Local Data Race Freedom with Non-multi-copy Atomicity

Tatsuya Abe(✉)

STAIR Lab, Chiba Institute of Technology, 2-17-1 Tsudanuma,
Narashino, Chiba 275-0016, Japan
abet@stair.center

Abstract. Data race freedom ensures the sequentially consistent behaviors of concurrent programs under relaxed memory consistency models (MCMs), and reduces the state explosion problem for software model checking with MCMs. However, data race freedom is too strong to include all interesting programs. In this paper, we define small-step operational semantics for relaxed MCMs, define an observable equivalence using the notion of bisimulation, and propose the property of local data race freedom (LDRF), which requires a kind of race freedom locally instead of globally. LDRF includes some interesting programs, such as the independent reads independent writes program, which is well known to exhibit curious behaviors under non-multi-copy atomic MCMs, and some concurrent copying garbage collection algorithms. In this paper, we introduce an optimization method called memory sharing for model checking of LDRF programs, and show that memory sharing optimization mitigates state explosion problems with non-multi-copy atomic MCMs through experiments.

Keywords: Data race freedom · Memory consistency model
Software model checking · State explosion problem
Non-multi-copy atomicity · Observable equivalence · Bisimulation
Independent reads independent writes program
Concurrent copying garbage collection algorithm

1 Introduction

Memory consistency models (MCMs), which specify how multiple threads use shared memory, have become extremely important because modern computer architectures have multiple cores. Their computing performance depends on the MCMs that the architectures adopt.

Saraswat et al. and Owens provided an insightful definition of MCM [19,25]. Saraswat et al. focused on an earlier study of observable equivalences between MCMs and data race freedom (DRF) by Gao and Sarkar [11]. Saraswat et al. defined some relaxed MCMs, and presented the so-called *fundamental property* such that DRF programs have only *observable* and *sequentially consistent (SC)*

© Springer International Publishing AG, part of Springer Nature 2018
M. M. Gallardo and P. Merino (Eds.): SPIN 2018, LNCS 10869, pp. 196–215, 2018.
https://doi.org/10.1007/978-3-319-94111-0_12

behaviors [17] on the relaxed MCMs. Saraswat et al. explained that the fundamental property ensures that most programmers writing DRF programs only have to be concerned about SC executions [25]. Owens considered that the fundamental property is the essence of MCMs and defined MCMs as rules that are designed to guarantee that DRF programs do not have non-SC behaviors [19].

The so-called *state explosion problem* of software model checking of DRF programs with MCMs is mitigated because non-SC behaviors of DRF programs on relaxed MCMs can be observably reduced to SC behaviors. However, DRF is so strong that we cannot expect it for all programs. Software model checking of *data racy* programs with MCMs still suffers from the state explosion problem.

Owens moderated a condition of DRF and presented the notion of *triangular race freedom (TRF)*, which includes the spinlock used in the Linux kernel [19]. Observable behaviors of TRF programs are reduced to SC behaviors on the x86-TSO [20], which is adopted by Intel architectures. However, TRF is a *global* property of a program; that is, *all* threads in the program must uniformly follow the TRF condition. TRF is also strongly specific to x86-TSO, which is stricter than modern MCMs, and cannot reduce behaviors on more relaxed MCMs to SC behaviors.

In this paper, we provide a formulation to define small-step operational semantics for relaxed memory consistency models and an observable equivalence on the semantics using the notion of bisimulation (note that Owens used trace semantics [19]), and propose the novel notion of *local data race freedom (LDRF)*, which claims that *some* threads follow a race free condition, but not all threads follow the condition, unlike DRF and TRF, which require that *all* threads must uniformly follow their conditions. LDRF is a variant of DRF in another direction that is different from the direction of TRF.

LDRF includes *the independent reads independent writes (IRIW) program*, which is well known to have curious behaviors under MCMs with *non-multi-copy atomicity*, which is more relaxed than x86-TSO. Although the IRIW program enjoys TRF, we cannot use the SC reduction for TRF programs on model checking with relaxed MCMs because TRF is specific to x86-TSO and we cannot observe the curious behavior on x86-TSO. Some *concurrent copying garbage collection (CCGC) algorithms* are typical LDRF programs. There exists no load-store race in any period between synchronization points at the garbage collection layer because end-user programmers must write DRF programs in programming languages with MCMs that require DRF, although it may be the case that a collector and mutators share variables because a collector communicates with mutators when collecting objects that have not been used.

In this paper, we also provide an optimization called *memory sharing* for the software model checking of LDRF programs with non-multi-copy atomic MCMs, which do not ensure atomicities among multiple effects (of a store) to multiple threads. We demonstrate the effectiveness of memory sharing optimization by conducting experiments for the IRIW program and some CCGC algorithms.

Related Work. To the best of our knowledge, there exists only the following one literature to propose an extension of DRF by focusing on locality.

Dolan et al. proposed a notion of local data race freedom independently, and showed that data race free portions of programs follow SC behaviors [9]. However, the MCMs in their paper are stricter than those in this paper. Actually, their LDRF do not provide any optimization on relaxed MCMs which allow load buffering and non-multi-copy atomicity.

There exists no relevant literature of an optimization specific to model checking with non-multi-copy atomic MCMs, although there exists a model checker such as Nidhugg [1] which supports the POWER MCM [12].

Owens used trace semantics and showed that x86-TSO behaviors of DRF programs can be reduced to SC behaviors [19]. In the present paper, we provide an alternative formalization of an observable equivalence using the notion of bisimulation, and show that behaviors of LDRF programs with non-multi-copy atomicity can be observably simulated by behaviors on a general machine with memory sharing optimization on a partition of threads.

Partitioning threads is reminiscent of *clustered-based partial order reduction*, which clusters multiple threads that are dependent on each other, and utilizes the independencies between the clusters to improve the efficiency of the partial order reduction [8]. However, the idea looks different from our memory sharing optimization, which makes independent threads on an LDRF program to share a common memory.

Ownership and separation are promising reasoning concepts regarding concurrency in program logic [18]. The author also proposed the notion of *observation invariants* in concurrent program logic [3]. However, this paper studies model checking, and provides no logic.

Outline. The remainder of this paper is organized as follows: In Sect. 2, we present a review of the observable equivalence of DRF programs. In Sect. 3, we introduce a general machine with non-multi-copy atomicity. In Sect. 4, we propose the notion of LDRF together with an optimization called memory sharing for software model checking with non-multi-copy atomic MCMs. In Sect. 5, we provide formal definitions that are introduced in Sects. 2, 3 and 4, and prove the validity of memory sharing under appropriate conditions. In Sect. 6, we explain how to implement memory sharing optimization in model checker VeriDAG that supports non-multi-copy atomic MCMs [2]. In Sect. 7, we present an assessment of the effectiveness of memory sharing using experiments. In Sect. 8, we conclude the paper by identifying future studies.

2 Observable Equivalence of Data Race Free Programs

In this section, we informally explain observable equivalence.

Saraswat et al. and Owens formally proved that non-SC behaviors of DRF and TRF programs, respectively, can be reduced to SC behaviors [19,25]. This means that non-SC behaviors do not disappear but cannot be observed. Non-SC behaviors exist internally on computer architectures.

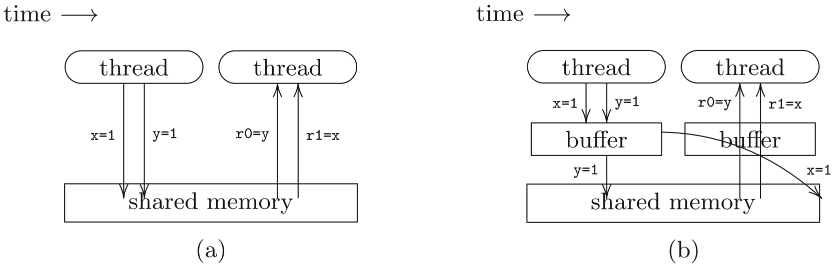

Fig. 1. Different behaviors without/with buffers

We can observe non-SC behavior for a data racy program: `(x=1; y=1)` ∥ `(r0=y; r1=x)`, where ∥ denotes parallel composition, `x` and `y` are shared variables, `r0` and `r1` are thread-local variables, and all variables are initialized to `0`. All six SC executions satisfy `r0==1 -> r1==1` when all four instructions are complete. However, there exists a non-SC execution, `x=1 (buffered); y=1 (buffered); y=1 (visible); r0=y; r1=x; x=1 (visible)` for which `r1==1 && r0==0` on modern computer architectures such that each thread might have one buffer that does not preserve the order of stores, as shown in Fig. 1(b); Fig. 1(a) shows SC behavior on computer architecture without a buffer.

Consequently, we cannot ignore relaxed MCMs on modern computer architectures. However, this is not the case for DRF programs. We consider a DRF program `(x=1; y=1)` ∥ `(r0=y'; r1=x')`, where `x'` and `y'` are shared variables. Although the buffer can delay the effects of the two stores to the shared memory, that can be ignored because the first thread cannot recognize whether the effects of the stores are delayed or the stores are not invoked.

To be precise, we cannot *describe* an assertion that distinguishes which effects are delayed in the assertion language; that is, the expressive power of the assertion language is not strong. We can distinguish them by describing an assertion `y==1 -> x==1` if the assertion language enables us to describe arbitrary states on computer architectures. However, it is reasonable to infer that the assertion language does not admit threads to read values of shared variables without loading the shared variables. The observable equivalence of a DRF program is defined as the non-existence of assertions that specify a non-SC behavior of the program, as formally defined in Sect. 5.3.

3 General Machine with Non-multi-copy Atomicity

In this section, we introduce a general machine, which assumes non-multi-copy atomicity [26].

There exist some computer architectures, such as ARMv7 [6] and POWER [12,26], that do not always assume multi-copy atomicity, that is, distinct threads can observe distinct behaviors of threads. We consider *the IRIW program* `(r0=y; r1=x)` ∥ `(r2=x; r3=y)` ∥ `x=1` ∥ `y=1`, where `r2` and

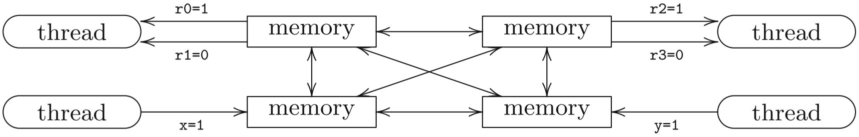

Fig. 2. A curious behavior on computer architectures with non-multi-copy atomicity

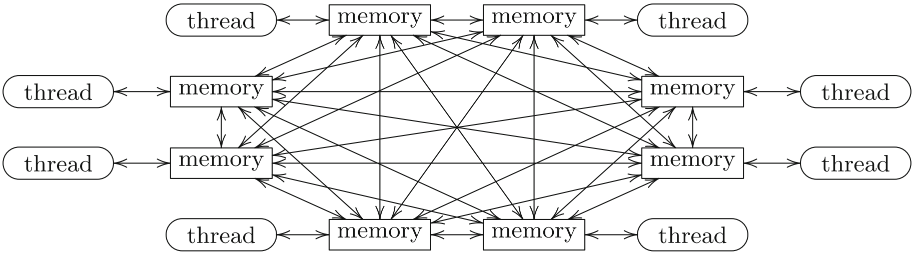

Fig. 3. General machine consisting of eight threads

`r3` are thread-local variables. The first and second threads read `x` and `y`, respectively, in program order. Therefore, the assertion `(r0==1 && r2==1) -> (r1==1 || r3==1)` appears to hold when the program ends. However, non-multi-copy atomic MCMs allow distinct threads to observe distinct behaviors of threads. For example, the first observes `y=1` and is invoked before `x=1` is invoked, whereas the second observes `x=1` and is invoked before `y=1` is invoked. This is natural for the computer architecture in Fig. 2.

As described in this paper, we consider a *general* machine with non-multi-copy atomicity. Each thread has its own *memory*. Each thread reads a shared variable from its own memory. A store to a memory is reflected to the other memories as shown in Fig. 3.

Every pair of memories is connected directly so that stores are passed through other memories, and buffers are separated to manage reflects of shared variables to memories. A question that arises is why no buffer exists in the general machine. Buffers are unnecessary for representing non-multi-copy atomic MCMs because each memory at each thread works as a buffer. The operational semantics of the general machine is formally defined in Sect. 5.2.

4 Memory Sharing and Local Data Race Freedom

In this section, we propose the notion of *LDRF* and an optimization called *memory sharing* for software model checking with non-multi-copy atomic MCMs.

LDRF is based on a simple concept. To introduce LDRF, we first consider DRF. Figure 4(a) denotes the behavior of a DRF program `x=1` ‖ `y=1` on computer architecture with buffers. The behavior is often regarded as being reduced to the behavior of computer architecture without a buffer, as shown in Fig. 4(b).

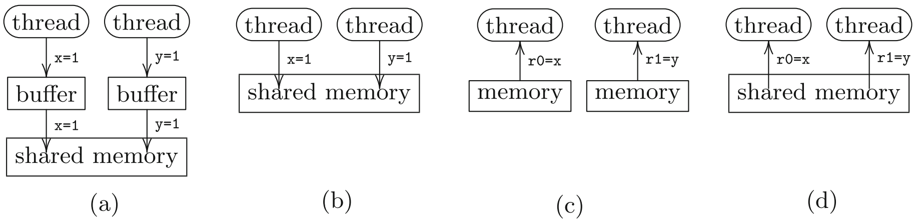

Fig. 4. Behaviors of a DRF program on various architectures

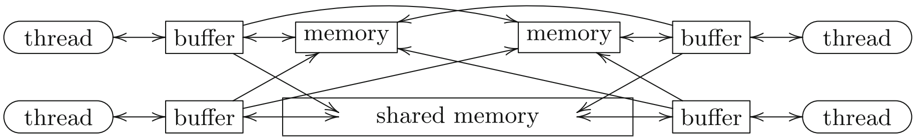

Fig. 5. Machine with memory sharing optimization

However, this can be also regarded such that two buffers are merged and integrated into the shared memory. We consider other threads that load `x` and `y` (to `r0` and `r1`, respectively). The loading of `x` and `y` from memories, as shown in Fig. 4(c), can be regarded as that from *one shared memory*, as shown in Fig. 4(d), by identifying buffers with memories.

This concept is the origin of the optimization of the general machine for which each thread has its own memory. If *some* threads enjoy load-store race freedom, then the threads can share their memories. Additionally, even if the threads do not enjoy load-load race freedom in a period between synchronization points, if there exists no store in the period on the other threads, then the threads can share their memories. We call that *local data race freedom (LDRF)*. This notion is formally defined in Sect. 5.

The behavior on the architecture shown in Fig. 2 can be observably simulated by behaviors on the machine shown in Fig. 5, which consists of four buffers, two memories, and one shared memory. The stores between writer threads on the general machine are ignored on the machine shown in Fig. 5. It must be the case that the curious behavior of the IRIW program can be observed, as shown in Fig. 6.

It may be considered slightly discouraging that memory sharing optimization cannot reduce behaviors on non-multi-copy atomic MCMs to SC behaviors; that is, memory sharing optimization does not improve model checking with non-multi-copy atomicity to a great degree, whereas SC reduction drastically addresses the state explosion problem; the sequential execution of multiple threads can simulate all parallel executions of multiple threads. However, LDRF includes IRIW programs, and memory sharing optimization mitigates the state explosion problem of model checking for LDRF programs. We demonstrate that memory sharing optimization is effective through experiments in Sect. 7.

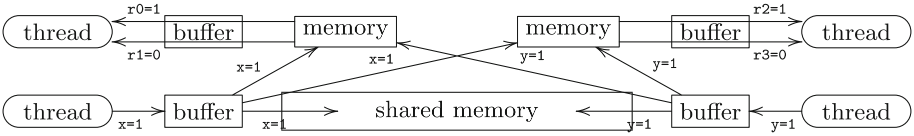

Fig. 6. Reduced behavior for memory sharing optimization

5 Formal Theory

In this section, we present formal definitions of the notions that were introduced informally in Sects. 2, 3 and 4, and prove that memory sharing is valid under appropriate conditions.

5.1 Concurrent Programs

The sets of instructions C^i and sequential programs S^i on thread i are defined as

$$C^i ::= \texttt{Nop}^i \mid r = \texttt{MV}^i\ e \mid r = \texttt{LD}^i\ x \mid x = \texttt{ST}^i\ e \mid r = \texttt{CAS}^i\ x\ e\ e$$
$$S^i ::= C^i \mid S^i; C^i,$$

where r denotes thread-local variables, x denotes shared variables, e denotes *thread-local* expressions (e.g., thread-local variables, constant value v, and arithmetic operations), and superscript i represents an identifier of the thread on which the associated statement is executed. In the remainder of this paper, this superscript is omitted when it is readily apparent from the context. The Nop statement represents an ordinary no-effect statement. We distinguish thread-local assignment statements from assignment statements to shared memory. MV denotes ordinary variable substitution. LD and ST denote read and write operations, respectively, for shared variables. CAS denotes *compare-and-swap* in a standard manner.

We adopt compare-and-swap as a primitive to ensure atomicity, whereas Owens adopted *locking* [19]. We adopted this approach because fine-grained synchronization, such as compare-and-swap, is preferred to coarse-grained synchronization, such as locking, on modern many-core computer architectures.

In this section, memory allocation, jump (conditional and loop statements), function call, and thread creation instructions are omitted, for simplicity. Actually, the model checker VeriDAG introduced in Sect. 6 and used at the experiments in Sect. 7 supports them by introducing the notions of the so-called addresses, labels, and basic blocks.

A concurrent program with N threads is defined as

$$P, Q ::= S^0 \parallel S^1 \parallel \cdots \parallel S^{N-1},$$

where $\parallel$ denotes a parallel composition of threads in a standard manner.

We assume that the number of threads is fixed during program execution.

To represent shared buffers and memories, we introduce the notion of partitions of a set of threads. We assume a partition $\{I_{(m)} \mid 0 \leq m < M\}$ of $\{0, \ldots, N-1\}$; that is, there exists M such that $0 \leq M \leq N$, $I_{(m)} \cap I_{(n)} = \varnothing$ for any $0 \leq m \neq n < M$, and $\bigsqcup\{\, I_{(m)} \mid 0 \leq m < M \,\} = \{0, \ldots, N-1\}$. An element $I_{(m)}$ of a partition is called a segment.

Threads in a common segment are regarded to share a common memory. For example, the writer threads (whose identifiers are 2 and 3) in the IRIW program `(r0=y; r1=x)` ‖ `(r2=x; r3=y)` ‖ `x=1` ‖ `y=1` can share a common memory as shown in Fig. 6. In the case, the partition of thread identifiers $\{0, 1, 2, 3\}$ is $\{\{0\}, \{1\}, \{2, 3\}\}$.

We define a state. Register ς takes a thread-local variable r and returns value v. Shared memory σ takes a segment $I_{(m)}$ of the partition and shared variable x, and returns value v. Buffer Σ takes a pair of thread identifier and a segment $\langle i, I_{(m)}\rangle$, and returns a *queue set*, where one queue is defined for *each* shared variable. Four methods *method* are defined for a queue set qs. One method qs.enqueue(x, v) enqueues v at x in qs. Another method qs.dequeue(x, v) dequeues a value at x in qs, and returns that the value is v. Another method qs.empty(x) determines that the queue at x in qs is empty. The other method qs.latest(x) returns the latest value at x in qs without dequeuing any element.

5.2 Operational Semantics

In the following, we assume a partition $\{\, I_{(m)} \mid 0 \leq m < M \,\}$ of $\{0, \ldots, N-1\}$, and write a meta-variable I as a segment $I_{(m)}$ of the partition.

For brevity, we write σ^I and $\Sigma^{i,I}$ as $\sigma(I)$ and $\Sigma(\langle i, I\rangle)$, respectively. We use an update operation of function f in a standard manner as follows:

$$f[a := b](c) = \begin{cases} b & \text{if } a = c \\ f(c) & \text{otherwise.} \end{cases}$$

We also write $\sigma[\sigma^I[x := v]]$ as $\sigma[I := \sigma^I[x := v]]$ because it is readily apparent that the update is about I. Similarly, for brevity, we express $\Sigma[\Sigma^{i,I}.\mathit{method}]$ as $\Sigma[\langle i, I\rangle := \Sigma^{i,I}.\mathit{method}]$. We respectively write $\sigma[\,\{\, \sigma^I[x := v] \mid I \,\}\,]$ and $\Sigma[\,\{\, \Sigma^{i,I}.\mathit{method} \mid I \,\}\,]$ as $\sigma[\sigma^{I_{(0)}}[x := v]] \cdots [\sigma^{I_{(M-1)}}[x := v]]$ and $\Sigma[\Sigma^{i,I_{(0)}}.\mathit{method}] \cdots [\Sigma^{i,I_{(M-1)}}.\mathit{method}]$.

Furthermore, we introduce an update of shared memory by a shared buffer as

$$\sigma^I[\Sigma^{i,I}](x) = \begin{cases} \Sigma^{i,I}.\text{latest}(x) & \text{if the queue at } x \text{ is not empty} \\ \sigma^I(x) & \text{otherwise.} \end{cases}$$

A *state* is defined as a triple: $\langle \varsigma, \sigma, \Sigma\rangle$. A *configuration* is defined as $\langle P, \langle \varsigma, \sigma, \Sigma\rangle\rangle$. A small-step operational semantics is defined as shown in Fig. 7. Transition $\xrightarrow{\text{c}}$ indicates that an instruction is invoked and that a state is updated.

$$\frac{}{\langle r = \mathtt{MV}^i\ e, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle \mathtt{Nop}^i, \langle \varsigma[r := \llparenthesis e \rrparenthesis_\varsigma], \sigma, \Sigma \rangle\rangle}$$

$$\frac{i \in I}{\langle r = \mathtt{LD}^i\ x, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle \mathtt{Nop}^i, \langle \varsigma[r := \sigma^I[\Sigma^{i,I}](x)], \sigma, \Sigma \rangle\rangle}$$

$$\frac{i \in I}{\langle x = \mathtt{ST}^i\ e, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle \mathtt{Nop}^i, \langle \varsigma, \sigma, \Sigma[\{\, \Sigma^{i,I}.\mathrm{enqueue}(x, \llparenthesis e \rrparenthesis_\varsigma) \mid I \,\}] \rangle\rangle}$$

$$\frac{i \in I \qquad\qquad \sigma^I[\Sigma^{i,I}](x) = \llparenthesis e_0 \rrparenthesis_\varsigma}{\begin{array}{l}\langle r = \mathtt{CAS}^i\ x\ e_0\ e_1, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \\ \quad \langle \mathtt{Nop}^i, \langle \varsigma[r := 1], \sigma[\{\, \sigma^I[x := \llparenthesis e_1 \rrparenthesis_\varsigma] \mid I \,\}], \Sigma[\{\, \Sigma^{i,I}.\mathrm{empty}(x) \mid I \,\}] \rangle\rangle\end{array}}$$

$$\frac{i \in I \qquad\qquad \sigma^I[\Sigma^{i,I}](x) \neq \llparenthesis e_0 \rrparenthesis_\varsigma}{\langle r = \mathtt{CAS}^i\ x\ e_0\ e_1, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle \mathtt{Nop}^i, \langle \varsigma[r := 0], \sigma, \Sigma \rangle\rangle}$$

$$\frac{\langle P, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle \mathtt{Nop}, \langle \varsigma', \sigma', \Sigma' \rangle\rangle}{\langle P; Q, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle Q, \langle \varsigma', \sigma', \Sigma' \rangle\rangle} \qquad \frac{\langle P, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle P', \langle \varsigma', \sigma', \Sigma' \rangle\rangle}{\langle P; Q, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle P'; Q, \langle \varsigma', \sigma', \Sigma' \rangle\rangle}$$

$$\frac{\langle P, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle P', \langle \varsigma', \sigma', \Sigma' \rangle\rangle}{\langle P \parallel Q, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle P' \parallel Q, \langle \varsigma', \sigma', \Sigma' \rangle\rangle} \qquad \frac{\langle Q, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle Q', \langle \varsigma', \sigma', \Sigma' \rangle\rangle}{\langle P \parallel Q, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{c}}{\to} \langle P \parallel Q', \langle \varsigma', \sigma', \Sigma' \rangle\rangle}$$

$$\frac{}{\langle P, \langle \varsigma, \sigma, \Sigma \rangle\rangle \overset{\mathrm{e}}{\to} \langle P, \langle \varsigma, \sigma[\sigma^I[x := v]], \Sigma[\Sigma^{i,I}.\mathrm{dequeue}(x, v)] \rangle\rangle}$$

Fig. 7. Operational semantics

Specifically, $r = \mathtt{MV}^i\ e$ evaluates e at ς and updates ς, where $\llparenthesis e \rrparenthesis_\varsigma$ represents the valuation of expression e as

$$\llparenthesis v \rrparenthesis_\varsigma = v \qquad \llparenthesis r \rrparenthesis_\varsigma = \varsigma(r) \qquad \llparenthesis e_0 + e_1 \rrparenthesis_\varsigma = \llparenthesis e_0 \rrparenthesis_\varsigma + \llparenthesis e_1 \rrparenthesis_\varsigma \qquad \cdots\ .$$

Instruction $r = \mathtt{LD}^i\ x$ evaluates x on $\Sigma^{i,I}$ if $\Sigma^{i,I}(x)$ is defined, and on σ^I otherwise, where $i \in I$, and updates ς. Instruction $x = \mathtt{ST}^i\ e$ evaluates e on ς and updates not σ^I but $\Sigma^{i,I}$ for any I. The effect of the store operation is buffered in $\Sigma^{i,I}$ for any I. Instruction $r = \mathtt{CAS}^i\ x\ e_0\ e_1$ atomically loads x, compares the evaluation of e_0, stores the evaluation of e_1 at x, and returns 1 to r if the values of x and e_0 are equal; it returns 0 otherwise. Sequential and parallel compositions follow standard methods. In this paper, parallel composition is defined as a non-commutative and non-associative operator because the indices of segments are sensitive to operational semantics.

Whereas a transition $\overset{\mathrm{c}}{\to}$ invokes and consumes one instruction, a transition $\overset{\mathrm{e}}{\to}$, which represents an effect that is reflected from a buffer to shared memory, does not invoke or consume any instructions.

5.3 Assertion Language

The assertion language is defined as

$$\varphi ::= e = e \mid e \leq e \mid \neg\,\varphi \mid \varphi \supset \varphi \mid \forall\, r.\, \varphi \ .$$

Relation $\varsigma \models \varphi$ is defined in a standard manner as

$$\begin{aligned} \varsigma \models e_0 = e_1 &\iff \llparenthesis e_0 \rrparenthesis_\varsigma = \llparenthesis e_1 \rrparenthesis_\varsigma & \varsigma \models e_0 \leq e_1 &\iff \llparenthesis e_0 \rrparenthesis_\varsigma \leq \llparenthesis e_1 \rrparenthesis_\varsigma \\ \varsigma \models \neg\,\varphi &\iff \varsigma \not\models \varphi & \varsigma \models \varphi \supset \varphi' &\iff \varsigma \models \varphi \text{ implies } \varsigma \models \varphi' \\ \varsigma \models \forall\, r.\, \varphi(r) &\iff \varsigma \models \varphi(v) \text{ for any } v \ . \end{aligned}$$

Relation $\langle P, \langle \varsigma, \sigma, \varSigma \rangle\rangle \models \varphi$, which indicates that the configuration satisfies the assertion, is defined as $\varsigma \models \varphi$. An assertion can be inserted anywhere in a program, while a system in which an assertion must be located at the end of a program cannot support divergent programs.

The assertion language has no shared variable. The satisfiability is defined by registers only. Consequently, the assertion language requires the loading of a shared variable to identify the value of the shared variable.

5.4 Local Data Race Freedom and Observable Equivalence

An objective of in this section is to define a relation $\sim$ between configurations satisfying:

1. if cfg_0 and cfg_1 are related by the memory sharing optimization, then $cfg_0 \sim cfg_1$ holds,
2. if $cfg_0 \sim cfg_1$ holds, then $cfg_0 \models \varphi$ coincides with $cfg_1 \models \varphi$ for any φ, and
3. the relation $\sim$ is preserved by transitions on operational semantics.

Claim 1 means that the relation $\sim$ contains a pair of configurations $\langle cfg_0, cfg_1 \rangle$ where cfg_0 and cfg_1 are related by the memory sharing optimization. Claim 2 means that satisfiability of an assertion on cfg_0 can be checked by checking satisfiability of the assertion on cfg_1 and vice versa. Claim 3 means that the relation $\sim$, in particular, the memory sharing optimization is robust to transition.

First, let us define a relation $\sim$ satisfying Claim 1. A set of sequential programs $\{S_0, \ldots, S_{n-1}\}$ is called *load-store race free* if $R(S_k) \cap W(S_l) = \varnothing$ for any $0 \leq k \neq l < n$, where

$$R(S) = \bigcup \{\, R(C) \mid C \in S \,\} \qquad R(C) = \begin{cases} \{x\} & \text{if } C \text{ is } r = \texttt{LD}\ x \text{ or } r = \texttt{CAS}\ x\ e_0\ e_1 \\ \varnothing & \text{otherwise.} \end{cases}$$

$$W(S) = \bigcup \{\, W(C) \mid C \in S \,\} \quad W(C) = \begin{cases} \{x\} & \text{if } C \text{ is } x = \texttt{ST}\ e \text{ or } r = \texttt{CAS}\ x\ e_0\ e_1 \\ \varnothing & \text{otherwise.} \end{cases}$$

A concurrent program $P \equiv S^0 \parallel \cdots \parallel S^{N-1}$ is called *LDRF with respect to partition* $\{I_{(m)} \mid 0 \leq m < M\}$ *of* $\{0, \ldots, N-1\}$ if for any segment $I_{(m)}$

- $\{S^i \mid i \in I_{(m)}\}$ is load-store race free, and
- for any x and $i \neq j \in I_{(m)}$, if $x \in R(S^i) \cap R(S^j)$ then $x \notin W(S^k)$ for any $0 \leq k < N$.

By definition, every DRF program that consists of N threads is LDRF with respect to discrete partition $\{\{m\} \mid 0 \leq m < N\}$, although memory sharing optimization based on the discrete partition never improves model checking with non-multi-copy atomicity.

Let $\mathcal{R}$ be a relation. Relation $\mathcal{R}^+$ represents the transitive closure of $\mathcal{R}$. Relation $\mathcal{R}^*$ represents the reflexive and transitive closure of $\mathcal{R}$.

Let $\{I_{j,(m)} \mid 0 \leq m < M_j\}$ be a partition of $\{0, \ldots, N-1\}$ for any $j = 0, 1$. We define an *expansion relation* as $\langle P_0, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle \succsim \langle P_1, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle$ if

- for any $j = 0, 1$
  - $P_j \equiv S^0 \parallel \cdots \parallel S^{N-1}$,
  - $\langle \varsigma, \sigma_j, \Sigma_j \rangle$ is defined on $\{\, I_{j,(m)} \mid 0 \leq m < M_j \,\}$, and
  - P_j is LDRF with respect to partition $\{I_{j,(m)} \mid 0 \leq m < M_j\}$,
- $\{I_{0,(m)} \mid 0 \leq m < M_0\}$ is a refinement of $\{I_{1,(m)} \mid 0 \leq m < M_1\}$, that is, for any $0 \leq m_0 < M_0$, there exists $0 \leq m_1 < M_1$ such that $I_{0,(m_0)} \subseteq I_{1,(m_1)}$, and
- for any x and $0 \leq i < N$,
  - if $x \in R(S^i)$, then ${\sigma_0}^{I_0}[{\Sigma_0}^{i,I_0}](x) = {\sigma_1}^{I_1}[{\Sigma_1}^{i,I_1}](x)$,
  - if $\langle P_0, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{+} \langle P_0, \langle \varsigma, {\sigma_0}', {\Sigma_0}' \rangle \rangle$, then there exist ${\sigma_1}'$ and ${\Sigma_1}'$ such that $\langle P_1, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{*} \langle P_1, \langle \varsigma, {\sigma_1}', {\Sigma_1}' \rangle \rangle$ and ${\sigma_0}'^{I_0}[{\Sigma_0}'^{i,I_0}](x) = {\sigma_1}'^{I_1}[{\Sigma_1}'^{i,I_1}](x)$, and
  - if $\langle P_1, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{+} \langle P_1, \langle \varsigma, {\sigma_1}', {\Sigma_1}' \rangle \rangle$, then there exist ${\sigma_0}'$ and ${\Sigma_0}'$ such that $\langle P_0, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{+} \langle P_0, \langle \varsigma, {\sigma_0}', {\Sigma_0}' \rangle \rangle$ and ${\sigma_0}'^{I_0}[{\Sigma_0}'^{i,I_0}](x) = {\sigma_1}'^{I_1}[{\Sigma_1}'^{i,I_1}](x)$,

  where I_0 and I_1 denote the unique segments such that $i \in I_0$ and $i \in I_1$, respectively.

Intuitively, $\langle P_0, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle \succsim \langle P_1, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle$ means that $\langle P_1, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle$ is the application of the memory sharing optimization to $\langle P_0, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle$.

It is noteworthy that the third condition is satisfied by executing programs with `initstate` $\equiv \langle \{\_ \mapsto 0\}, \{\, \_ \mapsto \{\_ \mapsto 0\}\}, \{\_ \mapsto \varnothing\} \rangle$ because no effect can be reflected from the state.

We describe a basic property that expansion relation $\succsim$ is a contextual relation. This property is useful to check whether two configurations are related by $\succsim$, because this property ensures that two configurations which are partially related by $\succsim$ are totally related by $\succsim$.

Proposition 1. *If* $\langle P_0, \langle \varsigma, \sigma_0 \restriction \mathcal{I}_0, \Sigma_0 \restriction (\mathcal{I}_0 \times \mathcal{I}_0) \rangle \rangle \succsim \langle P_1, \langle \varsigma, \sigma_1 \restriction \mathcal{I}_1, \Sigma_1 \restriction (\mathcal{I}_1 \times \mathcal{I}_1) \rangle \rangle$ *holds, then* $\langle P_0 \parallel S^{N-1}, \langle \varsigma, \sigma_0, \Sigma_0 \rangle \rangle \succsim \langle P_1 \parallel S^{N-1}, \langle \varsigma, \sigma_1, \Sigma_1 \rangle \rangle$ *holds where*

- $\mathcal{I}_j = \{\, I_{j,(m)} \mid 0 \leq m < M_j \,\}$ *is a partition of* $\{0, \ldots, N-2\}$ *for any* $j = 0, 1$,
- $\mathcal{I} = \{N-1\}$,
- $\sigma_0 \restriction \mathcal{I} = \sigma_1 \restriction \mathcal{I}$,
- $\Sigma_0 \restriction (\mathcal{I} \times \mathcal{I}) = \Sigma_1 \restriction (\mathcal{I} \times \mathcal{I})$, *and*

- *$\Sigma_0 \restriction (\mathcal{I}_0 \times \mathcal{I})$, $\Sigma_0 \restriction (\mathcal{I} \times \mathcal{I}_0)$, $\Sigma_1 \restriction (\mathcal{I}_1 \times \mathcal{I})$, and $\Sigma_1 \restriction (\mathcal{I} \times \mathcal{I}_1)$ are empty*

where $\operatorname{dom} f$ *is the domain of f, and $f \restriction A$ is the restriction of f on A for any $A \subseteq$* $\operatorname{dom} f$*.*

Proof. It is routine to check the conditions of $\succsim$ according to the definitions of $\succsim$ and the operational semantics in Fig. 7. □

Next, let us confirm that the relation $\sim$ satisfies Claim 2. Satisfiabilities of assertions on configurations are invariant to the memory sharing optimization, which is shown as follows:

Proposition 2. *If $cfg_0 \succsim cfg_1$, then for any φ, $cfg_0 \vDash \varphi$ coincides with $cfg_1 \vDash \varphi$.*

Proof. It is obvious because expansion relation $\succsim$ is defined between configurations that have the *same* register, and $\vDash$ is defined by the register and the assertion φ, which cannot refer to variables on memories and buffers. □

We define *observable equivalence* $\sim$ as the reflexive, symmetric, and transitive closure of the expansion relation $\succsim$. Intuitively, $cfg_0 \sim cfg_1$ means that cfg_0 and cfg_1 are related by the memory sharing optimization.

Proposition 3. *If $cfg_0 \sim cfg_1$, then for any φ, $cfg_0 \vDash \varphi$ coincides with $cfg_1 \vDash \varphi$.*

Proof. It is immediate from the definition of observable equivalence $\sim$. □

Finally, let us prove that the relation $\sim$ satisfies Claim 3. We define a property by which a relation is preserved by transitions on operational semantics. We designate $\mathcal{R}$ as a *bisimulation* if $cfg_0\ \mathcal{R}\ cfg_1$ implies

- if $cfg_0 \xrightarrow{\mathrm{c}} cfg_0{}'$, then $cfg_1{}'$ exists such that $cfg_1 \xrightarrow{\mathrm{c}} cfg_1{}'$ and $cfg_0{}'\ \mathcal{R}\ cfg_1{}'$,
- if $cfg_1 \xrightarrow{\mathrm{c}} cfg_1{}'$, then $cfg_0{}'$ exists such that $cfg_0 \xrightarrow{\mathrm{c}} cfg_0{}'$ and $cfg_0{}'\ \mathcal{R}\ cfg_1{}'$,
- if $cfg_0 \xrightarrow{\mathrm{e}} cfg_0$, then $cfg_1{}'$ exists such that $cfg_1 \xrightarrow{\mathrm{e}}{}^{*}\ cfg_1{}'$ and $cfg_0{}'\ \mathcal{R}\ cfg_1{}'$, and
- if $cfg_1 \xrightarrow{\mathrm{e}} cfg_1{}'$, then $cfg_0{}'$ exists such that $cfg_0 \xrightarrow{\mathrm{e}}{}^{*}\ cfg_0{}'$ and $cfg_0{}'\ \mathcal{R}\ cfg_1{}'$.

Proposition 4. *Let $\mathcal{R}_0$ and $\mathcal{R}_1$ be bisimulations. The inverse of $\mathcal{R}_0$, that is, $\{\langle cfg_1, cfg_0\rangle \mid \langle cfg_0, cfg_1\rangle \in \mathcal{R}_0\}$, and the composition of $\mathcal{R}_0$ and $\mathcal{R}_1$, that is, $\{\langle cfg_0, cfg_2\rangle \mid \langle cfg_0, cfg_1\rangle \in \mathcal{R}_0, \langle cfg_1, cfg_2\rangle \in \mathcal{R}_1$ for some $cfg_1\}$, are bisimulations. Therefore, the reflexive, symmetric, and transitive closure of $\mathcal{R}_0$ is a bisimulation.*

Proof. It is obvious by the definition of bisimulation.

LDRF is necessary to define $\succsim$ as a bisimulation. For example, $\langle x = \mathtt{ST}^0\ 1 \parallel r = \mathtt{LD}^1\ x \parallel r' = \mathtt{LD}^2\ x, \langle\{\_ \mapsto 0\}, \sigma_0, \Sigma_0\rangle\rangle \not\succsim \langle x = \mathtt{ST}^0\ 1 \parallel r = \mathtt{LD}^1\ x \parallel r' = \mathtt{LD}^2\ x, \langle\{\_ \mapsto 0\}, \sigma_1, \Sigma_1\rangle\rangle$, where

$$\begin{aligned}
\sigma_0 &= \{\,\{0\} \mapsto \{\_ \mapsto 0\}, \{1\} \mapsto \{\_ \mapsto 0\}, \{2\} \mapsto \{\_ \mapsto 0\}\} \\
\varSigma_0 &= \{\langle 0, \{0\}\rangle \mapsto \varnothing, \langle 0, \{1\}\rangle \mapsto \varnothing, \langle 0, \{2\}\rangle \mapsto \varnothing, \langle 1, \{0\}\rangle \mapsto \varnothing, \langle 1, \{1\}\rangle \mapsto \varnothing, \\
&\qquad \langle 1, \{2\}\rangle \mapsto \varnothing, \langle 2, \{0\}\rangle \mapsto \varnothing, \langle 2, \{1\}\rangle \mapsto \varnothing, \langle 2, \{2\}\rangle \mapsto \varnothing\} \\
\sigma_1 &= \{\,\{0\} \mapsto \{\_ \mapsto 0\}, \{1, 2\} \mapsto \{\_ \mapsto 0\}\} \\
\varSigma_1 &= \{\langle 0, \{0\}\rangle \mapsto \varnothing, \langle 0, \{1, 2\}\rangle \mapsto \varnothing, \langle 1, \{0\}\rangle \mapsto \varnothing, \langle 1, \{1, 2\}\rangle \mapsto \varnothing, \\
&\qquad \langle 2, \{0\}\rangle \mapsto \varnothing, \langle 2, \{1, 2\}\rangle \mapsto \varnothing\}
\end{aligned}$$

because $\langle r = \mathtt{LD}^1\ x \parallel r' = \mathtt{LD}^2\ x, \langle \{\_ \mapsto 0\}, \sigma'_0, \varSigma'_0\rangle\rangle \not\gtrsim \langle r = \mathtt{LD}^1\ x \parallel r' = \mathtt{LD}^2\ x, \langle \{\_ \mapsto 0\}, \sigma'_1, \varSigma_1\rangle\rangle$, where

$$\begin{aligned}
\sigma'_0 &= \{\,\{0\} \mapsto \{x \mapsto 1, \_ \mapsto 0\}, \{1\} \mapsto \{x \mapsto 1, \_ \mapsto 0\}, \{2\} \mapsto \{\_ \mapsto 0\}\} \\
\varSigma'_0 &= \{\langle 0, \{0\}\rangle \mapsto \varnothing, \langle 0, \{1\}\rangle \mapsto \varnothing, \langle 0, \{2\}\rangle \mapsto x = 1, \langle 1, \{0\}\rangle \mapsto \varnothing, \langle 1, \{1\}\rangle \mapsto \varnothing, \\
&\qquad \langle 1, \{2\}\rangle \mapsto \varnothing, \langle 2, \{0\}\rangle \mapsto \varnothing, \langle 2, \{1\}\rangle \mapsto \varnothing, \langle 2, \{2\}\rangle \mapsto \varnothing\} \\
\sigma'_1 &= \{\,\{0\} \mapsto \{x \mapsto 1, \_ \mapsto 0\}, \{1, 2\} \mapsto \{x \mapsto 1, \_ \mapsto 0\}\}
\end{aligned}$$

after $x = \mathtt{ST}^0\ 1$ is invoked.

Lemma 5. *Expansion relation $\gtrsim$ is a bisimulation.*

Proof. Each shared variable has its own queue. Therefore, any pair of e-transitions related to distinct shared variables can be reordered.

Let $\langle P_0, \langle \varsigma_0, \sigma_0, \varSigma_0\rangle\rangle \gtrsim \langle P_1, \langle \varsigma_1, \sigma_1, \varSigma_1\rangle\rangle$. Assume that S^j belongs to a common segment with S^i on $\langle P_1, \langle \varsigma_1, \sigma_1, \varSigma_1\rangle\rangle$. If $R(S^i) \cap R(S^j) = \varnothing$ holds, then the first item of the third condition of $\gtrsim$ does not matter. If $x \in R(S^i) \cap R(S^j)$, then there exists no $x = \mathtt{ST}^k\ e$ according to LDRF; the first item of the third condition of $\gtrsim$ also does not matter.

Otherwise, the queues of S^i and S^j are separated; any $\xrightarrow{\mathrm{e}}$ on P_0 can be simulated by $\xrightarrow{\mathrm{e}}^*$ on P_1. The second and third items of the third condition of $\gtrsim$ can be checked easily because it is sufficient to consider effects except those by invoking the $x = \mathtt{ST}^i\ e$ instruction.

The case of $\mathtt{CAS}$ is similar. The other cases related to the c-transition are routine because $\varSigma$ remains unchanged. The c-transition of P_1 is similar. The cases of e-transitions are readily apparent by definition. □

Theorem 6. *Observable equivalence $\sim$ is a bisimulation.*

Proof. It is immediate from Proposition 4 and Lemma 5. □

Thus, the objective has been accomplished because Claim 1 is satisfied by the definition of observable equivalence, Claim 2 is satisfied by Proposition 3 and Claim 3 is satisfied by Theorem 6.

6 Implementation of Memory Sharing Optimization

In this section, we explain the implementation of memory sharing optimization.

We implemented memory sharing optimization on a stateful model checker VeriDAG, which performs model checking not only with multi-copy atomicity, but also non-multi-copy atomicity [2]. VeriDAG takes a concurrent program written in the C programming language (or a sequence of LLVM-IRs) and an MCM as inputs, and generates a directed acyclic graph called a *program graph* as an intermediate representation, which was introduced in [4]. Program graphs are defined to support various MCMs, such as relaxed memory ordering (RMO) [5] that allows load-load reordering. Furthermore, the definition of program graphs was extended to support non-multi-copy atomicity [2]. Operational semantics for program graphs can simulate the general machine introduced in Sect. 3, and the formal discussion in Sect. 5 can be applied to program graphs and its operational semantics.

A node of a program graph corresponds to an instruction or an effect of an instruction from thread i to a set of threads I written as $x = \mathtt{E}^{i,I}\ v$, which makes the execution of the instruction visible to the other threads. For example, an effect means a reflect from the store buffer on thread i to shared memory under x86-TSO. An effect also means a reflect from the memory on thread i to the other memories on threads I under the POWER MCM.

Figures 8(a) and (b) depict program graphs that consist of $x = \mathtt{ST}^1\ 3$ and its effects on three threads under multi-copy atomicity and non-multi-copy atomicity, respectively. The edges of the program graph denote dependencies. In the figures, $x = \mathtt{E}^{1,\{0,1,2\}}\ 3$, $x = \mathtt{E}^{1,\{0\}}\ 3$, $x = \mathtt{E}^{1,\{1\}}\ 3$, and $x = \mathtt{E}^{1,\{2\}}\ 3$ are necessarily invoked after $x = \mathtt{ST}^1\ 3$ is invoked.

One method to implement memory sharing optimization is as follows: We introduced a partition of threads that corresponded to memory sharing by extending $\mathtt{E}$ to take not only the set of threads that corresponded to multi-copy atomicity or a singleton that corresponded to non-multi-copy atomicity, but also *any segment* of the partition. For example, $x = \mathtt{ST}^1\ 3$ on an LDRF program that consists of three threads was represented as shown in Fig. 8(c), where the two threads share one memory. Another method is to abandon unnecessary effects, although it was not adopted in this work. Because VeriDAG was well-designed to address reflected stores partially, the implementation of memory sharing optimization was straightforward.

Program graphs on VeriDAG have been modified from the original ones described in a previous paper [4], for performance improvement. An *atomic edge*, denoted by $\Longrightarrow$, means that if its source node is consumed, then its target node is preferably chosen at one of the roots of the updated program graph in the next step; that is, a pair of instructions related by an atomic edge is invoked instantaneously. Atomic edges are carefully implemented to not disturb the so-called partial order reduction based on the notion of *ample sets* [21] using invisibility that is already implemented in VeriDAG.

The previous program graphs in Fig. 8 were modified, as shown in Fig. 9. The validity of the optimization was ensured because there was no necessity to

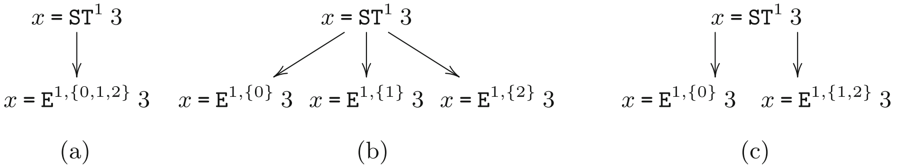
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0,1,2\}}\ 3$
(a)
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0\}}\ 3$
$x = \mathtt{E}^{1,\{1\}}\ 3$
$x = \mathtt{E}^{1,\{2\}}\ 3$
(b)
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0\}}\ 3$
$x = \mathtt{E}^{1,\{1,2\}}\ 3$
(c)

Fig. 8. Program graphs with multi-copy atomicity, non-multi-copy atomicity, and memory sharing optimization

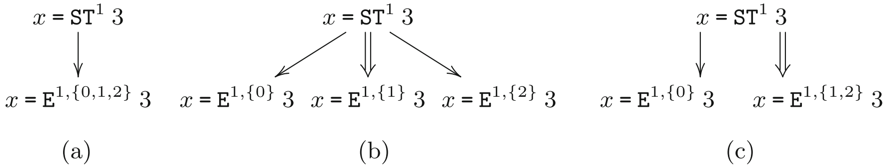
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0,1,2\}}\ 3$
(a)
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0\}}\ 3$
$x = \mathtt{E}^{1,\{1\}}\ 3$
$x = \mathtt{E}^{1,\{2\}}\ 3$
(b)
$x = \mathtt{ST}^1\ 3$
$x = \mathtt{E}^{1,\{0\}}\ 3$
$x = \mathtt{E}^{1,\{1,2\}}\ 3$
(c)

Fig. 9. Program graphs with atomic edges

consider interleavings between these nodes because every thread i used its own σ^I, and every σ^I was always used via $\Sigma^{i,I}$ in the form of $\sigma^I[\Sigma^{i,I}]$, where $i \in I$. Note that thread 2 did not read x in the last program graph according to the LDRF condition. Additionally, note that the first program graph had no atomic edge because the graph was generated with multi-copy atomicity.

7 Experiments

In this section, we demonstrate the effectiveness of memory sharing optimization by conducting model checking of the IRIW program and CCGC algorithms. We used VeriDAG, on which memory sharing optimization was implemented, as explained in Sect. 6. MCMs were RMO with and without multi-copy-atomicity. RMO with multi-copy-atomicity corresponds to the SPARC RMO MCM. RMO with non-multi-copy-atomicity corresponds to the POWER MCM. The experimental environment was as follows: the CPU was Intel Xeon E5-1620 v4 3.50 GHz, the memory was DDR4-2400 256 GB, the OS was Ubuntu 17.10, and VeriDAG was compiled using Glasgow Haskell Compiler 8.0.2.

7.1 Independent Reads Independent Writes Program

The IRIW program is `(r0=y; r1=x)` ‖ `(r2=x; r3=y)` ‖ `x=1` ‖ `y=1`, and an assertion to observe curious behaviors under non-multi-copy atomicity is `(r0==1 && r2==1) -> (r1==1 || r3==1)`, as explained in Sect. 3. We used *acquire loads* in the IRIW program, which prohibit load-load reordering because the IRIW program has curious behaviors with non-multi-copy atomicity even if load-load reordering is prohibited.

We increase the number of writer threads in the IRIW because memory optimization should be more effective when the number of threads whose buffers

Table 1. Experimental results of the IRIW programs

# of Ws	Multi-copy atomicity			Non-multi-copy atomicity			Memory sharing optimiz.		
	States (K)	Memory (MB)	Time (s)	States (K)	Memory (MB)	Time (s)	States (K)	Memory (MB)	Time (s)
2	1	2	0.02	2	3	0.08	1	2	0.05
3	3	3	0.16	35	9	2.06	10	4	0.54
4	17	5	1.15	718	142	63.91	79	17	5.83
5	92	20	12.14	16650	3339	2526.31	666	125	84.07
6	463	87	71.00	407439	80912	88382.43	5100	941	847.56

are shared is larger. The number of writer threads in the original IRIW program is two. Additionally, we conducted model checking with the IRIW programs with three to six writer threads. The additional writer threads wrote integer 1 to additional shared variables. The two reader threads read additional values from the shared variables. For example, in a case that the number of writer threads is 3, the program is `(r0=z; r1=y; r2=x)` || `(r3=x; r4=y; r5=z)` || `x=1` || `y=1` || `z=1`. The number of interleavings should have increased drastically.

Table 1 presents the experimentally obtained results for the IRIW programs. The first column shows the number of writer threads (denoted by Ws). The second, third, and fourth columns refer to multi-copy atomicity. Model checking with multi-copy atomicity was conducted to represent the difficulty of model checking with non-multi-copy atomicity. Of course, model checking with multi-copy atomicity does not ensure the correctness of programs with non-multi-copy atomicity. There might exist a counterexample with non-multi-copy atomicity even if model checking with multi-copy atomicity detects no counterexample.

The second, third, and fourth columns list the numbers of states visited, memory consumed, and time elapsed, respectively. Even if a counterexample was detected, model checking continued until an exhaustive search was complete. The command-line option of VeriDAG includes `-c0`, which denotes that an exhaustive search is complete even if a counterexample is detected and printed out, whereas the default is `-c1`, which denotes that if a counterexample is detected, model checking stops and returns the counterexample. Similarly, the fifth, sixth, and seventh columns refer to non-multi-copy atomicity, and the eighth, ninth, and tenth columns refer to memory sharing optimization.

Model checking with multi-copy atomicity was completed more rapidly than the others with non-multi-copy atomicity and memory sharing optimization. However, they printed out no counterexamples because the assertion `(r0==1 && r1==1) -> (r2==1 || r3==1)` holds with multi-copy atomicity when the IRIW program `(r0=y; r1=x)` || `(r2=x; r3=y)` || `x=1` || `y=1` is complete. They are just experiments to compare model checking with multi-copy atomicity with model checking with non-multi-copy atomicity.

The numbers of states increased drastically during model checking with non-multi-copy atomicity. Accordingly, the consumed memories and elapsed times

Table 2. Experimental results of the CCGC algorithms

CCGC algorithm	Non-multi-copy atomicity			Memory sharing optimization		
	States (K)	Memory (MB)	Time (s)	States (K)	Memory (MB)	Time (s)
Schism	1744	320	205.17	1131	210	128.80
Sapphire	159052	28576	22312.51	41926	7589	5503.61

increased. Memory sharing optimization mitigated the state explosion problem. The greater the number of writer threads, that is, the larger the shared buffer, the more effective it is because of the combinatorial explosion.

Memory sharing optimization mitigated the state explosion problem. When the number of writer threads was two, memory sharing optimization improved performance approximately twice. When the number of writer threads was six, memory sharing optimization improved performance by 79–105 times. Thus, we confirmed that memory sharing optimization was more effective when the number of threads whose memories were shared was larger.

7.2 Concurrent Copying Garbage Collection Algorithms

Some CCGC algorithms are typical LDRF programs. CCGCs consist of mutators, which correspond to threads in user programs, and a collector. We can assume that there exists no load-store race in any period between synchronization points because threads in user programs that do not exist at the garbage collection layer must be DRF in programming languages with MCMs that require DRF. However, it may be the case that mutators share variables because a collector communicates with mutators when collecting objects that have not been used.

We conducted model checking of popular CCGC algorithms [15] that were modeled in the paper [27]. We extended the models to those whose mutators consisted of two threads, whereas the number of mutators in the original models was one. The original models have restrictions that their behaviors are fixed to read-write or write-read flows. Details of the restrictions are in the paper [27]. We modified the models to include both the flows by adding non-deterministic choice statements to the models. The modified models are more realistic than the original models. Table 2 presents the experimentally obtained results of the CCGC algorithms. The first column shows the names of the CCGC algorithms. The remaining columns are similar to those in Table 1.

Table 2 shows that the experimental results for Schism [22] and Sapphire [24], even those that were real applications, had a similar feature to that for the IRIW program, which is a litmus test for multi-copy-atomicity. The experiment for the other larger CCGCs was not complete in half a day ($= 43,200$ s).

8 Conclusion, Discussion, and Future Work

In this paper, we provided small-step operational semantics for relaxed memory consistency models with store buffers, defined observable equivalence on the semantics, and proposed LDRF, a local property of concurrent programs, which runs on non-multi-copy atomic MCMs. LDRF includes the IRIW programs that DRF cannot include. We also introduced memory sharing optimization on model checking of LDRF programs, and demonstrated that memory sharing optimization mitigates state explosion problems with non-multi-copy atomic MCMs.

Multi-copy atomicity, also known as remote write atomicity, has been studied [6,13]. Although ARM adopted non-multi-copy atomicity until ARMv8 [7], Pulte et al. reported that ARMv8 will be revised to prohibit non-multi-copy atomicity because they claim that the complexity derived from non-multi-copy atomicity has a cost, particularly for architectures implemented by multiple vendors [23]. In contrast, Vafeiadis argued that multi-copy atomicity did not seem relevant because its enforcement of global orders between instructions prevents scalability of verification [28]. Thus, multi-copy atomicity is a topic of debate. We hope that the present paper helps to elucidate non-multi-copy atomicity, and therefore contributes to decisions to adopt various multi-copy atomicities.

The experimental results demonstrated that the performance of VeriDAG with the memory sharing optimization was not substantially high. This is because VeriDAG is a stateful model checker that adopts classical partial order reduction optimization [21], differently from Nidhugg [1] with the POWER MCM [12] and RCMC [16] with the repaired version of C/C++11 MCM [14] which adopt dynamic partial order reduction [10]. However, this is independent of the goal of demonstrating the effectiveness of memory sharing optimization.

Because TRF and LDRF are variants of DRF in different directions, it might be expected that a novel property can be defined by combining the two properties. However, this appears to be difficult because TRF is strongly specific to x86-TSO, which preserves the order of stores of different shared variables, whereas LDRF requires that store buffers consist of multiple queues.

This work has a few limitations. LDRF is syntactically and conservatively defined while DRF is defined by observing execution traces, because local data race detection has not been implemented yet.

It seems reasonable that LDRF should be defined as a pair-wise property because the locality may be a pair-wise property. For example, let us consider a case that threads 0 and 1 do not race, threads 0 and 2 do not race, but threads 1 and 2 do. However, LDRF in this paper cannot capture the case since LDRF is defined as a property for a partition of the set of threads.

The memory sharing optimization that mitigates the state explosion problem is not a scalable method for increasing the number of threads. We would like to revise LDRF as a pair-wise property, find any better characterization for scalability, and provide local data race detection.

Acknowledgments. The author thanks Toshiyuki Maeda and Tomoharu Ugawa. The motivation of this work was fostered through discussions with them. The author also thanks the anonymous reviewers for several comments to improve the paper. This work was supported by JSPS KAKENHI Grant Number 16K21335.

References

1. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8
2. Abe, T.: A verifier of directed acyclic graphs for model checking with memory consistency models. Hardware and Software: Verification and Testing. LNCS, vol. 10629, pp. 51–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_4
3. Abe, T., Maeda, T.: Observation-based concurrent program logic for relaxed memory consistency models. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 63–84. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_4
4. Abe, T., Maeda, T.: Concurrent program logic for relaxed memory consistency models with dependencies across loop iterations. J. Inf. Process. **25**, 244–255 (2017)
5. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer **29**(12), 66–76 (1996)
6. ARM Limited: ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition) (2012)
7. ARM Limited: ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (2017)
8. Basten, T., Bošnački, D., Geilen, M.: Cluster-based partial-order reduction. Autom. Softw. Eng. **11**(4), 365–402 (2004)
9. Dolan, S., Sivaramakrishnan, K., Madhavapeddy, A.: Bounding data races in space and time. In: Proceedings of PLDI (2018, to appear)
10. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of POPL, pp. 110–121 (2005)
11. Gao, G.R., Sarkar, V.: Location consistency-a new memory model and cache consistency protocol. IEEE Trans. Comput. **49**(8), 798–813 (2000)
12. IBM Corporation: Power ISA Version 3.0 (2015)
13. Intel Corporation: A Formal Specification of Intel Itanium Processor Family Memory Ordering (2002)
14. ISO/IEC 14882:2011: Programming Language C++ (2011)
15. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook. CRC Press, Boca Rotan (2012)
16. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL:17), 1–32 (2018)
17. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **c-28**(9), 690–691 (1979)
18. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
19. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_23

20. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
21. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_34
22. Pizlo, F., Ziarek, L., Maj, P., Hosking, A.L., Blanton, E., Vitek, J.: Schism: fragmentation-tolerant real-time garbage collection. In: Proceedings of PLDI, pp. 146–159 (2010)
23. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL:19), 1–29 (2018)
24. Ritson, C.G., Ugawa, T., Jones, R.: Exploring garbage collection with Haswell hardware transactional memory. In: Proceedings of ISMM, pp. 105–115 (2014)
25. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: Proceedings of PPoPP, pp. 161–172 (2007)
26. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of PLDI, pp. 175–186 (2011)
27. Ugawa, T., Abe, T., Maeda, T.: Model checking copy phases of concurrent copying garbage collection with various memory models. Proc. ACM Program. Lang. **1**(OOPSLA:53), 1–26 (2017)
28. Vafeiadis, V.: Sequential consistency considered harmful. In: New Challenges in Parallelism (Report from Dagstuhl Seminar 17451), p. 21 (2018)