Operational Characterization of Weak Memory Consistency Models

M. Senftleben(✉) and K. Schneider

TU Kaiserslautern, 67653 Kaiserslautern, Germany
{senftleben,schneider}@cs.uni-kl.de

Abstract. To improve their overall performance, all current multicore and multiprocessor systems are based on memory architectures that allow behaviors that do not exist in interleaved (sequential) memory systems. The possible behaviors of such systems can be described by so-called weak memory consistency models. Several of these models have been introduced so far, and different ways to specify these models have been considered like axiomatic or view-based formalizations which have their particular advantages and disadvantages. In this paper, we propose the use of operational/architectural models to describe the semantics of weak memory consistency models in an operational, i.e., executable way. The operational semantics allow a more intuitive understanding of the possible behaviors and clearly point out the differences of these models. Furthermore, they can be used for simulation, formal verification, and even to automatically synthesize such memory systems.

Keywords: Memory models · Weak memory consistency
Processor architecture · Memory architecture

1 Introduction

Historically, computer architectures were considered to consist of a single processor that is connected with a single memory via a bus (von Neumann architecture; 1945). The sequentialization of the read and write operations via the single bus ensured that each read operation returns the value most recently written to the corresponding memory location and that we can at all define *the most recently written value*. Even if the processor of such a computer architecture would be used to execute multiple processes by interleaving their executions, the memory operations would still take place one after the other and will therefore form a sequence where all memory operations are totally ordered.

Nowadays, essentially all computer architectures consist of multicore processors or even multiple processors which share a common main memory. Early multiprocessor systems still connected multiple processors via a single bus with the shared memory. This way, processors had to compete for bus access that still enforced an ordering of the memory operations in a linear sequence. Modern multiprocessor systems, however, are based on much more complex memory

© Springer International Publishing AG, part of Springer Nature 2018
M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 195–208, 2018.
https://doi.org/10.1007/978-3-319-77610-1_15

architectures that do not only make use of caches with cache coherence protocols, but also add further local memories to improve their performance. In particular, the use of local store buffers between the processor cores and the caches allows a significantly faster execution: Using store buffers, processors simply 'execute' store operations by putting a pair consisting of an address and a value to be stored at that address in a FIFO buffer. The processor can then continue with the execution of its next instruction and may consult its own store buffer in case a later load operation is executed. The store buffer will execute its store operations as soon as it is given access to the main memory. This avoids idle times due to waiting for the bus access for each store operation and allows a faster execution in general. However, since processors cannot see the store buffers of other processors, they will temporarily have different views on the shared memory. Note that after the store buffers were finally emptied, the cache coherence protocol ensures a coherent view on the shared memory, but before that point of time, the different views that exist due to the contents of the local store buffers allow executions that are otherwise impossible. For this reason, one speaks about weakly consistent memory models that do not impose as strong constraints as the traditional sequential memory models that just interleaved the memory operations of different processors.

Store buffers are one – but not the only – reason that lead to the introduction of weak memory consistency models [1,12,15,27]. For example, in distributed computer systems, the single memory is replaced by multiple distributed memories which can be specific to single processors or can be shared with all or some other processors. Depending on the implemented memory architecture, very different weak memory models were developed through the past decades, and some of them may lead to behaviors that are really unexpected by the programmers. It is therefore very important that the designers of modern computer systems are able to describe the potential memory behaviors of their systems in a precise but yet comprehensive way so that the programmers are able to determine when memory synchronization is required in their programs.

Memory consistency models have been defined in different ways: First descriptions of weak memory models were just given in natural language and were therefore often ambiguous. In fact, such ambiguous descriptions lead to non-equivalent versions of the processor consistency model [3,10].

Another way to define a memory consistency model is the so-called view-based approach where the different views processors may have during the execution of a multithreaded program are formally specified. From the viewpoint of a particular processor, this is usually done in that one has to determine which of the memory operations of other threads have to be interleaved with the memory operations of the own thread to define its local view. For example, for PRAM consistency, one would have to consider all store operations of all other threads, but not their load operations, while for other memory models other sets of store operations may be considered. The view-based approach can also be defined from the viewpoint of the memory, providing rules for the ordering of all operations as observed by the main memory. View-based definitions are quite popular, and

[27] showed how most of the existing weak memory models can be defined in this way. The authors of [27] even managed to organize many weak memory models in a hierarchy regarding their weakness, and they could describe most of the weak memory models systematically as combinations of four basic constraints.

However, the view-based approach remains quite abstract and formal, and while being precise for a formal analysis [9], it is not comprehensive enough to serve as a general description for programmers. A slightly different approach has been followed by the SPARC memory models TSO and PSO that are described in an axiomatic way [28]. Also being view-based in principle, these weak memory models were specified by just a few axioms that can be directly used for formal reasoning about the potential executions of a multithreaded system. While also lacking of comprehensiveness, these descriptions are much more succinct, and allow one to directly make use of formal verification that is not that directly applicable when the views are defined by a couple of total or partial orders.

More recent efforts made use of theorem provers to specify weak memory models, using e.g., higher order logic [18,22] or temporal logic [25]. The motivation for this choice is to ensure the well-definedness of the given non-trivial formalization, and to directly reason about properties of the specified memory models with verification tools. However, also these approaches tend to be too difficult to be used as a reference for programmers.

From programming languages, it is well-known that besides the axiomatic and denotational semantics, the operational semantics is often preferred for defining simulators or virtual machines [6]. Usually, programmers also prefer operational semantics, obviously since that kind of semantics directly determines how the programs are executed. Operational semantics are therefore usually the best means to define programming models.

In this paper, we therefore advocate the use of operational semantics for the specification of weak memory consistency models. We believe that operational semantics may also lead to formally precise, but still comprehensive specifications of weak memory consistency models that might be preferable for programmers. To specify such an operational description, one has to define for each weak memory model an abstract memory architecture with load/store ports for the processors. These operational/architectural models can be described using modern system-level languages to obtain precise and executable models. Using these operational models in teaching, we found that students were able to much better and quicker understand the subtle differences between the memory models. Moreover, our operational models can be directly used for simulation, formal verification, and also for synthesis. In particular, we will list such operational/architectural models for the important memory consistency models described in literature [5,10,14,16]. These reference architectures are obtained by directly deriving implementations of memory systems from the definitions of their memory consistency model. The resulting reference machines do not claim to be efficient (for synthesis), but minimal in terms of different components and structures to simplify verification of correctness and completeness of the implementation.

The outline of the paper is as follows: In the next section, we briefly review related work and then define some weak memory models according to [27]. Section 4 contains the core of this paper where we present operational architectures for the models considered before. Finally, we discuss future work with preliminary conclusions.

2 Related Work

Comprehensive introductions to memory consistency models are [1,19]. Reference [20] provides a good overview over many of the models known at that time and compares these with each other.

The formalism and some of the definitions used in this publication are based on [27] which introduced a systematic framework for view-based definitions for many common memory consistency models and revealed the relations between different memory models. In the next section, we list four of the many weak memory models of [27] which are described in an operational way afterwards. Similarly, [2,4] provided unified formalizations for multiple memory models. Other work on view-based definitions include [3,5]. Reference [26], on the other hand, introduces a framework for axiomatic definitions.

In our own previous work, we analyzed in [8,9] the complexity of testing whether given execution traces comply with a certain memory model. Similar to this publication, [13] provided definitions and comparisons of several consistency models and defined machines for the models.

The only previous works we are aware of that made also use of operational semantics were Lipton and Sandberg [16] who provided an implementation for PRAM by defining its structure and communication rules. Recently, [7] described the semantics of the ARMv8 multiprocessor architecture with an operational approach. Our approach is more general, and claims to have the potential to be used to describe most known weak memory models in an operational, and thus comprehensive way.

3 View-Based Definitions of Memory Consistency Models

In this section, we adapt the terminology introduced by Steinke and Nutt in [27] to provide formal definitions of four weak memory consistency models. Note that this formalism does not describe the multithreaded system in an executable/operational way. Instead, it just determines the set of possible executions in terms of possible traces of memory operations, but does not explain how or why these were generated. In the next section, we will then provide operational models for the memory models considered in this section.

In this setting, a memory operation is expressed as a quadruple (o, p, l, v) where o is either a read or write operation, p is the process id of the process executing the operation o, l is the memory location (address), and v is the value read or written. The local order relation $<_i$ reflects the execution order of all memory operations of process P_i in that it orders the operations according to the

program code of the process. The process order $<_P$ is the conjunction $\bigwedge_{i \in P} <_i$ of all local order relations. Definitions of memory models can then be given in the following form:

$$\forall_{i \in P} \exists SerialView\,(<|\,(*, i, *, *) \cup (w, *, *, *))$$

which means that for each process P_i, there must exist a serial view on all its own operations (using wildcards $(*, i, *, *)$) and all write operations of all processes $((w, *, *, *))$ which respects the ordering $<$. A serial view is thereby a total order $\triangleleft$ over the given set of operations and a superset of the provided relation $<$. Furthermore, in a serial view $\triangleleft$, the memory value of each read operation has to correspond to the most recent write operation to that location with respect to $\triangleleft$. This implicitly defines the writes-to order $w \mapsto r$ which maps each read operation r to the write operation it reads from.

3.1 Local Consistency

Local consistency was first defined by Heddaya and Sinha [11] as the weakest constraint that could be required of a shared memory system. In a locally consistent system, each process observes its own operations in local order while all other operations may be observed in an arbitrary order. Different processes' orders are not related at all in this memory model. Local consistency [5,11] can be expressed in the introduced formalism as follows [27]:

$$\forall_{i \in P} \exists SerialView\,(<_i|\,(*, i, *, *) \cup (w, *, *, *))\,.$$

3.2 Cache Consistency (CC)

In 1989, Goodman [10] provided a definition for cache consistency, which he called weak consistency since he assumed that it is the weakest form of memory consistency. Furthermore, he expected that no synchronization guarantees could be given in cache consistency. Meanwhile, both assumptions have been proven wrong by the existence of weaker models and algorithms that can ensure mutual exclusion in weaker models like slow consistency. Cache consistency [10] can be defined as follows, which means that each process observes the same ordering on memory operations regarding the same memory location, but processes may see operations regarding different memory locations in different orders:

$$\forall_{x \in V} \exists SerialView\,(<_P|\,(*, *, x, *))$$

3.3 Pipelined-RAM (PRAM) Consistency

One of the first well known weak memory models described was PRAM (Pipelined RAM) which was presented 1988 by Lipton and Sandberg [16,17].

They show that their shared memory system PRAM scales better than sequentially consistent systems as it is immune to high network latency. Additionally, synchronization costs remain low while performance increases significantly. Due to its informal textual definition, there exists an interpretation of Ahamad et al. [3], and another slightly different one by Mosberger [21] as shown in [24]. PRAM consistency based on Ahamad et al. [3] can be expressed as follows:

$$\forall_{i \in P} \exists SerialView\,(<_P |(*, i, *, *) \cup (w, *, *, *))$$

In a PRAM consistent execution, every process observes all the writes of all other processes in the order they were issued. However, different processes may see the writes of the other processes in a different order. A system implementing PRAM consistency therefore only has to ensure that the communication from one process to another does not reorder or lose writes, while the transmission delay between processors is arbitrary.

3.4 Sequential Consistency (SC)

While technically not a weak model, we include sequential consistency as defined by Lamport [14] as a base reference. Sequential consistency has been the preferred memory model for programmers since it just considers the interleaving of the single thread executions. The definition of sequential consistency [14] as expressed by [27] is:

$$\exists SerialView\,(<_P |(*, *, *, *))$$

which means that a system is sequentially consistent if for all executions, there exists a corresponding sequential order for all operations which respects the process order.

4 Operational Definitions of Memory Consistency Models

This section contains the main contribution of our paper, i.e., the operational/ architectural characterizations of the four weak memory models described in the previous section. To this end, we provide reference machines for each one of these memory models, and discuss then their correctness and completeness, i.e., that these reference machines can only execute computations that belong to the considered memory model (correctness), and that the reference machine can execute *all* computations that belong to the considered memory model (completeness). We have also developed reference machines for other memory models during our research [24] but these cannot be included in this paper due to lack of space.

In order to discuss these reference machines, we first introduce some common basic components in the next section. Then, the reference machines are presented and their correctness and completeness are briefly discussed. Finally, further details on the actual implementation in the synchronous language Quartz [23] are given.

4.1 Basic Components

The reference machines in this section are constructed with the basic components described in this section.

FIFO: The FIFO component is a First-In-First-Out Buffer which buffers memory operations as tuples. It holds the operation type (read or write), the issuing process' id, the memory address, and in case of a write operation the value to be written. The component's interface is defined as follows:

```
module FIFO(
    event ?pop,
    event ?push,
    event !isempty,
    event isfull,
    // input : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
    // output : writeCommand & target & value
    event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp
)
```

The outputs `isempty` and `isfull` signal the current state of the buffer. Both data channels `inp` and `outp` consist of a valid flag, the id of the originating processor, the memory location to write to and the actual value to write. Adding an entry to the buffer is handled by input signal `push` while providing the data to `inp`. Similarly, removing the first entry of the buffer is handled by input signal `pop` and reading data from `outp`.

BAG: The BAG component shares the same interface as the FIFO component but slightly differs in its semantics: While the FIFO component will always return and remove the oldest entry when signal `pop` is set, the BAG component may non-deterministically return and remove any stored entry.

MEM: The memory unit MEM stores the latest write to a location and returns for read operations the most recently written value of a location.

In the next subsections, we discuss reference machines for local consistency, cache consistency, PRAM consistency, and sequential consistency. To that end, we will first describe the architecture of the reference machine using the above mentioned basic components. After this, we briefly discuss the correctness and completeness of the given reference machine, where *correctness means that all computations of our reference machine belong to the considered weak memory consistency model*, and conversely, *completeness means that our reference machine can simulate all possible executions of the considered weak memory consistency model*. Hence, the reference machines exactly characterize the weak memory consistency model in an operational/architectural manner.

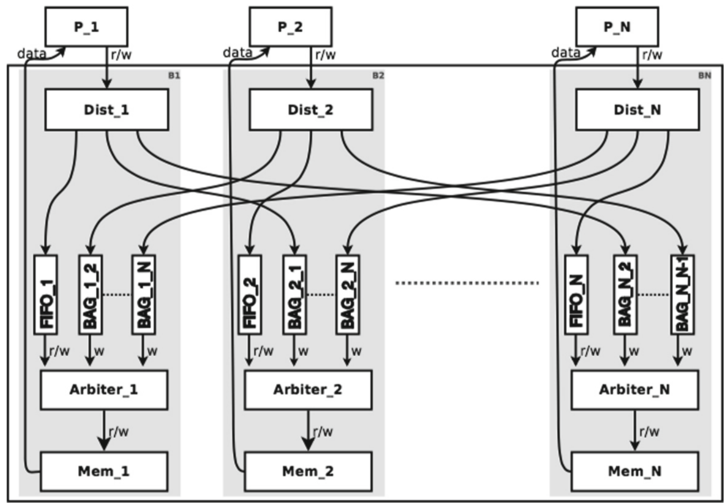

Fig. 1. Reference machine for local consistency

4.2 Reference Machine for Local Consistency

Architecture: The implementation of the reference machine for local consistency is shown in Fig. 1 for a given set P of n processes and m memory locations. For each process $P_i \in P$, the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i, a FIFO buffer $FIFO_i$, and $n-1$ different BAG structures $BAG_{i,j}$ with $j \in \{1, \ldots, n\}, j \neq i$. A distributor $Dist_i$ broadcasts received writes to its $FIFO_i$, and all corresponding $BAG_{j,i}, j \in \{1, \ldots, n\}, j \neq i$, and sends all received reads to its $FIFO_i$. The arbiters nondeterministically decide to idle or to nondeterministically choose a read from the connected FIFO and BAG structures. Any operation read from the selected FIFO or BAG is forwarded to the memory unit.

Correctness: By construction, a process' own memory operations are kept in order in the FIFO maintaining $<_i$. The arbiters generate a serial view covering all own ordered operations and all others' write operations.

Completeness: Consider now an arbitrary locally consistent execution. According to its definition, a serial view exists for each process. Now, the arbiter can choose to read from the BAG/FIFO structures as the order of the serial view suggests, or to idle as long as the next required value is not yet available. The given architecture allows to wait until the required values are available and therefore covers the required behavior.

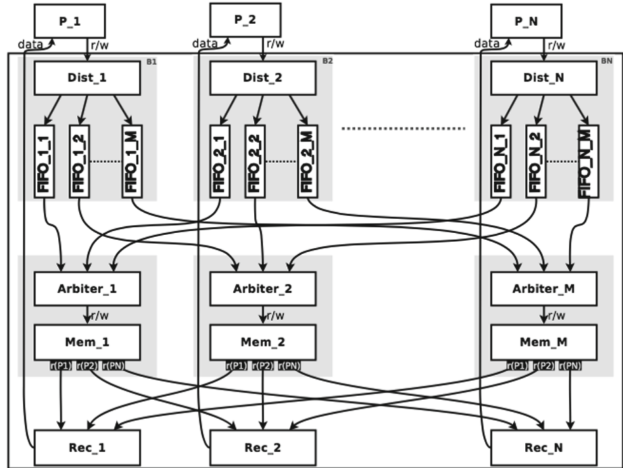

Fig. 2. Reference machine for cache consistency

4.3 Reference Machine for Cache Consistency

Architecture: The implementation of the reference machine for cache consistency is shown in Fig. 2 for a given set P of n processes. For each process $P_i \in P$, the memory system has a distributor $Dist_i$, a receiver Rec_i, and m different FIFO buffers $FIFO_{i,j}$ for $j \in \{1, \ldots, m\}$. For each memory cell M_j, the memory system provides a memory unit Mem_j and an arbiter $Arbiter_j$. A distributor $Dist_i$ passes the received memory command for memory cell M_j to the corresponding $FIFO_{i,j}$. The arbiters choose nondeterministically from the connected FIFOs to read from. The memory unit returns the result of a read operation to the receiver Rec_i of process P_i. The receiver Rec_i receives reads for its process and returns them to the process' data interface.

Correctness: The use of FIFO buffers ensures by construction that the read and write operations regarding a specific memory location of each process are kept in order (maintains $\leq_{PO}$ per variable). Therefore, each arbiter $Arbiter_j$ constructs a serial view on all read and write operations regarding its memory location j.

Completeness: Consider now an arbitrary cache consistent execution. If each arbiter selects its action according to the executions' serial view corresponding to its memory location, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. As no memory operations are lost, and the serial views adhere to the process order, it cannot be the case that the next required value is stuck behind another value in one of the FIFOs. Therefore, each arbiter can idle until eventually the next required operation will be available at the head of one of the connected FIFOs.

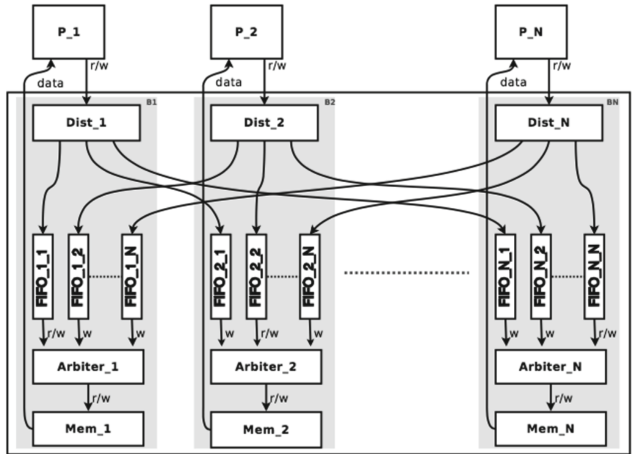

Fig. 3. Reference machine for PRAM consistency

4.4 Reference Machine for PRAM Consistency

As can be seen in Fig. 3, the reference machine for PRAM consistency provides a single memory for every process, so that this kind of memory model is typically found in distributed computing.

Architecture: The implementation of the reference machine for PRAM consistency is shown in Fig. 3 for a given set P of n processes. For each process $P_i \in P$, the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i, and n different buffers $FIFO_{i,j}$ for $j \in \{1, \ldots, n\}$. A distributor $Dist_i$ broadcasts received writes to all corresponding $FIFO_{i,j}$ for $j \in \{1, \ldots, n\}$, and sends all received reads to its $FIFO_{i,i}$. The arbiters choose nondeterministically from the connected FIFOs.

Correctness: Using FIFO buffers ensures by construction that the read and write operations of each process are kept in order (maintaining $\leq_{PO}$). The arbiter takes elements from the top of a FIFO buffer and issues the operation to the memory unit. Therefore, the arbiter constructs a serial view on write operations of all processes and the read operations of its corresponding process.

Completeness: Consider now an arbitrary PRAM execution. If each arbiter selects its actions according to the execution's serial view corresponding to its process, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. As said before, as no writes are lost and an arbiter can always wait until the required value is available, every PRAM consistent execution is covered by the reference machine.

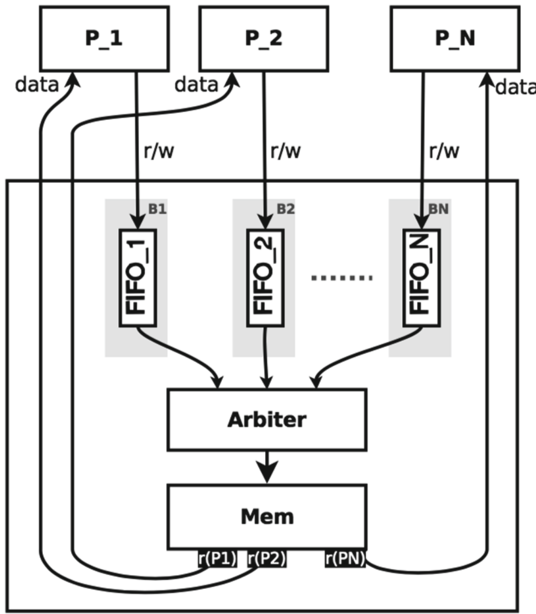

Fig. 4. Reference machine for sequential consistency

4.5 Reference Machine for Sequential Consistency

Architecture: Figure 4 shows an implementation of a reference machine for a sequential consistency. It consists of a FIFO buffer for each connected process, which are directly connected to their process interface, an arbiter which selects nondeterministically from all FIFOs and passes the operations to the memory unit or otherwise idles. The memory unit passes processed reads to the process that issued the read operation.

Correctness: Using FIFO buffers ensures by construction that the read and write operations of each process are kept in order (maintaining $\leq_{PO}$). The arbiter generates a serialization of all memory operations while maintaining the process order and therefore satisfies sequential consistency.

Completeness: If an arbitrary execution is sequentially consistent, then a serial view exists for all memory operations which respects $<_P$. If the arbiter uses this view to make its nondeterministic choices, then the resulting behavior is equivalent to the considered execution. Consequently, all sequentially consistent executions are covered by the given reference machine.

4.6 Implementation of Reference Machines

We have implemented all reference machines discussed in the previous sections as well as many others in the synchronous programming language Quartz [23]. The complete implementations can be found in [24].

To test their correctness, programs like Dekkers mutual exclusion protocol and programs derived from memory model litmus test suites have been successfully run on the presented reference machines.

While the introduced reference machines require unbounded buffers and true nondeterminism to guarantee the completeness of the memory model, their implementations in a system description language like Quartz have to specify bounds for such structures. Clearly, since we can determine the required buffer sizes for each finite execution, it is still possible to ensure the completeness. For simulation purposes, we can resort to randomizing the nondeterministic choices, and for verification or coverage checking, those have to be handled by oracle inputs which are controlled by the underlying tools.

5 Conclusions and Future Work

This paper presents reference machines to characterize weak memory consistency models in an operational manner. We have implemented these reference machines in the synchronous language Quartz so that their behaviors are precisely determined by the formal semantics of Quartz. All reference machines were implemented by means of some basic components that clearly reflect the intention of the considered memory model. The resulting reference machines are useful for simulation and verification, and can serve as a comprehensive specification that can be used as a programming model.

We have proved the correctness and completeness of our reference machines, i.e., that the reference machines can only perform computations that belong to the weak memory model (correctness), and that all possible computations of the memory model can also be performed by our reference machines (completeness). Hence, our reference machines characterize the memory models in an operational manner.

In our future work, we would like to develop reference machines closer to real implementations. As stated before, the provided implementations aimed to be both correct and complete following the corresponding definitions. As a result, their structure is more complex as a real implementation would be, including both unboundedness and nondeterminism which are not wanted in real implementations.

Furthermore, as we want to observe the behavior of programs developed for sequential machines on weak memory models, the memory models were all analyzed and defined without synchronization operations. Multicore processors offer synchronization operations for enforcing a desired behavior if needed. Therefore, it might be of interest to include in future also synchronization operations like fences in our architectures.

References

1. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
2. Adve, S., Hill, M.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. (TPDS) **4**(6), 613–624 (1993)
3. Ahamad, M., Bazzi, R., John, R., Kohli, P., Neiger, G.: The power of processor consistency. In: Snyder, L. (ed.) Symposium on Parallel Algorithms and Architectures (SPAA), pp. 251–260. ACM, Velen (1993)
4. Alglave, J.: A formal hierarchy of weak memory models. Form. Methods Syst. Des. (FMSD) **41**(2), 178–210 (2012)
5. Bataller, J., Bernabeu, J.: Synchronized DSM models. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 468–475. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0002771
6. Bruni, R., Montanari, U.: Models of Computation. Texts in Theoretical Computer Science. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-42900-7
7. Flur, S., Gray, K., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: Principles of Programming Languages (POPL), pp. 608–621. ACM (2016)
8. Furbach, F., Meyer, R., Schneider, K., Senftleben, M.: Memory model-aware testing - a unified complexity analysis. In: Application of Concurrency to System Design (ACSD), pp. 92–101. IEEE Computer Society, Tunis La Marsa (2014)
9. Furbach, F., Meyer, R., Schneider, K., Senftleben, M.: Memory-model-aware testing – a unified complexity analysis. Trans. Embed. Comput. Syst. (TECS) **14**(4), 63:1–63:25 (2015)
10. Goodman, J.: Cache consistency and sequential consistency. Technical report 1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991
11. Heddaya, A., Sinha, H.: Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical report BU-CS-92-004, Department of Computer Science, Boston University (1992)
12. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, Burlington (2003)
13. Higham, L., Kawash, J., Verwaal, N.: Weak memory consistency models - part I: definitions and comparisons. Technical report 98/612/03, Department of Computer Science, University of Calgary (1998)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. (T-C) **28**(9), 690–691 (1979)
15. Lawrence, R.: A survey of cache coherence mechanisms in shared memory multiprocessors (1998)
16. Lipton, R., Sandberg, J.: PRAM: a scalable shared memory. Technical report CS-TR-180-88, Princeton University (1988)
17. Lipton, R., Sandberg, J.: Oblivious memory computer networking. Patent US 5276806, January 1994
18. Mador-Haim, S., et al.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_36
19. McKenney, P.: Memory barriers: a hardware view for software hackers, June 2010. http://www.rdrop.com/users/paulmck
20. Mosberger, D.: Memory consistency models. ACM SIGOPS: Oper. Syst. Rev. **27**(1), 18–26 (1993)

21. Mosberger, D.: Memory consistency models. Technical report TR 93/11, Department of Computer Science, The University of Arizona, Tucson, Arizona, USA (1993)
22. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
23. Schneider, K.: The synchronous programming language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009
24. Senftleben, M.: Operational characterization of weak memory consistency models. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, March 2013
25. Senftleben, M., Schneider, K.: Specifying weak memory consistency with temporal logic. In: Ghazel, M., Jmaiel, M. (eds.) Verification and Evaluation of Computer and Communication Systems (VECoS). CEUR Workshop Proceedings, vol. 1689, pp. 107–122. Sun SITE Central Europe, Tunis (2016). http://ceur-ws.org/Vol-1689/
26. Sindhu, P., Frailong, J.M., Cekleov, M.: Formal specification of memory models. In: Dubois, M., Thakkar, S. (eds.) Scalable Shared Memory Multiprocessors, pp. 25–41. Kluwer, Dordrecht (1992)
27. Steinke, R., Nutt, G.: A unified theory of shared memory consistency. J. ACM (JACM) **51**(5), 800–849 (2004)
28. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual-Version 9. Prentice-Hall Inc., Upper Saddle River (1994)