平成３０年度修士論文

並列プログラム理解支援のための細粒度プログラムアニメーション

情報・ネットワーク工学専攻 コンピュータサイエンスプログラム

１７３１１３５ 藤本 明優

主任指導教員　寺田 実　准教授
指導教員　村尾 裕一 准教授

提出日 ２０１８年 ３月１１日

概要

目的

プログラムが実際にどのようにして動作しているかを理解することは, プログラミングにおいて重要なことの一つである. しかしマルチスレッドプログラムは逐次処理と異なり, ソースコードの見た目通りにプログラムが動いていないため, 実際の動作がわかりにくい. また, マルチスレッドのバグには発生確率の低いものや発生の有無が手元の環境に依存する場合がある. 本研究ではマルチスレッドプログラムがどのように動作しているのか, シングルスレッドプログラムと同じようにプログラミングしてしまうとどのような動作をしてどのような問題が発生するのかなどの動作原理の概要の理解支援を行う.

方法

手作業で全ての実行パスを再現できるプログラムアニメーションを行うシミュレータを開発する. ハードウェアレベルのプログラムアニメーションにより, マルチスレッドプログラムの動作原理の概要の理解を促す.

結論

評価実験により, 提案システムはマルチスレッドの並列プログラムの理解支援に一定の有効性を示した. 一方で, プログラムアニメーションが手動であるために見落としが発生した.

目次

図目次

第 1 章　序論

1.1　背景

プログラムが実際にどのようにして動作しているかを理解することは, プログラミングにおいて重要なことの一つである. しかしマルチスレッドプログラムは逐次処理と異なり, ソースコード上は一つの操作に見えても実は不可分でない操作やメモリの可視性などが原因でソースコードの見た目通りにプログラムが動いていないため, 実際の動作がわかりにくい. また, マルチスレッドのバグは発生確率が低いものがあるだけでなく環境によって全く発生しない場合があり, 実際に参考書などに記載されているバグのサンプルを手元の環境で試しても発生しないことがある.

1.2　マルチスレッドにおけるバグの種類

マルチスレッドプログラム特有のバグにはデッドロック, 競合状態, データ競合, メモリの可視性に由来するバグなどの種類がある.

この内, 競合状態の中には非ハードウェアレベルのプログラムアニメーションでは再現できない場合がある. 例えば変数への加算（例: *x* += 1;）は非ハードウェアレベルのアニメーションでは不可分に見えるので, 同期処理をしていないプログラムで同じ変数に複数のスレッドから加算を行ってもバグは発生しないように見えてしまう (図 1.1 左側). しかし加算の操作はメモリからレジスタへ変数を読み込む, 読み込んだ値に加算を行う, 加

	非ハードウェアレベルのアニメーション		ハードウェアレベルのアニメーション	
ステップ	スレッド 1	スレッド 2	スレッド 1	スレッド 2
1	*x* += 1 (*x*: 0 ⇒ 1)		x = 0 の読み込み (レジスタの値: 0)	
2		*x* += 1 (*x*: 1 ⇒ 2)	レジスタの値 += 1 (レジスタの値: 0 ⇒ 1)	*x* = 0 の読み込み (レジスタの値: 0)
3			メモリに書き戻し (*x*: 0 ⇒ 1)	レジスタの値 += 1 (レジスタの値: 0 ⇒ 1)
4				メモリに書き戻し (*x*: 1 のまま)
結果	*x* の値は 2		*x* の値は 1	

図 1.1: 非ハードウェアレベルとハードウェアレベルのアニメーションの比較

算した値をメモリに書き戻すという三つの操作から成り立っている. したがって図 1.1 右側のような順序で実行した場合, どちらか片方の加算の結果がもう片方の結果に上書きされて失われてしまう.

また, メモリの可視性に由来するバグはコンパイラによる最適化で行われる命令の省略や並び替え, ハードウェア側の実装による命令の並び替えなどを原因とするバグを指す. したがって使用する言語のメモリモデルやハードウェア側のメモリ一貫性モデルに影響を受けるため, 非ハードウェアレベルのアニメーションでは再現できない.

また, ハードウェアは各スレッドの実行順序を動的に決定する. これをスレッドスケジューリングと呼ぶ. マルチスレッド特有のバグはスケジュールによってバグが発生するか否かが決定される場合がある. スレッドスケジューリングはソフトウェア側で制御するものではないので, まれなスケジュールでのみ発生するバグは発見が困難であり, 一度発見しても再現には多大な労力と時間が必要になる.

1.3　メモリモデル

メモリモデルという言葉には複数の意味があるが, ここでは各プログラミング言語のメモリモデルとハードウェアのメモリ一貫性モデルについて言及する.

Java のようなプログラミング言語はメモリモデルを定義し, それに従ってコンパイルの際にプログラムの最適化を行う [1]. 例えば Java であれば主に synchronized ブロックや volatile 修飾子などを用いて同期し, それ以外は逐次処理で矛盾のない範囲で命令の並び替えや省略を許可する [2]. 並列処理プログラムで正しく同期処理をしなかった場合, 矛盾が発生しバグが起きる.

ハードウェアのメモリ一貫性モデル (メモリコンシステンシモデル) とは, 主にハードウェアがどのように機械命令の実行順序 (特にメモリアクセス命令) を並び替えるかという規則などのことを指す [3]. ハードウェアは逐次実行で矛盾のない範囲で機械命令の実行順序を実行時に動的に並び替えることで実行速度を上げることができる. 並列処理や並行処理では矛盾が発生するが, メモリバリア命令によって並び替えを制限することでそれを防ぐことができる. 例えば Java であれば synchronized ブロックに暗黙的にメモリバリアが実装されている [4]. アーキテクチャごとに一貫性モデルは異なり, メモリアクセス命令の並び替えを全く許さないメモリモデルから逐次実行に矛盾のない範囲であれば他には制限のないメモリモデルまで様々な種類がある. 前者は強いメモリモデルと呼ばれ, 後者は弱いメモリモデルと呼ばれる.

1.4　目的

本研究ではマルチスレッドプログラムがどのように動作しているのか, シングルスレッドプログラムと同じようにプログラミングしてしまうとどのような動作をしてどのような問題が発生するのかなどの動作原理の概要を学ぶためのツールを開発する.

マルチスレッド特有のバグのうち, デッドロックは既に可視化が行われている [5][6]. そこで, 本研究では競合状態やメモリの可視性など, よりハードウェア側に近い動作を可視化したいと考えた.

実機ではメモリモデルの違いやコンパイラによる最適化の違いなどにより全ての可能性を再現することはできないため, 逐次処理プログラミング経験者向けビジュアルプログラミング言語による入力およびハードウェアレベルのプログラムアニメーションを行うシミュレータの開発を目的とする.

全ての可能性を提示する方法としてモデル検査によりバグのあるパターンを検出 [7][8] して自動実行する方法と手作業で全パターンを実行できるようにする方法が考えられる. 本研究では学習を目標とするため, バグの

有無にかかわらず全ての可能性を実行できるように後者を選択した. ユーザは複数の可能な操作から一つを選ぶことで, スレッドスケジューリングやハードウェアによる動的な実行順序の並び替えをシミュレートできる.

1.5　本論文の構成

論文の構成を簡単に説明する．本章では，序論として研究の背景，目的について述べた．2 章では，関連研究について述べる．3 章では，マルチスレッドに影響するコンパイラやハードウェアのアルゴリズムや仕組みについて述べる．4 章では，提案システムの概要を述べる．5 章では，提案システムの実装について述べる．6 章では，評価の方法や結果について述べ, 考察する．7 章では，提案システムの有効性や問題点, 今後の課題と提案システムの応用や展望について述べる.

第 2 章　関連研究

2.1　並列処理プログラミングにおける Scratch の可能性についての考察 [9]

喜家村はプログラミング初級者向けのビジュアルプログラミング言語によるプログラミング教育ツールである Scratch[*1](図 2.1) を並列処理プログラミングの教育に応用することを提案し, その実用性を示した. Scratch はプログラミング初級者向けのプログラミング教育ツールであり, 複数のオブジェクトを並列に動かす機能を元々有しているため, ライブラリを用意するなどの手間が不要である. むしろオブジェクトが並列で処理される性質上, Scratch で全体を正しく処理するには各オブジェクト間の関係を正しく理解している必要があるため, Scratch でのプログラミングを学ぶことは並列処理の学習に役立つ.

しかし Scratch はプログラミング初級者向けのプログラミング教育ツールであるため, アニメーションは 1.2 節で言及した非ハードウェアレベルのアニメーションにあたる. したがってアニメーションの抽象度が異なるため, Scratch では再現できないバグを本研究で提案するツールは再現することができる.

また, Scratch では例えば変数への加算を繰り返すブロックのまとまりを複数同時に動かし始めた場合, 全てのブロックのまとまりは同じタイミングで加算する. つまり, 全てのブロックのまとまりの各命令は等しい速度で処理されていく. Scratch で書かれたプログラムをマルチスレッドプログラムとしてとらえると, これはスレッドスケジューリングが常に一定であるということになる. このような処理はプログラミングにおいて厳密な意味での並列処理および並行処理ではなく, スレッドスケジューリングについて誤解を招くことや厳密な並

図 2.1: Scratch による並列プログラミングの記述例 ([9] 図 2 より引用): 猫とオウムはそれぞれ独立に動作する. これは並列に動作するプログラムと言える.

[*1] https://scratch.mit.edu/

列処理ではバグの発生する可能性があるプログラムが固定されたスレッドスケジューリングによって常に正しく動いてしまうことが懸念される.

2.2　Visualizing Potential Deadlocks in Multithreaded Programs[5]

Byung-Chul Kim らはデッドロックの可視化のためのツールである VisualDeadlock を開発した (図 2.2). これは教育用ツールではなく, プログラム中のバグを発見するためのツールである. 実際のプログラムを静的に解析し, ロックを可視化している. ユーザは生成された図を見ることで, ソースコードを手作業で解析するより容易に潜在的なデッドロックを発見できる. この手法はデッドロックの発生確率によらずデッドロックを発見・可視化できるという点において, リアルタイムでの可視化や実際の実行履歴に基づく可視化よりも優れている.

2.3　ThreadMentor: A Pedagogical Tool for Multithreaded Programming[6]

マルチスレッドの可視化については, 他にも研究が行われている. Steve Carr らが作成した ThreadMentor はマルチスレッドのプログラムの実行を可視化できる教育および学習用ツールである (図 2.3). ロックの状態なども可視化され, デッドロックが起きた場合にはそれを検知できる. 可視化はリアルタイムでの解析と生成した実行履歴の解析の 2 種類を扱うことができる. コンパイル済みのデータに対して実行を可視化すること, スケジューリングはハードウェアにより動的に決定されることの 2 点から, 本研究と異なり任意の実行パスを可視化できない. また, 本研究と異なり不可分な命令単位で可視化されるわけではなく, 競合状態やメモリの可視性に由来するバグの詳細を可視化することはできない.

2.4　The design of a multithread programing course and its accompanying software tools[10]

Ching-Kuang らはマルチスレッドプログラミングの教育コースとそれを支援するツールについて提案した. 支援ツールの中には可視化の機能が含まれる.

Ching-Kuang らは可視化の方法として静的解析, リアルタイムでの解析, 過去に生成した実行履歴の解析の三つを提案した. しかし同時にマルチスレッドプログラムにおいて発生確率の低い振る舞いや環境によって発生しない振る舞いについて言及しており, 解決策としてあらかじめそのような振る舞いのアニメーションを用意し, web で利用できるようにするとしている.

本研究で提案するシステムでは対話的ステップ実行により実行パスをユーザが選べるようになっているため, 発生確率や環境によらずシミュレートすることが可能である. これによりユーザはあらかじめ用意されたもの以外のバグや振る舞いもシミュレート可能であり, また本研究で提案するシステムを使用することで, 小さなプログラムであれば発生確率の低いバグや環境によって発生しないバグなどについても容易にアニメーションのサンプルを作成することができる.

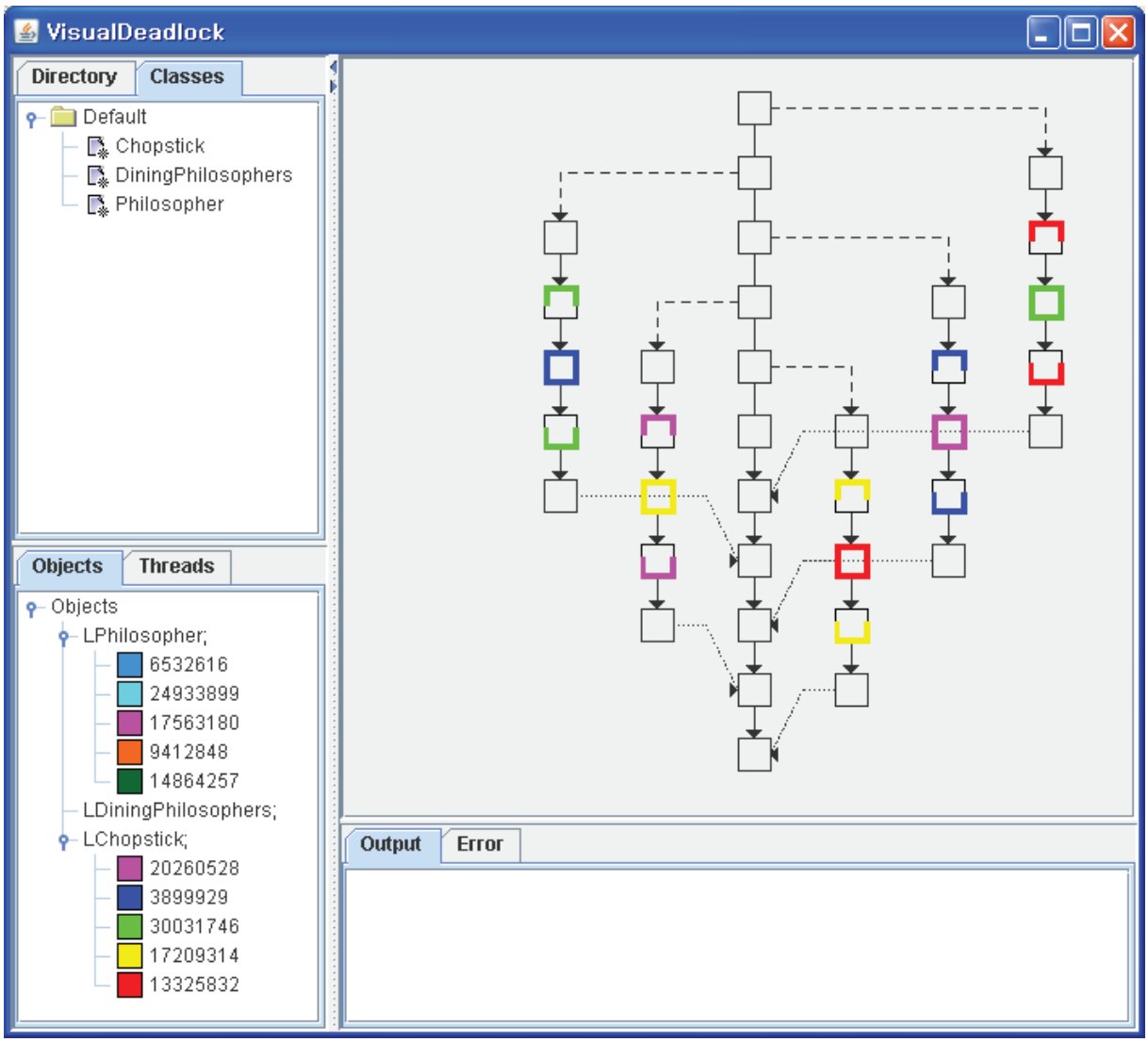

図 2.2: VisualDeadlock による可視化 ([5] Fig. 4 より引用)
画面右側の図の色のついた部分が各ロックとその順序を表している.

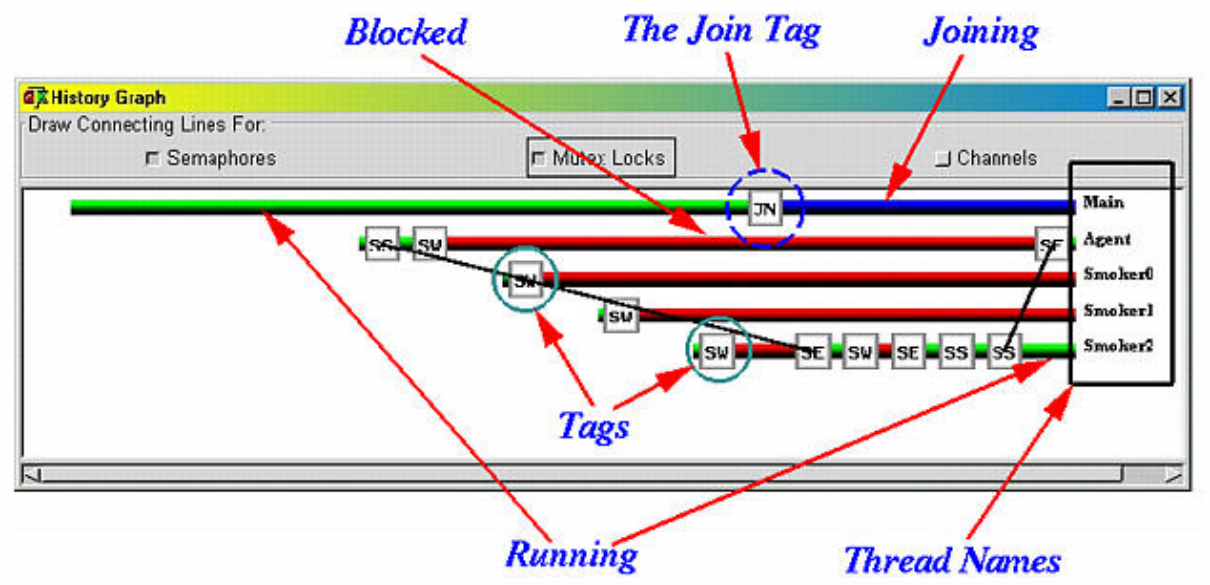

図 2.3: Thread Mentor の History Graph Window(ThreadMentor のホームページ [*1]から引用): バーは各スレッドの状態を表す. バーの右側はスレッドの名称を示している. バーの色は緑ならば実行中, 青ならば join で待機中, 赤ならばブロックされていることを表す. タグ (Tags で示されている四角形) は JN なら join, SS ならシグナル, SW ならウェイトを表し, SE ならばシグナルを受けて待機状態から解放されたことを表している. タグをクリックすると Source Window が開き, ソースコードを確認できる.

2.5 Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models[7]

安部らはメモリ一貫性モデルを論理式で記述し, そのメモリモデルに従って並列プログラムの振る舞いを検査できるモデル検査器 McSPIN を開発した. 論理式によりメモリモデルを記述できるため, 強いメモリモデルから弱いメモリモデルまで検査することができる.

本研究ではモデル検査を導入していないが, McSPIN によりモデル検査で並列処理特有のバグを検知できることが実証された. したがって, 本研究で提案するシミュレータにモデル検査を導入することができれば, バグの自動検知が可能になることがわかる.

[*1] https://pages.mtu.edu/ shene/NSF-3/e-Book/FUNDAMENTALS/VISUAL/basic.html

第 3 章　マルチスレッド特有のバグに影響する要素

マルチスレッド特有のバグに影響する要素として, ハードウェアのメモリ一貫性モデル, コンパイラによる最適化, アウト・オブ・オーダー実行, ストア・バッファが挙げられる.

3.1　ハードウェアのメモリ一貫性モデルによる load 命令と store 命令の並び替え制限

1.3 節で示したように, ハードウェアのメモリ一貫性モデルによって動的な実行順序の並び替えの制限が行われる. また, アーキテクチャによって一貫性モデルはそれぞれ異なる. したがって, メモリの可視性に由来するバグの中には手元の環境では起こらなかったが他のアーキテクチャでは発生した, という場合もあり得る. ハードウェアのメモリ一貫性モデルには例えば以下のような種類が挙げられる [11][12].

- SC(Sequential Consistency)
  - メモリアクセスの順序は並び替えないモデル
- TSO(Total Store Order)
  - load 命令がその前の store 命令より先に実行することを許すモデル (図 3.1①)
- PSO(Partial Store Order)
  - load 命令と store 命令がその前の store 命令より先に実行することを許すモデル (図 3.1①②)
- STO(Store Order)
  - load 命令がその前の load 命令や store 命令より先に実行することを許すモデル (図 3.1①③)
  - 順序制限の強さは PSO と相互互換
- RMO(Relaxed Memory Order)
  - 逐次実行に矛盾がなければメモリアクセスに制限は設けないモデル (図 3.1①②③④)

3.2　コンパイラによる最適化

コンパイラによる最適化によって, 逐次処理では矛盾のない範囲でプログラムの各命令の省略や並び替えが行われる. この仕組みにより, コンパイル後のプログラムは実際にはソースコード上での見た目とは全く異なる挙動をしている. したがって, 同期処理を正しく行っていない場合, コンパイラによる最適化で行われる命令の省略や並び替えはバグの原因になる.

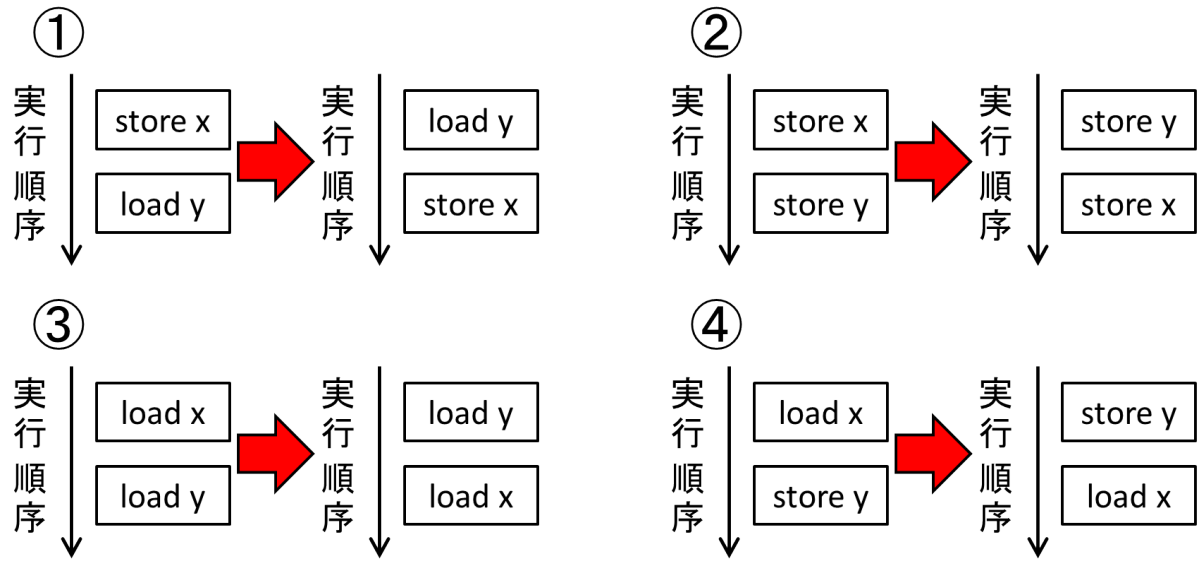

図 3.1: メモリ一貫性モデルによる順序制限: ①load 命令がその前の store より先に実行する例, ②store 命令同士の順序が逆になる例, ③load 命令同士の順序が逆になる例, ④store 命令がその前の load 命令より先に実行する例

	最適化前	最適化後
1	$x = 1;$	$x = 1;$
2	$a = x;$	$a = 1;$

図 3.2: 定数伝播

3.2.1　最適化手法

最適化には様々な手法があるが, 本項では提案システムに実装した最適化手法と並列処理との関係について記述する (参考 [13]).

3.2.1.1　定数伝播

最適化の手法として, 自明な定数が書き込まれている変数の読み込みを定数に置き換える手法がある. これを定数伝播と呼ぶ (図 3.2). また, オペランドが全て定数であるために式の計算結果が自明な場合, その式を計算結果に置き換える手法を定数の畳み込みと呼ぶ. 定数伝播と定数の畳み込みは効果を高めるために併用される. 正しく同期処理をしていないプログラムでは, 定数伝播により想定している読み込みが行われなくなることで想定外の挙動が発生することがある (図 3.3).

	最適化前	最適化後
1	$x = 1$;	$x = 1$;
2	while ($x == 1$) {}	while (true) {}

図 3.3: 定数伝播によるバグ: 他のスレッドによる x への代入によりループを抜けることを想定している場合でも, x の定数伝播により単なる無限ループになってしまう.

	最適化前	最適化後
1	while(x != 1) {	if(x != 1) {
2	y += 1;	while(true) {
3	}	y += 1;
4		}
5		}

図 3.4: 不変式のループの外への移動の例: 機械命令レベルの話だが, 仮に C 言語のプログラムで表している.

3.2.1.2　ループの外への移動

ループの外への移動という最適化手法およびそれによって発生するバグについて説明する. 例えば図 3.4 のようなプログラムを考える. 初期値は $x = y = 0$ とする. このプログラムをコンパイラが最適化する場合, まずループに対して同期処理がされていないので, コンパイラはこの while 文は逐次処理において矛盾のない範囲で最適化可能と考える. さらに条件式とループ内の処理を見比べると, 条件式で参照される x はループ内では変化していないことがわかる. ループ内の処理を減らすことはプログラムの実行速度を上げることに繋がるので, コンパイラはループ内の処理の影響を受けない条件式のチェックを最初だけ行うようにする. つまり, C 言語で表すならば図 3.4 右側のように書き換えられる. これは逐次処理では結果の変わらない書き換えだが, 並列処理では他のスレッドが x を変化させた場合でもその値の変化を受け取れず無限ループが終わらないようになる. したがって, 並列処理で他のスレッドの処理の終了待ち等をする場合に単なる while 文で待とうとすると, この最適化によって無限ループになる.

3.2.1.3　命令スケジューリング

最適化の他の手法として, 命令スケジューリングがある. これはアーキテクチャがスーパースカラと呼ばれる, 同一スレッドの命令を並列に実行できる機能を有している場合に特に有効な最適化手法である. 例えば図 3.5 左側のようなプログラムがあった時, x や y の読み込みの直後に読み込んだ値を使って計算をしている. これはスーパースカラを実装しているハードウェアでは読み込みと計算が同時に行えないことを表す. したがってコンパイラは値の読み込みと参照の間に依存関係のない別の処理を挿むことで実行効率を上げる. この手法が命令スケジューリングである. しかし正しく同期処理をしていない場合, この最適化が行われると他のスレッドから見た値の読み込みと書き込みの前後関係が変化してしまうため, 想定外の挙動が起きることがある.

	最適化前	最適化後
1	*x* の読み込み	*x* の読み込み
2	*y* の読み込み	*y* の読み込み
3	読み込んだ *x* と *y* の加算	*a* の読み込み
4	加算結果の *x* への書き戻し	読み込んだ *x* と *y* の加算
5	*a* の読み込み	*b* の読み込み
6	*b* の読み込み	加算結果の *x* への書き戻し

図 3.5: 命令スケジューリングの例 (機械命令レベル): 5, 6 行目が 2, 5 行目に移動している.

3.3　アウト・オブ・オーダー実行

アウト・オブ・オーダー実行 (以下, OoO 実行) とは, CPU による動的パイプライン・スケジューリングにより実行時に命令の実行順序を動的に並び替えることで効率化を図る仕組みである. 各命令の実行結果の出力, つまりレジスタやメモリへの値の書き込みは元の順序で行われるため, 逐次処理では問題は発生しない. 結果を元の順序で出力することをイン・オーダー確定と呼ぶ. 近年のコンピュータはそのほとんどが OoO 実行を実装している. しかし逐次処理と同じようにコードを記述するとマルチスレッドによる動作が考慮されず, バグの原因になる. 本研究では 1967 年に IBM の Robert Marco Tomasulo によって提唱されたアルゴリズム [14] を簡略化したものを実装する.

3.3.1　Tomasulo のアルゴリズム [14][15]

Tomasulo のアルゴリズムは OoO 実行を実現するアルゴリズムの一つである. 命令のフェッチ・デコードユニット, 機能ユニット, 確定ユニットの 3 種類のユニットからなる動的パイプラインによって機械命令が実行される (図 3.6). 機能ユニットは機械命令の実行ステージの種類ごとに一つ以上が存在する.

まず命令のフェッチ・デコードユニットで機械命令のフェッチとデコードが行われ, 結果がその命令の実行ステージの種類に対応する機能ユニットの一つに送られる. この処理はイン・オーダーで行われる.

命令とオペランドは機能ユニットごとのリザベーション・ステーションというバッファに蓄えられ, オペランドが全て準備されたものから実行される. これによって順不同な実行が行われる. したがって, 逐次実行として考えた際に矛盾がなければ, メモリアクセスなど時間のかかる命令の完了を待たずにそれより後の命令を実行できる.

実行結果は確定ユニットおよび機能ユニットに送られる. 機能ユニットに送られた結果はリザベーション・ステーション内の各オペランドに反映される. 確定ユニットに送られた結果は確定ユニット内のリオーダ・バッファに蓄えられる. 確定ユニットでは残っている命令の中でフェッチ・デコードされた時刻が最も古い命令の結果が送られてくるのを待つ. その最古の命令の結果がリオーダ・バッファに送られてきたならば, それを出力してリオーダ・バッファから取り除く. この処理をリタイアという. この確定ユニットの働きによってイン・オーダー確定がされる.

Tomasulo のアルゴリズムは投機的実行にも適している. 投機的実行を行い, 分岐の予測が正しければそのままリタイアし, 誤っていればリタイアせずに破棄して分岐地点からやり直せばよい.

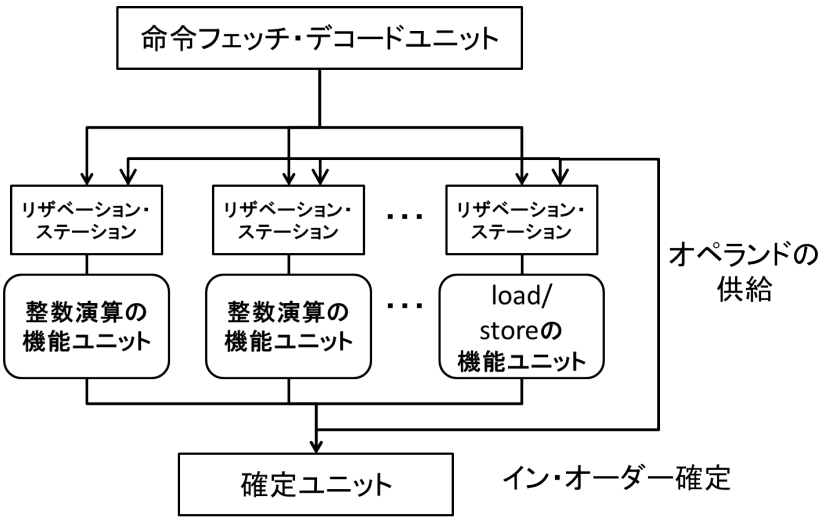

図 3.6: Tomasulo のアルゴリズム

load 命令の読み込みは機能ユニットによる実行時にオペランドの読み込みとして行われるため, この仕組みによって load 命令はそれより前の load 命令や store 命令より先に実行できる. store 命令の結果はリタイアの時点で反映されるため, 順序はイン・オーダーになる. したがって OoO 実行だけでは store 命令はそれより前の load 命令や store 命令より先に書き込みを行えない.

3.3.2　リネーミング [15]

OoO 実行について, 同じレジスタに対する操作を考える. この時, 読み込みの後の書き込み (Write After Read, WAR) と書き込みの後の書き込み (Write After Write, WAW) は順序関係を守らなければ, 正しい結果が得られない (図 3.7 左側, 図 3.8 左側). しかし, これらの順序関係はたまたま同じレジスタを使用しているがために起こるものである. したがって別のレジスタが仮に空いているとするならば, WAR における書き込み, および WAW における 2 回目の書き込みは同じレジスタを使用する必要はない. この関係性を偽の依存性と呼ぶ.

偽の依存性を取り除くためには, 命令パイプライン上でのレジスタ名を異なる名前にすればよい (図 3.7 右側, 図 3.8 右側). この処理によって WAR と WAW は順序関係を守る必要がなくなる. この処理をリネーミングと呼ぶ.

Tomasulo のアルゴリズムにおいては, 機能ユニットでの実行の結果がリザベーション・ステーション内の別のオペランドに供給される際に, その命令より後でその命令と同じレジスタへの書き込みより前にあるオペランドにのみ実行結果を供給することで, 実効的にリネーミングを実装することができる.

3.3.3　ロードキューとストアキュー [16]

OoO 実行において, load 命令と store 命令の実行はそれぞれロードキューとストアキューに依頼される.

load 命令の場合, 機能ユニットでの実行の際にロードキューにアクセス要求がされる. ロードキューはメモリやキャッシュがアクセスを許可してからメモリやキャッシュにアクセスし, 値を取得する. 取得した値は load 命令に返され, load 命令の実行が完了する.

	リネーミング前	リネーミング後
1	レジスタ A から読み込み	レジスタ A から読み込み
2	レジスタ A へ書き込み	レジスタ A’ へ書き込み
3〜	以下, レジスタ A からの読み込みなど	以下, レジスタ A’ からの読み込みなど

図 3.7: WAR のリネーミング: リネーミング後はステップ 1 とステップ 2 の間に依存関係がなく, 順序を交換できる.

	リネーミング前	リネーミング後
1	レジスタ A へ書き込み	レジスタ A へ書き込み
2	レジスタ A へ書き込み	レジスタ A’ へ書き込み
3〜	以下, レジスタ A からの読み込みなど	以下, レジスタ A’ からの読み込みなど

図 3.8: WAW のリネーミング: リネーミング後はステップ 1 とステップ 2 の間に依存関係がなく, 順序を交換できる.

store 命令の場合, 確定ユニットからのリタイアの際にストアキューにアクセス要求がされる. アクセス要求を送った時点で store 命令の実行は完了された扱いとなる. ストアキューはメモリやキャッシュがアクセスを許可してからアクセスし, 値を書き込む.

ロードキューとストアキューにより, メモリアクセスのスーパースカラ実行ができるため, 実行効率が良くなる.

ロードキューとストアキューの実行順序について, load 命令と store 命令はアクセスするメモリのアドレスが決定するまでは順序関係を確定することができない (3.3.4 項).

3.3.4　メモリディスアンビギュエーション [16]

同じアドレスへの書き込みの後の読み込み (Read After Write, RAW) がある場合, ロードキューはその同じアドレスへの書き込みをストアキューが終わらせてからでなければその読み込みを行うことができない. そのため, メモリのアクセスするアドレスを計算しなければロードキューは読み込みを行っていいかどうか判断することができない. アクセスするメモリのアドレスを計算し, 順序関係を確定させることをメモリディスアンビギュエーションと呼ぶ.

3.4　ストア・バッファ

Oracle によれば, SPARC などのハードウェアにはストア・バッファが実装されている [17](図 3.9). ストア・バッファは store 命令の実行を遅らせることで実際のメモリアクセスを減らし, 実行速度を上げる.

store 命令が行われた際に, ストア・バッファが実装されたハードウェアはキャッシュやメモリに直接書き込まず, ストア・バッファに書き込む. ストア・バッファに書き込まれた値はストア・バッファが一杯になった時やメモリバリア命令を行った場合などにキャッシュやメモリに書き込まれる. マルチコアのプロセッサなどでは各コアがストア・バッファを持ち, 同じコアのスレッドからはストア・バッファに書き込まれた値を読み込

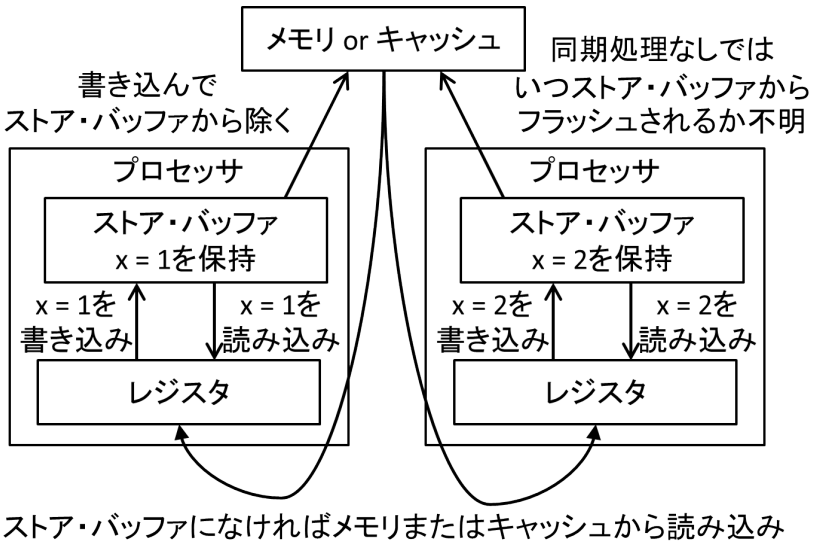

図 3.9: ストア・バッファの仕組み

むことができるが, 他のコアは別のコアのストア・バッファを参照できない. この仕組みによって store 命令の実行が遅延されるので, load 命令や store 命令をそれより前の store 命令より先に実行することができる.

第 4 章　提案システム

4.1　概要

ビジュアルプログラミング言語によりソースコードを入力し, それを擬似的な環境でコンパイルした結果を対話的なステップ実行によるプログラムアニメーションで可視化する. アニメーションは OoO 実行を実装する. 本ツールに付属するサンプルやネット上にあるマルチスレッドのバグの例を確かめることにより並列処理への理解を促すシステムを目標とする.

4.2　利用例

例として, 参考書に示されたバグのサンプルを確かめる場合を考える. まず, ユーザは参考書に示されたソースコードをページ上部のビジュアルプログラミング言語部分 (図 4.1①) で入力する. コンパイルは自動的に行われ, コンパイル結果がページ中段 (図 4.1②) に表示される. ユーザはページ下部のアニメーション部分 (図 4.1③) の実行ボタンをクリックすることでアニメーションを開始することができる. ユーザは実行時間を進めながら参考書に示された実行順序になるようにアニメーション部分の命令をクリックする. アニメーション部分のメモリに格納された値を見ることで実際にバグが発生していることを確認できる.

4.3　シミュレーションする計算機のモデル

並列処理および並行処理を扱うため, マルチコアのプロセッサを対象とする. メモリ一貫性モデルについては, オプションによって選択できるようにすることで複数種類に対応する (4.4.2 項).

各コアへのスレッドの配置は本来制御できないが, 提案システムはシミュレータであるため, どのコアにどのスレッドを配置できるかをユーザが選べるようにする. 同じコアに属するスレッドは並行処理はできるが並列処理はできない. つまり, 同時刻に命令を実行することができない. 別々のコアに属するスレッドは並列処理ができる. つまり, 同時刻に命令を実行することができる.

4.4　仕様

4.4.1　ビジュアルプログラミング言語による入力部分

ビジュアルプログラミング言語を用いることは文法エラーなどを極力排し, バグの検証をわかりやすくする目的を含む. ビジュアルプログラミング言語による入力部分では, 各命令のブロックの上下方向の高さによって不可分な命令単位を表現する. 例えば定数の代入文と変数の加算操作 (リード・モディファイ・ライト操作) を考えた場合, 定数の代入文は不可分な操作であるため, 1 段分の高さとなる. 変数の加算操作は変数のレジスタ

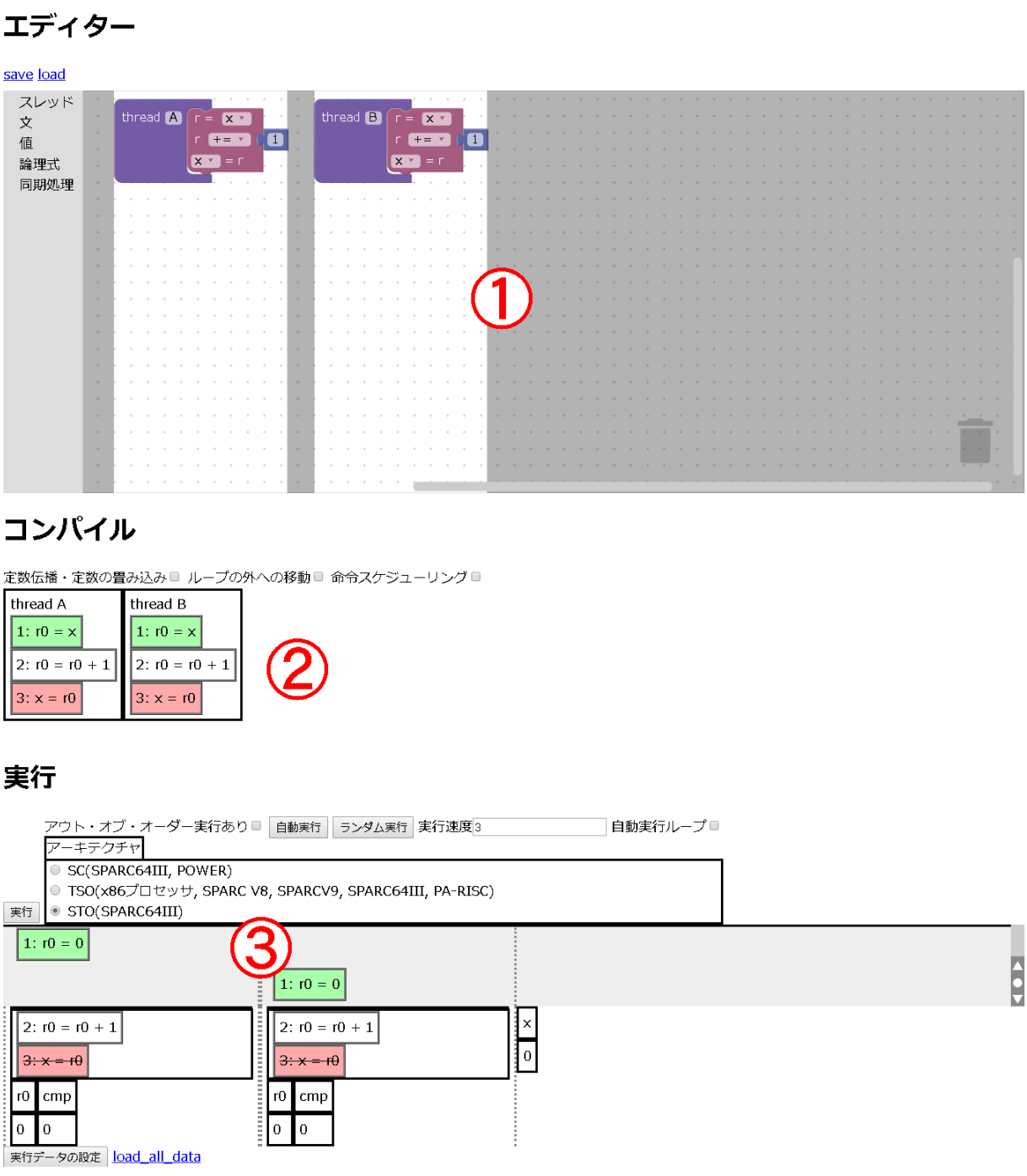

図 4.1: 全体のインタフェース: ①ビジュアルプログラミング言語による入力部分, ②コンパイル結果の表示部分, ③プログラムアニメーションの表示部分

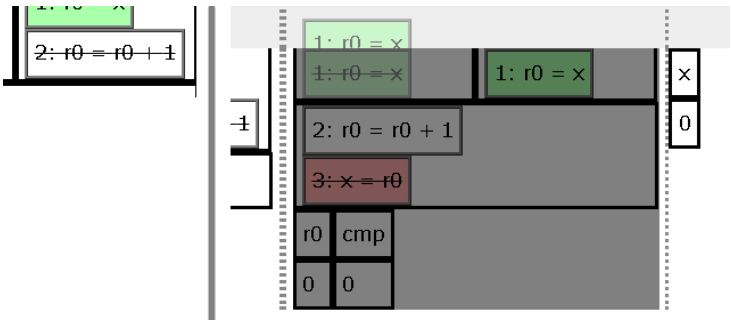

図 4.2: 実行できない箇所の表現: 左は実行できない命令に打ち消し線が引かれている. 実行できない原因になっている他の命令が実行やリタイアなどをするまで実行できない. 右の黒色の半透明の領域は一時的に実行できない箇所. 他の命令の実行などを伴わなくても実行時間が進むことで解放される.

への読み込み, 読み込んだ値への加算, レジスタからメモリへの書き込みの三つの不可分な操作から成り立つため, 定数の代入文の 3 倍, つまり 3 段分の高さとなる.

4.4.2　オプション

ユーザはコンパイルオプションや実行時のメモリ一貫性モデルを本ツールが提示するものの中から選択できる. これによってユーザの環境によらずバグの再現を行うことができる.

4.4.3　アニメーションの実行制御

アニメーションの実行制御は複数スレッドの制御を簡単化するために GUI を用いる. 実行不可能な個所を打ち消し線を引く, 黒色の半透明の領域で覆うなどで視覚的に表現する (図 4.2). 打ち消し線が引かれた命令は他の命令が実行・リタイアされるまで実行できないことを表す. 黒色の半透明の領域で覆われた部分は他の命令の実行などを伴わなくても, 実行時間を進めることで実行できるようになることを表す. これによってユーザは正確なハードウェアの仕組みを知らなくても実行順序を間違うことなくシミュレートできる. また, アーキテクチャを変更してシミュレートすることでハードウェアごとに実行順序の制限が異なることを理解できる.

4.5　仕様上簡略化したもの

以下の機能は他の機能により同じ種類の並び替えによるバグを再現できる. したがってレイアウトの煩雑化を防ぐため, 簡略化・省略した.

4.5.1　アウト・オブ・オーダー実行

OoO 実行について, 実際には複数の機能ユニットが存在し, 実行速度を高めている. しかし機能ユニットの個数はバグの発生の有無にかかわらないので, 機能ユニットは単一の物とする.

ロードキューとストアキューはキューなので, load 命令同士, および store 命令同士の順序関係は保存される.

store 命令のアクセス要求がされた時点でそれ以前の load 命令はリタイアしていなければならないので, この仕組みで store 命令を先行する load 命令より先に実行することもできない. また, OoO 実行により load 命令は先行する store 命令より先に実行できる. したがって, ロードキューとストアキューの描写の有無は実行パスに影響しないので, 省略した.

メモリディスアンビギュエーションがいつ行われたかは実行パスには影響しないので, 省略した.

4.5.2　ストア・バッファ

本研究ではストア・バッファは実装しない, 代替として, コンパイラによる最適化での機械命令の並び替えで store 命令のオーダリングによるバグを示す.

第 5 章　実装

5.1　概要

本ツールはソースコードの入力部分, コンパイル結果の表示部分, プログラムアニメーションのインタフェースによって構成される (図 4.1).

- ソースコードの入力は文や式などのブロックを組み合わせることで表現するビジュアルプログラミング言語によって行う (図 4.1①).
- コンパイルした結果を命令列として表示する (図 4.1②).
- コンパイル結果およびプログラムアニメーション部分 (図 4.1③) の命令列は枠線で囲った長方形 (以下, ボックス) で表現する.
- 命令列は Tomasulo のアルゴリズムに従った動的パイプラインでアニメーションさせる. アニメーションは GUI による手動の操作によって行う.

5.2　開発環境

html と JavaScript を用いることで, 他のソフトウェアなどをインストールせずに使えるように設計した. また, Google Blockly[*1]によりビジュアルプログラミング部分を作成した. また, html を用いているのでウェブ上に本システムを設置することが可能である. GoogleBlockly を用いた理由は, ビジュアルプログラミング部分の作成が容易になること, および命令の不可分性に合わせてブロックの高さを設定できること (4.4 節) が理由である.

5.3　ビジュアルプログラミング

ブロックの組み合わせによりソースコードを表現する. ビジュアルプログラミング言語による入力部分 (図 4.1①) の左端 (図 5.1①) からブロックを取得し, 白い背景の部分 (図 5.1②) で組み合わせる. ブロックの取得や移動はドラッグ&ドロップで行う.

5.3.1　実装したブロックの種類

実装したブロックについて, 一覧を表 5.1 に示す. 本システムの各命令は不可分であることを基本単位としているため, lock 命令などの実際には複数の機械命令から構成されているものも 1 段分の高さのブロックとして

[*1] https://developers.google.com/blockly/

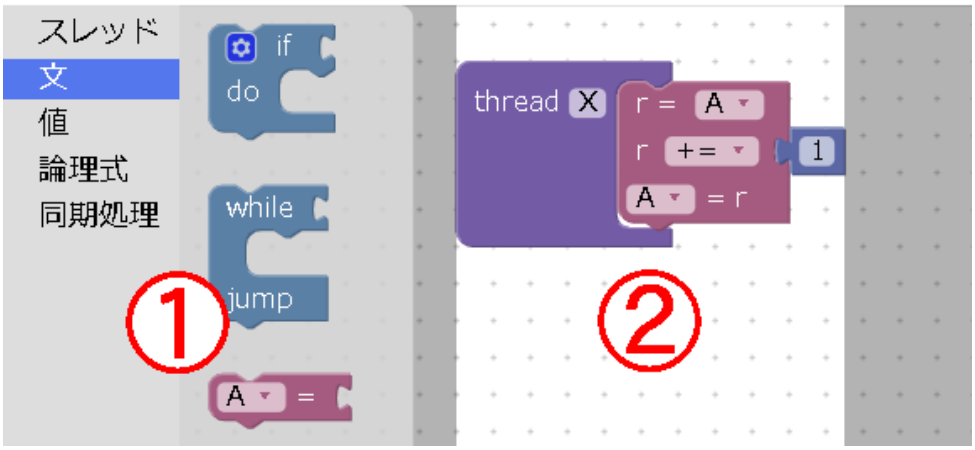

図 5.1: ビジュアルプログラミング言語による入力部分

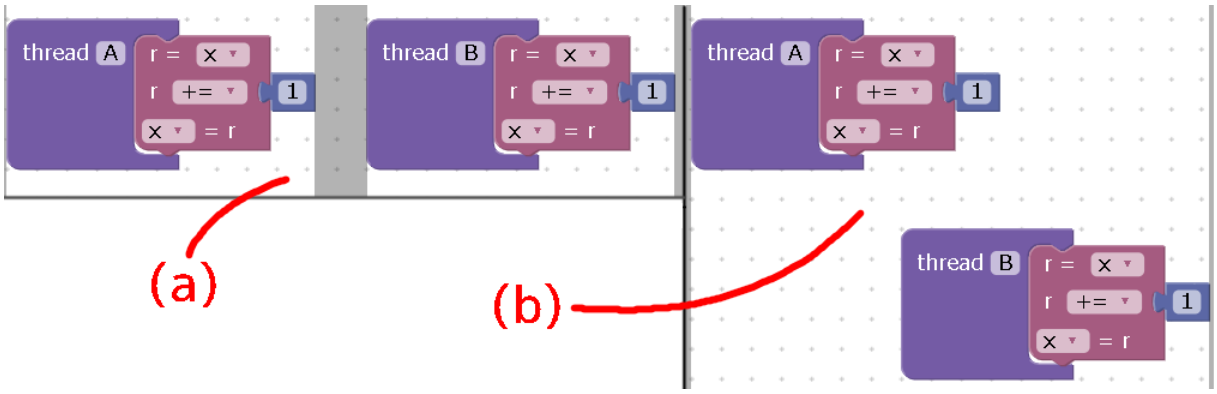

図 5.2: コアへのスレッドの分配: (a) 図の左半分, 異なるコアへのスレッドの配置 (b) 図の右半分, 同じコアへのスレッドの配置

いる.

5.3.2　コアの分配

図 5.2(a) のようにスレッドブロックとその中身が縦方向に重ならない位置に並べるとそれらのスレッドは別々のコアに属するものとして扱い, 間にグレーの領域が描画される. 図 5.2(b) のように縦方向に重なる位置に並べると, それらのスレッドは同じコアに属するものとして扱う. コアの分配と実行時のアニメーションの関係は 5.5.5 項で示す.

5.4　コンパイル

コンパイラによる最適化手法を厳密かつ大域的に行うには, jump 命令などによるプログラムの制御フローを把握し, データの流れを解析してループで不変な値や共通部分式, 無用命令を発見する必要がある [13]. しかし

表 5.1: ブロック一覧

分類	ブロックの種類	備考
スレッド	スレッド	このブロックの中身のみが結果として出力される. このブロックの中身が一つのスレッドの内容として扱われる.
文	代入文	
	四則演算文	リード・モディファイ・ライト操作であるため不可分な三つの操作から成り立つ.
	if 文	条件式には論理式のブロックか, true, false のブロックのみが使える.
	while 文	
値	定数	
	変数	レジスタへの読み込みと読み込んだ値を使う二つの操作から成り立つ.
	true	他の値と異なり, 論理式の代わりに用いる.
	false	
論理式	比較	C 言語において用いられるものと同様の比較演算子を実装している.
	論理積および論理和	二つの論理式に対して論理積や論理和を行う.
	否定	一つの論理式の否定を行う.
同期処理	メモリバリア	アーキテクチャによる動的な実行順序の入れ替えやコンパイラによる最適化の際の命令の並び替えをこのブロックを超えて行わないようにする. 実際にはアーキテクチャやプログラミング言語によっては load 命令とその後の store 命令との順序関係のみを制限するなど限定的な順序制限を提供するものもある [11][12][18] が, 本ツールでは単に全てのメモリアクセス順序を制限するもののみを実装した.
	lock	
	unlock	
	join	

本ツールでは最適化手法として定数伝播と定数の畳み込み, 不変式のループの外への移動, 命令スケジューリングを詳細なデータ解析などを行わずに簡易的に実装した. 簡易的に実装した理由は, コンパイラの最適化はあくまで同期処理をしていないと並列処理でバグを起こすということの再現であり. 厳密に最適化することは本研究の本質とは異なるためである.

これらは個別にオプションとし, ユーザが適用するかどうかを選べるようにした (図 5.3).

5.4.1　定数伝播と定数の畳み込み

定数伝播について, ある変数に対し定数が store される場合は, その後からその変数が変化するまでの間にある参照を定数に置き換える (図 5.4).

if 文がある場合, if 文の前から行われている定数伝播は if 文の中にも伝播し, if 文の中での定数伝播は原則として外部には波及しない. ただし, if 文の分岐の結果がいずれも同じ定数になるのであれば, if 文の後ろにその

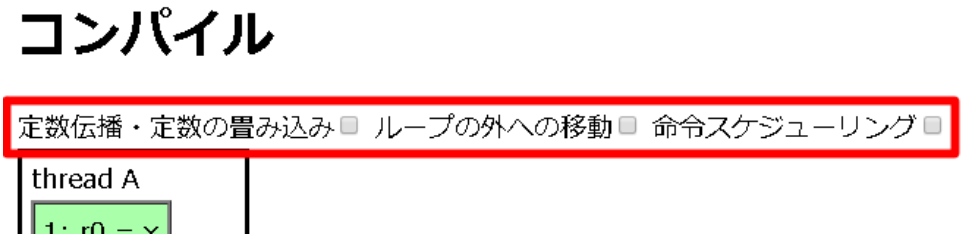

図 5.3: 最適化オプション

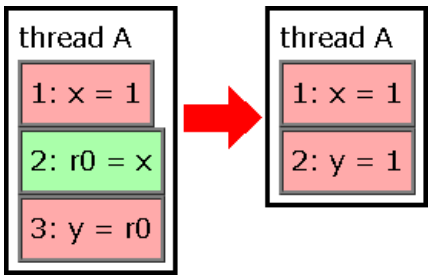

図 5.4: 定数伝播による命令列の変化: 左は最適化前, 右は最適化後

定数を伝播させる.

while 文について, 逐次処理で考えた時に何回ループしてもある変数の値が同じ定数になる場合のみ, while 文開始時点の定数を while 文の内部に伝播させる.

5.4.2　ループの外への移動

無限ループを引き起こす例の再現のため, 条件式のオペランドがループ内で変化しない場合にその条件式をループの前に 1 回のみ行うようにした (図 5.5).

5.4.3　命令スケジューリング

書き込んだ値が直後の命令で参照されている場合のみ, その後にある定数の store 命令の内, 移動しても逐次処理では問題のないものを挿むようにした (図 5.6). 図 5.6 の場合, y に 1 を代入する命令が移動しても逐次処理では問題のない命令に当たるため, $r0 = a$ と $z = r0$ の間に移動している.

5.5　プログラムアニメーション

以下にアニメーションのシステムの詳細を示す. なお, 本システムのアニメーションにおいて全ての変数の初期値は 0 である.

5.5.1　実行オプション

実行オプションを図 5.7 に示す.

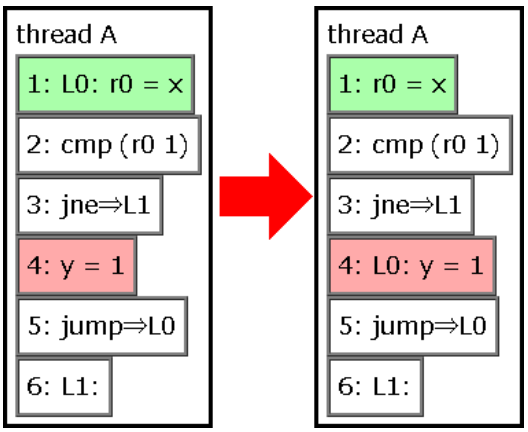

図 5.5: ループの外への移動による命令列の変化 (ソースコードは while($x == 1$) { $y = 1$; }): 左は最適化前, 右は最適化後. ラベル L0 の位置が 1 行目から 4 行目に移動する.

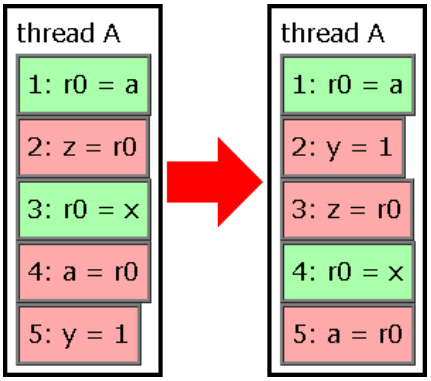

図 5.6: 命令スケジューリングによる命令列の変化: 左は最適化前, 右は最適化後. 移動しても逐次処理としては結果の変わらない $y = 1$ を 5 行目から 2 行目に移動している.

ユーザは SC, TSO, STO の中からアーキテクチャのメモリモデルを一つ選択できる. しかしマルチスレッドの順序制限の名称だけではわかりにくいため, 代表的なプロセッサ名で補足している.

また, OoO 実行を用いるか否かを設定することができる. 単純な競合状態など OoO 実行と関係ないバグの学習を行う際には, OoO 実行を行わずに実行することでアニメーションを単純化できる (5.5.2 項, 5.5.3 項).

他にも作成済みの実行履歴の再現実行 (5.6 節) をする自動実行ボタン, ランダム実行をするボタン, 再現実行やランダム実行の速度設定, 再現実行の繰り返しの有無を設定するチェックボックスを用意している.

5.5.2　インタフェース

実行履歴は薄いグレーの領域 (図 5.8①) に表示される. 右側のタイムスライダー (図 5.8②) を操作することで実行時間を制御し, 命令列 (図 5.8③) のクリックによって命令の実行を行う. 出力結果はレジスタやメモリに反映される (図 5.8④⑤). 実行時間が進むと実行履歴の領域が延長される. タイムスライダーの操作によって実行時間を巻き戻した時, 実行履歴の領域の短縮と実行された命令の回収がされ, 出力結果も巻き戻る.

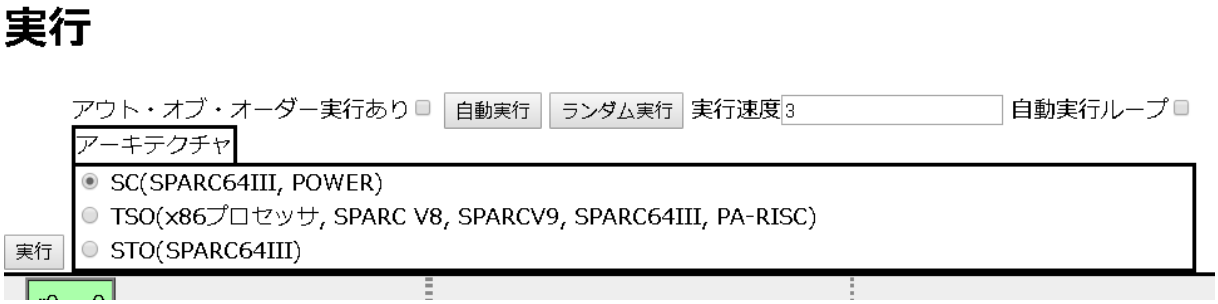

図 5.7: プログラムアニメーションの実行オプション

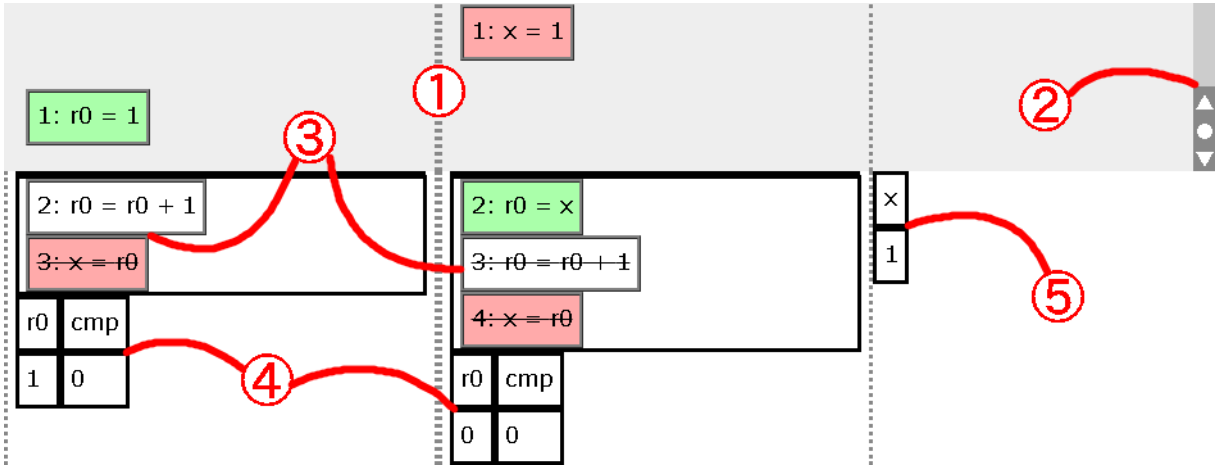

図 5.8: 非 OoO 実行時のアニメーション部分: ①実行履歴の表示領域, ②タイムスライダー, ③命令列, ④レジスタ, ⑤メモリ

実行オプションで OoO 実行を有効にしている場合, リザベーション・ステーションやリオーダ・バッファが描画される (図 5.9①②). 3.3.1 節で示した各ユニットは描画せず, その内部のリザベーション・ステーションとリオーダ・バッファを各コアに一つずつ描画する. リザベーション・ステーションとリオーダ・バッファは実行中のスレッドの命令列の上部にのみ描画される.

5.5.3　実行の手順

OoO 実行を無効にしている場合, 命令列 (図 5.8③) のいずれかのボックスをクリックすると一番上の命令が実行され, 結果が出力される. どのスレッドから実行するかはユーザが自由に選ぶことができる. ただし lock や join で制限されている場合はそれに従う必要があり, 実行できないスレッドの先頭の命令に打ち消し線が引かれる.

OoO 実行を有効にしている場合は以下のような手順で実行を進めることができる.

1. 元の命令列のボックスのいずれかをクリックする. 元の命令列の先頭のボックスが実行履歴に残り (図 5.10①), リザベーション・ステーションとリオーダ・バッファにコピーが生成される (図 5.10②③). この

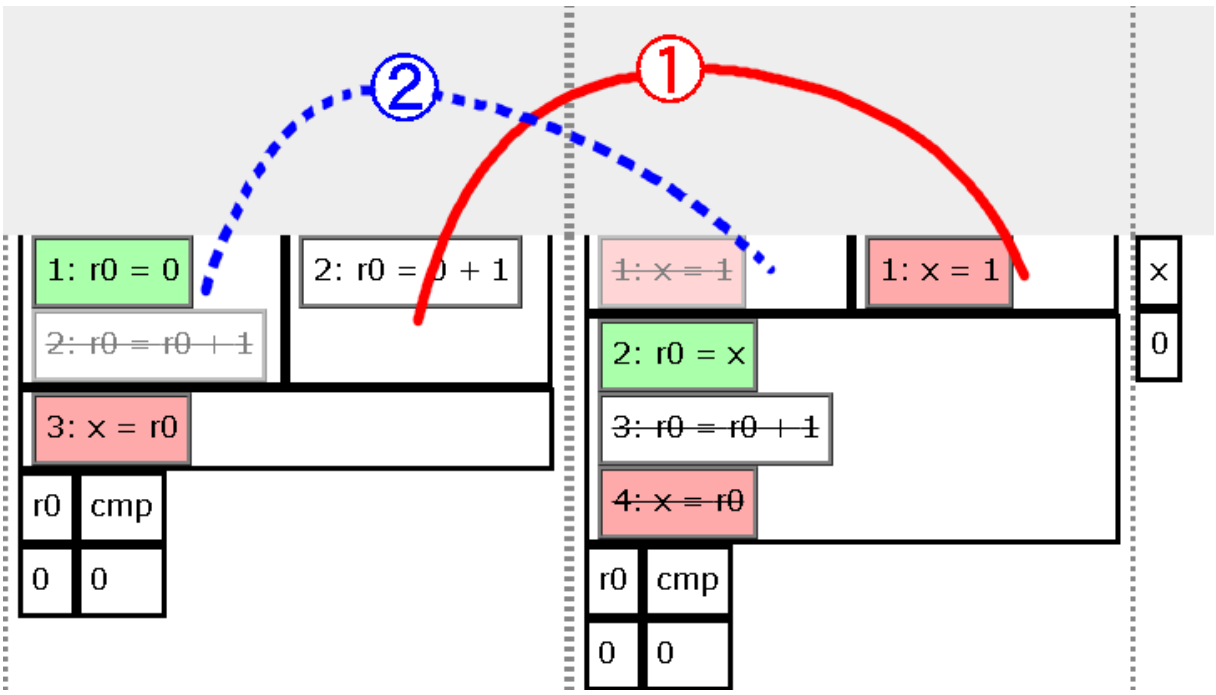

図 5.9: OoO 実行時のアニメーション部分: ①リザベーション・ステーション, ②リオーダ・バッファ

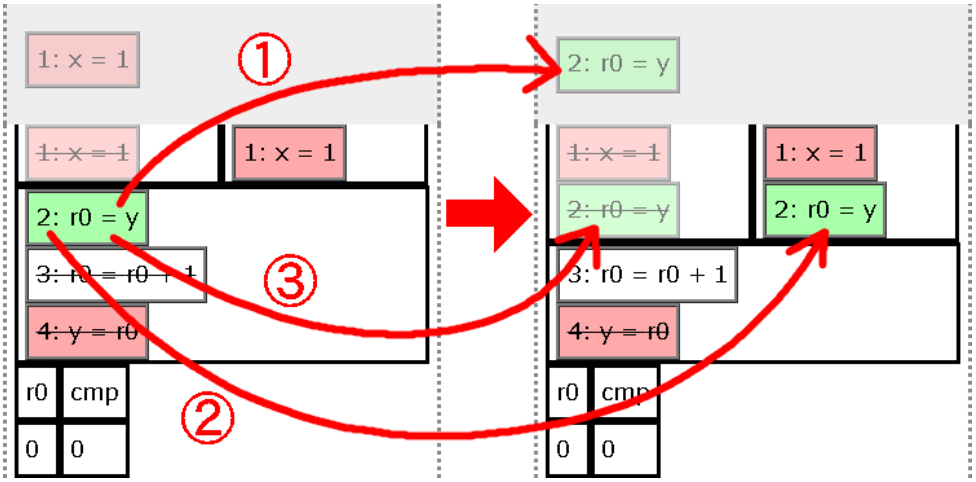

図 5.10: OoO 実行のアニメーション (機械命令のフェッチ・デコード): ①実行履歴への配置, ②リザベーション・ステーションへのコピー, ③リオーダ・バッファへのコピー

操作は Tomasulo のアルゴリズムにおける命令のフェッチ・デコードに当たる.

2. リザベーション・ステーションのボックスをクリックする. クリックされたボックスが実行履歴に残り (図 5.11①), 結果がリオーダ・バッファの同じボックスに書き込まれる (図 5.11②). 同時に, リザベーション・ステーションの適切なオペランドに実行結果が供給される (図 5.11③). この適切なオペランドへの供給により実効的にリネーミングしている. この操作は Tomasulo のアルゴリズムにおける機能ユニッ

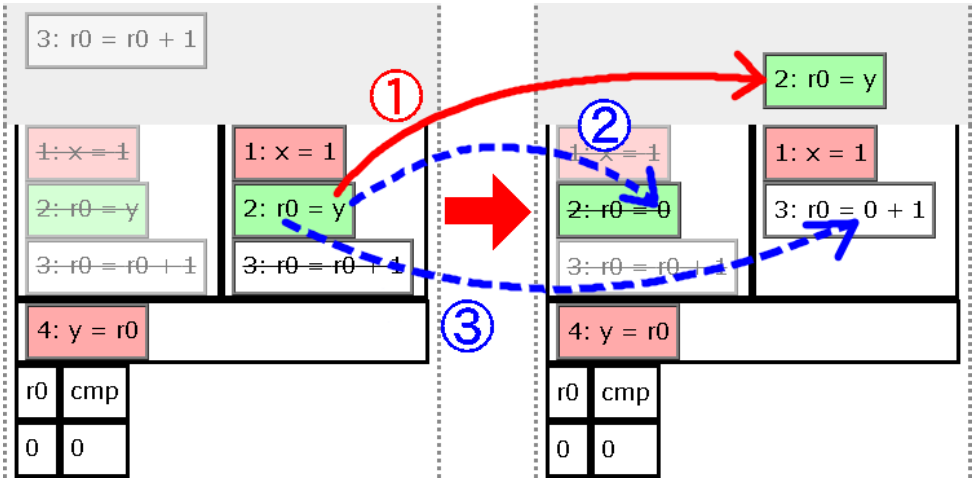

図 5.11: OoO 実行のアニメーション (機械命令の実行): ①実行履歴への配置, ②リオーダ・バッファへの書き込み, ③オペランドへの供給

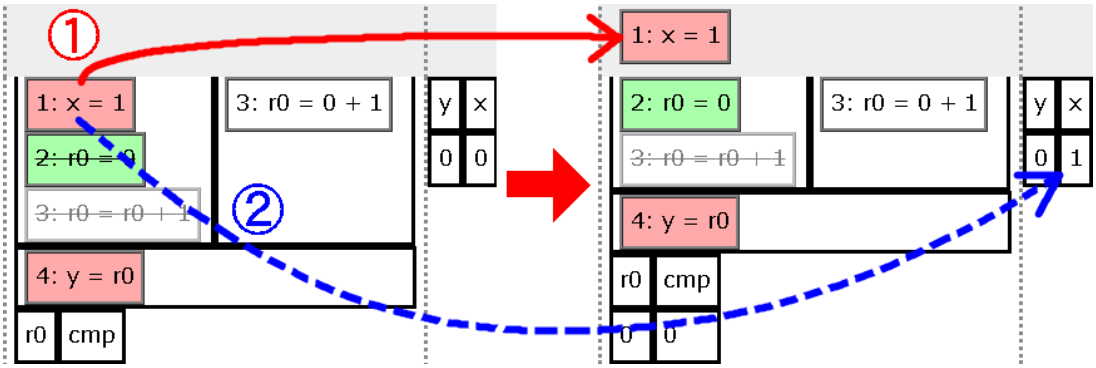

図 5.12: OoO 実行のアニメーション (リタイア): ①実行履歴への配置, ②レジスタやメモリへの結果の反映

トでの命令の実行に当たる.

3. リオーダ・バッファのボックスのいずれかをクリックする. リオーダ・バッファの先頭のボックスが実行履歴に残り (図 5.12①), 実行結果がリタイアされてレジスタまたはメモリに書き込まれる (図 5.12②). この操作は Tomasulo のアルゴリズムにおける確定ユニットからのリタイアに当たる.

以上の手順は一つの命令に対する順序であり, 複数の命令間でこの順序を守る必要はない. 例えばある命令をフェッチ・デコードした後, その命令を実行してもいいし, 次の命令をフェッチ・デコードしても良い (図 5.13). どの命令に対する操作を先に行うかはユーザが選択することができる. ただし, 各命令間の順序が実行オプションで設定したアーキテクチャによる制限を守り, 依存関係のある命令間の順序を守る必要がある. これらに反する命令のボックスには打ち消し線が引かれ, 実行できない.

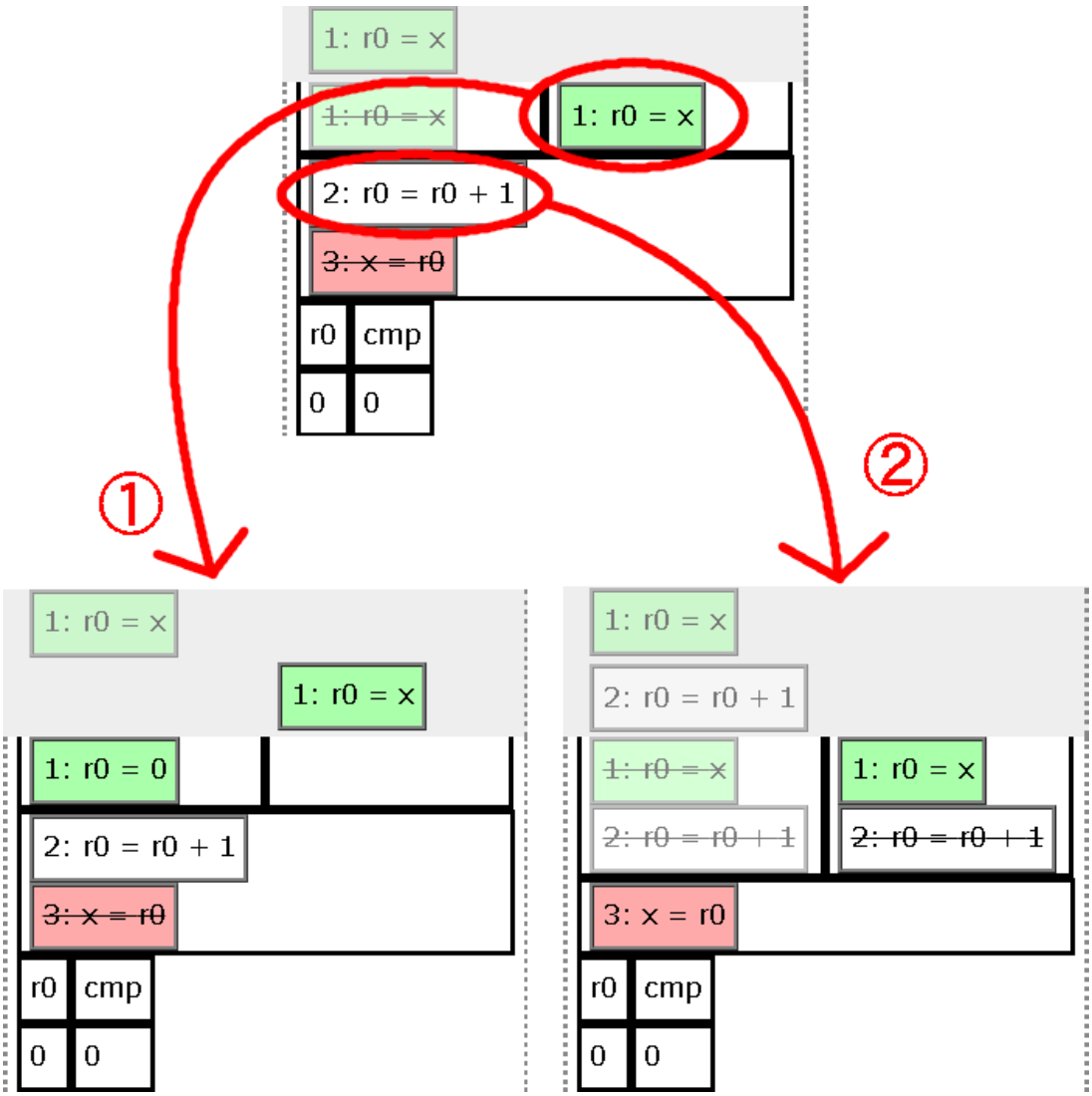

図 5.13: OoO 実行のアニメーション (ユーザの選択の自由): ①先にフェッチ・デコード済みの命令を実行した場合, ②先に次の命令のフェッチ・デコードをした場合

5.5.4　投機的実行

本システムは投機的実行のアニメーションを実装している. OoO 実行を有効にしている場合, if 文や while 文の条件式に対して用いられる. 図 5.14 のように投機的実行をしている命令のボックスは枠線が点線で表示される. 分岐の予測はジャンプをしない場合が真であると仮定している. 分岐の予測が正しいことが確定すると, ボックスの枠線は実線に戻る. 分岐の予測が誤っていることが確定すると, 投機的実行をされていたボックスは削除される. ただし, 既に実行履歴に置かれたボックスは枠線はそのままで削除もされない.

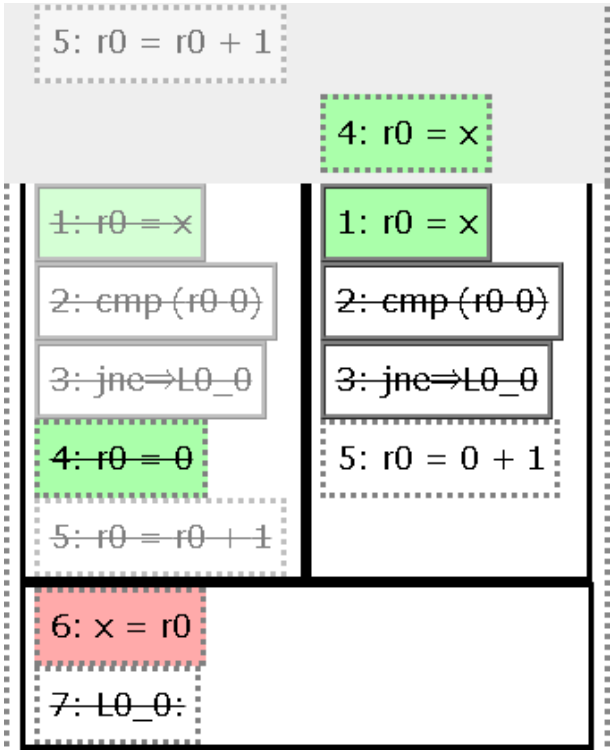

図 5.14: 投機的実行のアニメーション: 3 行目の jne 命令から先が投機的実行であるため, ボックスの枠線が点線になっている.

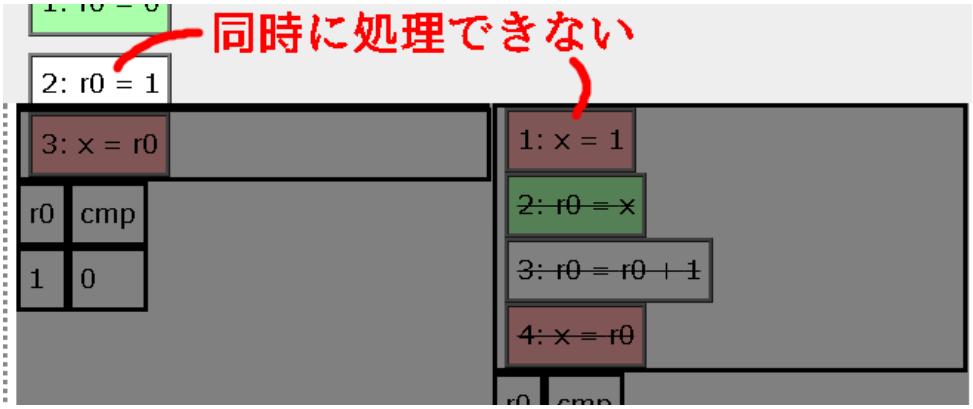

図 5.15: 並行処理のアニメーション

5.5.5　並列性

同じコアに属するスレッドは並行処理になる (図 5.15). 別々のコアに属するスレッドは並列処理になる (図 5.16). 4.3 節で示した通り, 並列処理ならば同時実行ができるが, 並行処理では同時に実行できない.

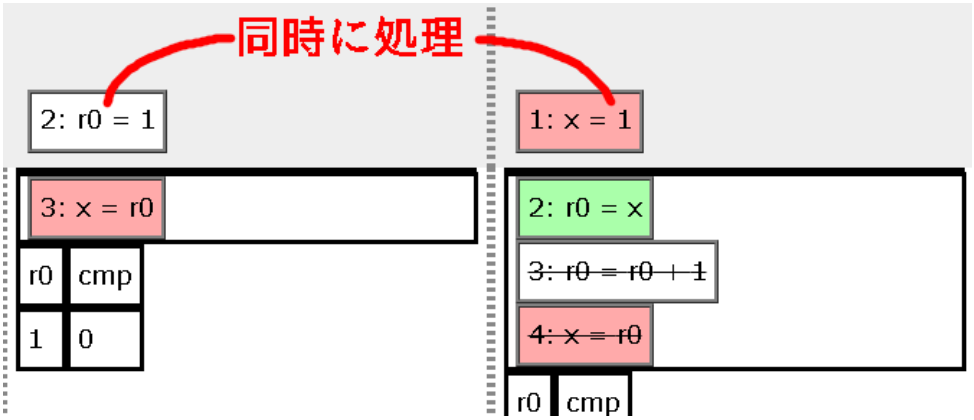

図 5.16: 並列処理のアニメーション

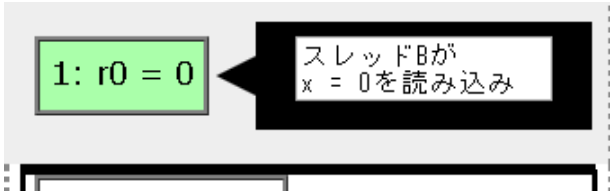

図 5.17: 注釈

5.6　実行履歴の保存と再現

プログラムアニメーションにおいて, 実行履歴の保存と保存された実行履歴に基づくアニメーションの再現を実装している. この自動実行には一時停止と再生の機能がある. 実行履歴の再現機能によってバグのサンプルをソースコードだけでなく実行履歴と共に保存することができる.

5.6.1　注釈

プログラムアニメーションの実行中に右クリックで注釈を書くことができる (図 5.17). 実行履歴の保存を行ったユーザとは別のユーザが実行履歴を再生する場合, そのユーザは注釈を読むことで, 実行履歴を保存したユーザがどのような意味を込めて実行履歴を作成したのかを知ることができる.

注釈は黒い吹き出しとして表示される. 注釈はドラッグで移動させることができ, 入力した文字や改行に合わせて自動的に大きさが変化する. 注釈は実行履歴と共に保存され, アニメーションの再現時に注釈を書いたのと同じタイミングで同じ場所に表示される.

第 6 章　評価

6.1　バグの再現機能の検証

まず, 本ツールは理解支援をするために, マルチスレッドプログラムに発生する各種バグを再現できる必要がある.

そこで各種バグについて, 本ツールによるアニメーションでバグが再現できる例を考案し, アニメーションさせて確認した. 以下, 再現したバグを示す.

6.1.1　競合状態

競合状態について, リード・モディファイ・ライト操作とチェック・ゼン・アクト操作の二つを確かめた.

6.1.1.1　リード・モディファイ・ライト操作

まずリード・モディファイ・ライト操作について, $x+=1$ を行う二つのスレッドによって発生する競合状態を確かめた. ただし, 初期値は $x = 0$ とする. これは 1.2 節で示した競合状態の例と同一のものである. ビジュアルプログラミング部分には図 6.1 のように入力した. コンパイラによる最適化と OoO 実行はバグの原因ではないので, 最適化はすべて無効にし, 実行オプションでは OoO 実行を無効にした. コンパイル結果を図 6.2, リード・モディファイ・ライト操作による競合状態が確認できる実行結果を図 6.3 に示す. よって, 本ツールでリード・モディファイ・ライト操作を示すことができた.

6.1.1.2　チェック・ゼン・アクト操作

チェック・ゼン・アクト操作は条件式のチェックとその結果に合わせた実行が不可分でないために発生するバグである. 例を図 6.4 に示す. ただし, 初期値は $x = y = 0$ とする. この例では x が 0 であれば y を 1 に, y が 0 であれば x を 1 にするようにしている. プログラマの意図としてはどちらか片方だけが 1 になることを期待している. しかし実際には条件式をチェックしてから代入を行うまでの間にもう片方のスレッドが条件式を

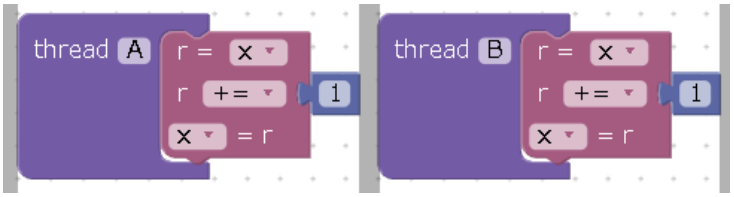

図 6.1: リード・モディファイ・ライト操作の例: ソースコード

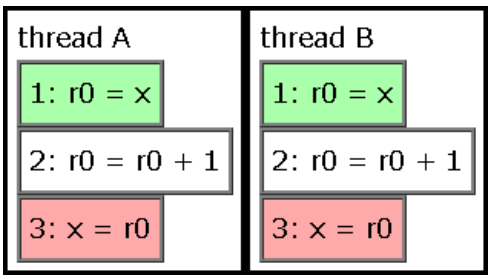

図 6.2: リード・モディファイ・ライト操作の例: コンパイル結果

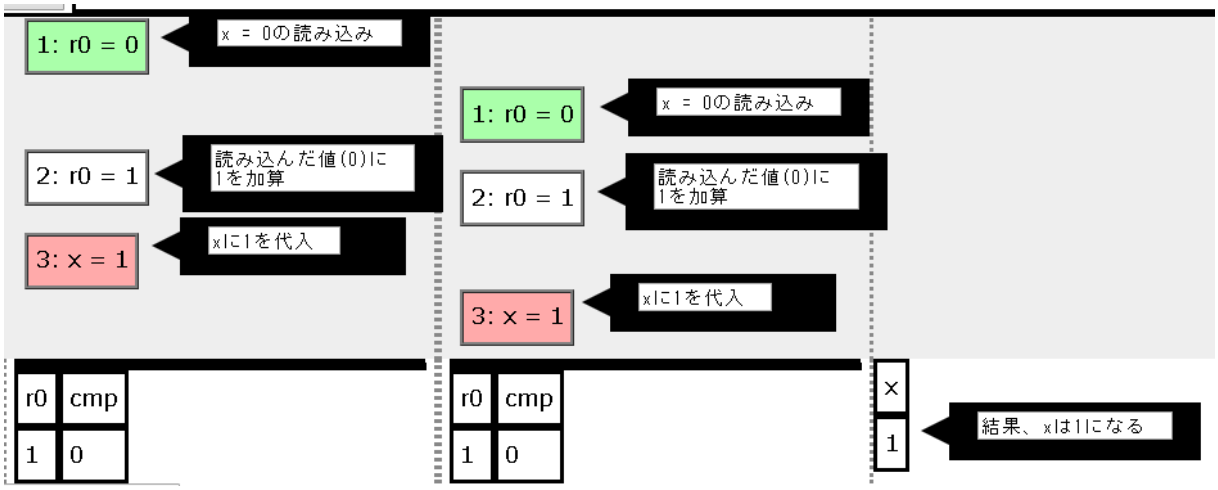

図 6.3: リード・モディファイ・ライト操作の例: 実行結果

	スレッド A	スレッド B
1	if (x == 0) {	if (y == 0) {
2	y = 1;	x = 1;
3	}	}

図 6.4: チェック・ゼン・アクト操作の例

チェックしてしまう可能性があるため, 両方とも 1 になる可能性がある. この例を本ツールで確かめる. 先と同様にコンパイラによる最適化と OoO 実行はバグの原因ではないので, 最適化はすべて無効にし, OoO 実行を無効にした. ビジュアルプログラミング部分への入力を図 6.5, コンパイル結果を図 6.6, チェック・ゼン・アクト操作による競合状態が確認できる実行結果を図 6.7 に示す. よって, 本ツールでチェック・ゼン・アクト操作を確かめることができた.

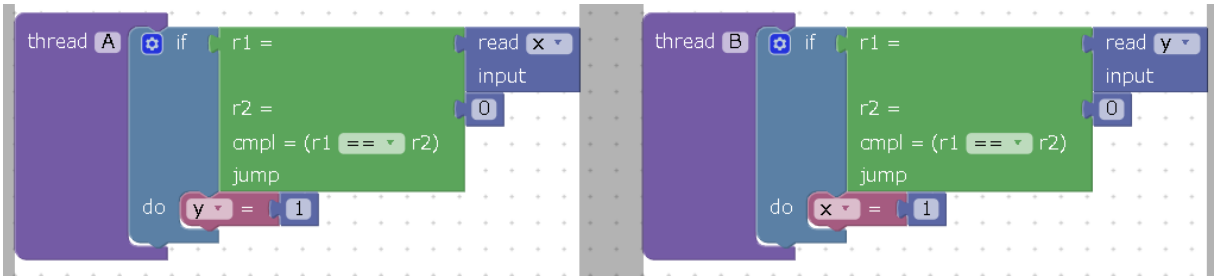

図 6.5: チェック・ゼン・アクト操作の例: ソースコード

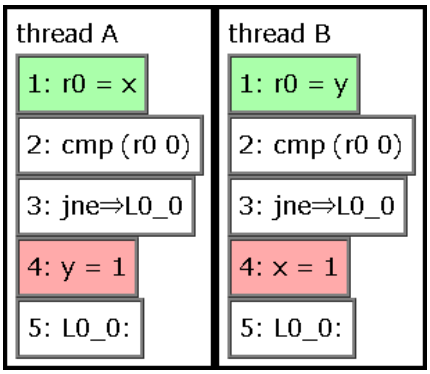

図 6.6: チェック・ゼン・アクト操作の例: コンパイル結果

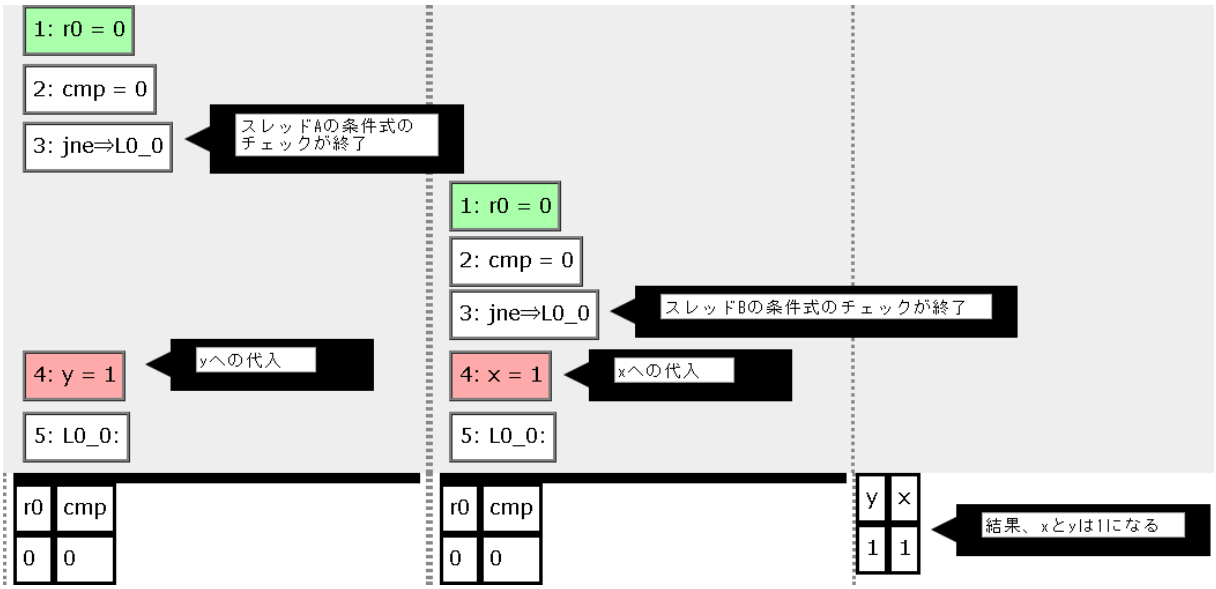

図 6.7: チェック・ゼン・アクト操作の例: 実行結果

	スレッド A	スレッド B
1	$x = 1;$	$y = 1;$
2	if($y == 1$) {	if($x == 1$) {
3	$y = 2;$	$x = 2;$
4	}	}

図 6.8: load 命令が先行する store 命令より先に実行されることで想定されない挙動が起こるプログラムの例

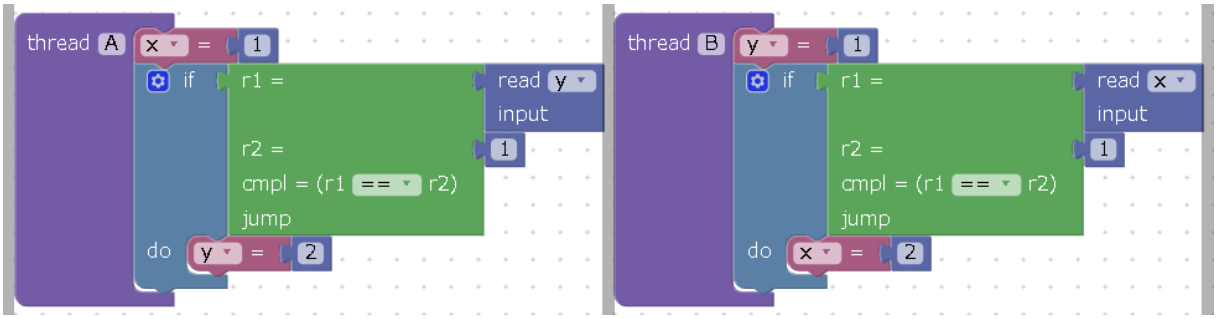

図 6.9: load 命令が先行する store 命令より先に実行される例: ソースコード

6.1.2　メモリの可視性: OoO 実行

メモリの可視性について, まずは OoO 実行によって発生するバグについて検証した.

6.1.2.1　load 命令と先行する store 命令の順序の入れ替え

load 命令が先行する store 命令よりも先に実行することで想定外の挙動をする場合を考える. 例えば図 6.8 のようなプログラムである. このプログラムではコンパイラによる最適化や OoO 実行を考慮しない場合, いずれかのスレッドの if 文の条件式がチェックされた時, そのスレッドの代入は完了している. したがって, 少なくとも片方は if 文の中身が実行されるように見える. しかし TSO など load 命令が先行する store 命令より先に実行できるメモリモデルでは, 代入より先に if 文の条件式にある変数の読み込みを行うことができるので, if 文の中身がどちらも実行されない可能性がある.

この例を本ツールで確かめた. コンパイラによる最適化はこの例では考慮しないので, 最適化はすべて無効にした. OoO 実行は有効にした. ハードウェアのメモリモデルは TSO を選択する. ビジュアルプログラミング部分への入力を図 6.9, コンパイル結果を図 6.10, どちらの if 文も実行されない挙動が確認できる実行結果を図 6.11 に示す. よって, 本ツールで load 命令が先行する store 命令より先に実行されることで想定外の挙動をする場合を確かめることができた.

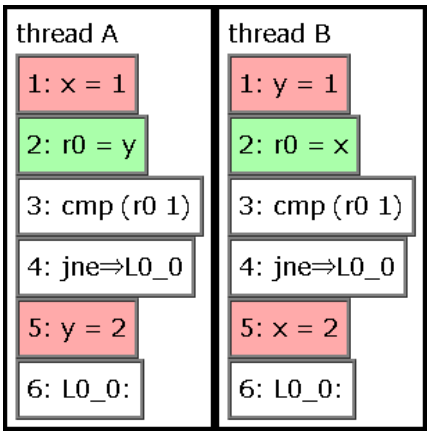

図 6.10: load 命令が先行する store 命令より先に実行される例: コンパイル結果

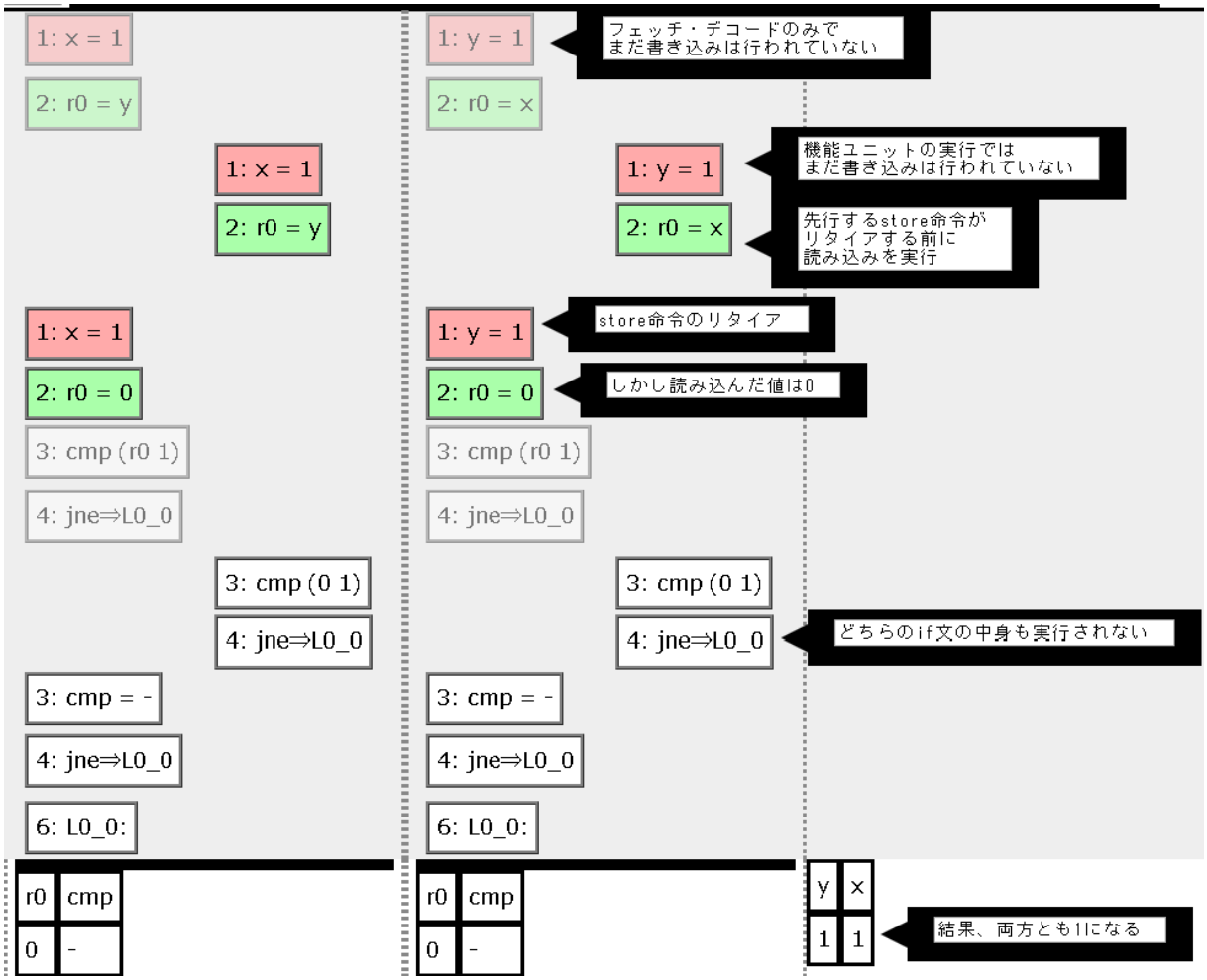

図 6.11: load 命令が先行する store 命令より先に実行される例: 実行結果: 各スレッドの 2 行目の load 命令が 1 行目の store 命令より先に実行されている

	スレッド A	スレッド B
1	$x = 1;$	$z = y;$
2	$y = 2;$	$z \mathrel{+}= x$

図 6.12: load 命令同士の順序が守られないことで想定されない挙動が起こるプログラムの例

6.1.2.2　load 命令同士の順序の入れ替え

load 命令同士の実行順序が入れ替わることで想定外の挙動をする場合を考える. 例えば図 6.12 のようなプログラムである. ただし, 初期値は $x = y = z = 0$ とする. この例について, コンパイラによる最適化や OoO 実行を考えない場合, スレッド A は x への代入の後で y への代入を行っており, スレッド B は y の値を読み込んで z へ代入してから x の値を z に加算しているので, スレッド B が読み込んだ y の値がスレッド A が書き込んだ値であれば, スレッド B が読み込んだ x の値はスレッド A が書き込んだ値である. したがって, z の値は初期値の合計である 0 か, x だけが代入後の値である場合の 1 か, 両方とも代入後の値である 3 のいずれかになるように見える. しかし, 先行する store 命令より先に load 命令を実行でき, かつ load 命令同士の順序が守られない STO などのメモリモデルでは, y の読み込みと x の読み込みの順序が逆になり, z が 2 になる場合がある.

この例を本ツールで確かめた. コンパイラによる最適化はこの例では考慮しないので, 最適化はすべて無効にした. OoO 実行は有効にした. ハードウェアのメモリモデルは STO を選択する. ビジュアルプログラミング部分への入力を図 6.13, コンパイル結果を図 6.14, z が 2 になる挙動を確認できる実行結果を図 6.15 に示す. よって, 本ツールで load 命令同士が順序を守らないことで想定外の挙動をする場合を確かめることができた.

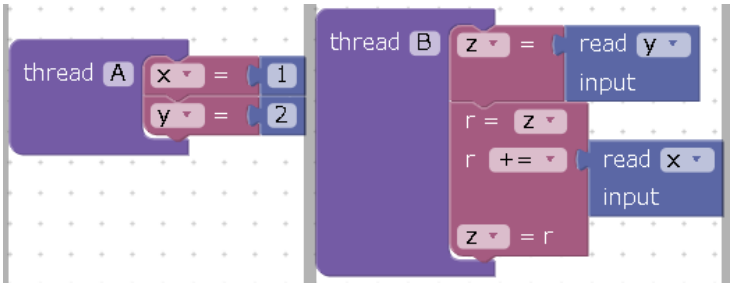

図 6.13: load 命令同士の順序が逆に実行される例: ソースコード

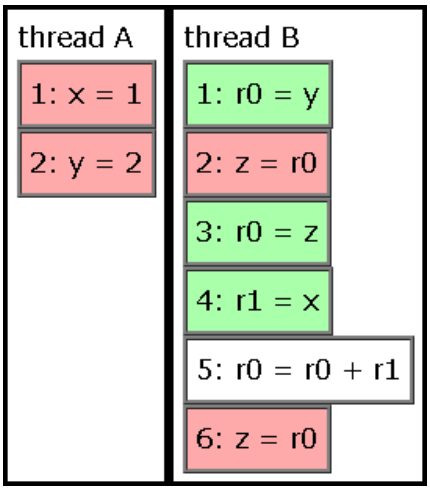

図 6.14: load 命令同士の順序が逆に実行される例: コンパイル結果

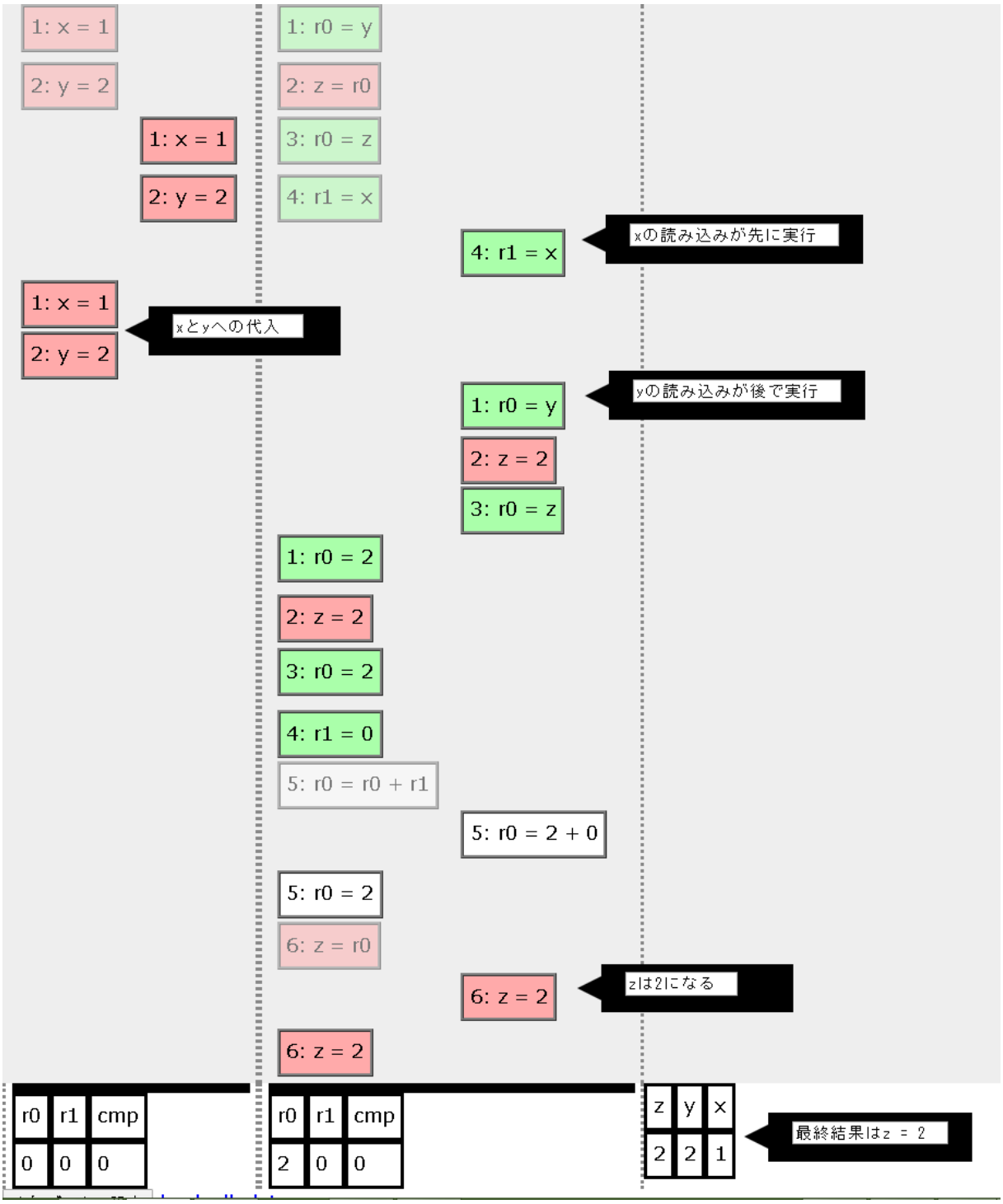

図 6.15: load 命令同士の順序が逆に実行される例: 実行結果

	スレッド A	スレッド B
1	$x = 1$;	if($y == 1$) {
2	メモリバリア	$x += 2$;
3	$y = 1$;	}

図 6.16: 投機的実行によりバグが起こるプログラムの例

6.1.2.3　投機的実行

投機的実行によりバグが発生する場合を考える. 例えば図 6.16 のようなプログラムでバグが起きる. ただし, 初期値は $x = y = 0$ とする. スレッド B は y が 1 であれば x に 2 を加算し, スレッド A は x に 1 を代入した後 y に 1 を代入するので, 投機的実行を考えない場合, x は 1 か 3 にしかならない. しかし, OoO 実行と投機的実行を考慮した場合, load 命令同士の実行順序に制限を設けないメモリモデルであれば, 投機的実行により if 文の中にある x の読み込みを先に実行して x へ書き込む値を 2 としてからスレッド A を実行し, そのあとでスレッド B の条件式をチェックすることができる. この時, if 文の中では x の初期値が読み込まれているにもかかわらず, 条件式では代入後の y を参照しているため if 文の中身が実行され, x の最終結果が 2 になる.

この例を本ツールで確かめた. コンパイラによる最適化はこの例では考慮しないので, 最適化はすべて無効にした. OoO 実行は有効にした. ハードウェアのメモリモデルは STO を選択する. ビジュアルプログラミング部分への入力を図 6.17, コンパイル結果を図 6.18, x が 2 になるバグを確認できる実行結果を図 6.19 に示す. よって, 本ツールで投機的実行でバグが発生する場合を確かめることができた.

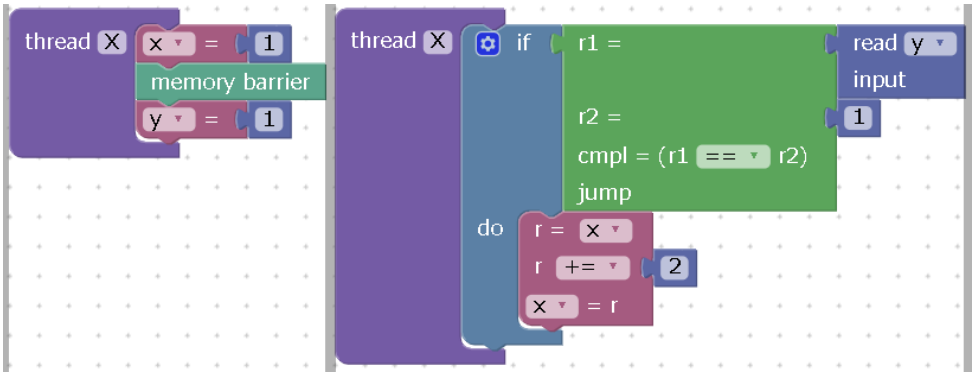

図 6.17: 投機的実行でバグが発生する例: ソースコード

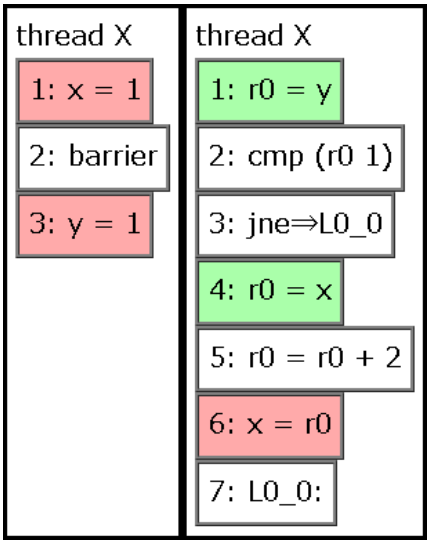

図 6.18: 投機的実行でバグが発生する例: コンパイル結果

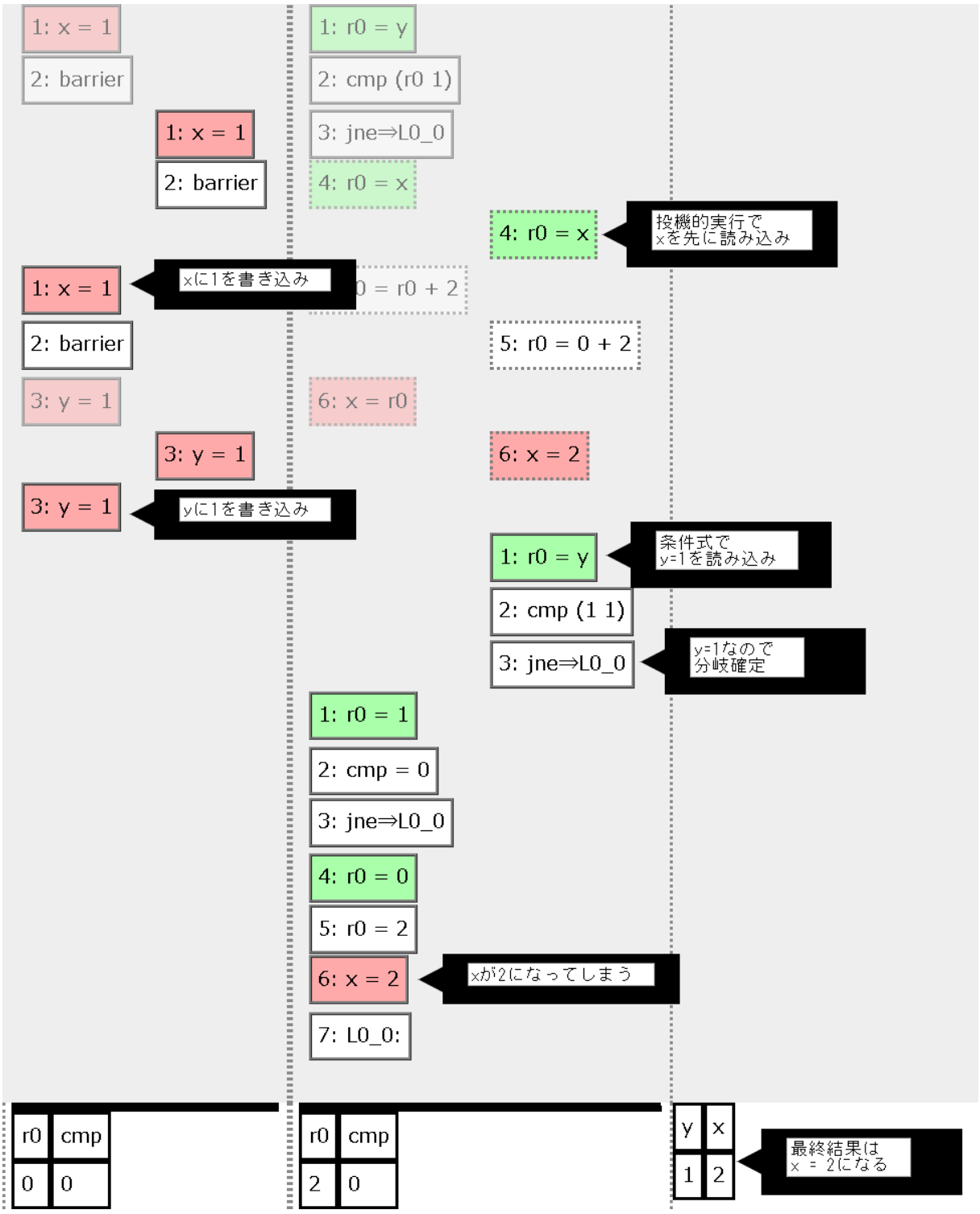

図 6.19: 投機的実行でバグが発生する例: 実行結果

	スレッド A	スレッド B
1	x += a;	z = y;
2	y = 2;	メモリバリア
3		z += x;

図 6.20: store 命令が同士の順序が入れ替わることで想定外の挙動が発生するプログラムの例

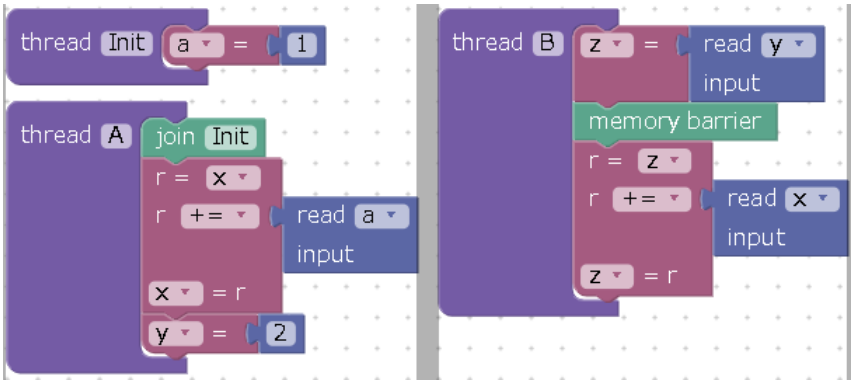

図 6.21: store 命令が同士の順序が入れ替わることで想定外の挙動が発生する例: ソースコード

6.1.3　メモリの可視性: コンパイラによる最適化

次に, コンパイラによる最適化によって想定外の挙動が発生する場合を考える.

6.1.3.1　store 命令同士の順序の入れ替え

コンパイラによる並び替えによって store 命令同士の実行順序が入れ替わり, 想定外の挙動が発生する場合を考える. 例えば図 6.20 のようなプログラムを考える. ただし初期値は $x = y = z = 0, a = 1$ とする. これは図 6.12 のスレッド B にメモリバリアを加え, スレッド A の x への代入を加算にしたものである. したがって load 命令同士の並び替えは起こらない. しかし命令スケジューリングや動的な実行順序の並び替えによって y への書き込みが x への書き込みより先に実行されてしまう可能性があり, 最終結果が $z = 2$ になる場合がある.

この例を本ツールで確かめた. まず図 6.20 のプログラムに初期値を代入するための処理とスレッドを加えた図 6.21 のようなプログラムを入力する. 最適化は命令スケジューリングのみにして, OoO 実行は無効にする. 最適化前のコンパイル結果は図 6.22, 最適化後は図 6.23 のようになる. 最適化前後で $y = 1$ の位置が変化しているのがわかる. 実行結果は図 6.24 のようになり, $z = 2$ になる実行パスがあることが確認できた.

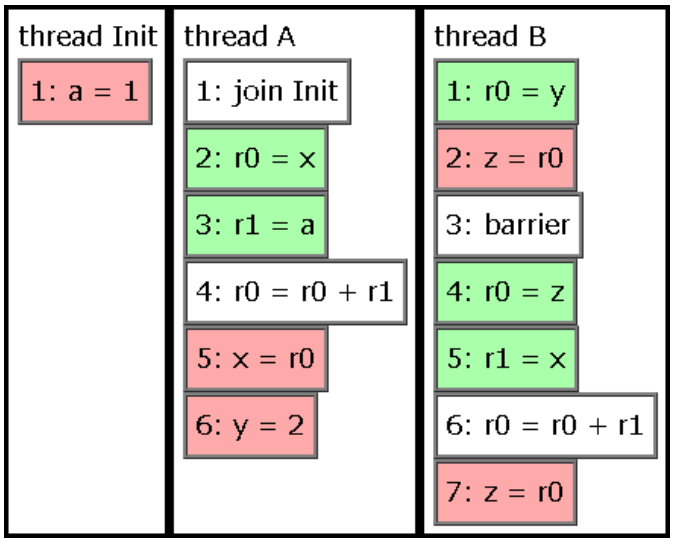

図 6.22: store 命令が同士の順序が入れ替わることで想定外の挙動が発生する例: コンパイル結果 (最適化前):
スレッド A の 5 行目, $x = r0$ は 6 行目の $y = 2$ より先に実行される.

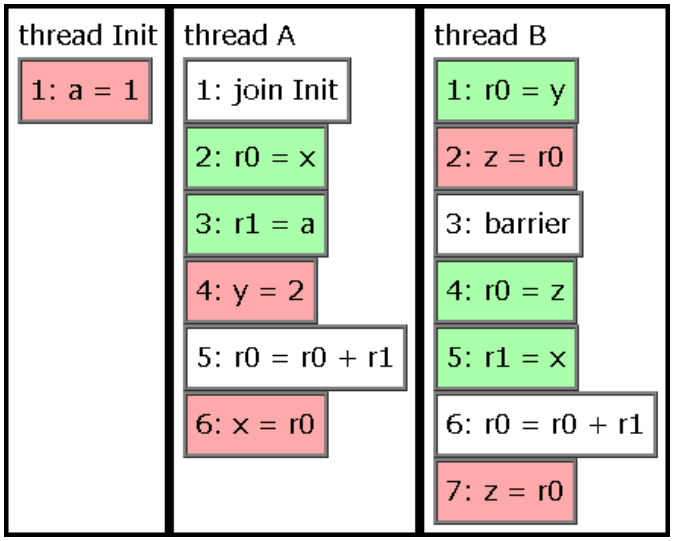

図 6.23: store 命令が同士の順序が入れ替わることで想定外の挙動が発生する例: コンパイル結果 (最適化後):
スレッド A の 6 行目, $x = r0$ は 4 行目の $y = 2$ より後に実行される.

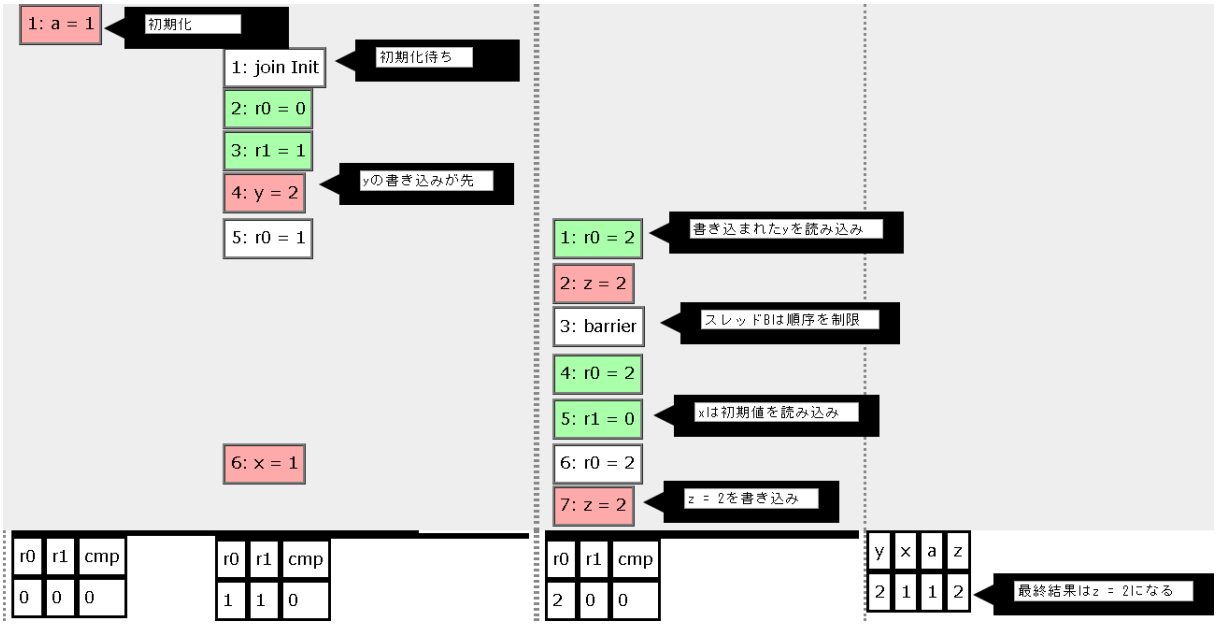

図 6.24: store 命令が同士の順序が入れ替わることで想定外の挙動が発生する例: 実行結果

	スレッド A	スレッド B
1	$z = a;$	$b = y;$
2	$a = x;$	$x = 1;$
2	$y = 1;$	

図 6.25: store 命令が先行する load 命令より先に実行されることで想定外の挙動が発生するプログラムの例

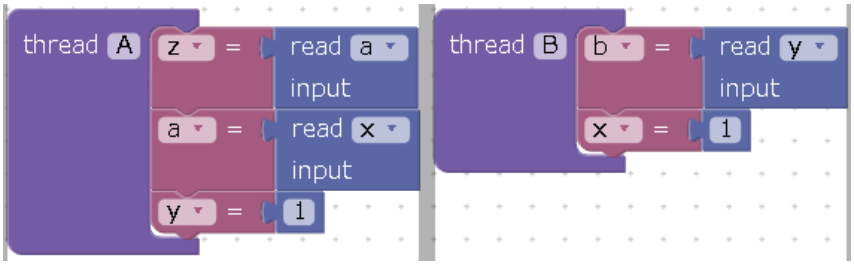

図 6.26: store 命令が先行する load 命令より先に実行されることで想定外の挙動が発生する例: ソースコード

6.1.3.2　store 命令と先行する load 命令の順序の入れ替え

コンパイラによる並び替えによって store 命令が先行する load 命令より先に実行され, 想定外の挙動が発生する場合を考える. 例えば図 6.25 のようなプログラムで想定外の挙動が発生する. 初期値は $a = b = x = y = z = 0$ とする. コンパイラによる最適化や OoO 実行を考慮しない場合, スレッド A は x の読み込み, y への書き込みの順, スレッド B は y の読み込み, x への書き込みの順で動作する. したがって a と b の内の片方が 1 になった時, もう片方は 0 になっているように見える. しかし命令スケジューリングや動的な実行順序の並び替えによって store 命令が load 命令より先に実行される可能性があるため, 実際には a と b の両方が 1 になる可能性がある.

この例を本ツールで確かめた. ビジュアルプログラミング部分への入力を図 6.26, 最適化前のコンパイル結果を図 6.27, 最適化後のコンパイル結果を図 6.28 に示す. スレッド A の $y = 1$ が load x より先になっていることがわかる. a と b の両方が 1 になる挙動を確認できる実行結果を図 6.29 に示す. よって, 本ツールで投機的実行でバグが発生する場合を確かめることができた.

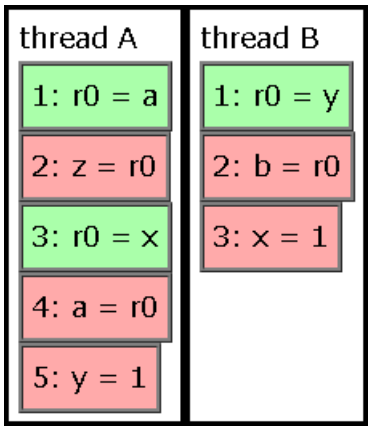

図 6.27: store 命令が先行する load 命令より先に実行されることで想定外の挙動が発生する例: コンパイル結果 (最適化前):
スレッド A の 3 行目, $r0 = x$ は 5 行目の $y = 1$ より先に実行される.

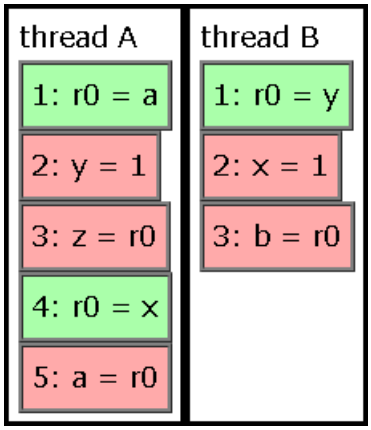

図 6.28: store 命令が先行する load 命令より先に実行されることで想定外の挙動が発生する例: コンパイル結果 (最適化後):
スレッド A の 4 行目, $r0 = x$ は 2 行目の $y = 1$ より後に実行される.

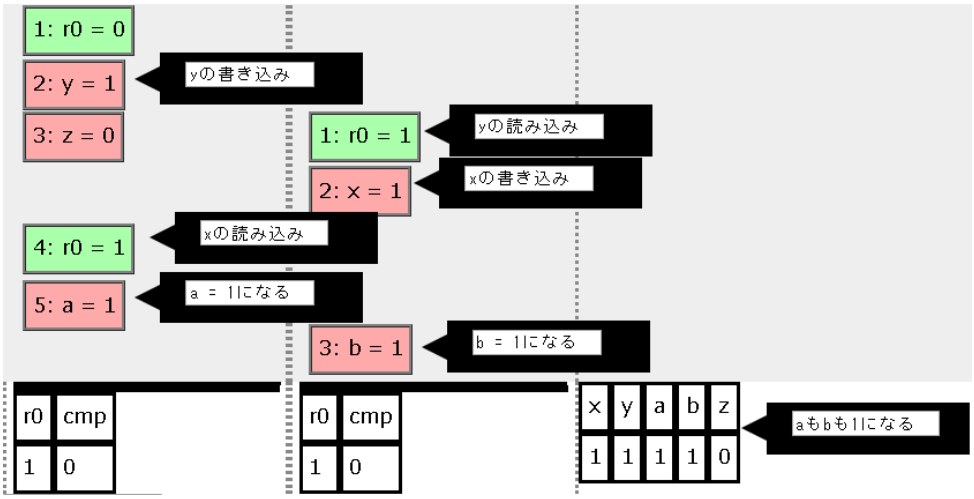

図 6.29: store 命令が先行する load 命令より先に実行されることで想定外の挙動が発生する例: 実行結果

	スレッド A	スレッド B
1	$x = 1$;	while(x != 1) {
2		y += 1;
3		}

図 6.30: ループの条件式がループの外へ出ることでバグが起こるプログラムの例

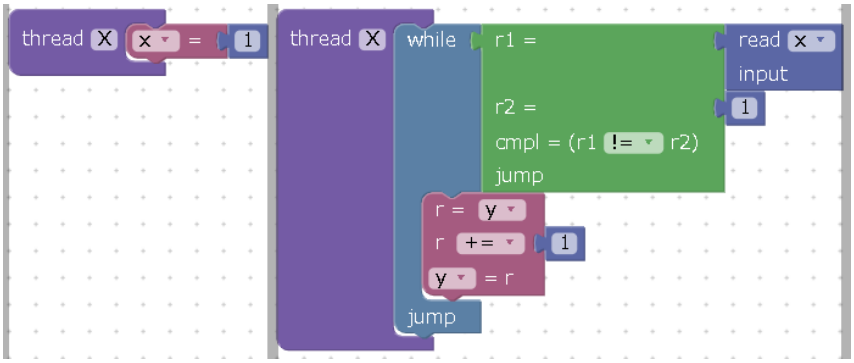

図 6.31: 条件式のループの外への移動でバグが発生する例: ソースコード

6.1.3.3　ループの外への移動

条件式がループの外へ移動されることで発生するバグを考える. 3.2 節で示した例を考える (図 6.30). ただし, 初期値は $x = y = 0$ とする. スレッド B はスレッド A が代入を行うまでループを続けるように見えるが, コンパイラによる最適化によって条件式が最初の 1 回のみチェックするようになっていた場合, 最初のチェックの時点でスレッド A による書き込みが完了していなければ, 無限ループが発生する.

この例を本ツールで確かめた. 簡単化のため, 最適化は条件式のループの外への移動のみにし, OoO 実行は無効にした. ビジュアルプログラミング部分への入力を図 6.31, 最適化前のコンパイル結果を図 6.32, 最適化後のコンパイル結果を図 6.33 に示す. 最適化によってラベルの位置が変わり, 条件式のチェックが最初の 1 回しか行われないようになっていることがわかる. 実際に無限ループが起きていることを確認できる実行結果を図 6.34 に示す. よって, 本ツールで条件式のループの外への移動でバグが発生する場合を確かめることができた.

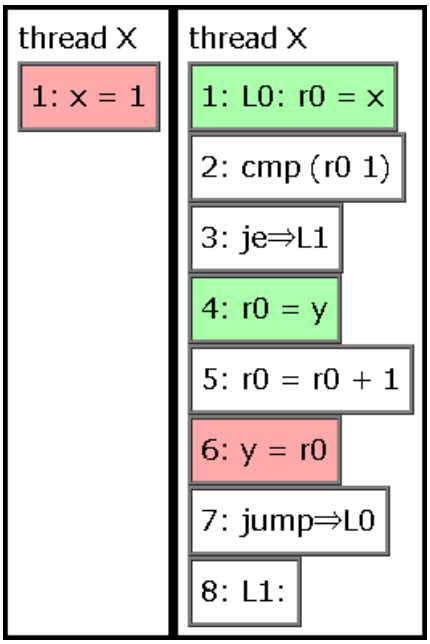

図 6.32: 条件式のループの外への移動でバグが発生する例: コンパイル結果 (最適化前): スレッド B のラベル L0 は条件式の直前である 1 行目に来ている.

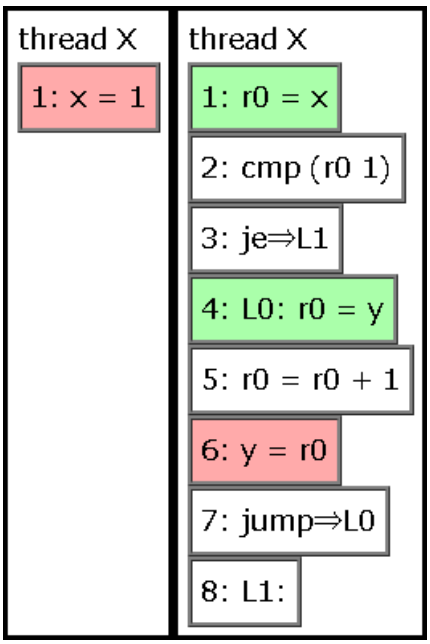

図 6.33: 条件式のループの外への移動でバグが発生する例: コンパイル結果 (最適化後): スレッド B のラベル L0 は条件式の直後である 4 行目に来ている.

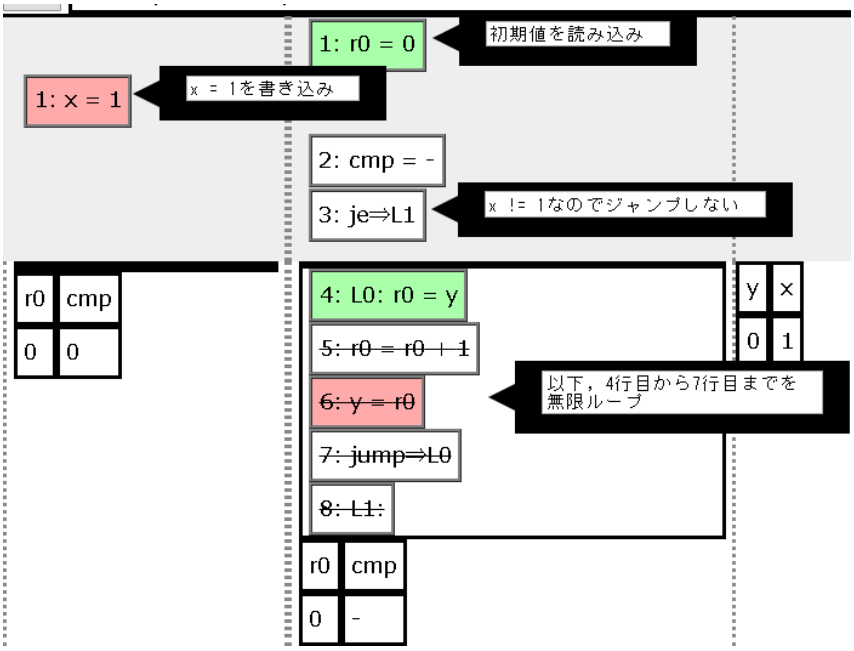

図 6.34: 条件式のループの外への移動でバグが発生する例: 実行結果

6.2　実験 1: インタフェースの評価

本システムのインタフェースはバグを再現するのに適切かどうかを評価する目的で実験を行った. 被験者は本学の学生 6 人である.

6.2.1　実験方法

本ツールの説明と実演を行い, 本ツールでバグを再現してもらった. 再現してもらうバグは, 6.1.1 項で示したリード・モディファイ・ライト操作によるバグと同様である. なお, OoO 実行はオプションで無効にした. 再現後, アンケートを取った. アンケート内容はバグの再現ができたかどうかおよび動作が快適だったかどうかである. バグの再現については, 変数 x が最終的に 1 になったことを被験者自身に確認してもらった. 動作の快適さは快適, 思ったように動かないことがある, 全く思ったように動かないのいずれに当てはまるかを答えてもらった.

6.2.2　結果

6 人全員がバグの再現に成功した. 動作の快適さは 5 人が快適であると答えたが 1 人は思ったように動かないことがあると答えた. その理由として応答性は良好であるがアニメーション部分のタイムスライダーの操作に難があるとのことだった. 他にも実行結果代入後に実行履歴に残るボックスに代入前の変数名も表示されるとわかりやすいかもしれない, 初期値である 0 という値を各スレッドの始めに改めて入力するか悩んだとの意見があった.

6.2.3　考察

インタフェースはバグの再現をするのに十分であることがわかったが, 操作性に多少の難があった. タイムスライダーの癖を修正する必要がある. 途中経過について, OoO 実行を無効にしている場合, リタイア後のボックスのみが残り, 命令のフェッチ・デコード時のボックスや実行結果のボックスが省略されるため, 途中経過がわかりにくくなったと考えられる. したがって, 特に OoO 実行を無効にしている時, 代入前の変数名も実行履歴に残るボックスに表示することでレイアウトを改善できると考えられる. また, 初期値がユーザにわかりにくいので, 初期値の表示と入力が必要であることも分かった.

6.3　実験 2: 有効性の評価

本ツールで並列プログラミングへの理解を促せるかどうかを評価する目的で実験を行った. 被験者は実験 1 と同一で, 実験 1 の直後に行った.

6.3.1　実験方法

OoO 実行はオプションで無効にし, OoO 実行を行わない場合について考えてもらった. まずはシミュレータを使わずに図 6.35 のソースコードを被験者に提示し, 以下の状況を示した.

	スレッド A	スレッド B
1	x = ①	if ($x < y$) {
2	y = ②	$z = -1$;
3		} else {
4		$z = 1$;
5		}
6		join A // スレッド A の終了を待機する
7		x -= y;
8		z *= x;

図 6.35: 実験 2 に使用したソースコード

(a)　①が 7, ②が 3 の時、z が-4 になることがある
(b)　①が 3, ②が 7 の時、z が-4 になることがある

この時, (a)(b) についてどちらがあり得るか, 両方あり得るか, 両方あり得ないかの 4 択を回答してもらい, 理由を記述してもらった. その後, シミュレータを使って改めて同じ問題に回答してもらった. 実験を円滑に進めるため, 本ツールに入力するべきソースコード (図 6.36, 6.37) は筆者が用意し, それを被験者に本ツールの読み込み機能で読み込んでもらった.

実験 1 によって被験者は正しく同期処理をしていないプログラムはスケジューリングによってバグが発生する場合があること, 不可分に見えるが実際には複数の命令から成り立っている操作がありバグの原因になることを学習していることが期待される.

したがって, (b) はスレッド A が動く前, およびスレッド A が x のみを書き込んでいる状態でスレッド B の条件式がチェックされると起こり得ることを被験者は予測できる. ソースコードの見た目と一致した挙動であるため, このことはツールを使用しなくともソースコードから理解しやすい.

しかし, この設問ではスレッド B の if 文の中の条件式において x の読み込みと y の読み込みが不可分でないことに気が付き, かつ x の読み込みと y の読み込みの間にスレッド A が x と y への書き込みを完了する場合を考えなければ (a) があり得ることがわからないようになっている (図 6.38).

したがって, 本実験での本ツール使用前に両方あり得ると答えられなかった被験者が使用後に両方あり得ること, およびその正しい理由が理解できている場合, 本ツールの使用によって使用者の不可分性への理解が深まったこと, および本ツールは並列処理の理解支援に有効であることがわかる.

6.3.2　結果

本ツール使用前では 6 人全員が (b) のみあり得る (不正解) と回答した. 本ツール使用後は 3 人が (b) のみあり得る (不正解), 3 人が両方あり得る (正解) と回答した.

6.3.3　考察

並列処理への理解度の促進に対し, 本ツールは一定の有効性を示した. しかし手動による実行パスの検査では見落としが発生するため, 自動でバグや学習において意味のある実行系列を検出する機能が必要であることが

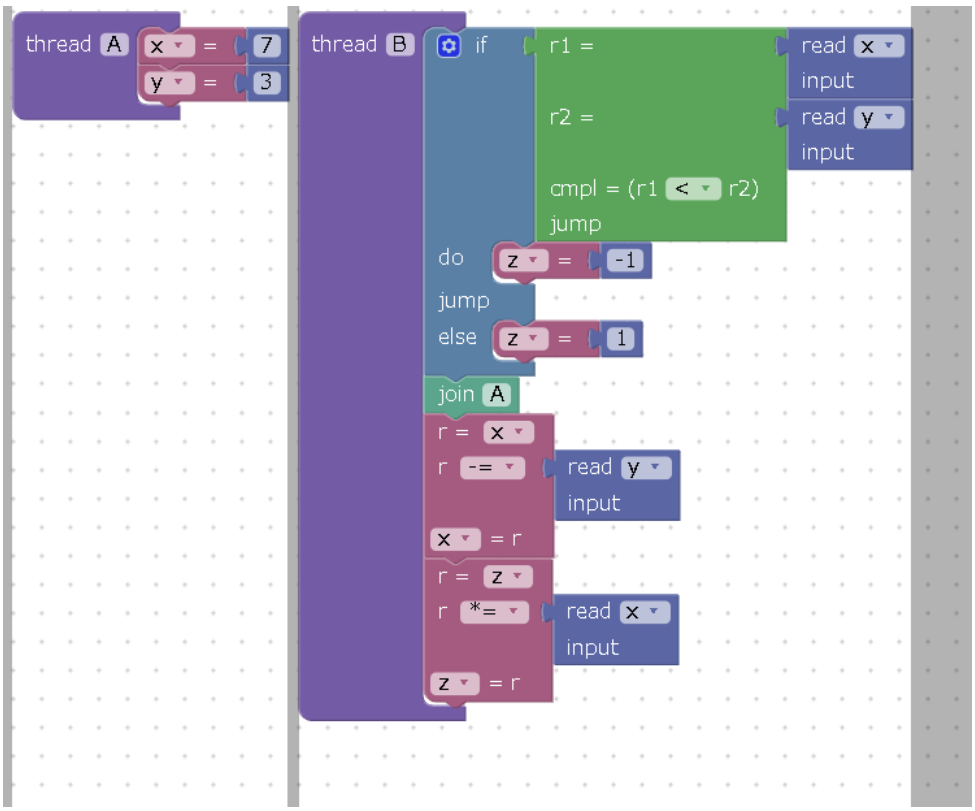

図 6.36: 実験 2 のサンプル: ソースコード

わかった.

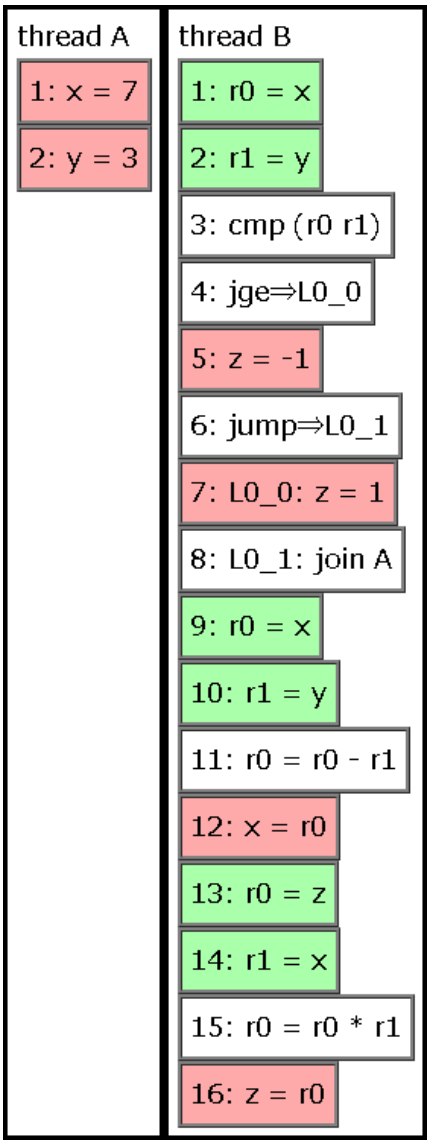

図 6.37: 実験 2 のサンプル: コンパイル結果

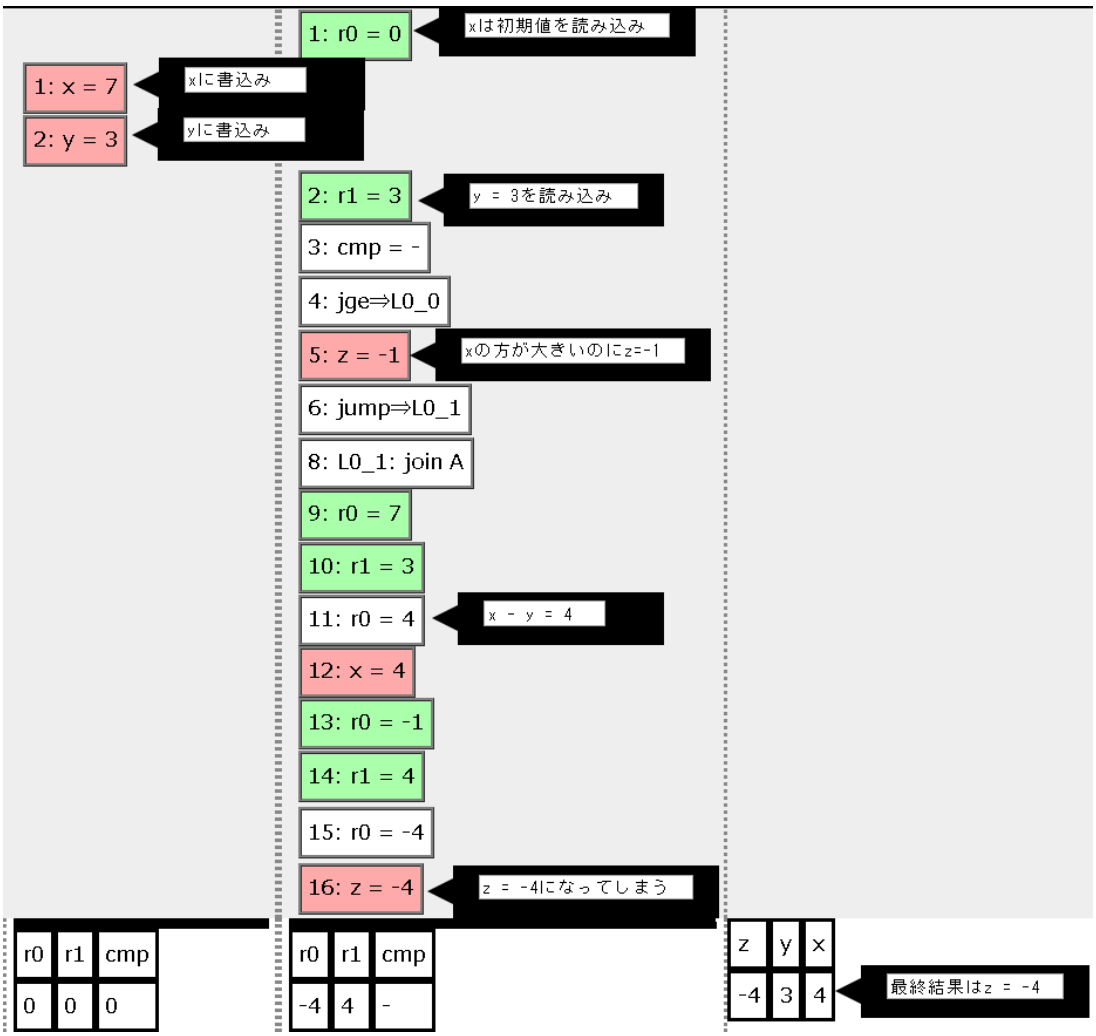

図 6.38: 実験 2 における (a) の状況

第 7 章　結論

7.1　まとめ

ビジュアルプログラミングと OoO 実行を再現したアニメーション, およびプログラムアニメーションの対話的なステップ実行により, 実行パスを手動で選択し, 環境や確率に左右されずマルチスレッドのバグをシミュレートすることができた.

実験 1 の結果より, 本ツールのインタフェースはバグの再現に有効だが, 人によっては操作性やわかりやすさに多少の難があることがわかった. タイムスライダーの調整や非 OoO 実行時の途中経過の表示, 初期値の表示が必要である.

また, 提案システムは並列プログラムへの理解度を高めることに一定の有効性を示した. しかし, 本シミュレータは手動で実行パスの検査を行っているため, バグの見落としがあることが実験 2 により示された. 本ツールの有効性を高めるためには, 自動でバグや学習において意味のある実行系列の検出する機能の実装が必要である.

7.2　今後の課題

現在は手動でアニメーションをさせているが, オプションとして意味のある実行系列の自動選択を実装することで有用性を高めることができる. モデル検査により競合状態を検出する, assert 文を実装しそれをモデル検査でチェックするという方法が考えられる. 本ツールの目的はマルチスレッドの学習であり小規模なプログラムを対象にするので, モデル検査で生成されるパターン数は抑えられると思われる.

7.3　展望

本ツールの応用として, マルチスレッドプログラミングの学習の導入, および教育目的での使用が考えられる. また, モデルチェッキングで出たバグシナリオの可視化や, OoO 実行などをより正確なモデルで実装することで, コンピュータアーキテクチャの学習への応用も期待される.

謝辞

本研究は, 電気通信大学大学大学院 情報理工学研究科 情報・ネットワーク工学専攻 コンピュータサイエンスプログラムの寺田研究室において, 寺田 実准教授のご指導のもと行われました.

寺田 実准教授には, 研究の方針やアイデア, 修士論文の書き方などについて様々な助力, 御指導を頂きました.

村尾 裕一准教授には, 修士論文の書き方などについて御指導を頂きました.

明星大学 情報学部 情報学科の丸山 一貴准教授には, 研究内容について御指導を頂きました.

2018 年 夏のプログラミングシンポジウムの参加者の方々には研究内容について御意見, 御質問を頂きました.

また, 寺田研究室の仲間である修士課程 2 年の村松 啓寛さん, 佐々木 透さん, 修士課程 1 年の飯尾 直樹さん, 三谷 将大さん, 毛利 勇摩さん, 学部 4 年の池田 東さん, 金田 侑大さん, 鄭 佳健さんには研究へのご協力や論文へのアドバイスを頂き, 研究室での生活など様々な面でお世話になりました.

ここに感謝の意を表します.

参考文献

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith. “The Java®Language Specification”. https://docs.oracle.com/javase/specs/jls/se11/html/index.html. August 2018. (参照 2019-1-18).

[2] Brian Goetz, Tim Peierls, Joshura Bloch, Joseph Bowbeer, David Holmes and Doug Lea. 岩谷 宏訳. “Java 並行処理プログラミング”. ソフトバンククリエイティブ, 2010.

[3] Sarita V. Adve and Kourosh Gharachorloo. “Shared Memory Consistency Models: A Tutorial”. Western Research Laboratory Research Report 95/7, September 1995.

[4] Doug Lea. “The JSR-133 Cookbook for Compiler Writers”. http://g.oswego.edu/dl/jmm/cookbook.html. March 2011. (参照 2018-11-15).

[5] Byung-Chul Kim, Sang-Woo Jun, Dae Joon Hwang, and Yong-Kee Jun. “Visualizing Potential Deadlocks in Multithreaded Programs”. Parallel Computing Technologies 2009, pp. 321330, 2009.

[6] Steve Carr, Jean Mayo, and Ching-Kuang Shene. “ThreadMentor: A Pedagogical Tool for Multithreaded Programming”. ACM Journal of Educational Resources, Vol. 3, No. 1, pp.1-30, March 2003.

[7] Tatsuya Abe, Tomoharu Ugawa, Toshiyuki Maeda, Kousuke Matsumoto. “Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models”. Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA’ 16), LNCS, Vol. 9984, pp.118-135. August 2016.

[8] 鵜川 始陽, 松本 稿如, 飯干 寛幸. “並列プログラムの Partial Store Ordering での実行をモデル検査するための Release メモリバリア”. 第 60 回プログラミング・シンポジウム予稿集, pp.119-128, January 2019.

[9] 喜家村 奨, “並列処理プログラミング教育における Scratch の可能性についての考察”. 帝塚山学院大学人間科学部研究年報, 18 号, pp.33-49, December 2016.

[10] Ching-Kuang Shene and Steve Carr. “The design of a multithreaded programming course and its accompanying software tools”. The Journal of Computing in Small Colleges, 1998.

[11] David L. Weaver and Tom Germond, “The SPARC Architecture Manual Version 9”. SPARC International, Inc. 1994.

[12] “SPARC64-III User’ s Guide”. HAL Computer Systems, Inc. 1998.

[13] 中田育男. “コンパイラの構成と最適化”. 朝倉書店, 初版, 1999.

[14] Robert Marco Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. IBM Journal of Research and Development, 11(1):25-33, January 1967.

[15] David A.Patterson and John L.Hennessy. 成田 光影訳.“パターソン&ヘネシー コンピュータの構成と設計”. 上巻. 日経 BP 社マーケティング, 第 5 版, 2015.

[16] Hisa Ando. “高性能コンピュータ技術の基礎”. 株式会社マイナビ. 2011.

[17] Oracle. “第 10 章 プログラミング上の指針 (マルチスレッドのプログラミング)”.

https://docs.oracle.com/cd/E19455-01/806-2732/6jbu8v6q9/index.html. 2010. (参照 2018-11-2).

[18] “ARM®Architecture Reference Manual ARMv8, for ARMv8-A architecture profile”. 2013-2018.