令和 3 年度

学士学位論文

不揮発性メモリ向けオブジェクト永続化アルゴリズムのモデル検査器 Spin を用いた検証

Verification of an Algorithm for Persistent Object in Non-Volatile Memory Using Spin Model Checker

1245115　　飯干 寛幸

指導教員　　高田 喜朗

2022 年 2 月 28 日

高知工科大学 情報学群

要 旨

不揮発性メモリ向けオブジェクト永続化アルゴリズムのモデル検査器 Spin を用いた検証

飯干 寛幸

近年，DRAM と同じようにバイト単位のアクセスが可能で，かつ計算機の電源が失われてもデータが残る不揮発性メモリが注目されている．不揮発性メモリを使ったシステムではメモリへ書き込むだけでデータを永続化できる．ただし，不揮発性メモリはキャッシュを介してアクセスすることに注意が必要である．キャッシュが書き戻されるタイミングはプログラマには分からないので，データの整合性を保つためには，キャッシュラインを書き戻す命令を使ってプログラマが永続化順序を制御する必要がある．

プログラマの負担を軽減するために，Java のようなマネージド言語でランタイムシステムが代わりに永続化順序を管理するシステムが研究されている．そのようなシステムでは，全てのデータを永続化するのではなく，オブジェクト単位で選択的にデータを永続化する．オブジェクト永続化アルゴリズムは，Java のオブジェクトを永続化するためのランタイムシステムのアルゴリズムである．一般に，DRAM 上のオブジェクトを不揮発性メモリ上へコピーすることでオブジェクトを永続化する．DRAM 上のオブジェクトを揮発性コピー，不揮発性メモリ上のオブジェクトを永続化コピーと呼ぶ．オブジェクトの永続化中にユーザプログラムがオブジェクトの値を書き換える可能性があるため，オブジェクト永続化アルゴリズムには適切な排他制御が必要であり，作るのが難しい．オブジェクト永続化アルゴリズムの正当性を確かめるために，形式的手法を使った検証は有用である．モデル検査器 Spin は，プログラムを抽象化したモデルに基づいて，特定の性質がモデルのあらゆる実行経路で成り立っているか網羅的に調べるツールである．本研究では，オブジェクト永続化アルゴリズム

の一つである複製による永続化のアルゴリズムの正当性を，モデル検査器 Spin を使った検査で確かめる．

複製による永続化では，static 変数の中から永続化する変数の集合をプログラマが指定する．この変数の集合を durable root という．そして，durable root から参照を辿って到達可能なオブジェクトをランタイムシステムが永続化する．

複製による永続化が満たすべき性質は 2 つあり，(1) 書き込み途中のオブジェクトを除き揮発性コピーの値と永続化コピーの値が一致しており，(2)durable root から参照を辿って到達可能なオブジェクトはすべて永続化コピーを持つ．これらの性質は，オブジェクトの参照が書き換えられたときに満たされなくなる可能性がある．そこで，複製による永続化アルゴリズムのうち，参照の書き換えとオブジェクトの永続化のモデルを作成した．本研究で作成したモデルでは，特定のメモリの状態から実行を始めて，複数スレッドでアルゴリズムに従って任意のオブジェクトへ参照を書き込む．そして，実行開始時点のメモリの状態を変えて検査することで，任意の実行経路を調べる．DRAM と不揮発性メモリの値が常に上記の性質を満たした状態になっていれば，アルゴリズムは正しい．

モデル検査を使って，現実的な検査時間やメモリ使用量で調べられる実行経路の数は有限なので，オブジェクトの数やユーザスレッドの数といったパラメータの制限が存在する．すべての実行経路を 1 回のモデル検査で調べようとすると，一度にメモリを大量に消費するため，小さなパラメータでしか検査できない．できる限りパラメータを大きくとるために，調べる実行経路を開始状態によって分割した．

本研究で作成したモデルを使って，パラメータをオブジェクト数が 5 つ，スレッド数が 2，スレッドごとの書き込み回数を 1 回として検査した．その結果，本研究で設定したパラメータの範囲内で，複製による永続化のアルゴリズムは 2 つの性質を満たしていることがわかった．

キーワード　並行アルゴリズム，モデル検査，オブジェクト永続化，直交永続性，到達可能性に基づく永続化，複製に基づく永続化

Abstract

Verification of an Algorithm for Persistent Object in Non-Volatile Memory Using Spin Model Checker

Hiroyuki Iiboshi

Emerging a non-volatile memory is byte-addressable and retains data even if a computer power is lost. A system using non-volatile memory can persist data simply by writing data to the memory. However, it should be noted that non-volatile memory is accessed through a volatile cache. Since a programmer can not know when the cache is written back, the programmer must control a persistence order by using instructions to write-back cache lines to keep data consistency.

In order to save the programmer's labor, some researchers are developing a runtime system of a managed language such as Java which manages a persistence order. In such a system, instead of persisting all data, the system persists data selectively on an object-by-object basis. An object persistence algorithm is an algorithm for the runtime system to persist Java objects. Generally, the system using the object persistence algorithm makes an object persistent by copying the object on DRAM to non-volatile memory. Objects on DRAM are called volatile copies, and objects on non-volatile memory are called persistent copies. Since a user program may rewrite a value of an object during persisting of the object, the object persistence algorithm requires appropriate exclusive control and is difficult to develop. Verification using formal methods is useful for verifying the correctness of the object persistence algorithm. Model checker Spin comprehensively checks whether a specific property holds in all execution paths of a model,

which abstracts a program. In this paper, we confirm the validity of the persistence by replication, which is one of the object persistence algorithms, by using the Spin.

In persistence by replication, the programmer specifies a set of variables to be persisted from static variables. This set of variables is called durable root. Then, the runtime system persists objects reachable by following a reference from the durable root.

There are two properties that the persistence by replication must satisfy: (1) a value of the volatile copy and a value of the persistent copy match except for an object being written, and (2) every object reachable by following the reference from the durable root has a persistent copy. These properties may not be satisfied when references to the object are rewritten. Therefore, we developed a model of reference rewriting and object persistence of the persistence by replication algorithm. In the model developed in this paper, execution is started from a specific memory state, and multiple threads write a reference to an arbitrary object according to the persistence by replication algorithm. We check an arbitrary execution path by changing a state of the memory at the start of execution. At any point of execution, if the values of DRAM and non-volatile memory always satisfy the above properties, the algorithm is correct.

Since the model checker can check a finite number of execution paths with realistic check time and memory usage, there are restrictions on parameters such as the number of objects and the number of user threads. If we try to check all execution paths with one model checking, it consumes a large amount of memory, so we can check with only small parameters. In order to take the parameters as large as possible, we divide the execution paths by a start state of the execution path.

Using the model we developed, we conducted model checking for the persistence by replication algorithm with the number of objects was 5, the number of threads was 2, and the number of writes per thread was 1. As a result, it was found that the persistence by replication algorithm satisfies two properties within the range of the parameters set

in this paper.

key words concurrent algorithm, model checking, persistent object, orthogonal persistency, persistence by reachability, persistence by replication

目次

目次

図目次

第 1 章

はじめに

近年，計算機の電源が失われてもデータが残るメモリとして不揮発性メモリが注目されている．不揮発性メモリにデータを書き込んでデータを保存することを**永続化する**という．不揮発性メモリは，DRAM と同じように高速なバイト単位でのアクセスが可能という特徴がある．そのため，不揮発性メモリを使ったシステムは単にメモリへ書き込むだけでデータを永続化することができる．

ただし，不揮発性メモリはキャッシュを介してアクセスすることに注意が必要である．キャッシュが書き戻されるタイミングはプログラマには分からない．そのため，プログラマの意図とは異なる順序でデータが永続化される可能性があり，データの整合性を保つためにはプログラマがキャッシュラインを書き戻す命令を使って永続化順序を制御する必要がある．

プログラマの負担を軽減するために，Java のようなマネージド言語でランタイムシステムが代わりに永続化順序を管理するシステムが研究されている [1, 2, 3]．そのようなシステムでは，全てのデータを永続化するのではなく，オブジェクト単位で選択的にデータを永続化する．オブジェクトは作られた時は永続化されていない．その一部が実行中に永続化され，以降，永続化されたオブジェクトへの書込みも不揮発性メモリに保存される．

オブジェクト永続化アルゴリズムは，Java のランタイムシステムが Java オブジェクトを永続化するためのアルゴリズムである．オブジェクトの永続化とユーザプログラムによるオブジェクトへの書き込みが並行に行われるため適切な排他制御が必要であり，作るのが難しい．そこで，形式的手法を使った検証によってオブジェクト永続化アルゴリズムの正しさを確かめることが考えられる．モデル検査器 Spin[4] は，プログラムを抽象化したモデルに基づいて，特定の性質がモデルのあらゆる実行経路で成り立っているか網羅的に調べるツール

であり，並行プログラムの検査にも使われる．

本研究では，オブジェクト永続化アルゴリズムの一つである複製による永続化 [1] のアルゴリズムが正しいことを，Spin を使った検査で確かめる．簡単のため，本研究ではキャッシュによる永続化順序の入れ替わりは考慮しない．それでも，オブジェクトの永続化とオブジェクトへの書き込みを並行に行うことは十分に難しい．Spin を使った検査のために，複製による永続化のアルゴリズムが満たすべき性質を定義し，Spin 用のモデルを作成した．現実的な時間で Spin が調べられる状態数には限りがあるので，オブジェクト数やスレッド数にパラメータ制限が必要である．そこで，より大きなパラメータを使った検査を行うために，複製による永続化のアルゴリズムが持つ性質を利用して調べる状態数を減らす方法を考案した．

本論文の構成は以下のとおりである．まず，本論文の理解に必要な事前知識について述べる（2 章）．次に，検査の対象である複製による永続化のアルゴリズムについて説明し（3 章），Spin 用のモデル記述言語 Promela を使って複製による永続化のアルゴリズムのモデルを作成する（4 章）．そして，調べる状態数を減らす方法について説明し，状態数の削減が検査結果に影響しないことを形式的に証明する（5 章）．その後，検査の結果について述べる（6 章）．

第 2 章

事前知識

本章では，永続化アルゴリズムとモデル検査の内容について説明するための事前知識について説明する．

2.1 Spin

Spin[4] はモデル検査器の一つである．モデルと呼ばれる検査対象システムの仕様とシステムで成り立って欲しい性質を入力とする．システムの仕様から，システムがとりうる状態を節点，状態遷移を辺とするグラフを作成して，グラフの全ての節点において性質が成り立つか調べる．Spin のモデルは，Spin 用モデル記述言語である Promela を使用して記述する．Promela は C 言語に似た文法を持ち，関数はインライン関数のみが記述できる．また，if 文や，while 文に相当する do 文によって制御構造を記述する．これらの文は条件を複数記述することができ，真になる条件が複数ある場合は非決定的な分岐となる．非決定的な分岐は，可能な分岐先の状態を全て探索する．性質は assertion の形で与えることができ，assertion を通過しうる全ての状態が assertion 違反でないか調べる．

2.2 メモリアクセス命令の入れ替え

近代的な CPU では，メモリアクセス命令の実行を最適化するためにプログラムと異なる順番で命令を実行する場合がある．例えば，Intel の x86 アーキテクチャでは依存関係のない書き込み命令を読み出し命令が追い越す可能性がある [5]．ここでいう依存関係がないとは，読み出すメモリアドレスと書き込むメモリアドレスが異なることをいう．図 2.1 にメ

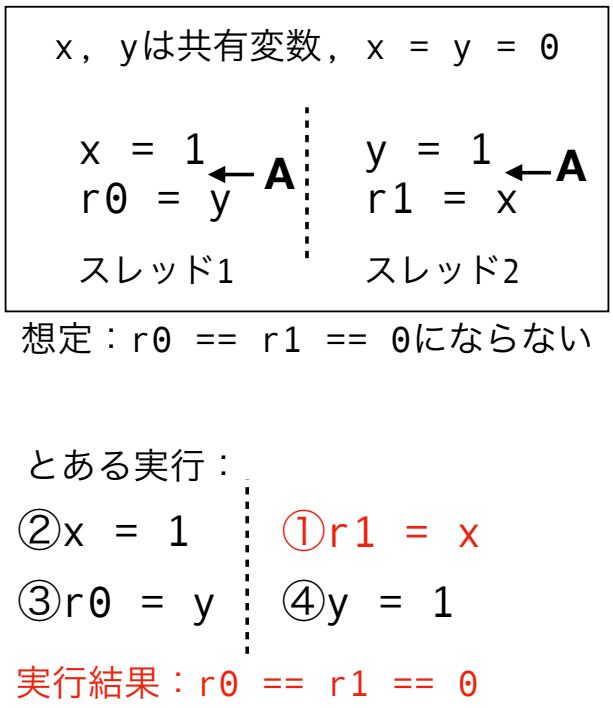

図 2.1　メモリアクセス命令の入れ替えによって問題が起きる例

モリアクセス命令の入れ替えによって問題が起きる例を示す．図上部のようなプログラム `x = 1; r0 = y || y = 1; r1 = x` を実行する場合について考える．`x` と `y` は共有変数で初期値は 0，`r0`，`r1` はローカル変数である．プログラムが書いてある順番に実行されて，`r0`，`r1` のいずれかには 1 が書き込まれることを期待して作られている．メモリアクセス命令が入れ替えられる環境で実行したとき，図下部のように丸付き数字の順に実行されたとする．すると，`r0`，`r1` 両方が 0 になってしまう．

フェンス命令は，フェンス命令より前のメモリアクセス命令がフェンス命令に続くメモリアクセス命令よりも先に実行されることを保証する命令である．図 2.1 の例では，A の位置にフェンス命令を挿入することでメモリアクセス命令の入れ替えを防ぐことができる．

さらに，x86 アーキテクチャではキャッシュへの書き込みが書き込まれた順に不揮発性メモリに反映されるとは限らない．本研究では，書き込みと永続化の並行処理について主に調べるため，キャッシュへの書き込みは書き込まれた順に反映されるものとする．

```
1 int x, y;
2 inline add_to_x(num) {
3     x = y + num;
4 }
```

```
1 #VARSIZE 2
2 #define x 1
3 #define y 2
4 inline add_to_x(num) {
5         int tmp = READ(y);
6     WRITE(x, tmp + num);
7 }
```

図 2.2 MMLib を使用したモデルの例（左：Promela モデル，右：MMLib モデル）

2.3 MMLib

Spin のみで，メモリアクセス命令の実行順序の入れ替えを考慮して検査するには，入れ替えが起こった場合と起こらなかった場合のいずれか一方を非決定的に実行するように，メモリアクセス命令のたびに記述する必要がある．そのため，本研究ではメモリアクセス命令の入れ替えを考慮した検査を行うための Spin 用ライブラリである MMLib[6] を使って検査する．MMLib を使ったモデルでは，スレッド間で共有する変数を MMLib の管理下に置き，専用の読み書き命令を使ってアクセスする．これによって，メモリアクセス命令の実行順序が入れ替わった場合を調べることができる．MMLib を使ったモデルの例を図 2.2 に示す．まず，使用する共有変数の数を VARSIZE として指定する．MMLib で使える命令には，共有変数に書き込む WRITE，共有変数から読み出す READ，フェンス命令である FENCE が存在する．MMLib では，入れ替えうる書き込み命令の最大数をパラメータ（BUFFSIZE）として指定する必要がある．検査の精度に影響するので，できる限り大きくすることが望ましい．

CPU によって入れ替わる可能性があるメモリアクセス命令のパターンは異なり，MMLib では 3 つのパターンから 1 つ指定することができる．そのうち，sequential consistency（SC）は全く入れ替わりが起きないパターンで，total store order（TSO）は Intel x86 アーキテクチャのパターンである．

第 3 章

オブジェクト永続化アルゴリズム

本章では，複製による永続化のアルゴリズムについて説明し，満たすべき性質について述べる．

3.1 永続化アルゴリズム

3.1.1 オブジェクトの永続化方法

複製による永続化のアルゴリズムは，不揮発性メモリ上にオブジェクトのコピーを作成することでオブジェクトを永続化する．以後 DRAM 上のオブジェクトを揮発性コピー，不揮発性メモリ上のオブジェクトを永続化コピーとよぶ．揮発性コピーは自身の永続化コピーへの参照を持つ．この参照を forwarding ポインタと呼ぶ．オブジェクトからの読み出しは揮発性コピーから行い，オブジェクトへの書き込みは揮発性コピーと永続化コピーの両方に行う．これにより，時間のかかる不揮発性メモリへのアクセスを減らすことができる．

3.1.2 永続化するオブジェクトの選択

不揮発性メモリへの書き込みは DRAM に比べると遅く，キャッシュラインを書き戻す時間まで含めると最大で 588 倍かかる [7]．すべてのオブジェクトを永続化すると，不揮発性メモリへのアクセスが多くなり実行が遅くなってしまう．そのため，オブジェクトを永続化するランタイムシステムでは一部のオブジェクトだけを永続化するアプローチが一般的である．複製による永続化のアルゴリズムは，**durable root** と呼ばれる静的フィールドの集合

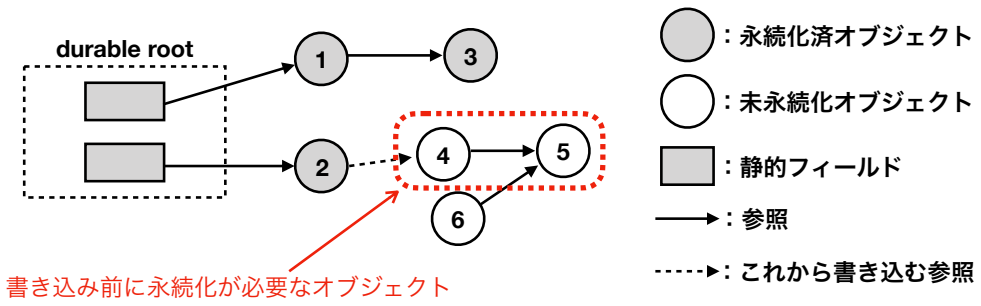

図 3.1　参照の書き換えによって永続化が必要になる例

から参照を辿って到達可能なオブジェクトをすべて永続化する方式を採用している．この方式では，プログラマは static 変数の中から durable root を指定し，ランタイムシステムは durable root から到達可能なオブジェクトがすべて永続化されていることを保証する．すると，計算機の電源が失われたときに，durable root から到達可能なオブジェクトの参照先が失われない．

プログラムの実行中にオブジェクトの参照関係が書き換えられると，永続化されたオブジェクトから永続化されていないオブジェクトへの参照ができる可能性がある．そのため，永続化されたオブジェクトへ参照の書き込みがあるたびに，新たに参照先になるオブジェクトを書き込みに先んじて永続化する必要がある．図 3.1 に参照を書き換える例を示す．durable root に 2 つ静的フィールドが含まれており，それぞれが別のオブジェクトを参照している．durable root から参照されているオブジェクト 1, 2，オブジェクト 1 から参照されているオブジェクト 3 は durable root から到達できるので永続化されている．さらに，永続化されているオブジェクト 2 からオブジェクト 4 への参照を書き込もうとする場合，オブジェクト 4 だけでなくオブジェクト 4 から参照されているオブジェクト 5 も参照を書き込む前に永続化する必要がある．

オブジェクト 4，5 のように，参照を書き込むときに永続化が必要になるオブジェクトの集合を定義することができる．このような集合を，参照を書き込もうとしているスレッドの**永続化対象集合**と呼ぶ．複製による永続化のアルゴリズムでは，書き込みを行うスレッドが書き込みに先んじて永続化対象集合のオブジェクトを永続化する．

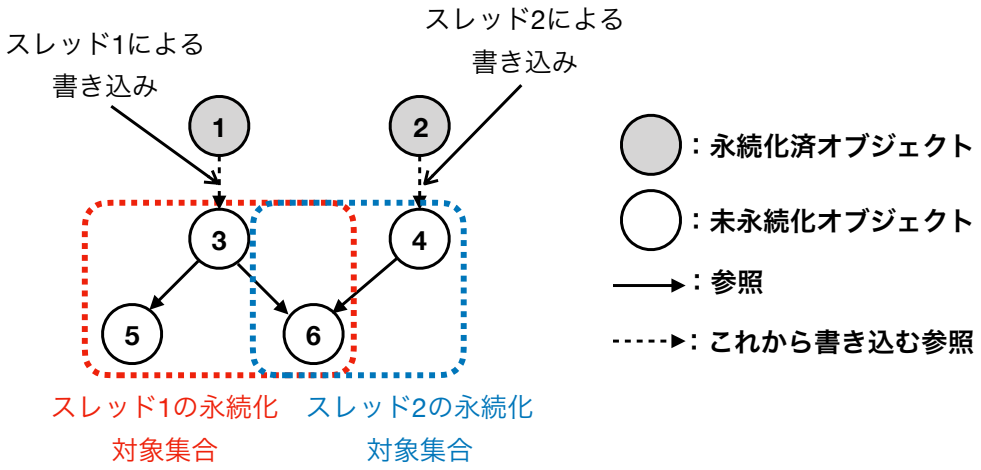

図 3.2　永続化対象集合に共通部分がある例

3.1.3　干渉スレッドグループ

複数のスレッドが同じオブジェクトを永続化しようとしたとき，永続化対象集合に共通部分ができる．永続化対象集合に共通部分が存在するスレッドの集合を**干渉スレッドグループ**（ITG）と呼ぶ．永続化対象集合の共通部分に属するオブジェクトは，ITG 内のいずれかのスレッドが責任を持って永続化する必要がある．そして，同じ ITG に属する他のスレッドは永続化が完了するまで書き込みを待たなければならない．さもなくば，永続化されたオブジェクトから永続化されていないオブジェクトへの参照ができてしまう．あるオブジェクトの永続化責任を負うスレッドを，そのオブジェクトの**永続化担当スレッド**と呼ぶ．例を図 3.2 に示す．スレッド 1，2 はともに永続化済みのオブジェクト 1, 2 から未永続化オブジェクト 3, 4 への参照をそれぞれ書き込もうとしている．このとき，スレッド 1 の永続化対象集合はオブジェクト 3, 5, 6 であり，スレッド 2 の永続化対象集合はオブジェクト 4, 6 である．このような状況では，スレッド 1, 2 のいずれかがオブジェクト 6 の永続化を担当し，もう一方のスレッドはオブジェクト 6 の永続化が完了するまで待つ．

3.1.4　発生しうる競合

複数のスレッドが同時に同じオブジェクトに対してメモリアクセスを行い，そのうち少なくとも 1 つが書き込みであるような状況を**競合**が起きたと表現する．複製による永続化のア

ルゴリズムは，ユーザプログラムが競合を起こさないことを仮定している．つまり，競合を起こさないようにプログラムを作るのはプログラマの責任である．

一方で，オブジェクトの永続化はランタイムシステムによって行われるので，永続化が関係する競合をプログラマが避けることはできない．そのため，永続化が関係する競合はランタイム側で解決する必要がある．永続化が関係する競合は次の 3 種類ある．

1. あるスレッドが forwarding ポインタを書き込んでいる最中のオブジェクトに，別のスレッドが書き込む
2. あるスレッドが揮発性コピーの値を永続化コピーに写している最中のオブジェクトに，別のスレッドが書き込む
3. 複数のスレッド間で永続化対象集合に共通部分がある

競合の種類によって，発生する問題が異なる．1. の競合について，オブジェクトに値を書き込むとき，forwarding ポインタを持つオブジェクトには永続化コピーが存在するので，揮発性コピーと永続化コピーの両方に書き込む必要がある．一方で，forwarding ポインタを持たないオブジェクトは永続化コピーが存在しないので，揮発性コピーにのみ書き込む．書き込みを行うスレッドが forwarding ポインタがないオブジェクトであると判断した後に別のスレッドが forwarding ポインタを書き込んだ場合，forwarding ポインタを持つオブジェクトであるにもかかわらず揮発性コピーにのみ値が書き込まれてしまう．

2. の競合について，揮発性コピーから値を読み出して永続化コピーに書き込むまでの間にはタイムラグが存在する．その間に新たな揮発性コピーと永続化コピーの値が書き込まれると，書き込み以前に読み出していた古い値で永続化コピーが上書きされてしまう．

3. の競合について，共通部分に属するオブジェクトの永続化が完了する前に，他のオブジェクトの永続化を終えたスレッドが参照を書き込んでしまうと，永続化されたオブジェクトから永続化されていないオブジェクトへの参照が作られてしまう．そのため，ITG に属するスレッド全体で，共通部分に属するオブジェクトが全て永続化されたことを確認する必要がある．

永続化が関係する競合の解決方法については 3.1.6 節で述べる．

3.1.5　参照書き込みのアルゴリズム

Java では，オブジェクトや静的フィールドへ書き込む際に putfield という関数を内部で用いる．本節では，複製による永続化における putfield 関数のアルゴリズムについて説明する．なお，キャッシュへの書き込みは書き込まれた順に不揮発性メモリに反映されると仮定したため，キャッシュラインを書き戻す命令（CLWB）についての説明は省略する．また，書き込みやキャッシュラインを書き戻す命令にだけ影響を及ぼすフェンス命令（SFENCE）は，キャッシュラインを書き戻す命令の実行順を制御するために使われているので同じく説明を省略する．

ここで，CAS 命令は第 1 引数の参照先の値と第 2 引数の値を比較し，それらが等しければ第 1 引数の参照先に第 3 引数の値を書き込む命令である．比較と書き込みの間に他スレッドによって第 1 引数の参照先が書き換えられることはない．書き換えに失敗した場合は第 1 引数の参照先の値をそのまま返し，成功した場合は書き換え前の第 1 引数の参照先の値を返す．

図 3.3 に参照を書き込むときに用いる putfield 関数のアルゴリズムを示す．o は書き込み対象のオブジェクト，f は書き込み対象のフィールド，v は書き込む参照である．VCopy は揮発性コピー，PCopy は永続化コピーを表す．永続化コピーのないオブジェクトに書き込むときには，forwarding ポインタの領域（図中の pcopy）を CAS 命令で BUSY という特別な値に置き換える（3–5 行目）．CAS 命令が成功した場合，o の forwarding ポインタは元々NULL だったということなので，揮発性コピーのみに書き込む（6 行目）．そして，書き込みを終えてから forwarding ポインタを NULL に戻す．CAS 命令が失敗した場合，o は forwarding ポインタを持ち，既に永続化されているか，永続化処理の途中である．いずれにせよ，o の参照先となる v も永続化する必要がある．ensure_recoverable 関数はオブジェクトを永続化する処理を行う関数であり，詳細は 3.1.6 節で述べる（10 行目）．その後，揮発性コピーと永続化コピーの両方に書き込む（12–13 行目）．

```
1  putfield(VCopy o, Field f, VCopy v) {
2      PCopy on;
3      while ((on = CAS(&o.pcopy, NULL, BUSY)) == BUSY);
4      if (on == NULL) {
5          o[f] = v;
6          o.pcopy = NULL;
7          return;
8      }
9      ensure_recoverable(v);
10     PCopy vn = v.pcopy;
11     o[f] = v;
12     on[f] = vn;
13     SFENCE;
14 }
```

図 3.3 putfield のアルゴリズム（[1] より引用，一部省略）

3.1.6 オブジェクト永続化のアルゴリズム

オブジェクトの永続化処理は，(1) 不揮発性メモリ上に永続化コピー用の領域を確保して forwarding ポインタに書き込む，(2) 揮発性コピーの値を永続化コピーにコピーする，の 2 つに分けることができる．(1) は shade 関数によって，(2) は ensure_recoverable 関数によって行う．さらに，これら 2 つの関数で ITG の処理も行う．

ensure_recoverable 関数のアルゴリズムを図 3.4 に示す．r が永続化対象となるオブジェクトである．r から参照を辿って到達できるオブジェクトを全て永続化する必要があるので，永続化が必要なオブジェクトを worklist で管理する．worklist 内のオブジェクトから参照されている永続化前のオブジェクトを worklist に追加する，ということを繰り返せば到達可能なオブジェクトで永続化が必要なものを網羅できる（12 行目）．worklist 内のオブジェクトは永続化処理を始めるときに worklist から削除される（5 行目）．worklist 内のオブジェクトが全て永続化され，空になったらそれ以上新たなオブジェクトを永続化する必要はない（4 行目）．

9–14 行目では o から参照されているオブジェクトを worklist へ追加するとともに，o

の揮発性コピーから永続化コピーへ値を写している．この間に他スレッドによって o の値が書き換えられる可能性がある（2. の競合）．そのため 18–20 行目の処理によって揮発性コピーの値と永続化コピーの値が一致することを確かめている．一致しなかった場合は，一致するまで値を写し直す．

15 行目の MFENCE はすべての種類のメモリアクセス命令に影響するフェンス命令である．もしここにフェンス命令がなかったとすると，x86 アーキテクチャにおいて，揮発性コピーの値を永続化コピーへ写す書き込みを，揮発性コピーの値と永続化コピーの値との比較が追い越す可能性がある．つまり，永続化コピーへの値のコピーが完了する前に，コピーが無事に完了したか確認し始めてしまう．

SYNC 関数については，shade 関数について述べた後で説明する．

shade 関数のアルゴリズムを図 3.5 に示す．o が永続化コピーをつくる対象となるオブジェクトである．まず不揮発性メモリ上にオブジェクトと同じ大きさの領域を確保しておき，CAS 命令を使って forwarding ポインタを書き換える（4 行目）．forwarding ポインタの値が BUSY だった場合は，BUSY でなくなるまで待機する．forwarding ポインタが BUSY になるのは，putfield 関数が揮発性コピーのみに値を書き込んでいるときである（1. の競合）．BUSY でなくなるまで待機することで，forwarding ポインタの書き込みと値の書き込みが競合を起こさなくなる．また，forwarding ポインタが BUSY でも NULL でもない場合は，他のスレッドか過去の自分自身が forwarding ポインタに書き込み終えている．複数のスレッドが同時に forwarding ポインタを書き込もうとした場合，CAS 命令を使っているため必ずどちらか一方が失敗する．永続化コピーの作成に成功したスレッドが永続化担当スレッドになる．

7–14 行目は既に永続化コピーが作られていた場合の処理である．responsible_thread は永続化担当スレッドを取得する関数である（8 行目）．既に o の永続化が完了している場合は何もしない．responsible_thread によって自分以外のスレッドが得られた場合，他のスレッドが先に永続化コピーを作成している．この場合は，永続化コピーを作成したスレッドと ITG を併合する必要がある．responsible_thread によって自身のスレッドが得

```
1  ensure_recoverable(VCopy r) {
2      if (shade(r) == TRUE) {
3          worklist.add(r);
4          while (not worklist.empty()) {
5              VCopy o = worklist.remove();
6              PCopy on = o.pcopy;
7  RETRY:
8              /* copy */
9              for (Field f in o.fields) {
10                 VCopy p = o[f];
11                 if (shade(p) == TRUE)
12                     worklist.add(p);
13                 on[f] = p.pcopy;
14             }
15             MFENCE;
16             CLWB_RANGE(on, on.size());
17             /* verify */
18             foreach (Field f in o.fields)
19                 if (o[f].pcopy != on[f])
20                     goto RETRY;
21             /* complete */
22         }
23         SFENCE;
24     }
25     SYNC();
26 }
```

図 3.4 ensure_recoverable のアルゴリズム（[1] より引用，一部省略）

られた場合は，過去に 1 度同じオブジェクトに対して shade 関数を呼び出しているということを意味する．これは，永続化対象集合のオブジェクト間で参照のループができていることを意味する．既に shade 関数の実行が終わっているので，o は worklist に追加されているはずである．そのため，それ以上何かする必要はない．depends_on 関数は引数として渡されたスレッドと呼び出したスレッドの ITG を併合する（11 行目）．

ensure_recoverable 関数の SYNC 関数は同じ ITG に属するスレッド間でバリア同期を

```
1  shade(VCopy o) {
2      PCopy on = alloc_NVM(o.size());
3      PCopy fwd;
4      while ((fwd = CAS(&o.pcopy, NULL, on)) == BUSY);
5      if (fwd != NULL) {
6          t = responsible_thread(fwd);
7          if (t != current_thread) {
8              /* thread t is making o recoverable */
9              depends_on(t);
10         }
11         return FALSE;
12     }
13     return TRUE;
14 }
```

図 3.5 shade のアルゴリズム（[1] より引用，一部省略）

行う．SYNC が完了したときには，r と r から到達可能なすべてのオブジェクトの永続化が完了している．これにより，3. の競合が解決される．このとき，自身が永続化担当スレッドになっているオブジェクトはすべて永続化が完了しているので，永続化担当スレッドを表すフィールドをクリアする必要がある．SYNC が完了してから，永続化担当スレッドを表すフィールドをすべてクリアするまでの間に時間的な差があり，その間に他のスレッドが responsible_thread 関数を呼ぶ可能性がある．もし，このまま depends_on 関数によって SYNC を既に終えたスレッドとこれから SYNC しようとするスレッドが同じ ITG に入ってしまうと，SYNC を終えたスレッドがもう一度 SYNC を呼び出すまでもう一方のスレッドは待たされ続けてしまう．そのため，SYNC を終えているスレッドと ITG を併合しないように depends_on 関数を実装しなければならない．

3.1.7 永続化担当スレッドの実装

forwarding ポインタを書き込んだスレッドと永続化担当スレッドを一致させるために，永続化コピーには永続化担当スレッドを記録するためのフィールドを用意する．shade 関数

で永続化コピーを作成するとき，不揮発性メモリ上に永続化コピー用の領域を確保した時点で，永続化担当スレッドを記録するフィールドに自身の ID を書き込んでおく．すると，CAS を使った forwarding ポインタの書き込みに成功したとき，同時に永続化担当スレッドを自分にすることができる．

3.2　複製による永続化のアルゴリズムが満たすべき性質

複製による永続化では，durable root から参照を辿って到達可能なオブジェクトを永続化する．すなわち，永続化対象となるオブジェクトは，少なくとも 1 つの durable root または永続化されているオブジェクトから参照されている．もし，永続化対象となるオブジェクトの中に永続化されていないオブジェクトがあった場合，計算機の電源が失われたときそのオブジェクトは消えてしまう．すると，消えたオブジェクトへの参照は不正な値になり，不揮発性メモリ上の値が不整合な状態になる．ユーザが指定できるのは durable root だけなので，durable root から参照を辿って到達できるオブジェクトを永続化するのはランタイムの責任である．したがって，durable root から到達可能なオブジェクトがすべて永続化されていなければならない．

さらに，プログラム実行中にユーザプログラムがアクセスするのは揮発性コピーである一方で，計算機の電源が失われたときには永続化コピーのみが残る．もし，ユーザプログラムによる書き込みの一部が永続化コピーに反映されていなかった場合，永続化コピーには新しい値と古い値が混在することになる．よって，揮発性コピーと永続化コピーが同じ値に保たれていることが必要である．ただし，揮発性コピーと永続化コピーにまったく同時に書き込むことは不可能なので，書き込み対象となるオブジェクトには揮発性コピーと永続化コピーの値が異なる期間が存在する．複製による永続化のアルゴリズムでは競合のないプログラムを仮定しているので，書き込み中の値が読み出されることはない．そのため，書き込みが完了するまでは揮発性コピーと永続化コピーの両方が，書き込み前と書き込み後のいずれかの値であれば良い．

3.2　複製による永続化のアルゴリズムが満たすべき性質

以上をまとめると，複製による永続化が満たすべき性質は次の 2 つである．

- durable root から参照を辿って到達可能なオブジェクトはすべて永続化コピーを持つ
- 書き込み途中のオブジェクトを除いて，揮発性コピーの値と永続化コピーの値が一致する

第 4 章

アルゴリズムのモデル化

4.1 検査方針

本研究では，複製による永続化のアルゴリズムが 3.2 節で述べた 2 つの性質を満たしているか Spin を使って調べる．特に，メモリアクセス命令の入れ替えを考慮してもなお性質が満たされているか調べるために MMLib も使用する．これらの性質は，オブジェクトに参照が書き込まれたときに満たされなくなる可能性がある．例えば，durable root から参照を辿って到達可能なオブジェクトから，永続化されていないオブジェクトへの参照を書き込もうとした場合，参照を書き込む前に参照先のオブジェクトを永続化する必要がある．オブジェクトに参照を書き込む場合の正当性を調べるために，複製による永続化のアルゴリズムのうち，3 章で疑似コードを使って説明した参照の書き込みと書き込みに伴うオブジェクトの永続化のモデルを作成した．

本研究で作成するモデルは，特定の初期状態から実行を始めて，複製による永続化の `putfield` のアルゴリズムに沿って複数スレッドで任意のオブジェクトへ参照を書き込む．このようなモデルを，初期状態を変えて複数個作成する．実行中のすべての状態が 2 つの性質を満たしていれば，検査したモデルに関して複製による永続化のアルゴリズムは正しい．以下，任意のオブジェクトへ参照を書き込むスレッドのことを**ワーカスレッド**と呼ぶ．図 4.1 に本研究のモデルの例を示す．この例では，永続化されていないオブジェクト間に 1 つだけ参照がある状態が初期状態である．実行が始まると，中央の状態のいずれかに遷移する．中央の状態には，1 つ目のワーカスレッドが先に参照を書き込んだ場合，2 つ目のワーカスレッドが先に参照を書き込んだ場合，その他書き込む参照が異なる状態が存在する．各

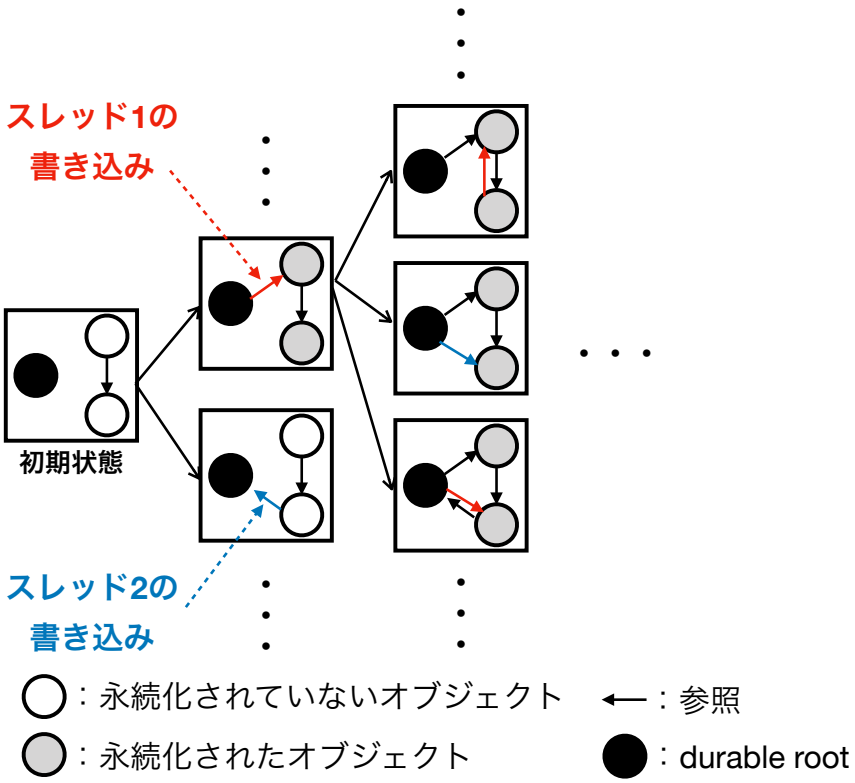

図 4.1　本研究で作成するモデルの例

ワーカスレッドは putfield のアルゴリズムに沿って参照を書き込むので，必要に応じてオブジェクトを永続化する．例えば，中央上の状態では durable root から永続化されていないオブジェクトへの参照をワーカスレッド 1 が書き込むので，同時に参照を辿って到達可能な 2 つのオブジェクトを永続化している．その後，中央の状態からさらに右の状態へ遷移して，実行が完了した状態になるまで遷移を繰り返す．このようにして初期状態から遷移できるすべての状態で，2 つの性質が満たされていることを確かめる．

オブジェクトを頂点，オブジェクト間の参照関係を辺とみなしたグラフを**オブジェクトグラフ**と呼ぶ．本研究で作成するモデルの初期状態の違いは，検査開始時点におけるオブジェクトグラフの違いによる．したがって，十分な検証のためには異なるすべてのオブジェクトグラフのパターンについて調べる必要がある．

すべてのオブジェクトグラフのパターンを 1 回のモデル検査で調べようとすると，一度にメモリを大量に消費するため，モデル検査のパラメータを小さくせざるを得なくなる．でき

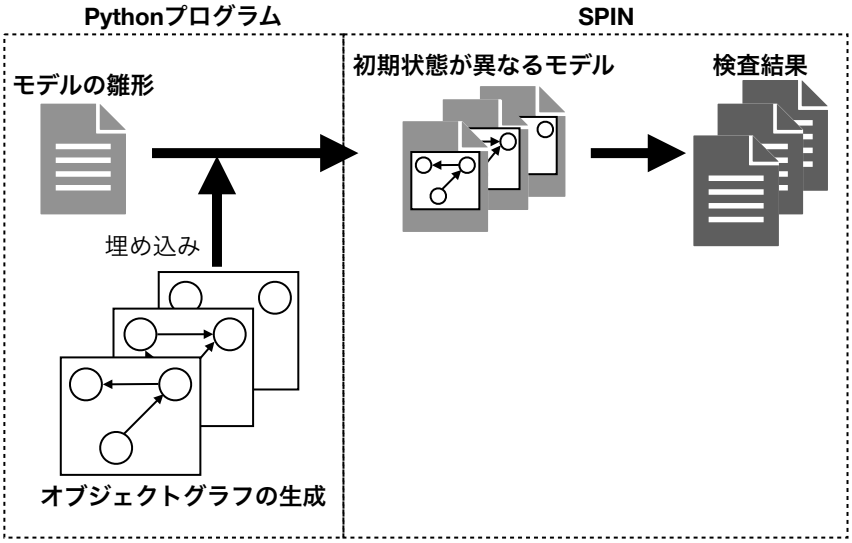

図 4.2　モデル検査の流れ

る限りパラメータを大きくするために，調べる実行経路を初期状態のオブジェクトグラフによって分割している．

モデル検査の流れを図 4.2 に示す．初期状態におけるオブジェクトグラフだけが決まっていないようなモデルの雛形を作成し，Python プログラムを使って別途作成したオブジェクトグラフを後から埋め込んでモデルを生成する．生成したモデルすべてについて検査してエラーが見つからなければ，複製による永続化のアルゴリズムが 2 つの性質を満たしており，正しい．

4.2　オブジェクトのモデル

揮発性コピーは参照を書き込める複数のフィールドと forwarding ポインタを書き込むフィールドを持つ．一方，永続化コピーは揮発性コピーと同じ数のフィールドと，永続化担当スレッドのスレッド ID を記録するフィールドを持つ．オブジェクトの永続化担当スレッドは，揮発性コピーの forwarding ポインタから参照されている永続化コピーにスレッド ID が記録されている．

4.3　パラメータ制限

現実的な実行時間と使用メモリ量では，Spin で調べることができる状態の数は有限なので，パラメータによる制限をかける必要がある．本研究では以下のパラメータを有限にしている．

- ワーカスレッドの数
- オブジェクトの数
- フィールドの数
- ワーカスレッドの書き込み回数

これらに加えて，MMLib のパラメータであるバッファサイズも制限される．特に，本研究ではワーカスレッドの数は 3 つ以下とする．すると，2 つ以上のスレッドが属するような ITG はたかだか 1 つになる．

4.4　複製による永続化のモデル

MMLib を使った，複製による永続化のモデルを作成した．作成したモデルの例として，ensure_recoverable 関数の一部を図 4.3 に示す．Promela の文法による違いを除けば，図 3.4 の擬似コードとほとんど対応するように記述している．Promela の関数はインライン関数なので，戻り値をそのまま変数に格納することはできない．そのため，インライン関数は引数に戻り値を格納するための変数を渡して呼び出す．例えば，擬似コードでは worklist_remove は引数を取らない関数であるが，モデルでは戻り値の格納先である `o` を引数にとっている．なお，`current_thread_id` はワークリストを識別するためのスレッド ID である．

検査対象アルゴリズムは競合のないプログラムを仮定していた．この仮定を満たすために，`putfield` を呼び出す前に書き込み対象となるオブジェクトをロックしている．そして，`putfield` の実行が終わってからロックを解放する．ロックを取得，解放する際にはフェン

ス命令を使用している．

4.4.1　Observer スレッド

ある状態がエラーかどうかを判定するために，Observer スレッドという検査用スレッドを作成した．Observer スレッドは任意の状態から実行を始めて，動作中はワーカスレッドを停止させる．そして，揮発性コピーと不揮発性コピーの値を見て，正しい状態かどうかを調べる．具体的には，durable root から揮発性コピーの参照を辿って到達できたオブジェクトについて (1) 全ての forwarding ポインタが null でないこと，(2) 揮発性コピーが持つ参照は永続化コピーが持つ参照と等しいことを確かめる．

ただし，ワーカスレッドが参照を書き込む途中のオブジェクトについては，(2) が満たされない場合がある．そのため，(2) については，変更前の参照と変更後の参照のいずれか一方であれば良いものとして扱う．ワーカスレッドは `putfield` を呼び出す前に，書き込み対象となるオブジェクトと書き込み前の値，書き込み後の値をワーカスレッドごとに記録しておく．そして，揮発性コピーと永続化コピーへの書き込みがメモリに反映されて，`putfield` の実行が終わったら記録しておいた値をクリアする．Observer スレッドは，ワーカスレッドが書き込み中のオブジェクトについては，書き込み前の値と書き込み後の値の両方と比較して，いずれか一方と一致すれば正しい状態であると判断する．

4.4.2　バリア同期の実装方法

`responsible_thread` と `depends_on`，`SYNC` は，ワーカスレッド数の制限によって 2 つ以上のスレッドが属する ITG はたかだか 1 つであることを使って実装している．各ワーカスレッドはどの ITG に属しているかを示す状態を持ち，(1)ITG に属していない，(2) 自身のみが属する，(3) 複数のスレッドが属している，のいずれかである．図 4.4 は ITG に関する状態の状態遷移図である．実行が始まったときのワーカスレッドは (1) の状態であり，`ensure_recoverable` を呼び出したときに (2) に遷移する．`depends_on` によって ITG が

```
1     VCopy o;
2     worklist_remove(current_thread_id, o);
3     PCopy on = READ(PCOPY_ADDR(o));
4  retry:
5     int f = 0;
6     do
7     :: f < FIELD_NUM ->
8         VCopy p = OBJ_FIELD(o, f);
9         if
10        :: p != NULL ->
11            shade(current_thread_id, p, result);
12            if
13            :: result == TRUE ->
14                worklist_add(current_thread_id, p);
15            :: result == FALSE -> skip;
16            fi;
17            WRITE(OBJ_FIELD_ADDR_NVM(on, f), READ(PCOPY_ADDR(p)));
18        :: else -> skip;
19        fi;
20        f++;
21    :: else -> break;
22    od;
23    FENCE();
```

図 4.3 永続化アルゴリズムのモデルの例（ensure_recoverable の一部）

併合された場合は，(3) に遷移する．SYNC は，自身の状態が (3) だったとき，ITG 内のスレッドが全て揃うまで待機する．一方，(2) だったときは (1) に遷移する．

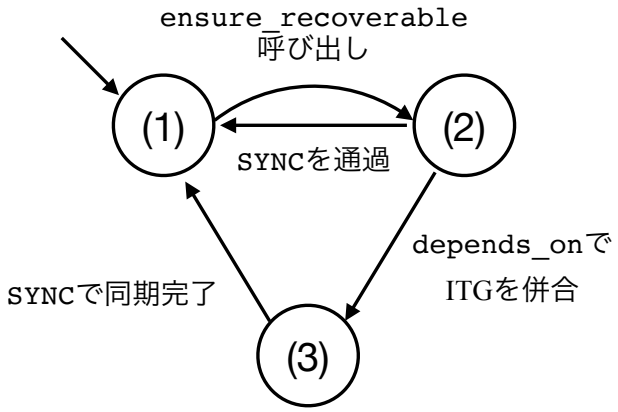

図 4.4　ワーカスレッドの ITG に関する状態遷移図

第 5 章

検査する状態数の削減

本研究で作成したモデルの一部は，初期状態におけるオブジェクトグラフが異なるモデルと検査結果が一致することが検査前にわかる． ほかのモデルと検査結果が一致するモデルについては検査を省略することで，検査する状態数を減らすことができる．本章では，検査結果が一致するパターンについて，そして本当に検査結果が一致することを示す証明について述べる．

5.1 オブジェクトグラフの削減による状態数の削減

2 つのオブジェクトグラフに対して，オブジェクト間の参照関係を保持しつつ揮発性コピーを揮発性コピーに、永続化コピーを永続化コピーに移すような一対一関係が存在するとき**同型である**という．複製による永続化のアルゴリズムはオブジェクトを ID によって区別しないので，ある状態が正しいならば，オブジェクトグラフが同型な状態も正しい．そこで，オブジェクトグラフを生成するとき，すでに生成したグラフと同型なグラフは生成せず，モデルの数を削減する．例として，図 5.1 のグラフ 1 とグラフ 2 は互いに同型なグラフであり，揮発性コピーのオブジェクト 1，3 とオブジェクト 2，4 がそれぞれ対応している．永続化コピーについても，オブジェクト 1，2 が交互に対応している．グラフ 1 においてオブジェクト 1，2 は永続化されているので，グラフ 2 においても対応するオブジェクト 2，1 が永続化されている．右のグラフにおいてオブジェクト 2 から 3 への参照を書き込む場合の検査結果について考える．ワーカスレッドは任意のオブジェクトに書き込むことができるので，左のグラフの検査においてオブジェクト 1 から 4 への参照を書き込むような状態は必ず

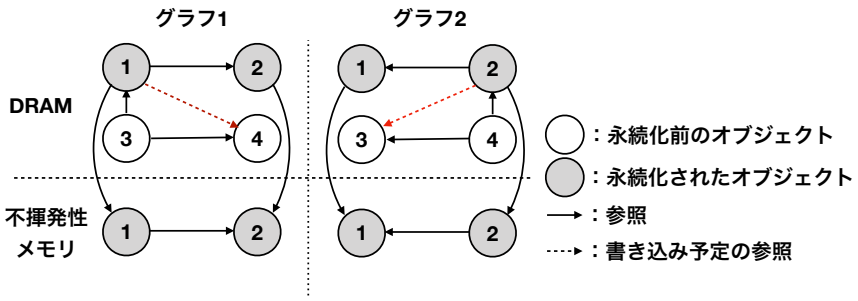

図 5.1　同型なグラフの例

検査済みである．左のグラフの検査結果におけるオブジェクトの ID を右のグラフに対応するよう置き換えれば，右のグラフの検査結果が得られる．

複製による永続化のアルゴリズムはスレッドを ID によって区別しないので，ワーカスレッド間で実行内容を交換した状態は交換前の状態と検査結果が一致する．そこで，ワーカスレッドごとに書き込み可能なオブジェクトを固定して，2 つのスレッド間で書き込む参照を交換した状態を調べないようにする．ただし，durable root は特別な変数なので，すべてのスレッドが書き込み可能である．

ワーカスレッドごとに書き込み可能なオブジェクトを固定する場合，オブジェクトグラフの同型判定の際にオブジェクトの書き込み担当スレッドを考慮する必要がある．1 つのワーカスレッドが複数のオブジェクトの書き込みを担当する場合，同じワーカスレッドが担当するオブジェクトに同時に書き込むような状態は調べることができない．そのため，担当するスレッドごとにオブジェクトの色を分けて区別をつけ，同じ色同士が対応するときのみ同型とする．なお，上で述べたようにワーカスレッドの実行内容は交換可能なので，色同士も交換可能であるとする．したがって，ワーカスレッド間で色を交換しただけのグラフは交換前のグラフと同型である．オブジェクトの色を考慮したオブジェクトグラフの同型判定の例を図 5.2 に示す．最も左のオブジェクトグラフが検査済みであったとして，元のグラフと呼ぶ．左から 2 つ目のグラフは，元のグラフからオブジェクト 1 と 3 を入れ替えたグラフである．オブジェクト 1 と 3 はどちらもスレッド 1 が書き込み可能なオブジェクトなので同型である．左から 3 つ目のグラフは，元のグラフからスレッド 1 とスレッド 2 の色を入れ替えただけのグラフなので同型である．左から 4 つ目のグラフは，元のグラフからオブジェクト 1 と

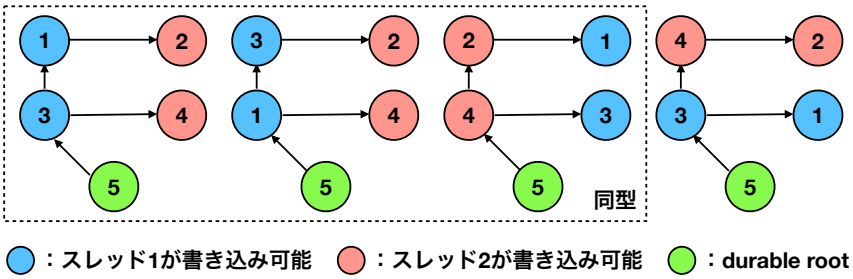

図 5.2　書き込み担当を考慮したオブジェクトグラフの同型判定の例

4 を入れ替えたグラフである．オブジェクト 1 と 4 はそれぞれスレッド 1，2 が書き込み可能なオブジェクトであり，互いに異なるので同型ではない．

5.2　削減方法が正しいことの証明

本節では，5.1 節で述べた状態数の削減方法によって，調べることができる状態は変わっていないことについての証明を述べる．

本研究で作成したモデルでは，モデルの初期状態によって探索する状態が変わる．そして，探索する状態の中に一つでもエラーが含まれていれば，検査結果はエラーになる．このことから，初期状態とモデル検査の結果を対応づけることができる．以下の証明では，任意の初期状態に対して，検査結果が一致するような検査済みの初期状態が存在することを示す．

5.2.1　定義

オブジェクトグラフ

$O = \{o_1, o_2, \ldots, o_n\}$ を**オブジェクト**（揮発性コピーと永続化コピーの両方を含む）の有限集合，$\mathcal{E} = O \times O$ の要素を**辺**と言う．**オブジェクトグラフ**を三字組 $G = \langle O, E_G, L_G \rangle$ で表す．ただし，$E_G \subseteq \mathcal{E}$ は辺の部分集合，$L_G : O \to Label$ はラベル付け写像とする．$Label$ は**ラベル**の有限集合である．ラベルは，揮発性コピーか永続化コピーか，durable root かどうか，書き込み可能なスレッドの情報を表す．ラベルは複数の情報を含みうる（例えば，durable root の永続化コピー）ものの，揮発性コピーかつ永続化コピーであることはない．

すなわち，O の各要素は**揮発性コピー**または**永続化コピー**のどちらかちょうど一方である．揮発性コピーから永続化コピーへの辺を *forwarding pointer* といい，それ以外の辺を**参照**と言う．

オブジェクトグラフ $G = \langle O, E_G, L_G \rangle$ が以下の条件を満たすとき，G は**妥当** (valid) であると言う．ただし，m は定数であり，フィールドの数を表している．妥当でないグラフは初期状態になり得ないので，以降は妥当なオブジェクトグラフのみ考える．

- 任意の揮発性コピー o について，o から出る forwarding pointer は高々1 個，o から出る参照は高々m 個．
- 任意の永続化コピー o' について，o' に入る forwarding pointer は高々1 個．
- 永続化コピーから揮発性コピーへの辺はない．
- 入る forwarding pointer がない永続化コピーは孤立頂点である（入る辺も出る辺もない）．

妥当なオブジェクトグラフの全体集合を $\mathcal{G}$ とする．

オブジェクトグラフ G に対し，揮発性コピーだけを集めた集合を $VC_G = \{o \in O \mid o$ は G において揮発性コピー $\}$ と定義する．$G = \langle O, E_G, L_G \rangle \in \mathcal{G}$ と集合 $V \subseteq VC_G$ が以下の条件を満たすとき，G において V は**正しく永続化されている**と言う．

- 任意の $v \in V$ に対して forwarding pointer $\langle v, n_v \rangle \in E_G$ が存在する．
- 任意の $v, v' \in V$ に対し，$\langle v, v' \rangle \in E_G \iff \langle n_v, n_{v'} \rangle \in E_G$.

O 上の全単射 $g : O \to O$ に対し，g の定義域を以下のように拡張する．

- $\langle o, o' \rangle \in \mathcal{E}$ に対し，$g(\langle o, o' \rangle) = \langle g(o), g(o') \rangle$ と定義する．
- $E \subseteq \mathcal{E}$ に対し，$g(E) = \{g(e) \mid e \in E\}$ と定義する．
- $L : O \to Label$ に対し，$g(L)$ を，$g(L)(g(o)) = L(o)$ であるラベル付け写像と定義する．
- $G = \langle O, E_G, L_G \rangle \in \mathcal{G}$ に対し，$g(G) = \langle O, g(E_G), g(L_G) \rangle$ と定義する．

検査モデル

本研究の**検査モデル**の振る舞いを，状態遷移系 $TS = \langle S, \to \rangle$ で表す．ただし，$S = \mathcal{G} \times \mathcal{E}^2 \times S_{\text{inner}}$ は状態集合，$\to \subseteq S \times S$ は遷移関係である．S_{inner} の要素は，プログラムカウンタやその他の内部データの状態を表す．直観的には，$\langle G, e_1, e_2, s_{\text{in}} \rangle \in S$ の第 1 成分 G は**現時点の**オブジェクトグラフ，第 2, 3 成分 e_1, e_2 はそれぞれスレッド 1 及び 2 が書き込む予定の参照である．

エラー状態の集合を $Err \subseteq S$ とする．$\to$ の反射的推移的閉包を $\to^*$ と書き，$s \to^* s'$ でないことを $s \not\to^* s'$ と書く．$S' \subseteq S$ について，$\forall s \in S', \forall s_e \in Err, s \not\to^* s_e$ であるとき，S' は**安全**と言う．

O 上の全単射 $g : O \to O$ について，g の定義域をさらに以下のように拡張する．

- $s_{\text{in}} \in S_{\text{inner}}$ に対し，$g(s_{\text{in}})$ を，s_{in} の中に含まれる $o \in O$ 及び $e \in \mathcal{E}$ をすべて $g(o)$, $g(e)$ に置換したものと定義する．
- $s = \langle G, e_1, e_2, s_{\text{in}} \rangle \in S$ に対し，$g(s) = \langle g(G), g(e_1), g(e_2), g(s_{\text{in}}) \rangle$ と定義する．

$\mathcal{G}_I \subseteq \mathcal{G}$ を，durable root から参照をたどって到達できる揮発性コピーの集合が正しく永続化されているようなグラフからなる集合とする．このとき，**初期状態集合** $I \subseteq S$ を $I = \{\langle G, \langle o_1, r_1 \rangle, \langle o_2, r_2 \rangle, s^0_{\text{in}} \rangle \mid G \in \mathcal{G}_I,\ o_1, r_1, o_2, r_2 \in VC_G\}$ と定義する．ただし，$s^0_{\text{in}} \in S_{\text{inner}}$ は検査モデルの初期状態の S_{inner} 成分を表し，O 上の任意の全単射 g に対して $g(s^0_{\text{in}}) = s^0_{\text{in}}$ とする（つまり s^0_{in} はオブジェクトや辺を含まない）．**モデル検査の目的**は，I が安全であることを示すことである．本研究で行ったモデル検査では，I の部分集合である $I_{\text{chk}} \subseteq I$ が安全であることを示した．I_{chk} を**検査済み初期状態集合**と呼ぶ．

5.2.2　本研究で行ったモデル検査の正当性

本研究における検査モデルは，定義から，オブジェクト ID の付け替えの影響を受けない．また，本研究で考えるエラー状態も，オブジェクト ID の付け替えの影響を受けない．従っ

て，以下の補題が言える．

補題 1. g を O 上の任意の全単射とする．

(1) $\forall s, s' \in S,\ s \to s' \iff g(s) \to g(s')$.

(2) $\forall s \in S,\ s \in Err \iff g(s) \in Err$.

補題 1 から以下が言える．

補題 2. 任意の $s_0 \in I$ について O 上の全単射 g が存在し，$\forall s_e \in Err,\ g(s_0) \not\to^* s_e$ ならば，I は安全．

また，I_{chk} の作り方から，以下の補題が言える．

補題 3. 任意の $s_0 \in I$ について，O 上の全単射 g が存在し，$g(s_0) \in I_{\mathrm{chk}}$.

補題 2, 3 から以下の定理が言える．

定理 1. I_{chk} が安全ならば I も安全．

第 6 章

検査結果

本章では，作成したモデルを使った検査の結果について述べる．検査に使用した環境は以下の通り．

- CPU：Intel Core i9-10920X CPU (3.50GHz，24 コア)
- メモリ：256GB
- OS：Ubuntu 20.04.3 LTS
- Spin：6.5.1
- GCC：9.3.0

検査では，オブジェクト数が異なる 2 種類のパラメータの設定を使用した．各パラメータの設定において生成したグラフの数と，同型なグラフを削減しない場合のグラフの数を表 6.1 に示す．同型なグラフを削減しない場合のグラフの数と生成したグラフの数を比べると，それぞれおよそ 0.26 倍，0.51 倍にまで削減できている．本研究ではグラフと同じ数のモデルを生成して検査するため，グラフ数の削減は結果的に検査時間の短縮になる．そのため，同型なグラフを削減したことで検査時間を短縮できたといえる．

MMLib によるメモリアクセス命令の入れ替えは，SC の場合と TSO の場合について調べた．1 つのモデルの検査に要したメモリ，検査時間の平均と，検査全体で要したメモリの総量を表 6.2 に示す．いずれの検査においても，バグは見つからなかった．平均メモリ使用量と平均検査時間のいずれも SC に比べると TSO の方が大きくなっており，メモリアクセス命令を入れ替えた場合の状態が増えていることがわかる．

表 6.1　設定したパラメータと生成したグラフの数

オブジェクト数	5	4
フィールド数	2	2
スレッド数	2	2
スレッドごとの書き込み回数	1	1
生成グラフ数	41,006	1,232
削減しない場合のグラフ数	161,051	2401

表 6.2　1 つのモデルの検査に要したメモリと検査時間の平均

メモリモデル	SC		TSO
オブジェクト数	5	4	4
平均メモリ使用量 (MB)	1,039	353	1,544
平均検査時間 (秒)	11.00	2.73	19.98

第 7 章

関連研究

本研究の検査対象である複製による永続化 [1] は，durable root から参照を辿って到達可能なオブジェクトを永続化する，到達可能性による永続化アルゴリズムである．到達可能性による永続化アルゴリズムは，本研究と同様に初期状態によって分割する方法で検証できる可能性がある．AutoPersist[2] はそのようなアルゴリズムの例である．永続化されたオブジェクトが DRAM 上にコピーを持たないという点で，複製による永続化とは異なるアルゴリズムである．

複製による永続化のアルゴリズムの，揮発性コピーの値を永続化コピーへ写す手順は，ガベージコレクタである Sapphire[8] のオブジェクトコピーアルゴリズムに基づいて作られている．Sapphire のオブジェクトコピーアルゴリズムは，ユーザプログラムによるオブジェクトの値の書き換えと並行して，DRAM から DRAM へとオブジェクトをコピーするためのアルゴリズムである．Sapphire のオブジェクトコピーアルゴリズムは [9] でも正しいことが確認されており，本研究の結果と一致する．

本研究では，キャッシュの書き戻しがキャッシュへの書き込みと同じ順で起こると仮定して検査した．しかし，実際の CPU ではキャッシュが書き込みと異なる順序で書き戻される可能性がある．[10] は，Intel x86 アーキテクチャと ARM v8 アーキテクチャ下でキャッシュの書き戻し順が異なる場合を考慮したメモリモデルを定義している．このメモリモデルを考慮した検査は今後の課題である．

第 8 章

おわりに

本研究では，複製による永続化のアルゴリズムの正しさをモデル検査器 Spin を使って確かめた．複製による永続化は，durable root から到達可能なオブジェクトがすべて永続化されており，揮発性コピーの値と不揮発性コピーの値が一致するという性質を持つ．パラメータによる制限の下でより多くの状態を調べるために，1 回のモデル検査で調べる状態数を減らす方法を 2 つ取り入れた．1 つは，ユーザスレッドによる書き込みの分割，もう 1 つは初期状態として生成するグラフの削減である．その結果，本研究で検査に用いたパラメータの範囲では，複製による永続化は 2 つの性質が満たされていることがわかった．さらに，生成するグラフの削減によって，検査時間を 1/4 程度短縮することができた．

今後の課題として，作成したモデルを使った複製による永続化のアルゴリズムの改良と，キャッシュの書き戻し順を考慮した検査がある．本研究で作成したモデルは，複製による永続化の擬似コードと似たような形で記述されているので，アルゴリズムの一部を変更したとき，容易にモデルに反映できる．そのため，本研究で作成したモデルを使うことで，変更後のアルゴリズムでも 2 つの性質が保たれていることが簡単に確認できる．このことを使って，2 つの性質を保ったまま複製による永続化のアルゴリズムを改良するための方法を考えることができる．

本研究ではキャッシュの書き戻しが書き込みと同じ順番で起こると仮定して検査したため，キャッシュの書き戻し順が入れ替わったような場合は調べていない．キャッシュを考慮して検査する場合，キャッシュの値が一箇所でも違うと別の状態として扱われるため，検査する状態数が大きく増加する可能性がある．キャッシュの書き戻し順を考慮した検査を行うためには，さらに検査する状態を減らす必要がある．

謝辞

本研究を行うにあたり，多くのご指導をいただきました高知工科大学情報学群高田喜朗准教授と東京大学大学院情報理工学系研究科鵜川始陽准教授に深く感謝いたします．また，副査をしてくださった高知工科大学の敷田幹文先生と横山和俊先生に心より御礼申し上げます．

参考文献

[1] 鵜川始陽, 松本康太郎, 岩崎英哉. オブジェクトの到達可能性による永続化をリードバリアを使わずに実現するアルゴリズムとその予備評価. 日本ソフトウェア科学会第 38 回大会講演論文集, Sep. 2021.

[2] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 316–332, 2019.

[3] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. Gcpersist: An efficient gc-assisted lazy persistency framework for resilient java applications on nvm. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pp. 1–14, 2020.

[4] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.

[5] Intel. Intel®64 and IA-32 architectures software developer's manual volume 3A: System Programming Guide. 2021-01-19 参照.

[6] 松元稿如, 鵜川始陽, 安部達也. メモリモデルを考慮したメモリアクセス命令を提供する SPIN 用ライブラリ. ソフトウェア工学の基礎, Vol. 23, pp. 63–72, 2016.

[7] 松本康太郎, 高田喜朗, 鵜川始陽. 不揮発性メモリを用いた java オブジェクト永続化のオーバヘッドの調査. 日本ソフトウェア科学会第 37 回大会講演論文集, Sep. 2020.

[8] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pp. 48–57, 2001.

[9] Tomoharu Ugawa, Tatsuya Abe, and Toshiyuki Maeda. Model checking copy phases

of concurrent copying garbage collection with various memory models. *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA, Oct. 2017.

[10] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. Revamping hardware persistency models: View-based and axiomatic persistency models for intel-x86 and armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 16–31, 2021.