Vol.2013-EMB-30 No.2
2013/9/17

マルチレート制御モデルのイベントドリブンプロセッサ実装

大川 禎[1]　枝廣 正人[1]

概要：近年，マルチコア・メニーコアが組込みシステムにおいても主流となりつつある．また，制御システムの大規模複雑化による開発工数の増大が課題となっている．そのため，メニーコアであるイベントドリブンプロセッサを用い，タスクをコアに対応させることで制御モデルの実装を行うことを考える．これにより，同一コア内でのタスクスケジューリングが不要となり，開発工数の削減が期待できる．このとき，複数の制御周期が混在するマルチレートモデル，特に制御周期が動的に変化する場合が課題となる．本論文ではマルチレート制御モデルをイベントドリブンプロセッサに実現する手法を提案し，ベンチマークとして制御モデルを実装することで，提案手法の有効性を評価した．

1. はじめに

近年，CPU 設計の大規模複雑化，消費電力の増大などの問題により，単一 CPU の性能向上は限界に到達しようとしている．そのため，現在は，CPU 内に複数のプロセッサ (コア) を搭載したメニーコアが組み込みシステムにおいても主流となりつつある．しかし，プロセッサのメニーコア化によるハードウェアの性能向上だけではシステムの高速化は期待できず，メニーコアリソースを効率的に活用するために，実装するソフトウェアが並列化を意識して設計されていなければならない．

並列化が有効なソフトウェアモデルとして，CSP(Communicating Sequential Processes)[2] があげられる．CSP は，システム全体をイベント駆動によって独立して実行されるプロセスの集合として並行処理システムを表現する．内部の処理と非同期に入力を受け付けるようなシステムや，リアルタイム性を要求されるシステムを記述する上ではモデル記述が容易である．

また，近年開発されているイベントドリブンプロセッサ [4] は，ハードウェアのサポートにより CSP モデルの実装を可能としているメニーコアプロセッサである．モデルの設計からソフトウェア開発を CSP によって記述し，イベントドリブンプロセッサへ実装することで，システム開発の抽象的なレベルからハードウェアレベルまで並列化を実現することができる．

もう一つの背景として，制御システムの大規模複雑化による開発工数の増大が課題となっている．本論文では，メニーコアであるイベントドリブンプロセッサを用い，タスクをコアに対応させることで制御モデルを実装する．これにより，同一コア内でのタスクスケジューリングが不要となり，開発工数の削減が期待できる．また，複数の制御周期が混在するマルチレートモデル，特に制御周期が動的に変化する場合を踏まえ，マルチレート制御モデルをイベントドリブンプロセッサに実現する手法を提案する．本論文で提案する手法によりベンチマークとして制御モデルを実装し，提案手法の有効性を評価した．

[1] 名古屋大学 情報科学研究科
Graduate School of Information Science, Nagoya University

2. CSP モデル

2.1 概要

CSP モデルとは，並行システムにおける相互作用のパターンを記述する仕様記述言語 CSP(Communicating Sequential Processes) によって記述されるモデルである．複数のプロセスと，プロセス間の通信路であるチャネルによってシステムが定義される [2]．各プロセスは独立して逐次実行され，他プロセスとメッセージ通信を行う．通信を必要とするプロセス間にはチャネルが定義され，プロセス間で通信されるデータはチャネルを介して送受信される．

CSP システム全体は複数のプロセスに分割することができ，各プロセスは独立して逐次実行される．プロセス間で依存するデータは，チャネルによってプロセス間で受け渡される．例えば，あるプロセス A が変数 n を操作し，別のプロセス B が操作された n を必要としているとき，プロセス B はプロセス A に依存し，プロセス A はチャネルを介してプロセス B へ変数 n を送信する．

CSP モデルを視覚化すると，次のようなデータフローダイアグラム (DFD) で表せられる．

楕円をプロセス，楕円をつなぐ矢印がチャネルを表す．

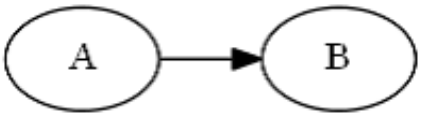

図 1 CSP モデル例

図 1 では，2 つのプロセスと 1 つのチャネルが定義されている．プロセス B が必要としているデータはプロセス A がチャネルを介してプロセス B へ受け渡している．図 1 のチャネルは，プロセス A の出力チャネルであり，プロセス B の入力チャネルである．

2.2 プロセス間の通信

プロセス間の通信はチャネルによって行われ，チャネルによる通信は次の 3 つの特徴を持つ．

(1) チャネル通信はバッファを持たない
(2) チャネルの接続形態は一対一
(3) データの流れは単方向

1 の性質から，プロセス間のデータの送受信は同期して行われる．例えば，送信側プロセスがチャネルへデータを送信できる (Ready) 状態になった際に受信側プロセスが受信できる (Ready) 状態でないとき，通信するデータを蓄積させて処理を続行することができないため，受信側のプロセスが Ready 状態になるまで送信側のプロセスは待たされることになる．逆に，受信側のプロセスが Ready 状態にあるとき，受信されるべきデータは蓄積されてない (バッファが存在しない) ため送信側のプロセスが Ready 状態になるまで受信側のプロセスは待たされることになる．したがって，チャネル通信における送信と受信はタイミングが同期される．

2.3 プログラミング言語 occam による CSP の実装

CSP モデルは並列プログラミング言語 occam によってプログラムとして実装される．本論文では occam を用いた実装は行わないが，occam の仕様を継承した言語によって CSP モデルを記述する．本節では，本論文に関連する occam の言語仕様について述べる．

SEQ は次に続く式のリストを逐次実行することを示す．他の言語とは異なり，occam においては SEQ のない式のリストを暗黙的に逐次実行することはない．

次の例は，変数 x を 1 だけインクリメントした後 x を 2 乗した値を変数 y に代入する命令を順に行う．

```
1: SEQ
2:     x := x + 1
3:     y := x * x
```

PAR は，それに続く式のリストは並列的に実行可能であることを示す．実装時は，PAR 内の命令はそれぞれ別スレッドとして実行されることになるが，実装の仕様によっては必ずしも並列に実行されるとは限らない，例えば，実装環境に並列に実行できるだけのリソースが足りなければ PAR 内の式をシーケンシャルに実行することになる．occam の記述は，並列実行を許可するだけであり，実際に並列実行されるかどうかは実装に依存する．

次の例は，変数 x を 1 だけインクリメントする命令と，変数 y に y*2 を代入する命令を並列に実行することを許可している．

```
1: PAR
2:     x := x + 1
3:     y := y * 2
```

2.4 チャネル

あるプロセスがチャネルへデータを送信するとき，'!' を用い，チャネルからデータを受信するとき，'?' を用いる．

次に，変数 c をチャネル channelA へ送信する例，および channelA から変数 c を受信する例をそれぞれ示す．

```
channelA ? c

channelA ! c
```

上記の 2 つの命令が並列に実行されることで，それぞれの命令が属するプロセスがチャネルによる通信を実現できる．

2.5 選択実行

あるプロセスが複数の受信チャネルを持ち，全て待ち状態であるとき，1 つのチャネルが Ready 状態になったときに対応するコードを実行することができる．複数のイベントの内どれか 1 つだけ発生したら動作を開始するようなシステムを記述する場合に用いられる．

選択実行を行うプロセスの例を図 2 に示す．

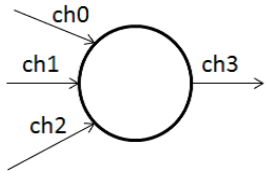

図 2 選択実行を行うプロセス例

また，コード例を次に示す．

```
1: ALT
2:     ch0 ? data
3:       SEQ
4:         data += 1
5:     ch1 ? data
6:       SEQ
7:         data += 2
```

```
 8:      ch2 ? data
 9:        SEQ
10:            data += 3
11: ch3 ! data
```

このコードは，ch0,ch1,ch2 の 3 つのチャネルのいずれかからデータを受け取ったとき，受け取ったデータに対し，受けとったチャネル対応した演算を行い ch3 にその値を送信する．このプロセスは，3 つのチャネルの内いずれかから受信するまで，チャネル送信は行われない．また，一つのチャネルから受信があったとき，他のイベントから発生する命令は実行されない．なお，待ち状態にある複数のイベントが同時に発生した場合，実行される命令は不定である．

3. イベントドリブンプロセッサ

本論文で用いるイベントドリブンプロセッサは，分散型のメニーコアであり，イベント通知によるプロセス間通信を前提としたシステムを実装することを目的としている．本論文では，CSP モデルの実装をおこなうイベントドリブンプロセッサとして XMOS を用いた．

3.1 マルチスレッド

XMOS は 1 つのコアで複数スレッドの並行実行をハードウェアレベルで実現できるマルチスレッドプロセッサである．XMOS はプロセッサ内に XCore と呼ばれるコアを搭載し，XCore は図 3 のような構造となっている．

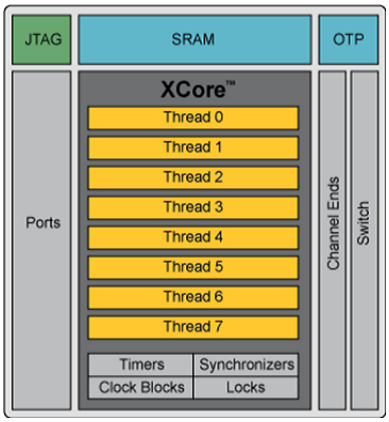

図 3 XCore

XCore は 8 つのレジスタ群 (スレッド) を持ち，このスレッドが CPU の役割を持つ．そして，それらのスレッドを時分割実行することで，1 つの XCore で 8 つのプロセスを並行実行する事ができる．これにより XCore はハードウェアマルチスレッディングを実現している．

XCore には 32 ビット RISC プロセッサとスレッド・スケジューラ回路，タイマ，クロック生成回路，同期/ロック機構がまとめて搭載され，XCore と，I/O ピンとのインターフェース回路や，RISC プロセッサで実行中の命令と処理対象データを置く SRAM，XC 言語で開発したソフトウエアを格納する OTP（One Time Programmable）メモリー，コア間のインターコネクト回路（「XLink」とよぶ）で単位プロセッサ・コア（「タイル」とよぶ）を構成した上で，チップ上では複数のタイルをアレイ状に接続している．

クロック周波数は最大 400〜500MHz．1 個のタイル当たり 400〜500MIPS の処理性能が得られる．例えば 4 つのタイルを集積すれば，処理性能は最大 1600〜2000MIPS に達する．

なお，本論文で用いる XMOS は浮動小数点演算器を実装しておらず，浮動小数点演算はソフトウェアエミュレーションを行う必要がある．

3.2 XC

XMOS の動作は XC 言語で記述される．XC 言語は前章で述べた occam の仕様を ANSI C に追加することで，ソフトウェア処理の並列性や動作タイミングを記述できるように拡張されている．

XC ではチャネル，ポート，タイマ変数が新しく定義される．チャネルはプロセス間で送受信されるデータを通信する．ポートはプロセッサが持つ I/O ポートから送受信されるデータの通信を行う．タイマ変数は，プロセッサの各コアが持つタイマを指し，コード上では時刻の取得先を示す．これらは記号：>，<：を用いて occam と同様に値の受け渡しをして通信することができる．言語仕様は C を踏襲しているため，occam と異なり SEQ 構文は存在せず，式のリストは C の仕様通り逐次実行される．また，'par' により囲まれたブロックは，ブロック内の各式を並列実行させる．XC では par 構文によってのみスレッドを新しく生成し，par 構文内の各式は必ず別スレッド領域に割り当てられる．ハードウェアリソースの不足によってスレッドが生成できなくなる場合，XC コンパイラはエラーを出力する．

CSP モデルの各プロセスは XC の関数で実装される．XC は main 関数と，各プロセスを記述するプロセス関数を記述する．main 関数は CSP モデルが持つチャネルの定義と，プロセス関数の呼び出しおよびプロセスのコア割り当てを行う．プロセス関数は par 構文内部で呼び出し，並行実行させる．また，on 構文によって割り当てコアを指定することができる．陽にプロセスの割り当てコアを指定しない場合，XC コンパイラはそのプロセスを 0 コア目に割り当てる．

3.3 スレッド制約

CSP モデルをイベントドリブンプロセッサへ実装する際，ハードウェアのリソース制限によって，プロセスのコア割り当てには制約条件が存在する．

複数の入力チャネルを持つプロセスのチャネル入力処理は，システムのデッドロックを回避するためにそれぞれのチャネル受信処理を並行実行しなけらばならない．よって，各チャネル受信処理を par 構文内に記述する必要がある．これにより，1 つのプロセスで入力チャネルの数だけスレッド領域を要することになる．

プロセスが出力チャネルを複数持つ場合も同じく，それぞれのチャネル出力処理を par 構文により並行実行する．これも必要とするスレッド領域数は出力チャネル数と一致する．

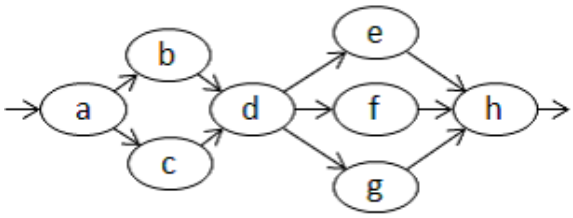

図 4 複数のスレッド領域を必要とするタスクをもつ CSP モデル例

図 4 において，プロセス d は入力チャネルを 2 つ持ち，出力チャネルを 3 つもつ，よって，プロセス d はスレッド領域を 3 つ必要とする．

ここで，図 4 の CSP モデルの各プロセスが必要とするスレッド領域数を表に示す．

表 1 図 4 の各プロセスの必要スレッド領域数

プロセス名	入力チャネル数	出力チャネル数	スレッド領域数
a	1	2	2
b	1	1	1
c	1	1	1
d	2	3	3
e	1	1	1
f	1	1	1
g	1	1	1
h	3	1	3

問題となるのは，1 つのプロセスは 1 つのコアにしか割り当てられないことと，1 つのコアが同時並列実行できるスレッド領域数は有限であるということである．つまり，コアのスレッド領域に空きがあっても，プロセスが必要とするスレッド領域数に満たない場合はそのプロセスは割り当てることができない．例えば，図 4 において，コアが持つスレッド領域数が 4 であるコアにプロセス a とプロセス b を割り当てたとき，プロセス a は 2，プロセス b は 1 のスレッド領域を使用する．したがって，そのコアの空きスレッド領域数は 1 となり，プロセス d は割り当てられなくなる．

CSP モデルのイベントドリブンプロセッサへの実装時にコア割り当てをおこなう際は以上のことが制約条件となる．本論文においてはこの制約条件下でコア割り当てをおこなう．

3.4 タイマ

XMOS はチップ上にナノ秒の分解能を持つタイマを実装している．XC コード上でタイマを用いる場合は，タイマ変数からチャネル受信をすることで時刻を取得することができる．タイマ変数はレジスタを示し，プロセッサ起動からの経過時間を示している．

タイマをもちいたコード例を次に示す．

```
1:void xc_task(chanend input1,
               chanend input2,
               chanend output1){
2: timer tmr;
3: unsigned int time;
4: unsigned int rate = 100000;
5:
6: tmr :> time;
7: tmr when timerafter(time+rate) :> void;
8:}
```

2 行目はタイマ変数を宣言し，3 行目で時刻を格納する変数を宣言している．時刻の取得は 6 行目で行う．7 行目は，タイマ変数が $(time + rate)$ 以上になるまで待機する処理である．これらを用いることで，プロセスの周期制御を記述することができる．

4. マルチレート制御モデル

本論文ではフィードバック制御を行うモデルを対象とし，制御モデルとして MATLAB Simulink モデル [5] を仮定する．フィードバック制御モデルは，外界からの信号を入力値として制御値を出力し，制御した結果を入力側へ戻す処理を行うものである．外界からの入力値に対し出力制御値を決定するコントローラと，コントローラから得られた信号を入力としフィードバック値を出力としてコントローラへ返すプラントで構成される．

フィードバック制御モデルへは複数回の入力が行われ，1 入力に対しコントローラ，プラントそれぞれで内部処理が行われる．コントローラは，外界からの入力値と，前回の入力に対するフィードバック値に対し，出力をプラントへ渡す．プラントは，1 入力に対し制御する機構の内部処理を行い，コントローラへフィードバック値を出力する．モデルへの 1 入力に対し，フィードバック値が出力されるまでを 1 ステップとする．

4.1 マルチレート

前述のとおり，CSP においてはチャネルによる通信は同期することから，制御モデルを複数ステップ実行する上で，送信側タスクと受信側タスクで通信回数は一致する必要が

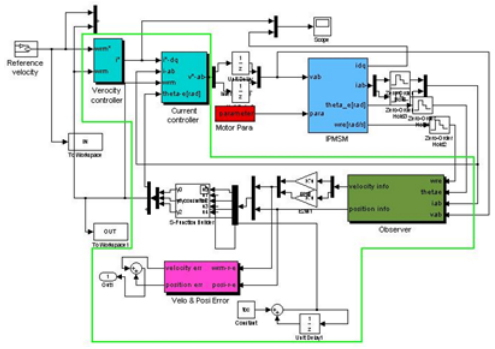

図 5 モーター制御を行う Simulink モデル

あり，通信の周期 (レート) が異なるタスクをチャネル接続すると，実行時にデッドロックが発生してしまう．しかしながら，マルチレート制御モデルにおいては，モデル内の各タスクが自身の制御周期に基づいて動作するため，タスク間のレートが異なることは避けられない．

レートの異なるタスク間通信手法についてはタスク間で共有メモリを用いて，各タスクが任意のタイミングで共有メモリへアクセスすることで実現する方法が考えられる．しかしながら，分散型であるイベントドリブンプロセッサにおいては，共有メモリをもちいることができないため，次章で実現手法を述べる．

5. 実装手法

本節では，マルチレート制御モデルからイベントドリブンプロセッサ実装までのフローを述べる．本論文では制御モデルとして MATLAB Simulink モデルを仮定する．

5.1 モデルベース並列化ツール

MATLAB Simulink により記述された制御モデルは MATLAB の機能である Real-Time Workshop Embedded Coder により，逐次 C コードへ変換される．なお，この C コードはマルチコアで実行可能な並列処理として記述されていない．そのため，この逐次コードはマルチコアにおいては高い性能を発揮できない．そこで，モデルベース並列化ツール [3] を用いて，Simulink モデルと逐次処理 C コードから，マルチコア向けに記述された並列処理 C コードに並列化する．これは CSP 理論に基づき記述されたモデルとなっている．

なお，モデルベース並列化ツールが Simulink モデルから並列 C コードを生成する手順は，以下のステップで構成される．

(1) C コード解析 Simulink から出力された逐次処理 C コードを基本ブロックに分割．
(2) モデル解析 Simulink モデルから，ブロックの接続関係，ブロックの階層構造を読取る．
(3) 中間モデル構築逐次処理 C コードとモデルの解析結果にもとづいて，並列 C コード生成用の中間モデルを作成．
(4) 並列コード生成ターゲット OS に合わせて，中間モデルから並列 C コード生成．

なお，モデルベース並列化ツールが出力するコードはすべてのタスクが等しいレートで動作する前提となっており，本論文で対象とするマルチレート制御モデルにはそのまま適用できない．

5.2 実装フロー

Simulink で記述されたモデルをイベントドリブンプロセッサへ実現するまでのフローを図 6 に示す．

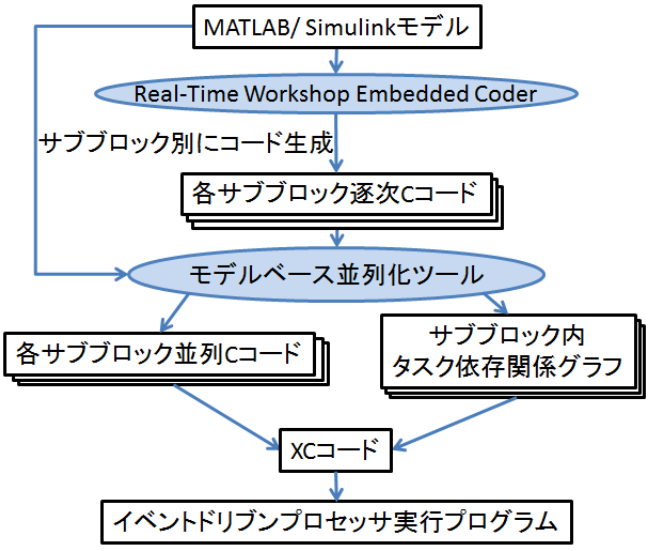

図 6 マルチレート制御モデルのイベントドリブンプロセッサ実装

まず，Simulink モデルにおいてレートが等しいタスクの集合を同一のサブブロックとしてまとめる．これにより，モデルを異なるレートで動作するサブブロックの集合とみなす．次に，Real-Time Workshop Embedded Coder により，サブブロックごとに逐次 C コードを生成する．これにより，サブブロック内はシングルレートで動作し，モデルベース並列化ツールをもちいた並列化が可能となる．モデルベース並列化ツールはサブブロックに対して並列 C コードおよびサブブロック内のタスク依存関係を生成する．これらをもとに，サブブロック同士の通信を記述した XC コードを生成する．XC コードは，XC コンパイラにより XMOS の実行プログラムへコンパイルされる．

5.3 レートの異なるタスク間通信

マルチレート制御モデルにおいて，レートが異なるタスク間の通信手法を提案する．

2 つの受信チャネルと 1 つの送信チャネルを持つタスクの例を，図 7 に示す．2 つの受信チャネルからは，それぞれ自身と異なるレートでチャネル受信を行う．

図 7 で示す CSP モデルにおけるコード例を示す

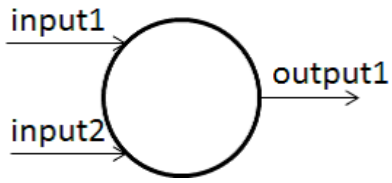

図 7 レートの異なるタスクから受信するタスク例

```
1:void xc_task(chanend input1,
               chanend input2,
               chanend output1){
2: int in1 = 0, in2 = 0;
3: int out1;
4:
5: timer t;
6: unsigned int time;
7: unsigned int rate = 100000;
8:
9: while(1){
10:  t :> time;
11:  select{
12:   case input1 :> in1:
13:    break;
14:   case input2 :> in2:
15:    break;
16:   case t when timerafter(time+rate) :> void:
17:    t :> time;
18:    out1 = in1 + in2;
19:    output1 <: out1;
20:    break;
21:  }
22: }
23:}
```

このコードはチャネル受信，タスク内部処理，チャネル送信を繰り返し無限に行う．select は Occam における ALT に相当する選択実行構文である．この例では，3 つの case 文により

- input1 チャネル受信
- input2 チャネル受信
- レート秒経過

のいずれかのイベントが発生したとき，対応する case 内の文が実行される．これらのイベントは実行される順番，タイミングは不定であり，タスクが自身の制御周期に達したときにチャネル送信を行い，それ以外のときは常に 2 つの受信チャネルが Ready 状態となっている．

このように記述することで，受信側タスクのレートに関わらず，レートの異なるタスク間通信を送信側タスクのレートで行うことができる．

6. 評価実験

イベントドリブンプロセッサへ実装したマルチレート制御モデルの評価を行う．ベンチマークとして，タスク数 27，サブブロック数 5 のモーター制御モデルを実装した．また比較対象として Intel プロセッサを用い，イベントドリブンプロセッサと比較を行った．

本実験における評価環境は以下のとおりである．

XMOS XK-XMP-64 400MHz 16 コア

Intel Intel Corei7 980X 3.33GHz 4 コア

Intel での実装においては，モデルベース並列化ツールから WindowsAPI によるコードを出力し，各サブブロックを接続するコードを C で記述した．サブブロック間通信は共有メモリを用い，サブブロック内のタスクが任意のタイミングで通信することでマルチレートを実現した．

6.1 実験結果

各プロセッサ上で実装したモーター制御モデルをそれぞれ 1000 ステップ実行し，プラント部が出力するフィードバック値と，MATLAB 上でのシミュレート結果を比較した結果，XMOS，Intel のどちらも出力結果は一致した．以上から，マルチレートによるモーター制御モデルを実現することができた．

6.2 性能評価実験

図 8 に示す小規模なモデルを各プロセッサで実行した．

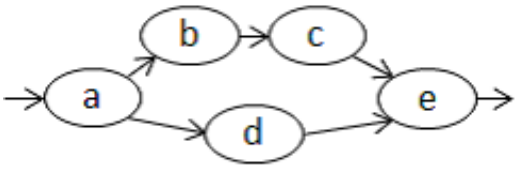

図 8 タスク数 5 のマルチレートモデル

図 8 のモデルは各タスクが入力値に対し加算命令を行い，異なる周期で通信を行うモデルであり，モデルは制御周期が出力結果に依存しない．

全体のタスクの制御周期を短くしていき，いずれかのタスクが制御周期を満たさなくなる限界を見つけることで，制御モデル実装において各チップの性能差を比較した

制御周期の限界を表 6.2 に示す．

表 2 各プロセッサにおける制御周期の限界

プロセッサ	制御周期の限界
XK-XMP-64 400MHz 16 コア	約 0.05 (μ s)
Intel Corei7 980X 3.33GHz 4 コア	約 20 (μ s)

表 6.2 より，同じモデルにおいて XMOS の方がより短い制御周期で動作可能である．したがって，XMOS が Intel に比べ高速に実行できることがわかった．

このモデルにおいては，演算量が少なく浮動小数点型の演算を持たないことと，Windows マルチスレッドの負荷が大きいこととが要因であると考えられる．

6.3 考察

各チップの消費電力を表 6.3 に示す [4][1].

表 3 消費電力

プロセッサ	最大消費電力
XK-XMP-64	32(W)
Corei7 980X 3.33GHz	130(W)

表 4 図 8 のモデル 1 ステップ実行における最大必要エネルギー

プロセッサ	最大必要エネルギー
XK-XMP-64	1.6(μ J)
Corei7 980X 3.33GHz	2600(μ J)

性能評価実験の結果から．XMOS は Intel に比べ高い性能が期待できることがわかった．そして，特に演算量が少なく，タスク数が多いモデルにおいては特にその傾向が強いと考えられる．また，XMOS チップは Intel に比べ消費電力が低く，組込みを想定した制御処理の実装において，イベントドリブンプロセッサに優位性があると考えられる．例えば，図 6.3 で示すマルチレートモデル 1 ステップを実行するために必要なエネルギーは，表 6.3 に示す通り 1000 倍を超える差となる．しかしながら，XMOS では 1 コア 8 スレッドまでしか実行できないことから，CSP モデルの生成時に制限が生まれてしまう点が課題である．

7. まとめ

モーター制御モデルのイベントドリブンプロセッサへの実装を行った．制御周期の異なるタスクの混在するマルチレートモデルにおいては，制御周期ごとにサブブロックとしてまとめ，サブブロック間の通信を任意の周期で行うことで実現した．評価結果から，本論文で提案する手法によりマルチレート制御モデルが正しく動作することを確認した．これにより，メニーコア実装における開発工数の削減が期待できる．また，演算量が少なく，タスク数が多いモデルの実装においては性能，消費電力の面でイベントドリブンプロセッサに優位性が期待できることがわかった．

参考文献

[1] ARK — Intel; Core i7-980X Processor Extreme Edition. http://ark.intel.com/ja/products/47932.
[2] C. A. R. Hoare, *Communicating Sequential Processes*. the United Kingdom, Prentice Hall, 2011.
[3] T. Kumura, Y. Nakamura, N. Ishiura, Y. Takeuchi, M. Imai, Model Based Parallelization from the Simulink Models and Their Sequential C Code. SASIMI 2012 Proceedings.
[4] D. May, The XMOS XS1 Architecture. XMOS Limited, 2009. http://www.xmos.com/published/xs1_jp.
[5] Simulink - シミュレーションおよびモデルベース デザイン (MBD) - MathWorks, http://www.mathworks.co.jp/products/simulink/, 2013/02/07.