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

Source: Massoud Pedram, USC
P 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

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

### Leakage Power Reduction Framework

1. ICFG construction
2. Component-Activity Data-Flow Analysis
3. Power-gating instruction scheduling
4. Sink-N-Hoist Analysis
5. Power-gating instruction generation

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

*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.

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.

Finding Concurrency

- Decomposition of problems
  - Data or Task
  - Architect Parallel Software
    - Structure Patterns
    - Computation Patterns

Algorithm Structure

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

Supporting Structures

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

Implementation Mechanisms

- 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

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

- OSCAR API
- OpenMP
- OpenACC

```c
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.
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 *uncertainty* 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 *uncertainty* of multithread program is reduced at the beginning and the ending of a *superstep*.

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.

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 and MHP analysis.
- 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.


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.

---

**Rate equations**

\[
\delta_a = \delta_b \\
\delta_{a1} = \delta_{b1} \\
\delta_{a2} = \delta_{b2} \\
\delta_{a3} = \delta_{b3} \\
\vdots \\
\delta_{am} = \delta_{bm}
\]

- Frequency adjustment
  \[
  \frac{1}{\text{Cyc}_a / f_a} = \frac{1}{\text{Cyc}_b / f_b} \\
  f_a = \frac{\text{Cyc}_a}{\text{Cyc}_b}
\]

**Consuming rate of b**

\[
\delta_a = \alpha \delta_{bptotal} \\
\delta_{bp1} = \delta_{c1} \\
\delta_{bp2} = \delta_{c2} \\
\vdots \\
\delta_{bpm} = \delta_{cm} \\
\]

- 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

**Code transformation**

(a). Code Skeleton of Pipe and Filter

```c
#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);
    }
}
```

(b). Code transformation for profiling instrumentation

```c
__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);
    }
}
```

(c). Frequency adjustment after profiling

```c
__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);
    }
}
```

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

**Code transformation**

(a). Code Skeleton of Pipe and Filter

```c
#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);
    }
}
```

(b). Code transformation for profiling instrumentation

```c
__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);
    }
}
```

(c). Frequency adjustment after profiling

```c
__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);
    }
}
```

---

**Input:**
- $A$: A multicore application with pipe and filter pattern that each filter is already mapping to each processor.
- $T_f$: Fitting table which contains the frequency configurations of each processor.

**Data:**
- $G = (V, E)$: Connection graph
- $R$: Rate equations
- $\delta$: Producing rate or consuming rate of each filter

```
1) $G \leftarrow$ build_connection_graph($A$);
2) $\forall e_i \in G$ do
3) \hspace{1em} $R_i \leftarrow$ build_rate_equation($e_i$);
4) $\forall v_i \in G$ do
5) \hspace{1em} $\delta_{v_i} \leftarrow$ simulation_profiling($A$);
6) $\forall f_id(a)$ do
7) \hspace{1em} frequency_adjustment($R_i, \delta_{v_i}, T_f$);
```
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™
  - Hierarchal Power Profiling Support

* 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 {}. |

<table>
<thead>
<tr>
<th>Compiler Directives for Low Power with Parallel Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma pattern filter filter_id()</td>
</tr>
<tr>
<td>#pragma pattern map on MapReduce</td>
</tr>
<tr>
<td>#pragma pattern reduce on MapReduce</td>
</tr>
</tbody>
</table>

Diagram:

- Source Code
  - Front end
    - #pragma processing
  - WHIRL Level Power Instrumentation
    - WHIRL Level Optimizers (IPA, WOPT, LNO,...)
  - IR Lowering
    - CGIR Optimizations
    - EBO optimizations
    - Register Allocation
    - Instruction Scheduling
  - Code Emission
  - Executable Code
  - MultiCore Power Simulator
  - Power Profiling Feedback
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

Code skeleton

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

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

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

```
foreach $P_i$ that early exits from the map function do
    $T' = T_{ave} - T_i$;
    if ($P(n,i) * T' + E_o < P(c,i) * T'$ then
        set_running_mode($P_i$);
    end
end
void set_power_mode ( Processor )
{ set Processor into low power running mode according to the hardware specification; }```

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 Co-efficient 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

//Shared co-efficient objects
#pragma pattern shared_coefficient_allocate
Face_MODELL facemodel;
Leye_MODELL leyemodel;
Reye_MODELL reyemodel;

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

#pragma pattern shared_coefficient_use
Face_MODELL *facemodel;
Leye_MODELL *leyemodel;
Reye_MODELL *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]);
        }
}

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

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 > size(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
```

---

**Diagram:**

```
local

DSP1

down

System Bus

down

DSP2

down

local

DSPx

down

local

Shared Memory
```
## Compiler Directives Support for Pattern-based Power Optimization

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma pattern BSP Powerhint</td>
<td>Multi-Threaded-Power-Gating(MTPG)</td>
</tr>
<tr>
<td>#pragma pattern filter filter_id</td>
<td>Rate-based profiling scheme</td>
</tr>
<tr>
<td>#pragma pattern map on MapReduce</td>
<td>Dynamic power management for early exits processor</td>
</tr>
<tr>
<td>#pragma pattern reduce on MapReduce</td>
<td>Dynamic power management for early exits processor</td>
</tr>
<tr>
<td>#pragma pattern shared_coefficient_allocate</td>
<td>Weight-based optimization scheme for shared coefficient objects</td>
</tr>
<tr>
<td>#pragma pattern shared_coefficient_use</td>
<td>Weight-based optimization scheme for shared coefficient objects</td>
</tr>
<tr>
<td>#pragma pattern shared_coefficient_powerhint</td>
<td>Weight-based optimization scheme for shared coefficient objects</td>
</tr>
<tr>
<td>#pragma pattern puppeteer</td>
<td>Power efficient communication and dvfs for each puppet</td>
</tr>
<tr>
<td>#pragma pattern puppet</td>
<td>Power efficient communication and dvfs for each puppet</td>
</tr>
<tr>
<td>#pragma pattern Agent-n-Depository</td>
<td>Decentralized localization</td>
</tr>
<tr>
<td>#pragma pattern Depository on Agent-n-Depository</td>
<td>Decentralized localization</td>
</tr>
<tr>
<td>#pragma pattern Deposit use on Agent-n-Depository</td>
<td>Decentralized localization</td>
</tr>
</tbody>
</table>

### Compiler Directives for Low Power with Parallel Patterns
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