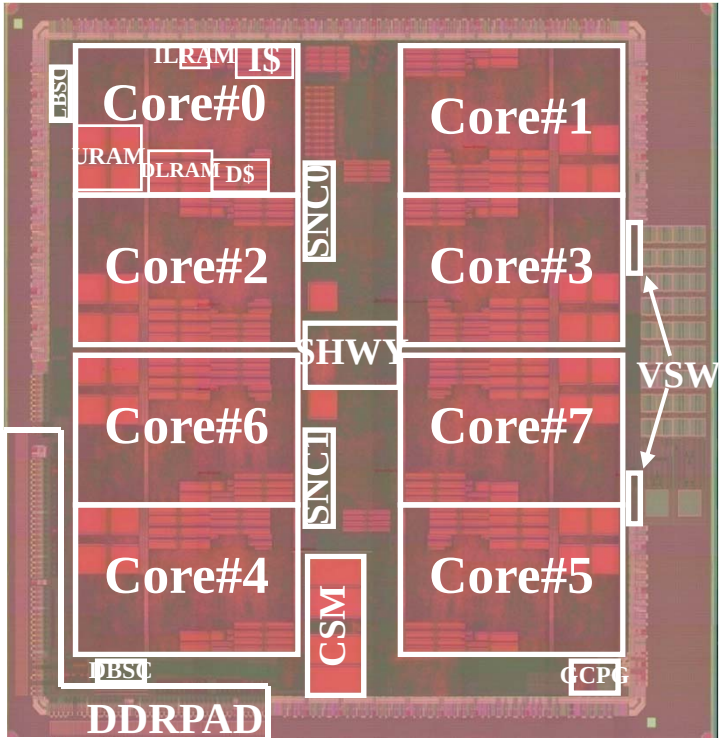
Automatic Cache and Local Memory Optimization for Multicores

Hironori Kasahara
Professor, Dept. of Computer Science & Engineering
Director, Advanced Multicore Processor Research Institute
Waseda University, Tokyo, Japan
IEEE Computer Society
President Elect 2017, President 2018
URL: http://www.kasahara.cs.waseda.ac.jp/

Multicores for Performance and Low Power

Power consumption is one of the biggest problems for performance scaling from smartphones to cloud servers and supercomputers (“K” more than 10MW) .



IEEE ISSCC08: Paper No. 4.5, M.ITO, … and H. Kasahara, “An 8640 MIPS SoC with Independent Power-off Control of 8 CPUs and 8 RAMs by an Automatic Parallelizing Compiler”

Power ∝ Frequency * Voltage^2

(Voltage ∝ Frequency)

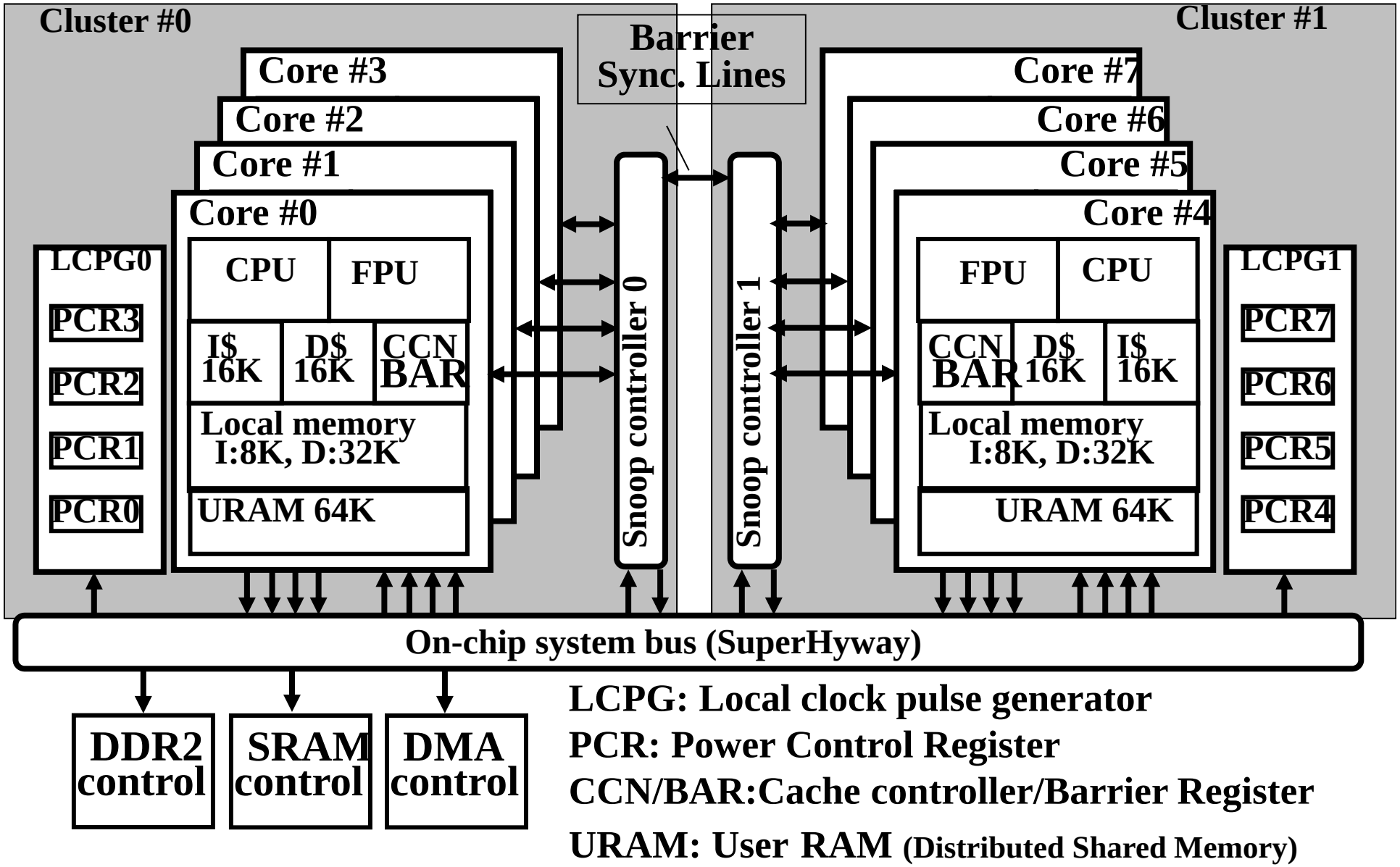
⇒ Power ∝ Frequency^3

If Frequency is reduced to 1/4 (Ex. 4GHz→1GHz), Power is reduced to 1/64 and Performance falls down to 1/4 .

<Multicores>

If 8cores are integrated on a chip, Power is still 1/8 and Performance becomes 2 times.

8 Core RP2 Chip Block Diagram



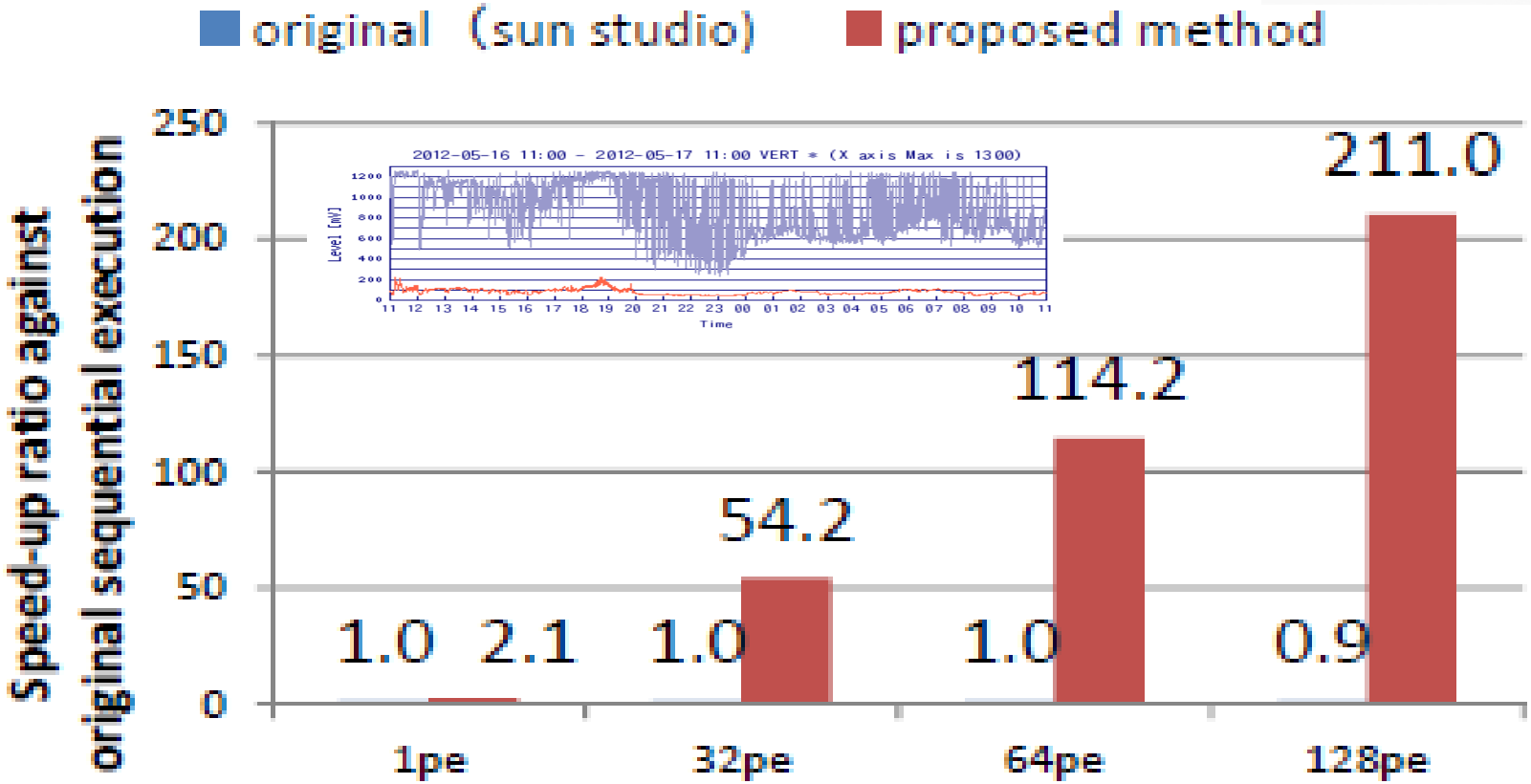
LCPG: Local clock pulse generator

PCR: Power Control Register

CCN/BAR:Cache controller/Barrier Register

URAM: User RAM (Distributed Shared Memory)

Earthquake Simulation "GMS" on Fujitsu M9000 Sparc CC-NUMA Server



With 128 cores, OSCAR compiler gave us 100 times speedup against 1 core execution and 211 times speedup against 1 core using Sun (Oracle) Studio compiler.

OSCAR Parallelizing Compiler

To improve effective performance, cost-performance and software productivity and reduce power

Multigrain Parallelization

coarse-grain parallelism among loops and subroutines, near fine grain parallelism among statements in addition to loop parallelism

Data Localization

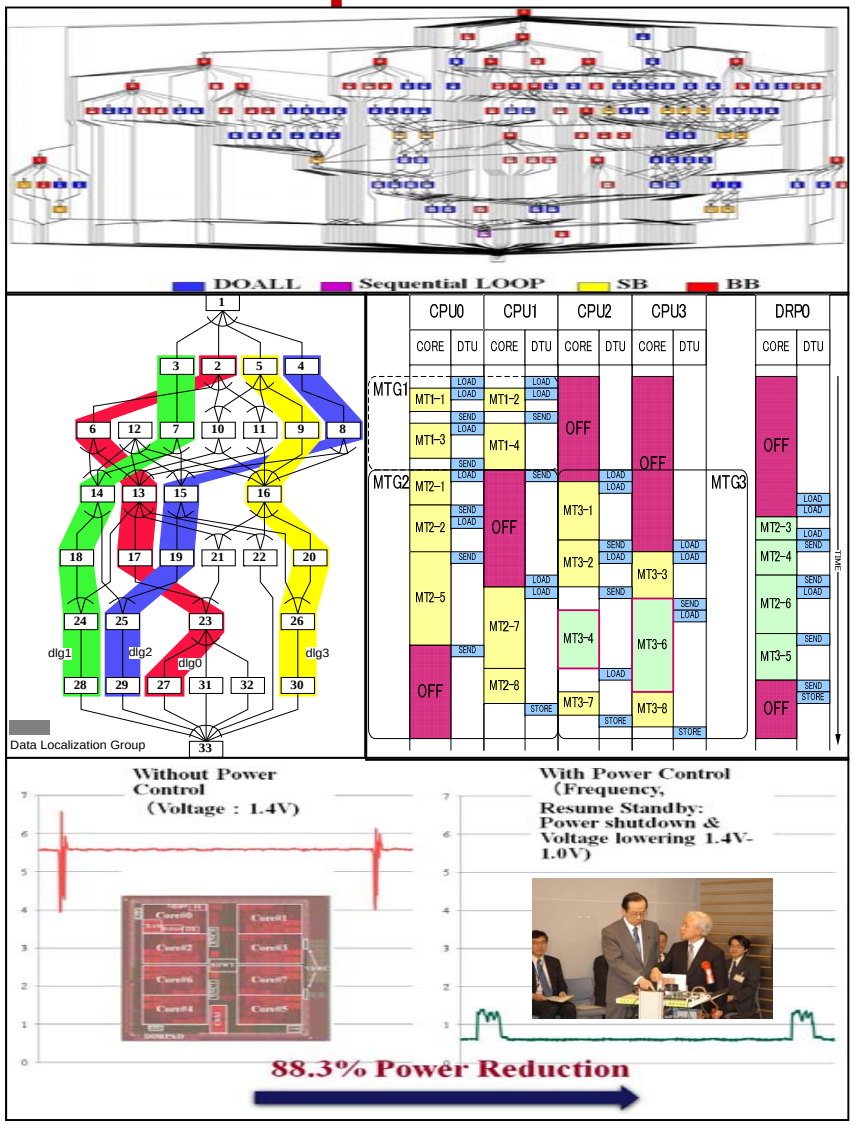
Automatic data management for distributed shared memory, cache and local memory

Data Transfer Overlapping

Data transfer overlapping using Data Transfer Controllers (DMAs)

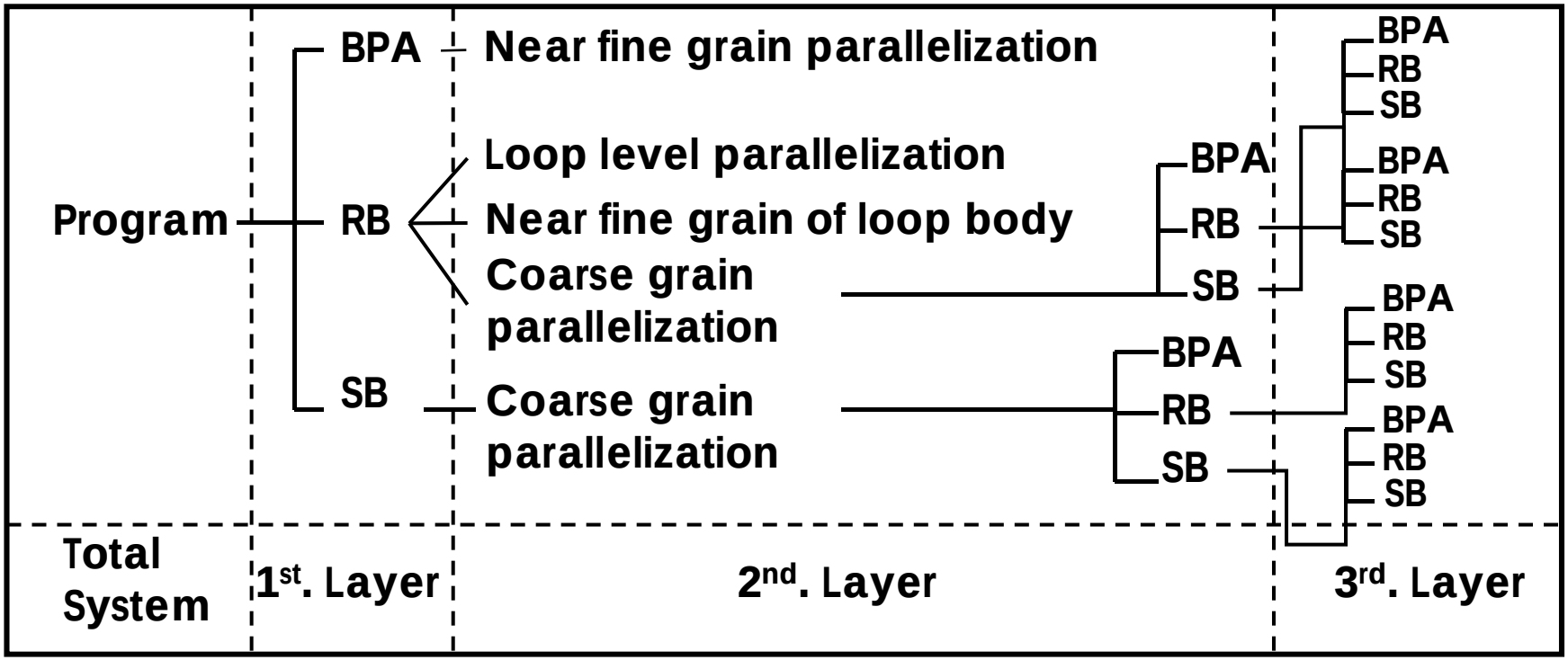
Power Reduction

Reduction of consumed power by compiler control DVFS and Power gating with hardware supports.

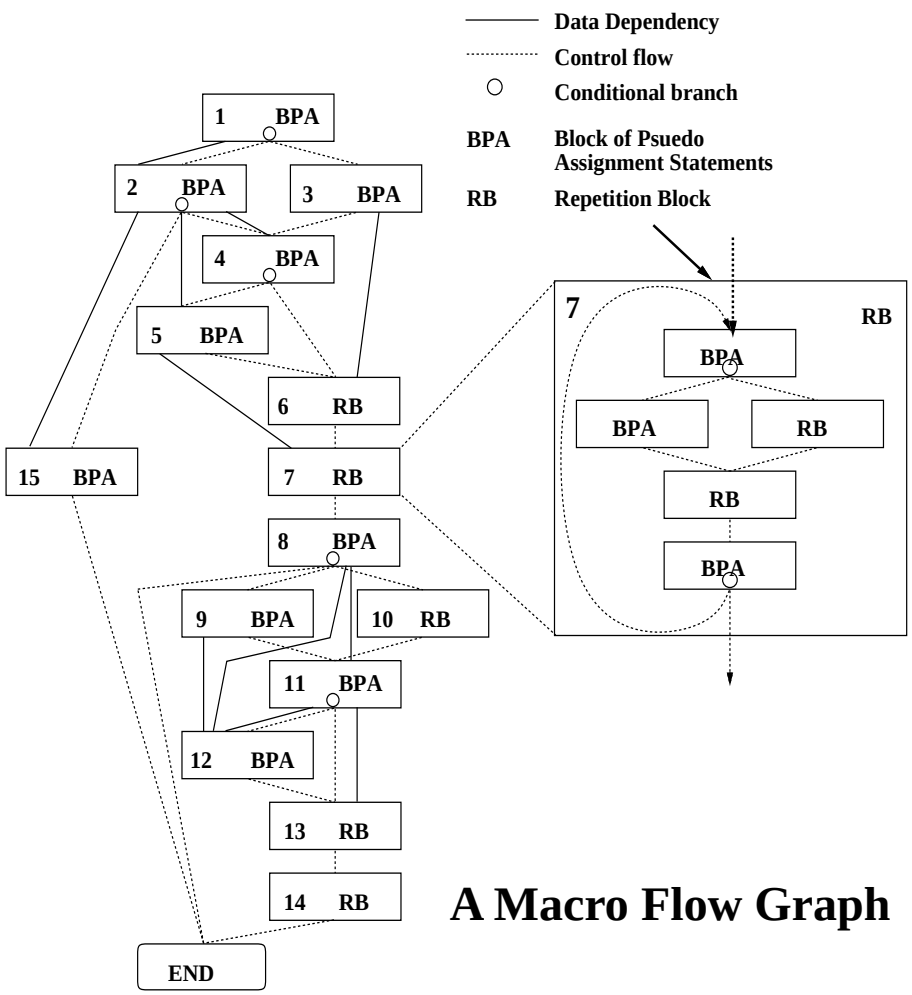


Generation of Coarse Grain Tasks

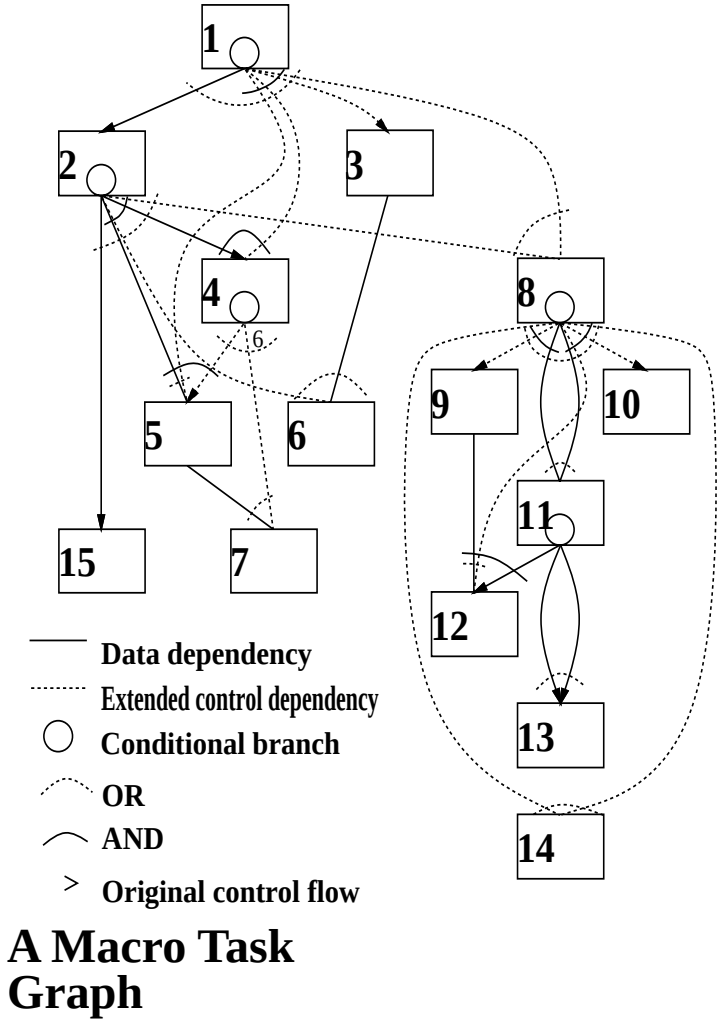
- **Macro-tasks (MTs)**
  - **Block of Pseudo Assignments (BPA): Basic Block (BB)**
  - **Repetition Block (RB) : natural loop**
  - **Subroutine Block (SB): subroutine**



Earliest Executable Condition Analysis for Coarse Grain Tasks (Macro-tasks)

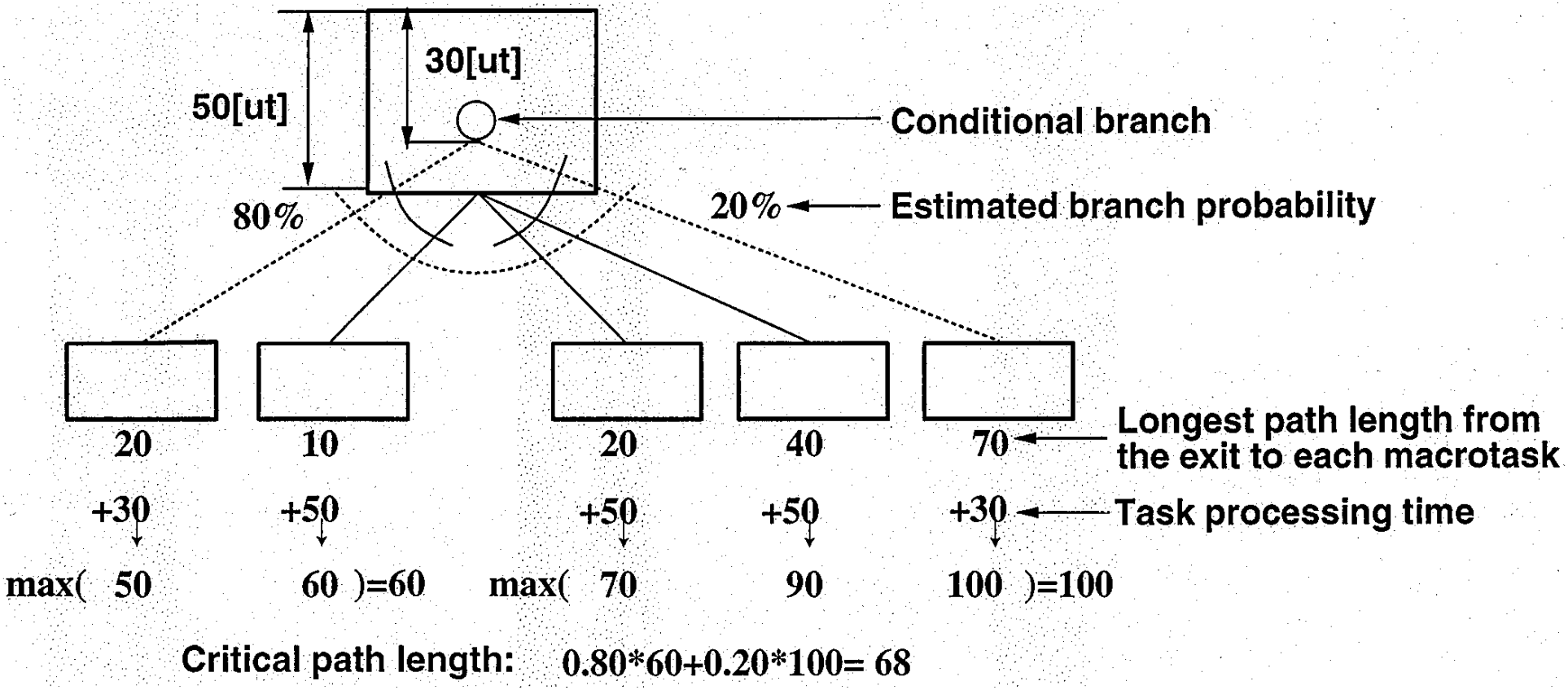


A Macro Flow Graph



A Macro Task Graph

PRIORITY DETERMINATION IN DYNAMIC CP METHOD



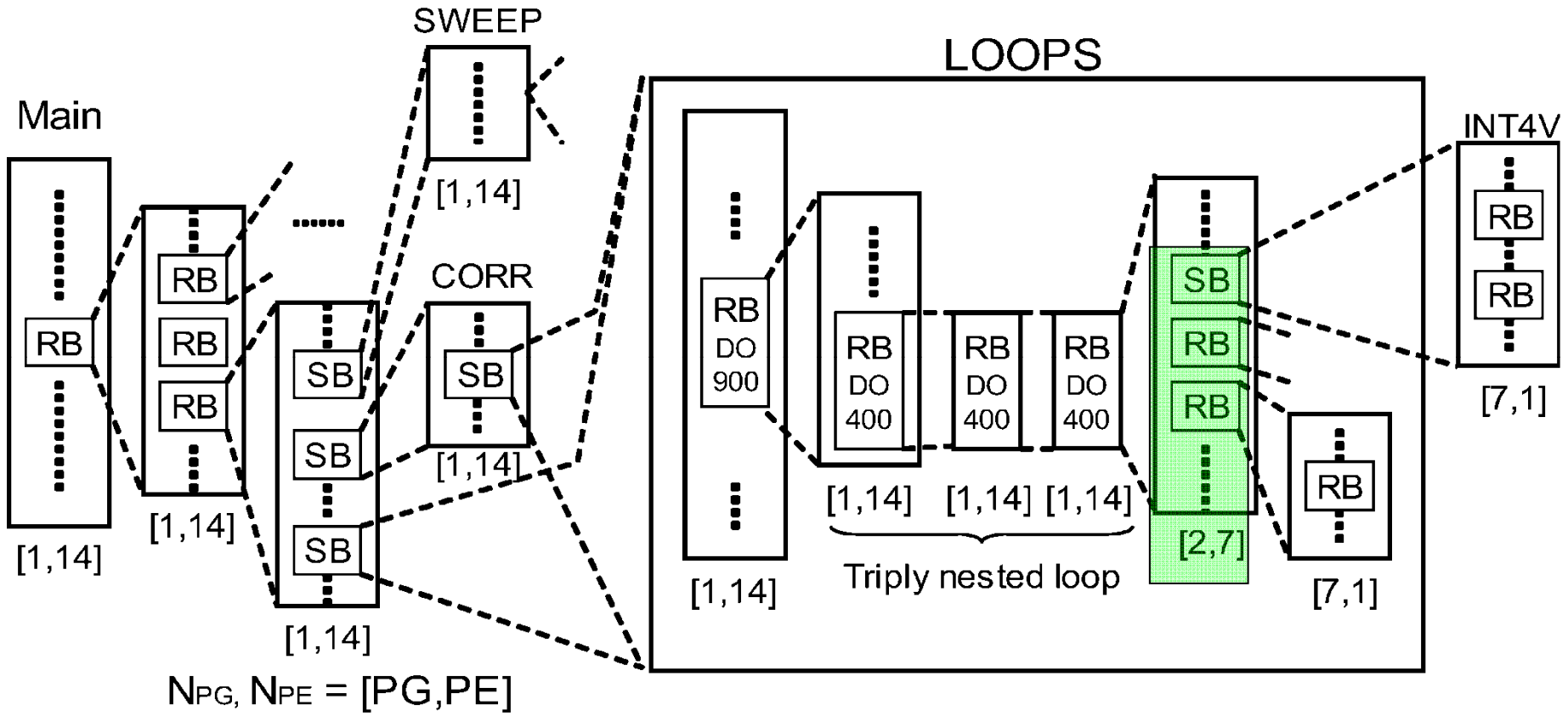
Earliest Executable Conditions

Macrotask No.	Earliest Executable Condition
1	
2	1_2
3	$(1)_3$
4	2_4 OR $(1)_3$
5	$(4)_5$ AND [2_4 OR $(1)_3$]
6	3 OR $(2)_4$
7	5 OR $(4)_6$
8	$(2)_4$ OR $(1)_3$
9	$(8)_9$
10	$(8)_{10}$
11	8_9 OR 8_{10}
12	11_{12} AND [9 OR $(8)_{10}$]
13	11_{13} OR 11_{12}
14	$(8)_9$ OR $(8)_{10}$
15	2_{15}

Automatic processor assignment in 103.su2cor

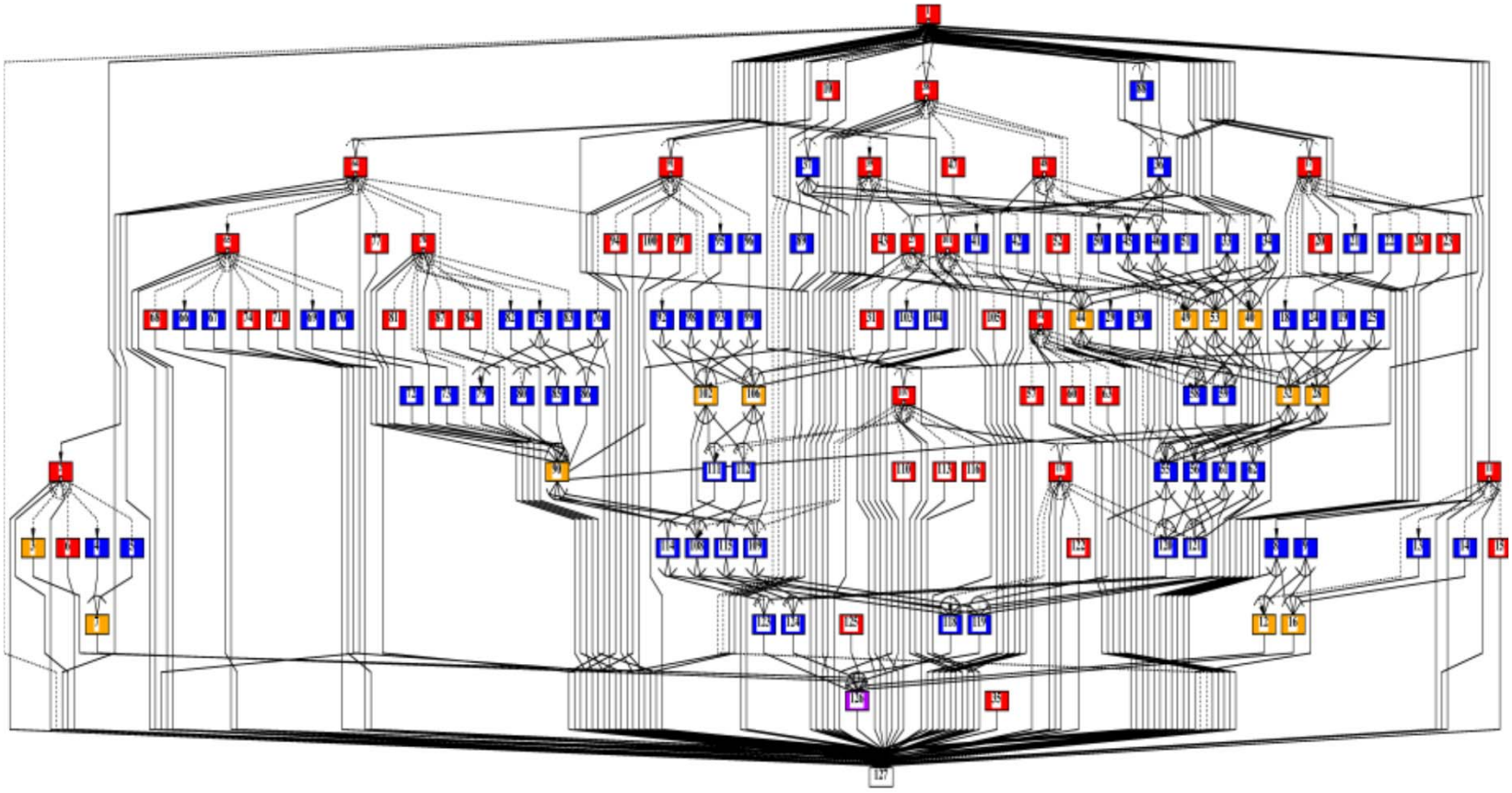
- **Using 14 processors**

Coarse grain parallelization within DO400



MTG of Su2cor-LOOPS-DO400

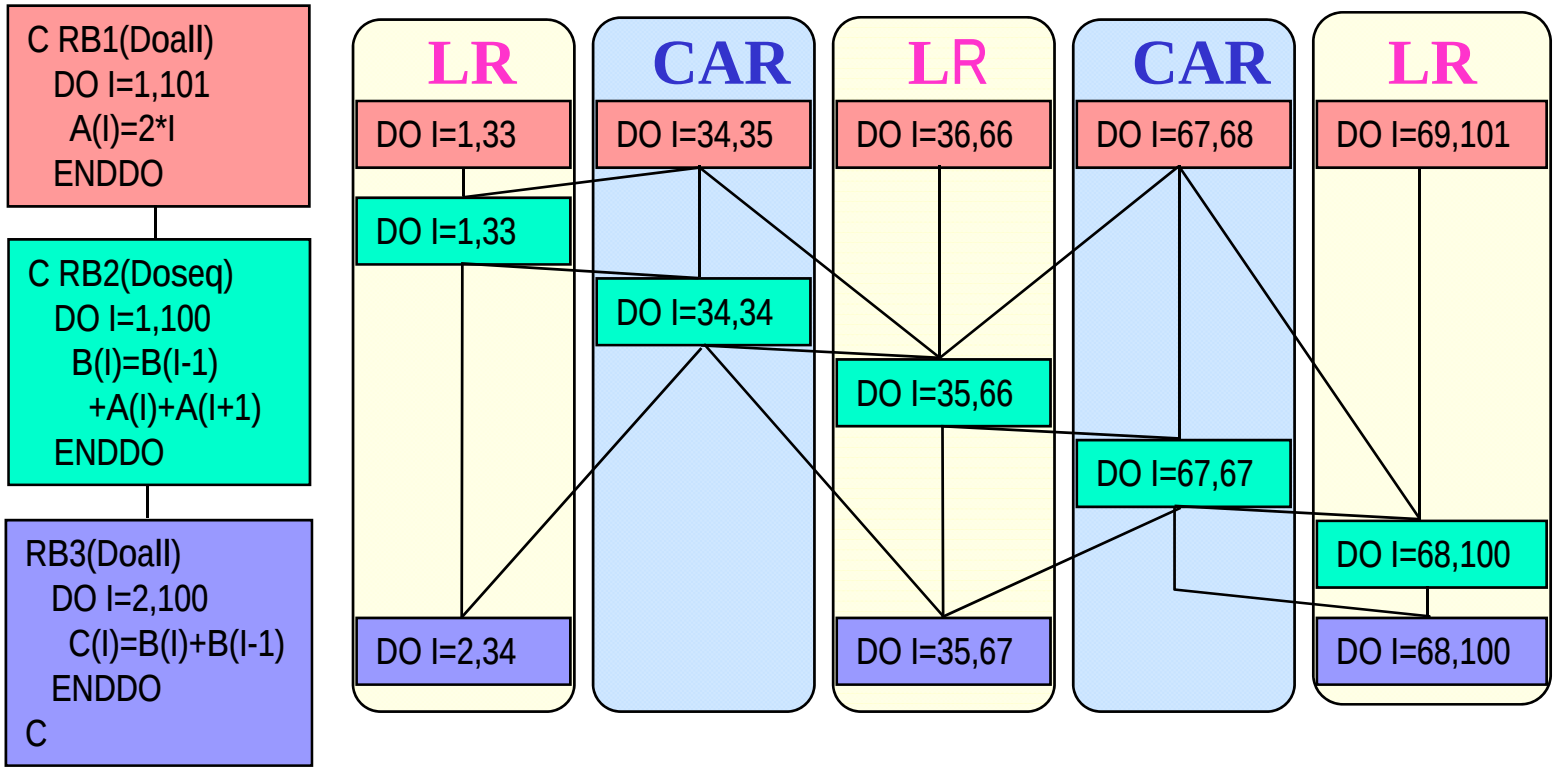
- **Coarse grain parallelism PARA_ALD = 4.3**



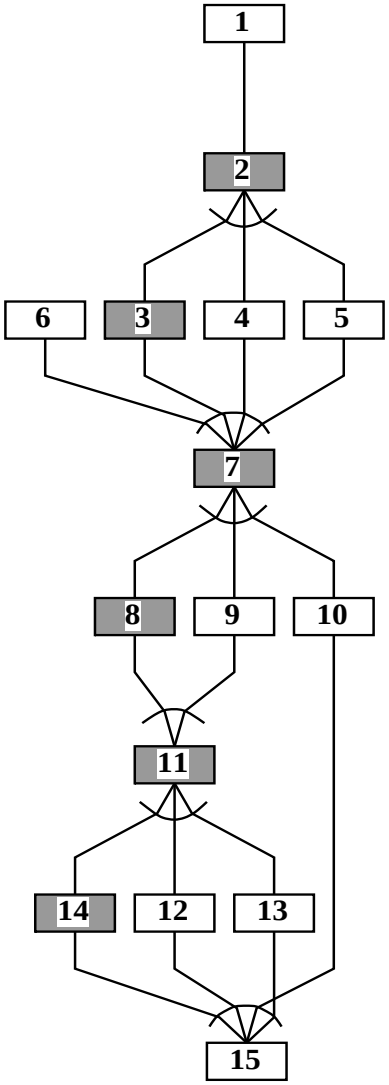
DOALL **Sequential LOOP** **SB** **BB**

Data-Localization: Loop Aligned Decomposition

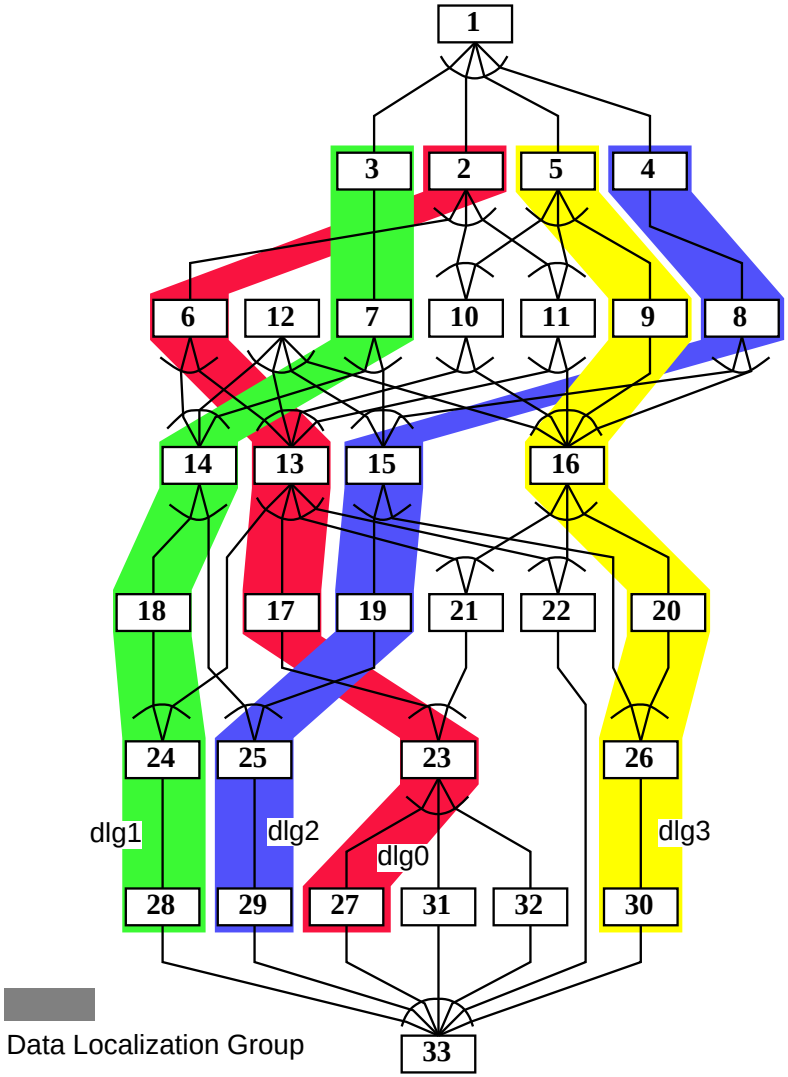
- **Decompose multiple loop (Doall and Seq) into CARs and LRs considering inter-loop data dependence.**
  - **Most data in LR can be passed through LM.**
  - **LR: Localizable Region, CAR: Commonly Accessed Region**



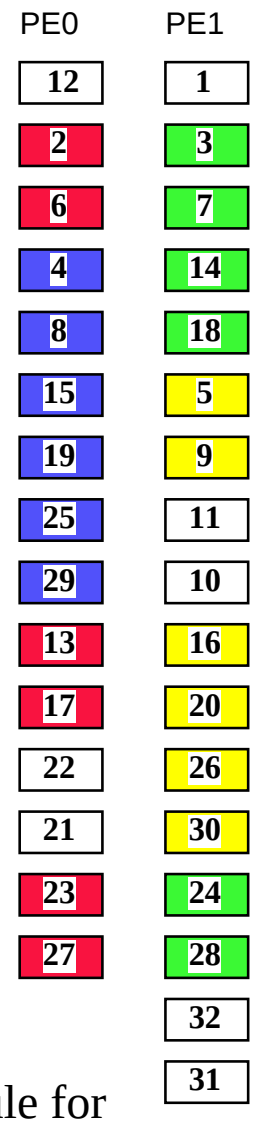
Data Localization



MTG



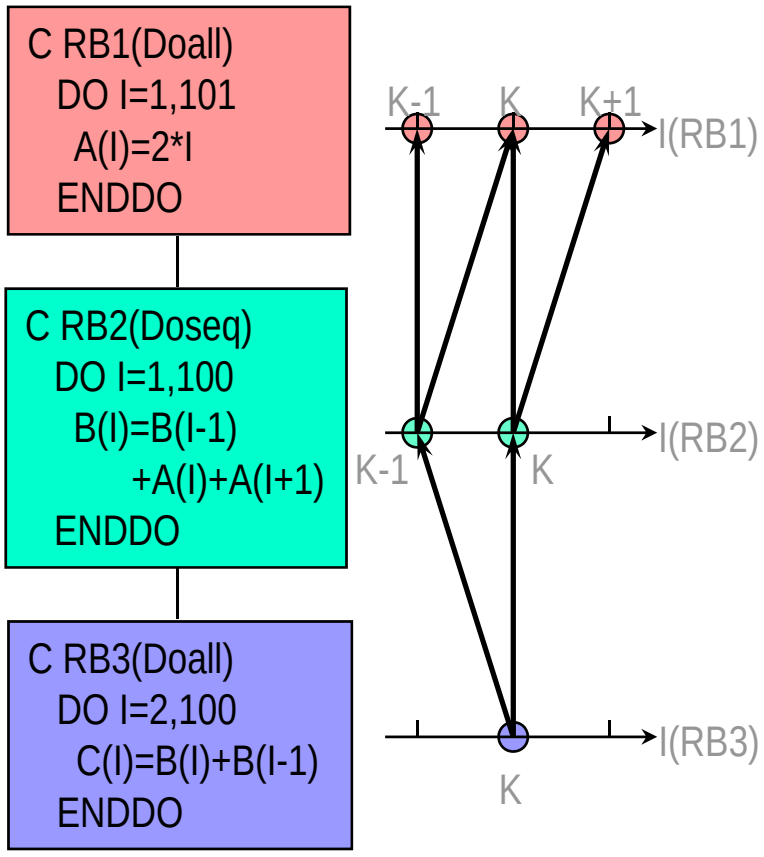
MTG after Division



A schedule for two processors

Inter-loop data dependence analysis in TLG

- Define exit-RB in TLG as Standard-Loop
- Find iterations on which a iteration of Standard-Loop is data dependent
  - e.g. K_{th} of RB3 is data-dep on $K\text{-}1_{th}$,K_{th} of RB2, on $K\text{-}1_{th}$,K_{th},$K\text{+}1_{th}$ of RB1



Example of TLG

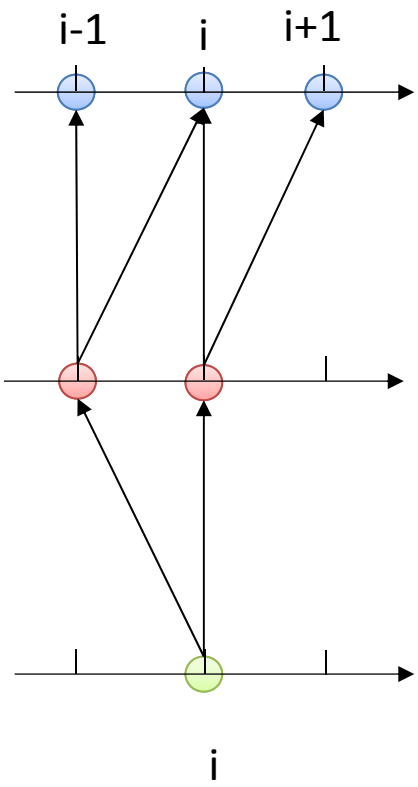
Target Loop Group Creation and Inter-Loop Dependence Analysis

- Target Loop Groups
  - grouped loops that access the same array
  - baseline loop chosen for each group
    - the largest estimated time loop
- Inter-Loop Dependency Analysis
  - data dependencies between loops within the TLGs
  - detects relevant iterations of those loops that have dependence with the iterations of the baseline loop

```
DO I=1,101
   A(I)=2*I
ENDDO
```

```
DO I=1,100
   B(I)=B(I-1)
  +A(I)+A(I+1)
ENDDO
```

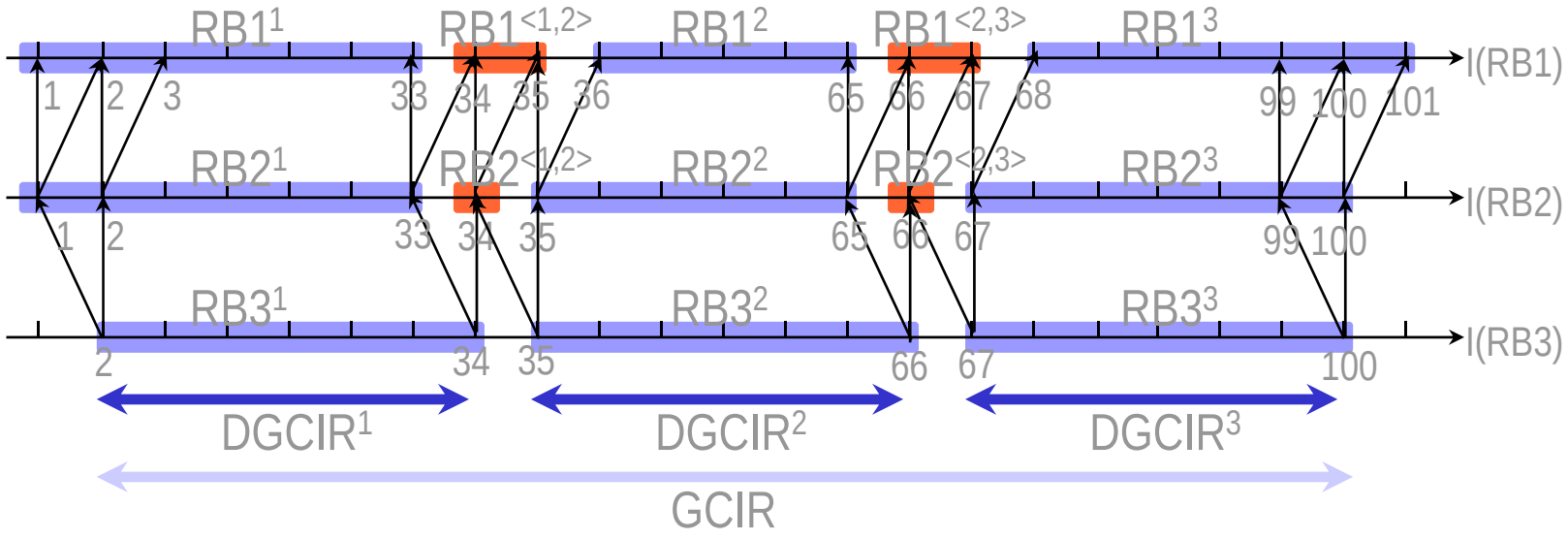
```
DO I=2,100
   C(I)=B(I)*B(I-1)
ENDDO
```



Inter-Loop dependence

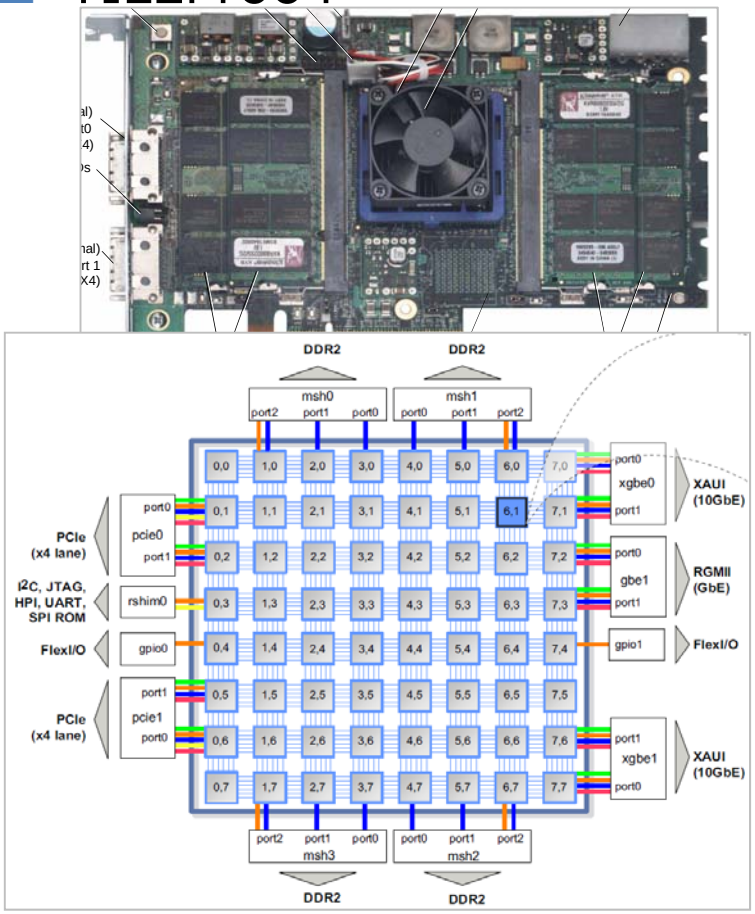
Decomposition of RBs in TLG

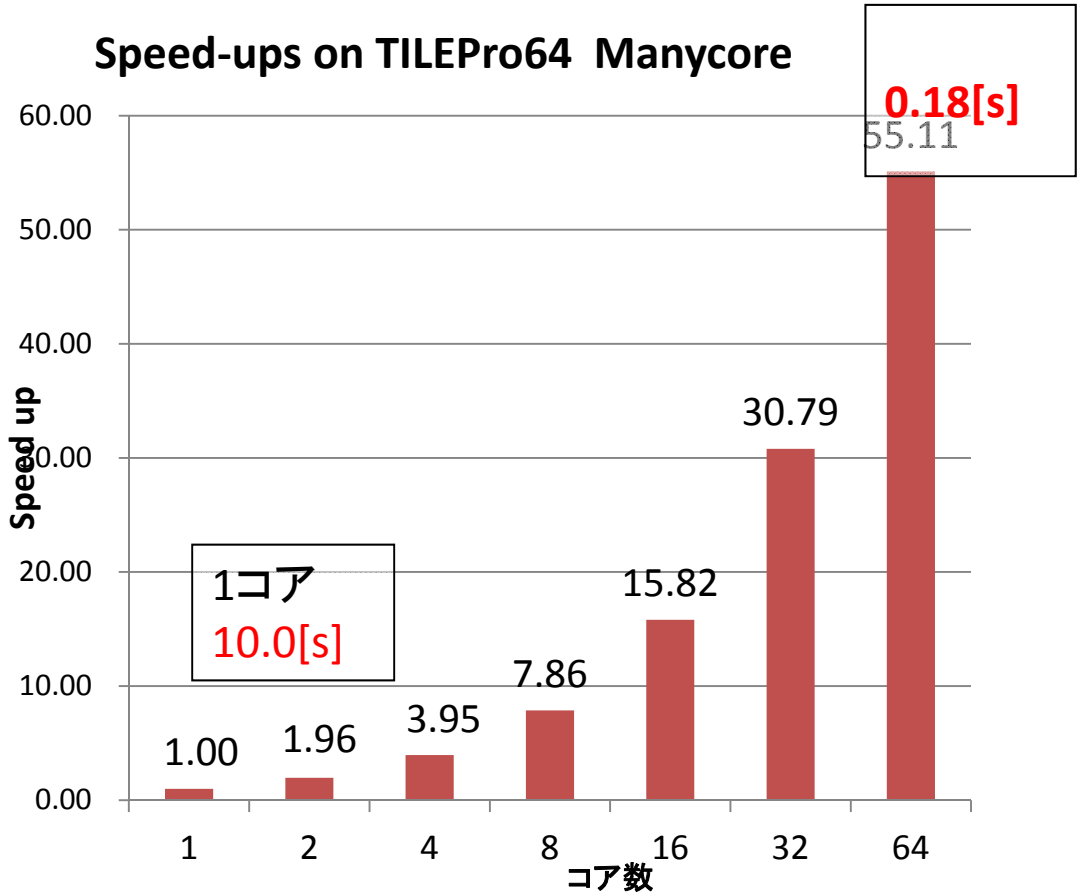
- Decompose GCIR into $DGCIR^p(1 \leqq p \leqq n)$
  - n: (multiple) num of PCs, DGCIR: Decomposed GCIR
- Generate CAR on which $DGCIR^p$&$DGCIR^{p+1}$ are data-dep.
- Generate LR on which $DGCIR^p$ is data-dep.



Automatic Parallelization of Still Image Encoding Using JPEG-XR for the Next Generation Cameras and Drinkable Inner Camera

- TILEPro64

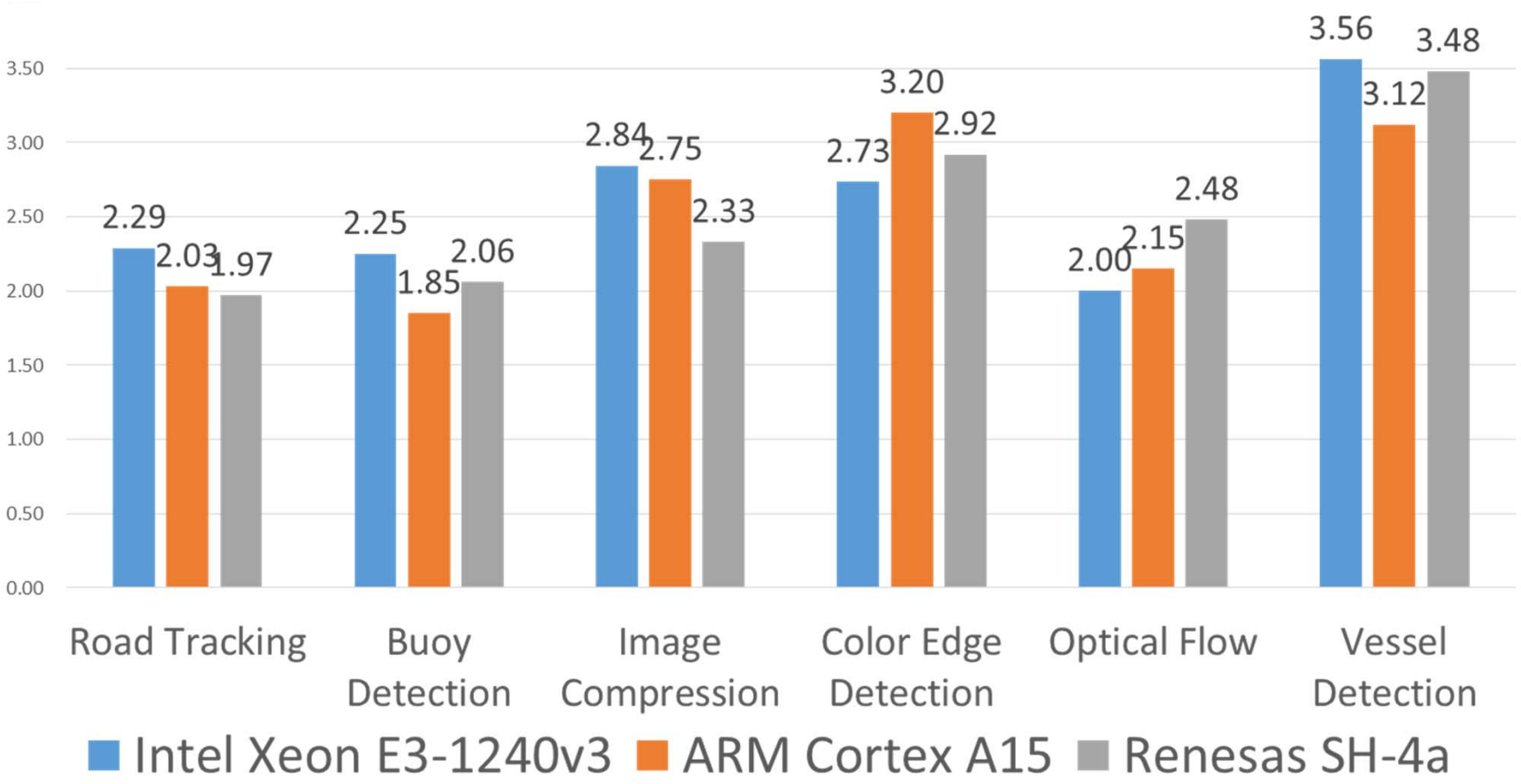




55 times speedup with 64 cores against 1 core

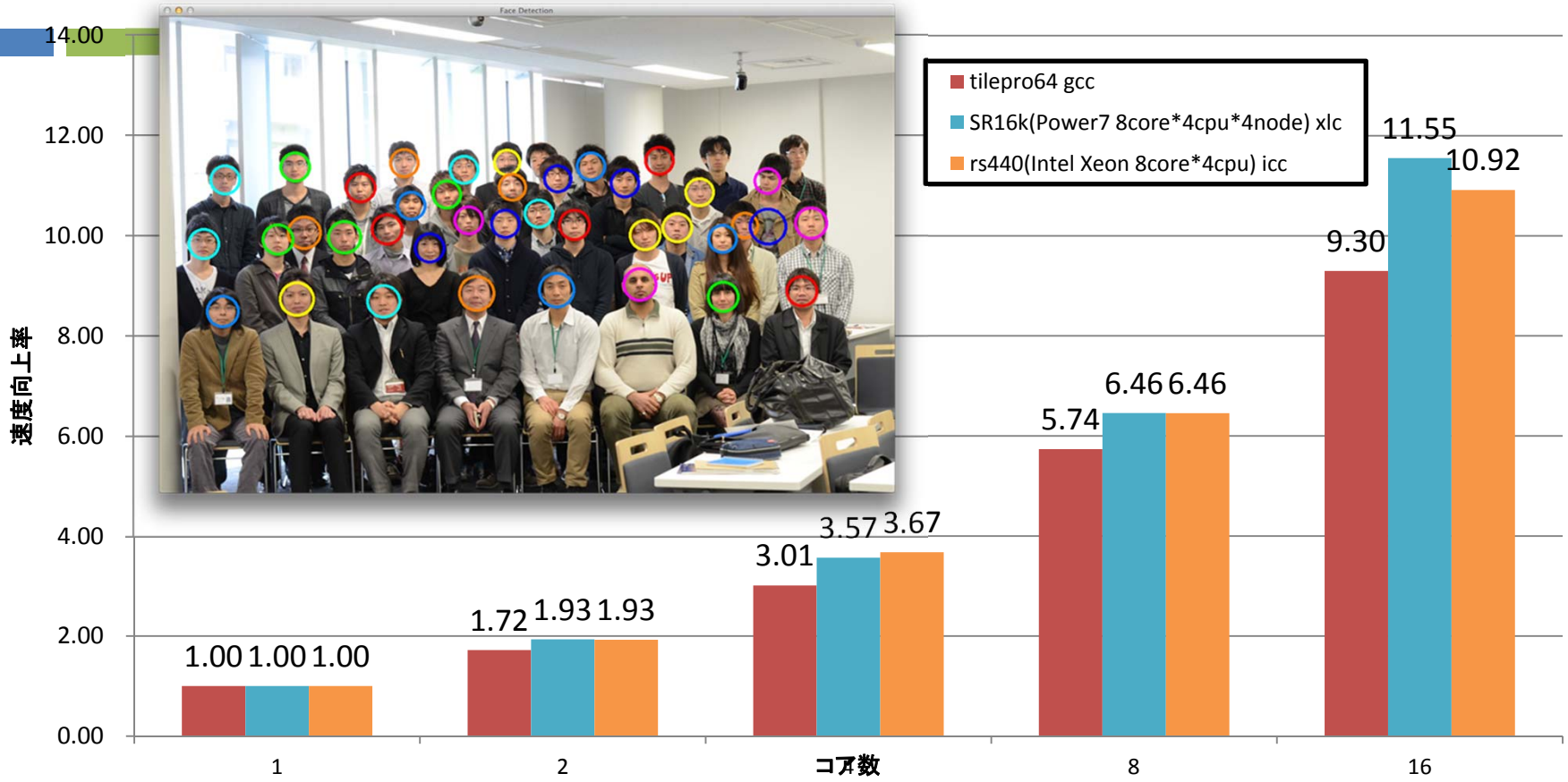
Speedups of MATLAB/Simulink Image Processing on Various 4core Multicores

(Intel Xeon, ARM Cortex A15 and Renesas SH4A)



Road Tracking, Image Compression : http://www.mathworks.co.jp/jp/help/vision/examples
Buoy Detection : http://www.mathworks.co.jp/matlabcentral/fileexchange/44706-buoy-detection-using-simulink
Color Edge Detection : http://www.mathworks.co.jp/matlabcentral/fileexchange/28114-fast-edges-of-a-color-image--actual-color--not-converting-to-grayscale-/
Vessel Detection : http://www.mathworks.co.jp/matlabcentral/fileexchange/24990-retinal-blood-vessel-extraction/

Parallel Processing of Face Detection on Manycore, Highend and PC Server

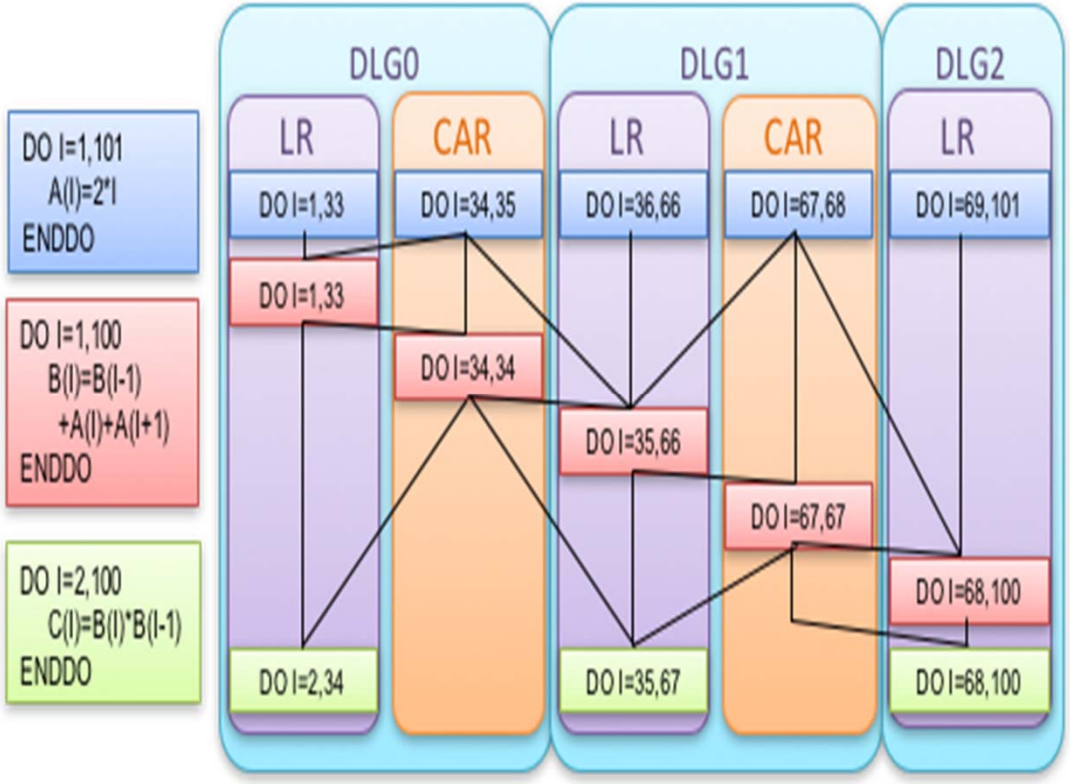


- OSCAR compiler gives us 11.55 times speedup for 16 cores against 1 core on SR16000 Power7 highend server.

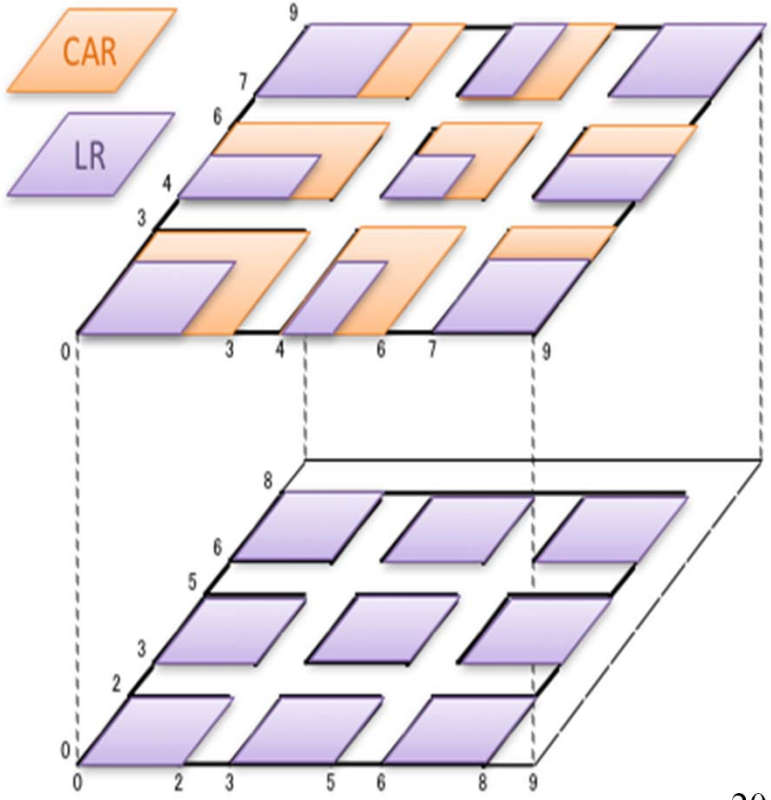
Data Localization: Loop Aligned Decomposition

- **Decomposed loop into LRs and CARs**
  - **LR (Localizable Region): Data can be passed through LDM**
  - **CAR (Commonly Accessed Region): Data transfers are required among processors**

Single dimension Decomposition

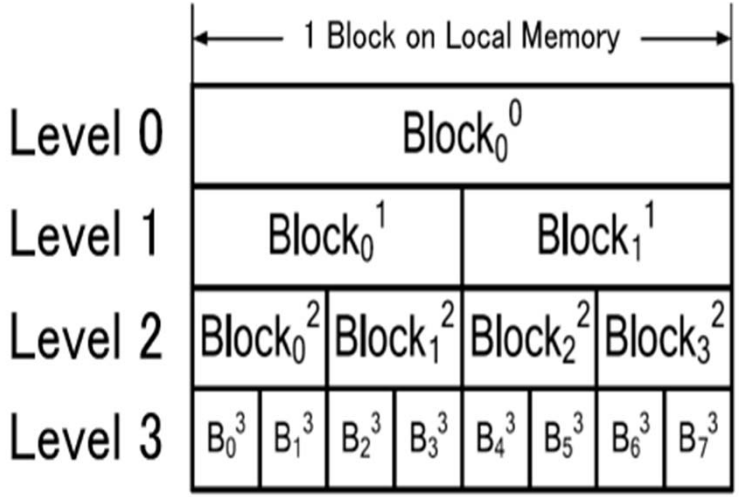


Multi-dimension Decomposition



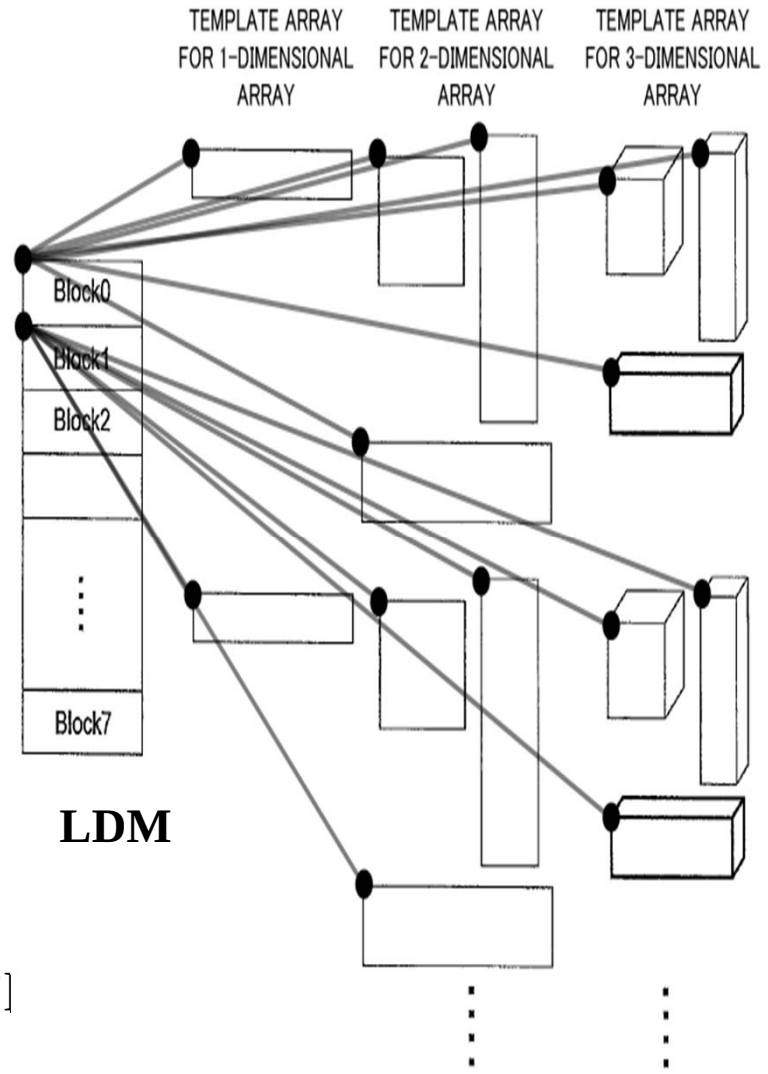
Adjustable Blocks

- Handling a suitable block size for each application
  - different from a fixed block size in cache
  - each block can be divided into smaller blocks with intege $\mathrm{Block}_{\mathrm{Number}}^{\mathrm{Level}}$ small arrays and scalar



Multi-dimensional Template Arrays for Improving Readability

- a mapping technique for arrays with varying dimensions
  - each block on LDM corresponds to multiple empty arrays with varying dimensions
  - these arrays have an additional dimension to store the corresponding block number
    - TA[Block#][] for single dimension
    - TA[Block#][][] for double dimension
    - TA[Block#][][][] for triple dimension
    - ...
- LDM are represented as a one dimensional array
  - without Template Arrays, multi-dimensional arrays have complex index calculations
    - A[i][j][k] -> TA[offset + i' * L + j' * M + k']
  - Template Arrays provide readability
    - A[i][j][k] -> TA[Block#][i'][j'][k']



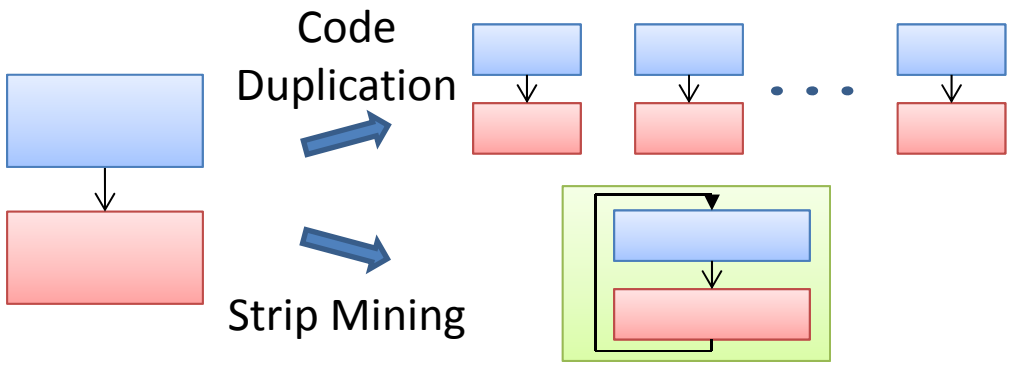
LDM

Block Replacement Policy

- Compiler Control Memory block Replacement
  - using live, dead and reuse information of each variable from the scheduled result
  - different from LRU in cache that does not use data dependence information
- Block Eviction Priority Policy
  1. (Dead) Variables that will not be accessed later in the program
  2. Variables that are accessed only by other processor cores
  3. Variables that will be later accessed by the current processor core
  4. Variables that will immediately be accessed by the current processor core

Code Compaction by Strip Mining

- Previous approach produces duplicate code
  - generates multiple copies of the loop body which leads to code bloat
- Proposed method adopts code compaction
  - based on strip mining
  - multi-dimensional loop can be restructured



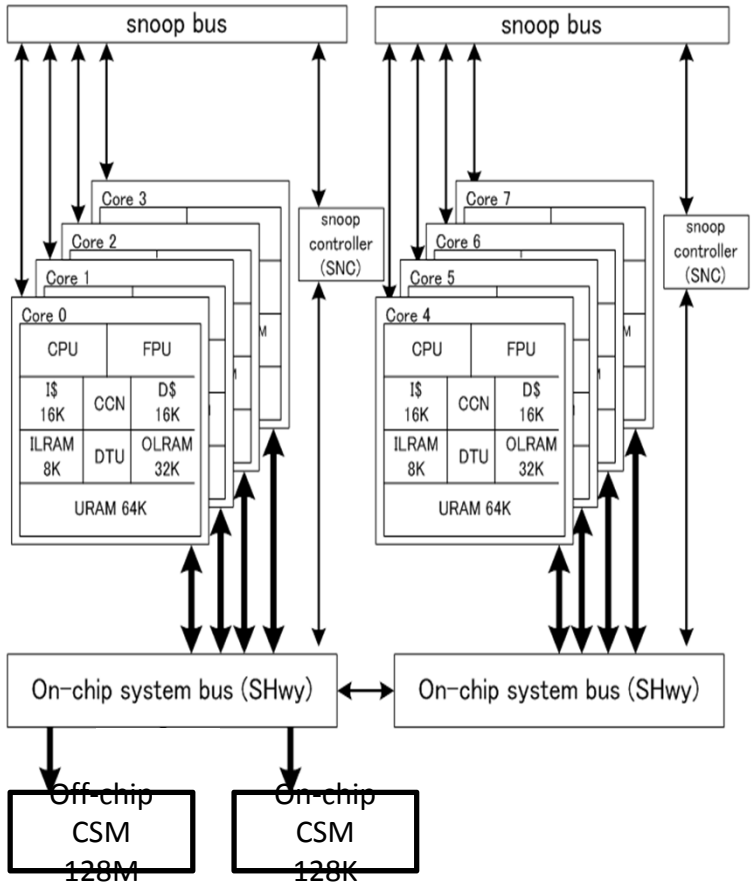

```
for (i = 0; i < 16; i++)
    for (j = 0; j < 64; j++)
        a[i][j] = i + j;

for (i = 0; i < 15; i++)
    for (j = 0; j < 63; j++)
        b[i][j] = a[i][j] + a[i+1][j+1];
```

⇩

```
for (ii = 0; ii < 15; ii+=8)
    for (jj = 0; jj < 63; jj+=32)
        for (i = ii; i < min(15,ii+8+1); i++)
            for (j = jj; j < min(63,jj+32+1); j++)
                a[i][j] = i + j;
        for (i = ii; i < min(15,ii+8); i++)
            for (j = jj; j < min(63,jj+32); j++)
                b[i][j] = a[i][j] + a[i+1][j+1];
```

Evaluation Environment

- Implemented on the OSCAR compiler
- Tested on RP2
  - SH4A with 600MHz processor based
  - 8 core homogeneous multicore processor
  - each processor core has 16KB LDM
    with a 1 clock cycle latency
  - equipped with 128MB DDR2 off-chip
    CSM (Central Shared Memory)
    with a 55 clock cycle latency

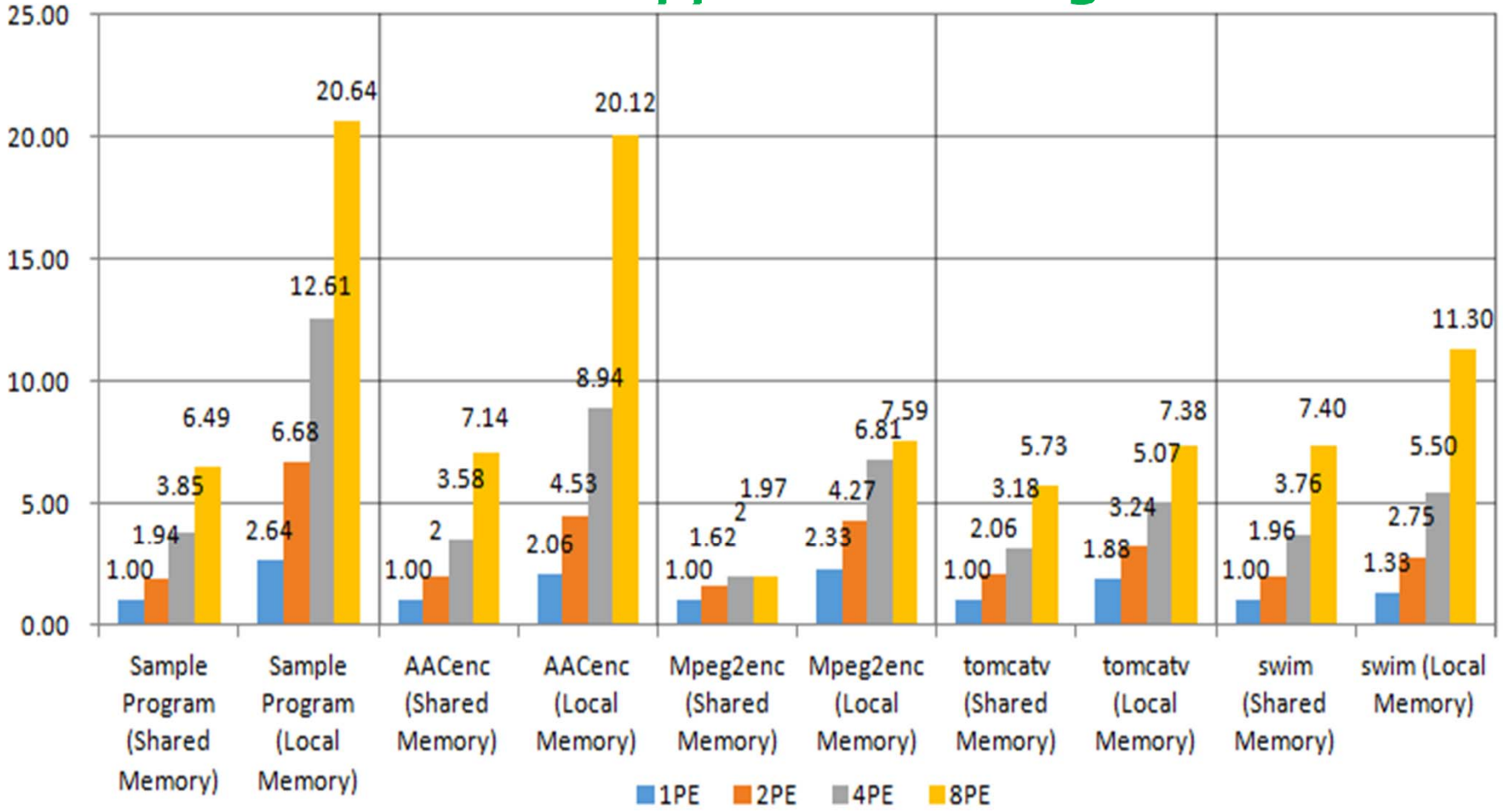


Architecture of the RP2 Multicore Pro

Applications for Evaluation

- Sequential C Applications
  - Example code in explanation of code compaction
  - AACenc (provided by Renesas Technology)
    - AAC encoder, input: a 30 second audio file
  - Mpeg2enc (part of the MediaBench benchmark suite)
    - MPEG2 encoder, input: a 30 frame video with a resolution of 352 by 256 pixels
  - SPEC95 Tomcatv
    - loop fusion and variable renaming were applied
  - SPEC95 Swim
    - loop distribution and loop peeling were performed

Speedups by the Proposed Local Memory Management Compared with Utilizing Shared Memory on Benchmarks Application using RP2



20.12 times speedup for 8cores execution using local memory against sequential execution using off-chip shared memory of RP2 for the AACenc

Conclusions

- This talk introduced automatic cache and local memory management method using data localization with hierarchical loop aligned decomposition, adjustable block tailored for each application, and block replace considering block reuse distance .
- The local memory management method was implemented on the OSCAR parallelization compiler.
- The performance on the RP2 8 core multicore gave us
- for example,
  - 20.12 times speedup on 8cores using local memory against sequential execution using off-chip shared memory for the AAC encoder though the 8 core execution using
  - shared memory gave us 7.14 times speedup.
  - 11.30 times speedup on 8cores execution using local memory against sequential execution using off-chip shared memory for the SPEC95 swim though the 8 core execution using shared memory gave us 7.40 times speedup.