Performance and Power Aware C Code Parallelization for Embedded Devices

Rainer Leupers
Chief Scientist
Silexica Software Solutions GmbH
MPSoCs and the Productivity Gap

**Demand for Complex Applications**

- Mobile HD videos
- Augmented reality
- Drive-by-Wire
- Human-machine Interaction

**Facts**

- Complex applications require multicore processors to run smoothly

**Multicores Becoming More Complex**

- Off-the-shelf platforms
  - 8 Cores + GPU
  - 4 Cores + 3 DSPs + GPU + 3 HW accelerators
  - 4 Cores + 8 DSPs

**Facts**

- No automated programming solution available in the market

→ Need better support for SW development in the MPSoC era
Key problem in MPSoC programming today

Lack of “the compiler” in the programming flow for MPSoCs leads to huge productivity loss!
SLX Multicore Toolsuite Overview

- **SLX Parallelizer**
  - Expose multiple forms of parallelism

- **SLX Mapper**
  - Define an optimized software distribution

- **SLX Generator**
  - Generate code for the target platform
• Information to parallelize a sequential application?
  • Dependencies in a Control and Data Flow Graph
    • Hybrid approach: static and dynamic data flow analysis
    • Performance information

• Partitioning process
  • Clustering heuristics at the C statement granularity
  • Search for different parallelism patterns
  • Understand cost of computation and communication
Supported Parallelism Types

Task Level Parallelism (TLP)

Data Level Parallelism (DLP)

Pipeline Level Parallelism (PLP)
SLX Parallelizer Flow

- Parsing, tracing, performance estimation, dynamic data flow analysis
- Graph clustering and pattern-based parallelism extraction
- Feedback to the programmer
Displaying Results

• Compiler graphs are difficult to handle for the programmer

• Therefore intuitive reports are presented to the user

✓ Source level annotations

✓ Source code highlighting
SLX Parallelizer – Code Generation

- Today: Identify parallelism automatically and convert to a shared memory API specification, e.g. OpenMP

- Future: Automatically convert to a Process Network specification
SLX Mapper

• How to distribute an application based on different optimization criteria and additional user constraints to a given multicore architecture?

• Challenges:
  ✓ Application specification
  ✓ Bottleneck identification
  ✓ Accurate performance estimation
  ✓ Simultaneous mapping of computations and communications
  ✓ Huge search space
SLX Mapper- Programming Model

- Many parallel programming models, but no winner

- Thoughts on programming models
  - Leveraged by compiler for **code generation** (correctness)
  - Formal properties for **optimizations**

- Practical considerations
  - **C dominance** is unchallenged
  - Domain-specific (wireless, multimedia): **Dataflow programming / Streaming applications for signal processing, multimedia application and more**
Parallel Specification: CPN

- CPN: C for Process Networks
  - *Intuitive parallel programming*

- Based on C
  - Processes in C
  - Channels support C types, structs, typedefs, …

```c
__PNkpn Amp __PNin(short A[2]) __PNout(short B[2]) __PNparam(int boost)
{
    while (1)
    {
        __PNin(A) __PNout(B) {
            for (int i = 0; i < 2; i++)
                B[i] = A[i]*boost;
        }
    }
}
__PNprocess AudioAmp1 = Amp __PNin(C) __PNout(F) __PNparam(3);
__PNprocess AudioAmp2 = Amp __PNin(D) __PNout(G) __PNparam(10);
```
SLX Mapper: Post-Mapping Analysis

- Processor Utilization
- Task States
- Schedule
SLX Generator

- Target specific code needs to be written manually

- Challenges:
  - Code generation for heterogeneous OS with different APIs
  - Source code is mapping-dependent
  - Cross-compilation for a huge variety of target MPSoCs
Smartphone case study

- Exynos Octa 5 multicore SoC
- Performance improvement of image codecs via parallelizing C compiler
- Side-effect: energy savings
- Verified via physical measurements

![Graphs showing performance improvement for encoder and decoder](image-url)
Towards system-level power optimization
Motivation

Impact of design decisions on power consumption

Level of design detail

- Electronic System Level (ESL)
- Register Transfer Level (RTL)
- Gate Level
- Layout Level
- Chip

Timing: ✔️

Power: ✗ → ✔️

- High simulation speed
- Early availability of power estimates
Power estimation methodology

ESL system containing component

- gate-level netlist
- layout
- hardware chip
- automated back-annotation

ESL power estimation in different scenario

ESL power estimation in different system

power trace of component
Power model calibration

- Run hardware and ESL model with same inputs
  - will be in same state
- Record ESL traces and hardware power as calibration data
- Assume linear power model
  - Determine factors $a$ with best fit

\[
P_{est} = S \cdot a
\]

\[
a = (a_1 \ldots a_N)^T
\]

\[
P_{est} \approx P_{ref}
\]
NoC case study

- Core: blackbox
- L1 Mem: placement blockage
- L2 Mem: blackbox
- L3 Mem: blackbox

Master subsystems

Pins of subsystems assigned close to NoC.
Speed and accuracy

- ESL power estimation methodology verified for AMBA AXI and complex custom NoC at VLSI layout level (post P&R)
- Estimation error < 22%
- Can predict different „power phases“ correctly
- 900x faster than low-level power simulation
ESL power estimation for clustered multicore platform

- Power estimation for entire SoC
- Focus on ARM A9 core power estimation
  - White box approach using instrumented gem5 simulator
  - Black box approach using OVP simulator TLM and activity traces
TI Panda Board ES

- OMAP4460
  - 2x ARM Cortex A9
  - 2x ARM Cortex M3
  - DSP
  - GPU
  - ...
- SW loaded via bootleader from SD card
  - bare-metal benchmarks possible
Power Measurement Setup

- **Standard Output**: via serial port
- **Test Automation Board**: control power via serial port
- **Benchmark Upload**: via Ethernet
- **Benchmark Time Synchronization**: via GPIO pin, sampled by power measurement circuit
- **Amplifier Circuit**: for power measurement
- **Data Logger**: USB-DUXfast
  - 16 channels
  - 12 bit
  - 5 kHz
White box estimation results

![Graph showing single core RMS power estimation error (%)]

White box estimation results

**Single Core RMS Power Estimation Error (%)**

**RMS**: root mean square

**Averages**: 11.1% 5.4%

**Formulas**:

- \( err^{RMS} = \sqrt{\frac{1}{T} \int_0^T (P_{est} - P_{ref})^2 dt} \)
- \( err^{RMS}_{avg} = \frac{\sum_{n=1}^{N} err^{RMS}_n}{N} \)
Black box estimation results

Dual Core Black Box OVP Power Estimation Error (%)

- Bare
- Computed
- Activity

Averages: 4.9%, 5.3%, 5.5%
Platform level power estimation

abstract power model:

\[ P_{bus} = b \times \text{<in>} \times \text{<out>} \times \text{<width>} \times (s + d \times \text{<freq>} \times \text{<usage>}) \]

abstract power model:

\[ P_{mem} = m \times \text{<capacity>} \times (s + d \times \text{<freq>} \times 5\% \times \text{<usage>}) \]
## Optimized SW task mapping (16 cores)

### Audio Filter

<table>
<thead>
<tr>
<th></th>
<th>Average Power</th>
<th>Peak Power</th>
<th>Time</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Mapping</td>
<td>2.52 W</td>
<td>4.82 W</td>
<td>24 us</td>
<td>61.6 uJ</td>
</tr>
<tr>
<td>One core</td>
<td>0.62 W</td>
<td>0.62 W</td>
<td>92 us</td>
<td>56.8 uJ</td>
</tr>
</tbody>
</table>

### LTE Benchmark

<table>
<thead>
<tr>
<th></th>
<th>Average Power</th>
<th>Peak Power</th>
<th>Time</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Mapping</td>
<td>1.98 W</td>
<td>9.70 W</td>
<td>115 us</td>
<td>0.23 mJ</td>
</tr>
<tr>
<td>One core</td>
<td>0.60 W</td>
<td>0.61 W</td>
<td>245 us</td>
<td>0.15 mJ</td>
</tr>
</tbody>
</table>

### Observations:
- Energy of an application rather constant
- Energy optimization opportunity for heterogeneous systems
- Huge optimization opportunity for average and peak power
- Trade-off of runtime vs power for multicore systems is a must
Thank you! Questions?