Vol.2017-ARC-225 No.36
Vol.2017-SLDM-179 No.36
Vol.2017-EMB-44 No.36
2017/3/10

パス解析を用いた並列実行時メモリ読み書き順序の変化検出

杉山 由芳[1,a)]　枝廣 正人[1]

概要：現在のパワートレーンシステムは環境対策による制御アルゴリズムの複雑化により，マルチコア化が期待されている．しかし，マルチコア化によるシステムへの影響を考慮しなければならない．そこで本研究では，マルチコア化による共有メモリへの読み書き順序の変化に対して，パス解析技術を使用して検出する．マルチコアチップの抽象記述である SHIM の性能情報から C 言語構造の命令の実行サイクル数を推測し，読み書き命令までの最大・最小サイクル数をパス解析によって見積もる．また，シンボリック実行によってそのパスの実行可能性を解析し，実行サイクル数の精度を向上させる．エンジン制御アプリケーションを模擬したベンチマーク用ソフトウェアに対して適用し，有効性を確認した．

Detection of change in memory read/write order during parallel execution using path analysis

Yuho Sugiyama[1,a)]　Masato Edahiro[1]

1. はじめに

パワートレーンシステムでは燃費向上や排ガス規制への対応に負荷の高い制御則を取り入れる必要があり，性能面やコスト面で優れるマルチコアプロセッサの導入が検討されている．しかし，開発者はマルチコア化によるシステムの振る舞いの影響を考慮する必要があり，この問題が組込みシステムのマルチコアプロセッサの導入を阻害している．

マルチコア化によって発生する実行順序の変化は共有メモリにアクセスする命令に影響を与える．そのため，読み書き順序の変化を防止するためにコア間で同期を行う必要がある．しかし，同期を闇雲に取り入れることによって並列動作可能な箇所が少なくなり，マルチコアプロセッサの性能を活用することができなくなる可能性がある．必要な箇所だけ同期を取り入れるために読み書き順序が変化する箇所を特定する必要があるが，実際のハードウェアを動作させて不具合箇所を特定することは困難である．また，静的解析を行う検証ツールの多くは大規模なシステム全体を現実的な時間で検証することが困難である．そのため，現状の開発現場では，読み書き順序が変化することで不具合が発生する箇所を，最終的にはシステム仕様から人手で判別している．

本研究の目的は，プログラムの振る舞いによる共有データへのアクセス順序の変化を可視化し，組込みシステムのマルチコア化を支援することである．読み書き順序は命令が実行されるタイミングの変化によって判別することができる．つまり，読み書き命令の最大・最小実行サイクル数を比較することで，読み書き順序が変化することを高速に判別することができる．ハードウェアやソフトウェアの特性により実行サイクル数の精度は落ちる可能性があるが，実行タイミングを取り入れることでシステムの振る舞いから不具合が発生しない共有データを除外することができる．これによって，検証が必要な箇所を判別する工数を大幅に削減することができる．エンジン制御アプリケーションを模擬したベンチマーク用ソフトウェアに提案手法を適用した結果，パス解析が可能でポインタを介さない共有データに対して読み書き順序が変化する可能性の有無を自動的に判別でき，検証作業の支援に有用であることを示した．

2. 並列化による問題

シングルコアプロセッサで動作していたソフトウェアを

[1] 名古屋大学 大学院情報科学研究科
Graduate School of Information Science, Nagoya University
[a)] yuho_s@ertl.jp

Vol.2017-ARC-225 No.36
Vol.2017-SLDM-179 No.36
Vol.2017-EMB-44 No.36
2017/3/10

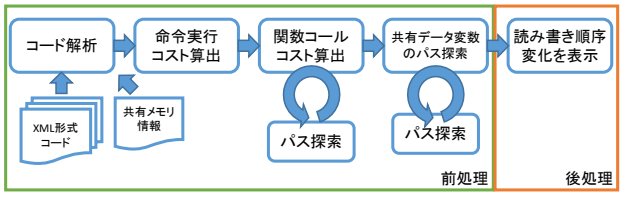

図 1　読み書き順序変化検出の手順
Fig. 1　Process of detecting change in the r/w order

マルチコアプロセッサで実行した場合，タスクを分散させなければマルチコアプロセッサが持つ性能を発揮することはできない．そのため，ソフトウェア開発者はタスクをどのように分割するか考慮する必要があり，さらにタスクを分割することによるソフトウェア品質の影響も考えなければならない．ソフトウェアの並列化によって発生する問題には，デッドロック・ライブロック，並列化オーバヘッド，実行順序の変化がある．

実行順序の変化はコア間で共有されるメモリにアクセスする命令に影響を与え，読み書き命令の実行順序によって，古い値を読み込んでしまう場合や新しい値を上書きしてしまう場合が考えられる．実行順序の変化が発生する要因には，プログラムの実行フローの変化と命令の実行サイクル数の変化が考えられる．実行フローの変化は，条件分岐によって処理量が大きく異なるプログラムを実行する場合，タスクの実行時間に影響を与える．命令の実行サイクル数の変化は，キャッシュやパイプラインなどの命令の実行履歴によって命令の実行サイクル数に影響を与える．

実行順序が変化しないようにするためには，同期を使用する必要がある．しかし，闇雲に同期を取り入れることによって，コア間通信によるオーバヘッドやタスクのイベント待ちによる並列度の低下が問題になる．同期はデッドロックが発生しない限り，実行順序の変化が発生する可能性がある箇所に取り入れる必要があるため，開発者は共有データへ読み書きする命令の全てに対して実行順序が変化するかどうか検討しなければならない．

3. パス解析を用いた読み書き順序の変化検出

本研究では実行フローの変化によって引き起こされる読み書き命令の実行順序の変化に着目し，実行順序が変化する可能性があるところを高速に判別することにより，同期を取り入れる必要がある箇所を開発者に伝えることを目的としている．図 1 は提案手法の流れを示す．前工程ではデータアクセス命令までの最大・最小サイクル数を推測し，後工程ではデータアクセス命令の実行順序の可視化を行う．

3.1 コード解析

まず，構造化されたソースコードについて紹介する．コードの構造化には C/C++コンパイラ clang[4] の AST（Abstract Syntax Tree）をベースに，ループの展開，CFG（Control Flow Graph）の出力等の機能を追加したツールを使用している．制御システムのような組込みプログラムでは，define やマクロ，ifdef などによってモジュールの接続，機能の有効・無効を設定しているため，プリプロセッサによって展開する必要がある．また，変数情報や命令情報を抽出することにより，モデル検査等の静的解析ツールへの適用が容易になる．本研究では，C 言語コードを構造化した xml ファイルを入力として与え，CFG と変数情報と命令情報を抽出した．

3.2 コスト算出

次に，CFG の各ノードの実行サイクル数を見積もる．構造化されたソースコードから各ノードに含まれる命令情報を抽出し，命令ごとのコストの合計を推測値とする．命令ごとのコストは SHIM[3] の性能情報から取得し，命令内容と SHIM 性能情報に記載されている LLVM IR ベースの命令と対応付ける．本研究ではルネサス エレクトロニクス社 RH850 ベースの評価用メニーコアチップの性能を計測した SHIM 性能情報 [5] を使用した．

C 言語レベルの構造では変数がメモリまたはレジスタのどちらに配置するか決定しないため，グローバル変数またはローカル変数の違いによってメモリまたはレジスタのどちらに配置するか推定する．グローバル変数は命令ごとのコストに加えてメモリロード・メモリストアのコストを加算し，ローカル変数はコンパイラの最適化によってレジスタに配置されると仮定し，命令ごとのコストのみを算出する．

命令のコストには SHIM 性能情報の typical 値を使用する．読み書き命令までの最大・最小サイクル数を推測する場合には best，worst の値を使用するべきである．しかし，SHIM 性能情報の best，worst にはハードウェアの状態が最良，最悪の場合を想定しており，ハードウェアの状態が常に最良，最悪の場合で実行サイクル数を見積もるのは楽観的，悲観的な結果になる．そのため，本研究では実行パスによる実行サイクル数の変化のみに着目するために typical 値を使用した．

3.3 関数コール算出

CFG には関数呼び出しによる制御フローは表現されていないため，関数呼び出しによる影響を考慮する必要がある．呼び出される関数の CFG を使用して関数呼び出し命令の箇所に展開する手法も考えられるが，CFG の規模が爆発的に増加する可能性がある．特に，複数の関数から呼び出される関数の場合は展開した数だけ計算コストが増加する．そのため，本研究では関数ごとに最長・最短パスを探索し，呼び出し先の関数の実行サイクル数のみを考慮する．また，呼び出される関数がどのタイミングで呼び出さ

Vol.2017-ARC-225 No.36
Vol.2017-SLDM-179 No.36
Vol.2017-EMB-44 No.36
2017/3/10

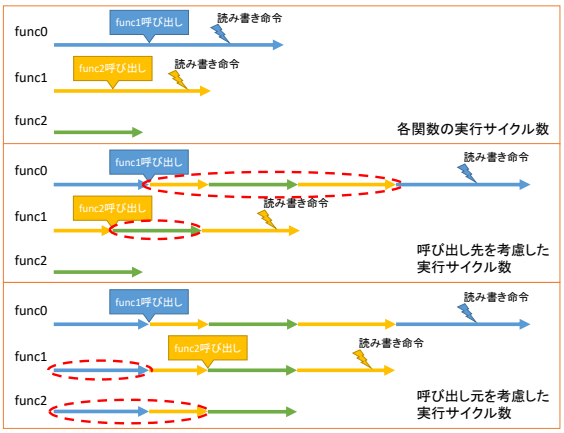

図 2　関数呼び出しによる実行サイクル数の影響
Fig. 2　Effect of number of execution cycles by function call

れるのかについても考慮する必要もある．

図 2 は関数呼び出しによる影響を実行サイクル数のみで表している．関数呼び出し命令が実行された後，制御フローは呼び出し先の関数へ移行し，呼び出し先でリターン命令が実行された後に元の関数へ制御フローが戻る．そのため，関数ごとに分けて最長・最短パスを探索するには，呼び出し元の関数は呼び出し先の関数全体の実行サイクル数を関数呼び出し命令に含める必要があり，呼び出し先の関数は呼び出されるまでに経過した実行サイクル数をオフセットとして含める必要がある．

関数全体の実行サイクル数を見積もるには，関数内の関数呼び出し命令が見積もられた後に行うべきである．そのため，関数呼び出しがないコールグラフ上で深い階層の関数から順に関数全体の実行サイクル数を見積もりを行う．また，呼び出されるまでのオフセットを見積もるには，関数呼び出し命令までの間の関数呼び出し命令が見積もられ，呼び出し元の関数のオフセットが見積もられた後に行うべきである．そのため，関数全体の実行サイクルをすべての関数対して見積もり，呼び出し元の関数呼び出し命令に反映した後に，関数が呼び出されないコールグラフ上で浅い階層の関数から関数呼び出し命令までの実行サイクル数を見積もり，呼び出し先の関数の開始ノードにコストとして加算する．

本研究では他の関数から呼び出されない関数の開始ノードを 0 サイクル目として基準を設定する．他の関数から呼び出される関数は関数呼び出しまでの実行サイクル数を開始ノードにオフセットとして代入されるため，すべての関数はコールグラフ上でトップの関数が基準となる．多くの組込みシステムでは，タイマーからの割込みなどを使用して周期的にタスクを実行する．そのとき，割込みによって最初に呼び出される関数から，機能ごとの処理を呼び出す構造となる．したがって，その関数をトップにしたコールグラフが形成され，実行サイクル数の基準はタイマーからの割込みが発生した瞬間となる．タイマーからの割込みがすべてのコアで同時に発生すると仮定した場合，コア同士の実行サイクル数の基準が一致するため，共有メモリに対する読み書き命令までの実行サイクル数を比較することができる．

3.4　パス探索

パス探索では実行可能性解析，実行不可能なパスの除去を行う．これは，読み書き命令までの最大・最小サイクル数が悲観的な結果になることを防止するためである．最大・最小サイクル数が悲観的な結果の場合，本来発生しない読み書き順序の変化を誤検出する可能性がある．ただし，実行不可能なパスの場合でも，読み書き命令までの最大サイクル数の上限，最小サイクル数の下限は保証される．

パス探索の流れは，関数ごとに読み書き命令までの最長・最短パス候補を探索する．次に，最長・最短パスの候補が実行可能かどうかシンボリック実行を使用して解析する．このとき，実行可能である場合は最長・最短パスとして採用され，実行不可能である場合は実行不可能となる原因を除去し，再び最長・最短パスの候補を探索する．

最長・最短パス探索

最長・最短パス探索では，CFG に付加された各ノードごとに実行サイクル数を重みとして，最大・最小パスをダイクストラ法 [2] を使用して探索する．本来，ループ構造がある CFG に対して，ダイクストラ法を使用して正しく最大パスを探索することはできない．本研究では構造化された C 言語コードの段階でループ展開がされているため，ダイクストラ法を使用して最大パスを探索することができる．

図 3 は最長・最短パスを探索した例である．楕円は CFG のノードを表し，中にノード番号と命令内容が記述されている．青線は最長パスを表し，赤線は最短パス，紫線は最長パスと最短パスが同じであることを表している．また，直前のノードが条件分岐命令の場合には，条件が成立するときは実線，成立しないときは破線の方向に分岐する．各パスには直前のノードの実行サイクル数が重みとして記述されている．なお，重みには最小（min）と最大（max）があるが，本研究ではどちらの場合でも typical の値を使用する．

実行可能性解析

最長・最短パス候補の実行可能性解析にはシンボリック実行を使用する．シンボリック実行は CBMC[1] のシンボリック実行を参考にしている．ただし，通常のシンボリック実行は計算コストが高いため，本研究では限定的なシンボリック実行を行う．また，最長・最短パスの探索は関数ごとに行われているため，シンボリック実行についても関数ごとに行う．

最長・最短パスにおいて実行不可能なパスとなる場合

Vol.2017-ARC-225 No.36
Vol.2017-SLDM-179 No.36
Vol.2017-EMB-44 No.36
2017/3/10

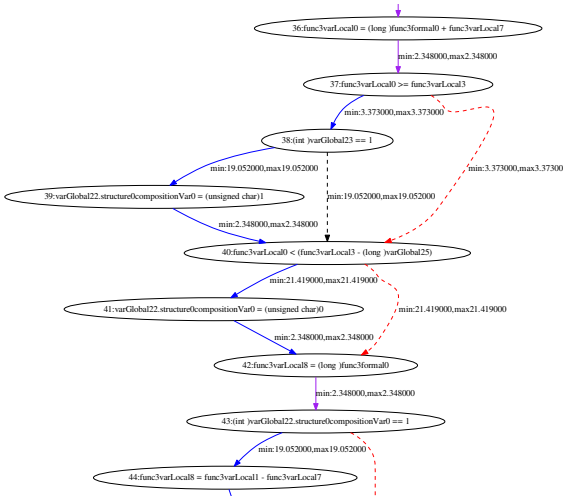

図 3　ダイクストラ法を使用用した最長・最短パスの探索

Fig. 3　Search for longest and shortest paths using Dijkstra's method

には，代入命令と条件分岐命令の矛盾がある．図 4 は図 3 の最大パスをシンボリック実行した結果の一部である．図 4 では最大パスが通ったノードの命令を並べている．ここで，変数 varGlobal22 の structure0compositionVar0 要素に着目すると，図 3 の 41 番ノードで 0 が代入されているが，43 番ノードの条件分岐命令で変数 varGlobal22 の structure0compositionVar0 要素が 1 であるという条件が真の方向に分岐していることがわかる．このような最大パスは実際にプログラムを実行した場合には起こりえないため，実行不可能なパスである．

本研究では組込みシステムの特性に合わせたシンボリック実行を考案した．組込みシステムでは状態遷移モデルで設計されていることが多い．状態遷移モデルは内部に状態を記憶し，入力と現在の状態から次の状態と出力を決定する．そのため，条件分岐の条件文には入力値または状態が使用されると考えられる．入力値はセンサー等の外部から与えられる値である場合が多く，値を推測するためにはシステムの仕様を確認する必要がある．一方，状態はシステム内部で使用される値であるため，ソフトウェア的には整数値を割り当てる場合が多い．そのため，状態を使用した条件分岐命令には，状態と状態を表す整数値の一致，不一致を条件とすることが考えられる．本研究では，このような状態を表す変数に対してシンボリック実行を行い，状態を更新する代入命令と状態を確認する条件分岐命令の矛盾を検出する．図 4 では，変数 varGlobal22 の structure0compositionVar0 要素が何らかの状態を表し，状態を更新する 41 番ノードと状態を確認する 43 番ノードで矛盾が発生するため，実行不可能なパスと判定できる．

実行不可能なパスの除去

シンボリック実行によって実行不可能と判定されたパス

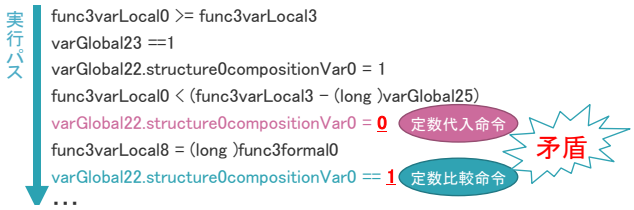

図 4　最長・最短パスのシンボリック実行

Fig. 4　Symbolic execution of the longest and shortest path

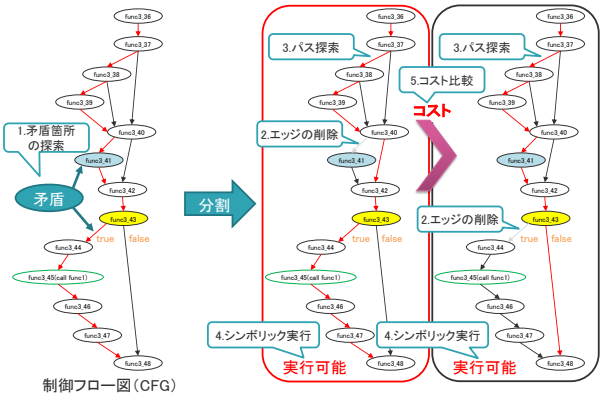

図 5　実行不可能なパスの除去

Fig. 5　Elimination of infeasible path

について，矛盾が発生する代入命令と条件分岐命令のいずれかに到達するエッジを除去する．図 5 は実行不可能なパスの除去の概要を表している．図 5 では図 4 で検出された実行不可能なパスの除去を行っている．まず，矛盾が発生する代入命令と条件分岐命令の分岐方向を抽出する．図 5 では，実行可能性解析で述べた矛盾が func3_41 と func3_43 の true 方向で表されている．次に，矛盾が発生しないようにするため，代入命令に到達するパスか条件分岐命令の分岐方向のいずれかのエッジを除去する．この場合，一方のパスをカットした 2 種類のグラフが生成される．図 5 では，func3_40 から func3_41 へのエッジを除去したグラフと func3_43 の true 方向のエッジを除去したグラフの 2 種類が生成される．そして，生成されたグラフについて，再度最長・最短パスの探索を行い，シンボリック実行によってそのパスが実行可能かどうか判定する．図 5 では，2 つのグラフの両方で最長パスを探索した結果，最長パスは実行可能であると判定されている．このとき，最短パスであればパスの実行サイクル数が小さい方を，最大パスであればパスの実行サイクル数が大きい方を選択する．図 5 では，左のパスの方が実行サイクル数が大きいため，左のパスが最大パスとして採用される．

矛盾する代入命令と条件分岐の組が複数ある場合，実行不可能なパスの除去によって複数の最長・最短パスの候補が生成される．上記の手法を用いると実行不可能なパスを除去した 2 つのグラフが生成され，それぞれのグラフから最長・最短パスの候補が探索される．その最長・最短パス

の候補が実行不可能である場合は，そのパスを除いた 2 つのグラフがさらに生成される．エッジの除去によって CFG の終端ノードに到達できなくなる場合を除いて，生成されたグラフから実行可能な最長・最短パスを求め，その中で最良・最悪なものを選択する必要がある．つまり，矛盾する代入命令と条件分岐の組が n 組ある場合，最大で 2^n 組の最長・最短パスの候補が生成される．そのため，本研究ではグラフの生成回数によって実行不可能なパスの除去を打ち切る処理を追加することで，パス探索やシンボリック実行を繰り返すことによる計算量の爆発的な増加を防止している．

3.5 読み書き順序変化の可視化

前工程によって，複数コアでアクセスされる可能性がある共有データごとに読み書き命令がある関数名，その関数が実行されるコア番号，読み書き命令までの最大・最小サイクル数，最大・最小サイクルとなるパスがまとめられた CSV ファイルが出力される．後工程ではその CSV ファイルのデータを使用して，読み書き命令の実行タイミングを表すグラフや読み書き順序が変化する可能性がある共有データの一覧を表示する．本研究では，Excel のマクロ機能を使用して共有データごとの CSV ファイルの取り込みと可視化を行う．

Excel ファイルは，すべての共有データの読み書き順序の結果を表すトップシートと共有データごとのシートから構成される．トップシートには，共有データの変数名とその共有データの読み書き順序が変化するかについて表す．共有データごとにシートには，読み書き命令の実行順序を表すグラフと最大・最小サイクルとなるパスを表示する．グラフは読み書き命令が実行される可能性のある実行サイクル数の範囲を棒グラフで表す．また，読み書き命令が実行されるコア番号，関数名，命令までの最大・最小実行サイクル数，最大・最小サイクル数となるパスを読み書き命令ごとに表す．パスの表示については，条件分岐命令とその方向，関数呼び出し命令といった制御フローの分岐点のみを表す．

共有データの読み書き順序の変化は読み込み命令の最大・最小サイクル数と書き込み命令の最大・最小サイクル数を比較して行う．それぞれの命令は最小サイクル数から最大サイクル数までの範囲のどこかで実行されるため，異なるコアで実行される読み書き命令の実行サイクル数の範囲が重なる場合に実行順序が変化する可能性があると判定する．また，読み込み命令と書き込み命令の実行サイクル範囲が重ならない場合は実行順序が変化しないと判定する．さらに，読み込み命令と書き込み命令が同じコアで実行される場合には，実行サイクル範囲が重なっていたとしてもプログラムの実行フロー上で順番が定められるため，実行順序が変化しないと判定する．なお，読み込み命令または書き込み命令の片方しかない場合は実行順序の判定を行わない．

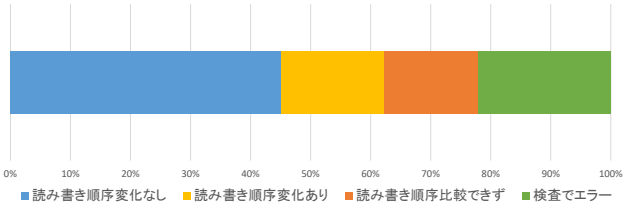

図 6 共有データの読み書き順序の変化を検出した結果
Fig. 6 Detecting change in the r/w order of shared data

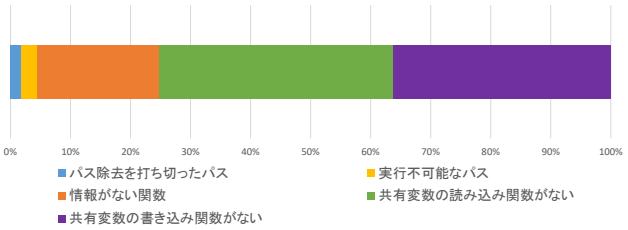

図 7 読み書き順序の変化を検出できなかったエラー要因
Fig. 7 Error factor that failed to detect changes in r/w order

4. 評価

構造化 C 言語コードで記述されたエンジン制御アプリケーションを模擬したベンチマーク用ソフトウェアを対象に，共有データに対する読み書き順序の変化を検査した．ベンチマークソフトウェアは 4 コアに分割され，3.1 で説明したツールを使用して構造化した．CFG はノード数約 15 万，エッジ数約 17 万であった．

提案手法を実装したツールを用いてベンチマークソフトウェアを評価した結果，約 2 時間ですべての変数の読み書き順序の変化を検査することができた．また，2 時間のうち，前処理には約 30 分，後処理には約 1 時間 30 分かかった．パス探索については，グラフの生成回数を 2 回までとした場合，1 秒から 30 秒程度かかった．

提案手法を用いて共有データの読み書き順序の変化を検出した結果を図 6 に示す．読み書き順序を比較できなかった共有データは，読み込み命令または書き込み命令の一方の情報がなかったことが原因であり，ベンチマークソフトウェアに含まれない OS 等と共有されるデータであったと考えられる．また，読み書き順序を検査することができなかった共有データについては，エラーによって検査することができなかった．

読み書き順序の変化検出で発生したエラーの原因の割合を図 7 に示す．「パス除去を打ち切ったパス」は 3.4 で述べたパス探索の打ち切りが行われたことを示す．さらに，打ち切り処理が行われるまでに実行可能なパスが発見できなかったパスを「実行不可能なパス」は示す．これは，ループ内部に実行不可能なパスを含むコードでは，実行パス上

Vol.2017-ARC-225 No.36
Vol.2017-SLDM-179 No.36
Vol.2017-EMB-44 No.36
2017/3/10

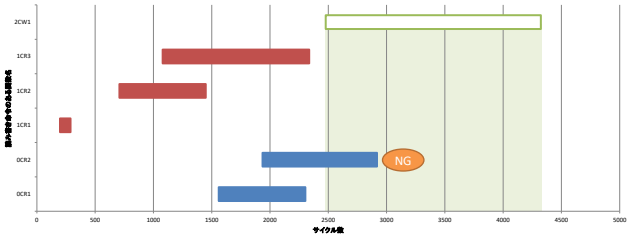

図 8　共有データの読み書き順序を可視化したグラフ
Fig. 8　Graph visualizing sharing data read / write order

に複数の矛盾する命令が含まれてしまうことが原因であると考えられる．

「情報がない関数」は 3.3 で述べた関数コール命令が呼び出す関数の情報がなかったことを示す．また，「共有変数の読み込み・書き込み関数がない」は共有メモリ情報にある共有データに読み書きを行う関数に関する情報がなかったことを示す．これらについては，ベンチマークソフトウェアに含まれないシステム関数が該当すると考えられる．また，現状ポインタ解析が不十分であり，共有データの実体が正しく認識されていない可能性がある．

最後に，読み書き順序を可視化したグラフを図 8 に示す．緑色で白抜きした棒グラフはコア 2 から書き込み命令が実行されることを表す．また，赤色で塗りつぶした棒グラフはコア 1 から読み込み命令が実行されることを表し，青色で塗りつぶした棒グラフはコア 0 から読み込み命令が実行されることを表す．図 8 では，NG と書かれた読み込み命令の実行順序が入れ替わる可能性があることを示している．評価結果から，複数のコアで共有される変数の多くは 1 つの関数から書き込まれ，複数の関数から読み込みされていることがわかった．その読み込み関数には最大サイクル数と最小サイクル数の差が大きい命令がいくつか存在した．このような場合は最長・最短パス探索の結果が悲観的であることが考えられ，実際の最大サイクル数と最小サイクル数の差よりも過大に表現されている可能性がある．

5. おわりに

5.1 まとめ

本研究ではパス解析技術を用いて並列動作による共有メモリの読み書き命令の実行順序の変化を検出する技術を提案した．マルチコアチップの抽象記述である SHIM の性能情報を用いてプログラムの実行サイクル数を推測し，パス探索によって共有メモリの読み書き命令までの最大・最小サイクル数を推測した．また，シンボリック実行によってパスの実行可能性を検証し，実行不可能なパスを除去することによって最大・最小サイクル数の精度を向上させた．

本研究の評価として，エンジン制御アプリケーションを模擬したベンチマーク用ソフトウェア対して，複数のコアから読み書きされる共有データの実行順序を検査した．また，読み書き順序をグラフによって可視化することにより，共有データの読み書き順序を容易に把握できることを示した．

5.2 今後の課題

現状，シンボリック実行が対象とするパスは関数内部に限定されているため，別のシステム関数等で書き込まれるフラグを条件とする条件分岐命令はシンボリック実行の対象外となる．シンボリック実行の対象範囲を広げるためにあらかじめ関数をインライン展開するといった処理が必要になる．

状態爆発を防止する打ち切り処理では，ループ内部に実行不可能なパスを含む場合，現状の実行不可能なパス除去では処理量が爆発的に増加してしまうため，多くの場合でパス探索が打ち切られていた．そのため，複数箇所に矛盾する命令が含まれている場合を考慮して，実行不可能なパスの除去を行う必要がある．

現状，3.1 で述べたツールはポインタ解析が不十分であり，ターゲット依存部の多いコードに対しては構造化を行うことができない．一般的にポインタを静的解析によって完全に解析することは困難であるため，抽象解釈を用いてポインタの指すメモリを推定する必要がある．また，ハードウェアに近いソフトウェアは一部がアセンブリ言語で記述されている場合があるため，解析する必要がある．

本研究では読み書き命令までの最大・最小サイクル数を静的に推測して読み書き順序の変化を検出したが，検出した読み書き順序の変化が実際に発生することを実機上では確認していない．そのため，静的解析によって得られた最大・最小サイクルのパスを通るテストパターンを生成して，実機上で読み書き順序が変化するかどうかを検証する必要がある．

謝辞　日頃議論いただくトヨタ自動車株式会社の皆様に深く御礼申し上げます．

参考文献

[1] Clarke, E., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (Jensen, K. and Podelski, A., eds.), Lecture Notes in Computer Science, Vol. 2988, Springer, pp. 168–176 (2004).
[2] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs, *Numer. Math.*, Vol. 1, No. 1, pp. 269–271 (online), DOI: 10.1007/BF01386390 (1959).
[3] Gondo, M., Arakawa, F. and Edahiro, M.: Establishing a standard interface between multi-manycore and software tools-SHIM, *COOL Chips XVII, 2014 IEEE*, IEEE, pp. 1–3 (2014).
[4] clang: a C language family frontend for LLVM. http://clang.llvm.org/.
[5] 溝口裕哉：ソフトウェア向けハードウェア性能記述を用いたプロセッサ性能見積手法に関する研究，修士論文，名古屋大学情報科学研究科　情報システム学専攻 (2016).