Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

モデル解析によるマルチレートSimulinkモデルの性能向上

池田 良裕[1,a)]　鈴木 均[2]　枝廣 正人[1]

概要：車載制御向けの設計において，モデルベース開発を用いることが多くなってきている．また，より高性能な制御のため，制御装置のマルチコア化も同時に進んでいる．ハードウェアのマルチコア化に伴い，ソフトウェアの自動並列化の研究も進められている．しかし，複数制御周期を持つマルチレートモデルに対し，異なる制御周期のコードを混在させつつ，効率のよい並列化を行うことは簡単ではない．本研究では，複数周期を持つ (マルチレート)Simulink モデルを対象として並列度向上手法について提案する．マルチレートモデルのうち長周期タスクに対してモデル構造を考慮した負荷分割を行い，短周期に合わせて一部ずつ実行していくことで性能を向上させる．提案手法により従来の実行時間に比べ，性能が向上していることを確認した．

Performance improvement of multirate Simulink models by model analysis

Yoshihiro Ikeda[1,a)]　Hitoshi Suzuki[2]　Masato Edahiro[1]

1. はじめに

車載分野を始めとする組込みシステムでは大規模・複雑化が進んでおり，モデルベース開発を用いることが多くなっている．制御システム設計でよく用いられるモデルベース開発支援ツールに MATLAB/Simulink[1] があり，シミュレーション・自動コード生成機能等が搭載されている．Simulink モデルの段階でシミュレーションを行うことで早い段階でエラー・誤動作等を発見でき，実機で動作させ確認を行いソフトウェアを修正するのに比べ，手戻りを減らす事ができる．また，自動コード生成機能があることにより，設計書に近い形の Simulink モデルで維持・管理が可能となるため，コードの状態で管理する必要が無くなる．これらの機能により MATLAB/Simulink を用いることでシステム設計をスムーズに進めることができる．

一方で, プロセッサの周波数の上昇が止まりつつあり，消費電力の面からも組込みシステム分野においてマルチコア CPU の導入が進んでいる．マルチコア CPU を利用するにあたりソフトウェアを並列化する必要があるが，これを人手で行うのは容易ではない．そのため，コード解析などによりソフトウェアの並列性を探し出し，自動でソフトウェア並列化を行う研究が盛んに行われている．我々もモデルベース開発を対象とし，Simulink モデル解析によるソフトウェア並列化技術の研究を進めている．

本研究では，MATLAB/Simulink によるモデルベース開発に対して，Simulink モデル解析によるソフトウェア並列度向上手法について提案する．複数周期を持つ (マルチレート)Simulink モデルを対象とし，長周期タスクに対して負荷分散を行い，その結果を Simulink モデルへ反映させることで短周期と同じ周期で動作させることを可能にし，性能を向上させる．

本論文の構成は以下の通りである．まず 2 章で我々が進めているモデルベース自動並列化技術について説明し，3 章でマルチレート Simulink モデルへのモデルベース自動並列化技術の適用における課題を述べる．4 章では性能向上のための負荷分散方法を述べ，5 章で Simulink モデルを一部変更することによる性能向上方法について説明する．6 章で負荷分散方法の妥当性と変更後の Simulink モデルか

[1] 名古屋大学 大学院情報科学研究科
愛知県名古屋市千種区不老町
[2] ルネサス　エレクトロニクス株式会社
東京都小平市上水本町 5-20-1
[a)] yoshi12@ertl.jp

Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

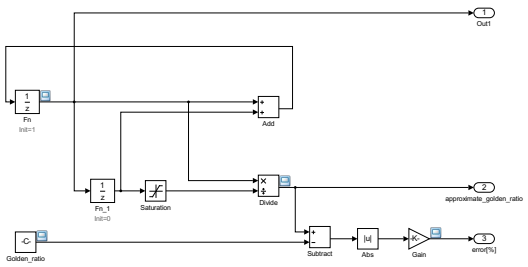

図 1　Simulink モデル

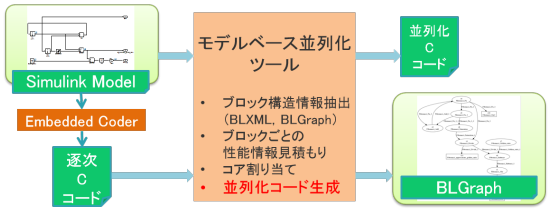

図 2　モデルベース自動並列化フロー

ら生成される並列化コードの評価を行う．

2. Simulink モデルからの並列化コード自動生成

2.1 MATLAB/Simulink[1]

MATLAB/Simulink はモデルベース開発支援ツールの一つであり，車載アプリケーションの開発では広く使われている．MATLAB/Simulink を用いたモデルベース開発では Simulink モデルと呼ばれるブロック線図によりアプリケーションを記述する．Simulink モデルの例を図 1 に示す．Simulink モデルを作成することで，シミュレーションや自動コード生成を行う事ができる．シミュレーション機能を使うことで開発工程の早い段階で修正することができ，手戻りを減らすことが可能である．

Simulink モデルはブロック毎に処理を行い，処理結果を信号線で表すデータフローダイアグラムである．システムを表現した Simulink モデルに対し全体の周期を設定するが，それとは別に各ブロックはそれぞれモデル全体の周期の整数倍の実行周期を設定することができる．また異なる周期を繋ぐ信号線には Rate Transition や Unit Delay といったブロックを用いてデータの整合性をとる必要がある．このような設定をすることで，本研究の対象である複数周期を持つシステムを記述した Simulink モデルを作成することが可能である．

2.2 並列化コード生成

現在，我々が進めているモデルベース自動並列化技術 [2] について説明する．モデルベース自動並列化フローを図 2 に示す．

まず，自動コード生成可能な Simulink モデルからブロック情報及びブロック間の接続情報を抽出する．次に，自動生成した逐次コードをブロック毎に対応させ，BLXML(Block Level XML) を生成する．BLXML は，ブロックの入力・出力，対応するコード情報を一つのグループとして，ブロック毎に記述される．最後に，ブロック毎の処理量や通信オーバヘッドを考慮してコア割り当てを行い，並列化コードを生成する．ブロック毎の処理量の見積もりについては 4 章で述べる．

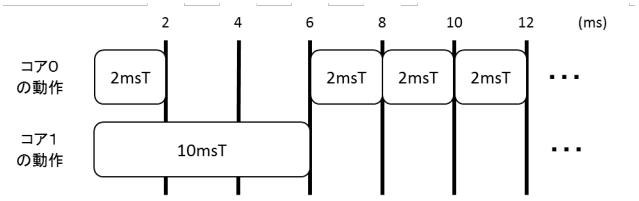

図 3　現状の並列化による実行パターン

並列化コードはターゲットの OS を設定し生成することが可能である．本研究では組込みシステム向けメニーコアプロセッサ対応リアルタイム OS である MCOS（Many-Core real-time OS）[3] をターゲット OS として並列化コードを生成した．コア毎にスレッドを生成することで並列実行が可能になっている．通信が必要な箇所ではバッファを使用し，データをバッファに送信・受信することで行う．その際にフラグを用いることでバッファに対して書き込み・読み込みが行われるデータの順序を保っている．Inport ブロックなどの入力側にポートを持たないブロックはコア毎のスレッドとは別に生成されるタスク 0 により同期される．

3. マルチレート Simulink モデルからの並列化コード生成における課題

前章でモデルベース自動並列化技術について簡単に述べたが，対象の Simulink モデルの中には現状の並列化を行っても十分に性能を発揮できないものがある．その一つが複数周期 (マルチレート) を持つ Simulink モデルである．

マルチレート Simulink モデルから短周期タスクをコア 0，長周期タスクをコア 1 に割り当てた場合を想定する．短周期タスクが長周期タスクから結果を受け取る際に，短周期タスクが長周期タスクの終了を待つことになる．そのとき，長周期タスクの実行時間が短周期を上回っていた場合，短周期タスクはスケジュール通り実行できない問題が発生する．上述の実行パターンを図 3 に示す．

図 3 では短周期が 2ms，短周期タスクの実行時間が 2ms，長周期が 10ms，長周期タスクの実行時間が 6ms としている．時間 0 にコア 0，コア 1 が両方起動する．短周期は時刻 2 に処理を終了し，次の短周期タスクを実行可能となるが，長周期タスクは時刻 6 まで処理が完了せず，短周期タスクは待機してしまうため処理を開始することができない．これでは，2 つのコアで並列動作させても，長周期タスクの空き時間を持て余し，更に短周期タスクはスケジューリング通りに実行できない．

Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

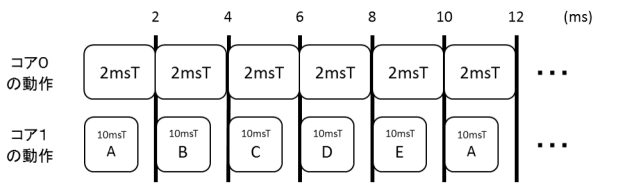

図 4　提案手法による実行パターン

長周期タスクの処理結果を短周期側に反映させるタイミングを次に長周期タスクが起動した時とすると図 4 に示す実行パターンが考えられる．図 4 では長周期タスク内の処理を短周期で実行可能なサイズに分割し実行している．毎周期，長周期タスクの完了を待つが，短周期タスクは周期通りの実行が可能であり，図 3 のように遅延することはない．4 章，5 章では図 4 の実行パターンを実現するための手法を提案する．

4. 負荷分散方法

本章では，長周期タスクを短周期時間内で実行するための分割方法を提案する．

4.1 SHIM による性能見積もり

本研究では長周期タスクをブロック毎の処理量を考慮して分割を行う．ブロック毎の処理量としてルネサス 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見積もりをハードウェア抽象化記述 SHIM(Software Hardware Interface for Multi-many-core) [4] で記述したものを使用した．

SHIM はボード，チップの特徴をパラメータ化し，ソフトウェアにより処理できる XML 形式で表現されている．ボード，チップの特徴にはハードウェアコンポーネントからプロセッサ，メモリ，レジスタなどの情報が記載されている．加えて通信の種類に対応した情報も記載されている．

モデルベース並列化技術では，SHIM により記述された各命令レベルの処理量からブロック毎に持つコード片の処理量を算出する．各命令レベルの処理量は最良実行時間の best，最頻実行時間の typical，最悪実行時間の worst が推定されており，これらを合算することでブロック毎の処理量の best 値，typical 値，worst 値を算出する．3 種の処理量は BLXML 内のブロック毎に要素として追加され，ソフトウェアで扱う事ができる．本研究ではブロック毎の処理量に typical 値を用いて負荷分散を行う．

4.2 グラフ探索による負荷分散

探索対象のグラフとして BLGraph を用いた．BLGraph は Simulink モデル内のブロックをノード，信号線による依存関係をエッジとして表現したグラフで BLXML から情報を抽出し作成される．図 1 の Simulink モデルから生成された BLGraph を図 5 に示す．BLGraph は BLXML からブロックの情報を引き継いでいるので，SHIM から算出した処理量を扱うことが可能である．

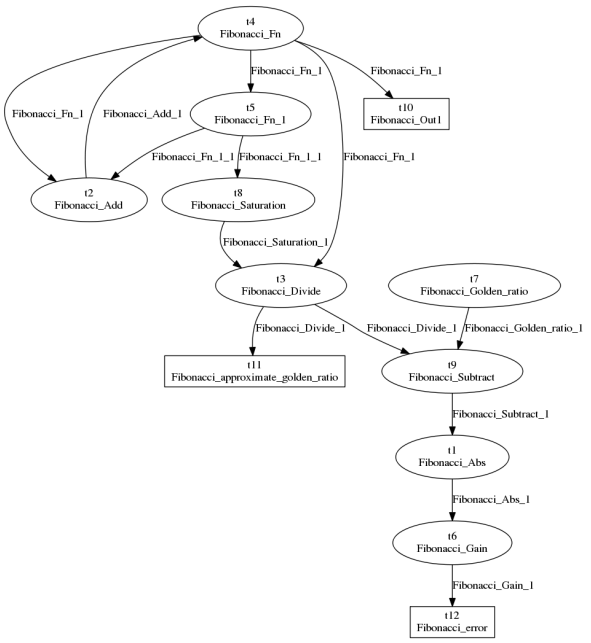

図 5　BLGraph

次に，グラフ探索による負荷分散方法について述べる．負荷分散は再帰的なクリティカルパス分割により行う．本論文において，クリティカルパスとは Simulink モデル内で始点から探索を開始し，終端に至るパスの内，最も処理に時間が必要なパスを指し，今回は処理時間の代わりに SHIM による性能見積もり情報による処理量の合計が最も大きくなるパスとした．

Simulink モデル内のクリティカルパス探索には WarshallFloyd Algorithm[5] を用いた．WarshallFloyd Algorithm はダイクストラ法 [6] を元に，多始点・多終点に対応させたアルゴリズムである．多始点・多終点のグラフ内の最短パスを求めるアルゴリズムであるが，重みの判定を逆にすることで最長パス (クリティカルパス) を求めることが可能である．

BLGraph ではエッジではなくノードに処理量を保持しているため，エッジを探索した後，ノードの処理量を確認し判定するようにアルゴリズムを変更している．クリティカルパス分割による負荷分散は，上述の方法で求めたパスをレート比 (長周期/短周期) に分割することで可能である．5 章で負荷分散情報を用いるため各ノードに分割した際の ID(**オフセット ID**) を振る．

クリティカルパスの分割では，まずパスの合計処理量を算出し，レート比で割ることで分割した 1 ブロック群の処理量の目安を出す．再びパスを始点側から処理量を順次加算しつつ辿り，1 ブロック群の処理量の目安に達するまでオフセット ID に 0 を割り当てる．1 ブロック群の処理量の目安に達したら，合計処理量が 1 ブロック群の処理量の

Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

Algorithm 1 Recursive partitioning algorithm

Input: BLGraph $G = (V, E)$, 始点ノード $S \subset V$, 終点ノード $T \subset V$, および各ノード $v \in V$ の処理量 u_v, グラフ全体の分割数 d, オフセット ID id ($0 \leqq id < d$).
Output: 分割後の BLGraph

```
procedure MAIN
    BLGraph 内の全ての始点ノードの入力ノードとなる v_0 を追加
    Recursive partitioning(v_0)
end procedure
procedure RECURSIVE PARTITIONING(v)
    ノード v からのクリティカルパス探索
    パスの始点と終点の id から分割数を決定
    パス分割及びノードに id 割当て (途中, 分割数に応じて id++)
    終点ノードに id を設定したノードを追加
    パス上のノードで出力エッジが 2 本以上存在するものを再帰探索開始キューに追加
    while 再帰探索開始キューが空でない do
        探索開始キューからノード v 取り出し
        Recursive partitioning(v)
    end while
end procedure
```

目安の 2 倍になるまでオフセット ID に 1 を割り当てる．このようにして，パスの終点までオフセット ID を割り当てる．

続いて，再帰的なクリティカルパス探索について述べる．これはクリティカルパスを分割するだけでは，クリティカルパスから枝分かれしたパスに関してはオフセット ID を振ることができないことに起因している．再帰的なクリティカルパス探索のアルゴリズムを Algorithm1 に示す．

Algorithm1 の Main 内で”BLGraph 内の全ての始点ノードの入力ノードとなるノード 0 を追加”しているのは，短周期タスクから長周期タスクへのエッジが 1 本のみであれば必要ない．しかし複数本あった場合，それぞれにオフセット ID を 0 から振るために記述している．出力側が分岐する箇所に関しては，分岐ノードからのクリティカルパスを算出後，始点ノードのオフセット ID と終点ノードのオフセット ID から分割数を決定する．このパスの終点ノードがオフセット ID が未設定のノードの場合，(レート比 −1) の値，つまり Simulink モデル内に振られるオフセット ID の中で一番大きい値で分割を行う．

例として，短周期 0.2ms，長周期 1.0ms のレート比 5 の場合を考える．この場合，分割数は 5 でオフセット ID は 0∼4 の範囲で設定される．始点ノードのオフセット ID が 1，終点ノードのオフセット ID が 3 の場合，$(3*0.2-1*0.2)/0.2$ で 2 つに分割される．始点ノードからオフセット ID に 1 を設定しつつ，処理量を加算していき，パスの合計処理量の 1/2 を越えたノードからオフセット ID に 2 を設定していく．

再帰的なクリティカルパス分割により，グラフ内のクリティカルパス上だけでなく，そこから分岐しまた合流するノード群，分岐後そのまま終点ノードにつながるノード群を含めたグラフ内全てのノードにオフセット ID を割り当てることができる．

5. オフセット値設定による性能向上

4 章で述べた負荷分散により，オフセット ID を BLXML に追加した．本章ではオフセット ID を設定された長周期タスクの実装方法について述べる．

5.1 Simulink モデルにおけるオフセットの概要

前述の通り，Simulink モデルには各ブロックにモデル全体のサンプル時間 (短周期) の整数倍のサンプル時間 (長周期) を設定することでき，この設定によりマルチレート Simulink モデルを作成することが可能となる．長周期が設定されるブロックには加えてオフセット値を設定することができる．オフセット値を設定することにより，長周期タスクの実行周期を変えずにタイミングを遅らせる事ができる [7]．本来，0.0ms，1.0ms，2.0ms，3.0ms,... と動作していたタスクをオフセット値を 0.2ms で設定することで，0.2ms，1.2ms，2.2ms，3.2ms,... といったタイミングで動作させることができる．

4 章でブロック毎にオフセット ID を設定した．そこでオフセット ID を元に Simulink モデル内の各ブロックのサンプル時間にオフセット値を設定していく．長周期タスク内のブロックへのオフセット値は (短周期)×(オフセット ID) となる．このようにオフセット値を設定することで，図 4 のコア 1 の長周期タスクの実行パターンのように動作させることが可能となる．

5.2 オフセット値設定後のコード

図 4 に対応する Simulink モデル内の長周期タスクに該当する部分のオフセット設定後のコード概要を図 6 に示す．本来，短周期タスクとの通信が存在するが，説明の簡略化のため省略した．短周期 : 2ms，長周期 : 10ms で 5 分割した場合のコードとなっている．rate が現在の周期を表しており，rate_s，rate_l がそれぞれ短周期，長周期を表している．

まず，rate が 0 でタスクループに入るため，ループの初回では rate % rate_l の結果が 0 となり，図 4 中の 10msT A に対応する処理の processing A が実行される．その processing A 完了後，rate += rate_s が実行され，rate の値が 2 になる．次のループに移ると，rate %rate_l の値が 2 となり，図 4 中の 10msT B に対応する処理の processing B が実行される．このようにして，長周期タスク内のコードを負荷分散して実行することが可能となる．

次に長周期タスクから短周期タスクへの接続が存在した場合を想定する．その場合，図 6 内の 15 行目の後に送信のための処理が行われる．したがって，長周期タスクの実行が始まって，最初の 4 回のループでは送信するデータが

Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

```
1  core1_thread
2  {
3      //initialize
4      rate = 0, rate_s = 2, rate_l = 10;
5      //task loop
6      while(1){
7          if(rate % rate_l == 0)
8              //図4中の 10msT A に対応する処理
9              processing A;
10         if(rate % rate_l == 2)
11             //図4中の 10msT B に対応する処理
12             processing B;
13         // 以下，同様の処理を分割数だけ実行
14         rate += rate_s;
15     }
16 }
```

図 6 図 4 に対応する長周期タスクコード例

得られないため，初期値が短周期タスクに送信される．5 回目のループの実行後に初めて長周期タスクの実行結果が得られ，短周期タスクに送信される．図 4 の実行パターンの前提として，1 回目の長周期タスクの実行は 6 回目の短周期タスクが開始する時に間に合えば良いということがあった．これにより，図 6 のコードから図 4 と同様の実行パターンを実現することが可能であると考えられる．

6. 評価

4，5 章で提案した手法について評価を行う．評価項目は”Simulink モデル内に複数の分岐・合流が存在していてもオフセット値を設定できているかの確認”と”一回一回の実行パターンの確認”についてである．

6.1 分岐箇所の分割確認

再帰的なクリティカルパス探索により，Simulink モデルをレート比の数に分割が実行できているか評価を行った．

6.1.1 評価対象モデル

対象の Simulink モデルは長周期タスクのみを抽出したもので，ランダムで分岐と合流を含むものを生成した．評価用の Simulink モデルの生成方法は以下の通りで，使用したブロックはモデルベース自動並列化フローで対応しているもののみである．

(1) 始点ブロックである Inport ブロックを設置
(2) 数個の 1 入力 1 出力ブロックを設置，ブロック間を信号線で接続
(3) 最後に設置したブロックの出力ポートを出力待ちリストに追加
(4) 1~3 個のブロックをランダムで設置，出力待ちリストの信号線からそれぞれのブロックの入力ポートに接続 (分岐発生)

表 1 分割結果 (test1)

分割 ID	part0	part1	part2	part3	part4
ブロック数	98	20	15	10	22
処理量	17295	4257	2464	1648	3757

表 2 分割結果 (test2)

分割 ID	part0	part1	part2	part3	part4
ブロック数	12	227	45	34	20
処理量	2288	43222	7173	3894	4072

(5) 出力待ちリストから選択された信号を排除，(4) で設置したブロックの出力ポートを出力待ちリストに追加
(6) 規定数のブロックを設置するまで (4) に戻りループ
(7) 出力待ちリスト内の出力ポートに終点ブロックの Outport ブロックを接続

(4) の手順では，その時接続する必要がある入力ポートと出力待ちリスト内の出力ポートの数を比較する．比較結果により以下の作業を行う．

入力数 > 出力数
出力ポートからランダムに入力ポートに接続 (分岐)

入力数=出力数
出力ポートからランダムに入力ポートに接続

入力数 < 出力数
入力ポートからランダムに出力ポートに接続，選択された出力ポートのみ出力待ちリストから排除

合流は多入力ブロックを用いて行った．このようにして，評価用モデルを作成した．test1 と test2 の 2 つのモデルを生成し，再帰的なクリティカルパス探索による負荷分散を行った．それぞれブロック数は 165 と 338 である．

6.1.2 評価結果

それぞれの分割結果は表 1，2 の通りである．各モデルとも分割は行えているが，test1 の part0 と test2 の part1 が他のブロック群に比べブロック数，処理量共に大きくなってしまっている．この結果から再帰的なクリティカルパス分割では対応しきれないパターンが存在することが分かる．

Simulink モデル内で再帰的にクリティカルパス分割を行う場合，パスの始点と終点にそれぞれ既にオフセット ID を設定されている場合がある．当然，そのオフセット ID が分岐・合流のパターンで同じの場合もあるため分割されないパターンが存在する．そういった場合が多くなると本評価のような偏りが発生してしまうと考えられる．

6.2 実行パターンの確認

5 章でオフセット値設定後のコードについて述べたが，実際に短周期と長周期を並列動作させた場合に，反映されていることを確認した．図 7 のマルチレート Simulink モデルを対象とし，長周期側の実行時間を約 0.1sec に調整した．短周期側には，ほとんど処理がない．

今回の評価では短周期タスクの最後で実行時間を計測し

Vol.2017-ARC-225 No.46
Vol.2017-SLDM-179 No.46
Vol.2017-EMB-44 No.46
2017/3/10

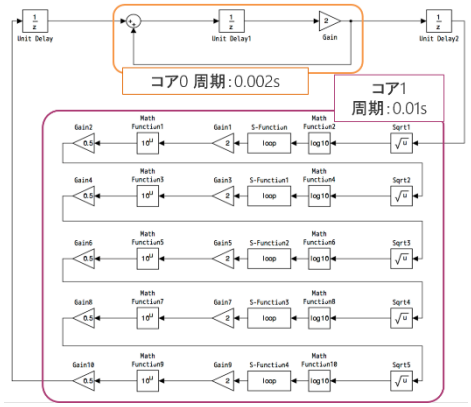

図 7　対象のマルチレート Simulink モデル

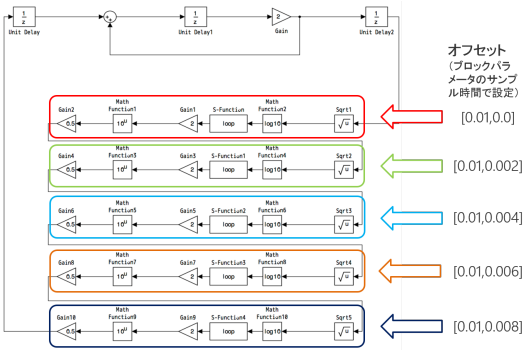

図 8　オフセット設定後の Simulink モデル

た．また実行時間がオフセット設定前後で変化することを確認することが本評価の主目的であるため，決められた時間間隔で周期実行するリアルタイム OS ではなく Linux 上で動作させた．そのため短周期・長周期タスク共に実行可能な状態になると次周期の実行に移る．

図 7 のマルチレート Simulink モデルへのオフセット値の設定は図 8 のようになった．コア 1 に割り当てられている長周期タスクに対して，始点側からオフセット値をサンプル時間に設定している．[0.01，0.002] というのはサンプル時間 0.01，オフセット値 0.002 ということである．

図 9 はオフセット値設定前後の実行結果である．オフセット値設定前は長周期タスクの実行がある場合にのみ実行時間が長くなっていることが確認できる．それに対して，オフセット値設定後は各サイクルでほぼ同じ実行時間である．よって，オフセット値を長周期タスクに適切に設定することにより，短周期タスク側で待つ時間が一定になることが確認できた．

また，本評価では短周期タスクの実行時間はほぼ 0sec に設定したが，短周期タスクの実行時間が約 0.02sec までであれば，図 9 の時間で実行可能である．これは短周期タスクは長周期タスクの実行を待っているだけで，内部の処理を行っていないからである．その場合，オフセット値設定前のコードに比べ式 1 の性能向上が可能と見込まれる．

$$performace = \frac{2 \times r - 1}{r} \qquad (1)$$

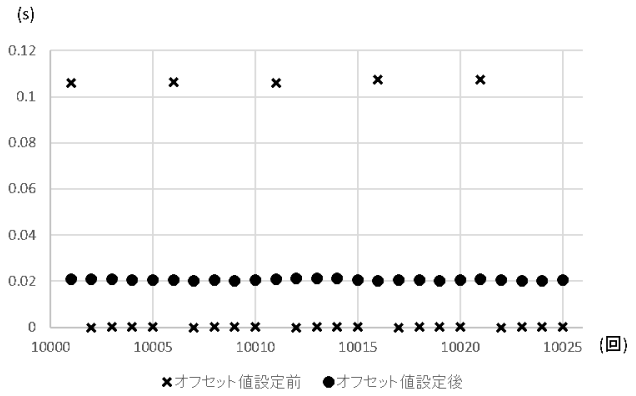

図 9　オフセット値設定前後の比較結果

r はリアルタイム OS の場合はレート比，汎用 OS の場合は長周期タスクと短周期タスクの実行時間の比である．

7. おわりに

本論文では, 複数周期を持つ Simulink モデルを対象としたソフトウェア並列度向上手法を提案した．評価では，再帰的なクリティカルパス探索による負荷分散方法とオフセット設定による並列度向上方法について行った．前者は評価モデルを作成し，オフセット ID の振り方を確認することで行った．後者はブロック毎の処理量を目安に長周期タスクを分割し，オフセット値を設定することで短周期タスクの実行を一定の周期で実行できることを確認した．

今後の課題としては，まず提案手法では処理量の偏りが発生してしまった負荷分散方法の改善が挙げられる．次に長周期タスク内の負荷分散の妥当性を確認する機構の実装が必要である．また長周期タスクと短周期タスクの処理量に大きな差がある場合に，異なる周期のタスクを一つのコアで動作させることを考慮した負荷分散方法について検討を進めていきたい．

参考文献

[1] MathWorks: MATLAB/Simulink, http://www.mathworks.co.jp
[2] 山口，池田，枝廣他: Simulink モデルからのブロックレベル並列化，組込みシステムシンポジウム 2015(ESS2015),2015,pp.123-124.
[3] eSOL:メニーコアプロセッサ対応スケーラブルリアルタイム OS http://www.esol.co.jp/embedded/emcos.html
[4] M. Gondo, F. Arakawa, and M. Edahiro: Establishing a Standard Interface between Multi-Manycore and Software Tools - SHIM, COOL Chips XVII, VI-1,2014.
[5] Floyd, Robert W. ”Algorithm 97: shortest path.” Communications of the ACM 5.6 (1962): 345.
[6] Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. In Numerische Mathematik, 1 (1959), S. 269 ～ 271.
[7] MathWorks: Specify Sample Time, https://jp.mathworks.com/help/simulink/ug/how-to-specify-the-sample-time.html?lang=en