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

藤本 明優[1,a)]　寺田 実[1,b)]

概要：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. 本研究は逐次プログラミング経験者がマルチスレッドプログラミングを学ぶ際に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, シングルスレッドと同じようにプログラミングするとどのように動いてしまうかなどの動作の概要を学ぶためのビジュアルプログラミング言語およびモデル化したハードウェアレベルのプログラムアニメーションの開発を目的とする.

キーワード：並列処理, マルチスレッド, プログラミング学習, プログラムアニメーション, ビジュアルプログラミング言語

1. はじめに

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] 電気通信大学
[a)] f1731135@edu.cc.uec.ac.jp
[b)] terada.minoru@uec.ac..jp

1.2 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 [1][2]. 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の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の最

適化の違いなど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手作業で全パターンを実行できるようにする方法と, モデル検査によりバグのあるパターンを検出して自動実行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前者を選択した.

1.3 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アニメーションでは再現できない場合がある. 例えば単純な変数への加算のような操作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 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 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ルの最適化によ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.

1.4 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. 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 [3].

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 [4]. 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 [5]. 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.

2. 関連研究

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) 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 [6].

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 1.3 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

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

	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 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

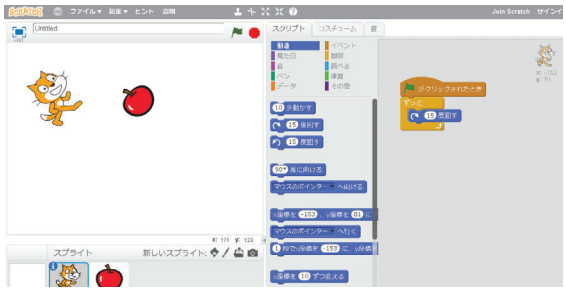

図 2 Scratch によるプログラミングと実行の例

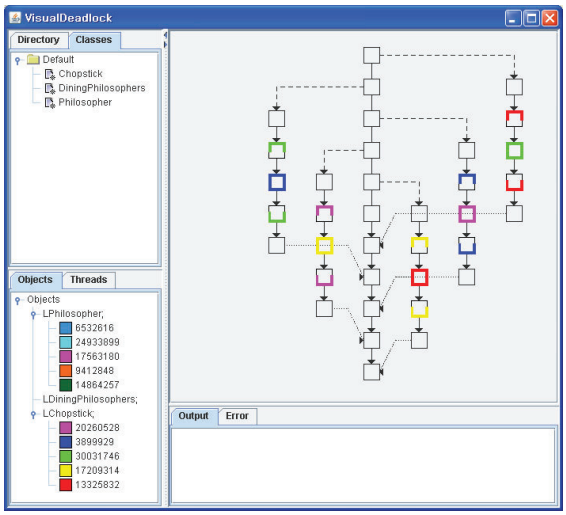

図 3 VisualDeadlock による可視化 ([1] より引用)
画面右側の図の色のついた部分が各ロックとその順序を表している.

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. これ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い.

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 [1] (図 3). 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.

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ツールである [2]. 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二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二点から, 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.

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 [7]. 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ことが示された.

本ツール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, また本ツールを使用することで, 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.

3. マルチスレッドに影響する要素

3.1 アーキテクチャによる load 命令と store 命令の並び替え制限

1.4 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, などが起こり得る. 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 [10][11].

- 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
- 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
- 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
  - 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

3.2 コンパイラによる最適化

正しく同期処理をしていないプログラムに対し, コンパイルの最適化は様々なバグを引き起こす. 例えばある変数が 0 の間は無限にループするスレッドと, その変数を操作するスレッドがある場合を考える (図 4). この時, その言語のメモリモデルや最適化オプションにもよるが, ループ内にその変数への操作や同期処理がない場合, その変数はループ内で変化しないので, 最初の一回だけ値を読み込んで比較し, 一度ループに入った後は条件の比較を行わずに無限ループするように書き換えることがコンパイラには許される. 実際, ループするスレッドだけでプログラムを考えた場合に, この最適化は正しい. しかしこの最適化はプログラマの意図しない無限ループを引き起こす.

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

最適化無し	最適化あり
.L2: movl a(%rip), %eax testl %eax, %eax je .L2	movl a(%rip), %eax testl %eax, %eax je .L4 rep; ret L4: jmp .L4

図 4 コンパイルの最適化に起因する無限ループ (while(a==0){} について)

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 [8] を簡略化したものを実装する.

3.4 Tomasulo のアルゴリズム [8]

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, 確定ユニットの三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 5). 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.

命令とオペランドは機能ユニット内のリザベーション・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.

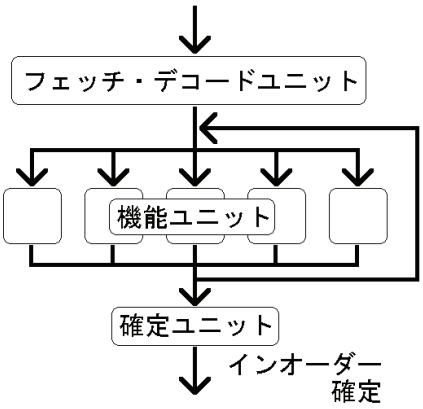

図 5 Tomasulo のアルゴリズム

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.

3.5 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 [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. 本システムには実装していないため詳細な説明は省略する. ストア・バッファによって 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.

4. 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. 本ツールに用意された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理解を促すシステムを目標とする.

例として, 参考書に示されたバグのサンプルを

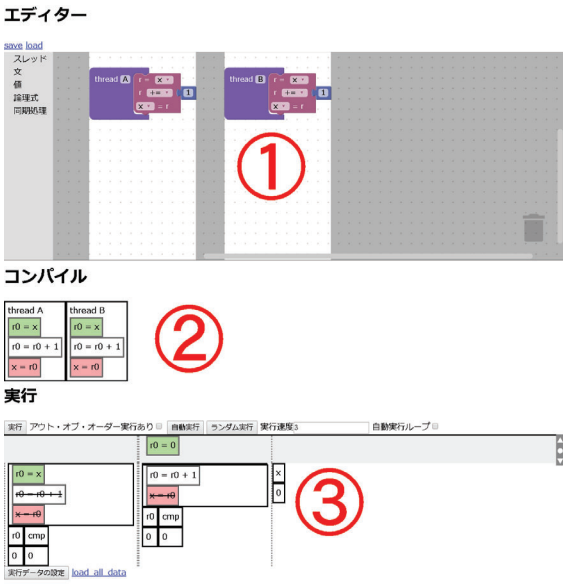

図 6　インタフェース全体

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 6①) 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 6②) 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 6③) 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することで実際にバグが発生していることを確認できる．

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, 一段分の高さとなる．変数の加算操作は変数のレジスタへの読み込み, 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, 定数の代入文の三倍, つまり三段分の高さとなる．また, 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．

ユーザはコンパイルオプションや実行時のアーキテクチャ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．

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．

4.2　実装上簡略化したもの

4.2.1　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, レイアウトやインタフェースの煩雑化を避けるために実装せず, 機能ユニットは単一の物とする．

4.2.2　ストア・バッファ

本研究ではレイアウトやインタフェースの煩雑化を防ぐため, ストア・バッファは実装しない, 代替として, コンパイルの最適化による並び替えで store 命令のオーダリングによるバグを示す．

5.　実装

5.1　概要

JavaScript によってウェブ上にビジュアルプログラミング言語を実装し, 結果を html の要素として出力する (図 6①)．ビジュアルプログラミング部分の実装には GoogleBlockly[*2] を用いた．ビジュアルプログラミング部分からの出力をオプションに応じて最適化, コンパイルした結果を命令列として表示する (図 6②)．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 6③) の命令列は div 要素による長方形 (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対話的に行う設計とする．

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

5.2 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が表現される．ビジュアルプログラミング言語による入力部分 (図 6①) の左端 (図 7①) からブロックを取得し，白い背景の部分 (図 7②) で組み合わせる．

5.2.1 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 1 に示す．

5.2.2 コアへの分配

図 8(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 8(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 5.4.4 項で示す．

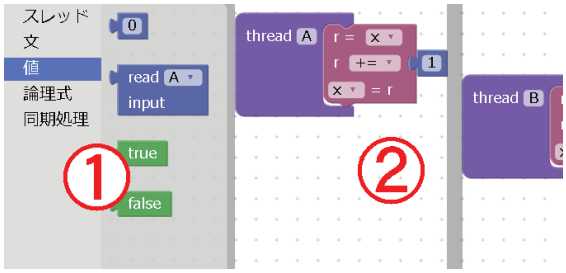

図 7 ビジュアルプログラミング言語による入力部分

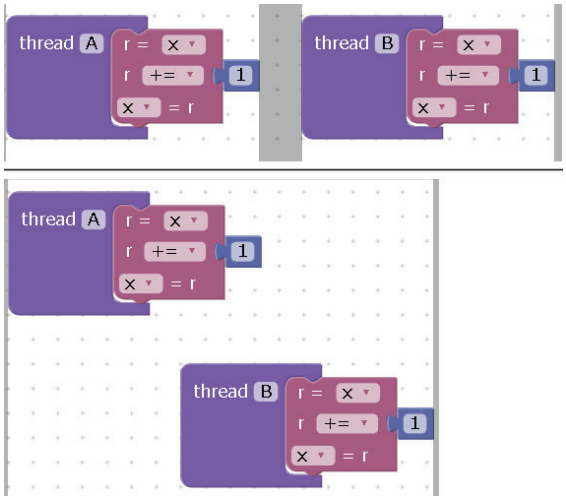

図 8 スレッドのコアへの分配: (a) 図の上半分，異なるコアへのスレッドの配置 (b) 図の下半分，同じコアへのスレッドの配置

5.3 コンパイル

コンパイルの最適化は未実装である．現在はただソースコードをスレッドごとにボックスの列にするのみである．

5.4 プログラムアニメーション

5.4.1 実行オプション

実行時のアーキテクチャの選択は未実装である．

OoO 実行を用いるか否かを設定することができる．

5.4.2 インタフェース

実行履歴はグレーの領域 (図 9①) に表示される．右側のタイムスライダー (図 9②) を操作することで実行時間を制御し，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 9③④)．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．

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 9⑤⑥)．3.4 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．図 9 左側のスレッドは実行中のため，リザベーションステーションとリオーダ・バッファが描画されている．図 9 右側のスレッドはまだ実行を開始していないため，リザベーションステーションとリオーダ・バッファはまだ描画されていない．

5.4.3 実行の手順

OoO 実行を無効にしている場合，命令列のいずれかのボックス (図 9⑦) 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．

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) 元の命令列のいずれかのボックスをクリック

表 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	
	break 文	while 文の中でのみ使える．
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．
	lock	
	unlock	
	join	

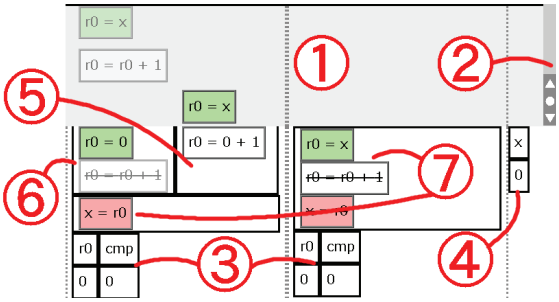

図 9　プログラムアニメーション部分

すると, 一番上の命令がフェッチ・デコードされたことになり, リザベーション・ステーションとリオーダ・バッファ (図 9⑤⑥) にその命令のコピーが生成される．load 命令以外の命令について, リオーダ・バッファに蓄えられている結果やレジスタから得られる値がリザベーション・ステーションにコピーされた命令のオペランドに供給される．

(2) リザベーション・ステーションに蓄えられた命令の内, オペランドの準備ができているものを順不同で実行できる．結果は各リザベーション・ステーション内の別の命令のオペランドやリオーダ・バッファに送られる．load 命令はこの手順の時点でリオーダ・バッファに蓄えられている store 命令の結果かメモリから値を得る．store 命令は同じアドレスに対する load 命令の内, その命令より後にあり直近の同じアドレスへの store 命令よりも前にある load 命令に実行可能になったことを知らせる．

(3) リオーダ・バッファの最古の命令 (リオーダ・バッファの一番上の命令) が結果を受け取っている場合, それをリタイアすることができる．これによって結果がレジスタやメモリに書き込まれる．

こ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．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．

5.4.4　並列性

同じコアに属するスレッドは同時刻に実行する

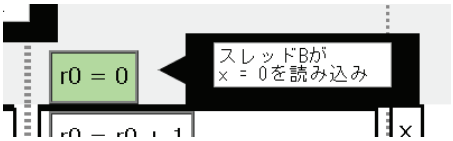

図 10　注釈

ことができない. 別々のコアに属するスレッドは同時刻に実行することができる.

5.5　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. この自動実行において, 一時停止と再生の機能を実装している. この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.

5.5.1　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 10). 注釈は黒い吹き出しとして表示される. 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.

6.　バグのサンプル

本ツールで作成できるバグのサンプルを例示する. なお, どちらの例でも各変数の初期値は 0 とする.

6.1　不可分でない操作による競合状態

$x+=1$ という操作を行う二つのスレッドによる競合状態の例である. OoO 実行は関係ないため, 無効にする. 加算が不可分であれば二回の加算により初期値+2 になるが, 実際には加算命令が不可分でないため, 結果として x は初期値+1 になってしまう場合がある. 結果が初期値+1 になってしまうという例を示す. ソースコードは図 11, コンパイル結果は図 12, 実行履歴は図 13 の通りである. リード・モディファイ・ライト操作が不可分でないことが可視化され, それによって引き起こされるバグがアニメーションによってシミューレートできている.

6.2　アウト・オブ・オーダー実行によるバグ

図 14 に示すプログラムについて, OoO 実行を考慮しなければ, x か y の少なくともどちらか片方は代入が加算より前に行われ, 結果が 3 になるように見える. しかし OoO 実行がされれば代入より先に加算のための読み込みが実行される可能性があり, その場合は両方とも代入前の初期値である 0 に 2 を加算して結果は 2 となる. 結果が両方とも 2 になる例を示す. ソースコードは図 15, コンパイル結果は図 16, 実行履歴は図 17 の通りである. OoO 実行により動的に実行順序が変わること, それによってバグが起こりうることを可視化できた.

7.　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.

8.　今後の予定と課題

8.1　実装の予定

コンパイルの最適化とそのオプションを実装する. ただし, 高度な最適化は困難であるため簡略化したモデルを用いる. また, 実行時のアーキテクチャの選択も行えるようにする. また, 同期処理についてもセマフォやバリア同期の実装を検討している. その他わかりやすいレイアウトの考案と実装, バグのサンプルの作成, アニメーションの再現の巻き戻しの実装なども行う予定である.

8.2　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などの方法がある. 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.

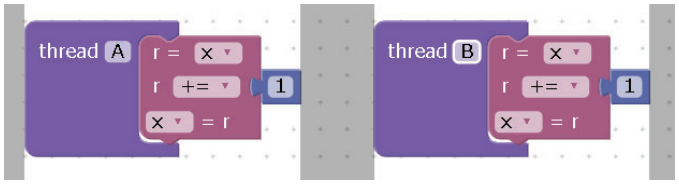

図 11　競合状態のサンプルのソースコード

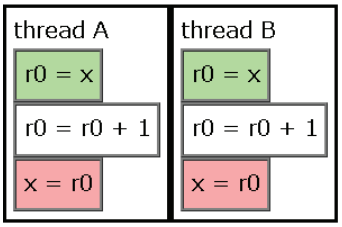

図 12　図 11 のプログラムのコンパイル結果

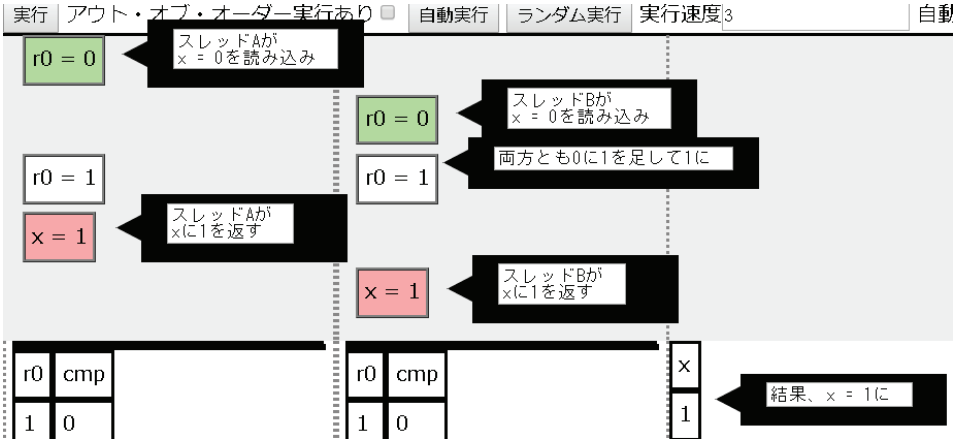

図 13　図 11 のプログラムの実行履歴

スレッド 1	スレッド 2
x = 1	y = 1
y += 2	x += 2

図 14　OoO 実行により想定外の動きをするプログラム

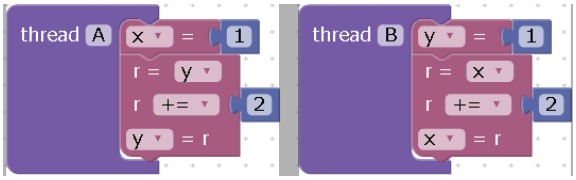

図 15　OoO 実行によるバグのサンプルのソースコード

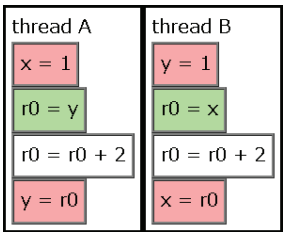

図 16　図 15 のプログラムのコンパイル結果

8.3　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　本稿の作成に当たり, アドバイスを下さった筆者所属研究室の皆様, 発表の場にてご意見を下さったプログラミングシンポジウム参加者の皆様に感謝の意を表します. 本当にありがとうございました.

参考文献

[1] 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.

[2] Steve Carr, Jean Mayo, and Ching-Kuang

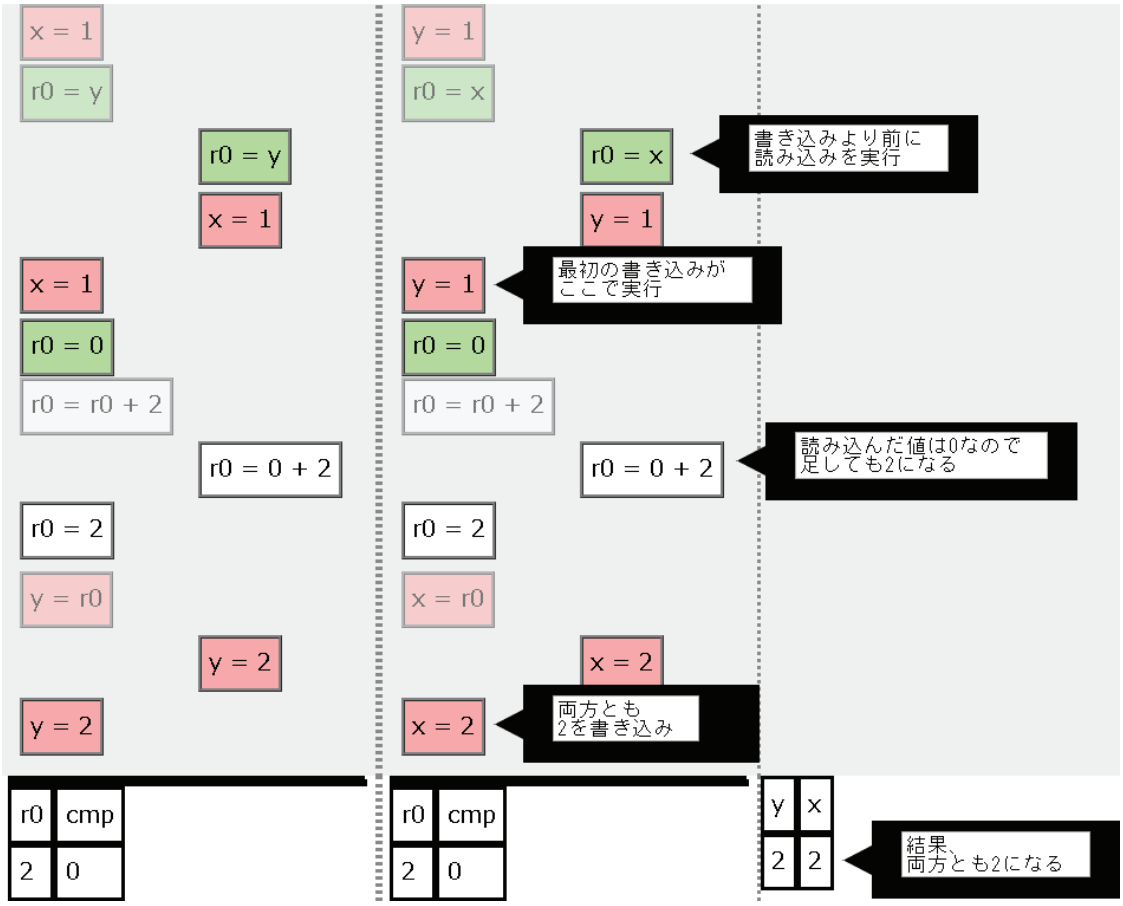

図 17 図 15 のサンプルの実行履歴

Shene. “ThreadMentor: A Pedagogical Tool for Multithreaded Programming”. ACM Journal of Educational Resources, Vol. 3, No. 1, pp.130, March 2003.

[3] 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.

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

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

[6] 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, pp33-49, December 2016.

[7] 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.

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

[9] 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).

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

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

質疑・応答・意見

国士館大学 中村 嘉志氏

質問　マルチスレッドプログラミングは上級者向けだと思うのだが, ビジュアルプログラミング言語をマルチスレッドの教育に使った事例はあるのか.

回答　自分の調べた限りでは発見できなかった. 本ツールの目的は教育というより学習であり, バグのサンプルが再現できないので自分で確かめたい場合などに使用することを想定している.

意見　それがどれくらい有効に働くか, 求められているかを調べた方が良い.

質問　命令レベルよりももう少し抽象的なレベルでの表現を狙わないのか, 今のままだと行数が増えるのではないか.

回答　本ツールの目的上, 大きいプログラムを実行することは考えていない.

意見　目的に対して応用がマッチしない印象を受ける.

意見　OS によってスケジューリングは異なるが, バグの要員として OS の影響を要素として加えるのはどうか.

明星大学 丸山 一貴氏

意見　自分で可能性を選べるのは面白いと思う.

質問　目的に対して粒度が細かすぎる. 粒度をもう少し粗くして, 抽象度を上げるのはどうか.

回答　デッドロックの可視化は既に盛んに行われているため, 競合状態などもう少し細かいレベルのバグの学習についてを狙った.

意見　現実的なプログラムを相手にするのは難しい.

意見　実際に実行した結果を確かめてみるのはどうか.

質問　自動実行でちょっと戻して変えてみるのはどうか.

回答　自動実行したいが, 実際にバグが発生したプログラムを再現するのは規模的に難しい.

意見　教科書レベルの初めてのマルチスレッドのサンプルとやや現実的なバグの両方を扱えると良い.

マッキンテリジェンス 大座畑 重光氏

質問　モデル検査はどのように使うのか.

回答　小さいプログラムへの適用を考えている. 大きいプログラムはレイアウトの関係で難しい.

質問　モデルチェックはやらずにいきなり試すのに使うのか.

回答　本ツールの目的は参考書やネットに落ちているバグのサンプルを自分で試すためであり, モデル検査は今後の課題.

東京大学 萩谷 昌己氏

意見　ビジュアル言語がどれくらい本質的か考えるといい.

質問　ブロックの縦の位置は実行に影響するか.

回答　今の実装では縦の位置は実行に影響しない.

質問　それではテキストベースと同じなのではないか.

回答　確かめたいバグに関係のないエラーを吐かない, 不可分な操作を可視化できるなどの利点がある.

意見　縦の位置に応じて実行の様子が変わるなど, 関係ある方が面白いのではないか.

高知工科大学 松崎 公紀氏

意見　メモリモデルの理解は難しいので, そのための取っ掛かりとして使えるのではないか.

意見　ストロングコンシステンシーでは実行できないが, x86 だと実行可能になるなど具体的に示すことでわかりやすくなるのではないか. 例えばタブで実行オプションのアーキテクチャを分けるなどすると良い.

意見　さらなる拡張を期待する.

ラムダ数学教育研究所 伊知地 宏氏

意見　自動実行で動かし, 結果をデータベースにまとめることでバグシナリオを蓄積, 解析して検索できるようにするとよいのではないか.