Formal Analysis of Distributed Shared Memory Algorithms

Muhammad Atif[1], Muhammad Adnan Hashmi[2], Mudassar Naseer[1], and Ahmad Salman Khan[3]
muhammad.atif@cs.uol.edu.pk, ahashmi@hct.ac.ae, mudassar.naseer@cs.uol.edu.pk, ahmad.salman@se.uol.edu.pk
[1] Department of Computer Science, The University of Lahore, Pakistan.
[2] Department of Computer Information Science, Higher Colleges of Technology, UAE.
[3] Department of Software Engineering, The University of Lahore, Pakistan.

Summary
The memory coherence problem occurs while mapping shared virtual memory in a loosely coupled multiprocessors setup. Memory is considered coherent if a read operation provides same data written in the last write operation. The problem is addressed in the literature using different algorithms. The big question is on the correctness of such a distributed algorithm. Formal verification is the principal term for a group of techniques that routinely use an analysis that is established on mathematical transformations to conclude the rightness of hardware or software behavior in divergence to dynamic verification techniques. This paper uses UPPAAL model checker to model the dynamic distributed algorithm for shared virtual memory given by K.Li and P.Hudak. We analyse the mechanism to keep the coherence of memory in every read and write operation by using a dynamic distributed algorithm. Our results show that the dynamic distributed algorithm for shared virtual memory partially fulfils its functional requirements.

Keywords:
Virtual memory, Distributed Algorithm, Formal Specification, Verification.

1. Introduction

The idea of virtual memory becomes inevitable when a system requires more memory than installed.Virtual memory is known as usage of other than main memory as a main memory. In the shared virtual memory, physically separated memories (on the network) are shared among the processors connected through a loosely coupled fashion. Processes while executing in different processors may use shared virtual memory like traditional virtual memory as shown in Figure 1. Formal methods offer a large potential to provide correctness measuring techniques [13][7][8][9]. This set of techniques helps us to avoid overlooking critical issues. Formal methods provide different techniques to model and check the complex systems as mathematical entities. These models make it possible to verify a system's specifications better than empirical testing [2][3]. We apply model-checking techniques to verify the memory coherence problem where the shared virtual memory is managed through distributed manager algorithms.

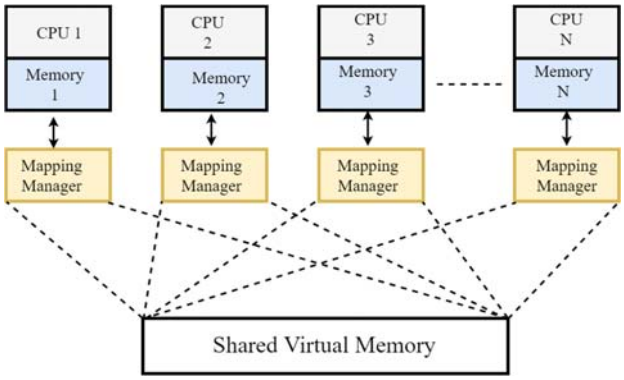

Fig. 1 Shared virtual memory [14].

In this paper, we study the dynamic distributed memory management algorithm given in [14] where other approaches, like centralized manager, fixed and broadcast are also given. Implementing the centralization algorithm becomes challenging when all of the traffic pases through a central manager for each type of page fault. An algorithm appears to have the best required results and general features named as dynamic distributed manager algorithm.

The dynamic distributed manager algorithm is comparatively better than other algorithms when there are a lot of page faults and we need to manage the network traffic in an efficient manner. The performance of this algorithm shows that it is probable to implement it on a huge scale multiprocessor. However, we verify the functional requirement of this algorithm by using formal methods. Formal Method is a standard word for system scheme, investigation, and application methods that are designated and used with scientific precision [4][5].

Manuscript received April 5, 2024
Manuscript revised April 20, 2024
https://doi.org/10.22937/IJCSNS.2024.24.4.22

2. Literature Review

In [1] Venkateswarlu Chennareddy et al. verified weak consistency model of distributed shared memory. CADP (Construction and Analysis of Distributed Processes) toolbox is used for design and implementation of model. In [4] Johan.B et al. modeled memory management system of virtual memory with MSVL tool. Memory Management System is formalized by using MSVL (Modeling Simulation and Verification Language) using Model Checking Approach. This approach is applied to verify the perfection, delay linked properties and regular repeated properties. Munez et al. present the formal verification of a sequentially consistent memory model where low level functions were considered as sequential [4]. In [4], Kim G. Larsen et al. perform model checking using UPPAAL and verify the audio protocol. Researchers describe the importance of structures in UPPAAL used for model checking. Another integration of formal verification through Cyber-Physical Systems (CPS) design process is presented in the literature which is consisted of executing transformation of AADL (Architecture Analysis and Design Language) models and represented them in timed automata. This was analyzed through model checking (MC)[11].

In [12], authors gift a roaring placing on DSMC (Deep Statistical Model Checking) to MECHATRONICUML, some sort of DSML (Domain Specific Model Checking) for the laptop software device kind of cycles/2d, victimization the version checker UPPAAL.

3. Memory Coherence Problem

A single address space is shared by several processors in shared virtual memory on the network as shown in Figure 1. It is allowed to all processors to directly access any memory address in address space. Memory mapping manager controls the implementation of mapping between shared virtual memory address space and local memories. Major responsibilities of a manager include to protect the system from memory coherent problem that means a read operation value on all processor must be same to the most recent write operation.

Address spaces of shared virtual memory are divided into pages. Pages are a point to a memory block. A different copy of pages with read-only operation take place in different processors physical memory at the same time, but the page with write operation just locate in one processor's physical memory. Memory mapping manager scans its local memory as well as address spaces of attached processors from the shared virtual memory cache. A page fault occurs due to memory reference where the page memory location is not in the current physical memory of the processor. So in the case of a page fault, manager rescue the page, get a page from disk or any other processor. If another processor has copies of the faulting memory page reference, then manager need to do some effort for maintaining the memory coherent. The memory coherent problem might be encountered as these algorithms are maintaining the memory. A shared virtual memory on loosely coupled systems has no physical shared memory, and the communication budget between processors is non-trivial.

3.1 Dynamic Distributed Memory Management

Dynamic distributed algorithm is a type of Distributed Manager Algorithms where tasks are divided among individual processors. In this algorithm, every processor has its local table for maintaining the ownership of all pages, which is known as PTable. This PTable has five columns naming page id, access field, copy-set, probowner and lock field [14].

i. Page id is the unique id of page.

ii. The access field shows the page accessibility roles i.e. either read or write.

iii. The copy-set contains IDs of the processors having copies of the page.

iv. Probowener mean a possible owner of a page

v. The lock field is used to avoid the race condition between/among processors demanding the same page.

In this algorithm probowner is set in a way that there is no loop for pointing out prbowner. For example, it is not possible that a node A says probowner is the node B and the node B says that the probowner is the node A.

In this protocol, every node sends requests to it's probowner and if that is actual owner it replies back, otherwise forwards the request to probowner. Eventually, a page is served from the actual owner.

3.1.1 Read Operation

Two processors are involved in each read operation, one is read fault handler (which is request for read access) and the other is read server (which is specified in probowner field). For read access, fault handler requests to a processor mentioned by probowner field. If read server is true owner of the requested page, then it needs to do following operations:

i. Add itself to copy set of requested page.
ii. Change access to "Read" in its PTable.

iii. Send page and page copy set to faulting processor.
iv. Add faulting processor in to probowner field of its PTable.

If the read server is not a true owner of requested page, then it forwards request to processor which is mentioned in probowner field of its PTable. It also updates its probowner field with requested node. Everytime a faulting processor receives a page copy, it updates its PTable along with probowner field with "self" and changes access to "read".

3.1.2 Write Operation

Write operation is also working same as read operation except invalidating pages according to the copy set. Two processors are also involved in each write operation, one is write fault handler (which requests for the write access) and other is write server (which is specified in probowner field). For write access, fault handler requests to a processor mentioned by probowner field. If write server is true owner of the requested page, then it needs to do the following operations:

1) Change access to "nill" in its PTable.

2) Send page and page copy set to faulting processor.

3) Add requested processor in to probowner field of its PTable.

If the write server is not the true owner of the requested page, then it forwards request to the processorhe true owner of the requested page, it forwards request to the processor mentioned in the probowner field of its PTable. It also updates its probowner field with the requested node. When faulting processor receives page copy, first it invalidates all copies from the copy set.

3.2 Formal Specification

In Dynamic Distributed Manager Algorithm of shared virtual memory, there are synchronized processes that we have previously discussed. We cultivate models for every synchronized process. We practice the UPPAAL tool suit [15][6] for modelling these processes. Modelling the "Dynamic Distributed Manager Algorithm of shared virtual memory" turned out to be suitable in definite situations to apply broadcast communications. Structures of UPPAAL with the broadcast frequencies and the dedicated positions let the broadcast communication categorized as the atomic arrangements of identical process organizations. Our foremost apprehension at investigates in this paper is to demonstrate the formal analysis of [14] through the use of UPPAAL. We deliver a complete examination of numerous protocol varieties in relation to the verification of complete functional requirements.

Let us discuss the summary of prescribed requirement in the toolset UPPAAL and then the formalism which is castoff in prescribed requirement of the Dynamic Distributed Manager algorithm. For demonstrating dynamic distributed manager algorithm in UPPAAL, we generate two local processes. These processes are named as processors and they request as well as serve all page read and write requests. These processes perform the following tasks:

1. Genrate a read fault.
2. Handle a read fault request.
3. Forward a read request to probowner.
4. Genrate a write fault.
5. Handle a write fault request.
6. Forward a write request to probowner.
7. Invalidate pages upon giving up ownership.

A processed named as Invalidate-process is modeled to address all the requests from all the processes when they want to invalidate old copies of pages. Essentially, the Invalidate-process behaves like a buffer to process requests one by one. A process invalidate pages according to the copy set while transferring ownership of a page. It means a a page is going to be updated and previously used copies of that page are invalidated. A process gets the updated copy of a page by generating a read fault request.

4. Results and Discussion

We present prescribed analysis of the dynamic distributed manager algorithm presented in [14]. The specification of this distributed algorithm in an automaton theoretic formalism is formalized and then functional requirements are verified.

4.1 Functional Requirements

We discover the following functional requirements of algorithm for formal analysis and formal verification.

R1: Deadlock freedom. No deadlock is supposed to be there when any processor request for the read or write the page in the system. System do not hang while anyone request for read, write or broadcast invalidate request

R2: Any Processor can get read access of any page.

R3: Any Processor can get write access of any page.

R4: When a processor request for read the page,then the processor must get the read access of the page. The true owner of the page must send the page copy to the requested processor.

R5: When a processor request for write the page. Then the processor must get the write access of the page. The true owner of the page must send the page write access to the requested processor.

4.2 Formal Specification of the Requirements

The principal requirement is that system does not contain any deadlock. According to requirement there is no valid deadlock in the system. The model must be deadlock free. So the query for verify this requirement is:

- A[] not deadlock

The query says that for all paths and states there is no deadlock. In query 'A' represent to all path and '[]' represent to all states. Any processor in the system can get read access of any page, this means any process can send read request for any page. Then the true owner of the page will send the page access to the processor page. The query for verify the R2 is given below:

- E<> forall (i:pro_id_t) forall (j:page_id_t) Process(i).PTable[j][1]==1

This query uses nested loop. Outer loop for the processor and inner loop is for the page. Query checks the read access from the PTable. Query verifies the read access for all pages under each processor. As described earlier the index 1 show the page access value whereas 0, 1 and 2 represent the nil, read and write access, respectively.

Any processor in the system can get the write access of any page, any process can send write request for any page. The true owner of the page sends the page for the write access to the requesting processor. The query for verify the R3 is given below:

- E<> forall (i:pro_id_t) forall (j:page_id_t) Process(i).PTable[j][1]==2. In the query 'E' represent to 'Some path' and '<>' represent to 'Some state'. This query is similar to the query in R2 except it checked the write access. For the write access, the value of index 1 in PTable must be equal to 2. When the processor requests for reading a page it must get read access of that page. So according to this requirement, when a page request for read page process reached at readFault state and when get the accesses it reached backed at the ideal. The formula of R4 requirement is given below.

- Process(1).ReadFault --> Process(1).Ideal

According to the R4 requirement, when Process (1) reached at the readFault state it will definitely go back to the ideal state. When the processor requests for write a page it must get the write access of that page. So, according to this requirement when a page request for write page, process reached at writeFault state and when get the accesses it reached backed at the ideal. The formula of R5 requirement is given below.

- Process(1).writeFault --> Process(1).Ideal

According to R5, when the Process (1) reaches at the writeFault state it definitely goes back to the ideal state.

5. Conclusions

We verify our model with respect to the given functional requirements and the results are shown in Table 3.1. We model the algorithm with 3 processors and 8 pages.

Table 1: Verification Results

Requirements	Results	Time	Memory
R1	Satisfied	19h25s	2.16GB
R2	Satisfied	0.046	102MB
R3	Satisfied	0.001s	29.16MB
R4	Satisfied	11.032s	100MB
R5	Satisfied	13.172s	102MB

We face some serious challenges in verifying and validating these requirements. Challenges are related to machine power in which we verify the requirements. First, we execute it on a machine with the specification, windows 10, 8GB RAM and Core i5 7th generation. On this machine, the model is executed for 20 minutes and then crashed due to state space problem. We also executed this query on MacBook 2016 with 16 GB RAM where it ran around 4 hours and then crashed. So, we need to execute it on a more powerful machine to verify this requirement. Other four requirements are satisfied with 2 processors and 2 pages. We present the results in Table 1.

References

[1] Venkateswarlu Chennareddy and Jatindra Kumar Deka. Formally Verifying the Distributed Shared Memory Weak Consistency Models. IEEE, 2006.

[2] Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In Dirk Beyer and Marieke Huisman, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 61–78, Cham, 2018. Springer International Publishing.

[3] Christel Baier, Joost-Pieter Katoen, et al. Principles of model checking, vol. 26202649. MIT Press Cambridge, 26:58, 2008.

[4] Johan Bengtsson, WO David Griffioen, Kåre J Kristoffersen, Kim G Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In Computer Aided Verification, pages 244–256. Springer, 1996.

[5] Lemerre M. Loulergue F. Blanchard A., Kosmatov N. A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. Springer, Cham, 2015.

[6] Peter Bulychev, Alexandre David, Kim Larsen, Marius Mikučionis, Danny Poulsen, Axel Legay, and Zheng Wang. Uppaal-smc: Statistical model checking for priced timed automata. EPTCS, 85, 07 2012.

[7] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR), 28(4):626–643, 1996.

[8] A. Fernandez G. David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. International Journal on Software Tools for Technology Transfer, 17:397–415, 2014.

[9] David Fabian and Radek Marik. Configuration dynamics verification using Uppaal. In Configuration Workshop, 2013.

[10] Y. Fei, H. Zhu, and X. Li. Modeling and verification of nlsr protocol using uppaal. In 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE), pages 108–115, Aug 2018.

[11] Eduardo Tovar Leandro Buss Becker Fernando Silvano Gonçalves, David Pereira. Formal Verification of AADL Models Using UPPAAL. IEEE, 2017.

[12] Christopher Gerking, Stefan Dziwok, Christian Heinzemann, and Wilhelm Schäfer. Domain-specific model checking for cyber-physical systems. 09 2015.

[13] Naseem Ibrahim and Ismail Khalil Al Ani. Verifying web services compositions using Uppaal. 12 2012.

[14] Paul Hudak and Kai Li. Memory coherence in shared virtual memory systems. In ACM Transactions on Computer Systems, 1989, Proceedings, The 5th Annual ACM Symposium on Principles of Distributed Computing, pages 321–359. ACM, 1989.

[15] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT), 1(1):134– 152, 1997.