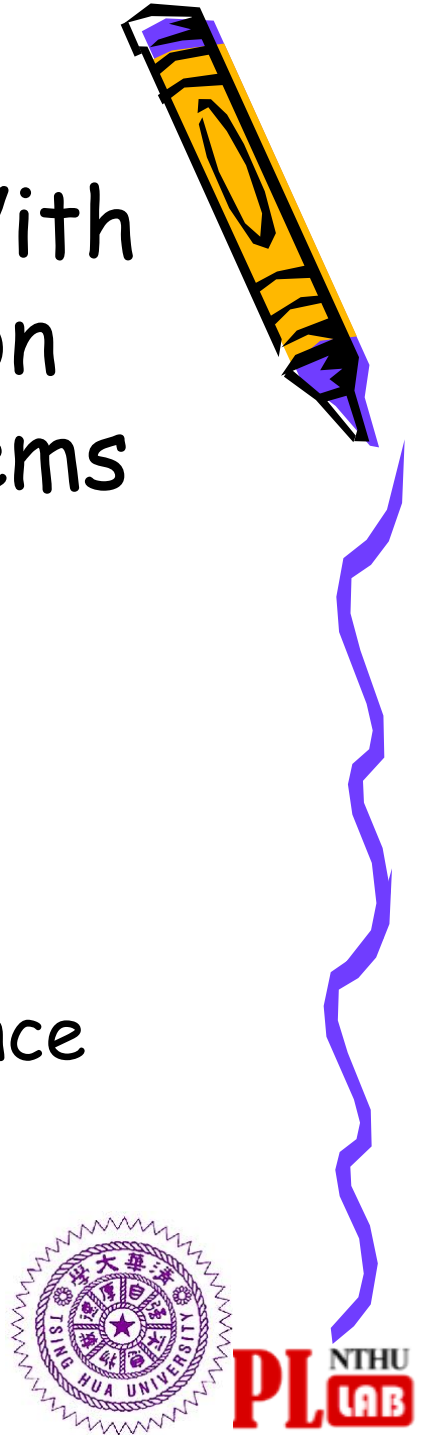
Compilers for Low Power With Parallel Design Patterns on Embedded Multicore Systems

Jenq Kuen Lee
Kenny Lin Wen-Li Shih
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan

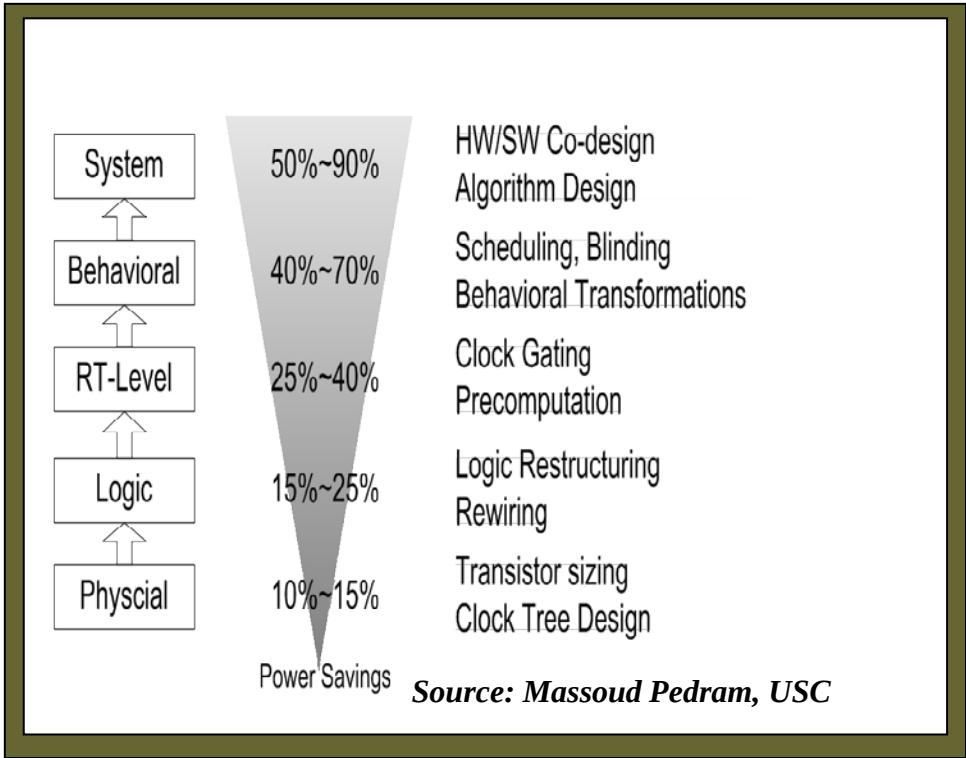


Outline

- History: Compilers for Low-Power
- Pattern-Based Power Optimizations
  - BSP Model
  - Producer-N-Consumer
  - MapReduce
  - Coefficient Objects
- Experiments
- Conclusion

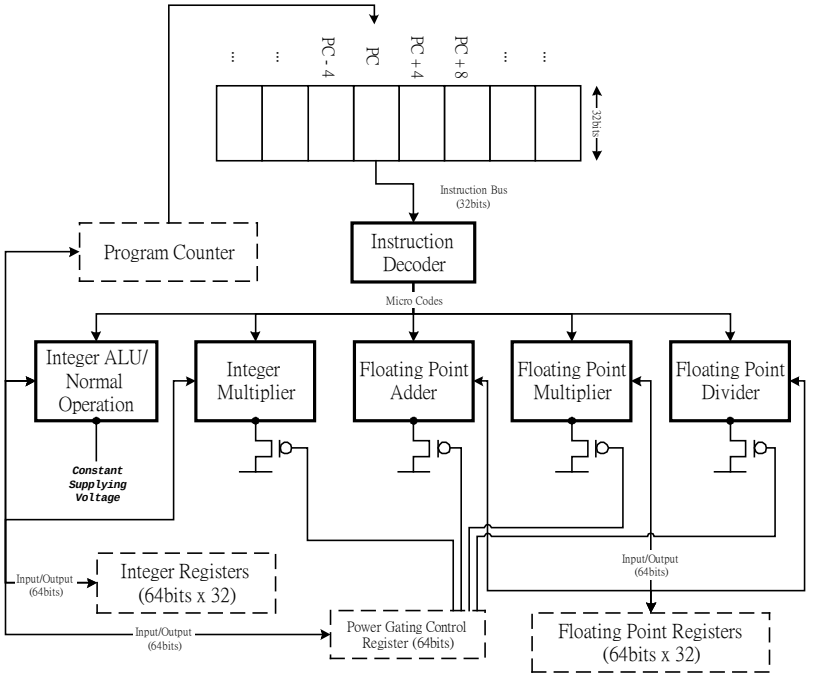
Low Power Design

- Power optimization are needed at all levels.
- Higher levels of application and system layer impact the power decision most.
- Software for Low-Power
  - Applications
  - OS
  - Compilers
  - Runtime Systems
  - Power Simulator



Compilers for Power-Gating (Static Power)

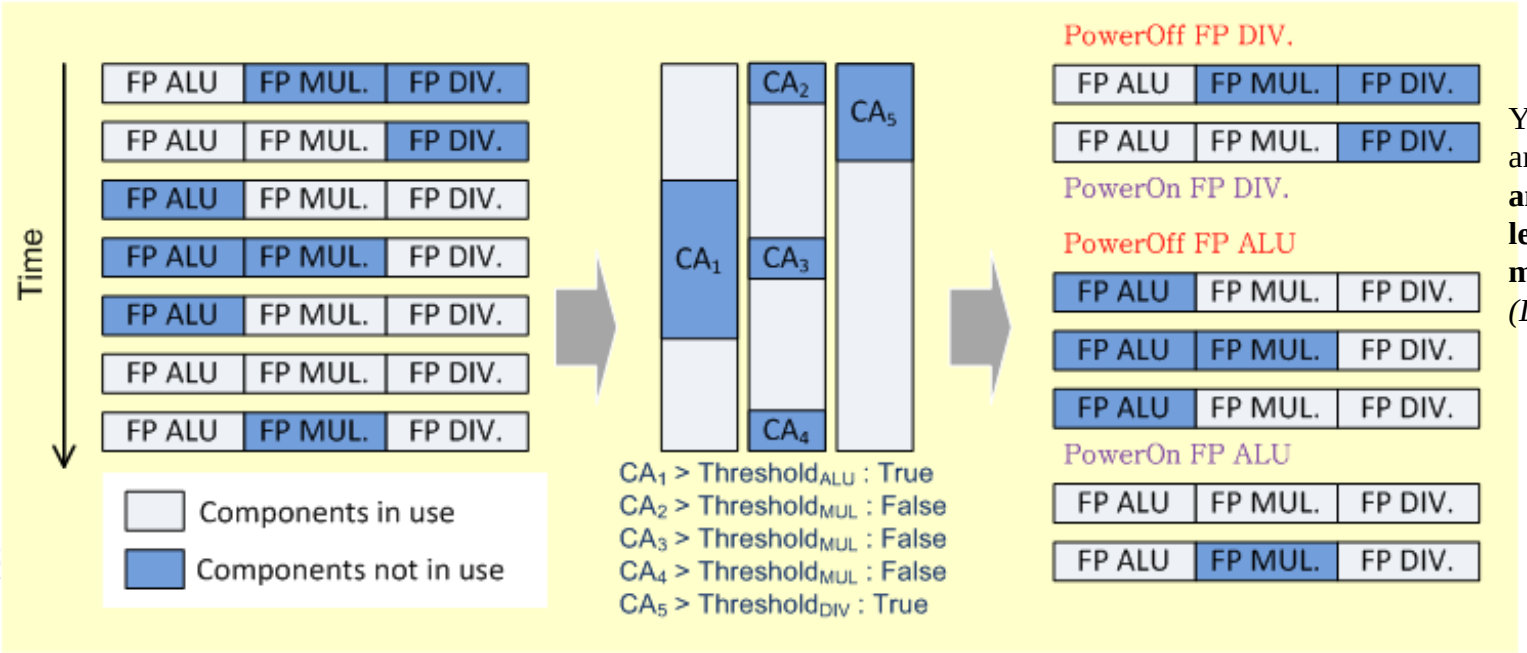
- Compiler frameworks employ power-gating instructions to reduce static powers
- To turn off useless components in processors
- Use compiler analysis techniques to analyze program behaviors



"Compilers for Leakage Power Reductions, Yi-Ping You, Ching-Ren Lee, Jenq-Kuen Lee, ACM Transactions on Design Automation of Electronic Systems, Jan. 2006.

Component Activity Data-Flow Analysis

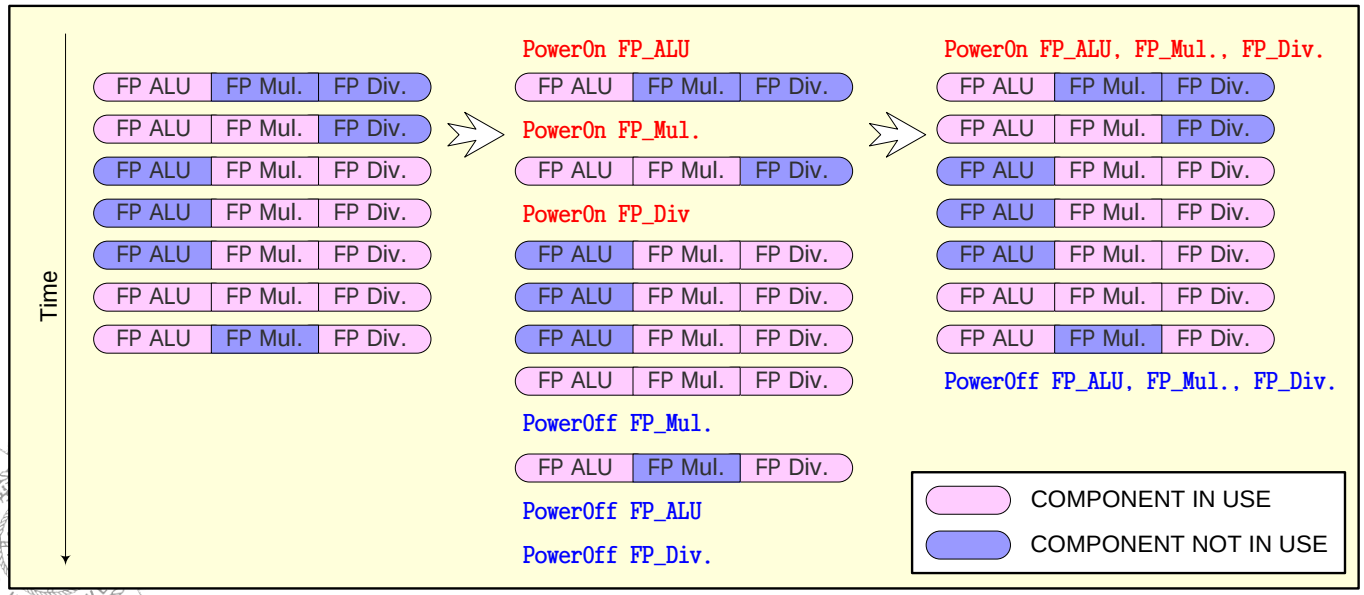
- Compiler frameworks employ power-gating instructions to reduce static powers.
- Use compiler analysis techniques to analyze program behaviors.
- To turn off useless components in processors.
- Wake up the components ahead of time considering the cycles needed for components to be ready.
- Consider Break-Even point/cost-model and incorporate edge profiling and branching situations.
- Suggest possible ways to work with Out-of-Order architectures.



Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. **Compiler analysis and supports for leakage power reduction on microprocessors.** *In (LCPC'02)*, pages 63–73,

Compact Power-Gating Control Placement

- The amount of power-gating instructions being added increases as the increasing amount of components equipped with power-gating control.
- Solution: code motion of power-gating instructions
  - "sink" power-off instructions
  - "hoist" power-on instructions



Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. **A Sink-N-Hoist framework for leakage power reduction.** *In Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*, pages 83–94, Jersey City, New Jersey, USA, September 2005.

Leakage Power Reduction Framework

i. ICFG construction

ii. Component-Activity Data-Flow Analysis

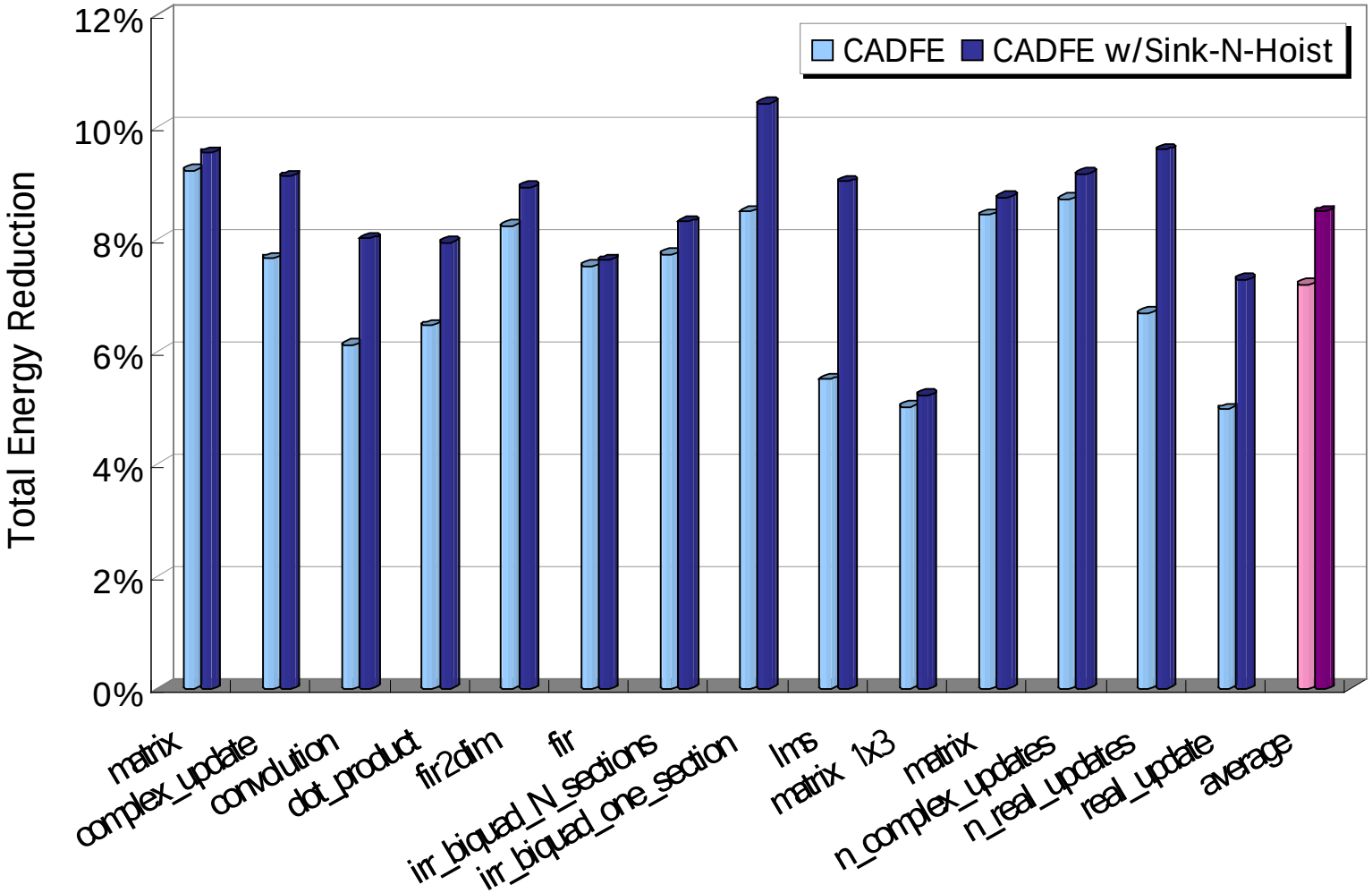
iii. Power-gating instruction scheduling

iv. Sink-N-Hoist Analysis

v. Power-gating instruction generation

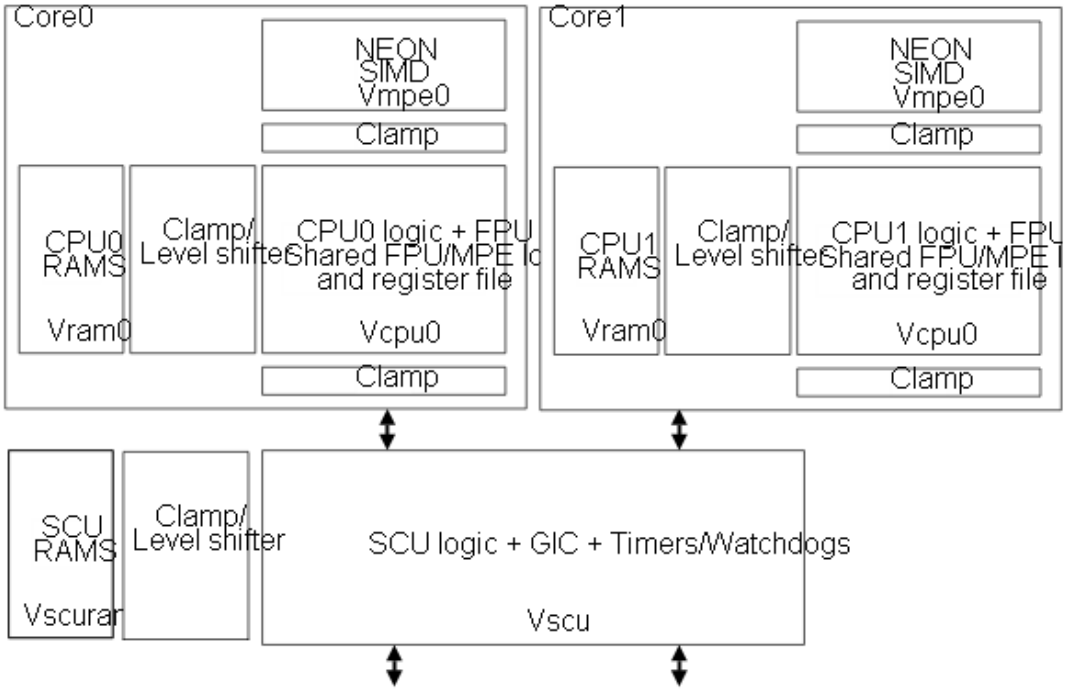
"A Sink-N-Hoist Framework for Leakage Power Reduction",
Yi-Ping You, C. W. Huang, Jenq-Kuen Lee, ACM EMSOFT, Sep. 2005.
(Also extended version in ACM TODAES 2007)

Results: Total Power Reduction



The Low Power Hardware Design Trend

- Power Gating Architecture
  - ARM Cortex A9 MPCore
    - Different Power Domains
    - Up to 14 Power Domains
      - Cortex-A9 processors(4)
      - Data Engine(4)
      - Processors Cache and TLB RAMs(4)
      - SCU duplicated TAG RAMs
      - SCU logic cells and private peripherals
    - Power Gating
      - Four running modes
        - Run, Standby, Dormant, Shutdown
        - Fine-grained pipeline shutdown
        - Faster register save and restore



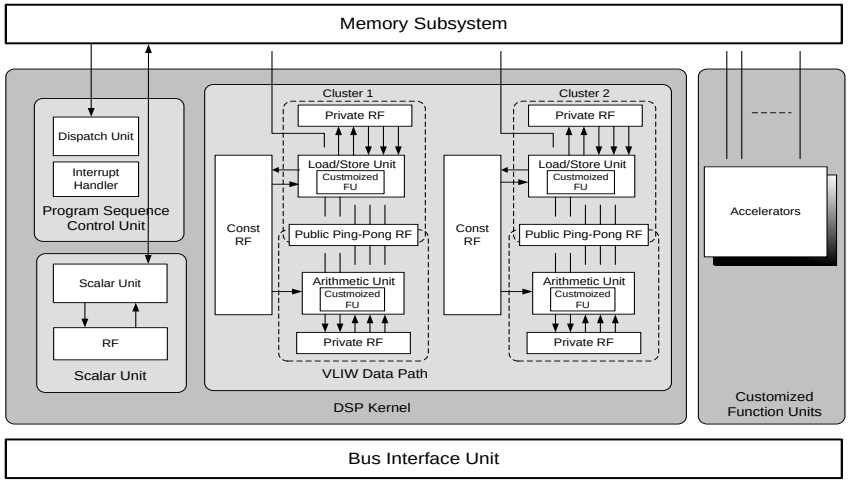
Cortex-A9 MPCore power domains and clamps*

***source: ARM, Cortex-A9 Mpcore Technical Reference Manual**

VLIW DSP with Distributed Register Files

- Distributed Register Files
- Cluster register files
- Local RF Accessible Only by Dedicated FU
- M-I Pair Access Limited by ***Ping-pong Switches***
  - **Maximal 2 Read Ports + 1 Write Port**

➔ **Low cost and low power**

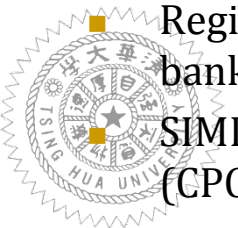


A VLIW DSP compiler for distributed register files to match the effort

- PALF scheduling policies for ILP (CPC 2006)
- GRA scheme for distributed register files (CPC 2007)
- Register spills among distributed register banks (CPC 2009)
- SIMD compiler optimizations with intrinsics (CPC 2010)

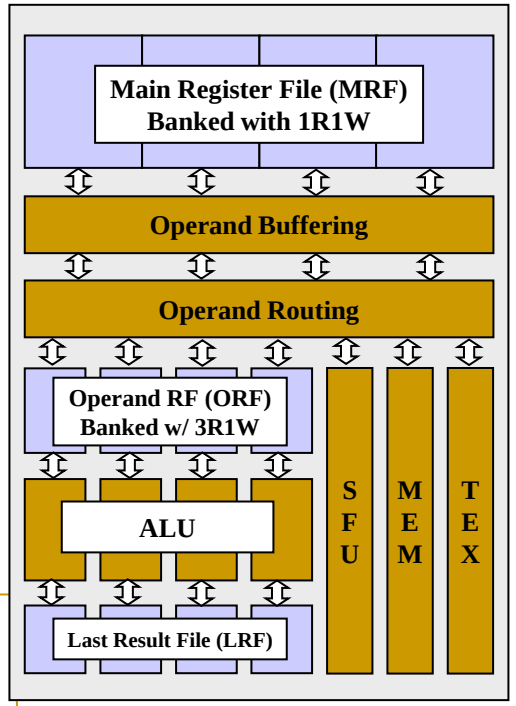
Compare with Centralized Register File

- Area : 76.8% area are saved
- Access Time : 46.9% access time are saved



Compiler Support for Low-Power with GPUs

- Register files consume 15%-20% of GPU dynamic energy – an attractive target to optimize.
- Energy-efficient register file designs:
  - Hierarchical register file (HRF [1, 2]) with compiler register allocation – 54% energy reduction on RFs
- Compiler supports are needed for a three-level register file:
  - Main register file (MRF):
    - Entries/thread for storing thread contexts
    - The biggest and the least energy-efficient
  - Operand register file (ORF):
    - Entries/thread for data with frequent reads
    - Medium size and energy-efficiency
  - Last result file (LRF):
    - Entry with thread for immediate read after write
    - The smallest and most energy-efficient
  - Re-allocate (replacement) data which were allocated to MRF to LRF or ORF whenever it is suitable.



Reference from [2]

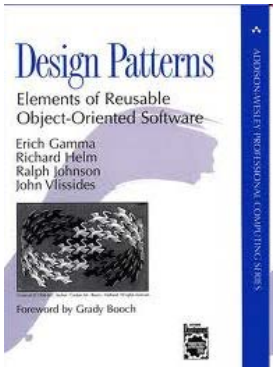
[1] Energy-efficient mechanisms for managing thread context in throughput processors. Mark Gebhart et. al, ISCA 2011.
[2] A compile-time managed multi-level register file hierarchy. Mark Gebhart et. al, MICRO 2011.

Outline

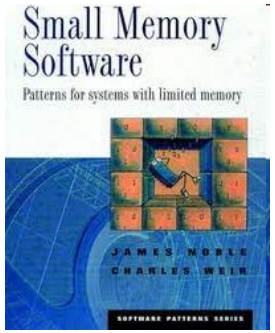
- Compilers for Low-Power
- **Pattern-Based Power Optimizations**
  - BSP Model
  - Producer-N-Consumer
  - MapReduce
  - Coefficient Objects
- Experiments
- Conclusion

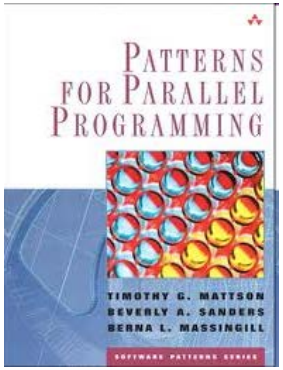
Design Patterns

- Brief introduction to design patterns
  - Proposed by C. Alexander for city planning and architecture.
  - Introduced to software engineering by Beck and Cunningham.
  - Become prominent in object-oriented programming by GoF.



- Design patterns describe “good solutions” to recurring problems in a particular context.
  - Patterns for **object-oriented programming**
    - Creational patterns, Structural patterns, Behavioral patterns, etc.
  - Patterns for **limited memory systems**
    - Compression, Small data structures, Memory allocation, etc.
  - Patterns for **parallel programming**
    - Finding concurrency, Algorithm structure, Supporting structures and Implementation mechanisms.





Parallel Design Patterns

Parallelization can be a process to transform problems to programs by selecting appropriate patterns.

Decomposition of problems

- Data or Task
- Architect Parallel Software
  - Structure Patterns
  - Computation Patters

Appropriate Algorithms

- By Tasks
  - Task parallelism
  - Divide & conquer
- By Data
  - geometric
  - Recursive

Program Constructs

- Data structures
  - Shared data
  - Shared queue
  - Shared coefficient object
  - Distributed array

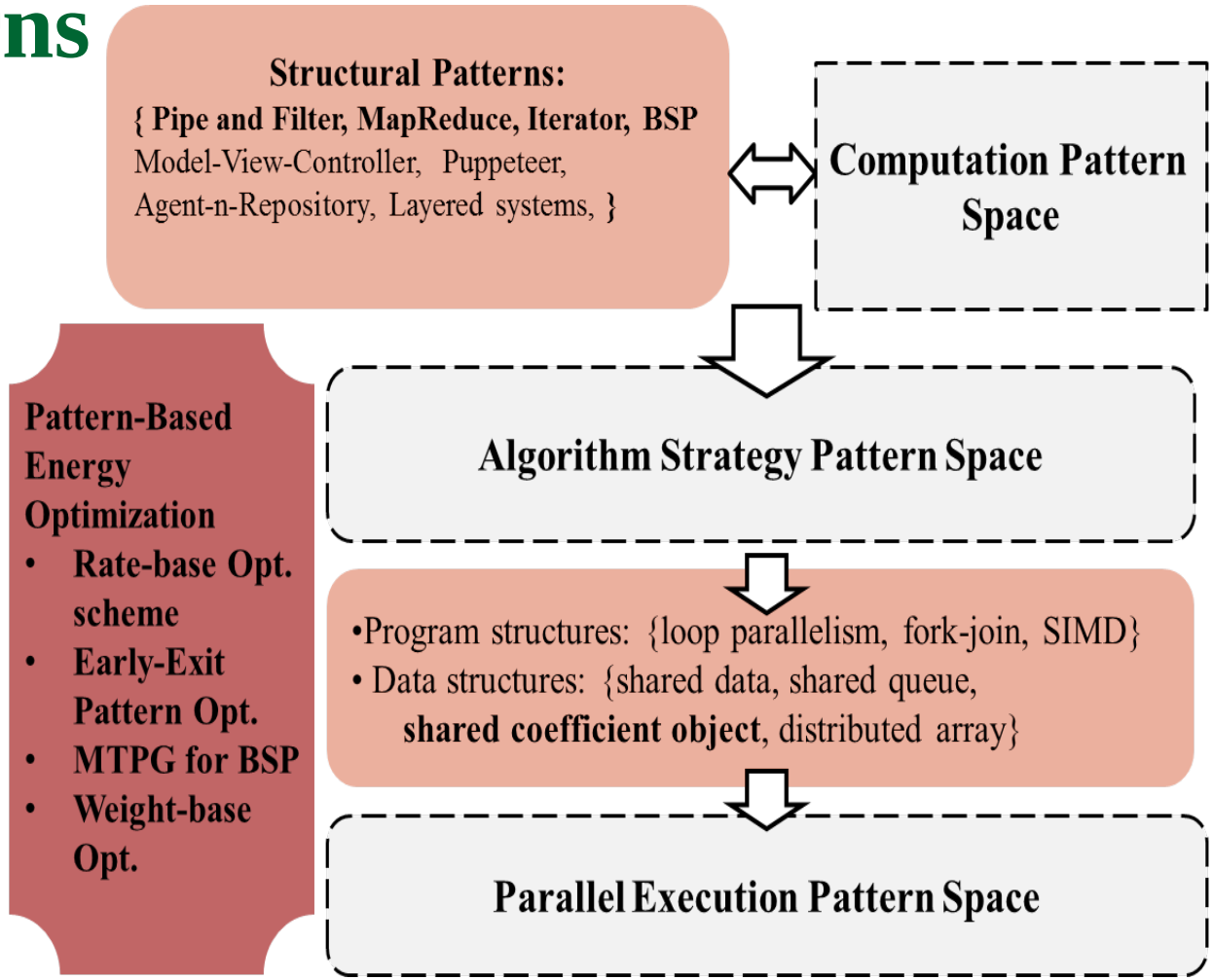
Parallelized Programs

- UE management
  - Thread/Processes
  - Work Group/Item
- Synchronization
  - barrier
  - Mutex
  - Memory fence
- Communication
  - Msg. passing
  - Collective comm.

These patterns are summarized from Our Pattern Language (OPL), Tim Mattson and the book, “Patterns for Parallel Programming” by Mattson et al

Energy Optimization with Parallel Design Patterns

- Exploit regular parallel patterns for power optimization in software layer.
- This work investigates compiler support for low power with parallel design patterns.
- Currently We are Targeting for
  - Pipe and Filter
  - MapReduce
  - Iterator
  - BSP
  - Shared Coefficient Object



These patterns are summarized from Our Pattern Language (OPL), Tim Mattson and the book, “Patterns for Parallel Programming” by Mattson et al

Compiler Directives Support for Pattern-based Power Optimization

Compiler Directives for Low Power with Parallel Patterns	
Name	**Description**
#pragma pattern BSP Powerhint	Multi-Threaded-Power-Gating(MTPG)
#pragma pattern filter filter_id	Rate-based profiling scheme
#pragma pattern map on MapReduce #pragma pattern reduce on MapReduce	Dynamic power management for early exits processor
#pragma pattern shared_coefficient_allocate #pragma pattern shared_coefficient_use #pragma pattern shared_coefficient_powerhint	Weight-based optimization scheme for shared coefficient objects
#pragma pattern puppeteer #pragma pattern puppet	Power efficient communication and dvfs for each puppet
#pragma pattern Agent-n-Depository #pragma pattern Depository on Agent-n-Depository #pragma pattern Deposit_use on Agent-n-Depository	Decentralized localization

Examples with Compiler Directives

- OSCAR API
- OpenMP
- OpenACC

```
void main() {
  /*Task Code*/
       …
  /*Sleep until someone wakes me
up*/

#pragma oscar fvcontrol
\((OSCAR_CPU( ), 0)

/*after wakeup do
synchronization
…
```

This example is from "OSCAR API for Real-Time Low-Power Multicores and Its Performance on Multicores and SMP Serves", Keiji Kimura et.al, , LCPC'09.

Re-visit with Compiler for Power-Gating

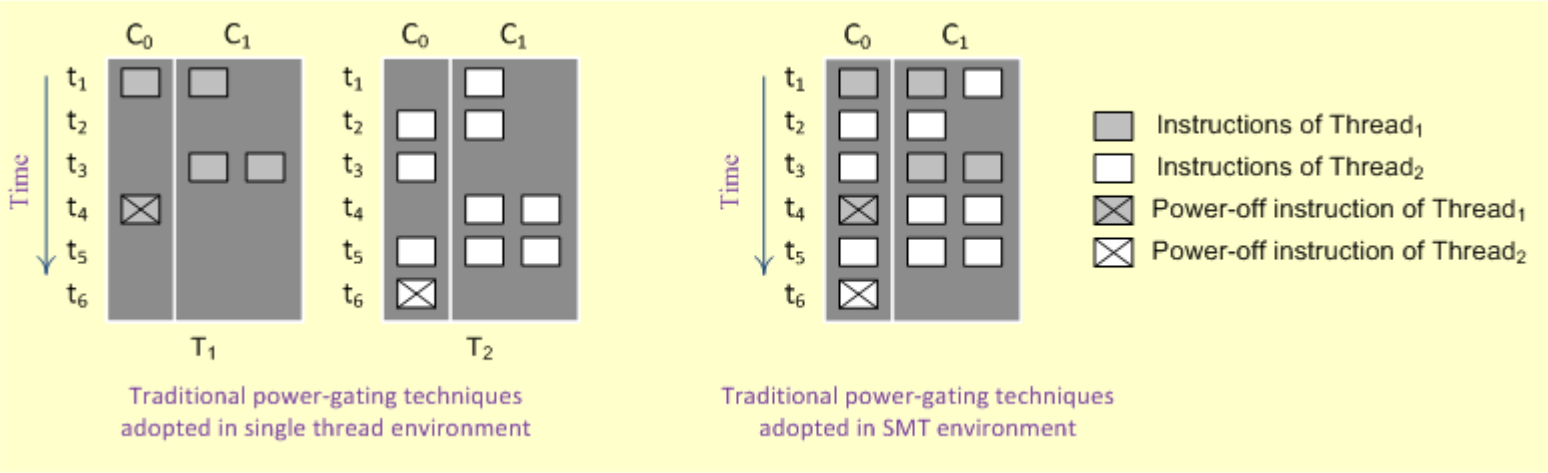
- Problem:

  In the case of multi-threading environment, original data flow analysis won't apply directly.
- It involves in the estimation of "May-Happen-In Parallel".

Assume some of parallel design patterns such as BSP is used, compilers can make the problem possible for management.

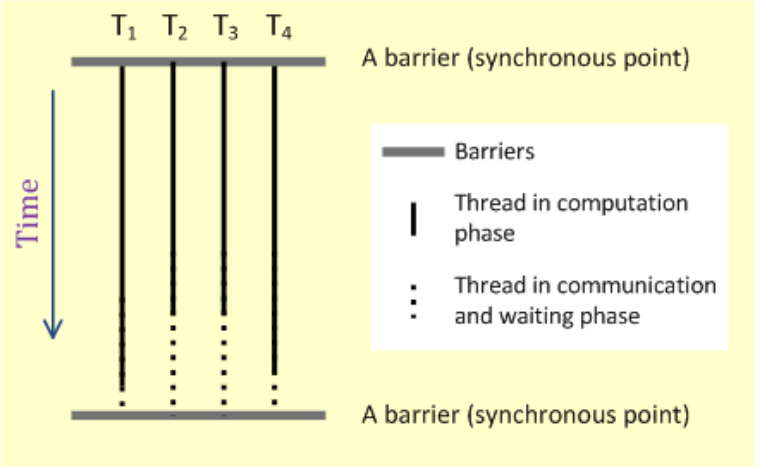
Low Power Optimization on BSP Model Motivation

- The *uncertaincy* of multithread programs is a big challenge on power-gating optimization
- On simultaneous multithreading (SMT) machines, functional units are shared by concurrent threads
- Simply applying traditional power-gating analyzing methods to multithreaded programs on SMT machines is improper:
  - Threads might mis-powered-off components still used by other threads
  - Hardware with “self power-on” mechanisms could power on components internally; however the internal power-on operation could cause processors to stall, thus resulting in more energy consumption than naïve one



Pattern: BSP Model

- Bulk-Synchronous Parallel (BSP) model♦, proposed by Valiant, is a bridging model for theory and practice of parallel computations
  - BSP model structures multiple processors with local memory and a global barrier synchronous mechanism
  - Threads processed by processors are separated by synchronous points, which forms *supersteps*
    - A *superstep* is consisted by computation phase and communication phase



- Threads in a *superstep* start at the same time and end at the same time; thus the *uncertaincy* of multithread program is reduced at the beginning and the ending of a *superstep*

♦ Valiant, L.G. 1990. **A bridging model for parallel computation**. *Commun. ACM 33*, 8, 103-111

Low Power Optimization on BSP Model Predicated-Power-Gating Operations

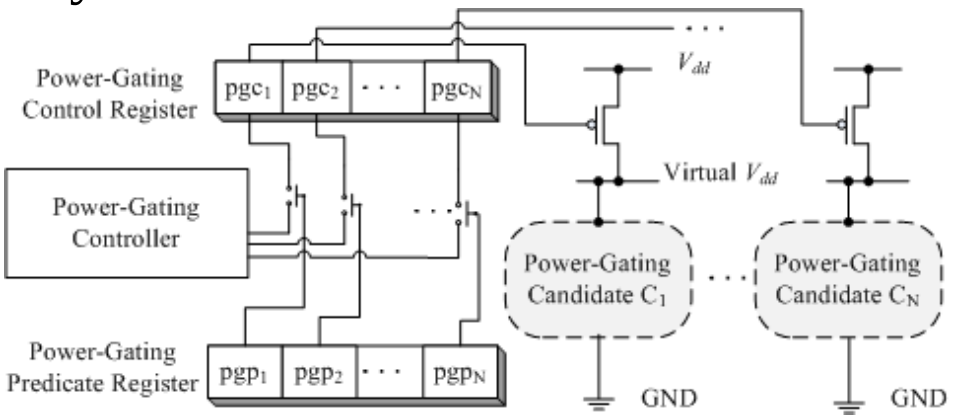
- We import the idea of conditional execution into conventional power-gating operations for solving the improper power-gating control among a set of concurrent threads
- Predicated-power-gating (PPG) operations are capable to reduce the amount of improper issued power-gating instructions
- The amount of PPG instructions could be further reduced by our MTPG analysis

Pseudo code for predicated-power-on operations

```
      lock lc1
(pgp1) power-on C1
      rc1 = rc1 + 1
      pgp1 = 0
      unlock lc1
```

Pseudo code for predicated-power-off operations

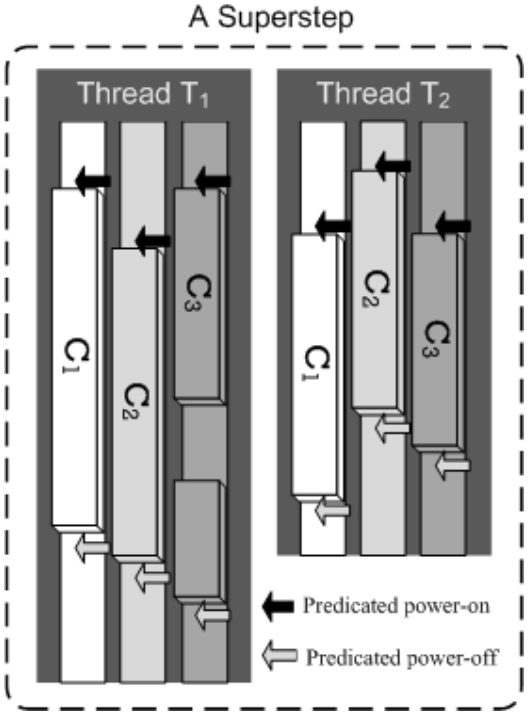
```
      lock lc1
      rc1 = rc1 - 1
      pgp1 = (rc1 == 0)
(pgp1) power-off C1
      unlock lc1
```



Yi-Ping You, Jeng Kuan Lee, Kuo Yu Chang, and Chung-Hsien Wu, **"Multi-thread power-gating control design",** US patent , NO. 7904736 B2, 2007

Low Power Optimization on BSP Model Multi-Threaded Power-Gating (MTPG)

- We design a multithreaded power-gating (MTPG) analysis for properly placing PPG instructions into BSP programs on SMT machines
- MTPG is proceeded on top of the results of CADFA[♦] with Sink-N-Hoist[Θ] and MHP analysis[Φ]
- The basic idea is to evaluate the power efficiency with dedicated power model and information from both CADFA with Sink-N-Hoist and MHP analysis



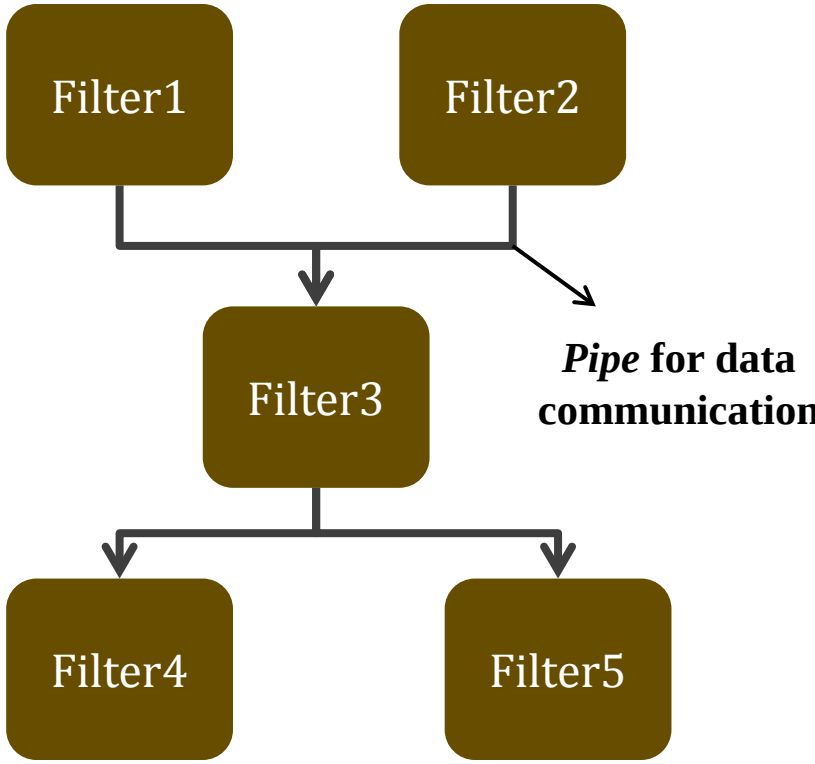
♦ Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. **Compiler analysis and supports for leakage power reduction on microprocessors.** *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, pages 63–73, Washington, D.C., USA, July 2002. Lecture Notes in Computer Science, Vol. 2481, Springer Verlag.

Θ Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. **A Sink-N-Hoist framework for leakage power reduction.** *In Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*, pages 83–94, Jersey City, New Jersey, USA, September 2005.

Φ Rajkishore Barik. **Efficient computation of may-happen-in-parallel information for concurrent Java programs.** *In Languages and Compilers for Parallel Computing (LCPC'05)*, volume 4339 of Lecture Notes in Computer Science. Springer-Verlag, 2005.

Pattern: Pipe and Filter

- The program can be decomposed into several *filters*.
- Each *filter* is a functional unit performing one or several computation tasks
- *Pipes* are used for data communication
  - Can be implemented as a shared queue, a circular buffer, or inter procedural communication (IPC)
- Concurrent execution for independent *filters*
- Examples
  - Streaming applications, Image processing



Low Power for Pipe and Filter

- Behavior between two filters is similar as *producer* and *consumer*
- Therefore processor may stall for buffer (empty or full) because of the imbalance producing rate and consuming rate
- Extra energy wasted
- Developers may try to solve the *rate equations* for balancing data computation rate
  - Figure out the relation between rate equation and processor frequency
  - Adjusting voltage & frequency for balanced rate equation

Three basic forms of pipe and filter

Rate equations

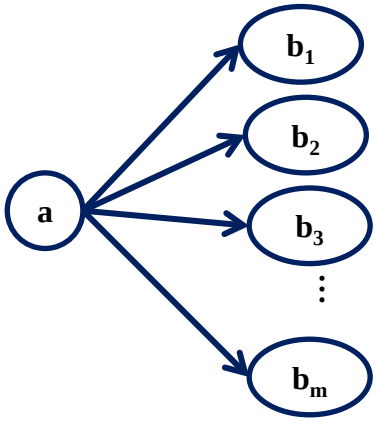
Frequency adjustment

(a). Basic Form

$\delta_a = \delta_b$

$$\frac{1}{Cyc_a/f_a} = \frac{1}{Cyc_b/f_b}$$

$$\frac{f_a}{f_b} = \frac{Cyc_a}{Cyc_b}$$

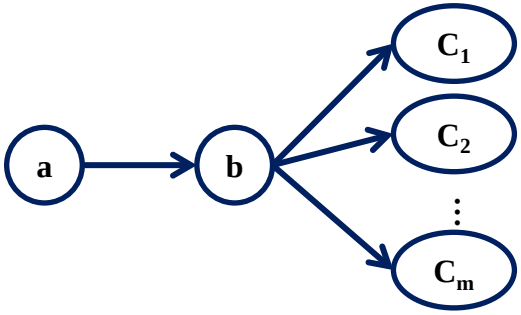


(b). One to Many

$\delta_{a1} = \delta_{b1}$
$\delta_{a2} = \delta_{b2}$
$\delta_{a3} = \delta_{b3}$
...
$\delta_{am} = \delta_{bm}$

$1/m\delta_{atotal} = \delta_{b1}$
$1/m\delta_{atotal} = \delta_{b2}$
$1/m\delta_{atotal} = \delta_{b3}$
...
$1/m\delta_{atotal} = \delta_{bm}$

Consuming rate of b



(C). Hierarchical

$\delta_a = \delta_{bc}$
$\delta_{bp1} = \delta_{c1}$
$\delta_{bp2} = \delta_{c2}$
...
$\delta_{bpm} = \delta_{cm}$

$\delta_a = \alpha\delta_{bptotal}$
$1/m\delta_{bptotal} = \delta_{b2}$
$1/m\delta_{bptotal} = \delta_{b3}$
...
$1/m\delta_{bptotal} = \delta_{bm}$

Producing rate of b_m to C_m

Rate-based profiling scheme for power optimization

- Compiler *pragma* support for ***pipe and filter***
- Rate-based profiling scheme to figure out proper *voltage & frequency* of each filter processor

Input:
1. $\mathcal{A}$: A multicore application with pipe and filter pattern that each filter is already mapping to each processor.
Input: 2. T_f : Fitting table which contains the frequency configurations of each processor.
Data: $G=(V,E)$: Connection graph
Data: R : Rate equations
Data: δ : Producing rate or consuming rate of each filter

```
1 G ← build_connection_graph(A);
2 foreach edge e_i ∈ G do
3 |   R_i ← build_rate_equation(e_i);
4 end
5 foreach vertex v_i ∈ G do
6 |   δ_{v_i} ← simulation_profiling(A);
7 end
8 frequency_adjustment(R_i, δ_{v_i}, T_f);
```

Code transformation

(a).Code Skeleton of Pipe and Filter

```
#pragma pattern pipe-n-filter f_id(a)
filter_a() {
  while(true) {
    /*User defined producing process*/
    producing_function(&DATA);
    /*Stall when the buffer is full*/
    put_data(f_id(b), &DATA, SIZE);
  }
}
```

```
#pragma pattern pipe-n-filter f_id(b)
filter_b() {
  while(true) {
    /*Stall when the buffer is empty*/
    get_data(f_id(a), &DATA, SIZE);
    /*User defined consuming process*/
    consuming_function(&DATA);
  }
}
```

(b). Code transformation for profiling instrumentation

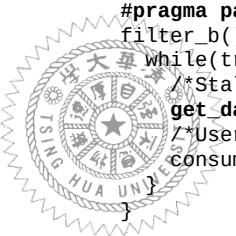
```
__inspector(f_id(a),START);
filter_a() {
  while(true) {
    /*User defined producing process*/
    producing_function(&DATA);
    /*Stall when the buffer is full*/
    put_data(f_id(b), &DATA, SIZE);
  }
}
__inspector(f_id(a), END)
```

```
__inspector(f_id(b), START);
filter_b() {
  while(true) {
    /*Stall when the buffer is empty*/
    get_data(f_id(a), &DATA, SIZE);
    /*User defined consuming process*/
    consuming_function(&DATA);
  }
}
__inspector(f_id(b), END)
```

(c). Frequency adjustment after profiling

```
__frequency_adjustment(f_id(a));
filter_a() {
  while(true) {
    /*User defined producing process*/
    producing_function(&DATA);
    /*Stall when the buffer is full
    put_data(f_id(b),&DATA, SIZE);
  }
}
```

```
__frequency_adjustment(f_id(b));
filter_b() {
  while(true) {
    /*Stall when the buffer is empty*/
    get_data(f_id(a), &DATA, SIZE);
    /*User defined consuming process*/
    consuming_function(&DATA);
  }
}
```



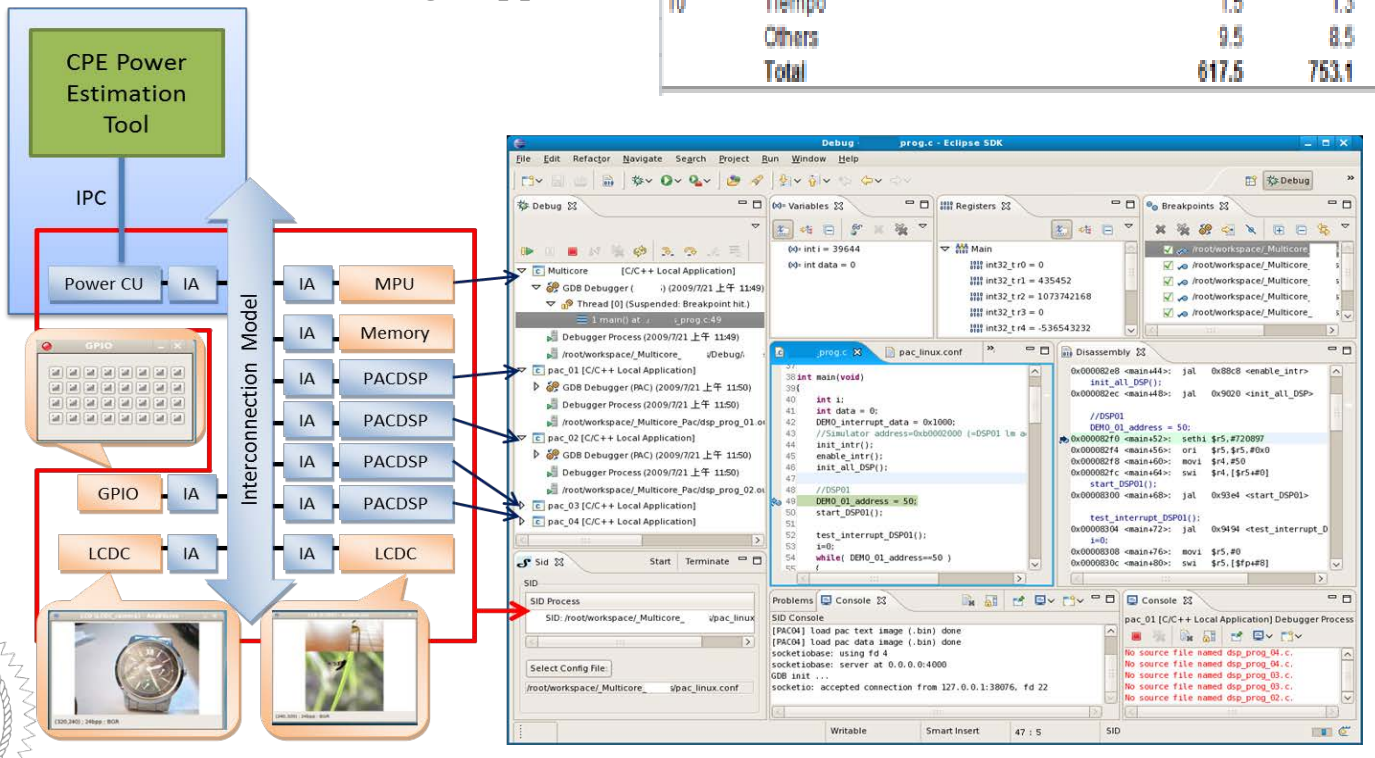
Evaluation Environment

- SID-Based Multicore Power Simulator
  - Configurable Heterogeneous Multicore Environment
    - Andes™ Core N1213 as MPU
    - A number of PAC-DSPs (from ITRI)
    - Other peripherals
  - Power Modeling Tool is based on PowerMixer^IP
  - Hierarchal Power Profiling Support

Semiconductor Design IP Revenue, Microprocessors, Worldwide, 2010 and 2011 (Millions of Dollars)

Rank	Company	2010	2011	Growth	2011 Share	Cumulative Share
1	ARM Holdings	490.7	634.8	29.4%	84.3%	84.3%
2	MIPS Technologies	85.3	72.1	-15.5%	9.6%	93.9%
3	Tensilica	11.9	13.6	14.2%	1.8%	95.7%
4	Synopsys	8.1	9.0	11.1%	1.2%	96.9%
5	Andes Technology	3.3	4.5	37.5%	0.6%	97.4%
6	Cortus	3.8	4.5	17.1%	0.6%	98.0%
7	Cast	1.7	2.1	20.8%	0.3%	98.3%
8	Western Design Center	1.7	1.6	-10.0%	0.2%	98.5%
9	Netdeus	0.0	1.3	NA	0.2%	98.7%
10	Tiempo	1.5	1.3	-13.3%	0.2%	98.9%
	Others	9.5	8.5	-9.9%	1.1%	100.0%
	Total	617.5	753.1	22.0%	100.0%	100.0%

From EE Times

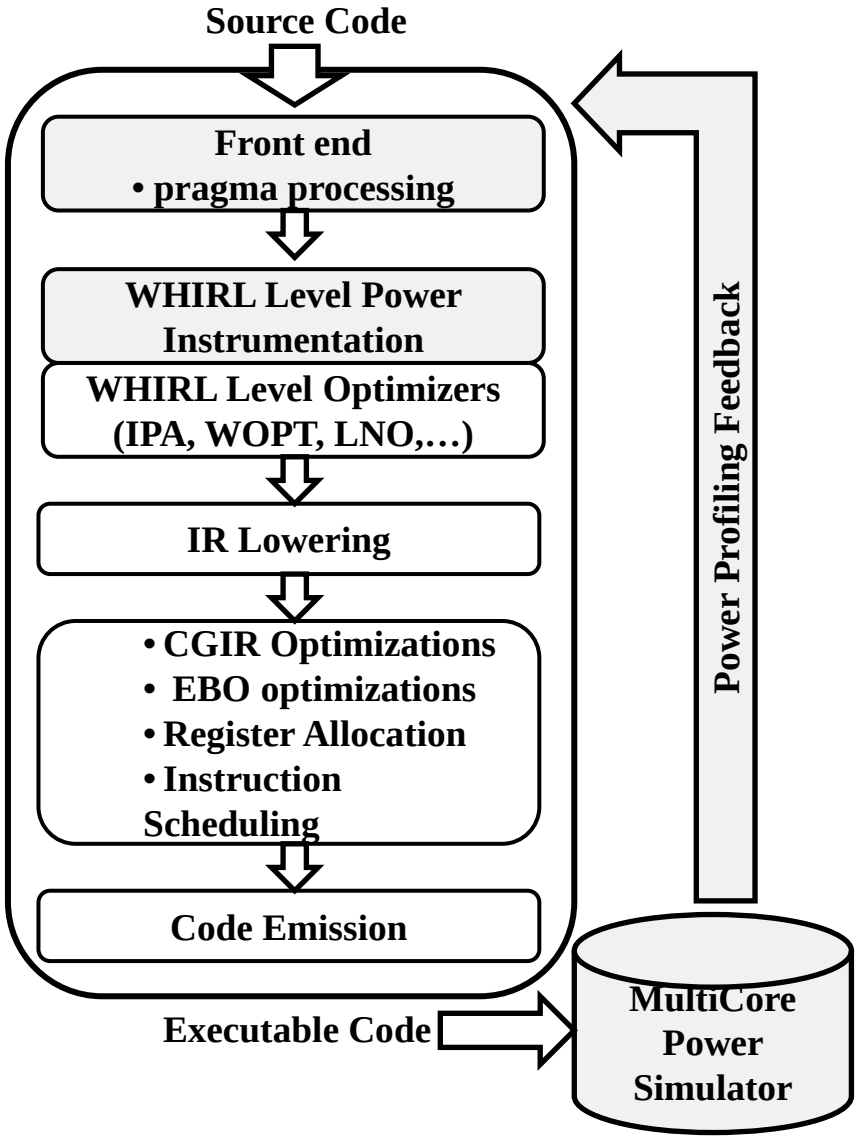


*** Power Aware SID-based Simulator for Embedded Multicore DSP Subsystems, Lin et. al, CODES+ISSS'10.**

The Compilation and Simulation Flow

- Open64 based VLIW DSP compiler
  - Optimizations for distributed register architecture
  - *Pragma* support for pattern-based power optimizations
  - WHIRL-level Power Instrumentation

Compiler Directives for Power Profiling Code Sections	
Name	**Description**
#pragma power_profiling function_name	Power profiling a specific function.
#pragma power_profiling for	Power profiling a specific for loop.
#pragma power_profiling while	Power profiling a specific for a while loop.
#pragma power_profiling section	Power profiling a user defined code section between { }.
Compiler Directives for Low Power with Parallel Patterns	
#pragma pattern filter filter_id()	
#pragma pattern map on MapReduce #pragma pattern reduce on MapReduce	

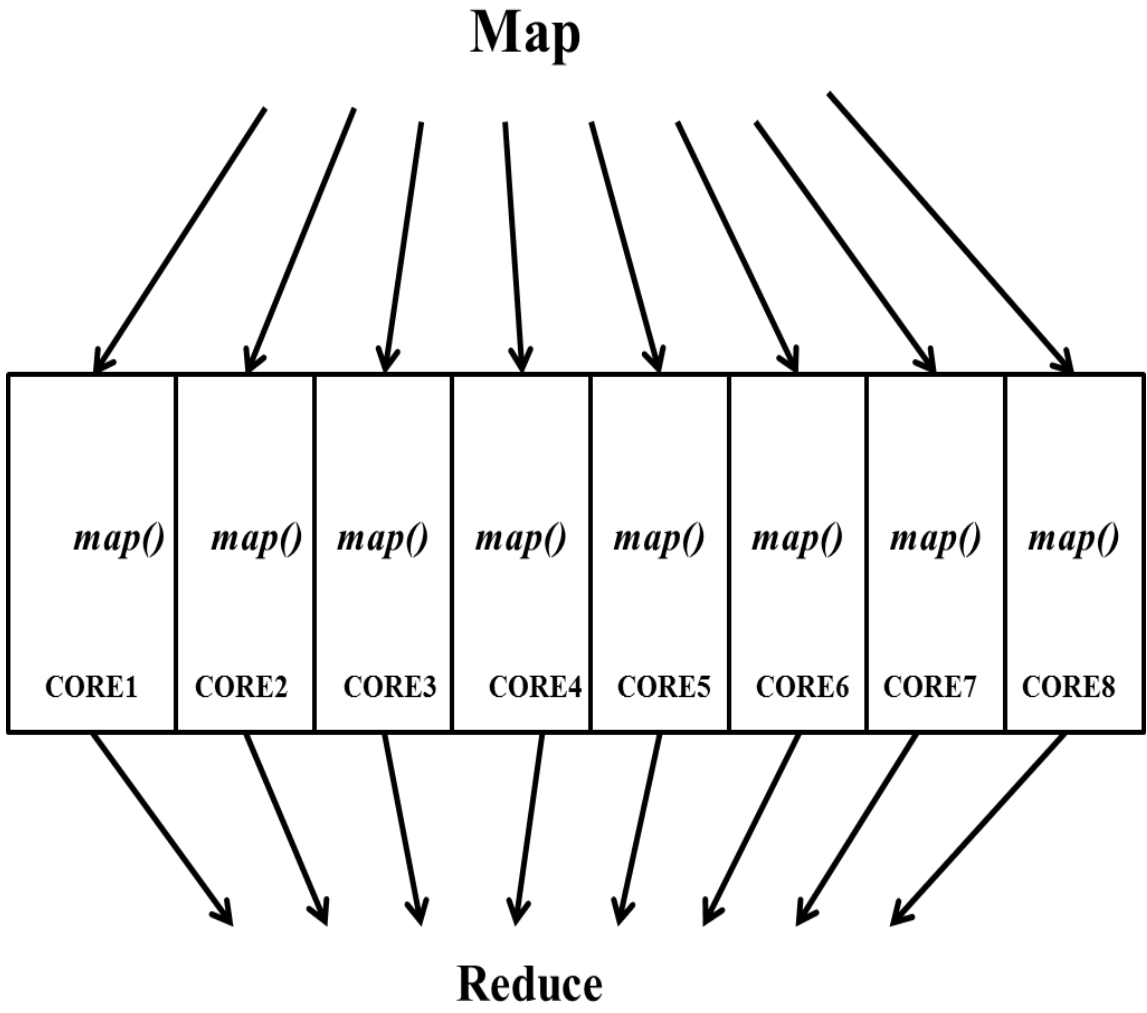


Related Work: Programming Model Supports

- OpenStream
  - A Stream programming model for OpenMP
  - Decoupled, producer/consumer task-parallel pipelines
  - *OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs, Albert Cohen et al., TACO' 13
- WeakRB (Weak consistency Ring Buffer)
  - An improved Single-producer, single-consumer (SPSC) FIFO with a portable C11 implementation
  - "Bringing Together FIFO Queues and Dynamic Scheduling for the Correct and Efficient Execution of Task-Parallel, Data Flow Programs", Alber Cohen et al., CPC'13

Pattern: MapReduce

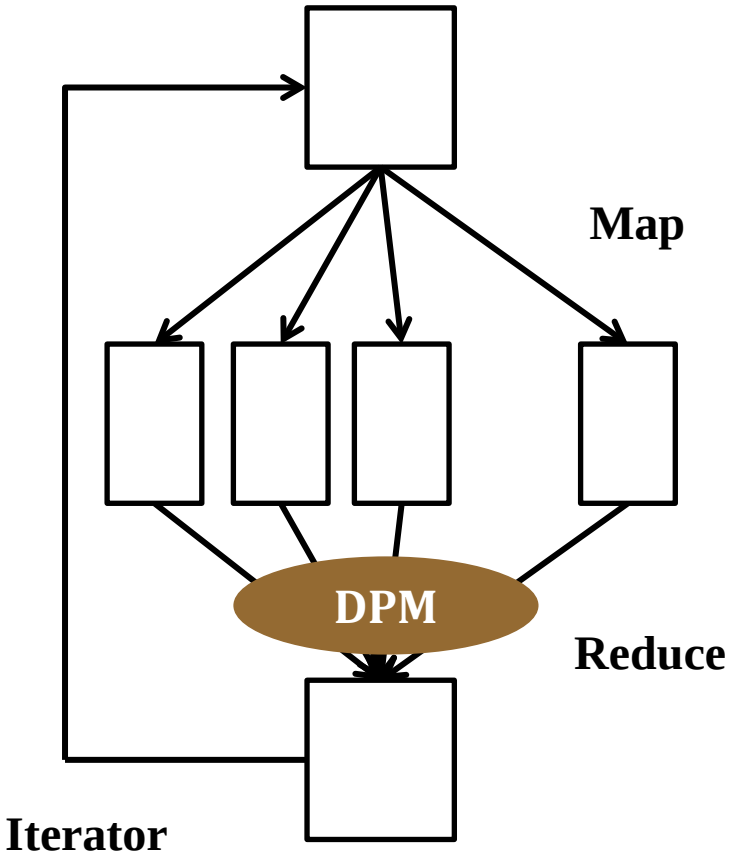
- Also used in cloud computing for data intensive task on distributed large scale systems*
- Decomposes task into two phases
  - Map
    - User defined *map* function for independent computation task
  - Reduce
    - User defined *reduce* function collecting and summarizing the results from map function



*MapReduce: simplified data processing on large clusters, Jeffer Dean, etc., In Proceedings of the 6th conference on Symposium on Operating System Design and Implementation, OSDI'04, 2004

Low Power for MapReduce with Iterator

- Some early returned processors may spend extra power for waiting the next iteration
- Dynamic Power Management (DPM) for such processors
  - Saving power of the idling processors



Power Management Scheme for MapReduce with Iterator

- Early exit optimization
- Compiler *pragma* support for MapReduce with Iterator
- MapReduce runtime with dynamic power management
  - Running mode configuration for early returned processors

Input:
1. P_i | i=1,...n., each processor P_i will execute a *map* function, respectively.
2. T_i | =i=1,...,n., the waiting time of the early exists processor P_i at reduction stage.
3. T_{ave} | The average waiting time of each processor at reduction stage.
3. $P_{(c,i)}$ | Power consumption of processor P_i at original running mode.
4. $P_{(n,i)}$ | Power consumption of processor P_i after configured to low power running mode.
5. E_o | Energy overhead for running mode configuration.

```
1 foreach P_i that early exits from the map function do
2     T' = T_ave - T_i;
3     if (P_(n,i) * T'+E_o) < P_(c,i) * T' then
4         set_running_mode (P_i);
5     end
6 end
7 void set_power_mode ( Processor )
8 {set Processor into low power running mode according to the hardware
  specification; }
```

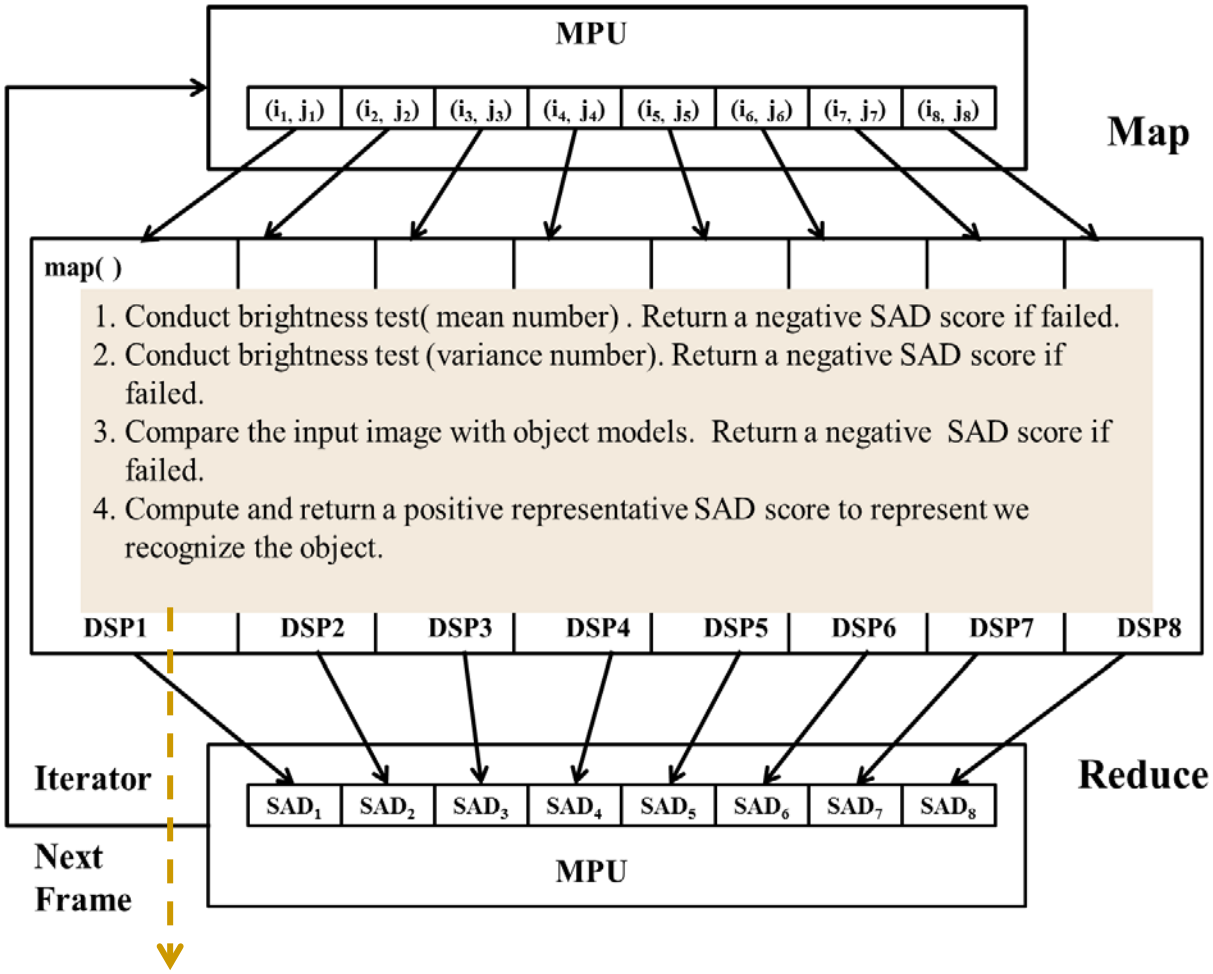
Code skeleton

```
#pragma pattern map on MapReduce
map(intermediate_key, input_value) {
        /*User defined steps
          to compute the intermediate results
          from input value*/
        …
}
```

```
#pragma pattern reduce on MapReduce
reduce(intermediate_key, intermediate_result) {
        /*User defined steps
         to summarize the intermediate results
         from each map function*/
             …
}
```

Preliminary Results: Object Detection

- Multicore object detection application
  - Detecting the target object from the input video
  - Parallelized by MapReduce with Iterator pattern
    - Each ***map*** function is mapping on each PAC-DSP
    - Iterative execution until finishing all frames



Every *map* function will go through four conditions to calculate the sum of *absolute differences (SAD)* score in order to determine if the target object exists in the input scope.

Pattern: Shared Coefficient Object

- Shared Co-efficient Object
  - First initialized in the external shared memory
  - Accessed by the parallel tasks simultaneously
  - Frequently used in embedded multicore applications
    - Image Recognition Applications
    - Voice Recognition Applications

(a). Multicore RMS: Code fragment at MPU site

```
//Shared coefficient objects
#pragma pattern shared_coefficient_allocate
Face_MODEL facemodel;
Leye_MODEL leyemodel;
Reye_MODEL reyemodel;

int SlideWinSearching(...)
{
   //Data initiation
   face_model_init(facemodel);
   leye_model_init(leyemodel);
   reye_model_init(reyemodel);
   start_DSP(); /*Perform RMS with 8 DSPs*/
}
```

(b). Multicore RMS: Code fragment at DSP site

```
#pragma pattern shared_coefficient_use
Face_MODEL *facemodel;
Leye_MODEL *leyemodel;
Reye_MODEL *reyemodel;

int CheckSlideWindow(...)
{
  /*A loop with shared coefficient object access*/
  #pragma pattern shared_coefficient_powerhint
  for(i=0; i<e_num; i++)
    fOr (j=0; j < p_num; j++){
      model[i] += facemodel->EigenVec[i*p_num+j] *
            (window[j] – facemodel->Mean[j]);
    }
}
```

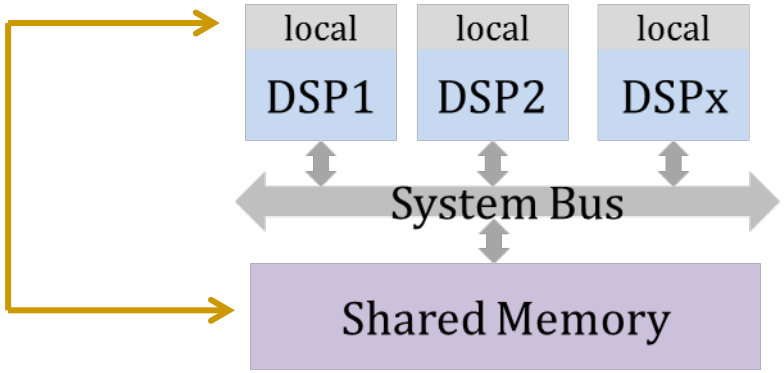
Running Example: A multicore RMS application with shared coefficient object

Power Optimization for Shared Coefficient Object

- Power Optimization with Data Localization
  - Make good use of local memory of each processor
  - Reduce External Memory Accessing
  - Weight-based algorithm

Input: 1. $Coeff_i | i=1,...,n$; 2. *Aval_Local_Size*.
Data: 1. *candidate_list*; 2. $Cand_p$—$p=1,...,n$.
Output: Assignment of coefficient object to local memory.

```
1  foreach Coeff_i do
2  |   candidate_list ← (access_counts_calculation(Coeff_i), Coeff_i);
3  end
4  Sort candidate_list in decreasing order according to the access counts of each
   coefficient object Cand_p, that (Cand_p—p=1,...,n)∈ candidate_list, ;
5  while Aval_local_Size is not empty and Aval_local_Size ≥ size(Cand_min) do
6  |   for p ← 1 to n do
7  |   |   if Aval_local_Size > sizeof(Cand_p) then
8  |   |   |   Assign Cand_p to local memory;
9  |   |   |   p++;
10 |   |   |   Aval_local_Size ← Aval_local_Size - size(Cand_p);
11 |   |   else
12 |   |   |   p++;
13 |   |   end
14 |   end
15 end
```



Compiler Directives Support for Pattern-based Power Optimization

Compiler Directives for Low Power with Parallel Patterns	
Name	**Description**
#pragma pattern BSP Powerhint	Multi-Threaded-Power-Gating(MTPG)
#pragma pattern filter filter_id	Rate-based profiling scheme
#pragma pattern map on MapReduce #pragma pattern reduce on MapReduce	Dynamic power management for early exits processor
#pragma pattern shared_coefficient_allocate #pragma pattern shared_coefficient_use #pragma pattern shared_coefficient_powerhint	Weight-based optimization scheme for shared coefficient objects
#pragma pattern puppeteer #pragma pattern puppet	Power efficient communication and dvfs for each puppet
#pragma pattern Agent-n-Depository #pragma pattern Depository on Agent-n-Depository #pragma pattern Deposit_use on Agent-n-Depository	Decentralized localization

Summary

- We present compiler techniques for low power.
- Pattern-based energy optimization method is presented
  - Pipe and Filter
  - MapReduce + Iterator
  - BSP
  - Shared Coefficient Object
- Significant power reduction is observed from preliminary results
- Power optimizations with parallel patterns can be an important direction for power optimization in the software layer.
- Related references can be seen in http://www.cs.nthu.edu.tw/~jklee