On some Potential Research Contributions to the Multi-Core Enterprise

Oded Maler

CNRS - VERIMAG
Grenoble, France

February 2009

Background

- This presentation is based on observations made in the ATHOLE project with the participation of STM, CEA-LETI, THALES, CWS and VERIMAG
- Opinions are mine and do not necessarily represent other partners nor the ultimate truth
- The project is centered around low-power multi-core mobile architectures for stream-processing applications (video, audio, radio)
- Not all these observation are valid for all potential applications of the multi-core concept
- More on high-level performance modeling and analysis, less on programming

Motivation

- Complex electronic gadgets are designed and sold under tough competition constraints
- They should satisfy the following conflicting goals:
  - High performance
  - Low power
  - Short development time, adaptation to changes in standards and market needs
- Low-level development (hardware, micro-code, assembly) is better for optimizing performance
- High-level software: more flexible, effective and reliable development process
- Hardware is inherently parallel
- Software (and algorithms in general) is traditionally more of a sequential nature

Why Multicore for Mobile Streaming Applications?

- Today such systems are realized by special-purpose hardware
- The same silicon area can be allocated differently:
- A computation and communication fabric consisting of computation nodes connected via a network on chip (NoC)
- Computation nodes are simple general-purpose processors with local memory
- Some nodes may be special-purpose hardware accelerators obeying the same unified communication regime
- To be viable the platform should combine the relative simplicity and flexibility of software without too much performance penalty
- This means executing the software in a parallel fashion

Parallelism

- We are not concerned with the following problem:
- Take a sequential program, identify its maximal parallelism and find an optimal or satisfactory schedule
- Our starting point: stream-processing applications, described naturally in a dataflow style
- An application is viewed as a block diagram, a network of communicating “filters”
- With this formalism the dependence and independence between tasks is visible and the inherent parallelism is already “exposed”
- The body of a filter can be written as a sequential acyclic C-like program obeying some input-output convention

Task Graph Scheduling

- In principle, if we annotate filters with their execution times and know the number of processors we can apply our favorite task graph scheduling algorithm
- There are some features that render the straightforward application of standard scheduling algorithms difficult if not impossible:
  - The problems are recurrent: a stream of instances arrives from the outside (unlike loop parallelisation)
  - Performance optimization should be combined with power minimization
  - The application are data-intensive: communicating and transferring data among filters is sometime more significant and resource-consuming than the computations themselves
- Below we sketch some preliminary work to identify and tackle these problems

Recurrent Scheduling (with Aldric Degorre, FORMATS'08)

- The model:
  - Job types: a combination of task-graph (partial order) and job-shop (different types of machines)
  - Request generator: generates non-deterministically (but with bounded frequency) jobs of different types
  - Execution platform: a given number of machines for each type
  - Admissible request streams: accumulated demand for work does not exceed platform capacity
  - Scheduling policy/strategy: allocate machines to task instances, without knowing the future requests
- Results (theoretical):
  - Negative: some admissible request streams admit no schedule of bounded latency
  - Positive: a scheduling policy that can guarantee bounded backlog for all admissible request streams
  - Better understanding of the notion of pipelinability

Meeting Dealines Cheaply (with Julien Legriel, 2008)

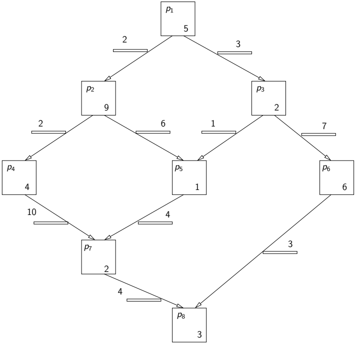
- Motivation: the execution platform may have different models varying in the number of processors
- Moreover, to control power consumption, the architecture is configurable and processors can be turned off or slowed down
- What is the cheapest (in terms of power) configuration (number of processors and their speeds) on which an application can be excuted with a reasonable performance (design-space exploration)

Meeting Dealines Cheaply (contd.)

- Modified task graph model: tasks defined in terms of quantity of work, not execution times (those depend on the speed of the processor)
- A (static) cost function on architecture configurations (linear function of the number of processors at each speed)
- Given a deadline, what is the cheapest architecture on which the graph can be scheduled to meet the deadline
- The problem is formulated as an SMT (SAT modulo theories, constrained optimization) and solved using the Yices solver
- Can solve problem with up to 40 tasks and 3 processor speeds
- Extension to periodic problems via finite unfolding

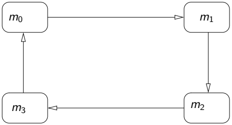
Handling Data (preliminary)

- Exposing non-parallelism: tasks that exchange a lot of data (directly or indirectly) should be executed on the same machine
- Task-data graph: the precedence between two tasks is also annotated by the volume of data communicated between them



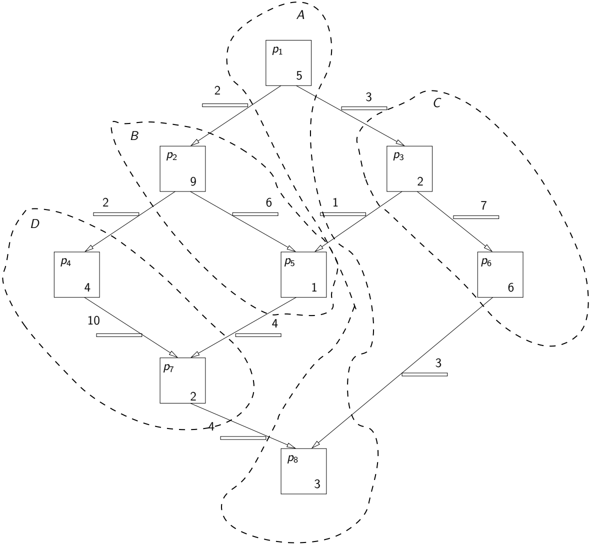
Network Topology

- The topology is not the full graph and some pairs of processors have distance > 1

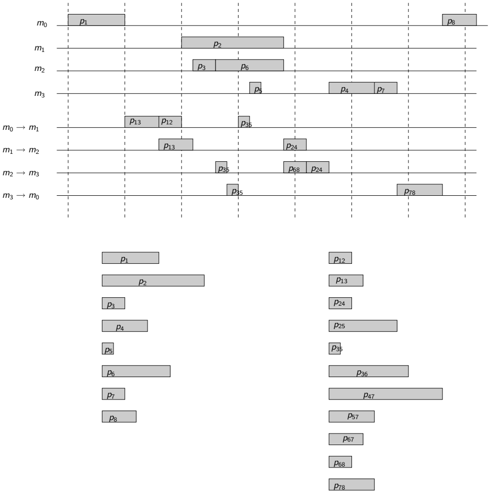


- Mapping: deciding which task runs on which processor and what path the communication between a pair of tasks uses
- Sometimes the path is fixed once the processors are determined
- Heuristic: try to minimize communication by running communicating tasks on the same machine (filter merging) or as close as possible
- Balance the computation load

A Mapping Example



From a Mapping to a Schedule



This is Not a Good Solution

- We assumed each task waits for all its data before executing and sends all its output upon termination
- Hence we could use scheduling in the classical sense: allocate resources (processors and communication channels) deterministically
- This is not the underlying philosophy and methodology for these applications (unlike hard real time)
- Computation and communication are interleaved, tasks are executed in data-driven multi threading
- The behavior of the network is more statistical in nature, bulk reasoning, load balancing, etc.

Half Baked Ideas

- Current conceptual challenge: how to combine these points of views, exact scheduling and throughput reasoning
- Reasoning only by quantity of work ignores precedence constraints
- On the othe hand, pipelined execution may render precedence less important
- Maybe should invent or reinvent a computational model with computation and transportation as basic entities
- You start with x at some location and want to have some complex $f(x)$ at another location
- How you map the "parse tree" of f on the architecture

Timed Automata

- We use timed automata (and the IF toolbox) as the underlying model for performance analysis
- Ideal for modeling a task that takes some time to execute
- Can express timing nondeterminism (lower and upper bound)
- In the past (with Y. Abdeddaim and E. Asarin) we have shown how to reduce optimal scheduling to shortest path in timed automata
- Showed also how to derive dynamic scheduling strategies for task graph with temporal undertainty (unfortunately it does not scale up)
- Currently (with JF Kempf, M. Bozga and R. Ben Salah) we develop a toolset for defining tasks, architectures, request generators and scheduling policies, and evalutate their performance using IF
- Can serve for design-space exploration at early stages of the development process before code is written

Thank you