Specifying Weak Memory Consistency with Temporal Logic

Maximilian Senftleben and Klaus Schneider

Department of Computer Science University of Kaiserslautern http://es.cs.uni-kl.de

Abstract. Modern multiprocessors only provide weak memory consistency that deviates from the traditional sequential consistency in many unintuitive ways: In essence, the read and write operations issued by one processor might be observed by other processors in a different order than executed by the issuing processor. It is therefore strongly recommended to formally verify multithreaded systems. To this end, it is necessary to offer unambiguous and complete descriptions of the used memory consistency models. Different formal methods have been used so far to specify the possible executions under particular weak memory consistency models. However, some of them cannot be directly used for verification by model checking. In this paper, we present a new way for specifying weak memory consistency models that is based on temporal logic. Using Linear Temporal Logic (LTL) to define weak memory systems, we can easily check properties of one and the same multithreaded program against several consistency models. This way, we can determine the weakest consistency guarantees required to ensure given specifications.

Keywords: temporal logic, model checking, weak memory consistency, specification.

1 Introduction

Communication between several threads over a shared memory is one of the major bottlenecks in modern multiprocessor systems. As processors have to compete for accessing the shared memory and have to synchronize their memory operations, the processor performance would significantly decrease with an increasing number of processors if no countermeasures would be taken: all modern multicore and multiprocessor systems therefore employ optimizations in their memory architecture to further increase their performance.

While some of these optimizations do not modify the result of programs like the use of local caches with a snooping-based cache coherence protocol [27], other optimizations like store buffers might sometimes give unexpected results [33]. To be more precise, snooping-based cache coherence protocols maintain the sequentially consistent behavior [20] as known from the execution of processes on uniprocessor systems while other optimizations weaken the consistency to allow further behaviors that are not possible by simply interleaving the behaviors of the threads. A memory system is thereby called weak if different processors may disagree about the order of observed memory operations.

As single-threaded programs still account for a large part of the workload of general purpose processors, weak memory goes unnoticed most of the time. Multithreaded programs which rely on inter-thread communication however may result in different results when executed on a single core or multiple cores.

```
bool b;
int v;
thread P() {
    v = value;
    b = true;
}
thread Q() {
    while(not(b)) {};
    r = v;
}
```

Fig. 1. Two threads P and Q demonstrating weak memory consistency.

For example, consider the two processes P and Q shown in Figure 1 and assume variable b has initially the value false. Process P writes a value to variable v and then signals its completion by setting another variable b to true. Process Q observes variable b, and as soon it finds a value true in b, it proceeds with reading the value of variable v. Obviously, this program is written with the assumption that write operations of one process (P) are observed in the same order by all other processes (Q). If we relax this restriction in a weak memory model, then it may happen that Q first observes the assignment to b and proceeds with reading from v even though the assignment to v is not yet visible to Q, resulting in an unexpected behavior.

Each possible optimization of a memory system may result in different possible behaviors, and therefore define their own so-called memory consistency model. For example, current multicore processors maintain for each core a load/ store buffer where they buffer their load/store instructions to be issued to the memory system. While these instructions may not yet have been accepted by the memory system, they can already be seen by the processor core that owns the buffer. If even different buffers were used by one core, e.g., for different memory banks, this leads to a weak memory consistency model where assignments executed later by a processor may arrive first in memory.



Fig. 2. Hierarchy of weak memory models (see [32]).

Many different optimizations of memory architectures have been implemented so far that lead to different weak memory consistency models. Figure 2 gives an overview of some of these models (see also [24, 1, 32, 13] for recent surveys). A memory model is thereby called *weaker* than another one if it allows more possible executions as denoted by the arrows in the figure. For example, all executions that are sequentially consistent (SC) are total store order consistent (TSO) as well, but there may exist TSO-consistent executions that are not sequentially consistent.

Since the presence of weak memory models often leads to unintuitive behaviors, it is strongly recommended to formally verify multithreaded programs taking the underlying weak memory model of the hardware platform into account. To this end, one obviously needs precise formal descriptions of the possible behaviors of each model that can be used for formal verification of the multithreaded programs. However, there is still no commonly accepted formalism to define weak memory models which makes it also very difficult to formally reason about them (see next section). The original descriptions of the weak memory models were unfortunately written only in natural language and are therefore often ambiguous. Recent efforts use formal methods to specify weak memory models, as e.g., higher order logic [26], and partial orders [32], which are both not well-suited for model checking.

In this paper, we propose the use of temporal logic as a new way to formally define weak memory models. We emphasize in this paper that the so far mainly used weak memory models can all be conveniently defined by temporal logic. This directly allows us to formally verify temporal logic specifications for a given system with different weak memory models in a flexible way. In particular, we can determine the minimal requirements for a memory system for correctly implementing a multithreaded system. The outline of the paper is as follows: Section 2 surveys the state of the art in specifying weak memory consistency models. Section 3 is the core of the paper that presents the use of linear temporal logic for specifying different weak memory models. In Section 4, we present first experimental results, and finally list first conclusions in the final section of the paper.

2 Related Work

[23] as well as [1] provide recommendable introductions to memory consistency in general. [24] gives a good overview over many of the models known at that time and compares these with each other. [32] introduced a unified framework that derives weak memory models as the composition of four independent basic properties and defined this way a lattice of memory models. In our previous work, we analyzed the complexity of testing whether given execution traces comply with a certain memory model in [12, 13]. In most cases, this problem is NP-complete even if some parameters like the number of variables are kept constant. Other unified frameworks were introduced by [2], [18], and [4].

The first descriptions of weak memory models were only given in an informal manner and sometimes lead to misinterpretations, e.g., the interpretations of [24] and [3] for PRAM-consistency which was informally introduced by [21] were different: In contrast to the definition by [3], the definition by [24] assumes that a processor first updates its local memory before it broadcasts a write operation to other processors. When manufacturers recognized the impact of weakly consistent systems on programming, they provided distinguishing test cases (so-called litmus tests [15,5]). Litmus tests are either examples of possible weak behaviors or examples of behaviors that may not occur at all[15,5].

It became quickly clear that the inherent incompleteness given by sample programs as well as the ambiguity given by informal definitions are inadequate for any kind of formal reasoning about multithreaded programs. [25] discusses the problems that arise from ambiguously defined memory models in modern architectures and high-level languages, and [29] revealed that the Java memory model was flawed for many years.

It is therefore very important to come up with formally precise, but still comprehensive definitions of memory models. There are already many established ways of specifying the behavior of different memory systems. In the remainder of this section, we classify the so-far given approaches in different categories.

2.1 Operational Definitions

Providing a full system description even at an abstract level would provide an operational semantics of a memory system, but due to the proprietary nature of processors such descriptions are usually not publicly available. Nonetheless, in some cases like publicly available processor cores, the system implementation might be available, and in other cases, an operational semantics can be re-engineered from an available formal definition [7, 31]. However, the operational models introduce a big burden for formal verification since, in addition to the multithreaded program, one also has to model the underlying hardware platform with its memory transactions for the verification.

2.2 Axiomatic Definitions

Another way to specify a weak memory model is to list axioms that have to be satisfied by the allowed behaviors. This has been done first for the SPARC processors [33], but would have to be re-engineered for the other models. Axiomatic semantics makes use of quantifiers and often of higher order logic [26], and is therefore not well-suited for verification by model checking.

2.3 View-based Definitions

View-based definitions describe the ordering of operations seen from the individual processors or the memory system. This type of definition is superior in that it allows one to abstract from the actual implementation and focuses on the processors' view which simplifies its use for the programmer. Some view-based definitions use the notion of an execution which consist of write and read operations that might occur in that way, more specifically, the read operations are already annotated with the value they will read. A benefit of view-based definitions is that they allow to introduce unified ways of defining different memory models as shown by [32]. This concept can be helpful for analyzing specific cases, but may be hard to follow, as it is rather related with litmus tests than with actual programs because in a program, we usually do not know the read value in advance.

3 Specifying Weak Consistency by Temporal Logic

As early as 1981, it was suggested by [28] that temporal logics would be a suitable formalism to describe the behavior of concurrent systems. Since then, specifying and verifying the concurrent behavior of reactive and other systems became one of the success stories of computer science [16] and lead to many Turing awards. Surprisingly, to the best of our knowledge, no one described the behavior of different weak memory models using temporal logic until today. Therefore, this is the first paper that systematically introduces linear time temporal logic (LTL) as an alternative, uniform way to specify weak memory models. In the following, we will first define the general setting using LTL for specifying weak memory models in Section 3.1, and will then apply this to specific memory models in Section 3.2.

3.1 The General Setting

Linear Temporal Logic (LTL). LTL is a variant of temporal logic whose models are single execution paths of a system. As LTL is based on a discrete notion of time, each point of time can be denoted as an integer value. The semantics is defined for a labelled transition system (a Kripke structure) K = (S, I, R, L) that consists of a set of states S, initial states $I \subseteq S$, a transition relation $R \subseteq S \times S$, and the label function L that maps each state to the variables that hold there. Each path consists of a sequence of states determined by the transition relation R. An LTL formula is satisfied for a given structure K if it is satisfied for all infinite paths starting in any of the initial states I.

The temporal operators used in this paper are:

- $G\varphi$ (Globally): φ holds in the current state and all future states.
- $F\varphi$ (Finally): φ eventually holds at least once (now or in future).
- X φ (Next): φ holds in the next state of the path.
- $[\psi \cup \varphi]$ (Until): φ holds until the first time when ψ holds, and ψ has to hold eventually.
- $[\psi \overline{\mathsf{U}} \varphi]$ (Weak Until): φ holds until the first time when ψ holds, and ψ may never hold (in which case φ holds ad infinitum).

For more information about temporal logics, see e.g., [11, 22, 6, 10, 30].

Read/Write Events. In general, a multiprocessor system can be modeled by a set of processors that execute a set of programs connected to a central memory system via a well-defined interface. In the following, we will provide LTL specifications for various memory consistency models. The specifications describe the behavior of the memory system in a specific environment. The environment consists of multiple processors which interact with the memory system via events. For each processor p, we distinguish write events W^p and read events \mathbb{R}^p . A global unique identifier (an integer value) is assigned to each write event W^P denoted by $Id(W^P)$. We write W_i^p as a shorthand for $W^p \wedge$ $Id(W^p) = i$. A write instruction which is issued multiple times, e.g., in a loop, will result in different write events with different identifiers. The expression $Loc(W^p)$ denotes the memory location to write to, and $Val(W^p)$ the value to be written. Similarly, $Loc(\mathbb{R}^p)$ denotes the memory location to read from, and $Val(\mathbb{R}^p)$ the value returned by the memory system. Note that only $Val(\mathbb{R}^p)$ is an output of the memory system and everything else is considered as an input. In addition to the mentioned events W^p and R^p , the specifications use an additional event ^qO which models that a processor q has observed a write event. $Prc(^{q}O)$ denotes the originating processor and $Id(^{q}O)$ the identifier of the observed write. For conciseness, we write ${}^{q}O_{i}^{p}$ for ${}^{q}O \wedge Prc({}^{q}O) = p \wedge Id({}^{q}O) = i$. A processor is assumed to observe at most one write at a time. Similarly to the previous definitions, $Loc(^{q}O)$ denotes the memory location, and $Val(^{q}O)$ the value of the observed write.

Minimal Requirements for all Models. To provide a well-defined context for the specifications, the environment has to fulfill certain assumptions. First, quite obviously the same event should not occur more than once (1), next the identifiers of the writes should be globally unique (2), and for each processor strictly increasing (3). Furthermore, it is assumed that a processor only issues either a write or a read event at a time (4).

$$\mathsf{G}(W_i^p \to \mathsf{XG}(\neg W_i^p)) \tag{1}$$

$$\mathsf{G}(W_i^p \to \neg \bigvee_{q \neq p} (W_i^q)) \tag{2}$$

$$\mathsf{G}(W_i^p \to \mathsf{XG}(\bigwedge_{j \le i} \neg W_j^p)) \tag{3}$$

$$\mathsf{G}(\neg(W_i^p \land R^p)) \tag{4}$$

A system has to satisfy at least some basic properties to be considered a reasonable memory system. First, observation events should be causally related to write events, i.e., an observation event O_i^p may only occur if there was a corresponding write event W_i^q before (5). Next, a processor should observe each write event only once (6). Naturally, we would like Loc(O) and Val(O) to return the same values as their corresponding write event (7).

$$\mathsf{G}\left[\left(\neg {}^{q}O_{i}^{p}\right)\overline{\mathsf{U}}W_{i}^{p}\right] \tag{5}$$

$$\mathsf{G}({}^{q}O_{i}^{p} \to \mathsf{X}\mathsf{G}\neg\,{}^{q}O_{i}^{p}) \tag{6}$$

$$\mathsf{F}(W_i^p \wedge Loc(W^p) = l \wedge Val(W^p) = v)$$

$$\rightarrow \mathsf{C}({}^q \mathcal{O}^p \rightarrow Loc({}^q \mathcal{O}^p) - l \wedge Val({}^q \mathcal{O}^p) - v)) \tag{7}$$

$$\rightarrow \mathbf{G}({}^{\prime}O_{i}^{\prime} \rightarrow Loc({}^{\prime}O_{i}^{\prime}) = l \land Val({}^{\prime}O_{i}^{\prime}) = v)) \tag{1}$$

Read operations should either return the default value (in this paper denoted as \perp) as long as there was no observed write to that location (8) or the value of the latest observed write event (9).

$$([R^q \wedge Loc(R^q) = l] \rightarrow Val(R^q) = \bot) \overline{\mathsf{U}} (^qO \wedge Loc(^qO) = l)$$
(8)

$$\mathsf{G}\left(\left({}^{q}O_{i} \wedge Val({}^{q}O) = v \wedge Loc({}^{q}O) = l\right) \rightarrow \left[\left(R^{q} \rightarrow Val(R^{q}) = v\right) \overline{\mathsf{U}}\left({}^{q}O \wedge Loc({}^{q}O) = l \wedge Id({}^{q}O) \neq i\right)\right]\right)$$
(9)

3.2 Specifying Particular Memory Models

In the following, we will introduce the additional properties that are needed to specify some of the known memory models.

Local Consistency. Local consistency was defined by [17] as the weakest memory model. It requires each process to observe its own writes in the order they were issued, but allows other processes' writes to be observed in any order. This correlates to the *Read-My-Writes* consistency property known in the database community. This property can be expressed in LTL as follows:

$$\mathsf{G}(W_i^p \to {}^pO_i^p)$$

In other words, a write event requires the issuing processor to immediately observe its own write. Figure 3 shows an example execution for local consistency. It shows that local consistency allows reordering of writes of another processor, i.e., only the issuing processor is required to observe its own writes in order.

Write(X = 1)	ا ۲	$\operatorname{Read}(X =$	2)
Write(X=2)	¥	$\operatorname{Read}(X =$	1)

Fig. 3. TestLocal: Local consistent execution which is not Slow consistent.

Slow Consistency. Slow consistency [19] was introduced to increase memory performance by reducing consistency maintenance. Slow consistency extends local consistency by requiring processors to observe the writes of another processor to the same location in the order they were issued. The corresponding LTL representation of that property is as follows:

$$\mathsf{G}\big({}^{q}O_{i}^{p} \to \mathsf{XG}\bigwedge_{j \leq i} \neg \big[{}^{q}O_{j}^{p} \wedge Loc({}^{q}O_{i}^{p}) = Loc({}^{q}O_{j}^{p})\big]\big)$$

This means that if a write is observed then no earlier write of that process to the same location may be observed in the future any more. As shown in Figure 4, slow consistency allows reordering of writes if they do not target the same location. In this case, the write to Y is visible before the second write to X was observed.

Write
$$(X = 1)$$

Write $(X = 2)$
Write $(Y = 1)$
Read $(X = 1)$
Read $(X = 1)$
Read $(X = 2)$

Fig. 4. TestSlow: Slow consistent execution which is not PRAM consistent.

PRAM Consistency. Pipelined random access memory (PRAM) consistency was first introduced in 1988 by [21]. PRAM requires each process to respect the order of the writes of other processes, but not their read operations. This means that two writes of the same process will always be observed in the same order by all other processes. Therefore, the PRAM specification extends slow consistency by an additional property:

$$\left[\mathsf{F}({}^{q}O_{i}^{p})\right] \to \bigwedge_{j \leq i} \left([\mathsf{F}W_{j}^{p}] \to \left[(\neg {}^{q}O_{i}^{p}) \ \overline{\mathsf{U}} \ {}^{q}O_{j}^{p}\right]\right)$$

That is, if a processor observes a write operation, then it has to observe all earlier writes of that processor beforehand. While the execution in Figure 5 is PRAM consistent, it is not cache consistent (see next paragraph) as cache consistency requires writes to the same location to be observed in the same order by all processors.

Write(X = 1)	<u>۴</u>	Write(X=2)
$\operatorname{Read}(X=2)$	¥-^^-¥	$\operatorname{Read}(X=1)$

Fig. 5. TestPRAM: PRAM consistent execution which is not CC consistent.

Cache Consistency. In 1989 [14] introduced weak consistency and claimed that it would be the weakest form of consistency. Later on, after shown not to be the weakest model (see Local or Slow consistency), it became known as Cache consistency (CC). Cache consistency is stronger than slow consistency and extends it by the following property:

$$\begin{split} & \left[\mathsf{F}\left({}^{q}O_{i}^{p} \wedge \left[\mathsf{F}\left({}^{q}O_{j}^{p} \wedge Loc({}^{q}O_{i}^{p}) = Loc({}^{q}O_{j}^{p})\right)\right]\right)\right] \\ & \rightarrow \bigwedge_{r} \left[\mathsf{F}({}^{r}O_{i}^{p} \wedge (\mathsf{F}{}^{r}O_{j}^{p}))\right] \end{split}$$

This implies that if a processor observes two writes to the same location, then all other processors observe them in the same order. In the case of Figure 6, the second processor does not observe all writes of the first processor in the order they were issued and therefore can not be PRAM consistent, but as the observations are not in conflict with the location-specific ordering of the first processor, the execution is still cache consistent.

Write
$$(X = 1)$$

Write $(X = 2)$
Write $(Y = 1)$
Write $(Y = 1)$

Fig. 6. TestCC: CC consistent execution which is not PRAM consistent.

Sequential Consistency. Sequential consistency was defined by [20] and defines a behavior that may occur if programs are executed on a single processor (core). It requires all processors to agree upon a single sequential total ordering of all write operations they observe. The first required property (Totality) can be expressed as:

$$\mathsf{G}\big[W_i^p \to \bigwedge_q \left(\mathsf{F}\,^q O_i^p\right)\big]$$

That means, whenever a write event occurs, then each processor has to observe that write operation some time in the future. The other property to ensure a unique sequential representation is as follows:

$$\left[\mathsf{F}({}^{q}O_{i}^{p}\wedge\mathsf{F}\,{}^{q}O_{j}^{p'})\right]\to \bigwedge_{q'}\left[\mathsf{F}({}^{q'}O_{i}^{p}\wedge\mathsf{F}\,{}^{q'}O_{j}^{p'})\right]$$

This implies that if one processor observes two writes in a specific order, then all other processors have to observe these two writes in the same order.

4 Experimental Results

The environment and specifications described in the previous section have been implemented in the SMV [9] input language. This allows us to utilize NuSMV [9] and NuXMV [8] for LTL verification either using BDD-based or SAT-based bounded model checking (BMC). For the processor implementation, the environment follows an assembler representation of the programs or test cases to analyze.

4.1 Property Verification

To verify a safety property, we abstract from the actual memory system implementation and only provide a memory specification to check the property against all possible behaviors.

Multithreaded programs often come along with parts of the code that requires mutually exclusive access to some variables to achieve the correct behavior, i.e., they contain a critical section. For example, an instance of Peterson's algorithm should never allow two processes to access the critical section at the same time. When implementing such programs as parallel modules in SMV and expressing the properties in LTL, we can verify them using LTL model checking.

This can be written as follows:

$$(Process modules) \models (Memory Spec) \rightarrow (Property)$$

Using tools like NuSMV or NuXMV, properties can be verified using BDD approaches or by searching for counterexamples using a SAT-based bounded model checking approach.

4.2 Examples and Results

In the following, several algorithms are described and analyzed for their minimal required memory models. We start with the well-known mutual exclusion algorithm due to Peterson, and then consider a simple consumer-producer algorithm.

First, Peterson's mutual exclusion algorithm (see Figure 7) will be shown to work as expected with sequential consistency, but fails with weaker consistency models.

bool flagP, flagQ; int turn, data;

```
thread P() {
                                     thread Q() {
  flagP = true;
                                        flagQ = true;
  turn = 0;
                                        turn = 1;
  while(flagQ & turn==0) {};
                                        while(flagP & turn==1) {};
  // Begin of Critical Section
                                        // Begin of Critical Section
  data = data + 1;
                                        data = data + 1;
  data = data - 1;
                                        data = data - 1;
                                        // End of Critical Section
   // End of Critical Section
                                        flagQ = false;
  flagP = false;
}
                                     }
```

Fig. 7. Peterson mutual exclusion protocol

Peterson's mutex algorithm works as follows: Whenever a process wants to enter the critical section, it sets its own flag. Then, it sets the turn variable to the id of the other process. Afterwards, the algorithm checks whether the other process indicated a critical section request with its flag, too. If so, depending on the state of the turn variable, it will either idle as long as the others process flag holds, or it proceeds to the critical section. After a process finished its critical operations, it resets its flag to signal the other process that it is safe to progress. In the example provided the critical section contains of an increment and an decrement of *data*. Assuming the mutual exclusive execution of the critical sections the value of *data* should always be either 0 or 1.

To examine the memory behavior of that algorithm, it has to be translated in a (pseudo) assembler representation which reveals the individual load-store instructions as shown in Figure 8. The representation uses memory locations already present in the high-level implementation (see Figure 7): flag[0], flag[1], turn, and data. Consider the non-atomic instructions data = data+1 and data = data - 1. Assuming the initial value of data is 0, then data should only alternate between 0 and 1 and after both processes are finished we expect it to be 0. But if the reading and writing part of the instructions are interleaved, data may have more intermediate values -1, 0, 1, 2 and either -1, 0,or -1 in the end. Therefore, the instructions can be used to model a critical section, as it adds unexpected behavior if the mutual exclusion is not ensured. In this case, we would like to verify that the location data is 0 or 1 all the time.

```
1 write(flag[id],1)
2 write(turn,1-id)
3 reg = read(flag[1-id])
4 if (reg=0) then goto 7
   reg = read(turn)
5
6
    if (reg=(1-id)) then goto 3
\overline{7}
    reg = read(data)
    write(data,reg+1)
8
9
    reg = read(data)
10
    write(data,reg-1)
11
    write(flag[id],0)
12
    goto 12
```

Fig. 8. Peterson's algorithm for two processes in an 'assembler' representation. Constant id is 0 for the first process and 1 for the other.

To achieve this verification, we model the instructions depicted in Figure 8 on two processor modules in the SVM input language. Then, we add an LTL specification which reads (*ModelSpecification*) \rightarrow (*G* (*data* = 0 \lor *data* = 1))). This means whenever a path of the state transition system satisfies the specification of the memory model, then we imply that it will never be the case that *data* is a value other than 0 or 1. Using NuSMV, we proved the safety property to be valid for sequential consistency, and to be invalid for the other models described in this paper: Local, Slow, CC, and PRAM consistency by providing counterexamples.

The second algorithm we analyze is a simple producer-consumer algorithm as seen in Figure 9. Producer P writes one data value and then waits until it has been read by consumer C. P signals the availability of data by setting variable *ready* to *true* and C signals that it read the value by setting *ready* to *false* again.

```
bool ready = false; int data = 0;
thread P() {
                                     thread C() {
  for(i=0..N-1) {
                                      int local[N]; bool done;
   data = i;
                                      for(i=0..N-1) {
   ready = true;
                                        while(!ready) {};
   while(ready) {};
                                        local[i] = data;
                                        ready = false;
  }
}
                                      }
                                      done = true;
                                     7
```

Fig. 9. Producer-Consumer Algorithm

Clearly, the depicted producer-consumer algorithm works as expected for sequential consistency: Whenever C observes *ready* to be *true*, the corresponding write to *data* is visible to C, too. Therefore, there is only a single possible outcome for the values of the *local* registers in a sequentially consistent environment. Analogously to the first algorithm, using a suitable low-level representation and defining a correctness property like (*ModelSpecification*) \rightarrow (G (*done* $\rightarrow \bigwedge_{i=0}^{N-1}(local[i] = i)$)), we were able to prove that the algorithm works as expected for sequential consistency. While we were able to find counterexamples for Local, Slow and CC consistency, we could prove that PRAM consistency is sufficient to ensure the specified correctness property. This shows that a more efficient, but weakly consistent memory system can be used for this algorithm without compromising the expected behavior.

Note that in these examples, it is quite easy to determine a suitable depth as both processes terminate after a predetermined number of steps. This approach can be used for repetitive programs as well, but suitable bounds have to be determined based on the number of steps required to cover all relevant behavior.

Using the described technique, we were able to verify other small litmus-testalike examples for different memory models, and to disprove them for weaker models. This way, we are able to determine the minimally required consistency models/guarantees to ensure that a given property holds. Verifying against multiple models is as easy as replacing the LTL specification. Neither the processor representation nor the property description have to be changed.

5 Conclusion and Future Work

In this paper, we introduced a novel approach to specify weak memory systems using temporal logic. Using temporal logic, we were able to describe the behavior of different memory consistency models, i.e., restricting the allowed read results in correspondence to the history of issued memory write operations. This itself is already a useful result, as it offers a new perspective and makes the topic more accessible for programmers already familiar with property specifications in LTL. Model checking can directly use our LTL specifications, so that we can use established tools to verify multithreaded programs. Moreover, the approach allows us to easily determine the weakest consistency requirements a program needs to satisfy a given property.

However, the approach suffers from the state explosion problem as weak consistency considers all possible write events and therefore has to quantify over time, processes, variables, and all possible values. Verification with NuSMV's BDD model checking of non-trivial examples like the Peterson mutual exclusion introduced in Figure 7 already require several GB of memory to finish. Using NuSMV's BMC with reasonable bounds allows us to inspect more examples, but inevitable will run out of memory for more complex examples, too.

In the future, we would like to provide specifications for more models, including an analysis how models like TSO act differently than the presented models. Furthermore, we are looking at finding better representations, especially for a possibility to get rid of the global write identifier. In this context, it may be possible to reuse already computed information like the reachable states for model checking when only the memory model is changed. To strengthen the confidence in the presented LTL specifications, we are interested in proving the equivalence between the LTL specification and reference implementations, and furthermore to prove the relationship between different models using our representations (e.g., that sequential consistency implies PRAM consistency).

References

- S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- S. Adve and M. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 4(6):613–624, June 1993.
- M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In L. Snyder, editor, *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 251–260, Velen, Germany, 1993. ACM.
- J. Alglave. A formal hierarchy of weak memory models. Formal Methods in System Design (FMSD), 41(2):178–210, October 2012.
- J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In P. Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *LNCS*, pages 41–44, Saarbrücken, Germany, 2011. Springer.
- C. Baier and J.-P. Katoen. Principles of Model Checking. MIT Press, Cambridge, Massachusetts, USA, 2008.
- G. Boudol and G. Petri. Relaxed memory models: an operational approach. In Z. Shao and B. Pierce, editors, *Principles of Programming Languages (POPL)*, pages 392–403, Savannah, Georgia, USA, 2009. ACM.
- R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 334–342, Vienna, Austria, 2014. Springer.
- A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification* (*CAV*), volume 1633 of *LNCS*, pages 495–499, Trento, Italy, 1999. Springer.
- 10. E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.
- E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier, 1990.
- F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory model-aware testing - a unified complexity analysis. In *Application of Concurrency to System Design (ACSD)*, pages 92–101, Tunis La Marsa, Tunisia, 2014. IEEE Computer Society.
- F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory-model-aware testing – a unified complexity analysis. *Transactions on Embedded Computing* Systems (TECS), 2015.

- M. Goodrich. Constructing arrangements optimally in parallel (preliminary version). In F. Leighton, editor, Symposium on Parallel Algorithms and Architectures (SPAA), pages 169–179, Hilton Head, South Carolina, USA, 1991. ACM.
- R. Grisenthwaite. Barrier litmus tests and cookbook. Technical Report PRD03-GENC-007826 1.0, ARM Limited, November 2009.
- O. Grumberg and H. Veith, editors. 25 Years of Model Checking History, Achievements, Perspectives, volume 5000 of LNCS. Springer, 2008.
- A. Heddaya and H. Sinha. 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.
- L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, University of Calgary, 1998.
- P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 302–309, Paris, France, 1990. IEEE Computer Society.
- L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (T-C)*, 28(9):690–691, September 1979.
- R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.
- 22. Z. Manna and A. Pnueli. The temporal Logic of Reactive and Concurrent Systems. Springer, 1992.
- 23. P. McKenney. Memory barriers: A hardware view for software hackers. http://www.rdrop.com/users/paulmck, June 2010.
- D. Mosberger. Memory consistency models. ACM SIGOPS: Operating Systems Review, 27(1):18–26, January 1993.
- 25. F. Nardelli, P. Sewell, J. Ševčík, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave. Relaxed memory models must be rigorous. In *Exploiting Concurrency Efficiently and Correctly (EC2)*, Snowbird, Utah, USA, 2011.
- 26. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5674 of *LNCS*, pages 391–407, Munich, Germany, 2009. Springer.
- M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In 25 Years of the International Symposia on Computer Architecture (ISCA), pages 284–290, Barcelona, Spain, 1998. ACM.
- A. Pnueli. The temporal semantics of concurrent programs. Theoretical Computer Science (TCS), 13(1):45–60, 1981.
- W. Pugh. The Java memory model is fatally flawed. Concurrency: Practice and Experience, 12(6):445-455, May 2000.
- 30. K. Schneider. Verification of Reactive Systems Formal Methods and Algorithms. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.
- M. Senftleben. Operational characterization of weak memory consistency models. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, March 2013. Master.
- 32. R. Steinke and G. Nutt. A unified theory of shared memory consistency. *Journal* of the ACM (JACM), 51(5):800–849, September 2004.
- D. Weaver and T. Germond, editors. The SPARC Architecture Manual-Version 9. Prentice-Hall, Inc., 1994.