マルチコア適用ガイド Ver1.0

2021 年 6 月

組込みマルチコアコンソーシアム

マルチコア適用ワーキンググループ

Copyright © 2021 Embedded Multicore Consortium ALL RIGHTS RESERVED.

本資料は、組込みマルチコアコンソーシアムのマルチコア適用ワーキンググループ活動で 2019 年度に作成した、4 つのマルチコア適用ガイド資料を統合化し、公開するものです。

<統合化された 4 つのガイドライン>

- マルチコア適用ガイド
- 制御系マルチコア・ハードウェアの特徴とユースケース
- 自動車 機能安全へのマルチコア適用
- 並列処理ソフトウェアの課題と対策技術

更新履歴

Rev.	日付	更新内容
Ver1.0	2021/06/30	初版

本資料の作成及び公開において、ご協力頂いた方々に感謝申し上げます。

- 作成協力者【カッコの中は 2019 年度在籍の機関名を記載】

  稲垣浩之※（アイシン精機株式会社）、本田晋也※（名古屋大学大学院）、枝廣正人※（名古屋大学大学院）、納富昭※（オスカーテクノロジー株式会社）、倉田信一郎※（オスカーテクノロジー株式会社）、竹内成樹※（ガイオ・テクノロジー株式会社）、生沼正博※（ガイオ・テクノロジー株式会社）、鈴木均（ルネサス エレクトロニクス株式会社）、藤本洋（イーソル株式会社）

  注：※は、本資料のベースとなる統合化前ガイドライン（MS-PowerPoint）作成協力者です。

  本資料の MS-Word 文書化及び統合化には、関与されておりません。

Contents

1 ＜並列化フロー＞　完成度の高いマルチコアソフトウェアを効率よく作成するための開発手順

プログラムを並列化し、静的・動的解析により性能を確認

■ 本章の対象読者

知識・経験：レベル 1（入門者）シングルコアの知識・開発経験のみ

プロセス　：設計、実装、テスト

ドメイン　：組込み全般

キーワード　　：タスク、性能解析、依存解析、デバッグ、プロファイリング

■ 本章を読んで得られるもの

マルチコア向けにソフトウェアを並列化する手順が分かる。

ソフトウェアの並列化に際して確認すべき事項や開発環境の概要が分かる。

■ 要旨

従来の製品開発プロセスの下でマルチコアソフトウェアの完成度を高めようとすると、いくつかの課題に突き当たるが、その中でも特に大きな課題は「並列性能の確保」と「品質の確保」だろう。

並列性能は、マルチコアのハードウェアアーキテクチャ、その上で動作するソフトウェアの構成や処理特性、およびコーディングスタイルなどに左右される。期待する性能を確保するためには、これらを総合的に勘案したシステム設計、および実際にソフトウェアを並列化し、性能を確認するための手段が必要になる。一方、高い品質を確保するためには、テスト環境やテスト手法の見直しが求められる。場合によっては、従来のプロセスに手を加える必要があるかもしれない。

本章では、これら二つの課題のうち、特に並列性能の確保に有効なマルチコアソフトウェアの開発手順について説明する。

1.1 ソフトウェアをどのように並列化するのか

ソフトウェアを、マルチコアで動作させること、もしくはマルチコアで動作するように変換することを「並列化」と呼ぶ。最初に、ソフトウェアを並列化するための手法や開発プロセスの課題について説明する。

1.1.1　並列化の粒度

並列化の粒度について説明する。なお第 1 部では、OS を採用するしないにかかわらず、組込みソフトウェアシステムにおいて処理する仕事の実行単位を「タスク」と呼ぶ。

マルチタスクの組込みソフトウェアシステムを並列化する際には、どのような単位（粒度）でソフトウェア

をコアに割り振るかという点で、いくつかの選択肢がある。

1.1.1.1 タスク内並列処理

一つのタスクの中で同時に実行できる処理を抽出し、異なるコアに割り当てる。並列化できる単位はソフトウェア内のあらゆる処理になるため、並列化効率を最大限に高められる。

タスク内の処理の中から並列実行できる処理を抽出するのは人手では困難。自動並列化ツールなどの支援が必要になる。

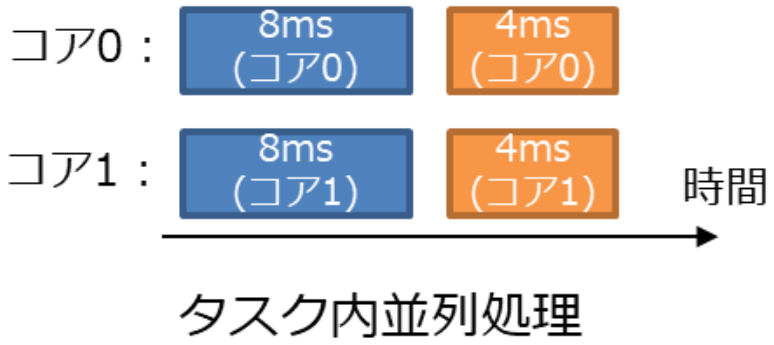


図 1-1 タスク内並列処理

1.1.1.2 タスク間並列処理

異なるタスクを異なるコアに割り当てる。タスク間の依存関係を考えて時間とコアの配分を考える必要はあるが、タスク内並列処理より難易度が低い。

並列化できる単位がタスクの大きさ（粒度）に制限されるため、並列化の効率も制限を受ける。

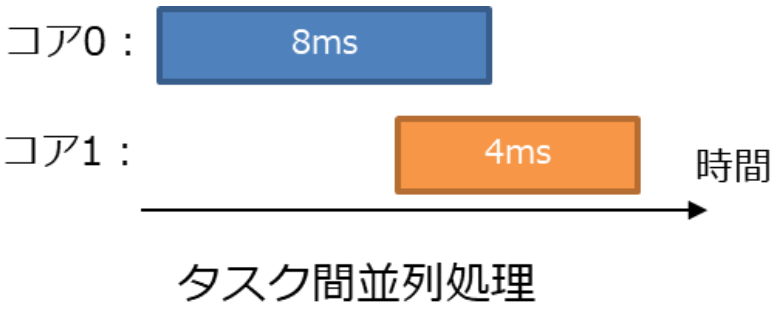


図 1-2 タスク間並列処理

1.1.1.3 ハイブリッドな並列処理

前述のタスク内並列処理、タスク間並列処理を合わせたハイブリッドな並列処理も考えられる。

タスク内並列にしても、タスク間並列にしても、同時に実行可能な処理の量には限界がある。これは、後述する並列化の阻害要因が存在するためである。

それらの要因を排除しつつ、並列化効率を最大にするためには、下図のようなハイブリッドなタスク並列が必要になると考えられる。

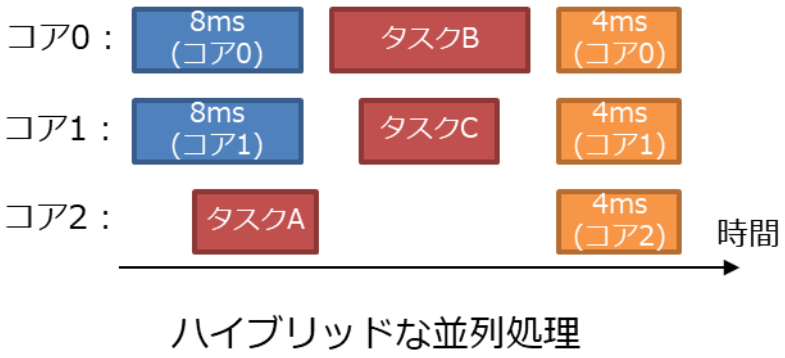


図 1-3 ハイブリッドな並列処理

1.1.2 タスク内で並列化する際の課題

タスク内並列処理では、タスクを記述するソフトウェアの中から並列に実行可能な箇所を抽出し、マルチコアに割り付ける。

1.1.2.1 コア間の同期処理が必要な場合

1. 図 1-4 は、タスク A の中の処理について、関数 func1 と func2、関数 func3 と func4 が同時に実行できる（依存関係がない）ものと判断した例。
2. 関数 check は前後の関数との依存関係があるため、別のコアに分割して割り当てるには、コア間の同期処理が必要になる。

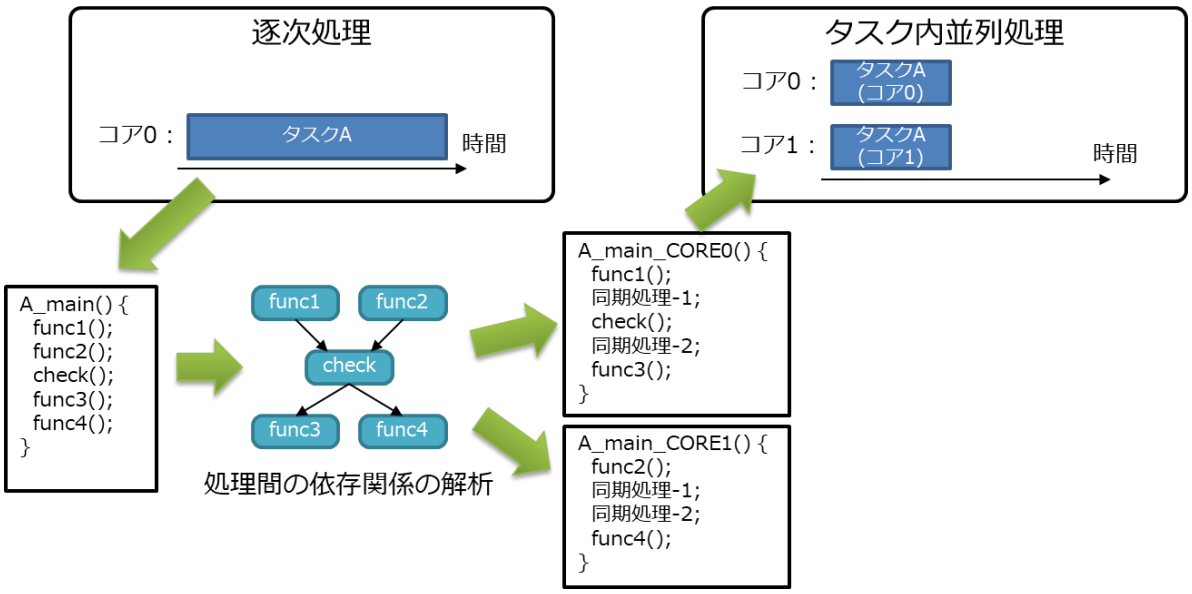


図 1-4　コア間の同期処理が必要な場合

1.1.3　ソフトウェアの並列化から性能確認、再設計までの流れ

ソフトウェアを並列化して性能を向上させることは、解決するべき課題が非常に多く、困難を伴う。

1.1.3.1 並列化の開発プロセス

ソフトウェアを並列化した後、性能を確認し、必要に応じて再設計を行う。並列化の工程では、自動並列化ツールなどを活用して作業効率の向上を図れる。

並列化によりソフトウェアの性能を向上させるには、局所的なソースコードへの対処だけでは不十分。大局的な情報が必要になる。そのため、ソフトウェア開発工程において、実装と単体テストの完了後に並列化を行い、統合テストやシステムテストの工程で性能を確認する。

統合テストの工程で性能上・品質上の問題が判明すると大きな手戻りが発生し、開発スケジュールや工数への影響が甚大になる。量産開発時の手戻りを防ぐためには、試作段階でどこまで精度の高い並列化の方針が立てられるかが非常に重要。

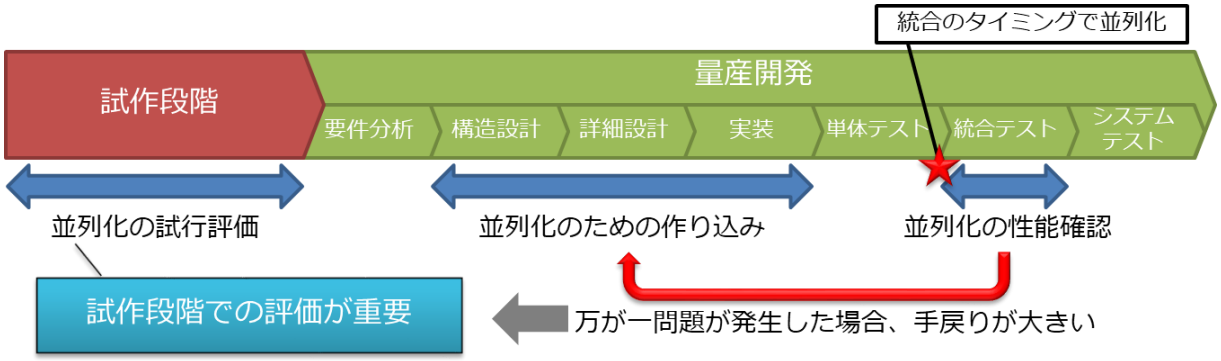


図 1-5 ソフトウェアの並列化から性能確認、再設計までの流れ

1.1.3.2 ソフトウェアの並列化のフロー

ソフトウェアの並列化のフローの概略は、以下のようになる。

「並列化」は、開発工程上の構造設計（アーキテクチャ設計）から詳細設計、実装にまで及ぶ可能性がある。

並列化した後の性能は、1.1.4 以降で示す「静的解析で確認するべき項目」と「動的解析で確認するべき項目」の 2 段階で確認する。一般には、「静的解析による性能確認」に対する再設計は「詳細設計・実装」の工程への手戻りに、「動的解析による性能解析」に対する再設計は「構造設計」、もしくはシステム設計の工程への手戻りになる可能性が高い。

前項で述べたように、量産開発において構造設計や詳細設計の工程に立ち返ることは、極力避けたい。そのため、図 1-6 と同等のフローを試作段階で適用し、並列化の課題を事前に解決しておくことが重要になる。

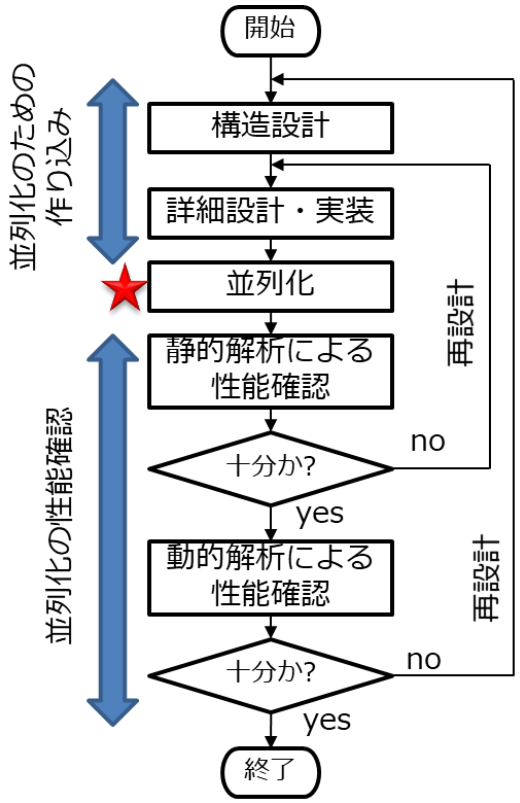


図 1-6　ソフトウェアの並列化のフローの概略

1.1.4　静的解析と動的解析で確認するべき事項

並列性能の向上のために確認するべき事項には、以下の項目がある。

1.1.4.1 静的解析で確認するべき項目

ソースコードや設計情報など、プログラムを実行させる前に分かる情報を用いて解析し、確認するべき情報。

1. ソフトウェアの依存関係：ソフトウェアの処理と処理の間の依存関係（順序関係）（2-1 参照）
2. 依存関係が判別できない要因：実行時にしか決まらない情報、設計情報として開示されていない情報など（2-2 参照）
3. ソフトウェア構造上のボトルネック：並列化を考えるうえで障害となるソフトウェアの構造（2-3 参照）
4. マルチコアへの割り当て：処理をマルチコアに分散して割り当てる上で確認するべき項目（2-4 参照）

1.1.4.2 動的解析で確認するべき項目

評価ボードや量産ハードウェアなどでプログラムを実行させないと確認できない、もしくは確認が困難な情報。

1. 並列処理の開始・終了による性能オーバーヘッド：並列処理に特有の開始・終了時のオーバーヘッド（3-1 参照）
2. 同期による性能オーバーヘッド：マルチコア間の依存関係(順序関係)を維持するために必要な同期処理に伴うオーバーヘッド（3-2 参照）
3. メモリ配置の影響：マルチコア環境でのメモリ配置による並列性能への影響（3-3 参照）
4. リソース競合：マルチコア間でリソースへのアクセスが競合することによる並列性能への影響（3-4 参照）

1.2 静的解析による並列性能の確認とソフトウェアの再設計

次に、並列性能の向上のために確認するべき項目のうち、ソースコードや設計情報など、プログラムを実行させる前に分かる情報を用いた項目について説明する。

1.2.1　ソフトウェアの依存関係

ソフトウェアの依存関係には、制御の依存関係と変数（データ）の依存関係がある。ここでは、変数の依存関係の種類とその対処方法について述べる。

1.2.1.1 変数の依存関係

変数の依存関係には、変数の定義・参照関係の違いから、順依存、逆依存、出力依存の 3 種類がある。

種別	順依存	逆依存	出力依存
説明	定義後に参照がある	参照後に定義がある	定義後に再定義がある
コード例	a = b + c　; 定義 e = a − f　; 参照	e = a + c　; 参照 a = b − f　; 定義	a = e + c　; 定義 a = b − f　; 定義

表 1-1　依存関係の種別とコード例

順依存は、プログラムの自然な流れで発生する依存関係。解消するためには、プログラムのアルゴリズムの変更が必要になる。

逆依存と出力依存は、主に変数の使いまわしからくる依存関係。変数名を変えることにより、依存関係を解消できる。

1.2.1.2 逆依存の解消

新たな変数名を導入することで逆依存が解消する例を示す。

変更前	変更後
func1()とfunc2()の間には並列性があるが、変数gに逆依存があるため、func1()とfunc2()を並列実行できない	変数gの使いまわしを止め、新たな変数g2を導入することで、逆依存が解消され、func1()とfunc2()を並列実行できる

int a, b, h, g;
void func1(){ a = **g** + 1; }
void func2(){ b = **g** + 2; }
void func3(){ h = a + b; }
 	
int main(){
  g = 3;
  func1();
  g = 5;
  func2();
  func3();
}
	int a, b, h, g, **g2**;
	void func1(){ a = g + 1; }
	void func2(){ b = **g2** + 2; }
	void func3(){ h = a + b; }
	
	int main(){
	  g = 3;
	  func1();
	  g2 = 5;
	  func2();
	  func3();
	}

表 1-2　逆依存の解消

新たな変数を導入することで依存関係が解消し、並列実行が可能となるが、実行に必要となるメモリサイズが増えることに注意しなければならない。

1.2.2　依存関係が判別できない要因

プログラム内の変数の依存関係を解析する際、判別できない要因がいくつか考えられる。ここでは、代表的な要因とその対策を挙げる。

判別できない主な要因	対策案
ポインタ変数の指し先が不明である	• ポインタ変数の指し先が分かるようにコードを修正する
ポインタ変数が初期化されていない	• ポインタ変数の初期化処理を追加する
定義の無い外部関数が呼ばれている	• 関数の定義を追加する • 外部関数の呼び出しに対して、入力変数と出力変数の各情報をツールに与える（与え方はツールに依存）
関数ポインタが使われている	• 関数ポインタ記述をif文による条件判定と関数呼び出しの形に変換する

表 1-3　依存関係が判別できない主な要因と対策案

1.2.3　ソフトウェア構造上のボトルネック

論理的に並列性のある処理がソフトウェアの呼び出し構造の深い場所にあると、ソフトウェア構造が性能向上のボトルネックになることがある。

プログラムの論理的な意味を変更せずにプログラム構造を変更することで、並列性を抽出したり、より効率よく並列実行が可能な場合がある。

プログラム構造を変更する手段の一つとして、インライン展開がある。

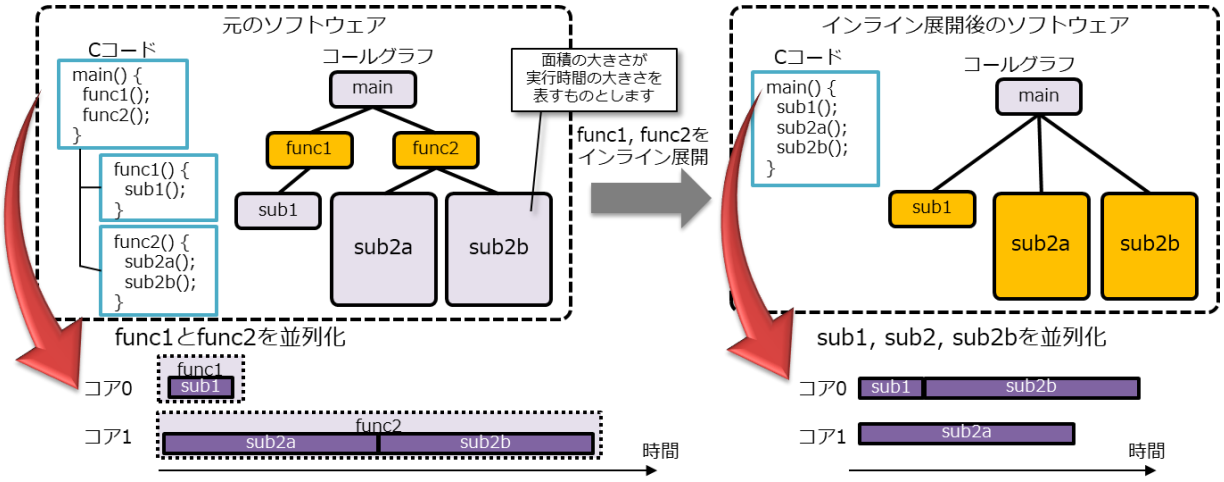


図 1-7　インライン展開

1.2.4　マルチコアへの割り当て

1.2.1~1.2.3 に示した課題を解決しても、マルチコアへの処理の割り当てによっては、並列化効率が向上しない場合がある。処理の割り当てについて、以下のような課題を検討する必要がある。

1.2.4.1 コアごとの処理量の均一化

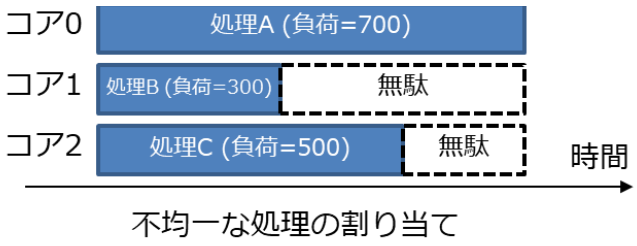


図 1-8　不均一な処理の割り当て

図 1-8 では、各コアに割り当てる処理負荷が均一ではないため、コア 1 とコア 2 の CPU 時間に無駄が生じてしまう。処理 A～C の負荷の総計=1,500 に対し、コア 0 の処理 A の負荷=700 が支配的になるため、並列化効率は 2.14 程度となっている。

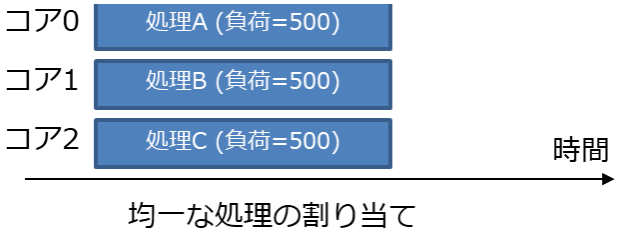


図 1-9　均一な処理の割り当て

図 1-9 のような理想的な割り当てを目指すには、処理負荷を均一にするための施策が必要。まず、性能向上

のボトルネックになっている処理 A を分解することを検討するべきである。分解の手法としては、2-3 で述べたインライン展開などのリストラクチャリングや、ループ処理の分割などが考えられる。

1.2.4.2 割り当ての最適化

図 1-10 左上のような依存関係と処理負荷のある五つの処理を、二つのコアに割り当てることを考える。

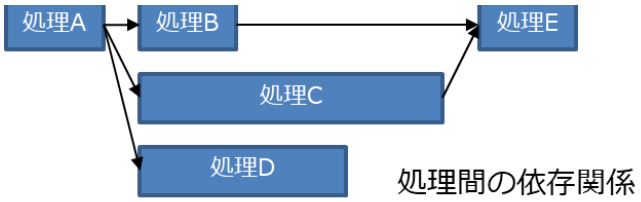


図 1-10　割り当ての最適化

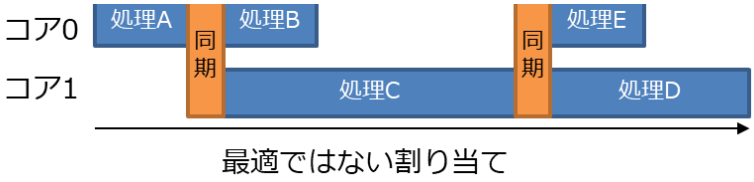


図 1-11　最適ではない割り当て

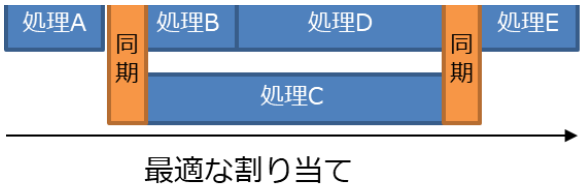


図 1-12　最適な割り当て

図 1-11 では、処理 D をコア 1 に割り当ててしまったため、最適な割り当てである右下の図よりも処理時間が増加している。依存関係の条件を満たす範囲で、図 1-12 のような最適な割り当てを目指す必要がある。

この例では処理が 5 個と少ないため、最適な割り当てを行うのは容易だが、現実の組込みソフトウェアでは非常に多くの処理を対象とする必要がある。

– タスク間並列では、タスクの数の組み合わせを考慮する必要がある。

– タスク内並列では、分解能に従って処理の数はさらに増大する可能性がある。

非常に多くの処理を、限られた開発の時間内で最適に割り当てるのは難しい。自動並列化コンパイラは、最適に近い解を実用的な時間で得られる工夫がなされている。

1.3 動的解析による並列性能の確認とソフトウェアの再設計

最後に、並列性能の向上のために確認するべき項目のうち、評価ボードや量産ハードウェアなどでプログラ

ムを実行させないと確認できない、もしくは確認が困難な項目について説明する。

1.3.1　並列処理の開始・終了による性能オーバーヘッド

タスクを新規に開始するには、通常、下記のような手順が必要であり、オーバーヘッドを伴う。

1. （割り込みによる）タスクマネージャの開始
2. 実行中のタスクの終了処理、または退避処理（プリエンプション）
3. 新規タスクの開始

1.3.1.1 タスクの終了・起動

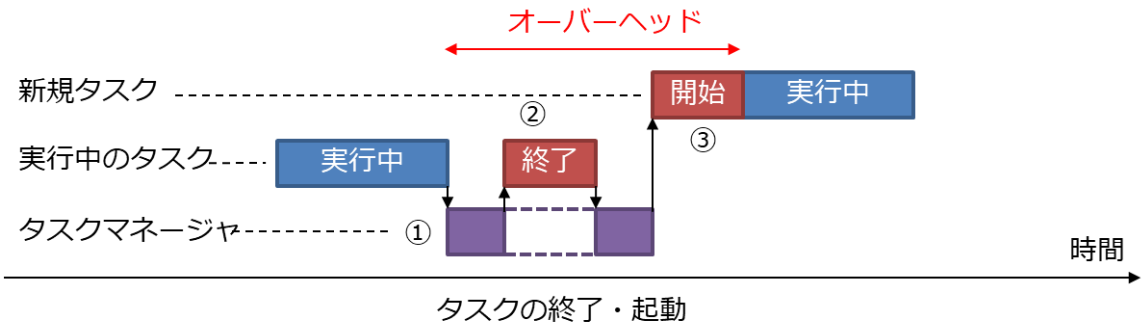


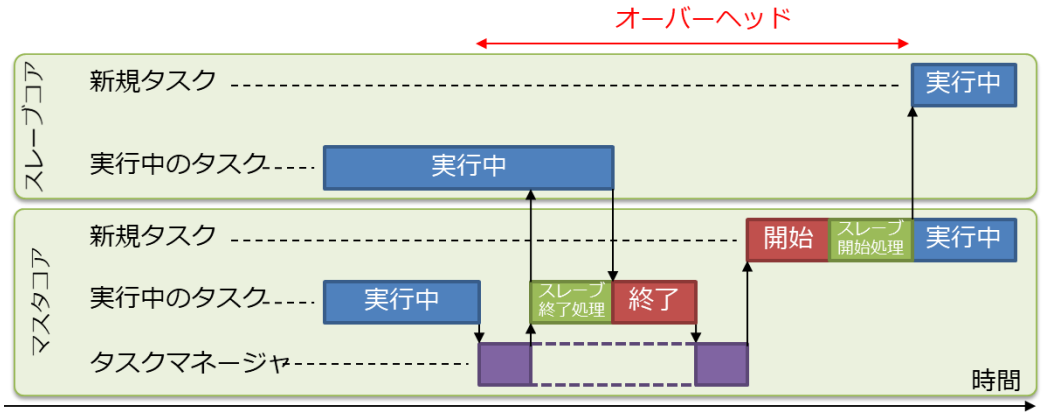
図 1-13　タスクの終了・起動のオーバーヘッド

新規タスクをタスク内並列処理するには、上記のような通常のオーバーヘッド以外に、特有のオーバーヘッドを考慮する必要がある。

1.3.1.2 マルチコアにおけるタスクの終了・起動

並列処理の開始・終了に伴うオーバーヘッドの量は、タスクをどのように管理するか、というシステム設計に強く依存する。

ここでは、マルチコアの構成はマスタ（1 コア）とスレーブ（複数コア）に分かれており、タスクマネージャ、およびタスクの開始・終了の動作はマスタコアが主導するものとする。



マルチコアにおけるタスクの終了・起動

図 1-14　マルチコアにおけるタスクの終了・起動のオーバーヘッド

並列処理の開始・終了のオーバーヘッドを最小にするためには、下記のようなシステムレベルの施策が必要になる。

1.3.1.3 タスクマネージャの分散化

タスクマネージャがマスタコアでのみ動作する集中管理のシステムでは、タスクの遷移のたびにマスタコアの負荷が増加する。さらに、タスクマネージャの処理中にスレーブコアが実質的な処理を行えないため、並列処理の効率が悪化する。

タスクマネージャをマルチコア全体で分散処理し、タスク遷移の際のタスクマネージャの負荷を軽減する必要がある。

1.3.1.4 タスク遷移頻度の削減

タスクの開始・終了は、そもそもタスクを遷移させるために必要な処理である。例えば、タイマにより一定間隔で開始・終了するタスクに対し、周辺デバイスからの割込みを契機に非同期に処理しなければならない高優先のタスクが発生した場合、タスクを遷移することを考えなければならない。

シングルコアのシステムでは、一時にたかだか一つのタスクしか実行できないため、タスクの遷移が必須。しかし、マルチコアのシステムでは、1-1 で述べたタスク間並列処理やハイブリッドな並列処理のように、依存関係のないタスクの並列実行が可能。マルチコアのシステム構成と静的なタスクの割り付けの改善により、タスク遷移の頻度を削減できる。

1.3.2　同期による性能オーバーヘッド

処理と処理の間の依存関係（順序関係）を維持し、かつマルチコア間に処理を分散配置するためには、コア間の同期処理が必須である。

1.3.2.1 コア間の同期処理

下図は処理間の依存関係をグラフで表しているものとする。処理 1 と処理 2 は別のコアで並列に実行できる。同じように、処理 4 と処理 5 も並列に実行できる。

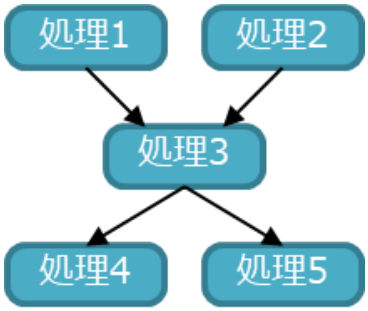


図 1-15　処理間の依存関係の解析

処理 2 と処理 5 を別のコアに割り当てた場合を考える。

処理 1 と処理 2 の実行時間に差があると、処理 2 の終了を待たずに誤って処理 3 を開始してしまう、という状況が起こりえる。

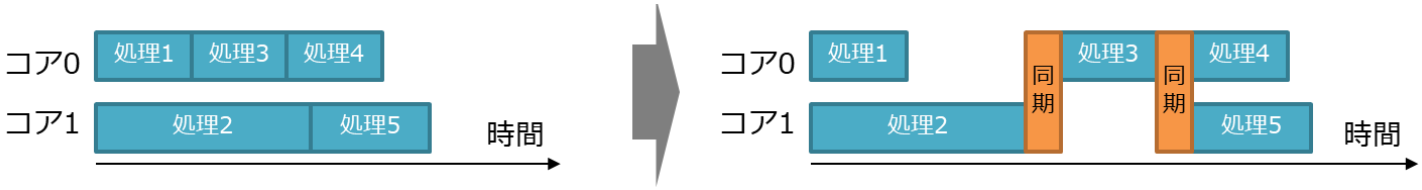


図 1-16　同期処理の挿入

ソフトウェアを正しく動作させるためには、上記のように同期処理を挿入する必要がある。

シングルコアで実行する場合は不要な処理であり、並列化特有のオーバーヘッドとして考慮しなければならない。

1.3.2.2 同期処理の実現手法

同期処理を実現するには、以下のような手法がある。

- 同期フラグの実装方法による分類
  - グローバル変数（コア間の共有変数）をフラグとして利用した同期処理
  - 専用ハードウェアによる同期処理
- API の違いによる分類
  - リアルタイム OS などで規定された API を利用する同期処理
  - ソフトウェア独自の同期処理

1.3.2.3 同期によるオーバーヘッドの縮小

同期による性能オーバーヘッドを縮小するために、以下を考慮する必要がある。

- 同期処理そのものの最適化

- グローバル変数へのアクセス時間や、専用ハードウェアの利用可能性を考慮した同期処理の実装方法を見直す。

■ 同期処理の回数の削減

- ソフトウェアを分割するほど並列に実行できる箇所は増加するが、並列に実行できる処理の粒度は小さくなる。一方、同期処理はほぼ一定の時間を必要とする。さらに、並列に実行できる箇所が増えるほど、同期が必要な箇所も増加する。その結果、ソフトウェアの分割が進むほど、同期によるオーバーヘッドの割合は増大する。
- 並列性の向上と同期のオーバーヘッドの削減はトレードオフの関係。システムごとの同期処理の時間を見極めたうえで、最適なポイントを探る必要がある。

1.3.3 メモリ配置の影響

PC やサーバはもちろん、組込み用の SoC やマイコンにおいてもメモリの階層化やローカル化が進んでいる。マルチコアでは、各コアがアクセスするコードやデータの、メモリへの最適な配置を考えないと、シングルコアの場合よりも性能が悪化する、という事態も起こりえる。

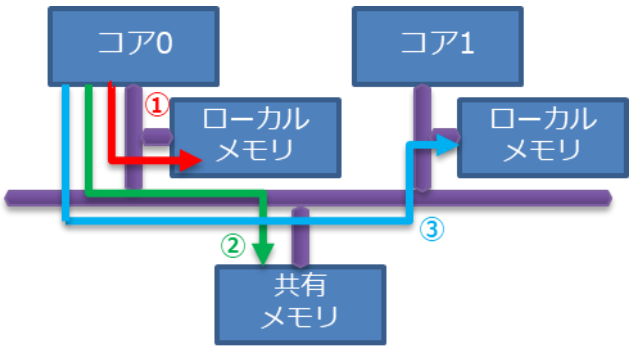


図 1-17 ハードウェア構成の例

図 1-17 のようなハードウェア構成の場合、コア 0 からのメモリアクセスは下記の 3 通り。

① 自コアのローカルメモリへのアクセス

② 共有メモリへのアクセス

③ 他コアのローカルメモリへのアクセス

ローカルメモリを搭載する最大の目的はアクセス速度の向上。コア 0 からの各メモリへのアクセス時間は、①＜②＜③となるのが一般的。

上記のようにメモリの階層化、ローカル化が進んだ SoC やマイコンでは、下記のようなメモリ配置を実現する必要がある。

■ コア 0 からのアクセス頻度が最も大きいデータを、コア 0 のローカルメモリに配置

- コア 1 からのアクセス頻度が最も大きいデータを、コア 1 のローカルメモリに配置
- コア間で共通にアクセスするデータは、共有メモリに配置

最適なメモリ配置のためには、コアごとのデータのアクセス頻度を測定する必要がある。自動並列化コンパイラなどのツールの中には、静的に解析できる範囲でアクセス頻度を算出できるものがある。

1.3.4 リソース競合

マルチコア環境では、各コアで共通のリソースへのアクセスが競合し、並列性能が阻害されることがある。

1.3.4.1 バスの競合

コアから、共有メモリや周辺デバイスへのアクセス経路が単一のバスに限られていると、バスの調整において競合し、性能が阻害される。

マルチレイヤ構成（マルチマスタ構成）可能なバスを採用するなど、システムレベルの対策が必要。

1.3.4.2 共有メモリの競合

1.3.3 では「コア間で共通にアクセスするデータは共有メモリに配置」という指針を示した。しかし、共有メモリへの各コアからのアクセスが頻発すると、メモリアクセスが競合し、性能が阻害される可能性がある。

共有メモリをアドレスドメインごとに分散する、あるいは共有メモリをマルチポート RAM で構成するなど、システムレベルの対策が必要。

1.3.4.3 周辺デバイスの競合

メモリだけでなく、SoC 上の周辺デバイス（ペリフェラル）についても、アクセスが競合する可能性がある。

コアごとに機能を分割することにより、アクセス対象の周辺デバイスを分散させるようなソフトウェア設計が必要。

1.4 まとめ

マルチコアの性能を向上させるには、ソフトウェアを並列化しなければならない。ソフトウェアの並列化にはさまざまな粒度が考えられ、システムに合った構成を選択する必要がある。

並列化したソフトウェアに対して、まず、静的解析による並列性能の確認を実施する。並列性能を満たせない場合は、依存関係の解消やリストラクチャリングなどの再設計を実施する。

次の段階では、実機や評価ボードを用いた動的解析による並列性能の確認を実施する。並列性能の問題が判明した場合、システム設計やソフトウェアの構造設計を見直す必要がある。

並列性能不足による開発の手戻りを最小限にするため、試作段階の性能評価と並列化方針の確定が重要である。

2 ＜動作の見える化＞マルチコアの問題解決に役立つ可視化の技術

CPU 負荷、依存関係、OS 状態からバス利用まで、多岐にわたる情報を活用

■ 本章の対象読者

知識・経験：レベル 1（入門者）　シングルコアの知識・開発経験のみ

プロセス：実装、テスト

ドメイン：組込み全般

キーワード：可視化、性能解析、依存解析、デバッグ、プロファイリング

■ 本章を読んで得られるもの

マルチコアの理想的な実行状態（あるべき姿）が分かる。

実行状態に問題が発生した場合の解決のポイントを把握できる。

問題の解析や解決に役立つ可視化の技術を理解できる。

■ 要旨

マルチコア実行には、アプリケーションの高速化や低消費電力化など、さまざまな利点がある。ただし、その長所を引き出すためには、複数のコアを有効に利用できるように、適切な方法でソフトウェアを作成及び配置する必要がある。

マルチコア実行では、複数のプログラムが同時並列に動作するため、デッドロックなど、さまざまな問題が起こる。さらに、OS などのタスクのコアに対する動的割り当てを利用する場合は、問題が発生した状況の把握や問題の再現性困難となる現象が起こりがちである。

こうしたマルチコア特有の問題を解析したり、解決したりする際には、CPU 負荷やプログラム内部の依存関係などを可視化する手法が有効である。マルチコア向けソフトウェア開発支援ツール製品の中には、さまざまな可視化機能を備えているものが少なくない。

本章では、マルチコア上での理想的な実行状態を阻害するいくつかの課題を取り上げ、それらの課題を解析、解決するための情報の可視化について説明する。

2.1 マルチコアソフトウェア開発になぜ可視化が必要か？

最初に、マルチコアソフトウェア開発における可視化の有用性について述べる。

■ シングルコア実行とマルチコア実行

- シングルコア実行では、動作プロセッサが一つであるため、時間順に表示されたログのような情報でも、ある程度の解析ができる。

- マルチコア実行の場合、複数プロセッサが同時に動作するため、その動作状況を表示する際には、コア方向と時間方向の少なくとも２軸が必要であり、視覚的表示が有効である。

■ マルチコア動作状況と可視化

- マルチコア動作に問題がある場合、可視化により、短時間で問題点を発見できることが多い。
- 以下では、1-1 において「理想的な動作状況」、1-2 において「問題がある状況」について説明する。

2.1.1 マルチコア実行の理想像（性能向上）

次に、マルチコア実行の理想的状況と可視化について述べる。

図 2-1 のように、各プロセッサ（コア 0～コア 2）ごとの実行状況について、時間軸を合わせて可視化することにより、CPU 利用状況や負荷分散状況が分かる。

「ガントチャート」や「タイミングチャート」などの呼び方がある。

図 2-1 の例は、すべてのコアにおいて、常に処理が実行され、同時に終了しているため、理想的な状況と言える。

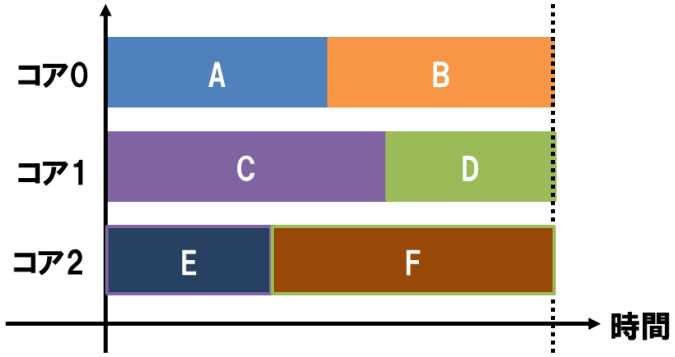


図 2-1 マルチコア実行の可視化

2.1.2 可視化を行う理由

2.1.2.1 並列実行の可視化

並列実行に課題がある例を紹介し、可視化を行う理由について説明する。

図 2-2 は、並列実行があまりうまくいっていない例

■ 各プロセッサに空き時間がある。

■ 同時に終わっていない。

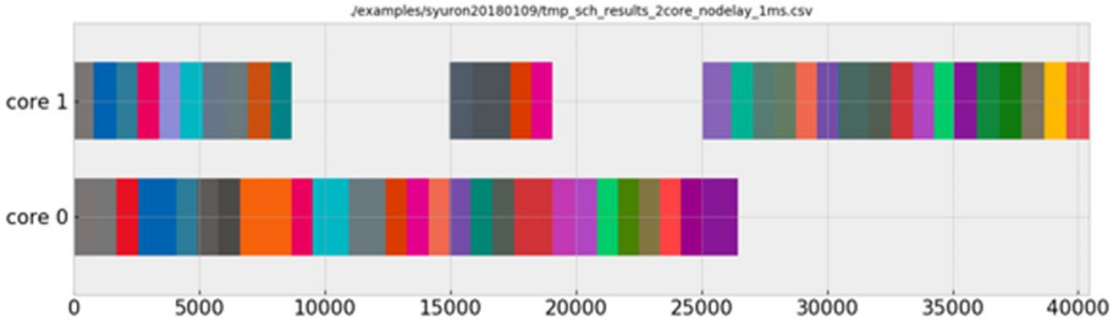


図 2-2 並列実行の可視化

可視化により、ひと目で問題の有無や深刻度、状況が分かる。さらなる原因解析についても、可視化した結果（グラフや図）を参照しながら進めることができる。

2.1.2.2 OS オブジェクト状態の可視化

組込みシステムでは OS（リアルタイム OS）を使うことが多い。リアルタイム OS には OS オブジェクトの状態をログとして記録する機能があるが、ログだけから現象を把握することは容易ではないため，可視化が重要である。

ログでは、各コアのイベントが 1 次元で並んでいるため、コア間の関係が読み取りづらい。

デットロックが発生している例

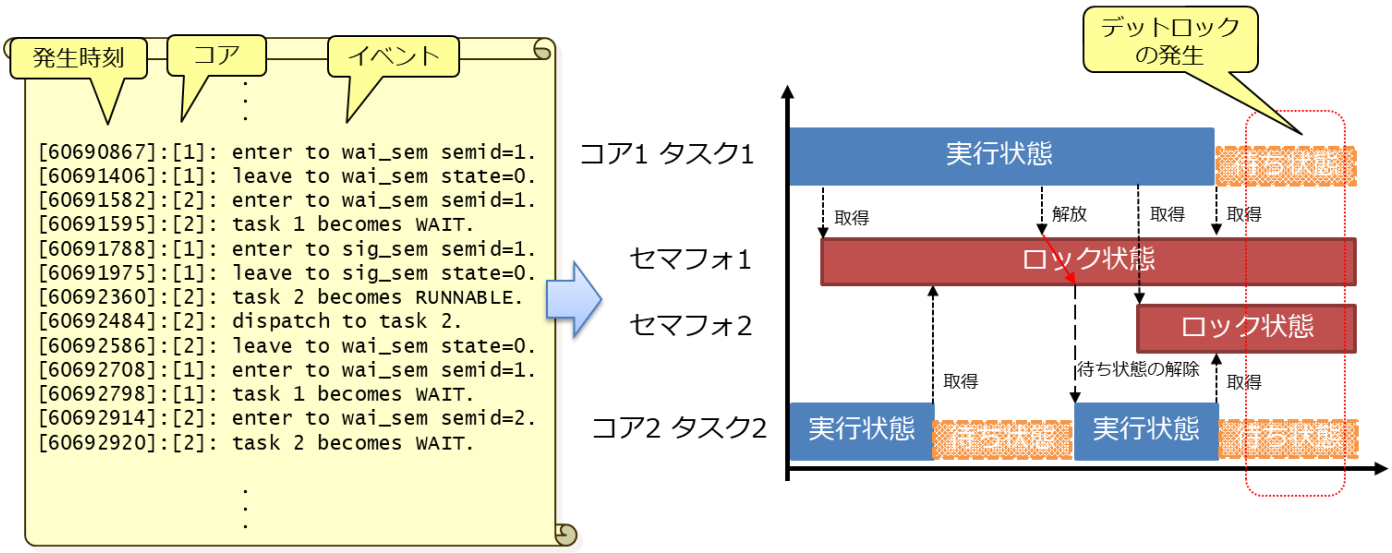


図 2-3　デッドロックが発生している例

2.2 マルチコア実行の課題とさまざまな可視化

次に、マルチコア実行に課題がある場合の原因についていくつか述べ、可視化により原因解析する例について説明する。

2.2.1　負荷分散の可視化

負荷分散に問題があると、性能に影響する。ここでは負荷分散に関するマルチコア実行の課題と可視化について紹介する。

図 2-4 の例では、改善前、コア 0 は常に処理を実行しているにもかかわらず、コア 1 には時間の空きがある。可視化により、負荷分散に問題があることがひと目で分かる。

例えば、処理Dをコア 1 に移すことにより、性能を改善できる。

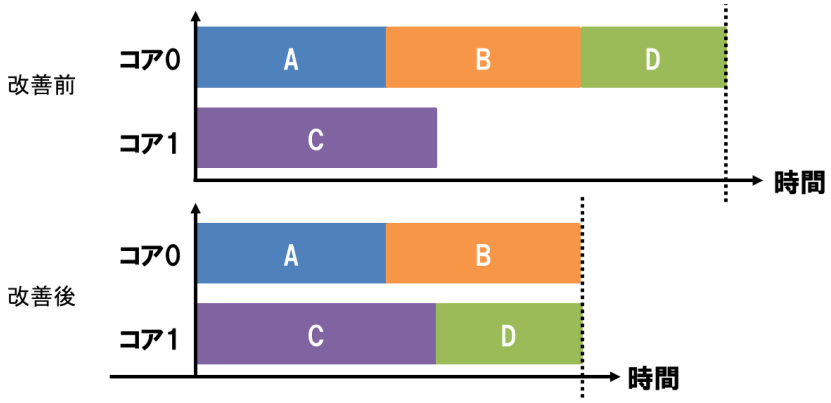


図 2-4　負荷分散の可視化

2.2.1.1 負荷分散の可視化に関連する用語

負荷分散の可視化に関連する用語を説明する。

- 処理単位
  - 前ページでは単に「処理」と記載したが、処理の単位としてはさまざまな可能性がある。典型例は「スレッド」、「タスク」、「プロセス」など。
- CPU 利用率（アイドル時間）
  - 各プロセッサコアで有効な処理を実行している割合を「CPU 利用率」と呼ぶ。アイドル状態になっている時間が「アイドル時間」である。
  - マルチコアの場合、他プロセッサが利用している資源を待ってビジーウェイトしている時間など、実行しているが有効とは言えない処理時間もあるため、注意が必要である。
- アプリケーションの逐次実行部分
  - ある処理を実行している時、同時に実行できる処理が他に存在しない時間帯を「逐次実行部分」と呼ぶ。アムダールの法則をもとに理論並列性能向上率を考える上で、逐次実行部分の割合は重要である。

2.2.2　依存関係の可視化

依存関係は並列実行に制約を与えるため、性能に影響する。ここでは依存関係によるマルチコア実行の課題と可視化について紹介する。

図 2-5 は処理の実行順序制約を示している。

処理 C、Dはともに、処理Aと処理Bの両方が終了していないと実行できない。

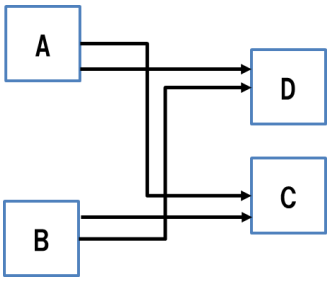


図 2-5　タスクの依存関係の例

実際の実行が図 2-6 のように可視化されたとすると、コア 0 に長い待ちがあり、効率のよいマルチコア実行ができていないことがひと目で分かる。

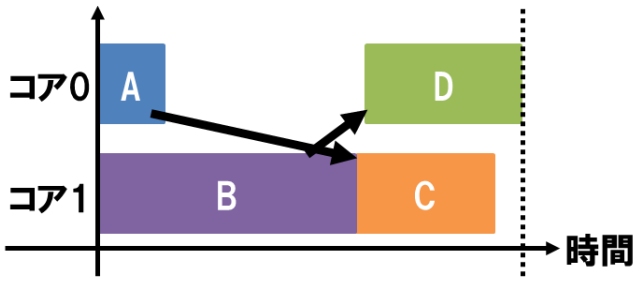


図 2-6　依存関係の可視化の例

このような処理の関係を「依存関係」と呼ぶ。

2.2.2.1 制御依存，データ依存

依存関係には「制御依存」と「データ依存」がある。

- 制御依存：プログラム制御上の順序関係
  - 例： if 処理 A then 処理 B else 処理 C
- データ依存：データ（変数など）の読み書き順序によって規定される順序関係
  - 例：　i = f(x);（処理 A）

    　　j = g(i);（処理 B）

    としたとき、処理 B は処理 A の結果データを使っているのでデータ依存。

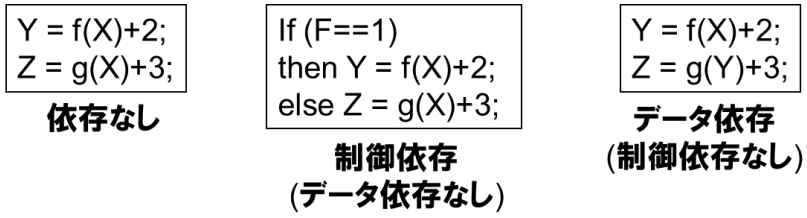


図 2-7　制御依存とデータ依存

2.2.3　メモリ読み書き順序の可視化

同一メモリアドレスに対する読み書き順序の問題により、不具合が起こる場合がある。ここではメモリ読み書き順序にかかわるマルチコア実行の課題と可視化について紹介する。

- 並列化により、処理が同時並行に実行される。同一メモリアドレスに対する読み書き順序が変わると、実行結果が変わることがある。
  - 並列化の際には、データ依存の関係（2.2.2 参照）を見つけ、読み書き順序が変わらないように、処理順序に従って並列化を行う。
- 同一メモリアドレスに対して、異なるプロセッサがほぼ同時に値を更新しようとすると、誤動作を引き起こすことがある。
  - 並列化の際には、排他制御（2.2.5.1 参照）により更新処理を保護する必要がある。

いずれの場合も、その対応により並列性が下がるので、対応箇所の範囲は最小限に抑える必要がある。ここで、ポインタの利用などを見逃すと、不具合が生じる。不具合箇所発見のため、可視化が重要になる。

2.2.3.1 メモリ読み書き順序の可視化の例

逐次実行および並列実行に対し、トレースなどを用いた動的解析によりメモリアクセス履歴を取り、対応する命令アドレス（プログラム上の場所）、メモリアドレス（プログラム内の共有変数名）、アクセス属性（Read/Write、Lock など）、値、時刻などを表示する。

同一メモリアドレスに対し、逐次実行におけるメモリアクセス順と異なる箇所、あるいは同時実行可能性がある箇所の色分け表示などを行って可視化する。

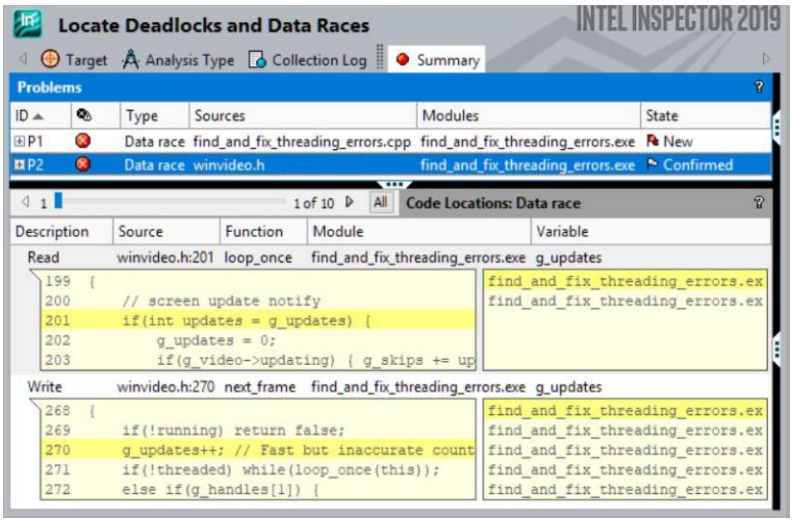


図 2-8　Intel 社 Inspector における可視化の例

図 2-8 は右図は Intel 社 Inspector における可視化の例。

保護されていないデータ競合（同時アクセスの可能性）箇所、および該当共有変数をソースコード上で色分け表示している。

2.2.3.2 RAW、WAW、WAR

メモリ読み書き順序が問題となる場合には、RAW（Read After Write）、WAW（Write After Write）、WAR（Write After Read）の 3 種類がある。Read After Read は順序が変わっても不具合が起こらないため、考慮しない。

図 2-9、図 2-10、図 2-11 は、逐次実行ではスレッド 1→スレッド 2 の順だったものが、同時実行により、メモリアクセス順が入れ替わる例である。すべての Read、Write は同じメモリアドレスへのアクセスとする。

図 2-9 RAW

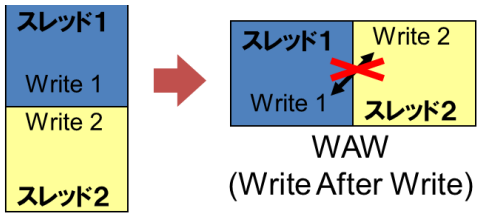


図 2-10　WAW

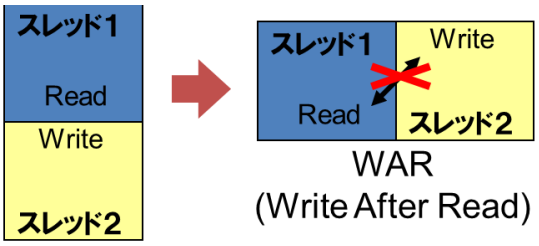


図 2-11　WAR

2.2.3.3 動的解析を使う理由

メモリ読み書き順序を解析するためには、プログラム上のメモリアクセス箇所と、その時のメモリアドレスを特定する必要がある。解析には「静的解析」と「動的解析」があるが、以下の理由で可視化には動的解析が使われることが多い。

大域変数宣言された共有変数などは静的解析で特定可能であり、コンパイラなどで自動的に対応できる。

Ｃ言語のポインタ変数などは、多くの場合、実際に実行させなければ同一アドレスになるかどうかは分からず、動的解析が必要。すべてのポインタ変数への読み書きを保護する、あるいは順序制約をつけると性能が大幅に落ちるため、必要な箇所のみの保護にしたい。

例えば、次の関数

```
void func1(int *a, int *b) { … }
```

において変数 a と b が同一アドレスになるかどうかは、呼び出し側に依存している。開発プロジェクト内の関数利用規則で制約をかけたとしても、人的ミスにより誤った使い方をすると不具合が生じる。このような不具合の発見には、動的解析による可視化が有効である。

2.2.4　OS オブジェクトの状態の可視化

2.2.4.1 タスクとは

可視化では、リアルタイム OS の考慮が必要になる。

- タスク
  - リアルタイム OS 上の処理単位であり、状態を持つ。
- タスクの基本的な状態（図 2-12 参照）

- 実行状態
  - プロセッサで実行されている状態
- 実行可能状態
  - 実行すべき処理があるが、他に優先すべき処理があり、実行が保留されている状態
- 待ち状態
  - 事象の通知を待っている状態
- 休止状態
  - 起動していない状態

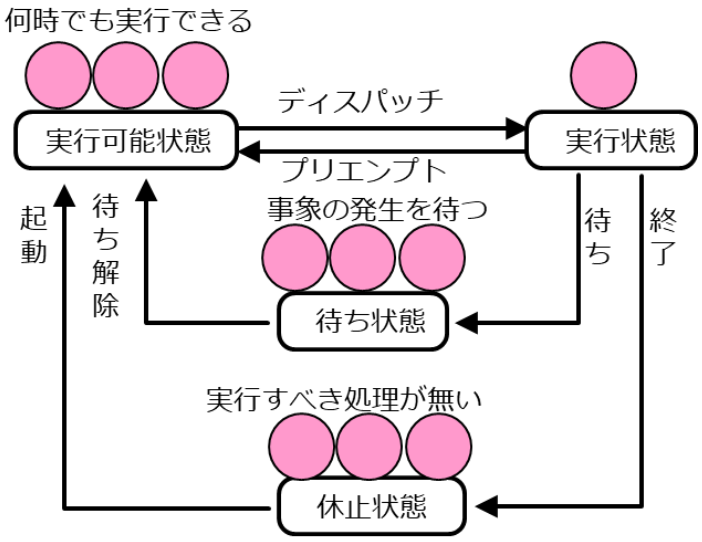


図 2-12　タスクの状態遷移

2.2.4.2 シングルコア実行のタスクスケジューリング

タスク状態を可視化することにより，処理が滞りなく進んでいるかどうかを判断できる。

■ シングルコア実行におけるスケジューリング
- プリエンプティブな優先度ベーススケジューリング。
- 優先度の高いタスクが存在すれば、低いタスクは実行されない。

■ 可視化（下図）において着目する点
- どのタスクによりプリエンプトされ、処理が止まっているか
- 何が原因で待ち状態となり、処理が止まっているか

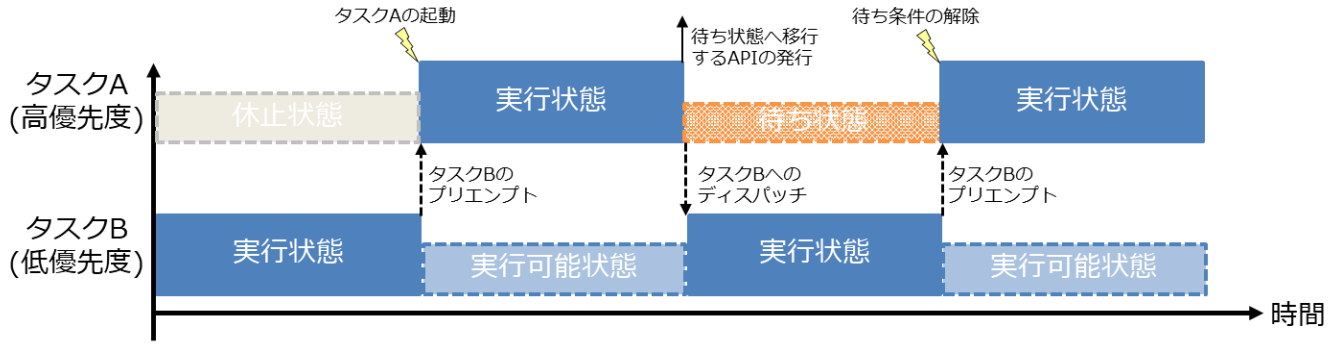


図 2-13　シングルコア実行のタスクスケジューリング

2.2.4.3 マルチコア実行のタスクスケジューリング

マルチコア実行におけるタスクスケジューリングの基本は、以下の二つ。

- AMP スケジューリング（図 2-14）
  - タスクは設計時、（静的に）設計者によって特定のコアに割り付けられる。
  - コア内はプリエンプティブな優先度ベーススケジューリング。

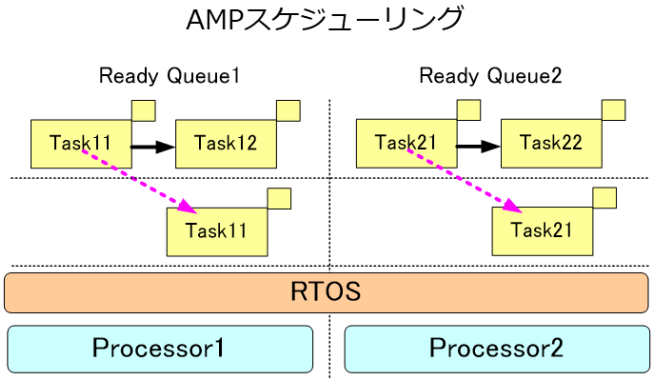


図 2-14　AMP スケジューリング

- SMP スケジューリング（図 2-15）
  - タスクは実行時、（動的に）OS により実行するコアが決定される。

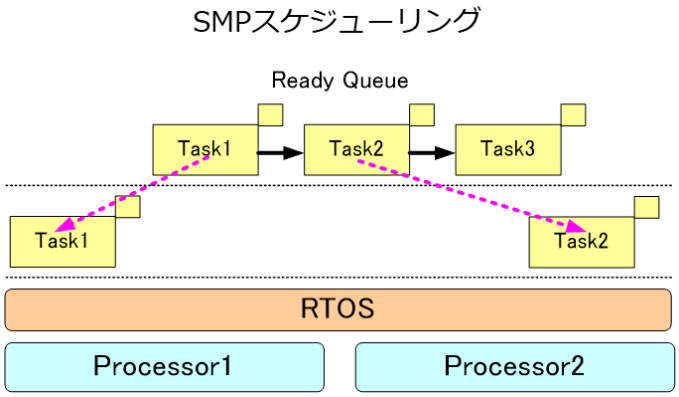


図 2-15　SMP スケジューリング

2.2.5　マルチコアにおけるタスク状態の可視化

マルチコア実行では、コアごとのタスクの状態を可視化する。

- AMP スケジューリングの場合可視化（図 2-16）において着目する点
  - それぞれのコアで独立してスケジューリングされるため，並列実行されるタスク間の実行タイミングが非決定的となり，予想していない実行タイミングが実行されデットロック等の発生条件の見落としが発生する。
  - 実行タイミングが変化する要因の例
    - 割込み発生タイミング・キャッシュ状態

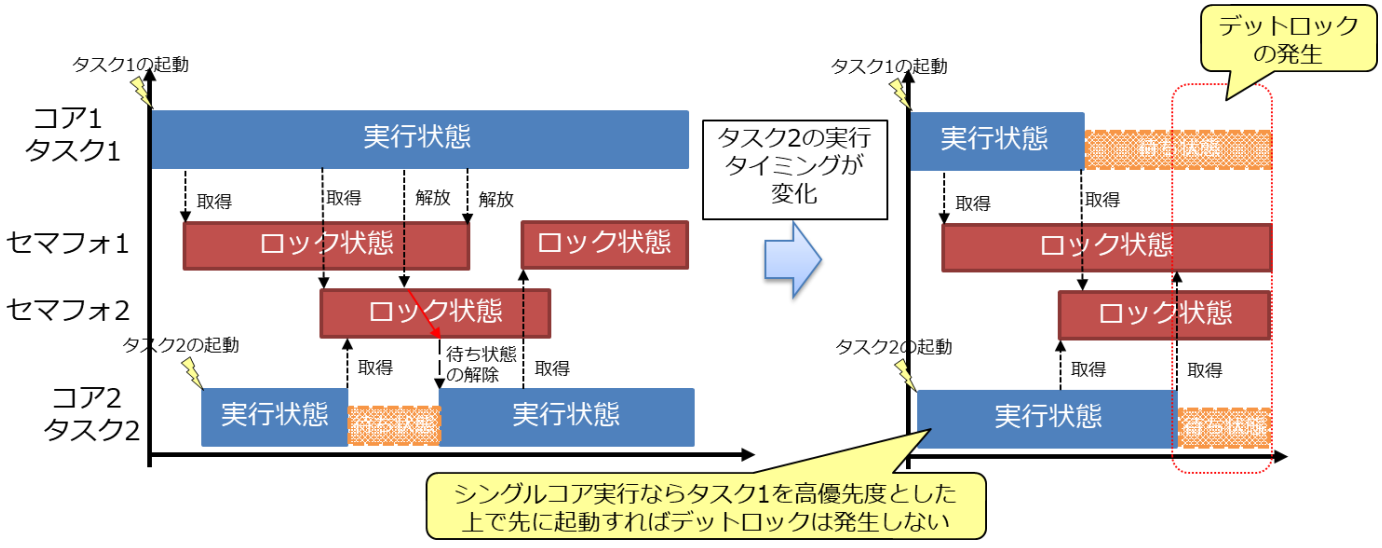


図 2-16　AMP スケジューリングの可視化

- SMP スケジューリングの場合可視化（図 2-17）において着目する点
  - タスクがいつどのコアに割り付けられて実行されたか

- あるタスクの実行タイミングが変化すると、他のタスクの実行されるコアと実行タイミングが変化する可能性がある

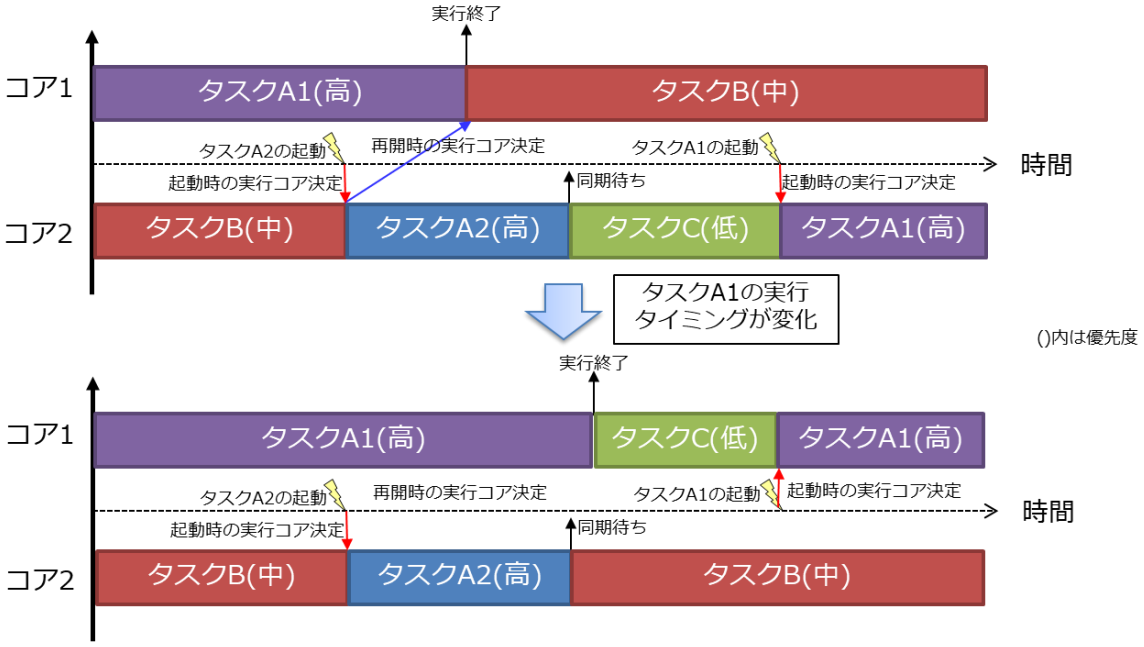


図 2-17　SMP スケジューリングの可視化

2.2.5.1 同期通信オブジェクト

リアルタイム OS には、数種類の同期通信オブジェクトがある。マルチコア実行で処理が理想的に動作している場合、これらのオブジェクトの状態を可視化することにより原因を解析が容易となる。

- FIFO
  - 通信のためのオブジェクト。
  - 内部にバッファを持ち、送受信が可能。バッファがフルの場合に送信すると、待ち状態になる。
  - バッファがエンプティの場合に受信すると、データが来るまで待ち状態となる。
- セマフォ
  - コア間のタスクの排他制御のためのオブジェクト。
  - 共有資源を操作する前に取得し、操作後に返却する。
  - 資源数をカウントするカウンタを持つ。
  - 取得時に資源数が 0 なら待ち状態となり、他のタスクが資源を返却すると待ち状態が解放される。
- スピンロック

- コア間の割込みハンドラ間、および割込みハンドラ-タスク間の排他制御を実現するためのオブジェクト。
- セマフォと異なり，資源数が 0 の場合は実行状態のまま資源の解放を待つ。
- 取得した際には割込みも禁止するのが一般的。
- 取得を試みている間も、割込みを禁止する場合がある（複数のスピンロックを取得する場合）。

2.2.5.2 FIFO

FIFO によるデータ通信で知りたい情報（図 2-18 の可視化例を参照）

■ タスクがどの程度待っているのか、どの程度の頻度で送受信しているのか
- タスクの状態を可視化

■ バッファサイズは十分なのか
- バッファの状態を可視化

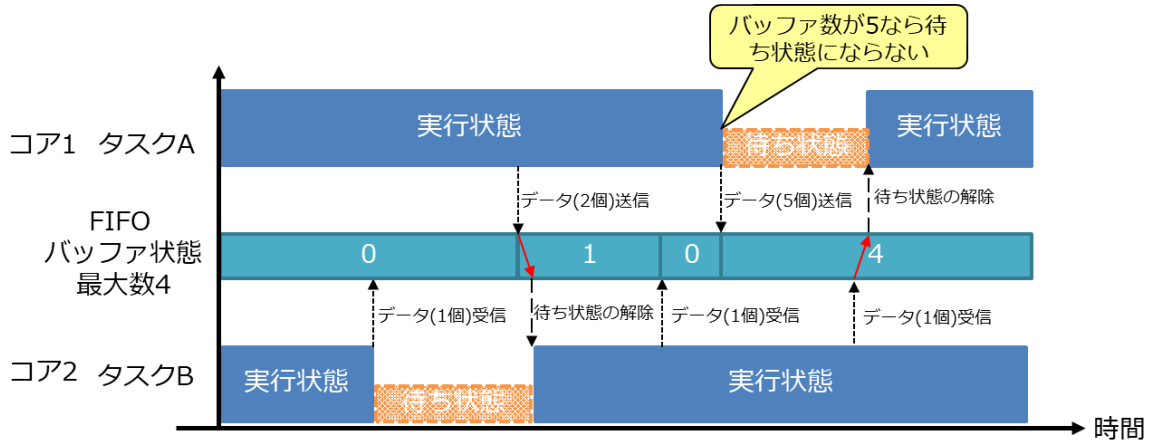


図 2-18　FIFO の可視化

2.2.5.3 セマフォ

セマフォによる排他制御で知りたい情報（図 2-19 の可視化例を参照）

■ タスクがどれだけ待っているのか、誰に待たされているのか
- タスクの状態を可視化

■ 優先度逆転やデッドロックは起きていないか
- タスクとセマフォの状態の可視化

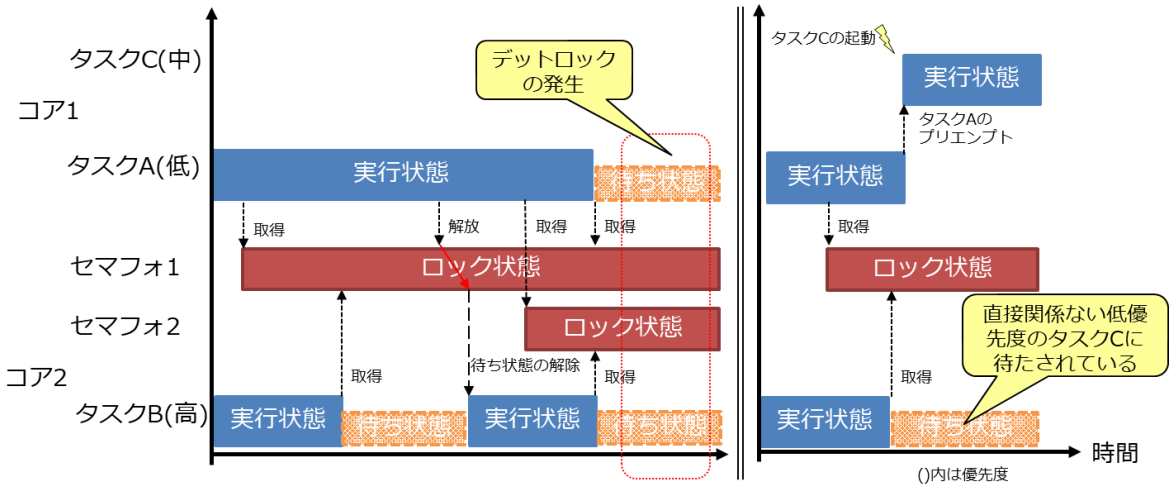


図 2-19　セマフォの可視化

2.2.5.4 スピンロック

スピンロックによる排他制御で知りたい情報（図 2-20 の可視化例を参照）

- タスクがどれだけ待っているのか、誰に待たされているのか
  - ✧ タスクの状態を可視化
- 優先度逆転やデッドロックは起きていないか
  - ➢ タスクとスピンロックの状態の可視化
- スタベーション（飢餓）が発生していないか
  - ➢ 各コアのタスクや ISR（interrupt service routine）の状態を可視化

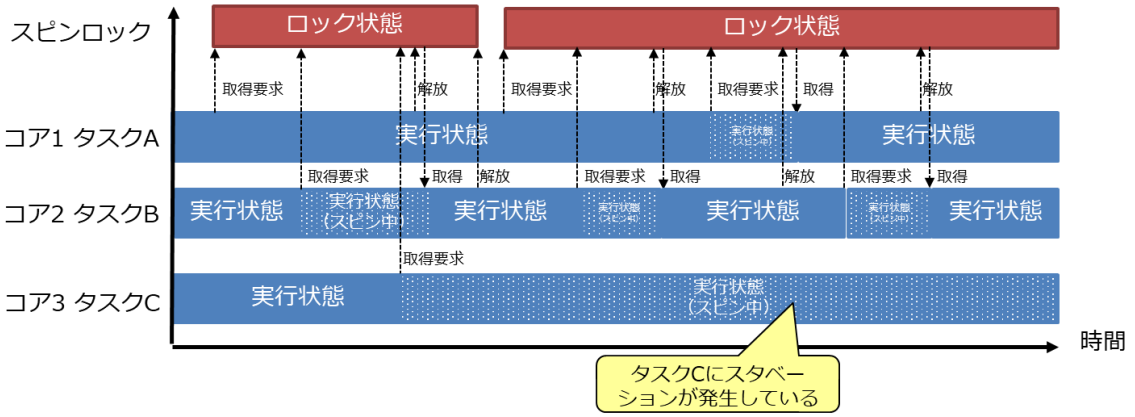


図 2-20　スピンロックの可視化

2.2.6　割り込み応答時間の可視化

2.2.6.1 割り込み応答時間の可視化

割込み応答時間の悪化の原因を特定するために、可視化が有効。

- 割込み応答時間の最悪値に着目
  - 物理的な割込みが入ってから割込みハンドラが起動されるまでの時間
- 割込み応答時間が悪化する要因は割込み禁止にある
  - シングルコアの場合
    - 最悪値はコード中の最長の割込み禁止区間の長さと一致する。

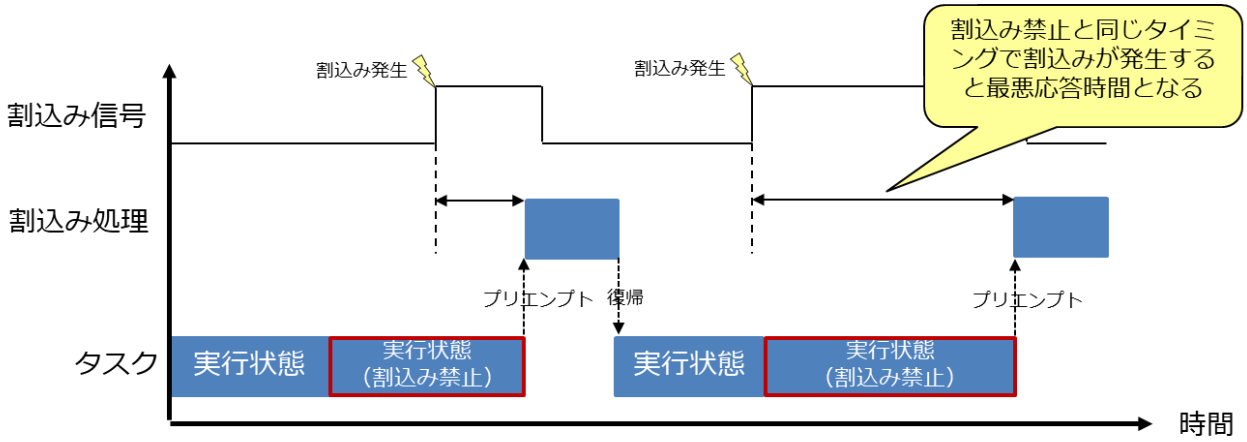


図 2-21　割り込み応答時間の可視化

2.2.6.2 スピンロックの解析

マルチコア実行で発生する状況を解析するために、可視化が有効。

- スピンロックの取得と共に割込みを禁止するため，スピンロックを取得している区間は割込み処理が遅延する。
- 2 段以上のスピンロックの取得の場合、割込みを禁止して取得を試みるため，他のコアのタスクの振る舞いにより、割込み禁止区間が変化する。

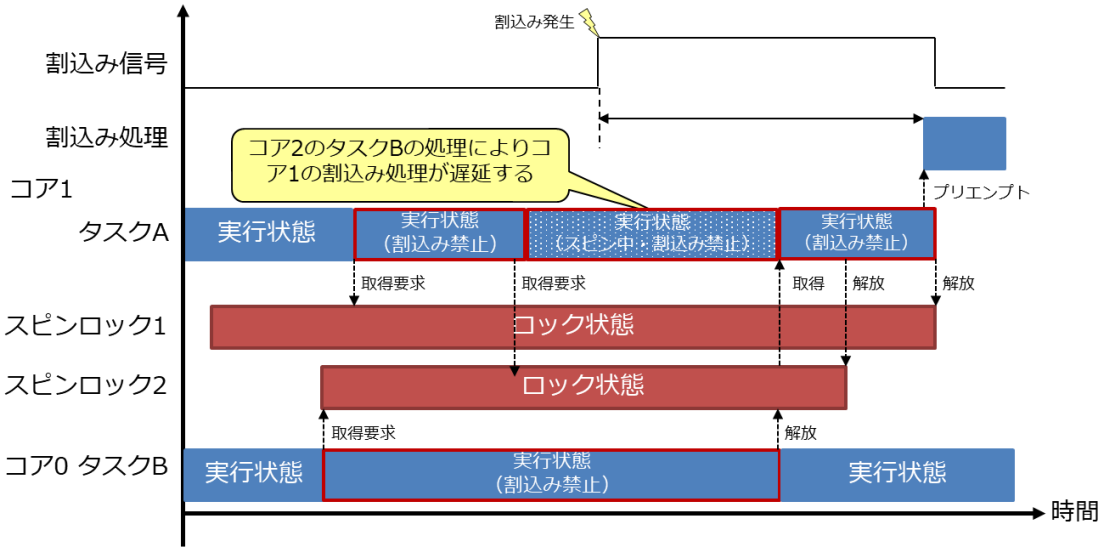


図 2-22　スピンロックの解析

2.2.7　プロセッサ情報の可視化

近年のプロセッサは、高速化・高性能化のためにさまざまな手法を取り入れている。これらの手法がプログラムの性能に影響を及ぼすため、性能解析のためにプロセッサの情報を可視化したい。

- キャッシュ、分岐予測、TLB（translation lookaside buffer）
  - これらの機構はミスが発生する場合がある。ミス時はプログラムの実行性能が低下する。
- プログラムのどの箇所で発生したかを可視化したい（図 2-23 参照）
  - プロセッサ情報はシミュレータでも取得可能。ただし、キャッシュや TLB までシミュレーションすると、シミュレータの実行速度が遅くなる。
  - 実機では、単位時間当たりのミス回数の統計を取得して可視化する。

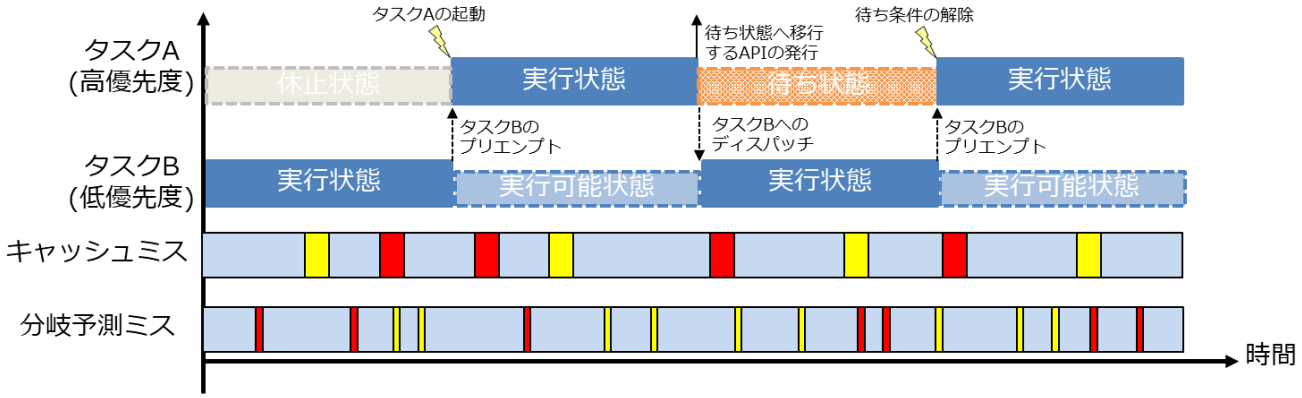


図 2-23　プロセッサ情報の可視化

2.2.8　バスネットワークの利用情報の可視化

ここまで説明した要因について問題がないにもかかわらず並列実行性能が向上しない場合、バスネットワーク競合が考えられる。この問題の解析に対しても、可視化が有効である。

- プログラム中において共有メモリや共有資源の利用を避けることが難しい場合も少なくない。その場合、プロセッサ（クラスタ）ごとのローカルメモリやキャッシュなどを利用し、共有メモリ（共有資源）への同時アクセスを避ける工夫が必要になる。
- 例えば、データ並列（各プロセッサは同一プログラムで異なるデータを処理）利用時にデータを共有メモリに置くと、同一プログラムであるがゆえに、同じタイミングでデータアクセスが起こることがあり、バスやネットワークの競合が起こりがちである。そのような状況を見るために、可視化が有効（右図参照）。

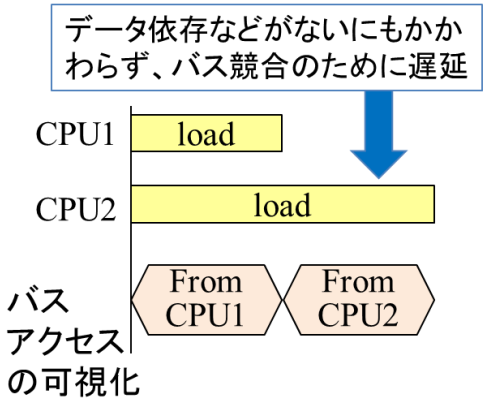


図 2-24　バスネットワークの利用情報の可視化

2.3 まとめ

マルチコア実行において、すんなりと最高並列性能が達成できることは少ない。場合によっては途中でデッドロックするなど、並列要因により実行が止まることすらある。

そのような場合のデバッグや性能解析においては、時間軸のほかに、複数コアや複数タスクの状況を解析しなければならず、少なくとも 2 次元の解析が必要となる。

そのため、可視化を行い、問題発生時刻付近における複数コアや複数タスクの状況を見ることが、問題解析を容易にする。

並列実行を妨げる要因にはさまざまなものがあり、本資料ではそのいくつかの要因について説明し、可視化の例、およびその見方を紹介した。

コア数が多くなると、画面上の情報量が増え、解析が困難になるため、必要最小限、かつ、設計者に分かりやすい表示方法が重要となる。

3 <テスト設計> マルチコア用プログラムを対象としたテストの勘所

バックツーバック手法で並列処理特有の欠陥を効率的に検出

■ 本章の対象読者

知識・経験：レベル 1（入門者）シングルコアの知識・開発経験のみ

プロセス　：実装、テスト

ドメイン　：組込み全般

キーワード：ソフトウェアテスト、並列処理プログラム、競合、相互排除、排他制御、デッドロック
スタベーション、飢餓、ライブロック、バックツーバックテスト

■ 本章を読んで得られるもの

マルチコア用プログラムに特有の欠陥についての理解が深まる。

マルチコア用プログラムを対象としたテスト設計が行えるようになる。

■ 要旨

シングルコア用プログラムをマルチコア用プログラムへ移植する際に、移植元のプログラムでは問題が発生しなかったにもかかわらず、移植先のプログラムで不具合が発生するケースがよくある。

これは、並列に処理するプログラムには、逐次的に処理するプログラムでは考えなくてよかった、特有の欠陥が潜んでいる可能性があるためである。そこでソフトウェアテストの分野では、並列処理のプログラムやシステムに対するテストの難しさの原因、およびテスト手法について、長らく議論されてきた。

本章では、マルチコア用プログラムに潜む欠陥として、並列処理プログラムに特有の「安全性の破壊」、「生存性の破壊」、「公平性の破壊」の三つについて説明する。

こうしたマルチコア用プログラムの欠陥を効率的に検出できる「バックツーバックテスト（back-to-back testing）」を紹介する。

3.1 マルチコア用プログラムとテスト

マルチコアプロセッサを使ったシステムの開発では、マルチコア用プログラムを対象としたテストが課題となっている。

各プロセッサメーカが CPU の動作周波数を、これ以上飛躍的に向上させることは困難。

一つのプロセッサチップに複数の CPU コアを内蔵するマルチコアへ向かっている。

マルチコアやマルチスレッドを用いることにより、ソフトウェア処理の並列性を高め、システム全体の処理性能を引き上げようという考え方。

しかし、ソフトウェアが複数の CPU コアを効率的に利用できなければ、意味がなくなる可能性がある。

並列コンピューティングに対応したプログラミングスキルが必要になる。

マルチコア用プログラム（並列処理プログラム）では、シングルコア用プログラム（逐次処理プログラム）では発生しない、並列処理特有の欠陥が発生してしまう。

複数のプロセスが利用できる共有資源に対して、複数のプロセスからの同時アクセスにより、競合が発生する場合がある。

複数のプロセスの処理が共有資源にアクセスする時は、ロックなどを使った排他制御（相互排除）が必要である。

排他制御とは、あるプロセスに資源を独占的に利用させている間、他のプロセスがその資源を利用できないようにすることによって、データの一貫性（整合性）を保つ処理のことである。

従来の逐次処理の欠陥の知識に加えて、並列処理特有の欠陥の知識を理解することが不可欠である。

ただし、並列プログラミングで発生するバグを発見することは容易ではない。

こうしたバグの多くが、プログラミング作業の終わりの段階で見つかることになる。

3.1.1 マルチコア用プログラムの事例

マルチコア用プログラムの動作確認は難しい。以下に、コンパクトデジカメの事例を紹介する。

二つの CPU コアを搭載している。

- Core 0：キャプチャ処理（イメージセンサからベイヤ配列の RGB データを取り込む）
- Core 1：現像処理（RAW データを JPEG データに変換する）

これら二つのデータ処理を同時並行に実行する。

ユーザが次の被写体を素早く狙えるように、キャプチャ処理と現像処理を別々の CPU コアで実行する。

画像データは共有メモリに格納されており、両方の CPU コアからアクセス可能である。

CPU コア間の通信には、AMP（asynchronous multi-processing）対応のリアルタイム OS の機能を利用する。

しかし，最初の機種の開発では、多くのトラブルを経験した。

- メモリ競合（排他制御）
- キャッシュコヒーレンシにかかわる不具合

シングルコア用プログラム（逐次処理プログラム）では発生しない並列処理特有の欠陥が、マルチコア用プログラム（並列処理プログラム）で発生した。

3.1.2　並列プログラムのテスト

並列処理プログラムを対象としたソフトウェアテストの技術は、長年、研究されてきた。

並列処理のプログラムやシステムに対するテストの難しさの原因、およびテスト手法については、1980 年ころから議論。

当時、「コンピュータシステムの処理性能向上のため、並列分散処理の時代が来る」と言われていた。

実際は、ハードウェアの発展、特に CPU の動作周波数の向上が著しかったため、並列処理が強く望まれる応用領域は限られていた。

並列処理が持つ本質的な難しさやその原因については、このころから指摘されており、現在もそれほど変わっていない。

3.2 マルチコア用プログラム特有の欠陥

マルチコア用プログラム特有の欠陥は、以下の 3 種類に分類できる。

- 安全性の破壊
  - 排他制御の失敗により、データの一貫性（整合性）が失われる欠陥
- 生存性の破壊
  - 同期の失敗により、デッドロックと呼ばれる状態が発生する欠陥
- 公平性の破壊
  - ある処理だけが不当に実行されないライブロック（スタベーション、飢餓）と呼ばれる状態が発生する欠陥

3.2.1　安全性の破壊

安全性の破壊とは、プロセス間で利用される共有資源の排他制御の失敗により、データの一貫性（整合性）が失われる欠陥である。

■　データの一貫性が失われる例

例えば、二つのプロセスで同じ変数の値（仮に x=0 とする）を読み出し、一方のプロセスが 1 加え（x=x+1）、もう一方のプロセスが 2 加えて（x=x+2）から値を書き込んだとする。

この場合、値を同時に読み込んでしまうと、後に書き込んだ値だけが有効になるので、結果が一意に定まらない状況が生じる（x=1 or 2 or 3）。

並列処理が行われているプログラムのテストでは、同期が適切に制御されているかどうかを確認するため、さまざまなタイミングでデータの読み書きを行うテストが要求される。

ホテルの部屋、列車や飛行機の座席のオンライン予約などでも、この問題は起きる。

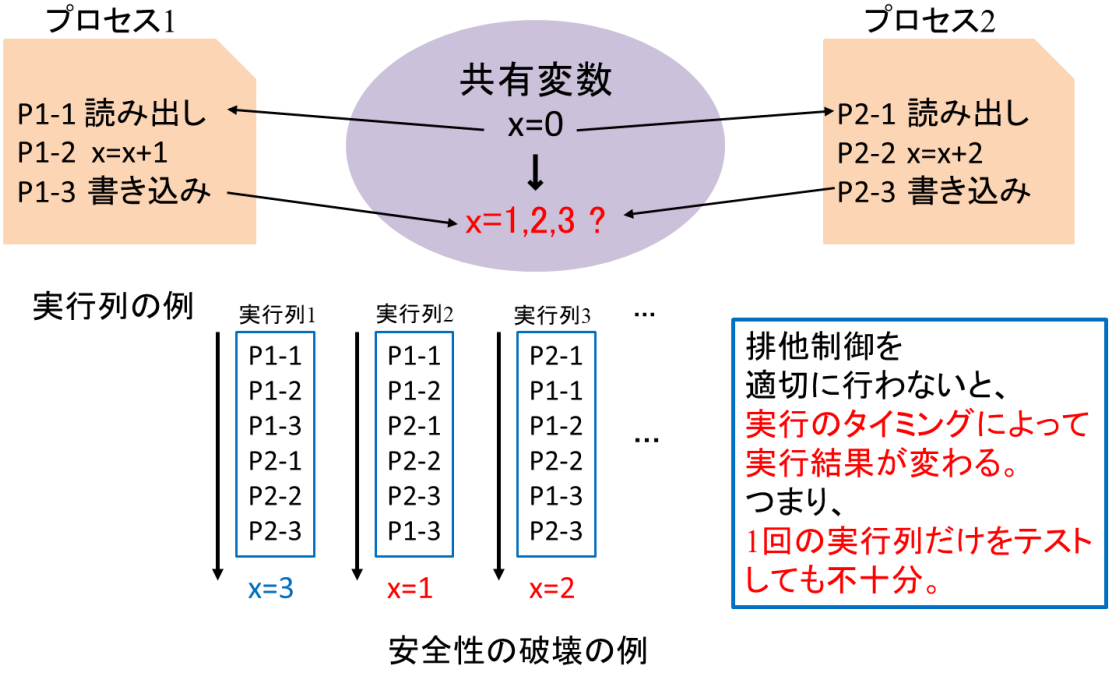


図 3-1　安全性の破壊の例

3.2.2　生存性の破壊

生存性の破壊とは、プロセス間の同期の失敗によって、プロセスが永久に待ち状態となってしまうデッドロックと呼ばれる状態が発生する欠陥。

■　デッドロックの例（Dijkstra が提案した「哲学者の食事問題」）

中央にスパゲティが盛られた円卓のまわりに 5 人の哲学者が座っている。

それぞれの前に皿が 1 枚置かれ、その皿の両側にフォークが 1 本置かれている。

各哲学者から見ると、左右に 1 本ずつフォークが置かれているように見えるが、テーブル全体では 5 本しかフォークはない。

哲学者は 2 本のフォークを使って食事をしなければならないとする。

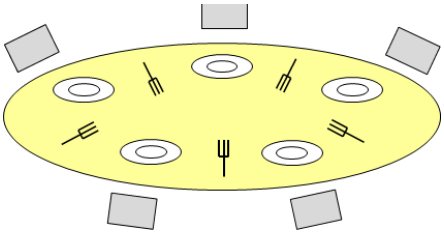
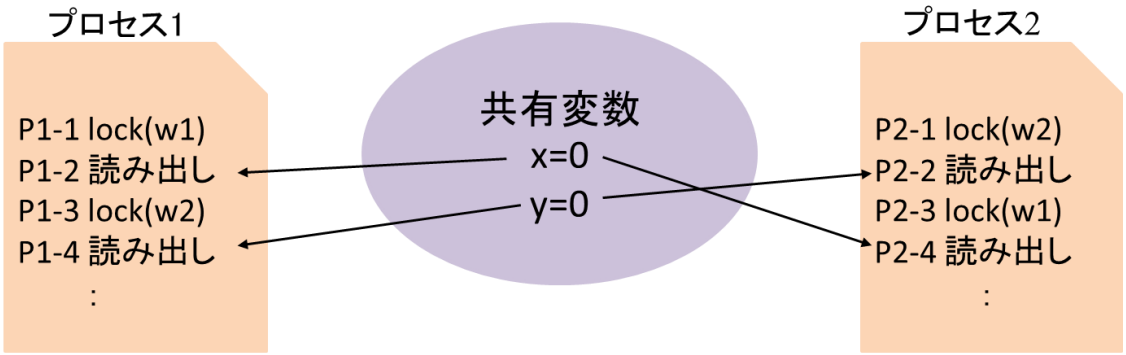


図 3-2　哲学者の食事問題

全員が 1 本ずつフォークを取ってしまうと、誰も永久に食事ができなくなってしまう。(デッドロックの状態)



排他制御を行うため、ロック（あるいはセマフォ）を導入する。
共有変数xに関するロックをw1、共有変数yに関するロックをw2とする。
上記のコードで、例えば p1-1 → p1-2 → p2-1 → p2-2 と実行した場合、
プロセス1とプロセス2は共に、永久に待ち状態となってしまう。

生存性の破壊の例

図 3-3　生存性の破壊の例

3.2.3　公平性の破壊

公平性の破壊とは、ある処理（プロセス）だけが不当に実行されないライブロック（スタベーション、飢餓）と呼ばれる状態が発生する欠陥。

■　ライブロックの例（先ほどの「哲学者の食事問題」で）

一方のフォークを取った状態で、もう一方のフォークを 5 分間待った場合は、いったんフォークを置いて 5 分間待ってから、再度食事を試みる、という規則を設定する。

システムが異なった状態に変化していくので、デッドロックは回避できる。

(ライブロックの状態は回避できない！)

しかし、もし 5 人の哲学者がまったく同時に食卓に着いたとしたら、いっせいに左のフォークを取って 5 分間右のフォークを待ち、左のフォークをいっせいに置いて 5 分間待つ、という状況が発生する。

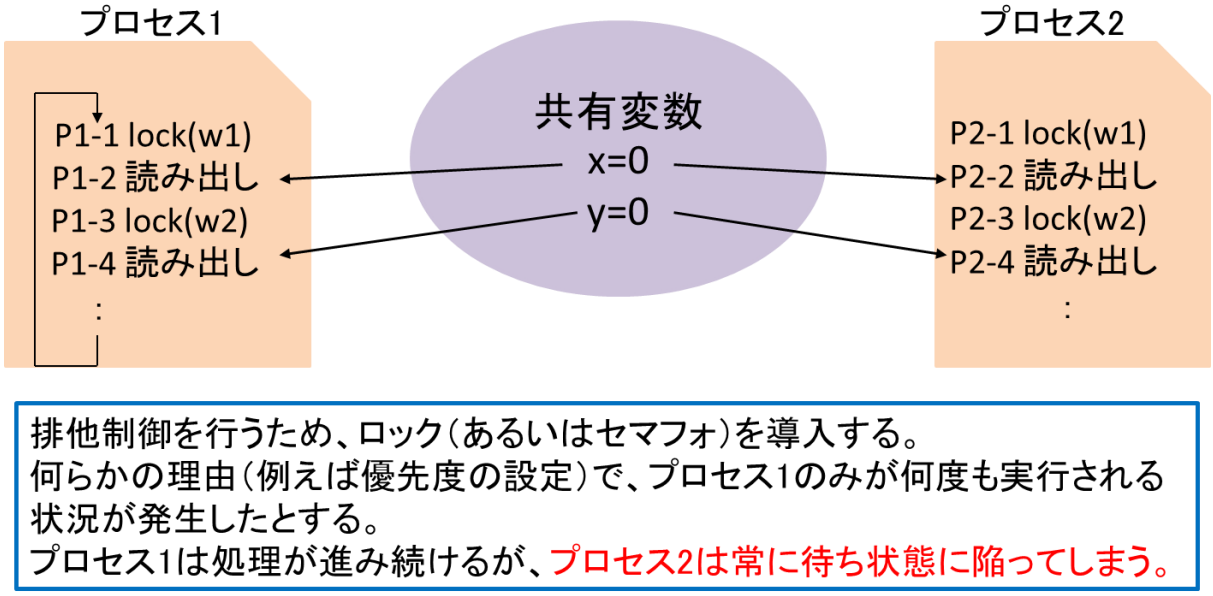


図 3-4　公平性破壊の例

3.3 マルチコア用プログラムを対象としたテスト設計

マルチコア用プログラムを対象としたテスト設計を行う際には、以下のことに気をつける。

■　逐次処理でも発生する欠陥を見つける。

- ➢　のテスト設計技法の活用
- ➢　仕様ベース、構造ベース、経験ベースのテスト

■　並列処理特有の欠陥を見つける。

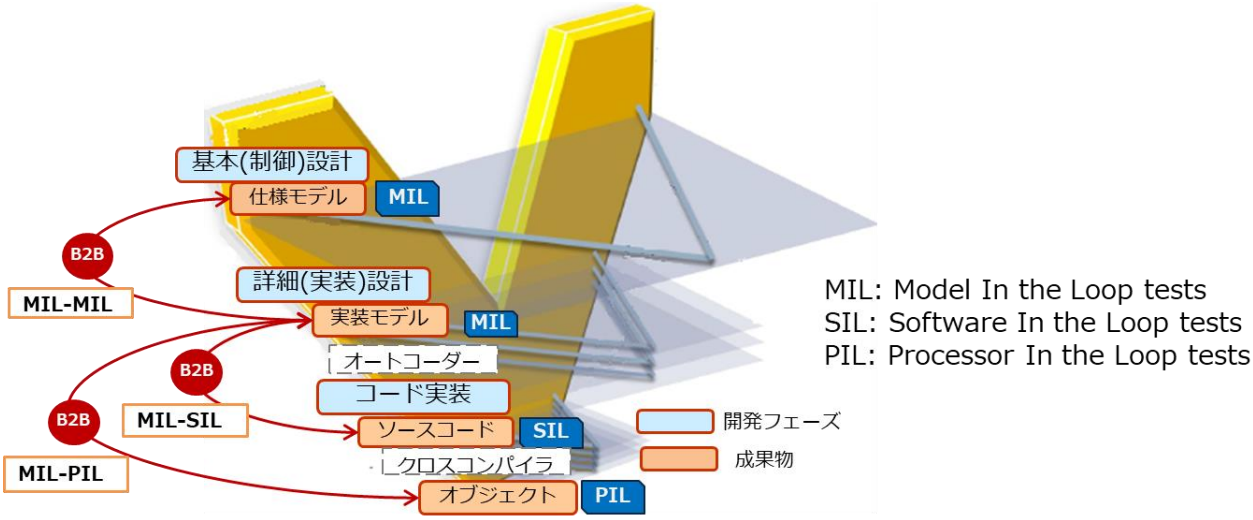
- 安全性の破壊、生存性の破壊、公平性の破壊
- すべての実行列を考慮
- 適切なロックを考慮
- 適切な実行順序（タイミング）を考慮

3.3.1 バックツーバックテスト

3.3.1.1 バックツーバックテスト(B2B テスト)の定義

「二つ以上の異なるコンポーネントまたはシステムに対して同じ入力で実行し、出力を比較する。その結果、相違がある場合はそれを分析する」。(参考文献(5))

適用例として車載機器に対する機能安全規格（ISO 26262）では、モデルベース開発のユニットテストや統合テストでの検証方法として規定されている。



IPA モデルベースシステムズエンジニアリングとSysML -システムズモデリング言語
(https://www.ipa.go.jp/files/000005428.pdf P7) 図を引用

図 3-5　IPA モデルベースシステムズエンジニアリングと SysML

3.3.1.2 バックツーバックテストのマルチコア用プログラムへの適用ポイント

並列処理特有の欠陥の検出には、バックツーバックテスト(B2B テスト)が有効。

あらかじめ実行したシングルコア用プログラムの出力と、マルチコア用プログラムの出力を比較することで、マルチコア化によりプログラムの機能性が損なわれていないことの検証に適用できる。

データの精度や時間応答性に対して、完全一致が求められるのか、一定の閾値範囲内での許容誤差を設けるのか、B2B テスト前に事前検討が必要となる。

CPU クロックに依存した応答性など、ペリフェラル側の挙動にも影響するかどうか検討した上で、B2B テ

ストの環境を構築する必要がある。

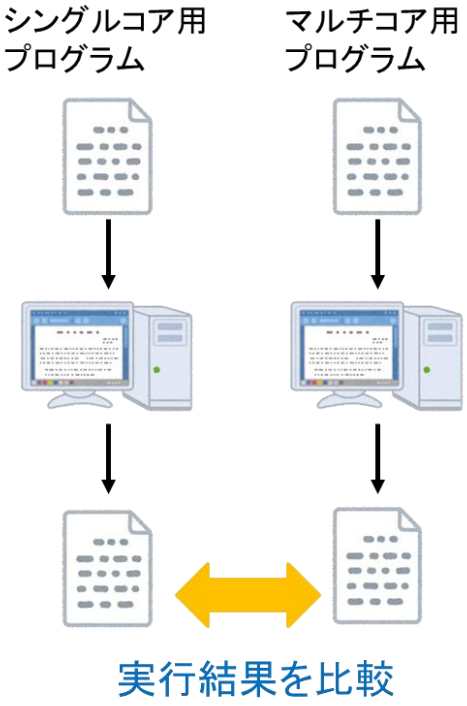


図 3-6　バックツーバックテストのマルチコア用プログラムへの適用ポイント

3.3.2　タイミングを考慮したテスト

マルチコア用プログラムのテストでは、タイミング（適切な実行順序）の考慮が重要。

複数考えられる実行列のうち、たった一つだけをテストして終わってしまってはいけない。

- 実行列の洗い出し
- 実行列の強制実行

さまざまなタイミングでプログラムを実行させることにより、再現しにくい並列処理特有の、タイミング依存の欠陥を検出する。

例えば、Java マルチスレッドのテストフレームワークとして提案されている ConTest である。

3.4 まとめ

各プロセッサメーカは、一つのプロセッサチップに複数の CPU コアを内蔵するマルチコアに向かっており、その性能を引き出すため、マルチコア用プログラムの開発が必要になっている。

高品質なマルチコア用プログラムを開発するためには、マルチコアの特性を考慮したテストが重要になる。

十分なテストを実施するためには、マルチコア特有の欠陥を理解していないといけない。

- 安全性の破壊

- 生存性の破壊
- 公平性の破壊

マルチコア用プログラムを対象としたテスト設計を実施しよう！

従来のシングルコア用プログラムで起こる欠陥も忘れずに…。

- バックツーバックテスト
- タイミングを考慮したテスト

3.5 参考文献

[1] 古川 善吾、伊東 栄典、片山 徹郎；「並行処理プログラムの試験」、情報処理、Vol.39、No.1、pp.7-12 （1998 年）.

[2] M. Ben-Ari；Principles of Concurrent Programming, Prentice Hall, 1982.

[3] R.N. Taylor, D.L. Levine, and C. D. Kelly；“Structural Testing of Concurrent Programs”, IEEE Trans. Softw. Eng., Vol.18, No.3, pp.206-215 (1992).

[4] 片山 徹郎、菰田 敏行、古川 善吾、牛島 和夫；「並行処理プログラムにおけるテストケースの定義と生成ツールの試作」、情報処理学会論文誌、Vol.34、No.11、pp.2223-2232（1993 年）.

[5] Japan Software Testing Qualifications Board (JSTQB)；「JSTQB ソフトウェアテスト標準用語集 日本語版 Version 2.3.J02」, http://jstqb.jp/dl/JSTQB-glossary.V2.3.J02.pdf（2019 年 2 月 12 日アクセス）

[6] Japan Software Testing Qualifications Board (JSTQB)；「ISTQB テスト技術者資格制度 Foundation Level シラバス日本語版 Version 2011.J02」, http://jstqb.jp/dl/JSTQB-SyllabusFoundation_Version2011.J02.pdf（2019 年 2 月 12 日アクセス）

[7] 内海 武之、高木 智彦、八重樫 理人、古川 善吾；「back-to-back テストを実行するテストツールの実装と評価」、情報処理学会 第 73 回全国大会、Vol.1、pp.499-500（2011 年）.

[8] IBM Developer；「ConTest を使用したマルチスレッド・ユニットのテスト」、https://www.ibm.com/developerworks/jp/java/library/j-contest/（2019 年 2 月 12 日アクセス）

4 ＜品質評価＞組込みシステムをマルチコア化したときに確保すべき品質とは

要求品質を軸に、品質特性で捉える評価・検証の全体像

■ 本章の対象読者

知識・経験：レベル 1（入門者）シングルコアの知識・開発経験のみ

プロセス　：設計、実装、テスト

ドメイン　：組込み全般

キーワード：タスク、性能解析、依存解析、デバッグ、プロファイリング

■ 本章を読んで得られるもの

マルチコア向けにソフトウェアを並列化する手順が分かる。

ソフトウェアの並列化に際して確認すべき事項や開発環境の概要が分かる。

■ 要旨

シングルコアのプロセッサで動作する組込みシステムとマルチコアのプロセッサで動作する組込みシステムでは、ソフトウェアに求められる要求品質の細目が異なる。

本章では、組込みシステムがシングルコアからマルチコアへ移行したときに、開発者が押さえておくべき要求品質の概要について解説する。

まず、ソフトウェアの要求品質と品質特性の関係をひも解き、品質を評価・検証へ結びつけていく流れを説明する。その上で、静的テスト（レビューや静的解析ツールの適用）によって評価・検証するアプローチと、動的テストによるアプローチの両方の例を示す。

4.1 マルチコア化したときの要求品質とは

「品質」という言葉が持つ意味から考えてみる。

品質の定義はとらえ方次第

■ JIS
- 品物またはサービスが、使用目的を満たしているかどうかを決定するための評価の対象となる固有の性質・性能の全体　→　標準的な定義

■ クロスビー
- 要求に対する適合　→　一般の工業プロダクト寄り

■ 石川

- 「製品の品質」は欧米の考えである狭義の「質」であり「広義の質」は仕事の質、サービスの質、情報の質、工程の質、部門の質、人の質、システムの質、会社の質など、これらすべてを含めとらえる　→　日本人が持つこだわり

■ ワインバーグ

- 品質は誰かにとっての価値である... ここで価値という言葉は,「人々はその要求が満たされるなら，喜んで対価を支払う（または何かをするか？」ということを意味している　→　欧米の価値観

■ 保田

- 品質は，ユーザーにとっての価値である... 近代的な品質管理における品質の定義：品質は，ユーザーの満足度である　→日本人的発想

4.1.1　組込みシステムに求められる普遍的な品質

4.1.1.1 組込みシステムに求められる品質

組込みシステムの一般的な特性を理解したうえで品質を考えなくてはならない。

NTCR 特性　：Nature,　Time,　Constrain,　Reliability

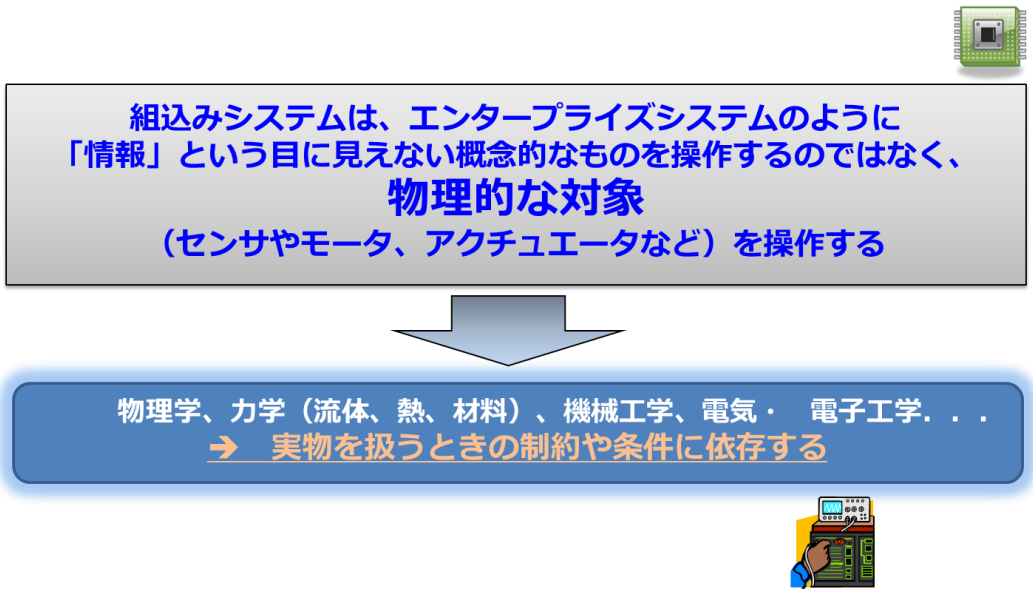


図 4-1　NTCR 特性

4.1.1.2 規格や法規への準拠

求められる品質は、分野ごとのミッションクリティカル度合いで異なる。

安全性やセキュリティ関連の規格や法規への準拠

■ 機能安全系：IEC 61508 をメタ規格としたアンブレラ規格群

- 自動車　　：ISO 26262
- 鉄道　　：IEC 62278 (RAMS)
- 産業機械　　：IEC 62061
- etc.

■ セキュリティ：サイバーセキュリティに対応するための規格群
- ISMS（情報セキュリティマネジメントシステム）：ISO/IEC 27000 シリーズ
- セキュリティ評価基準（CC:コモンクライテリア）：ISO/IEC 15408
- 制御システム ：IEC 62443
- 車載　　：SAE J-3061、TP 15002
- etc.

4.1.2　マルチコア化によって新たに求められる品質

「何のために」マルチコア化するのかという目的から、要求を明らかにしていく。これにより、どのような品質を満足するべきかを検討・整理できるようになる。以下、一般的に考えられるものを五つ例示する。

■ 「機能安全」上の冗長化機構の実現
- 例えばマイコンのロックステップ機構による演算の冗長化など、安全メカニズムとしての要求

■ システム「性能（パフォーマンス）」の向上
- 消費電力量や発熱量の増加なしに性能を向上させたい、という要求

■ 搭載する「ECU（electronic control unit）数の削減」
- ECU 間の高密度な情報連携の実現など、アーキテクチャの側面からの要求

■ 搭載する「機能数の増加」への対応
- 単一 ECU に搭載する機能（処理の組み合わせを含む）の数を可能な限り増やしたい、という要求

■ 「大容量」のグラフィカルデータへの対応
- 大容量かつ複数の画像データを並行に処理し、求められる応答時間の制約を満たしたい、という要求

4.2 要求品質を満たすための品質特性

品質のとらえ方は、作り手の立場か利用者（ユーザ）の立場かによって異なる。双方の立場を理解したうえで、要求品質をとらえ、整理することができる。

利用者の立場で品質のとらえ方（測り方）を整理すると、図 4-2 のようになる。

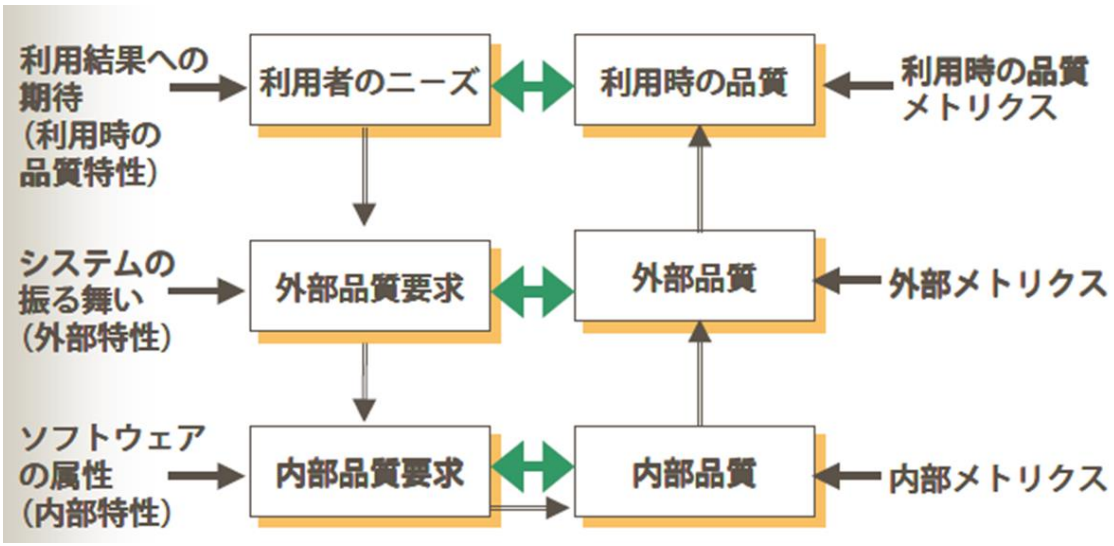


図 4-2　ソフト品質の測定と評価　標準化の状況

早稲田大学 東 基衛 著より引用

ユーザーは外からの見方（外部品質）、作り手は中からの見方（内部品質）をする。品質特性は両方の見方でとらえていく。

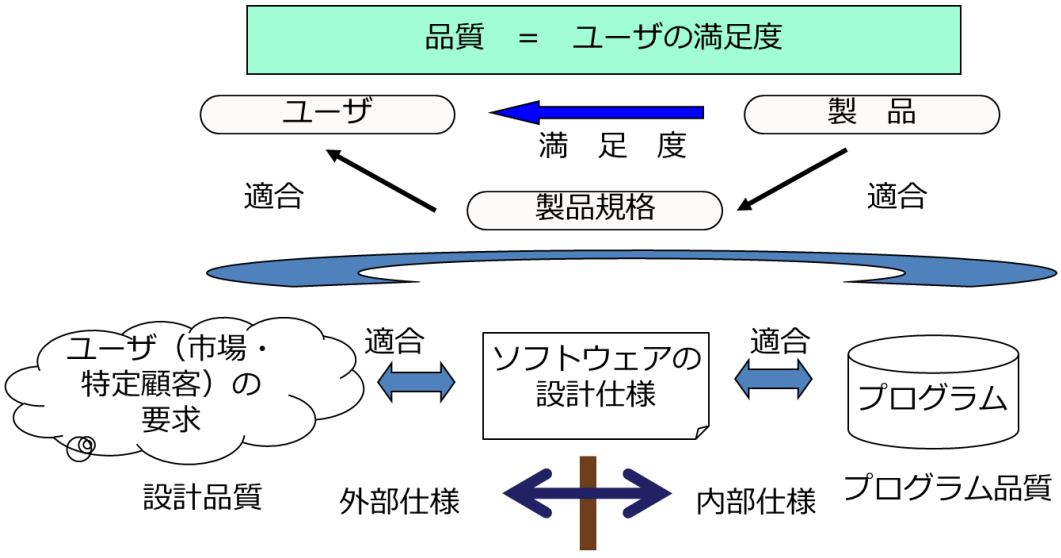


図 4-3　ソフトウェア品質保証の考え方と実際

日科技連出版、保田 勝通 著より引用

システム開発におけるＶ字プロセスで外部品質と内部品質を整理すると、以下のようになる（機能安全のＶ字プロセスで例示）。

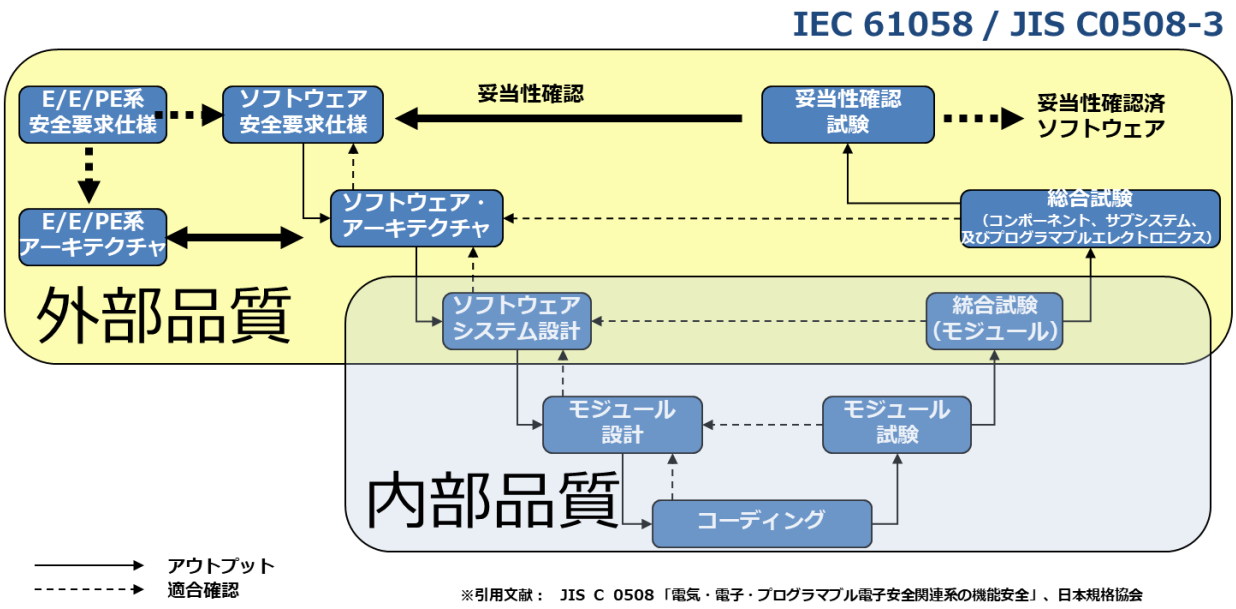


図 4-4　電気・電子・プログラマブル電子安全関連系の機能安全

4.2.1　マルチコア化で押さえるべき外部品質

- 外部品質（利用時の品質）を ISO 25000（JIS X 25010）の品質特性で例示する。
- 有効性：車両性能に対して体感（官能）できるメリット、適用コスト
- 効率性：処理速度の向上、消費電力と熱効率
- 満足性
  - 実用性：メモリ消費の最適化、搭載機能が増やせる。
  - 信用性：処理破たんしない。
  - 快感性：コストダウン
- リスク回避性
  - 経済リスク緩和性：コスト増加抑制
  - 健康・安全リスク緩和性：安全機構の適用（機能安全観点）
  - 環境リスク緩和性：なし
- 利用状況網羅性
  - 利用状況完全性：機能的責務配分の最適化
  - 柔軟性：物理アーキテクチャへの対応

4.2.2　マルチコア化で押さえるべき内部品質

内部品質（作り手の品質）を ISO 25000（JIS X 25010）の品質特性で例示する。

- 機能適合性
  - 機能完全性：組み込むべき機能をすべて搭載できる。
  - 機能正確性：シングルコアと機能的に等価に動作する。
  - 機能適切性：機能間の干渉なく動作する。
- 性能効率性
  - 時間効率性：スケジューリングが仕様どおりに機能し、リアルタイム性を確保する。
  - 資源効率性：オーバーフローやスタック、割り込みの干渉なく、各コアの機能が動作する。タスク動作のリアルタイム性を確保できるメモリ消費量に収まる。
  - 容量満足性：メモリ配置が適切に行われることにより、タスク動作のリアルタイム性を確保できる容量内に収まる。
- 互換性
  - 共存性：他タスクとの共存性を維持する。
  - 相互運用性
    - 他アーキテクチャコアとの安定した連携動作
  - MPU⇔DPU（deep learning processing unit）など、ペリフェラルとの安定した連携動作
- 使用性：マルチコア化で見るべき内部品質から除外できる。
- 信頼性：シングルコアと変わらない品質目標となる。
- セキュリティ：シングルコアと変わらない品質目標となる。
- 保守性
  - モジュール性：コア間の責務配置の保守ができる。
  - 再利用性：マルチコア化で見るべき内部品質から除外できる。
  - 解析性：コア内およびコア間の動作がタスク/メモリの通信レベルで解析できる。
  - 修正性：マルチコア化で見るべき内部品質から除外できる。
  - 試験性：コア内およびコア間の動作がタスク/メモリの通信レベルで試験できる。
- 移植性：コア数の増加に対する論理的構造（アーキテクチャ）上での適切な配慮が必要。

4.3 品質特性を評価・検証する手段

人手またはツールを用いた静的、動的手段の両方を組み合わせて、品質特性を評価・検証していく。

- 肝となること

- コアごとの特性を理解したうえで、最大の性能（パフォーマンス）が得られる機能配置、および設計を実現していることを検証する。
- コア間の不要な干渉を避ける機能配置、および設計を実現していることを評価する。
- ECU の統合時に設計破たんしないように、アーキテクチャとして統制のとれた設計を意識する。

■ 勘所

- 評価・検証するための開発環境や仕組みを開発の早期から構築していく。
- 性能を評価・検証するための現実的な手段を用意する。
- 物理、OS、アプリケーション各層の切り分けが可能な評価・検証環境を用意する。

4.3.1 静的に評価・検証するアプローチ

マルチコアの動作/開発環境において静的に評価・検証を実現する手段として、以下の機能や手法が挙げられる。

4.3.1.1 設計 DR（デザインレビュー）による確認

実機を用いた動作確認以前の開発工程において、マイコンの所定の機能により、メトリクスを満たせることをレビューの場で論理的に検証する。

例えば、以下のような内容を確認する。

■ インターフェース（I/F）仕様にてシンボルと信号の接続を確認する。

- 接続することで、所定の機能を満たすか
- （結果として）不要な機能を搭載していないか

■ インターフェース仕様にてコア間通信のシーケンスを確認する。

- 信号の順序性に問題がないか
- 輻輳（ふくそう）状態に陥らないか
- ライブロックやデッドロックの状態にならないか

■ リソースマッピングの妥当性を確認する。

- 正しい領域に配置できているか

4.3.1.2 静的解析ツールによる確認

コンパイルしたオブジェクトを、ツールで静的解析した結果を用いて分析する。例えば、以下のような内容を確認する。

■ コア間通信の信号変数が競合しないこと

- Read/Write の順序に問題がないか

- 排他が確立できており、参照関係における干渉がないか

■ コア間でリソース配置（シンボル）を照合し問題ないこと
- 未定義/多重定義がなされていないこと

■ コアごとのリソース消費量（オブジェクト実体、マップファイル）が想定内であること
- リソース破たんしていないか。予兆がないか

4.3.2 動的に評価・検証するアプローチ

マルチコアの動作/開発環境において動的に評価・検証を実現する手段として、以下の機能や手法などが挙げられる。

■ 故障診断機能
- マイコンが具備している故障診断機能によって故障条件を検出し、機能仕様に基づく所定の動作によりクエスション（目標を達成したかどうかを評価するための質問）を満たせることを検証する。

■ 故障注入テスト
- 故障診断機能が検出する故障を人為的に発生させる。これを検証の手段として用いる。

■ 計測器による計測
- 計測機器または計測器を用いた実測により、メトリクスを満たせることを検証する。
  - ✧ 計測機器の例：電流・電圧計（テスタ）、オシロスコープ、メモリハイコーダ、プロトコルアナライザ...
  - ✧ 計測器の例：メジャー、質量計り...

■ 内蔵ソフトウェアによる計測
- マイコン開発環境（SDK）として具備する機能を用いて計測することにより、メトリクスを満たせることを検証する。

■ 動作確認
- マルチコア化した場合、メトリクスで定められたマイコンの所定の機能が仕様どおりに動作すること（想定するしきい値内かどうか）を検証・評価する。

■ OS 機能による計測
- OS が具備する機能を用いて計測することにより、メトリクスを満たせることを検証する。

4.4 まとめ

マルチコア化した目的を果たしていることを証明するための品質確認材料をそろえることが肝要。

■ マルチコア化が経済合理性に適っていることを証明できる材料であること

■ トータルコストが見合っていることを示せること

- 性能目標を満たしていることを説明できること
- アーキテクチャ（ホモ/ヘテロ）に適合した冗長化、堅ろう性を持つことを証明できること

シングルコアからデグレード（性能劣化）していないことを検証すること。

- メモリ間の干渉により問題が起きていないか
- コアごとの性能を平準化し、かつリアルタイム性を維持できているか

マルチコア化した製品の品質を確認するため、分析、評価・検証できる環境や仕組み、プロセスを、開発の早い段階から検討し、準備を抜かりなく、かつ責任を持って行うこと。

- 容易に性能評価（処理負荷計測）が行えるようなお膳立て
- コア間のリソース干渉を分析するための手立ての用意など

4.5 参考文献

[1] 保田 勝通；『ソフトウェア品質保証の考え方と実際』、日科技連出版社、1995 年 11 月.

5 ＜自動車応用＞車載システム向けのドメインごとの特徴とマルチコア対応

ADAS/AD 系・制御系をいかに適切に並列化するか

■ 本章の対象読者

知識・経験：レベル２（初級者）

マルチコアの知識あり、車載システムの開発経験少ない

プロセス：要件定義、設計、実装

ドメイン：自動車

キーワード：コデザイン、ヘテロジニアスマルチコア、ホモジニアスマルチコア、リアルタイム制約

■ 本章を読んで得られるもの

車載システムのドメインごとのハードウェアアーキテクチャの違いが分かる。

車載システムのドメインごとのソフトウェア処理特性の違いが分かる。

マルチコアやメニーコアへの実装を前提とした車載システムの要件定義や設計指針の立案が、スムーズに進められるようになる。

■ 要旨

車載システムは、提供する役割により、ADAS（先進運転支援システム）/AD（自動運転）系や制御系など、いくつかのドメインに分類できる。現状の開発では、開発者はドメインの違いをあまり意識しないまま、モデルベース開発を行ったり、ハンドコードされた過去の資産を活用したりしている。

シングルコアの上でシステムを動かす場合は、ドメインの違いを意識しなくても、開発は順調に進んだ。しかし、従来の意識のまま、マルチコアやメニーコアのシステム開発に臨むと、さまざまなトラブルに遭遇する。例えば「期待した性能を引き出せない」、「リアルタイム制約を満たせない」などのトラブルである。このような問題を回避するためには、ドメインごとに異なるハードウェアアーキテクチャやソフトウェアの処理特性の理解が欠かせない。

本章では、まず車載システムのドメインごとの特徴について整理し、その後、各ドメインのシステムをマルチコアやメニーコアへ対応させる際に考慮すべき事項について説明する。

■ 車載システムの略語について

本章をはじめ、車載システムの説明の中でよく登場する略語を以下に記載する。

- AD（autonomous driving）：自動運転
- ACC（adaptive cruise control）：車間距離制御
- ADAS（advanced driver assistance systems）：先進運転支援システム

- AEB（advanced emergency braking）：衝突被害軽減制動制御
- EV（electric vehicle）：電気自動車
- FCW（forward collision warning）：前方衝突警告
- LDW（lane departure warning）：車線逸脱警報
- LKA（lane keeping assist）：車線逸脱防止支援
- IVI（in-vehicle infotainment）：車載情報通信システム
- TSR（traffic sign recognition）：交通標識認識

5.1 ドメインによるハードウェアとソフトウェアの基本構造の違い

車載システムは、その特徴によりいくつかのドメインに分類できる。ドメインには最適なソフトウェアとハードウェアの組み合わせが存在する。ここではソフトウェアを論理層、ハードウェアを物理層ととらえ、それぞれの基本構造を述べていく。

- ソフトウェア（論理層）
  - ADAS/AD
  - エンジン/EV モータ制御
  - その他（コネクテッド/IVI など）
- ハードウェア（物理層）
  - ヘテロジニアス（マルチコア）SoC
  - ホモジニアス（マルチコア）SoC
  - ヘテロジニアス SoC with セキュアゾーン
  - マルチコア/メニーコア MCU（マイクロコントローラ、汎用品）

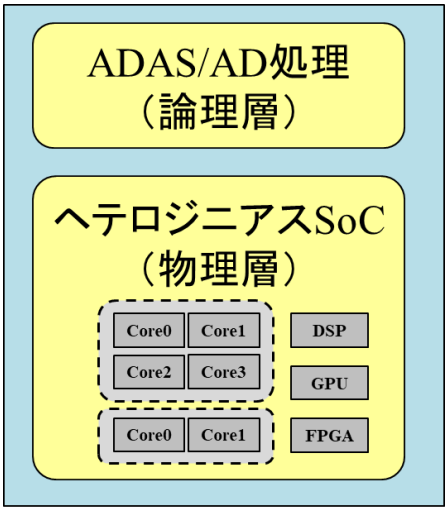
基本構造

図 5-1　基本構造

5.1.1　ADAS/AD 系の基本構造

ADAS/AD 処理には、センシング処理や識別処理、プランニング処理、アクチュエータ処理などがある。これらを同時に処理するため、タイプの異なるプロセッサコアを搭載したヘテロジニアス SoC を使用する。

- 論理層の特徴
  - センサからの識別処理
  - 車両制御
  - 高負荷制御
- 物理層の特徴
  - 画像処理
  - 電波処理
  - ディジタル信号処理
  - GPU 処理



ADAS/AD系ドメインの基本構造例

図 5-2　ADAS/AD 系ドメインの基本構造例

5.1.2　制御系（エンジン、EV モータ）の基本構造

制御系処理（エンジン、EV モータ）には、リアルタイム処理が必須である。ハードウェア処理とソフトウェア処理を組み合わせてリアルタイム制約をクリアする。リアルタイム処理を行うこと、および安全機構を組み込むことを目的とするため、高性能な SoC ではなく、ロックステップ機構などを持つ汎用品のマルチコア/メニーコア MCU を使用する。

- 論理層の特徴
  - 回転制御
  - 内燃機関制御
  - リアルタイム制約
- 物理層の特徴
  - ロックステップ機構
  - 複数のセンサ入力
  - 耐熱

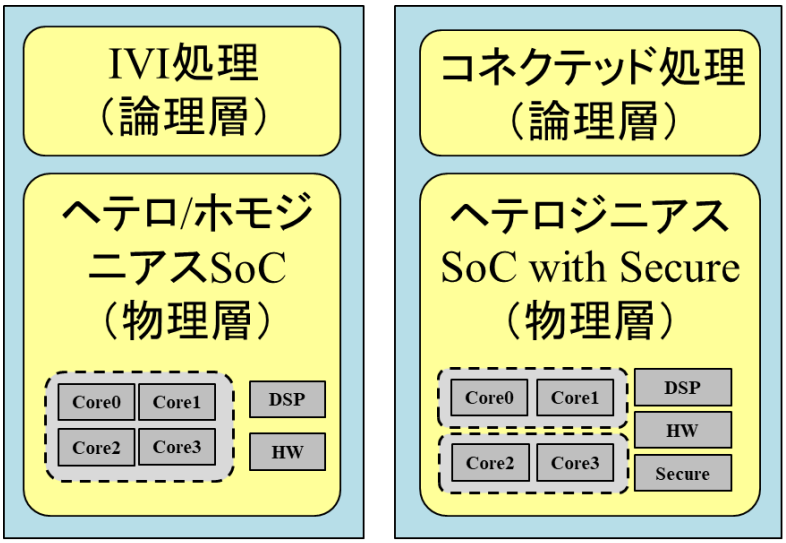
制御系ドメインの基本構造例

図 5-3　制御系ドメインの基本構造例

5.1.3　その他のシステムの基本構造

このほかに、IVI やコネクテッドといったドメインがある。特徴としては、車載（車内）の情報通信処理、および車外との情報通信インターフェースを持つ。そのため、さまざまな情報を取りまとめるための各種プロセッサコアを搭載した SoC を使用する。

- 論理層の特徴
  - 通信ネットワーク処理
  - マルチメディア処理
  - セキュリティ
- 物理層の特徴
  - ディジタル信号処理
  - セキュアチップ搭載



その他のシステムの基本構造例

図 5-4　その他のシステムの基本構造例

5.2 ドメインによるソフトウェア処理の違い

車載システムは、ドメインごとにソフトウェア処理に求められる要件が異なり、それぞれの特徴に合ったソフトウェアアーキテクチャが存在する。

- ADAS/AD 系
  - 高負荷処理の制御とリアルタイム制御を共存させなければならない。
- 制御系（エンジン、EV モータ）
  - 厳格なリアルタイム制御が必須となっている。

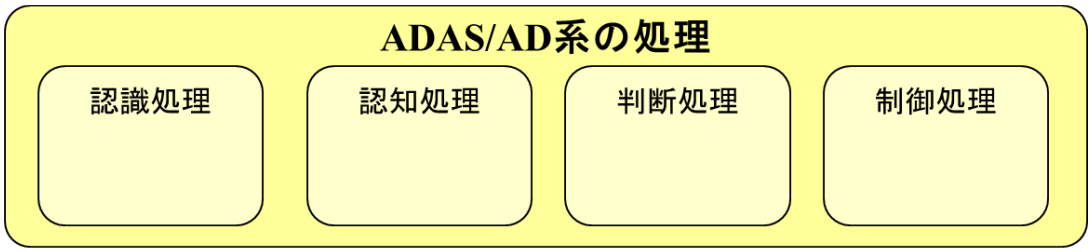
5.2.1　ADAS/AD 系のソフトウェア処理

5.2.1.1 ADAS/AD 系のソフトウェア処理

ADAS 処理では、人が運転してそれをソフトウェアがアシストする。AD 処理では、ソフトウェアが運転してそれを人がアシストする。運転主体は異なるが、ADAS 処理、AD 処理ともにソフトウェアは、認識処理、認知処理、判断処理、制御処理から構成される。

ADAS/AD 系の認識処理と認知処理には、高い処理性能（高負荷処理）が要求される。

ADAS/AD 系の制御処理には、「アクチュエータに指示を出す」などのリアルタイム処理が要求される。



ADAS/AD系のソフトウェア処理

図 5-5　ADAS/AD 系のソフトウェア処理

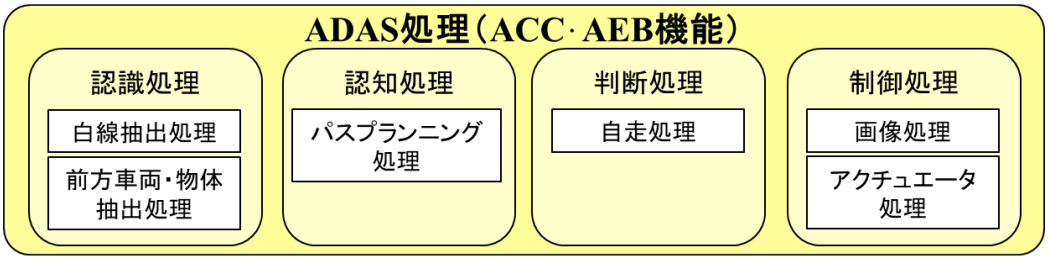
5.2.1.2 ADAS のソフトウェア処理(ACC・AEB 機能の例)

ADAS が人の運転をアシストする機能は、非常に多岐にわたる。ここでは、カメラを使用した ACC･AEB 機能を例に説明する。

ACC 機能の基本動作は、道路の白線を検出しながら、前方の車両を検出して追走する。前方の車両が検出されない場合は、自走で速度を一定に保ちながら車の挙動を制御する。

AEB 機能の基本動作は、車両前方に突然飛び出してきた物体に対し、衝突による衝撃を軽減するため、ブレーキを制御する。

これらのソフトウェア処理は、白線抽出処理、前方車両抽出処理、（白線に沿った）パスプランニング処理、自走処理、画像処理、アクチュエータ処理から構成される。



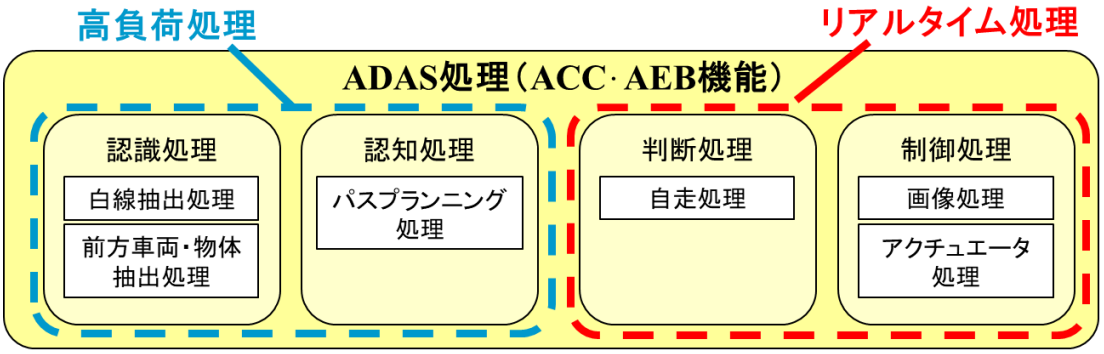
ACC･AEB機能のソフトウェア構成例

図 5-6　ACC・AEB 機能のソフトウェア構成例

5.2.1.3 ADAS のソフトウェア処理に求められる要件

白線抽出処理、前方車両抽出処理、パスプランニング処理は、前回の処理結果に今回の処理結果を重畳する場面が多い。場合によっては識別する物体が極端に増え、突然、高負荷の状態になることもある。このため、リアルタイム処理よりもむしろ、高負荷時に破たんしないように、いかにして適切に処理（タスク）をさばいていくかが重要になる。

自走処理やアクチュエータ処理は、車の挙動を決定し、目的のタイミングで必要な指示を出さなければならないので、リアルタイム処理が重要になる。



ADASのソフトウェア処理に求められる要件

図 5-7　ADAS のソフトウェア処理に求められる要件

5.2.1.4 AD のソフトウェア処理

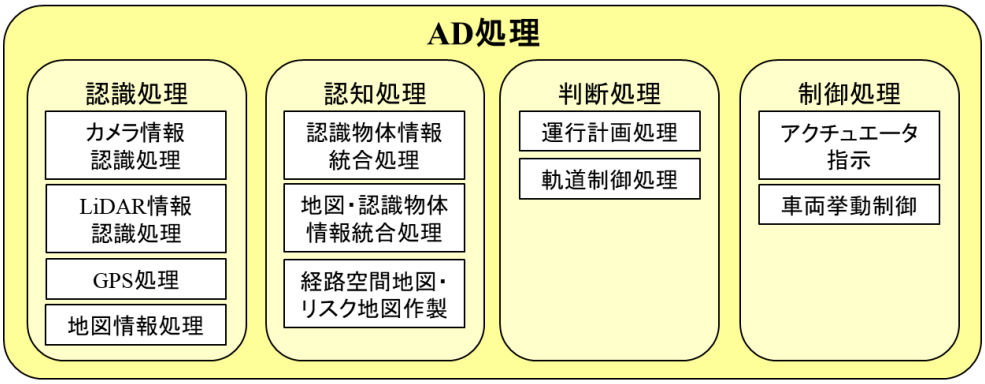
AD 処理では、ソフトウェアが運転を行う。その多くは、認識処理、認知処理、判断処理、制御処理から構成される。

認識処理では、主にカメラや LiDAR（light detection and ranging）、ミリ波レーダなどを利用して物体を認識する。

認知処理では、認識物体情報の統合処理（フュージョン）、GPS などから入力される地図情報と認識物体情報の統合処理、自車周辺の移動体の行動予測、経路・空間地図やリスク地図の作製などを行う。

判断処理では、運行計画処理や軌道制御処理などを行う。

制御処理ではアクチュエータ群に指示を出し、車両の挙動を制御する。



ADのソフトウェア構成例

図 5-8　AD のソフトウェア構成例

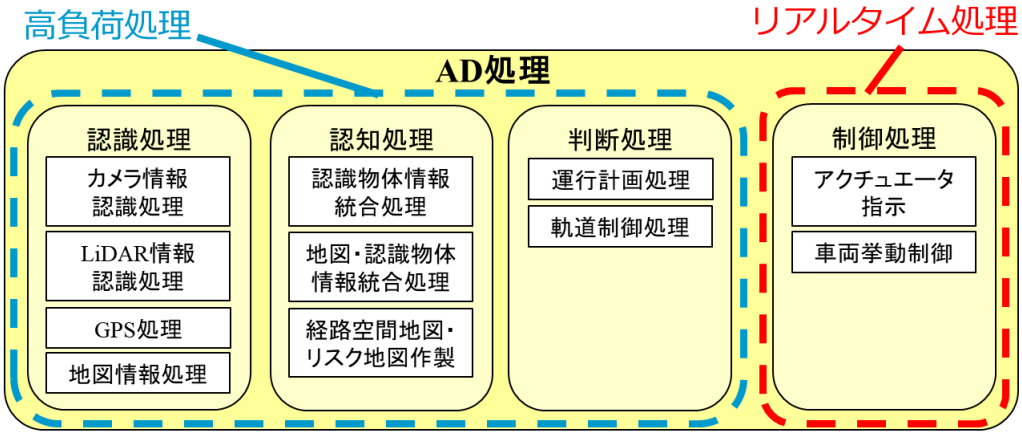
5.2.1.5 AD のソフトウェア処理に求められる要件

認識処理、認知処理、判断処理については、各種情報の統合処理、地図作製、運行処理など、扱う処理量が ADAS と比較にならないほど多くなる。

試作段階でそれぞれの処理量を確認し、どのような負荷分散方式を量産設計に盛り込むかを検討することが

望ましい。

制御処理は、車両の制御指示を行うため、リアルタイム処理が要求される。



ADのソフトウェア処理に求められる要件

図 5-9　AD のソフトウェア処理に求められる要件

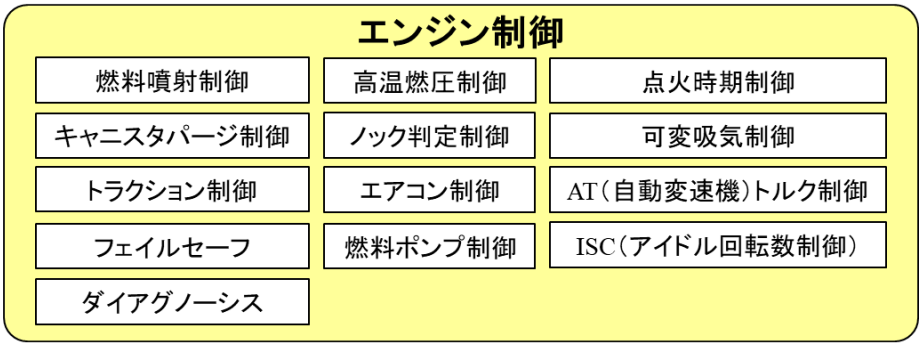
5.2.2　制御系のソフトウェア処理

制御系（エンジン、EV モータ）のソフトウェア処理には、厳格なリアルタイム処理が要求される。エンジンやモータの回転数に応じて、適切に制御することが求められる。

5.2.2.1 エンジン制御のソフトウェア処理

エンジンの回転数に応じて吸気→圧縮→燃料噴射→点火を行う。

それぞれの制御処理が、V 型エンジン、直列型エンジン、気筒ごとに複雑にからみ合う。



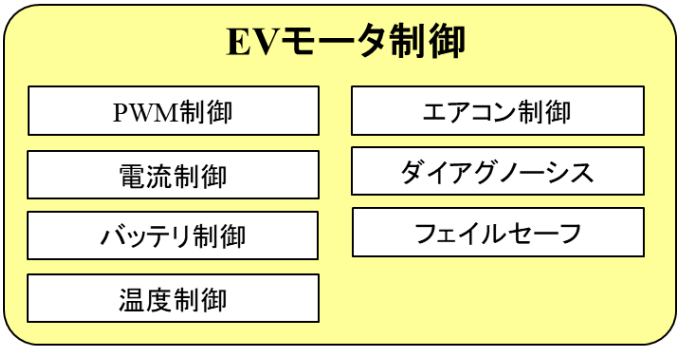
エンジン制御のソフトウェア構成例

図 5-10　エンジン制御のソフトウェア構成例

5.2.2.2 EV モータ制御のソフトウェア処理

EV モータ制御には、エンジン制御のような内燃機関を制御するための処理は存在しないが、低速域から高速域（毎分 0～18000 回転くらい）までモータの回転角速度制御をシームレスに行う必要がある。

回転角速度を処理するためのタイマ割込みは、μs オーダの処理となる。



EVモータ制御のソフトウェア構成例

図 5-11　EV モータ制御のソフトウェア構成例

5.3 ドメインごとの並行処理とマルチコア/メニーコア対応

これまで、車載システムはドメインごとに特徴ある性質を有していることを述べた。以下では、これらの性質を押さえながら、マルチコア/メニーコアに対応していくための指針を述べる。

5.3.1　ADAS/AD 系の並行処理とマルチコア/メニーコア対応

5.2.1 で示した ADAS の ACC･AEB 機能のソフトウェア処理を例に、マルチコア/メニーコア対応の構成例と指針を述べる

5.3.1.1 SMP（symmetrical multi-processing）OS を導入した場合

5.2.1 で示したように、ADAS 処理は、

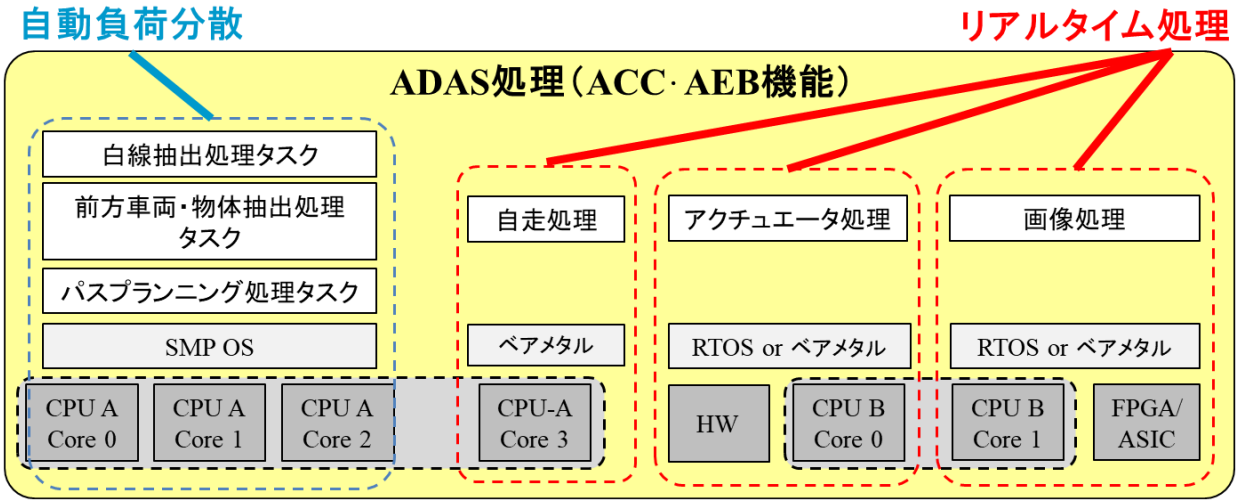
- 高負荷処理：高い負荷性能が求められる処理
- リアルタイム処理：リアルタイム制約を満たすことが求められる処理

に分けられる。それぞれの処理は、別々のプロセッサコアに割り当てる。

高負荷処理については、複数のプロセッサコアを割り当て、SMP OS を導入して負荷分散を行う（SMP OS がタスクを複数コアへ自動的に振り分ける）。

リアルタイム処理に対してリアルタイム OS（RTOS）を導入するかどうかは、接続ペリフェラルや他のプロセッサコアから入ってくる情報をどのくらいの速さでさばかなければならないか（ソフトリアルタイム性）に基づいて決定する。ベアメタル（OS なし）で十分の場合は、無理にリアルタイム OS を導入する必要はない。

マルチコア/メニーコア対応の構成例は、図 1-1 図 5-12 のようになる。



ACC･ABE機能のマルチコア/メニーコア対応（SMP OSを導入した場合）

図 5-12　ACC・ABE 機能のマルチコア/メニーコア対応

5.3.1.2 SMP OS 導入時のタスクの分化について

SMP OS を採用することで、タスクをそれぞれのコアへ自動的に振り分ける負荷分散のメリットを享受できる。このときのタスクの分化（分割）は必要最小限に留めることが重要。タスク数は「割り当てコア数＋せいぜい 1～2」となることが望ましい。

コア数以上にタスクを分化しても、コンテキストスイッチやタスク状態遷移のオーバーヘッドなどにより、性能が頭打ちになることが知られている。関数コールの方が、必要なレジスタ情報をスタックに積むだけなので、当然、処理は早い。タスクを増やしすぎると、逆に性能は出なくなる。

タスクの分化の際には、採用した OS のタスクスイッチングオーバーヘッドまで考慮して、タスク処理時間を設計しておく。こうすることで、タスクのデッドラインに抵触せず、OS 導入のメリットを最大限享受できる。ただし、複数コアになることでデッドライン設計や検証作業はかなり難しくなる。

組込みシステムで使用する OS は、それぞれ性格や特性が異なるので、OS を採用する前に調査を行い、その特徴を把握してから導入することを推奨する。

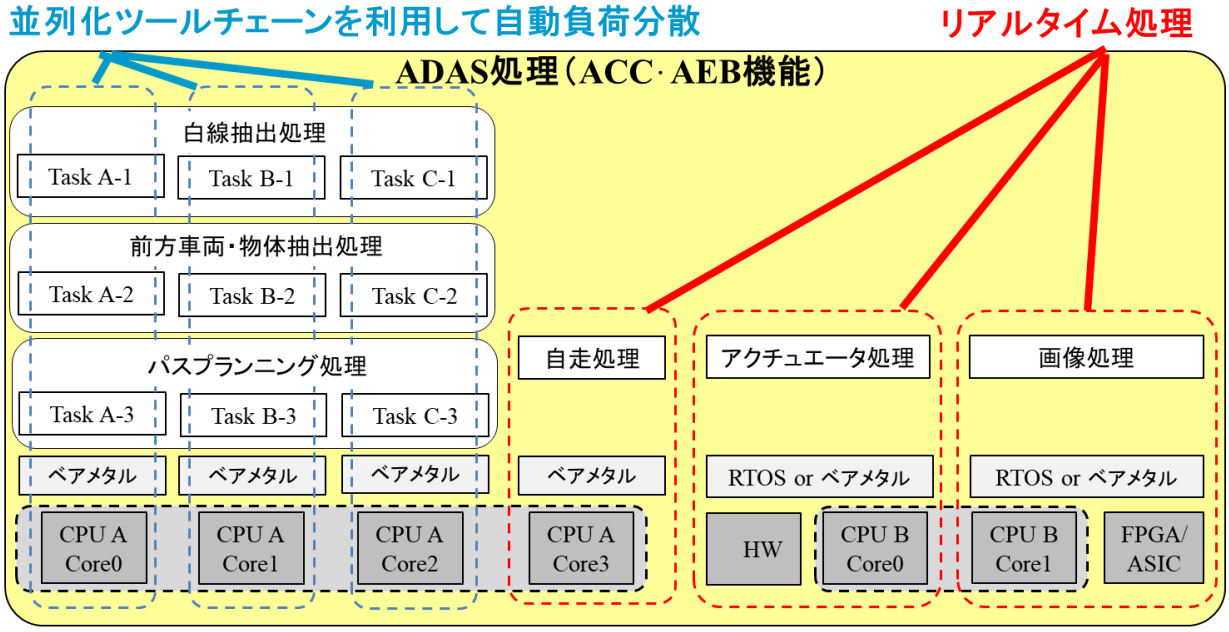
5.3.1.3 SMP OS を導入しない場合

SMP OS を導入しない場合は、各処理をタスク化し、それらを人手で（または、ツールの支援を受けながら）コアへ割り付ける。コア数が、そのままタスク化した処理数になる。

人手で割り付ける場合、プロセッサ資源を効率よく使用できるかどうかは、設計者のスキルに依存する。

SMP OS を採用する、採用しないにかかわらず、マルチコア・メニーコアでプロセッサ資源を効率よく利用するためには、設計から検証まで並列化ツールチェーンを利用することを推奨する。人海戦術には限界がある。

マルチコア/メニーコア対応の構成例は、図 5-13 のようになる。



ACC･ABE機能のマルチコア/メニーコア対応（SMP OSを導入しない場合）

図 5-13　ACC・ABE 機能のマルチコア/メニーコア対応

5.3.2　制御系の並行処理とマルチコア/メニーコア対応

制御系のソフトウェア処理について、マルチコア対応の構成例と指針を述べる。

5.3.2.1 制御系のマルチコア対応

制御系のソフトウェア処理は、割込み処理、タイマ処理、main ループが基本となる。

リスクを上げないように、またなるべく安全にマルチコアを使用するため、

- 仕様の分割
- 割込みの分散
- フィードバック制御
- 処理タイミングによる分割

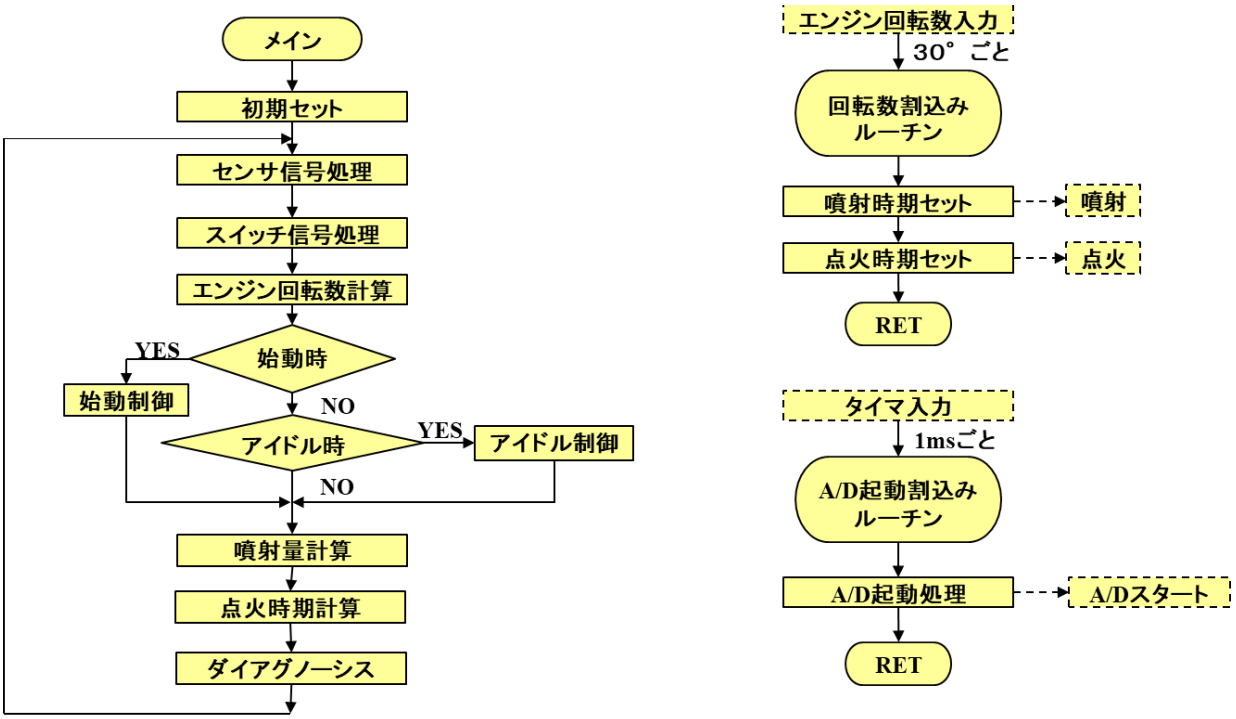
に配慮して、処理を複数コアへ割り付ける。

メモリ参照を可能な限りローカル化する設計、およびコア間におけるペリフェラルのアクセス競合（例えば GPIO など）を生じさせない設計が前提となる。

資源共有は、スピンロックの発生によって処理が止まる可能性がある。これは、ハードリアルタイムが求められる制御処理では致命傷となる。アクセス競合を発生させないように十分に注意を払って、各コアへメモリやペリフェラルを割り当てる。

5.3.2.2　　エンジン制御フローチャート

エンジン制御の基本的なフローチャート例を、図 5-14 に示す。



エンジン制御フローチャート例

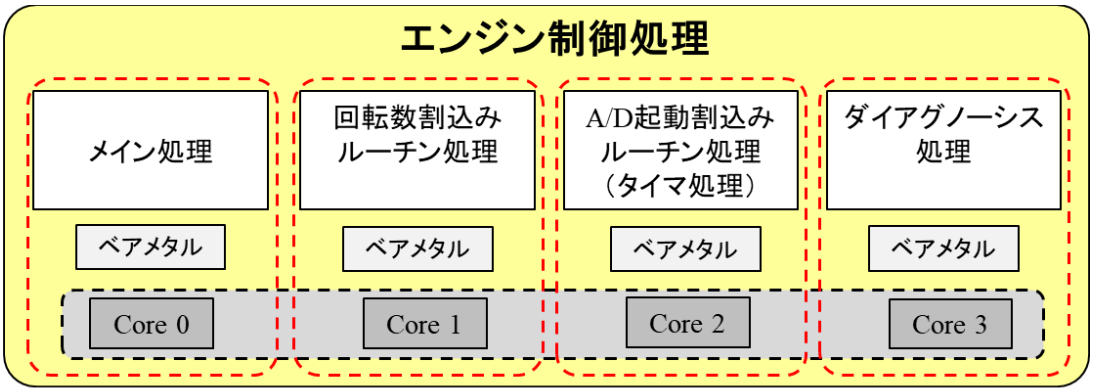
図 5-14　エンジン制御フローチャート例

5.3.2.3 マルチコア構成例

図 5-14 のエンジン制御フローを例に、マルチコア対応の構成例を図 5-15 に示す。

コア割り当て

- Core 0：エンジン制御のメイン処理
- Core 1：回転数割込み処理
- Core 2：1ms タイマ処理
- Core 3：ダイアグノーシス処理
  - メイン処理のダイアグノーシスは、メイン処理に影響を与えない範囲で別のコアに割り当てることが可能。



エンジン制御処理のコア割り当て例

図 5-15　エンジン制御処理のコア割り当て例

5.3.2.4 その他の注意事項

- ■ ここでは 4 コアへ割り当てる例を示したが、メニーコア MCU を使用することも可能。
  - ➢ 割込み処理が複数ある場合に、Core 1、Core 2 以外のコアにも割込み処理を割り当てる（割込み処理負荷分散）。
  - ➢ タイマ処理も、同じように複数コアへ割り当てる（タイマ処理負荷分散）
- ■ エンジンの気筒ごとにコアを割り当てる案も考えられる。しかし、コアごとの物理特性（クロックのぶれなど）が微妙に異なるため、注意が必要。
  - ➢ 上死点、下死点前後の制御処理やクランク角の制御処理などに影響を与え、処理が破綻してしまう可能性がある。

5.4 まとめ

車載システムは、提供する役割により、いくつかのドメインに分類できる。本資料では、組込みソフトウェア開発の立場で、それぞれの特徴について記述した。

マルチコア・メニーコア開発を行うにあたり、ドメインごとに異なるハードウェアアーキテクチャやソフトウェアの処理特性の違いを理解しておくことが重要になる。

ADAS/AD 系と制御系のドメインについて、それぞれのソフトウェア処理、およびマルチコア/メニーコアに対応するための構成例や指針を記載した。

5.5 参考文献

[1]　新井 宏 ; オーム社『自動車の電子システム』,「2 章 エンジン制御」、pp.50-53、2014 年 10 月.

[2] 児島 隆生、長田 健一、伊藤 浩朗、堀田 勇樹、広津 鉄平、小野 豪一；「自動運転の高度化を支える知能化技術」、日立評論、
[3] http://www.hitachihyoron.com/jp/archive/2010s/2017/05/pdf/p52-56_10A03.pdf，2017 年 5 月.
[4] 東芝デバイス＆ストレージ株式会社；「HEV/EV システム」、自動車， https://toshiba.semicon-storage.com/jp/application/automotive/ecology/hev-ev.html
[5] Evsmart Blog；「電気自動車の仕組み」、電気自動車 一般， http://blog.evsmart.net/electric-vehicles/how-electric-car-works/
[6] Autoware；「Open-Source To Self-Driving」、Autoware， https://github.com/CPFL/Autoware/
[7] 株式会社 ZMP；「ADAS の開発に関する基本情報」、自動運転・ADAS を知る、https://www.zmp.co.jp/knowledge/adas_dev/
[8] Technical Direct；「次世代車載情報通信システム（In-Vehicle Infotainment system、IVI システム） スマートドライブによる無限の可能性を開発 未来の新テクノロジーに、ドライブ･イン！」、http://www.technical-direct.com/jp/category/ivi/ 、2013 年 12 月 3 日.
[9] Technical Direct；「IAA2015 レポート：自動車技術の最重要トレンド、コネクテッド・カー」、 http://www.technical-direct.com/jp/category/ivi/ 、2015 年 10 月 2 日.

6 制御系マルチコア・ハードウェアの特徴とユースケース

6.1 背景

近年、組込み装置における高性能化、高集積化を背景にソフトウェアの規模が急速に増大している。2010 年時点においても、スマートフォンに搭載される一般的なソフトウェアの規模は 1200 万ラインを超えており、高級車を対象とした車載ソフトウェアでは約 8 倍の総計 1 億ラインを超える規模となっている。2020 年での推定は高級車 1 台に搭載されるソフトウェア規模は 3 億ライン程度の見通しである。

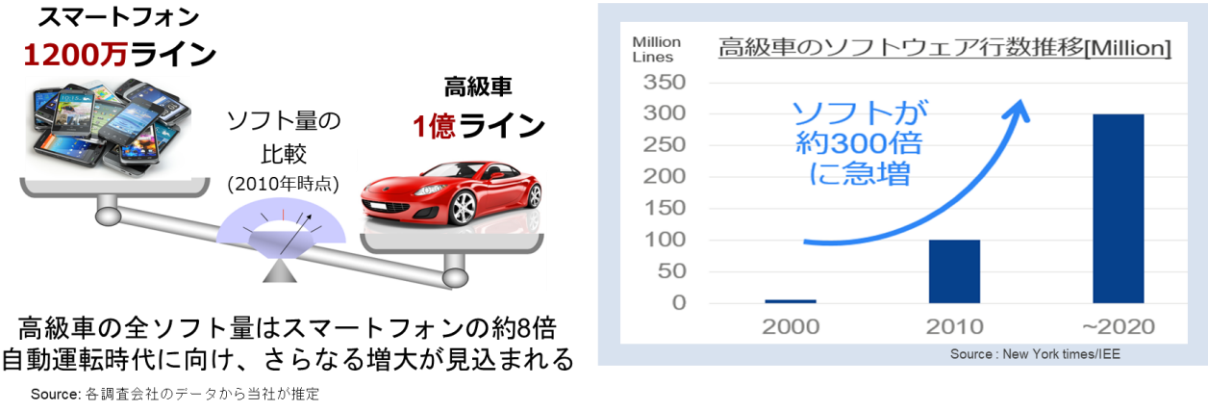


図 6-1 ソフトウェア規模のトレンド

このような大量のコードを実行するには、マイクロプロセッサ(MPU)の能力を非常に高める必要がある。一方で、MPU1 つの処理能力は、物理的な限界から、数 GHz 程度の周波数からの伸びは非常に小さいのが現状である。このため、大量のコードを実行するためには、MPU を複数同時に動作させるマルチプロセッサ構成、あるいは 1 つの MPU のパッケージ内に複数の演算装置(Core:コア)を搭載したマルチコア構成の MPU が必要となる。また、組込み装置は設置場所、形状等の問題から、1 つの基板上に配置可能な MPU パッケージの数量が限定されることもあり、特にマルチコア MPU への期待が大きい。

更に、組込み制御システムでは高いリアルタイム性(ハード・リアルタイム)が要求されることもあり、MPU の動作方式についても一般的な PC 等に搭載される MPU のように、整理され体系化された統一的な構造がとれず、各用途に合わせたハードウェア構成を採用するケースが多い。このため、組込み制御システム向けのマルチコア MPU においては、ソフトウェアの構築あたり、ハードウェアが持つ特徴をよく捉えて設計しなければならず、従来のソフトウェア設計スキルより、より広範な知識が必要となる。

本ドキュメントではそういった組込み特有の制約が多い中で、制御マイコンにおけるマルチコア・アーキテクチャの代表的構成、マルチコア支援機能、高いリアルタイム性を維持するためのハードウェアの仕組みと、それらを前提としたユースケースについて解説する。

6.2 制御システムにおけるマルチコアユースケース

組込み制御向けのマルチコアが想定する代表的なユースケースとして、次の 3 つの課題を解決するアーキテクチャが考えられる。

- ✓ 制御アプリケーションの高性能化 (負荷分散型)
- ✓ 複数のアプリケーションの組合せ・統合 (機能統合型)
- ✓ 機能安全への対応 (機能分離型・時間分離型)

それぞれの課題に応じて適切な構成があり、システム全体の目的として複数の課題に対応する場合には、組み合わせた形のアーキテクチャが採用されるケースも存在する。

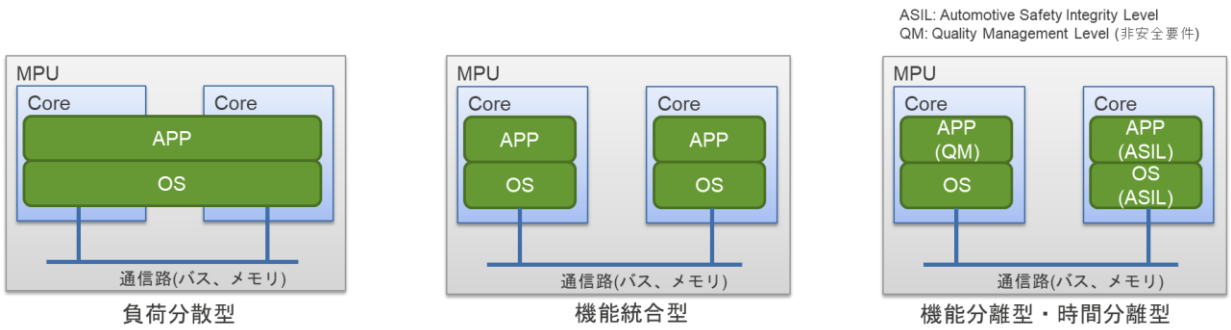


図 6-2 制御システムにおけるマルチコア・ユースケース

6.2.1.1 アーキテクチャ・パタン 1:負荷分散型

負荷分散型は、単一の機能の実行時間性能を向上させるために、本来 1 つのソフトウェアとして動作するコードを分割し、並列に実行することで、実行時間(レイテンシ)を減少させることを目的とする。

本方式は、複数のコアを利用することで並列に動作させた分だけ、最終的な実行完了時刻が早まる(レイテンシが短縮される)。一方で、並列動作をさせる過程で、分割した処理間での通信や、同期待ちなどの分割前のソフトウェアでは生じなかったオーバーヘッドが生じる。このオーバーヘッドが、処理を並列化したことによる短縮された時間に見合わない場合は、性能向上が得られず、場合によっては劣化してしまうことがある。

図 6-3 に示すように 1 つのソフトウェアが複数のコアにまたがるように配置し、またそのプログラムを管理する OS もマルチコアに対応した OS を用いるのが理想である。独立した複数の OS 上(あるいは OS レス)に構築することも可能だが、その際は並行動作するプログラムの権限や実行の管理はすべてユーザが行う必要があり、難易度が高い。

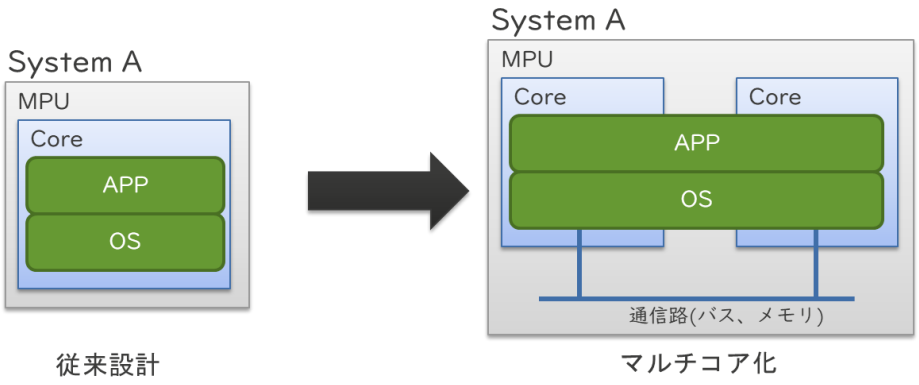


図 6-3 負荷分散型マルチコア

本方式の特徴と必要とされる技術を示す。

- 利点
  - ✓ マルチコア化による実行時間性能の向上
-
- 欠点
  - ✓ 並列化に適したソフトウェア構成でない場合、性能が期待通り向上しない
  - ✓ バスや共有メモリを介したデータ交換のオーバーヘッドが生じる
-

- 必要とされる技術要素
  - ✓ 並列化の粒度：アプリケーション以下
  - ✓ ソフトウェアの並列化支援ツール
  - ✓ マルチコア OS
  - ✓ 高速な共有メモリ
  - ✓ 高速な CPU 間イベント通信

この方式では、複数のコア間では同一の性質、権限を持ったソフトウェアが動作し密接に関わり合うため、特に高性能を追求する場合は、ハードウェアが持つバスや共有メモリの機能・構成について熟慮する必要がある。

ソフトウェア構築にあたっては、マルチコア OS の機能支援や、ハードウェアが用意する拘束な CPU 間イベント通信の機能を積極的に利用することで、ミスや設計負担を減少させることができる。また、単一のソフトウェアを並列動作可能な形に分割することは非常に困難であり、コードの構文や構造から自動的に並列化する支援ツールを利用することも視野にいれたい。

6.2.1.2 アーキテクチャ・パタン 2: 機能統合型

機能統合型は、複数の独立したシステム(または機能)を 1 つのマルチコアに集積し、BOM コストを低減、またはパッケージの設置面積を縮小することが目的である。

本方式はもともと異なる MPU で実行していたシステムをそれぞれのコアに搭載し、複数の機能を 1 つの MPU で実現するものである。それぞれのコアに搭載されたシステムは基本的に独立して動作し、干渉を最小限に保つ必要がある。

図 6-4 に示すように、それぞれのコアには、それぞれシングルコア用の OS が搭載され、あたかも独立した MPU であるかのように動作する。協調動作が必要な場合は、システム内の共有資源を用いて通信を行うが、ソフトウェアの概念上は複数の MPU に分かれていた場合と同様に取り扱う。

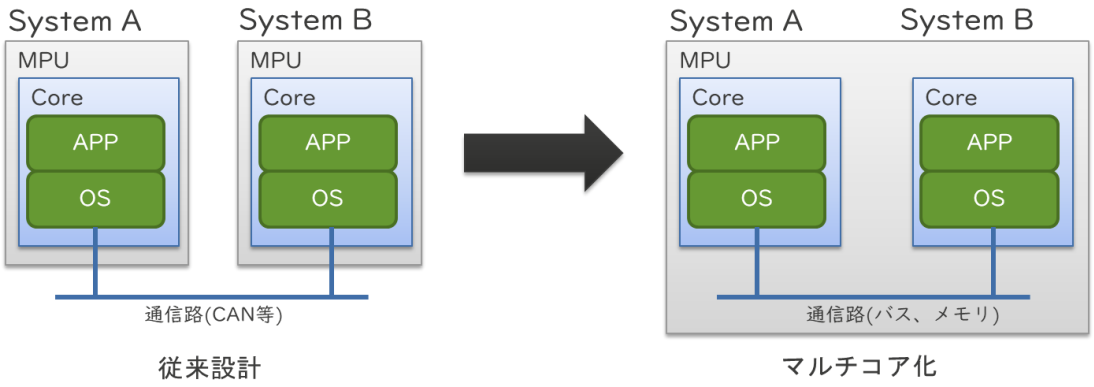


図 6-4 機能統合型マルチコア

本方式の特徴と必要とされる技術を示す。

- 利点
  - ✓ 部品点数削減による BOM コストの低減
  - ✓ パッケージの設置面積をマルチプロセッサ構成より低減
  - ✓ 通信路が MPU 内部で閉じるため、通信レイテンシがマルチプロセッサ構成より小さく、効率的な協調動作が可能

- 欠点
  - ✓ バスや共有資源の競合による性能劣化(干渉)が発生するため、マルチプロセッサ構成よりも取り扱いが難しい
  - ✓ 統合により共通化された部品、共有資源などの故障により、同時に障害が起きるケースがある(共通故障)

- 必要とされる技術要素
  - ✓ 並列化の粒度：アプリケーション単位
  - ✓ 仮想化技術／ハイパーバイザ
  - ✓ OS 間通信フレームワーク
  - ✓ メモリ・周辺装置のパーティショニング技術
  - ✓ 干渉の少ないバス・システム

この方式では、異なるコアに搭載したソフトウェア・システム間の独立性の保証や、干渉の低減について強

く注意する必要がある。このため、バスや共有資源の分離性や、性能の干渉などについて一定の保護(パーティショニング)がハードウェア的に施されている場合がある。

またソフトウェア側も共有する資源が存在する場合には、その利用方法などに一定のマナーを課すことや、性能低下に対する許容範囲などを事前に検討が必要となり、これをサポートするための仮想化技術や、統合前と透過に通信が行えるような通信フレームワーク技術 t などが、OS やハイパーバイザと呼ばれる管理プログラムによって提供されるべきである。

アーキテクチャ・パタン 3: 機能分離・時間分離

機能分離型・時間分離型は、図 6-4 の機能統合型と構成は似ているが、それぞれのコアに割り当てられたソフトウェアが必ずしも独立したシステムではなく、時に協調や監視といった密接な関係で動作する状況が存在する。機能分離はソフトウェアの実行結果への影響、時間分離はソフトウェアの実行時間への影響を示す。

図 6-5 に示すように、異なるコアで安全水準が異なるソフトウェア(QM,ASIL)を動作させ、高い安全水準を持つソフトウェアの動作が、低い水準のソフトウェアにより阻害されることを防止する。そのためにハードウェア機能と、その上で動作するソフトウェアの開発手法を組み合わせて、システムの安全性を保つことを保証する。

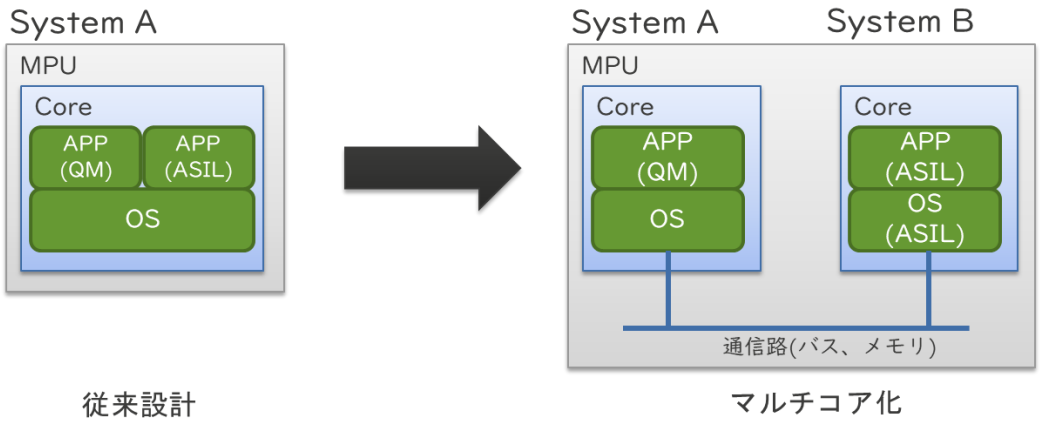


図 6-5 機能分離型・時間分離型マルチコア

本方式の特徴と必要とされる技術を示す。

- 利点
  - ✓ 異なる性質のソフトウェア間の干渉が低減可能
  - ✓ OS を含めて分離することで、個々の OS は機能最小限のコンパクトな実装が可能となる

- 欠点
  - ✓ バスや共有資源を介したデータ交換などの際に、安全水準を保つための追加手順が必要となる
  - ✓ 分離を保つ管理プログラムとして品質の高いハイパーバイザ・プログラムが必須であり、このハイパーバイザ処理によるオーバーヘッドが生じる可能性がある。

- 必要とされる技術要素
  - ✓ 並列化の粒度 アプリ単位 or アプリ以下
  - ✓ 仮想化技術／ハイパーバイザ
  - ✓ 共有メモリ
  - ✓ メモリ・周辺装置の分離技術
  - ✓ 帯域分離が可能なバス・システム

この方式では、機能統合型では必須ではなかった保護(パーティショニング)が必須となる。システムの目的は安全水準に従って、保護すべきソフトウェアの動作を正常に保つことを第一義として、性能を目的として条件を緩和することは許容されない。一方でソフトウェアが定められた時間で所定の処理を完了させることも、安全性の要求に含まれるため、マルチコアで動作した場合の干渉等の予測可能性も非常に高いレベルで要求される。

6.3 制御系マルチコア・ハードウェア

本章では組込み制御システム向けのマルチコア MPU において採用されるハードウェア要素技術について紹介し、それぞれの役割、特徴点を説明する。これらは前述したユースケースを成立させる上で必要な要素技術である。

6.3.1 マルチコアの種類

マルチコア MPU の大きな種別を把握する方法として、コアの構成とその利用分野ついて紹介する。

マルチコアの種類は、構造から「対称型マルチコア」「非対称型マルチコア」、応用から「制御系マルチコア」「情報系マルチコア」に分けられる。それぞれの詳細は本章の各節で述べる。

6.3.1.1 対称型マルチコア(ホモジニアス)

同一性能のコアを複数搭載したマルチコア MPU は、対称型マルチコア(ホモジニアス・マルチコア)と呼ばれる。すべてのコアが同性能のため、ソフトウェアの配置を変更した際にも同一の実行時間で動作することが期待され、ソフトウェアを分割配置する際に結果が予測しやすい。このため汎用的な目的で利用される MPU や、大規模な処理を行う多並列処理を行う MPU で採用されることが多い。

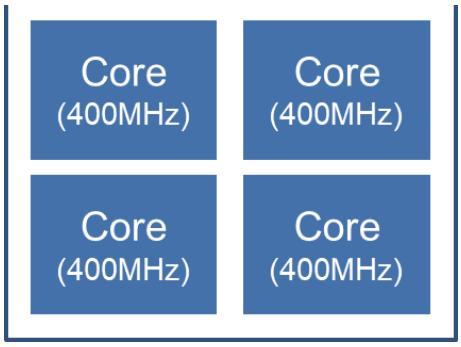


図 6-6 対称型マルチコア(ホモジニアス)

6.3.1.2 非対称型マルチコア(ヘテロジニアス)

異なる性能のコアを搭載したマルチコア MPU は、非対称型マルチコア(ヘテロジニアス・マルチコア)と呼ばれる。対称型マルチコアと異なり、性能が異なるコアが搭載されているため、ソフトウェアを分割し配置した場合に、配置先のコアに応じて分割したソフトウェアごとの性能が変動する。このため、ソフトウェアの配置最適化の検討時に難易度が上昇する。しかし、使用用途が明確であり、各コアに配置するソフトウェアの要求する性能に大きく差がある場合、コアの計算能力が無駄にならない利点がある。

例えば、組込み用途ではメインの計算処理を行うソフトウェアと、比較的軽量なシステムを保全する処理(システム制御)を行うソフトウェアに分割される場合などに、処理性能の高いコアにメイン処理、低いコアにシステム制御処理を割り当てることで、最適な動作が可能となる。これにより全体の消費電力の低減や、待機状態での電力消費を最小化が実現できる。

また、非対称型の場合は性能だけでなく、命令セットアーキテクチャなどから異なるコアを採用する場合がある。

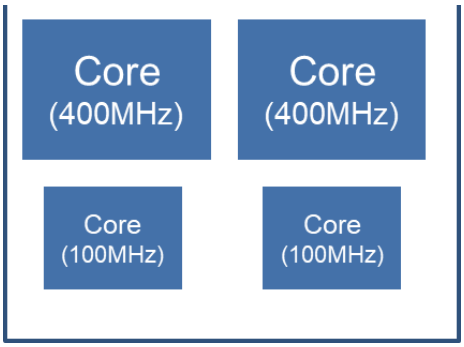


図 6-7 非対称型マルチコア(ヘテロジニアス)

6.3.1.3 制御系マルチコアの代表構成(MCU:マイクロコントローラ)

システムの目的に応じて、最適なマルチコア構成は変化する。組込み制御システムにおいては、基板の設置場所の容積や熱容量の制約があり、マルチコアが要求されるケースにおいても、200 MHz～400 MHz 程度の処理性能を持つコアを 2 コア～4 コア程度集積する場合が多い。

制御系の処理は多数の並列性を抽出しにくい性質であると共に、画像処理のような同一処理内容で大量の並列実行が可能なデータ並列性が少なく、様々な内容の処理を細かく小さく実行するタスク並列性が中心である。これらの処理量が異なる処理を、ある程度まとめて少数のコアに配置していくにあたり、非対称マルチコアでは最適な構成の探索が非常に困難となる。

このような背景により、用途がはっきりしているという印象の制御系マイコンにおいては非対称型マルチコアよりも対称型マルチコアが主流である。もちろん、一部の特定用途の処理などを行う補助的なコアが搭載されることもあるが、通常のプログラミング空間からは分離されている場合が多く、ここでは除外して考える。

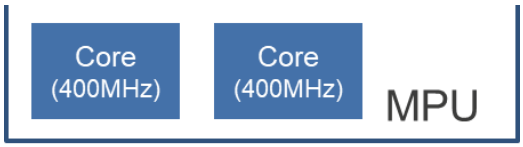


図 6-8 代表的な制御系マルチコア構成

6.3.1.4 情報系マルチコアの代表構成(SoC:システム・オン・チップ)

組込み機器向けであっても、大規模な SoC チップを用いた情報系処理を担う MPU も存在する。本ドキュメントでは主に制御系を対象としているが、ここでは簡単に情報系処理におけるマルチコアについても簡単に言及する。

情報系マルチコアでは、対称型マルチコアと非対称型マルチコアが組み合わせられるケースが多い。大規模なデータ並列性の存在するメインの処理群を高性能(1GHz 超，4 コア以上)の対称型マルチコア部分で行い、それ以外の雑多な機器メンテナンスを行う入出力の処理を比較的低性能(数 100MHz)のコアで行う形が一般的である。

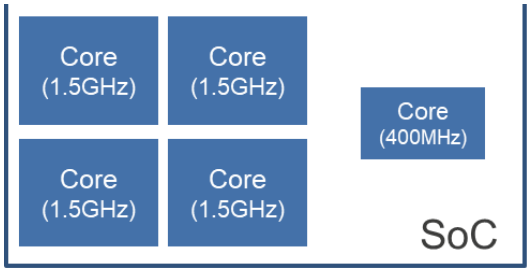


図 6-9 代表的な情報系マルチコア構成

6.3.2 メモリ構成

マルチコアにおいてメモリ構成は最も重要とも言える要素である。メモリはコアに配置したプログラムの実行時間、コア間のデータ共有のための通信時間に関わり、メモリへの操作を効率化することがマルチコアでのトータル性能を向上させる際に最も影響の大きいパラメータとなる。

あるコアから見たメモリ操作に関わるパラメータとして、重要なものは以下の 3 つである。これは組込みマルチコアコンソーシアムが提唱するハードウェア抽象記述 SHIM(Software Hardware Interface for Multi-Many-core)[^shim] に規定されたメモリ表現とも一致する。

- ✓ アクセシビリティ
- ✓ レイテンシ
- ✓ ピッチ

アクセシビリティは、いずれのメモリがどのコアからアクセスが可能かを示す。協調動作をする際には各コアが同じメモリにアクセスが可能でなくてはならないが、組込み制御システムにおいては特定のコアからのみアクセスが可能なローカル・メモリが定義されている場合がある。

レイテンシは、コアが当該メモリにアクセスした際にデータが返却されるまでの時間を示す。ピッチは、連続で要求を発行した際に、返却されるデータ間の間隔を示す。詳細は SHIM ドキュメントを参照のこと。

バス・システムを介したメモリへのアクセスが常に 1 つの要求のみを受付け、データが返却されるまで他の要求を実行しない場合(ブロッキング・アクセス)、レイテンシとピッチの値は等しくなる。これに対して最初のデータが返却されるまでの間に、後続の要求の処理がある程度進み、最初のデータが返却された後に、短い間隔で次のデータが返却されるような場合(ノンブロッキング・アクセス)、ピッチの値はレイテンシより小さくなる。

このように、SHIM におけるメモリ表現はバス・システムを含むメモリ操作に関わるハードウェア機能を隠蔽し、比較的取り扱いの簡単なパラメータで表現可能としている。本章では、この SHIM のメモリ表現で説明できる粒度で、マルチコア・ハードウェアへの理解を深めるため、いくつかの特徴的な構成や機能について紹介する。

[^shim]: https://standards.ieee.org/standard/2804-2019.html

6.3.2.1 TCM: Tightly coupled memory(Local memory)

各コアが高速に利用できるメモリとして、コア近傍に持つ小容量のデータ・メモリを TCM(Tightly coupled memory)と呼ぶ。コアが局所的に保持するメモリという意味で Local Memory と呼ばれることもある。

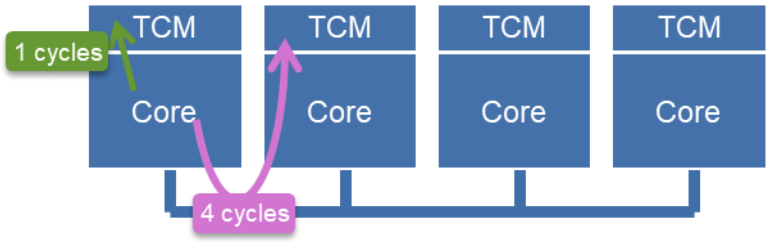


図 6-10 TCM: Tightly coupled memory (Local memory)

カップリングされたコアのみが高速に実行できるため、そのコアに割り当てられたソフトウェアのスタック・メモリや、プライベートなデータを配置するために用いるのが一般的である。通常は、このメモリへの操作はコアにとってはペナルティ(待ち時間等)のない形でデザインされる。このため、非常に実行時間の予測性が高いのも特長の一つである。

カップリングされたコア以外からのアクセスは、ハードウェアのポリシによって許可する場合と禁止する場合がある。禁止する理由は、他コアからの干渉を防ぎ、予測性や安全性を高めるためである。アクセスを許可する場合も、他コアからのアクセスには余分なレイテンシが生じる。

TCM を共有メモリとして利用する場合は、主に単方向通信の場合である。通常、送信コアから受信コアの TCM に向かって書込みが行われる。一般的には書込みは結果を待たずに後続の処理が実行できるため、レ

イテンシの大きなメモリへ行ってもペナルティが顕在化することは少なく、読込みは結果を必要とする処理が後続に控えているため、可能な限り速やかに完了できることが望ましいからである。

6.3.2.2 等距離共有メモリ

等距離共有メモリは、最も一般的な形態の共有メモリである。すべてのコアから同じレイテンシでアクセスできるため、ソフトウェアをどのコアに配置しても性能が大きく変動せず、非常に使い勝手の良い共有メモリである。

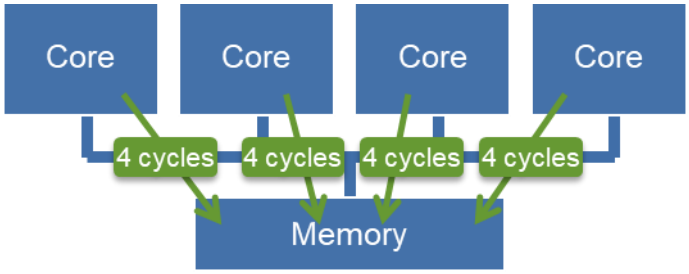


図 6-11 等距離共有メモリ

一方で、等距離であることを維持するために、コア数が増えた場合にレイテンシが増加しがちであり、大規模なマルチコアで頻繁な共有を行うケースでは、性能のボトルネックを生じさせやすい。

6.3.2.3 非等距離共有メモリ

非等距離共有メモリは、コア毎にアクセスする際のレイテンシが異なる共有メモリである。等距離共有メモリと異なり、近傍のコアに対しての性能が高く、コア数が増えた場合にも性能が劣化しにくい。反面、ソフトウェアの配置はレイテンシが異なるグループに移動した場合には変動してしまうため、やや難しくなる。

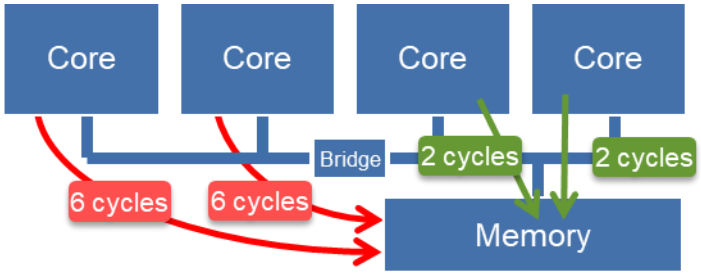


図 6-12 非等距離共有メモリ

図 6-10 の TCM は非等距離共有メモリの一種だが、同じレイテンシを持つグループのコアが存在しない特

殊例となる。非等距離共有メモリは、6.3.3 節で説明するクラスタ構造にも密接に関わる。

6.3.3　クラスタ構造

大規模なマルチコアの場合、一部のコアをグループ化して取り扱うクラスタ構造(サブシステム、アイランドなどとも呼ばれることがある)を取る場合がある。その際、6.3.2.1， 6.3.2.2 で説明した各種メモリ構成を組合わせて、図 6-13 のような構造を取ることが多い。

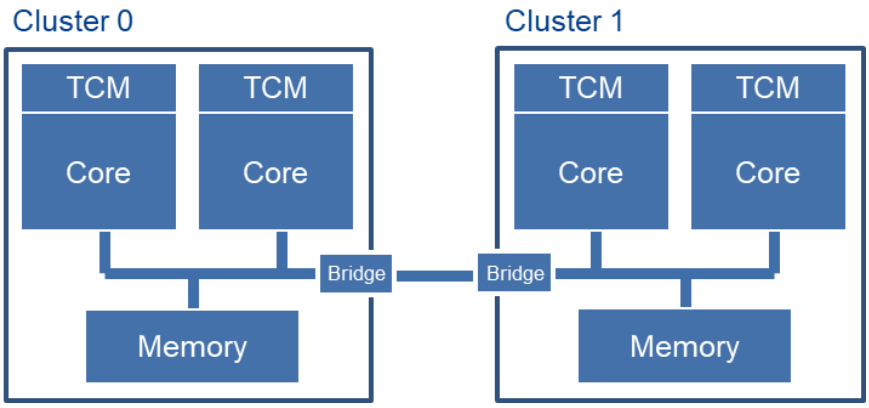


図 6-13 代表的なクラスタ構造例

各コアに TCM を搭載しプライベートな処理は原則的に TCM 上で行うことで高速性を保ちながら、クラスタごとに非等距離共有メモリを持つ。これにより、クラスタ内でのソフトウェアから見ると、比較的取り扱いやすい等距離共有メモリが存在するように見えるため、負荷分散等の検討が容易になる。

これによる利点として、次の点が挙げられる。

- ✓ クラスタ毎に回路レイアウトを行うことで、共有メモリのアクセス性能や周波数性能等の引き上げが容易
- ✓ クラスタ毎に独立したハードウェア構成であり、互いに鑑賞しないソフトウェア群の性能保証や安全性を保ちやすい
- ✓ 電力管理などハードウェア資源の階層的な管理単位としても利用できる

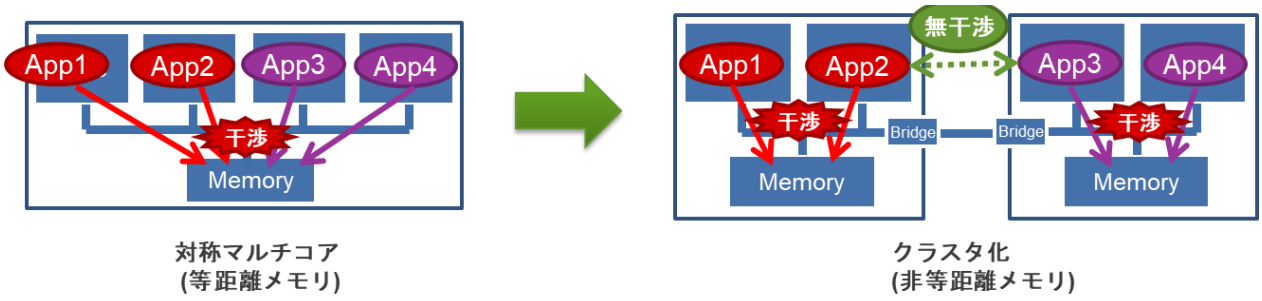


図 6-14 クラスタ構造の利点:相互干渉の低減

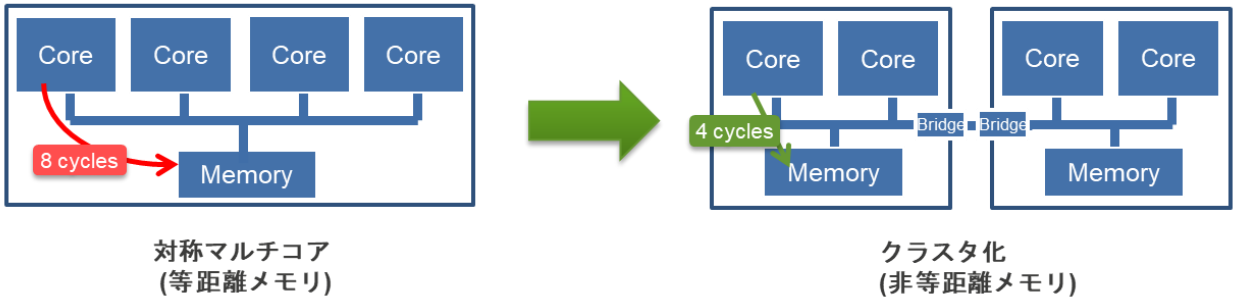


図 6-15 クラスタ構造の利点:性能引き上げ

この例では 4 コアと少数のマルチコア構成であるが、8 コア、16 コアとコアが増加するに従ってクラスタ構造のメリットは大きくなる。また、更に外側に大容量の等距離共有メモリ(DRAM など)を配置することも SoC などでは行われることが多い。

また、少数コアでのクラスタ構成としては、車載システムなど安全性が重要なシステムでは実際に採用されるケースもある。

6.3.4 メモリ・インターフェース

マルチコアを用いる際、必ず発生するのが共有メモリへのアクセス競合である。複数のコアが同時に同じメモリを使うことで、メモリアクセスを順番に行うために性能が著しく劣化することがある。

この問題に対してハードウェアではメモリ・インターフェースにも工夫を凝らしている。これらの仕組みはメモリ・アクセスのレイテンシやピッチの期待値(平均値)として現れてくるものである。

6.3.4.1 メモリ・バンク

メモリ・バンクはメモリの内部を、連続したアドレスの範囲で区切って並行的にアクセスを可能とする。

図 6-16 の例では 2 MiB のメモリを 1 MiB 毎に区切って 2 バンクの構成をとってものである。それぞれ上位、下位の範囲で並行してアクセスが可能であり、それぞれ別のコアが利用するようにプログラムを構成することで、メモリ衝突によるペナルティを排除できる。

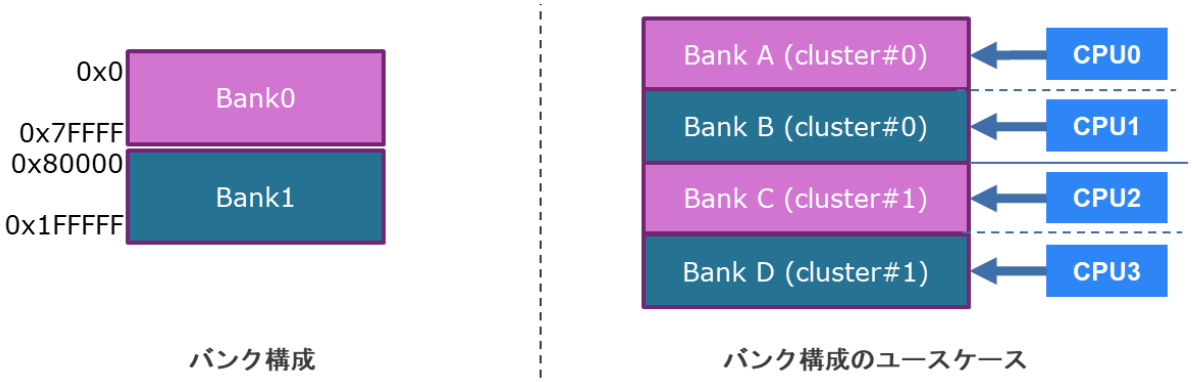


図 6-16 メモリ・バンク

メモリ・バンクはコンパイラ(リンカ)の設定によって、意識的に制御しメモリの競合を回避できるため、ソフトウェア設計時にメモリ・バンクの構成を考慮して検討を行うことが重要となる。

6.3.4.2 メモリ・サブバンク

メモリ・サブバンクはメモリの内部を一定のアドレス毎にスキップしたデータを 1 組としてバンクとし、オフセットの異なるデータ同士は異なるメモリとして並行アクセス可能とする。

図 6-17 の構造例では、16 B の範囲を 4 B 毎にサブバンクとしている。すなわち、アドレス 0x0, 0x10, 0x20, 0x30...の各 4 バイトをサブバンク 0、アドレス 0x4, 0x14, 0x24, 0x34...をサブバンク 1 のようにグルーピングしている。これにより比較的近傍の範囲のデータ(例えばコア間の共有データなど)へ、複数のコアが同時にアクセスした際にもペナルティが生じない可能性が確率的に高まる。

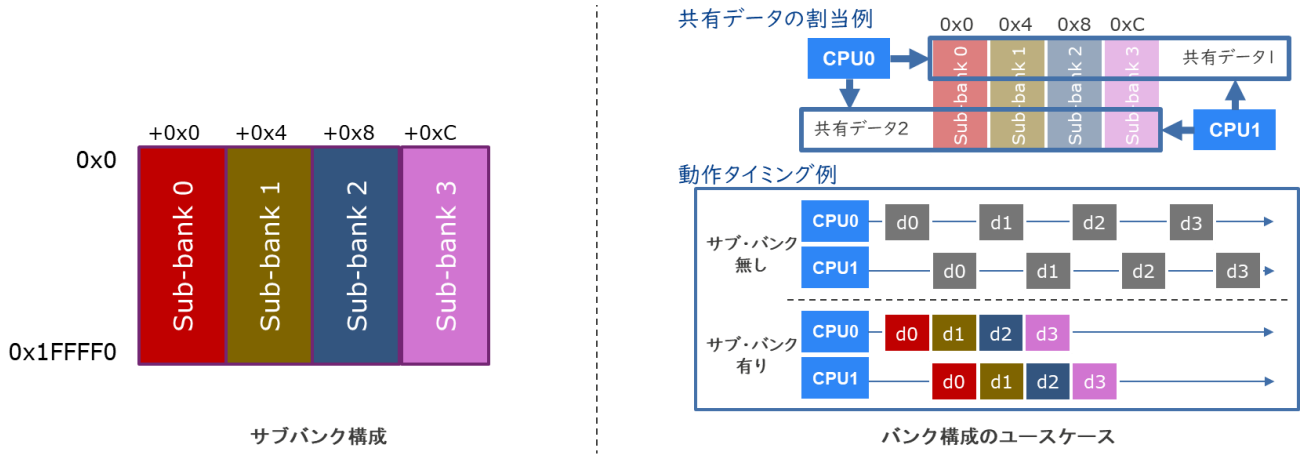


図 6-17 メモリ・サブバンク

図 6-17 の動作例では、それぞれのコアが同じオフセットアドレスにアクセスを行った際にもサブバンクよってペナルティを除去できる。この仕組みは、ソフトウェアではあまり意識する必要はないが、平均的なアクセス性能に大きく貢献する。

6.3.5 プロセッサ間割込み／イベント・フラグ

マルチコアを利用する際に、複数のコアでの協調動作が必要となる。データの共有は共有メモリを介して行うが、ソフトウェアによっては処理がある状態に至ったことを相手に伝えるためのイベント通知の必要が生じる。

マルチコア向けハードウェアでは、通知の形態によって大別して次の 2 種類の通知方法をサポートする。

- ✓ プロセッサ間割り込み
- ✓ プロセッサ間イベント・フラグ

それぞれ特性が異なるため、システムが要求する動作条件に従って選択する。また、マルチコア OS が本機能を用いてメッセージ通信機能などを提供することもあり、この場合は、より高機能な通信 API が提供される。

6.3.5.1 プロセッサ間割り込み

プロセッサ間割込みはシングルコア MPU でも備えている割込みをマルチコア MPU 向けに MPU 内部のコアで相互に割込み要求を行えるようにする機能である。通常、プロセッサ間の処理起動を目的として、

Remote Procedure Call(RPC)を実現する手段として用いる。

図 6-18 のように、左の要求側コアからソフトウェア操作によって要求レジスタ(IRQ)を操作することで、通知先のコアに割込み要求がなされる。割込みは通常の割込みコントローラに接続され、他の割込みと同様にマスクや要求取下げなどの管理を受信側が任意に行うことが可能である。

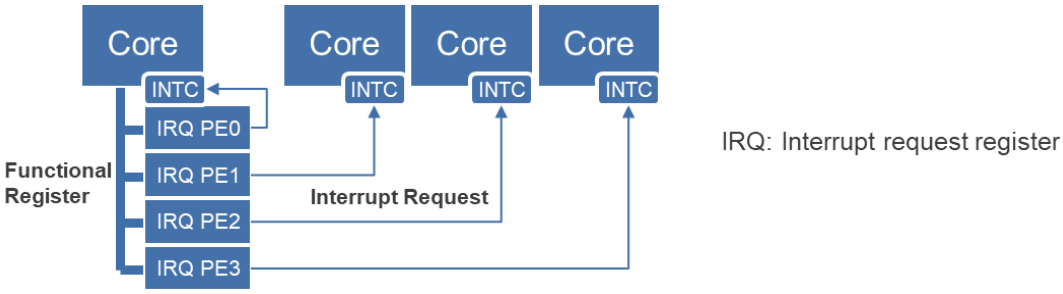


図 6-18 プロセッサ間割り込み

プロセッサ間割込みは対象コアの処理状態を変化させるため強制力・即時性がある反面、要求されたコアの処理中断を引き起こし、システム全体の処理能力の低下やリアルタイム性の減少を引き起こす場合がある。ただし受信側は割込みのマスクが可能であり、真にクリティカルな状況では中断を拒否できる。

マルチコア向けの機能としては、1 操作で複数のコアに同時に割込み通知を行う同報通知や、要求回数をカウントするような仕組みを備える場合がある。

6.3.5.2 プロセッサ間イベント・フラグ

プロセッサ間イベント・フラグはプロセッサ間割込みとは異なり、割込みを介さずにイベントの有無を受信側に伝える仕組みである。

図 6-19 のように、基本的にはプロセッサ間割込みと同様の構成だが、要求レジスタ(Snd)から接続される先は割込みコントローラではなく、ただの機能レジスタとなっている。このため、受信側の処理が中断されることはなく、受信側のプログラム進行に合わせて都合の良いタイミングでイベントの有無が確認できる。

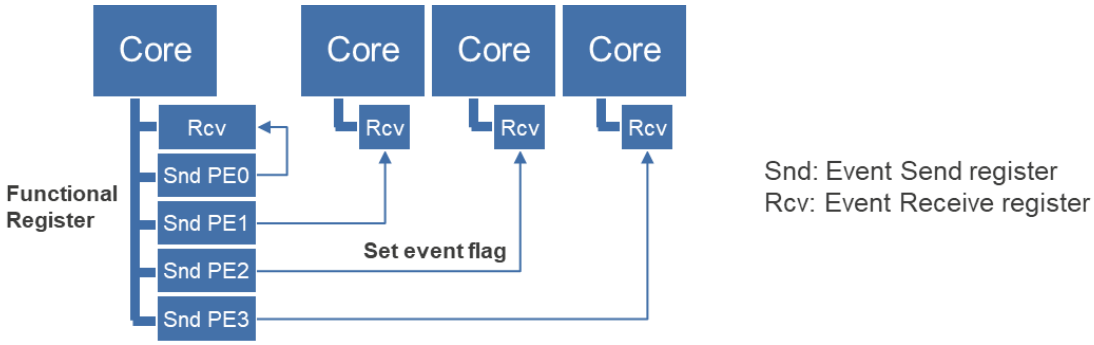


図 6-19 プロセッサ間イベント・フラグ

この手順のみであれば共有メモリ上でも同等のことが行えるが、プロセッサ間イベント・フラグの特有の価値として補助的な機能が備わっていることが多い。例えば、イベント・フラグのセットに応じて、受信側のコアがスリープ状態などであった場合に起床可能であるなど、システム制御面での付加価値がある。

また、共有メモリを介すると他コアのプログラムの動作次第でイベントの通知にかかるレイテンシが変動してしまう。これらを避けて一定の時間での通知完了を確定できる点が強みとなる。

6.3.6 排他制御用レジスタ (Mutex register)

プロセッサ間で複数の共有データの組(セット)を相互に参照・更新する場合、複数の共有データ間での一貫性を保持するために、互いに現在、参照中または更新中であることを把握するための手順が必要となる。通常、この Mutex(MUTual EXclusion)と呼ばれる手順を実現するためには共有メモリを介してやり取りを行うが、ハードウェアによっては専用の機能レジスタを備えている場合がある。

Mutex を構成する場合、鍵となるデータ(ロック変数)を互いにアクセスし、空き状態であれば利用状態に更新した後、共有データ本体を操作する。利用状態であれば、空き状態になるまで何度も繰り返し確認(ポーリング)し、空き状態になるのを待ち続ける。この過程で、頻繁にロック変数へのアクセスが発生することで、本来行うべき処理とは無関係なアクセスが頻出することとなる。これはシステムのバス資源を無駄に消費し、システム全体の性能を低下させるが、一方で各コアの処理のレイテンシ短縮のみを考えると、全力でロック変数にアクセスすることは必要な処理である。従って、マルチコア・システムはこのロック変数への頻繁なアクセスと、それによるシステム性能の劣化防止を両立する必要がある。

排他制御用レジスタと呼ばれる機能レジスタは、共有メモリが接続されているメイン・バス・システムから独立した各コアに直結したパスで接続された小さな共有メモリ(レジスタ)である。これにより、どんなに各コアが鍵にアクセスを行ったとしても、メインとなるバス・システムには一切の影響を与えないことを実現している。

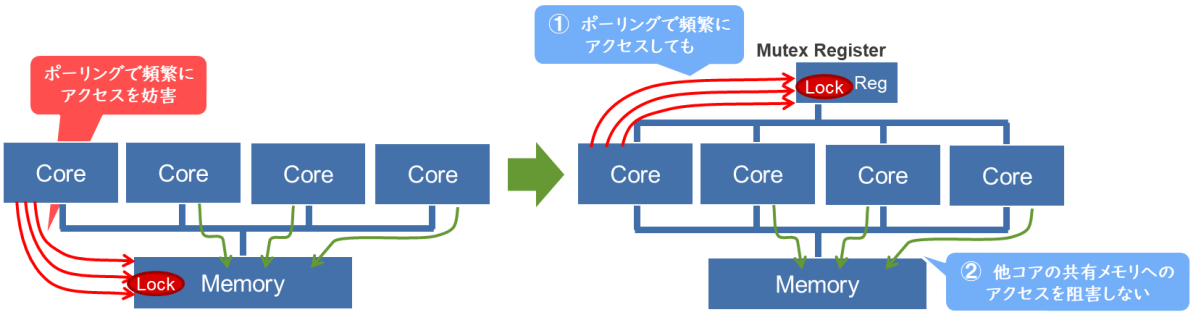


図 6-20 排他制御用レジスタ(Mutex register)

排他制御用レジスタは、アクセス経路以外は特長を持たないただのレジスタであり、時に 1 ビットではなく複数ビットの語長を持つ場合がある。ソフトウェアはこれらのビットを使い、意味をもたせることで単純な mutex だけでなく、バリア同期や Counting semaphore などを実現することができる。

6.3.7 同期化処理

共有メモリを介してコア間でメッセージのやりとりを行う際に、あるコアから書き込んだ値を確実に別のコアに伝える必要が生じる場合がある。これを実現するために、同期化処理が必要となる。

共有データに書込み命令を実行後 Mutex を解放し、受信側コアが操作可能にするといった手順を考えた場合、共有メモリ上にデータを書込みした後に何もせずに Mutex を解放すると、受信側が読出す前に共有メモリへの書込みが終わっていることが保証できない可能性がある。

近年の MPU では処理性能の向上のために、書込み命令を実行しても実際に共有メモリへの書込みが終わる前に、後続の命令を実行する。これは、都度書込みの完了を待っているとアクセス先によっては書込み命令の実行に長い時間がかかり、性能が低下する可能性があるからである。

このため、図 6-21 のようにコア 1 による Write (B)の結果が共有データに書かれる前にコア 2 が Mutex を獲得し、更新前の A という値を読み出してしまうことが起こりうる。これは共有データを低速な等距離共有メモリに配置し、Mutex を排他制御レジスタに配置した場合などに起きる可能性がある。

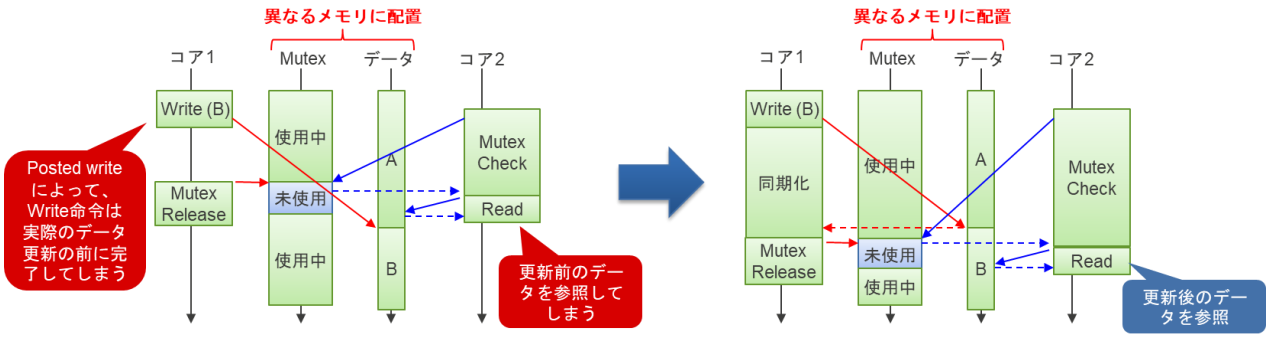


図 6-21 同期化処理

これに対して、ハードウェアは一定の同期化処理を定義しており、その手順をソフトウェア的に実行することで、先行する書込みが完了することを保証可能としている。ただし、この手順はハードウェアのバス構成やコアの機能と密接に関わっており、ハードウェア毎に手順が大きく異なる。

ハードウェア・マニュアルには「同期化処理」「完了保証」などのワードで注記されている場合が多いため、ハードウェア・マニュアルをよく確認する必要がある。多くの場合は、特定の命令実行、特定のレジスタアクセス、同じメモリからの読み出しといった操作の組み合わせで実現される。

また、この種の問題はロック変数と共有データを同じメモリに配置すると解消することが多いが、前述したメモリ・バンクや特殊なメモリ領域などもあるため、十分な確認が必要となる。

6.3.8　マルチコア用ハードウェア機能の有用性

本章で述べた各ハードウェアに機能について、6.2 章のユースケース毎に有用性と SHIM での対応状況を表 6-1 にまとめた。

機能	負荷分散型	機能統合型	機能分離型 時間分離型	SHIM 対応状況
対称型マルチコア	Excellent	Good	Good	Yes
非対称型マルチコア	Poor	Fair	Fair	Yes
TCM	Good	Good	Good	Yes
等距離共有メモリ	Excellent	Fair	Fair	Yes

非等距離共有メモリ	Poor	Good	Good	Yes
クラスタ構造	Poor	Excellent	Excellent	Yes
メモリ・バンク	Poor	Good	Good	Yes
メモリ・サブバンク	Good	Good	Good	Yes [注]
プロセッサ間割込み	Fair	Good	Good	Yes
プロセッサ間イベント・フラグ	Good	Good	Good	Yes
排他制御用レジスタ	Excellent	Fair	Fair	Yes
同期化処理	Excellent	Excellent	Excellent	No

注: 間接的にはアクセス時間のパラメータに反映される

表 6-1 各ハードウェア機能のユースケースごとの有用性

負荷分散型はあるタスクを分割した際の実行レイテンシが最も重要となる。また、元は密接な関わりのある一機能を分割するため通信が頻繁に発生するため、特に低レイテンシな通信が必要である。処理分割の際にどのような分割方法がいいかを入念に検討する必要があり、ソフトウェアの配置が容易になる対称性も重要となる。

機能統合、機能分離・時間分離では、処理間の影響を低減する仕組みが有用である。通信頻度は低めで、機能毎に分けるためそれぞれの処理量はまちまちであるため、事前に処理量が推定できる場合は、非対称型マルチコアが有効に活用できる局面も多い。

また、ハードウェアがこれらの機能を備えているかどうかについては、組込みマルチコアコンソーシアムが提案した SHIM 仕様に基づいて確認が可能である。メモリ・サブバンクに関しては機能的には直接表現されず、アクセス性能の数値に反映されて表現する形となる。

この中で唯一 SHIM により表現できていないのは、同期化処理についてであるが、これは同期化処理の対象範囲や手順などが画一的に取り扱えず、標準的な仕様で表現しにくいことが原因である。しかしながら、マルチコアを扱う上で重要な要素となるため、対応方法を継続して検討する。

6.4 車載制御向けマルチコアの実例

本章では実際のマルチコア・デバイスの仕様として、ここまで説明したハードウェア機能がどのような形で

搭載されており、またその意図はどのようなところにあるか、ルネサス エレクトロニクス社のデバイスを例にあげて紹介する。

6.4.1 製品: Renesas RH850 シリーズのマルチコア機能

ルネサス エレクトロニクス社製の RH850 シリーズは、車載システム向けの高性能・高信頼性を両立したマルチコア・マイクロコントローラを発売している。

- RH850 シリーズ : RH850/E2x-FCC2
- https://www.renesas.com/jp/ja/about/press-center/news/2018/news20180327.html
- https://monoist.atmarkit.co.jp/mn/articles/1803/28/news039.html

6.3 章で説明したハードウェア機能の RH850/E2x-FCC2 での対応状況は、表 6-2 のようになっている。

マルチコア機能	**対応状況**
対称型マルチコア	Yes (6 コア:同一性能コア)
非対称型マルチコア	No
TCM	Yes
等距離共有メモリ	Yes (クラスタ内)
非等距離共有メモリ	Yes (クラスタ間)
クラスタ構造	Yes (3 クラスタ)
メモリ・バンク	Yes
メモリ・サブバンク	Yes
プロセッサ間割込み	Yes
プロセッサ間イベント・フラグ	Yes 注
排他制御用レジスタ	Yes
同期化処理	Yes

注: プロセッサ間割込みの要求フラグで代用可

表 6-2 製品: Renesas RH850 シリーズのマルチコア機能

RH850 は車載制御システムの特性に合わせて、高性能・高信頼を目的としたマイクロコントローラ(MCU)である。6.2 章で挙げた 3 つのユースケースのいずれにも対応できるクラスタ構成の対称型マルチコアを採用している。

- ✓ 車両内の設置場所の制約から複数の機能を統合する要求がある: 機能統合型
- ✓ 車載安全基準(ASIL)へ対応したソフトウェア構成が必須: 機能分離・時間分離
- ✓ 制御処理の高性能化が求められる: 負荷分散

制御処理の高性能化については、制御処理自体はあまり並列性が高くなく、2～3 並列程度の異なる処理を実行することが求められる。一部のデータ並列処理(行列演算など)は、各コアに搭載している SIMD 演算器により、対応するという方針である。

図 6-22 に RH850 マルチコア・デバイスの構成概要とその特長を示す。同一性能の CPU コアを 2 つずつクラスタ化し 3 つのクラスタを備えており、それぞれのコアに TCM(LRAM)、クラスタ毎に共有メモリ(Cluster RAM)を配置している。Cluster RAM は、クラスタ内では等距離メモリとして 2 つのコアでの制御処理の再配置を容易にし、クラスタ間では非等距離メモリとして、機能間の比較的低速なデータ通信に利用する。また、プログラムを供給する命令メモリはクラスタ毎に Flash ROM を個別に備えることでクラスタ間の影響を排除している。

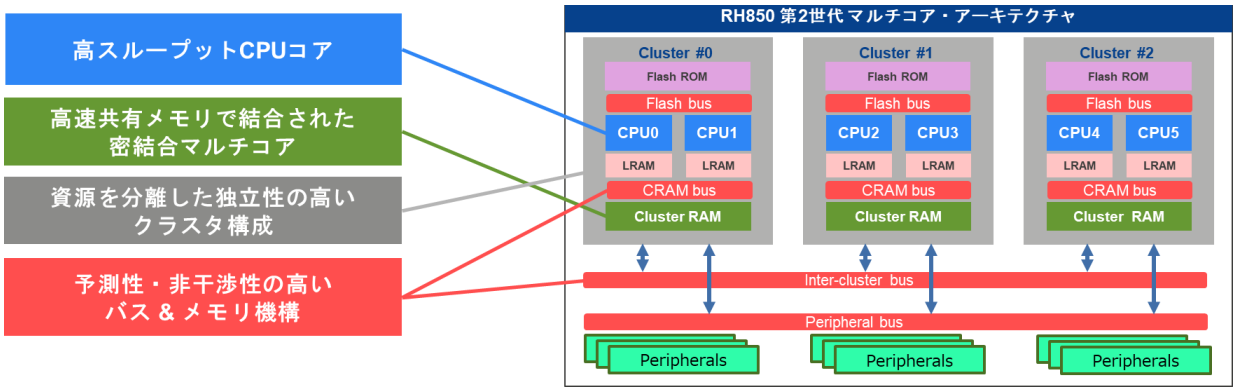


図 6-22 RH850 マルチコア・デバイスの特長

この構成の意図は、負荷分散として単機能を分割する際にはクラスタ内の 2 コアを利用し、複数の機能の統合ではクラスタを分けることで干渉の少ない、安全基準を満たした Freedom from interference の実現が容易に可能であるように考えられたものである。

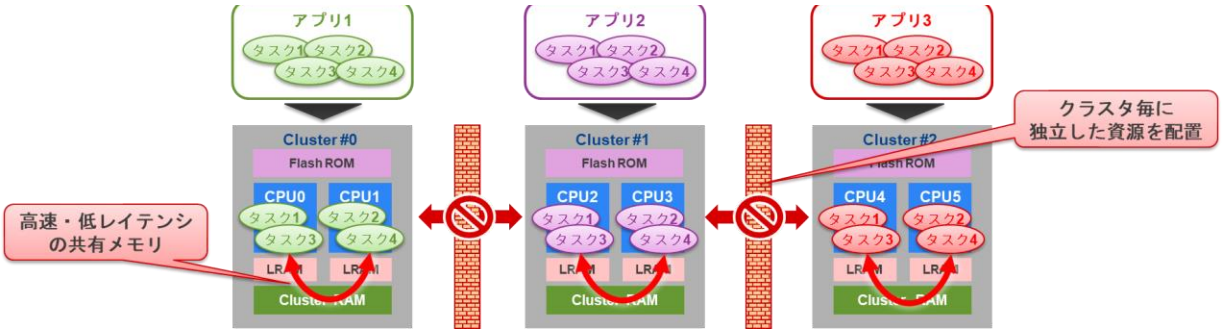


図 6-23 RH850 マルチコアの狙い

6.5 制御系ソフトウェアの並列化とツール

本章では制御系ソフトウェアをマルチコアに適用するに当たり、並列化のための考え方と、並列化を支援するツールについて簡単に説明する。

6.5.1　制御系ソフトウェア並列化の粒度

制御ソフトウェアの並列化を行う場合、その並列性をどのような箇所から見つけ出すかによって、並列化ソフトウェアの性質が異なってくる。ここでは、主にソフトウェアを分割する際に着目する粒度について解説する。

一般的な制御ソフトウェアは、より大きな粒度から考えると次の 3 つのレベルで整理することができる。

- ✓ アプリケーション・レベル(機能レベル)
- ✓ タスク・レベル(スレッド・レベル)
- ✓ コード・レベル

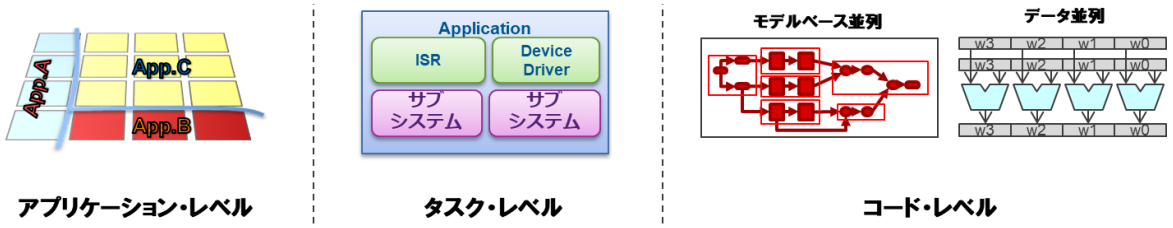


図 6-24 並列化の粒度

これらのレベル毎に並列化で取り扱う単位が異なり、それぞれに必要な考え方や支援ツールが異なってくる。また、ユースケースごとに当てはめやすい方式が異なるため、システムの目的に合わせて適切に選択する必要がある。表 6-3 にユースケースとそれぞれの並列化粒度の適性についてまとめた。

並列化の粒度	負荷分散	機能統合	機能分離 時間分離
アプリケーション・レベル	No	Yes	Yes
タスク・レベル	Yes	No	Yes

コード・レベル	Yes	No	No

表 6-3 並列化の粒度とユースケース

各節ではそれぞれの粒度で並列化を行う際の考え方、特長、注意点等を述べる。

6.5.1.1 アプリケーション・レベル(機能レベル)

アプリケーション・レベルは設計者が意図する大きな単位での機能を構成するソフトウェア群をひとかたまりとして、複数の機能を並列に配置する考え方となる。すなわち、ソフトウェアが持つ誰が見ても明らかな意味的な並列性である機能／目的の違いを基準に並列に配置する。

最も単純な形としては、1 つのコアを 1 つの機能に割り当てることである。この場合、ソフトウェアの配置は非常に容易だが、一方で機能ごとに負荷が不均等であるケースが多く、コアによっては性能が不足したり、逆に余剰が大きく遊んでしまう可能性がある。このため、利用効率をあげることが難しい。

ただし設計自体は単純で、設計者の意図を反映しながら、OS やハイパーバイザなどのアプリケーション管理機構を持ったプラットフォームに乗せていくことで、比較的容易に実現できる。

- 必要となる支援ツール
  - ✓ ハイパーバイザ (仮想化)
  - ✓ マルチタスク OS

6.5.1.2 タスク・レベル(スレッド・レベル)

リアルタイム OS のタスクにあたる粒度でソフトウェアを分割する方式である。タスクは小規模な機能であれば、タスクひとつがそのまま単一の機能となる場合もあるが、機能内の役割に応じて複数のタスクに分割する場合もある。これらのタスクは動作のタイミングが異なったり、概念的には同時にアクティブとなるような性質のものが存在する。タスク・レベルの並列化とは、それらの同時に実行可能なタスクに着目して並列に動作させる考え方である。

制御系ソフトウェアではある機能を実現するのに、複数の入出力装置を取り扱い、またそれと並行して制御方針を定める定周期のメイン処理が存在するような構成が取られることが多い。この時、それぞれ入出力装置の都合に合わせて起動されるプログラム(機器からの割込みや、サンプリング周波数に依存した定周期で起動)と、常に一定周期で実行されるメイン処理とを並列に動作させることで、全体の処理能力の向上が図

れる。

OS を利用するソフトウェアであれば、事前にある程度タスクへの分割はなされていることから並列化自身は容易であるが、動作タイミングを含めた関係性を考慮して、効果的な配置を行うことは難しい。また、タスク間に処理順序が必要な場合には、適切な手法で排他制御を構築する必要があり、その結果としてタスク間の動作関係が変動するなど、実際に動作させてみないと最終性能が予測しにくい。

このように動作関係の定義・検証や、並列化の効果測定のために、タスク配置支援ツールや解析系ツールが必要となる。

- 必要となる支援ツール
- マルチタスク OS
- タスク配置支援ツール
- タスク動作解析ツール

6.5.1.3 コード・レベル

ソフトウェアのコードに記述された処理構造から並列化を行う方式である。例えば、関数や基本ブロックに相当する一連の処理群、またはループ構造からくるデータ並列性などを活用する。

コードから並列性を抽出するために、変数などの利用関係を明確にする必要があり、コードの詳細解析が必要となる。また、組込みソフトウェアなどでコンパクトな実装を目指した結果、変数などが再利用される場合、本来の処理構造としては依存関係がなかったにも関わらず、文脈による依存が発生してしまい、並列実行を阻害するケースが生じる場合がある。

またデータ並列については、ループの添字毎に依存がないことを利用する必要があるが、制御処理は通常、配列の添字を連続的に処理していくようなわかりやすいデータ並列性が少ないため、不適であることが多い。

このように既存のコードから並列性を抽出するのは、特殊なスキルや理解力が必要となり、手動で実施するのはコストが高い。このため、並列化コンパイラなどのコード解析と自動並列化を備えた支援ツールなどの利用が望ましい。

また、コードを出力する前段階の制御処理の論理構造を扱うようなモデリング言語をベースに並列化することも一つの方法である。これは前述の文脈による依存関係が存在しない、純粋なデータ依存のみが表現されている可能性が高いからである・

- 必要となる支援ツール
  - ✓ 並列化コンパイラ
  - ✓ コード解析ツール
  - ✓ モデルベース並列化ツール

6.5.2 マルチコア支援ツール

6.5.1 で説明した並列化手法を実現するために利用できるマルチコア支援ツールについて列挙する。本章では各ツールの詳細については割愛する。

- タスク・レベル設計支援
  - ✓ TA Tool Suite (Timing Architect)
  - ✓ T1 (GLIWA)
  - ✓ SymTA/S (Symta vision)
  - ✓ Task Analyzer (Renesas Electronics)

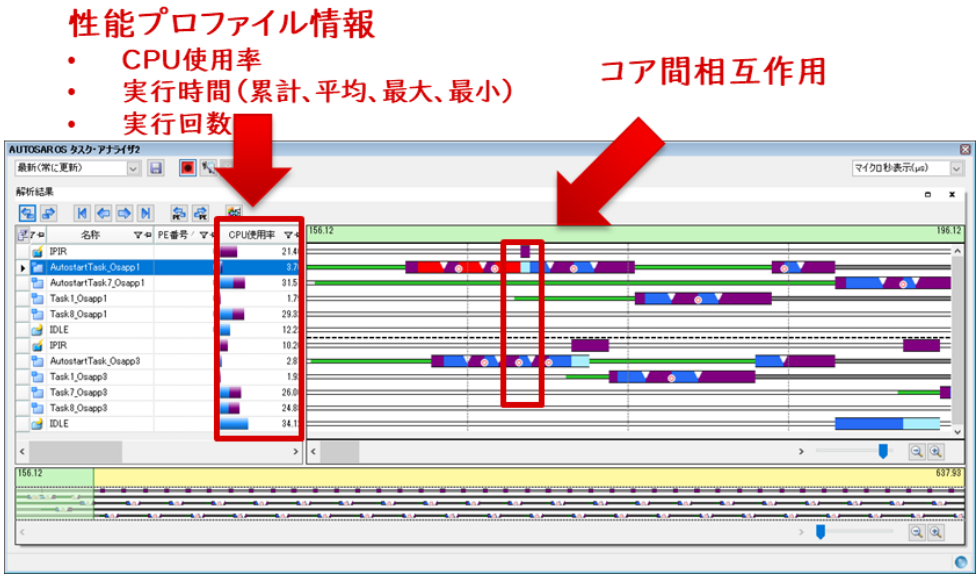


図 6-25 Task Analyzer

- 並列コンパイラ
  - ✓ OSCARTech®Compiler (Oscar Technology)

✓ SLX Tool Suite (Silexica)

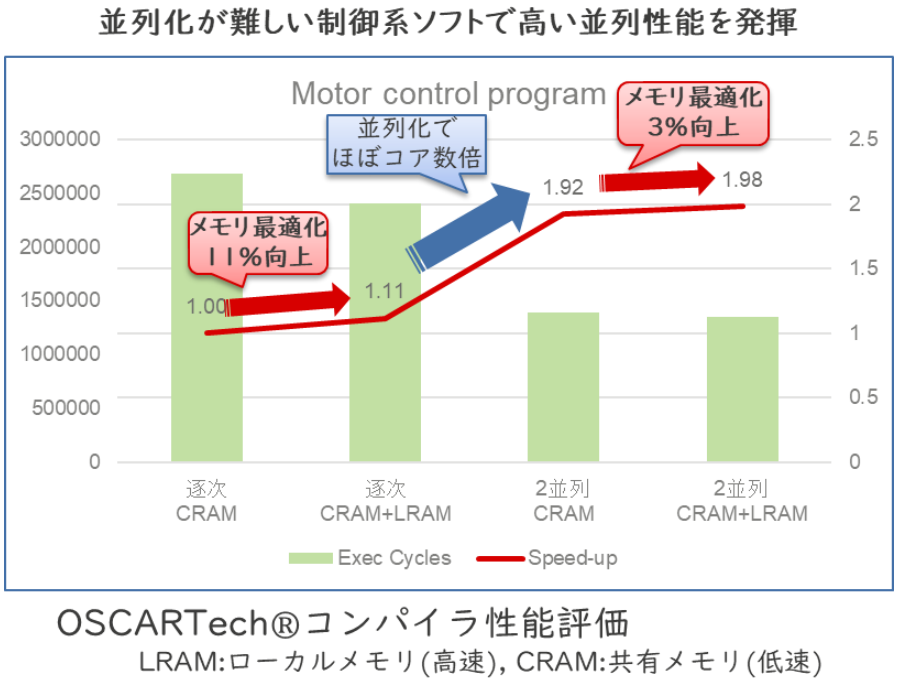


図 6-26 OSCARTech® Compiler

➢ モデルベース並列化

✓ eMBP (eSOL)
✓ Embedded Target for RH850 Multicore(Renesas)

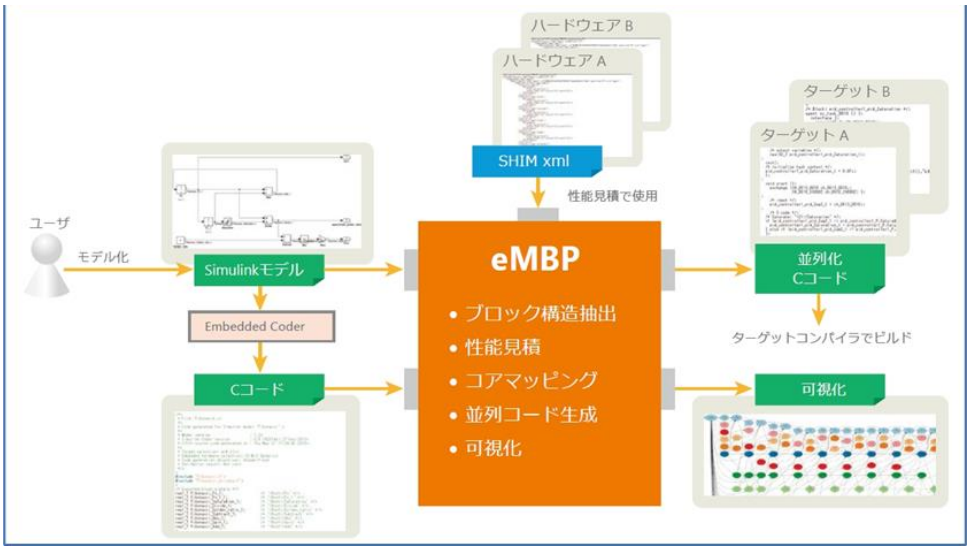


図 6-27 eMBP

6.6 その他の制御系マルチコア技術

6.6.1　冗長構成(ロックステップ)

マルチコアの一種としてロックステップ構成の MPU が存在する。ロックステップ構成のマルチコアは、複数のコアに同じ計算をさせ結果を比較し、片方が誤った場合を検出し、異常発生を速やかに検出するため、特に高信頼が必要な分野で用いられる。

図 6-28 では 2 つのコアは同時、あるいは数クロックずらした状態で同一の入力を用いて動作する。したがって、出力は完全に一致するはずであり、相違がある場合はどちらかが動作を誤った状態であり、システムは速やかに安全な状態へと制御する必要が生じる。通常、演算器を二重化し、メモリ等は ECC などで冗長性を高めることで高信頼を担保する。

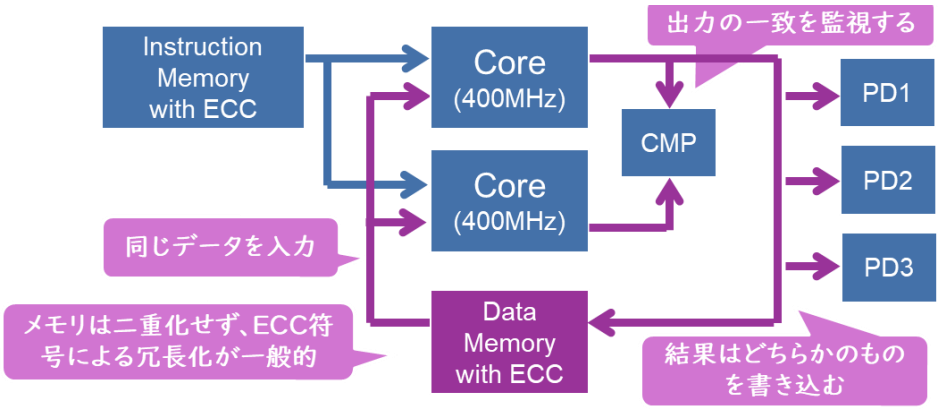


図 6-28 ロックステップ構成

このように同一動作を行わせることで、速やかに故障を検出できるが、2 コア構成の場合は誤りがあったかどうかのみ検出可能であり、誤りを訂正して継続して実行することはできない。これは 2 つの CPU のいずれかが故障はしているが、どちらが故障しているかまでは区別がつかないためである。故障後も継続実行を行うには、3 つ以上の CPU を動作させる必要がある。

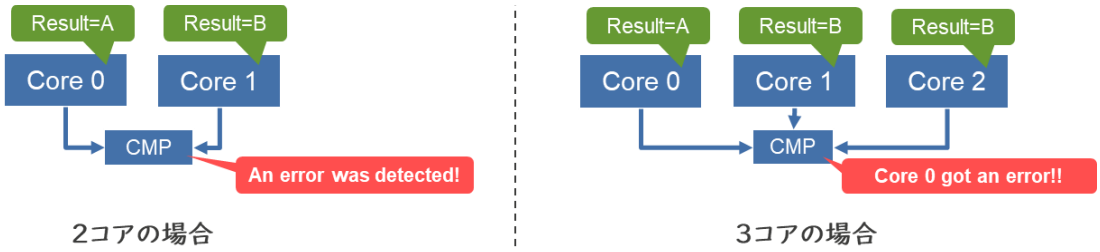


図 6-29 2 コア・ロックステップと 3 コア・ロックステップ

ロックステップ構成は非常にコストが高いため、通常のマルチコアとしても使えるモードとロックステップ動作を行うモードを実行時に切替え可能な MPU も存在する。

また、ロックステップ構成はソフトウェアから見た場合、単一のコアとしてしか見えないため、プログラミング上はマルチコアとして扱うものではない。

6.7 今後の制御マルチコアと EMC の活動

本ドキュメントで説明したマルチコア・ハードウェアについては、今後ますます採用するデバイス・ベンダが増えていくことが予想される。短期的にも、下記のような観点でマルチコアへのニーズが高いと感じられる。

- ✓ エッジ・コンピューティングへの Deep Learning の適用
- ✓ 制御アルゴリズムの非線形化と最適化アルゴリズムの採用
- ✓ 車載機器のセントラル・コンピューティング化(機能統合)

これに伴って従来の設計に加えて、並列化への対応が求められるようになり、ソフトウェア設計はより難易度が上がっていくであろう。しかしながら、マルチコアを取り巻くツール環境や情報整備はまだまだ不十分である。組込み向けデバイスのユーザーズマニュアルを見ても、ソフトウェア設計の観点から見て、マルチコアとして必要な情報は綺麗に整備されておらず、ハードウェア・ソフトウェアの双方に造詣の深い技術者が数千ページあるマニュアルの様々な箇所から、必要な情報の断片をピックアップせねばならない。また、ソフトウェアの並列化も自動並列化ツールや、並列化した結果の確認、検証などを設計の上流で簡易に行えるようなツールが普及しているとは言えない。

このような局面において、

- ✓ マルチコア・ハードウェア仕様を正確に把握する仕組み
- ✓ マルチコア設計手法の一般化、ツール支援

がますます重要になるであろう。

組込みマルチコアコンソーシアムでは、この課題に対して図 6-30 に示す 3 つの委員会を以て対策を講じ、広くオープンに周知することで、日本におけるマルチコア活用の推進を行っていく計画である。

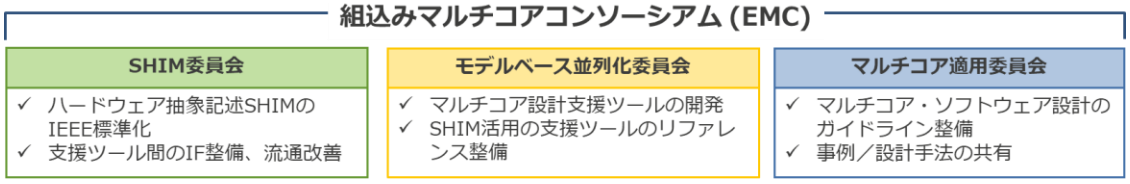


図 6-30 組込みマルチコアコンソーシアムの活動

7　自動車 機能安全へのマルチコア適用

7.1 はじめに

安全性が欠かせない自動車の組込み機器では、コンピューティングパワーのニーズが急速に高まり、複雑さが増している。自動車の安全に関する国際規格においても、半導体に関する重要性が増しており、 2018 年 12 月に公開された機能安全規格の 2nd Edition である ISO 26262:2018 では、半導体に関する章が新設されている。本稿では、 ISO 26262:2018 で記載されているマルチコアプロセッサを設計に用いる際の規格の要求事項について説明する。また、組み込み開発者がマルチコアを新規に採用する際に直面する課題についても説明する。

7.2 テクノロジーの進化

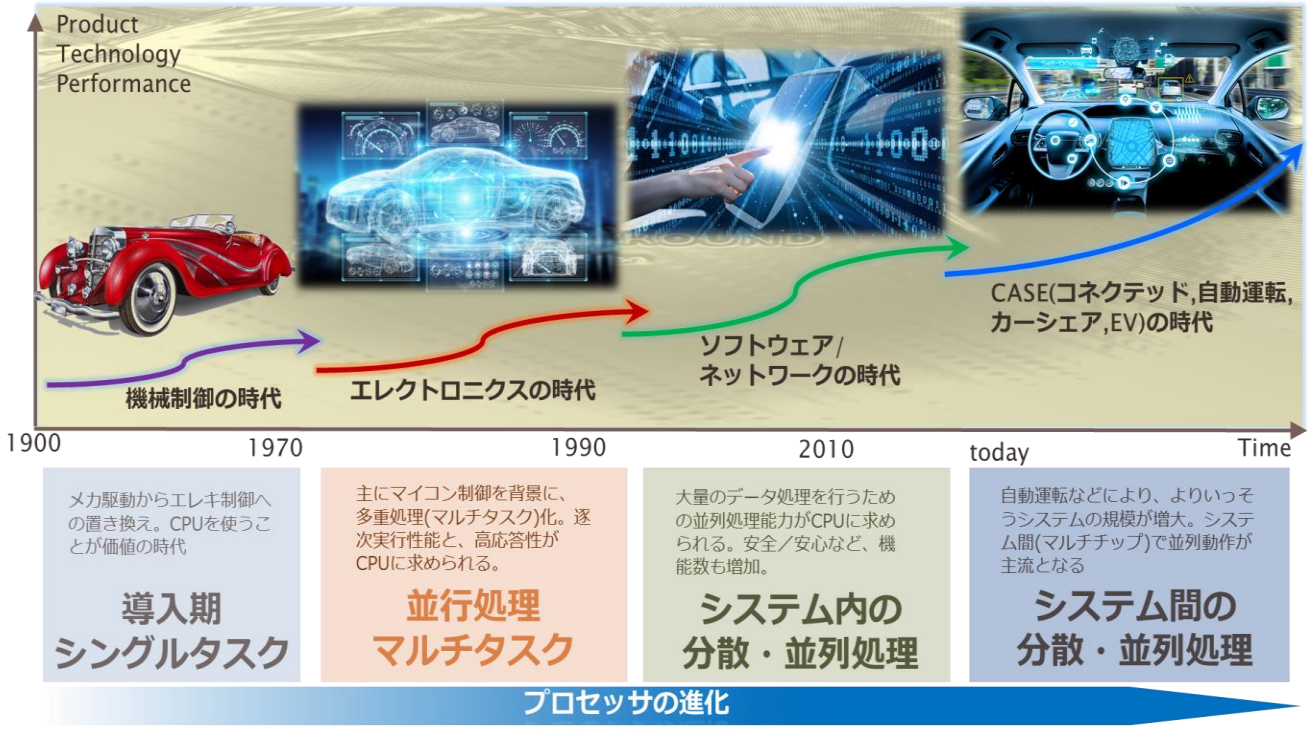


図 7-1 テクノロジーの進化

7.3 マルチコアテクノロジの必要性

あるシステムでは、マルチコアプロセッサを使用して必要な目標を達成するのは自然な解決策であるが、他の方法で効率的に実現できることもある。 一方で、マルチコアプロセッサは過剰品質のように見えることもあるかもしれないが、良い設計の選択肢にもなり得る。

このように、自動車のシステム開発の課題に取り組む前に、マルチコアが本当に必要かどうかという疑問に対処することはプロダクト開発において非常に重要である。

ここでは、マルチコアの必要性を 3 つに分類して扱う。

- 安全性の向上
- パフォーマンスの向上
- 効率の向上

どのような機能を達成したいのか？なぜマルチコア技術がこのシステムに役立つのか？

マルチコア採用時はこれらの疑問に対する回答を明確にすることが必要である。

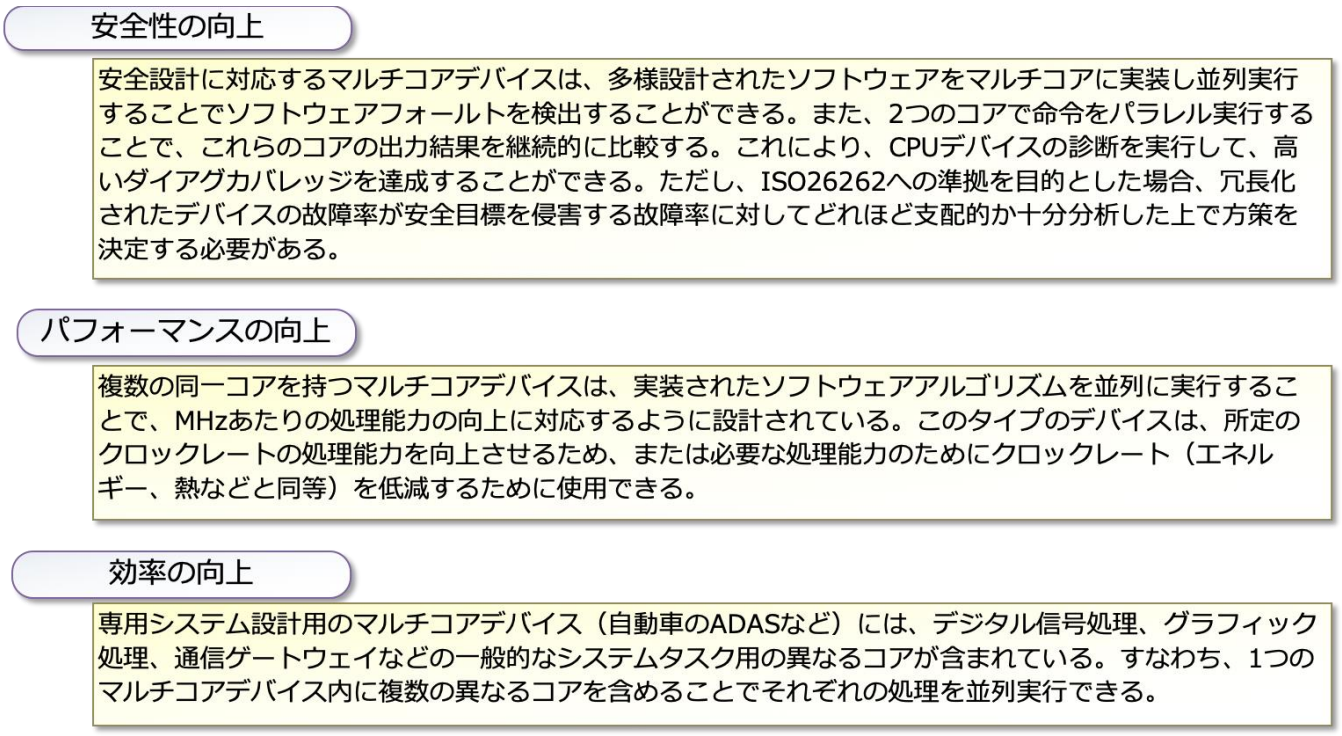


図 7-2 マルチコアテクノロジの必要性

7.4 機能安全規格 ISO 26262 2nd Edition

- Guidelines on application of ISO26262 to semiconductors
  - **自動車用途に使用される半導体のガイドラインを提供する**
  - **半導体コンポーネントの開発**
    - ✧ 半導体コンポーネントはエレメントの 1 つとして安全分析を行う
  - **半導体コンポーネントに関するガイドライン**
  - **半導体技術に関するガイドライン**

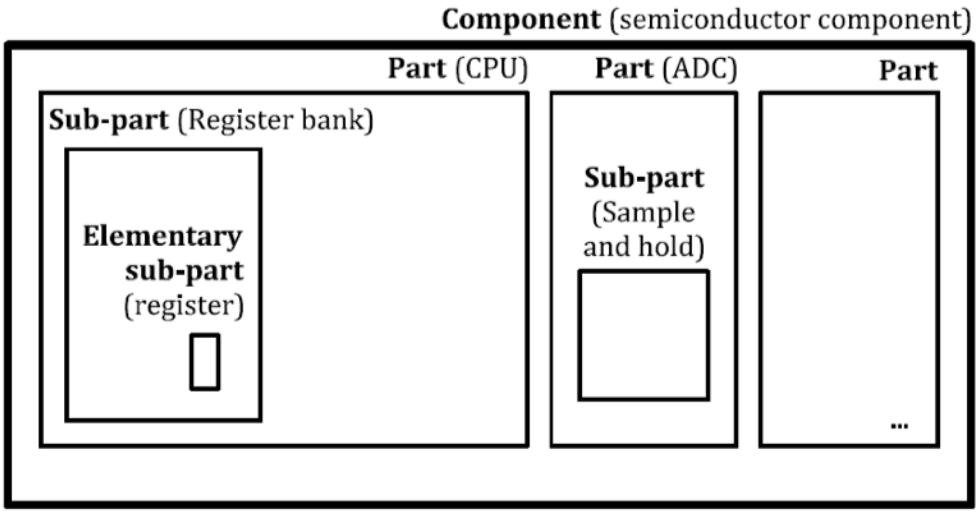


Figure 2 — A semiconductor, its parts and subparts

図 7-3 Guidelines on application of ISO26262 to semiconductors

引用：ISO26262-11:2018

本節では ISO 26262:2018 で記載されているマルチコアを設計に用いる際の規格の要求事項について説明する。

7.4.1 ISO 26262 : 2018 Part6 (ソフトウェア開発) clause5

モデリングガイドラインとコーディングガイドラインでカバーすべき内容にマルチコアに搭載したソフトウェアの同時処理が追加された。

Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1i	Representation of concurrency aspects [f]	+	+	+	+
[f] Concurrency of processes or tasks may be a topic when executing software in a multi-core or multi-processor runtime environment					

引用：ISO26262-6:2018

7.4.2 ISO 26262 : 2018 Part11（半導体） clause5

複数のコアに安全要求をデコンポジションして割り当てた場合、当然妥当性説明必要になる。

すなわち、コア間の従属故障と共通原因故障がないことを安全分析結果から示すことになる。

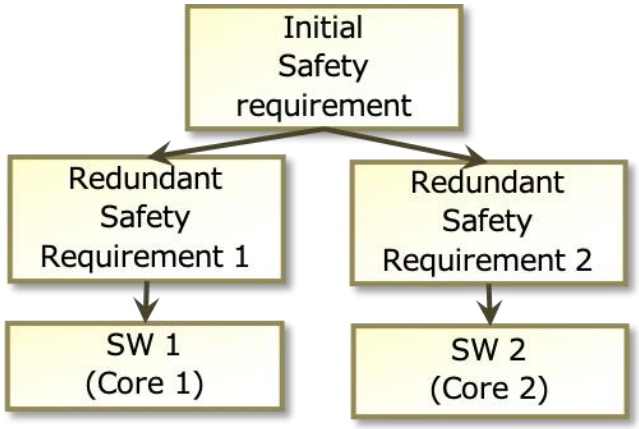


図 7-4 ASIL decomposition in the context of multi-core

引用：ISO26262-11:2018

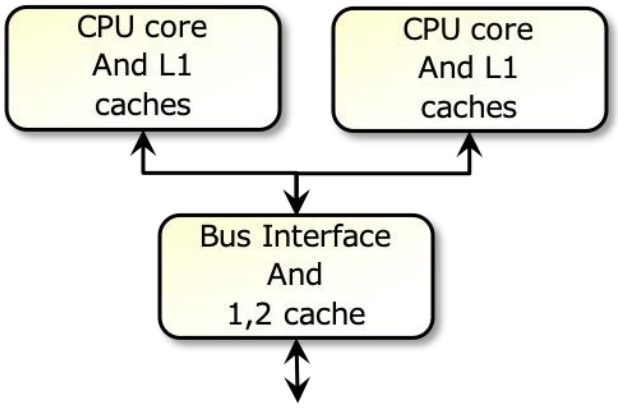


図 7-5 Generic diagram of a dual-core system

引用：ISO26262-11:2018

7.4.3 ISO 26262:2018 Part11 clause5

自動車の製品開発プロジェクトでは、異なるソフトウェアモジュールを 1 つの制御ユニットに組み合わせてコストを節約することがある。このような開発状況でマルチコアプロセッサを利用する場合、コア間でセーフティクリティカルなモジュールへの潜在的な干渉が無いことを論証する必要がある。

マルチコアに搭載したソフトウェアに対して、無干渉の妥当性説明が追加された。

すなわち、ソフトウェアの従属故障分析が必要となる。

5.4.2.3 Clarifications on Freedom from interference (FFI) in multi-core components
If in a multi-core context multiple software elements with different ASIL ratings coexist, a freedom from interference analysis according to ISO 26262-9:2018, Clause 6 is carried out.

引用：ISO26262-11:2018

仮想技術の有効性も記載されている。

7.4.4 ISO26262:2018 Part11 clause5

マルチコアに搭載したソフトウェアコンポーネントのタイミング故障に関する分析と適切な方策が要求されている。

5.4.2.4 Timing requirements in multi-core component
Multi-cores are potentially subject to timing faults; therefore the previous listed clauses are carefully considered with dedicated analyses and adequate countermeasures identified.
EXAMPLE 1 Typical dedicated analyses for the identification of timing faults potentially violating the safety goal are based on the upper estimation of execution time .
EXAMPLE 2 Typical hardware-based countermeasures for detection of violation of timing requirements are watchdogs, timing supervision units and specific hardware circuits. Software-based countermeasures are also possible.

引用：ISO26262-11:2018

7.4.5　ソフトウェアコンポーネント間の非干渉 IS0 26262

- エレメント共存の法則
  - ソフトウェアメカニズムで FFI を保証する場合は、共存させる ASIL の中で高い方の ASIL で実装される。

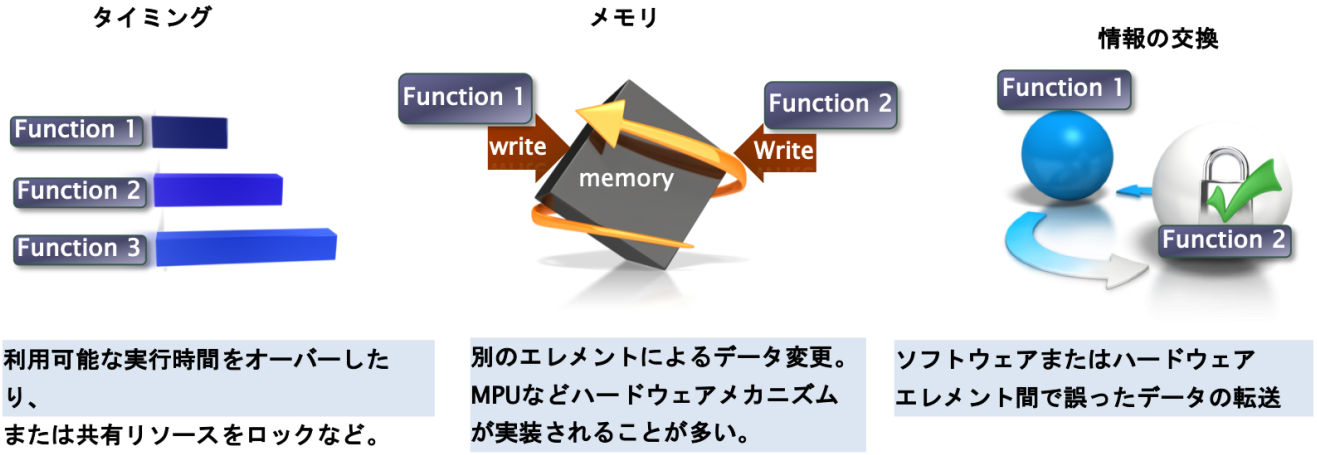


図 7-6 ソフトウェアコンポーネント間の非干渉 IS0 26262

7.4.6　2nd Edition の追加トピックスから読取れる主な機能安全設計課題

- 安全要求を満たすと同時に、並行性の効果を最大化する
- 並行処理に適切な設計法及び表記法を用いる
- 並行処理特有のフォールトに対し従属故障分析を行い、コア間の非干渉を達成する
- 主機能の可用性を継続しながら、いくつかのアプリケーションをシャットダウンすることができるフェイルオペレーショナルな安全アーキテクチャを設計する

7.5 マルチコア適用の機能安全プロセス

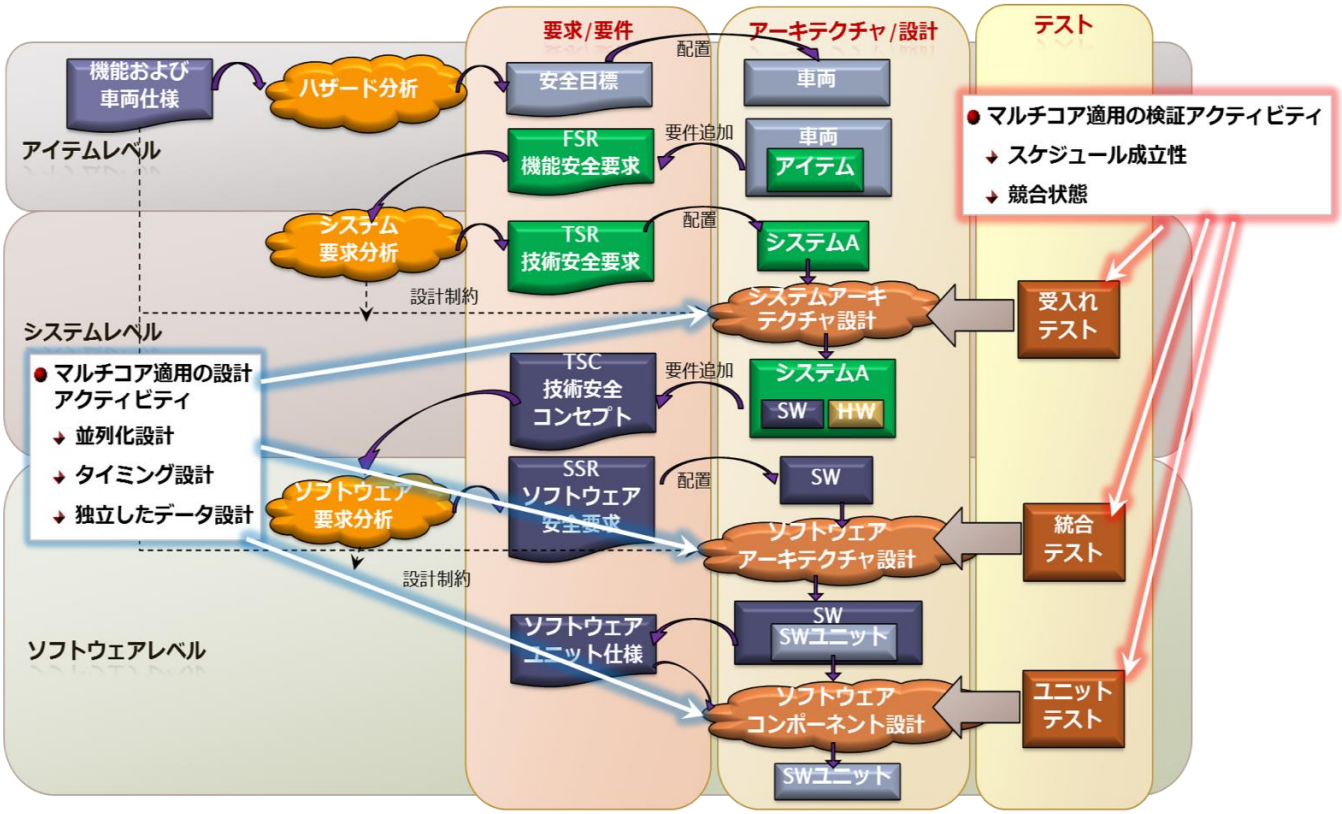


図 7-7 マルチコア適用の機能安全プロセス

7.6 マルチコア適用の機能安全アーキテクチャデコンポジションの例

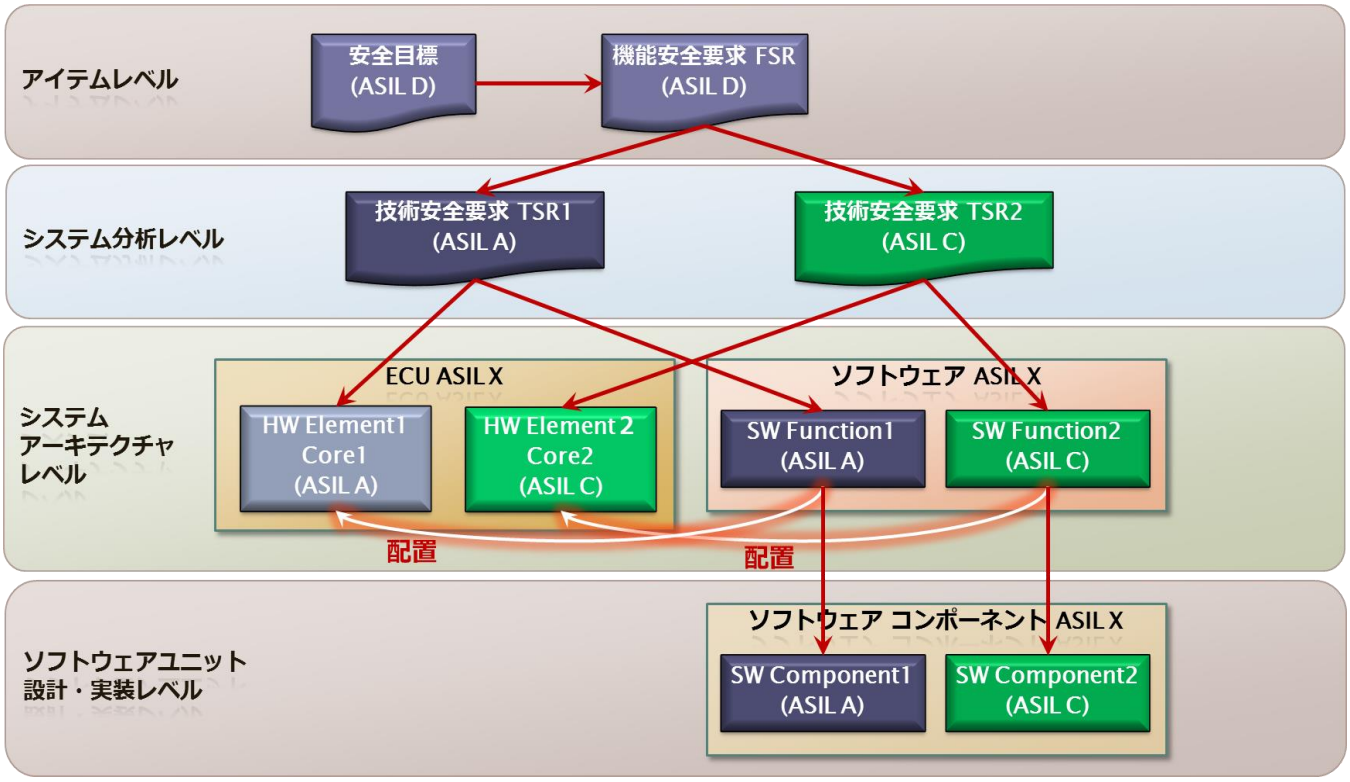


図 7-8 マルチコア適用の機能安全アーキテクチャデコンポジションの例

7.7 並行処理でのソフトウェアフォールトの例

① スケジューリングエラー

② 共有メモリエラー

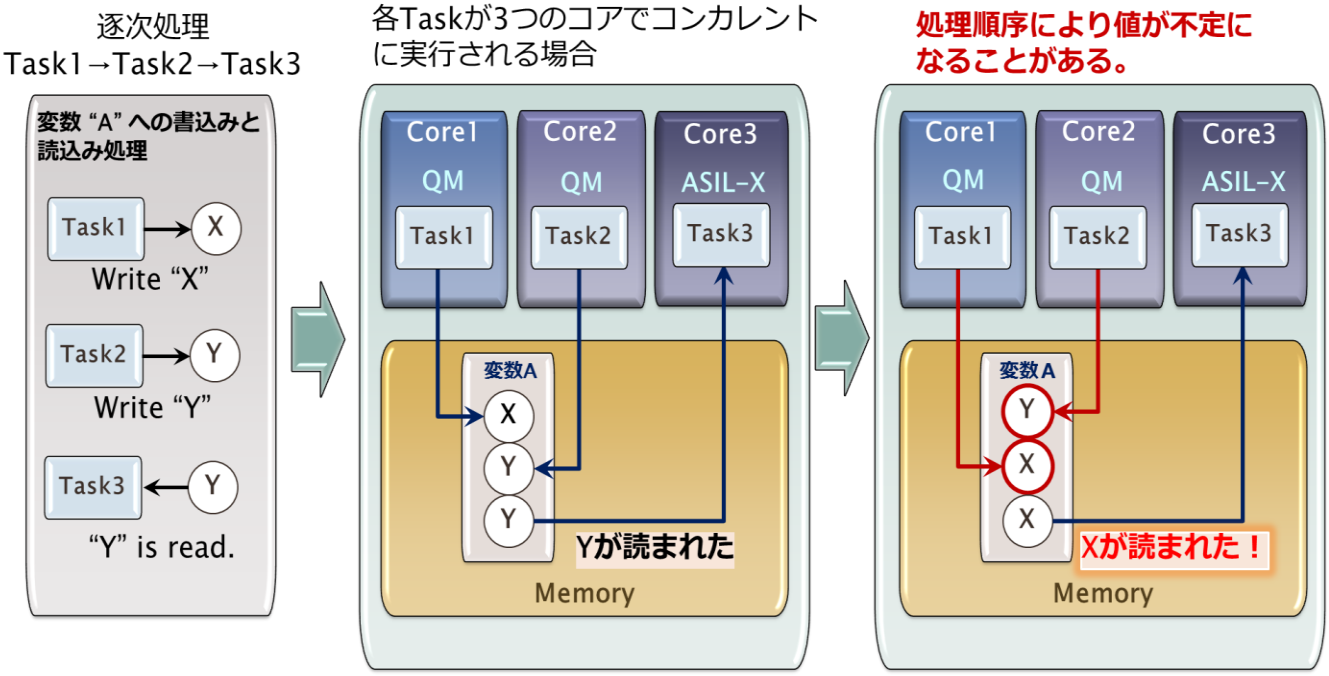


図 7-9 並行処理でのソフトウェアフォールトの例

7.8 ソフトウェアフォールト伝搬の例

④　タイミングエラー

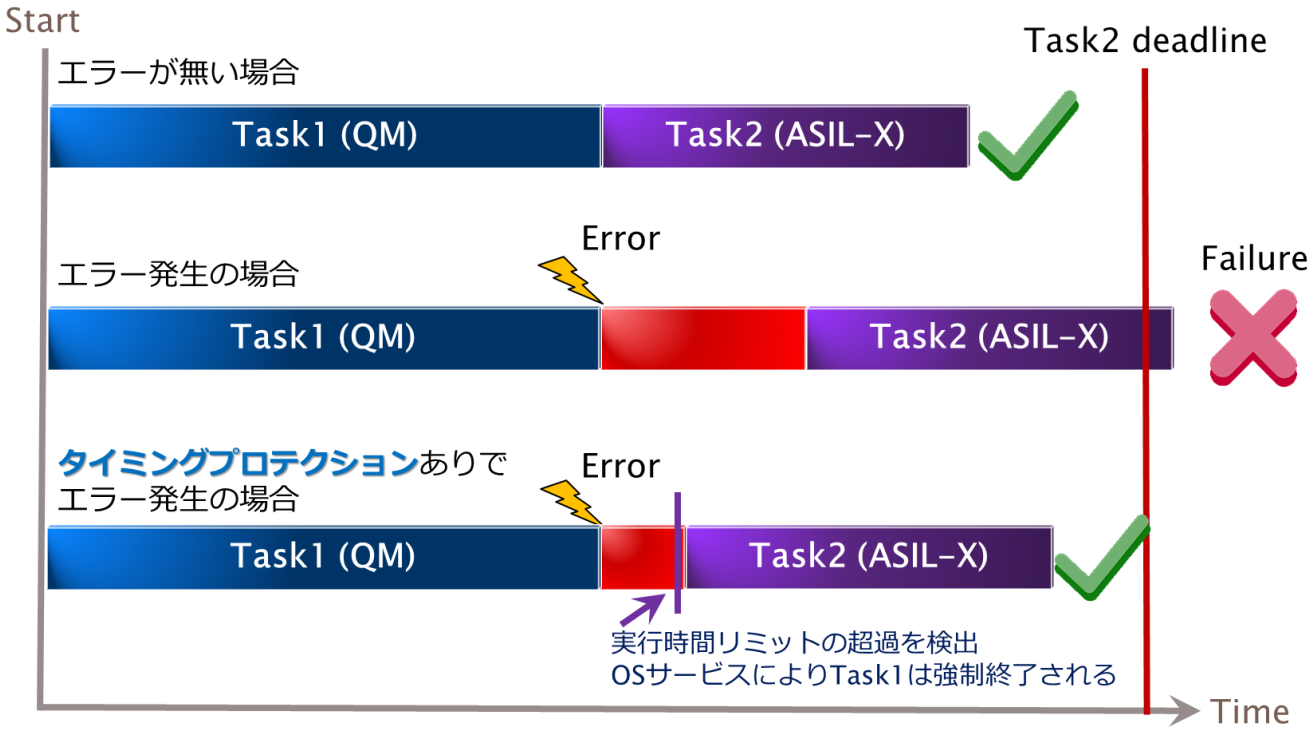


図 7-10 ソフトウェアフォールト伝搬の例

7.9 FFI 対応：Memory Protection による共有メモリアクセス制御

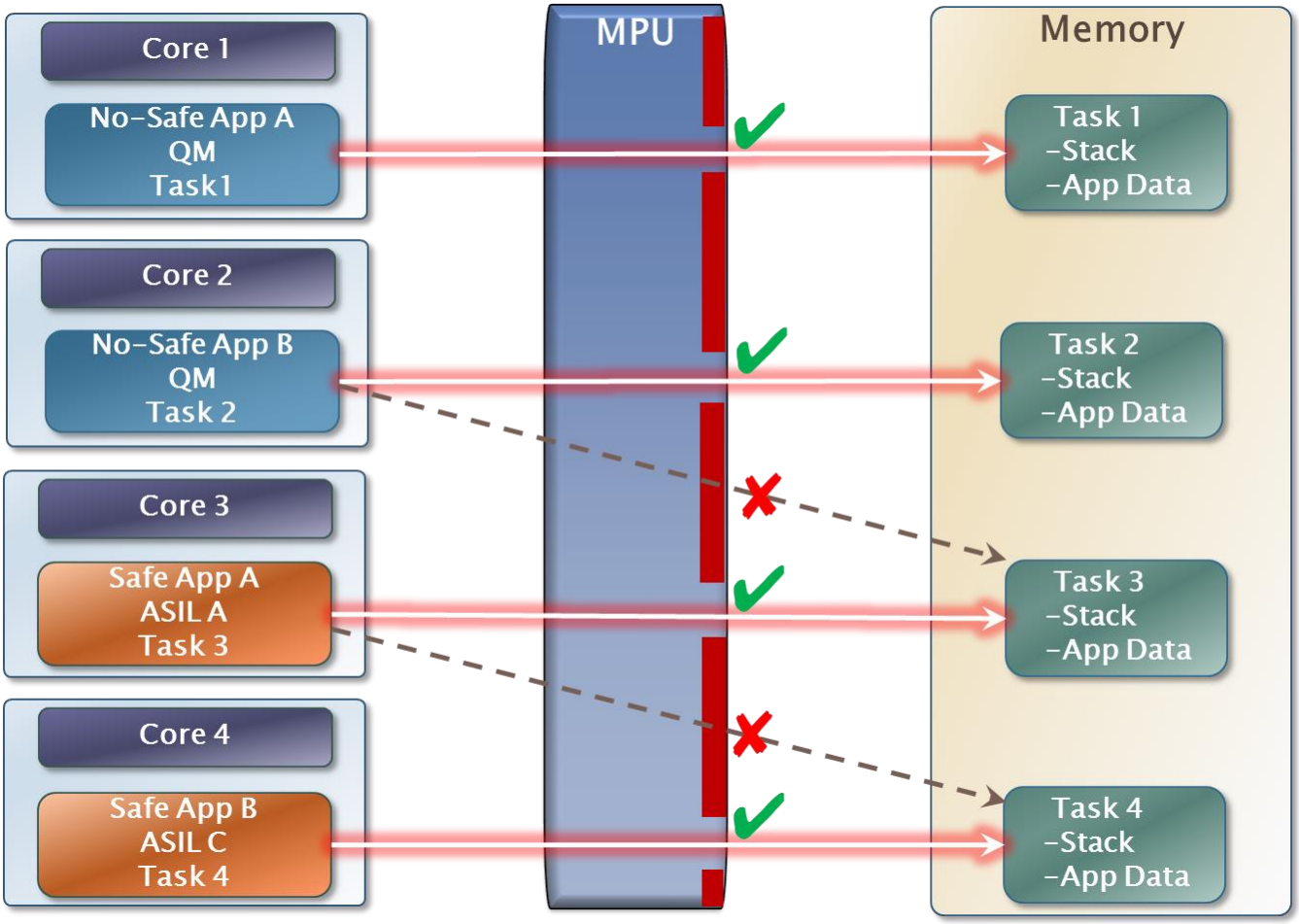


図 7-11 FFI 対応：Memory Protection による共有メモリアクセス制御

7.10　FFI 対応：Program Flow Monitor によるタイミング保護

アプリケーション内のチェックポイントに加え、フローを監視することで正しいシーケンスを保証する。

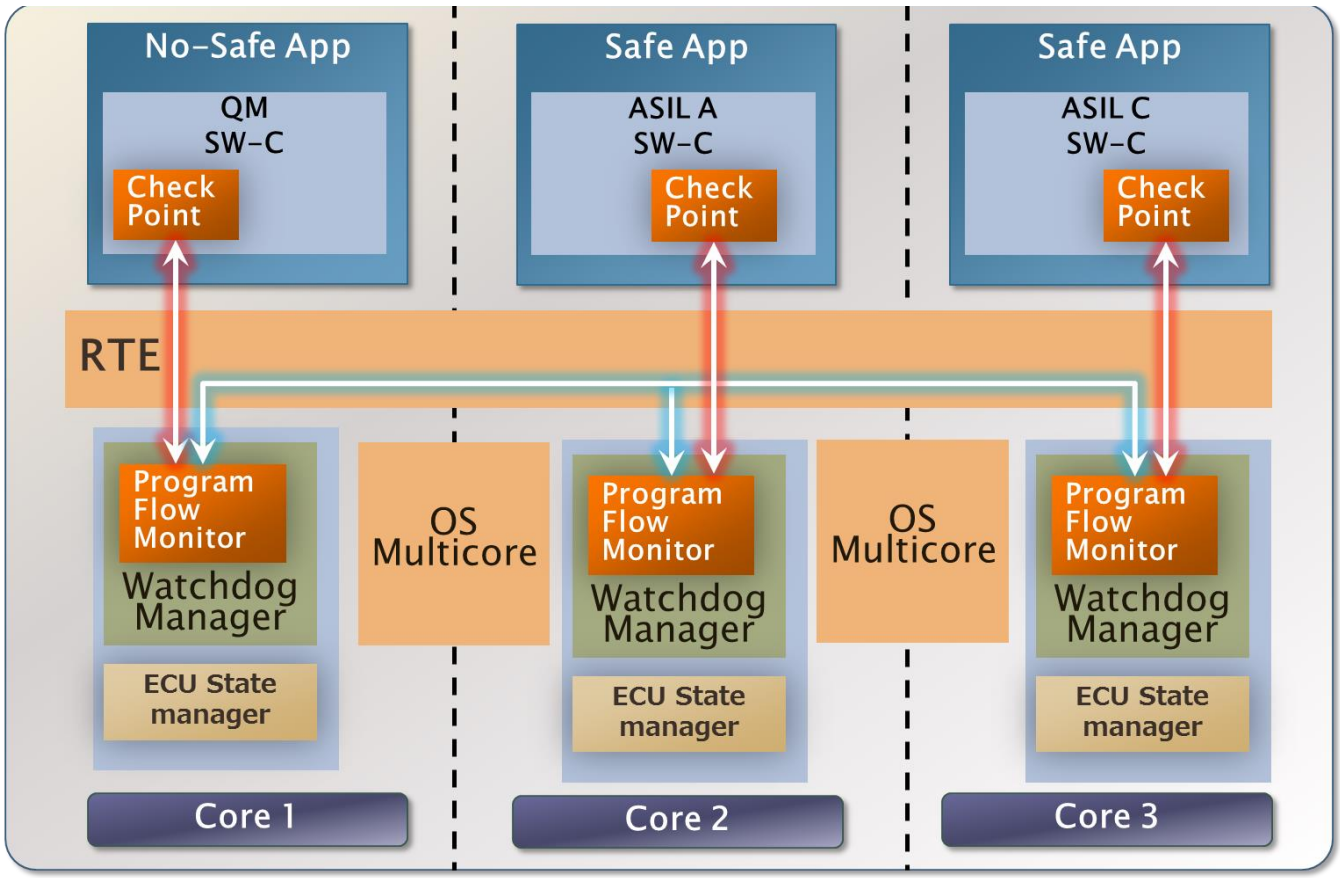


図 7-12 FFI 対応：Program Flow Monitor によるタイミング保護

7.11　分散・並列処理ソフトウェアのデザインパターン

図 7-13 分散・並列処理ソフトウェアのデザインパターン

7.12　まとめ

- コアへの安全要求の配置、ASIL 共存及び非干渉化の効率的な実現がマルチコア適用のカギ
- マルチコアを安全設計で使用する場合、分散・並列処理アーキテクチャ設計技術が必要となる。
- 分散・並列処理アーキテクチャにはデザインパターンを活用することで効率化と信頼性の確保に貢献できる

8 並列処理ソフトウェアの課題と対策技術

8.1 はじめに

マルチコア上で動作するプログラムは、コアを有効利用するために並行性を持つ様に構成（Thread 化）する。それらのスレッドは別なコア上で動作する場合、物理的に並列動作する。

このようなシステムで、プログラムの並列性を理解して設計する事は次の様な観点から必須となる。

- 計算資源の有効利用
- ソフトウェアの並列性に起因する不具合の回避

8.2 並列プログラムの課題

ソフトウェアを、マルチコアで動作させること、もしくはマルチコアで動作するように変換することを「並列化」と呼ぶ。最初に、ソフトウェアを並列化するための手法や開発プロセスの課題について説明する。ソフトウェアの並列動作可能な計算資源へのマッピングとして、計算資源の有効利用が課題とされている。また、並列プログラムは以下のような挙動理解が困難である。

8.2.1　非決定性

並列プログラムの挙動には「非決定性」を含む場合がある。非決定性を持つプログラムでは、状態遷移の分岐が排他的でない。これはテストプログラムや不具合事象の再現性が悪く、デバッグしにくい。

8.2.2　複雑性

並列に動作する別別のスレッドが別々の状態遷移を持つため、システムの状態はその組み合わせになり、複雑で、膨大な振る舞い空間をもつ。これはシステム挙動の網羅的な検証が困難である。

8.3 問題のある振る舞い

マルチコア上で動作する並列プログラムの挙動には非決定性が含まれる。

8.3.1　振る舞いの非決定性

プロセス PA と PB が並行に動作する場合、「可能な挙動」は組み合わせが膨大になりうる。

```
PA = ea1->ea2->ea3->Stop;
PB = eb1->eb2->eb3->Stop;
```

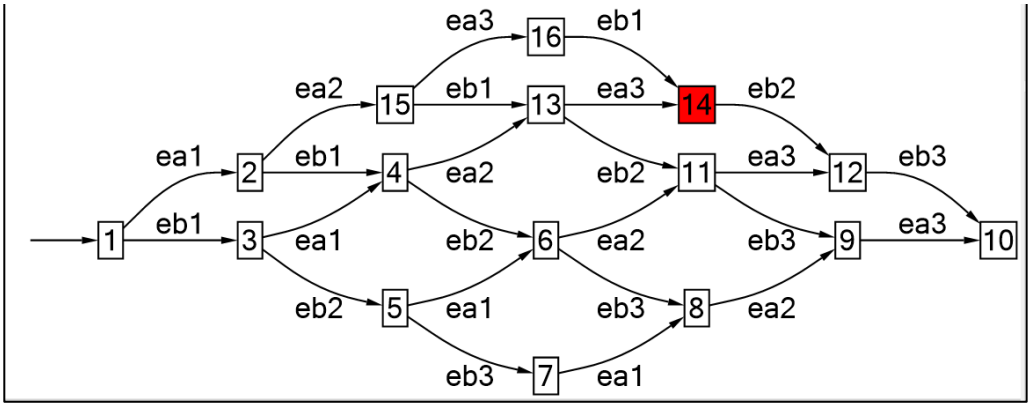


図 8-1 並列プログラムの振る舞い

8.3.2　データ破壊・不整合

複数スレッドが共有メモリ領域を読み書きする場合に起きる。並列に動作する複数のスレッドが共有の記憶領域に対して同時に書き込むような場合に、データが適切に更新されず、矛盾・破壊する事がある。これではプログ

ラムの機能（入出力仕様）が保証されない。データ破壊は同時書き込み時に起こる。また、データ不整合は複数のメモリ内の情報の間に成り立っていなければならない関係（条件）が崩れる。

8.3.3 デッドロック

並列に動作する複数のスレッドが共有の記憶域にアクセスするために組み込んだ「排他制御」が想定通りに動作せず、システムが停止してしまう。これではプログラムの可用性が保証されない。また、共有リソースのロックが外れなくなる。

デッドロックが発生する例

タイミング１：

ThreadA がリソース R1 を利用(R1 をロック)

ThreadB がリソース R2 を利用(R2 をロック)

タイミング２：

ThreadA がリソース R2 を使おうとして R1 のアンロックを待つ

ThreadB がリソース R1 を使おうとして R2 のアンロックを待つ

＜＜＜Deadlock＞＞＞

8.4 形式検証

形式検証では、対象システムの挙動をモデル化する。このモデルは、形式仕様記述言語（プログラムの構造を数学的な意味で厳密に記述するために設計された言語）で記述したモデルである。これにより、非決定的なモデルが表現でき、そのモデルに対して網羅的な検証が可能である。

挙動のモデルは「状態遷移」の考え方の数学的な表現。次の２種類が知られている。

8.4.1 オートマトン (A = {S, s0, E, T})

有限オートマトンを用いる

8.4.2 プロセス代数 (X = A op B)

並行プロセスを代数的な操作ができる「式」として表現する事で、並行動作するシステムの振る舞いを解析する。プロセス代数(CSP 等)で、並行動作するシステム同士が協調動作する事を表現するのに「チャネル」の概念を使う。ここで、プリミティブ要素として「同期チャネル」を用意しているが、拡張要素として、「通信バッファ数」を指定できるモデルもある。

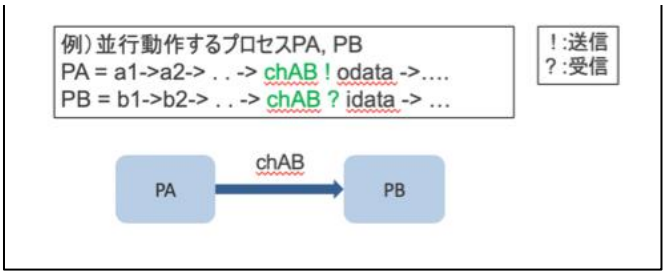


図 8-2 CSP による記述

記述例

- プロセス PA とプロセス PB の並行合成演算

```
PA || PB
```

- 仕様を表現したプロセス Spec と設計を表現したプロセス Dsg が「詳細化関係(refinement)を満たすか」

```
Spec <refine> Dsg
```

8.4.3 現存するツール

Name	Model	language	Distributed by	feature	
SPIN	automaton	Promela	Gerard J. Holzmann	Most measure	
NuSMV	automaton	SMV	ITC-IRST (Italy)	many model	
UPPAAL	Timed automaton	NTA	UP4ALL(Demmark)	Timed model Easy to use(GUI)	
PAT	CSP(base)	CSP#	Semantic Engineering	Many model	
SCADE	Scade	Lustre	ANSYS/ESTEREL	Model Base	
Simulink Design Verifier	Simulink/Matlab	Simulink model	Mathworks	Model Base	
・・・・https://en.wikipedia.org/wiki/List_of_model_checking_tools					

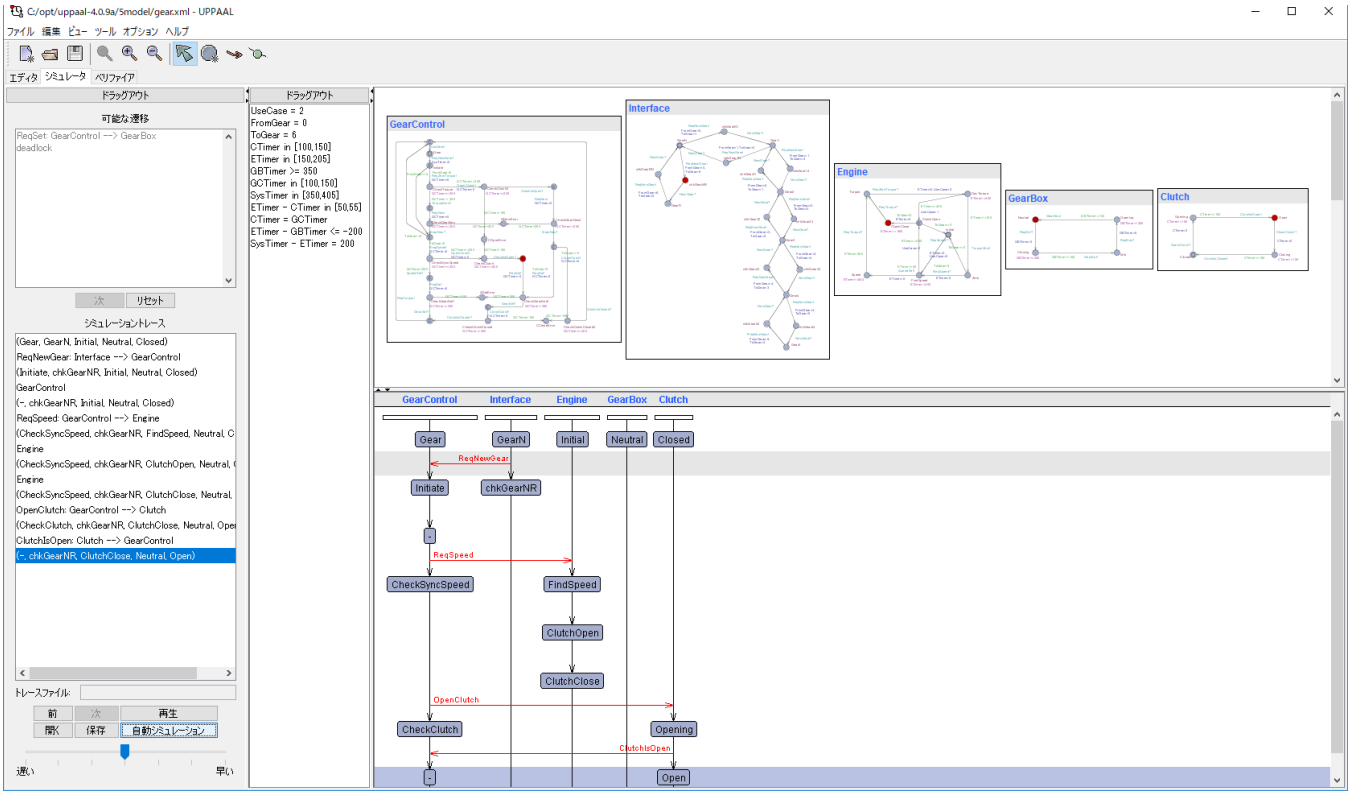


図 8-3 ツール画面

8.4.4 モデル検査ツール適用の流れ

① モデル作成
挙動のモデル

② 検証式(assertion)記述
要件由来（機能・非機能）など、システムが満たしているべき性質を記述

抽象的には以下のようなものが記述可能

- 到達可能性
- 安全性
- 他(Deadlock しない、Livelock しない、など)

8.5 適用事例の情報

- サイト
  - ✧ IPA「報告書：形式手法導入課題を解決する「形式手法活用ガイドならびに参考資料」(https://www.ipa.go.jp/sec/softwareengineering/reports/20120928.html)
  - ✧ ディペンダブルシステムのための形式手法の実践ポータル/モデル検査(MRI)(http://formal.mri.co.jp/db/fmcategory/cat65/)
- Paper

- ✧ 形式手法に基づく並行処理可視化ツールの紹介(産総研・磯部)
  (https://www.ipa.go.jp/files/000004054.pdf)
- ✧ モデル検査ツールの利用 マルチコア環境におけるタスク設計検証(藤倉インターフェース / CQ 出版社)
  (http://iss.ndl.go.jp/books/R000000004-I8952420-00?ar=4e1f)

8.6 プログラミング言語/ライブラリ

8.6.1 並列処理に向いたプログラミング言語

8.6.1.1 並列処理指向のプログラミング言語

並列処理に必要な要素を言語要素として持つ

- Thread 生成
- 同期・通信
- 安全機構 (データ破壊防止等)

8.6.1.2 関数型プログラミング言語

1. （より純粋な）関数内の変数は ThreadSafe
   - Closure
2. 関数の論理構造の中に並列性が明示される
   - 破壊的な代入が無いため、データフローは、関数の引数と返却値のみを追いかければよい

8.6.2 並列指向言語/ライブラリ

8.6.2.1 言語

1. Go – Google
   - https://golang.org/
2. Rust - Mozilla の公式プロジェクト
   - http://www.rust-lang.org
3. Erlang – 公開された Ericsson 内製言語
   - https://www.erlang.org/
4. Occam – Transputer/Inmos 用の言語

8.6.2.2 ライブラリ

1. OpenMP – Shared memory Base
2. OpenMPI – Message Passing Base
3. OSCAR API - OpenMP + α + β + …
4. PPL(Parallel Patterns Library) - Microsoft
5. TBB(Threading Building Block) – Intel

8.6.3 並列処理指向言語の例 – Go

■Kenneth Lane Thompson(C の開発者)らによって開発された。

■言語要素として Thread、Channel の概念を持つ。

8.6.3.1 Channel 定義

プロセス代数(CSP)由来のもの。バッファサイズ指定が可能

```
ch := make(chan int,8)  --- バッファサイズ8のint型のチャネル
```

8.6.3.2 Thread 定義

定義した関数を「go」というキーワードを付けて呼び出す事でスレッド生成を意味する（「go routine」と呼ぶ）

```
go hello();  go world();
          ---- hello, world関数を別Threadで実行
Hello, worldはこの言語で定義された通常の関数。
```

8.6.4 並列処理指向言語の例 – Rust

- Mozilla Research の公式プロジェクト
- 速度、並行性、安全性を言語仕様として保証する C 言語、C++に代わるシステムプログラミングに適したプログラミング言語を目指しており、強力な型システムとリソース管理の仕組みにより、メモリセーフな安全性が保証されている。

「所有権(ownership)」概念の言語要素化

→安全でなければコンパイルが通らない

8.6.4.1 Channel 定義

```
let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();
```

8.6.4.2 Thread 生成

```
let child = thread::spawn(move || {<Thread手続き本体> })
```

8.6.5 マルチスレッド/通信 API

- マルチコアの活用のためにはプログラムが複数の並列実行可能な単位(Thread)に分割されている必要がある。プログラムをそのように構成するためには、Thread の生成/開始のような Thread の実行制御に関わるものと、仮に共通にアクセスできるメモリがない場合に通信可能な、「Thread 間通信」が実現できることが好ましい。

API Spec.	Thread生成	Thread開始	メッセージ送信	メッセージ受信	送受信ポイント
eMCOS	mcos_thread_create	mcos_thread_start	mcos_message_send	mcos_message_receive	送信先Thread指定
pthread	pthread_create		mqueueへのenque	mqueueからのdeque	mqueueに対して
MPI			MPI_SEND	MPI_RECV	endpointQNX
MCAPI/MTAPI	mtapi_action_create	mtapi_task_start	mcapi_msg_send	mcapi_msg_recv	endpoint
QNX			Msg_Send	Msg_Receive	

8.6.6 メッセージ送受信 API

- 2 つの End-Point を指定し、「チャネル」オブジェクトを定義し、それに対して送受信する。

8.6.6.1 チャネル定義

```
1.  EndPoint ep1, ep2;
2.  Channel ch =  new Channel<DataType>(ep1, ep2)
```

8.6.6.2 送受信

```
1.  DataType data;
2.  send(ch, data)  /  Code in Thread 1
3.  receive(channel, &data)    / Code in Thread 2
```

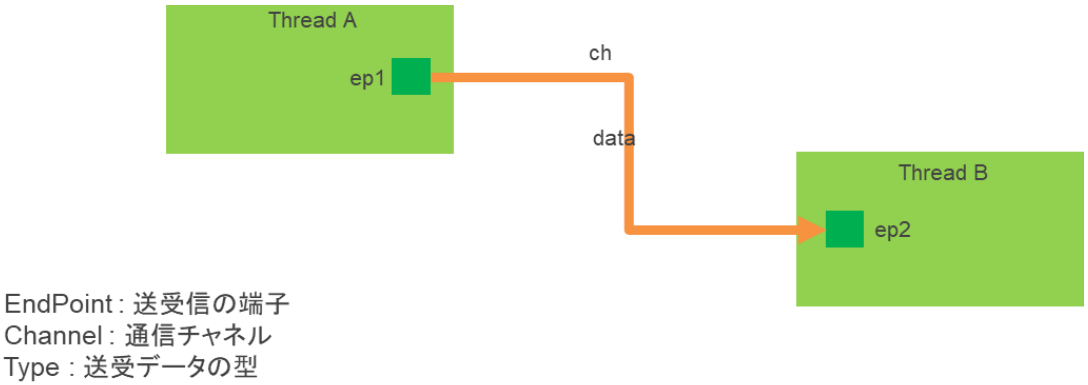


図 8-4 メッセージ送受信チャネル

8.6.7 参考情報(MCAPI)

- MCAPI - The Multicore Association
  - https://www.multicore-association.org/workgroup/mcapi.php
- Wikipedia
  - https://en.wikipedia.org/wiki/MCAPI
- MCAPI 事例(Mentor Graphics)
  - http://www.mentorg.co.jp/training_and_services/news_and_views/2010/autumn/feature_story2/index.html

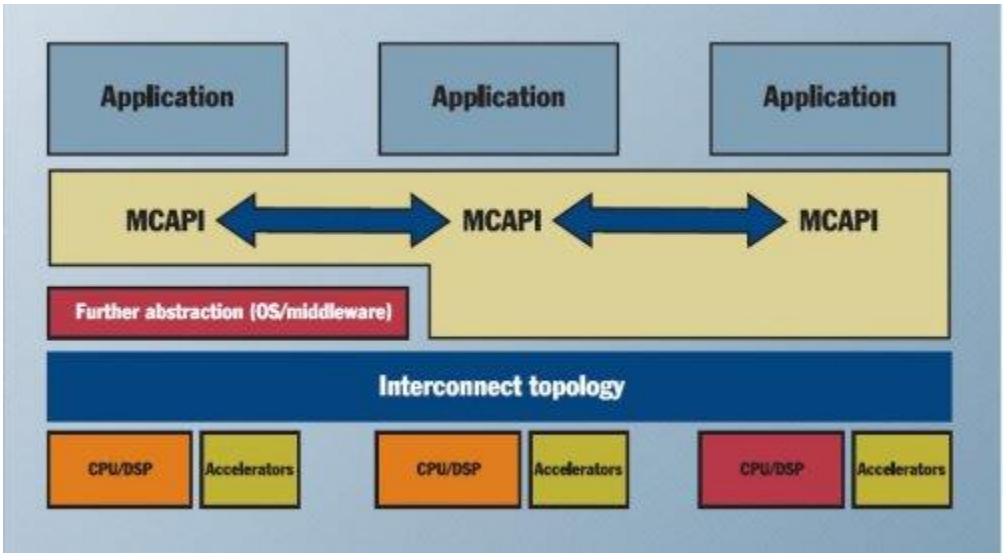


図 8-5 MCAPI

8.7 マルチコア対応 RTOS

8.7.1　マルチコアに対応した OS の機能

■Thread-Core Affinity (Static)

- Configuration によってあらかじめ指定

■Thread-Core Migration (Dynamic)

- 動的に Core 割り当てを変える事ができる

■Message Passing

- スレッド間通信機能

8.7.2 MulticoreOS の特徴(動作モデル)

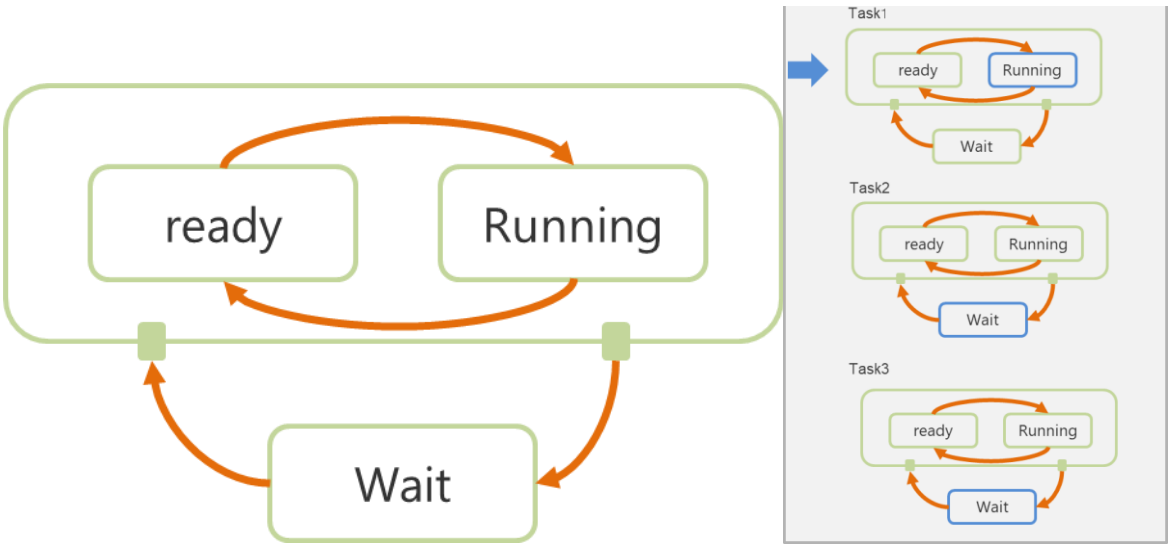


図 8-6 Task の状態遷移モデル

遷移イベントはアプリケーションによる OS の API コール

1. Single Core の場合

   → Running になれるタスクは 1 個

2. Multi- Core(N-Core)の場合

   → Running になれるタスクは N 個

8.7.3 Single Core 3 task

■Single Core - 3 task

■優先度： task1 > task2 > task3

<制約>

1. あるタイミングで Running になれるタスクは最大 1 つ
2. もっとも優先度が高い ready タスクが Running になる
3. ready タスクがあるのに Running タスクがないことはない

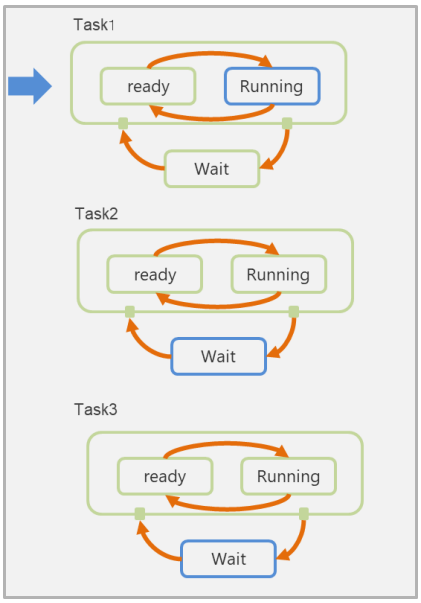


図 8-7 Single Core 3 task

8.7.4 2 Core 3 task

■Single Core - 3 task

■優先度： task1 > task2 > task3

<制約>

1. あるタイミングで Running になれるタスクは最大 2 つ
2. もっとも優先度が高い ready タスクが Running になる
3. ready タスクがあるのに Running タスクがないことはない

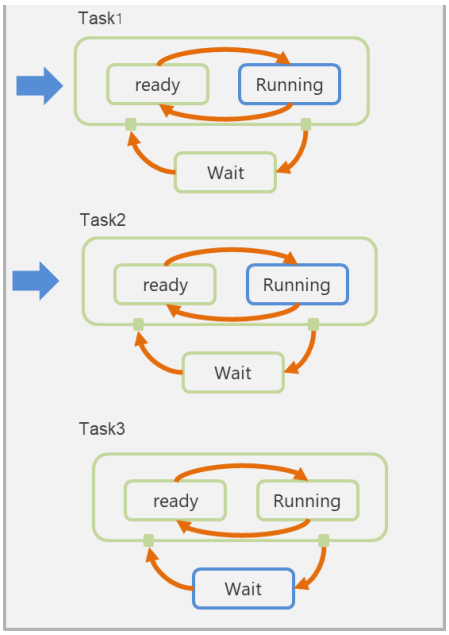


図 8-8 2 Core 3 task

8.7.5 マルチコア対応 RTOS の例

- eMCOS
  - https://www.esol.co.jp/embedded/emcos.html
- QNX
  - https://blackberry.qnx.com/jp
- POSIX 対応 OS (Posix Thread)
  - https://en.wikipedia.org/wiki/POSIX_Threads
- TOPPERS/FMP
  - https://www.toppers.jp/fmp-kernel.html

8.8 設計パターン

8.8.1 設計パターンとは

ソフトウェアのアーキテクチャ、デザイン、コーディングなどに対して、解決策としてのパターンを共有する。パターンはお互いに関連しあい、連携して利用されるパターン言語を形成する。

ルーツ

- Pattern Language(Christopher Alexander)
- Software Design Pattern (GOF ※)
- Pattern Oriented Architecture Pattern (bushman)

- Taming Java Threads (Allen Holub)

図 8-9 設計パターンの種類

現在は様々な領域でのパターン活動がある

※ Eric Gamma, Rechard Helm, Ralph Johnson, john Vissides

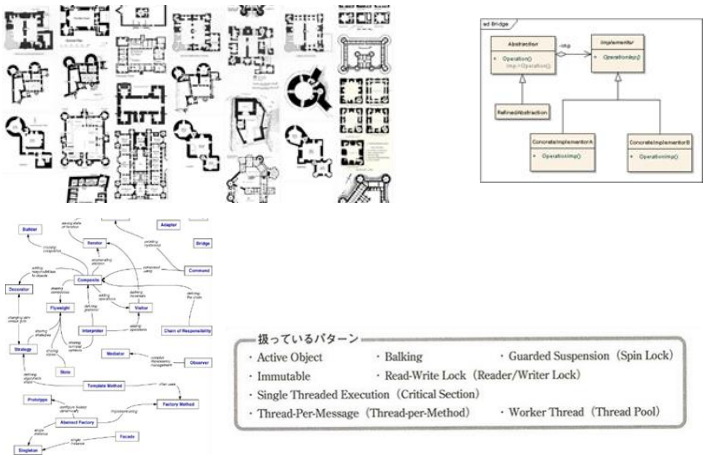


図 8-10 設計パターンのイメージ図

8.8.2 並列化ソフトウェア設計パターン

マルチ/メニーコアを利用したハードウェア上で動作する「並列処理ソフトウェア」に対しても、この「パターン」の蓄積・共有が有効だと思われる。

これらの設計パターンを用いる動機は、起こって欲しくない「設計由来の問題」をなくしたいことである。

- デッドロック
- 共有データの破壊
- 理解しがたい複雑さ
- デッドライン違反
- より効率的に処理を構成したい

複数コアの有効活用

- 無駄な通信を少く
- コア数非依存アルゴリズム（スケーラブル）

8.8.2.1 Pattern Template

活用可能なパターンカタログを作るために以下のような項目がそろっているのが望まれる

section	description
Pattern Name and Classification	A descriptive and unique name that helps in identifying and referring to the pattern.
Intent	A description of the goal behind the pattern and the reason for using it.
Also Known As	Other names for the pattern.
Motivation (Forces)	A scenario consisting of a problem and a context in which this pattern can be used.
Applicability	Situations in which this pattern is usable; the context for the pattern.
Structure	A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
Participants	A listing of the classes and objects used in the pattern and their roles in the design.
Collaboration	A description of how classes and objects used in the pattern interact with each other.
Consequences	A description of the results side effects and trade offs caused by using the pattern.
Implementation	A description of an implementation of the pattern; the solution part of the pattern.
Sample Code	An illustration of how the pattern can be used in a programming language.
Known Uses	Examples of real usages of the pattern.
Related Patterns	Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

8.8.2.2 Pattern Example

From Eun-Gyu Kim / 2004

http://snir.cs.illinois.edu/patterns/patterns.pdf

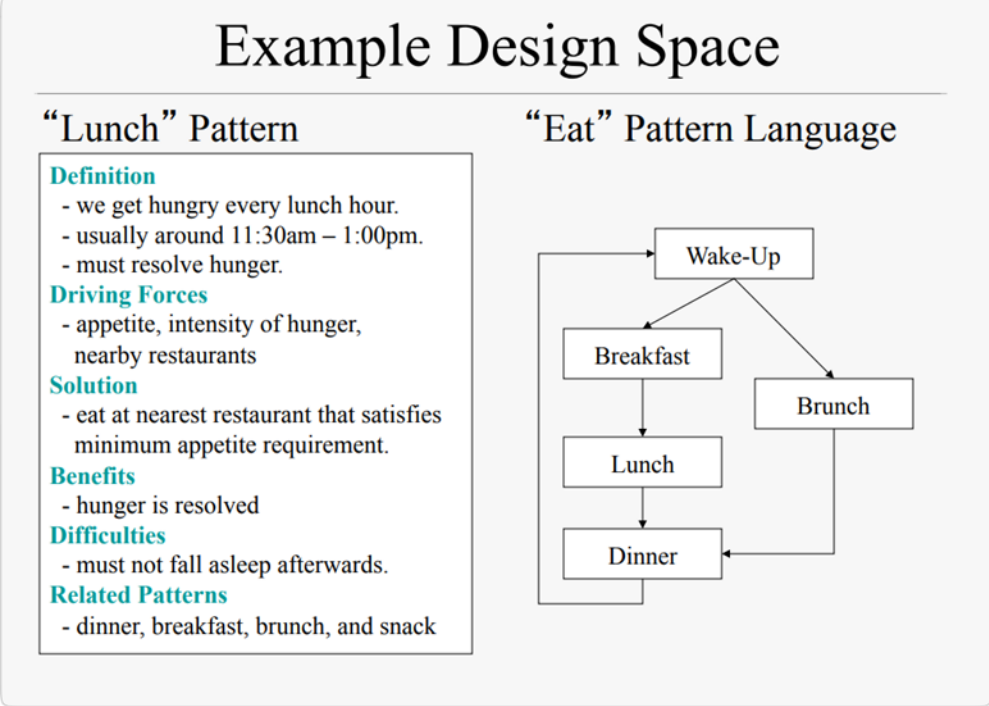


図 8-11 設計パターンのサンプル

8.8.2.3 Concurrency Patterns (POSA2)

POSA：Pattern Oriented Software Architecture

ソフトウエアアーキテクチャパターンの本

並列処理パターンもいくつか紹介されている

Name	Description
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.
	Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.
Thread-specific storage	Static or "global" memory local to a thread.

Wikipedia：Design Pattern / Concurrency pattern

8.8.2.4 Intel – Design Pattern

Intel 発信の情報

https://software.intel.com/en-us/node/506112

以下のようなパターンが紹介され、それをどの様に実装するか解説されている

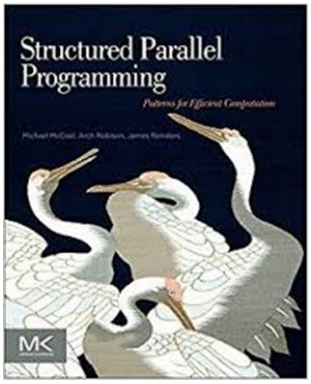
- Agglomeration
- Elementwise
- Odd-Even Communication
- Wavefront
- Reduction
- Divide and Conquer
- GUI Thread
- Non-Preemptive Priorities
- Local Serializer
- Fenced Data Transfer
- Lazy Initialization
- Reference Counting
- Compare and Swap Loop
- General References

8.8.2.5 Structured Parallel Programming

Subtitle：Patterns for Efficient Computation

Michael McCool, Arch D. Robinson, James Reinders

並列プログラムを構成するためのパターンを紹介し、後半にはそれらを使った並列アルゴリズムの解説を行っている



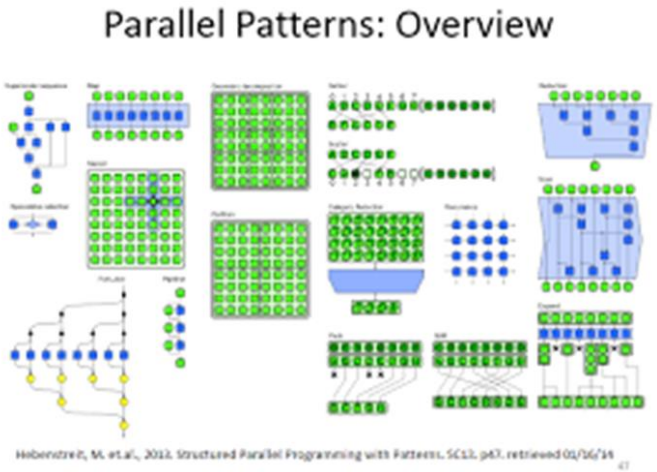


図 8-12 並列プログラムのための設計パターン

8.8.2.6 anti-pattern

アンチパターン (英: anti-pattern) とは、ある問題に対する、不適切な解決策を分類したものである。 語源は、ソフトウェア工学におけるデザイン**パターン**である。 主に失敗した開発プロセスに焦点を当てて失敗に陥る**パターン**を類型化する。 そうすることで、そのような事例の早期発見と対応策に関しての提案を目的とする。

マルチコア・システム開発に置いても、さまざまな不具合事象をパターンとして整理する活動を行う事で、同様の効果を期待できる。

8.9 並列化支援技術

マルチコアプロセッサと並列指向の言語やライブラリが与えられただけでは、それを有効活用するソフトウェアを構築するのは簡単ではない。

現状はある程度専門的なエンジニアがこれらを担っている。今後は、マルチコアを活用するソフトウェアをより多くのエンジニアが開発できるように、つかえる支援ツールの充実が望まれる。

8.9.1 並列化支援環境

並列化されていないプログラムのどこが並列化可能か、並列化する事による性能改善への寄与が大きいかといった情報を開発者へ提示。Thread プログラミング支援。

8.9.2 自動並列化技術

並列化されていないプログラムをスレッドライブラリなどを利用した並列化されたプログラムに変換する。

それにあたって、並列化されていないプログラムを解析（基本的にはデータフロー解析）して並行動作可能部分を抽出しスレッド化プログラムとして再構成する。さらに、限られたコア数に対してそれらのスレッドを割り当て配分する。

■ CUDA – nVidia

GPGPU 向け

■ OpenCL - Khronos Group

Embedded Profile (組み込み向け)

■ Intel Parllel Studio

統合開発環境として提供

■ Eclipse Parallel Tools Platform (PTP)

MPI, OpenMP, UPC(Unified Parallel C), OpenSHMEM ,OpenACC をサポート

8.9.3 自動並列化技術（具体例）

- eMBP(eSOL) based on MBP (Nagoya Univ.)
  - Input : (Simulink Block Model ， C Code generated by Embedded
  - Output : Parallelized C Code（use multi-thread library)
- SILEXICA (SILEXICA)
  - Tool Set for support parallelization
- Automatic Parallelization Compiler
  - OSCAR (waseda Univ)
    - C source(※) -> Parallelized C (with OSCAR API)
      - ※Parallelizable
  - Intel Compiler
    - C source -> Parallelized C(with openMP)
      - (--parallel option)
  - Solaris Stuio Fortran (Loop optimizatation)
    - -autopar option

- PLUTO (Not general)
  - Automatic parallelizer and locality optimizer fo raffine loop nests
  - http://pluto-compiler.sourceforge.net/

8.9.4 自動並列化の考え方

- 入力：並列化されていないプログラム
- 出力：並列化(Multi-Thread 化）されたプログラム
- 並列化プロセス：
  - データフロー解析
    - 変数の代入関係を解析し、処理単位のデータ依存グラフ、タスクグラフを抽出。グラフ構造から並列可能な部分（ノード）が分かる
  - コア割り当て
    - ノード（タスク）が動作するコアを割り当てる。
  - 並列化コード生成
    - 別なコアに割り当てられた処理単位は別なスレッドとして動作するようなマルチスレッド・コードとして生成する。
- 並列化のゴール
  - 処理速度の向上（処理時間を最小化）
  - 低消費電力化（プロセッサの消費電力を最小化）
  - 上記 2 つのゴール指標はトレードオフ

8.9.5 モデルベース並列化技術

- 入力
  - データフローとして解釈できるモデル
  - そのモデルに対応して作成/生成されたプログラムコード(C 言語プログラムなど)
- 出力
  - Thread 化されたプログラム
  - Thread へのコア割り当て情報
- 並列化プロセス
  - データフローはモデルから取得
  - 負荷分散、クリティカルパスなどの制約を満たすコア割り当てのルールを適用する

8.9.6 MBD は並列設計に利用可能

モデルベースの設計ツールが対象としている「モデル（ソフトウェア観点）」には、「データフロー」モデル、および、「状態遷移モデル」をがあり、これらをグラフィカルエディタなどの支援のもとに設計すると、そこから並列性を自動抽出することが可能となる。

- データフロー
  - Matlab/Simulink

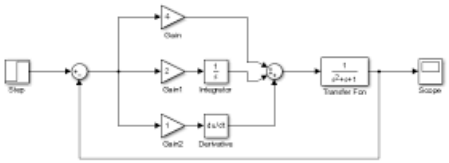


図 8-13 MATLAB/Simulink

  - LabView

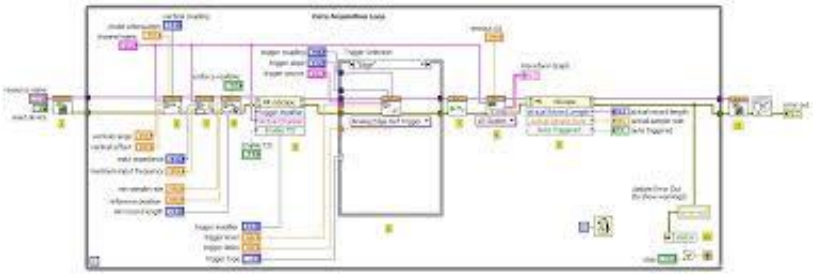


図 8-14 LabView

- 状態遷移モデル
  - Stateflow

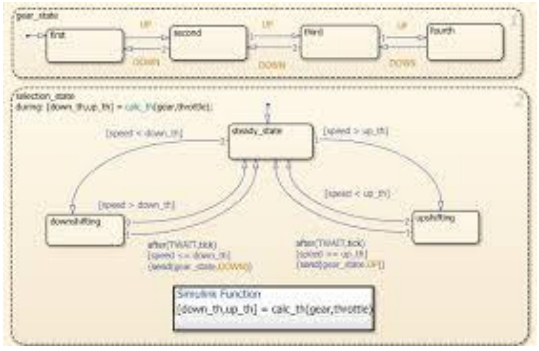


図 8-15 Stateflow (MATLAB/Simulink)

  - EHSTM

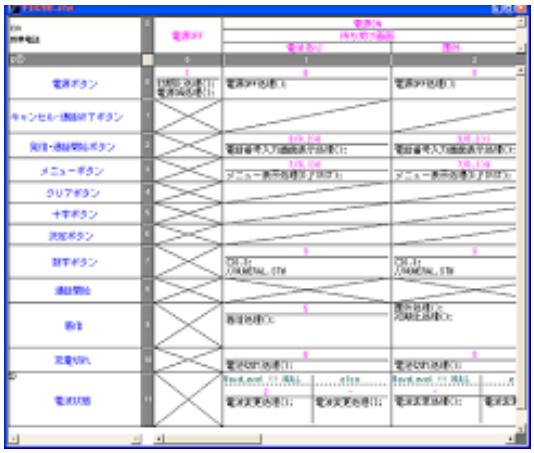


図 8-16 EHSTM

8.9.7 eMBP / eSOL

Simulink モデルと Simulink モデルから生成された C ソースコードから並列化（Thread 化）された C ソースコードを出力。

Thread のコア割り当てのために SHIM(※)を利用した性能見積もりを行う。

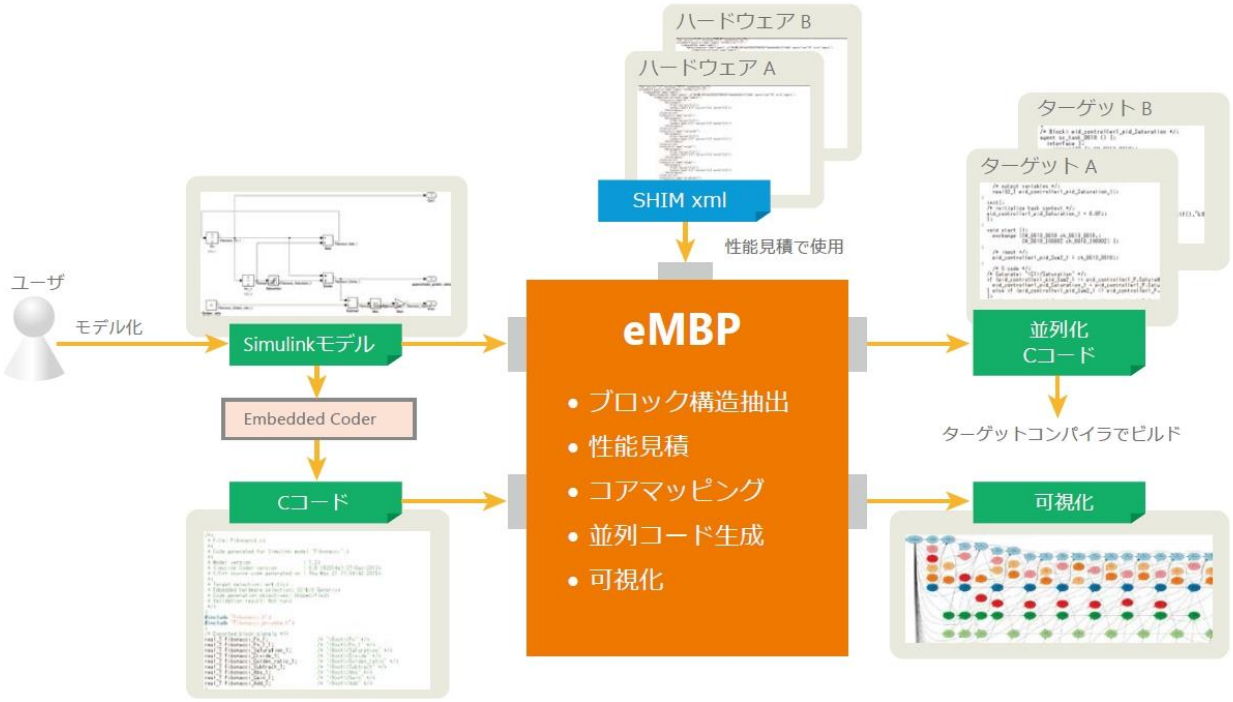


図 8-17 eMBP (eSOL)

8.10 トレースフォーマット

マルチコア環境でのプログラム・デバッグでは、ステップ実行やブレークポイントを利用した操作でのデバッグが難しい。そこで、タイムスタンプのついたトレースをとり、実行後に可視化などをする事で現象分析する事が有効となる

しかし、可視化機能、解析機能などを持ったツールはトレースフォーマットに依存したものになるため、標準的なフォーマットが必要となる。

8.10.1 トレースフォーマット CTF

- フォーマット名: CTF(Common Trace Format)
  - https://www.efficios.com/ctf
- 対応ツール
  - TraceCompass
    - Eclipse-plug-in

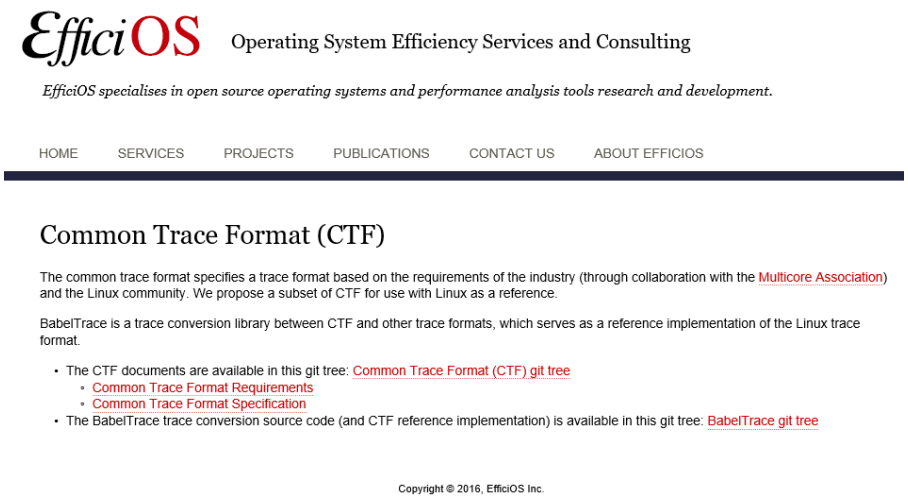


図 8-18 TraceCompass

8.10.1.1 CTF 特徴

- 対象ファイルフォーマット
  - トレース要素定義ファイル(DSL) meta
    - イベントを定義
  - 対象ファイル
    - 計測データ
- MCA からも参照されている
  - TOOLS INFRASTRUCTURE WORKING GROUP (TIWG™)
    - https://www.multicore-association.org/workgroup/tiwg.php

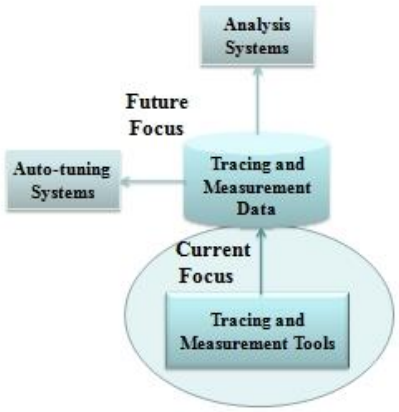


図 8-19 CTF

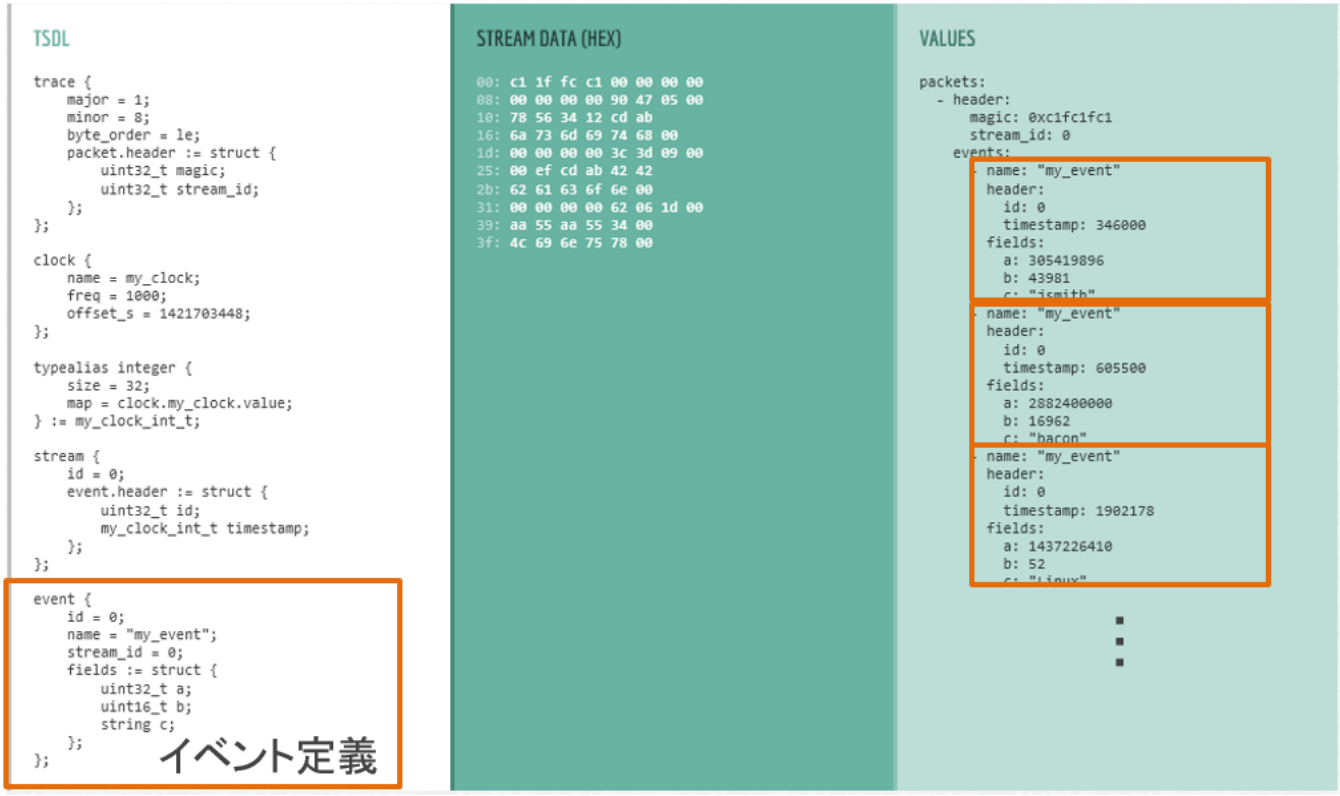


図 8-20 CTF 具体例

8.10.2 可視化技術

8.10.2.1 CTF 対応ツール

- TraceCompass
  - Eclipse-plug-in

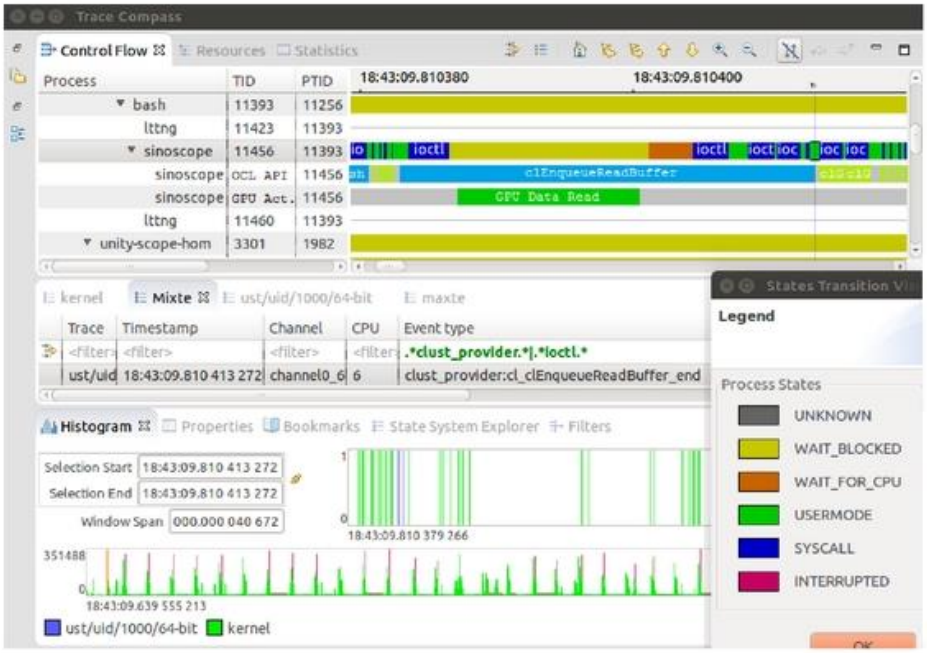


図 8-21 TraceCompass ツール画面

8.10.2.2 TOPPERS/TLV

- Trace Log Visualizer
  - https://www.toppers.jp/tlv.html

図 8-22 Trace Log Visualizer ツール画面

9 <Appendix>組込みマルチコア用語集

■ 本章の対象読者

知識・経験：レベル 1（入門者）マルチコアの知識なし

プロセス：要件定義、設計、実装、テスト

ドメイン：組込み全般

キーワード：用語集、技術用語、テクニカルターム、略語、同義語、対義語

■ 本章を読んで得られるもの

マルチコア技術の専門用語の意味が分かる。

車載電子システムの専門用語の意味が分かる。

■ 要旨

組込みマルチコアコンソーシアム マルチコア適用委員会（WG3）が作成した「マルチコア適用ガイド（2017 年度作成分）」の解説資料の中から、70 程度の専門用語を選定し、用語集としてまとめた。

用語の説明のほか、用語の英語表記、略語を開いたときのフルスペル、同義語、対義語を記載した。

9.1 ア行

■ アクセラレータ（accelerator）

- システムの中核をなす標準的な CPU（central processing unit）に対して、負荷の大きい特定の処理を大幅に高速化（accelerate）して、CPU の負荷を肩代わりする周辺プロセッサや周辺ハードウェアを「アクセラレータ」と呼ぶ。例えば、GPU（graphics processing unit）や DSP（digital signal processing unit）、IPU（image processing unit）、NPU（neural network processing unit）、ビデオ CODEC（coder decoder）、暗号回路、ユーザ定義のカスタム回路などがある。アクセレータは、SoC（system on a chip）として CPU コアと 1 チップ化される場合もあるし、サーバシステムなどに組み込む拡張ボードの形をとる場合もある。

■ アサーション（assertion）

- → 「検査式」を参照。

■ エミュレータ（emulator）

- 広義の意味では、オリジナルのシステムの動作を模倣（emulate）する装置やソフトウェアを指す。例えば、ある命令セットのプロセッサの動作を別の命令セットのプロセッサを使って実現するため、命令列の変換処理（バイナリ変換）などを行うソフトウェアを「エミュレータ」と呼ぶ。組込みシステム開発では、ソフトウェアのデバッグやシステムの評価に利用するフル ICE（in-circuit emulator）や JTAG（Joint Test Action Group） ICE を「エミュレータ」と呼ぶことが多い。こうしたデバッグ用アダプタ（ハードウェア）は、デバッグ回路を内蔵するプロセッサやその代替となる評価用チップを利用して、プロセッサ内部の状態を監視したり、プログラムの実行を制御したり、プロセッサの実行履歴を取得したりする機能を備えている。

■ オートマトン（automaton）

- コンピュータの計算機構を抽象的に表したモデル。複数の内部状態を持ち、現在の状態と入力の組み合わせに基づいて次の状態へ移行する。状態遷移図や状態遷移表に対応する。状態数が有限のものを「有限オートマトン」と呼び、さらに現在との状態と入力によって次の状態が一意的に決まるものを「決定性有限オートマトン（DFM：deterministic finite automaton）」、一意的に決まらないものを「非決定性有限オートマトン（NFM：nondeterministic finite automaton）」と呼ぶ。また、ソフトウェアの挙動を検証する際には、これらを拡張したモデルを利用することが多い。このような拡張モデルの例として、SPIN（Simple Promela Interpreter）が採用としている Büchi オートマトンや、UPPAAL が採用している NTA（Networks of Timed Automata）などがある。

9.2 カ行

■ 機能安全（functional safety）

- 人や環境に危害を加えるリスクそのものを取り除くことを「本質安全」と呼ぶ。これに対して、システムに機能的な工夫を施し、許容できるレベルの安全を確保することを「機能安全」と呼ぶ。機器やシステムの開発において、設計段階でリスクを 100%取り除くことが望ましいが、運用時に想定外の事態が生じたり、対策に膨大なコストがかかったりするなど、現実にはリスクを完全に排除することが難しい。そこで、合理的なリスク分析に基づき、設計段階では排除しきれない残留リスクへの対策となる機能を、別途付加しておく。機械製造やプラント、交通輸送、医療、防衛などの分野で、機能安全への対応が求められる。機能安全の国際標準規格として、電気電子システム全般を対象とする「IEC 61508」をベースに、「ISO 26262」（自動車）や「IEC 61513」（原子力）、「JAR/FAR25 1309」（航空機）などの分野別の規格が制定されている。

- 機能分散（functional distribution）
  - マルチプロセッサやマルチコアによる分散コンピューティングの一形態。ソフトウェアを複数のまとまった機能に分割し、それぞれの機能を個々のプロセッサ（または CPU コア）に割り当てて、同時並行に実行する。ソフトウェアの分割は人手で行うことが多く、開発者が考えやすい単位で機能を分割するのが一般的。
  - 対義語　負荷分散

- 形式手法（formal method）
  - 数学的に厳密に意味づけられた言語を用いて情報システム（ソフトウェア、ハードウェア）の要求や設計などを記述する手法。例えば、情報システムがユーザの要求を満たしているか否かを、論理的に推論するための仕組みを提供する。証明したい仕様や論理的な性質は、「形式仕様記述言語」と呼ぶ言語で記述する。鉄道や航空宇宙、電力など、高い信頼性と安全性、セキュリティが求められる分野で利用されている。

- 形式仕様記述（formal specification description）
  - プログラムが満たすべき仕様や論理的な性質を記述するための言語。形式手法で用いられ、システムの状態とその変化、事前・事後条件、並行動作、データ構造などを表現する。Z、VDM、B、CSP、Event-B、Alloy など、形式手法ごとにさまざまな言語が提案されている。

- 決定性（deterministic）
  - 情報システムにおいて、現在の状態と入力により、次の状態が一意的に決まる性質を「決定性」と呼ぶ。「決定性有限オートマトン」や「決定性（決定的）アルゴリズム」といった使い方をする。決定性のあるアルゴリズムを実行すると、常に同じ手順で計算を行い、常に同じ結果を出力する。
  - 対義語　非決定性

- 検査式（assertion）
  - プログラムが満たすべき論理的な性質（成立条件）を表現した数式。モデル検査（形式手法）では、検査式の表現として時相論理式（temporal logic）を利用することが多い。
  - 同義語　アサーション

- 故障注入（fault injection）

- システムに故障が発生したとき、実際に安全な状態へ移行するかどうかをテストする手法。故障が想定される箇所の信号やメモリ、レジスタの値を強制的に変化させて擬似的に故障状態を再現し、システムの挙動を観測する。制御用ボードなどの実機（ハードウェア）上で実際の信号の値を変化させる方法や、制御プログラムを改変してメモリやレジスタの値を書き換える方法、ICE などのデバッガを使ってメモリやレジスタの値を変更する方法、シミュレーション（仮想プラットフォーム）を用いてシステムの故障状態を模擬する方法など、さまざまな手法がある。自動車向け機能安全規格の ISO 26262 では、故障注入テストの適用が推奨されている。

■ コンテキスト（context）

- 本来の意味は、文章などの前後の脈絡（文脈）。情報システムでは、割り込みなどが原因で CPU がタスクの処理を中断したとき、タスク実行の状態にかかわる CPU の情報（データ）をいったんレジスタやメモリに保存する。このときに保存する情報を「コンテキスト」と呼んでいる。マルチタスクシステムにおいて、処理するタスクを切り替えるために CPU の状態情報を保存したり復元したりする動作を「コンテキストスイッチ」と呼ぶ。

9.3 サ行

■ 自動並列化コンパイラ（automatic parallelizing compiler）

- → 「並列化コンパイラ」を参照。

■ 状態遷移（state transition）

- 状態機械（state machine）において、ある状態から別の状態へ移行すること。状態機械は「状態」と「状態間の遷移」から構成される。状態機械の動作は、状態遷移図や状態遷移表として表現される。

■ 診断カバレッジ（diagnostic coverage）

- → 「ダイアグカバレッジ」を参照。

■ スレッドセーフ（thread safe）

- 複数のスレッドが並行に動作するマルチスレッドシステムでは、複数のスレッドが同時に、同一のコードを実行する状態が起こりうる。このような状況でも問題が発生しない性質を「スレッドセー

フ」と呼ぶ。スレッドセーフを維持するためには、例えばリエントラント（再入可能）な関数を使用したり、共有データへのアクセスを管理する排他制御を行ったりする必要がある。

■ 静的解析（static analysis）

- ソフトウェアやシステムの解析手法のうち、実際のプログラムの実行を伴わないものを「静的解析」と呼ぶ。テストやデバッグなどに利用する。例えば、プログラムの構造（制御フロー、データフロー）を解析して、バッファオーバフローやポインタの範囲外参照を指摘する手法や、ソースコード記述を解析して、プログラムが指定されたコーディング規約に従っているかどうかを判定したりする手法などが、静的解析に分類される。
- 対義語　動的解析

■ 静的テスト（static test）

- 静的解析の手法を利用するテスト技術。原理的にはプログラムのすべてのパスを解析できるので、プログラムの実行を伴う動的テストと比べて、テストのカバレッジ（網羅率）を引き上げやすい。また、テストケースを用意する必要もない。ただし、プログラムの規模が大きくなると解析に時間がかかることが多く、頻発する誤検知（実用上問題がなくても不具合と判定してしまう状態）に悩まされることも少なくない。
- 対義語　動的テスト

■ 設計パターン（design pattern）

- 開発者がよく出会う問題に適切に対処するための指針をまとめた定石集（カタログ）。設計パターンを利用することにより、再利用性の高いソフトウェアを効率よく作成できるようになる。例えば、インスタンスの生成方法やクラスの抽出方法、実装とインターフェースの切り分け方などがまとめられている。システム全体の構造を決める際に役立つパターン（アーキテクチャパターン）や、失敗に陥るパターン（アンチパターン）などを含めて「設計パターン」と呼ぶ場合もある。
- 同義語　デザインパターン

9.4 タ行

■ ダイアグカバレッジ（diagnostic coverage）

- 故障には、システムの状態を安全側へ導く安全側故障と、危険側へ導く危険故障がある。危険側故障のうち、故障検出機能によってその発生を検知できる（危険に対処できる）ものの割合を「ダイ

アグカバレッジ」と呼ぶ。機能安全の考え方では、危険側故障が発生しても、それを故障検出機能によって検知できるのであれば、すなわちダイアグカバレッジが高ければ、「安全度が高い」とする。

- 同義語　診断カバレッジ

■ タスク（task）

- 組込みシステムでは広く OS（operating system）が普及しており、OS 上から見たときの処理の実行単位を「タスク」と呼ぶことが多い（第 2 部 <動作の見える化> を参照）。その一方で、OS を使用しないベアメタル方式の組込みシステムもあり、単に処理する仕事の実行単位を「タスク」としている場合もある（第 1 部 <並列化フロー> を参照）。

■ デザインパターン（design pattern）

- →　「設計パターン」を参照。

■ データ競合（data race）

- 複数の異なるスレッドがメモリ上の同一のアドレスに対して同時アクセス（ただし、少なくとも一つは書き込み）を行うと、実行結果は予期できないものになる。このような状態を「データ競合」と呼ぶ。これを回避するには、スレッドの実行順序関係が明確になるように排他制御を行う必要がある。

■ デッドロック（deadlock）

- 二つのタスクがそれぞれ別々のリソースをロックしている時に、同時に相手がロックしているリソースの解放を待つ状態に陥り、その状態から抜けられなくなる現象。マルチコアシステムや、シングルコアのマルチタスクシステムにおいて発生する可能性がある。デッドロックは、非常にレアな条件が成立したときにしか発生しないことが多く、短時間のテストでは現象を再現することが難しい。

■ デバッグインターフェース（debag interface）

- プロセッサが備えるデバッグ専用のシリアル通信ポート。プロセッサ内部のデバッグ機能の制御に用いる。標準的な仕様として、4～5 ピンの JTAG（Joint European Test Action Group）や 2 ピ

ンの SWD（Serial Wire Debug）がある。JTAG ICE などのデバッグ用アダプタは、デバッグインターフェースに接続して使用する。

- デバッグモニタ（debag monitor）
  - プロセッサ上で動作するデバッグ用のプログラム。例えばハードウェアを初期化したり、レジスタの値を読み出したり、ブレークポイントを設定したりする。外部の開発用 PC（その上で動作するデバッグアプリケーション）との間でコマンドやデータを受け渡しながら動作する。

- 動的解析（dynamic analysis）
  - ソフトウェアやシステムの解析手法のうち、実際のプログラムの実行を伴うものを「動的解析」と呼ぶ。テストやデバッグに利用する。例えば、プログラムモジュールに入力パターンを与えて実行し、出力結果の期待値照合を行う手法や、プログラムの中にチェックコードを埋め込むなどして観測点を設け、プログラムを実行しながらデバッグに有用な情報を取得（トレース）する手法などが、動的解析に分類される。。
  - 対義語　静的解析

- 動的テスト（dynamic test）
  - 動的解析の手法を利用するテスト技術。動的解析によるテストの良し悪しは、用意するテストケースに依存する。起こりうるすべての状態を再現するテストケースを用意することは、現実には難しく、静的テストのような網羅的なテストは行えない。そのため、検証の進捗を計測する「カバレッジ（網羅率）解析」と組み合わせて適用することがある。
  - 対義語　静的テスト

- トレース（trace）
  - システムの内部状態を監視・記録すること。ソフトウェア開発では、主にデバッグに役立つ各種の実行履歴（ログ）を取得する作業を指す。例えばプロセッサは、プログラムの実行中に命令の実行履歴やレジスタ値の推移をデータパケットにして外部へ転送する機能を備えている。OS には、システムの動作中にプロセスやタスクの実行履歴を記録する機能がある。こうした機能を利用してシステムを評価したり、プログラムをデバッグしたりする作業を「トレース解析」と呼んでいる。

9.5 ハ行

■ ハイパーバイザ（hypervisor）

- コンピュータを仮想化し、種類の異なる複数の OS を一つのシステムの中で並列稼動させるためのソフトウェア。ハードウェアの動作を模擬する「仮想マシン（VM：virtual machine）」を利用して実現する。ハイパーバイザには、CPU 上で直接動作し、その上で複数の OS が稼動するタイプ（いわゆるハイパーバイザ型。Xen や VMware ESX、Hyper-V、KVM など）と、ある特定の OS の上のアプリケーションソフトウェアとして動作し、その上で別の OS が稼動するタイプ（ホスト型。VMware Server や Virtual PC、Microsoft Virtual Server、Parallels Desktop、QEMU など）の 2 種類がある。ハイパーバイザは、サーバやクラウドコンピューティングの基盤技術として発展した。組込み OS ベンダの多くは、汎用 OS（Linux や Windows）とリアルタイム OS を同時に動かせる組込みシステム向けのハイパーバイザを提供している。

■ 非決定性（nondeterministic）

- 情報システムにおいて、現在の状態と入力により、次の状態が一意的に決まらない性質を「非決定性」と呼ぶ。「非決定性有限オートマトン」や「非決定性（非決定的）アルゴリズム」といった使い方をする。例えば入力以外の条件によって次の状態が変わったり、タイミングに依存する処理を行ったりした場合、決定性の条件がくずれて非決定性となる。
- 対義語　決定性

■ 負荷分散（load distribution、load balancing）

- マルチプロセッサやマルチコアによる分散コンピューティングの一形態。特定のプロセッサ（または CPU コア）に負荷が偏らないように、OS や負荷分散ソフトウェアが特定のルールに基づいてソフトウェアのタスクを複数のプロセッサ（または CPU コア）へ振り分け、全体としてバランスよく並行処理を進める。同種のプロセッサを組み合わせるホモジニアス構成のコンピュータ上で実現することが多い。
- 対義語　機能分散

■ プロセス代数（process algebra）

- 並行システムの動作や構造を形式的に記述する手法の一つ。並行に動作するプロセスの間の相互作用や通信、同期といった関係を数学的な代数式で表現する。式（プロセス記述）を操作する代数的規則も定義されている。例えば並列プログラムが仕様どおりに作成されているかどうかの検証や、二つのプロセスが等価かどうかの分析などに用いる。プロセス代数体系の例として、CSP

（Communicating Sequential Process）や ACP（Algebra of Communication Process）、CCS（Calculus of Communicating Systems）などがある。

■ プロファイリング（profiling）

- システムの性能を解析するため、プログラムを実行しながらその挙動や特性を調べる作業。動的解析手法の一つ。プログラムの実行中に関数呼び出しなどのイベントとその処理時間を記録する。記録したデータに統計的な処理を施し、実行により多くの時間がかかっている関数（性能向上のボトルネックとなっている箇所）を見つけ出す。このような部分を修正・最適化すれば、システム全体の性能を改善できる。プロファイリングの作業を支援するツールを「プロファイラ」と呼ぶ。

■ 並行処理（concurrent processing）

- 「並行処理」とは、複数の処理が（少なくとも論理的に）同時に走っていて、処理やイベントが起きるタイミングが任意のものを指す。

■ 並列化コンパイラ（parallelizing compiler）

- 単一のプロセッサで動作する「逐次プログラム」から並列化が可能な箇所を抽出し、並列コンピュータやマルチコアプロセッサで実行可能な「並列プログラム」に変換するソフトウェアツール。並列化コンパイラは、まずプログラム内部のデータ依存や制御依存の関係を解析して並列処理構造（データフローグラフや制御フローグラフなど）を抽出し、等価な処理を行う並列表現に置き換える。次に、分割された並列実行可能な箇所をターゲットとなる複数のプロセッサ（または複数の CPU コア）へ割り当て、処理の実行順序を決定する。続いて、データ転送のオーバーヘッドやメモリの使用効率を考慮して、メモリ上のデータの分散配置を決定する。最後に、各プロセッサ（各 CPU コア）で動作する C 言語プログラム、または並列化指示の記述を含むスレッド化されたソースコードを生成する。オスカーテクノロジーやドイツ Silexica 社などが並列化コンパイラ製品を開発・販売している。
- 同義語　自動並列化コンパイラ

■ 並列処理（parallel processing）

- 「並列処理」とは、実際に処理が同時に起こっていることを指す。目的は、主に計算速度の向上。従って、1 コアの CPU（シングルコアプロセッサ）を一つだけ使うシステムの場合、「並列処理」という表現は意味をなさない。

- 並列プログラム（parallel program）
  - 並列計算によって処理性能を向上できるプログラム。プログラム内部の並列性（データ並列化やタスク並列化、パイプライン並列化）を利用して、処理性能を引き上げる。並列コンピュータやマルチコアプロセッサの上で動作する。並列プログラムを開発する環境として、人手で並列性を記述する OpenMP や Intel TBB（Threading Building Blocks）、POSIX thread（pthread）などのスレッドライブラリがある。並列化コンパイラを利用すると、単一プロセッサ向けの通常のプログラム（逐次プログラム）を並列プログラムへ自動変換できる。
  - 対義語　逐次プログラム

- ヘテロジニアス（heterogeneous）
  - 「異質の」、「異種の」を意味する言葉。「ヘテロジニアスマルチコア」と言った場合、異なるハードウェアアーキテクチャ、異なる命令セットアーキテクチャを備える複数のプロセッサ、およびアクセラレータのコア（大規模回路ブロック）を集積した SoC（system on a chip）タイプの LSI を指す。例えば、汎用的な CPU コアに加えて、GPU（graphics processing unit）コアや NPU（neural network processing unit）コア、FPGA（field programmable gate array）のプログラマブル論理のコアなどを混載した SoC が考えられる。
  - 対義語　ホモジニアス

- ボディ系（body system）
  - 自動車の車体（ボディ）まわりの電子システム。ドアやパワーウィンドウ、ミラー、ライト、エアコン、キーレスエントリーなどの制御を行う。自動車の駆動に直接かかわる「パワートレイン系」や人命にかかわる「安全系」と比べると、安全性や信頼性の要求レベルは相対的に低い。

- ホモジニアス（homogeneous）
  - 「同質の」、「同種の」を意味する言葉。「ホモジニアスマルチコア」と言った場合、同じハードウェアアーキテクチャ、かつ同じ命令セットアーキテクチャを採るプロセッサコアを複数内蔵したプロセッサ LSI を指す。SMP（symmetrical multi-processing）構成のコンピュータを実現する場合、ホモジニアスマルチコアのプロセッサ LSI を使うことが多い。
  - 対義語　ヘテロジニアス

9.6 マ行

■ マルチコア OS（multicore operating system）

- マルチコアプロセッサ上での動作に対応した OS。SMP 構成に対応したものや AMP 構成に対応したもの、その両方に対応したものなどがある。SMP OS は、負荷分散アルゴリズムに従って、複数のコアへ動的にタスクを振り分ける機能を備えている。AMP OS が備える機能は開発元によって異なるが、多くの AMP OS はコア間の同期・通信処理をコーディングしやすいように、セマフォやイベントフラグ、データキューなどの API（application programming interface）を用意している。T-Engine フォーラムが開発した AMP OS「AMP T-Kernel」の場合、ファイル管理機能もマルチコア向けに拡張されている。TOPPERS プロジェクトの「TOPPERS/FMP」の場合、リアルタイム性の確保のため、基本はタスクをコアに固定する ASP OS だが、システムの稼働中にタスクをあるコアから別のコアへ移動させる API を用意しており、この API を使用することにより、SMP OS と同等の機能を実現している。

■ マルチタスク（multitasking）

- 人やコンピュータが、複数の作業（タスク）を切り替えながら同時並行に実行すること。コンピュータにマルチタスクを導入する場合、ごく短い時間（例えば数ミリ～数十ミリ秒）ごとに実行タスクを切り替えることで、I/O 処理や通信処理に伴う CPU の待ち時間を削減できることが多い。ユーザからは複数のアプリケーションが同時に実行されているように見える。タスク切り替えの管理は、マルチタスクに対応した OS が行う。

■ メトリクス（metrics）

- 「尺度」、「評価基準」を意味する言葉。「ソフトウェアメトリクス」と言った場合、ソフトウェアの品質を定量的に測定する手法やその際の評価基準を指す。例えば、コード行数（規模）やモジュール間の呼び出し関係（結合度）、if や for、which といった分岐が発生する構文の数（複雑度）、同一または類似コードの量（コードクローン）、テスト項目数、バグの数、レビュー時の指摘件数などを指標として品質管理を行う。これらの数値は、ツールや開発環境を使って自動的に収集することが多い。

■ メニーコア（many core）

- 複数のプロセッサコアを 1 チップに集積した LSI（マルチコアプロセッサ）のうち、コア数が非常に多いものを指す。「何個以上がメニーコア」という明確な基準はないが、10 コア以上、あるいは 16 コア以上のチップをメニーコアと呼ぶことが多い。GPU（graphics processing unit）や NPU

（neural network processing unit）、データフロープロセッサなど、演算ユニットの数が多いアーキテクチャを採るプロセッサ（アクセラレータ）をメニーコアに含める場合もある。コア数が増えると、分散処理やコア間通信の管理が複雑になる。そのため、コア数の増加にスケーラブルに対応できるメニーコア向けの OS が開発されており、組込みシステム向けにはイーソルが製品化している。

■ モデル検査（model checking）

- 形式手法の一つ。ツールを使って、すべての状態と実行パスを網羅的に検証する。ユーザは、専用言語を使って検証対象となる仕様書やソースコードからモデルを作成し、併せてそのモデルが満たすべき条件式（検証式）を用意する。モデルが条件式を満たしているかどうかをツールが自動的に判定し、条件を満たしていない場合は反例（不具合が生じる実行パスの例）を出力する。モデル検査には、モデルの規模が大きくなると計算時間が非常に長くなるという課題がある。その場合は、モデルの分割や絞り込み、抽象化といった対策を施す必要がある。モデル検査ツールには、SMV（Symboric Model Verifier）や SPIN（Simple Promela Interpreter）など、多くの種類がある。

■ モデルベース開発（model based development）

- グラフィカル形式やテキスト形式のモデリング表記を用いて、開発の各工程の作業（要求分析、設計、検証・テストなど）を行う方法論。モデリングは、実現すべき機能性（開/閉ループの制御、監視）の概念の獲得、および現実の物理的システム（例えば、自動車の場合は車両環境）の挙動のシミュレーションなどに使用する。モデリング表記には、形式的（例えば、基礎となる数学的定義による表記）と、準形式的（例えば、不完全に定義された意味論を有する構造化表記）がある。モデルベース開発パラダイムの一つの特徴は、機能モデルが、その望まれている機能を仕様化するだけでなく、設計情報も提供する点にある。そして最後には、コード生成手段による実装の基盤も提供する。すなわち、そのような機能モデルは、設計および実装の側面だけでなく、仕様の側面も表している。モデリング表記法として、UML（Unified Modeling Language）や SysML（Systems Modeling Language）、米国 MathWorks 社の「MATLAB/Simulink」などがある。
- 同義語　MBD

■ モデルベース並列化（model based parallelization）

- 米国 MathWorks 社のシステム設計ツール「Simulink」で作成したブロック線図のモデルからマルチコア向けの並列プログラムを生成する技術。やみくもにソフトウェアの並列性を探索するので

はなく、モデルベース開発手法にのっとって作成されたブロック線図の構造を生かしながら並列化する箇所を決定する。ソースコードではなく抽象度の高いモデルの段階で並列化を検討することから、開発コストや開発期間の削減を期待できる。モデルベース並列化は、名古屋大学大学院 枝廣研究室が研究・開発しており、イーソルがその成果を利用した商用ツールを開発・販売している。

- 同義語　MBP

■ モデル予測制御（model predictive control）

- 動的な計算モデルを使って制御対象の未来の応答を予測し、その予測に基づいて最適制御する手法。性能や安全などの制約を反映しやすく、多入力の複雑なシステムにも適用できる、という特徴がある。ただし制御対象のふるまいを随時計算する必要があるので、PID（proportional integral differential）制御などと比べて計算量は多くなる。プラント制御やエンジン制御などに使われている。

9.7 ラ行，ワ行

■ リエントラント（re-entrant）

- あるプログラム（関数やサブルーチン、メソッド）の実行中に割り込み処理などが発生し、その実行が完了しないうちに、再びそのプログラムが呼び出されることがある。このように同一のプログラムが多重に起動し、並列実行が生じても、問題が発生しない性質を「リエントラント（再入可能）」と呼ぶ。

■ ロックステップ（lock-step）

- 「足並みをそろえた行進」を意味する言葉。コンピュータシステムの場合、二つ、またはそれ以上の CPU に同一のプログラムを実行させ、それらの計算結果を比較してシステムの異常を検出する耐故障（フォールトトレラント）設計の技術を指す。マルチコアプロセッサ内部の二つの CPU コアにこの技術を適用した場合、「デュアルコアロックステップ」と呼ぶ。航空機や自動車など、主に信頼性や機能安全に対する要求の厳しいシステムに利用されている。

■ ワイヤハーネス（wire harness）

- 複数の電線（ワイヤ）を束ねたケーブルやコネクタなどの配線部品を指す。機器や部品への電源供給、機器間の通信などに用いる。自動車や航空機では、ワイヤハーネスの重量が燃費や航続距離に

影響を与える。そのため、高速なシリアル通信（車載ネットワーク）を導入したり、複数の ECU（electronic control unit）を一つに集約したりするなどして、配線量の削減を図っている。

9.8 A~Z

- エーダス
- ADAS（advanced driver assistance systems）
  - 日本語は「先進運転支援システム」。自動車などの輸送システムにおいて、ドライバの運転操作の負担を軽減したり、事故を回避したりする機能全般を指す。例えば、ヘッドライト自動点灯、緊急時自動ブレーキ制御、歩行者・障害物検知、交通標識認識、車線逸脱警告、車線維持制御、先行車速度追随型のクルーズ制御、自動駐車支援などがある。可視光カメラや赤外線カメラ、ミリ波レーダ、LiDAR（レーザーレーダ）、光センサ、加速度/ジャイロセンサ、GPS などの各種センサを利用する。ADAS をベースに、自動車の周囲地図作成や走行ルート計画、行動方針決定などの機能を強化し、運転操作におけるドライバの関与を減らしていくと、自動運転の技術になる。

- エーエムピー
- AMP（asymmetrical multi-processing）
  - 日本語は「非対称型マルチプロセシング」。複数の CPU を使用した並列分散処理における、タスクの処理方式の一つ。各 CPU に対して、実行するタスクを事前に人手で（静的に）割り付ける方式。システムの実行中に OS が動的にタスクを振り分ける SMP（symmetrical multi-processing）とは異なり、タスク分割（機能分散）の方針を開発者自身が考えなければならない。リアルタイム制約や信頼性の要件が厳しい機器の場合、開発者がシステム全体のふるまいを見通しやすく、動作保証しやすい AMP 方式が好んで採用されている。ただし、CPU の数やタスクの数が多くなるにつれて、人手による適切なタスクの割り付けが難しくなる。
  - 対義語　SMP

- エーシル
- ASIL（Automotive Safety Integrity Level）
  - 「SIL（Safety Integrity Level）」は、電気電子システム全般を対象とする機能安全規格である IEC 61508 が規定する、システムが備えるべき安全性能の尺度。日本語では「安全性要求レベル」という。これに対して ASIL は、自動車向けの機能安全規格である ISO 26262 が規定する、車載電子システム（automotive）の安全性要求レベル。ASIL では、安全性要求の水準を A～D の 4 段階で示す（D がもっとも厳しく、A がもっともゆるい）。障害が発生したときの被害の大きさ

（severity）、障害の発生頻度（exposure）、障害に対する対処のしやすさ（controllability）をもとに算出する。

- ビーエムピー
- BMP（bound multi-processing）
  - 複数の CPU を使用した並列分散処理における、タスクの処理方式の一つ。SMP（symmetrical multi-processing）システムの中に、AMP（asymmetrical multi-processing）的なタスク割り付けの制約を取り込んでいる。一部のタスクは事前に人手で特定の CPU へ割り付け、それ以外のタスクは OS が CPU へ自動的に振り分ける。例えば、単一 CPU 上での実行を前提に開発された既存資産のタスクは同一の CPU 上で実行し、それ以外のタスクは、CPU の負荷に応じて OS が自動振り分けを行う、といった使い方が考えられる。

- ビーエスダブリュ
- BSW（basic software）
  - AUTOSAR（Automotive Open System Architecture）が規定する階層化アーキテクチャでは、上位の「サービス層（services layer）」、中位の「ECU 抽象化層（ECU abstraction layer）」、下位の「マイクロコントローラ抽象化層（MCAL：microcontroller abstraction layer）」、これらの階層とは別枠の「複合ドライバ（complex drivers）」の四つを合わせて、「BSW」と呼んでいる。AUTOSAR とは、ECU（車載電子制御ユニット）に搭載するソフトウェアの部品化・共通化を推進している業界団体の名称。サービス層には、OS、ネットワークの通信・管理、メモリ管理、システム診断などの機能が含まれる。ECU 抽象化層には、マイコンとインターフェースするための API（application programming interface）などが含まれる。マイクロコントローラ抽象化層には、マイコンが内蔵する周辺機能や、メモリマップされた外部のデバイスへアクセスするためのデバイスドライバが含まれる。複合ドライバはマイコンへ直接アクセスするソフトウェアモジュールで、センサやアクチュエータなどを操作するための特殊な機能を実装するために用いる。

- キャン
- CAN（Controller Area Network）
  - 自動車や船舶、産業用機器、医療機器などで使われているシリアル通信プロトコル。ドイツ Robert Bosch 社が 1986 年に仕様を公開し、1994 年に ISO（International Organization for Standardization）が標準化した。通信速度は最大 1Mbps。OSI（Open System

Interconnection）参照モデルのトランスポート層（再送制御）、データリンク層（メッセージのフレーム化、アービトレーション、エラー検出など）、物理層（ビットタイミング、同期方式など）が定義されている。2012 年には Robert Bosh 社が、CAN の拡張版である「CAN FD（CAN with Flexible Data-Rate）」の仕様を公開。2015 年に ISO の規格にも反映された。CAN FD ではデータ転送速度が可変になり、最大 2M～5Mbps 程度の通信に対応できるようになった。

- シーシーエフ
- CCF（common cause failure）
  - 日本語は「共通原因（による）故障」。複数の系統が存在するシステムにおいて、共通の根本原因によって複数の系統が損なわれる故障を指す。安全性を考慮してシステムを二重化・冗長化した場合でも、同じ部品や同じ回路、同じ電源、同じソフトウェアを用いていると、複数の系統が故障する可能性がある。多くの機器やシステムに影響を与える CCF は重大事故の原因となることが多い。そのため、機能安全設計やリスクアセスメントでは、CCF に配慮する必要がある。

- ディーアールピー
- DRP（dynamic reconfigurable processor）
  - 日本語は「動的再構成可能プロセッサ」。回路の動作中に任意のタイミングで、演算回路の一部（または全部）の回路構成を変更できるようにしたディジタル LSI の総称。「時刻 A には演算回路 A」、「時刻 B には演算回路 B」、…というように、時間によって異なる演算回路をチップ上に展開する。当初は、FPGA が内蔵するプログラマブル論理の構造やアイデアをもとに研究・開発が進んだ。最近では、メニーコア的に小規模な演算素子や演算ユニットをチップ上に多数配置し、それらをフレキシブルに組み合わせて任意のパイプライン演算回路を構成するものが多い。従来の演算回路の設計に加えて、時系列の回路の切り替えを考慮する必要があるので、設計は複雑になる。そのため開発元は、所望の機能やアルゴリズムを記述した C 言語プログラムを分割・スケジューリングして、ターゲットとなる DRP へ展開するコンパイラ（回路合成ツール）を用意している。ルネサスエレクトロニクスや東京計器などが DRP 製品の開発・事業化を手がけている。FPGA（field programmable gate array）の中にも、動的再構成の機能を正式にサポートしているものが多い。

- イーシーユー
- ECU（electronic control unit）

- 日本語は「(車載）電子制御ユニット」。自動車に搭載する組込みコンピュータ全般を指す。もともと排出ガスを減らしたり、燃費を改善したりするため、エンジンの点火や燃料噴射のタイミングを制御する目的で導入が進んだ。現在では、変速装置（トランスミッション）や ABS（anti-lock brake system)、エアバッグ制御、ヘッドライト制御、ミラー制御、ADAS（advanced driver assistance systems）など、自動車の中のあらゆる電子制御に用いられている。ECU の内部には、車載専用のマイクロコントローラとその周辺回路、通信制御回路、診断回路などが実装されている。ECU 間の通信には、CAN や LIN、FlexRay などの車載ネットワーク規格が用いられる。自動車の内部は温度条件や振動条件、ノイズ環境が劣悪。加えて、ECU の誤動作は人命に影響を与える可能性がある。そのため、ECU には高度の信頼性と安全性が要求される。高級車の中には、100 個以上の ECU が搭載されているものもある。

■ エフピージーエー

■ FPGA（field programmable gate array）

- ユーザの手元で回路構成を変更（プログラム）できるディジタル LSI。高額なマスク代や開発費を支払わなくても、ユーザ仕様のカスタム LSI を入手できる、という特徴がある。FPGA は、多数の論理素子や論理ブロックと、それらの間を結線するプログラム素子（SRAM セルなど）を組み合わせて、大規模な論理回路（プログラマブル論理）を実現する。現在はプログラマブル論理に加えて、メモリブロックや乗算器ブロック、高速シリアル伝送用の SerDes（serialization/deserialization）ブロック、CPU コア、通信インターフェース回路なども集積した SoC タイプの FPGA が増えている。FPGA ベンダは、所望の回路をユーザが HDL（ハードウェア記述言語）で記述すると、FPGA に回路を書き込むためのコンフィグレーションデータを生成する統合開発環境を提供している。最近では、ビヘイビア合成ツールを使って、C 言語プログラム（アルゴリズム）から FPGA に実装する回路を生成するケースも増えている。米国 Cypress Semiconductor 社や米国 Intel 社（旧 Altera 社)、米国 Lattice Semiconductor 社、米国 Microsemi 社（旧 Actel 社、現在は Microchip Technology 社の一部門)、米国 Xilinx 社などが FPGA 製品を開発・販売している。

■ ジーキューエム

■ GQM（Goal-Question-Metric）

- ソフトウェア開発にかかわる計測の枠組み、およびモデル化の手法を指す。University of Maryland の Victor R. Basili 教授が提唱した。GQM モデルでは、計測の目標（Goal)、目標を達

成したかどうかを評価するための質問（Question）、質問に定量的に答えるためのデータ（Metric）の 3 階層が定義されている。

イソ ニロクニロクニ

- ISO 26262（International Organization for Standardization 26262）
  - 車載電子システム向けの機能安全規格。電気電子システム全般を対象とする IEC 61508 をたたき台として策定が進み、ISO（International Organization for Standardization）が 2011 年 11 月に正式発行した。ISO 26262 は、機能安全設計にかかわる組織や人員、安全性要求レベル（ASIL）、システム/ハードウェア/ソフトウェアレベルの製品開発、生産、安全性分析などの要件を細かく定義している。

- ジス エックス ニマンゴセンジュウ
- JIS X 25010（Japanese Industrial Standard X 25010）
  - JIS（日本工業規格）が 2013 年に公開した、システムおよびソフトウェアの品質モデルを規定した規格。2011 年に発行された ISO/IEC 25010 をもとに、技術的内容や構成を変更することなく、そのまま JIS 規格とした。製品品質として八つの特性（機能適合性、性能効率性、互換性、使用性、信頼性、セキュリティ、保守性、移植性）を、利用時の品質として五つの特性（有効性、効率性、満足性、リスク回避性、利用状況網羅性）を定義している。製品開発の視点に加えて、利用者の視点で品質を考えることを求めている。これとは別に、JIS X 25012（ISO/IEC 25012）ではデータ品質のモデルも規格化されている。

- エムビーディー
- MBD（model based development）
  - → 「モデルベース開発」を参照。

- エムビーディー
- MBP（model based parallelization）
  - → 「モデルベース並列化」を参照。

- エムキャル
- MCAL（microcontroller abstruction layer）

- 日本語では「マイクロコントローラ抽象化層」。AUTOSAR（Automotive Open System Architecture）が規定する階層化アーキテクチャの階層の一つで、ソフトウェアの中では最下層（マイコンの上）に位置する。MCAL には、マイコンが内蔵する周辺機能や、メモリマップされた外部のデバイスへアクセスするためのデバイスドライバが含まれる。例えば、マイコンのクロックやタイマ、入出力ポートなどの設定を行うドライバ、メモリや通信を制御するドライバなどがある。

■ キューエム

■ QM（quality management）

- 日本語では「品質管理」。製品の質が、決められたある水準を満たすように、製品を検査したり、開発や製造の工程を管理したりすること。ISO 26262 の ASIL（車載電子システムの安全性要求レベル）では、機能安全を適用する必要がない通常の品質管理のレベル（ASIL A より安全性要求がゆるいレベル）を「QM」と呼んでいる。

■ シム

■ SHIM（Software-Hardware Interface for Multi-many-core）

- マルチコアプロセッサごとに異なるアーキテクチャの情報を XML で記述するための標準仕様。マルチコア技術の普及推進団体である米国 Multicore Association が策定し、2015 年 2 月に公開した。SHIM 記述には、プロセッサコアの種類と数、メモリの容量と性能、共有メモリの有無、キャッシュメモリの構造など、マルチコアソフトウェア開発の観点で必要となる情報を記載する。例えばマルチコア対応 OS や並列化コンパイラ、マルチコア対応の性能解析ツールへ、ターゲットとなるプロセッサのアーキテクチャ情報を提供するために用いる。SHIM のコンセプトは、独立行政法人 新エネルギー・産業技術総合開発機構（NEDO）が 2011 年 11 月に採択したプロジェクト「多様なマルチ・メニーコアの高度な活用を可能にする標準プラットフォーム開発とエコシステム構築による省エネルギー技術の実用化」がベースになっている。

■ エスエムピー

■ SMP（symmetrical multi-processing）

- 日本語は「対称型マルチプロセシング」。複数の CPU を使用した並列分散処理における、タスクの処理方式の一つ。システムの実行中に、OS が各 CPU の負荷を考慮しつつ、動的にタスクを CPU へ振り分ける。すべての CPU を等しく扱うという意味で、「対称型」と呼ぶ。SMP では、

システム全体を一つの OS が統括する。CPU の空き状態や無駄な待ち状態が少なく、計算リソースを有効活用しやすい。反面、開発者の意図しない順序でタスクを実行する可能性があり、リアルタイム制約や信頼性の要件が厳しい機器に適用する場合は、システムの設計や検証を注意深く行う必要がある。

- 対義語　AMP

■ エスピーエフ

■ SPF（software platform）

- 日本語は「ソフトウェアプラットフォーム」。OS やミドルウェア、デバイスドライバ、ライブラリなど、多くのユーザが共通的に使うソフトウェアで、システムの土台（プラットフォーム）部分に位置するもの（アプリケーションソフトウェア以外）を指す。AUTOSAR（Automotive Open System Architecture）を取り扱う車載電子システムや車載用半導体の業界では、BSW（basic software）、RTE（runtime environment）、ユーザ側で開発する API（application programming interface）のライブラリなどを含めて、「SPF」と呼ぶこともある。BSW は、OS やミドルウェア、デバイスドライバに相当するソフトウェア群を指す。RTE はソフトウェア部品（SW-C）間、およびソフトウェア部品と BSW の間で通信するためのインターフェースソフトウェアで、RTE ジェネレータと呼ぶツールを使って生成する。