Optimization of a General Model Checking Framework for Various Memory Consistency Models

Tatsuya Abe
RIKEN AICS
Kobe, Hyogo, Japan
abet@riken.jp

Toshiyuki Maeda
RIKEN AICS
Kobe, Hyogo, Japan
tosh@riken.jp

ABSTRACT

While relaxed memory consistency models contribute optimizations of compilers on multicore CPUs and shared memory distributed programming languages, their relaxedness makes it difficult to write programs correctly. To address this problem, the authors proposed a general model checking framework and implemented a prototype tool McSPIN, which can take a memory consistent model as an input, as well as a program and a property to be checked, in their previous works. However, one big problem of McSPIN was that it was prone to suffer from the state explosion problem, and difficult to be applied to programs other than small example programs. In this paper, we propose optimization approaches for McSPIN to largely reduce the number of state transitions to be explored during model checking so that it can be applied to larger programs. In addition, we actually implemented the optimizations to McSPIN, and this paper gives several experimental results with the optimized McSPIN to show effectiveness of the proposed optimization approaches.

1. INTRODUCTION

A memory consistency model [9] is a formal specification of a behavior of memory shared among multiple processes/threads in multicore CPUs and shared-memory distributed programming languages. Relaxed memory consistency models allow the shared memory behaves differently from the sequential consistency model, that is, the results of simultaneous accesses to the shared memory by multiple processes can be different from any of the results obtained by executing them in an interleaving manner. The importance of relaxed memory consistency models increases as the number of cores in multicore CPUs and the number of nodes in shared memory distributed programming environments increase. This is because the overhead of ensuring the sequential consistency model becomes large (due to communication and synchronization costs among processes). Under the relaxed memory consistency models, compilers and/or runtimes are allowed to optimize programs aggressively (e.g., reorder instructions, delay memory operations, and so on).

However, one big problem of relaxed memory consistency models is that their relaxedness makes it more difficult to write correct programs. To see the difficulty, let us consider an example program shown in Fig. 1, so called *double checked locking* [28]. The intention of the program is to initialize a singleton object (line 4) share by multiple processes, and prevent the processes from initializing multiple objects concurrently. More specifically, the program first checks the variable `x` whether the singleton object is initialized or not (line 1). Please note that the check is performed without any synchronization with other processes in order to avoid the cost of synchronization. If the singleton object seems to be not initialized, the program acquires a synchronization lock (line 2), and checks again whether the singleton object is initialized by examining `x` (line 3), because it may be already initialized by the other processes. If the singleton object is not yet initialized at this point, it is finally initialized (line 4), and the acquired lock is released (line 6).

```
1: if (x == NULL) {
2:   lock();
3:   if (x == NULL) {
4:     x = new Singleton();
5:   }
6:   unlock();
7: }
```

Figure 1: Double checked locking

At first glance, it seems that the program of Fig. 1 correctly initialize the singleton object without any race condition. In fact, the program behaves correctly under the sequential consistency model. However, it may not work under some relaxed memory consistency models (e.g., Java programming language [35], Unified Parallel C (UPC) [40], Intel Itanium CPU architecture [31], and so on), because memory operations can be reordered unexpectedly from the viewpoint of the sequential consistency model (for the details, please refer to [15] and/or Sec. 4.1, for example).

In the previous works [5, 6], in order to address the difficulty in programming under relaxed memory consistency models, the authors developed a model checking framework McSPIN[1], which is able to support various memory consistency models. One advantage of McSPIN compared to conventional model checking approaches is that McSPIN is able to take a specification of a memory consistency model as an input, and performs model checking under the specified memory consistency model, while the conventional model checkers typically do not consider relaxed memory consistency models at all, or support only a few relaxed memory models but they are embedded into the core algorithms of model checking,

that is, it is not apparent how to apply them to different memory models. Actually, we defined the memory consistency models of UPC [40], Coarray Fortran [36], and Itanium [31] in McSPIN, and formally verified the differences among them [6].

This advantage contributes to design and implementation of shared-memory concurrent/distributed programming languages. Concretely, language designers and implementors are able to express their memory consistency models formally. One benefit of the formalization is that, for example, they are able to confirm that their memory consistency models are defined precisely as they intend by checking sample programs with McSPIN.

However, the original implementation of McSPIN has a big problem that it is prone to suffer from the state explosion problem, and only applicable to small example programs. This is because McSPIN uses a very relaxed memory consistency model as its base model in order to cover a wide range of memory consistency models. Because any program executions and memory operations can be freely reordered under the base model, the number of state transitions to be explored during model checking increases drastically.

In this paper, in order to address the state explosion problem of the original McSPIN, we show 4 optimization approaches that can reduce the number of state transitions to be explored. Although the optimization approaches described in this paper seem to be specific to the implementation of McSPIN, we believe that they can be applied to other model checking approaches that adopt program translations, that is, translate programs under relaxed memory consistency models into ones under sequential consistency, and use model checkers under sequential consistency not limited to SPIN.

This paper also presents experimental results with McSPIN. The experimental results show that the optimization approaches are actually effective, and McSPIN can be applied to larger programs compared to the previous implementation.

The rest of this paper is organized as follows. First, Sec. 2 briefly summarizes the previous works [5, 6], that is, the model checking framework which supports various memory consistency models and its implementation McSPIN. Next, Sec. 3 explains optimization approaches for McSPIN that can reduce the number of state transitions to be explored during model checking. Then, Sec. 4 shows the effectiveness of the optimization approaches by experimental results with McSPIN. Finally, Sec. 5 discusses related work, and Sec. 6 concludes the paper.

2. SUMMARY OF OUR ORIGINAL MODEL CHECKER

In this section, we summarize our previous works [5, 6] which describe our original model checker which is able to support various memory consistency models. Main contributions of this paper, that is, optimization approaches for the original model checker are described in the next sections (Sec. 3 and 4).

The key idea of the original model checker is as follows. First, we defined a very relaxed base model in which instruction executions and memory operations can be almost freely reordered so that various kinds of relaxed memory consistency models can be covered. Based on the base model, memory consistency models can be defined as constraint rules that restrict how instruction executions and memory operations are reordered. Our model checker checks whether the given program satisfies the given property by exploring all the states that can be reached during the execution of the given program in consideration of the reordering of instruction executions and memory operations and the given constraint rules that represent memory consistency models.

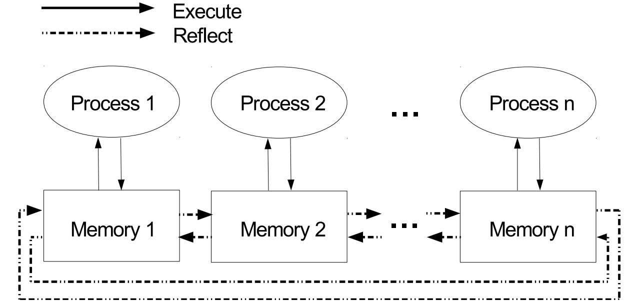

Figure 2: Overview of our abstract machine

(State)	$S ::= \langle\langle P_1, P_2, \ldots, P_m\rangle, O\rangle$
(Proc.)	$P ::= \langle R_1, R_2, \ldots, R_m\rangle$
(Raw Proc.)	$R ::= \langle\{x \mapsto v\}, \{\ell \mapsto v\}, L, I, j\rangle$
(Insts.)	$I ::= i_1; i_2; i_3; \ldots; i_k$
(Inst.)	$i ::= \langle L, A, \iota\rangle$
(Raw Inst.)	$\iota ::= \texttt{Move}\ x \leftarrow t \mid \texttt{Load}\ x \leftarrow [x'] \mid \texttt{Store}\ [x] \leftarrow x' \mid \texttt{Jump}\ L\ \texttt{if}\ x \mid \texttt{Nop}$
(Term)	$t ::= x \mid v \mid t + t' \mid t - t'$
(Attrs.)	$A ::= \{a, \ldots, a\}$
(Attribute)	a (user-defined attributes)

where

(Process Identifier)	p, q	(finite set of process identifier)
(Variable)	x	(finite set of local variables)
(Location)	ℓ	(finite set of addresses in memory)
(Label)	L	(finite set of labels in instructions)
(Value)	v	$::= n \mid \ell \mid L$
(Branch Counter)	j	(integers)

(Operations)	$O ::= \{o, \ldots, o\}$
(Operation)	$o ::= \texttt{Fetch}^j_q\ p\ i \mid \texttt{Issue}^j_q\ p\ i \mid \texttt{Execute}^j_q\ p\ i\ \ell\ v \mid \texttt{Reflect}^j_q\ [p \Rightarrow p']\ i\ \ell\ v$

Figure 3: Definition of our abstract machine

2.1 Base Model

Fig. 2 shows a very brief overview of our base model. The base model consists of multiple processes and each process has its own memory. The memories associated with processes are isolated each other, that is, one process cannot access the memories of the other processes. Each process manipulates its own memory by executing memory access instructions (Execute in Fig. 2), and the effects of the memory manipulation are reflected (propagated) to the other memories (Reflect in Fig. 2). In the base model, Executes and Reflects are freely reordered except for several straightforward constraints (e.g., Execute of a memory access instruction never occurs before the instruction is issued, Reflect of an instruction execution never occurs before its corresponding instruction is executed, and so on).

A more formal definition of the abstract machine of our base model is given in Fig. 3. The state of the abstract machine S consists of a fixed number of processes P, and a set of operations O. O denotes actions that can be taken by the abstract machine. Its details are

explained later in the end of this section. Each process P_q consists of a set of raw processes R (which represent all the processes in the abstract machine), and each raw process R consists of a state of variables (a map from local variables x to values v, where v denotes an integer value n, a memory address ℓ, or an instruction label L), a state of a memory (a map from addresses ℓ to values v), a label L (which represents the program counter), instructions I, and a branch counter j (which represents the number of executed branch instructions).

The reason why each process holds raw processes for all the processes is that several relaxed memory consistency models allow one process to speculate the behaviors of the other processes, and do not require the behaviors of a process speculated by different processes to be consistent in some cases.

I is an ordered list of instructions i, which is a tuple of a label L, attributes A, and a raw instruction ι. ι consists of local variable access instructions, memory access instructions, branch instructions, and nop. More specifically, `Move` $x \leftarrow t$ stores the evaluated value of term t to variable x. `Load` $x_d \leftarrow [x_s]$ loads the value stored at the address represented by x_s, and stores it to local variable x_d. `Store` $[x_d]{\leftarrow}x_s$ stores the value of variable x_s to the address represented by x_d. `Jump` L `if` x branches to the instruction labeled L, if the condition value represented by x is not equal to the integer value 0. `Nop` does nothing. The reason why we introduced `Nop` to the abstract machine is that it is useful to represent special instructions that are specific to individual memory consistency models by utilizing the attributes A, where A represents a fixed number of user-defined attributes a. Details of how to use the attributes to describe memory consistency model-specific instructions are explained in Sec. 2.3.

As slightly mentioned above, O represents a set of operations o that can be performed by the abstract machine. Basically, the abstract machine non-deterministically takes one operation (action) from O, and performs it.

More specifically, o consists of four operations. $\texttt{Fetch}_q^j\ p\ i$ represents a fetch of an instruction i on process P_p observed (or speculated) by P_q, where j represents how many times branch instructions are executed. When the program counter L points to an instruction in I, its corresponding fetch operation is added to O of the state of the abstract machine. Please note that adding a fetch operation to O does not mean that its corresponding instruction is actually fetched. Rather, it indicates that the instruction is ready to be fetched. If the instruction is actually fetched, the fetch operation is removed from O.

$\texttt{Issue}_q^j\ p\ i$ represents an issue of the fetched instruction i on process P_p observed by P_q, where j has the same meaning as in `Fetch`. `Issue` is added to O when its corresponding instruction is fetched (that is, `Fetch` is removed from O), and removed from O if the instruction is actually issued.

$\texttt{Execute}_q^j\ p\ i\ \ell\ v$ represents an execution of a memory access instruction which is processed by P_p and observed by P_q, where j has the same meaning as in `Fetch`, ℓ represents the target memory address, and v denotes the operand value. `Execute` is added to O when its corresponding memory access instruction is issued (that is, `Issue` is removed from O), and removed from O if the instruction is actually executed. Remark that states (variables and locations) of P_p are updated when the issued instruction is executed, but the effect cannot be seen from the other processes.

$\texttt{Reflect}_q^j\ [p \Rightarrow p']\ i\ \ell\ v$ represents a reflection (propagation) of the effect of an execution of memory store instruction on process P_p to the other process P'_p (which is observed by P_q), where j has the same meaning as in `Fetch`, ℓ represents the target memory address, and v denotes the operand value to be stored. `Reflect` is added to O when its corresponding memory store instruction is executed (that is, `Execute` is removed from O), and removed from O if the effect of the instruction is reflected on the other process. One thing different from `Fetch`, `Issue`, and `Execute` is that multiple `Reflect`s for all the processes are added to O. This is because it is necessary to handle the reflections to each process separately.

2.2 Execution Traces

In addition to the base model defined in the previous section, we also define *execution traces* which represent the behavior (state transition) of the abstract machine. Informally, an execution trace represents the order of actions during the state transition of an abstract machine, that is, the order in which operations are taken out from O in the abstract machine state S (see Fig. 3). More specifically, an execution trace (denoted as τ) is defined as an ordered (finite or infinite) sequence of operations o (defined in Fig. 3) as follows:

$$\text{(Trace)} \quad \tau ::= o_1;\ \ldots;\ o_n;\ \ldots$$

In addition, we denote all the admissible traces under all the state transitions (as slightly mentioned in Sec. 2.1) from the state S as T_S.

2.3 Memory Consistency Model

Under the base model and the execution traces, memory consistency models can be defined as constraints which restrict the order among operations, that is, narrows the range of the admissible traces (described in the previous section). For example, the sequential consistency model [9] can be represented as constraint rules that allow no reordering among operations.

More formally, we define the equational first-order predicate logic with formulae φ which consists of a binary predicate ($\leq$) and logical connectives $\neg$ (negation), $\supset$ (implication), and $\forall$ (universal quantifier). Intuitively, $o \leq o'$ denotes a constraint which ensures that o is performed before o' is performed (or o is equal to o'). It can be considered to be an extension of the so-called *happens-before relation* [33]. In addition, we define $\tau \vDash \varphi$ if the trace τ satisfies the formula φ.

With the equational first-order predicate logic, memory consistency models are defined as a set of axioms (Φ) of the logic. For example, the following axiom (formula) ensures that the effect of an executed store instruction on one process is immediately reflected to the other processes:

$$\texttt{Execute}_q^{j_1}\ p_1\ i_1\ \ell\ v < o_2 \supset \texttt{Reflect}_q^{j_1}\ [p_1 \Rightarrow p_3]\ i_1\ \ell\ v < o_2$$

where $i_1 \equiv \texttt{Store}\ [x] \leftarrow x'$ and $o_2 \equiv \texttt{Issue}_q^{j_2}\ p_2\ i_2$, which means that any o_2 must waits reflections of an executed instruction i_1 if it advances o_2 (universal quantifiers are omitted and $o < o'$ denotes $o \leq o'$ where $o \neq o'$). In our previous work [6], we confirmed that the equational first-order predicate logic can express all the rules of Sec. 3 of the Itanium specification manual [30] and Appendix A of the UPC specification manual [40] can be expressed. The concrete definitions of the Itanium and UPC memory consistency models for McSPIN are also bundled in the distribution of McSPIN [1]. Their details will be explained in our forthcoming working paper [4].

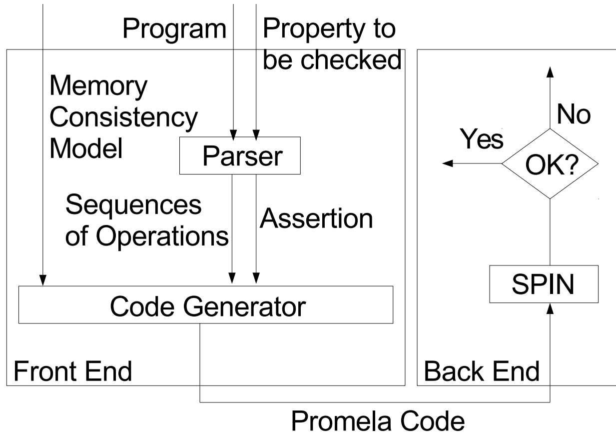

Figure 4: An outline of McSPIN

In addition, we also define all the admissible traces of the abstract machine state S under a memory consistency model Φ (denoted as $T_s(\Phi)$) as follows:

$$T_S(\Phi) \equiv \{\tau \mid \tau \in T_S \text{ and } \forall\varphi \in \Phi.\, \tau \vDash \varphi\}$$

2.4 Model Checking Framework

Based on the notions of the abstract machine, the execution traces, and the representations of memory consistency models (explained in the previous sections), model checking which checks whether a given program (S) satisfies a given property under a given memory consistency model (Φ) can be implemented by exploring all the state transitions derived from $T_S(\Phi)$ and verifying that all of them satisfy the given property. In this section, we briefly describe the implementation of our model checking framework [5, 6], named McSPIN.

Fig. 4 shows the overview of of our model checking framework. As shown in the figure, McSPIN consists of a front-end and back-end. The back-end utilizes the SPIN model checker [29] for actual model checking, that is, exploring all the state transitions. The front-end takes a program and a memory consistency model as inputs, and translates them into PROMELA code, which is the input format of the SPIN model checker.

One big difference from conventional model checkers is that McSPIN takes a memory consistency model as an input, while in the conventional checkers specific memory consistency models are embedded into them and it is necessary to re-design their model checking theories and re-implement them in order to switch one memory consistency model to another. On the other hand, because McSPIN is able to take a memory consistency model as an input, we can easily perform model checking under various memory consistency models without worrying the re-design and re-implementation.

As the input format of the front-end of McSPIN, we designed a simple modelling language equipped with variables/attributes declarations, and assignment/conditional statements. In addition, the language also has the notion of synchronization primitives (e.g., mutex locks, barriers, fences, etc.). In this paper, we do not elaborate the formal definition of our modelling language because it is simple and straightforward. Instead, we explain how to translate the input language to PROMELA code in the rest of this section.

The basic idea of the translation from our language to PROMELA is to enumerate all the possible operations from the instructions of processes, and generate the *do* statement (a repetition with non-deterministic choice) of PROMELA whose choice clauses correspond to the enumerated operations. For example, let us consider the following two processes P_1 and P_2:

$$P_1 : \texttt{Load}\ r_1 \leftarrow [x] \qquad P_2 : \texttt{Store}\ [x] \leftarrow r_2$$

The front-end of McSPIN enumerates the following 16 operations that are possible during the execution of the two processes:

$$\begin{array}{lll}
\texttt{Fetch}^1_{P_1}\ P_1\ i_1 & \texttt{Issue}^1_{P_1}\ P_1\ i_1 & \texttt{Execute}^1_{P_1}\ P_1\ i_1\ \ell\ v_1 \\
\texttt{Fetch}^1_{P_1}\ P_2\ i_2 & \texttt{Issue}^1_{P_1}\ P_2\ i_2 & \texttt{Execute}^1_{P_1}\ P_2\ i_2\ \ell\ v'_1 \\
 & & \texttt{Reflect}^1_{P_1}\ [P_2 \Rightarrow P_1]\ i_2\ \ell\ v'_1 \\
 & & \texttt{Reflect}^1_{P_1}\ [P_2 \Rightarrow P_2]\ i_2\ \ell\ v'_1 \\
\texttt{Fetch}^1_{P_2}\ P_1\ i_1 & \texttt{Issue}^1_{P_2}\ P_1\ i_1 & \texttt{Execute}^1_{P_2}\ P_1\ i_1\ \ell\ v_2 \\
\texttt{Fetch}^1_{P_2}\ P_2\ i_2 & \texttt{Issue}^1_{P_2}\ P_2\ i_2 & \texttt{Execute}^1_{P_2}\ P_2\ i_2\ \ell\ v'_2 \\
 & & \texttt{Reflect}^1_{P_2}\ [P_2 \Rightarrow P_1]\ i_2\ \ell\ v'_2 \\
 & & \texttt{Reflect}^1_{P_2}\ [P_2 \Rightarrow P_2]\ i_2\ \ell\ v'_2
\end{array}$$

where $i_1 \equiv \texttt{Load}\ r_1 \leftarrow [x]$, $i_2 \equiv \texttt{Store}\ [x] \leftarrow r_2$, ℓ is a location of x, v_1 and v_2 are the values stored in ℓ observed by P_1 and P_2, respectively, and v'_1 and v'_2 are the values of r_2 observed by P_1 and P_2, respectively. Please recall that, in our base model, each process can have its own view of the abstract machine represented by the subscripts of the operations, as described in Sec. 2.1.

With the enumerated operations $o_1, o_2, \ldots$, the front-end generates the do statement of PROMELA as follows:

```
do
 :: (translation of o1);
 :: (translation of o2);
 :: ...
od;
```

Roughly speaking, the do statement of PROMELA chooses and executes one of clauses (begin with '::') non-deterministically, and repeats the choice and execution (until the *break* statement is executed).

More specifically, an operation o is translated to a clause of the do statement as follows:

```
:: end_o == 0 -> o; clock++; end_o = clock;
```

where `clock` is a variable which represents a global time counter, and `end_o` is a variable which holds the value of the time counter when o is performed. Both the variables are initialized to `0`. Therefore, the guard condition of the clause (the left-hand side of '->') `end_o == 0` indicates that o has not been executed yet. Thus, the clause is only executed once. The reason why we introduce the global time counter is that using the counter is a straightforward to record orders in which operations are performed.

For example, $\texttt{Execute}^1_{P_2} P_1\, i_1\, l v_1$ (one of the operations enumerated above) is translated as follows:

```
:: end_execute_on_1_by_2_L1 == 0 ->
   r1_on_1_by_2 = memory_on_1_by_2[x_on_1_by_2];
   clock++; end_execute_on_1_by_2_L1 = clock;
```

where $_L_1$ denotes the label of i_1, and `_on_1` and `_by_2` represent that P_2 observes that the operation is performed by P_1. In addition, `memory_on_1_by_2` denotes the memory of P_1 observed by P_2.

The explanation so far does not take account of loops. In order to handle loops, McSPIN uses a bounded model checking approach [18, 17]. More specifically, instead of introducing the scalar variable end_o for each operation o, we use a fixed length array end_o_[LEN] whose each element represents o with a different branch counter j. Please note that j denotes how many times branch instructions are executed (as explained in Sec. 2.1). That is, LEN denotes the bound of the bounded model checking. For example, $\mathtt{Execute}^{j}_{P_2}\ P_1\ i_1\ l\ v_1$ is translated as follows:

```
:: end_execute_on_1_by_2_L1_[j-1] == 0 ->
    r1_on_1_by_2 = memory_on_1_by_2[x_on_1_by_2];
    clock++; end_execute_on_1_by_2_L1_[j-1] = clock;
```

The front-end of McSPIN then generates the PROMELA code which launches PROMELA processes that contain the above-mentioned do statements for each pair of processes as follows:

```
active main () {
    ...
    run process_on_1_by_1();
    run process_on_2_by_1();
    ...
    run process_on_n_by_1();

    ...

    run process_on_1_by_n();
    run process_on_2_by_n();
    ...
    run process_on_n_by_n();
    ...
}
```

where the PROMELA process process_on_p_by_q represents the execution of instructions on the process P_p which is observed by the process P_q.

During the execution of the launched PROMELA processes, the execution traces (explained in Sec. 2.2) are recorded as the introduced variables end_o. Please note that the variables end_o hold the information when the operation o is performed. Therefore, the execution trace can be obtained by sorting them.

Finally, the front-end of McSPIN generates an assertion statement of PROMELA which checks whether the obtained execution trace is admissible under the base model and the specified memory consistency model, and satisfies the specified property.

3. OPTIMIZATIONS

One big problem of the model checking approach described in the previous section is that it performs unnecessary exploration of state transitions, thus it easily suffers from the state explosion problem. For example, as described in the last paragraph of the previous section, apparently inadmissible execution traces (e.g., traces in which Executes performed before their corresponding Issues are performed) are examined in the unnecessary precise way.

In this section, we introduce four optimization approaches which are able to dramatically reduce the number of the state transitions during model checking. The experimental results of measuring the effects of the optimization approaches are shown in the next section (Sec. 4).

3.1 Enhancing Guards

Let us consider a formula $o_1 < o_2$ that represents a constraint rule which denotes that o_1 has to be performed before o_2 (e.g., o_1 is Fetch and o_2 is Issue). The original algorithm of translation (described in Sec. 2.4) generates the following PROMELA code:

```
:: end_o1 == 0 -> o1; clock++; end_o1 = clock;
:: end_o2 == 0 -> o2; clock++; end_o2 = clock;
```

However, the above PROMELA code is apparently inefficient because the code allows o_2 to be performed before o_1, which is inadmissible under the condition $o_1 < o_2$.

In order to reduce the number of state transition to be examined, the front-end of McSPIN modifies the guard condition of o_2 as follows:

```
:: end_o1 == 0 -> o1; clock++; end_o1 = clock;
:: end_o1 > 0 && end_o2 == 0 ->
                    o2; clock++; end_o2 = clock;
```

That is, o_2 will not be chosen until o_1 is performed. This enhancement of the guard condition can reduce the number of state transitions to half.

Furthermore, a more general formula $o_1 < o_2 \supset o_3 < o_4$ can be represented as the following code:

```
...
:: (!(end_o1 < end_o2) || end_o3 > 0) && end_o4 == 0
    -> o4; clock++; end_o4=clock;
...
```

Thus, this optimization excludes apparently impossible traces by examining the specified memory consistency model. More specifically, McSPIN takes and follows a memory consistency model, and generates a PROMELA code that does not contain such traces.

3.2 Disabling Speculation

While some memory consistency models allow processes to speculate behaviors of the others (e.g., UPC [40]), there also exist a large number of memory consistency models that do not allow the speculation. Under the non-speculative memory consistency models, it is unnecessary to distinguish who observes execution of the abstract machine. For example, if the abstract machine consists of two processes, the original algorithm (described in Sec. 2.4) generates the following PROMELA code:

```
active main () {
    ...
    run process_on_1_by_1();
    run process_on_2_by_1();
    run process_on_1_by_2();
    run process_on_2_by_2();
    ...
}
```

However, under the non-speculative memory consistency models, the behaviors of process_on_1_by_1 and process_on_1_by_2 are non-distinguishable. Therefore, it is suffice to generate the following code:

```
active main () {
    ...
    run process_on_1_by_1();
    run process_on_2_by_1();
    ...
}
```

3.3 Prefetching Instructions

Generally speaking, instructions to be fetched depend on the results of branch instructions (Jump), and the results vary depending on memory consistency models. Therefore, the original algorithm (explained in Sec. 2.4) handles instruction fetches (Fetch) non-deterministically as follows:

```
do
 :: (Fetch^1_{P_1} P_1 i_1);
 :: (Fetch^1_{P_1} P_1 i_2);
 :: ...
od;
```

However, if the instructions to be fetched are known statically (e.g., if there are no branch instructions in the program), their Fetch can be performed in any order without loss of generality. Thus, it is suffice to generate the following code (instructions are fetched in program order before any other operations):

```
(Fetch^1_{P_1} P_1 i_1);
(Fetch^1_{P_1} P_1 i_2);

do
 :: ...
od;
```

3.4 Removing Global Time Counter

Although the original algorithm of Sec. 2.4 handles the value of the global time counter explicitly, but it is sometimes unnecessary to handle the exact value. Instead, it is suffice to keep track of the order in which operations are performed. Forgetting the exact value of the time counter and keeping track of only the order among operations can reduce the number of state transitions to be explored, depending on constraint rules of memory consistency models.

For example, let us consider a constraint rule $o_1 < o_2 \supset o_3 < o_4$. When considering the constraint rule, it is suffice to keep track of the order of o_1 and o_2, and that of o_3 and o_4. More specifically, we can remove the counter variable clock by introducing two new variables ord_o_1_o_2 and ord_o_3_o_4 which record the orders among the operations, as in the following code:

```
do
:: end_o1 == 0 -> o1; end_o1 = 1;
:: end_o2 == 0 -> o2; end_o2 = 1; ord_o1_o2 = end_o1;
:: end_o3 == 0 -> o3; end_o3 = 1;
:: (!(ord_o1_o2 > 0) || end_o3 > 0) && end_o4 == 0
     -> o4; end_o4 = 1; ord_o3_o4 = end_o3;
od;
```

Please note that, in the above code, the variables end_o_i no longer hold the values of the time counter when the operations o_i are performed. Instead, they only hold the information whether the operations are performed or not. In addition, the variable ord_o_1_o_2 (ord_o_3_o_4) holds the information whether o_1 (o_3) is performed before o_2 (o_4) or not.

After all the operations are performed (that is, $\forall i.$end_$o_i = 1$), the number of possible states is 3 ({ord_o_1_$o_2 = 0$, ord_o_3_$o_4 = 0$}, {ord_o_1_$o_2 = 0$, ord_o_3_$o_4 = 1$}, and {ord_o_1_$o_2 = 1$, ord_o_3_$o_4 = 1$}), while the number is 24 (= 4!) (the factorial of the number of operations) in the original algorithm of Sec. 2.4.

4. EXPERIMENTS

Process P_1	Process P_2
1: Load $r_1 \leftarrow [x]$	1: Load $r_1 \leftarrow [x]$
2: Jump 10 if r_1	2: Jump 10 if r_1
3: Nop : *lock*	3: Nop : *lock*
4: Load $r_2 \leftarrow [x]$	4: Load $r_2 \leftarrow [x]$
5: Jump 9 if r_2	5: Jump 9 if r_2
6: Move $r_3 \leftarrow 1$	6: Move $r_3 \leftarrow 1$
7: Store $[c] \leftarrow r_3$	7: Store $[c] \leftarrow r_3$
8: Store $[x] \leftarrow r_3$	8: Store $[x] \leftarrow r_3$
9: Nop : *unlock*	9: Nop : *unlock*
10: Load $r_4 \leftarrow [x]$	
11: Load $r_5 \leftarrow [c]$	

Figure 5: Double checked locking in our modelling language

In order to show the effectiveness of the optimization approaches explained in Sec. 3, this section describes three experimental results with our model checker McSPIN. More specifically, Sec. 4.1 shows the result of model checking the double checked locking algorithm shown in Sec. 1, Sec. 4.2 shows the result of model checking Dekker's mutual exclusion algorithm, and Sec. 4.3 measures the effects of optimization approaches by model checking small example programs taken from the specification documents of Itanium [31] and UPC [40].

The experimental environment is as follows:

CPU	Intel Xeon E5-2670 2.6GHz 8cores × 4
Memory	DDR3-1066 1.5TB
SPIN version	6.3.2
GCC version	4.4.6

4.1 Double Checked Locking Algorithm

Fig. 5 shows the double checked locking algorithm in our modelling language. Please note that, compared to Fig. 1, if statements are replaced by branch instructions (Jump), and the instantiation of an object (new Singleton()) is represented by the store instruction to the variable c. In addition, please also note that lock() and unlock() are represented as Nops with user-defined attributes lock and unlock. Each memory consistency model defines the semantics of the attributes lock and unlock in our experiments, but we do not elaborate the semantics in this paper (please refer to our previous paper [6] for details).

In the experiment, we checked whether the property $r_4 = 1 \supset r_5 = 1$ holds after the two processes finished under various memory consistency models with McSPIN. Under the sequential consistency model, we confirmed that the code of Fig. 5 satisfies the property. McSPIN took 0.113 seconds and used 2.530 MB memory to examine 3638 state transitions exhaustively, when all the optimization approaches are enabled.

On the other hand, under relaxed memory consistency models, the property may not hold. For example, under Itanium memory model [31], McSPIN found a counterexample by examining 2000 state transitions in 0.096 seconds with 903 KB memory usage with all the optimizations enabled.

Roughly speaking, the obtained counterexample is as follows. First, let us suppose that the process P_2 holds the lock (line 3) and stores the value 1 to the variables c and x (lines 7 and 8). Under Itanium

```
Process P1                Process P2
x=1;                      y=1;
while (y==1) {            while (x==1) {
  if (t==1) {               if (t==0) {
    x=0;                      y=0;
    while (t==1) {}           while (t==0) {}
    x=1;                      y=1;
  }                         }
}                         }
z++;                      z++;
fence;                    fence;
t=1;                      t=0;
x=0;                      y=0;
```

Figure 6: Dekker's algorithm

Process P_1	Process P_2
1: Move $r_1 \leftarrow 1$	1: Move $r_1 \leftarrow 1$
2: Store $[x] \leftarrow r_1$	2: Store $[y] \leftarrow r_1$
3: Load $r_2 \leftarrow [y]$	3: Load $r_2 \leftarrow [x]$
4: Move $r_2 \leftarrow 1 - r_2$	4: Move $r_2 \leftarrow 1 - r_2$
5: Jump 17 if r_2	5: Jump 17 if r_2
6: Load $r_3 \leftarrow [t]$	6: Load $r_3 \leftarrow [t]$
7: Move $r_3 \leftarrow 1 - r_3$	7: Jump 15 if r_3
8: Jump 15 if r_3	8: Move $r_4 \leftarrow 0$
9: Move $r_4 \leftarrow 0$	9: Store $[y] \leftarrow r_4$
10: Store $[x] \leftarrow r_4$	10: Load $r_5 \leftarrow [t]$
11: Load $r_5 \leftarrow [t]$	11: Move $r_5 \leftarrow 1 - r_5$
12: Jump 11 if r_5	12: Jump 10 if r_5
13: Move $r_6 \leftarrow 1$	13: Move $r_6 \leftarrow 1$
14: Store $[x] \leftarrow r_6$	14: Store $[y] \leftarrow r_6$
15: Move $r_7 \leftarrow 1$	15: Move $r_7 \leftarrow 1$
16: Jump 3 if r_7	16: Jump 3 if r_7
17: Load $r_8 \leftarrow [z]$	17: Load $r_8 \leftarrow [z]$
18: Move $r_8 \leftarrow r_8 + 1$	18: Move $r_8 \leftarrow r_8 + 1$
19: Store $[z] \leftarrow r_8$	19: Store $[z] \leftarrow r_8$
20: Nop:fence	20: Nop:fence
21: Move $r_9 \leftarrow 1$	21: Move $r_9 \leftarrow 0$
22: Store $[t] \leftarrow r_9$	22: Store $[t] \leftarrow r_9$
23: Move $r_{10} \leftarrow 0$	23: Move $r_{10} \leftarrow 0$
24: Store $[x] \leftarrow r_{10}$	24: Store $[y] \leftarrow r_{10}$

Figure 7: Dekker's algorithm in our modelling language

memory model,

$$\texttt{Reflect}^1_{P_1}\ [P_2 \Rightarrow P_1]\,(7, \emptyset, \texttt{Store}\ [c] \leftarrow r_3)\ \ell_1\ 1 <$$
$$\texttt{Reflect}^1_{P_1}\ [P_2 \Rightarrow P_1]\,(8, \emptyset, \texttt{Store}\ [x] \leftarrow r_3)\ \ell_2\ 1$$

is not ensured where ℓ_1 and ℓ_2 are locations of c and x, respectively. That is, a reflection of Store $[x] \leftarrow r_3$ to P_1 can be reordered with that of Store $[c] \leftarrow r_3$. Now, let us suppose that the process P_1 observes the reflection of Store $[x] \leftarrow r_3$, that is, $x = 1$, at line 1. Then, the process P_1 jumps to line 10, and tries to load values from x and c (lines 10 and 11). At this point, because Itanium memory model allows that the effect of Store $[c] \leftarrow r_3$ of P_2 is not reflected to P_1 as described above, Load $r_5 \leftarrow [c]$ of P_1 at line 11 can observe $c = 0$. Thus, the property $r_4 = 1 \supset r_5 = 1$ does not hold.

4.2 Dekker's Algorithm

Fig. 6 shows Dekker's mutual exclusion algorithm in pseudo code. The intention of the algorithm is to prevent the two processes P_1 and P_2 simultaneously access the variable z, that is, avoid race condition on z. The reason why we chose Dekker's algorithm as the target of our experiment is that it consists of a slightly large number of instructions and contains (nested) loops.

Table 1: Results of (bounded) model checking Dekker's algorithm under the sequential consistency model

bound of iterations	# of state transitions	memory (MB)	time (sec.)
1	37862	71.037	1.044
2	76960	423.094	5.222
3	131334	1459.484	17.468
4	204008	3828.921	45.421

In our modelling language, the pseudo code of Fig. 6 can be translated to the instructions shown in Fig. 7. Please note that fence in the pseudo code (which ensures that any non-reflected memory effects are reflected to the other processes) is represented by Nop with a user-defined attribute fence. In the same way as the attributes lock and unlock of Sec. 4.1, each memory consistency model defines the semantics of the attribute fence in our experiments, but we do not elaborate it in this paper.

In the experiment, we checked whether $z = 2$ holds after the two processes finishes under various memory consistency models with McSPIN. Under the sequential consistency model, we confirmed that z = 2 holds on all the execution traces when increasing the bound of bounded model checking (that is, the number of iterations when handling loops) up to 4. More specifically, McSPIN took about 45.4 seconds and used about 3800 MB memory when the bound is set to 4. Tab. 1 shows more detailed results.

On the other hand, under relaxed memory consistency models, $z = 2$ may not hold. For example, under Itanium memory model [31], McSPIN found a counterexample by examining 9301219 state transitions in 111.279 sec. with 3889.837 MB.

Roughly speaking, the obtained counterexample is as follows. First, let us suppose that that P_1 issues Store $[x] \leftarrow r_1$ and P_2 issues Store $[y] \leftarrow r_1$ at line 2. Because the effects of the two Stores may not be reflected to the other processes under Itanium memory model, both the processes can branch to line 17 simultaneously, that is, the two processes can observe $z = 0$ when issuing Load $r_8 \leftarrow [z]$ at line 17. Thus, because it is possible that $z = 1$ after the two processes finish, the property $z = 2$ may not hold under Itanium memory model.

4.3 Measuring Effects of Optimizations

In this section, we measure how the optimization approaches (described in Sec. 3) improve the performance of our model checking. In the experiments, we used the same sample programs as our previous work [6]. More specifically, they are taken from the specification documents of Itanium [31] and UPC [40].

Table 2 shows a comparison between our previous work [6] and the current work. The first column lists the names of the sample programs, where itanium(n) and upc(n) represent the programs shown in Sec. n of the specification documents of Itanium [31] and UPC [40]. The second column lists the number of processes in the programs, the third column lists the total number of instructions of all the processes. The fourth column lists the types of property checking. More specifically, V.C.means that we need to verify va-

Table 2: Comparison with our previous work

	program			Our previous work [6]			All the possible optimizations applied		
				# of state	memory	time	# of state	memory	time
	#p	#i	type	transitions	(MB)	(sec.)	transitions	(MB)	(sec.)
itanium(A.1.4)	2	4	C.E.	1484	2.658	1.654	580	0.114	0.025
itanium(A.2.5)	2	4	V.C.	31215	12.200	1.688	216	0.062	0.019
itanium(A.3.6)	2	4	C.E.	55	2.219	1.642	635	0.118	0.026
itanium(A.4.7)	2	6	V.C.	168840	60.675	1.993	271	0.081	0.009
itanium(A.5.8)	2	5	V.C.	275298	100.925	2.128	185	0.061	0.012
itanium(A.5.9)	2	4	C.E.	31215	12.419	1.690	192	0.053	0.023
itanium(A.6.10)	2	6	C.E.	37346	16.696	1.977	293	0.088	0.018
itanium(A.7.11)	2	8	V.C.	2871250	1197.534	7.010	365	0.123	0.027
itanium(A.8.12)	4	6	V.C.	—	—	—	106627	21.076	0.591
itanium(A.9.13)	3	6	V.C.	5.889462e+08	327968.442	1839.239	2983	0.803	0.033
upc(B.5.1)	2	4	C.E.	5960	4.084	1.588	171	0.064	0.021
upc(B.5.2)	2	4	V.C.	1801	2.768	1.688	461	0.180	0.019
upc(B.5.3)	2	4	C.E.	414	2.329	1.566	177	0.065	0.019
upc(B.5.4)	2	2	C.E.	458	2.329	1.565	671	0.233	0.024
upc(B.5.5)	2	4	V.C.	1944	2.877	1.705	449	0.176	0.019
upc(B.5.6)	2	5	C.E.	162	2.219	1.700	321	0.127	0.023
upc(B.5.7)	2	6	V.C.	1250299	551.563	4.033	3639	1.639	0.024
upc(B.5.8)	2	4	V.C.	14417	7.264	1.599	1457	0.536	0.026
upc(B.5.9)	2	4	C.E.	9865	5.839	1.594	964	0.349	0.012
upc(B.5.10)	2	5	C.E.	2364	3.097	1.667	379	0.149	0.023
upc(B.5.11)	2	6	V.C.	145	2.219	1.552	330	0.119	0.019
upc(B.5.12)	4	6	V.C.	44238	19.438	1.729	607	0.243	0.024

lidity of the given property, and C.E.means that we need to find a counterexample. The fifth, sixth, and seventh columns list the number of state transitions, the amounts of consumed memory, and the elapsed times in our previous work[6], while the eighth, ninth, and tenth columns lists the numbers when all the optimization approaches described in Sec. 3 are applied (if possible). Please note that dashes (—) in the table means that model checking cannot be finished due to out of memory.

As Tab. 2 shows, the amounts of consumed memory and elapsed time are largely reduced in all the sample programs compare to our previous work. The numbers of state transitions are also largely reduced in almost all the cases, but the numbers sometimes slightly increased in a few cases. This is because we newly introduced the operation `Execute` in this paper, which was not explicitly handled in our previous works. In our previous works, when a thread P_i issues an instruction, its local effect in the thread (denoted by `Execute` P_i) and remote observation by the thread (denoted by `Reflect` $[P_i \Rightarrow P_i]$) are not distinguished because it is unnecessary to distinguish them in verifying sample programs written in the specification documents of Itanium [31]. However, this work explicitly distinguished them, and we precisely expressed all the rules by using `Execute`.

Tab. 3 and 4 show breakdowns of the effects of each optimization approach. More specifically, they compare the number of state transitions during model checking with varying the optimization approaches described in Sec. 3. In the tables, DS means that disabling speculation (explained in Sec. 3.2) is applied, PI means that prefetching instructions (explained in Sec. 3.3) is applied, RC means that removing the global time counter (explained in Sec. 3.4) is applied. Please note that the optimization of enhancing guards (explained in Sec. 3.1) is always applied in all the cases. In addition, please also note that DS (the optimization of disabling speculation)

Table 4: Comparison of Number of State Transitions with Varying Optimizations (under UPC memory model)

	PI,RC	RC	PI	
upc(B.5.1)	171	262	601	6726
upc(B.5.2)	461	1113	3517	7031898
upc(B.5.3)	177	282	983	8497
upc(B.5.4)	671	1348	90177	21473581
upc(B.5.5)	449	1193	3655	7332411
upc(B.5.6)	321	584	3607	103253
upc(B.5.7)	3639	12045	2862369	—
upc(B.5.8)	1457	2871	16733	40178890
upc(B.5.9)	964	1865	49961	18666427
upc(B.5.10)	379	727	3519	126362
upc(B.5.11)	330	1628	13128	—
upc(B.5.12)	607	3015	93369	—

is not applied to Tab. 4 because UPC memory model allows speculative behaviors of processes.

As shown in Tab. 3, when applying each optimization approach individually, the most effective optimization was PI (prefetching instructions), while RC (removing the global time counter) had no effect under Itanium memory model. DS (disabling speculations) showed its effectiveness when applied in conjunction with PI and/or RC. In all the cases, the best results were obtained when all the optimization approaches were applied.

On the other hand, under UPC memory model, individually applying each optimization approach reduced the number of state transitions largely (as shown in Tab. 4), while the best results were obtained when all the (applicable) optimization approaches were applied, in the same way as Itanium memory model.

Table 3: Comparison of Numbers of State Transitions with Varying Optimizations (under Itanium memory model)

	DS,PI,RC	DS,RC	DS,PI	DS	PI,RC	PI	RC	
itanium(A.1.4)	580	1111	1721761	37941192	4284594	4284594	97051291	97051291
itanium(A.2.5)	216	659	49941	14067196	123201	123201	35688796	35688796
itanium(A.3.6)	635	1130	297695	4978718	740983	740983	12717672	12717672
itanium(A.4.7)	271	2149	352871	—	915719	915719	—	—
itanium(A.5.8)	185	536	39853	9109062	103084	103084	24035899	24035899
itanium(A.5.9)	192	635	49941	14067196	123201	123201	35688796	35688796
itanium(A.6.10)	293	1513	306560	1.0639732e+09	823970	823970	—	2.8950955e+09
itanium(A.7.11)	365	4657	5016477	—	14157037	14157037	—	—
itanium(A.8.12)	106627	366795	—	—	—	—	—	—
itanium(A.9.13)	2983	8451	1.2706668e+08	—	4.5898596e+08	4.5898596e+08	—	—

5. RELATED WORK

While the conventional model checkers typically do not consider relaxed memory consistency models (e.g., [29, 25, 27, 7, 8], and so on), a number of works that tackle program verification under relaxed memory consistency models have been proposed in recent years [38, 39, 43, 41, 42, 21, 37, 20, 26, 32, 12, 13, 34, 14, 19, 10, 11, 24, 2, 3]. However, there exist few works on optimization of model checking under relaxed memory models [24, 2, 3].

Dan et al. [24] proposed a predicate abstraction approach for model checking under relaxed memory consistency models. In their approach, a program is first checked under the sequential consistency model with a standard predicate abstraction approach. Next, the program is transformed in the way that the characteristics of the memory consistency model is embedded in the translated program so that it can be checked under the sequential consistency model (similar to [10]). Then, in order to avoid the state explosion problem in model checking, the translated program is abstracted to a boolean program with the predicates inferred from the predicates obtained in the first predicate abstraction.

One difference from their approach and ours is that their approach is able to fully leverage the power of predicate abstraction by finding predicates from the property to be checked, while our approach utilizes the notion of predicate abstraction only in removing the global time counter by replacing it with the predicates that represent the order between two operations (Sec. 3.4).

Another difference is that our approach can support various memory consistency models because it is able to take a specification of a memory consistency model as an input, while their approach is specific to Partial Store Order memory model (PSO) and Total Store Order memory model (TSO), and it is not apparent whether their approach can be applied straightforwardly to the other memory consistency models.

Abdulla et al. [2, 3] proposed a program verification framework for safety properties under relaxed memory consistency models. In their approach, an input program to be checked is abstracted by combination of predicate abstraction and abstraction of store buffers, and the problem of checking safety properties is reduced to the reachability problem on the abstracted program. Then, the reachability problem is solved by existing model checkers such as BLAST [16] and CBMC [23]. Regarding predicate discovery, they adopt a conventional counterexample guided abstraction refinement (CEGAR [22]) approach.

One difference from their approach and ours is that, as explained above, our approach does not fully leverage the power of predicate abstraction (only used in abstracting away the global time counter) unlike their approach.

Another difference is that, as also described above, our approach supports various memory consistency models, while their approach is specific to TSO.

Burckhardt et al. [21] also proposed a program verification framework under relaxed memory consistency models. In their approach, an input program to be checked is translated into an intermediate representation including stores, loads, fences, and synchronization instructions. Finally, the intermediate representation with a memory consistency model is translated into a SAT formula, which is checked by a SAT solver. Their approach is similar to our work with respect to translating input programs into intermediate representations and using solvers. However, our approach supports various memory consistency models, while their approach is specific to sequential consistency and a certain relaxed memory consistency model (called *Relaxed* in [21]).

6. CONCLUSION

This paper described the optimization approaches for our model checker McSPIN. As shown in our previous papers [5, 6], McSPIN is able to perform model checking under various memory consistency models by taking a specification of a memory consistency model as an input. However, the original implementation of McSPIN is prone to suffer from the state explosion problem because the base model of model checking of McSPIN is very relaxed in order to support a wide range of relaxed memory consistency models. More specifically, the optimization approaches consist of enhancing guards (Sec. 3.1), disabling speculations (Sec. 3.2), prefetching instructions (Sec. 3.3), and removing the global time counter (Sec. 3.4). This paper also showed the effectiveness of the proposed optimization approaches by showing the experimental results with the optimized McSPIN.

One direction for future work is to apply predicate abstraction and CEGAR approaches to McSPIN (as in [24, 2, 3]) in order to further reduce the number of state transitions during model checking, which are necessary to verify larger programs.

Acknowledgments

In this research work we used the supercomputer of ACCMS, Kyoto University. This work was supported by JSPS KAKENHI Grant Numbers 25871113 and 25730050.

7. REFERENCES

[1] McSPIN. https://bitbucket.org/abet/mcspin/.

[2] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Automatic fence insertion in integer programs via

predicate abstraction. In *Proc. of SAS2012*, pages 164–180, 2012.

[3] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Memorax, a precise and sound tool for automatic fence insertion under tso. In *Proc. of TACAS2013*, pages 530–536, 2013.

[4] T. Abe and T. Maeda. A formal system of memory consistency models. In preparation for submission.

[5] T. Abe and T. Maeda. Model Checking with User-Definable Memory Consistency Models. In *Proc. of PGAS2013, short paper*, pages 225–230, 2013.

[6] T. Abe and T. Maeda. A general model checking framework for various memory consistency models. In *Proc. of HIPS2014*, pages 332–341. IEEE, 2014.

[7] T. Abe, T. Maeda, and M. Sato. Model Checking with User-Definable Abstraction for Partitioned Global Address Space Languages. In *Proc. of PGAS2012*, 2012.

[8] T. Abe, T. Maeda, and M. Sato. Model Checking Stencil Computations Written in a Partitioned Global Address Space Language. In *Proc. of HIPS2013*, pages 365–374, Cambridge, May 2013.

[9] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[10] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software Verification for Weak Memory via Program Transformation. In *Proc. of ESOP2013*, pages 1–20, 2013.

[11] J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Proc. of CAV2013*, pages 141–157. 2013.

[12] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. of CAV2010*, pages 258–272, July 2010.

[13] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. *ACM SIGPLAN Notices*, 45(1):7–18, Jan. 2010.

[14] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *Proc. of ESOP2012*, pages 26–46, Mar. 2012.

[15] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Mitchell, K. Nilsen, et al. The "double-checked locking is broken" declaration, 2000.

[16] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[18] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS1999*, pages 193–207, 1999.

[19] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and Enforcing Robustness against TSO. In *Proc. of ESOP2013*, pages 533–553. 2013.

[20] G. Boudol and G. Petri. Relaxed memory models: an operational approach. *ACM SIGPLAN Notices*, 44(1):392–403, Jan. 2009.

[21] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *Proc. of PLDI2007*, pages 12–21. ACM, 2007.

[22] E. M. Clarke. Counterexample-guided abstraction refinement. In *Proc. of TIME2003*, page 7, 2003.

[23] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Proc. of TACAS2004*, pages 168–176, 2004.

[24] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Predicate Abstraction for Relaxed Memory Models. In *Proc. of SAS2013*, 2013.

[25] A. Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. In *Proc. of PGAS2011*. ACM, 2011.

[26] R. Ferreira, X. Feng, and Z. Shao. Parameterized Memory Models and Concurrent Separation Logic. *Proc. of ESOP2010*, pages 267–286, 2010.

[27] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: Model Checking a New Programming Language with an "Old" Model Checker. In *Proc. of ICST2012*, pages 11–20, 2012.

[28] P. Haggar. *Practical Java: Programming Language Guide*. Addison-Wesley, 2000.

[29] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[30] Intel Corporation. *Itanium Architecture Software developer's manual*.

[31] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, 2010.

[32] R. Jagadeesan, C. Pitcher, and J. Riely. Generative Operational Semantics for Relaxed Memory Models. In *Proc. of ESOP2010*, pages 307–326, 2010.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[34] S. Mador-Haim, R. Alur, and M. Martin. Generating Litmus Tests for Contrasting Memory Consistency Models. In *Proc. of CAV2010*, pages 273–287. 2010.

[35] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, Jan. 2005.

[36] J. Reid and R. W. Numrich. Co-arrays in the next Fortran Standard. *Scientific Programming*, 15(1):9–26, 2007.

[37] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A Theory of Memory Models. In *Proc. of PPoPP2007*, pages 161–172, Mar. 2007.

[38] X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (crf): A new memory model for architects and compiler writers. In *Proc. of ISCA1999*, pages 150–161, 1999.

[39] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, Sept. 2004.

[40] The UPC Consortium. *UPC Language Specifications V1.2*, 2005.

[41] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *Proc. of ICFEM2004*, pages 30–45, 2004.

[42] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487, 2005.

[43] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos : A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. In *Proc. of IPDPS2004*, 2004.