Formal Specification and Verification of Memory Consistency Models of Shared Memory Multiprocessors

by

Prosenjit Chatterjee

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

School of Computing

The University of Utah

March 2002

Copyright © Prosenjit Chatterjee 2002

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Prosenjit Chatterjee

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

____________ ____________________

Chair: Ganesh Gopalakrishnan

____________ ____________________

Gary Lindstrom

____________ ____________________

Michael Jones

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of The University of Utah:

I have read the thesis of Prosenjit Chatterjee in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to the Graduate School.

Date

Ganesh Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Weak shared memory consistency models, especially those used by modern microprocessor families, are quite complex. The bus and/or directory-based protocols that help realize shared memory multiprocessors using these microprocessors are also exceedingly complex. Thus, the *correctness problem* – that all the executions generated by the multiprocessor for any given concurrent program are also allowed by the memory model – is a major challenge. In this thesis, we present a formal approach to verify protocol implementation models against weak shared memory models through automatable *refinement checking* supported by a *model checker.* We define a taxonomy of weak shared memory models that includes most published commercial memory models, and detail how our approach applies over all these models. In our approach, the designer follows a prescribed procedure to build a highly simplified intermediate abstraction for the given implementation. The intermediate abstraction and the implementation are concurrently run using a model-checker, checking for refinement. The intermediate abstraction can be proved correct against the memory model specification using theorem proving. We have verified four different Alpha memory model implementations and four different[1]Itanium memory model implementations against their respective specifications. The results are encouraging in terms of the uniformity of the procedure, the high degree of automation, acceptable run-times, and empirically observed bug-hunting efficacy. The use of parallel model-checking, based on a version of the parallel Murϕ model checker we have recently developed for the MPI library,

[1]We designed both these protocols [1] - one with a split-transaction bus and one with Scheurich's optimization - as there are no public domain Itanium protocols as far as we know.

has been essential to finish the search in a matter of a few hours.

For our verification method to be applicable, the specification of the memory model must be described in terms of an identifiable set of events that have unique completion points. We thus introduce a new definitonal framework to specify memory models that fulfills the above mentioned requirement. In addition, given such a specification we propose an algorithm to generate a corresponding executable specification. This alternative specification can be used to generate all possible outcomes of small assembly-language multiprocessor programs in a given memory model, which is very helpful for understanding the subtleties of the model. The executable specification can also check the correctness of assembly language programs including synchronization routines.

To my dadu Pratul, my baba Prasanta, my ma Shukla and my wife Nandita

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **x**

LIST OF TABLES . . . . **xi**

ACKNOWLEDGMENTS . . . . **xii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Our Definitional Framework to Specify Memory Consistency Models 3
1.2 Generating Executable Specifications Automatically 4
1.3 Our Approach to verification 6
1.3.1 The Systematic Two Phase Steps in our Verification Methodology 7
1.3.2 The Partial Re-Usability in our Verification Methodology . 8
1.3.3 Scalability in our Verification Methodology 9
1.3.4 Handling Complex Protocols where the Temporal and Logical Orders differ 9
1.3.5 Handling protocols with large state-spaces 10
1.3.6 Summary of Results 10
1.3.7 Verification Property 11
1.3.8 Roadmap 12

2. RELATED WORK **13**

2.1 Formal Specification of memory consistency models 13
2.2 Automatic generation of executable specifications of memory consistency models 14
2.3 Formal Verification of memory consistency models of shared memory multiprocessors 15

3. FORMAL SPECIFICATION OF MEMORY CONSISTENCY MODELS **20**

3.1 Definitions 20
3.2 Hierarchy of Memory Models 22
3.2.1 *Strong* Memory Models 25
3.2.1.1 Memory ordering rules 25
3.2.1.2 Examples of *Strong* Memory Models : 26

3.2.2 *Weak* memory models: 26
3.2.2.1 Memory ordering rules: 27
3.2.2.2 Examples of *Weak* Models : 28
3.2.3 *Weakest* memory models: 29
3.2.3.1 Memory Ordering Rules 29
3.2.3.2 Examples of *Weakest* Memory Models 31
3.2.4 *Hybrid* memory models: 32
3.2.4.1 Examples of *Hybrid* Memory Models 33

4. EXECUTABLE SPECIFICATION OF MEMORY CONSISTENCY MODELS 34

4.1 Strong and Weak memory models 34
4.1.1 State Transition Rules of Generalized Weak memory model 35
4.2 Weakest and Hybrid memory models 37
4.2.1 State Transition Rules of Generalized Weakest Memory Model 39
4.3 A Hybrid Memory Model- Itanium 40
4.3.1 Litmus Test examples of Itanium memory model 41
4.3.2 The Operational Model of Itanium 42
4.3.3 State Transition Rules of Itanium Operational Model . . . 43
4.3.4 Analysis of Itanium Operational Model 47
4.3.4.1 Ordering Relaxations 49
4.3.4.2 How $R\rangle\rangle R$ may impact causality 50
4.3.5 Tool User Guide to specifying memory models 51

5. FORMAL VERIFICATION THROUGH AUTOMATED REFINEMENT CHECKING 54

5.1 Phase 1 of our Verification Method 54
5.2 Itanium memory model specification 55
5.3 Itanium implementation model 57
5.4 The Intermmediate Abstraction 60
5.5 Automatic Refinement Checking 61
5.5.1 Synchronization scheme 62
5.6 Handling aggressive optimizations 63
5.7 Derivation of Intermmediate Abstraction 65
5.8 Specifying, Designing and Verifying an implementation 67

6. EXPERIMENTS- SETUP, TEST CASES AND RESULTS 69

6.1 Experimental Setup 69
6.2 Test Cases 70
6.2.1 Internal Partition 70
6.2.2 External Partition 70
6.2.2.1 Split Transaction Bus Cache Invalidate Protocol . 71
6.2.2.2 Multiple Interleaved Bus Cache Invalidate Protocol 73
6.3 Summary of Results 73

7. CONCLUSION **75**

REFERENCES **77**

LIST OF FIGURES

3.1 The four classes of memory models 23

4.1 Generalized Weak Memory Model. 34

4.2 Generalized Weakest Memory Model. 37

4.3 An Operational Model of Itanium 43

4.4 Memory model specification tool interface- specifying Alpha memory model 51

4.5 Assembly level concurrent program as typed by user 53

4.6 Tool Output - All possible outcomes of the litmus test w.r.t. Alpha memory model 53

5.1 An Itanium Implementation 57

5.2 The derived Itanium Intermmediate Abstraction 60

5.3 Modified Itanium abstract Model 64

5.4 Specifying, Designing and Verifiying an Implementation 67

LIST OF TABLES

3.1 Splitting of store and data structures for each memory model class 24

4.1 Transition System of Generalized Weak Memory Model 35

4.2 Transition System of Generalized Weakest memory model 38

4.3 Transition System of Itanium Operational Model 44

5.1 Protocol Transactions 58

5.2 Completion steps of all events of implementation and abstract model 62

5.3 Splitting of store and external partition for each memory model class 66

6.1 Experimental Setup 69

6.2 Internal Partition of Alpha and Itanium Implementation 70

6.3 Protocol Actions and Reactions 72

6.4 Experimental Results 74

ACKNOWLEDGMENTS

Imagination is more important than knowledge.

— Albert Einstein .

I would like to thank many people who contributed directly or indirectly to my learning. First among them is my advisor, Prof. Ganesh Gopalakrishnan. I have had many stimulating discussions with him on both technical as well as non-technical matters. He gave me freedom to explore topics, and his hands-off style of functioning allowed me to learn through my own mistakes. Some other directions in which I made progress with his help include technical writing, presentation, paper reading, and teaching. I would like to thank Prof. John Carter for many of the same reasons, and for the courses he taught which to my opinion were the best. I have learned much in the process. I would like to thank Prof. Gary Lindstrom and Prof. Mike Jones for serving on my committee.

I want to thank my school friends Kali, Vikrant, Shailesh, Mukul, Shashank, Chandru, Arvinder and Bhasin and my college friends specially Nitesh and Bhootnath who have been an integral part of my life.

I had a great time being part of the Utah Verifier group, and enjoyed the many interesting discussions I had with my office-mates Ali, Ritwik, and Mohan.

Finally, I would like to thank my parents and my wife for freeing me of some family responsibilities to allow me to come to the graduate school, and for their constant encouragement during the last two years.

CHAPTER 1

INTRODUCTION

Weak shared memory consistency models [2], especially those used by modern microprocessor families, are quite complex. They allow many subtle reorderings among *load*s and *store*s, ultimately helping to hide *load* latencies, and allowing aggressive compiler optimizations. Manufacturers are nowadays committed to formal methods for specifying the memory models, as these specifications are the basis for many generations of microprocessors. The protocols that help realize shared memory multiprocessors using modern microprocessors are also exceedingly complex, being highly performance oriented. They exploit a rich pallet of optimizations, including snooping bus protocols, directory protocols, out-of-order buses, multiple interleaved buses, arbitrary interconnects, write buffers, load queues, invalidation queuing, superscalar execution, out-of-order execution and speculative execution. Despite considerable research, formal verification of shared memory consistency protocols against weak shared memory models is largely an unsolved problem [3, 4]. Most reported successes have been either for far simpler models such as cache coherence or sequential consistency (e.g., [5, 6, 7, 8]), or approaches that took advantage of existing architectural test program suites and created finite-state abstractions for them [9].

We present a methodology that systematically applies to a whole spectrum of weak memory consistency models. It supports a simple *deterministic* finite-state execution based method (using an explicit-state model-checker) to check the refinement relation between the memory model and the protocol. It requires very little formal verification expertise to be employed. It avoids taking the cartesian product

of the already large state-space of protocol implementations with the state-space of the specifications. Instead, it generates *no more* states than in the protocol. The state vector size during verification is only slightly higher than that for the protocol implementation.

We demonstrate our results on four different Alpha memory model implementations and four different Itanium memory model implementations against their respective specifications. The results are encouraging in terms of effort, potential for reuse of models, the degree of automation, run-times, and bug-hunting efficacy (it found many subtle coding errors). Our results, obtained using the parallel Murϕ model-checker (recently ported by us to run using the MPI library on our in-house Network Testbed[1]), strongly indicates that parallel model-checking will play a crucial role in finishing verification within acceptable durations.

To summarize, this thesis contributes in:

1. providing a new **definitional framework** to specify a wide spectrum of memory consistency models that aids in the verification process and forms as an input to

2. enable automatic generation of their corresponding **executable specifications** and

3. providing an **automatic**, **scalable**, **partially re-usable** and **systematic two phase** approach to formally verify memory consistency models of **complex** shared memory multiprocessors.

The challenges we faced, and the solutions we adopted in our work are now described - the rest of the thesis being an elaboration of these, in the indicated chapters.

[1] http://www.emulab.net

1.1 Our Definitional Framework to Specify Memory Consistency Models

While a plethora of approaches are available to specify memory models, not all of them are equally suited for model-checking based refinement. The main difficulty stems from the internal nondeterminism of specifications and implementations. To illustrate this difficulty, consider one source of internal non-determinism - namely *local bypassing.* Local bypassing allows a *load* to pick its value straight out of the store buffer, as opposed to allowing the store to post globally, and then reading the global store. Consider the operational-style specification of a memory model, such as the one illustrated in Figure 5.1(b) (the details of this figure are not important for our illustration). Consider one such model M_1, and make an *identical copy* calling it M_2. Let the external events of M_1 and M_2 (its alphabet) be *load* and *store*. Certainly we expect M_2 to refine M_1. However, both M_2 and M_1 can have different executions starting off with a common prefix, signifying non-determinism. For instance, if $P1$ runs program "$store(a, 1)$; $load(a)$," and P2 runs program "$load(a)$," one execution of M_1 obtained by annotating *load*s with the returned values is

$$\text{Exec1} = P1 : store(a, 1);\ P1 : load(a, 1);\ P2 : load(a, 1),$$

while an execution of the M_2 is

$$\text{Exec2} = P1 : store(a, 1);\ P1 : load(a, 1);\ P2 : load(a, \top),$$

where $\top$ is the initial value of a memory location. Note that after a common prefix, the *load* values disagree. Exec2 exercises the bypass option while Exec1 didn't do so.

It is easy to see that the above kind of non-determinism can be avoided if one uses *internal events* also in writing specifications. Fortunately, this is also the popular contemporary approach to specifying shared memory consistency (e.g., [10, 4, 11]).

These methods specify the so called *visibility order* of executions in terms of a richer alphabet than just *load* and *store*. For example, enriching the alphabet to $\Sigma' = \{store_{p1}(a,1), store_g(a,1), load(a,1), load(a,\top\}$, where $store_{p1}$ refers to the store being visible to P1 and $store_g$ refers to the store being visible globally, Exec1 and Exec2 detailed in terms of this alphabet to Exec1' and Exec2' show no non-determinism:

Exec1' $= P1 : store_{p1}(a,1);\ P1 : load(a,1);\ store_g(a,1);\ P2 : load(a,1)$

Exec2' $= P1 : store_{p1}(a,1);\ P1 : load(a,1);\ P2 : load(a,\top);\ store_g(a,1)$.

In general, there are many sources for non-determinism than just local bypassing. However, visibility order based specifications can be written in such a way that this non-determinism is avoided. The thing to note is that each *load* is associated with a *unique* past store event, thus avoiding non-determinism. In Chapter 3, we sketch how this approach can be applied to an entire spectrum of weak memory models. In Chapter 5, we illustrate the visibility order based approach that we have adopted to writing specifications on the Intel Itanium memory model.

1.2 Generating Executable Specifications Automatically

In a shared memory multiprocessor architecture, a memory model specifies the semantics of memory operations when multiple processors load and store shared memory locations. The precise details of this model are crucial to several people. Obviously, the design of the cache coherence scheme must respect the model. Also, processor designers must ensure that, for example, out-of-order issue of memory transactions conforms to the model. Programmers must be aware of the model because, for example, it affects the correctness of synchronization routines. Compiler writers may also have to consider the memory model in some optimizations.

Several memory models for shared-memory multiprocessor architecture have been

proposed. An early model, sequential consistency [12], simply required that multiprocessors simulate atomic reads and writes to a common global memory. This model is (relatively) easy to understand but has strong constraints which hinder performance improvements. During the past decade, a lot of effort has been made to design weaker memory models, such as release consistency [2], Itanium[13].

Weaker memory models are attractive because they allow more concurrency in memory systems and processors, resulting in improved performance. However, weaker memory models are generally very subtle. Dealing with concurrency is never easy, and even sequential consistency can be counter-intuitive at times. Hence, it is vital to specify a memory model precisely and an executable specification we believe can serve the purpose. Such a description provides a precise specification of the machine architecture, both for implementors and programmers.

Developing executable models greatly enhanced our understanding of their corresponding memory models for several reasons. First, being required to write a precise description points out ambiguities and inconsistencies to the author, even if the description is not executed. Second, we were able to analyze the possible outcomes of illustrative examples and synchronization programs rapidly and automatically, when there was a question about the implication of a change in some ordering rule. Hence, *this alternative specification serves as an useful aid in the initial stages of specifying a memory model.*

While in [14], Park uses an executable description to specify Ultra Sparc TSO, PSO and RMO it does not provide a uniform approach for generating formal executable specifications given a memory model.

The approach here is different from that used by Collier [15], who infers the behavior of programs from a set of ordering relations, which are not necessarily easy to convert into an operational form (our descriptions are executable). Gharachorloo [16] and Sindhu and Frailong [17] have used similar methods. Our method more

closely resembles that of Gibbons, et al. [18], who give I/O automata specifications of memory models. The primary differences here are the description languages, and more importantly, our emphasis on support for automatic processing (verification with I/O automata is generally by hand). Our approach to these problems is to describe the memory model by giving a maximally general executable description. Such a description provides a precise specification of the machine architecture, both for implementors and programmers.

We use an explicit state model checker SPIN [19] which apart from being an automatic formal verification system have been used to also describe the executable specification. This is a tool that supports exhaustive checking of all the reachable states of a description for deadlocks or violations of user-specified properties. It also allows the printing of the state of concurrent system at user-specified point while exploring the reachable states; this feature can be used, for example, to list all of the possible register values that can occur when an example program terminates. In our tool the user defines a memory model of his choice using our specification framework [20] and also enters the assembly language program he wants to run and check all possible outcomes w.r.t his specified model. The PROMELA code (input language of SPIN) for both the executable specification as well as the assembly program is then generated. The SPIN verifier is then used to run the assembly program on the executable specification and all possible outcomes of the program are delivered as output in terms of the data value returned to the load instructions. The tool is available at http://www.cs.utah.edu/formal_verification/ESGtool.

1.3 Our Approach to verification

We provide an **automatic**, **scalable**, **partially re-usable** and **systematic two phase** approach to formally verify memory consistency models of **complex** shared memory multiprocessors. We now elaborate on the above mentioned features in

our verification method in the following subsections.

Since model-checking is easier to use and more **automatable** than theorem proving, we prefer to base our verification approach predominantly on model-checking. Since writing a collection of temporal logic properties as assertions has not been shown to be feasible for verifying weak memory models, we prefer to employ model-checking as a way to run the implementation model against an abstract model, thus verifying the existence of a refinement mapping between the model. To make this approach practically feasible, memory model specifications must be written in a deterministic style, as we elaborated previously.

1.3.1 The Systematic Two Phase Steps in our Verification Methodology

In our approach, verification is divided into two distinct phases as follows:

1. **Phase 1:** In this phase, an intermediate abstraction Imp_{abs} which highly simplifies the implementation Imp is created, and Imp is verified against it through model-checking.

2. **Phase 2:** In this phase, Imp_{abs} is verified against $Spec$, the visibility-order based specification of the memory model.

We believe (as we will demonstrate) that the first phase can in itself be used as a very effective bug-hunting tool. The second phase can be conducted using theorem-proving, in the manner done in [21]. For the sake of completeness, in Appendix B, we show the paper-and-pencil proof-sketch for the intermediate abstraction we created for our Itanium implementation; most of the thesis is concerned with Phase 1.

1.3.2 The Partial Re-Usability in our Verification Methodology

For a large class of implementations, the second phase does not actually change, as the very same Imp_{abs} results from these implementations, thus permitting verification reuse. In fact, the Imp_{abs} models we end up creating are often the same as **operational style** models, such as the UltraSparc operational model of [22] or the Itanium operational model [13]. We expect the designer to annotate the Imp he created with a specific set of internal events. Such annotations are, essentially, designer assertions as to when (they think) the internal events are completing. This step is very natural since (in the ideal case) we expect a designer to study the visibility order style of the *Spec* which defines the set of internal events to annotate with and then only design Imp.

The creation of Imp_{abs} is based on the following observation. Most protocol implementations possess two distinct partitions: one (*small*) partition – called *internal partition* – containing the *load* and *store* buffers, and the other (*much larger*) partition – called *external partition* – containing the *cache*, *main memory*, and the (multiple) buses and directories. (In Chapter 5, we present such a protocol model that implements the Itanium memory model.) For a wide spectrum of memory models, we can create the Imp_{abs} model by retaining the internal partition, but replacing the external partition by a highly simplified operational device. The latter, in most examples is merely a single-port memory accessed through a multiplexed switch. In Chapter 5, we show the details of the construction of the intermediate abstraction. This approach also makes it possible to consistently annotate the internal and external partitions of Imp and Imp_{abs} with events from the enriched alphabet.

1.3.3 Scalability in our Verification Methodology

Another interesting practical advantage is that we can actually *share* the internal partition during our deterministic model-checking based refinement. This means that the implementation state-space is not multiplied out with the *Spec* state-space - rather, we only suffer from a modest increase in *state-vector size*.

1.3.4 Handling Complex Protocols where the Temporal and Logical Orders differ

In many aggressive protocols, the *logical* order of events (the "explanation" constructed by the verification method) is different from the *temporal* order in which the protocol performs these events. For example, consider an optimization described by Scheurich in the context of an invalidation-based directory protocol. In the ordinary version, a store request sent to the directory causes invalidations to be sent to each read-shared copy. The store can proceed only after the invalidations have been performed. Scheurich observed that the read-sharers can merely queue the invalidations, send "fake" acknowledgements back to the directory, and then perform invalidations in the background, so long as the read-sharers are "isolated" from the rest of the system. Thus, with Scheurich's optimization, even after a processor P_1 writes new data to a cache line, a *load* from some other processor P_2 to the same cache line can read stale data. Thus, in the logical order, these stores must be situated after the loads, even though in temporal order, the store is done before the loads on the stale lines. Most existing *automated* verification approaches have not considered such complications in the context of verifying weak memory models. In Chapter 5, we discuss how such protocols are handled. The creation of the intermediate abstraction, Imp_{abs}, again helps us partition our concerns.

1.3.5 Handling protocols with large state-spaces

Shared memory consistency protocols can easily have several billions of states, with global dependencies caused by pointers and directory entries that defy compact representation using BDDs. The use of a parallel model-checker is essential to make things practical. In some cases, we aborted runs that took more than 55 hours (due to thrashing) on a sequential model-checker even on machines with about 4GB of real memory. These runs finished in less than a few hours on our parallel model-checker.

1.3.6 Summary of Results

We applied our method to an implementation of the Alpha processor [23] that was modelled after a multiprocessor using the Compaq (DEC) Alpha 21264 microprocessor. The cache coherence protocol is a Gigaplane-like split transaction bus [24] protocol. We also verify an Alpha implementation with an underlying cache coherence protocol using multiple interleaved buses, modelled after the Sun *Ultra*^{TM} *Enterprise*^{TM} 10000 [25]. Both these implementations were verified with and without Scheurich's optimization. We also applied our method to our own implementation of the Itanium memory model. To our knowledge, nobody has verified such a list of protocols against different weak memory models using a uniform approach. These protocols finished in anywhere between 54 to 240 minutes on 16 processors, visiting up to 250 million states. The diameter of the reachability graph (indicative of the degree of parallelism in the problem) was sometimes in excess of 5,000. To better understand these numbers, we compare them with the highest numbers reported in [26], namely for the Stanford FLASH and the SCI protocols, where only 1 million states were explored, with the state graph diameter being close to 50.

We believe that the level of effort in applying our methodology is within practical limits. While the abstract model always retains, without change, the internal partition of the implementation, the *external* partition is considerably simplified. Designer insight is required in selecting the external partition, as this the external partition, as this depends on the memory model under examination. However, the effort needed here is not case-specific, but instead *class specific*. That is (as shown in Chapters 3 and 4), we can taxonomize memory models into four categories, and once and for all develop external partitions for each branch of the taxonomy. Designer insight is also required in attaching events to the abstract model.

1.3.7 Verification Property

As mentioned before, we use a model checker to verify that the implementation Imp refines its derived intermediate abstraction Imp_{abs}. Consequently, we prove that all possible executions generated by Imp given any concurrent program can also be generated by Imp_{abs}. The "final property" verified during model-checking in our approach is quite simple to state: that the loads completing in the implementation and specification models do return the same data. Thus, it is fair to say that the level of difficulty of the above two designer insights is comparable to that of writing properties for verification while using a model-checker, in general. Since a model checker can only deal with finite state processes, our intermediate abstraction modeled in Murϕ has bounded data structures and hence all possible executions of a concurrent program is only a subset of the executions of the intermediate abstraction with unbounded data structures. However, for the model checking based refinement where both the implementation and the intermediate abstraction are finite state processes, the bounded data strcutures suffice to enable the refinement checking.

1.3.8 Roadmap

The rest of the thesis is organized as follows. In Chapter 2 we detail related work by others and how our work compares to theirs. In Chapter 3 we introduce our new definitional framework to specify memory consistency models and in Chapter 4 we describe the automatic generation of the corresponding executable specification given the user defined specification in our newly introduced framework. In Chapter 5 we will explain **Phase 1** of our verification method in context of a specific example by applying it to a shared memory multiprocessor that claims to satisfy the Itanium Shared Memory Consistency Model. The breakup of this chapter is as follows:

1. We first specify the Itanium shared memory consistency model (*Spec*) in our own definitional framework.

2. We present a shared memory multiprocessor implementation *Imp* that claims to satisfy the Itanium shared memory consistency model *Spec*.

3. We then demonstrate how to derive the simplified model Imp_{abs} from *Imp*.

4. Finally, we explain how our Automated Refinement Technique works on the above example where we prove that the *Imp* refines Imp_{abs}(Appendix B provides a proof-sketch that Imp_{abs} satisfies the *Spec*).

We then illustrate how the creation of abstract models is influenced by non-trivial optimizations such as Scheurich's optimization [27]. We offer guidelines for the creation of abstracted models by presenting a taxonomy of shared memory consistent models and discussing the kinds of abstract models one can create with respect to this taxonomy.

CHAPTER 2

RELATED WORK

This thesis contributes in three areas mainly as follows:

1. Formal Specification of memory consistency models.

2. Automatic generation of executable specifications of memory consistency models, and

3. Formal Verification of memory consistency models of shared memory multiprocessors.

We now present the related work by others in each of these areas.

2.1 Formal Specification of memory consistency models

There has been a plethora of methodologies to specify memory consistency models. Collier [15] infers the behavior of programs from a set of ordering relations, which are not necessarily easy to convert into an executable form. Gharachorloo [16] and Sindhu and Frailong [17] have used methods similar to Collier's. Gibbons, et al. [28] give I/O automata specifications of memory models. However, verification with I/O automata is generally by hand. Higham[29] and Ahmed[30] proposed a history based specification of memory models. While, a plethora of these approaches are available to specify memory models, not all of them are equally suited for model-checking based refinement. The main difficulty stems from the internal nondeterminism of specifications and implementations and has been explained in the previous chapter. However, visibility order based specifications can be written

in such a way that this non-determinism is avoided. The thing to note is that each *load* is associated with a *unique* past store event, thus avoiding non-determinism.

The above approach is also the popular contemporary approach to specifying shared memory consistency (e.g., [10, 11]). These methods specify the so called *visibility order* of executions in terms of a richer alphabet than just *load* and *store*. In [10], Sorin specifies the Ultra Sparc TSO, Alpha and in [11], Neiger specifies Itanium. However, it does not explain how any memory model can be specified in their definitional framework thus not providing a common platform to specify and compare memory models. Moreover in [10], their definitional framework does not scale to specify memory models that do not require Write Atomicity or memory models that support more than one kind of load and more than one kind of store instructions. They also lack the proper formal specification framework that unambiguously can specify memory models.In [11], Gil Neiger specifies Itanium, a modern memory model that supports multiple flavors of load and store instructions and can also handle non atomic weak store instructions. However, the insight as to how his specification methodology can specify any memory model is not provided and does not propose how such specifications can enable automatic translation to corresponding executable specifications.

2.2 Automatic generation of executable specifications of memory consistency models

In [21], Park introduced this approach for specifying a memory model for multiprocessors. They describe the memory model by giving a maximally general executable description [31], using a high-level description language for concurrent systems called Murϕ [32]. Such a description provides a precise specification of the machine architecture, both for programmers and hardware implementors. They used this new alternative specification technique to successfully specify Ultra Sparc TSO, PSO and RMO [14]. We also specified the Intel Itanium memory model

in this similar executable fashion in [13]. However, these specifications have been manually developed and there is no general set of rules specified to describe any memory model in this style.

This thesis ameliorates the above two problems by *automatically* generating an unique executable specification using SPIN[33] as the description language given *any* user defined memory model specification in our new definitional framework.

The major advantage of using SPIN is that it is also an automatic verification system. There is a tool that supports exhaustive checking of all the reachable states of a description for violations of user-specified properties. Using the verifier allows users to experiment with the effects of the memory model on programs being executed in the memory system. Running the verifier can greatly help clarify the subtle details of the models.

2.3 Formal Verification of memory consistency models of shared memory multiprocessors

Memory consistency models can be quite complex and subtle, and their implementations is obviously even more complex creating a real possibility of design errors, especially for those used in large-scale multiprocessor systems. Formal verification is desirable because the bugs can be quite subtle and hard to capture by simulation. Several coherence protocols have been proposed but few are formally verified [34, 35, 36, 37].

In [38], Giorgio Delzanno proposed a formal method to verify snoopy cache coherent protocols where he used arithmetic constraints to model possiblly infinite sets of global states of multiprocessor systems with many identical caches. However, his method is not applicable to directory based protocols. Moreover, cache coherent protocols is just a subset of memory consistency models that have cache coherence as a requirement and is relatively simpler to verify.

Only simple memory models such as Sequential Consistency[12] have been verified

for very simple implementations like Lazy Caching protocol[39]. In [6], Qadeer presents a model checking algorithm to verify sequential consistency on implementations that possess some desired properties for a finite number of processors and memory locations and an arbitrary number of data values. This method has been applied to a simple snoopy cache coherent protocol and is restricted to implementations where logical order and temporal order of completion of events differ. Existing verification approaches to verify weaker and complex memory models of complex implementations have resorted to finite state abstractions of multiprocessor test programs [40] but their approach is incomplete in the sense that if a bug is not detected then the fact that the implementation is bug free cannot be guaranteed. Moreover, they only handle one kind of cacheable load and store instructions whereas current memory models support multiple cacheable and multiple non cacheable load and store instructions, semaphores, atomic read modify writes, memory barriers. Our verification provides a scalable technique to verify complex weak memory models of real life shared memory implementations that support multiple load and store instructions and memory barriers and can easily be extended to support the whole set of instructions. To our knowledge, no one has verified such complex memory models of such various real life shared memory multiprocessors that can have billions of states.

One of the effective ways to validate shared memory models of implementations is using finite-state methods (model checking). Finite-state methods enumerate the states of the reachable state graph of the system, searching for states that violate a specified property (e.g. Murϕ [32], SMV [41], SPIN [33], COSPAN [42]). A model checker supports expressing properties in a temporal logic like CTL, LTL, CTL* [43]. For our purpose, the specification is the corresponding memory model of the protocol so it is required to encode a full memory model in temporal logic. However, it has never been shown that memory models can be specified using temporal logic.

Graf [44] uses a strong tempotal logic CTL* to specify the sufficient conditions for Sequential Consistency and hence not the precise specification. Lamport specifies Alpha [45] memory model using TLA [46] which can be fed into a model checker TLC [4] but it still has not been applied as a specification in a model checker to verify implementations. We avoid this problem in model checking by verifying the implementation against a simpler abstract model which is similar to an executable specification or operational model.

Another approach to formal verification is computer assisted theorem proving. Theorem-provers make available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods. However, the major problem with theorem proving is that considerable manual labor is involved.

Consequently, previous theorem proving approaches have not been able to verify a problem of the scale of a full multiprocessor cache coherence protocol. The most significant result to date is a manual proof of "lazy caching," a simple and abstract cache coherence algorithm [39].

Overall, the finite-state methods have been applied to simple and straightforward protocol models. However, we expect that the complexity of memory model implementations will continue to increase as faster and larger relaxed memory models are implemented. Then verifying them becomes a serious challenge which must be met with appropriate and efficient techniques.

We have proposed a verification technique that takes advantage of the good properties of both model checking and computer assisted theorem proving. The relevant feautures are as follows.

1. We use model checking to verify that a shared memory multiprocessor refines its derived and far simpler abstract model and use theorem proving to verify that this simple abstract model satisfies the visibility based specification.

2. We handle the state space problem by using a parallelized model checker that helped us explore on the order of billions of states.

3. In our model checking approach we need to explore only the state space of the implementation as opposed to the multiplied state space of both the implementation and the abstract model, hence avoiding adding any overhead.

4. Our work imposes far less burden on designers, as opposed to that imposed by an exclusively theorem-proving based approach.

5. We use theorem proving in verifying the very simplified abstract model which is only composed of simple atomic transactions, the manual effort of which is negligible as compared to theorem proving a complex implementation.

6. Moreover, the same abstract model can be used for verifying different implementations of the same memory model. Hence, the reusability of abstract models avoids the manual theorem proving effort in many cases.

In [47], an approach is proposed in which event sequences generated by protocol implementations are verified by a much simpler (and hence more trustworthy) protocol processor. While this approach is closest in spirit to our approach, their proposal helps verify only the final silicon, and not designs themselves. We emphasize design-level verification where bugs are often the cheapest to detect. Moreover, their approach is for verification against cache coherence, and not memory consistency models. The approach in [47] was inspired by [48], which uses a similar approach to verify processor pipelines. While the approach used in this thesis is similar to the above approaches, nobody to our knowledge has attempted to verify exactly what we have verified: that several non-trivial protocols used to realize two separate modern weak memory models, namely Alpha and Itanium, are correct. While parallel and distributed model-checking in itself forms a rapidly growing area,

nobody to our knowledge has used it for refinement checking for weak memory models, as we do.

CHAPTER 3

FORMAL SPECIFICATION OF MEMORY CONSISTENCY MODELS

We now introduce our new definitional framework to specify memory consistency models in the following sections.

3.1 Definitions

A concurrent shared memory program is viewed as a set of sequences of instructions, one sequence per processor. Each sequence in the set captures the *program order* at that processor. An *execution* of this program is obtained by running the shared memory program on an MP system. Formally, an execution is the above set of sequences of instructions with each *load* labelled with its returned value. Every instruction in an execution can be decomposed into one or two[1] *events* (*local* and *global* in the latter case; in the former case, we shall use the words 'instruction' and 'event' synonymously). Each event t is defined as a tuple (p, l, o, a, d) where

- $p(t)$: processor in whose program t originates from.
- $l(t)$: label of instruction t in p's program .
- $o(t)$: event type
- $a(t)$: memory address
- $d(t)$: data

[1] In general, as discussed later, there could be more events.

The approach presented in this thesis only deals with cacheable memory instructions *acquire load*s (written *ld.acq*), *ordinary load*s (*ld*), *release store*s (*st.rel*), *ordinary store*s (*st*) as well as memory fences. Handling atomic read-modify-writes, non-cacheable memory locations, and special rules pertaining to register data dependencies can be easily extended using our specification methodology. Some of the instructions like *acquire load*s and *release store*s are only applicable to certain classes of memory models where more than one kind of load and store instructions are supported.

All instructions except store instructions are decomposed into exactly one event. Each *st* is decomposed into one or more events depending upon the memory model under consideration. An *execution* obeys a memory model if all the memory events of the *execution* form at least one logical total order $\rightarrow$ which obeys:

- the *Per Processor Order* stipulated by the memory model, and
- the *Read Value* Rule stipulated by the memory model. In addition,
- some store events may or may not appear atomically to all processors.

The *Read Value* Rule specifies the data value to be returned by the load events in an execution. The *Per Processor Order* includes both program order as well as data dependence order. The logical order ($\rightarrow$) of completion of events may not be the same as the temporal order of completion. For example, consider a processor that allows *local bypassing*, *i.e*, early retirement of loads that directly read off the store buffer. Suppose the processor also implements memory fences aggressively by not waiting for the existing entries in the store buffer to be flushed before retiring the fence (they are flushed only when a subsequent coherent transaction is encountered). An instruction sequence `store(a); fence; load(a)` can be executed by this processor as *store(a).local; load(a); store(a).global*, in temporal order. In other words, the `load` is 'globally performed' before the `store` is globally

performed. However, in a logical explanation (that respects the fence semantics) `store(a).global` must precede the `load(a)`.

3.2 Hierarchy of Memory Models

An executable specification (operational model) of any memory model can be visualized as a correct and simple 'implementation' whose every run satisfies the memory model definition and also every run that satisfies the memory model can be generated by the executable specification. The selection of this simplified 'implementation' depends on the memory model under examination. In this section we categorize memory models into four classes and show how common data structures to design the operational model can be derived for memory models belonging to a particular memory model class, thus providing a systematic approach to deriving the operational model. The four classes of memory models are as follows::

1. *Strong*: requires *Write atomicity* and does not allow local bypassing. (e.g. Sequential Consistency [12], IBM-370[49]).

2. *Weak*: requires *Write atomicity* and allows local bypassing (e.g. Ultra Sparc TSO, PSO and RMO [50], Alpha[51])

3. *Weakest*: does not require *Write atomicity* and allows local bypassing (e.g.PC [28], PowerPC [2], PRAM [52], Slow Memory [53]).

4. *Hybrid*: supports weak load and store instructions that come under *Weakest* memory model class and also support strong load and store instructions that come under *Strong* or *Weak* memory model class. (e.g. Itanium [13], DASH-RC [2], RC_{PC} [2]).

Memory models that are weaker than the *Strong* class do not usually require that an *execution* have all its load and store instructions form a total order as a part of their specification. However, the prevalent style (e.g. [10, 4]) of specifying even

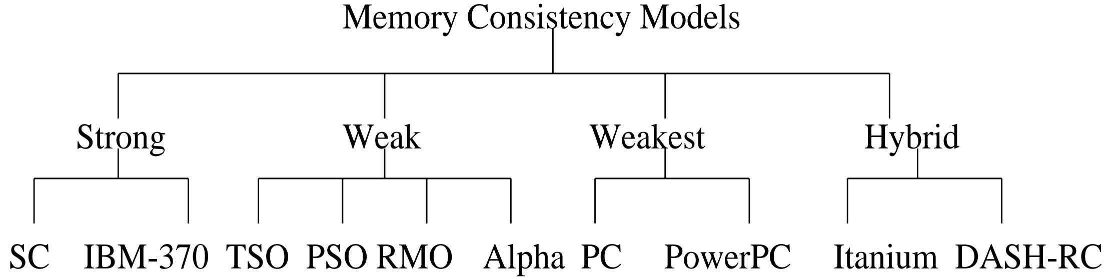

Figure 3.1. The four classes of memory models

these memory models is to adopt the approach of identifying a logical total order such as $\rightarrow$ described earlier. In our work, we also adopt this point of view. In other words, even in case of these weaker memory models, by splitting loads and stores to finer events, we define a logical total order of all the memory events.

Depending upon the category a memory model falls under, we split a store instruction into one or more events. Load instructions for any memory model can always be treated as a single event. Here are a few examples of splitting events. In case of Sequential Consistency, we do not split even the stores as sequential consistency demands a single global total order of the loads and stores. For a weak memory model such as the Ultra Sparc TSO, we split the store instruction into two events, a local store event (which means that the store is only visible to the processor who issued it) and a global event (which means that the store event is visible to all processors). Since the *Weakest* category of memory models lack write atomicity, we need to split stores into $p+1$ events, where p is number of processors, thus ending up with a local store event and p global events (global event i would mean that the store event is visible to processor i). Table 3.1 summarizes these splitting decisions for various memory models. It also shows the data structures

chosen for various memory models in order to design the corresponding executable specification.

Memory Model	Splitting of store instructions	Data Structures[2]
Strong	store unsplit	single port memory
Weak	store split to local and global	single port memory
Weakest	store split to local and $(p+1)^3$globals	memory and re-order buffer per processor
Hybrid	store split to local and $(p+1)$ globals	memory and re-order buffer per processor

Table 3.1. Splitting of store and data structures for each memory model class

In case of *Strong* and *Weak* memory models, the external data structures is just a single port memory M. The intuition behind having M is that both these classes of memory models require *Write Atomicity* and hence a store instruction should be visible to all processors instantaneously. *Weakest* and *Hybrid* memory models require more involved data structures where each processor i has its own memory M_i and also a re-ordering buffer that takes in incoming store instructions posted by different processors including itself from their internal store buffers. Store instructions residing in this buffer eventually get flushed to memory. The combination of M_i and a re-ordering buffer simulates a processor seeing store instructions at different times and different relative order as that of another processor. An algorithm that generates the correct executable specification, given the memory model will be discussed shortly. A remotely executable web-based tool is available

[3]each memory model class also includes internal load and store buffers

[3]p is number of processors

for experimenting with the operational models thus generated.

We now specify each class of memory models in details.

3.2.1 *Strong* Memory Models

Each memory instruction including store is split into one event.

3.2.1.1 Memory ordering rules

We now define *Read Value* rule that must be satisfied by any *Strong* memory model and also define all possible sub-rules of the *Per Processor Order* rule where a *Strong* memory model may obey zero, some or all of these sub-rules. *Write Atomicity* is also defined and should be obeyed by any *Strong* memory model.

1. *Read Value:* Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store instruction t_2 to the same memory location as t_1 in the total order relation $\rightarrow$. i.e

   (a) $o(t_2) = st, a(t_1) = a(t_2), t_2 \rightarrow t_1$ and

   (b) there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$ and $t_2 \rightarrow t_3 \rightarrow t_1$.

2. *Per Processor Order:* Let t_1 and t_2 be two instructions s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$. *Per Processor Order* rules constitutes of five sub-rules.

   (a) $ld \rightarrow ld$: $o(t_1) = ld, o(t_2) = ld$.

   (b) $ld \rightarrow st$: $o(t_1) = ld, o(t_2) = st$.

   (c) $st \rightarrow ld$: $o(t_1) = st, o(t_2) = ld$.

   (d) $st \rightarrow st$: $o(t_1) = st, o(t_2) = st$.

   (e) fence: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$.

Any number of these sub-rules may be applicable to a memory model. For any sub-rule that holds true for that memory model, $t_1 \rightarrow t_2$.

We use concise representations to express the list of sub-rules that holds for a memory model(e.g if both $ld \rightarrow ld$ and $ld \rightarrow st$ sub-rules are satisfied then we can represent that memory model to satisfy $ld \rightarrow X$ where X means any memory event)

3. *Write Atomicity:* All the store events form a single totat order in a way that every store instruction appear to be visible atomically to all processors.

3.2.1.2 Examples of *Strong* Memory Models :

An *execution* satisfies a *Strong* memory model if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* and *Write Atomicity* rules and also obeys zero or more sub-rules of the *Per Processor Order* rules depending upon that memory model.

The sub-rules of *Per Processor Order* rules that are applicable to the following memory models are as follows :

1. Sequential Consistency: All subrules except *fence* holds(SC is defined in absence of fence instructions). Hence we can concisely say that $X \rightarrow X$ holds.

2. IBM-370: Sub-rules $ld \rightarrow X$, $st \rightarrow st$, $st \rightarrow ld$ *per memory location* and fence holds.

3.2.2 *Weak* memory models:

Every store instruction is split into two memory events . Hence a store instruction $t = (p, l, st, a, d)$ is split into $t^{local} = (p, l, st_{local}, a, d)$ and $t^{global} = (p, l, st_{global}, a, d)$, both of which have the same data value. Each load instruction may get its *Read Value* from a t^{local} for which the corresponding t^{global} has not yet occurred. The

goal in this case is to model "local bypassing" i.e to model a store buffer bypassing where stores enters the store buffer as t^{local} when its visible to *only* the processor that issued it and exit with a t^{global} when its visible to all processors. $t^{local} \rightarrow t^{global}$ always.

3.2.2.1 Memory ordering rules:

1. *Read Value*: Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store event split from the store instruction t_2, to the same memory location as t_1 in the total order relation $\rightarrow$. i.e

   (a) if

      i. $p(t_1) = p(t_2)$, $a(t_1) = a(t_2)$, $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ and

      ii. there does not exist a store instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$, $t_2^{local} \rightarrow t_3^{local} \rightarrow t_1$.

   (b) else if

      i. $a(t_1) = a(t_2)$, $t_2^{global} \rightarrow t_1$ and

      ii. there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$, $t_2^{global} \rightarrow t_3^{global} \rightarrow t_1$.

   (c) else, t_1 receives the initial value 0.

2. *Per Processor Order:* Let t_1 and t_2 be two events s.t $p(t_1) = p(t_2)$,$l(t_1) < l(t_2)$. There are six possible sub-rules as follows:

   (a) $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

   (b) $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{global}$.

   (c) $st \rightarrow ld$: $o(t_1) = st_{global}$, $o(t_2) = ld$.

   (d) $st \rightarrow st$: $o(t_1) = st_{global}$, $o(t_2) = st_{global}$.

(e) $fence$: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$.

(f) Memory Data Dependence: This sub-rule pertains to instrcutions involving the same memory location. Hence, for $a(t_1) = a(t_2)$,

i. $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

ii. $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{local}$.

iii. $st \rightarrow ld$: $o(t_1) = st_{local}$, $o(t_2) = ld$.

iv. $st \rightarrow st$: $o(t_1) = st_{local}$, $o(t_2) = st_{local}$.

For any sub-rule that holds true for a memory model, $t_1 \rightarrow t_2$. Whenever we use "X" we refer to it as ld or st instruction.

3. *Write Atomicity:* All the store events t where $o(t) = st_{global}$, form a single totat order in a way that every store event appear to be visible atomically to all processors.

3.2.2.2 Examples of *Weak* Models :

An *execution* satisfies a memory model in this category if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* and *Write Atomicity* rules and obeys zero or more sub-rules of the *Per Processor Order* rules depending upon that memory model.
The sub-rules of *Per Processor Order* rules that are applicable to the following memory models are as follows :

1. TSO: Sub-rules $ld \rightarrow X$, $X \rightarrow st$, *Memory Data Dependence* and fence holds.

2. PSO: Sub-rules $ld \rightarrow X$, *Memory Data Dependence* and fence holds.

3. RMO: Sub-rules *Memory Data Dependence* and fence holds.

4. Alpha: Sub-rules *Memory Data Dependence* and fence holds. Note that Alpha has two kinds of fences, MB which is equivalent to our definition of fence and

$MB - WW$ which is applicable between store instuctions only ($MB - WW$ can be defined in a simlar way as MB was defined).

3.2.3 *Weakest* memory models:

Every store instruction is split into $p + 1$ events where p is the number of processors. Thus t where $o(t) = st$ is split into t^{local} where $o(t^{local}) = st_{local}$ and $t^1, .t^k...t^p$ where $o(t^k) = st^k_{global}$ and $t^{local} \to t^{p(t)} \to t^k, \forall k(1 \leq k \leq p$ and $k \neq p(t))$. Since *Weakest* memory models do not require *Write Atomicity*, a store instruction may be visible to one processor earlier or later than when its visible to another processor. Hence the need to identify when a store event is visible to all processors at different times where t^k event corresponds to when the store event t is visible to processor k.

3.2.3.1 Memory Ordering Rules

1. *Read Value*: Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store event split from the store instruction t_2, to the same memory location as t_1 in the total order relation $\to$. i.e

   (a) if

   - $p(t_1) = p(t_2)$, $a(t_1) = a(t_2)$, $t_2^{local} \to t_1 \to t_2^{global}$ and
   - there does not exist a store instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$, $t_2^{local} \to t_3^{local} \to t_1$.

   (b) else if

   - $p(t_1) = p(t_2)$, $a(t_1) = a(t_2)$, $t_2^{p(t_1)} \to t_1$ and
   - there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$, $t_2^{p(t_1)} \to t_3^{p(t_1)} \to t_1$.

   (c) else, t_1 receives the initial value 0.

2. *Per Processor Order*: Let t_1 and t_2 be two events(can be $ld.acq, st^{local}, st^k$ for any $k \in$ processors) s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$. There are six possible sub-rules.

   (a) $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

   (b) $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st^{p(t_1)}$.

   (c) $st \rightarrow ld$: $o(t_1) = st^{p(t_1)}$, $o(t_2) = ld$.

   (d) $st \rightarrow st$: $o(t_1) = st^{p(t_1)}$, $o(t_2) = st^{p(t_1)}$.

   (e) fence: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$ and $o(t_1) = o(t_2) = st^{p(t_1)}$.

   (f) Memory Data Dependence: This sub-rule pertains to instructions involving the same memory location. Hence, for $a(t_1) = a(t_2)$,

      i. $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

      ii. $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{local}$.

      iii. $st \rightarrow ld$: $o(t_1) = st_{local}$, $o(t_2) = ld$.

      iv. $st \rightarrow st$: $o(t_1) = st_{local}$, $o(t_2) = st_{local}$.

   If any one of the above sub-rules hold then $t_1 \rightarrow t_2$. Again, if we use "X" then we refer to it as ld or st event.

3. *Coherence*: All this time we never referred to *Coherence* as for *Strong* and *Weak* memory models *Coherence* is a subrule of *Write Atomicity*. However now we need to define *Coherence* separately simply because *Weakest* memory models does not support *Write Atomicity* but may or may not support *Coherence*. *Coherence* requires that if t_1 and t_2 be two store instructions s.t $a(t_1) = a(t_2)$ then

   (a) If $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$, then $t_1^k \rightarrow t_2^k, \forall$ processors k.

(b) If $t_1^p \rightarrow t_2^p$ for some processor p then $t_1^k \rightarrow t_2^k, \forall$ processors k.

4. *Write Atomicity*: Although *Weakest* memory models do not require *Write Atomicity* we will still define it when store instructions are split to $p+1$ events, its importance will be apparent in the next section where we will define *Hybrid* models.

   Here we have two cases as follows:

   (a) case 1: If a store instruction t is split into $p+1$ events i.e $t^{local}, t^1, ...t^p$ then they all appear to occur atomically i.e in the total order relation if $t^i \rightarrow t^j$ where $i, j \in$ {local,1,..,p} and if $t^i \rightarrow$t'$\rightarrow t^j$ then t' can only be $\in \{t^{local}, t^1, .., t^p\}$. Thus, all the events of t are ordered in a way s.t no instruction that does not belong to an event of t can come in between.

   (b) case 2: If a store instruction t is split into $p+1$ instructions i.e $t^{local}, t^1, ...t^p$ then except t^{local} they all appeat to occur atomically i.e in the total order relation if $t^i \rightarrow t^j$ where $i, j \in$ {local, 1 ,.., p} and if $t^i \rightarrow$t'$\rightarrow t^j$ then t' can only be $\in \{t^1, .., t^p\}$.

Note that the definition of case 1 is same as the definition of *Write Atomicity* for *Strong* memory models and the definition of case 2 is same as the definition of *Write Atomicity* for *Weak* memory models where they only differ in how store instructions are split.

3.2.3.2 Examples of *Weakest* Memory Models

An *execution* satisfies a memory model in this category if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* , may or may not satisfy *Coherence* and obeys zero or more sub-rules of the *Per Processor Order* rules depending upon that memory model.

The sub-rules of *Per Processor Order* rules that are applicable to the following memory models are as follows :

1. PC: Sub-rules $ld \rightarrow X$, $X \rightarrow st$, *Memory Data Dependence* and fence holds.

2. PowePC: Sub-rules *Memory Data Dependence* and fence holds.

For both PC and $PowerPC$, Coherence is satisfied.

3.2.4 *Hybrid* memory models:

All memory models that support more than one kind of load and store instructions where all *execution*s containing only the strong load and store instructions obey a memory model under *Strong* or *Weak* memory model class and all *execution*s containing only the weak load and store instructions obey a memory model under *Weakest* memory model class.
In such models where both strong and weak store instructions exist there can be two ways to view these store instructions as follows:

- split the strong store instructions t into t^{local} and t^{global} and split the weak store instructions into $p+1$ events i.e $t^{local}, t^1, .., t^p$.

- split both the weak and strong store instructions into $p+1$ events i.e $t^{local}, t^1, .., t^p$.

Although breaking up the strong store instructions into $p+1$ events seems redundant in order to uniformly define their properties and to be able to explain the interaction between weak and strong store events more clearly the latter method is more logical.

Hence every strong and weak store instructions are split into $p + 1$ events as explained before. The memory ordering rules like *Read Value*, *Per Processor Order*, *Coherence*, *Write Atomicity* are same as defined for *Weak* memory ordering rules. However, a reference to ld or st instruction corresponds to weak load and store instructions respectively. Instructions ld.acq and st.rel refer to strong load and store instructions respectively, unlike as for *Weakest* memory models where ld and

st corresponds to one unique type of load and store instruction. Consequently, for example, a sub-rule $ld \rightarrow st$ of *Per Processor Order* rules defined for *Weakest* memory models corresponds to $ld.acq \rightarrow st$, $ld.acq \rightarrow st.rel$, $ld \rightarrow st$, and $ld \rightarrow st.rel$ subrules for *Hybrid* memory models where each of these four sub-rules are defined similar to $ld \rightarrow st$ sub-rule for *Weakest* memory models .

If for a rule or sub-rule the type of load or store instruction is not mentioned then that rule or sub-rule is applicable to both types of load and store instructions respectively. "X" would refer to $ld, ld.acq, st$, or $st.rel$ event.

3.2.4.1 Examples of *Hybrid* Memory Models

1. $Itanium_{TM}$: An *execution* obeys Itanium memory model if there exists a logical total order of all memory events (ld, st, ld.acq, st.rel, fence) that obeys *Read Value*, *Coherence*, *Write Atomicity*(case 2) for all st.rel store events and *Per Processor Order* rules which include subrules $ld.acq \rightarrow X$, $X \rightarrow st.rel$, *Memory Data Dependence* except $ld \rightarrow ld$ and fence. Note that *Coherence* is defined irrespective of whether a store instruction is strong or weak.

   RC_{PC}, RC_{SC} are some more examples and in the latter the strong store instructions obey *Write Atomicity* (case 1).

CHAPTER 4

EXECUTABLE SPECIFICATION OF MEMORY CONSISTENCY MODELS

We now explain the method by which the user specified memory consistency models defined in our framework as explained in the previous chapter can be automatically translated to an executable specification or an operational model.

4.1 Strong and Weak memory models

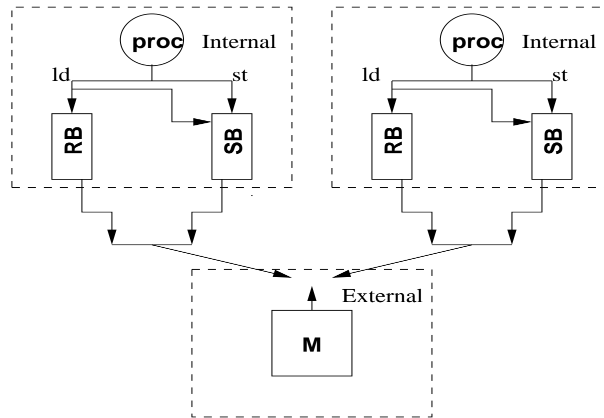

Figure 4.1. Generalized Weak Memory Model.

We will now describe the operational semantics of the Generalized Weak memory model described in terms of its data structures (see Figure 4.1), and how each instruction tuple t that is issued updates these data structures and/or returns the read value, as per Table 4.1. The data structure elements are:

1. a single port memory M that spans the entire address-space and holds word-sized data in each location. M is updated when the $MW(t)$ event of Table 4.1 fires, which removes an entry from $SB_{p(t)}$ writes into M. Initially, each location of M carries data 0.

2. a write out re-order buffer SB_i into which st instructions are enqueued. When the $st_{global}(t)$ event of Table 4.1 fires and $p(t) = i$, the entry t is removed from SB_i and atomically copied into M. By 'buffer' we mean an unbounded structure in which the entries maintain their arrival order as in a FIFO, but entries may be removed from anywhere provided a removal condition is satisfied. The oldest entry is always at the head and the youngest at the tail. Initially, all buffers are empty.

3. a re-order load buffer RB_i into which ld instructions are enqueued when event $(ld(t))$ of Table 4.1 fires. Eventually, t is removed from RB_i, and data $d(t)$ corresponding to this tuple gets returned.

4.1.1 State Transition Rules of Generalized Weak memory model

Event	*Guard*	*Actions*
$ld(t)$ (hit)	$\exists$ youngest $t' \in SB_{p(t)} : a(t') = a(t) \wedge d(t') = d(t)$	
$ld(t)$ (miss)	$\neg\exists\ t' \in SB_{p(t)} : a(t') = a(t)$	Issue($RB_{p(t)}$, t)
$ld(t)$ (miss) *contd.*	t$\in RB_{p(t)} \wedge$ Allowed$(RB_{p(t)}, t) \wedge M[a(t)] = d(t)$	Delete$(RB_{p(t)}, t)$
$st_{local}(t)$	True	Issue$(SB_{p(t)}, t)$
$st_{global}(t)$	$t \in SB_{p(t)} \wedge$ Allowed$(SB_{p(t)}, t)$	$M[a(t)] \leftarrow d(t)$; Delete$(SB_{p(t)}, t)$;
$Fence(t)$	True	Flush(t)

Table 4.1. Transition System of Generalized Weak Memory Model

Table 4.1 defines the operational semantics of Generalized Weak memory model. The first column shows *Event*s that happen if the *Guard* condition in the second

column is true, performing the *Actions* shown in the last column. At any time, any one of the eligible events may be picked in a *fair* manner. Each event happens when the next instruction t is issued by processor $p(t)$ ($ld(t)$,$st(t)$,$Fence(t)$). Notice that in case of event $ld(t)$, tuple t carries the data $d(t)$ being returned (following the convention used in [39]). When these events fire, a constraint expressed in the Guard field shows what this data is. We use $=$ for equality testing, and $\leftarrow$ for assignment.

$\underline{ld(t)}$: We seek an entry t' in $SB_{p(t)}$ such that $a(t') = a(t)$, and t' is the youngest such entry, if multiple entries exist. If t' exists (hit), the returned data $d(t)$ is the same as $d(t')$. If no such entry exists (miss), t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$. Eventually, the $ld(t)$ event completes by being serviced by M which provides the data $d(t)$. Its guard 'Allowed' captures when tuple t, which is present in $RB_{p(t)}$, can be processed ahead of all the other tuples within $RB_{p(t)}$.

$\underline{st_{local}(t)}$: results in t being enqueued into $SB_{p(t)}$ via procedure $Issue$.

$\underline{st_{global}(t)}$ updates the memory array M from $SB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $SB_{p(t)}$, can be processed ahead of all the other tuples within $SB_{p(t)}$.

$\underline{Fence(t)}$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $SB_{p(t)}$ entry, where the entry comes from $p(t)$ and occurs earlier than t in program order. The functions used in the transition system are now described.

Allowed($SB_{p(t)}$, t): The function evaluates to true if

1. $\neg\exists t' \in SB_{p(t)}$ s.t $l(t') < l(t) \wedge (st \rightarrow st^1 \vee (a(t_1) = a(t_2) \wedge$ *Memory Data Dependence:*$st \rightarrow st))$.

2. $\neg\exists t' \in RB_{p(t)}$ s.t $l(t') < l(t) \wedge (ld \rightarrow st \vee (a(t_1) = a(t_2) \wedge$ *Memory Data Dependence:*$ld \rightarrow st))$.

Allowed($RB_{p(t)}$, t): The function evaluates to true if

1. $\neg\exists t' \in RB_{p(t)}$ s.t $l(t') < l(t) \wedge (ld \rightarrow ld \vee (a(t_1) = a(t_2) \wedge$ *Memory Data Dependence*:$ld \rightarrow ld))$

2. $\neg\exists t' \in SB_{p(t)}$ s.t $l(t') < l(t) \wedge (st \rightarrow ld \vee (a(t_1) = a(t_2) \wedge$ *Memory Data Dependence*:$ld \rightarrow st))$

Issue$(Buffer, t)$: Add t to the tail of Buffer queue.

Delete$(Buffer, t)$: This procedure deletes t wherever it may be in Buffer.

Although we designed a Generalized Weak memory model, Strong memory models can also be designed with the same data structures and operational semantics except that now a load instruction cannot hit the SB buffer (local bypassing) due to Strong memory models not supporting local bypassing.

4.2 Weakest and Hybrid memory models

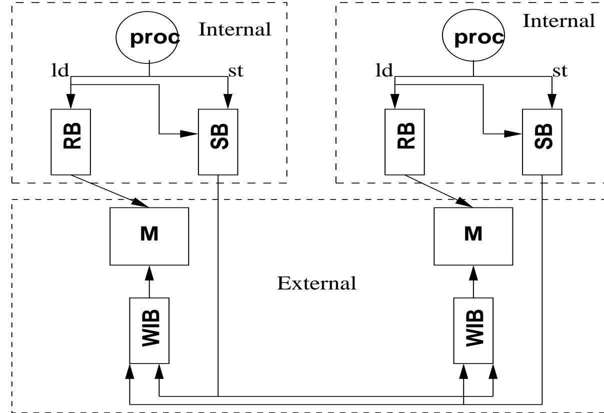

Figure 4.2. Generalized Weakest Memory Model.

The operational semantics of the Generalized Weakest memory model is now described in terms of its data structures (see Figure 4.2), and how each instruction

[1]Here $st \rightarrow st$ refers to *Per Processor Order* sub-rule $st \rightarrow st$ as a boolean condition which *IF* **True** means that the user specified memory model whose operational model is being generated requires this sub-rule to be satisfied

tuple t that is issued updates these data structures and/or returns the read value, as per Table 4.2. Initially, all buffers are empty. The data structure elements are:

1. memory M_i per processor i that spans the entire address-space and holds word-sized data in each location. M_i is updated when the st_{global} *contd.* event of Table 4.2 fires, which removes an entry from $WIB_{p(t)}$ and writes into M_i. Initially, each location of M_i carries data 0.

2. a write out re-order buffer SB_i into which st instructions are enqueued. When the $st_{global}(t)$ event of Table 4.2 fires, the entry t is removed from SB_i and atomically copied into WIB_i for every processor i.

3. a re-order load buffer RB_i into which ld instructions are enqueued when event $(ld(t))$ of Table 4.2 fires. Eventually, t is removed from RB_i, and data $d(t)$ corresponding to this tuple gets returned.

4. a write in re-order buffer WIB_i into which st instructions are enqueued. When the $st_{global}(t)$ *contd.* event of Table 4.2 fires, the entry t is removed from WIB_i and atomically copied into M_i.

Event	*Guard*	*Actions*
$ld(t)$ (hit)	$\exists$ youngest $t' \in SB_{p(t)} : a(t') = a(t) \wedge d(t') = d(t)$	
$ld(t)$ (miss)	$\neg\exists\ t' \in SB_{p(t)} : a(t') = a(t)$	Issue($RB_{p(t)}$, t)
$ld(t)$ (miss) *contd.*	t$\in RB_{p(t)} \wedge$ Allowed$(RB_{p(t)}, t) \wedge M_{p(t)}[a(t)] = d(t)$	Delete$(RB_{p(t)}, t)$
$st_{local}(t)$	True	Issue$(SB_{p(t)}, t)$
$st_{global}(t)$	$t \in SB_{p(t)} \wedge$ Allowed$(SB_{p(t)}, t)$	$\forall$ processors i Issue(WIB_i, t); Delete$(SB_{p(t)}, t)$;
$st_{global}(t)$ *contd.*	$t \in WIB_{p(t)} \wedge$ Allowed(WIB_i, t)	$M_i[a(t)] \leftarrow d(t)$; Delete$(WIB_i, t)$;
$Fence(t)$	True	Flush(t)

Table 4.2. Transition System of Generalized Weakest memory model

4.2.1 State Transition Rules of Generalized Weakest Memory Model

Table 4.2 defines the operational semantics of Generalized Weakest memory model.

$ld(t)$: We seek an entry t' in $SB_{p(t)}$ such that $a(t') = a(t)$, and t' is the youngest such entry, if multiple entries exist. If t' exists (hit), the returned data $d(t)$ is the same as $d(t')$. If no such entry exists (miss), t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$. Eventually, the $ld(t)$ event completes by being serviced by $M_p(t)$ which provides the data $d(t)$. Its guard 'Allowed' captures when tuple t, which is present in $RB_{p(t)}$, can be processed ahead of all the other tuples within $RB_{p(t)}$.

$st_{local}(t)$: results in t being enqueued into $SB_{p(t)}$ via procedure $Issue$.

$st_{global}(t)$ updates the memory array M_i from $WIB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $WIB_{p(t)}$, can be processed ahead of all the other tuples within $WIB_{p(t)}$.

$Fence(t)$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $SB_{p(t)}$ entry, where the entry comes from $p(t)$ and occurs earlier than t in program order. The functions **Allowed**($SB_{p(t)}$), **Allowed**($RB_{p(t)}$), **Issue**($Buffer, t$) and **Delete**($Buffer, t$) used in the transition system are same as that of the Generalized Weak memory model. Function Allowed($WIB_{p(t)}$ is now described.

Allowed($WIB_{p(t)}$,t): The function evaluates to true if $\neg\exists t' \in WIB_{p(t)}$ s.t $l(t') < l(t)$ and *Write Atomicity* (case 1 or 2) is required.

Although we designed a Generalized Weakest memory model, Hybrid memory models can also be designed with the same data structures and operational semantics but now due to multiple load and store instruction types, there will be additional rules pertaining to these new instructions, for example, reordering between st and $st.rel$ in SB and WIB. We now focus on the operational model for Itanium which can be referred to understand how our Generalized Weakest memory model can

be easily extended to handle Hybrid memory models. It also explains how subtle ordering rules pertaining to causality can be handled also.

4.3 A Hybrid Memory Model- Itanium

The Itanium memory model can be understood in terms of *program* and *global visibility* ("visibility") orders. For memory operations of type 'store', visibility refers to when the effects of the store become apparent to all processors. For memory operations of type 'load,' visibility refers to when the execution of load appears to have been carried out for the processor carrying out the load. (All other processors do not directly observe the load happening.) As in [54], for two different memory operations X and Y, $X \rangle\rangle Y$ specifies that X is before Y in program order, $X \rightarrow Y$ indicates that Y must be visible only after X is visible. Further, if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. Now, if X and Y are two memory operations in the same program, and $X \rangle\rangle Y$, Itanium requires the following:

1. If X is a load (store) and Y is a store (load) to the same location, WAR (RAW) hazards must be avoided.
2. If X and Y are stores to the same location, WAW hazards must be avoided. In addition, $X \rightarrow Y$.
3. If X and Y are memory operations to any location, and have a fence between them, then $X \rightarrow Y$.
4. If X is an Acquire load and Y is any other memory operation to any location, then $X \rightarrow Y$.
5. If X is any memory operation to any location and Y is a Release store, then $X \rightarrow Y$.

Under all other circumstances, X and Y can get executed in any order. [54] also asserts the following constraints on executions:

Coherence: There is a single visibility order (which is also a total order) of all stores per memory location observed by all the processors. Further, this total order is consistent with the program order for memory operations on that location in each processor,

RC_tso: Intuitively, *ld.acq* and *st.rel* are used to "bracket" instruction sequences, to permit more liberal execution orders for instructions in-between. To a crude approximation, *ld.acq* and *st.rel* are strongly ordered as in sequential consistency [2]. However, more precisely viewed, RC_tso [11] captures the orderings involving *ld.acq* and *st.rel* as per *Release Consistency* [2], with *ld.acq* and *st.rel* obeying TSO [22]. Under RC_tso, there is a single global visibility order of all Release Stores, with the exception that each processor may see (via ordinary or acquire loads) its own updates earlier than when other processors see it. Further, this global visibility order is a total order consistent with program order of all release stores in each processor.

Causality: When a *st.rel* instruction X in some processor P1 is read by an *ld.acq* instruction Y in another processor P2, then no store instruction following Y in program order must be visible to any processor before X is visible.

4.3.1 Litmus Test examples of Itanium memory model

The following examples (some from [54]) illustrate the Itanium ordering rules (assume that each memory location has value 0 in the beginning).

- The following execution is *invalid* due to Rule 2 pertaining to WAW, Rule 4 pertaining to Acquire, and the requirement of Coherence,

```
  P               Q
st(A,1)         ld.acq(A,2)
st(A,2)         ld(A,1)
```

- Fence (Rule 3) is illustrated by the following *invalid* execution. Here, `ld(B,0)` and `ld(A,0)` are seen after a fence, while the stores that supply new values into `A` and `B` are not getting flushed as is required by *fence*s:

```
  P               Q
st(A,1)         st(B,1)
Fence           Fence
ld(B,0)         ld(A,0)
```

• Acquire and Release (Rules 4 and 5) are illustrated by the following *invalid* execution. Here, `st(A,1)` precedes `st.rel(B,1)`, while `ld.acq(B)` precedes `ld(A)`. However, we see `ld(A,0)` happening instead of `ld(A,1)`.

```
   P              Q
st(A,1)        ld.acq(B,1)
st.rel(B,1)    ld(A,0)
```

• Coherence is illustrated by the following *invalid* execution. This is because the Acquire semantics forces the `ld` instructions to occur after the `ld.acq` instructions. However, processors `R` and `S` are observing the updates to `A` in different orders:

```
   P            Q          R            S
st(A,1)      st(A,2)   ld.acq(A,1)  ld.acq(A,2)
                       ld(A,2)      ld(A,1)
```

• RC_tso is illustrated by the following *valid* execution. The store of P into A is locally visible to P (say, via a cache or store buffer) before it becomes visible to all other processors (and similarly for Q and variable B): that

```
    P              Q
st.rel(A,1)    st.rel(B,1)
ld.acq(A,1)    ld.acq(B,1)
ld(B,0)        ld(A,0)
```

• Another aspect of release stores is that all the `st.rel` of all the processors taken together forms *a single global visibility order* that is also a total order. Considering this, the following outcome is *not valid*, because `Q` and `R` are observing `st.rel(A,1)` and `st.rel(B,1)` in different orders.

```
     P            Q            R            S
st.rel(A,1)  ld.acq(A,1)  ld.acq(B,1)  st.rel(B,1)
             ld(B,0)      ld(A,0)
```

• The following example violates the causality rule. The `st.rel(A,1)` of `P` is observed by `Q` via a `ld.acq`. Further, `st(B,1)` of `Q` is observed by `ld.acq` of `R`. Causality now requires that `st(B,1)` must be visible to `R` only after `st.rel(A,1)`. However, in this example, `R` sees a different order.

```
     P            Q            R
st.rel(A,1)  ld.acq(A,1)  ld.acq(B,1)
             st(B,1)      ld(A,0)
```

4.3.2 The Operational Model of Itanium

Apart from the four data structures as used in the Generalized Weakest Operational Model we use an additional data structure to account for causality as defined via a litmus test.It is a label-vector L_i held by each processor p_i. This is a vector of natural numbers, with each entry initialized to 0. Specifically, $L_i[j]$ holds the

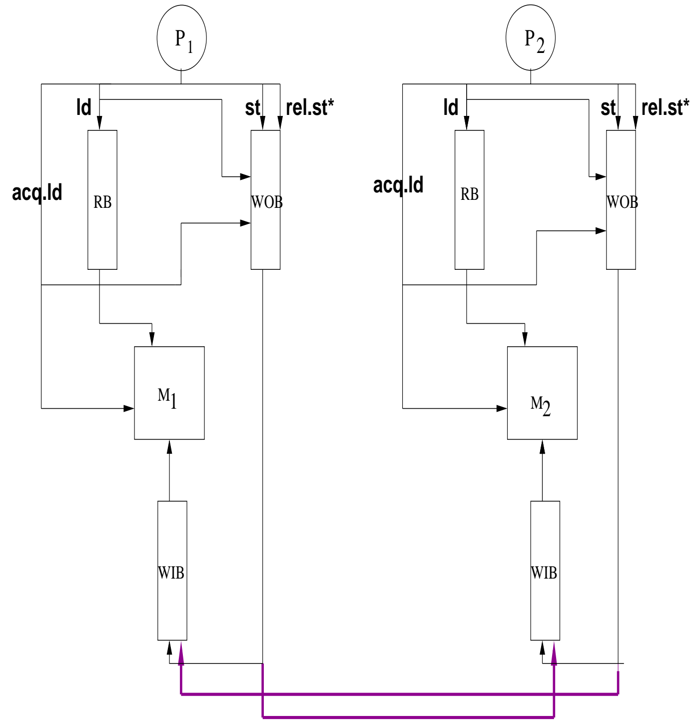

Figure 4.3. An Operational Model of Itanium

label of the last $st.rel$ instruction of p_j that has already been written into M_i. In other words, $L_i[j]$ indicates the (release-store) instruction of p_j upto which M_i has "caught up." To maintain this invariant, whenever any memory location in M_i gets updated by a release store operation represented by the tuple t, $L_{p(t)}[p(t)]$ gets set to the value $l(t)$, which is the label of the release instruction represented by t. Before an *st* instruction is enqueued into SB_i, the $v(t)$ field of this instruction is set to the current L_i value.

4.3.3 State Transition Rules of Itanium Operational Model

Table 4.3 defines the operational semantics of the Itanium shared memory model. The first column shows *Event*s that happen if the guard condition in the second column is true, performing the actions shown in the third column. At any time,

Event	Guard	Actions
$ld.acq(t)$	$\exists t' \in SB_{p(t)}$: $a(t') = a(t) \wedge d(t') = d(t)$ $\wedge\ t'$ youngest for address $a(t)$ **else** $M_{p(t)}[a(t)] = d(t)$ $\wedge$ $\neg\exists t' \in WIB_{p(t)}$: $a(t') = a(t) \wedge p(t') = p(t)$	none
$ld(t)$	$\exists t' \in SB_{p(t)}$: $a(t') = a(t) \wedge d(t') = d(t)$ $\wedge\ t'$ youngest for address $a(t)$ else True	Issue($RB_{p(t)}$, t)
st.rel(t)	True	Issue($SB_{p(t)}, t$)
$st(t)$	True	$t \leftarrow$ t[$L_{p(t)}/v$]; Issue($SB_{p(t)}, t$)
$Fence(t)$	True	Flush(t)
$MW(t)$	$t \in WIB_{p(t)}$ $\wedge$ Allowed($WIB_{p(t)}, t$)	$M_{p(t)}[a(t)] \leftarrow d(t)$; Delete($WIB_{p(t)}, t$); if($o(t) = st.rel$) then $L[p(t)] = l(t)$
$MR(t)$	t$\in RB_{p(t)}$ $M_{p(t)}[a(t)] = d(t)$ $\wedge \neg\exists t' \in WIB_{p(t)}$: $a(t') = a(t) \wedge p(t') = p(t)$	Delete($RB_{p(t)}, t$)
$Del(t)$	true	ProcSB($SB_{p(t)}, t$)

Table 4.3. Transition System of Itanium Operational Model

any one of the eligible events may be picked in a *fair* manner. Each event happens when the next instruction t is issued by processor $p(t)$ (for events $ld.acq(t)$ through $Fence(t)$), or when an instruction is removed from one of the internal buffers and is carried out (for events $MW(t)$, $MR(t)$, and $Del(t)$). Notice that in case of events $ld.acq(t)$, $ld(t)$, as well as $MR(t)$, tuple t carries the data $d(t)$ being returned (following the convention used in [39]). When these events fire, a constraint expressed in the Guard field shows what this data is. We use = for equality testing, and $\leftarrow$ for assignment.

$\underline{ld.acq(t)}$: If the next instruction tuple t of processor $p(t)$ is a $ld.acq$, we perform the $ld.acq(t)$ event. We seek an entry $t^{'}$ in $SB_{p(t)}$ such that $a(t^{'}) = a(t)$, and $t^{'}$ is the youngest such entry, if multiple entries exist. If $t^{'}$ exists, the returned data $d(t)$ is the same as $d(t^{'})$. If no such entry exists ("else"), $ld.acq$ must get serviced from the memory $M_{p(t)}$, and that too, only when there is *no* tuple $t^{'}$ in the $WIB_{p(t)}$ buffer such that $p(t^{'}) = p(t)$ and $a(t^{'}) = a(t)$. The condition $p(t^{'}) = p(t)$ prevents a $ld.acq$ from bypassing an earlier issued st or $st.rel$ on the same address.

$\underline{ld(t)}$: As with $ld.acq(t)$, the $ld(t)$ event is serviced directly by $SB_{p(t)}$ upon a 'hit'; otherwise, t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$.

$\underline{st.rel(t)}$: results in t being enqueued into $SB_{p(t)}$ via procedure Issue.

$\underline{st(t)}$ first updates the v field of tuple t with the label vector $L_{p(t)}$ (shown by t ← t[$L_{p(t)}/v$]), and then enqueues the resulting tuple t into $SB_{p(t)}$ via procedure Issue.

$\underline{Fence(t)}$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $SB_{p(t)}$ entry, and every WIB_j entry for all j, where the entry comes from $p(t)$ and occurs earlier than t in program order.

$\underline{MW(t)}$ updates the memory array $M_{p(t)}$ from $WIB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $WIB_{p(t)}$, can be processed ahead of all the other tuples within $WIB_{p(t)}$. This is precisely when there isn't an older $WIB_{p(t)}$ entry $t^{'}$ *and* **one** of the following four conditions hold: (i) $a(t) = a(t^{'})$, (ii) both t and $t^{'}$ are $st.rel$, (iii) both come from $p(t)$ with $o(t) = st.rel$, (iv) the label of $t^{'}$ matches $v(t)[p(t^{'})]$, which is the label of the last $st.rel$ from $p(t^{'})$ seen by $p(t)$, $o(t^{'}) = st.rel$, and $o(t) = st$. Condition (iv) blocks the st from happening until after $M_{p(t)}$ also has assimilated $t^{'}$, ensuring causality. When event $MW(t)$ fires, $M_{p(t)}$ is first updated, and tuple t is then deleted from $WIB_{p(t)}$ by procedure Delete. Also, if the operation of tuple t is $st.rel$, the label-vector $L_{p(t)}$ is updated to the label $v(t)$ carried by tuple t to record the release-store upto which $M_{p(t)}$ has caught up.

$\underline{MR(t)}$ represents when a tuple t buffered in $RB_{p(t)}$ (corresponding to an ld instruction) gets serviced. This event is allowed when memory array $M_{p(t)}$ holds $d(t)$ at

address $a(t)$, and there is no t' in $WIB_{p(t)}$ with a matching address from the same processor.

$\underline{Del(t)}$ calls procedure ProcSB which first checks if $o(t)$=st and there is an entry t' in $RB_{p(t)}$ with address $a(t)$, *or* if $o(t) = st.rel$ and there is an entry t' in either $RB_{p(t)}$ or $SB_{p(t)}$ with a lower label. If neither, ProcSB deletes t from SB, copying it atomically into every WIB. The functions used in the transition system are now described.

```
 Flush(t): WHILE ∨ (len(SB_p(t))  >  0)
         ∨ (len(RB_p(t))  >  0)
         ∨ (∃ i,  t'∈WIB_i : p(t)=p(t') ∧ l(t')<l(t))
DO FOR t'' ∈ RB_p(t) DO an MR(t'') event
   FOR t'' ∈ SB_p(t) DO ProcSB(SB_p(t),t)
   FOR t'' ∈some WIB_i where p(t)=p(t'') ∧ l(t'')<l(t)
                                   DO MW(t'')
END WHILE

 ProcSB(SB_p(t),t):
IF ∨(o(t) = st ∧ ¬∃ t'∈{RB_p(t),SB_p(t)}:
              a(t) = a(t') ∧  l(t') < l(t))
   ∨(o(t) = st.rel ∧ ¬∃ t'∈{RB_p(t),SB_p(t)}:
              l(t')<l(t))
THEN
 Delete t from SB_p(t);
 FOR all i DO Issue(WIB_i,t)
END IF

 Allowed(WIB_p(t),t): ¬∃ t' ∈  WIB_p(t) :
t' older than t ∧
```

$$
\begin{aligned}
(\vee(a(t) &= a(t')) \\
\vee(o(t) &= o(t') = st.rel) \\
\vee(p(t) &= p(t') \wedge o(t) = st.rel) \\
\vee(l(t') &= v(t)[p(t')]) \\
&\wedge \\
&(o(t') = st.rel \wedge o(t) = st) \\
)
\end{aligned}
$$

Issue$(Buffer, t)$: Add t to the tail of Buffer as in a FIFO queue.

Delete$(Buffer, t)$: Here, Buffer is either RB or WIB. This procedure deletes t wherever it may be in Buffer.

4.3.4 Analysis of Itanium Operational Model

We now show how our operational model meets the requirements of the Itanium specification.

1. RAW for load operation r and store operation w earlier in program order: (i) If r is satisfied when it hits a w in SB (Table 4.3, event $ld.acq(t)$ or $ld(t)$), RAW is satisfied. (ii) If r is satisfied from memory, in case w has already been written into memory, the RAW hazard is avoided. If however w is in WIB hence blocks r (which is in RB) from issuing, we freeze r till w is written into the memory (Table 4.3, event $MR(t)$). Thus, here also the RAW hazard is avoided.

2. WAR for load r and store w from the same processor: Note that w cannot move from SB to WIB until the load is drained from WIB (see ProcSB). Hence, WAR hazards are avoided.

3. WAW, as well as visibility order for stores to the same location are guaranteed as follows. If there are two stores to the same address in SB, event $Del(t)$

removes them in the oldest-first order. If the second store comes while the first has gone into WIB, then the "t' issued before t" check in function Allowed prevents a younger write from overtaking an older one.

4. Fence: Procedure Flush carries out all "preceding" instructions before allowing instruction issuing to resume. Hence Rule 3 is obeyed.

5. Acq: Hazard aspects of Acq have already been covered. Since Acq blocks further instruction issuing till it gets carried out (see event $ld.acq(t)$), Rule 4 pertaining to visibility is satisfied.

6. Rel: Loads that come before $st.rel$ are handled by ProcSB that checks for loads with lower labels. Stores before $st.rel$ are also checked in a similar manner. A $st.rel$ that enters WIB when there is another store in WIB from the same processor is prevented from reordering by function Allowed. This meets Rule 5.

7. Coherence: The rules for handling SB and WIB ensure Coherence.

8. RC_tso: Handling of SB and WIB ensure a total global visibility order of release stores. The "TSO" aspect of RC_tso comes naturally because each processor may see its own update early via the SB, exactly as in classical TSO [22].

All rules except for causality have been discussed. We now discuss causality in some detail. Causality can be summarized at a high level as follows: "Before any st operation o is posted into any M_i, ensure that every $st.rel$ operation r that o is "causally dependent upon" has already been updated into M_i. "Causally dependent on" means o was issued by some p_j after it had updated its own store M_j with the value provided by r.

Causality is obeyed to a certain extent. Specifically, if a $st.rel$ satisfies a $ld.acq$ instruction then all subsequent store operations following that $ld.acq$ instruction in

program order will be visible to all processors after that *st.rel* operation. It suffices to prove this condition by proving that if X is a *st.rel* from any processor $p(X)$ satisfying Y which is a *ld.acq* in processor $p(Y)$, Z is a *st* to any memory address in $p(Z)$ where $Y \rangle\rangle Z$ (hence $p(Y) = p(Z)$), and $X \to Y$ in $p(Y)$, then $X \to Z$ for *any* processor p_k. Since $X \to Y$,

- X must have been updated in M_Y by the time Y is carried out,
- the label vector $v(Z)$ must reflect the update of X, *i.e.*, $v(Z)[p(X)] \geq l(X)$, and
- for any other processor p_k, either X is updated in M_k or else it resides in WIB_k. When Z gets issued to all WIB buffers, and in particular WIB_k, it cannot participate in the $MW(t)$ event before X can do so, due to the behavior of function Allowed.

As an example, consider the earlier discussed example, now with labels:

```
     P             Q              R
1:st.rel(A,1) 1:ld.acq(A,1)  1:ld.acq(B,1)
              2:st(B,1)      2:ld(A,0)
```

The label vector carried by instruction `st(B,1)` would be `[1,0,0]` because `Q` would have seen the `st.rel(A,1)` instruction of `P` situated at label `1` when it issues `st(B,1)`. If `st.rel(A,1)` still resides in the WIB_R buffer when `st(B,1)` also enters WIB_R, function Allowed ensures that the former is posted into M_R before the latter. Thus, `ld(A,0)` is impossible in `R`.

4.3.4.1 Ordering Relaxations

```
P1
st(a,1)
st.rel(b,1)
st(c,1)
```

We now discuss a few examples of *ordering relaxations* correctly supported by our model.Releases can be bypassed by subsequent operations. Moreover, these

operations may bypass operations preceding release. In the following program, `st(c,1)` can bypass both `st.rel(b,1)` and `st(a,1)`. This is supported by our operational model as follows. Suppose these instructions are in SB. ProcSB will consider `st(c,1)` as well as `st(a,1)` eligible for movement into WIB, because, for `st` instructions, the label comparisons are done address-wise. However, ProcSB will *not* be able to move `st.rel(b,1)` into WIB before it moves `st(a,1)`, because for release stores, label comparisons are across all addresses.

Itanium is not required to provide any global total order for `st` instructions. In this example,

```
  P1       P2         P3           P4
st(a,1)  st(b,2)  ld.acq(a,1)  ld.acq(b,2)
                  ld(b,0)      ld(a,0)
```

it allows `P3` to see `st(a,1)` before `st(b,2)` and vice versa in `P4`. This relaxation is supported by function Allowed. Suppose `st(a,1)` and `st(b,2)` are both in WIB_{P3} and WIB_{P4} in some order. Function Allowed can pick `st(a,1)` to post first in M_{P3}, while it can pick `st(b,2)` to post first in M_{P4}.

4.3.4.2 How $R \rangle\rangle R$ may impact causality

It is unclear by reading [54] whether the following execution is legal or not:

```
   P1              P2          P3       P4
ld(A,1)       st.rel(A,1)  st(A,2) ld.acq(B,1)
ld.acq(A,2)                        ld(A,0)
st(B,1)
```

If the instructions `ld(A,1)` and `ld.acq(A,2)` are ordered because they are loads on the same location, **then** the following consequences of causality emerge. We have `st.rel(A,1)` being ordered before `ld.acq(A,2)` in the visibility order of `P1`. Due to the acquire semantics, `st(B,1)` is performed after `ld.acq(A,2)`. The situation is quite analogous to the Causality example on Page 42, except the causal chain forms through a load-to-load order. Now, since `st(B,1)` is observed by `ld.acq(B,1)`, we cannot have `ld(A,0)` in `P4` due to causality. It is unknown to us whether load-to-load orderings such as between `ld(A,1)` and `ld.acq(A,2)` are to be obeyed, and if so must cause causal chains in this fashion.

4.3.5 Tool User Guide to specifying memory models

In our tool the user defines a memory model of his choice using our specification framework as explained in Chapter 3 and also enters the assembly language program he wants to run and check all possible outcomes w.r.t his specified model. The PROMELA code (input language of SPIN) for both the executable specification as well as the assembly program is then generated. The SPIN verifier is then used to run the assembly program on the executable specification and all possible outcomes of the program are delivered as output in terms of the data value returned to the load instructions.

The tool is available at http://www.cs.utah.edu/formal_verification/ESGtool.

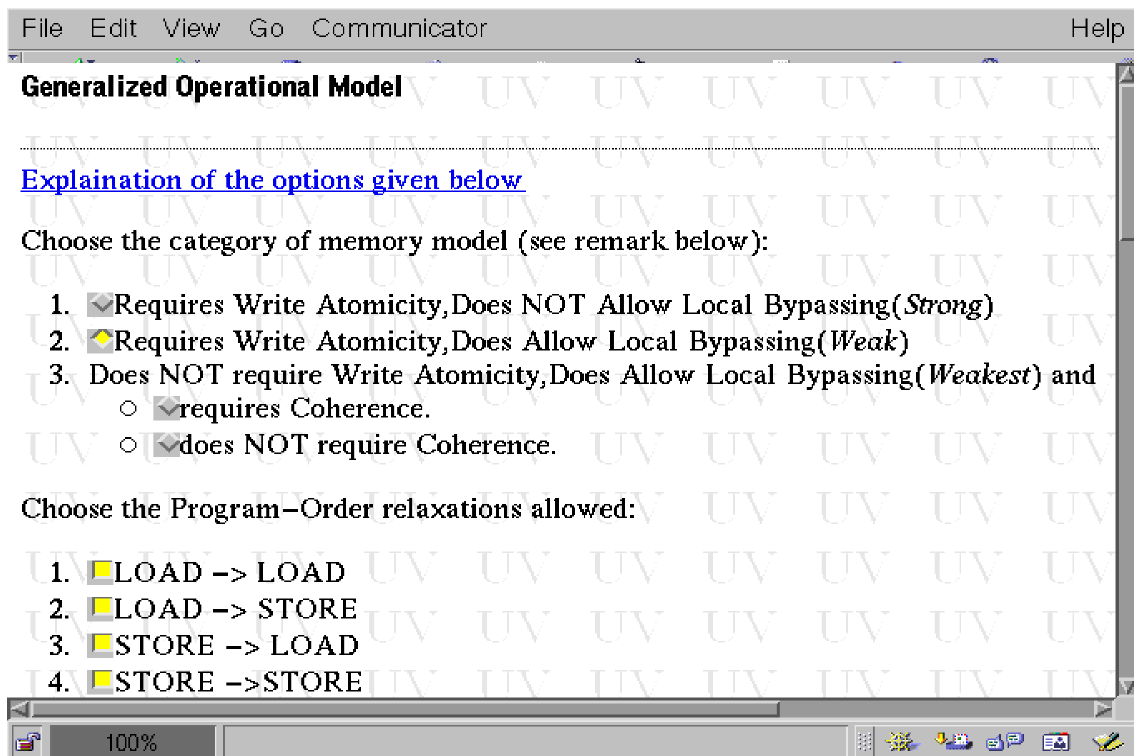

Figure 4.4. Memory model specification tool interface- specifying Alpha memory model

Suppose the user wants to define the Alpha memory model. He can specify that by first stating that Alpha falls under the Weak memory model class as it requires Write Atomicity and allows local bypassing. Among the *Per Processor Order* sub-rules only Memory Data Dependence and *Fence* is required. The sub-rules that are not required are $ld \rightarrow ld$, $ld \rightarrow st$, $st \rightarrow ld$, $st \rightarrow st$. The user specifies these requirements by simply stating the sub-rules he **does not** require. For Alpha he thus clicks on the sub-rules $ld \rightarrow ld$, $ld \rightarrow st$, $st \rightarrow ld$, $st \rightarrow st$ as they are not required.

The next step is for the user to enter an assembly level concurrent program and check all possible outcomes of such a program with respect to his defined memory model. For example, suppose he wants to check all possible outcomes of the following code with respect to the Alpha memory model.

```
    P               Q
st(A1,1)        r1 = ld(A2)
st(A2,2)        r2 = ld(A1)
```

The user can enter this code in a format as shown in Figure 4.5. All possible outcomes of the user entered program in terms of the date values returned to the registers are outputted as shown in Figure 4.6.

The final outcome agrees with the Alpha memory model which does not require *Per Processor Order* sub-rules $st \rightarrow st$ and $ld \rightarrow ld$. Hence, the store instructions in processor "P1" can complete in any order and the load instructions in processor "P2" can complete in any order respectively.

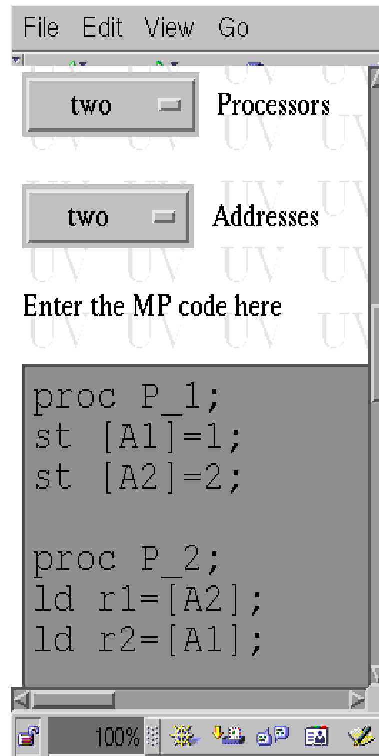

Figure 4.5. Assembly level concurrent program as typed by user

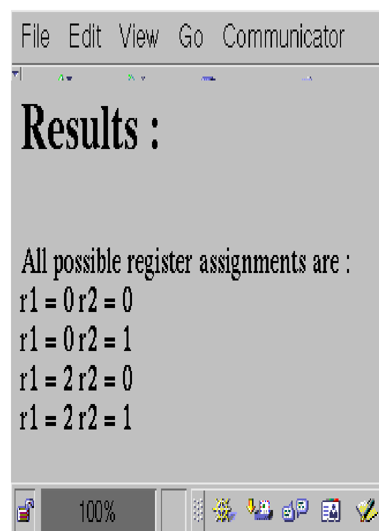

Figure 4.6. Tool Output - All possible outcomes of the litmus test w.r.t. Alpha memory model

CHAPTER 5

FORMAL VERIFICATION THROUGH AUTOMATED REFINEMENT CHECKING

The breakup of this chapter is as follows:

1. We first specify the Itanium shared memory consistency model (*Spec*) in our own definitional framework.

2. We present a shared memory multiprocessor implementation *Imp* that claims to satisfy the Itanium shared memory consistency model *Spec*.

3. We then demonstrate how to derive the simplified model Imp_{abs} from *Imp*.

4. Finally, we explain how our Automated Refinement Technique works on the above example where we prove that the *Imp* refines Imp_{abs}(Appendix B provides a proof-sketch that Imp_{abs} satisfies the *Spec*).

We then illustrate how the creation of abstract models is influenced by advanced optimizations such as Scheurich's optimization [27]. We offer guidelines for the creation of abstracted models by presenting a taxonomy of shared memory consistent models and discussing the kinds of abstract models one can create with respect to this taxonomy.

5.1 Phase 1 of our Verification Method

This entire Section explains and applies **Phase 1** of our Verification Method in context of a specific example where we verify that an Itanium Implementation refines its corresponding Abstract model.

5.2 Itanium memory model specification

We now summarize a subset of the Itanium memory model specification, from [54] and [11].

The approach presented in this thesis only deals with cacheable memory instructions, and that too those consisting of *acquire loads* (written *ld.acq*), *ordinary loads* (*ld*), *release stores* (*st.rel*), as well as memory fences. Handling ordinary (non-atomic) stores, atomic read-modify-writes, non-cacheable memory locations, and special rules pertaining to register data dependencies [54, Section 13.2] will be part of future work. Even with these limitations, our technique can handle memory models such as TSO [22], the Alpha memory model [45], or a subset of the Itanium memory model.

All instructions except $st.rel$ are decomposed into exactly one event. Each $st.rel$ is decomposed into a $st.rel_{local}$ and a $st.rel_{global}$. An *execution* obeys a memory model if all the memory events of the *execution* form at least one total order which obeys:

- the *Per Processor Order* stipulated by the memory model, and
- the *Read Value* Rule stipulated by the memory model. In addition,
- the $st.rel_{global}$ events must appear atomically to all processors.

The Read Value Rule specifies the data value to be returned by the load events in an execution. The Per Processor Order includes both program order as well as data dependence order. The fact that $st.rel_{global}$ operations are atomic is modeled by generating only "one copy" of a $st.rel_{global}$ event corresponding to each $st.rel$ instruction, and situating the $st.rel_{global}$ events in the total order '$\rightarrow$'. Since Itanium allows local bypassing, we split any store instruction t into two events t^{local} and t^{global} (and also create the corresponding tuples) where $o(t^{local})=st.rel_{local}$ and $o(t^{global})=st.rel_{global}$.

More specifically, an execution satisfies the Itanium memory model if there exists a logical total order '$\rightarrow$' of all the load (ld or $ld.acq$), $st.rel_{local}$ and $st.rel_{global}$ events and memory fence events present in the execution, such that '$\rightarrow$' satisfies the following clauses:

1. *Per Processor Order*: Let t_1 and t_2 be two events s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$ (t_1 appears earlier in program order than t_2).

   (a) If either

      i. t_1 is a $ld.acq$ event, or

      ii. t_2 is a $st.rel_{global}$ event, or

      iii. $a(t_1) = a(t_2)$ and,

         A. $o(t_1) = st.rel_{local}$, $o(t_2) = ld$, or

         B. $o(t_1) = ld$, $o(t_2) = st.rel_{local}$, or

         C. $o(t_1) = st.rel_{local}$, $o(t_2) = st.rel_{local}$,

      then $t_1 \rightarrow t_2$.

   (b) If there exists a fence(mf) instruction t_f s.t. $l(t_1) < l(t_f) < l(t_2)$ then $t_1 \rightarrow t_2$.

   (c) If $o(t_1) = o(t_2) = st.rel_{global}$ then $t_1 \rightarrow t_2$.

2. *Read Value*: This definition follows the style in which *Read Value* is defined in [23] for TSO. Formally, let t_1 be a load (ld or $ld.acq$) event. Then the data value of t_1 is the data value of either the most recent local or the most recent global store event (in the total order relation $\rightarrow$) to the same memory location as t_1. *i.e.*,

   (a) if

      i. $a(t_1) = a(t_2)$, $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ and

ii. there does not exist a $st.rel$ instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$ and $t_2^{local} \rightarrow t_3^{local} \rightarrow t_1$.

then $d(t_1) = d(t_2^{local})$;

(b) else if

i. $a(t_1) = a(t_2)$, $t_2^{global} \rightarrow t_1$ and

ii. there does not exist a $st.rel$ instruction t_3 s.t $a(t_1) = a(t_3)$ and $t_2^{global} \rightarrow t_3^{global} \rightarrow t_1$.

then $d(t_1) = d(t_2^{global})$;

(c) else, $d(t_1)$ is the "initial memory value" (taken to be 0 in the thesis).

Since all the global store events t ($o(t) = st.rel_{global}$) form a single total order within $\rightarrow$, we naturally end up modeling the fact that these global stores appear atomically to all processors.

5.3 Itanium implementation model

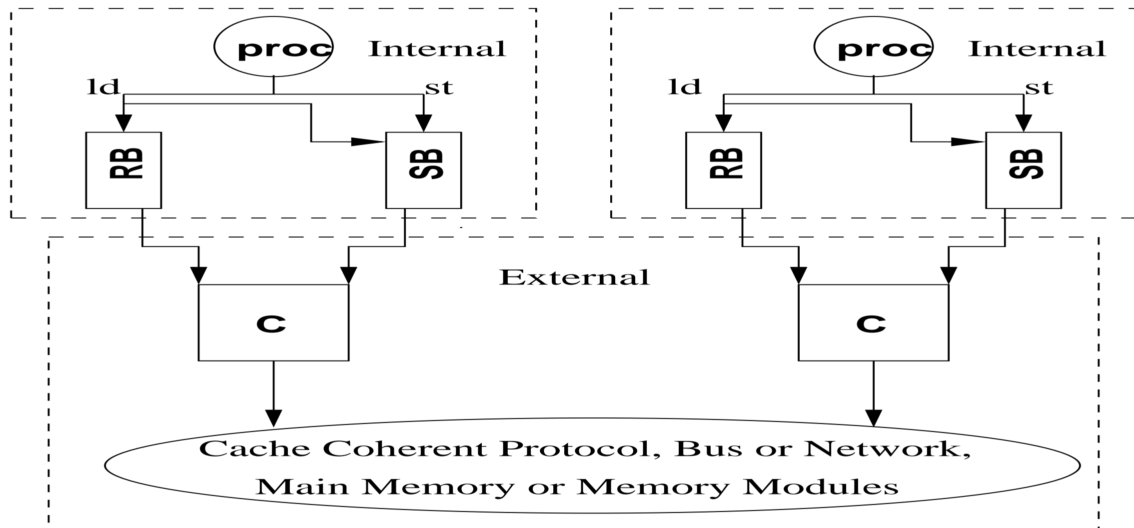

Figure 5.1. An Itanium Implementation

In our Itanium implementation we separate each processor from its cache (situated in the external partition) with a FIFO store buffer SB and a re-order read buffer RB (situated in the internal partition). Caches are kept coherent with a write-invalidate coherence protocol [10] details of which is explained in the next chapter. The data structure of caches is a two dimensional array C where, for event t, $C[p(t)][a(t)].a$ refers to data value of address $a(t)$ at processor $p(t)$ and $C[p(t)][a(t)].st$ refers to its address state (A-state). We begin with a brief explanation of the cache coherence protocol we use. This protocol is the same as the one used in [24] to describe a Gigaplane-like split-transaction bus. Table 5.1 summarizes the protocol transactions. Memory blocks may be cached Invalid(I),

trans-action	initial state of cache	initial state of other cache(s)	provider of data to cache	final state of other cache(s)	final state of cache
GETX	I	I	memory	I	E
GETX	I	S	memory	I	E
GETX	I	E	old owner	I	E
GETS	I	I	memory	I	S
GETS	I	S	memory	S	S
GETS	I	E	old owner	S	S
UPG	S	I		I	E
UPG	S	S		I	E
WB	E	I		I	I
PUTS	S	I		I	I
PUTS	S	I		S	I

Table 5.1. Protocol Transactions

Shared(S), or Exclusive(E). The A-state (address state) records how the block is cached and is used for responding to subsequent bus transactions. The protocol seeks to maintain the expected invariants (e.g, a block is Exclusive in at most one cache) and provides the usual coherent transactions: Get-Shared (GETS), Get-Exclusive (GETX), Upgrade (UPG, for upgrading the block from Shared to Exclusive) and Writeback (WB). As with the Gigaplane, coherence transactions

immediately change the A-state, regardless of when the data arrives. If a processor issues a GETX transaction and then sees a GETS transaction for the same block by another processor, the processor's A-state for the block will go from Invalid to Exclusive to Shared, regardless of when it obtains the data. The processor issues all instructions in program order. Below, we specify exactly what happens when the processor issues one of these instructions.

1. *st.rel*: A *st.rel* instruction first gets issued to SB, completing the $st.rel_{local}$ event. Entries in the buffer are the size of processor words.

   Deletion of entries from the head of the SB happen as follows. The processor first makes sure there is no earlier issued ld instruction pending in RB (if any, those RB instructions must be completed before deleting an entry from SB). It then checks if the corresponding block's A-state is Exclusive(E). If not, the coherence protocol is invoked to change the A-state to E. Once in E state, the entry is atomically deleted from SB and written into the cache, thus completing the $st.rel_{global}$ event.

2. *ld.acq*: To issue a *ld.acq* instruction, the processor first checks in its SB for a *st.rel* instruction to the same word. If there is one, the *ld.acq* gets the value of the youngest such $st.rel_{local}$ in program order. If there is none, the processor proceeds to first establish that the A-state of the block in the cache is Shared or Exclusive, if necessary invoking the coherence protocol (the details of which are as described in [10]). The *ld.acq* event completes once the cache has acquired E or S state and the data has been received.

3. *ld:* Issuing a *ld* instruction is similar to that of a *ld.acq* which first tries to hit in the SB and on a miss tries to get its data from the cache provided its in E or S state. However, if not in E or S state, it gets buffered in RB. In future, when in E or S state, that entry in RB gets its data from cache. Entries from

RB can be deleted in any order since ordering among ld instructions is *not* required by the specification.

4. *mf*:Upon issuing a *mf* instruction, all entries in SB are flushed to the cache and all entries in RB are deleted after returning their values from cache, hence completing the corresponding *mf* event[1]. While flushing an entry from SB, the processor checks that there is no earlier issued ld instruction residing in RB. We call this entire process as $flush_{imp}$. This implementation of mf is similar to the Sun Ultra Enterprise 6000 with UltraSparc II processors.

5.4 The Intermmediate Abstraction

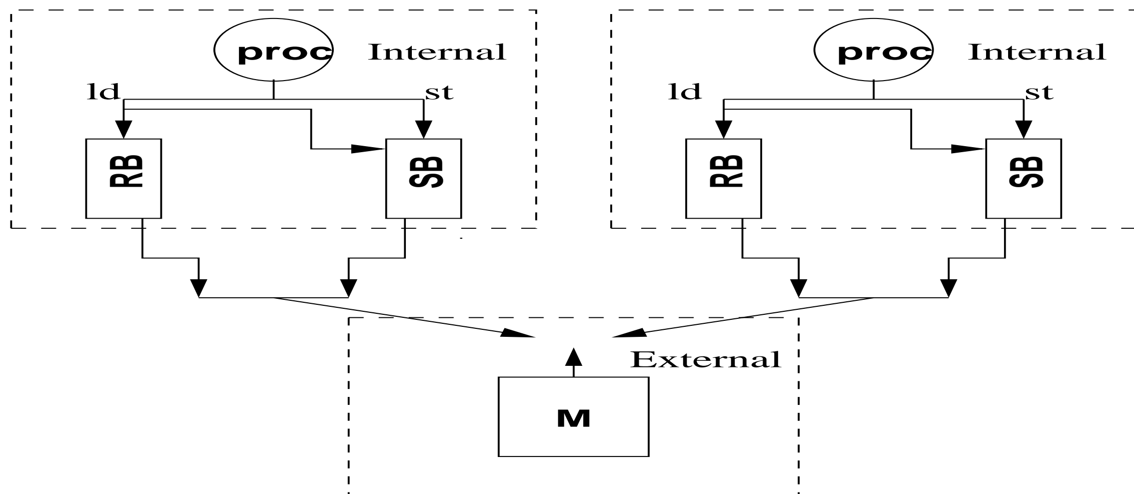

Figure 5.2. The derived Itanium Intermmediate Abstraction

The Itanium abstract model retains the internal data partition of the implementation *without any changes*. However, the cache, the cache coherent protocol, bus and main memory in the implementation which belong to the external partition are all replaced by a single port main memory M in the abstract model. This replace-

[1]Appropriate cache entries need to be in E state before flushing

ment (justified in Chapter 3) is because of write atomicity obeyed by the Itanium model. We now take a look at how each of the instructions get implemented. As with the implementation, the processor issues all instructions in program order.

1. *st.rel*: A *st.rel* instruction first gets issued to SB, completing the $st.rel_{local}$ event. At any time, provided there is no earlier issued *ld* instruction pending in RB, the entry at the head of SB can be atomically written to the single port memory M, completing the $st.rel_{global}$ event.

2. *ld.acq:* Similarly, as in implementation, a *ld.acq* instruction tries to hit SB but on a miss, it receives its data from M atomically.

3. *ld:* Similarly, as in implementation, a *ld* instruction tries to hit SB and on a miss, it gets buffered in RB. However, any entry in RB can be deleted once it receives its data from M, both the steps being performed in one atomic step.

4. *mf*: Upon issuing a *mf* instruction, all entries in SB are flushed to M and all entries in RB are deleted after returning their values from M. While flushing from SB the processor checks that there is no earlier issued load event residing in RB. We call this entire process as $flush_{abstract}$

5.5 Automatic Refinement Checking

The events $st.rel_{local}, st.rel_{global}, ld.acq, ld$ and mf have been defined for both the implementation and the abstract model. Every event of the implementation is composed of multiple steps. However, in the abstract model each event except *ld* is composed of a single atomic step. For example, for a *ld.acq* event to complete, if there is a miss in SB and the concerned address's A-state is Invalid, the processor will need to send a request on the bus to get a shared copy, and wait till it receives

[2]Here $d(t) \leftarrow C[p(t)].[a(t)].a$ refers to the load instruction t receiving its data from the updated cache entry

Event	Implementation	Operational Model
ld.acq(t) ($SB_{p(t)} hit$)	read from $SB_{p(t)}$	read from $SB_{p(t)}$
ld.acq(t) ($SB_{p(t)} miss$)	$C[p(t)][a(t)].st$=S or E and $d(t) \leftarrow C[p(t)][a(t)].a$ [2]	$d(t) \leftarrow M[a(t)]$
ld(t) ($SB_{p(t)} hit$)	read from $SB_{p(t)}$	read from $SB_{p(t)}$
ld(t) ($SB_{p(t)} miss$)	Issue to $RB_{p(t)}$; $C[p(t)][a(t)].st$=S or E and $d(t) \leftarrow C[p(t)][a(t)].a$	Issue to $RB_{p(t)}$; $d(t) \leftarrow M[a(t)]$
$st.rel_{local}(t)$	Issue($SB_{p(t)}, t$)	Issue($SB_{p(t)}, t$)
$st.rel_{global}(t)$	$C[p(t)][a(t)].st$=E and $C[p(t)][a(t)].a \leftarrow d(t)$	$M[a(t)] \leftarrow d(t)$
$mf(t)$	$flush_{imp}$	$flush_{abstract}$

Table 5.2. Completion steps of all events of implementation and abstract model

the data. During this process many intermediate steps take place which include other processors and main memory reacting to the request. However, in the abstract model, on a miss while handling the *ld.acq* event, the data value can simply be read off the single port memory.

5.5.1 Synchronization scheme

The discovery of the synchronization sequences between the implementation and the specification is the crux of our verification method. In manual approaches to verification, this step is very labor intensive. *In contrast, in our approach, a model-checker automatically "discovers" these sequences, if the implementation is correct, or flags an error pinpointing exactly where the synchronization breaks down.*

Table 5.2 provides an overview of the overall synchronization scheme. This table compares the completion steps of both the implementation and the abstract model, and highlights all synchronization points. Let us briefly elaborate the actions taken for *ld.acq* upon an SB miss. In the implementation, coherence actions are first invoked to promote the cache line into an Exclusive or Shared state. Thereafter, the

implementation receives data from the bus and at this point completes the *ld.acq* event. At this point, the model-checker will immediately make the same event complete in the abstract model by simply returning the data from $M[a(t)]$ through the multiplexor switch. Synchronization happens if the same datum is returned. In general, the last step that completes any event in the implementation and the single step that completes the same event in the abstract model are performed atomically.

The synchronization scheme for instructions that may get buffered and get completed later are slightly more elaborate. Basically, synchronization must be performed both when the instruction is entered into the buffer and later when they complete. For example, since a ld instruction may miss the SB and hence may not complete immediately, we will have to synchronize both the models when ld gets buffered, and finally synchronize again when the ld event completes. The synchronization of mf is accomplished indirectly, by the already existing synchronizations between the models at ld or $st.rel_{global}$. This is because an mf completes when the above instructions occurring before it complete.

We use the explicit state model checker Murϕ to model both the implementation and the abstract model for two processors, two memory locations, and two data values. Writing the verification property is straightforward: *the invariant verified was that whenever a ld.acq or ld event completes in both the implementation and the abstract model, the data values returned in both cases are identical.*

Our experimental results are shown in Table 6.2 in the next chapter.

5.6 Handling aggressive optimizations

Our Itanium abstract model can be re-used for a large family of cache coherent protocols. However, an optimization described by Scheurich requires our abstract model to be modified. With a standard invalidation-based directory protocol (like that of Stanford DASH), a store request to the directory that finds many read-only copies outstanding causes an invalidation message to be sent to each copy and then

acknowledged. Scheurich observed that processors can queue invalidations, send acknowledgements, and then perform invalidations, as long as the processors are "isolated" from the rest of the system. In our implementation, a $st.rel_{global}$ event completes when its written to cache and hence it would mean that this event is now visible to all processors. With Scheurich's optimization, even after writing the data to cache, a *ld.acq* or *ld* from some other processor to the same address can read stale data by delaying the invalidation of their shared copies.

Consider this situation where a $st.rel_{global}$ event completes by writing to cache. By our original scheme we we will immediately make the abstract model synchronize by writing the data to single port memory. But for the optimized implementation we can still read stale data value for the same address but this is not possible for the abstract model hence giving rise to a false negative. Hence, to disallow such a situation we need to modify our abstract model to a one that that can account for such an optimization.

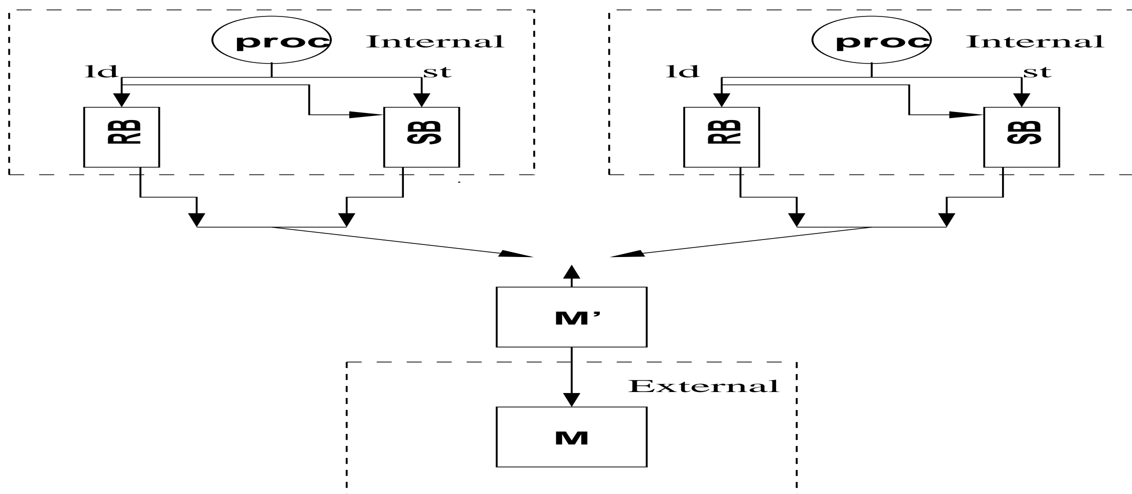

Figure 5.3. Modified Itanium abstract Model

The completion step of a $st.rel_{global}$ in the optimized implementation is not when it writes to cache, but rather when it is in the Exclusive A-state and also all the other caches have invalidated their shared copies. We modify our abstract model by inserting another "transient" single port memory M' in between the buffer and the original single port memory as shown in Figure 4 to model this situation. When a $st.rel$ gets written to cache but there are shared copies still not invalidated, we synchronize by making that event write to M' and keep track that the concerned address in this memory is visible only to the processor that issued it. Whenever all shared copies get invalidated, the entry in M' gets deleted and then copied to the original memory, completing the $st.rel_{global}$ event. With these changes, the abstract model can be used for refinement checking in the manner in which we have already described. The results are summarized in Table 6.2 in the next chapter. We believe that these manual steps in modifying the external partition of an abstract specification can also be cataloged and reused, as the optimization techniques employed in one architecture tend to be used in others also.

5.7 Derivation of Intermmediate Abstraction

In our verification methodology, the abstract model always retains, without change, the internal partition of the implementation. However, the *external* partition is considerably simplified. Designer insight is required in the selection of a simplified external partition, as this depends on the memory model under examination. In Chapter 4 we categorized memory models into four classes and showed how a common external partition can be derived for memory models belonging to a particular memory model class, thus providing a systematic approach to deriving the abstract model. Depending upon the category a memory model falls under, we split a store instruction into one or more events. Load instructions for any memory model can always be treated as a single event. Here are a few examples of splitting events. In case of Sequential Consistency, we do not split even the

stores as sequential consistency demands a single global total order of the loads and stores. For a weak memory model such as the Ultra Sparc TSO, we split the store instruction into two events, a local store event (which means that the store is only visible to the processor who issued it) and a global event (which means that the store event is visible to all processors). Since the *Weakest* category of memory models lack write atomicity, we need to split stores into $p+1$ events, where p is number of processors, thus ending up with a local store event and p global events (global event i would mean that the store event is visible to processor i). Table 5.3 summarizes these splitting decisions for various memory models. It also shows the nature of the external partition chosen for various memory models.

Memory Model	Splitting of store instructions	External Partition
Strong	store unsplit	single port memory
Weak	store split to local and global	single port memory
Weakest	store split to local and $(p+1)^3$globals	memory and re-order buffer per processor
Hybrid	store split to local and $(p+1)$ globals	memory and re-order buffer per processor

Table 5.3. Splitting of store and external partition for each memory model class

In case of *Strong* and *Weak* memory models, the external partition is just a single port memory M. The intuition behind having M is that both these classes of memory models require *Write Atomicity* and hence a store instruction should be visible to all processors instantaneously. *Weakest* and *Hybrid* memory models require more involved data structures where each processor i has its own memory M_i and also a re-ordering buffer that takes in incoming store instructions posted

[3] p is number of processors

by different processors including itself from their SB. Store instructions residing in this buffer eventually get flushed to memory. The combination of M_i and a re-ordering buffer simulates a processor seeing store instructions at different times and different relative order as that of another processor.

5.8 Specifying, Designing and Verifying an implementation

In this thesis we proposed a systematic process to first specify, then design and finally verify a shared memory multiprocessor so that it correctly implements the

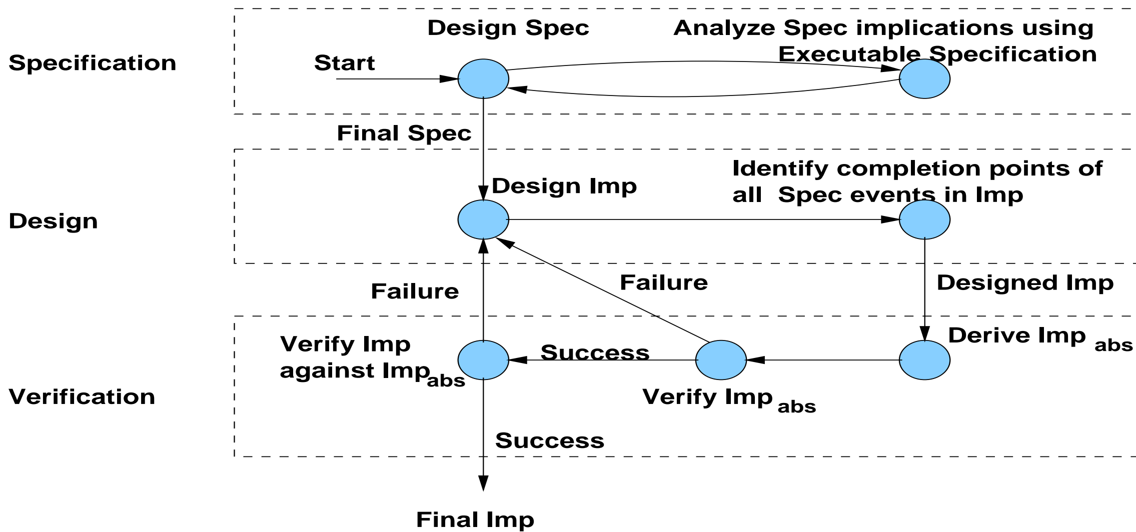

Figure 5.4. Specifying, Designing and Verifiying an Implementation

desired memory model. The entire process is depicted in Figure 7.1 and summarized as follows:

1. Specification: The memory model to be defined is first specified in our new definitional framework. The implications of ordering between various memory events are unambiguosly analyzed using the automatically generated executable specification and this cyclic process of specifying and analyzing the memory model is continued until the desired memory model specification $Spec$ is derived

2. Design: The implementation Imp is designed and completion points of all the memory events(e.g. $ld, fence, st_{local}, st_{global}$ for Alpha) as specified by $Spec$ are identified with respect to Imp (e.g. a st_{local} completes when buffered in SB for Alpha).

3. Verification: Imp_{abs} is derived from Imp where the internal partition is same as Imp but external partition is chosen based on which class the memory model belongs to. Then the two phase verification starts where

   (a) **Phase 1:** Imp is verified against Imp_{abs} through model-checking.

   (b) **Phase 2:** Imp_{abs} is verified against $Spec$ through theorem-proving.

   Phase 2 may be avoided if Imp_{abs} corresponds to a previously existing and verified intermmediate abstraction.

CHAPTER 6

EXPERIMENTS- SETUP, TEST CASES AND RESULTS

6.1 Experimental Setup

Number of procesors	16
Memory per processor	256 MB
Frequency	850 MHz
Algorithm	Breadth first search with symmetry
Message-Passing Library	MPI
Compiler Options (.m to .C)	mu -p
Compiler Options (.C to $< exec >$)	mpiCC -o $< exec >$ filename.C -lm
Runtime Options	mpirun -np 16 -machinefile nodenames.freebsd $< exec >$ -m256

Table 6.1. Experimental Setup

The experimental setup is shown in Table 6.1. We use the parallel Murϕ model-checker 3.0 Release Beta Version for our model checking effort. It was ported by us to run using the MPI library on our in-house Network Testbed. We ran our executables on 16 850MHz procesors, using 256MB memory per node. We use the default breadth first seacrh technique including symmetry. When we compile our protocol (.m file) into C++, the "-p" (parallel) option is used to generate code for parallel Murϕ instead of the default sequential Murϕ. Parallel Murϕ always does a breadth-first search of the state space. To get error traces, one has to start parallel Murϕ with the option "-d $< dir >$", where $< dir >$ denotes the directory, in which a file to store information for error traces will be written. Every node of the parallel system has to have this directory and it should be on a local harddisk. The C++

file is compiled using "mpiCC" that compiles and links MPI programs written in C or C++. Finally, one has to login to one of the 16 nodes in order to run the executable. Among the runtime options, one has to specify the number of nodes, machine names of all these nodes specified in a file say "nodenames.freebsd" and finally the amount of RAM to use per node.

6.2 Test Cases

We applied our method to an implementation of the Alpha processor [23] that was modeled after a multiprocessor using the Compaq (DEC) Alpha 21264 microprocessor. The cache coherence protocol is a Gigaplane-like split transaction bus [24] protocol. We also verify an Alpha implementation with an underlying cache coherence protocol using multiple interleaved buses, modeled after the Sun *Ultra*^{TM} *Enterprise*^{TM} 10000 [25]. Both these implementations were verified with and without Scheurich's optimization. We also applied our method to our own implementation of the Itanium memory model.

6.2.1 Internal Partition

The internal partitions chosen for any Alpha and Itanium implementation is summarized in Table 6.2.

	Alpha Internal Partition	Itanium Internal Partition
Load Buffer	Partial Re-Order	Fully Re-Order
Store Buffer	Fully Re-Order Coalescing	FIFO

Table 6.2. Internal Partition of Alpha and Itanium Implementation

6.2.2 External Partition

We will now explain the external partitions we used in details.

6.2.2.1 Split Transaction Bus Cache Invalidate Protocol

We will now discuss the protocol in details which is similar to several current bus protocols. A common feature of most complicated buses is that a transaction does not have to be completed before the next transaction can begin. In simple, circuit-switched bus protocols, coherence transactions are serialized (i.e., if processor P1 has requested a block in the EXCLUSIVE state and memory has not responded yet, then P1 will not release the bus until it receives the block from memory). Until P1's transaction has completed, a circuit-switched bus disallows other transactions from this processor as well as transactions from other processors. Split-transaction buses, on the other hand, allow systems to pipeline requests and responses. However, split-transaction buses are more difficult to prove correct, because transactions are not always atomic. Many split transaction buses, such as Sun Microsystem's Gigaplane(TM)[24], also permit data to be returned out of order with respect to the requests for it. In a split-transaction protocol, a transaction is composed of an action and zero or more reactions, where the action is a request and the reaction consists of the responses of all processors and memory modules to the action. For example, if a processor needs a block in the EXCLUSIVE state, it arbitrates for the bus, makes a request on the bus, and then it releases the bus. If memory or another processor has to respond, it will eventually send a reply on the bus, thus completing the transaction. Bus utilization is improved by allowing multiple transactions to proceed in parallel. Table 6.2 defines the actions and reactions for the protocol that we shall use in this section. All actions except PUTS use the bus.

The split-transaction nature of the protocol may require transfer of coherence permissions from processor to processor even before the data has arrived in response to an original coherence request. For example, suppose processor P1 makes a request for block B in the EXCLUSIVE state. Before memory has a chance to

Actions		Reactions	
Code	Description	Code	Description
GX	Get EXCLUSIVE	INV	Invalidate
GS	Get Shared	DWG	Downgrade (EXCLUSIVE to SHARED)
WB	Writeback	SEND	Send data
PUTS	Put SHARED		
UPG	Upgrade (SHARED to EXCLUSIVE)		

Table 6.3. Protocol Actions and Reactions

respond, processor P2 can make a request for the same block B in the EXCLUSIVE state. Now how do we handle this? One alternative would be to disallow it. This prevents P2 from making a request for B until memory has responded to P1's request. Another possible alternative would be to let memory (or possible a dedicated bus controller) keep track of outstanding transactions. This agent could then send blocks to requesting processors. A third alternative would be for ownership to transfer immediately upon requests. In this case, processor P1 becomes the owner of the block immediately after it makes its request. It then sees P2's request and records this fact so that it can send the block to P2 when it receives the block from memory. The protocol would need to impose certain constraints to avoid livelock and ensure forward progress. We pursue this last approach in our proposed protocol. Our protocol will transfer permissions immediately upon requests, even though the data may not get transferred for some amount of time after the transfer of ownership. This suggests the concept of maintaining two separate states for each block – one state (the address state) is maintained at the bus interface while the other state (the data state) is maintained at the processor's cache. The address state (A-state) changes immediately on coherence actions while the data state (D-state) may lag behind while waiting for the reaction(s). The

address tags at each processor maintain one of three states for each block: A_X (EXCLUSIVE), A_S (SHARED), or A_I (INVALID). In addition, the memory node has an A-state for each block. For the memory to be A_X means that all other nodes have the block A_I, and if the memory is A_I then one node has the block in A_X. Similarly, the data tags indicate D_X, D_S, or D_I. Furthermore, the data tags record information regarding pending transactions for a block (e.g., an invalidation may have been received for a block before the block has actually arrived at the cache from the bus). A queue is used to buffer messages from the bus (including a processor's own messages) before they are processed by the cache controller. Appendix C contains the entire Murϕ Code that models this protocol.

6.2.2.2 Multiple Interleaved Bus Cache Invalidate Protocol

System designers have implemented coherence protocols on systems with multiple buses. The Sun Ultra(TM) Enterprise(TM) 10000, for example, uses four buses. In a system with k buses, bus access is interleaved by address such that traffic involving address A uses the bus with number A modulo k. A multiple bus system could provide an increase in bus bandwidth. Appendix D contains the entire Murϕ code that models this protocol.

6.3 Summary of Results

As mentioned before,we applied our method to an implementation of the Alpha processor [23] that was modeled after a multiprocessor using the Compaq (DEC) Alpha 21264 microprocessor. The cache coherence protocol is a Gigaplane-like split transaction bus [24] protocol. We also verify an Alpha implementation with an underlying cache coherence protocol using multiple interleaved buses, modeled after the Sun $Ultra^{TM}$ $Enterprise^{TM}$ 10000 [25]. Both these implementations were verified with and without Scheurich's optimization. We also applied our method to

	Alpha Implementation			Itanium Implementation		
Cache Coherent Protocol	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)
Split Trans. Bus	64.16	470.52	0.95	111.59	985.43	1.75
Split Trans. Bus with Scheurich's Opt.	251.92	1794.96	3.42	325.66	2769.77	4.80
Multiple Interleaved Buses	255.93	1820.38	3.65	773.27	2686.89	10.97
Multiple Interleaved Buses with Scheurich's Opt.	278.02	1946.67	3.90	927.31	3402.41	12.07

Table 6.4. Experimental Results

our own implementation of the Itanium memory model. To our knowledge, nobody has verified such a list of protocols against different weak memory models using a uniform approach. These protocols finished in anywhere between 54 to 240 minutes on 16 processors, visiting upto 250 million states. The diameter of the reachability graph (indicative of the degree of parallelism in the problem) was sometimes in excess of 5,000. To better understand these numbers, we compare them with the highest numbers reported in [26], namely for the Stanford FLASH and the SCI protocols, where only 1 million states were explored, with the state graph diameter being close to 50.

CHAPTER 7

CONCLUSION

In this thesis, we presented a verification methodology for shared memory consistency protocols, and reported results on verifying four realistic protocols against the Alpha shared memory consistency model. We also experimented with four Intel Itanium protocols that we have developed. The results are summarized in Table 6.4. The fact that a variety of protocols gave rise to the same intermediate abstraction, that we could apply the technique on two different weak memory models, and the fact that we have encouraging results with respect to the use of parallel model-checking in this realm are the main contributions of this thesis.

Our approach fits today's design flow where aggressive, performance oriented protocols are first designed by expert designers, and handed over to verification engineers. The verification engineer, in turn, follows a systematic method for deriving an abstract reference model, and then uses a parallel model-checker to conduct verification. Effort-wise, our proposed method of obtaining the abstract model compares favorably with the effort of writing verification properties for a model checker. Our approach does not require special training to use, and can benefit from the use of multiple high-performance PCs to conduct parallel/distributed model checking, thereby covering large state spaces. We demonstrate promising initial results pertaining to the Alpha memory model on our parallel model-checking infrastructure.

In our ongoing work, we are verifying directory based implementations for the Alpha and Itanium memory model. We are also improving our parallel model-checker to scale up to very large state spaces.

Apart from being used in our verification process, the operational specification can be used to generate all possible outcomes of small assembly-language multiprocessor programs in a given memory model, which is very helpful for understanding the subtleties of the model. The executable specification can also check the correctness of assembly language programs including synchronization routines. [14] used this methodology for Ultra Sparc TSO memory model. However, we proposed an algorithm that can generate an operational model given any memory model formally specified in our own definitional framework.

To summarize our work we

1. provided a new **definitional framework** to specify a wide spectrum of memory consistency models that
   (a) aids in the verification process,
   (b) provides a common platform to compare memory models and
   (c) forms as an input to
2. enable automatic generation of their corresponding **executable specifications** and
3. developed an
   (a) **automatic**,
   (b) **scalable**,
   (c) **partially re-usable** and
   (d) **systematic two phase** approach to
   (e) formally verifying **complex** memory consistency models of shared memory multiprocessors.

REFERENCES

[1] http://www.cs.utah.edu/˜prosen/cav02.html.

[2] Sarita V. Adve and Kourosh Gharachorloo, "Shared memory consistency models: A tutorial", *Computer*, vol. 29, n. 12, pp. 66–76, December 1996.

[3] 2001, MPV: Workshop on Specification and Verification of Shared Memory Systems, Austin, Texas, October 31, 2001 (workshop held prior to FMCAD'2000).

[4] Leslie Lamport, "The Wildfire Challenge Problem", http://www.google.com - search for Wildfire Challenge.

[5] Thomas Henzinger, Shaz Qadeer and Sriram Rajamani, "Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems", in Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification99*, volume 1633 of *Lecture Notes in Computer Science*, pp. 301–315, Trento, Italy, July 1999, Springer-Verlag.

[6] Shaz Qadeer, "Verifying sequential consistency on shared-memory multiprocessors by model checking.", Technical report, SRC, December 2001, Research Report 176.

[7] Anne Condon and Alan J. Hu, "Automatable Verification of Sequential Consistency", in *Symposium on Parallel Algorithms and Architectures (SPAA)*, July 2001.

[8] Michael Merritt, "Guest Editorial: Special Issue on Shared Memory Systems", *Distributed Computing*, vol. 12, n. 12, pp. 55–56, 1999.

[9] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem and Ganesh Gopalakrishnan, "The 'Test Model-Checking' Approach to the Verification of Formal Memory Models of Multiprocessors", in Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pp. 464–476, Vancouver, BC, Canada, June 1998, Springer-Verlag.

[10] D. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. Martin and D. A. Wood, "Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol", Technical Report #1412, Computer Sciences Department, U. Wisconsin, Madison, March 2000.

[11] Gil Neiger, 2001, http://www.cs.utah.edu/mpv/papers/neiger/fmcad2001.pdf.

[12] Leslie Lamport, "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor", Technical report, Digital Equipment Corporation, Systems Research Center, February 1993.

[13] Prosenjit Chatterjee and Ganesh Gopalakrishnan, "Towards a Formal Model of Shared Memory Consistency for Intel Itanium", in *International Conference on Computer Aided Design*, page fill this, Austin, USA, 2001.

[14] David L. Dill, Seungjoon Park and Andreas Nowatzyk, "Formal Specification of Abstract Memory Models", in Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pp. 38–52. MIT Press, 1993.

[15] W. W. Collier, *Reasoning About Parallel Architectures*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[16] Kourosh Gharachorloo, *Memory Consistency Models for Shared Memory Multiprocessors*, PhD thesis, Stanford University, jun 1996, Department of Electrical Engineering.

[17] Pradeep S. Sindhu, Jean-Marc Frailong and Michel Cekleov, "Formal specifica tion of memory models", Technical report, Xerox Palo Alto Research Cent er, Palo Alto, California, December 1991.

[18] Phillip B. Gibbons and Ephraim Korach, "Testing Shared Memories", *SIAM Journal on Computing*, vol. 26, n. 4, pp. 1208–1244, August 1997.

[19] G.J. Holzmann, D. Peled and M. Yannakakis, "On Nested Depth First Search", in *The* SPIN *Verification System*, pp. 23–32. American Mathematical Society, 1996, Proc. of the Second SPIN Workshop.

[20] Prosenjit Chatterjee and Ganesh Gopalakrishnan, "Formally Specifying Memory Consistency Models and Automatically Generating Executable Specifications", Technical Report UUCS-398, University of Utah, Salt Lake City, UT, USA, October 2001.

[21] Seungjoon Park, *Computer Assisted Analysis of Multiprocessor Memory Systems*, PhD thesis, Stanford University, jun 1996, Department of Computer Science.

[22] David L. Weaver and Tom Germond, *The SPARC Architecture Manual – Version 9*, P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.

[23] Anne Condon, Mark Hill, Manoj Plakal and David Sorin, "Using Lamport Clocks to Reason About Relaxed Memory Models", in *Proceedings of the Fifth International Symposium On High Performance Computer Architecture (HPCA-5)*, January 1999.

[24] A.Singhal, D.Broniarczyk, F.Cerauskis, J.Price, L.Yuan, C.Cheng, D.Doblar, S.Fosth, N.Agarwal, K.Harvey, E.Hangersten and B.Liencres, "Gigaplane: A High Performance Bus for Large SMPs", in *Proc. of the 4th Annual Symposium on High Performance Interconnects at Stanford University*, pp. 41–52, 1996.

[25] http://www.sun.com/servers/datacenter/whitepapers/E10000.ps.

[26] Ulrich Stern and David Dill, "Parallelizing the Murϕ Verifier", *Formal Methods in System Design*, vol. 18, n. 2, pp. 117–129, 2001, (Journal version of their CAV 1997 paper).

[27] Christoph Scheurich, *Access Ordering and Coherence in Shared Memory Multiprocessors*, PhD thesis, University of Southern California, May 1989.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennes sy, "Memory consistency and event ordering in scalable shared-memory multiprocessors", in *17th Annual International Symposium on Computer Architec ture*, pp. 15–26. ACM Press, May 1990.

[29] L.Higham, J.Kawash and Nathaly Verwaal, "Defining and Comparing Memory Consistency Models", in *10th International Conference on Parallel and Distributed Computing Systems*, New Orleans, October 1997.

[30] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli and Phillip W. Hutto, "Causal Memory: Definitions, Implementation and Programming", *Distributed Computing*, vol. 9, n. 1, pp. 37–49, 1995.

[31] Seungjoon Park and David Dill, "An executable specification, analyzer and ve rifier for RMO (Relaxed Memory Order)", in *Symposium on Parallel Algorithms and Architectures (SPAA*, pp. 34–41. ACM Press, July 1995.

[32] David Dill, "The Stanford Murphi Verifier", in Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pp. 390–393, New Brunswick, New Jersey, July 1996, Springer-Verlag, Tool demo.

[33] Gerard Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[34] J. Archibald and J. Baer, "An economic solution to the cache coherence proble m", in *11th Annual International Symposium on Computer Architec ture*, pp. 355–362. ACM Press, June 1985.

[35] L. Censier and P. Feautrier, "A new solution to coherence problems in multica che systems", *IEEE Transactions Computers*, December 1978.

[36] D. Lenosky, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor", in *17th Annual International Symposium on Computer Architec ture*, pp. 148–159. ACM Press, May 1990.

[37] L. Censier and P. Feautrier, "Data coherence problem in a multicache system", *IEEE Transactions Computers*, January 1985.

[38] Giorgio Delzanno, "Automatic Verification of Parameterized Cache Coherence Protocols", jul 2000.

[39] Rob Gerth, "Sequential Consistency and the Lazy Caching Algorithm", *Distributed Computing*, vol. 12, n. 12, pp. 57–59, 1999.

[40] Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu and Ganesh Gopalakrishnan, "Using "test model-checking" to verify the Runway-PA8000 memory model", in *Tenth Annual ACM Symposium On Parallel Algorithms and Architectures*, pp. 231–239, Puerto Vallarta, Mexico, June 1998, ACM Press, Program Chair: Phillip B. Gibbons.

[41] Kenneth L. McMillan, *Symbolic Model Checking*, Kluwer Academic Press, 1993.

[42] Z. Har'El and R.P. Kurshan, "Software for Analysis of Coordination", in *Proc. Int'l Conference on System Science*, 1988.

[43] Edmund Clarke, Orna Grumberg and Doron Peled, *Model Checking*, MIT Press, 2000.

[44] Susanne Graf, "Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction", *Distributed Computing*, vol. 12, n. 12, pp. 75–90, 1999, Special Issue on Shared Memory Systems, Michael Merritt (ed.).

[45] Richard L. Sites, *Alpha Architecture Reference Manual*, Digital Press, 1992.

[46] Leslie Lamport, "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems*, vol. 16, n. 3, pp. 872–923, May 1994, Also appeared as SRC Research Report 79.

[47] Jason F. Cantin, Mikko H. Lipasti and James E. Smith, "Dynamic Verification of Cache Coherence Protocol", in *?*, June 2001, Workshop on Memory Performance Issues, in conjunction with ISCA.

[48] Todd M. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification", *Journal of Instruction-Level Parallelism*, vol. 2, n. 5, pp. 1–6, May 2000.

[49] R. P. Case and A. Padegs, "Architecture of the IBM System 370", *Communications of the ACM*, vol. 21, n. 12, pp. 73–96, January 1978.

[50] D. L. Weaver and T. Germond, *The Sparc Architcture Manual- Version 9*, NJ 07632, September 1994.

[51] R. L. Sites, *Alpha Architcture Reference Manual- Version 9*, September 1992.

[52] R.J. Lipton and J.S. Sandberg, "PRAM: A scalable shared memory", Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.

[53] P. W. Hutto and M. Ahamad, "Slow Memory: Weakening Consistency to enhance to enhance concurrency in distributed shared memories", in *10th International Conference on Distributed Computing Systems*, May 1990.

[54] Intel, *The IA-64 Architecture Software Developer's Manual Vol. 2 rev. 1.1: Itanium (TM); System Architecture*, Intel, 2000.

APPENDIX A

Details of function Allowed of Itanium Operational Model

Why $l(t^{'}) = v(t)[p(t^{'})]$ and not $l(t^{'}) \leq v(t)[p(t^{'})]$ is used in function Allowed: Suppose $l(t^{'}) < v(t)[p(t^{'})]$. Then the value L $= v(t)[p(t^{'})]$ corresponds to some instruction, say $t^{''}$. There are two cases: (i) $t^{''}$ is in $WIB_p(t)$. In this case, function Allowed ensures that $t^{'}$ gets posted into $M_{p(t)}$ before $t^{''}$. Then, we will be back to the = test. (ii) $t^{''}$ has already posted into M, in which case it isn't in $WIB_{p(t)}$. This is a contradiction because $t^{'}$ is still in WIB, violating Allowed.

APPENDIX B

Proof of the abstract model satisfying the logical specification

We show that any execution of the abstract model satisfies the logical specification. Since the temporal order and the logical order of completion of events do not differ in the abstract model, we just need to prove that the temporal order of completion of events is consistent with the *Read Value* and *Per Processor Order*. That the *Read Value* is satisfied is proved as follows. There are two possible situations as given in the *Read Value* definition:

1. In the first case let t_2^{local} be the most recent *st* s.t $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ as defined. Since instructions are issued in program order and issue intervals are non-overlapping, t_2^{local} is in $p(t_1)$'s WB before the processor checks for the youngest matching entry to address $a(t_1)$ in WB . t_2^{local} is the latest such local *st* entry as SB is a FIFO buffer and if two stores write to the same word, the youngest entry will hold the value written by the store that was issued later. t_2^{local} is still present in WB when this check for the matching entry is

done. Otherwise, t_2^{global} must have completed, making t_1 complete after it. Consequently, $t_2^{global} \rightarrow t_1$ which is a contradiction. Hence $d(t_1) = d(t_2^{local})$

2. In the second case let t_2^{global} be the most recent st s.t $t_2^{global} \rightarrow t_1$ as defined. In this case, $M[a(t_1)] = d(t_2^{global})$ and as access to M is an atomic step, no new t_3^{global} can overwrite $M[a(t_1)]$ while $p(t_1)$ reads from M. Hence $d(t_1) = d(t_2^{global})$

Per Processor Order is satisfied as follows. We analyse some sub-cases as for others, similar reasoning will suffice. Let t_1 and t_2 be two events s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$, $a(t_1) = a(t_2)$ and

1. $o(t_1) = o(t_2) = st_{local}$: Since SB is FIFO buffer and t_2 is issued after t_1, t_2 will flushed after t_1 and hence t_2 completes after t_1.

2. $o(t_1) = o(t_2) = st_{global}$: There are two possibilities. They are when

   (a) prior to being written to M, both their corresponding local events say $t_1^{'}$ and $t_2^{'}$ are in SB. In that case $t_2^{'}$ will be flushed after $t_1^{'}$. Hence, t_1 completes before t_2.

   (b) $t_2^{'}$ in SB and t_1 has written to M. In this case, eventually $t_2^{'}$'s data overwrites current value of M. Hence t_1 completes before t_2.

Since the st_{global} events complete by writing to single port memory, each in one single step via the multiplexor switch, the events naturally form a total order.

The abstract model that accounts for Scheurich's optimization can also be proved similarly with the additonal steps to account for the temporal ordering being different from logical ordering. The modification of defining the completion point of st_{local} and st_{global} bridges this difference. [1] can be referred for the complete proof.

APPENDIX C

Murϕ Code of Itanium Split Transaction Bus Protocol

```
Const
  PROCESSORS:               2;
  ADDRESSES:                2;
  DATA_VALUES:              2;
      P:                        0;
      Q:                        1;
      A:                        0;
      B:                        1;
      WRITE_BUFFER_CAPACITY:    2;
      INVALID:                  3;
  SHARED:                   4;
  EXCLUSIVE:                5;
  EX_GRANTED:               6;
      EMPTY:                    7;

Type
       PROC: 0..PROCESSORS-1;
       DATA: 0..DATA_VALUES;
       ADDR: 0..1;
       STATE:3..6;
       TRI_LOGIC:0..2;
       RQ_RS:0..7;
       BUFFER_SIZE:0..WRITE_BUFFER_CAPACITY;

tuple1:
```

```
    Record
        addr:ADDR;
                data:DATA;
    End;
tuple2:
    Record
        data:DATA;
        state:STATE;
    End;

buffer:
    Record
        writeBuffer: Array[0..1] of tuple1;
                writeBufferSize: BUFFER_SIZE;
    End;


cache:
    Record
        addr:Array[0..1] of tuple2;
    End;
chan:
    Record
        addr:ADDR;
        rq_rs:RQ_RS;
    End;
```

```
Var
    M: Array[0..1] of DATA;
    C: Array[0..1] of cache;
       SB: Array[0..1] of buffer;
       RB: Array[0..1] of buffer;
      bus: Array[0..1] of chan;
      pending:Array[0..1] of TRI_LOGIC;
procedure st_local(p:PROC;a:ADDR;d:DATA);
begin
SB[p].writeBuffer[SB[p].writeBufferSize].addr:=a;
SB[p].writeBuffer[SB[p].writeBufferSize].data:=d;
SB[p].writeBufferSize:=SB[p].writeBufferSize+1;
End;

procedure ld_bufferize(p:PROC;a:ADDR);
begin
RB[p].writeBuffer[RB[p].writeBufferSize].addr:=a;
RB[p].writeBufferSize:=RB[p].writeBufferSize+1;
End;



procedure st_global(p:PROC;a:ADDR;d:DATA);
begin
SB[p].writeBufferSize:=SB[p].writeBufferSize-1;
SB[p].writeBuffer[0].addr:=SB[p].writeBuffer[1].addr;
SB[p].writeBuffer[0].data:=SB[p].writeBuffer[1].data;
```

```
C[p].addr[a].state:=EXCLUSIVE;
C[p].addr[a].data:=d;
M[a]:=d;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Ruleset k: 1..2 Do
Rule   "st_local"
    (SB[i].writeBufferSize < 2 )
    ==>
    begin
    st_local(i,j,k);
    End;
End;
End;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule   "ld bufferize"
    (RB[i].writeBufferSize < 1 )
    ==>
    begin
    ld_bufferize(i,j);
    End;
End;
```

```
End;



Rule  " st_global P1"
(C[P].addr[SB[P].writeBuffer[0].addr].state =  EXCLUSIVE) &
(SB[P].writeBufferSize > 0)
    ==>
    begin
    switch SB[P].writeBuffer[0].addr
        case 0:
      st_global(P,0,SB[P].writeBuffer[0].data);
        case 1:
      st_global(P,1,SB[P].writeBuffer[0].data);
        end;
    End;
Rule  "st_global P2"
(C[P].addr[SB[P].writeBuffer[0].addr].state != EXCLUSIVE) &
(C[Q].addr[SB[P].writeBuffer[0].addr].state  > 3) &
(SB[P].writeBufferSize > 0) &
bus[P].rq_rs=EMPTY &
(pending[Q]!=SB[P].writeBuffer[0].addr)
    ==>
        begin
    pending[P]:=SB[P].writeBuffer[0].addr;
        bus[P].addr:=SB[P].writeBuffer[0].addr;
```

```
        bus[P].rq_rs:=EXCLUSIVE;
        End;
Rule  "st_global P3"
  (bus[P].addr=SB[P].writeBuffer[0].addr) &
(bus[P].rq_rs=EX_GRANTED)
    ==>
    begin
      bus[P].rq_rs:=EMPTY;
      pending[P]:=2;
    switch SB[P].writeBuffer[0].addr
        case 0:
      st_global(P,0,SB[P].writeBuffer[0].data);
        case 1:
      st_global(P,1,SB[P].writeBuffer[0].data);
        end;
    End;

Rule  " st_global Q1"
(C[Q].addr[SB[Q].writeBuffer[0].addr].state =  EXCLUSIVE) &
(SB[Q].writeBufferSize > 0)
    ==>
    begin
    switch SB[Q].writeBuffer[0].addr
        case 0:
      st_global(Q,0,SB[Q].writeBuffer[0].data);
        case 1:
      st_global(Q,1,SB[Q].writeBuffer[0].data);
```

```
        end;
    End;
Rule  "st_global Q2"
(C[Q].addr[SB[Q].writeBuffer[0].addr].state != EXCLUSIVE) &
(C[P].addr[SB[Q].writeBuffer[0].addr].state  > 3) &
(SB[Q].writeBufferSize > 0) &
bus[Q].rq_rs=EMPTY  &
(pending[P]!=SB[Q].writeBuffer[0].addr)
    ==>
        begin
    pending[Q]:=SB[Q].writeBuffer[0].addr;
        bus[Q].addr:=SB[Q].writeBuffer[0].addr;
        bus[Q].rq_rs:=EXCLUSIVE;
        End;
Rule  "st_global Q3"
  (bus[Q].addr=SB[Q].writeBuffer[0].addr) &
(bus[Q].rq_rs=EX_GRANTED)
    ==>
    begin
      bus[Q].rq_rs:=EMPTY;
      pending[Q]:=2;
    switch SB[Q].writeBuffer[0].addr
        case 0:
      st_global(Q,0,SB[Q].writeBuffer[0].data);
        case 1:
      st_global(Q,1,SB[Q].writeBuffer[0].data);
        end;
```

```
    End;


Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule  " ld 1"
(C[i].addr[j].state > 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 )
    ==>
    begin
        assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
Rule  "ld 2"
(C[i].addr[j].state = 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 ) &
bus[i].rq_rs=EMPTY  &
((i=P & pending[Q]!=j) | (i=Q & pending[P]!=j))
    ==>
    begin
    pending[i]:=j;
        bus[i].addr:=j;
```

```
        bus[i].rq_rs:=SHARED;
    End;
Rule "ld 3"
  (bus[i].addr=j) &
  ((bus[i].rq_rs =0 ) |
   (bus[i].rq_rs =1 ) |
   (bus[i].rq_rs =2 ))
    ==>
    begin
        C[i].addr[j].state:=SHARED;
        C[i].addr[j].data:=bus[i].rq_rs;
        bus[i].rq_rs:=EMPTY;
    pending[i]:=2;
        assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
End;
End;


Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule  " ld_buffer 1"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
(C[i].addr[j].state > 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 )
```

```
    ==>
    begin
        RB[i].writeBufferSize:=0;
        assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
Rule  "ld_buffer 2"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
(C[i].addr[j].state = 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 ) &
bus[i].rq_rs=EMPTY  &
((i=P & pending[Q]!=j) | (i=Q & pending[P]!=j))
    ==>
    begin
    pending[i]:=j;
        bus[i].addr:=j;
        bus[i].rq_rs:=SHARED;
    End;
Rule "ld 3"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
  (bus[i].addr=j) &
  ((bus[i].rq_rs =0 ) | (bus[i].rq_rs =1 ) |(bus[i].rq_rs =2 ))
    ==>
    begin
        C[i].addr[j].state:=SHARED;
```

```
        C[i].addr[j].data:=bus[i].rq_rs;
        bus[i].rq_rs:=EMPTY;
    pending[i]:=2;
    RB[i].writeBufferSize:=0;
        assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
End;
End;
```

```
Ruleset i: P..Q Do
 Rule "protocol response 1 "
 bus[i].rq_rs=SHARED &
 ((i=P & pending[Q]!=bus[i].addr) |
  (i=Q & pending[P]!=bus[i].addr) )
    ==>
 begin
 switch  i
 case P:
 C[Q].addr[bus[i].addr].state:=SHARED;
 bus[i].rq_rs:=C[Q].addr[bus[i].addr].data;
```

```
case Q:
 C[P].addr[bus[i].addr].state:=SHARED;
 bus[i].rq_rs:=C[P].addr[bus[i].addr].data;
 end;
 End;

 Rule "protocol response 2"
bus[i].rq_rs=EXCLUSIVE &
((i=P & pending[Q]!=bus[i].addr) |
 (i=Q & pending[P]!=bus[i].addr) )
    ==>
 begin
 switch  i
 case P:
 C[Q].addr[bus[i].addr].state:=INVALID;
 bus[i].rq_rs:=EX_GRANTED;
case Q:
 C[P].addr[bus[i].addr].state:=INVALID;
 bus[i].rq_rs:=EX_GRANTED;
 end;
 End;



End;

Startstate
```

```
begin
    For j:A..B Do
    C[P].addr[j].state:=EXCLUSIVE;
    C[Q].addr[j].state:=INVALID;
    C[P].addr[j].data:=0;
    C[Q].addr[j].data:=0;
    M[j]:=0;
    End;
For i:P..Q Do
pending[i]:=2;
    SB[i].writeBufferSize:=0;
    RB[i].writeBufferSize:=0;
bus[i].addr:=0;
    bus[i].rq_rs:=EMPTY;
    SB[i].writeBuffer[0].addr:=0;
    SB[i].writeBuffer[0].data:=0;
    SB[i].writeBuffer[1].addr:=0;
    SB[i].writeBuffer[1].data:=0;
    RB[i].writeBuffer[0].addr:=0;
    RB[i].writeBuffer[0].data:=0;
    RB[i].writeBuffer[1].addr:=0;
    RB[i].writeBuffer[1].data:=0;
    End;


End;
```

Murϕ Code of Itanium Split Transaction Bus Protocol with Scheurich's Optimization

The unoptimized and the optimized Murϕ code varies only in the way the protocol responses to coherence requests. The code added to the unoptimized protocol is as follows.

```
  Rule "protocol response 3_Scheurich"
  bus[i].rq_rs=EXCLUSIVE &
  ((i=P & pending[Q]!=bus[i].addr) |
   (i=Q & pending[P]!=bus[i].addr) )
  & ((i=P & (C[Q].addr[bus[i].addr].state=SHARED))|
     (i=Q & (C[P].addr[bus[i].addr].state=EXCLUSIVE |
      C[P].addr[bus[i].addr].state=SHARED)))
    ==>
 begin
 switch  i
 case P:
 bus[i].rq_rs:=EX_GRANTED;
case Q:
 bus[i].rq_rs:=EX_GRANTED;
 end;
 End;
```

APPENDIX D

Murϕ Code of Itanium Multiple Interleaved Bus Protocol

```
  Const
    PROCESSORS:                   2;
    ADDRESSES:                    2;
    DATA_VALUES:                  2;
        P:                            0;
        Q:                            1;
        A:                            0;
        B:                            1;
        WRITE_BUFFER_CAPACITY:        2;
        INVALID:                      3;
    SHARED:                       4;
    EXCLUSIVE:                    5;
    EX_GRANTED:                   6;
        EMPTY:                        7;

Type
       PROC: 0..PROCESSORS-1;
       DATA: 0..DATA_VALUES;
       ADDR: 0..1;
       STATE:3..6;
       TRI_LOGIC:0..2;
       RQ_RS:0..7;
       BUFFER_SIZE:0..WRITE_BUFFER_CAPACITY;

tuple1:
    Record
        addr:ADDR;
```

```
                data:DATA;
    End;
tuple2:
    Record
        data:DATA;
        state:STATE;
    End;

buffer:
    Record
        writeBuffer: Array[0..1] of tuple1;
                writeBufferSize: BUFFER_SIZE;
    End;


cache:
    Record
        addr:Array[0..1] of tuple2;
    End;
chan:
    Record
        addr:ADDR;
        rq_rs:RQ_RS;
    End;
bustype:
    Record
      multi:Array[0..1] of chan;
```

```
    End;

Var
    M: Array[0..1] of DATA;
    C: Array[0..1] of cache;
       SB: Array[0..1] of buffer;
       RB: Array[0..1] of buffer;
      bus: Array[0..1] of bustype;
      pending:Array[0..1] of TRI_LOGIC;
procedure st_local(p:PROC;a:ADDR;d:DATA);
begin
SB[p].writeBuffer[SB[p].writeBufferSize].addr:=a;
SB[p].writeBuffer[SB[p].writeBufferSize].data:=d;
SB[p].writeBufferSize:=SB[p].writeBufferSize+1;
End;

procedure ld_bufferize(p:PROC;a:ADDR);
begin
RB[p].writeBuffer[RB[p].writeBufferSize].addr:=a;
RB[p].writeBufferSize:=RB[p].writeBufferSize+1;
End;



procedure st_global(p:PROC;a:ADDR;d:DATA);
begin
SB[p].writeBufferSize:=SB[p].writeBufferSize-1;
```

```
SB[p].writeBuffer[0].addr:=SB[p].writeBuffer[1].addr;
SB[p].writeBuffer[0].data:=SB[p].writeBuffer[1].data;
C[p].addr[a].state:=EXCLUSIVE;
C[p].addr[a].data:=d;
M[a]:=d;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Ruleset k: 1..2 Do
Rule   "st_local"
    (SB[i].writeBufferSize < 2 )
    ==>
    begin
    st_local(i,j,k);
    End;
End;
End;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule   "ld bufferize"
    (RB[i].writeBufferSize < 1 )
    ==>
    begin
    ld_bufferize(i,j);
```

```
    End;
End;
End;


Ruleset i: P..Q Do
Rule  " st_global_no_coherent_transaction_required"
(C[i].addr[SB[i].writeBuffer[0].addr].state =  EXCLUSIVE) &
(SB[i].writeBufferSize > 0)
    ==>
    begin
    switch SB[i].writeBuffer[0].addr
        case 0:
      st_global(i,0,SB[i].writeBuffer[0].data);
        case 1:
      st_global(i,1,SB[i].writeBuffer[0].data);
        end;
    End;
    End;

Ruleset i: A..B Do
Rule  "st_global P2"
(C[P].addr[SB[P].writeBuffer[0].addr].state != EXCLUSIVE) &
(C[Q].addr[SB[P].writeBuffer[0].addr].state  > 3) &
(SB[P].writeBufferSize > 0) &
(SB[P].writeBuffer[0].addr=i) &
bus[i].multi[P].rq_rs=EMPTY &
```

```
(pending[Q]!=SB[P].writeBuffer[0].addr)
    ==>
        begin
    pending[P]:=SB[P].writeBuffer[0].addr;
        bus[i].multi[P].addr:=SB[P].writeBuffer[0].addr;
        bus[i].multi[P].rq_rs:=EXCLUSIVE;
        End;
Rule  "st_global P3"
  (bus[i].multi[P].addr=SB[P].writeBuffer[0].addr) &
(bus[i].multi[P].rq_rs=EX_GRANTED)
    ==>
    begin
      bus[i].multi[P].rq_rs:=EMPTY;
      pending[P]:=2;
    switch SB[P].writeBuffer[0].addr
        case 0:
      st_global(P,0,SB[P].writeBuffer[0].data);
        case 1:
      st_global(P,1,SB[P].writeBuffer[0].data);
        end;
    End;

Rule  "st_global Q2"
(C[Q].addr[SB[Q].writeBuffer[0].addr].state != EXCLUSIVE) &
(C[P].addr[SB[Q].writeBuffer[0].addr].state  > 3) &
(SB[Q].writeBufferSize > 0) &
(SB[Q].writeBuffer[0].addr=i) &
```

```
bus[i].multi[Q].rq_rs=EMPTY  &
(pending[P]!=SB[Q].writeBuffer[0].addr)
    ==>
        begin
    pending[Q]:=SB[Q].writeBuffer[0].addr;
        bus[i].multi[Q].addr:=SB[Q].writeBuffer[0].addr;
        bus[i].multi[Q].rq_rs:=EXCLUSIVE;
        End;
Rule  "st_global Q3"
  (bus[i].multi[Q].addr=SB[Q].writeBuffer[0].addr) &
(bus[i].multi[Q].rq_rs=EX_GRANTED)
    ==>
    begin
      bus[i].multi[Q].rq_rs:=EMPTY;
      pending[Q]:=2;
    switch SB[Q].writeBuffer[0].addr
        case 0:
      st_global(Q,0,SB[Q].writeBuffer[0].data);
        case 1:
      st_global(Q,1,SB[Q].writeBuffer[0].data);
        end;
    End;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule  " ld 1"
```

```
(C[i].addr[j].state > 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 )
    ==>
    begin
        --assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
Rule  "ld 2"
(C[i].addr[j].state = 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 ) &
bus[j].multi[i].rq_rs=EMPTY  &
((i=P & pending[Q]!=j) | (i=Q & pending[P]!=j))
    ==>
    begin
    pending[i]:=j;
        bus[j].multi[i].addr:=j;
        bus[j].multi[i].rq_rs:=SHARED;
    End;
Rule "ld 3"
   ((bus[j].multi[i].rq_rs =0 ) |
    (bus[j].multi[i].rq_rs =1 ) |
    (bus[j].multi[i].rq_rs =2 ))
```

```
    ==>
    begin
        C[i].addr[j].state:=SHARED;
        C[i].addr[j].data:=bus[j].multi[i].rq_rs;
        bus[j].multi[i].rq_rs:=EMPTY;
    pending[i]:=2;
        --assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
End;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
Rule  " ld_buffer 1"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
(C[i].addr[j].state > 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 )
    ==>
    begin
        RB[i].writeBufferSize:=0;
    --    assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
Rule  "ld_buffer 2"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
```

```
(C[i].addr[j].state = 3) &
!(
  (SB[i].writeBufferSize>0 & SB[i].writeBuffer[0].addr=j) |
  (SB[i].writeBufferSize=2 & SB[i].writeBuffer[1].addr=j)
 ) &
bus[j].multi[i].rq_rs=EMPTY  &
((i=P & pending[Q]!=j) | (i=Q & pending[P]!=j))
    ==>
    begin
    pending[i]:=j;
        bus[j].multi[i].addr:=j;
        bus[j].multi[i].rq_rs:=SHARED;
    End;
Rule "ld 3"
(RB[i].writeBufferSize > 0) & (RB[i].writeBuffer[0].addr=j) &
   ((bus[j].multi[i].rq_rs =0 ) |
    (bus[j].multi[i].rq_rs =1 ) |
    (bus[j].multi[i].rq_rs =2 ))
    ==>
    begin
        C[i].addr[j].state:=SHARED;
        C[i].addr[j].data:=bus[j].multi[i].rq_rs;
        bus[j].multi[i].rq_rs:=EMPTY;
    pending[i]:=2;
    RB[i].writeBufferSize:=0;
        --assert (C[i].addr[j].data=M[j]) "read-data mismatch\n";
    End;
```

```
End;
End;

Ruleset i: P..Q Do
Ruleset j: A..B Do
 Rule "protocol response 1 "
 bus[j].multi[i].rq_rs=SHARED &
 ((i=P & pending[Q]!=bus[j].multi[i].addr) |
  (i=Q & pending[P]!=bus[j].multi[i].addr) )
    ==>
 begin
 switch  i
 case P:
 C[Q].addr[bus[j].multi[i].addr].state:=SHARED;
 bus[j].multi[i].rq_rs:=C[Q].addr[bus[j].multi[i].addr].data;
case Q:
 C[P].addr[bus[j].multi[i].addr].state:=SHARED;
 bus[j].multi[i].rq_rs:=C[P].addr[bus[j].multi[i].addr].data;
 end;
 End;
```

```
 Rule "protocol response 2"
 bus[j].multi[i].rq_rs=EXCLUSIVE &
 ((i=P & pending[Q]!=bus[j].multi[i].addr) |
  (i=Q & pending[P]!=bus[j].multi[i].addr) )
    ==>
 begin
 switch  i
 case P:
 C[Q].addr[bus[j].multi[i].addr].state:=INVALID;
 bus[j].multi[i].rq_rs:=EX_GRANTED;
case Q:
 C[P].addr[bus[j].multi[i].addr].state:=INVALID;
 bus[j].multi[i].rq_rs:=EX_GRANTED;
 end;
 End;



End;
End;


Startstate
    begin
        For j:A..B Do
        C[P].addr[j].state:=EXCLUSIVE;
        C[Q].addr[j].state:=INVALID;
        C[P].addr[j].data:=0;
```

```
    C[Q].addr[j].data:=0;
    M[j]:=0;
    End;
    For i:P..Q Do
    pending[i]:=2;
    SB[i].writeBufferSize:=0;
    RB[i].writeBufferSize:=0;
    SB[i].writeBuffer[0].addr:=0;
    SB[i].writeBuffer[0].data:=0;
    SB[i].writeBuffer[1].addr:=0;
    SB[i].writeBuffer[1].data:=0;
    RB[i].writeBuffer[0].addr:=0;
    RB[i].writeBuffer[0].data:=0;
    RB[i].writeBuffer[1].addr:=0;
    RB[i].writeBuffer[1].data:=0;
    End;

    For i:P..Q Do
    For j:A..B Do
    bus[j].multi[i].addr:=0;
    bus[j].multi[i].rq_rs:=EMPTY;
    End;
    End;
End;
```