Model Checking with User-Definable Memory Consistency Models

Tatsuya Abe and Toshiyuki Maeda

RIKEN AICS, Kobe, Hyogo, Japan
{abet, tosh}@riken.jp

Abstract

From the viewpoint of performance and scalability, relaxed memory consistency models are common and essential for parallel/distributed programming languages in which multiple processes are able to share a single global address space, such as Partitioned Global Address Space languages. However, a problem with relaxed memory consistency models is that programming is difficult and error-prone because they allow non-intuitive behaviors that do not occur in the ordinary sequential memory consistency model.

To address the problem, this paper presents a model checking framework in which users are able to define their own memory consistency models, and check programs under the defined models. The key point of our model checking framework is that we define a base model that allows very relaxed behavior, and allow users to define their memory consistency models as constraints on the base model.

1 Introduction

One of the problems of Partitioned Global Address Space (PGAS) languages from the viewpoint of ease of writing programs is that they are based on relaxed memory consistency models [3], like multicore CPUs [12] and conventional distributed shared memory systems [14]. A memory consistency model is a formal model that defines the behavior of shared memory accessed simultaneously by multiple processes. In a relaxed memory consistency model, the shared memory behaves differently from that in a sequential one. More specifically, the behavior defined by a relaxed memory consistency model may not match any possible behavior of a sequential process that simulates the behavior of multiple processes by executing their instructions in an arbitrary interleaved way.

To prevent non-intuitive behaviors of relaxed memory models, programmers must insert explicit synchronization operations in their programs; however, this is difficult and error-prone. Conservatively inserting synchronization operations severely degrades the performance of the program. However, it is difficult to reduce the number of synchronization operations correctly because even a slight lack of synchronization can introduce unpredictable and/or non-reproducible bugs.

One possible approach to address the above mentioned problem of relaxed memory consistency models is to apply model checking to verify programs. Model checking is a formal program verification approach that explores all the possible program states that can be reached during program execution. Software model checkers are available for parallel/distributed programs including for PGAS languages [10, 17, 20, 7, 9, 1, 2].

However, most of the existing studies of model checking parallel/distributed programs consider the sequential memory consistency model only. Therefore, they cannot be used to verify programs executing on shared memory with relaxed memory consistency models. Although several studies of program verification considering relaxed memory consistency models have been carried out (e.g., [11, 5, 4, 6]), most of these support only one or a small number of fixed theoretical models, and are thus not suitable for handling practical relaxed memory consistency models that may vary from language to language. Some of the works try to handle multiple relaxed memory consistency models uniformly.
Figure 1: Overview of our abstract machine

(e.g., [18, 22, 21, 16, 8]), but these are still limited to a few existing relaxed memory consistency models and it is not apparent how to adapt and support the other models not covered by these.

To address the problem, this paper presents a model checking framework in which users are able to define their own memory consistency models, and verify programs under the defined models. The key point of our approach is that we define and provide a base model that allows very relaxed behaviors, and allows users to define their memory consistency models as constraints on the base model.

The rest of this paper is organized as follows. Section 2 describes our base model on which memory consistency models can be defined. Section 3 explains how to define memory consistency models using example constraint rules representing rules of memory consistency models for Coarray Fortran. Section 4 briefly introduces the implementation of our model checker. Finally, Section 5 concludes the paper and discusses future work.

2 Base Model

In this section, we introduce our base model on which various memory consistency models can be defined. More specifically, in Section 2.1 we introduce an abstract machine for the model. Then, we describe the execution traces of the abstract machine in Section 2.2.

2.1 Abstract Machine

An overview of our abstract machine is shown in Fig. 1. The abstract machine consists of a fixed number of processes. Each process has its own state (that is, program instructions, local variables, and memory). Basically, execution of the instructions is closed within each process, and communication between processes occurs only when a process performs a store operation on its memory, and the update is broadcast to the other processes.

A more formal definition of the abstract machine is given in Figs. 2 and 3. Owing to space limitations, the semantics of the abstract machine is not given in this paper. The semantics is standard and straightforward, except that the instructions can be reordered and memory store instructions send requests for memory updates to other processes. The next section describes how to handle instruction reordering in our base model.
(State) \[ S ::= (P_1, \ldots, P_n) \]
(Process) \[ P ::= (I, V, M) \]
(Store) \[ V ::= \{x_1 \mapsto v_1, \ldots, x_i \mapsto v_i\} \]
(Memory) \[ M ::= \{l_1 \mapsto v_1, \ldots, l_j \mapsto v_j\} \]

where
(Variable) \( x \) (local variables)
(Location) \( l \) (addresses in memory)
(Label) \( L \) (labels in instructions)
(Value) \( v \) \( n \mid \ell \mid L \)

Figure 2: Definition of our abstract machine

(Insts.) \[ I ::= i_1; \ldots; i_n \]
(Raw Inst.) \[ i ::= \text{Move } x \leftarrow t \mid \text{Load } x \leftarrow [x] \]
\[ \mid \text{Store } [x] \leftarrow r \mid \text{Jump } t, i \mid t, i \mid \text{Nop} \]
(Term) \[ t ::= x \mid v \]
(Attrs.) \[ A ::= \{a_1, \ldots, a_n\} \]
(Attribute) \( a \) (user-defined attributes)

Figure 3: Definition of the instructions

2.2 Execution Traces

This section explains how to handle relaxed memory consistency models in our base model; that is, how to simulate the effect that memory accesses performed by one process can be observed in a different order by other processes. The key ideas are threefold. First, our model decomposes an instruction into the fetch and issue of the instruction. Additionally, the issue of a memory instruction is further decomposed into the issue itself and its corresponding memory operation. Specifically, the issue of a memory load instruction is decomposed into the issue itself and the memory access, which stores the obtained value in a variable local to its own process. On the other hand, the issue of a memory store instruction is decomposed into the issue itself and memory updates on each process. Second, each process in the abstract machine executes not only its own instructions, but also the other processes' instructions and memory operations. At first sight, this may appear redundant, but it is necessary to handle some relaxed memory consistency models that allow processes to observe inconsistent shared memory images. Third, our model considers all the possible permutations of instruction issues and memory operations performed by all the processes. In this paper, we call the permutations execution traces (or simply traces).

A more formal definition of execution traces is given in Fig.4. Trace \( \tau \) is defined as an ordered (finite or infinite) sequence of operations \( o \), where \( o \) is either an instruction fetch (denoted by the tag \text{Fetch}), an instruction issue (denoted by the tag \text{Issue}), or a memory operation (denoted by the tag \text{Rflct}).

(All Traces) \[ T_S ::= [\tau \mid S \vdash \tau] \]
(Trace) \[ \tau ::= a_1; \ldots; a_n; \ldots \]
(Operation) \[ o ::= \text{Fetch}_p p i \]
\[ \mid \text{Issue}_p^o p i \]
\[ \mid \text{Rflct}_p^o [\Rightarrow p] i \ell v \]

Figure 4: Definition of execution traces
Please note that the subscript $p'$ of the operation tags indicates that the operations are included in the execution of the abstract machine by process $p'$. (Recall that as mentioned above, each process executes not only its own instructions but also the instructions and memory operations of the other processes.) A more detailed explanation is omitted owing to space limitations.

3 Defining Consistency Models

In this section, we explain how to define memory consistency models on the base model described in Section 2 by showing an example rule that represents a barrier instruction of Coarray Fortran [15]. Informally speaking, memory consistency models are defined as constraints on execution traces, and the traces satisfying the constraints are considered to be valid under the memory consistency models.

Coarray Fortran has a barrier instruction called sync all, and the memory consistency model of Coarray Fortran [15] ensures that instructions that are fetched before (or after) the barrier instruction have to be completed before (or after) the barrier instruction completes. For example, the following rule preserves the order between the barrier instruction (sync all) and a store instruction:

$$\forall p, p', i, i'.
\begin{align*}
&\text{Fetch}_{p'} pi \downarrow \text{Fetch}_p pi' \supset \\
&\forall p''. \text{Issue}_{p'} p i \downarrow \text{Rflct}_{p'} [\Rightarrow p''] i' \ell v \\
&\text{and} \text{Fetch}_{p'} pi' \downarrow \text{Fetch}_p pi \supset \\
&\forall p''. \text{Rflct}_{p'} [\Rightarrow p''] i' \ell v \downarrow \text{Issue}_{p'} p i \\
&\text{if } i \equiv (\ldots, \text{A, Nop}). i' \equiv (\ldots, \text{Store }[x] \leftarrow t), \\
&\text{and } \text{sync-all} \in A
\end{align*}$$

In the above rule, $o_1 \downarrow o_2$ means that $o_1$ appears before $o_2$ in execution traces. In addition, $m$ and $m'$ represent the positions of $i$ and $i'$ in the trace, respectively, $\ell = V(x)$, and $v = V(t)$, where $V$ is the state of the local variables of process $P_p$ when instruction $i'$ is issued. Note that the attribute of instruction $i$ is used to indicate that $i$ is a barrier operation.

4 Implementation

Based on the approach explained in Sections 2 and 3, we implemented a prototype model checker. Basically, the model checker takes a user program and a memory consistency model as inputs, and generates a model in PROMELA, which is the model description language of the SPIN model checker [10]. Then, the generated model is passed to the SPIN model checker to perform model checking. Note that model checking is conducted in such a way that each process defined in the input program explores the generated model independently (refer to Section 2.2).

In fact, using the proposed model checker, we successfully checked several small example programs taken from the specification documents of UPC [19] and Itanium [13]. In addition, the model checker was used to analyze differences between three relaxed memory consistency models: UPC [19], Coarray Fortran [15], and Itanium [13].

5 Conclusion and Future Work

In this paper we described an approach to model checking parallel/distributed programs with shared memory (e.g., PGAS programs) under relaxed memory consistency models. In our approach, users are
able to define memory consistency models as constraint rules on our quite relaxed base model. Based on this approach, we implemented a prototype model checker with which we conducted several preliminary experiments.

Future work includes two directions. First, we intend on improving the algorithm for model checking by utilizing, for example, partial order reduction. Because the current implementation of our model checker is prone to the state explosion problem, it is necessary to further optimize the algorithm to enhance its practicality. Second, we aim to develop a formal system for our base model based on Kripke semantics to verify properties directly by theorem proving.

References