MULTICORE DIGITAL SIGNAL PROCESSING

Karol Desnos – kdesnos@insa-rennes.fr

Slides from M. Pelcat, K. Desnos, J.-F. Nezan, D. Ménard, M. Raulet, J. Gorin
Previously on MDSPs
Design Challenges for MPSoC-Based Systems

- Exploit architecture parallelism
  - Express application parallelism
  - Balance computational load on PEs

- Hardware/Software co-design process
  - Complex design-space exploration

- Respect constraints
  - Predict/guarantee application performances
  - Reuse legacy code
Previously on MDSPs

Grail of Heterogeneous MPSOcs Programming

Algorithm + High-level desc.

Architecture + High-level desc.

Multicore Compiler

Simulator + Debugger + Profiler

Portable Multicore Program

Multicore Runtime

Multicore DSPs – Karol Desnos (kdesnos@insa-rennes.fr)
Previously on MDSPs

Properties

• Synchronous Dataflow (SDF)
  • Data-driven execution: An actor is fired when its input FIFOs contain enough data-tokens.

Previously on MDSPs

Properties

• Synchronous Dataflow (SDF)
  • Parallelisms: Task / Data / Pipeline / Internal

Course Outline

• Lecture 1 – Maxime Pelcat
  • Introduction to the course
  • Applications for MDSPs

• Lecture 2 – Karol Desnos
  • Languages and MoCs
  • Programming MPSoCs
  • Dataflow MoCs

• Lecture 3 – Maxime Pelcat
  • Hardware Architectures
• Lecture 4 – Karol Desnos
  • Theoretical Bounds
  • Mapping/Scheduling Strategies

• Lecture 5 – Karol Desnos
  • Lab Session
Theoretical Bounds
Amdahl’s Law

- Developed in 1967 by Gene Amdahl
- A generic performance metric for applications

**Notations**
- $x$: ratio of the code that is perfectly parallel, the rest is sequential
- $N$ processing elements
- Speedup $S$ refers to the acceleration brought by adding cores

**Formulation**

- Ideal speedup for $N$ PE:

  $$S = \frac{1}{(1-x)+\frac{x}{N}}$$

- Maximum achievable speedup:

  $$S_{max} = \lim_{N \to \infty} S = \frac{1}{(1-x)}$$
Amdahl’s Law

- Example: with 70% of parallel code

As many threads as we want for 70% of code

A single thread for 30% of code
Amdahl’s Law

• Example: with 70% of parallel code
  • Speedup is limited to 1.0 on 1 core (no kidding !)
  • Speedup is limited to 1.5 on 2 cores
  • Speedup is limited to 2.1 on 4 cores
  • Speedup is limited to 2.6 on 8 cores
  • …

Max. speedup = 3.33
Amdahl’s Law

- Example:
  - Max speedup of 5.0 for 80%
  - Max speedup of 3.3 for 70%
  - Max speedup of 2.5 for 60%
Amdahl’s Law

- Limitation:
  - Inter-process communications are ignored
  - No computation is perfectly parallel

- Amdahl’s law has brought many doubts on multicores
  - Why add more cores if the parallelism of applications limits speedups so much?
Gustafson’s Law

- Developed in 1988 by John Gustafson

**Hypothesis: More cores imply more parallelism**
- Sequential code latency remains constant regardless the number of PE
- Parallel code is increased (by the developer) to fit the number of PE

**Notations**
- $S + P$: Sequential time + Parallel time with 1 PE
- $S + N \cdot P$: Sequential time + Parallel time with N PE
- $x = \frac{S}{S+P}$: Ratio of sequential time over total time (/!\ ≠ Amdahl /!)  

**Formulation**
- Ideal speedup for N PE:

$$Speedup = \frac{S + N \cdot P}{S + P} = \cdots = N - x \cdot (N - 1)$$
Gustafson’s Law

• Example: With 70% of parallel code
  • Speedup is limited to 1.7 on 2 cores (Amdahl: 1.5)
  • Speedup is limited to 3.1 on 4 cores (Amdahl: 2.1)
  • Speedup is limited to 5.9 on 8 cores (Amdahl: 2.6)
Dataflow Speedup

- Maximum speedup is given by finding the critical path
  - Data path whose sum of actor execution times is the largest

- Example (communications not considered):
  - Critical path length = 1 + 6 + 3 + 1 = 11 ms
  - Total work = 23 ms
  - Max speedup = 23 / 11 = 2.09
Dataflow Speedup

- **PREESM Speedup Assessment Chart**
  - Evaluate quality of a schedule

![Diagram showing the speedup assessment chart with tasks A, B, C, D, E, F, and G, and their respective processing times (1ms, 2ms, 3ms, 4ms, and 5ms). The chart illustrates the critical path length and architecture limit with a green and yellow shaded area, and a red line indicating dummy scheduling (fast but far from optimal).]
Dataflow Speedup

• Limitations of PREESM Speedup Assessment Chart
  • Only latency is considered
  • Software pipelining is not considered
    – Example: New critical path: 1+6 / New max speedup = 3.8
  • All cores are identical
  • All communications have the same speed
Mapping/Scheduling Strategies
• Heterogeneous Mapping/Scheduling Problem

• Heuristic Algorithms

• Load Balancing

• Runtime Systems
Schedu-what?

- Reminder

Mapping/Scheduling Strategies

Tasks and architecture

Mapping

Scheduling

Timing

Multicore DSPs – Karol Desnos (kdesnos@insa-rennes.fr)
Different Strategies

- Choices can be made during compile or run time.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Scheduling</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>fully dynamic</td>
<td>run</td>
<td>run</td>
</tr>
<tr>
<td>static-assignment</td>
<td>compile</td>
<td>run</td>
</tr>
<tr>
<td>self-timed</td>
<td>compile</td>
<td>compile</td>
</tr>
<tr>
<td>fully static</td>
<td>compile</td>
<td>compile</td>
</tr>
</tbody>
</table>

Source: E. Lee, “Scheduling Strategies for Multiprocessor real-time DSP”
Multicore DSPs – Karol Desnos (kdesnos@insa-rennes.fr)
Mapping/Scheduling Strategies

About Mapping, Scheduling, and Timing

• Part of “Operational Research”
  • How to organize a company
  • How to organize a project (Gantt Chart, …)
  • How to make decisions in general

• NP-Hard Problem
  • Verifying the validity of a solution to the problem can be computed in polynomial time (eg. verifying that a schedule is valid).

• No polynomial time algorithm for solving NP-complete problems is known (and it is likely that none exists.)

• When the problem grows (eg. number of cores or actors), solving it is becoming more complex exponentially.
Mapping/Scheduling Strategies

About Mapping, Scheduling, and Timing

• Multicore scheduling is equivalent to quadratic assignment NP-Hard problem
  • N facilities, each pair of facilities (f,g) associated to a flow of communication
  • N locations to put the facilities, each pair of locations (l,m) associated to a distance

• Objective: Put each facility on a location and minimize traffic (i.e. the sum of the distances multiplied by the corresponding flows)
Mapping/Scheduling Strategies

About Mapping, Scheduling, and Timing

• Real problem is even more complex
  • M facilities (i.e. actors)
  • N<M locations (i.e. cores) to put the facilities
  • Heterogeneity: actors have different costs on different cores

• Objective is not only communication minimization but also latency, throughput, memory, power…
• Heterogeneous Mapping/Scheduling Problem

• Heuristic Algorithms

• Load Balancing

• Runtime Systems
Mapping/Scheduling Strategies

Heterogeneous Mapping/Scheduling

• **Exact vs Heuristic Algorithms**
  • Exact algorithms find the optimal solution (exponential time)
  • Heuristics explore only parts of the given problem.

• **Many heuristics exist**
  • List scheduling, greedy scheduling
  • FAST scheduling (Y.-K. Kwok)
  • Hybrid flow-shop scheduling (J. Boutellier)
  • Meta-heuristics (genetic algorithms, ant colonies…)
  • …

• **Quality of heuristic results can not be predicted**
  • But models should contain enough information to make decisions
## Mapping/Scheduling Strategies

### Heterogeneous Mapping/Scheduling

- Several class of heuristic algorithms

<table>
<thead>
<tr>
<th>Constructive</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Specific</td>
<td>Generic Algorithms</td>
</tr>
<tr>
<td>- List scheduling</td>
<td>- Divide and conquer</td>
</tr>
<tr>
<td>- Greedy scheduling</td>
<td>- Branch and bound</td>
</tr>
<tr>
<td>- Hybrid flow-shop</td>
<td>- Integer Linear Programming</td>
</tr>
<tr>
<td>- FAST scheduling</td>
<td>- Genetic Algorithms</td>
</tr>
<tr>
<td></td>
<td>- Simulated annealing</td>
</tr>
<tr>
<td></td>
<td>- Ant colony</td>
</tr>
</tbody>
</table>

Source: Z. Peng, lecture notes of “Computer aided design of electronics”, LiU
List Scheduling Algorithm

1. Create a list of actors sorted in:
   - Topological order (i.e. data dependency order)
   - When equivalent, secondary sorting criteria is used: longest execution time, critical path before last task, …
**List Scheduling Algorithm**

2. Map and schedule actors to the first available PE:

**Longest Critical Path**

- A 1ms
- C 6ms
- B 4ms
- D 3ms
- F 5ms
- G 1ms

**With longest execution time**

- Core$_1$: A → C → F → G
- Core$_2$: B → D → E

14ms
Heterogeneous List Scheduling Algorithm

- Core that can finishes actor execution first wins
  - For heterogeneous targets

A

B

C

D

E

F

Longest (shortest exec. time) order: A C F B E D

CPU

DSP

Acc.
Heterogeneous List Scheduling Algorithm

- Scheduling order is important
- An optimal order always exists
  - Try all orders (exhaustive search to find the optimal
### FAST Iterative Heuristic

1. Create an initial solution with list scheduling
2. Iteratively
   1. Select a random actor from the critical path
   2. Change its mapping
   3. Reschedule and evaluate the resulting latency
3. Keep the best result

<table>
<thead>
<tr>
<th>Actor</th>
<th>CPU</th>
<th>DSP</th>
<th>Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9ms</td>
<td>2ms</td>
<td>3ms</td>
</tr>
<tr>
<td>B</td>
<td>4ms</td>
<td>8ms</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>5ms</td>
<td>2ms</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1ms</td>
<td>1ms</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3ms</td>
<td>4ms</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>8ms</td>
<td>4ms</td>
<td>.5ms</td>
</tr>
</tbody>
</table>
Genetic Iterative Heuristic

1. Create a pool of solutions with list/fast scheduling
   - Each solution is represented by its ordered list
2. Iteratively
   1. Discard the worst solutions
   2. Produce new solutions using cross-over and mutation
   3. Reschedule and evaluate the resulting latency
3. Keep the best result

Mapping/Scheduling Strategies

### Mutation

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

### Cross-over

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

### Example

- A = 2ms, B = 4ms, C = 3ms, D = 2ms, E = 3ms, F = 8ms
- New solutions generated through cross-over and mutation
- Rescheduling and evaluating the latency
Mapping/Scheduling Strategies

Scheduling under multiple constraints

• Example with latency and power

Time

<table>
<thead>
<tr>
<th>CPU</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2ms</td>
<td>4ms</td>
<td>5ms</td>
<td>3ms</td>
<td>2ms</td>
<td>8ms</td>
<td>4ms</td>
</tr>
</tbody>
</table>

Power

<table>
<thead>
<tr>
<th>CPU</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2J</td>
<td>4J</td>
<td>5J</td>
<td>3J</td>
<td>2J</td>
<td>10J</td>
</tr>
</tbody>
</table>

8ms – 11.1J

10ms – 9.1J
Mapping/Scheduling Strategies

• Heterogeneous Mapping/Scheduling Problem
• Heuristic Algorithms
• Load Balancing
• Runtime Systems
Load Balancing

Unbalanced power consumption

Unbalanced computational load

Multicore DSPs – Karol Desnos (kdesnos@insa-rennes.fr)
Load Balancing Strategies

- With total predictability (i.e. known number of tasks – SDF-like)
- Decentralized static decision
  - No adaptivity to algorithm modifications
  - No decision overhead
  - Self-timed execution
Load Balancing Strategies

- With high predictability (i.e. reconfigurable tasks)
- Master/Slave
  - Adaptivity to algorithm variations
  - Master core can become a bottleneck
Load Balancing Strategies

- Without predictability (i.e. highly dynamic number of tasks)
- Work Queueing
  - Implemented over multi-threading
  - Great freedom in thread creation
- The shared task queue becomes the bottleneck
Mapping/Scheduling Strategies

Load Balancing Strategies

- Without predictability (i.e. highly dynamic number of tasks)
- Job Stealing
  - One task queue per core: No more bottleneck
  - Hard to predict performance

Job stealing (Cilk, Intel Threading Building Blocks)
Mapping/Scheduling Strategies

• Heterogeneous Mapping/Scheduling Problem

• Heuristic Algorithms

• Load Balancing

• Runtime Systems
Mapping/Scheduling Strategies

Runtime systems

• Role of Runtime Systems for dynamic adaptation

Algorithm

+ High-level desc.

Architecture

+ High-level desc.

Multicore Compiler

Portable Multicore Program

01 11 01 10

Simulator + Debugger + Profiler

Multicore Runtime

Main Proc.

Main Proc.

Main Proc.

Main Memory

Peripherals

Multicore DSPs – Karol Desnos (kdesnos@insa-rennes.fr)
Runtime Systems

• Role of Runtime Systems for dynamic adaptation
  • A runtime system is the extension of an operating system for distributed hardware

• 2 types of multiprocessing management systems:
  • AMP (Asymmetric Multiprocessing)
    • complexity not masked each core has its own OS, e.g. SYS/BIOS, no runtime
  • SMP (Symmetric Multiprocessing)
    • simulating a unique core an OS controls the whole architecture, e.g. SMP Linux

• Limited notions, compilers and runtime systems are now combined
• For instance, OpenMP is based on a runtime system to dispatch threads
Runtime Systems

- The industry develops new runtime systems
  - Apple Grand Central Dispatch
  - Intel Threading Building Blocks
  - Texas Instruments Open Event Machine

- They are based on task and data synchronization descriptions
  - The semantics are getting close to dataflow

- Several runtime systems are experimented at IETR
  - Based on dataflow algorithm descriptions
  - Spider: Synchronous Parameterized and Interfaced Dataflow Embedded Runtime
Mapping/Scheduling Strategies

Runtime Systems

Parameter Values

Template graph → Dataflow Management → Master/Slave Phase → uC-OS/II based RTOS Scheduling → uC-OS/II Tasks

CLK

Core → Core → Core → Core
General Conclusion

• Applications and architectures are increasingly complex
  • Model-based system design helps at several design stages

• To evaluate languages/models: focus on MoC
  • MoCs offer « pure » semantics, free of syntax

• No one-fit-all solution to design Multicore DSP systems
  • Many solutions exist now, complex choices have to be made