Simulinkモデルベース自動並列化とSHIM

- Simulinkブロックレベル並列化
- ブロック内自動並列化は従来から多くの研究あり（例：早大OSCAR）
- 本発表では車載制御を仮定
  - 自動運転に関しては加藤真平准教授から

名古屋大学大学院情報科学研究科
枝廣　正人

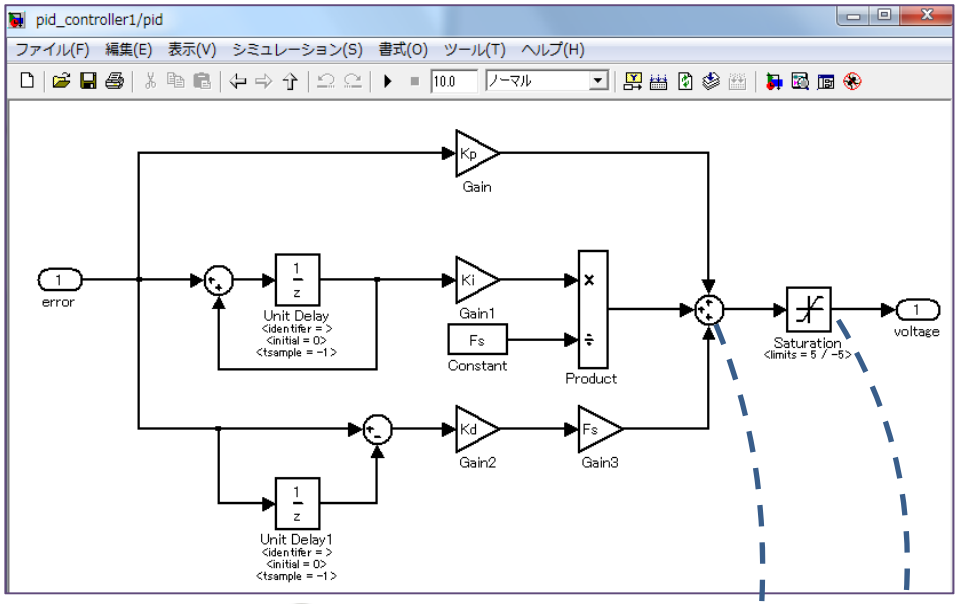
アウトライン

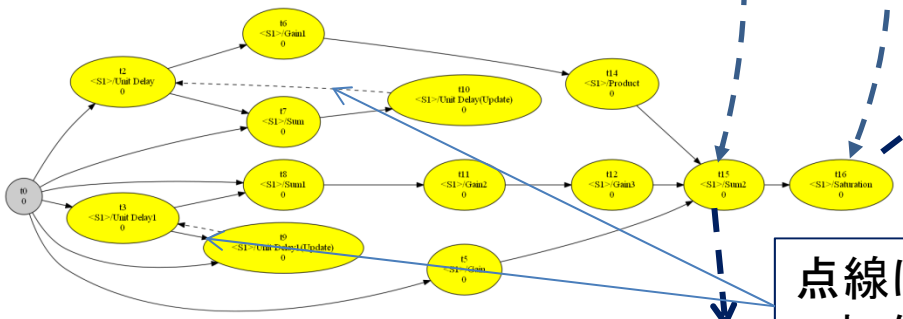
- Simulinkモデルベース開発とコード生成の特徴
- SHIMを用いたSimulinkモデルからの並列コード生成
- 並列アーキテクチャ向け最適化

Simulinkモデルベース開発とコード生成の特徴

- Simulinkモデルとコード生成
  （一つのブロックにMATLABまたはCで記載した場合と比べ）
  - ブロックレベルの構造が明確
  - 通信は単方向
  - ブロック内の粒度（処理量）が小さい。ほとんどが一行の数式。例外：飽和、Enable, 数値積分など
  - 離散系の場合、1ステップ（サンプル周期、積分などの時間刻み幅等）の処理は「入力または遅延素子」から開始し「遅延素子または出力」で終わる
    - 多くの場合、遅延素子を「入力から記憶部」と「記憶部から出力」に分離することにより、（かつ有限回転ループ展開により）有限の無閉路有向グラフ（DAG (Directed Acyclic Graph) ）となる
  - （Mathworks社のCoderでは）ブロック（タスク）間依存関係は明示的な通信のみ（Data Storeなど特殊なブロックを使用した場合を除く）
  - 制御設計者は、構造はともかく数式までばらばらにすることには抵抗感

例：Simulinkモデルで記載されたPID制御





```
/* Saturate: '<S1>/Saturation' */
 if (pid_controller1_pid_Sum2_1 >=
     pid_controller1_P.Saturation_UpperSat) {
   pid_controller1_pid_Saturation_1 =
     pid_controller1_P.Saturation_UpperSat;
 } else if (pid_controller1_pid_Sum2_1 <=
     pid_controller1_P.Saturation_LowerSat)
 {
   pid_controller1_pid_Saturation_1 =
     pid_controller1_P.Saturation_LowerSat;
 } else {
   pid_controller1_pid_Saturation_1 =
     pid_controller1_pid_Sum2_1;
 }
```

点線は、遅延素子の入力側と出力側をつなぐ線。
これを除けばDAGになっていることがわかる

```
/* Sum: '<S1>/Sum2' */
 pid_controller1_pid_Sum2_1 = (pid_controller1_pid_Gain_1 +
   pid_controller1_pid_Product_1) + pid_controller1_pid_Gain3_1;
```

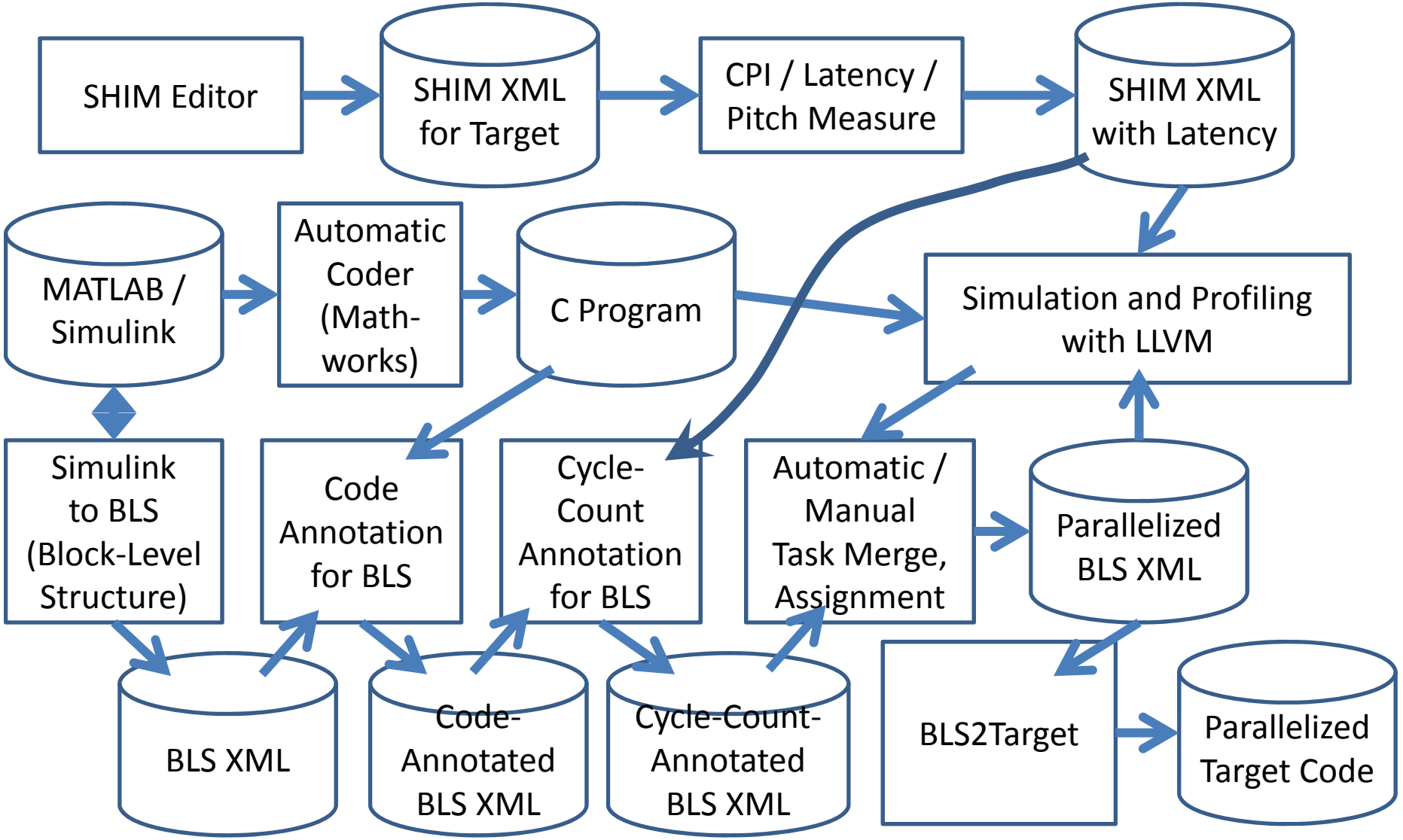
並列化の意味で良い点、難しい点

- 良い点（現在我々が見ている範囲で）
  - 静的解析、実行時間見積が進めやすい
    - μs～msオーダーの制御周期を守るように設計されているので、回転数不明のループや再帰呼び出しがない（注：デッドラインで打ち切る処理は存在）
    - Simulinkが生成するコードは、静的解析が難しい依存関係やポインタがない。依存関係はすべてブロック間の結線で記載されている
    - ブロック特有の制御構造がわかる（飽和演算、Enable信号、数値積分等）
- 難しい点
  - 各ブロックの粒度が小さく、通信オーバーヘッドの影響が強く出る
  - 並列性が低い（これはモデルベースが理由ではなく、もとの制御問題の特性）

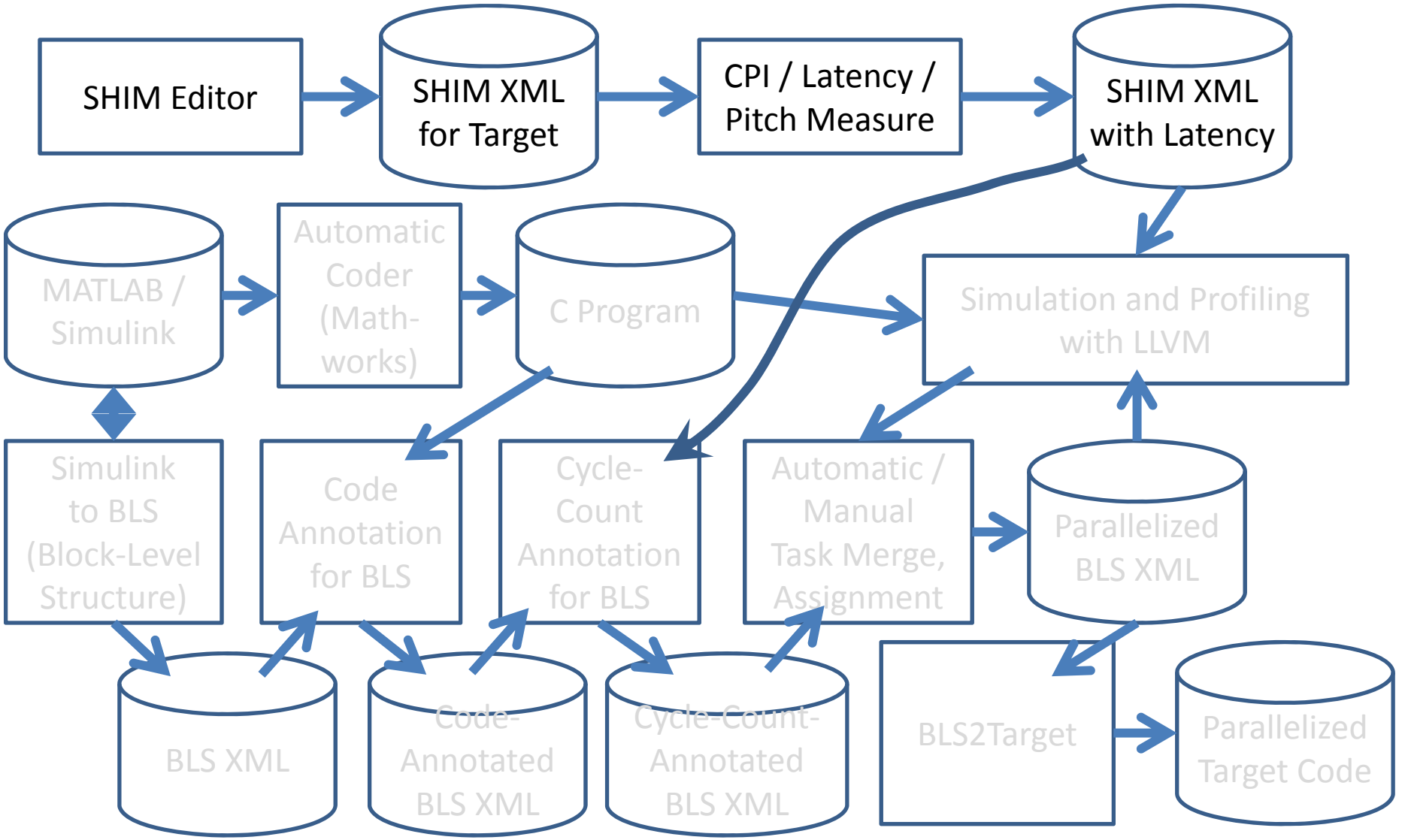
アウトライン

- Simulinkモデルベース開発とコード生成の特徴
- SHIMを用いたSimulinkモデルからの並列コード生成
- 並列アーキテクチャ向け最適化

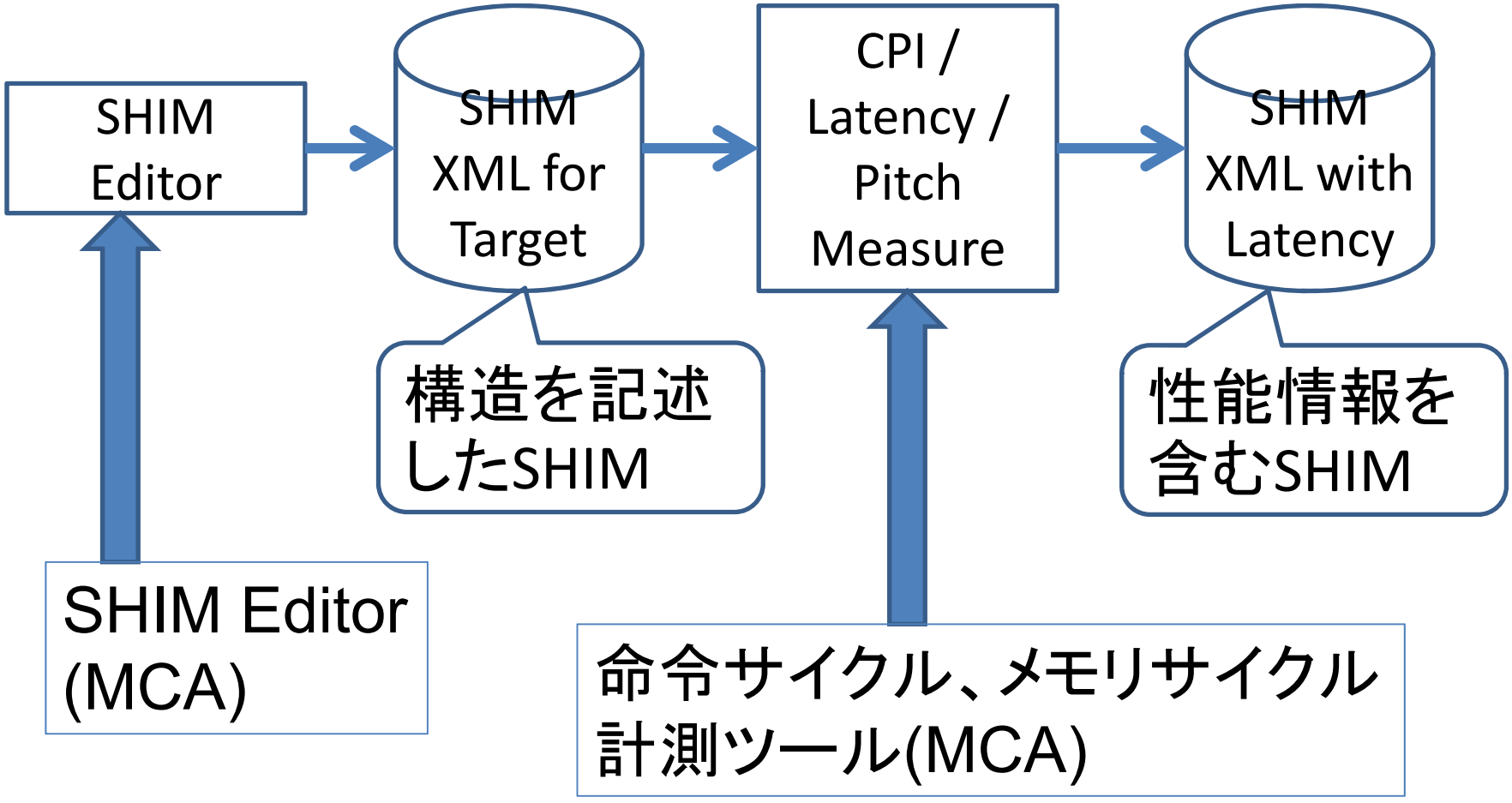
MATLAB / Simulinkからの並列化フロー



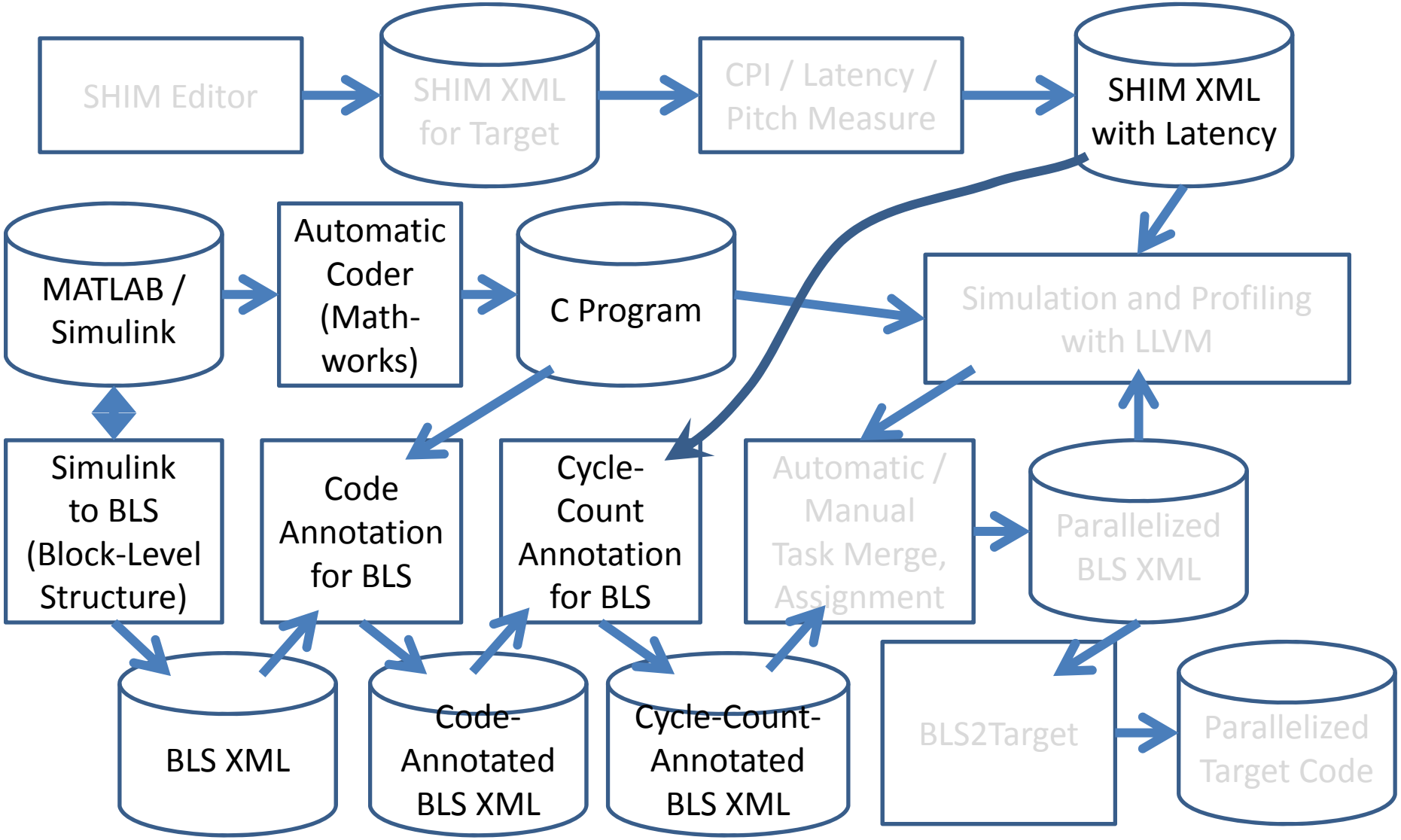
ターゲットプロセッサ向けSHIM生成



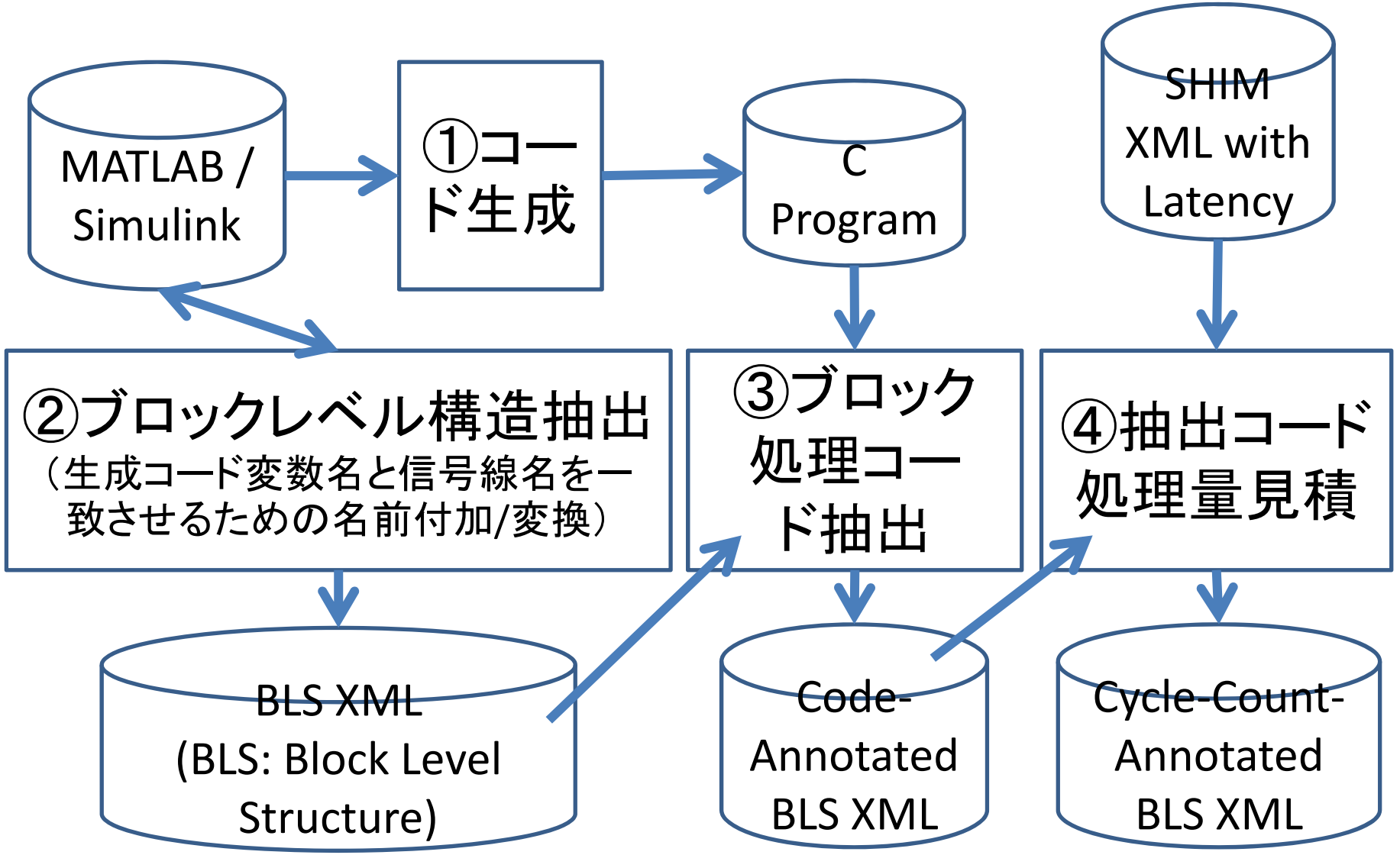
ターゲットプロセッサ向けSHIM生成



Simulinkモデルからの並列構造抽出

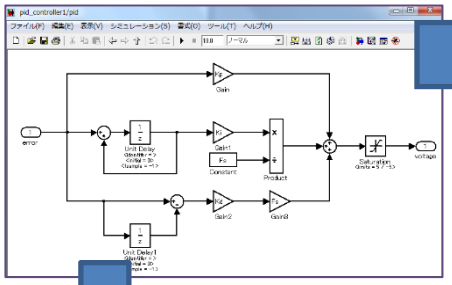


Simulinkモデルからの並列構造抽出



Simulinkモデルからの並列構造抽出

Simulinkモデル



②ブロックレベル構造抽出 (CSPグラフ構造)

①コード生成

```
/* Saturate: '<S1>/Saturation' */
if (pid_controller1_pid_Sum2_1 >= p
  pid_controller1_pid_Saturation_1
} else if (pid_controller1_pid_Sum2
{
  pid_controller1_pid_Saturation_1
} else {
  pid_controller1_pid_Saturation_1
}

/* End of Saturate: '<S1>/Saturatio

/* Sum: '<S1>/Sum' incorporates:
 *  Inport: '<Root>/error'
```

③ブロック処理コード抽出

```
<block blocktype="Saturate" name="pid_controller1_pid
  <input line="pid_controller1_pid_Sum2_1" port="pid_
    <connect block="pid_controller1_pid_Sum2" port="p
  </input>
  <output line="pid_controller1_pid_Saturation_1" port
    <connect block="pid_controller1_pid_voltage" port
  </output>
  <var line="pid_controller1_pid_Sum2_1" mode="input"
  <var line="pid_controller1_pid_Saturation_1" mode="
  <param name="Saturation_UpperSat" storage="pid_cont
  <param name="Saturation_LowerSat" storage="pid_cont
  <code file="models/pid/pid_controller1_ert_rtw/pid_
    /* Saturate: '&lt;S1&gt;/Saturation' */
    if (pid_controller1_pid_Sum2_1 &gt;= pid_controlle
      pid_controller1_pid_Saturation_1 = pid_controlle
    } else if (pid_controller1_pid_Sum2_1 &lt;= pid_co
    {
      pid_controller1_pid_Saturation_1 = pid_controlle
    } else {
      pid_controller1_pid_Saturation_1 = pid_controlle
    }
    /* End of Saturate: '&lt;S1&gt;/Saturation' */
  </code>
  <code file="models/pid/pid_controller1_ert_rtw/pid_
    pid_controller1_pid_Saturation_1 = 0.0F;
  </code>
  <performance best="26" type="task" typical="31" wors
  <performance best="15" type="init" typical="15" wors
  <forward block="pid_controller1_voltage" type="port"
    <var line="pid_controller1_pid_1" mode="input" na
  </forward>
  <backword block="pid_controller1_pid_Sum2" type="dat
    <var line="pid_controller1_pid_Sum2_1" mode="outpu
  </backword>
</block>
```

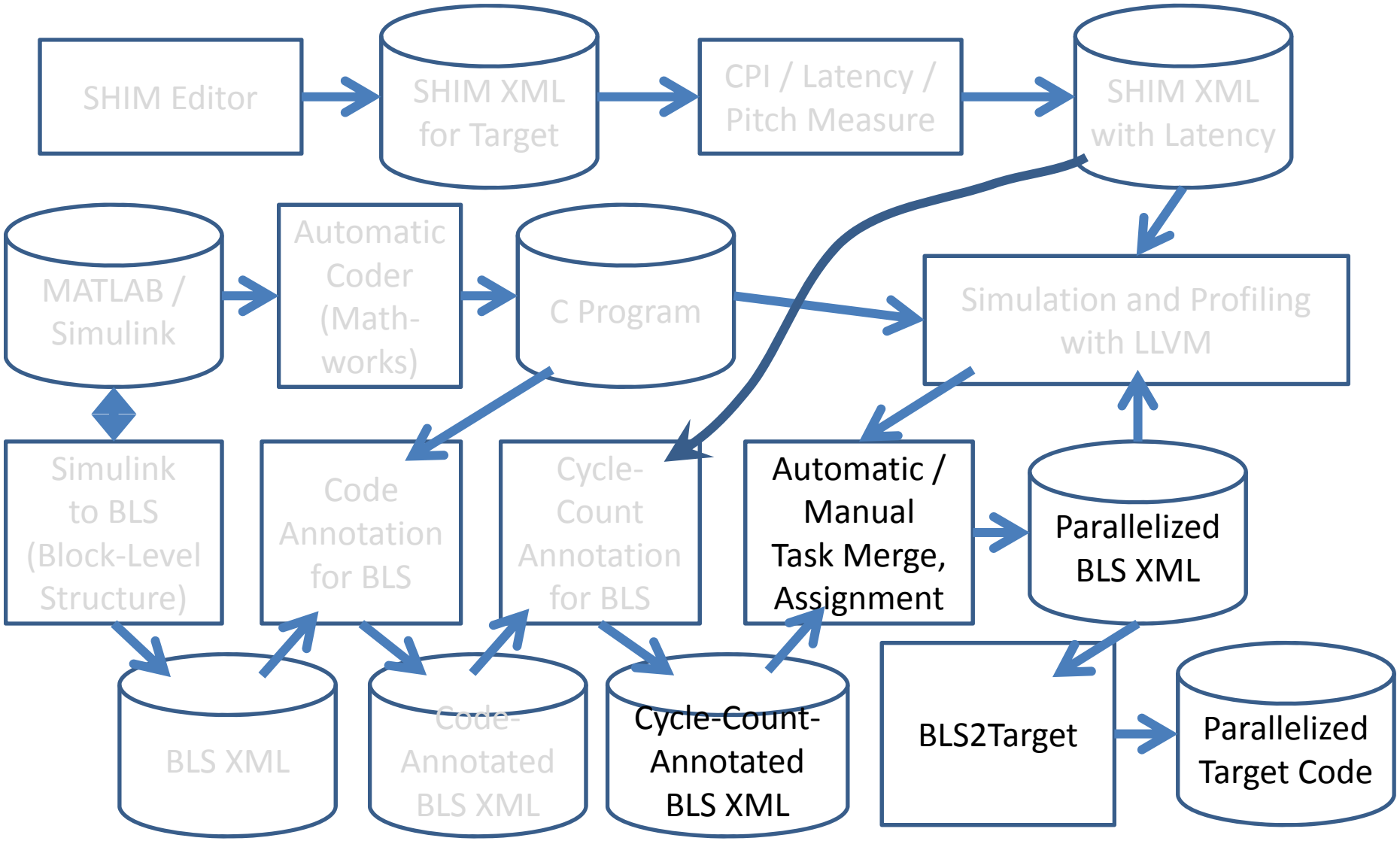
④抽出コード処理量見積

```
<ComponentSet name="Board_BO">↓
    <ComponentSet name="Cluster_BOCO">↓
        <ComponentSet name="PE_BOCOP1">↓
            <SlaveComponent name="LRAM_BOCOP1"
            <MasterComponent name="CPU_BOCOP1"
                <CommonInstructionSet name="
                    <Instruction name="ret">
                        <Latency best="10.0"
                        <Pitch best="10.0" typical="10.0"
                    </Instruction>↓
                    <Instruction name="br">↓
```

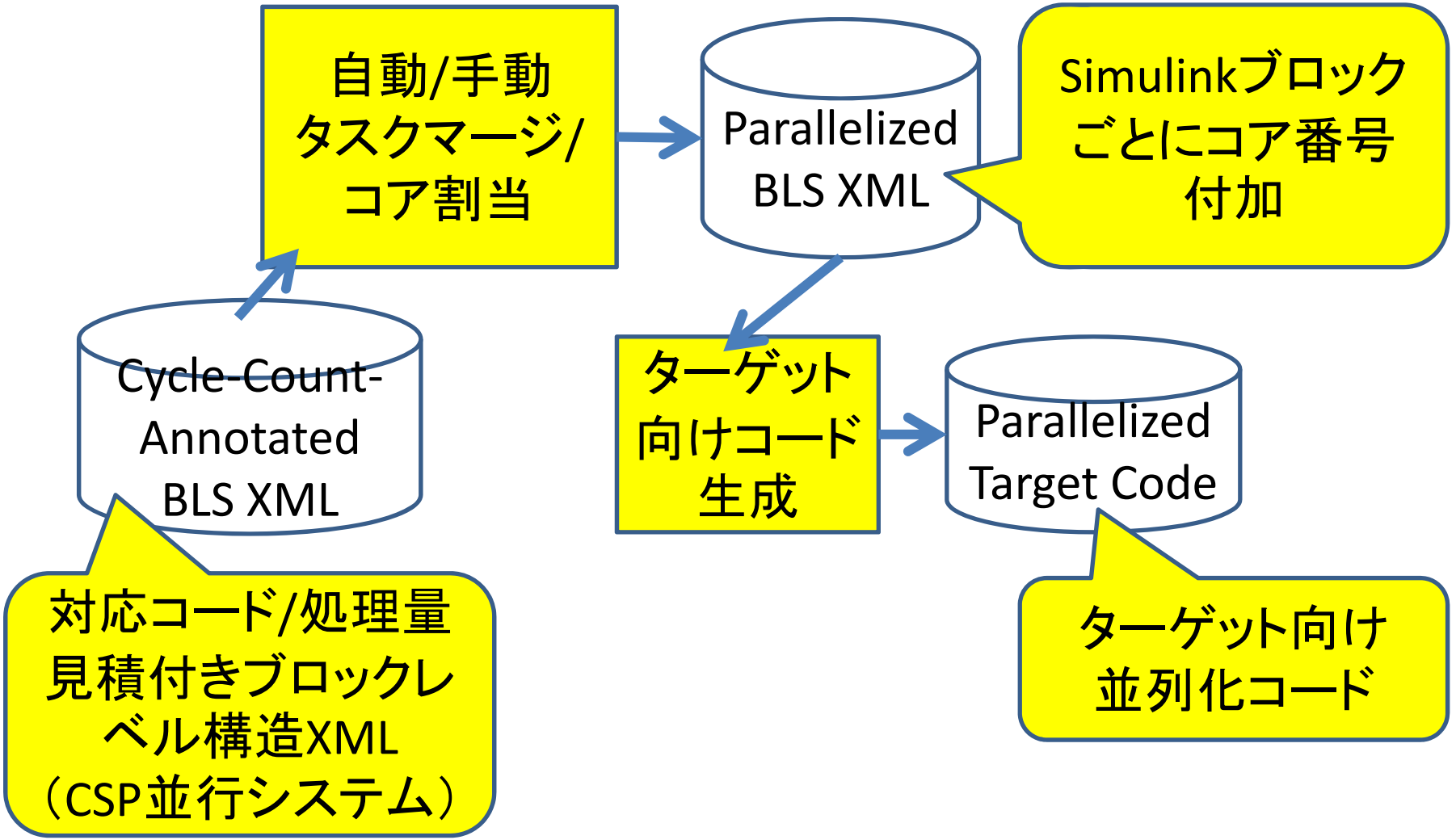
SHIM

ブロックレベル構造XML (BLS-XML)

BLS-XMLからの並列コード生成



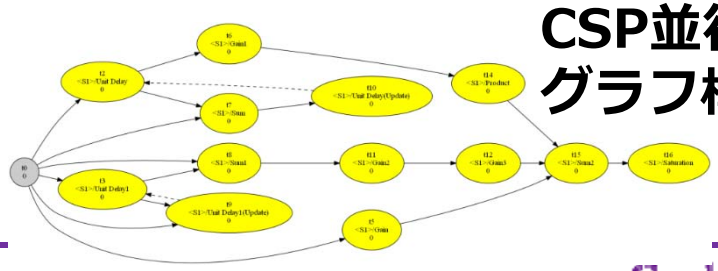
BLS-XMLからの並列コード生成



BLS-XMLからの並列コード生成

- ブロックレベル構造XML
  - CSP並行システム
    (Concurrent Sequential Processes)
  - ブロックごとのコード
  - ブロックごとの処理量見積

  - 複数ブロックを併合してタスク化しコード生成
  - タスクのコア割当

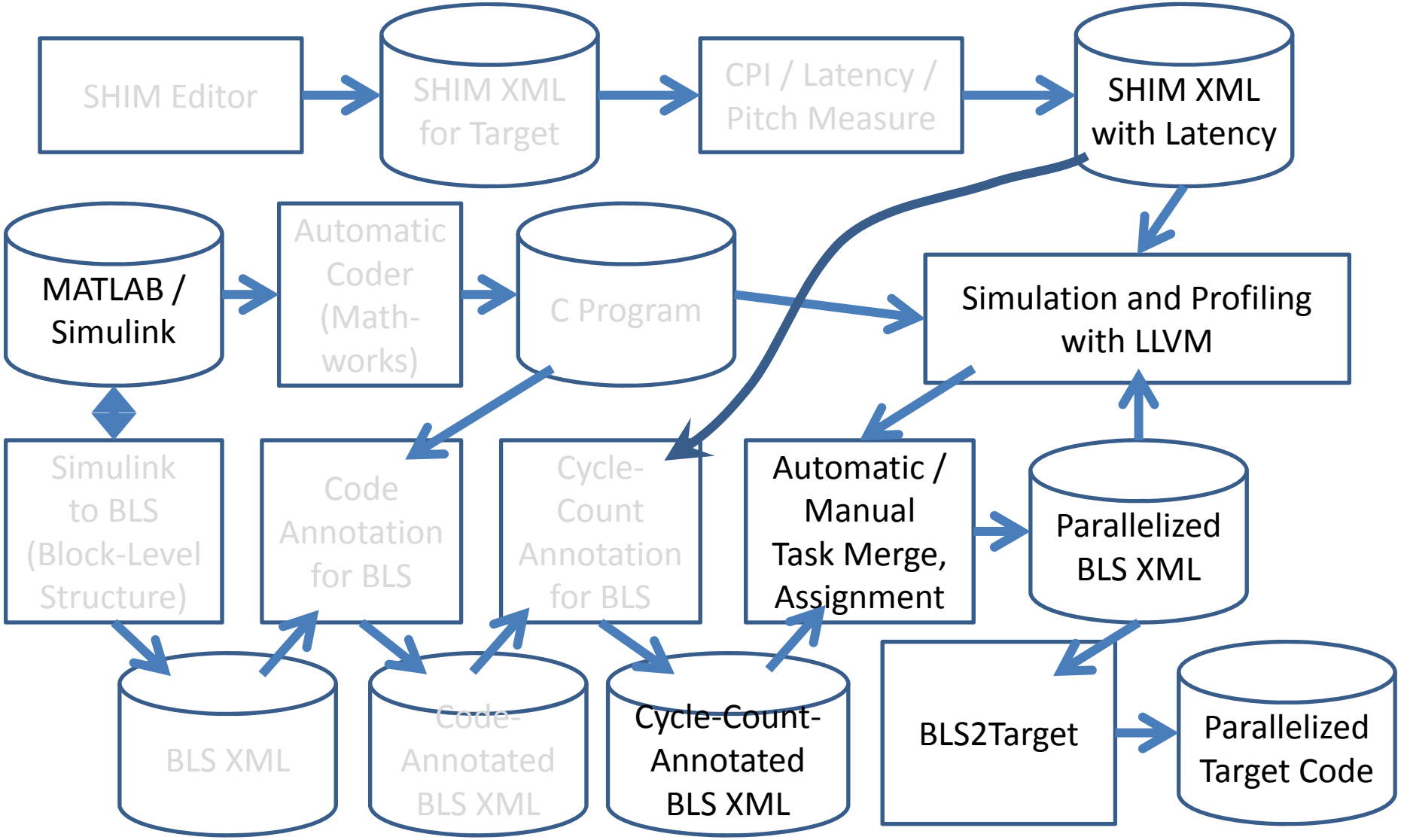


**CSP並行システム
グラフ構造**

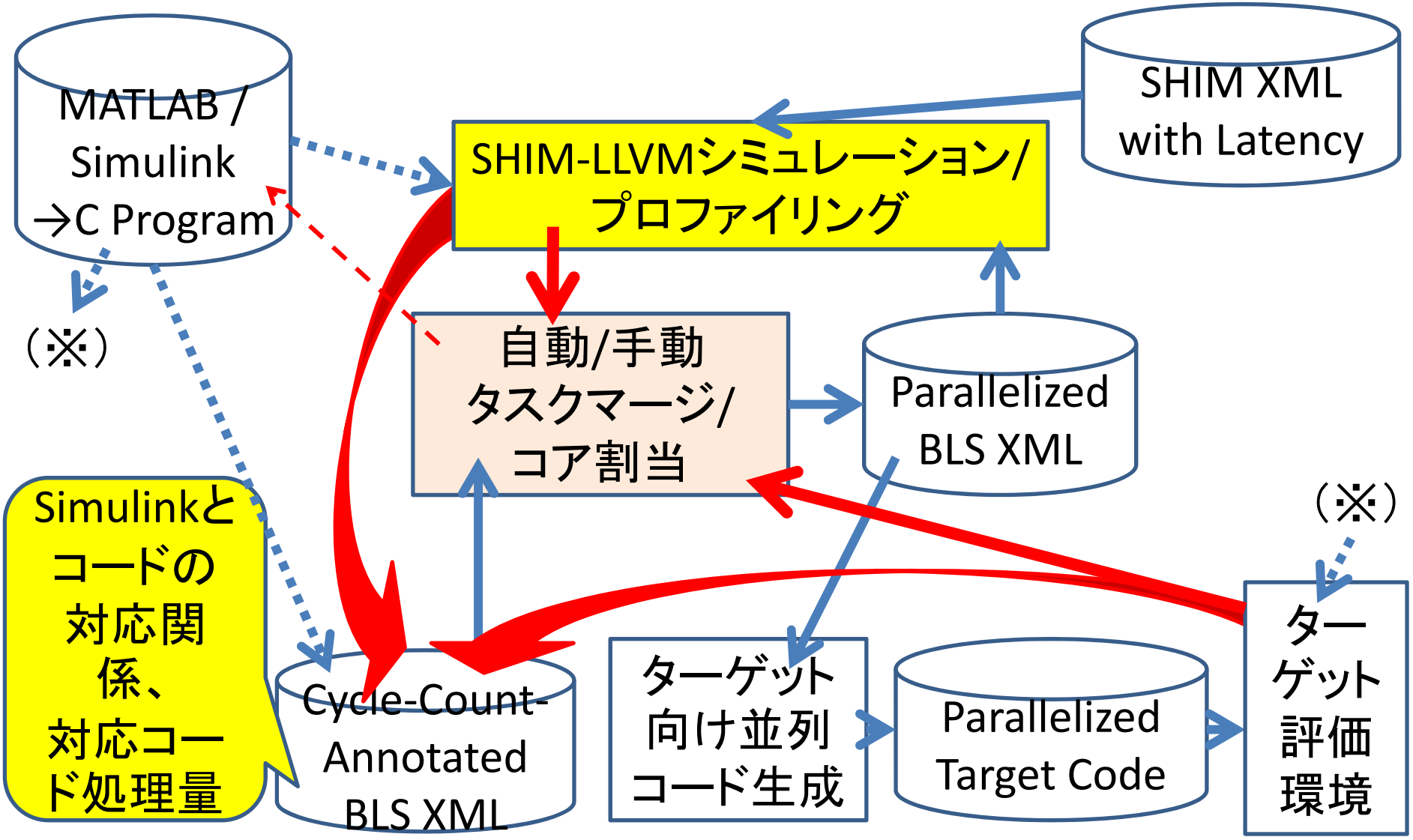
```
<block blocktype="Saturate" name="pid_controller1_pid_
  <input line="pid_controller1_pid_Sum2_1" port="pid_
    <connect block="pid_controller1_pid_Sum2" port="p
  </input>
  <output line="pid_controller1_pid_Saturation_1" port
    <connect block="pid_controller1_pid_voltage" port=
  </output>
  <var line="pid_controller1_pid_Sum2_1" mode="input"
  <var line="pid_controller1_pid_Saturation_1" mode="
  <param name="Saturation_UpperSat" storage="pid_cont
  <param name="Saturation_LowerSat" storage="pid_cont
  <code file="models/pid/pid_controller1_ert_rtw/pid_
    /* Saturate: '&lt;S1&gt;/Saturation' */
    if (pid_controller1_pid_Sum2_1 &gt;= pid_controlle
      pid_controller1_pid_Saturation_1 = pid_controlle
    } else if (pid_controller1_pid_Sum2_1 &lt;= pid_co
    {
      pid_controller1_pid_Saturation_1 = pid_controlle
    } else {
      pid_controller1_pid_Saturation_1 = pid_controlle
    }
    /* End of Saturate: '&lt;S1&gt;/Saturation' */
  </code>
  <code file="models/pid/pid_controller1_ert_rtw/pid_
    pid_controller1_pid_Saturation_1 = 0.0F;
  </code>
  <performance best="26" type="task" typical="31" wors
  <performance best="15" type="init" typical="15" wors
  <forward block="pid_controller1_voltage" type="port"
    <var line="pid_controller1_pid_1" mode="input" nam
  </forward>
  <backword block="pid_controller1_pid_Sum2" type="dat
    <var line="pid_controller1_pid_Sum2_1" mode="outpu
  </backword>
</block>
```

**ブロックレベル
構造XML (BLS-XML)**

最適化フェーズ

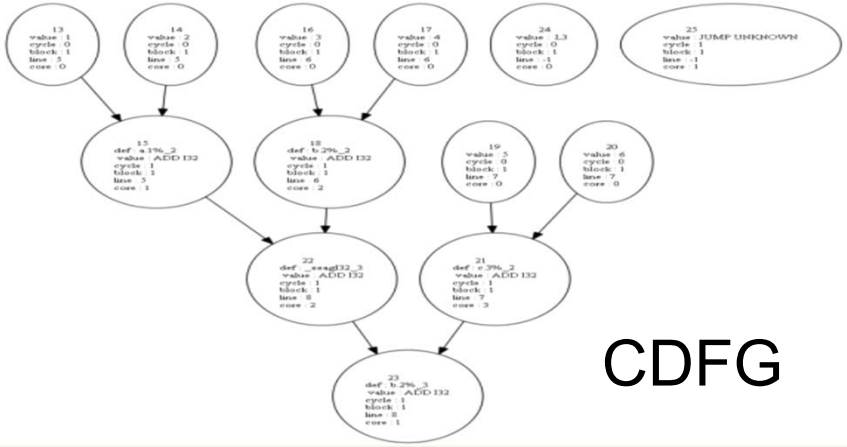

SHIM Editor
SHIM XML for Target
CPI / Latency / Pitch Measure
SHIM XML with Latency
MATLAB / Simulink
Automatic Coder (Math-works)
C Program
Simulation and Profiling with LLVM
Simulink to BLS (Block-Level Structure)
Code Annotation for BLS
Cycle-Count Annotation for BLS
Automatic / Manual Task Merge, Assignment
Parallelized BLS XML
BLS XML
Code-Annotated BLS XML
Cycle-Count-Annotated BLS XML
BLS2Target
Parallelized Target Code

最適化フェーズ


MATLAB / Simulink →C Program
（※）
SHIM XML with Latency
SHIM-LLVMシミュレーション/プロファイリング
自動/手動 タスクマージ/ コア割当
Parallelized BLS XML
Simulinkとコードの対応関係、対応コード処理量
Cycle-Count-Annotated BLS XML
ターゲット向け並列コード生成
Parallelized Target Code
（※）
ターゲット評価環境

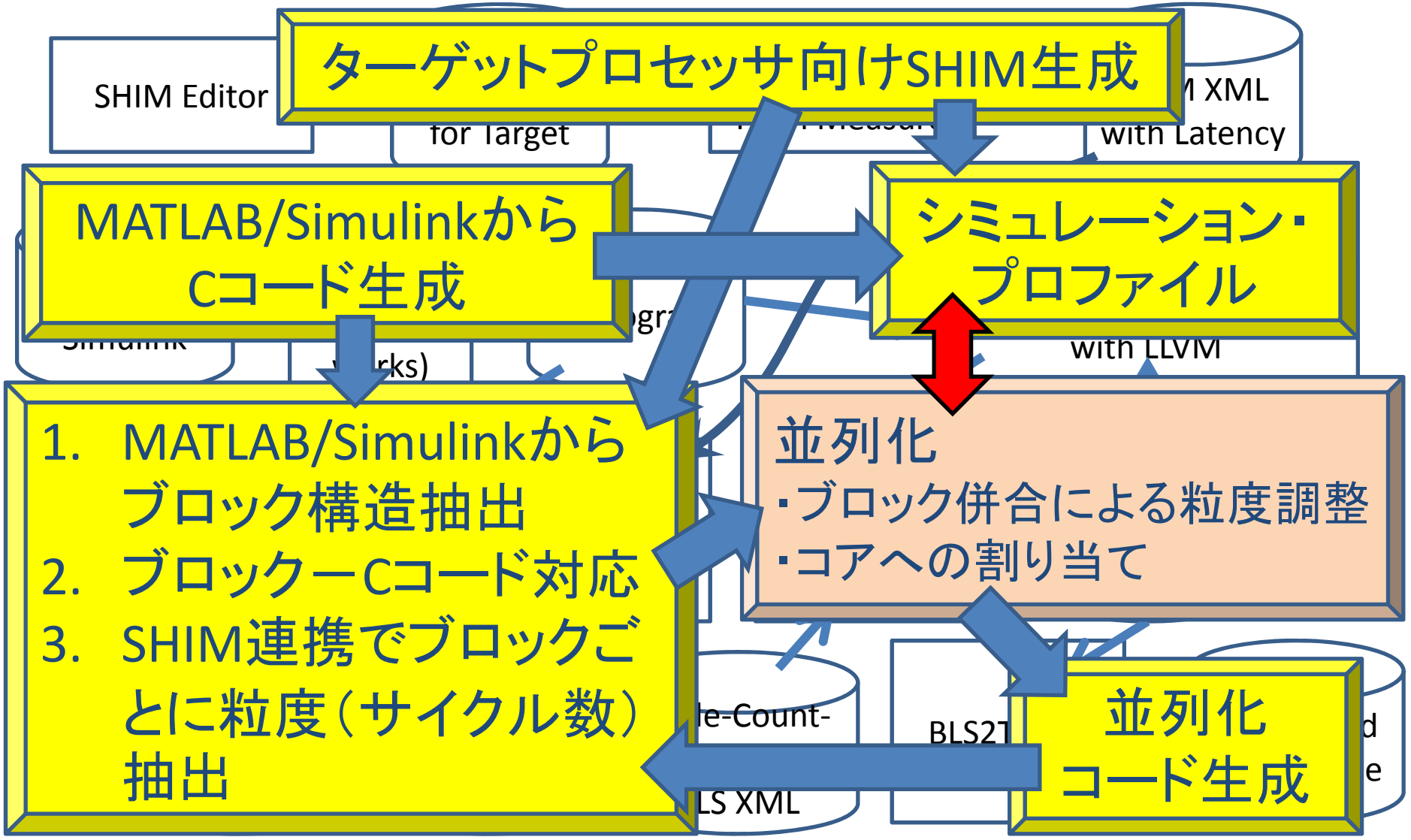
SHIM-LLVMシミュレーション/プロファイリング

1. LLVM中間言語レベルまでコンパイル
2. シミュレーション/プロファイリング
   - 動的手法
     - LLVMプロファイラにより、LLVM中間言語レベルで命令ごとの実行回数を測定
     - SHIM性能情報、コア割当情報を用いて見積
   - 静的手法
     - CDFG（制御/データフローグラフ）抽出
     - SHIM性能情報、コア割当情報を用いて見積



CDFG

MATLAB / Simulinkからの並列化フロー



アウトライン

- Simulinkモデルベース開発とコード生成の特徴
- SHIMを用いたSimulinkモデルからの並列コード生成
- 並列アーキテクチャ向け最適化

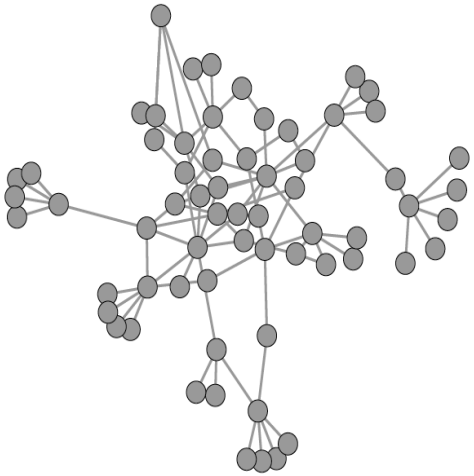
２種類のアーキテクチャ

- １対多アーキテクチャ
  - 1コアに複数タスク、コア内でリアルタイム・スケジューリング（名古屋大学高田研究室でも精力的に研究）
  - コア割当に関しては歴史的に数多くの研究が存在
  - 階層クラスタリング手法を用いたタスクマージ、コア割当

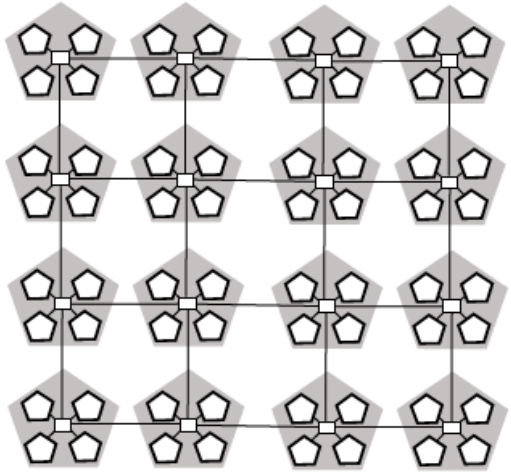
- １対１アーキテクチャ
  - 1コア1タスク、コア内スケジューリング、優先度設計不要
  - 限定された通信
  - シングルレート、決定的な計算順序を保つようにタスクマージ

階層クラスタリングによるタスク配置

- LSI配置問題において,構成要素を効率良くマッピングする手法である階層クラスタリング法を、階層構造を持つマルチコア向けタスク配置手法に発展
  - 実行順序などタスク配置特有の問題を考慮

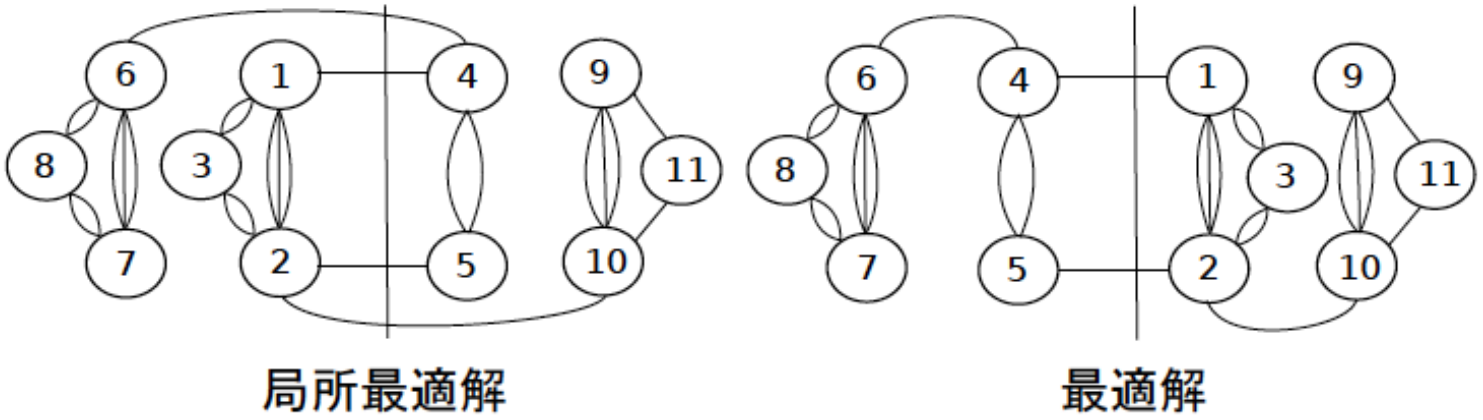


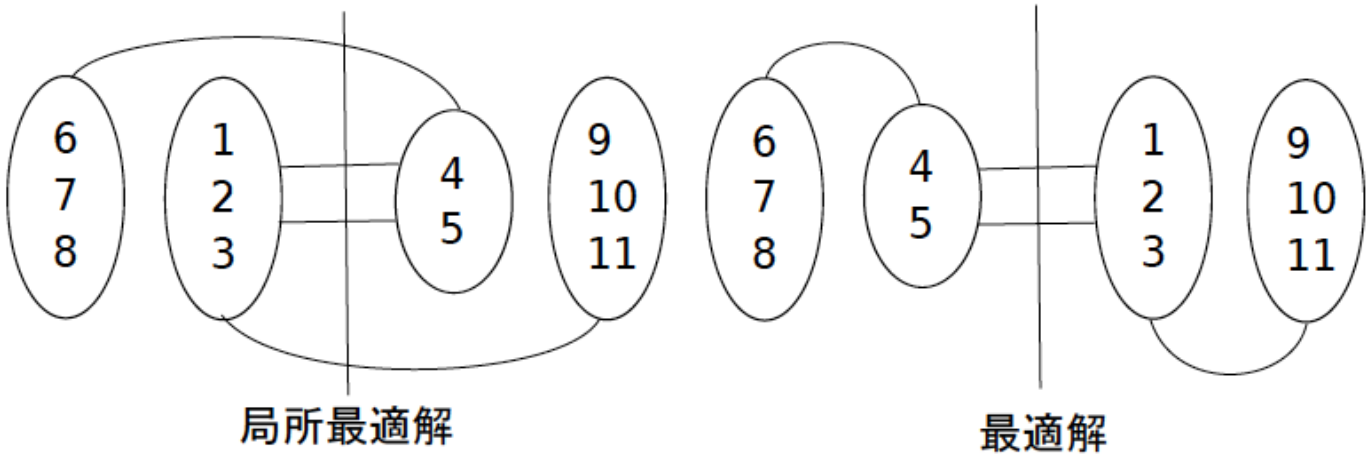
タスクグラフ



階層型マルチコア

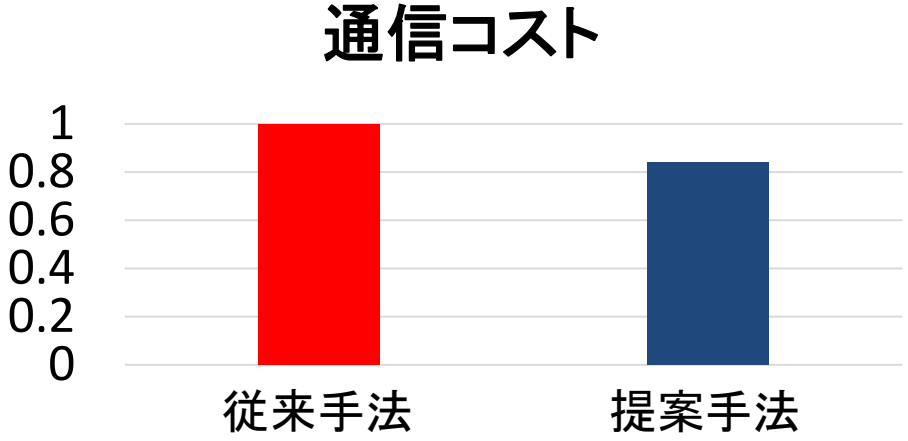
階層クラスタリング手法

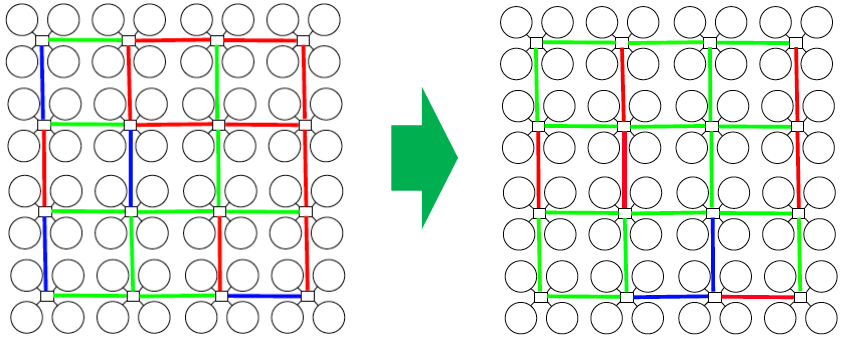




評価結果

- 64コア集中メッシュ
- 通信コスト
  - 16%削減
- 通信の分散
  - 配線全体に対する以下の項目に当てはまる配線の割合
    - 青：使用回数平均値の0.8倍未満の使用回数
    - 緑：使用回数平均値の0.8倍以上1.2倍以下の使用回数
    - 赤：使用回数平均値の1.2倍超の使用回数
  - 従来手法と比較して緑増加
    - 右図は典型的な例





２種類のアーキテクチャ

- １対多アーキテクチャ
  - 1コアに複数タスク、コア内でリアルタイム・スケジューリング（名古屋大学高田研究室でも精力的に研究）
  - コア割当に関しては歴史的に数多くの研究が存在
  - 階層クラスタリング手法を用いたタスクマージ、コア割当

- １対１アーキテクチャ
  - 1コア1タスク、コア内スケジューリング、優先度設計不要
  - 限定された通信
  - シングルレート、決定的な計算順序を保つようにタスクマージ

１対１アーキテクチャでの方針

- 実行モデル
  - １コア１タスク
    - スケジューリング・オーバーヘッドをなくす
  - タスクマージ
    - 同じ周期のタスクのみをマージ
    - 決定的に動作する性質は変えない
    - それでもタスク数がコア数よりも減らない場合、並列性を削減
  - シンプルな単方向１：１高速通信を使う
    - バス競合を最小にする

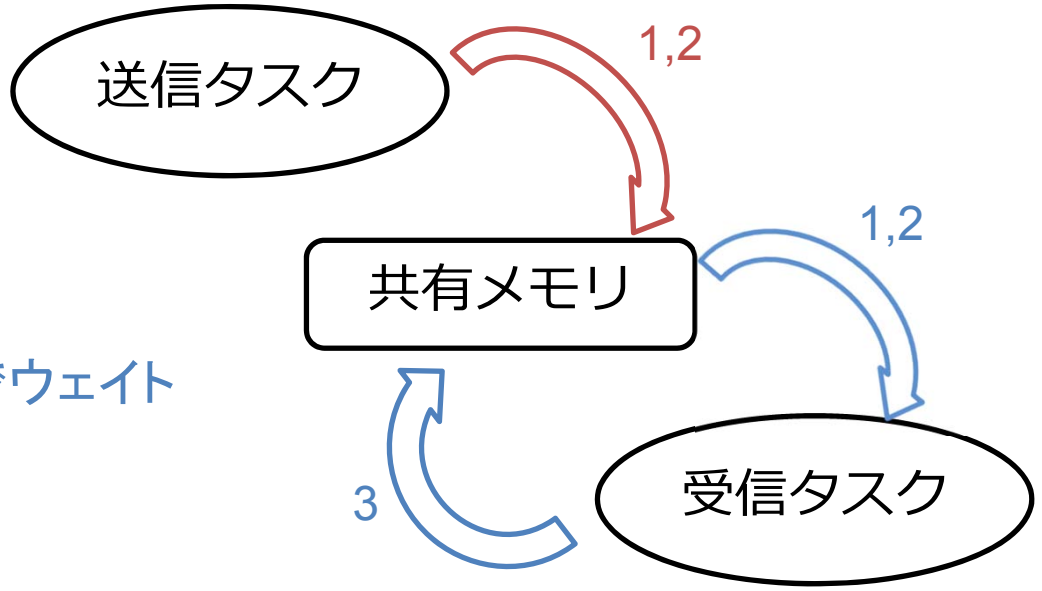
自動コード生成例 for Many Core

```
/* Block: pid_controller1_pid_Saturation */
agent sc_task_0010 () {
  interface {
    in<real32_T> CH_0013_0010;

    out<real32_T> CH_0010_I00002;

    spec {
      CH_0013_0010;
      CH_0010_I00002;
    };
  }

  map {
    SigmaC_agent_setUnitType(SigmaC_agent_self(),"k1
  }

    /* params */
    struct {
      real32_T Saturation_UpperSat;
      real32_T Saturation_LowerSat;
    } pid_controller1_P = {
  5.0F,                                 /* Computed P
                                         * Referenced
                                         */
  -5.0F                                 /* Computed P
                                         * Referenced
                                         */
    };

    /* input variables */
    real32_T pid_controller1_pid_Sum2_1;

    /* output variables */
    real32_T pid_controller1_pid_Saturation_1;

  init{
  /* initialize task context */
  pid_controller1_pid_Saturation_1 = 0.0F;
  }

  void start ()
    exchange (CH_0013_0010 ch_0013_0010,
              CH_0010_I00002 ch_0010_I00002) {

    /* input */
    pid_controller1_pid_Sum2_1 = ch_0013_0010;

    /* C code */
  /* Saturate: '<S1>/Saturation' */
  if (pid_controller1_pid_Sum2_1 >= pid_controller1_P.Saturat
    pid_controller1_pid_Saturation_1 = pid_controller1_P.Satu
  } else if (pid_controller1_pid_Sum2_1 <= pid_controller1_P.
  {
    pid_controller1_pid_Saturation_1 = pid_controller1_P.Satu
  } else {
    pid_controller1_pid_Saturation_1 = pid_controller1_pid_Su
  }

  /* End of Saturate: '<S1>/Saturation' */


    /* output */
    ch_0010_I00002 = pid_controller1_pid_Saturation_1;
  }
}
```

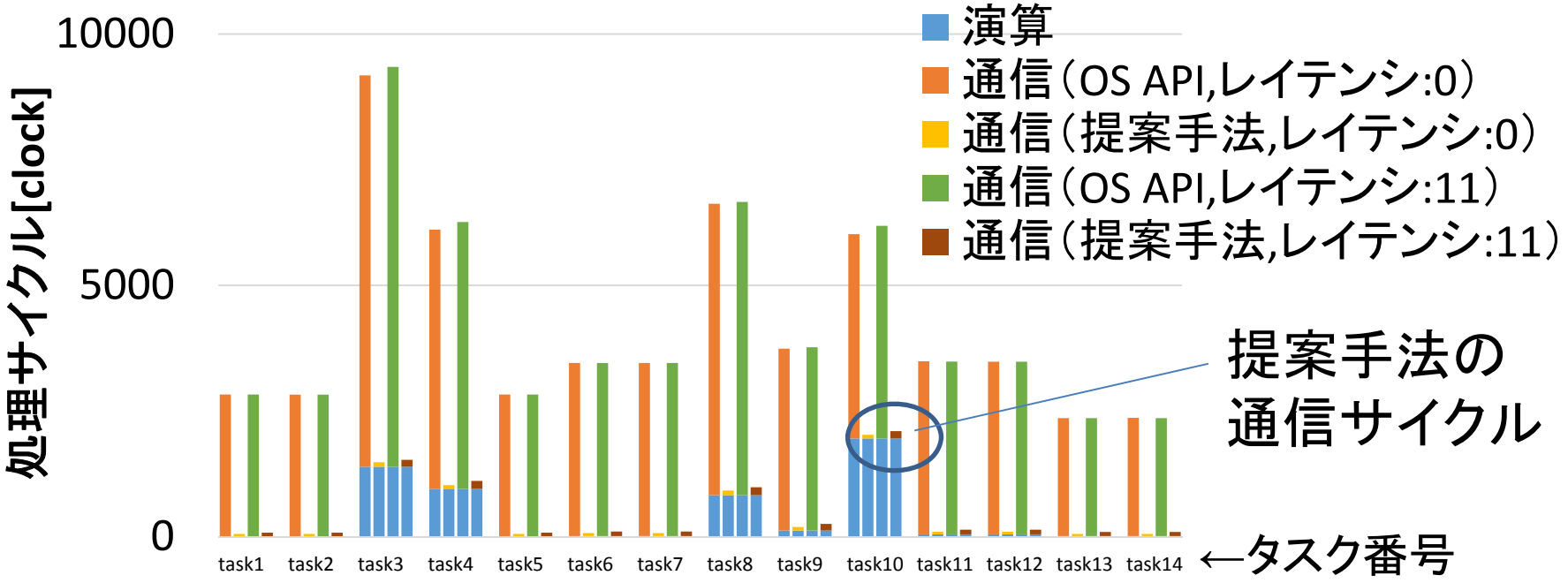
単方向1:1高速通信

- 評価方法(シミュレータによる実現)
  - 通信を行うそれぞれのタスク間に共有メモリ領域(フラグ、通信データ用)を設ける
  - 隣接プロセッサ間共有レジスタ、ローカルメモリを想定
    ⇒バスのアービタの影響が大きい領域は使用しない

- 通信プロセス
  - 送信タスク
    1. データを書き込む
    2. フラグをセットする
  - 受信タスク
    1. フラグが1となるまでウェイト
    2. データ読み込み
    3. フラグリセット



単方向1:1高速通信の評価

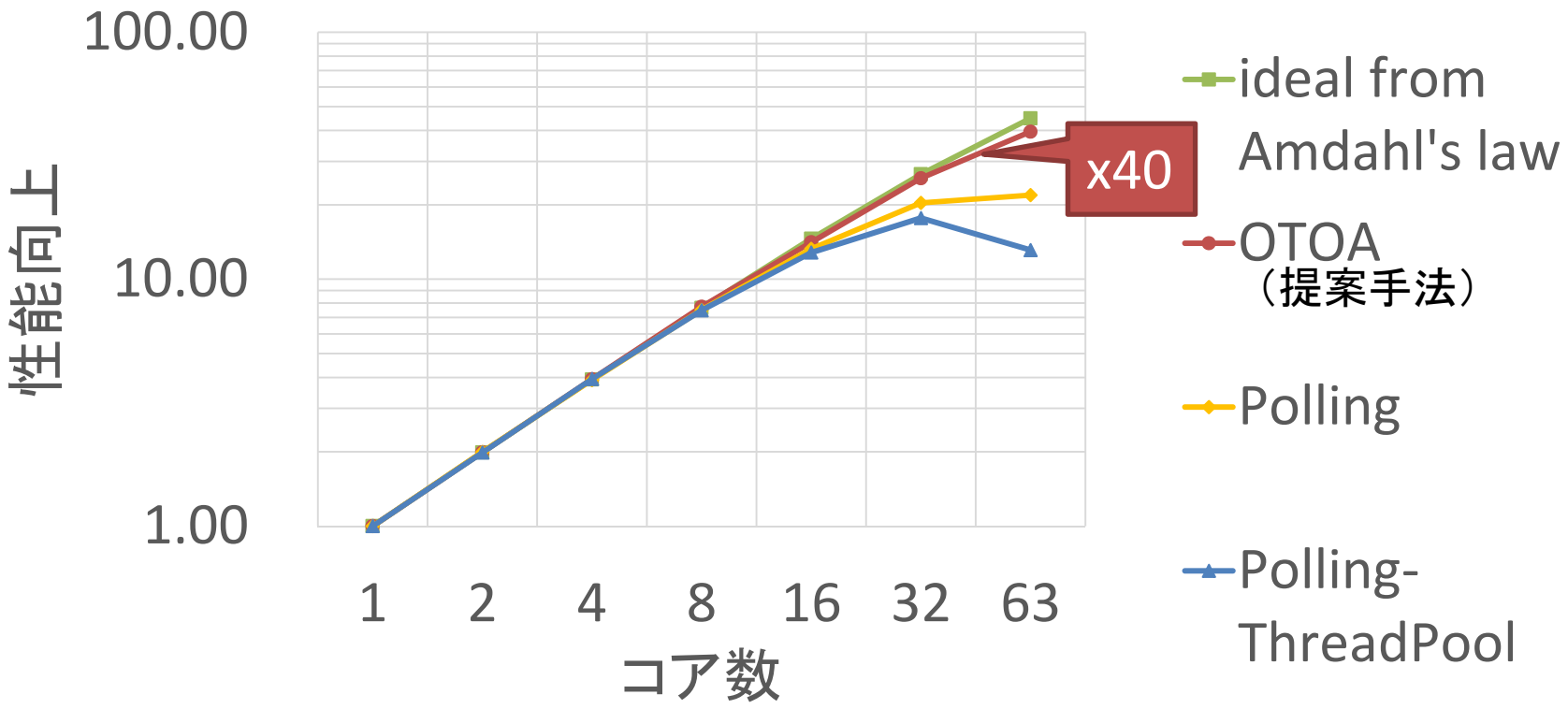


- OS APIではタスクの処理サイクルの68%以上が通信
- 提案手法の通信サイクルはOS APIの1%~4%

OS API は通信経路の決定や優先度による調停などを行うためオーバヘッドが大きい。
組込み制御向けには提案方式の方が適している。

評価結果

- モデル予測制御に基づく永久磁石同期モータのトルク制御系の改善(with 道木研究室(名古屋大))
- RH850ベース64コア サイクルレベルシミュレータでの実験
  - 最初の10 ステップのmodel_step()のサイクル数を計測



まとめ

- MATLAB/SimulinkモデルからのSHIM利用並列化に関する研究紹介
  - Simulinkモデルベース開発とコード生成の特徴
  - SHIMを用いたSimulinkモデルからの並列コード生成
  - 並列アーキテクチャ向け最適化
- 今後の課題
  - SHIMを用いたマルチ・メニーコア向けモデルベース設計方法論の確立、ガイドラインの定義
  - プロトタイプ試作、公開
  - 自動生成系だけでなく検証系の研究開発
- **コンソーシアムに入会いただき、この分野の重要性を各方面に訴えていく力を貸してください！**