JAIST Repository
https://dspace.jaist.ac.jp/

Title	Verification of automotive operating systems for multi-core processors
Author(s)	マリーヌアン, パッターウット
Citation	
Issue Date	2015-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/12933
Rights	
Description	Supervisor:Toshiaki Aoki, School of Information Science, Master

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Verification of automotive operating systems for multi-core processors

Pattaravut Maleehuan

School of Information Science
Japan Advanced Institute of Science and Technology
September, 2015

Master's Thesis

Verification of automotive operating systems for multi-core processors

1310205 Pattaravut Maleehuan

Supervisor : Toshiaki Aoki
Main Examiner : Toshiaki Aoki
Examiners : Masato Suzuki
Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology

August, 2015 (submitted)

Abstract

Currently, the automotive systems provide many functionalities for automobile. Although those functionalities are able to support our lives, the violation of those functionalities may be serious issue that we have to concern. Obviously, the automotive systems require high-reliability of the system to ensure safeness of our lives. In this research, we specifically focus on *operating systems* of automotive systems. The operating system is the basis component which provide services to serve application software. Thus, the correctness of the operating systems is the serious condition to correctly implement the operating system kernel.

In addition, since the demand of usage is increasing, the *multi-core processors* are adopt in the automotive systems for performance improving. The multi-core system is the system that have multi-processors with shared-memory. In the shared-memory systems, according to [Gha95], the results of programs might be not same as the *sequential execution*, which is the execution which follow the program order specified by programs. As programmer's point-of-view, these results maybe unexpected results. These results are affected by *memory consistency models* which are define the behaviors of memory in shared-memory systems.

In the shared-memory systems, the systems allow each processor to access the memory locations simultaneously. Moreover, each processor are able to issue the memory accesses out-of-order because of optimization mechanisms. Since each processor is independent to each other, the access order of memory access might be different. Hence, the memory maybe not consistence among processors. The specifications of memory consistency models ensure the execution order of memory accessing to shared-memory locations. Unfortunately, these behaviors are not appeared explicitly. It might be difficult to consider the behaviors of program execution.

Our research aims to verify the automotive operating systems for multi-core systems. Since the behaviors of program execution are not appeared explicitly, the verification might be difficult. In addition, the behaviors of program executions are affected by the hardware architecture. That means we cannot verify the programs on multi-core systems by themselves. Hence, this research will provide the verification which take the hardware behaviors into account for ensure the correctness. Moreover, the verification of the whole operating systems will be difficult because operating system is a complex system. Therefore, the scope of this research considers only multiprocessor programs to provide the verification for multi-core systems.

In software development, there are many approaches to ensure the correctness of the software. Due to the automotive operating systems require high-reliability, the *formal verification* is adopt in this research. Since the behaviors of program execution in multi-core systems might be complicated, we apply the *theorem proving* approach instead of model checking. Therefore, due to the behaviors of hardware is significant issue, this research provides the formal model which represent such behaviors. Then, the verification method is proposed to provide the proofs based on our formal model.

Acknowledgment

First of all, I would like to express my sincere gratitude to my supervisor, Associate Professor Toshiaki Aoki, for his guidance, support, and encouragement. Since I was a student at Chulalongkorn University in Thailand, he gave opportunity to interview for studying abroad at JAIST. In addition, he also assisted me in advanced to get to Japanese Government (Monbukagakusho) Scholarship. As for my academic life in JAIST, every time I have problems about the research or stuck on some issues, he always give a time to discuss and give guidance to solve those problems. In addition, as my presentation skills are not good, he always fix my mistakes by exploring the issues that need to be explained, then give examples and guidance. His guidance and comments usually are helpful for me to improve my skills.

Beside my supervisor, Associate Professor Toshiaki Aoki, I would like to show gratitude to Assistance Professor Yuki Chiba. He usually help me in advance to do the research and my life in Laboratory. Due to my research uses many mathematics definitions, he usually suggest the correct mathematics definition in advanced. Every times I discuss with him, he usually check it in detail, and give the comments to improve and correct my works.

Finally, I also would like to give my special thank to my friends and seniors in JAIST for their helps, suggestions and encouragements during live in JAIST. I also need to give my thanks to Thais friends, whose usually help and make me have a great time in Japan.

Contents

1 Introduction **1**
- 1.1 Problems . . . 2
- 1.2 Objective . . . 3
- 1.3 Approach . . . 3
- 1.4 Organization . . . 4

2 Multi-core systems and Verification **5**
- 2.1 Multi-core systems . . . 5
  - 2.1.1 Terminology . . . 6
  - 2.1.2 Behaviors for performing programs on multi-core systems . . . 7
  - 2.1.3 Hardware optimization . . . 9
  - 2.1.4 Memory consistency models . . . 11
- 2.2 Formal Verification . . . 15

3 Formalization for multi-core systems **16**
- 3.1 Target multi-core systems . . . 16
  - 3.1.1 Abstract model . . . 17
  - 3.1.2 Fetching instruction . . . 17
  - 3.1.3 Hardware operations . . . 18
  - 3.1.4 Target memory locations . . . 21
  - 3.1.5 Memory consistency models . . . 21
- 3.2 Formal model . . . 25
  - 3.2.1 Preliminaries . . . 25
  - 3.2.2 Syntax . . . 26
  - 3.2.3 Semantics . . . 32
  - 3.2.4 Conditions for memory consistency models . . . 39
  - 3.2.5 Instruction set . . . 44
- 3.3 Sample executions based on our semantics . . . 49

4 Verification method for multi-core systems **53**
- 4.1 Verification of multiprocessor programs based on our semantics . . . 53
- 4.2 Verification method . . . 55
  - 4.2.1 Provide *invariant* as the predicate . . . 55
  - 4.2.2 Define the sets of states . . . 56

4.2.3 Provide the induction proofs . . . 56

5 A case study : a mutex lock verification **57**
5.1 Apply verification method . . . 58
5.2 Proofs . . . 60

6 Evaluation **66**
6.1 Evaluation of our formal model . . . 66
6.2 Evaluation of our verification method . . . 67

7 Related works **68**
7.1 OS Verification . . . 68
7.2 Formalization . . . 69
7.3 Formal Verification . . . 69
7.4 Originality . . . 69

8 Conclusions and Future works **71**
8.1 Discussion . . . 71
8.1.1 Formal model . . . 71
8.1.2 Verification method . . . 71
8.2 Future works . . . 72
8.2.1 Ensuring the proposed formal model . . . 72
8.2.2 Improving the verification method . . . 72

Bibliography **73**

List of Figures

1-1 Sample programs on different cores 2

2-1 General shared-memory systems 6
2-2 Simple instructions 7
2-3 Sample for shared-memory systems 8
2-4 Example for bypassing read access 10
2-5 Representation for the sequential consistency (SC) model. 12
2-6 Sample programs 13
2-7 Representation for the total store ordering (TSO) model 13
2-8 Example programs for the TSO model 14
2-9 The partial store ordering (PSO) model 14

3-1 Abstract model for multi-core system 17
3-2 Example of out-of-order updates 19
3-3 Example of read-modified-write requirement 20
3-4 General model for shared-memory 22
3-5 A sample program for considering execution 49

4-1 A sample program 54
4-2 Set of states of programs 4-1 54

Chapter 1

Introduction

Currently, computer systems are attached to electronic systems around us. This research is interested in *automotive systems* which relate with our lives, such as automobile. In the *automobile systems*, computer systems are adopt to control many parts inside the automobiles, such as ABS braking systems. In addition, the most systems of automotive systems relate to our lives. Due to the fact that some violations of the systems may risk our lives, , we should ensure that the automotive systems is working correctly. In the automotive systems, *automotive operating systems* are the basis components which provide services for application software to control the hardware architecture. Hence, *automotive operating systems* are significant components which we focus in this research.

To improve the performance of the systems, the *multi-cores systems* are used in the modern automotive systems. In the multi-core systems, there are numerous optimization techniques to reduce the *memory latency*, such as bypass the read accesses. However, such techniques will change the *order of executions* in the systems. Even if the order of executions is changed, uniprocessor systems[1] are able to keep the correct results follow the *program orders*[2] Due to the fact that multi-core system is *multiprocessors system* with *shared-memory*, each core are able to read/write the shared-memory simultaneously. In addition, the order of updates, the results of write accesses to shared-memory, maybe appeared to each core independently in shared-memory systems. To keep *consistency* of the orders among cores, such optimization techniques have to restrict some behaviors in the hardware. So, we may loose the performance of optimization techniques. However, modern processor architectures assume that some executions of programs are acceptable. Hence, there are *memory consistency models*, or *memory models*, which allow some out-of-order executions to be happened for improving the performance of the models. Nevertheless, due to some executions may produce unexpected results from the programs, ensuring the correctness of the programs for multi-core systems becomes complex.

In software development, there are many techniques to confirm the correctness of the programs, such as providing test cases. Nevertheless, the automotive systems need high-reliability of the systems. Hence, in this research, *formal verification approach* is proper

[1] A computer system that has a single CPU

[2] The order of instructions that appear in the program

```
1  A = 1
2  b = B
3
```

Program A

```
1  B = 1
2  a = A
3
```

Program B

Figure 1-1: Sample programs on different cores

to ensure the correctness of the systems. Because the formal verification approaches offer *rigorous approaches* for the verification.

Since the operating systems are close to hardware architecture. Most of implementations are usually implemented by *mixing assembly and c languages*. As we said before, multi-core systems are able to allow out-of-order executions of *memory accesses*. Nevertheless, programmers usually assume that the executions will be *sequential executions*. Many *program verification methods*, such as Hoare logic, consider the instructions will be executed as atomic instructions in a sequential way. Therefore, unexpected results might be happened in the multi-core systems.

1.1 Problems

As we mention before, the *hardware behaviors* might change the order of executions of programs in multi-core systems. Even if we have ensured the programs, it's not enough to confirm programs are able to be performed correctly in multi-core systems. Firstly, let's consider the programs in figures 1-1. As for the programs, we define *capital letters*, A and B, as *global memory locations* and small letters, a and b, as *private memory locations*. Each program will be performed on different core on same multi-core systems. As verification of programs, we usually verify the programs in the interleaving ways as sequential executions. However, as we mention before, the order of executions will be changed due to hardware behaviors in multi-core systems. As the programmer's point of view, the results of (a, b) will never be $(0, 0)$. Nevertheless, some *multi-core systems* allow operations *'b = B'* and *'a = A'* to be executed before *'B = 1'* and *'A = 1'*. Therefore, the results of $(a, b) = (0, 0)$ might be possible in the multi-core systems. Obviously the verification of programs performed on multi-cores systems *cannot be verified by themselves*. Because behaviors of multi-core systems have not shown explicitly in the programs.

Due to the behaviors of multi-core systems can change the order of executions, the degree of non-determinism of multi-core systems will be greater than programs considered as *interleaved executions*. The verification becomes difficult because of degree of non-determinism. Since the behaviors of multi-core systems are significant issues to be concern, the verification should be able to ensure the correctness of programs with the behaviors of multi-core systems.

1.2 Objective

The objective of this research is to propose a verification method for ensure the correctness of *automotive operating systems* for multi-core systems. Since the operating systems are close to hardware and the behaviors of multi-core systems affect the execution of the programs, behaviors of multi-core systems are the most importance in the verification. Therefore, the hardware behaviors of multi-core systems will be taken into account. The proposed verification method also should able to ensure the correctness of the programs in multi-core systems.

In addition, since operating systems are the complex systems which contain a lot of functionalities to service the software application. The verification of the operating systems become difficult. As this research we focus on verification of programs on multi-core systems, thus verification of programs which service the multi-core functionalities is enough to ensure the correctness of programs in multi-core systems. Because the out-of-order execution is the significant issue to be concerned when the programs are executed on different cores.

1.3 Approach

Since the operating systems usually implemented by mixing assembly with C languages. These languages offer different programming paradigms to write programs. In addition, assembly language is more close to the hardware which affect the execution order of programs. Moreover, considering the complied programs as assembly languages also avoid the compiler optimization which also change the program implementation. Therefore, this research will focus on assembly programs in the verification.

In order to propose a verification method for programs executed on multi-core systems, as we mention before, the behaviors of multi-core systems should be taken into account. Thus, the verification also need a *formal model* which represent the multi-core systems. The formal models should be an abstraction of multi-core systems and capable for verification of programs. In addition, the execution order of memory accesses in multi-core systems is influenced by *memory consistency models*. In the multi-core processors, there is a variation of memory consistency models which allow different behaviors of executions to be happened in the systems. Therefore, the memory consistency models are necessary issues to be formalized in the formal model.

Then, the verification method will be proposed to deal with programs executing on multi-core systems based on our formal model. The verification should be able to cover the executions of programs in the multi-core systems. As the executions of instructions will not be an atomic step and more hardware behaviors will be appeared in the formal model, the verification becomes more complex due to the degree of non-determinism. Thus the verification method should be capable for programs in our formal model which aren't executed as *sequential executions*.

1.4 Organization

In this thesis, the next chapter will explain the preliminaries for this research. Firstly, we will consider the behaviors of multi-core system. We specifically consider the behaviors of program executions which is necessary in verification. Then, we will explain about formal verification. The formal verification will be adopted to our research. In the verification, we specifically use the theorem proving for ensuring the correctness.

In chapter 3, we provide the formal model of our work for verification. The model that we have proposed captures the behaviors of program execution on multi-core processors. Since the optimization mechanisms inside multi-core systems change the order of execution, the formal model is provided to capture those behaviors in the verification. In the last section of this chapter, we show the sample of execution based on our formal model as an evidence to convince that our model can be used.

In chapter 4, we will adopt the proposed formal model in the verification. In that chapter, the method will cover the possible execution paths produced from our formal model. In addition, we also show the proofs as a case study in chapter 5 to show that our method can verify the programs executing on multi-core systems.

Since we have already proposed the formal model and verification method, those model a method will be evaluated in the chapter 6. Then, the chapter 8 will discuss about them and conclude the research.

Chapter 2

Multi-core systems and Verification

This section will explain about the underlying system behaviors that perform programs on multi-core systems. Normally, we know that the assembly instructions will be fetched to a processor and be executed in that processor. Behind an instruction execution, the processor uses *micro-operations* and *memory accesses* to execute the fetched instruction. That means a single instruction isn't appeared as atomic. In addition, there are many mechanisms to improve the performance of the systems. Such improving techniques will be used for reducing the *memory latency*. Some techniques allow the latter memory accesses to be executed before the earlier memory accesses. Hence improving techniques will affect the execution order of *memory access*. In the section 2.1 will explain the behaviors of multi-core systems which affect the executions of the programs in details.

In multi-core systems, memory accesses are used in programs which communicate via shared-memory. There are optimization mechanisms provided to reduce the *memory latency* of memory accesses. The optimization mechanisms will be in either *compiler optimization* or *hardware optimization*. In the compiler optimization, the compiler uses profile-guide optimization to reordering the assembly instructions. Anyway, in this research, we focus on hardware optimization mechanisms which affect the hardware behaviors. The section 2.1.3 will explain the behaviors of optimization mechanisms in detail. Moreover, the multi-core systems will have it's own memory consistency model which affects the order of executions of programs. That means the possible executions of the programs may not be as sequential executions. The section 2.1.4 will explain the behaviors of memory consistency models in details.

To ensure the correctness of the programs for multi-core systems, we choose *formal verification* approach. The section 2.2 explain the idea of *formal verification*, describes the existing approaches, and consider the related works which apply the formal verification to verify operating systems.

2.1 Multi-core systems

Normally, the multi-core systems are the *multiprocessor systems* with *shared-memory systems*. The program will be performed in a processor that belongs to one core and

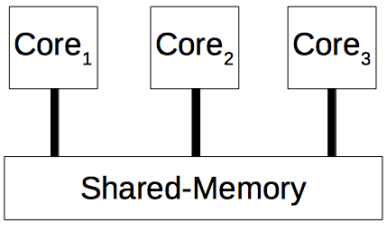

Figure 2-1: General shared-memory systems

communicate each other by *memory accesses* via *shared-memory* as the figure 2-1. The programs usually are written in assembly code which belongs to *Instruction Set Architecture (ISA)* of processors, such as ARM processors. For each instruction appeared in the program will be fetched to the *execution unit*, *or pipeline*, inside a processor to be executed as *micro-operations* and *memory accesses*. Each of *micro-operations* and *memory access* will be issued from the processor to compute in the core or issue to outside the core for accessing the shared-memory. This section will consider the behaviors of executing programs on multi-core systems.

2.1.1 Terminology

Firstly, to describe the behaviors inside the multi-core systems, we would like to introduce overview of terminologies that will be appeared in this research.

A **multi-core systems** is the *multiprocessor with shared-memory* systems. It consists of a collection of *cores*, *shared-memory locations*, and *a network*, which connect cores and shared-memory locations. A **core** is a group of components that contains *a processor*, *execution units*, *buffers*, and *private memory locations* such as *registers*. The main role of the **processor** is to fetch the instructions from *a program* to be performed in the *execution unit*. A **program** is the sequence of instructions that will be performed in a processor. Each of programs will be placed on a core separately in multi-core systems. The fetched instructions are usually called **instances** to identify the execution in that time, because same instruction is able to fetched many times. The **execution unit** will issue a sequence of *micro-operations* and *memory accesses* for performing the fetch instructions. The **micro-operations** are the low-level instructions which transfer data among registers and buses inside a core. The **memory access** is the low-level instruction that access the memory location either update or read the value. There are situations that the processor try to perform a micro-operation and a memory access while earlier operations or accesses is not finished yet. The **buffers** will be used to handle such micro-operations or memory accesses which cannot be performed yet. The memory accesses which need to access the shared-memory locations will be issued to the *network* outside the core.

The **network** is an external bus that connect the cores and *shared-memory locations*. The memory accesses which access shared-memory locations might have a delay of memory latency to access the external memory locations. The modern processors usually have *caches* inside a core to reduce memory latency.

In the systems, we have information of **program order of instances** and **program**

```
1 MOV r1, #1
2 MOV r2, r1
```

Program A

```
1 MOV r1, #1
2 MOV r2, #2
```

Program B

Figure 2-2: Simple instructions

order of micro-operations and memory accesses. These order is specified by the time that instances, micro-operations, or memory accesses are instantiated. These *program orders* are *total orders* in the it's own core. However they will be appeared as *partial orders* in the multi-core systems. Such program order will be used to consider about execution behaviors in the future, because *memory consistency models* will be generalized in formal model. Nevertheless, these information is not appeared in the hardware explicitly.

2.1.2 Behaviors for performing programs on multi-core systems

As the programs will be placed on each core separately, first we will consider the program executed in a one core. Then, the *memory accesses* which issued to the network will be considered later to interact with shared-memory locations.

Performing instructions

In order to execute a program assigned to the core, a processor inside a core will fetch an instruction to the *execution unit*. The fetched instructions are considered as *instruction instances*. Even though such instances are instantiated from same instruction, the instances should be different to each other. The *program order* of instances is depend on the order of fetching instances from a programs. Obviously the program order of instances is a *partial order* among cores.

After an instruction is fetched, execution unit will issue *micro-operations* and *memory accesses* to perform an instance of such instruction. These micro-operations and memory accesses also have their *program order* to keep order of executions of instance's operations.

Moreover, fetching instruction is able to fetch the next instruction immediately if the registers which will be used by such instructions are available to use. Let's consider the programs in figure 2-2, In the *Program A*, we have to wait instruction at line 1 before fetching the next instruction because of dependency of register $r1$. Nevertheless, in the *Program B*, we are able to fetch the instruction at line 2 even if the instruction at line 1 is not finish yet. The fetched instructions will be placed on available *execution units*, as known as *pipeline* in modern processors.

Read and write accesses and their buffers

Let's consider the memory accesses which issued from execution units. The accesses are either read accesses or write accesses to memory locations. Generally, accessing the memory always have a memory latency of memory operations. Due to memory latency, a processor have to be *stall* until the outstanding memory access is finished. Hence, the

```
A = 1
B = 1
```

Program A

```
u = B
v = A
```

Program B

```
x = B
y = A
```

Program C

Figure 2-3: Sample for shared-memory systems

read and write buffers have been introduced to reduce memory latency. If there are any read or write accesses as the next memory accesses which independent from outstanding access, such read or write accesses are able to be issued from the buffers and performed immediately. Because such accesses will be perform in different locations which do not need to stall the processor. The access locations of memory accesses can be either private or shared memory locations.

Atomic operations

In the hardware architecture, the instructions are not be appeared as atomic instructions. Each instruction described in assembly language is performed as *micro-operations* and *memory accesses*. However, such atomic instructions are necessary in the multi-core systems. The hardware also have mechanisms to service the instructions which require atomicity. Normally, atomicity behaviors will consist of read and write access which access the same location. The atomicity behaviors ensure that once the read access is executed, there is no any write access from another processors which access the same location can update the value before the atomic's write access. The such mechanism can be implemented in either processors or memory. In a processors, atomicity can be supported by controlling the cache coherency protocol. Once the atomicity is needed, the processor will request the exclusive owner of the locations. Then, the cache coherency protocol will lock that location until the write access is finished. In the related work [Gha95], they refer to such kind of operations as *read-modified-write operations*.

Accessing shared-memory locations

Let's consider the memory accesses which access the memory locations. Once the read and write accesses are finished by accessing the caches, the caches also have mechanisms to maintain the coherency of locations among cores. Although the *cache protocol* is able to handle the coherency among cores, there are the execution order issues to be concerned. Due to each core is independently and the optimization mechanisms are able to change the order of execution, the order of updates might be different. In multi-core systems, there are *memory consistency models* which describe the behaviors of memory accesses for shared-memory systems. Such consistency models will affect the order of updates among cores. Let's consider the programs in figure 2-3, once the 'A = 1' and 'B = 1' are issued, some *memory consistency models* can allow *programs B* to read the value of 'A = 0' and 'B = 1', and *program C* can read the value of 'A = 1' and 'B = 0'. We can see that the order of updates is appeared to each core independently.

In the related work [AAS03], they propose the formal model for share-memory systems. That model also take the memory consistency models into account for capturing the behaviors of shared-memory. As for shared-memory systems, they have proposed *view-orders* to each core to describe the order appeared to the core in each step of executions. That approach also applicable to simulate the order which is appeared to each core independently.

Fencing operations

Due to some systems allow out-of-order executions to be happened, such systems also provide *fencing operations*, or *memory barrier*, to enforce some parts of programs to be executed follow the program order. However, in each processor architecture also have different mechanisms of fencing operations. Fencing operations will separate the group of operation to be two groups. They usually define groups as *past operations* and *future operations*. For the future operations, each system is able to define it's group, such as consider only write operations to the same location as operations in *past operation* group. To control the order of executions, some systems put the *fence operations* into a write buffer to enforce the executions.

2.1.3 Hardware optimization

This section will explain the hardware optimization mechanisms that usually be used in the multi-core systems. However, some optimization mechanisms, such as lock-up free cached, are ignored in this research. Due to we need to abstract the behaviors of hardware for programs verification, the cache behaviors might not necessary to consider. If the cache components are taken into account, the cache coherency protocols also might be considered in details. Moreover, the effect of some mechanisms also lead to reordering of executions. Such reordering also is able to be simulated by *buffers behaviors*. In addition, to maintain the *coherency*, *memory consistency models* are also used to control the order of updates among cores. Therefore, as optimization mechanisms proposed in this section might be sufficient to verify the programs for multi-core systems.

Out-of-order issuing

In the fetching instruction, normally the instructions are fetched in order decided by a program counter. Sometimes the next instruction should be wait until the necessary registers are available, as described in section 2.1.2. Instead of waiting, this technique stores the instruction into *reservation station*, or *instruction buffer*. The *reservation station* will issue an instruction that it's required registers are available. That means this behaviors will allow *out-of-order issuing* of instructions. In some cases that the programs are executed in multi-core systems, this behavior is known by only it's own processor. Therefore, the unexpected results maybe produced.

```
1 A = 1
2 B = 1
3 a = A
```

Program A

```
1 x = B
2 y = A
```

Program B

Figure 2-4: Example for bypassing read access

Non-blocking read access

In the execution units, the instruction will be performed as *micro-operations* and *memory accesses*. As for the read accesses, there are some situations that the read access cannot be performed immediately. The causes maybe *read miss* in caches or the memory locations is not available yet. Hence, this mechanisms have been introduced to skip such read accesses to perform next *micro-operation* or *memory access*. However this read access will be performed again once the value of the read is needed. This behavior is able to be realized by *read buffers*.

Bypassing read access

The write accesses normally are putted into write buffers. To issue the read access as the program order, the read operation usually have to wait until the previous write accesses already be issued from buffers. In this case, the processor should be stall itself before perform the next operations. To reduce the stalls, *bypassing read access* have been introduced. The read access can be performed immediately if and only if there are no write accesses that access to the same address as the read access. Hence, this behavior will cause that a read access may be executed before write accesses specified as earlier operations. In the same processor, this behavior will not produce unexpected results. Nevertheless, in the multi-core systems, the order of some write accesses and read accesses maybe significant order to be considered.

For example, let's consider the figure 2-4, we define 'A =1' and 'B=1' as write access, and the remaining are read accesses. Assume that the write access 'A=1' already executed in the shared-memory and 'B=1' is stored in the *write buffers*. In this case, the read access 'a = A' is able to read the value 'A = 1' from shared-memory immediately, even if the write access 'B = 1' is not executed yet. In this case, the result of (x, y) is $(0, 1)$ can be happened in the multi-core systems.

Read forwarding

This mechanism also reduces the stalls of that processor by immediately issue the read access if and only if there is a write access stored in write buffers which access the same memory locations as the read access. However, such return value should be the value from the last write access that appeared in the buffer. Although this mechanism can reduce the stalls in processors, some *hardware* are not allow this mechanism to be implemented due to it may provide some unexpected results.

Non-FIFO read/write buffers

Generally, buffers usually act like queues which issue an entity in order as First-In-First-Out(FIFO). In some cases the earlier read or write accesses cannot be executed yet, because accessing memory location is not available. Moreover, In the case of *out-of-order issuing*, some accesses should be issued in the program order, but buffers might have earlier accesses which should be issued later. Hence, such accesses will be selected to be issued before the earlier accesses in the buffers.

2.1.4 Memory consistency models

Due to the fact that the execution order on multi-core systems is significant issue to verify the correctness of the programs, this section will explain about memory consistency models. Currently, there are many models have been proposed in the computer systems. In each systems also provide different hardware behaviors to keep the memory consistency among cores. Fortunately, there is a related work [Gha95] that generalize the memory consistency models to their framework. They provide constrains to consider valid executions on each model. So, such framework will be applied to formalize the multi-core systems in our work.

A *memory consistency model*, or *memory model*, is the model to describe how read and write access will be executed in the multiprocessor with shared-memory systems. Due to the order of read and write accesses are able to be changed, the memory consistency models are used to specify how it can be changed. As the programmer's point of views, the program should be executed follow the program order. Such behaviors that every executions are executed in order are called *sequential consistency model.Relaxed models* have been introduced to allow more optimization techniques can exploit the hardware architecture.

Framework for representing memory consistency models

First of all, to describe the memory consistency models, we need a framework to describe such model in a formal way. The framework that we use to describe the various of memory consistency models is adopt from [Gha95]. The framework provides conceptual abstract structure of shared-memory systems. The figure 2-5 show basic representation for sequential consistency model. In their conceptual system for sequential consistency consists of n processors sharing a single logical memory. In the conceptual systems, they show the concept in the programmers point-of-view. That means caches components is not appeared in this systems, even if it's used to implement in the practical systems.

The read and write accesses are treated as **R** and **W** for representing as read and write operations. A read operation is assumed to complete once the return value is bound. A write operation is assumed to complete once the shared-memory is updated to be new value. This framework also assume that read and write operations are issued from processor as *program order* though the completed order might be out-of-order. A atomic operation is represented as read-modified-write operation in this framework. The read-

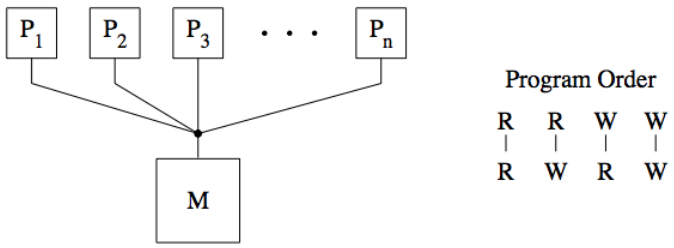

Figure 2-5: Representation for the sequential consistency (SC) model.

modified-write operation is treated as both a read operation and a write operation. The most of memory consistency models require that there is no write operations appeared in execution order between the execution of atomic operation.

To represent memory consistency models, this framework proposes a set of constraints for the model. There are types of constraints for this framework. The first type of constraint on the execution of memory operations relates to program order . This is referred to as the *program order requirement*. Such requirements are figured in the right-side of the figure 2-5, they represent all pairs of read and write operations issued by same processor that follow one another in program order: read followed by a read, read followed by a write, write followed by a read, and write followed by a write. The *lines* provide constraints that the operations between the line should be maintained the program order in the execution order. The second type of constraint relates to the values returned by reads and is referred to as the *value requirement*. As for the sequential consistency, the value requirement is as follows :*a read is required to return the value of the last write to the same location that completed in memory before the read completes.* This requirement is referred as *memory value requirement*. Some of the memory consistency model have different value requirement that allows a processor to read the value of its own write before the write completes in memory. This latter requirement is affected by optimization mechanisms such as *read forwarding* that allows a read to return the value of a write from a write buffer. This requirement is referred as the *buffer-and-memory value requirement*. Other models may impose additional types of constraints to describe the models.

Sequential consistency model

According to framework in [Gha95], the conceptual systems as shown in figure 2-5 must obey the program order and memory value requirement described above to satisfy sequential consistency. An implementation obeys a given model if the result of any execution is the same as if the program was executed on the conceptual system. Therefore, a practical system need not satisfy the constraints imposed by conceptual systems (e.g. program order) as long as the results of its executions appear as if these constrains are maintained.

Let's consider the programs in figure 2-6, this example is taken from the related work [Gha95]. Assume that these programs are working together in a multi-core system, the capital letters, A and B, refer to the shared memory locations. the small letters, a and b, refer to private memory locations inside a core. Obviously these programs are working

```
1 A = 1
2 B = 1
```

Program A

```
1 b = B
2 a = A
```

Program B

Figure 2-6: Sample programs

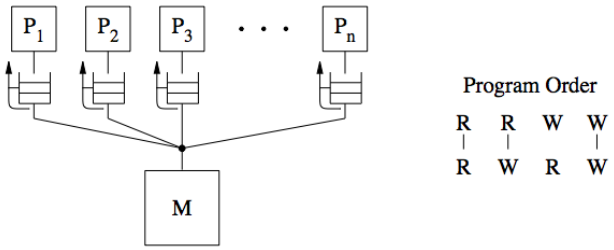

Figure 2-7: Representation for the total store ordering (TSO) model

on different cores. As for sequential consistency models, the results of executions should follow the *program order*. That means some out-of-order executions also are allowed to be happen in the sequential consistency models if and only if the correct results is maintained. Hence the possible results of *(a,b)* from this programs are $\{(0,0),(1,0),(1,1)\}$. In the *sequential consistency models*, they allow operation *'A=1'* and *'B=1'* to be changed the order of executions. Nevertheless, if *'B=1'* already executed, the model will enforce the operation *'A=1'* to be executed before *'a = A'*. Therefore, the result of *(a,b)* is (0,1) will not be happened in this model.

Since multi-core system has many hardware components working together. Moreover each core is working independently. To maintain the sequential consistency among cores, there is a work [Lam79] that describe how to correctly execute multiprocessor program on multiprocessor computer. However, there are another models which allow some results to be happened called *relaxed models*. These models assume that some results of the programs are acceptable and the significant order in the program should be maintained by programmers. Moreover, to control the significant order in some parts of programs, they provide fencing instructions to enforce the order of operations. This kind of models will be explained in the next section.

Relaxed models

The relaxed models are memory consistency models which allow the execution order of memory access to be changed. Each model will provide different constraints for its own conceptual system. Although the order of execution is allowed to be changed implicitly, they provide some mechanisms to restrict the significant program order to avoid the out-of-order execution of the order. We will give some sets of constraints for memory consistency models. Each of models will provides different requirements to maintain the behaviors of system.

The figure 2-7 represent the conceptual system for total store ordering. The systems

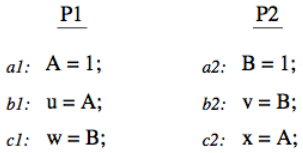

Figure 2-8: Example programs for the TSO model

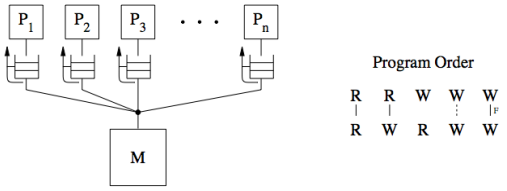

Figure 2-9: The partial store ordering (PSO) model

that use this model will be allow to finish the write operations out-of-program order. This conceptual systems is similar to figure 2-5 with a single logical memory shared by the processors. In contrast, this system provide a buffer buffer between each processor and the memory. Since we assume that operations will be issued in program order. they use the buffer to capture the behaviors of the operations are not necessary issued in the same order to memory. Hence, *program order requirement* places the constraints on the reordering that can tale place within the buffer. They also use the buffer to capture the behavior of models that allow a read to return the value of conflicting write in the buffer before the write is actually issued to the memory. The program order requirement for the TSO model is shown on the right side of figure 2-7. It similar to sequential consistency model, the only difference is in the program order from a write followed by a read is allowed to complete out-of-order.

Let's consider the example programs from [Gha95] in figure 2-8, under the sequential consistency model, the outcome (u,v,w,x) = (1,1,0,0) is disallowed. However, this outcome is possible under TSO model because read operations are allowed to bypass previous write operations, even if they are to the same location. Therefore the sequence (b1,b2,c1,c2,a1,a2) is a valid order for TSO model. Obviously the value requirement still requires b1 and b2 to return the values of a1 and a2, respectively, even though the reads occur earlier in the sequence than the writes.

The partial store ordering model(PSO) is an extension of the TSO models. Figure 2-9 shows the representation for this model. The conceptual system is same as the TSO model. The program order requirement is also similar, the difference is the order of a write followed by a write. The dotted line represent that the order of a write followed by a write should be maintained if and only if both writes access the same location. Moreover, the line with the description *F* represent that the order of a write follow by a write, which do not access to the same location, will be maintained if there is a fence instruction between them. Thus, the PSO model provides a fence instruction for maintaining the order. Programmers have to use such instructions for maintaining the order by themselves.

2.2 Formal Verification

In software development, there is a number of techniques to ensure the correctness of the software, such as *unit testing*. Since the software which is focus in this research is *automotive operating system*. The services of this operating systems will be used to control the behaviors of automotive systems which might relate to our lives. The errors of the systems also be significant issues which we should concern. Hence, to ensure the correctness of *automotive operating systems*, we need *rigorous approaches* to ensure the correctness of the systems. Therefore, *formal verification* is the appropriate approach for this research.

Formal verification is the approach to ensure the systems whether satisfy the given properties or not. Generally, it's used to *prove* or *disprove* the given properties by using *formal methods*. According to [But], "*Formal Methods*" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process).

There are approaches for formal verification. One approach is *model checking*, which we formalize the systems as the *mathematical model* and exploring all states and transitions in the model. The properties to be verified usually given in *temporal logic*. The great advantage of this techniques is fully automatically exploring all states of the models. However, It's doesn't scale for large models, the state explosion will be happened in the exploring process. Hence, there are many approaches to avoid such behaviors, such as providing boundaries for exploring.

Another one is *deductive verification*, which construct the collection of mathematical *proof obligations*, the truth of which imply conformance of the system to its specification. Then, provide proofs for these *proof obligations* to verify the correctness. Generally, we usually use either interactive theorem provers, automatic theorem provers, or satisfiability modulo theories (SMT) solvers. The disadvantage of this technique is that it requires user to understand the systems in detail, and how to convey the information for verification systems.

There are related works that also use *formal verification* to ensure the correctness of *automotive operating systems*. In the related work [Cho13], they use model checking techniques in this work. The formal model is translated from the kernel code of the operating systems. Thus, the *model checking* approach is adopted for this model. Then, the verification of this model is using the SPIN model checker. In this work, the safety properties are considered in the SPIN model checker to check the correctness of the operating system which already formalized as formal model.

There is another related work [KAE+14] that use *theorem provers*, which is used for *deductive verification*, to verify the correctness of it's operating systems. Since it has the advantage that it is not constrained to specific properties or finite, possible state spaces, unlike more automated methods of verification such as *static analysis* or *model checking*.

Chapter 3

Formalization for multi-core systems

This section will propose a formal model to represent multi-core systems. Since the behaviors of multi-core systems is necessary issue in verification, the systems should be formalized to verify the correctness of the programs for multi-core systems. In order to formalize the multi-core systems, the formal model should keep the essences of the multi-core system and sufficient to capture the behaviors of multi-core system. The first section will provide the details of target multi-core system which is focused in this research. The target system consists of components and behaviors which are sufficient for verifying programs on multi-core systems. The next section will propose our formal model based on *operational semantics*. Then, we give a sample execution of programs based on our formal model.

3.1 Target multi-core systems

First of all, to capture the essences of multi-core systems, let's consider the procedures to execute programs on multi-core systems. In the section 3.1.1 will proposed the abstract model of the target systems which represent multi-core systems. The model will capture only the components that sufficient for program verification. To execute programs on multi-core system, normally, each program will be assigned to each core. Then, a processor in the core will *fetch an instruction* from the assigned program. The fetched instruction will be performed by *an execution unit*. The details to performed each instruction will be explained in section 3.1.2.

Then, the execution unit will issue *micro-operations* and *memory accesses* to complete assigned instruction. We refer *micro-operations* and *memory accesses* as *hardware operations* in the target systems for the formal model. There is a variation of hardware operations to complete the instruction that provide semantics by *Instruction set architecture*. However, the target system captures only the operations which influence the execution order issue. The behaviors of hardware operation will be explained in section 3.1.3.

Next, we will consider memory operations which accesses the memory locations either shared-memory or private memory. In the practical systems, there are many components

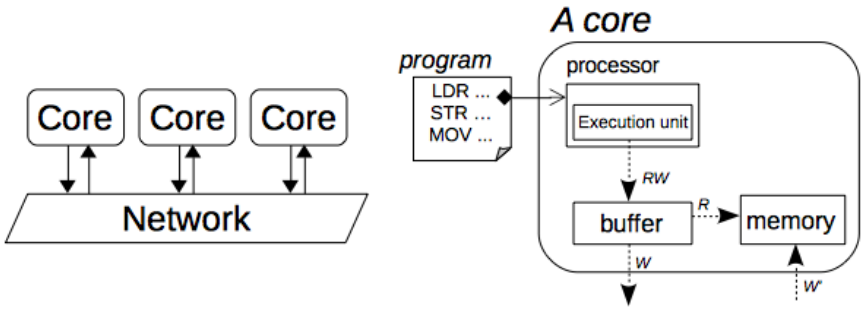

Figure 3-1: Abstract model for multi-core system

to store data, such as cache. In addition, the address locations and arithmetic calculation are used in the low-level architecture. Since the formal model will keep only the essences of the systems, the unnecessary components and behaviors will be eliminated in the target systems.

Since memory consistency models are used for describing the behaviors of shared-memory. As memory operations which access the shared-memory, the order of executions should be controlled by memory consistency models. Thus, the section 3.1.5 will explain idea to adapt the constraints which are proposed by [Gha95].

3.1.1 Abstract model

The figure 3-1 represent the abstract model that represent the multi-core system in the target system. This model consists of a group of *cores*, and *a network* for communicating. As for shared-memory locations, these locations are separated and be attached to each core to capture the *memory consistency* of target system. Inside a core, It consists of a *processor*, *buffers*, reservation stations, an *execution unit*, and *memory location*. In target system, *buffers* is used for keeping the read/write operations. *Memory locations* in the core is used to represent both shared-memory location and private memory locations. The memory locations are also refer to a *cache* which can be appeared in the core. However, we did not model the cache separately to take the *cache coherency protocol* into account because cache coherency can be controlled by *memory consistency* which we focus in this research. As for *reservation station*, we use this component as instruction buffer for the core. The behaviors of this component will be explained in the followed section.

3.1.2 Fetching instruction

In order to execute assigned program in the core, the processor will fetch the instruction indicated by *program counter*, which is *a register* in the core. Once the instruction was fetched, such instruction will be placed into an available execution unit. Note that, in the modern processors, there is a *pipeline* which consists a collection of execution units. When the execution unit access the registers inside a core, sometimes the registers are not updated immediately. Thus, the execution unit will *lock the registers* which is not

updated yet.

In target systems, the processor is allowed to fetch the next instruction immediately. However, such instruction cannot be performed immediately because there is dependence of locked registers. In some cases, processor has *reservation stations*, or instruction buffer, for keeping the instruction to be issue later. In addition, the reservation station allows the out-of-order issuing to be happened. The followed topics will explain more about *reservation stations*.

Moreover, there are some instructions that can update the value of *program counter*. So, such instructions will lock the register that represent *program counter* in the core. In the target system, we will eliminate the some behaviors which predict the next instruction such as *pre-fetch* and *branch-prediction*. Therefore, to fetch next instruction, we will make sure that the program counter is available.

Reservation stations for instructions

Reservation station, or *instruction buffer*, is a buffer to store the instructions which cannot be performed immediately. As the programmer point-of-view, the program should issue the instructions as program order. Nevertheless, to improve the performance of the system, there is mechanism, called out-of-order issue, to issue an instruction from reservation station out-of-order if that instruction is ready to be performed. To consider the ready instructions to be performed, processor have to check whether the required registers of such instructions is available to be accessed.

3.1.3 Hardware operations

Hardware operations refer to the operations issued by execution unit for performing the assigned instruction. In the target system,these operations are either memory access, arithmetic calculation, or fence operation for relaxed memory consistency models. In addition, in the modern processors, the predicated instruction set have been introduced. For example, Instruction set architecture of ARM processor provides instruction with condition can be attached to it such as 'MOV**NE** r1, #1'. The condition of 'MOV**NE** r1, #1' is '**NE**' used to consider whether this condition will be performed or not. Thus, the behaviors of *predicated instruction* also be captured in the target system. In the follow topics will describe some hardware operations in details.

Read/Write operation

The read and write operations refer to memory operations which provide memory access to either private memory, such as register and program status registers, or shared-memory. These operations usually used in the programs to communicate each other among cores. As for read operations, the read will access the memory location to read the value. Then, the read will return the value to the issued processor. To keep the intermediate value of read, target system will store that value into the *internal bus* that will be used in the next operations. The target system allows *non-blocking read access* behaviors, which was

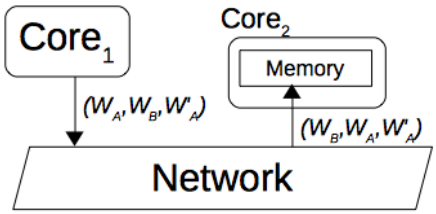

Figure 3-2: Example of out-of-order updates

described in section 2.1.3, to be happened in the system. Hence, the read operations will be putted into the buffer if the hardware uses this optimization mechanism.

As for write operations, the write will get the value to update the memory location in the write operation. The value should be came from the *internal bus*. Once the write operation is issued from the execution unit, the write operation will be putted into the buffer. In the target system, read and write operations are stored in the same buffer. The practical systems can contains many buffers inside a core. However, to capture the behaviors of reordering of memory operations, we have decided to use only single buffer to represent.

The effect of write operation will update the value of memory location. In the case of shared-memory location. The memory location will be appeared in each core separately as copies of shared-memory. In order to keep coherency among cores, the write operations will be distributed into the network. The network will update the shared-memory location for each core. However, the order of write operations is allow to be changed the order, except the writes to the same location for keeping *coherency of memory* among cores.

Let's consider the figure 3-2, the figure shows that write operations W_A, W_B and W'_A are issued from $Core_1$ in that order. Note that the subscriptions A and B locate the shared-memory locations. These write operations will be putted into the *Network* and update the memory in $Core_2$. In some systems, it's possible to update by the order (W_B, W_A, W'_A). In this order, W_B is possible to be executed before W_A. Nevertheless, W'_A is not allowed to be executed before W_A because *coherency properties* should be kept.

Lock/Unlock operation

These operations are used to own the permission for accessing the memory location. In the private memory locations, the *execution unit* will lock the *used registers* for the assigned instruction. This will keep the dependency among instructions even if the instructions are allowed to be issue out-of-order. As for shared-memory locations, the lock/unlock operations will be used to facilitate the read-modify-write operation which will be described in the next topic.

Read-modified-write operation

The *read-modified-write* operation is proposed to capture the atomicity behaviors. This operation will consists of a lock, an unlock , a read, and a write operation. The read

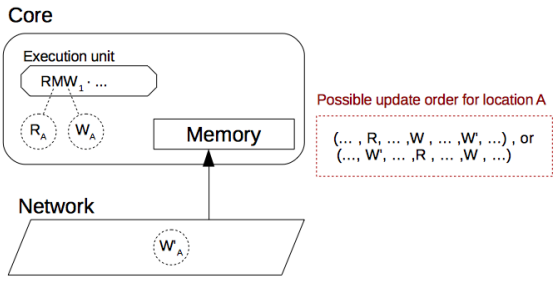

Figure 3-3: Example of read-modified-write requirement

and write operations will access the same shared-memory location. Moreover, the lock and unlock operations will used to ensure that during the execution of read and write operations, there are no another read-modified-write operations can access this location.

In addition, in the target systems, we have to control the behaviors of write operations that access the same location of read-modified-write operation will not interrupt execution of the read-modified-write operation. Let's consider the figure 3-3, RMW_1 refers to read-modified-write operation which consists of a read and a write, R_A and W_A, respectively. Moreover, network also has a write operation W'_A. These read and write operations need to access the memory location A in the core, which represent the copy of shared-memory location. However, the read-modified-write operation requires that during the executions, there is no write operations from another cores can interrupt the executions. Hence, the possible executions of these operations will be either $(..., R_A, ..., W_A, ..., W'_A, ...)$ or $(..., W'_A, ..., R_A, ..., W_A, ...)$. Obviously it's possible to allow another read/write operations to interrupt executions unless they are not write operations which access to the same location.

Fence operation

Since the memory consistency models are provided to allow out-of-order execution, the critical section of the programs should be kept the order to avoid the violation. The fence operation is introduced for that purpose. Normally the fence operation will indicate two groups of operations as *past operations* and *future operations*. In some architecture will treat this operation as read or write operation. To separate the groups of operations, fence operations will be putted in the write buffers or read buffers. To issue the next operations, the fence operation will check that the previous operations already issued from the buffers or cores.

However, the essence of the fence operation is to keep the order of execution to be executed as the program order. Hence, in the formal model, we will just keep the information of program order of fence operation to enforce the execution in the future.

3.1.4 Target memory locations

The target multi-core system consists of a collection of cores and a network that connect each core together. The memory locations appeared to each core consist of *private memory locations* and *shared-memory locations*. These locations are accessed by *read/write operations* which are issued from the *buffer* or *network*.

In the practical systems, the locations of memory are described as the *binary address* and the index of registers. In order to abstract the system, the target system define address of memory location as a *variable*. This means that the address of location cannot be considered as consequent address. Hence, some optimization mechanisms are not able to be taken into account, such as merging write. As for the value to be kept inside the memory location, the target system will consider the value as a *natural number* instead of binary bits. Due to we need to deal with the execution order as a major topic, we will keep the calculation inside the model to be simply.

3.1.5 Memory consistency models

For now the behaviors of read and write operations are able to be executed out-of-order by hardware optimization mechanisms. Although the processor is able to keep the correct result of programs by itself, another processors should be able to keep the correct result too. Since the *memory consistency models* are introduced to describe the behaviors of memory, we will adopt the constraints from [Gha95] for enforcing the possible execution to produce the valid executions based on its memory consistency model.

First of all, we would like to introduce some of framework that propose by [Gha95] in order to specify the system requirements. The system requirement is used to identify the valid systems for the memory consistency model. The specification of system requirements directly defines the *ordering constraints* for a given model. This framework is appropriate to deal with the memory consistency models which are defined differently in each system. Hence, we adopt the idea of [Gha95] to realize the behaviors based on a given model. Nevertheless, we cannot use those specification directly, because the framework captures the order constraints to be happened based on the given model. In contrast, our formal model captures the execution behaviors that should be appeared in the verification. However, we will briefly describe this framework for identifying the specification of system requirement.

Read and write operations

The figure 3-4 show the conceptual model for memory of their abstraction. In this model, the system consists of n processors, $P_1, P_2, ..., P_n$. The processor will issue the operations and put it into the *buffer*. Each node of processors has its own memory, M_i which belong to P_i. The memory, M_i, is a complete copy of shared-memory in the system. the each node will be connected by network. As processors use memory operations to access the memory and the presence of the copies memory, they introduce the notion of sub-operations for each memory operation. A read operation R by P_i is comprised of two atomic sub-

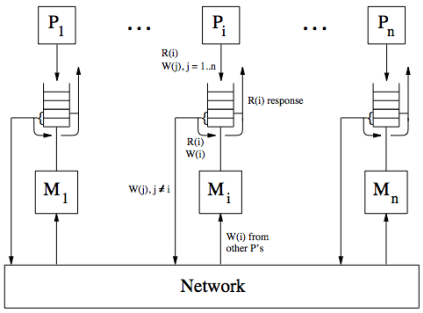

Figure 3-4: General model for shared-memory

operations: an initial sub-operation $R_{init}(i)$ and a single read sub-operation$R(i)$. A write operation W by P_i is comprised of at most $n+1$ atomic sub-operations: an initial write sub-operation $W_{init}(i)$ and at most n sub-operations $W(1), W(2), ..., W(n)$. A read operation on P_i results in the read sub-operation $R(i)$ being placed into P_i's memory buffer. Similarly, a write operation on P_i results in write sub-operations $W(1), ..., W(n)$ being placed in its processor's memory buffer. The initial sub-operations $R_{init}(i)$ and $W_{init}(i)$ are mainly used to capture the program order among conflicting operations. Conceptually, $R_{init}(i)$ for a read corresponds to the initial event of placing $R(i)$ into a memory buffer and $W_{init}(i)$ for a write corresponds to the initial event of placing $W(1), ..., W(n)$ into the memory buffer.

Once the sub-operations are placed in the processor's buffer, these operations will be issued by system. However, it's not necessary to issue the operations as first-in-first-out order. As for $W(j)$ issued by P_i will be placed into the network for updating the memory which belong to processor P_j. Similarly, the operation $R(i)$ issued by P_i will be used to read the memory of processor P_i. Since the $R(i)$ and $W(i)$ are allowed to change the execution order, $R_{init}(i)$ and $W_{init}(i)$ are provided to capture the program order of these memory operations. Obviously, $R_{init}(i)$ and $W_{init}(i)$ are not appeared explicitly.

As for target system, we do not split the memory operation into sub-operations. In order to capture the program order, we keep such information in the semantic configuration in formal model. Moreover, to update the memory location among cores, our formal model will distribute the write operation to the network as many operations Hence, the update of memory location will be came from network.

Program order and execution order

The definitions 3.1 and 3.2, which are defined in [Gha95], describe the program order in detail. However, this section will briefly explain the program order and execution order. As the conceptual model describe in figure 3-4, the program order and execution order are described to capture the behaviors of the systems. A *program order* is a partial order (denoted by $\xrightarrow{po}$) on the *instruction instances* and on the *memory operations*. $R \xrightarrow{po} RW$ means the read operation R is followed by RW in *program order*. A *execution order* is a total order (denoted by $\xrightarrow{xo}$) on the *sub-memory operations*.

In addition, there is *conflicting order* denoted by $\xrightarrow{co}$. For execution order $\xrightarrow{xo}$ and two conflicting memory operation X and Y, $X \xrightarrow{co} Y \text{iff} X(i) \xrightarrow{xo} Y(i)$ holds for any i. If X and Y are on different processors and $X \xrightarrow{co} Y$,then X and Y are also ordered by the *interprocessor conflict order* $X \xrightarrow{co'} Y$. Note that neither $\xrightarrow{co}$ nor $\xrightarrow{co'}$ are partial orders; for example, $R \xrightarrow{co} W$ and $W \xrightarrow{co} R'$ do not imply $R \xrightarrow{co} R'$.

These order will be used to constrain the execution of the memory consistency model. For example, to constrain the behavior of coherency on multi-core system, for any W and W' that $W \xrightarrow{co'} W'$, the valid executions should be the execution that $W(i) \xrightarrow{xo} W'(i)$ for all i. So, these orders are used to enforce the executions of the system. However, in some systems may require more order relations to specify their behaviors.

Definition 3.1: Program Order among Instruction Instances

(Note that, this definition is defined in [Gha95] as definition(4.3))
The *program order*, denoted by $\xrightarrow{po}$, is a partial order on the dynamic instruction instances in a run of the program. The program order is a total order among instruction instances from the same process, reflecting the order specified by the next instruction relation described in Definition 4.2. Dynamic instruction instances from different processes are not comparable by program order.

Definition 3.2: Program Order among Memory Operations

(Note that, this definition is defined in [Gha95] as definition(4.4))
Two memory operations, $o1$ and $o2$, are ordered by program order ($o1 \xrightarrow{po} o2$) if either (a) their corresponding instruction instances, $i1$ and $i2$, respectively, are ordered by $\xrightarrow{po}$ ($i1 \xrightarrow{po} i2$) as specified by Definition 4.3, or (b) they both belong to the same instruction instance and $o1$ is partially ordered before $o2$ by the instruction instance (as specified by Definition 4.2(ii)). In contrast to program order among instruction instances, the program order among memory operations from the same process is not necessarily a total order (because an instruction instance does not necessarily impose a total order among its own memory operations).

Conditions for system requirements

To consider the valid systems, there are some conditions to make sure that the system will work properly. The first two conditions are *Initial condition for reads and writes*(3.1), and *Termination condition for writes*(3.2).These conditions are the initial condition which system should hold in the implemented systems. The next two conditions are *Return value for Read sub-operations*(3.3) and *Atomicity of read-modified-write operations*(3.4). These conditions are provided to maintain the correct results which produce from the systems.

Condition 3.1: Initiation Condition for Reads and Writes

(Note that, this condition is defined in [Gha95] as condition(4.4))
Given memory operations by P to the same location, the following conditions hold: If $R \xrightarrow{po} W$, then $R_{init}(i) \xrightarrow{xo} W_{init}(i)$. If $W \xrightarrow{po} R$, then $W \xrightarrow{po} (i) \xrightarrow{xo} R_{init}(i)$. If $W \xrightarrow{po} W$, then $W_{init}(i) \xrightarrow{xo} W_{init}(i)$.

Condition 3.2: Termination Condition for Writes

(Note that, this condition is defined in [Gha95] as condition(4.5))
Suppose a write sub-operation $W_{init}(i)$ (corresponding to operation W) by P_i appears in the execution. The termination condition requires the other n corresponding sub-operations, $W(1), ..., W(n)$, to also appear in the execution. A memory model may restrict this condition to a subset of all write sub-operations.

Condition 3.3: Return Value for Read Sub-Operations

(Note that, this condition is defined in [Gha95] as condition(4.6))
A read sub-operation $R(i)$ by P_i returns a value that satisfies the following conditions. If there is a write operation W by P_i to the same location as $R(i)$ such that $W_{init}(i) \xrightarrow{xo} R_{init}(i)$ and $R(i) \xrightarrow{xo} W(i)$, then $R(i)$ returns the value of the last such $W_{init}(i)$ in $\xrightarrow{xo}$. Otherwise, $R(i)$ returns the value of $W(i)$ (from any processor) such that $W(i)$ is the last write sub-operation to the same location that is ordered before $R(i)$ by $\xrightarrow{xo}$. If there are no writes that satisfy either of the above two categories, then $R(i)$ returns the initial value of the location.

Condition 3.4: Atomicity of Read-Modify-Write Operations

(Note that, this condition is defined in [Gha95] as condition(4.7))
If R and W are the constituent read and write operations for an atomic read-modify-write ($R \xrightarrow{po} W$ by definition) on P_j , then for every conflicting write operation W from a different processor P_k , *either* $W(i) \xrightarrow{xo} R(i)$ and $W(i) \xrightarrow{xo} W(i)$ for all i *or* $R(i) \xrightarrow{xo} W(i)$ and $W(i) \xrightarrow{xo} W(i)$ for all i.

Aggressive specification for system requirements

Since to describe the requirements for memory consistency model we can strict all of executions to maintain the results. However, that means the optimization mechanisms also have been restricted the behaviors and cannot improve the performance of the system. Thus, [Gha95] proposed the *aggressive specifications* of memory consistency models which fully exploits the features of their general abstraction to place fewer restrictions on the execution order. The specification 1 show the aggressive condition for sequential consistency model which proposed by that work.

Specification 1: Aggressive conditions for sequential consistency

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X and Y are to *different* locations and X $\xrightarrow{spo}$ Y
define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{co'}$ Y
R $\xrightarrow{co'}$ W $\xrightarrow{co'}$ R
Condition on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; *applies to all write sub-operations.*
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: $RW \xrightarrow{po} W$
coherence: $W \xrightarrow{co'} W$
multiprocessor dependence chain: one of
$W \xrightarrow{co'} R \xrightarrow{po} RW$
$RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$
$W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$

3.2 Formal model

In order to describe the behaviors of hardware based on multi-core systems, we choose *operational semantics* to represent the multi-core system. In this section will describe the intuition idea to propose the semantics. Then, the specification of the memory consistency model will be adopt to the semantics by providing conditions for enforcing the semantics.

3.2.1 Preliminaries

Before we introduce the formal model, we would like to define some preliminaries that will be used to define semantics.

- The set of **lists** of a set S is denoted as S^*,
  we use ε for an empty list.
- An **ordered set**, or **partial ordered sets**, is a pair $(S, \prec)$,
  where $\prec$ is a *binary relation* on S such that
  - If $(a \prec b)$ and $(b \prec c)$ then $a \prec c$ (transitivity)
  - Let $(X, \prec_X)$ and $(X, \prec_Y)$ be ordered sets.

$$(X, \prec_X) \oplus (Y, \prec_Y) = (X \cup Y, \prec_X \cup \prec_Y \cup (X \times Y))$$

- The set of **ordered sets** of a set S is denoted as Order$\langle S \rangle$, such that
  Order$\langle S \rangle = \{(S, \prec) \mid \ \prec \subseteq S \times S\}$

- **Substitution** of a function $f : X \to Y$ is defined as

$$(f[x \mapsto y])(z) = \begin{cases} y & \text{If x = z} \\ f(z) & \text{Otherwise} \end{cases}$$

  Where $x, z \in X$ and $y \in Y$.

3.2.2 Syntax

First of all, we need to identify each core in the multi-core systems. Since each core issues the memory accesses to the network, we provide *identifier* to track the source core that issues the memory operation. Thus, we provide he definition of the set of *identifier* in 3.3 to keep the collection of identifiers in the system.

Definition 3.3:
The set of *identifiers* is denoted by ID.
The set of *memory locations* is denoted by Location.
the set of *values* that are used in the systems is denoted by Val.

In addition, we also define the set of*memory locations*, Location, which is the collection of locations which appeared to one processor. The *memory locations* are known by the processor's point-of-view. In the memory location, we separate the location into *private memory*, *internal buses* and *shared-memory*, which are defined in the following definitions.

Definition 3.4: Private memory locations
The set of *private memory locations*,denoted by Local, is the subset of Location. An element of this set may be the member of either:
(a) The set of *registers*, denoted by Reg, or
(b) The set of *application status registers*, denoted by APSR.
Note that, the definitions of *registers set* and *program status register set* are defined as $\{\mathrm{R}_1, \mathrm{R}_2, ..., \mathrm{R}_{14}, \mathrm{pc}\}$ and $\{\mathrm{Z}, \mathrm{N}, \mathrm{C}, \mathrm{V}\}$, respectively. By the definitions, these sets have the relations as:

$$\begin{aligned} \text{Local} &= \text{Reg} \cup \text{APSR}, \\ \text{Reg} &\subseteq \text{Location}, \text{ and} \\ \text{APSR} &\subseteq \text{Location} \end{aligned}$$

Definition 3.5: Internal buses
The set of *internal buses* is denoted by internalBus, where internalBus $\subseteq$ Location. An *internal bus* is an intermediate storage to store the value, such as data or address, which flows in a core.

Definition 3.6: Shared memory locations
The set of *shared-memory location* is denoted by Shared, where Shared = Location $\setminus$ (Local$\cup$internalBus). A *shared-memory location* is denoted by Mem[v], where $v \in$ Val.

Next, we would like to introduce the grammar of *expressions* that capture the *arithmetic calculation* inside a processor. The expression is defined as the definitions 3.7. In addition, since the target system supports the *predicated instruction*, we need to introduce the grammar of *conditional expressions* to supports the predicated instruction inside the target system. Hence, the definition is defined by definition 3.8.

Definition 3.7: Expressions

The grammar of an expression, denoted by Expr, is

$$\begin{aligned}\text{Expr} ::= &\text{ Val} \mid \text{internalBus} \mid (\text{Expr} + \text{Expr}) \mid (\text{Expr} - \text{Expr}) \mid \\ &(\langle \text{CondExpr} \rangle ? \text{Expr} : \text{Expr})\end{aligned}$$

Definition 3.8: Conditional expression

The grammar of a *conditional expression*, denoted by CondExpr, is

$$\begin{aligned}\text{CondExpr} ::= &\text{ True} \mid \text{False} \mid (\text{Expr} = \text{Expr}) \mid (\text{Expr} \mathrel{!}= \text{Expr}) \mid \\ &(\text{Expr} > \text{Expr}) \mid (\text{Expr} < \text{Expr}) \mid (\text{CondExpr} \wedge \text{CondExpr}) \mid \\ &(\text{CondExpr} \vee \text{CondExpr})\end{aligned}$$

The *hardware operations* are the operations that are issued by processors. In the general context, these operations are *micro-operations* and *memory accesses* which are generated to complete the fetched instruction inside execution units. Thus, the definitions of hardware operations are provided as follow.

Definition 3.9: Hardware operations

The set of *hardware operations* is denoted by Ops. A hardware operation is an operation issued by *processor*. These operations are provided to complete the fetched instructions. A hardware operation can be either:

(a) *memory operation*,
(b) *lock or unlock operation*,
(c) *assignment operation*, or
(d) *condition operation*.

Definition 3.10: Memory Operations

The set of *memory operations* is denoted by MemOp,where MemOp $\subseteq$ Ops. The memory operations are used to operate with the memory locations by the processor. These operations can be either:

(a) read/write operations,
(b) read-modified-write operations, or
(c) fence operations.

The sets of these operations will be denoted by RW, RMW and FenceOp, respectively. Moreover, these sets are the subset of the *memory operations*, $\text{RW}, \text{RMW}, \text{FenceOp} \subseteq \text{MemOp}$.

Definition 3.11: Read/Write operations

The set of *read and write operations* is denoted by RW. This set is consists of two sets,

- Read operation set, denoted by R, and
- Write operation set, denoted by W.

Where RW = R ∪ W and RW ⊆ MemOp.

Note that, in order to access the *memory location*, the read and write operations have attributes for accessing the locations. The attributes of read and write operations are extracted by following functions:

access :RW → Location	Access **location**
data :RW → Expr	**updated value** or **stored register**

Definition 3.12: Read-modified-write Operations

The set of *read-modified-write operations* is denoted by RMW, where RMW ⊆ MemOp. This operation is a composite operation which consist of lock operation, unlock operation, read operation, and write operation.

The following functions are used to extract the necessary information to produce the *hardware operations* in semantics.

access :RW → Location	Access **location**
data :RW → Expr	**updated value** or **stored register**

Definition 3.13: Fence Operations

The set of *fence operations* is denoted by FenceOp, where FenceOp ⊆ MemOp.

This fence operation will be used in each memory consistency model as different context. Thus, we define *type* of the FenceOp as

$$\begin{aligned}\text{fenceType} : \text{FenceOp} \rightarrow & \{\text{STBAR}, \text{MB}, \text{SYNC}, \text{WMB}\} \cup \\ & \{\text{MEMBAR(type)} \mid \text{type} \in \{\text{RR}, \text{RW}, \text{WR}, \text{WW}\}\}\end{aligned}$$

Definition 3.14: Assignment Operations

The set of *assignment operations* is denoted by AssignOp, where AssignOp ⊆ Ops.

Definition 3.15: Condition Operations

The set of *condition operations* is denoted by CondOp, where CondOp ⊆ Ops.

These assignment operations are used for calculation value of Expr. The results of calculation will be store in the internalBus. The grammar of these operations are

$$\text{AssignOp} = \{data := cval \mid (data \in \text{internalBus}) \text{ and } (cval \in \text{Expr})\}$$

Condition operations are provided to support predicated instructions. This operation will have the condition expression, CondExpr, to decide that which operation should be executed in the next step. The grammar of this operation is

$$\text{CondOp} = \{(\langle cond \rangle path_1 : path_2) \mid (cond \in \text{CondExpr}) \text{ and } (path_1, path_2 \in \text{Ops}^*)\}$$

In addition, before we introduced the lock/unlock operations, we would like to define the definition of *lock entities* as the follow definition.

Definition 3.16: Lock entities
The set of **lock entities** is denoted by Lock. These entities will be used for checking registers dependency mechanisms and supporting atomicity of read-modified-write operations.

Definition 3.17: Lock/Unlock Operations

The sets of *lock operations* and *unlock operations* are denoted by LockOp and UnlockOp, respectively, where $\text{LockOp}, \text{UnlockOp} \subseteq \text{Ops}$.

We define the functions to extract the information from lock/unlock operations as,

$$\text{lockEntity} : \text{ID} \times (\text{LockOp} \cup \text{UnlockOp}) \rightarrow \text{Lock}$$
$$\text{lockLocation} : \text{LockOp} \cup \text{UnlockOp} \rightarrow \text{Location}$$

Since the lock/unlock operation have two behaviors to be captured:
(1) Checking dependency of required registers of instructions, and
(2) Facilitate the atomicity of read-modified-write operations.
Therefore, we provide the restriction for the *lock entities* that the lock entities of shared-memory location will be same among cores. In contrast to lock entities of local memory location, the lock entities should be different among cores even if they lock the same location. We provide the restriction as followed.
Restriction: *For make sure that only shared location will lock the same lock*

$$\forall lock_1, lock_2 \in \text{LockOp} \cup \text{UnlockOp}. \forall i, j \in \text{ID}.$$
$$((i \neq j) \wedge (\text{lockLocation}(lock_1) = \text{lockLocation}(lock_2))$$
$$\implies ((\text{lockLocation}(lock_1) \in \text{Shared}) \iff (\text{lockEntity}(i, lock_1) = \text{lockEntity}(j, lock_2))))$$

In the verification, we also need the information to indicate the instances which are produced from programs. In addition, we also need to identify the owner of *hardware operations* in the verification for considering significant operations. Thus, we introduce the *timestamps* to capture the specific behaviors. The definition is below,

Definition 3.18: Timestamps
The set of *timestamps* is denoted by TS.

We define the function $\text{uniqueTime} : 2^{\text{TS}} \rightarrow \text{TS}$ for generating a fresh timestamp. In addition, the function $\text{timePC} : \text{TS} \mapsto \text{Val}$ is used to capture the value of *program counter* which the timestamps is generated. We also define annotations as below

Definition 3.19: Annotations for timestamps
For any $o \in \text{Ops}$ and $t \in \text{TS}$,
o^t is an operation that generated at timestamp t
Ops^{TS} is the set $\{o^t \mid o \in Ops \text{ and } t \in \text{TS}\}$.
MemOp^{TS} is the set $\{o^t \mid o \in \text{MemOp} \text{ and } t \in \text{TS}\}$.
RW^{TS} is the set $\{rw^t \mid rw \in \text{RW} \text{ and } t \in \text{TS}\}$.

Then, we will introduce the Instruction which will be used as interface to communicate the assembly programs and hardware operations.

Definition 3.20: Instructions
InstrCond is the set of *conditions* for instructions. InstrName is the set of *names* for instructions. Operand is the set of *operands*,where Operand $=$ Val $\cup$ Reg $\cup$ $\{\text{Mem}(val) \mid val \in \text{Expr}\}$. An *instruction* is a tuple $\langle name, cond, args\rangle$ of

$name \in$ InstrName	The name of the instruction
$cond \in$ InstrCond	Condition for performing the instruction, and
$args \in$ Operand*	Operands for the instruction.

The set of *instructions* is denoted by Instr.

For the instruction, we define a function ISA : Instr $\rightarrow$ Ops* to translate the *instruction* to be a *sequence of hardware operations*. Note that, if $ops = o_1 \cdot o_2 \cdot ... \cdot o_n$ for any $ops \in$ Ops* and $t \in$ TS, then we provide the annotation of the sequence *ops* as

$$ops^t = o_1^t \cdot o_2^t \cdot ... \cdot o_n^t$$

A *core* is the component that is consisted in the *multi-core system*, which is defined in the definition 3.21.

Definition 3.21: Core component
A **core** is a tuple $\langle id, \sigma, B, L, exec, P\rangle$ of

$id \in$ ID	Identifier
σ : Location $\rightarrow$ Val	Memory location of processor *id*
$B \subseteq$ RW	buffer for **operations**
$L \subseteq$ TS $\times$ Ops*	buffer for **instances**
$exec$: (Ops^{TS})*	Execution unit, and
P : Val $\rightarrow$ Instr	Program.

Then, as we consider the optimization mechanisms, we try to give a hardware configuration to consider the behaviors in our semantics.

Definition 3.22: Hardware Configuration
HW is the set of **hardware configurations**.

The following functions are describe the hardware configuration whether use that opti-

mization mechanism or not.

$$\begin{aligned} \text{inOrderIssue} &: \text{HW} \to \text{Boolean} \\ \text{nonBlocking} &: \text{HW} \to \text{Boolean} \\ \text{readForwarding} &: \text{HW} \to \text{Boolean} \\ \text{bypassing} &: \text{HW} \to \text{Boolean} \end{aligned}$$

As we also take the *memory consistency model*'s behaviors into account. We should also define the definition of memory consistency model in the target system.

Definition 3.23: Memory consistency model
The set of *memory consistency models* is denoted by MemModel.

We provide the *uniprocessor condition* and *network condition* to enforce the execution for the memory consistency model. The following functions are used to extract the information of the memory consistency model to our system.

$$\begin{aligned} \text{model} &: \text{HW} \to \text{MemModel} \\ \text{networkCond} &: \text{MemModel} \times \text{Order}\langle \text{ID} \times \text{MemOp} \rangle \times \text{Order}\langle \text{ID} \times \text{ID} \times \text{RW} \rangle \\ &\quad \times \text{ID} \times \text{ID} \times \text{RW} \to \text{Boolean} \\ \text{uniprocessorCond} &: \text{MemModel} \times \text{Order}\langle \text{ID} \times \text{MemOp} \rangle \times \text{Order}\langle \text{ID} \times \text{ID} \times \text{RW} \rangle \\ &\quad \times \text{ID} \times \text{RW} \to \text{Boolean} \end{aligned}$$

Configuration
In order to provide the semantics, we provide the *configuration* to represent the *state* as a tuple $\langle hw, \langle to, po, xo \rangle, locked, N, C \rangle$ of

$$\begin{aligned} C &: \text{ A set of cores,} \\ hw &\in \text{HW} && \text{A configuration of hardware,} \\ locked &\subseteq \text{Lock} && \text{A set of current lock,} \\ N &\subseteq \text{ID} \times \text{ID} \times \text{W} && \text{A network component,} \\ to &\in \text{Order}\langle \text{TS} \rangle && \text{A timestamp order,} \\ po &\in \text{Order}\langle \text{ID} \times \text{MemOp}^{\text{TS}} \rangle && \text{A program order, and} \\ xo &\in \text{Order}\langle \text{ID} \times \text{ID} \times \text{RW}^{\text{TS}} \rangle && \text{An execution order.} \end{aligned}$$

Restrictions:

$$\begin{aligned} \forall c_1, c_2 \in C.(\ & (c_1 = \langle id_1, \sigma_1, B_1, L_1, exec_1, p_1 \rangle) \wedge \\ & (c_2 = \langle id_2, \sigma_2, B_2, L_2, exec_2, p_2 \rangle) \wedge \\ & (c_1 \neq c_2) \implies (id_1 \neq id_2)) \end{aligned}$$

3.2.3 Semantics

First of all, we would like to provide the *evaluation context* for describing the configuration. The context for this formal model is defined as followed,

$$\begin{aligned} E ::= {} & [] \mid E \cdot \mathrm{Ops}^* \mid data := E \mid (\langle E \rangle \mathrm{Ops}^* : \mathrm{Ops}^*) \mid \\ & (E + \mathrm{Expr}) \mid (\mathrm{Expr} + E) \mid (E - \mathrm{Expr}) \mid (\mathrm{Expr} - E) \mid \\ & (\langle E \rangle ? \mathrm{Expr} : \mathrm{Expr}) \end{aligned}$$

Where $\diamond \in \{=,\ != , >, <\}$ and $\circ \in \{\vee, \wedge\}$
Then, the semantics will consider the behaviors of the multi-core systems as the following topics.

Fetching instruction

The first topic we consider to fetch an instruction from the program. For the fetching instruction, out-of-order issuing can be apply to the system to improve the performance of the systems. Before proposed the semantics, we would like to define some predicates as condition for each rule. The predicates are:

$$\begin{aligned} \mathrm{notUsePC}(o_1 \cdot o_2 \cdot ... \cdot o_n) \iff {} & \forall i \in \mathbb{N}, pid \in \mathrm{ID}.(\\ & (o_i \in \mathrm{UnlockOp} \implies \\ & \qquad \mathrm{lockLocation}(o_i) \neq (pid, \mathrm{pc})) \wedge \\ & (o_i = (\langle cond \rangle ops_1 : ops_2) \implies \\ & \qquad \mathrm{notUsePC}(ops_1) \wedge \mathrm{notUsePC}(ops_2))) \\ \mathrm{requireRegs}(\langle name, cond, a_1 \cdot ... \cdot a_n \rangle) = {} & \{a_i \mid i \in \mathbb{N} \wedge a_i \notin \mathrm{Local} \wedge \\ & \exists expr \in \mathrm{Expr}.((a_i = \mathrm{Mem}[expr]) \\ & \qquad \implies (expr \in \mathrm{Reg}))\} \\ \mathrm{regsAvailable}(pid, regs, o_1 \cdot o_2 \cdot ... \cdot o_n) \iff {} & \forall i \in \mathbb{N}, pid \in \mathrm{ID}, reg \in \mathrm{Local}. \\ & ((o_i \in \mathrm{UnlockOp} \wedge (\mathrm{lockLocation}(o_i) = reg) \\ & \implies (reg \notin regs)) \wedge \\ & ((o_i = (\langle cond \rangle ops_1 : ops_2) \implies \\ & \qquad (\mathrm{regsAvailable}(pid, regs, ops_1) \wedge \\ & \qquad \mathrm{regsAvailable}(pid, regs, ops_2))))) \end{aligned}$$

- **Rule 1.** In-order issuing

$$\langle hw, \langle (S_{ts}, \prec_{ts}), po, xo \rangle, locked, N, \{\langle pid, \sigma, B, L, exec, p \rangle\} \uplus C \rangle \rightarrow$$
$$\langle hw, \langle (S_{ts}, \prec_{ts}) \oplus (\{t'\}, \emptyset), po', xo \rangle, locked, N, \{\langle pid, \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1], B, L, exec \cdot ops^t, p \rangle\} \uplus C \rangle$$

If $\text{inOrderIssue}(hw) \wedge \text{notUsePC}(exec) \wedge \text{regsAvailable}(pid, \text{requireRegs}(p(\sigma(\text{pc}))), exec)$
Where

$$\begin{aligned}
\text{uniqueTime}(S_{ts}) &= t \\
\text{timePC}(t) &= \sigma(\text{pc}) \\
\text{ISA}(p(\sigma(\text{pc}))) &= ops \\
ops^t &= o_1^t \cdot o_2^t \cdot ... \cdot o_n^t \\
S_{ops} &= \{(pid, o_i^t) \mid (i \in \mathbb{N}) \text{ and } (o_i^t \in \text{MemOp}^{TS})\} \\
(id, o_i^t) \prec_{ops} (id', o_j^t) &\iff (id = id') \text{ and } (id = pid) \text{ and } (i, j \in \mathbb{N}) \text{ and} \\
&\quad\ (i < j) \text{ and } (o_i^t, o_j^t \in \text{MemOp}^{TS}) \\
po' &= po \oplus (S_{ops}, \prec_{ops})
\end{aligned}$$

- Out-of-order issuing
  - **Rule 2.** Putting

    $\langle hw, \langle (S_{ts}, \prec_{ts}), po, xo \rangle, locked, N, \{\langle pid, \sigma, B, L, exec, p \rangle\} \uplus C \rangle \rightarrow$
    $\langle hw, \langle to', po', xo \rangle, locked, N, \{\langle pid, \sigma[\text{pc} \mapsto \sigma(\text{pc}) + 1], B, L \cup \{(t', ops)\}, exec, p \rangle\} \uplus C \rangle$

    If $\neg\text{inOrderIssue}(hw) \wedge \text{notUsePC}(exec) \wedge \forall (t, ops) \in L.(\text{notUsePC}(ops)) \wedge$
    $\forall w \in B.(w \in \text{W} \implies \text{access}(w) \neq \text{pc})$
    Where

$$\begin{aligned}
\text{uniqueTime}(S_{ts}) &= t \\
\text{timePC}(t) &= \sigma(\text{pc}) \\
\text{ISA}(p(\sigma(\text{pc}))) &= ops \\
ops^t &= o_1^t \cdot o_2^t \cdot ... \cdot o_n^t \\
to' &= (S_{ts}, \prec_{ts}) \oplus (\{t\}, \emptyset) \\
S_{ops} &= \{(pid, o_i^t) \mid (i \in \mathbb{N}) \text{ and } (o_i^t \in \text{MemOp})\} \\
(id, o_i^t) \prec_{ops} (id', o_j^t) &\iff (id = id') \text{ and } (id = pid) \text{ and } (i, j \in \mathbb{N}) \text{ and} \\
&\quad\ (i < j) \text{ and } (o_i^t, o_j^t \in \text{MemOp}) \\
po' &= po \oplus (S_{ops}, \prec_{ops})
\end{aligned}$$

  - **Rule 3.** Issuing

    $\langle hw, \langle (S_{to}, \prec_{to}), po, xo \rangle, locked, N, \{\langle pid, \sigma, B, L \uplus \{(t, ops)\}, exec, p \rangle\} \uplus C \rangle \rightarrow$
    $\langle hw, \langle (S_{to}, \prec_{to}), po, xo \rangle, locked, N, \{\langle pid, \sigma, B, L, exec \cdot ops^t, p \rangle\} \uplus C \rangle$

    If $\neg\text{inOrderIssue}(hw) \wedge \text{execDependCond}(ops, exec) \wedge$
    $\forall (t', ops') \in L.(t' \prec_{to} t \implies \text{dependCond}(ops, ops'))$

Where

$$\begin{aligned}
&\text{execDependCond}(o_1 \cdot o_2 \cdot ... \cdot o_n, \\
&\qquad o'_1 \cdot o'_2 \cdot ... \cdot o'_n) \iff \forall i, j \in \mathbb{N}, o_i \in \text{LockOp}. \\
&\qquad (((o'_j \in \text{UnlockOp}) \implies \\
&\qquad (\text{lockLocation}(o_i) \neq \text{lockLocation}(o'_j)))\vee \\
&\qquad ((o'_j = (\langle cond \rangle ops_1 : ops_2)) \implies \\
&\qquad (\text{execDependCond}(o_i, ops_1)\wedge \\
&\qquad \text{execDependCond}(o_i, ops_2))))
\end{aligned}$$

$$\begin{aligned}
&\text{dependCond}(o_1 \cdot o_2 \cdot ... \cdot o_n, \\
&\qquad o'_1 \cdot o'_2 \cdot ... \cdot o'_n) \iff \forall i, j \in \mathbb{N}, o_i \in \text{LockOp}. \\
&\qquad (((o'_j \in \text{LockOp}) \implies \\
&\qquad (\text{lockLocation}(o_i) \neq \text{lockLocation}(o'_j)))\vee \\
&\qquad ((o'_j = (\langle cond \rangle ops_1 : ops_2)) \implies \\
&\qquad (\text{dependCond}(o_i, ops_1) \wedge \text{dependCond}(o_i, ops_2))))
\end{aligned}$$

Perform an operation

As the execution unit has a list of operations to be compute, this topics will provide the semantics to deal with the issued operations from the execution unit as follow,

- Write operation
  Before provide semantics for write operations, we would like to propose the conditions to facilitate the atomicity condition, which define in condition 3.4, as:

$$\begin{aligned}
\text{isAtomic}(pid, (S_{po}, \prec_{po}), r, w) &\iff (r \in \text{R}) \wedge (w \in \text{W}) \wedge (\text{access}(r) = \text{access}(w)) \\
&\quad \wedge ((pid, r) \prec_{po} (pid, w)) \\
\text{atomicCond}(i, po, (S_{xo}, \prec_{xo}), w_j) &\iff \forall w_i \in \text{W}, r_i \in \text{R}.((w_j \in \text{W}) \\
&\quad \wedge \text{isAtomic}(i, po, r_i, w_i)\wedge \\
&\quad (\text{access}(r_i) = \text{access}(w_j)) \implies \\
&\quad (((i, r_i) \in S_{xo}) \iff ((i, w_i) \in S_{xo}))
\end{aligned}$$

  - **Rule 4.** Putting to buffer

$$\begin{aligned}
&\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[w], p\rangle\} \uplus C\rangle \rightarrow \\
&\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B \cup \{w\}, L, E[\varepsilon], p\rangle\} \uplus C\rangle
\end{aligned}$$

    If $(w \in \text{W}) \wedge (\text{data}(w) \in \text{Val})$

– **Rule 5.** Issuing a write operation

$$\langle hw, \langle to, po, xo\rangle, locked, N, \langle pid, \sigma, B \uplus \{w\}, L, exec, p\rangle\rangle \rightarrow$$
$$\langle hw, \langle to, po, xo\rangle, locked, N', \langle pid, \sigma', B, L, exec, p\rangle\rangle$$

If uniprocessorCond(model(hw), po, xo, pid, w)
Where

$$\text{performWrite}(xo, N, \sigma, w) = (xo', N', \sigma')$$

The performWrite function is defined as:

$$\text{performWrite}((S_{xo}, \prec_{xo}), N, \sigma, w) =$$
$$\begin{cases} (xo \oplus (\{(pid, pid, w)\}, \emptyset), N, \sigma[\text{access}(w) \mapsto \text{data}(w)]) & \text{if access}(w) \in \text{Local} \\ ((to, po, xo), N \cup \{(i, pid, w) \mid i \in \text{ID}\}, \sigma) & \text{if access}(w) \notin \text{Local} \end{cases}$$

– **Rule 6.** Update from network

$$\langle hw, \langle to, po, xo\rangle, locked, N \cup \{(i, j, w_j)\}, \{\langle i, \sigma, B, L, exec, p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \langle to, po, xo \oplus (\{(i, j, w_j)\}, \emptyset)\rangle, locked, N, \{\langle i, \sigma[\text{access}(w_j) \mapsto \text{data}(w_j)], B, L, exec, p\rangle\} \uplus C\rangle$$

If networkCond(model(hw), Φ, i, j, w_j) $\wedge$ $((i \neq j) \implies \text{atomicCond}(i, po, xo, w_j))$

- Read operation
  Before introduce the semantics for read operations, we would like to define predicates for optimization mechanisms as:

$$\begin{aligned} \text{readForwardingCond}(pid, po, B, r) \iff & \forall w' \in B.((w' \in \text{R}) \wedge \\ & (\text{access}(w') = \text{access}(r)) \implies \\ & ((pid, w') \prec_{po} (pid, r))) \end{aligned}$$

$$\text{bypassCond}(hw, pid, (S_{po}, \prec_{po}), B, r) \iff$$
$$\begin{cases} \forall w \in B.((w \in \text{W}) \wedge (pid, w) \prec_{po} (pid, r) & \text{if } bypassing(HW) \\ \qquad \implies \text{access}(w) \neq \text{access}(r)) & \\ \forall w \in B.((w \in \text{W}) \implies \neg((pid, w) \prec_{po} (pid, r))) & \text{Otherwise} \end{cases}$$

In addition, we also define a function to return the value of *read forwarding* mechanism as:

$$\text{forwardValue}(pid, (S_{po}, \prec_{po}), B \uplus \{w\}, loc) =$$
$$\begin{cases} val(w) & \text{if } (w \in \text{W}) \wedge (\text{access}(w) = loc) \wedge \\ & \forall w' \in B.((w' \in \text{W}) \wedge ((pid, w') \prec_{po} (pid, w))) \\ \text{forwardValue}(pid, (S_{po}, \prec_{po}), B, loc) & \text{Otherwise} \end{cases}$$

Then, the semantics for read operations are:

– **Rule 7.** Blocking read

$\langle hw, \langle to, po, xo\rangle, locked, N, \{\langle pid, \sigma, B, L, E[r], p\rangle\} \uplus C\rangle \rightarrow$
$\langle hw, \langle to, po, xo \oplus (\{(pid, pid, r)\}, \emptyset)\rangle, locked, N, \{\langle pid, \sigma[\text{data}(r) \mapsto val], B, L, E[\varepsilon], p\rangle\} \uplus C\rangle$

If $\neg\text{nonBlocking}(hw) \wedge (r \in \text{R}) \wedge$
$\begin{cases} \text{readForwardingCond}(pid, po, B, r) & \text{if readForwarding}(hw) \\ \text{bypassCond}(hw, pid, po, B, r) \wedge & \text{Otherwise} \\ \text{networkCond}(\text{model}(hw), po, xo, pid, pid, r) \wedge & \\ \text{uniprocessorCond}(\text{model}(hw), po, xo, pid, r) & \end{cases}$
Where,

$$val = \begin{cases} \text{forwardValue}(pid, po, B, \text{access}(r)) & \text{if readForwarding}(hw) \\ \sigma(\text{access}(r)) & \text{Otherwise} \end{cases}$$

– Non-blocking read
  * **Rule 8.** Putting a read operation

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[r], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B \cup \{r\}, L, E[\varepsilon], p\rangle\} \uplus C\rangle$$

If $\text{nonBlocking}(hw) \wedge (r \in \text{R}) \wedge (\text{data}(r) \notin \{\text{data}(r') \mid (r' \in B) \text{ and } (r' \in \text{R})\})$
  * **Rule 9.** Issuing a read operation

$\langle hw, to, po, xo, locked, N, \{\langle pid, \sigma, B \uplus \{r\}, L, exec, p\rangle\} \uplus C\rangle \rightarrow$
$\langle hw, to, po, xo \oplus (\{(pid, pid, r)\}, \emptyset), locked, N, \{\langle pid, \sigma[\text{data}(r) \mapsto val], B, L, exec, p\rangle\} \uplus C\rangle$

If $\text{nonBlocking}(hw) \wedge (r \in \text{R}) \wedge$
$(\text{readForwarding}(hw) \implies \text{readForwardingCond}(pid, po, B, r)) \wedge$
$(\neg\text{readForwarding}(hw) \implies \text{bypassCond}(hw, pid, po, B, r) \wedge$
$\text{networkCond}(\text{model}(hw), po, xo, pid, pid, r) \wedge$
$\text{uniprocessorCond}(\text{model}(hw), po, xo, pid, r))$

Where

$$val = \begin{cases} \text{forwardValue}(pid, po, B, \text{access}(r)) & \text{if readfForwarding}(hw) \\ \sigma(\text{access}(r)) & \text{Otherwise} \end{cases}$$

• **Rule 10.** Lock operation

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[lock], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked \cup \{\text{lockEntity}(pid, lock)\}, N, \{\langle pid, \sigma, B, L, E[\varepsilon], p\rangle\} \uplus C\rangle$$

If $lock \in \text{LockOp} \wedge \text{lockEntity}(pid, lock) \notin locked$

- **Rule 11.** Unlock operation

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[unlock], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked \setminus \{\text{lockEntity}(pid, unlock)\}, N, \{\langle pid, \sigma, B, L, E[\varepsilon], p\rangle\} \uplus C\rangle$$

If $unlock \in$ UnlockOp

- **Rule 12.** Read-modify-write operations

$$\langle hw, \langle to, po, xo\rangle, locked, N, \{\langle pid, \sigma, B, L, E[rmw], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \langle to, po \oplus (\{r, w, rmw\}, \{r \prec_{rmw} w\}), xo\rangle, locked, N, \{\langle pid, \sigma, B, L, E[ops_{rmw}], p\rangle\} \uplus C\rangle$$

If $rmw \in$ RMW, Where

$$\begin{aligned} lock &\in \text{LockOp} \\ unlock &\in \text{UnlockOp} \\ \text{lockLocation}(lock) &= \text{locationRMW}(rmw) \\ \text{lockLocation}(unlock) &= \text{locationRMW}(rmw) \\ \text{componentR}(rmw) &= r \\ \text{componentW}(rmw) &= w \\ ops_{rmw} &= lock \cdot r \cdot w \cdot unlock \end{aligned}$$

- **Rule 13.** Fence operations

$$\langle hw, \langle to, po, xo\rangle, locked, N, \{\langle pid, \sigma, B, L, E[fence], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \langle to, po \oplus (\{fence\}, \emptyset), xo\rangle, locked, N, \{\langle pid, \sigma, B, L, E[\varepsilon], p\rangle\} \uplus C\rangle$$

If $fence \in$ FenceOp

- Expression valuation
  Let Val2N : Val $\rightarrow \mathbb{N}$ be a function to convert Val for computation in semantics. In addition, we need a function N2Val : $\mathbb{N} \rightarrow$ Val for converting to Val. We define a valuation alu : Expr $\rightarrow \mathbb{N}$ to compute expressions as followed,

$$\begin{aligned} [\![e]\!]_{\text{alu}} &= \text{Val2N}(e) && \text{If } e \in \text{Val} \\ [\![e_1 + e_2]\!]_{\text{alu}} &= [\![e_1]\!]_{\text{alu}} + [\![e_2]\!]_{\text{alu}} \\ [\![e_1 - e_2]\!]_{\text{alu}} &= [\![e_1]\!]_{\text{alu}} - [\![e_2]\!]_{\text{alu}} \end{aligned}$$

  The semantics will be

– **Rule 14.** Read data from bus

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[expr], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[\sigma(expr)], p\rangle\} \uplus C\rangle$$

If $(expr \in \text{internalBus}) \wedge$
$(\text{nonBlocking}(hw) \implies expr \notin \{\text{data}(r) \mid (r \in \text{R}) \wedge (r \in B)\})$

– **Rule 15.** Calculate value

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[e_1 \circ e_2], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[[\![e_1 \circ e_2]\!]_{\text{alu}})], p\rangle\} \uplus C\rangle$$

If $e_1, e_2 \notin \text{Val}$,
Where $\circ \in \{+, -\}$

- **Rule 16.** Assign operation

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[data := val], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma[data \mapsto val], B, L, E[\varepsilon], p\rangle\} \uplus C\rangle$$

If $val \in \text{Val}$

- Condition valuation Let condVal : CondExpr $\rightarrow \{0, 1\}$ be a valuation to compute value of condition as followed,

$$[\![e_1 = e_2]\!]_{\text{condVal}} = \begin{cases} 1 & \text{If } [\![e_1]\!]_{\text{alu}} = [\![e_2]\!]_{\text{alu}} \\ 0 & \text{Otherwise} \end{cases}$$
$$[\![e_1 \mathrel{!}= e_2]\!]_{\text{condVal}} = 1 - [\![e_1 = e_2]\!]_{\text{alu}}$$
$$[\![e_1 > e_2]\!]_{\text{condVal}} = \begin{cases} 1 & \text{If } [\![e_1]\!]_{\text{alu}} > [\![e_2]\!]_{\text{alu}} \\ 0 & \text{Otherwise} \end{cases}$$
$$[\![e_1 < e_2]\!]_{\text{condVal}} = \begin{cases} 1 & \text{If } [\![e_1]\!]_{\text{alu}} < [\![e_2]\!]_{\text{alu}} \\ 0 & \text{Otherwise} \end{cases}$$
$$[\![e_1 \wedge e_2]\!]_{\text{condVal}} = \begin{cases} [\![e_1]\!]_{\text{condVal}} & \text{If } [\![e_1]\!]_{\text{condVal}} > [\![e_2]\!]_{\text{condVal}} \\ [\![e_2]\!]_{\text{condVal}} & \text{Otherwise} \end{cases}$$
$$[\![e_1 \vee e_2]\!]_{\text{condVal}} = \begin{cases} [\![e_1]\!]_{\text{condVal}} & \text{If } [\![e_1]\!]_{\text{condVal}} < [\![e_2]\!]_{\text{condVal}} \\ [\![e_2]\!]_{\text{condVal}} & \text{Otherwise} \end{cases}$$

- Condition operation

– **Rule 17.** Opt operations

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[(\langle cond\rangle ops_1 : ops_2)], p\rangle\} \uplus C\rangle \rightarrow$$
$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[ops'], p\rangle\} \uplus C\rangle$$

Where $ops' = \begin{cases} ops_1 & \text{if } [\![cond]\!]_{\text{condVal}} = 1 \\ ops_2 & \text{Otherwise} \end{cases}$

– **Rule 18.** Opt values

$$\langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[(\langle cond\rangle ? expr_1 : expr_2)], p\rangle\} \uplus C\rangle \rightarrow \langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, E[expr'], p\rangle\} \uplus C\rangle$$

$$\text{Where } expr' = \begin{cases} expr_1 & \text{if } [\![cond]\!]_{\text{condVal}} = 1 \\ expr_2 & \text{Otherwise} \end{cases}$$

3.2.4 Conditions for memory consistency models

In order to enforce the execution of our formal model, we have introduced two predicated functions, uniprocessorCond and networkCond, which enforce the execution in the semantics. As we have proposed rules to perform the read/wrire operations, these predicate functions are used as conditions for applying these rules. These predicated functions are defined the structures as:

$$\begin{aligned} \text{uniprocessorCond} :& \text{MemModel} \times \text{Order}\langle \text{ID} \times \text{MemOp}\rangle \times \text{Order}\langle \text{ID} \times \text{ID} \times \text{RW}\rangle \\ & \times \text{ID} \times \text{RW} \rightarrow \text{Boolean} \\ \text{networkCond} :& \text{MemModel} \times \text{Order}\langle \text{ID} \times \text{MemOp}\rangle \times \text{Order}\langle \text{ID} \times \text{ID} \times \text{RW}\rangle \\ & \times \text{ID} \times \text{ID} \times \text{RW} \rightarrow \text{Boolean} \end{aligned}$$

These predicated functions are adopt idea from the framework of [Gha95], which specify the system requirements of a given model. As we have introduced the the framework in section 3.1.5, we adopt the idea of the constraints on *program order* and *execution order* for a given model. As for uniprocssorCond, we constraint the execution order of read/write operations on the core. In contrast, networkCond constraint the execution order happened among cores. Hence, these functions also need the information of *memory consistency model*,*program order*, *execution order* and the read/write operation, which consider whether can be performed.

In order to provide these predicated functions based on memory consistency model, we adopt the constraints provided by the framework in [Gha95]. In the specifications which they proposed, we focus on *aggressive specifications* which are proposed to exploit the optimization mechanisms. In such specifications, we will adopt the constraints on the *program order*($\xrightarrow{po}$) and *execution order*($\xrightarrow{xo}$) which are proposed in the condition (b), as we give an example in specification 1 for sequential consistency model. As our model have proposed the program order and execution order as $(S_{po}, \prec_{po})$ and $(S_{xo}, \prec_{xo})$, these orders correspond to the program order($\xrightarrow{po}$) and execution order($\xrightarrow{xo}$) in that framework. The following list is described the correspondence of their framework and our semantics. Let X and Y are read or write operations.

- $X \xrightarrow{po} Y$ corresponds to $(pid_x, X) \prec_{po} (pid_y, Y)$,
  where X and Y are issued from processor P_{pid_x} and P_{pid_y}, respectively.
- $X(i) \xrightarrow{xo} Y(j)$ corresponds to $(pid_X, i, X) \prec_{xo} (pid_Y, j, Y)$,
  where X and Y are issued from processor P_{pid_x} and P_{pid_y}, respectively.
- $X \xrightarrow{co} Y$ corresponds to $X <_{co(xo)} Y$

- $X \xrightarrow{co'} Y$ corresponds to $X <_{co'(xo)} Y$

The $(S_{po}, \prec_{po})$ and $(S_{xo}, \prec_{xo})$ are the information contained in the *configuration*, $\langle hw, \Phi, locked, N, C\rangle$ and $\Phi = \langle to, (S_{po}, \prec_{po}), (S_{xo}, \prec_{xo})\rangle$,of semantics. These information are necessary for considering the execution behaviors on our semantics. However, the last two relations are not appeared in our semantics. The definitions are as following:

$$\begin{aligned}
\text{conflict}(x, y) &\iff (x, y \in \text{RW}) \wedge (\text{access}(x) = \text{access}(y)) \wedge ((x \in \text{W}) \vee (y \in \text{W})) \\
x <_{co((S_{xo}, \prec_{xo}))} y &\iff \text{conflict}(x, y) \wedge \exists i, k \in \text{ID}.((i, k, x) \prec_{xo} (i, k, y)) \\
x <_{co'((S_{xo}, \prec_{xo}))} y &\iff \text{conflict}(x, y) \wedge \exists i, j, k \in \text{ID}.(j \neq k \implies (i, j, x) \prec_{xo} (i, k, y))
\end{aligned}$$

Next, we will consider the *aggressive conditions* provided by [Gha95]. Each model will provide different constraints and additional relations to consider the execution behaviors. Hence, the following topics will consider *sequential consistency model* and *partial store ordering model*.

Sequential consistency model

Firstly, let's consider the easiest model to consider the conditions for execution. The aggressive constraints of sequential consistency is provided as specification 1. To provide the conditions, we consider only condition *(b)* which is provided by that figure because the condition *(b)* is the condition for enforcing the behaviors of executions. In addition, this model requires additional relations, $\xrightarrow{spo}$ and $\xrightarrow{sco}$. Those relations are defined based on our semantics as followed:

$$\begin{aligned}
x <_{spo(pid,(S_{po}, \prec_{po}))} y &\iff ((pid, x) \prec_{po} (pid, y)) \wedge (\text{access}(x) \neq \text{access}(y)) \\
x <_{sco(xo)} y &\iff ((x <_{co'(xo)} y) \vee \\
&\qquad ((x, y \in \text{R}) \wedge \exists w \in \text{W}.((x <_{co'(xo)} w) \wedge (w <_{co'(xo)} y))))
\end{aligned}$$

The condition *(b)* of sequential consistency in specification 1 is defined as:

Giving memory operation X and Y, if X and Y are conflict and X, Y are the first and last operations in one of:

- Uniprocessors: $RW \xrightarrow{po} W$
- Coherence: $W \xrightarrow{co'} W$
- Multiprocessor dependence chain: one of
  - $W \xrightarrow{co'} R \xrightarrow{po} RW$
  - $RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$
  - $W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$

Then $X(i) \xrightarrow{xo} Y(i)$ for all i.

To provide condition based on our semantics, we provide this condition *(b)* as

- *uniprocessorCond* that consists only Uniprocessor ($RW \xrightarrow{po} W$)
- *networkCond* which consists of
  - Coherence : $W \xrightarrow{co'} W$, and
  - Multiprocessor dependence chain.

Therefore, these conditions are defined as followed:

$$\text{uniprocessorCond}(\text{SC}, (S_{po}, \prec_{po}), xo, i, w) \iff (w \in \text{W}) \wedge \forall x \in \text{RW}.(\text{conflict}(x, w) \wedge ((i, x) \prec_{po} (i, w)) \implies (i, i, x) \prec_{xo'} (i, i, w)$$

Where $(S_{xo'}, \prec_{xo'}) = xo \oplus (\{(i, i, w)\}, \emptyset)$

$$\text{networkCond}(\text{SC}, po, xo, i, j, rw) \iff \forall x \in \text{RW}, k \in \text{ID}.((\text{conflict}(x, rw) \wedge (\text{co_cond}(xo, x, rw) \vee \text{mul_chain}(po, xo, k, x, j, rw))) \implies (i, k, x) \prec_{xo''} (i, j, rw))$$

Where $(S_{xo''}, \prec_{xo''}) = xo \oplus (\{(i, j, rw)\}, \emptyset)$
For networkCond, we need additional predicated functions for representing co_cond and mul_chain. Such predicated functions are described as:

- co_cond $(W \xrightarrow{co'} W)$:

$$\text{co_cond}(xo, w, w') \iff (w \in W) \wedge (w' \in W) \wedge (w <_{co'(xo)} w')$$

- Multiprocessor dependence chain

$$\text{mul_chain}(po, xo, i, x, j, y) \iff \text{mul_cond1}(po, xo, x, j, y) \vee \text{mul_cond2}(po, xo, i, x, j, y) \vee \text{mul_cond3}(po, xo, x, y)$$

- mul_cond1 $(W \xrightarrow{co'} R \xrightarrow{po} RW)$:

$$\text{mul_cond1}((S_{po}, \prec_{po}), xo, w, j, rw) \iff (w \in \text{W}) \wedge \exists r \in \text{R}.(w <_{co'(xo)} r \wedge (j, r) \prec_{po} (j, rw))$$

- mul_cond2 $(RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW)$:

$$\text{mul_cond2}(po, xo, i, rw, j, rw') \iff \exists a, b, c \in \text{RW}.((rw <_{spo(i,po)} a) \wedge (c <_{spo(j,po)} rw') \wedge (a <_{sco(xo)} b) \wedge \text{trans_rel1}(po, xo, b, c))$$

$$\text{trans_rel1}(po, xo, b, c) \iff \begin{cases} \text{True} & \text{if } b = c \\ \exists_{a,b' \in \text{RW}, i \in \text{ID}}(& \\ (b <_{spo(i,po)} a) \wedge (a <_{sco(xo)} b') \wedge & \\ \text{trans_rel1}(po, xo, b', c)) & \text{if } b \neq c \end{cases}$$

- mul_cond3 ($W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$):

$$\text{mul_cond3}(po, xo, w, j, r) \iff (w \in \text{W}) \wedge (r \in \text{R}) \wedge \exists r_1 \in \text{R}.\exists a, b, c \in \text{RW}.\exists i \in \text{ID}.((w <_{sco(xo)} r_1) \wedge (r_1 <_{spo(i,po)} a) \wedge (a <_{sco(xo)} b) \wedge (c <_{spo(j,po)} r) \wedge \text{trans_rel1}(po, xo, b, c))$$

Partial store ordering model

The next memory consistency model which describe in this work is *partial consistency model*. The conceptual system of this model is shown as specification 2.

Specification 2: Aggressive conditions for partial store ordering

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ W
W(in RMW) $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Condition on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; *applies to all write sub-operations.*
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: $RW \xrightarrow{po} W$
coherence: $W \xrightarrow{co'} W$
multiprocessor dependence chain: one of
$W \xrightarrow{co'} R \xrightarrow{po} RW$
$RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$
$W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$

This model requires the fence instruction, STBAR, to consider the relation $\xrightarrow{spo}$. The additional relations of this model are defined based on our semantics as followed:

$$
\begin{aligned}
x <^{pso}_{spo(pid,(S_{po},\prec_{po}))} y \iff & (((x \in \mathrm{R}) \wedge ((pid,x) \prec_{po} (pid,y)))\vee \\
& ((x,y \in \mathrm{W}) \wedge \exists stbar \in \mathrm{FenceOp}.((\mathrm{fenceType}(stbar) = STBAR) \\
& \wedge ((pid,x) \prec_{po(po)} (pid,stbar)) \wedge ((pid,stbar) \prec_{po} (pid,y))))\vee \\
& (\exists rmw \in \mathrm{RMW}((\mathrm{componentW}(rmw) = x) \wedge ((pid,x) \prec_{po} (pid,y))))\vee \\
& ((x \in \mathrm{W}) \wedge (y \in \mathrm{R}) \wedge \exists rmw \in \mathrm{RMW}, stbar \in \mathrm{FenceOp}.(\\
& ((pid,x) <_{po} (pid,stbar)) \wedge ((pid,stbar) <_{po} (pid,rmw))\wedge \\
& ((pid,rmw) \prec_{po} (pid,y)))))
\end{aligned}
$$

$$
\begin{aligned}
x <^{pso}_{sco(xo)} y \iff & ((x <_{co(xo)} y)\vee \\
& ((x,y \in \mathrm{R}) \wedge \exists w \in \mathrm{W}.((x <_{co(xo)} w) \wedge (w <_{co(xo)} y))))
\end{aligned}
$$

The condition *(b)* of *partial store ordering* is defined as:

Giving memory operation X and Y, if X and Y are conflict and X, Y are the first and last operations in one of:

- Uniprocessors: $RW \xrightarrow{po} W$
- Coherence: $W \xrightarrow{co'} W$
- Multiprocessor dependence chain: one of
  - $W \xrightarrow{co} R \xrightarrow{spo} RW$
  - $RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$
  - $W \xrightarrow{co} R \xrightarrow{spo} \{A \xrightarrow{spo} B \xrightarrow{spo}\} + R$

Then $X(i) \xrightarrow{xo} Y(i)$ for all i.

We have seen that the conditions of partial store ordering are similar to sequential consistency model. Nevertheless, the different are the additional relations, $\xrightarrow{spo}$ and $\xrightarrow{sco}$. Therefore, the conditions of partial store ordering based on our semantics are:

$$\begin{aligned}\text{uniprocessorCond}(\text{PSO}, (S_{po}, \prec_{po}), xo, i, w) \iff & (w \in \text{W}) \wedge \forall x \in \text{RW}.(\text{conflict}(x, w) \wedge \\ & ((i, x) \prec_{po} (i, w)) \implies (i, i, x) \prec_{xo'} (i, i, w)\end{aligned}$$

Where $(S_{xo'}, \prec_{xo'}) = xo \oplus (\{(i, i, w)\}, \emptyset)$

$$\begin{aligned}\text{networkCond}(\text{PSO}, po, xo, i, j, rw) \iff & \forall x \in \text{RW}, k \in \text{ID}.((\text{conflict}(x, rw) \wedge \\ & (\text{co_cond}(xo, x, rw) \vee \text{mul_chain}(po, xo, k, x, j, rw))) \\ & \implies (i, k, x) \prec_{xo''} (i, j, rw))\end{aligned}$$

Where $(S_{xo''}, \prec_{xo''}) = xo \oplus (\{(i, j, rw)\}, \emptyset)$

For networkCond, we need additional predicated functions for representing co_cond and mul_chain. Such predicated functions are described as:

- co_cond ($W \xrightarrow{co'} W$) :

$$\text{co_cond}(xo, w, w') \iff (w \in W) \wedge (w' \in W) \wedge (w <_{co'(xo)} w')$$

- Multiprocessor dependence chain

$$\begin{aligned}\text{mul_chain}(po, xo, i, x, j, y) \iff & \text{mul_cond1}(po, xo, x, j, y) \vee \\ & \text{mul_cond2}(po, xo, i, x, j, y) \vee \text{mul_cond3}(po, xo, x, y)\end{aligned}$$

- mul_cond1 ($W \xrightarrow{co'} R \xrightarrow{po} RW$):

$$\begin{aligned}\text{mul_cond1}((S_{po}, \prec_{po}), xo, w, j, rw) \iff & (w \in \text{W}) \wedge \\ & \exists r \in \text{R}.(w <_{co'(xo)} r \wedge (j, r) \prec_{po} (j, rw))\end{aligned}$$

- mul_cond2 ($RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$):

$$\begin{aligned}\text{mul_cond2}(po, xo, i, rw, j, rw') \iff & \exists a, b, c \in \text{RW}.((rw <^{pso}_{spo(i,po)} a) \wedge (c <^{pso}_{spo(j,po)} rw') \wedge \\ & (a <^{pso}_{sco(xo)} b) \wedge \text{trans_rel1}(po, xo, b, c))\end{aligned}$$

$$\text{trans_rel1}(po, xo, b, c) \iff \begin{cases} \text{True} & \text{if } b = c \\ \exists_{a,b' \in \text{RW}, i \in \text{ID}}(& \\ (b <^{pso}_{spo(i,po)} a) \wedge (a <^{pso}_{sco(xo)} b') \wedge & \\ \text{trans_rel1}(po, xo, b', c)) & \text{if } b \neq c \end{cases}$$

- mul_cond3 ($W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$):

$$\begin{aligned}\text{mul_cond3}(po, xo, w, j, r) \iff & (w \in \text{W}) \wedge (r \in \text{R}) \wedge \\ & \exists r_1 \in \text{R}. \exists a, b, c \in \text{RW}. \exists i \in \text{ID}. (\\ & (w <^{pso}_{sco(xo)} r_1) \wedge (r_1 <^{pso}_{spo(i,po)} a) \wedge (a <^{pso}_{sco(xo)} b) \wedge \\ & (c <^{pso}_{spo(j,po)} r) \wedge \text{trans_rel1}(po, xo, b, c))\end{aligned}$$

3.2.5 Instruction set

Since the semantics focus on the behaviors that instruction, the definitions of *instruction set architecture* are not taken into account,yet. Due to there are many instruction set architectures, we do need to generalize the behaviors of hardware that affect the memory operations inside the hardware. Nevertheless, in this subsection, we will give some example of instruction definition based on ARM Instruction set architecture[Lim08]. The definition of the ARM instructions will be provided by ISA functions defined in the syntax.

- MOV{$\langle cond \rangle$} Rd, #imm32

$$\text{ISA}(\langle mov, cond, rd \cdot imm \rangle) = (\langle \text{passCond}(cond) \rangle (lock \cdot w \cdot unlock) : \varepsilon)$$

If $((rd \in \text{Reg}) \wedge (imm \in \text{Val}))$
Where

$w \in \text{W}$	
$\text{access}(w) = rd$	access Rd
$\text{data}(w) = imm$	Rd = imm32
$lock \in \text{LockOp}$	
$unlock \in \text{UnlockOp}$	
$\text{lockLocation}(lock) = \text{access}(w)$	lock Rd
$\text{lockLocation}(unlock) = \text{access}(w)$	unlock Rd

- MOV{$\langle cond \rangle$} Rd, Rm

$$\text{ISA}(\langle mov, cond, rd \cdot rm \rangle) = (\langle \text{passCond}(cond) \rangle) lock_{rd} \cdot lock_{rm} \cdot r \cdot w \cdot unlock_{rm} \cdot unlock_{rd} : \varepsilon)$$

If $((rd \in Reg) \wedge$
$(rm \in Reg))$

Where

$$
\begin{aligned}
w &\in \mathrm{W} \\
r &\in \mathrm{R} \\
result &\in \mathrm{internalBus} \\
(\mathrm{access}(r), \mathrm{data}(r)) &= (Rm, result) && \mathrm{result} = \mathrm{Rm} \\
(\mathrm{access}(w), \mathrm{data}(w)) &= (Rd, result) && \mathrm{Rd} = \mathrm{result} \\
lock_{rd}, lock_{rm} &\in \mathrm{LockOp} \\
unlock_{rd}, unlock_{rm} &\in \mathrm{UnlockOp} \\
\mathrm{lockLocation}(lock_{rd}) &= Rd \\
\mathrm{lockLocation}(unlock_{rd}) &= Rd \\
\mathrm{lockLocation}(lock_{rm}) &= Rm \\
\mathrm{lockLocation}(unlock_{rm}) &= Rm
\end{aligned}
$$

- B{$\langle cond \rangle$} imm32

$$\mathrm{ISA}(\langle b, cond, imm \rangle) = (\langle \mathrm{passCond}(cond) \rangle ops_B : \varepsilon)$$

If $imm \in \mathrm{Val}$,
Where

$$
\begin{aligned}
w &\in \mathrm{W} \\
r &\in \mathrm{R} \\
result &\in \mathrm{internalBus} \\
(\mathrm{access}(r), \mathrm{data}(r)) &= (pc, result) && \mathrm{result} = \mathrm{pc} \\
(\mathrm{access}(w), \mathrm{data}(w)) &= (pc, result) && \mathrm{pc} = \mathrm{result} \\
lock_{pc} &\in \mathrm{LockOp} \\
unlock_{pc} &\in \mathrm{UnlockOp} \\
\mathrm{lockLocation}(lock_{pc}) &= \mathrm{pc} \\
\mathrm{lockLocation}(unlock_{pc}) &= \mathrm{pc} \\
ops_B &= lock_{\mathrm{pc}} \cdot r \cdot (result := result + imm) \cdot w \cdot unlock_{pc}
\end{aligned}
$$

- CMP{$\langle cond \rangle$} Rn, Rm

$$
\begin{aligned}
\mathrm{ISA}(\langle cmp, cond, rn \cdot rm \rangle) = (&\langle \mathrm{passCond}(cond) \rangle lock_{rn} \cdot lock_{rm} \cdot r_{rn} \cdot \\
&r_{rm} \cdot (result := (\langle tmp_{rn} = tmp_{rm} \rangle ?1 : 0)) \cdot \\
&w_Z \cdot (result := (\langle tmp_{rm} > tmp_{rn} \rangle ?1 : 0)) \cdot \\
&w_N \cdot unlock_{rn} \cdot unlock_{rm} : \varepsilon)
\end{aligned}
$$

If $rn, rm \in \text{Reg}$
Where

$$
\begin{aligned}
r_{rn}, r_{rm} &\in \text{R} \\
w_Z, w_N &\in \text{W} \\
tmp_{rm}, tmp_{rn}, result &\in \text{internalBus} \\
(\text{access}(r_{rn}), \text{data}(r_{rn})) &= (rn, tmp_{rn}) \\
(\text{access}(r_{rm}), \text{data}(r_{rm})) &= (rm, tmp_{rm}) \\
(\text{access}(w_Z), \text{data}(w_Z)) &= (Z, result) \\
(\text{access}(w_N), \text{data}(w_N)) &= (N, result) \\
lock_{rn}, lock_{rm} &\in \text{LockOp} \\
unlock_{rn}, unlock_{rm} &\in \text{UnlockOp} \\
\text{lockLocation}(lock_{rn}) = \text{lockLocation}(unlock_{rn}) &= rn \\
\text{lockLocation}(lock_{rm}) = \text{lockLocation}(unlock_{rm}) &= rm
\end{aligned}
$$

- CMP{$\langle cond \rangle$} Rn, imm (ignore overflow and currying bits)

$$
\begin{aligned}
\text{ISA}(\langle cmp, cond, rn \cdot imm \rangle) = (&\langle \text{passCond}(cond) \rangle \\
&lock_{rn} \cdot r_{rn} \cdot (result := \langle tmp_{rn} = imm \rangle ? 1 : 0) \cdot w_Z \cdot \\
&(result := \langle imm > tmp_{rn} \rangle ? 1 : 0) \cdot w_N \cdot unlock_{rn} : \varepsilon)
\end{aligned}
$$

If $(rn \in \text{Reg})$
Where

$$
\begin{aligned}
r_{rn} &\in \text{R} \\
w_Z, w_N &\in \text{W} \\
tmp_{rn}, result &\in \text{internalBus} \\
(\text{access}(r_{rn}), \text{data}(r_{rn})) &= (rn, tmp_{rn}) \\
(\text{access}(w_Z), \text{data}(w_Z)) &= (Z, result) \\
(\text{access}(w_N), \text{data}(w_N)) &= (N, result) \\
lock_{rn} &\in \text{LockOp} \\
unlock_{rn} &\in \text{UnlockOp} \\
\text{lockLocation}(lock_{rn}) = \text{lockLocation}(unlock_{rn}) &= rn
\end{aligned}
$$

- SWP{⟨*cond*⟩} Rt, Rt2, [Rn]

$$\text{ISA}(\langle swp, cond, rt \cdot rt2 \cdot Mem[rn]\rangle) = (\langle \text{passCond}(cond)\rangle lock_{rt} \cdot lock_{rt2} \cdot lock_{rn} \cdot r_{rt2} \cdot r_{rn} \cdot rmw_{mem[rn]} \cdot w_{rt} \cdot unlock_{rt} \cdot unlock_{rt2} \cdot unlock_{rn} : \varepsilon)$$

If $(rt, rt2, rn \in \text{Reg})$
Where

$$\begin{aligned}
r_{rt2}, r_{rn}, r_{mem[rn]} &\in \text{R} \\
w_{rt}, w_{mem[rn]} &\in \text{W} \\
rmw_{mem[rn]} &\in \text{RMW} \\
data, tmp_{rn}, tmp_{rt}, tmp_{rt2} &\in \text{internalBus} \\
(\text{access}(r_{rt2}), \text{data}(r_{rt2})) &= (rt2, tmp_{rt2}) \qquad (tmp_{rt2}) = \text{Rt2} \\
(\text{access}(r_{rn}), \text{data}(r_{rn})) &= (rn, tmp_{rn}) \qquad (tmp_{rn}) = \text{Rn} \\
(\text{access}(r_{mem[rn]}), \text{data}(r_{mem[rn]})) &= (Mem[tmp_{rn}], data) \qquad \text{data} = \text{Mem[Rn}(tmp_{rn})] \\
(\text{access}(w_{mem[rn]}), \text{data}(w_{mem[rn]})) &= (Mem[tmp_{rn}], tmp_{rt2}) \qquad \text{Mem[Rn]} = \text{Rt2}(tmp_{rt2}) \\
(\text{access}(w_{rt}), \text{data}(w_{rt})) &= (rt, data) \qquad \text{Rt} = \text{data} \\
\text{locationRMW}(rmw_{mem[rn]}) &= Mem[rn] \\
\text{componentR}(rmw_{mem[rn]}) &= r_{mem[rn]} \\
\text{componentW}(rmw_{mem[rn]}) &= w_{mem[rn]} \\
lock_{rt}, lock_{rt2}, lock_{rn} &\in \text{LockOp} \\
unlock_{rt}, unlock_{rt2}, unlock_{rn} &\in \text{UnlockOp} \\
\text{lockLocation}(lock_{rt}) &= rt \\
\text{lockLocation}(unlock_{rt}) &= rt \\
\text{lockLocation}(lock_{rt2}) &= rt2 \\
\text{lockLocation}(unlock_{rt2}) &= rt2 \\
\text{lockLocation}(lock_{rn}) &= rn \\
\text{lockLocation}(unlock_{rn}) &= rn
\end{aligned}$$

- LDR{⟨*cond*⟩} Rt, [Rn]

$$\begin{aligned}\text{ISA}(\langle ldr, cond, rt \cdot Mem[rn]\rangle) =& (\langle \text{passCond}(cond)\rangle lock_{rt} \cdot lock_{rn} \cdot r_{rn} \cdot r_{mem[rn]} \cdot \\ & w_{rt} \cdot unlock_{rt} \cdot unlock_{rn} : \varepsilon)\end{aligned}$$

If $(rt, rn \in \text{Reg})$
Where

$$\begin{aligned} r_n, r_{mem[rn]} &\in \text{R} \\ w_{rt} &\in \text{W} \\ tmp_{rn}, data &\in \text{internalBus} \\ (\text{access}(r_n), \text{data}(r_{rn})) &= (rn, tmp_{rn}) && (tmp_{rn}) = \text{Rn} \\ (\text{access}(r_{mem[rn]}), \text{data}(r_{mem[rn]})) &= (Mem[tmp_{rn}], data) && \text{data} = \text{Mem[Rn]} \\ (\text{access}(w_{rt}), \text{data}(w_{rt})) &= (rt, data) && \text{Rt} = \text{data} \\ lock_{rt}, lock_{rn} &\in \text{LockOp} \\ unlock_{rt}, unlock_{rn} &\in \text{UnlockOp} \\ \text{lockLocation}(lock_{rt}) = \text{lockLocation}(unlock_{rt}) &= rt \\ \text{lockLocation}(lock_{rn}) = \text{lockLocation}(unlock_{rn}) &= rn \end{aligned}$$

- STR{⟨*cond*⟩} Rt, [Rn]

$$\begin{aligned}\text{ISA}(\langle str, cond, rt \cdot Mem[rn]\rangle) =& (\langle \text{passCond}(cond)\rangle lock_{rt} \cdot lock_{rn} \cdot r_{rt} \cdot r_{rn} \cdot \\ & w_{mem[rn]} \cdot unlock_{rt} \cdot unlock_{rn} : \varepsilon)\end{aligned}$$

If $(rt, rn \in \text{Reg})$
Where

$$\begin{aligned} r_{rt}, r_{rn} &\in \text{R} \\ w_{mem[rn]} &\in \text{W} \\ (\text{access}(r_{rt}), \text{data}(r_{rt})) &= (rt, data) && \text{data} = \text{Rt} \\ (\text{access}(r_{rn}), \text{data}(r_{rn})) &= (rn, tmp_{rn}) && (tmp_{rn}) = \text{Rn} \\ (\text{access}(w_{mem[rn]}), \text{data}(w_{mem[rn]})) &= (Mem[tmp_{rn}], data) && \text{Mem[Rn]} = \text{data} \\ lock_{rt}, lock_{rn} &\in \text{LockOp} \\ unlock_{rt}, unlock_{rn} &\in \text{UnlockOp} \\ \text{lockLocation}(lock_{rt}) = \text{lockLocation}(unlock_{rt}) &= rt \\ \text{lockLocation}(lock_{rn}) = \text{lockLocation}(unlock_{rn}) &= rn \end{aligned}$$

```
1 MOV r2, #1
2 STR r2, [r3]
```

ProgramA

```
1 LDR r1, [r3]
```

ProgramB

Figure 3-5: A sample program for considering execution

3.3 Sample executions based on our semantics

As we have proposed operational semantics for describing the abstract multi-core systems. In this section, we would like to show the sample execution of a sample program.We will consider the program shown as figure 3-5. Note that, the register $r3$ on both cores refer to the same shared-memory location.

Due to there are many possible executions that can be produced by this programs, we will consider only one path of the execution to show the execution of our semantics. First of all, we consider the initial state as:

$$\langle hw, \langle(\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset)\rangle, \emptyset, \emptyset, C\rangle$$

Where

$$\begin{aligned} Cores &= (\langle 1, \sigma^1_{init}, \emptyset, \emptyset, \varepsilon, ProgramA\rangle \cdot \\ &\qquad \langle 2, \sigma^2_{init}, \emptyset, \emptyset, \varepsilon, ProgramB\rangle) \\ \sigma^1_{init} &= \sigma^2_{init} \\ &= \{loc \mapsto 0 \mid \forall loc \in \text{Location}\}[pc \mapsto 1] \end{aligned}$$

As for hardware configuration, we will let attributes of the hw as

$$\begin{aligned} \text{inOrderIssue}(hw) &= True \\ \text{nonBlocking}(hw) &= True \\ \text{readForwarding}(hw) &= True \\ \text{bypassing}(hw) &= False \\ \text{model}(hw) &= \text{SC} \end{aligned}$$

Then, let's consider the execution from the initial state,

$\langle hw, \langle(\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset)\rangle, \emptyset, \emptyset, C\rangle$

$= \langle hw, \langle to_{init}, po_{init}, xo_{init}\rangle, \emptyset, \emptyset, C\rangle$ (Simplify)

$= \langle hw, \Phi_{init}, \emptyset, \emptyset, C\rangle$ (Simplify)

$= \langle hw, \langle(\emptyset,\emptyset),(\emptyset,\emptyset), xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{init}, \emptyset, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$ (Consider core 1)

$\rightarrow \langle hw, \langle(\{t_1\},\emptyset), po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{init}[pc = 2], \emptyset, \emptyset, (\langle True\rangle ops^{t_1}_{mov} : \varepsilon), ProgramA\rangle\} \uplus C\rangle$

Where $ops^{t_1}_{mov} = ISA^{t1}(ProgramA(\sigma^1_{init}(pc))) = lock_{r2} \cdot w_{r2} \cdot unlock_{r2}$, and

$po_1 = (\{w_{r2}\}, \emptyset)$ (Fetching in-order)

$\rightarrow \langle hw, \langle(\{t_1\},\emptyset), po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{init}[pc = 2], \emptyset, \emptyset, ops^{t_1}_{mov}, ProgramA\rangle\} \uplus C\rangle$ (Condition operations)

$= \langle hw, \langle to_1, po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_1}, \emptyset, \emptyset, ops^{t_1}_{mov}, ProgramA\rangle\} \uplus C\rangle$ (Simplify)

$= \langle hw, \langle to_1, po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_1}, \emptyset, \emptyset, E[lock_{r2}], ProgramA\rangle\} \uplus C\rangle$ (Lock operation)

$\rightarrow \langle hw, \langle to_1, po_1, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_1}, \emptyset, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$ (Lock semantic)

$= \langle hw, \langle to_1, po_1, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_1}, \emptyset, \emptyset, E[w_{r2}], ProgramA\rangle\} \uplus C\rangle$ (Write operation)

$\rightarrow \langle hw, \langle to_1, po_1, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_1}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$ (Write semantics)

$= \langle hw, \langle to_1, po_1, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_1}, \{w_{r2}\}, \emptyset, E[unlock_{r2}], ProgramA\rangle\} \uplus C\rangle$ (Unlock operation)

$\rightarrow \langle hw, \langle to_1, po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_1}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$ (Unlock semantic)

$= \langle hw, \langle to_1, po_1, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_1}, \{w_{r2}\}, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$ (Simplify)

$\rightarrow \langle hw, \langle to_2, po_2, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_1}[pc = 3], \{w_{r2}\}, \emptyset, ops^{t_2}_{str}, ProgramA\rangle\} \uplus C\rangle$

Where $ops^{t_2}_{str} = ISA^{t_2}(ProgramA(2))$

$= (\langle \text{True}\rangle lock_{r2} \cdot lock_{r3} \cdot R_{r2} \cdot R_{r3} \cdot W_{mem[r3]} \cdot unlock_{r3} \cdot unlock_{r2} : \varepsilon)$

$to_2 = to_1 \oplus (\{t_2\}, \emptyset) = (\{t_1, t_2\}, \{t_1 \prec t_2\})$,

$po_2 = po_1 \oplus (\{R_{r2}, R_{r3}, W_{mem[r3]}\}, \{R_{r2} \prec R_{r3}, R_{r3} \prec W_{mem[r3]}\})$ (Fetching in-order)

$= \langle hw, \langle to_2, po_2, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, (\langle \text{True}\rangle ops' : \varepsilon), ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, (ops'), ProgramA\rangle\} \uplus C\rangle$ (Condition operations)

$= \langle hw, \langle to_2, po_2, xo\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, E[lock_{r2}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$ (Lock semantics)

$= \langle hw, \langle to_2, po_2, xo\rangle, \{ent_{r2}\}, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, E[lock_{r3}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo\rangle, \{ent_{r2}, ent_{r3}\}, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo\rangle, \{ent_{r2}, ent_{r3}\}, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}\}, \emptyset, E[R_{r2}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo\rangle, \{ent_{r2}, ent_{r3}\}, \emptyset, \{\langle 1, \sigma^1_{t_2}, \{w_{r2}, R_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo \oplus (\{(1, R_{r2})\}, \emptyset)\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'_2}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

Where $\sigma^1_{t'_2} = \sigma^1_{t_2}[data \mapsto 1]$

$lock_{r2,r3} = \{ent_{r2}, ent_{r3}\}$ (Read-forwarding behavior)

$= \langle hw, \langle to_2, po_2, xo_1\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'_2}, \{w_{r2}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_1\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'_2}, \{w_{r2}\}, \emptyset, E[R_{r3}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_1\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'_2}, \{w_{r2}, R_{r3}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_1\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'_2}, \{w_{r2}, R_{r3}\}, \emptyset, E[W_{mem[r3]}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_2\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t''_2}, \{R_{r3}\}, \emptyset, E[W_{mem[r3]}], ProgramA\rangle\} \uplus C\rangle$

Where $\sigma^1_{t''_2} = \sigma^1_{t'_2}[R_2 \mapsto 1]$

$xo_2 = xo_1 \oplus (\{(1, 1, R_2)\}, \emptyset)$

$\rightarrow \langle hw, \langle to_2, po_2, xo_3\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \emptyset, \emptyset, E[W_{mem[r3]}], ProgramA\rangle\} \uplus C\rangle$

Where $\sigma^1_{t'''_2} = \sigma^1_{t''_2}[tmp_{r3} \mapsto \text{Adr}]$

$xo_3 = xo_2 \oplus (\{(1, 1, R_{r3}\}, \emptyset)$ (Adr refers to a constant)

$\rightarrow \langle hw, \langle to_2, po_2, xo_3\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \{W_{mem[r3]}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_3\rangle, locked_{r2,r3}, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \{W_{mem[r3]}\}, \emptyset, E[unlock_{r3}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_3\rangle, \{ent_{r3}\}, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \{W_{mem[r3]}\}, \emptyset, E[\varepsilon], ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_3\rangle, \{ent_{r3}\}, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \{W_{mem[r3]}\}, \emptyset, E[unlock_{r2}], ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_3\rangle, \emptyset, \emptyset, \{\langle 1, \sigma^1_{t'''_2}, \{W_{mem[r3]}\}, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_3)\rangle, \emptyset, \{(1, 1, W_{mem[r3]}), (1, 2, W_{mem[r3]})\}, \{\langle 1, \sigma^1_{t'''_2}, \emptyset, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_3\rangle, \emptyset, \{(1, 1, W_{mem[r3]}), (1, 2, W_{mem[r3]})\}, \{\langle 2, \sigma^2_{init}, \emptyset, \emptyset, \varepsilon, ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_3\rangle, \emptyset, N_1 \uplus \{(1, 2, W_{mem[r3]})\}, \{\langle 2, \sigma^2_{init}, \emptyset, \emptyset, \varepsilon, ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2, po_2, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_{init}[\text{Mem}[Adr] \mapsto 1], \emptyset, \emptyset, \varepsilon, ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_2, po_2, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_1, \emptyset, \emptyset, \varepsilon, ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_2 \oplus (\{t_3\}, \emptyset), po_3, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_1[\text{pc} \mapsto 2], \emptyset, \emptyset, ops^{t_3}_{ldr}, ProgramB\rangle\} \uplus C\rangle$

Where $\text{ISA}(ProgramB(1)) = ops_{ldr}$

$= \langle \text{True}\rangle(lock_{r1} \cdot lock_{r3} \cdot R_{r3} \cdot R_{mem[r3]} \cdot W_{r1} \cdot unlock_{r3} \cdot unlock_{r1} : \varepsilon)$

$po_3 = po_2 \oplus (\{(2, R_{r3}), (2, R_{mem[r3]}), (2, W_{r1})\},$

$\{(2, R_{r3}) \prec_{po} (2, R_{mem[r3]}), (2, R_{mem[r3]}) \prec_{po} (2, W_{r1})\})$

$= \langle hw, \langle to_3, po_3, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, ops^{t_3}_{ldr}, ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, (\langle \text{True}\rangle ops' : \varepsilon), ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, ops', ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_3\rangle, \emptyset, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, E[lock_{r1}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_3\rangle, \{ent_{r1}\}, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_3\rangle, \{ent_{r1}\}, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, E[lock_{r3}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_3\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_3\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma^2_2, \emptyset, \emptyset, E[R_{r3}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_3\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma^2_2, \{R_{r3}\}, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma^2_2[tmp_{r3} \mapsto Adr], \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

Where $xo_4 = xo_3 \oplus (\{(2, 2, R_{r3})\}, \emptyset)$

$= \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma^2_3, \emptyset, \emptyset, E[R_{mem[r3]}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_3^2, \{R_{mem[r3]}\}, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_3^2[data \mapsto 1], \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_4^2, \emptyset, \emptyset, E[W_{r1}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_4\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_4^2, \{W_{r1}\}, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_5\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_4^2[r1 \mapsto 1], \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

Where $xo_5 = xo_4 \oplus (\{(2, 2, W_{r1})\}, \emptyset)$

$= \langle hw, \langle to_3, po_3, xo_5\rangle, \{ent_{r1}, ent_{r3}\}, N_1, \{\langle 2, \sigma_5^2, \emptyset, \emptyset, E[unlock_{r3}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_5\rangle, \{ent_{r1}\}, N_1, \{\langle 2, \sigma_5^2, \emptyset, \emptyset, E[\varepsilon], ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_5\rangle, \{ent_{r1}\}, N_1, \{\langle 2, \sigma_5^2, \emptyset, \emptyset, E[unlock_{r1}], ProgramB\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_5\rangle, \emptyset, N_1, \{\langle 2, \sigma_5^2, \emptyset, \emptyset, \varepsilon, ProgramB\rangle\} \uplus C\rangle$

$= \langle hw, \langle to_3, po_3, xo_5\rangle, \emptyset, \{(1, 1, W_{mem[r3]})\}, \{\langle 1, \sigma_{t_2'''}^1, \emptyset, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$

$\rightarrow \langle hw, \langle to_3, po_3, xo_6\rangle, \emptyset, \emptyset, \{\langle 1, \sigma_{t_2'''}^1[\mathrm{Mem}[Adr] \mapsto 1], \emptyset, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$

Where $xo_6 = xo_5 \oplus (\{(1, 1, W_{mem[r3]})\}, \emptyset)$

$= \langle hw, \langle to_3, po_3, xo_6\rangle, \emptyset, \emptyset, \{\langle 1, \sigma_3^1, \emptyset, \emptyset, \varepsilon, ProgramA\rangle\} \uplus C\rangle$

Chapter 4

Verification method for multi-core systems

It's so difficult to verify the operating system because the operating system is a complex system which has a lot of functionalities. Since this research focuses on the out-of-order execution which is affect the correctness of the programs executed in multi-core system. This research will take *multiprocessor programs* into account. Moreover, to adopt our formal system in the verification, the *multiprocessor programs* will be complied as *assembly* language which is close to the hardware. However, as we see the sample execution in the previous section, the computation is too long for a simple program. In addition, the execution of instruction in the programs will not be atomic. Moreover, the memory accesses issued from the processor are able to be executed out-of-order. Thus, this section will propose a verification method for multiprocessor programs.

4.1 Verification of multiprocessor programs based on our semantics

Normally the execution of our semantics is change the state as step by step. Hence, we adopt the *induction* for providing a verification method. To verify the correctness of the program, we usually consider the required properties of the programs for ensuring the correctness. For example, in the *mutex lock* program, it provides the *safety properties* that is "There is no more one process can enter the critical section simultaneously".

To use induction in the verification, the properties of programs should be formalized as *invariant*. To ensure the correctness of programs, each state of execution should hold the invariant in each step of execution. For the induction proofs, first, we have to show the initial state holds the invariant. Then, the induction step will verify every arbitrary states to hold the invariant. Thus, to verify the programs based on our semantics, we have to show:

- Base case: $INV(s_{init})$
- Induction step: $\forall s, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s'))$

```
1 MOV  r1,#1
2 CMP  r1,#1
3 LDREQ  r2,[r3]
```

Figure 4-1: A sample program

initial program		$(\tilde{s}_{init})$
MOV	$r1, \#1$	$(\tilde{s}_{beforeLDR})$
CMP	$r1, \#1$	
LDREQ	$r2, [r3]$	$(\tilde{s}_{executingLDR})$

Figure 4-2: Set of states of programs 4-1

The s_{init}, s and s' are the configurations,$\langle hw, \Phi, locked, N, C\rangle$, based on our semantics. The INV represents the invariant which should be held for each arbitrary states. The 's_{init}' is an initial state of the verification. However, the initial state may not be defined concretely, because we have the arbitrary configuration of hw. So, we will introduce *a set of states* for verification. The set of states will be denoted as $\tilde{s}$. The set of state that refer to initial state will be defined as,

$$\begin{aligned}\tilde{s}_{init} = \{&\langle hw, (\langle(\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset)\rangle), \emptyset, \emptyset, Cores_{init}\rangle \mid hw \in HW \wedge \\ &\sigma_{init} = \{loc \to 0 \mid loc \in \text{Location}\}[pc \mapsto 1] \wedge \\ &\forall c \in Cores_{init}, pid \in ID, p \in 2^{Val \to Instr}.(c = \langle pid, \sigma_{init}, \emptyset, \emptyset, \varepsilon, p\rangle)\}\end{aligned}$$

Therefore, to show the based case, we have to show $INV(\tilde{s}_{init})$ holds.

Moreover, since the properties of the programs usually consider the execution of a program. To show the invariant hold the arbitrary states in the induction step, we assume that the properties usually concern about the steps that appeared to the program. In addition, the arbitrary state is able to consider the next state as *non-determinism*. Hence, for a single program, we define the *sets of states* to identify each set of states should holds the invariant.

Let's consider the program on figure 4-1, assume that the properties of this program will ensure the instruction 'LDREQ r2,[r3]' is always executed in this program, the value of program status register Z always be 1 at line 3. Assume that we already have invariant for verify the execution of this program. The invariant will ensure that the read access of 'LDREQ r2,[r3]', which access the location 'Z', always return 1 as a result. Hence, to consider the induction step, we group the arbitrary states as $\tilde{s}_{init}, \tilde{s}_{beforeLDR}$, and $\tilde{s}_{executingLDR}$. These sets of states are attached to the program as figure 4-2 . However, these states are considered in *a single program*. Hence, in the verification, we indicate sets of states for specific program as $\tilde{s}_{beforeLDR}(i)$ and $\tilde{s}_{executingLDR}(i)$ for the program in processor i. Even if the another programs are executed based on our semantics, we also consider such states as arbitrary states inside our defined states.

As induction proof, we have to show that $\forall s, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s'))$ as usual. The verification will consider the invariant based on executions of a single program. So, we split the induction step on processor i by using the *sets of states* as:

$$\forall s \in \tilde{s}_{init}, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \tag{4.1}$$
$$\forall s \in \tilde{s}_{beforeLDR}(i), s'.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \tag{4.2}$$
$$\forall s \in \tilde{s}_{executingLDR}(i), s'.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \tag{4.3}$$

To prove each case, we have to consider the rules that we have proposed in previous chapter. The rules which are able to apply to the source state will be considered the next state. For the case (4.1),'$\forall s \in \tilde{s}_{init}, s'.(INV(s) \wedge s \rightarrow s' \implies INVs')$', the possible rules which can apply are Rule 1 and Rule 2. For every $s \in \tilde{s}_{init}$, we consider the next set of states that apply rules Rule 1 and Rule 2 as,

$$\begin{aligned}
\tilde{s}_{Rule(1)} = \{\langle hw, \Phi', \emptyset, \emptyset, \{\langle i, \sigma[pc \mapsto 2], \emptyset, \emptyset, ops^t, p\rangle\} \uplus C\rangle \mid \\
\Phi' = \langle(\{t\}, \emptyset), po', (\emptyset, \emptyset)\rangle \wedge (\mathrm{ISA}(p(\sigma(\mathrm{pc}))) = ops) \wedge i \in \mathrm{ID}\} \\
\tilde{s}_{Rule(2)} = \{\langle hw, \Phi', \emptyset, \emptyset, \{\langle i, \sigma[pc \mapsto 2], \emptyset, \{(t, ops)\}, \varepsilon, p\rangle\} \uplus C\rangle \mid \\
\Phi' = \langle(\{t\}, \emptyset), po', (\emptyset, \emptyset)\rangle \wedge (\mathrm{ISA}(p(\sigma(\mathrm{pc}))) = ops) \wedge i \in \mathrm{ID}\}
\end{aligned}$$

Therefore, to prove the case (4.1), we will consider the sub-cases,

$$\forall s \in \tilde{s}_{init}, s' \in \tilde{s}_{Rule(1)}.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \tag{4.1-1}$$
$$\forall s \in \tilde{s}_{init}, s' \in \tilde{s}_{Rule(2)}.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \tag{4.1-2}$$

Moreover, to finish another cases, we will consider the possible rules which can apply to $\tilde{s}_{beforeLDR}$ for proving case (4.2) and $\tilde{s}_{executingLDR}$ for proving case (4.3).

To summarize, this section has proposed the idea to deal with our semantics for verification. In the next section we will propose the verification method as formal for the verification for multi-core systems. Then, we will show the verification for *mutex lock programs* as a case study.

4.2 Verification method

This section provides our verification method to verify programs for multi-core processor. However, the method is not provided as *systematically* because of the context of each programs to be verified might be different. Hence, the method is provided like a guidance to provide proofs. The following subsection will describe the steps of the method.

4.2.1 Provide *invariant* as the predicate

First of all, the *invariant*,$INV(s)$, should be formalized for ensuring the correctness. Since the properties are described as natural languages, we must formalize those properties to be ensured in our semantics.

By a given state, $\langle hw, \Phi, locked, N, C \rangle$, we are allowed to use these information to consider the *invariant* relate to the properties. Moreover, the syntax of our semantics also provide the function timePC to indicate the line of instruction in the programs. This might be useful to indicate the significant instructions for checking the correctness of the program. However, the formalization of properties should be manually formalized.

4.2.2 Define the sets of states

The sets of states are the states to be verified in the induction proofs. These sets will be used to split the case in induction step for induction proofs. In order to define the *sets of state*, we provide the mechanisms as followed:

Step 1 Firstly, we should identify the *significant* states. These states usually consider the *cutting-points* which consider the loop of the program. Some of the states might be determined by *defined invariant*. For example, the sets of states $\tilde{s}_{executingLDR}$, which describe in programs 4-2, is determined by the requirement of that program.

Step 2 Propose the intermediate states for the two states which cannot reachable another state by a single step. Note that, each states should be reachable by initial state, $\tilde{s}_{init}$. For example, in the program as figure 4-2, the set of states $\tilde{s}_{beforeLDR}$ has been introduced for connecting $\tilde{s}_{init}$ and $\tilde{s}_{executinhLDR}$.

Then, these proposed sets of states, included $\tilde{\mathrm{s}}_{\mathrm{init}}$, will be considered in the induction proof for case splitting. Assume that, from this step, we will get the $(n+1)$ sets of states as: $\tilde{s}_{init}, \tilde{s}_1, \tilde{s}_2, ..., \tilde{s}_n$.

4.2.3 Provide the induction proofs

To provide the induction proof on process i, we have to show:

- Base case: $\forall s \in \tilde{s}_{init}.INV(s)$
- Induction step: $\forall \tilde{s} \in \{\tilde{s}_{init}, \tilde{s}_1(i), \tilde{s}_2(i), ..., \tilde{s}_n(i)\}.\forall s \in \tilde{s}, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s'))$

The most difficult proof is the *induction step*. We have seen that the number of cases is relate to the number of sets of states.Moreover, each case will be split based on the possible rules that can apply to an arbitrary state in the source set of states. In some cases, the proof might be too difficult, because of the formal model complexity. Since the our formal model provides many semantics to represent the hardware behaviors, the verification should cover such behaviors to ensure the correctness. As we consider $s \rightarrow s'$, from the one arbitrary state, the number of transitions is depend on the semantics that can be applied to that state. For the initial state, only *fetch behaviors* are taken into account. Nevertheless, for another arbitrary state, the semantics that can be applied might be too much because our semantics are considered as non-determinism.

Chapter 5

A case study : a mutex lock verification

This section provides a case study that we verify the correctness of multiprocessor programs for multi-core systems. The programs that we consider as case study is *mutual exclusion programs*. This programs facilitate the mutual exclusion of the process among cores. We assume that the implementation of mutual exclusion will be as a code segment of the program. Hence, we define a segment of the program for facilitate mutual exclusion as,

```
    ...
α : MOV  r2, #1          (Acquire lock)
    SWP  r1, r2, [r0]
    CMP  r1, r2
    BNE  # − 3
β : ...                  (critical section)
    ...
γ : MOV  r2, #0          (Release lock)
    STR  r2, [r0]
    ...
```

In this program, the location r0 will be the location of the locked variable. The line α indicates the fist instruction of *acquire lock* mechanism. This mechanism will try to lock by using SWP instruction to acquire the lock variable in shared-memory location. The line β indicates the first line of *critical section*. The line γ indicates the first instruction of release lock mechanisms.

As for the properties, the mutual exclusion programs usually confirm that there are not more one process can be enter the critical section simultaneously. Basically, we should consider the fetched instructions at every execution that should not have more than one process are executing in the critical section. To consider more about memory operations, we should ensure that the memory operations which issued from one processor can not

be interrupted by memory operations from instructions in another processors. Therefore, we provide the *safety properties* to be ensured as

Safe$_1$: more than one process can never enter their critical section simultaneously, and

Safe$_2$: all read and write operations from instructions in one's critical section never be interrupted by another write accesses from another critical section.

The second property, Safe$_2$, is introduced because the fact that memory operations might return the result to its processor, but those operations maybe not completed, yet. Moreover, the order of executions is the significant issue to consider. Hence, we also should consider the second properties.

5.1 Apply verification method

First of all, we will propose an invariant to ensure the correctness of the program. The properties Safe$_1$ and Safe$_2$ will be formalized as $Inv_1(s)$ and $Inv_2(s)$ as followed,

$$
\begin{aligned}
Inv_1(s) &\iff \forall i, j \in ID.((i \neq j) \implies \neg(isEnterCrit(i,s) \wedge isEnterCrit(j,s))) \\
Inv_2(s) &\iff \forall rw_1, rw_2, rw' \in CritRW. \forall i, j \in ID.((i \neq j) \wedge \\
&\quad ((i, rw_1), (i, rw_2), (j, rw') \in S_{po}) \wedge (rw_1 \prec_{po} rw_2) \implies \\
&\quad \neg((rw_1 \prec_{xo} rw') \wedge (rw' \prec_{xo} rw_2)))
\end{aligned}
$$

Where

$$
\begin{aligned}
s &= \langle hw, \langle to, po, xo \rangle, locked, N, C \rangle \\
to &= (S_{to}, \prec_{to}) \\
po &= (S_{po}, \prec_{po}) \\
xo &= (S_{xo}, \prec_{xo}) \\
isEnterCrit(i,s) &= ((pc(i,s) >= \beta + 1 \wedge pc(i,s) <= \gamma) \\
\sigma(pc) &= pc(pid, \langle hw, \Phi, locked, N, \{\langle pid, \sigma, B, L, exec, p \rangle\} \uplus C \rangle) \\
CritRW &= \{rw^t \mid (timePC(t) = pc) \wedge (pc \in CritPC)\} \\
CritPC &= \{pc \in PC \mid (pc >= \beta) \wedge (pc <= \gamma)\}
\end{aligned}
$$

The invariant for the proofs is

$$INV(s) \iff Inv_1(s) \wedge Inv_2(s)$$

Then, we should provide the sets of states to identify the significant states to be considered. As we consider the *cut-points*, we will get the sets $\tilde{s}_{loop}$ and $\tilde{s}_{branch}$. Moreover, once we consider the *invariant*, the sets that we should consider are $\tilde{s}_{crit}$, $\tilde{s}_{try-release}$ and $\tilde{s}_{release}$. Next we will provide the remain sets to connect each step of execution. Therefore the

sets of states for the code segment as:

$$
\begin{array}{lr}
 & //initial \quad (\tilde{s}_{init}) \\
 & ... \quad (\tilde{s}_{1-\alpha}) \\
\alpha : & \text{MOV r2, \#1} \quad (\tilde{s}_{loop}) \\
 & \text{SWP r1, r2, [r0]} \quad (\tilde{s}_{\alpha-\beta}) \\
 & \text{CMP r1, r2} \\
 & \text{BNE \# - 3} \quad (\tilde{s}_{branch}) \\
\beta : & ... \quad (\tilde{s}_{crit}) \\
 & ... \quad (\tilde{s}_{\beta-\gamma}) \\
\gamma : & \text{MOV r2, \#0} \quad (\tilde{s}_{try-release}) \\
 & \text{STR r2, [r0]} \quad (\tilde{s}_{\gamma-realease}) \\
 & ... \quad (\tilde{s}_{release})
\end{array}
$$

Note that, the descriptions of the significant sets of states are:

- $\tilde{s}_{loop}$: The states that already fetch the instruction at line α. It can be either put the instruction into *instruction buffer* or directly put into the *execution unit*.
- $\tilde{s}_{branch}$: The states which have decided the next instruction should be at line α or β. That means these states are already to fetch the next instruction.
- $\tilde{s}_{crit}$: The states that already fetch the first instruction in critical section.
- $\tilde{s}_{try-release}$: The states that already fetch the instruction at line γ.
- $\tilde{s}_{release}$: The states that finish execute the instruction at line $\gamma + 1$.

As for the induction proofs, we will show:

- Based case: $\forall s \in \tilde{s}_{init}.INV(s)$
- Induction step : $\forall \tilde{s}.\forall s \in \tilde{s}, s'.(INV(s) \wedge s \to s' \implies INV(s'))$

As for the proofs of *induction step*, we split the case as:

$$
\begin{aligned}
\forall s \in \tilde{s}_{init}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{1-\alpha}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{loop}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{\alpha-\beta}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{branch}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{crit}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{\beta-\gamma}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{try-release}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{\gamma-release}, s'.(INV(s) \wedge s \to s' &\implies INV(s')) \\
\forall s \in \tilde{s}_{release}, s'.(INV(s) \wedge s \to s' &\implies INV(s'))
\end{aligned}
$$

Since, it's too difficult to provide the all of proofs by manual proof. In the next section, we will choose some cases to show the proof and idea to provide a proof for some cases.

5.2 Proofs

In this section will show the proofs of *mutual exclusion programs* for multi-core processors. However, we will show some proofs to show that our verification method can deal with the programs executed in multi-core processors. As for induction proof, we will show that,

$$\forall s \in \tilde{s}_{init}.INV(s) \qquad \text{(based case)}$$
$$\forall \tilde{s}, s \in \tilde{s}, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s')) \qquad \text{(Induction case)}$$

Based case

We define $\tilde{s}_{init}$ as

$$\tilde{s}_{init} = \{\langle hw, \langle(\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset)\rangle, \emptyset, \emptyset, Cores_{init}\rangle \mid hw \in \text{HW}\}$$

In order to consider the mutual exclusion properties, consider only 2 programs which each program consists of a code segment of mutual exclusion program. So, the definition of $Cores_{init}$ is defined as,

$$Cores_{init} = \langle 1, \sigma_{init}, \emptyset, \emptyset, \varepsilon, P_{mutex}\rangle \cdot \langle 2, \sigma_{init}, \emptyset, \emptyset, \varepsilon, P_{mutex}\rangle$$

where $\sigma_{init} = \{loc \rightarrow 0 \mid loc \in \text{Location}\}[pc \mapsto 1]$. The proofs of based cases will show that for any arbitrary $s \in \tilde{s}_{init}$, the invariant $INV(s)$ holds.

$$\begin{aligned}
INV(s) &\iff Inv_1(s) \wedge Inv_2(s) && \text{(Extract the invariant)}\\
&\iff \forall i,j \in ID.((i \neq j) \implies \\
&\quad \neg(isEnterCrit(i,s) \wedge isEnterCrit(j,s))) \wedge Inv_2(s) && \text{(Consider } Inv_1\text{)}\\
&\iff ((\mathbf{1} \neq \mathbf{2}) \implies \\
&\quad \neg(isEnterCrit(\mathbf{1},s) \wedge isEnterCrit(\mathbf{2},s))) \wedge Inv_2(s) \\
& && \text{(Initiate } i=1, j=2\text{)}\\
&\iff ((\text{True}) \implies \\
&\quad \neg(isEnterCrit(1,s) \wedge isEnterCrit(2,s))) \wedge Inv_2(s)\\
&\iff (\neg(isEnterCrit(1,s) \wedge isEnterCrit(2,s))) \wedge Inv_2(s)\\
&\iff (\neg(((pc(1,s) >= \beta+1 \wedge pc(1,s) <= \gamma) \wedge isEnterCrit(2,s)))\wedge \\
&\quad Inv_2(s) && \text{(Extract the } isEnterCrit(1,s)\text{)}\\
&\iff (\neg(((1 >= \beta+1 \wedge 1 <= \gamma) \wedge isEnterCrit(2,s))) \wedge Inv_2(s) \\
& && \text{(The initial pc is 1)}\\
&\iff (\neg(((\text{False} \wedge \text{False}) \wedge isEnterCrit(2,s))) \wedge Inv_2(s)\\
&\iff (\neg((\text{False} \wedge isEnterCrit(2,s))) \wedge Inv_2(s)\\
&\iff \neg(\text{False}) \wedge Inv_2(s)\\
&\iff Inv_2(s)
\end{aligned}$$

$$
\begin{aligned}
\Longleftrightarrow\ & \forall rw_1, rw_2, rw' \in CritRW. \forall i, j \in ID. ((i \neq j) \wedge \\
& ((i, rw_1), (i, rw_2), (j, rw') \in S_{po}) \wedge (rw_1 \prec_{po} rw_2) \implies \\
& \neg((rw_1 \prec_{xo} rw') \wedge (rw' \prec_{xo} rw_2))) \qquad \text{(Consider the invariant } Inv_2\text{)} \\
\Longleftrightarrow\ & \forall rw_1, rw_2, rw' \in CritRW. \forall i, j \in ID. ((i \neq j) \wedge (\text{False}) \wedge (\text{False}) \implies \\
& \neg((\text{False}) \wedge (\text{False}))) \qquad \text{(There is no execution, yet)} \\
\Longleftrightarrow\ & \forall rw_1, rw_2, rw' \in CritRW. \forall i, j \in ID. (\text{False} \implies \text{True}) \\
\Longleftrightarrow\ & \text{True} \qquad \square
\end{aligned}
$$

Induction case

As for induction step, we have to show

$$\forall \tilde{s}, s \in \tilde{s}, s'. (INV(s) \wedge s \to s' \implies INV(s'))$$

As we consider before, we will split this case by the *sets of states* defined before. Therefore the cases that we will consider are:

$$\forall s \in \tilde{s}_{init}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.1}$$
$$\forall s \in \tilde{s}_{1-\alpha}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.2}$$
$$\forall s \in \tilde{s}_{loop}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.3}$$
$$\forall s \in \tilde{s}_{\alpha-\beta}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.4}$$
$$\forall s \in \tilde{s}_{branch}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.5}$$
$$\forall s \in \tilde{s}_{crit}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.6}$$
$$\forall s \in \tilde{s}_{\beta-\gamma}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.7}$$
$$\forall s \in \tilde{s}_{try-release}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.8}$$
$$\forall s \in \tilde{s}_{\gamma-release}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.9}$$
$$\forall s \in \tilde{s}_{release}, s'. (INV(s) \wedge s \to s' \implies INV(s')) \tag{5.10}$$

For each cases, we should consider the possible rules, which are defined in section 3.2.3, that can be applied to an arbitrary state. Since the number of rules is 18 and the number of cases to be verified is 10, the proofs of induction cases might be so difficult for the manual proof. Thus, we just show some proofs and the idea to prove each case in this section.

Case (5.1) $\forall s_{init} \in \tilde{s}_{init}, s'.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies INV(s'))$
Let's consider the *possible rules* which can be applied to any arbitrary state $s \in \tilde{s}_{init}$. Thus, the possible rules are:

- Rule 1. Fetching in-order
  After we apply the rule, there are many concrete states that next state(s') can be. Hence, we define a set of states which refers to fetched states as

$$\tilde{s}_{fetch1} = \{s \mid s = \langle hw, \Phi', \emptyset, \emptyset, \{\langle pid, \sigma[pc \mapsto 2], \emptyset, \emptyset, \mathrm{ISA}^t(P_{mutex}(pid)), P_{mutex}\rangle\} \uplus C\rangle \wedge \\ \Phi' = \langle(\{t\}, \emptyset), po', (\emptyset, \emptyset)\rangle \wedge pid \in \mathrm{ID}\}$$

  The states in the set $\tilde{s}_{fetch1}$ have chosen one core to *fetch* an instruction for execution.
- Rule 2. Fetching out-of-order (put instruction into the instruction buffer)
  This rule also able to produce many possible states (s'). We also define a set of states which refers to *out-of-order* fetched states as

$$\tilde{s}_{fetch2} = \{s \mid s = \langle hw, \Phi', \emptyset, \emptyset, \{\langle pid, \sigma[pc \mapsto 2], \emptyset, \{(t, \mathrm{ISA}^t(P_{mutex}(1)))\}, \varepsilon, P_{mutex}\rangle\} \uplus C\rangle \\ \wedge \Phi' = \langle(\{t\}, \emptyset), po', (\emptyset, \emptyset)\rangle \wedge pid \in \mathrm{ID}\}$$

Hence, we have to show that

$$\forall s_{init} \in \tilde{s}_{init}, s' \in \tilde{s}_{fetch1} \cup \tilde{s}_{fetch2}(INV(s_{init}) \wedge (s_{init} \rightarrow s') \implies INV(s'))$$

So, It's better to split the case to be

$$\forall s_{init} \in \tilde{s}_{init}, s' \in \tilde{s}_{fetch1}(INV(s_{init}) \wedge (s_{init} \rightarrow s') \implies INV(s')) \quad (5.11)$$
$$\forall s_{init} \in \tilde{s}_{init}, s' \in \tilde{s}_{fetch2}(INV(s_{init}) \wedge (s_{init} \rightarrow s') \implies INV(s')) \quad (5.12)$$

First of all, for any arbitrary s_{init}, let's show the case (5.11) as

$$\begin{aligned}
& \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies INV(s')) \\
\iff & \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \boldsymbol{Inv_1(s')} \wedge Inv_2(s')) \\
\iff & \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \\
& \forall i, j \in \mathrm{ID}.((i \neq j) \implies \neg(\boldsymbol{isEnterCrit(i, s')} \wedge isEnterCrit(j, s'))) \\
& \wedge Inv_2(s')) \\
\iff & \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \forall i, j \in \mathrm{ID}.((i \neq j) \implies \neg(\\
& (\boldsymbol{pc(i, s') >= \beta + 1}) \wedge (pc(i, s') <= \gamma)) \wedge isEnterCrit(j, s'))) \\
& \wedge Inv_2(s'))
\end{aligned}$$

We know that $pc(i, s') = \begin{cases} 2 & \text{if } i = pid \\ 1 & \text{if } i \neq pid \end{cases}$,for all $s' \in \tilde{s}_{fetch1}$.
According to the program, we also know that $1 <= \alpha$ and $\alpha + 4 = \beta$.

Hence $(pc(i,s') > \beta + 1) \iff$ False, for all $i \in$ ID

$$
\begin{aligned}
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \forall i,j \in \text{ID}.((i \neq j) \implies \neg(\\
&\quad (\textbf{False}) \wedge (pc(i,s') <= \gamma)) \wedge isEnterCrit(j,s'))) \\
&\quad \wedge Inv_2(s')) \\
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \forall i,j \in ID.((i \neq j) \implies \neg(\\
&\quad (\textbf{False}) \wedge isEnterCrit(j,s'))) \\
&\quad \wedge Inv_2(s')) \\
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \textbf{True} \wedge Inv_2(s')) \\
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \boldsymbol{Inv_2(s')}) \\
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \\
&\quad \forall rw_1, rw_2, rw' \in CritRW.\forall i,j \in \text{ID}.((i \neq j) \wedge \\
&\quad ((i,rw_1),(i,rw_2),(j,rw') \in S_{po}) \wedge (rw_1 \prec_{po} rw_2) \implies \\
&\quad \neg((rw_1 \prec_{xo} rw') \wedge (rw' \prec_{xo} rw_2)))) \\
&\iff \forall s' \in \tilde{s}_{fetch1}.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies \\
&\quad \forall rw_1, rw_2, rw' \in CritRW.\forall i,j \in \text{ID}.((i \neq j) \wedge \\
&\quad ((i,rw_1),(i,rw_2),(j,rw') \in S_{po}) \wedge (rw_1 \prec_{po} rw_2) \implies \\
&\quad \neg(\textbf{False})) \qquad \text{(No execution)} \\
&\iff \text{True} \qquad \square
\end{aligned}
$$

For the case (5.12), It will provide a proof in the same way. Because, the proof considers only **pc's value** and **executions**. Hence, we can conclude that

$$\forall s_{init} \in \tilde{s}_{init}, s'.(INV(s_{init}) \wedge s_{init} \rightarrow s' \implies INV(s')) \quad \square$$

Sketch proof for case (5.2)

$$\forall s \in \tilde{s}_{1-\alpha}, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s'))$$

First of all, we need to indicate the target processor for verification. As for the processor i, we denote the states $\tilde{s}_{1-\alpha}$ and $\tilde{s}_{loop}$ of the program which belong to processor i as $\tilde{s}_{1-\alpha}(i)$ and $\tilde{s}_{loop}(i)$. Then, we define the definition of $\tilde{s}_{1-\alpha}$ and $\tilde{s}_{loop}$ as:

$$\begin{aligned}
\tilde{s}_{1-\alpha}(i) = \{&\langle hw, \Phi, locked, N, \{\langle i, \sigma, B, L, exec, \mathrm{P}_{mutex}\rangle\} \uplus C\rangle \mid (\sigma(pc) \leq \alpha)\} \\
\tilde{s}_{loop}(i) = \{&\langle hw, \langle to \oplus (t, \emptyset), po.xo\rangle, locked, N, \{\langle i, \sigma, B, L, exec, \mathrm{P}_{mutex}\rangle\} \uplus C\rangle \mid \\
&(\sigma(pc) = \alpha + 1) \text{ and } (ops = \mathrm{ISA}(\mathrm{P}_{mutex}(\alpha))) \text{ and } (\\
&(\neg(\mathrm{inOrderIssue}(hw)) \implies ((t, ops) \in L)) \text{ or} \\
&((\mathrm{inOrderIssue}(hw)) \implies exec = (exec' \cdot ops)))\}
\end{aligned}$$

Then, let's consider the *possible rules* which can be applied to any arbitrary state $s \in \tilde{s}_{1-\alpha}(i)$, for all $i \in \mathrm{ID}$. In this case, all 18 rules can be applied to arbitrary states $s \in \tilde{s}_{1-\alpha}(i)$. So, we will group the possible next state, $s \rightarrow s'$, as follow:

- $s' \in \tilde{s}_{loop}(i)$, this case is produced from the Rules 1 and 2 that apply to processor i.
- $s' \in \tilde{s}_{1-\alpha}(i)$, this case is produced form the remaining rules.

So, to proof the case (5.2), we will show

$$\forall s \in \tilde{s}_{1-\alpha}(i), s' \in \tilde{s}_{loop}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.2\text{-}1)$$
$$\forall s \in \tilde{s}_{1-\alpha}(i), s' \in \tilde{s}_{1-\alpha}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.2\text{-}2)$$

Let's consider the case (5.2-1), the Rules 1 and 2 can be applied to the arbitrary state s, where $s = \langle hw, \langle to, po, xo\rangle, locked, N, \{\langle i, \sigma, B, L, exec, \mathrm{P}_{mutex}\rangle\} \uplus C\rangle$, if and only if the state s satisfies the rule's condition. The applied states s that satisfy the rules 1 and 2 are defined as follow, respectively.

$$\begin{aligned}
\tilde{s}_{fetch1}(i) = \{&\langle hw, \Phi', locked, N, \{\langle i, \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1], B, L, exec \cdot ops^t, P_{mutex}\rangle\} \uplus C\rangle \mid \\
&\Phi' = \langle to \oplus (\{t\}, \emptyset), po', xo\rangle \text{ and } ops = \mathrm{ISA}(\mathrm{P}_{mutex}(i))\} \\
\tilde{s}_{fetch2}(i) = \{&\langle hw, \Phi', locked, N, \{\langle i, \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1], B, L \cup \{(t, ops)\}, exec, P_{mutex}\rangle\} \uplus C\rangle \mid \\
&\Phi' = \langle to \oplus (\{t\}, \emptyset), po', xo\rangle \text{ and } ops = \mathrm{ISA}(\mathrm{P}_{mutex}(i))\}
\end{aligned}$$

Therefore, to conclude the case (5.2-1), we will show the cases below:

$$\forall s \in \tilde{s}_{1-\alpha}(i), s' \in \tilde{s}_{loop}(i) \cap \tilde{s}_{fetch1}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.2\text{-}1\text{-}1)$$
$$\forall s \in \tilde{s}_{1-\alpha}(i), s' \in \tilde{s}_{loop}(i) \cap \tilde{s}_{fetch2}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.2\text{-}1\text{-}2)$$

As for the case (5.2-2), we have to consider all possible rules, 18 rules for our semantics, to produce the possible next states like $\tilde{s}_{fetch1}(i)$ and $\tilde{s}_{fetch2}(i)$.

By intuition, the states in the sets $\tilde{s}_{fetch1}(i)$ and $\tilde{s}_{fetch2}(i)$ is obviously hold the invariant. Because, in this induction proof, we consider only 2 processors and this processor i is not enter the critical section, yet.

Sketch proof for case (5.5)

$$\forall s \in \tilde{s}_{branch}, s'.(INV(s) \wedge s \rightarrow s' \implies INV(s'))$$

Firstly, we will define $\tilde{s}_{branch}$ as

$$t\tilde{s}_{branch}(i) = \{\langle hw, \Phi, locked, N, \langle i, \sigma, B, L, exec, \mathrm{P}_{mutex}\rangle\rangle \mid (\sigma(\mathrm{pc}) = \beta)\} \cap (\tilde{s}_{cRule_1}(i) \cup \tilde{s}_{cRule_2}(i))$$

This set will represent the states that are ready to fetch the next instruction in the next step. Then, we will consider the possible states as s', where $s \in \tilde{s}_{branch}(i)$ and $s \rightarrow s'$. The next states of $s \in \tilde{s}_{branch}(i)$ will be a state in the set either $\tilde{s}_{branch}(i)$, $\tilde{s}_{crit}(i)$, or $\tilde{s}_{loop}(i)$. So, we have to show that

$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{branch}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}1)$$
$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{crit}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}2)$$
$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{loop}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}3)$$

As for (5.5-2) and (5.5-3), they allow only fetching rules to be applied to be either $s' \in \tilde{s}_{crit}(i)$ or $s' \in \tilde{s}_{loop}(i)$. In addition, to show (5.5-2) and (5.5-3), we need some atomicity properties of the systems. Since we need to ensure that only one processor can enter the critical region, the 'SWP' instruction at line $\alpha + 1$ should be able to keep the atomicity requirement, Condition 3.4, of the read and write operations. As we provide the condition atomicCond for the rule 6, we can ensure that the systems will hold the atomicity properties. Then, for cases (5.5-2) and (5.5-3), we will provide that set of states that satisfy the Rule 1 and Rule 2 as

$$\tilde{s}_{fetch1}(i) = \{\langle hw, \Phi', locked, N, \{\langle i, \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1], B, L, exec \cdot ops^t, P_{mutex}\rangle\} \uplus C\rangle \mid \Phi' = \langle to \oplus (\{t\}, \emptyset), po', xo\rangle \text{ and } ops = \mathrm{ISA}(\mathrm{P}_{mutex}(i))\}$$
$$\tilde{s}_{fetch2}(i) = \{\langle hw, \Phi', locked, N, \{\langle i, \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1], B, L \cup \{(t, ops)\}, exec, P_{mutex}\rangle\} \uplus C\rangle \mid \Phi' = \langle to \oplus (\{t\}, \emptyset), po', xo\rangle \text{ and } ops = \mathrm{ISA}(\mathrm{P}_{mutex}(i))\}$$

Then, to show the case (5.5-2) and (5.5-3), we need more 4 cases as

$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{crit}(i) \cap \tilde{s}_{fetch1}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}2\text{-}1)$$
$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{crit}(i) \cap \tilde{s}_{fetch2}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}2\text{-}2)$$
$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{loop}(i) \cap \tilde{s}_{fetch1}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}3\text{-}1)$$
$$\forall s \in \tilde{s}_{branch}(i), s' \in \tilde{s}_{loop}(i) \cap \tilde{s}_{fetch2}(i).(INV(s) \wedge s \rightarrow s' \implies INV(s')) \quad (5.5\text{-}3\text{-}2)$$

To show these cases, we consider any arbitrary s' in the specific sets. So, we can conclude the cases (5.5-2) and (5.5-3).

As for cases (5.5-1), this cases can apply all rules to consider the next state s'. So, we also need to verify all rules that can apply to $\tilde{s}_{branch}(i)$. So, we have to show more 18 cases to verify the case (5.5-1). By intuition, this case can be verified, because the next arbitrary state of this case is the element the same set which already hold invariant.

Chapter 6

Evaluation

This chapter will evaluate our proposed formal model and verification method. Since this research focuses on multi-core behavior issues, the verification of programs for multi-core systems will become complex. As we have proposed the formal model to represent multi-core systems in chapter 3, we also need evidences that the proposed formal model is applicable to capture the multi-core behaviors. Moreover, the verification of the programs for multi-core systems also need the method to ensure the correctness based on the proposed formal model. In the chapter 4, we have proposed the verification method as a way to verify the correctness based on our semantics. This method also need some evidences to convince that the method is applicable for verification.

6.1 Evaluation of our formal model

The formal model defined in section 3.2.3 has abstracted the behaviors of the multi-core systems for executing the programs. The formal model is represented as *operational semantics* which are appropriate to describe the behaviors of defined operations.

The significant behavior considered in this research is *out-of-order* execution. Since this behavior affects the possible results of programs among cores, we have provided an abstract model to represent this behaviors. In addition, the optimization mechanisms cause the changing order of memory operation. Thus, our formal model provides the rules to capture such optimization mechanisms and out-of-order execution. In addition, the memory consistency models are captured in the model by providing conditions to restrict execution behaviors for the rules. Therefore, we have proposed 18 rules to represent the behaviors for executing programs on multi-core systems.

The section 3.3 have show the execution of programs which are placed on each core separately. That section shows only one path of executions because our semantics provide non-determinism executions. Moreover, the degree of non-determinism is more than we consider the programs as an interleaving way. Thus, the execution of the programs becomes complicated. Nevertheless, such execution show that our formal model is *able to capture the out-of-order execution of the programs on multi-core systems.*

6.2 Evaluation of our verification method

The verification method proposed in chapter 4 is provided to capture the execution states of the programs. The proof of this method uses the induction to provide the proof. The proof will consider each arbitrary step of executions from the initial state. To ensure the correctness of the programs, we consider the requirements or properties of the programs as usual. Such requirements or properties will be formalized as *invariant* to hold for every arbitrary states that can be reached by the initial state. Thus, the proposed method provides a way to verify the invariant to hold on every execution steps of programs which can be reached from initial states.

The characteristic of the method considers the execution of a single program to split the cases in the induction steps for verification. We assume that the properties to be verified usually consider the some significant states during the execution of the program. In the method, we assume that those significant states can be group as *a set of states*. Moreover, the intermediate states also can be group as a set of states. Hence, this method will consider the induction on groups of states based on the execution of the program.

As we show in chapter 5, the verification method can be applied to a mutex lock verification. In the verification, we proposed the way to verify the safety properties of the programs. Since the complexity of our formal model, we didn't show all the proofs to cover the all executions of the program. As we consider the cases provided in the case study, in the induction step, we have to show about 10 cases in the verification. In addition, for each case, we have to take the rules which are proposed in the semantics, 18 rules, into account. For the simple cases, case (5.1), we provide about 10 lines to show the proof. The whole proof might be at least 1800 lines in the verification. Hence, we decide to provide some proofs to show the idea to apply the formal model to capture the behaviors of execution on the multi-core system. Although the proof is quite complex even if the programs contains a few instructions, the verification is *able to capture the states which are not appeared explicitly in the programs*. That means we can *realize the behaviors of hardware in the verification*.

Chapter 7

Related works

This section will consider the related works that we consider in this research. Since our target verification is the operating system, there are another OS verification which provide another approaches for ensure their correctness. In addition, as we have proposed the formal model to represent the execution of programs for multi-core system, there are another approaches to represent the multi-core system which also take the out-of-order execution into account. Moreover, there are verification methods which can be used for program verification. This section will explain their approach and provide originality of this research.

7.1 OS Verification

In the field of verification, there are related works that also take the operating systems into account. As the [Kle09] has survey operating system verification which use the formal verification, obviously the operating system is a complex systems and there are different level of the proof to ensure the correctness of the operating system. As for our work, we consider the low-level implementation to capture the out-of-order execution inside the hardware.

As our motivation come from verification of automotive systems, there are related works that also verify the automotive operation systems. In each work also provide the different approach to deal with its own system, such as [Cho, KAE+14, Cho13, YH11].

The [Cho, Cho13, YH11] use *model checking* approach to ensure the correctness of *TramplineOS* which is implemented based on *OSEK/VDX OS* specification. That work formalizes the kernel code of operating systems as a model, then the model checking tool is applied to ensure the properties based on that model. This approach automatically check all of the possible states which can be produced from the formal model. Nevertheless, it might have the state explosion problem as a limitation of this approach.

The [KAE+14] provides the verification of its own micro-kernel, named *seL4*. This work uses *theorem prover assistances* to deduction proofs as formal verification. Although this techniques require users to understand the underlying of the systems, it has no limitation of the verification like model checking.

7.2 Formalization

As the formal model is necessary to capture the behaviors of program execution, there are related works that formalize their target system which take memory consistency models into account, such as [AAS03, BP09, FM10, AFI+09].

The [BP09] has proposed the model that represent *weakly ordering* model. This model is proposed for high-level language such as java language. The model is used to capture the memory model provided by its language, such as java memory model (JMM).

The [FM10] formalizes the Instruction Set Architecture(ISA) of ARM processors as a formal model. This model is described followed the ARM manual document, [Lim08]. This model is quite concrete to represent the execution of each instruction in ARM processor. Since this model is validated by using *Random testing* approach with the practical system to conform their model do represent the real behaviors of instructions.

The [AAS03, AFI+09] have captured the out-of-execution behaviors by events. The related work [AAS03] have propose the model for PowerPC shared-memory architecture by using in-out operations. Such operations are used to capture the behaviors of information that flow in the systems. In contrast, the related work [AFI+09] has proposed the *axiomatic model* to consider the *valid execution* of the events. In this work, they map the program as events to consider by extend the instruction semantics from the [FM10] to capture the micro-operation inside the hardware.

7.3 Formal Verification

In order to ensure the correctness of the programs by using formal verification, there are related works that proposed method to verify the program, such as [Moo03].

As chapter 3 in the book [Man03] proposes an approach to verify the *partial correctness* of the program. In this method simply consider the program as flow-chart. Each step of execution will have pre and post-conditions. However, this approach is not appropriate for verify programs in multi-core systems because of out-of-order executions.

The [Moo03] has proposed a method to verify the programs which executing on *operational semantics*. This method will attach the assertions to the each cut-point, which are considered by loops. In addition, this method also consider the *induction proof* to consider the induction step of execution based on provided operational semantics. This method will extend the initial states as symbolic state produced by the rules in semantics. Once the state is match to the significant state that attach the assertion, such assertion should hold to ensure the correctness during execution.

7.4 Originality

This research focuses on the verification of operating systems for multi-core systems. The motivation of our research focus specifically on *multi-core operating systems*.The related works [Cho13, Cho] have provide the formal verification by using *Model checking*.

Nevertheless, those works didn't take the out-of-order execution into account because the multi-core systems is not taken into account in those operating systems, yet. Since the multi-core systems is taken into account in this research, Model checking approach might not be appropriate to ensure the correctness of the system. Thus, we focus on theorem proving to ensure the correctness of the programs for multi-core systems.

As for the proposed formal model, we use the *operational semantics* to describe the behaviors of program execution on the multi-core system. Our formal model is an abstraction of the multi-core system. We do capture only essences of program execution to eliminate the unnecessary behaviors for program verification, such as cache behaviors. Although the [FM10] is quite useful model to represent the ISA of ARMv7 architectures, in our research we focus to abstract such behaviors, not in detail, to provide program verification for multi-core system. In contrast to the [BP09] that provides the model to represent the weakly model to consider program in the high-level, this research considers the programs in low-level which memory consistency model affect the behaviors of hardware architecture.

The most similar works that propose the formal model for multi-core system are [AFI+09, AAS03]. [AFI+09] describes an instruction inside a program as events. The executions of programs are the paths of the events, which are produced form the programs. They consider the valid executions for the ARM processor and PowerPC processor by providing the *axiomatic models*. As for [AAS03], they have introduce the *view orders* appeared to each processor. They also provide the definition to constraints the *view orders* of each core to follow the memory consistency model of target system. In contrast, our formal model *capture the behavior of program execution step by step*. In each step of fetching instruction, they also produced micro-operations and memory operations to complete the instruction. Then, the semantics will control the execution by using conditions which adopt from [Gha95].

The related work [Moo03] has provided an appropriate method to verify the correctness of programs based on proposed *operational semantics*. The approach will construct the arbitrary state by extend the initial state based on operational semantics. However, the degree of non-determinism of program execution on multi-core might cause the arbitrary state become complicate to consider. Our verification method will consider the group of arbitrary states to be hold for proposed invariant. Such groups will be considered from the significant states as we have explained in chapter 4.

Chapter 8

Conclusions and Future works

In this research, we have proposed the *formal model* which represent the hardware behaviors to capture the program execution for the multi-core system. As we show in chapter 3, the proposed formal model as operational semantics is able to provide the execution of programs which can capture the out-of-order execution of memory operation. As for memory consistency models, we capture just only *sequential consistency model* and *partial consistency model*. Such models are adopt the constraints from [Gha95] to provide conditions in operational semantics.

Moreover, as we need to ensure the correctness of programs on the multi-core system, we proposed the verification method, as described in chapter 4, based on our formal model. As we have shown in section 5, we can use this method to verify the safety properties of the programs executing on multi-core system.

8.1 Discussion

8.1.1 Formal model

As the formal model has been proposed, we still have to consider whether the formal model really represent the multi-core behaviors. Although we abstract the behaviors from the related works and the execution of program can be produced as we show in section 3.3, the model still need another evidences to convince that it can correctly represent the program execution in multi-core behaviors.

8.1.2 Verification method

As the verification method has been proposed, we have shown our idea to verify the partial correctness of safety properties. However, we didn't finish the proof, yet, because of there are many cases which are generated based on the formal model. Thus, the automate tools are needed to help us in verification.

In addition, there are another requirements of the programs to be ensure, such as starvation problems. That means another properties should be taken into account, such as

liveness properties. The another formal verification, such as model checking, use *temporal logics* to represent the liveness properties and check them in the execution.

8.2 Future works

As we have discussed, the formal model and verification method, which we have proposed, should be improved to convince that our method is appropriate for program verification on multi-core systems.

8.2.1 Ensuring the proposed formal model

In order to ensure, first of all, we should adopt our approach to proof assistance tool. Although the formal model have been proposed, we also need the tools to check that our formal model is really correct implement.

In addition, to provide the evidences for the formal model, we should *validate* our model. As the related works [FM10] use the practical hardware to ensure their formal model of instruction set architecture by using random testing techniques. Therefore, we also need the validation techniques to ensure that our abstract model can realize the program execution behaviors on multi-core systems.

8.2.2 Improving the verification method

As we discuss about the verification method, we should provide a way to support another properties. As model checking is able to adopt the temporal logic for checking the correctness. In theorem proving, there are related work [OF99] that adapt the idea of liveness verification. Thus, such approach might be adopt to improve the verification method for program execution in multi-core system. Moreover, since the proof became large even if we consider the few instructions, it's better to apply the automate tools to provide proofs.

Bibliography

[AAS03] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the powerpc architecture. In *Journal IEEE Transactions on Parallel and Distributed Systems*, volume 14, pages 502–515, May 2003.

[AFI+09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *The 4th workshop on Declarative aspects of multicore programming*, pages 13–24, 2009.

[BP09] Gerard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *POPL'09*, pages 392–403, 2009.

[But] Ricky W. Butler. What is formal methods?

[Cho] Yunja Choi. Safety analysis of trampoline os using model checking: An experience report.

[Cho13] Yunja Choi. Model checking an osek/vdx-based operating system for automobile safety analysis. In *IEICE TRANSACTIONS on Information and Systems*, volume E96-D, pages 735–738, 2013.

[FM10] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *Interactive Theorem Proving*, volume 6172, pages 243–258, 2010.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for shared-memory multiprocessors*. PhD thesis, Stanford University, December 1995.

[KAE+14] Gerwin Klein, JUNE ANDRONICK, KEVIN ELPHINSTONE, TOBY MURRAY, THOMAS SEWELL, RAFAL KOLANSKI, and GERNOT HEISER. Comprehensive formal verification of an os microkernel. In *Journal ACM Transactions on Computer Systems (TOCS)*, volume 32. NICTA and UNSW, Sydney, Australia, February 2014.

[Kle09] Gerwin Klein. Operating system verification - an overview. In *Sadhana*, volume 34, pages 27–69, 2009.

[Lam79] LESLIE Lamport. How to make a multiprocessor computer that correctly execute multoprocess program. In *Journal IEEE Transactions on Computers*, volume 28, pages 690–691, September 1979.

[Lim08] ARM Limited. *ARM architecture reference manual: ARMv7-A and ARMv7-R edition*, 2008.

[Man03] Zohar Manna. *Mathematical Theory of computation.* Dover Publications, Inc., 2003.

[Moo03] J Strother Moore. Inductive assertions and operational semantics. In *Correct Hardware Design and Verification Methods*, 2003.

[OF99] Kazuhiro Ogata and Kokichi Futatsugi. Formal verification of the mcs list-based queuing lock. In *ASIAN '99 Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 281–293, 1999.

[YH11] Longfei Zhu Qin Li Huibiao Zhu Jianqi Shi Yanhong Huang, Yongxin Zhao. Modeling and verifying the code-level osek/vdx operating system with csp. In *IEEE International Conference on Theoretical Aspects of Software Engineering*, pages 142 – 149, 2011.