Embedded Multicore Consortium
www.embeddedmulticore.org

マルチ・メニーコア向けOSと ソフトウェア開発ツールの最新動向

2015-11-18
Masaki Gondo
CTO, eSOL

Abstract

- マルチ・メニーコアの活用には、スケーラブルなランタイムのプラットフォーム、並列化やソフトウェアのコアへのマッピングのための開発支援ツールなどが重要になる。本講演では、世界初の商用メニーコアRTOSの最新情報、そしてSHIMの活用を予定している欧州を中心としたマルチ・メニーコア向け開発支援ツールの最新情報を紹介する。

Agenda

- マルチコアプロセッサ動向
- スケーラブルメニーコアRTOS
- 最新ツール事例

マルチコアツールの背景

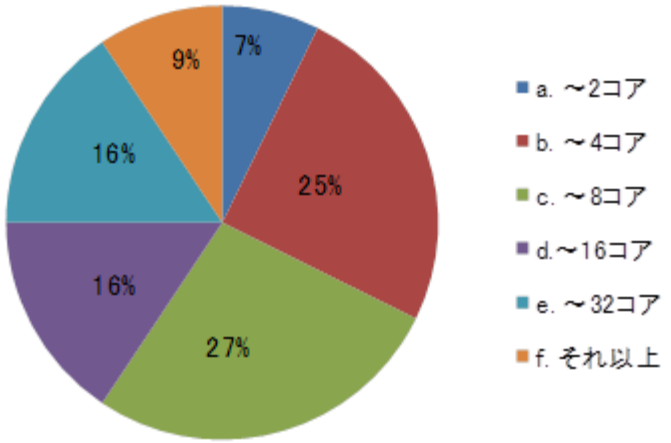
マルチコアプロセッサ動向

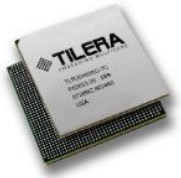
増えるコア数

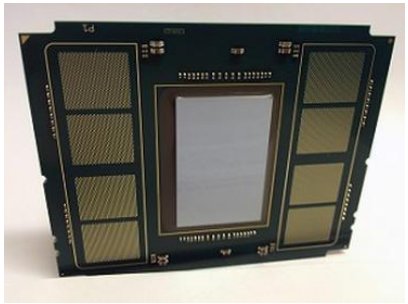
- 求められるコア数は増加の一途、**41%が8コアでは不足**（JEITA調査）
- KALRAY（256コア）Intel MIC（60コア）、TILERA TILE Gx（9～72コア）、Adaptiva Epiphany(64コア)など**新アーキテクチャ**のチップが量産中
- 4コアまでの従来マルチコアではARMのbig.LITTLEのように**ISAは同じでヘテロジニアス**なアーキテクチャで8コア構成

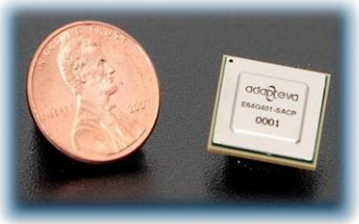
JEITA 2011～2014調査結果

項目7. 5年後に必要になると思われる
最大コア数をお選びください。(複数選択可)



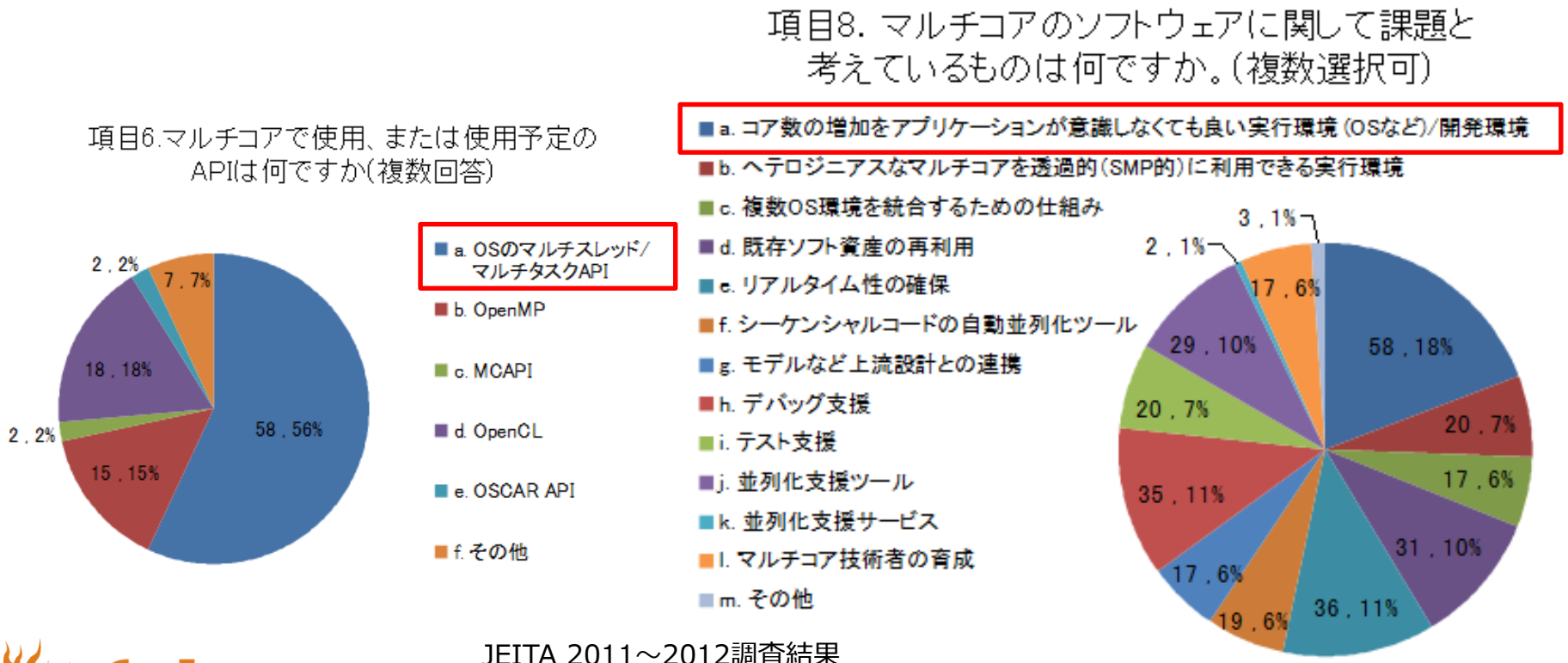






コアが増えてもプログラミングモデルは変えたくない

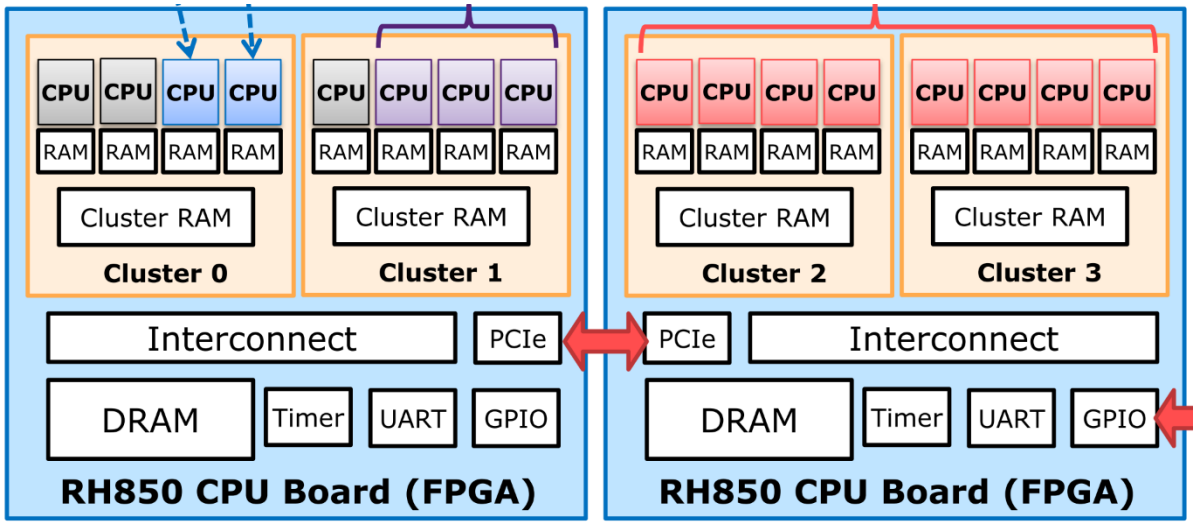
- **約60%がOSのマルチスレッド/マルチタスクAPIを使う**予定
- ソフトウェア課題の一番は**「コア数の増加をアプリケーションが意識しなくても良い実行環境(OSなど)/開発環境」**



JEITA 2011～2012調査結果

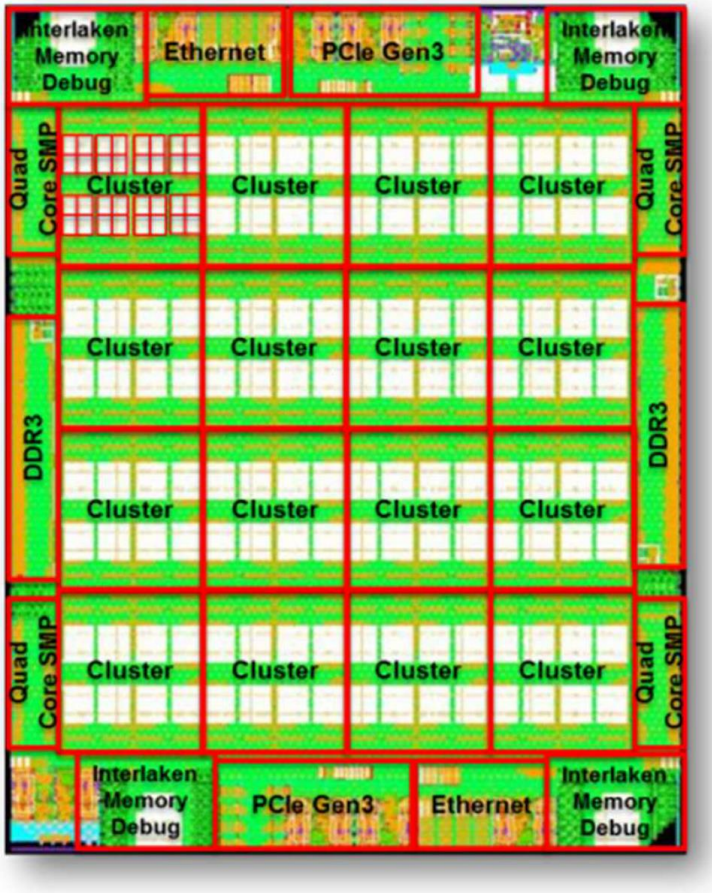
Renesas RH850 prototype

- 16 core, distributed shared memory, 200MHz, 1.5W
- 4-core cluster x 2 x 2
- eMCOS clustering can be mapped per 4-core cluster, 4 x 2 cluster, or all 16 cores
- Memory heaps are grouped into core local, cluster local, and DDRs



KALRAY MPPA

- 16 x 16 + 4 x 4 manycore processor, predictable, realtime inter-connects
- 400MHz 5-10W
- 2MB Cluster local shared memory without cache coherency
- Only the quad-cores have DDR access
- Message passing over NoC
- All cores are connected via message passing
- Threads are migratable within a cluster



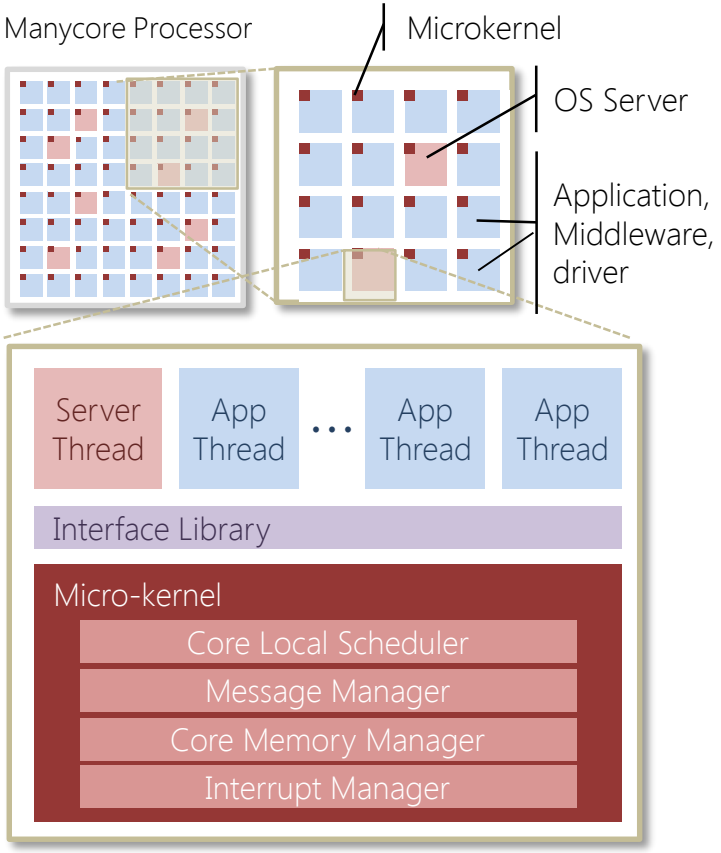
スケーラブルな世界初商用メニーコアRTOS

スケーラブルメニーコアRTOS

Background

- **2005**年に世界初のARM MPCoreのSMP/AMP対応のRTOS **eT-Kernel Multi-Core Edition**発表
  - その後、多くの車載機器、民生機器、工業機器で採用
- 2010年に既存の**マルチコアOSの限界**を認識、開発着手
  - **キャッシュコヒーレンシ**、**NoC**、**NUMA**、**Time to Space**
- MIT、チューリッヒ工科大、UCBなど海外のメニーコアOS研究を調査/分析、オリジナルのメニーコアOS開発着手
- **2012**年に**世界発の商用メニーコアRTOS**として発表（TILERA）
- 2013年～2015年に**NEDOプロジェクト**採択、実用化開発としてRH850メニーコア、KALRAY、各種ツールを対応
- 現在も**新ハードウェア対応**、**各種標準仕様**や**OSS**対応中

eSOL eMCOS – a manycore OS

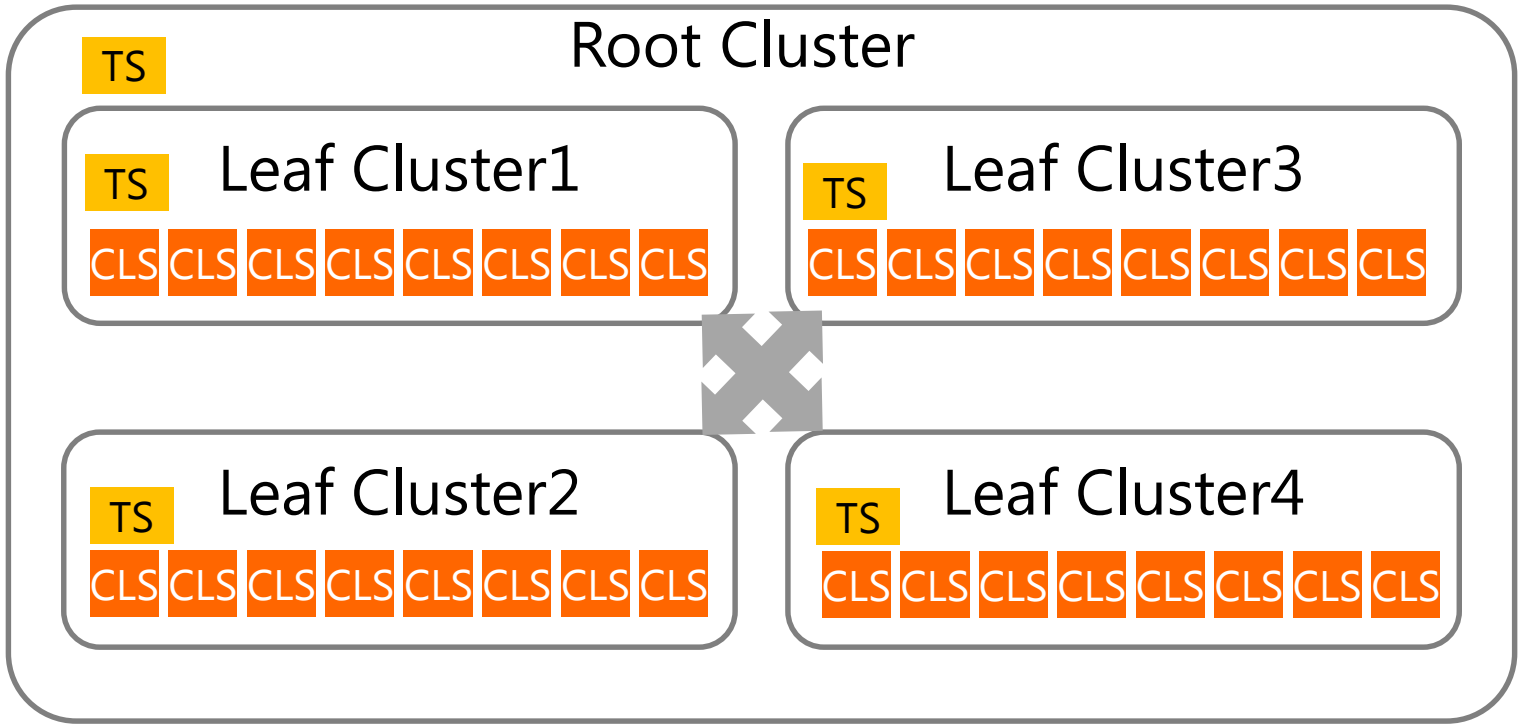


- **Distributed** microkernel
  - A micro kernel for every core
  - Message passing
  - Server/Client
  - OS services as distributed servers
  - **A single OS view from applications**
- Clustering (hierarchical)
- Layered scheduling
- Both **homogeneous and heterogeneous cores** supported
- **NUMA-capable** memory management

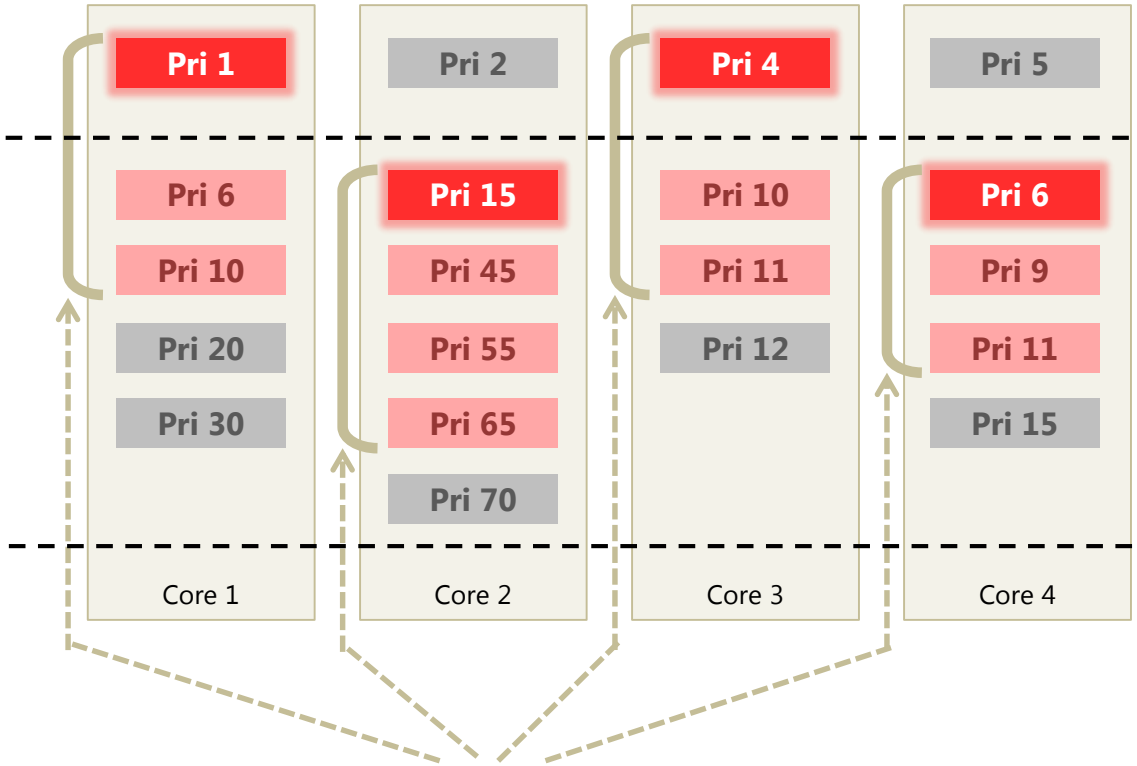
Clustering and layered schedulers

CLS = Core Local Scheduler

TS = Thread Scheduler (server)



Semi-priority based scheduling



Hard real-time threads: These threads are guaranteed to run whenever its ready

Soft real-time threads: These threads are load-balanced

Threads used for load calculation:
Any ready thread, including running, regardless of thread group

Pri #	RUNNING
Pri #	READY
Pri #	Non-READY

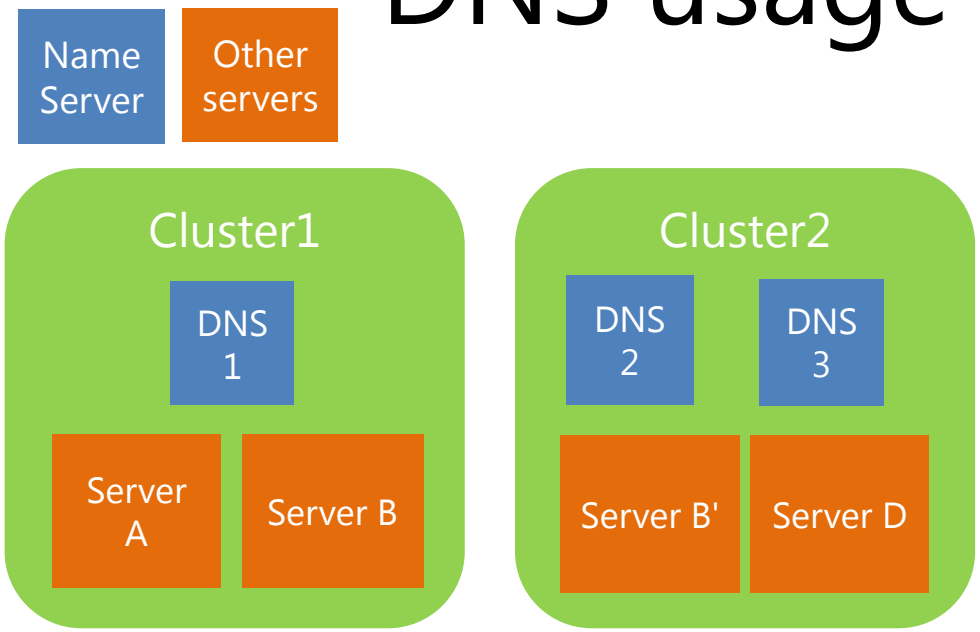
$$Wi = \sum_{j=1}^{c} (256 - Pj_{ready-thread})^2$$

$$D = \sum_{i=1}^{n} Wi^2$$

Distributed Name Service

- A **globally distributed** name service
- It provides a way to **configure the client-server routing of messages by system configuration**
- DNS enables **server multiplexing and optimization of server placement**
- **Name caching** for efficient processing

DNS usage example

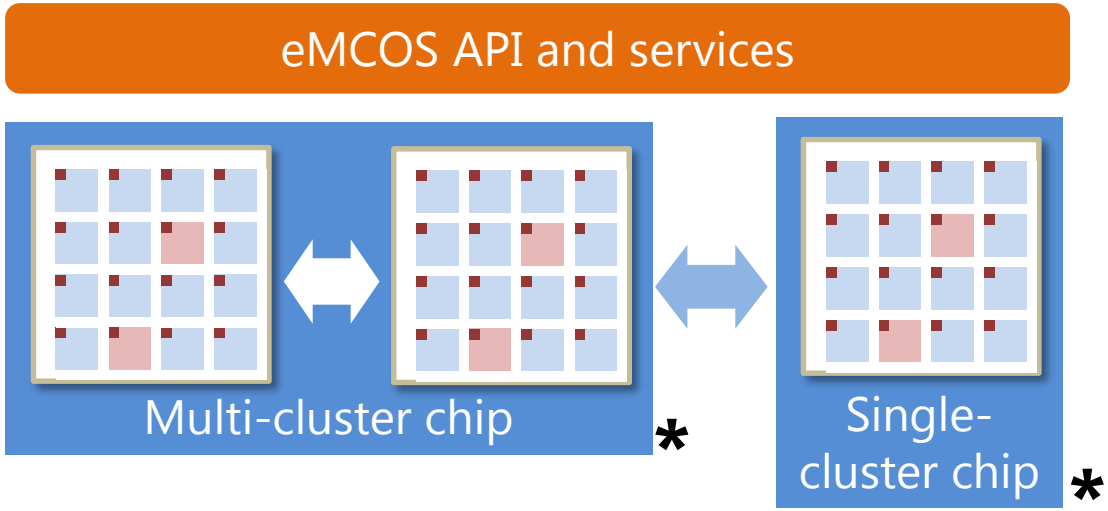


DNS Server	Servers registered
DNS1	Server A, Server B
DNS2	Server B'
DNS3	Server D

DNS server query order configuration	Cluster1	Cluster2
1' DNS Srv.	DNS1	DNS2
2' DNS Srv.	DNS3	DNS3

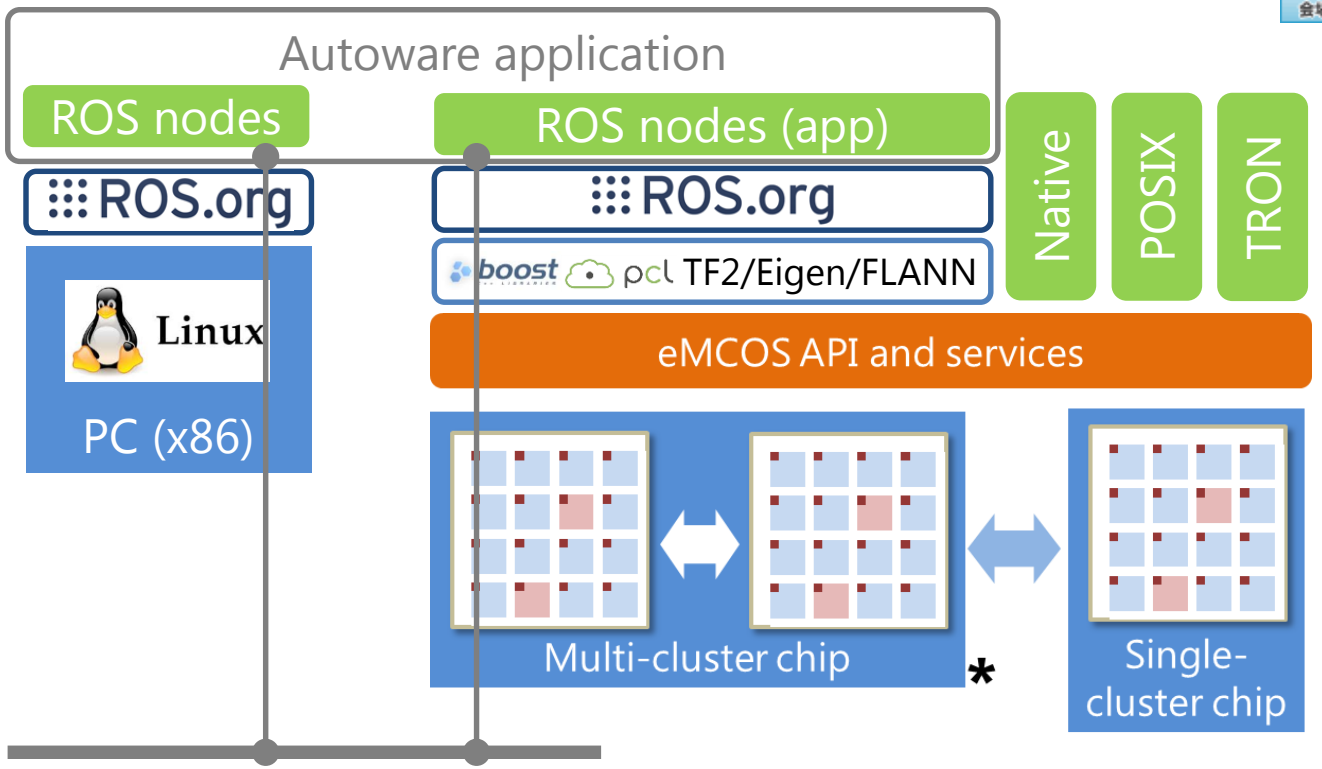
- DNS1 is "used" as the Cluster1 local name server
- DNS2 is "used" as the Cluster2 local name server
- DNS3 is "used" as the **global name server**
- This usage model is **determined by system configuration**, which configures **placement of these servers** and also the preset **DNS server search order table**
- The system configuration is performed by the system designer
- eMCOS SDK is to provide the configuration script
- Servers A/B/B'/C/D registers themselves using macros defining the DNS server they are to be registered to. The macros are defined in conjunction with the above system configuration

Heterogeneous compute/hardware clusters



- **Integrates** heterogeneous/homogeneous multi-manycore system over the message passing
- From single, multi, and manycore, with different memory subsystems
- *Any number of chips may be connected together

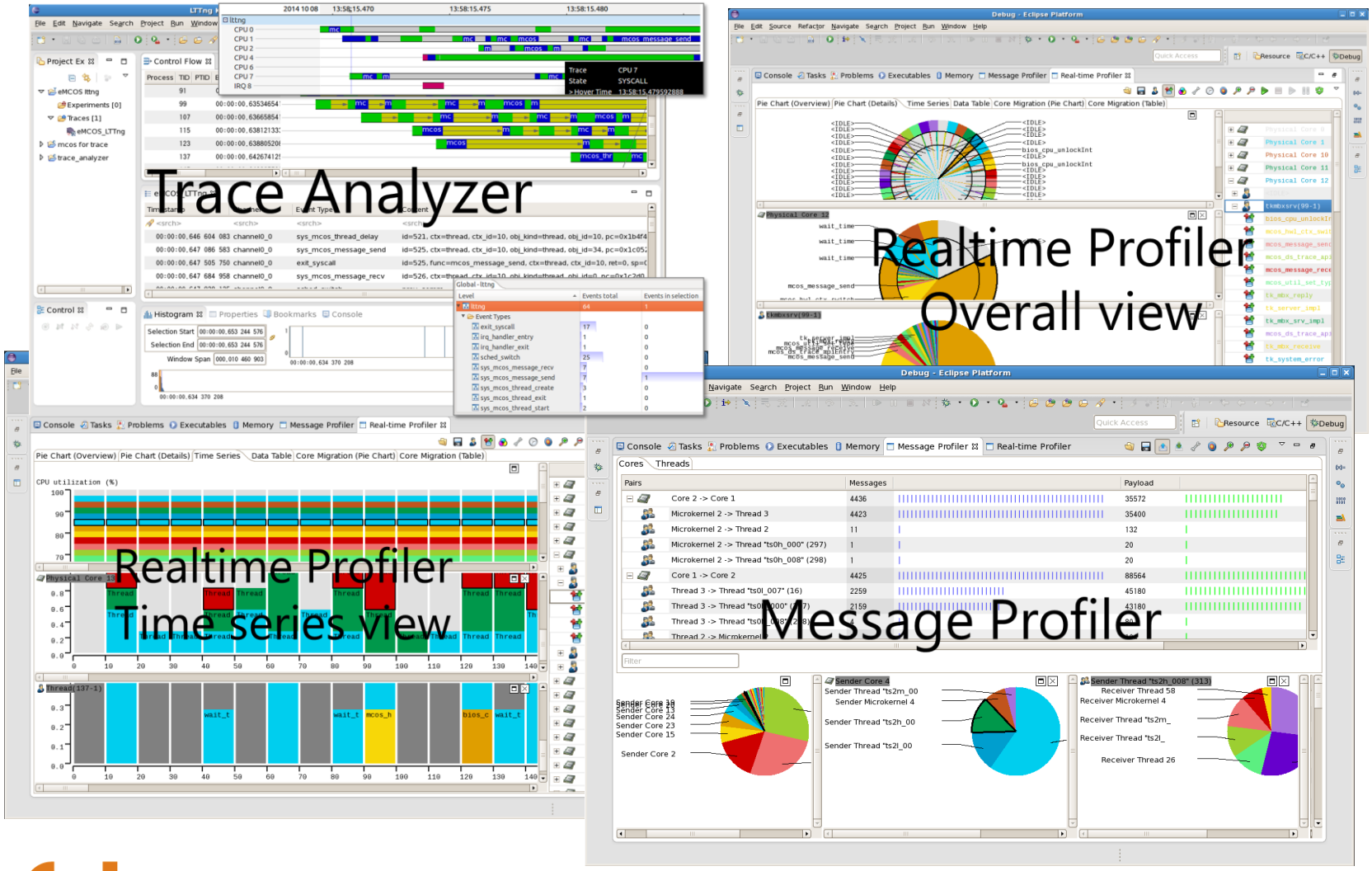
ROS on eMCOS – running *Autoware* autonomous driving system



Supported architecture

- KALRAY **MPPA255**
  - 255+16 manycore processor, predictable, realtime inter-connects
  - 400MHz 5-10W
- Renesas Electronics **RH850** manycore prototype
  - Based on the leading share RH850 for automotive
  - 16 core, distributed shared memory, 200MHz, 1.5W
- TILERA **TILE-Gx** family (9, 16, 36, 72 cores)
- A couple **more architectures under development**
- eMCOS is **scalable** to adapt to **any architecture**, from single-core to manycore

eMCOS IDE Plug-ins



SHIM

THE Multicore ASSOCIATION
www.multicore-association.org

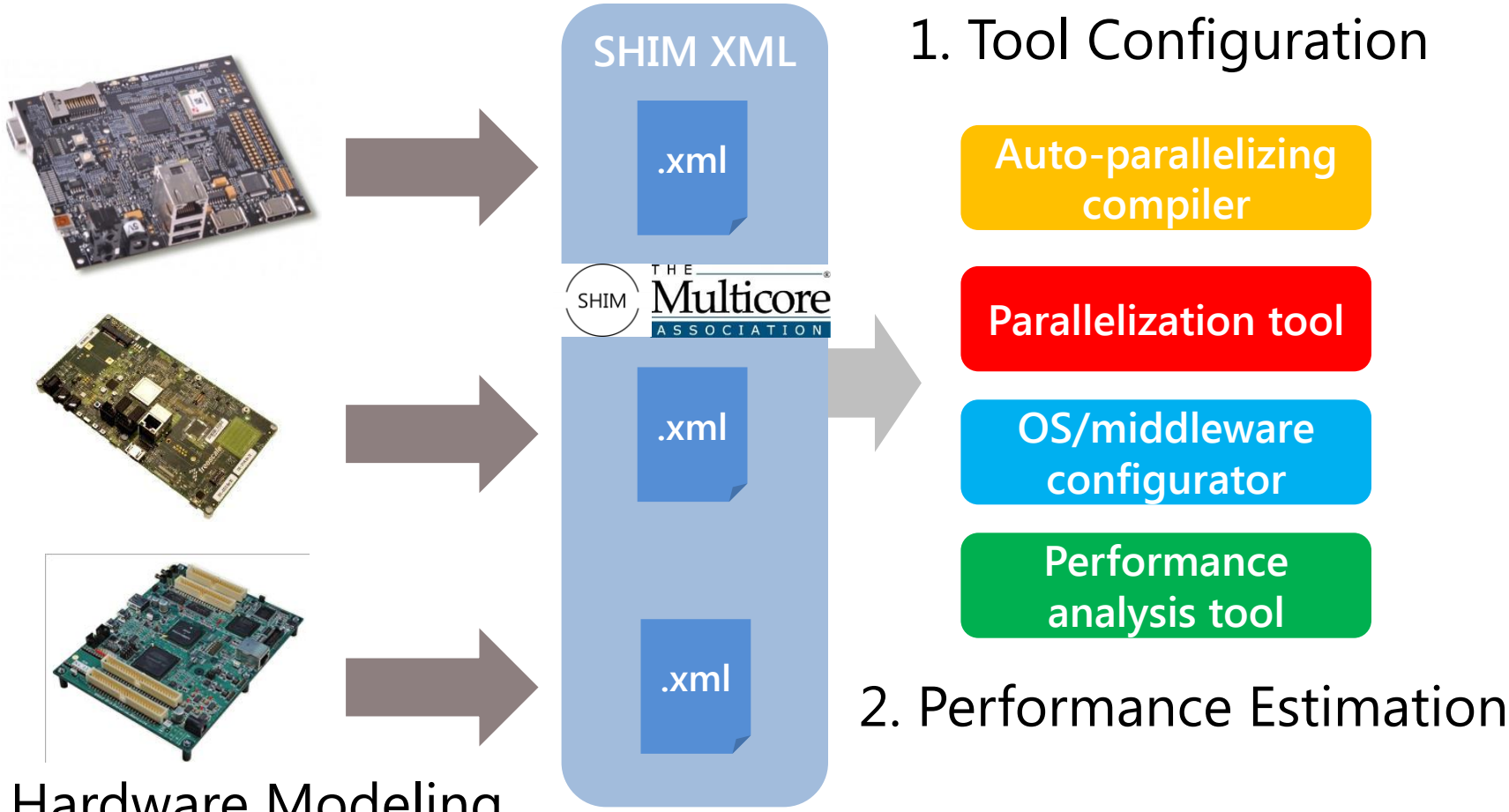
ドイツSilexica社のマルチコア設計支援ツールSLXによる
既存Cコードの並列化

最新ツール事例

SILEXICA

eSOL

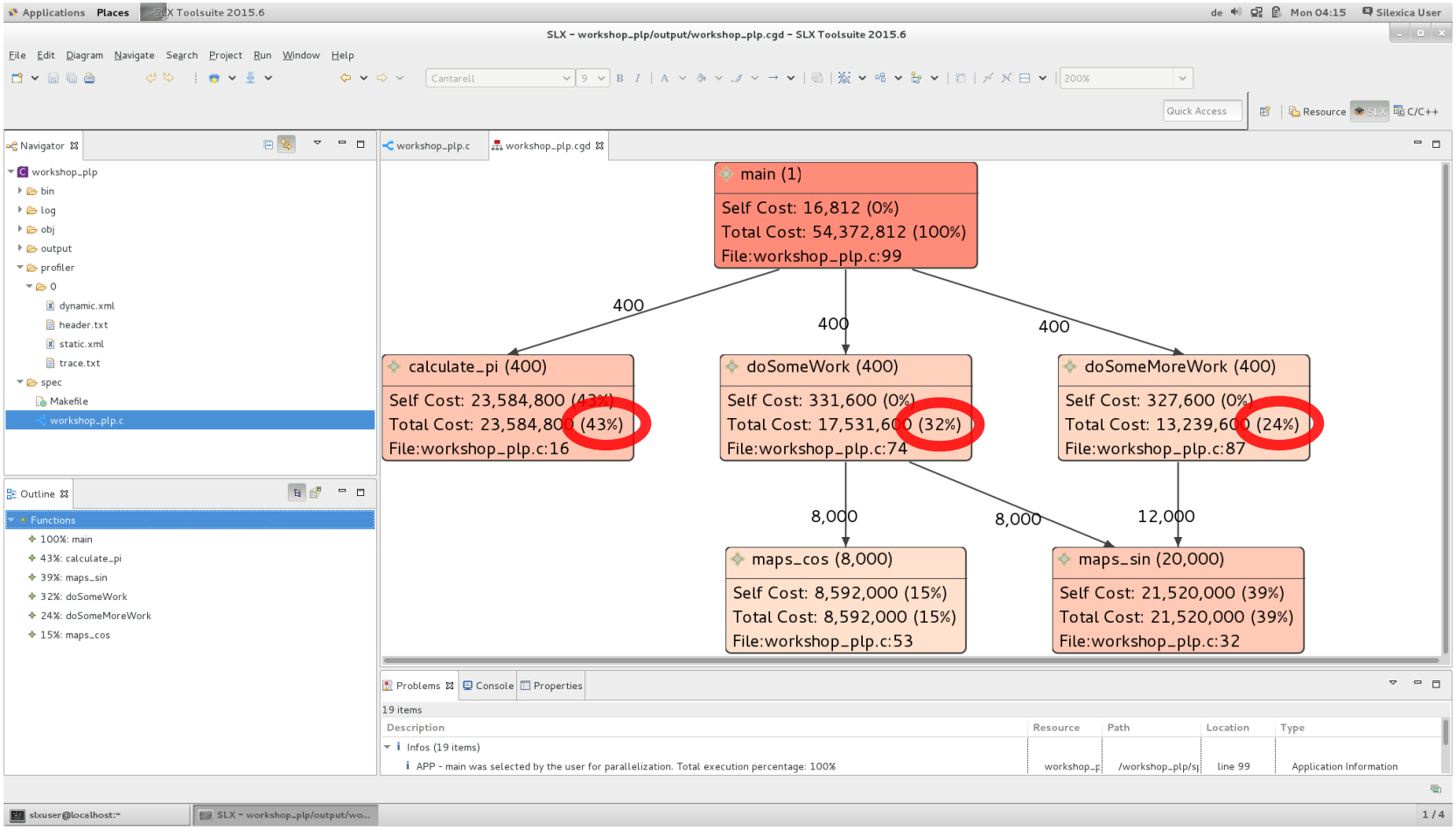
SHIM Relationship and use-cases



並列化のステップ（1）

1. 並列化する処理を決める
   - 現在の**逐次処理の処理時間プロファイルを取得/分析**する
   - 最適化の基本である「でかい所」を最初の並列化ターゲットとする
2. **並列性**（の可能性）を抽出/分析する
3. 並列化の**手法**を決める
4. **並列化**する
5. **評価**する
6. **繰り返す**

Parallelizerによる処理時間の解析

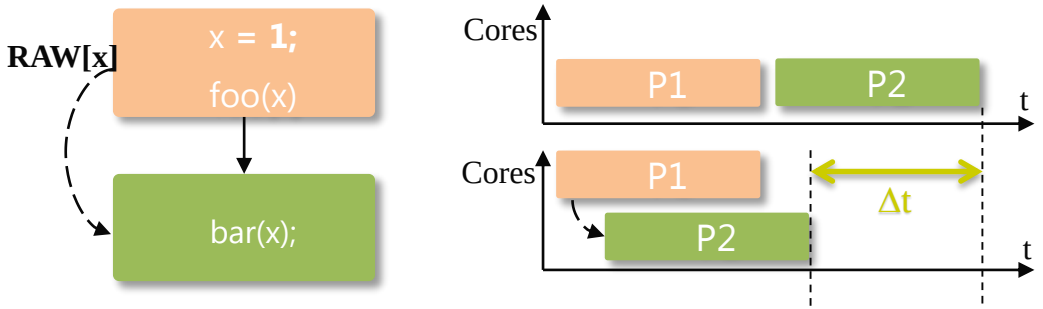


並列化のステップ（2）

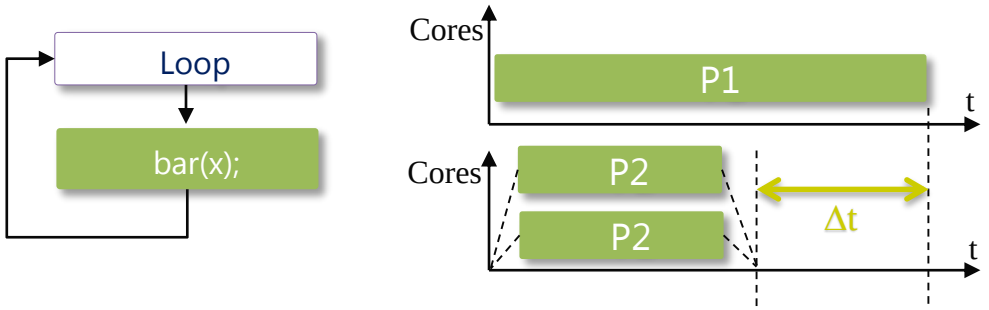
1. 並列化する処理を決める
2. **並列性**（の可能性）を抽出/分析する
   - データ依存関係（**Data dependency**）と制御順序（**Control dependency**）を抽出し、適用可能な**並列処理モデルを決める**
3. 並列化の**手法**を決める
4. **並列化**する
5. **評価**する
6. **繰り返す**

3タイプの並列性

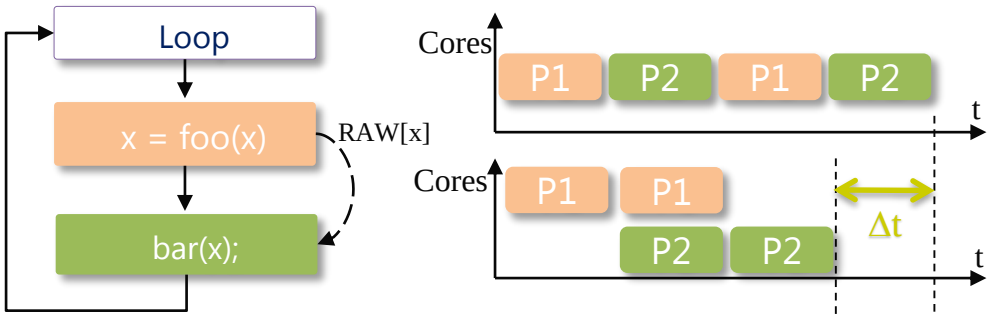
Task Level Parallelism (TLP)



Data Level Parallelism (DLP)



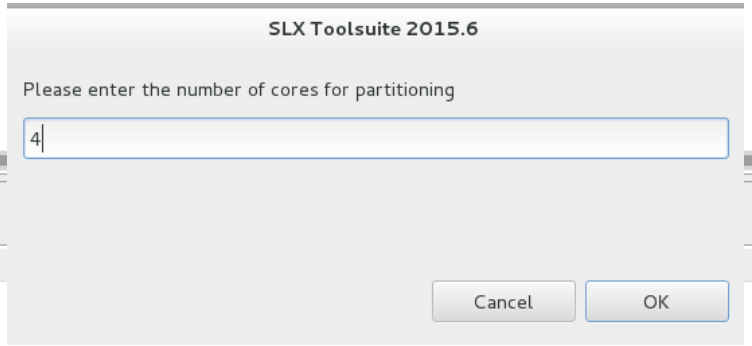
Pipeline Level Parallelism (PLP)



Parallelizerによるパーティショニング（並列性抽出と性能向上見積）

workshop_plp.c | workshop_plp.cgd

```
100 {
101
102     float x = 0.0;
103
104     float y = 0.0;
105
106     float z = 0.0;
107
108     float a = 9.2;
109
110     /* A for loop that can be broken into stages (e.g, PLP) */
111     for (int i = 0; i < MAIN_LOAD_PLP; i++)
112     {
113         x = calculate_pi();
114         y = doSomeWork(x / 2);
115         y = y + a;
116         z = doSomeMoreWork(y / 4 + z);
117     }
118
119     printf("z=%f\n", z);
120     return 0;
121 }
122
```



Problems | Console | Properties

SLXConsole

```
Partitioning process started...

-> Creating statement partition
   Finished!


-> Analyze DLP/PLP partitions for function main

workshop_plp.c:111:0: phint: DLP - Loop (111:117) does not provide DLP because of loop-carried dependency on variable z [RAW]
workshop_plp.c:111:0: phint: PLP - Loop (111:117) has a pipeline with 2 stages and an estimated speedup of 1.74968. Efficiency: 0.87484
workshop_plp.c:111:0: phint: PLP - Loop (111:117) has a pipeline with 3 stages and an estimated speedup of 2.27321. Efficiency: 0.757737
workshop_plp.c:111:0: phint: PLP - Loop (111:117) has recommended pipeline configuration with 3 stages. Estimated speedup is:  Local 2.27321, Global 2.27304

-> Analyze TLP partitions for function main


-> Report DLP/PLP partitions for function main

workshop_plp.c:111:0: phint: PLP - Loop (111:117) presents PLP parallelism pattern with a local speedup of 2.27321
------------------------------------------------------------


-> Report TLP partitions for all functions

workshop_plp.c:99:0: phint: TLP - No task-level parallelism selected for function main
workshop_plp.c:99:0: ahint: APP - Speedup for application workshop_plp due to all kinds of parallelism is estimated to be: 2.27304
Partitioning process finished   1.20 s
------------------------------------------------------------

[ Running partitioning algorithms finished ]
```

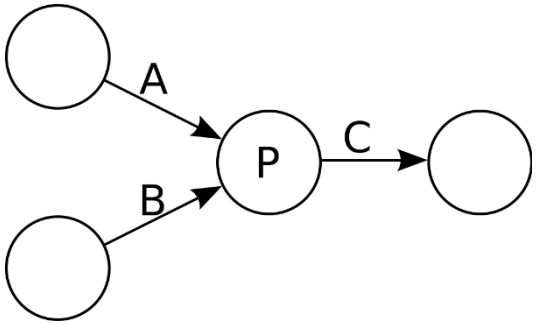
並列化のステップ（3と4）

1. 並列化する処理を決める
2. **並列性**（の可能性）を抽出/分析する
3. 並列化の**手法**を決める
   - 既存Cコードの対象箇所に適用可能な並列処理モデルを実装する方法を決める
4. **並列化**する
   - 決めた手法に従って実装する
5. **評価**する
6. **繰り返す**

既存Cコードの並列化

- 色々な並列化の手法
  - **マルチスレッド**＆**排他**、マルチスレッド＆**メッセージング**
  - **OpenMP**、Intel **TBB**、**OpenCL**、MS **C++AMP**…
    - どの手法も最後は**複数**の計算機/**演算ユニットに処理を振分け**る
    - するとHWを直接制御しない限りは**OSやドライバが間に入り**、**スレッドなどへのマッピング**、**スレッド間の通信**等が使われる
- 手でやるのは大変なので並列化支援ツールが出現しつつある

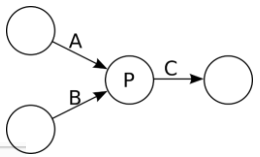
Process network

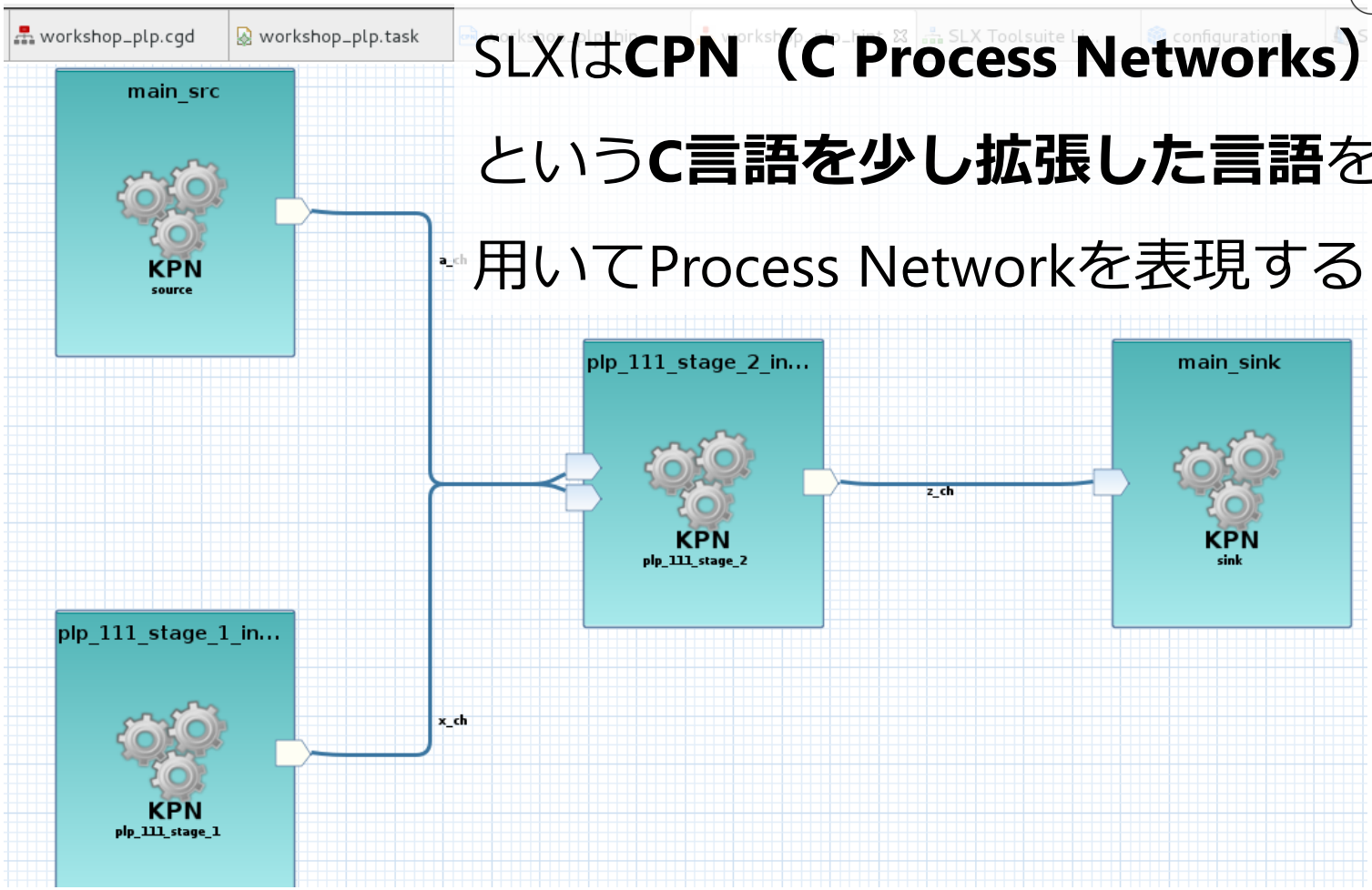


- 世界的には組込みシステムやシグナルプロセッシング、HPCに用いられる**分散型計算モデル**の一つ
- 逐次処理を含む「**プロセス**」とプロセス間を接続する「**チャネル**」と呼ぶ無限FIFOで構成
- チャネルの**書き込みはnon-blocking**、**読込はblocking**
- 有名なもので**KPN**（Kahn process networks）
- 元は**並列プログラミング言語**として考えられたもの

K. Gilles, "***The semantics of a simple language for parallel programming***," in *In Information Processing'74: Proceedings of the IFIP Congress*, **1974**, vol. 74, pp. 471–475.

2ステージパイプラインのCPN



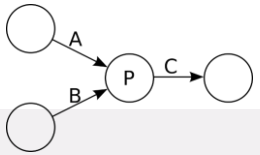


SLXは**CPN（C Process Networks）**

という**C言語を少し拡張した言語**を

用いてProcess Networkを表現する

2ステージパイプラインのCPN記述（1/2）



CPNコード

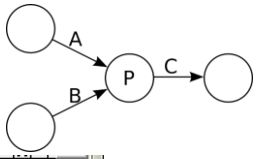
```
__PNkpn plp_111_stage_1 __PNout(float x)
{
    /* First stage: follow hint by copying the loop
     * statement and accessing the variable x within
     * the loop */
    float i;
    for (i = 0; i < MAIN_LOAD_PLP; i++)
    {
        __PNout(x)
            x = calculate_pi();
    }
}

/*************************************************************/
/*    Process for stage 2 of PLP loop at line 111            */
/*************************************************************/
__PNkpn plp_111_stage_2 __PNin(float a, float x) __PNout(float z)
{
    float y = 0.0;

    /* Second stage: follow hint by copying the loop
     * statement, accessing the variable x within
     * the loop and variables 'a' and 'z' outside
     * the loop  */
    __PNin(a) __PNout(z) {
    float i;
        for (i = 0; i < MAIN_LOAD_PLP; i++)
        {
            __PNin(x)
            {
                y = doSomeWork(x / 2);
                y = y + a;
                z = doSomeMoreWork(y / 4 + z);
            }
        }
    }
```

CPN化

既存Cコード

```
float doSomeMoreWork(float x)
{
    float tmp = 0.0;

    for (float i = 0.0; i < 1.5 * WORKER_LOAD_PLP; i++)
    {
        tmp += maps_sin(x) / (i + 1);
    }

    return tmp;
}

int main(void)
{

    float x = 0.0;

    float y = 0.0;

    float z = 0.0;

    float a = 9.2;

    /* A for loop that can be broken into stages (e.g, PLP) */
    for (int i = 0; i < MAIN_LOAD_PLP; i++)
    {
        x = calculate_pi();
        y = doSomeWork(x / 2);
        y = y + a;
        z = doSomeMoreWork(y / 4 + z);
    }

    printf("z=%f\n", z);
    return 0;
}
[EOF]
```

2ステージパイプラインのCPN記述（2/2）

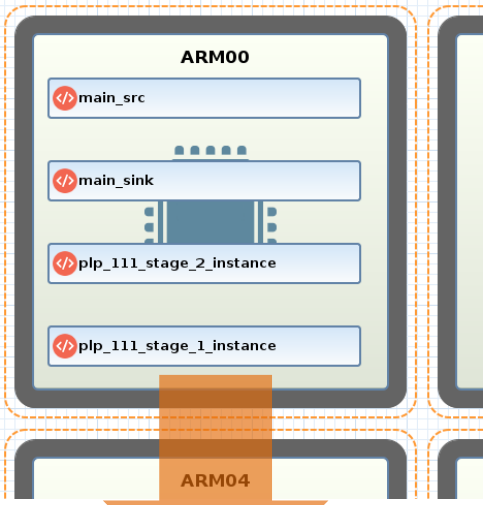


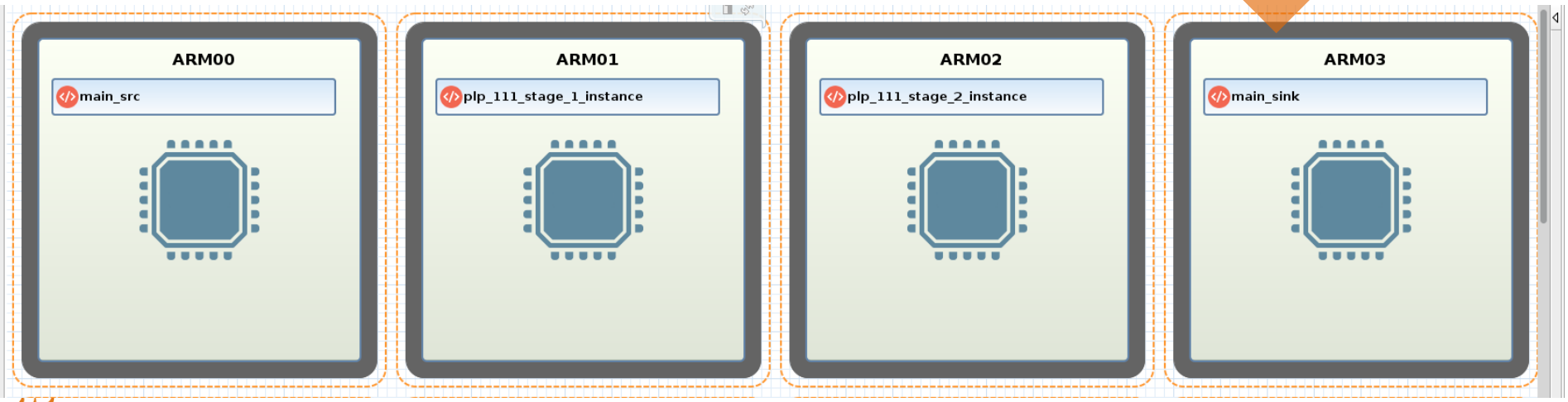

```
/*************************************************************/
/*    Source and sink processes with code from main          */
/*************************************************************/
__PNkpn source __PNout(float a)
{
    __PNout(a)
        a = 9.2;
}


__PNkpn sink __PNin(float z)
{
    __PNin(z)
        printf("z=%f\n", z);
}

/*************************************************************/
/*    Instantiate and connect proceses (done by hand)        */
/*************************************************************/
/* Channels to connect processes: Written by hand */
__PNchannel float a_ch, x_ch, z_ch;

/* Instantiate and connect processes */
__PNprocess main_src = source __PNout(a_ch);
__PNprocess plp_111_stage_1_instance = plp_111_stage_1 __PNout(x_ch);
__PNprocess plp_111_stage_2_instance = plp_111_stage_2 __PNin(a_ch, x_ch) __PNout(z_ch);
__PNprocess main_sink = sink __PNin(z_ch);
```

CPNによる2ステージパイプライン化：**開始プロセスと終了プロセス、全プロセスの定義、チャンネルの定義**

2ステージパイプラインでの自動マッピング

- **自動でCPNプロセスをコアにマッピング**
- ここでは16コアのハードウェアモデルを使用しているが、**異なるハードウェアを選択可能**



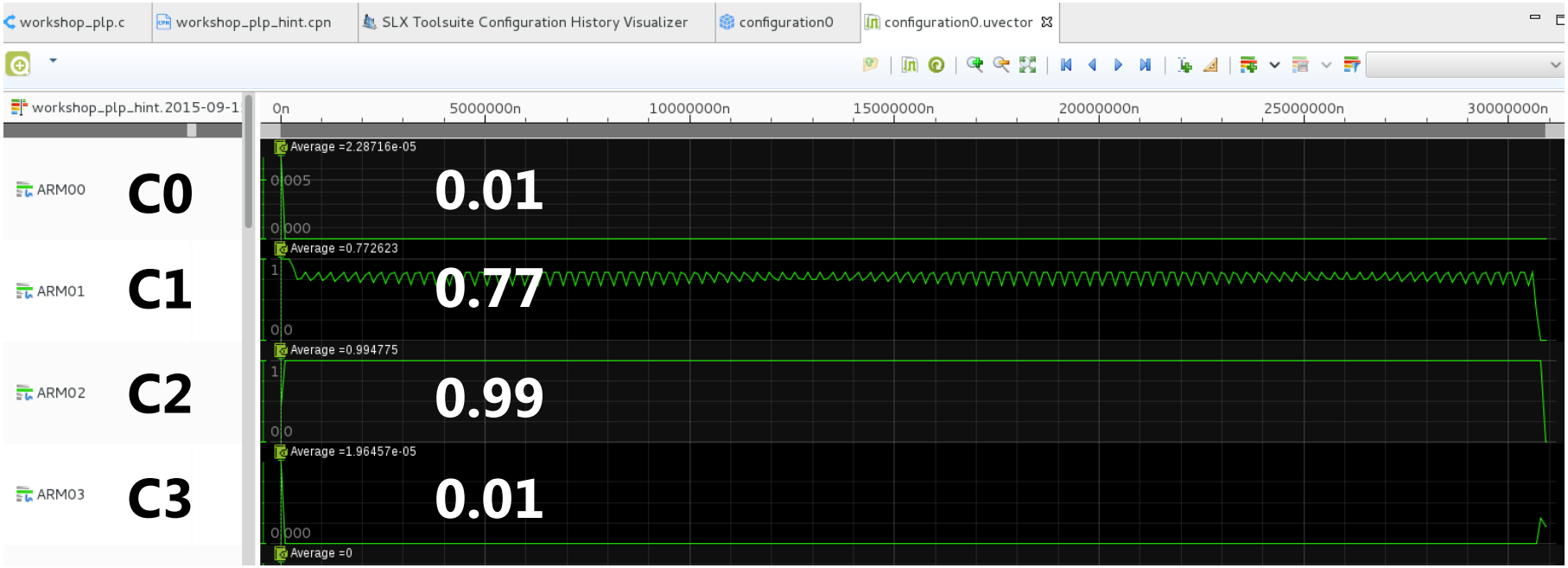


並列化のステップ（5）

1. 並列化する処理を決める
2. **並列性**（の可能性）を抽出/分析する
3. 並列化の**手法**を決める
4. **並列化**する
5. **評価**する
   - 並列化した結果の性能向上をチェックする
6. **繰り返す**

2ステージパイプラインのCPU利用率

- C2がボトルネック
- なので、これを更にパイプライン化する



2ステージパイプラインでのプロセッサコアの利用率

並列化のステップ（6）

1. 並列化する処理を決める
2. **並列性**（の可能性）を抽出/分析する
3. 並列化の**手法**を決める
4. **並列化**する
5. **評価**する
6. **繰り返す**
   - 結果が不足していれば、１、２、３、または4に戻る

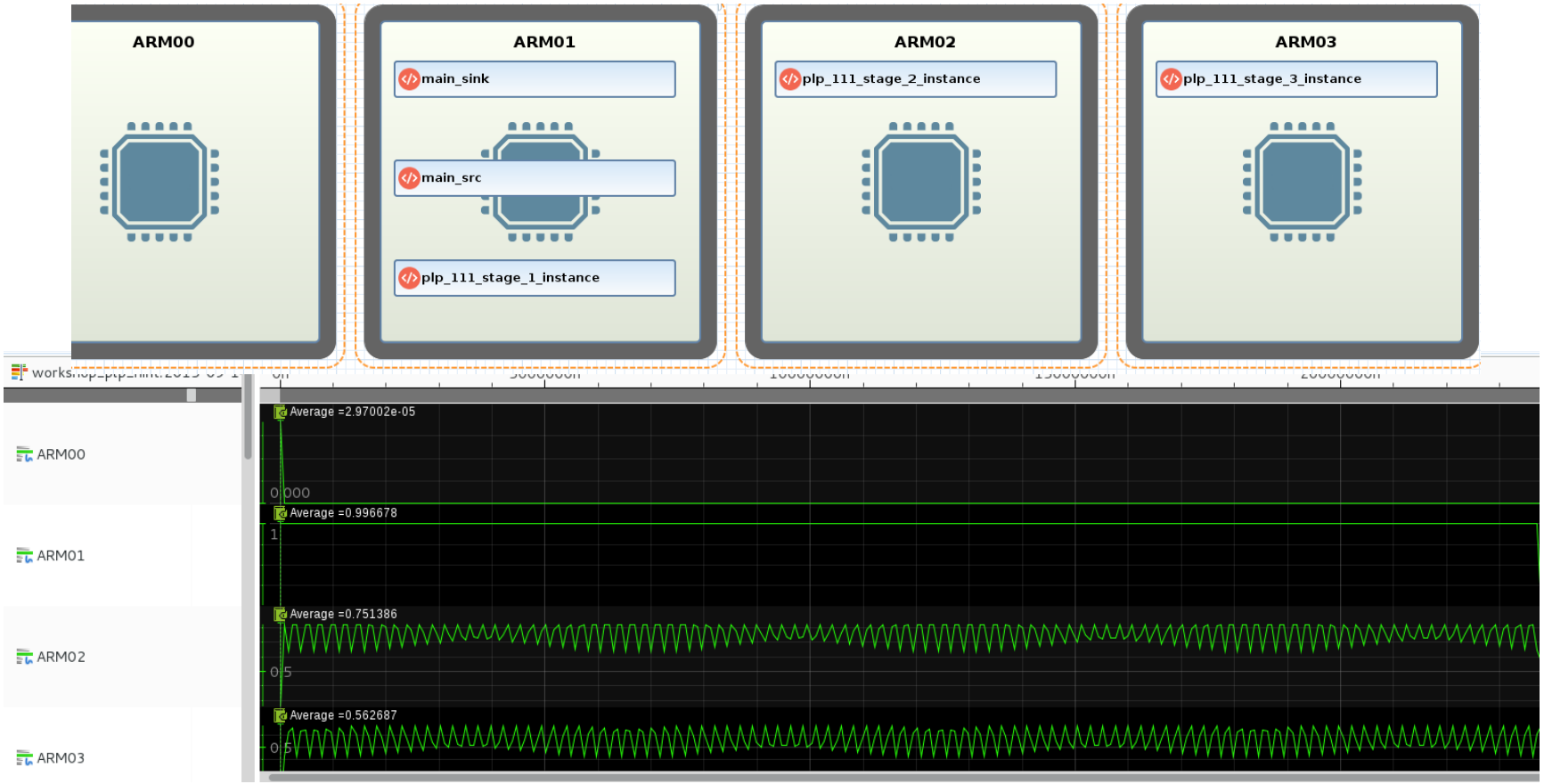
パイプラインの3ステージ化

```
155 /*************************************************************/
156 __PNkpn plp_111_stage_2 __PNin(float a, float x) __PNout(float z)
157 {
158     float y = 0.0;
159
160     /* Second stage: follow hint by copying the loop
161      * statement, accessing the variable x within
162      * the loop and variables 'a' and 'z' outside
163      * the loop  */
164     __PNin(a) __PNout(z) {
165     float i;
166         for (i = 0; i < MAIN_LOAD_PLP; i++)
167         {
168             __PNin(x)
169             {
170                 y = doSomeWork(x / 2);
171                 y = y + a;
172                 z = doSomeMoreWork(y / 4 + z);
173             }
174         }
175     }
176 }
177
178 #else   //added another stage
179 /*************************************************************/
180 /*    Process for stage 2 of PLP loop at line 111            */
181 /*************************************************************/
182 __PNkpn plp_111_stage_2 __PNin(float a, float x) __PNout(float y)
183 {
184     /* Second stage: follow hint by copying the loop
185      * statement, accessing the variable x within
186      * the loop and variables 'a' and 'z' outside
187      * the loop  */
188     __PNin(a){
189     float i;
190         for (i = 0; i < MAIN_LOAD_PLP; i++)
191         {
192             __PNin(x) __PNout(y)
193             {
194                 y = doSomeWork(x / 2);
195                 y = y + a;
196             }
197         }
198     }
199 }
200
```

```
201 /*************************************************************/
202 /*    Process for stage 3 of PLP loop at line 111            */
203 /*************************************************************/
204 __PNkpn plp_111_stage_3 __PNin(float y) __PNout(float z)
205 {
206     __PNout(z) {
207     float i;
208         for (i = 0; i < MAIN_LOAD_PLP; i++)
209         {
210             __PNin(y)
211             {
212                 z = doSomeMoreWork(y / 4 + z);
213             }
214         }
215     }
216 }
217 #endif
218
219 /*************************************************************/
220 /*    Instantiate and connect proceses (done by hand)        */
221 /*************************************************************/
222 /* Channels to connect processes: Written by hand */
223 #if TWO_STAGE //original
224 __PNchannel float a_ch, x_ch, z_ch;
225 #else
226 __PNchannel float a_ch, x_ch, y_ch, z_ch;
227 #endif
228
229 /* Instantiate and connect processes */
230 __PNprocess main_src = source __PNout(a_ch);
231 __PNprocess plp_111_stage_1_instance = plp_111_stage_1 __PNout(x_ch);
232 #if TWO_STAGE //original
233 __PNprocess plp_111_stage_2_instance = plp_111_stage_2 __PNin(a_ch, x_ch) __PNout(z_ch);
234 #else
235     __PNprocess plp_111_stage_2_instance = plp_111_stage_2  __PNin(a_ch, x_ch)  __PNout(y_ch);
236 __PNprocess plp_111_stage_3_instance = plp_111_stage_3  __PNin(y_ch)  __PNout(z_ch);
237 #endif
238 __PNprocess main_sink = sink __PNin(z_ch);
239 [EOF]
```

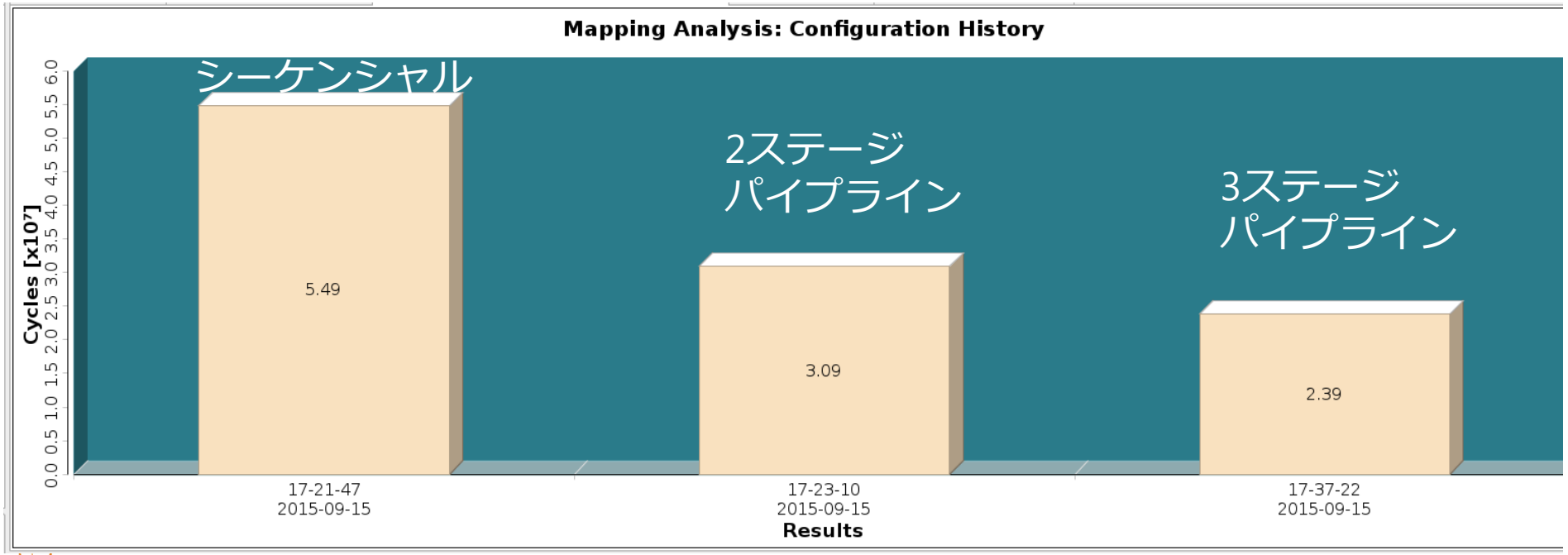
手動によるマッピング調整

- 処理の軽い2コア分の処理をパイプラインの1ステージのコアに



パイプラインの3ステージ化による スピードアップ

- 2ステージ（4コアマッピング、実質2コア）で約1.8倍
- 3ステージ（3コアマッピング）で約2.3倍



pthreadで自動生成されたコード

```
838 ↓
839 int main(int argc, char **argv, char **env) {↓
840     LLfifo_chan *const a_ch = LLfifo_create(0, sizeof(float), 8UL, 0UL);↓
841     LLfifo_chan *const x_ch = LLfifo_create(1, sizeof(float), 8UL, 0UL);↓
842     LLfifo_chan *const y_ch = LLfifo_create(2, sizeof(float), 8UL, 0UL);↓
843     LLfifo_chan *const z_ch = LLfifo_create(3, sizeof(float), 8UL, 0UL);↓
844     LLfifo_init();↓
845     {↓
846         struct PNargs_source const PNargs_main_src = { a_ch };↓
847         struct PNargs_plp_111_stage_1 const PNargs_plp_111_stage_1_instance = { x_ch };↓
848         struct PNargs_plp_111_stage_2 const PNargs_plp_111_stage_2_instance = { a_ch, x_c
849         struct PNargs_plp_111_stage_3 const PNargs_plp_111_stage_3_instance = { y_ch, z_c
850         struct PNargs_sink const PNargs_main_sink = { z_ch };↓
851         pthread_t main_src;↓
852         pthread_t plp_111_stage_1_instance;↓
853         pthread_t plp_111_stage_2_instance;↓
854         pthread_t plp_111_stage_3_instance;↓
855         pthread_t main_sink;↓
856         pthread_create(&main_src, 0, (void *(*)(void *))&PNwrapped_source, (void *)&PNarg
857         pthread_create(&plp_111_stage_1_instance, 0, (void *(*)(void *))&PNwrapped_plp_11
    111_stage_1_instance);↓
858         pthread_create(&plp_111_stage_2_instance, 0, (void *(*)(void *))&PNwrapped_plp_11
    111_stage_2_instance);↓
859         pthread_create(&plp_111_stage_3_instance, 0, (void *(*)(void *))&PNwrapped_plp_11
    111_stage_3_instance);↓
860         pthread_create(&main_sink, 0, (void *(*)(void *))&PNwrapped_sink, (void *)&PNargs
861         pthread_join(main_src, 0);↓
862         pthread_join(plp_111_stage_1_instance, 0);↓
863         pthread_join(plp_111_stage_2_instance, 0);↓
864         pthread_join(plp_111_stage_3_instance, 0);↓
865         pthread_join(main_sink, 0);↓
866     }↓
867     return 0;↓
868 }↓
869 ↓
```

まとめ

- プロセッサ**コア数は増える**
- 増えたコアを**使いこなせるOS、そしてツールが出現**しつつある
- マルチコアハードとツール群を対応しやすくし**マルチコアエコシステムを拡充する標準化活動**も進んでいる