISD that can cause non-SC behaviors.

Summary
Our contributions in this paper are as follows:

1. We define a formal model of the semantics of programs running under the SISD weak memory model. Following the semantics, we feature a set of fences with different costs which can be used to prevent instruction reordering in a program.
2. We design an algorithm which inserts an optimal set of fences in the program under SISD to satisfy a given safety property.
3. We compare the SISD memory model with other models, and conclude that it is weaker than SC but not comparable with TSO, PSO or POWER.
6. Conclusion and Future Work

In this thesis, we have considered the analysis of the models based on transactional memories, cache coherence protocols in the presence of transactional memories, and weak memory models induced by some cache coherence protocols.

First, we have proposed detailed models of the FlexTM hybrid transactional memory for two modes, namely a lazy mode and an eager mode. We have adapted a small model theorem for software transactional memories and established a number of properties that are satisfied by FlexTM. The properties allow us to apply the theorem that reduces the problem of verifying safety and liveness properties of transactional memories with an arbitrary number of variables and threads to checking the properties for a fixed number of these components. To verify the transition systems with a fixed number of variables and threads, we have built four automata of FlexTM and two reference transactional memories, and automatically checked language inclusion between them. The results show that the two properties are indeed satisfied by FlexTM. Furthermore, we have established obstruction freedom and provided a counterexample showing that livelock freedom is not satisfied by FlexTM. An interesting direction for further work is to develop a method that can automatically establish the properties allowing the application of the small model theorem and extract the corresponding systems to carry out verification.

Second, we have for the first time addressed parameterized verification of cache coherence protocols running under transactional memories. We have built a formal model of such protocols, and defined a filter which represents the conflict resolution policies in order to restrict the input sequences to the cache protocol. We have shown two properties of the model that allow us to apply a small model theorem, based on which we can reduce the verification problem to verifying parameterized cache protocols with only one cache line. We have also applied the technique of monotonic abstraction to use a classical backward reachability algorithm for infinite state systems. In order to take the conflict resolution strategies into account, we have extended the backward reachability algorithm by removing the traces that are not allowed by the filter. We have run a tool which implements the algorithm to verify two cache coherence protocols, namely TMESI in the FlexTM transactional memory and UTCP in DynTM. We have concluded that cache coherence is maintained by both protocols in the presence of the respective transactional memories. In the future, we intend to verify cache coherence protocols based on more detailed models. An example of such models can be the ones that contain intermediate states which are induced by processors waiting for the reply from the memory.
Finally, we consider verification of safety properties of programs running over Self-Invalidate and Self-Downgrade (SISD) cache coherence protocols. We have defined a formal model capturing the semantics of such programs. We also have designed an algorithm for inserting a set of optimal fences in the program under SISD, taking into consideration that the fences are with different costs. The goal is to make the program satisfy some safety property while maintaining the maximal degree of efficiency. Furthermore, we have compared SISD with other memory models and concluded that it is weaker than SC while not comparable with PSO, TSO or POWER. For future work, we will consider the robustness property. More precisely, we will check whether a given program running under SISD presents any behaviors not observed under SC.
References


[26] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free


Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1665

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-347787