Motivation for Course (1)

According to forecasts, the future of IT is characterized by terms such as

- disappearing computers,
- ubiquitous computing,
- pervasive computing,
- ambient intelligence,
- the Post-PC era, and
- cyber-physical systems.

Basic technologies:

- Embedded Systems
- Communication technologies
Motivation for Course (2)

“Information technology (IT) is on the verge of another revolution. ..... networked systems of embedded computers ... have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways. ...

The use ... throughout society could well dwarf previous milestones in the information revolution.”

National Research Council Report (US)
Embedded Everywhere, 2001

What is an embedded system?
Embedded Systems & Cyber-Physical Systems

“Dortmund“ Definition: [Peter Marwedel]

Embedded systems are information processing systems embedded into a larger product

Berkeley: [Edward A. Lee]:
Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems.

Definition: Cyber-Physical (cy-phy) Systems (CPS) are integrations of computation with physical processes [Edward A. Lee, 2006].

Growing importance of embedded systems

- the global mobile entertainment industry is now worth some $32 bln…predicting average revenue growth of 28% for 2010 [www.itfacts.biz, July 8th, 2009]
- …, the market for remote home health monitoring is expected to generate $225 mln revenue in 2011, up from less than $70 mln in 2006, according to Parks Associates. [www.itfacts.biz, Sep. 4th, 2007]
- Funding in the 7th European Framework
- Creation of the ARTEMIS Joint Undertaking in Europe
- Funding of CPS research in the US
- Joint education effort of Taiwanese Universities
- …
Application areas and examples

Automotive electronics

Functions by embedded processing:
- ABS: Anti-lock braking systems
- ESP: Electronic stability control
- Airbags
- Efficient automatic gearboxes
- Theft prevention with smart keys
- Blind-angle alert systems
- ... etc ...

Multiple networks
- Body, engine, telematics, media, safety, ...

Multiple networked processors
- Up to 100


Microsoft copyright restrictions apply.

© Jakob Engblom
Transportation (continued)

Avionics
- Flight control systems,
- Pilot information systems,
- ...

Railways
- Safety features contribute significantly to the total value of trains.
- Integrated systems are required, especially for high speeds.
- Example: European Rail Traffic Management System

Dependability is of utmost importance.

---

Telecommunication & Consumer electronics

Telecommunication:
- Mobile phones (one of the fastest growing markets in the recent years),
- Geo-positioning systems,
- Closed systems for police, ambulances, rescue staff.

Consumer electronics:
- TV sets,
- Smart personal assistants.
IT in Healthcare, Biometric systems, security

IT in Healthcare
- Artificial eyes:
  - Connection to brain [www.dobelle.com]
  - Translation into sound; [http://www.seeingwithsound.com/etumble.htm]
- ...

Biometric systems
- Finger print sensors,
- Face recognition,
- Handwriting,
- ...

Industrial automation & smart buildings

Industrial automation
- Factories, ...

Smart buildings
- Integrated cooling, lightning, room reservation, emergency handling, communication.
- Goal: “Zero-energy building”
- Expected contribution to fight against global warming
Common characteristics

ES must be dependable,

- **Reliability** $R(t) = \text{probability of system working correctly provided that it was working at } t=0$
- **Maintainability** $M(d) = \text{probability of system working correctly } d \text{ time units after error occurred.}$
- **Availability** $A(t): \text{probability of system working at time } t$
- **Safety**: no harm to be caused
- **Security**: confidential and authentic communication

*Even perfectly designed systems can fail if the assumptions about the workload and possible errors turn out to be wrong.*

*Making the system dependable must not be an afterthought, it must be considered from the very beginning.*
Examples of problems

- Non-real time protocols used for real-time applications (e.g. Berlin fire department)
- Over-simplification of models (e.g. aircraft anti-collision system)
- Using unsafe systems for safety-critical missions (e.g. voice control system in Los Angeles; ~ 800 planes without voice connection to tower for > 3 hrs)

Efficiency

- ES must be **efficient**
  - Code-size efficient (especially for systems on a chip)
  - Run-time efficient
  - Weight efficient
  - Cost efficient
  - Energy efficient
Importance of Energy Efficiency

Efficient software design needed, otherwise, the price for software flexibility cannot be paid.

© Hugo De Man, IMEC, Philips, 2007

---

Embedded System Hardware

Embedded system hardware is frequently used in a loop ("hardware in a loop"):

A/D converter → sensors

information processing → (physical) environment

display → D/A converter

Cyber-physical systems
Real-time constraints

- Many ES must meet **real-time constraints**
  - “A real-time system must react to stimuli from the controlled object (or the operator) within the time interval dictated by the environment”.
  - For real-time systems, right answers arriving too late are wrong.
  - “A real-time constraint is called hard, if not meeting that constraint could result in a catastrophe“ [Kopetz, 1997].
  - All other time-constraints are called **soft**.
  - A guaranteed system response has to be explained without statistical arguments.

Reactive & hybrid systems

- Typically, ES are **reactive systems**: “A reactive system is one which is in continual interaction with its environment and executes at a pace determined by that environment“ [Bergé, 1995]
  - Behavior depends on input and current state.
  - Automata model appropriate, model of computable functions inappropriate.

- **Hybrid systems** (analog + digital parts).
Dedicated systems

- **Dedicated** towards a certain application
  Knowledge about behavior at design time can be used to minimize resources and to maximize robustness

- **Dedicated user interface**
  (no mouse, keyboard and screen)

It is not sufficient to consider ES just as a special case of software engineering

EE knowledge must be available,
Walls between EE and CS must be torn down

The same for walls to other disciplines and more challenges ….
Specification techniques for embedded systems

Hypothetical design flow

- Specification
- ES-hardware
- System software (RTOS, middleware, …)
- Design repository
- Application mapping
- Optimization
- Evaluation (energy, cost, performance, …)
- Validation & test
- Design

Generic loop: tool chains differ in the number and type of iterations
The V-model as a special case

Motivation for considering specs

- Why considering specs?
- If something is wrong with the specs, then it will be difficult to get the design right, potentially wasting a lot of time.
- Typically, we work with models of the system under design (SUD)
  - What is a model anyway?
Models

Definition: “A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.”

[Jantsch, 2004]:

Which requirements do we have for our models?

Requirements for specification techniques (1): Hierarchy

- Hierarchy
  Humans not capable to understand systems containing more than ~5 objects.
  Most actual systems require more objects
  – Hierarchy (+abstraction)
    - Behavioral hierarchy
      Examples: states, processes, procedures.
    - Structural hierarchy
      Examples: processors, racks, printed circuit boards
Requirements for specification techniques (2): Component-based design

- Systems must be designed from components
- Must be “easy” to derive behavior from behavior of subsystems
  - Work of Sifakis, Thiele, Ernst, ...
- Concurrency
- Synchronization and communication

Requirements for specification techniques (3): Timing

- Timing behavior
  - Essential for embedded and cy-phy systems!
    - Additional information (periods, dependences, scenarios, use cases) welcome
    - Also, the speed of the underlying platform must be known
    - Far-reaching consequences for design processes!

“The lack of timing in the core abstraction (of computer science) is a flaw, from the perspective of embedded software” [Lee, 2005]
Requirements for specification techniques (4): Support for reactive systems

- **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.

- **Event-handling**
  (external or internal events)

- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state

We will see, how all the arrows labeled \( k \) can be replaced by a single one.

Requirements for specification techniques (5)

- Presence of programming elements
- Executability (no algebraic specification)
- Support for the design of large systems (\( \mathcal{O} \) OO)
- Domain-specific support
- Readability
- Portability and flexibility
- Termination
- Support for non-standard I/O devices
- Non-functional properties
- Support for the design of dependable systems
- No obstacles for efficient implementation
- Adequate model of computation

*What does it mean “to compute”?*
What does it mean, “to compute”?

Models of computation define:
- Components and an execution model for computations for each component
- Communication model for exchange of information between components.

Communication

- Shared memory

Variables accessible to several components/tasks.

Model mostly restricted to local systems.
Shared memory

Potential race conditions (inconsistent results possible)

Critical sections = sections at which exclusive access to resource $r$ (e.g. shared memory) must be guaranteed.

```plaintext
task a {
.. P(S) //obtain lock
.. // critical section
V(S) //release lock
}
task b {
.. P(S) //obtain lock
.. // critical section
V(S) //release lock
}
```

P(S) and V(S) are semaphore operations, allowing at most $n$ accesses, $n = 1$ in this case (mutex, lock)

Non-blocking/asynchronous message passing

Sender does not have to wait until message has arrived;

Potential problem: buffer overflow
Blocking/synchronous message passing *rendez-vous*

Sender will wait until receiver has received message

No buffer overflow, but reduced performance.

---

**Organization of computations within the components (1)**

- Finite state machines
- Data flow (models the flow of data in a distributed system)
- Differential equations

\[
\frac{\partial^2 x}{\partial t^2} = b
\]
Organization of computations within the components (2)

- Discrete event model

<table>
<thead>
<tr>
<th>queue</th>
<th>5</th>
<th>10</th>
<th>13</th>
<th>15</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>time action</td>
<td>a:=5 b:=7 c:=8 a:=6 a:=9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Von Neumann model

Sequential execution, program memory etc.

Questions?

Q&A?
# Specification techniques for embedded systems

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.

## Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP, ADA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
Support for early design phases

- Informal text

The system must respond to incoming calls. It must play the welcome message followed by a beep and then start recording ...

- (Message) sequence charts

Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td></td>
<td></td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td></td>
<td>SDL</td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td></td>
<td>Kahn networks, SDF</td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td></td>
<td>C/E nets, P/T nets, ...</td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, ...</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td>CSP, ADA</td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
StateCharts

Extending classical automata to model ES & CPS

- Adding timing with timed automata (François Friday tutorial)
- Adding hierarchy:
  Complex graphs cannot be understood by humans.
  François Introduction of hierarchy François StateCharts [Harel, 1987]

StateChart = the only unused combination of „flow“ or „state“ with „diagram“ or „chart“

Used here as a (prominent) example of a model of computation based on shared memory communication, appropriate only for local (non-distributed) systems

Introducing hierarchy

FSM will be in exactly one of the substates of S if S is active (either in A or in B or ..)
Definitions

- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states**.
- States containing other states are called **super-states**.
- For each basic state \( s \), the super-states containing \( s \) are called **ancestor states**.
- Super-states \( S \) are called **OR-super-states**, if exactly one of the sub-states of \( S \) is active whenever \( S \) is active.

![Diagram of superstate and ancestor state of E](image)

---

Default state mechanism

Try to hide internal structure from outside world!

(Default state)

Filled circle indicates sub-state entered whenever super-state is entered.

Not a state by itself!
Concurrency

Convenient ways of describing concurrency are required.

**AND-super-states:** FSM is in **all** (immediate) sub-states of a super-state.

---

Types of states

In StateCharts, states are either

- **basic states,** or
- **AND-super-states,** or
- **OR-super-states.**
Timers

Since time needs to be modeled in embedded systems, timers need to be modeled. In StateCharts, special edges can be used for timeouts.

If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

Using timers in an answering machine

Using timers in an answering machine.
The StateCharts simulation phases (StateMate Semantics)

How are edge labels evaluated?
Three phases:
1. Effect of external changes on events and conditions is evaluated,
2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
3. Transitions become effective, variables obtain new values.
Separation into phases 2 and 3 guarantees and reproducible behavior.

Example

In phase 2, variables $a$ and $b$ are assigned to temporary variables:

In phase 3, these are assigned to $a$ and $b$.

As a result, variables $a$ and $b$ are swapped.
Example (2)

In a single phase environment, executing the left state first would assign the old value of $b$ (=0) to $a$ and $b$:

Executing the right state first would assign the old value of $a$ (=1) to $a$ and $b$.

The result would depend on the execution order.

Reflects model of clocked hardware

In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.
Steps

Execution of a StateMate model consists of a sequence of (status, step) pairs

\[ \text{Status} \quad \text{Step} \quad \text{Status} \quad \text{Step} \quad \text{Status} \quad \text{Step} \quad \text{Status} \]

Status = values of all variables + set of events + current time
Step = execution of the three phases (StateMate semantics)

Other implementations of StateCharts do not have these 3 phases (and hence are non-determinate)!

Other semantics

Several other specification languages for hierarchical state machines (UML, ...) do not include the three simulation phases.

These correspond more to a SW point of view with no synchronous clocks.

Some systems allow turning the multi-phased simulation on and off.
Broadcast mechanism

Values of variables are visible to all parts of the StateChart model. New values become effective in phase 3 of the current step and are obtained by all parts of the model in the following step.

- StateCharts implicitly assumes a broadcast mechanism for variables (→ implicit shared memory communication—that other implementations would be very inefficient -).
- StateCharts is appropriate for local control systems (😊), but not for distributed applications for which updating variables might take some time (😢).

Evaluation of StateCharts

Pros:

- Hierarchy allows arbitrary nesting of AND- and OR-super states.
- (StateMate-) Semantics defined in a follow-up paper to original paper.
- Large number of commercial simulation tools available (StateMate, StateFlow, ...)
- Available “back-ends“ translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

Cons:

- Not useful for distributed applications,
- No program constructs,
- No description of non-functional behavior,
- No object-orientation,
- No description of structural hierarchy.
### Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Undefined components</strong></td>
<td>Plain text, use cases</td>
<td>Synchronous</td>
</tr>
<tr>
<td><strong>Communic. finite state machines</strong></td>
<td>StateCharts</td>
<td>SDL</td>
</tr>
<tr>
<td><strong>Data flow</strong></td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
</tr>
<tr>
<td><strong>Petri nets</strong></td>
<td></td>
<td>C/E nets, P/T nets, …</td>
</tr>
<tr>
<td><strong>Discrete event (DE) model</strong></td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
</tr>
<tr>
<td><strong>Von-Neumann model</strong></td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP, ADA</td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue

---

### SDL

Language designed for distributed systems.

- Dates back to early 70s,
- Formal semantics defined in the late 80s,
- Defined by ITU (International Telecommunication Union): Z.100 recommendation in 1980
- Provides textual and graphical formats to please all users,
- Just like StateCharts, it is based on the CFSM model of computation; each FSM is called a **process**,  
- However, it uses message passing instead of shared memory for communications,
- SDL supports operations on data.
Communication among SDL-FSMs

Communication between FSMs (or “processes“) is based on message-passing, assuming a potentially indefinitely large FIFO-queue.

- Each process fetches next entry from FIFO,
- checks if input enables transition,
- if yes: transition takes place,
- if no: input is ignored (exception: SAVE-mechanism).
Determinate?

Let tokens be arriving at FIFO at the same time:
- Order in which they are stored, is unknown:

All orders are legal: simulators can show different behaviors for the same input, all of which are correct.

Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP, ADA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
Data flow as a “natural” model of applications

Example: Video on demand system

Data flow modeling

Definition: Data flow modeling is ... “the process of identifying, modeling and documenting how data moves around an information system. Data flow modeling examines

- processes (activities that transform data from one form to another),
- data stores (the holding areas for data),
- external entities (what sends data into a system or receives data from a system, and
- data flows (routes by which data can flow).

Kahn process networks

- Each component is a program/task/process, not an FSM
- Communication is by FIFOs; no overflow considered
  - writes never have to wait,
  - reads wait if FIFO is empty.
- Only one sender and one receiver per FIFO
  - no SDL-like conflicts at FIFOs

Example

Process f(in int u, in int v, out int w){
    int i; bool b = true;
    for (;;) {
        i= b ? wait(u) : wait(v);
        //wait returns next token in FIFO, waits if empty
        send (i,w);  //writes a token into a FIFO w/o blocking
        b = !b;
    }

© R. Gupta (UCSD), W. Wolf (Princeton), 2003
Properties of Kahn process networks

- Communication is only via channels;
- Mapping from \( \geq 1 \) input channel to \( \geq 1 \) output channel;
- Channels transmit information within an unpredictable but finite amount of time;
- In general, execution times are unknown.

Key beauty of KPNs

- A process cannot check whether data is available before attempting a read.
- A process cannot wait for data for more than one port at a time.
- Therefore, the order of reads depends only on data, not on the arrival time.
- Therefore, for a given input, for Kahn process networks the result will always the same, regardless of the speed of the nodes.
- Many applications in embedded system design: simplifies emulation of real systems.
## Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td></td>
<td>Kahn networks, SDF</td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td>CSP, ADA</td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue.

---

**SDF**

Less computationally powerful, but easier to analyze:

Synchronous data flow (SDF).

Again using asynchronous message passing.
Synchronous data flow (SDF)

Synchronous data flow =
- global clock controlling “firing” of nodes
Asynchronous message passing =
- tasks do not have to wait until output is accepted.

In the general case, a number of tokens can be produced/consumed per firing; firing rate depends on # of tokens ...

Parallel Scheduling of SDF Models

SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.

Many scheduling optimization problems can be formulated. Some can be solved, too!
Balance equations (one for each channel)

\[ f_A N = f_B M \]

- number of tokens consumed
- number of tokens produced
- number of firings per “iteration”

fire A { 
... produce N  
... 
}
channel
N
fire B { 
... consume M  
... 
}

Schedulable statically
In the general case, buffers may be needed at edges.
Decidable:
- buffer memory requirements
- deadlock

Source: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

---

Similar MoC: Simulink
- example -

Semantics? “Simulink uses an idealized timing model for block execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps.” [Nicolae Marian, Yue Ma]

From www.mathworks.co.uk/access/helpdesk/help/toolbox/fuzzy/fuzzyt25.shtml
Summary

Specifications and Modeling
- Early phases
  - Text
  - Use Cases
  - (Message) Sequence Charts
- FSM-based models
  - Shared memory-based (StateCharts)
  - Message passing-based (SDL)
- Data flow
  - Kahn process networks
  - Synchronous data flow

Questions?

Q&A?
## Specifications and Modeling (2)

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.

## Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/ local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP, ADA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
Introduction

Introduced in 1962 by Carl Adam Petri in his PhD thesis. Focus on modeling causal dependencies; no global synchronization assumed (message passing only).

Key elements:

- **Conditions**
  - Either met or no met.

- **Events**
  - May take place if certain conditions are met.

- **Flow relation**
  - Relates conditions and events.

Conditions, events and the flow relation form a **bipartite graph** (graph with two kinds of nodes).

Example: Synchronization at single track rail segment
Playing the "token game"

Conflict for resource "track"
Petri nets & UML: Activity diagram

Extended Petri nets. Include decisions (like in flow charts). Graphical notation similar to SDL.

Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP, ADA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
HDLs using discrete event (DE) semantics

Used in hardware description languages (HDLs):
Description of concurrency is a must for HW description languages!

- Many HW components are operating concurrently
- Typically mapped to “processes“
- These processes communicate via “signals“
- Examples:
  - MIMOLA [Zimmermann/Marwedel], ~1975 …
  - VHDL (very prominent example in DE modeling)
    One of the 3 most important HDLs:
    VHDL, Verilog, SystemC
    Definition started in 1980, updated every 5 years

Discrete event semantics

Basic discrete event (DE) semantics

- Queue of future actions, sorted by time
- Loop:
  - Fetch next entry from queue
  - Perform function as listed in entry
    - May include generation of new entries
- Until termination criterion = true

<table>
<thead>
<tr>
<th>queue</th>
<th>time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 10 13 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:=5</td>
<td>b:=7</td>
<td>c:=8</td>
</tr>
<tr>
<td>a:=6</td>
<td>a:=9</td>
<td></td>
</tr>
</tbody>
</table>
Processes will wait for changes on their input ports. If they arrive, processes will wake up, compute their code and deposit changes of output signals in the event queue and wait for the next event. If all processes wait, the next entry will be taken from the event queue.

**Simple example (VHDL notation)**

gate1:

```
process (a,b)
begin
  c <= a nor b;
end;
```

gate2:

```
process (a,b)
begin
  c <= a nor b;
end;
```

**VHDL processes**

Delays allowed:

```vhdl
process (a,b)
begin
  c <= a nor b after 10 ns;
end;
```

Equivalent to

```vhdl
process
begin
  c <= a nor b after 10 ns;
  wait on a,b;
end;
```

- `<=`: signal assignment operator
- Each executed signal assignment will result in adding entries in the projected waveform, as indicated by the (optional) delay time
- Implicit loop around the code in the body
- Sensitivity lists are a shorthand for a single `wait on`-statement at the end of the process body
**δ-simulation cycles**
Simulation of an RS-Flipflop

![Petri net diagram]

<table>
<thead>
<tr>
<th>0ns</th>
<th>0ns+δ</th>
<th>0ns+2δ</th>
<th>0ns+3δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nQ</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**δ cycles reflect the fact that no real gate comes with zero delay.**

Should delay-less signal assignments be allowed at all?

---

**Models of computation considered in this course**

<table>
<thead>
<tr>
<th>Communication/ local computation</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>(Message) sequence charts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communic. finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, SDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog, SystemC, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Von-Neumann model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue
Imperative (von-Neumann) model

The von-Neumann model reflects the principles of operation of standard computers:

- Sequential execution of instructions (sequential control flow, fixed sequence of operations)
- Possible branches
- Partitioning of applications into threads
- In most cases:
  - Context switching between threads, frequently based on pre-emption (cooperative multi-tasking or time-triggered context switch less common)
  - Access to shared memory

From implementation concepts to programming models

Example languages

- Machine languages (binary)
- Assembly languages (mnemonics)
- Imperative languages providing a limited abstraction of machine languages (C, C++, Java, ...)

Threads/processes

- Initially available only as entities managed by the operating system
- Made available to the programmer as well
- Languages initially not designed for communication, availability of threads made synchronization and communication a must.
Communication via shared memory

Several threads access the same memory
- Very fast communication technique (no extra copying)
- Potential race conditions:

```plaintext
thread a {
  u = 1;
  if u<5 {u = u + 1; ..}
}

thread b {
  ..
  u = 5
}
```

Context switch after the test could result in u == 6.
- inconsistent results possible
- Critical sections = sections at which exclusive access to resource r (e.g. shared memory) must be guaranteed

Shared memory

```plaintext
thread a {
  u = 1; ..
  P(S) //obtain mutex
  if u<5 {u = u + 1; ..}
    // critical section
  V(S) //release mutex
}

thread b {
  ..
  P(S) //obtain mutex
  u = 5
    // critical section
  V(S) //release mutex
}
```

S: semaphore
P(S) grants up to n concurrent accesses to resource
n=1 in this case (mutex/lock)
V(S) increases number of allowed accesses to resource

**Imperative model should be supported by:**
- mutual exclusion for critical sections
- cache coherency protocols
Synchronous message passing: CSP

- CSP (communicating sequential processes) [Hoare, 1985],
  Rendez-vous-based communication:
  Example:

    process A
    ...
    var a ...
    a:=3;
    c!a; -- output
    end

    process B
    ...
    var b ...
    ...
    c?b; -- input
    end

Communication/synchronization

- Special communication libraries for ES & CPS
  - OSEK/VDX COM
  - ...

- Adopted communication libraries for general computing
  - CORBA (Common Object Request Broker Architecture)
  - Message passing interface (MPI)
  - Posix threads (PThreads)
  - OpenMP
  - UPnP, DPWS, JXTA, ...

  Frequently not easy to adjust to real-time requirements
Deadlocks

Deadlocks can happen, if the following 4 conditions are met [Coffman, 1971]:

- **Mutual exclusion**: a resource that cannot be used by >1 thread at a time
- **Hold and wait**: thread already holding resources may request new resources
- **No preemption**: Resource cannot be forcibly removed from threads, they can be released only by the holding threads
- **Circular wait**: ≥ 2 threads form a circular chain where each thread waits for a resource that the next thread in the chain holds

There is no general, always applicable technique for turning one of these conditions false.

In non-safety-critical software, it is “ok” to ensure that deadlocks are “sufficiently” infrequent.

Mutual exclusion in Java

“The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995
Mutexes using monitors are minefields

```java
public synchronized void addListener(listener) {
    (...)
}
public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue)
    }
}
```

`valueChanged()` may attempt to acquire a lock on some other object and stall. If the holder of that lock calls `addListener()`: deadlock!

---

Problems with imperative languages and shared memory

- Potential deadlocks
- Specification of total order of operations is an over-specification. A partial order would be sufficient. The total order reduces the potential for optimizations
- Timing cannot be specified
- Access to shared memory leads to anomalies, that have to be pruned away by mutexes, semaphores, monitors
- Access to shared, protected resources leads to priority inversion
- Termination in general undecidable
- Preemptions at any time complicate timing analysis
Comparison of models

Expressiveness of data flow MoCs

KPN

CSDF

SDF

HSDF

Turing-complete

Not Turing-complete

HSDF = Homogeneous synchronous data flow (all firing rates are the same)

CSDF = Cyclo static data flow (rates vary in a cyclic way)

The expressiveness/analyzability conflict

Expressiveness and succinctness

- Kahn process networks
- SDF
- Homogeneous SDF (HSDF)

Analyzability

Implementation efficiency


How to cope with MoC and language problems in practice?

Mixed approaches:

(RT-) UML or equivalent

SDL

C-programs

Assembly programs

Objectcode

(RT-) UML or equivalent

(VHDL

Net list

Hardware

Objectcode

Mixing models may require formal models of MoCs
Mixing models of computation: Ptolemy

Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation.

http://ptolemy.berkeley.edu/

Available examples are restricted to a subset of the supported models of computation.

Newton’s cradle

Mixing MoCs: Ptolemy
(Focus on executable models; “mature” models only)

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>FSM, synchronous/reactive MoC</td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td></td>
<td>Kahn networks, SDF, dynamic dataflow, discrete time</td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>DE</td>
<td>Experimental distributed DE</td>
</tr>
<tr>
<td>Von Neumann model</td>
<td></td>
<td>CSP</td>
</tr>
<tr>
<td>Wireless</td>
<td></td>
<td>Special model for wireless communication</td>
</tr>
<tr>
<td>Continuous time</td>
<td></td>
<td>Partial differential equations</td>
</tr>
</tbody>
</table>
Mixing models of computation: UML
(Focus on support of early design phases)

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Use cases</td>
<td>Synchronous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>State diagrams</td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Data flow</td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>Activity charts</td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Von Neumann model</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

UML for embedded systems?

Initially not designed for real-time.

Initially lacking features:
- Partitioning of software into tasks and processes
- specifying timing
- specification of hardware components

Projects on defining profiles for embedded/real-time systems
- Schedulability, Performance and Timing Analysis
- SysML (System Modeling Language)
- UML Profile for SoC
- Modeling and Analysis of Real-Time Embedded Systems
- UML/SystemC, …

Profiles may be incompatible
There is a tradeoff between the power of a modeling technique and its analyzability. In any case, open your eyes & think about the model before you write down your spec! Be aware of pitfalls. It may be necessary to combine modeling techniques. You may be forced, to use imperative models, but you can still implement, for example, finite state machines or KPNs in Java.
Summary

- Imperative Von-Neumann models
  - Problems resulting from access to shared resources and mutual exclusion (e.g. potential deadlock)
  - Communication built-in or by libraries
- Comparison of models
  - Expressiveness vs. analyzability
  - Process creation
  - Mixing models of computation
    - Ptolemy & UML
    - Using FSM and KPN models in imperative languages, etc.

Structure of this course

- Specification
- ES-hardware
- System software (RTOS, middleware, ...)
- Design repository
- Application mapping
- Optimization
  - Evaluation (energy, cost, performance, ...)
- Validation & test
- Design
Trends for the Speeds

Similar problems also for embedded systems & MPSoCs

- In the future:
  Memory access times >> processor cycle times
- “Memory wall” problem

Memory

For the memory, efficiency is again a concern:
- speed (latency and throughput); predictable timing
- energy efficiency
- size
- cost
- other attributes (volatile vs. persistent, etc)
Energy consumption in mobile devices

![Energy consumption chart](chart.png)

- **Audio codec and amplifiers**: 10%
- **Color display and backlighting**: 10%
- **RF modem and amplifier**: 30%
- **Other peripherals**: 10%
- **Memories**: 20%
- **Application processor**: 20%

Source: Siemens

[O. Vargas (Infineon Technologies): Minimum power consumption in mobile-phone memory subsystems; Pennwell Portable Design - September 2005;] Thanks to Thorsten Koch (Nokia/Univ. Dortmund) for providing this source.

Access times and energy consumption for multi-ported register files

<table>
<thead>
<tr>
<th>Cycle Time (ns)</th>
<th>Area ($\lambda^2 \times 10^6$)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File Size</td>
<td>GP6M2</td>
<td>GP6M3</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>1.1</td>
<td>2.2</td>
</tr>
<tr>
<td>64</td>
<td>1.3</td>
<td>2.4</td>
</tr>
<tr>
<td>128</td>
<td>1.5</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Rixner's et al. model [HPCA'00], Technology of 0.18 µm

Source and © H. Valero, 2001
Exploitation of the memory hierarchy
Access times and energy consumption increases with the size of the memory

Example (CACTI Model):

"Currently, the size of some applications is doubling every 10 months" [STMicroelectronics, Medea+ Workshop, Stuttgart, Nov. 2003]

Hierarchical memories using scratch pad memories (SPM)

SPM is a small, physically separate memory mapped into the address space

Hierarchical memories using scratch pad memories (SPM)

Address space

0

scratch pad memory

FFF...

no tag memory

Selection is by an appropriate address decoder (simple!)

Example

ARM7TDMI cores, well-known for low power consumption
Comparison of currents using measurements

E.g.: ATMEL board with ARM7TDMI and ext. SRAM

![Comparison of currents using measurements](image)

Why not just use a cache?

Energy for parallel access of sets, in comparators, muxes.

![Energy for parallel access of sets, in comparators, muxes](image)
Influence of the associativity

![Graph showing energy per access vs memory size for different cache configurations.]

Parameters different from previous slides

[P. Marwedel et al., ASPDAC, 2004]

Very limited support in ARMcc-based tool flows

1. Use pragma in C-source to allocate to specific section: For example:
   ```
   #pragma arm section rdata = "foo", rodata = "bar"
   int x2 = 5; // in foo (data part of region)
   int const z2[3] = {1,2,3}; // in bar
   ```

2. Input scatter loading file to linker for allocating section to specific address range

   ```
   LOAD_ROM_1 0x0000
   {
     EXEC_ROM_1 0x0000
     {
       program1.o (+RO)
     }
   }

   DRAM 0x18000 0x6000
   {
     program1.o (+RW,+ZI)
   }
   ```

Migration of data & instructions, global optimization model (TU Dortmund)

Example:

For i .{ } for j ..{ } while ...
Repeat call ...

Which memory object (array, loop, etc.) to be stored in SPM?

Non-overlaying (“Static”) allocation:

Gain $g_k$ and size $s_k$ for each object $k$. Maximise gain $G = \sum g_k$, respecting size of SPM $SSP \geq \sum s_k$.

Solution: knapsack algorithm.

Overlaying ("dynamic") allocation:

Moving objects back and forth

IP representation
- migrating functions and variables-

Symbols:

$S(var_k) = \text{size of variable } k$

$n(var_k) = \text{number of accesses to variable } k$

$e(var_k) = \text{energy saved per variable access, if } var_k \text{ is migrated}$

$E(var_k) = \text{energy saved if variable } var_k \text{ is migrated} = e(var_k) n(var_k)$

$x(var_k) = \text{decision variable, } = 1 \text{ if variable } k \text{ is migrated to SPM, } = 0 \text{ otherwise}$

$K = \text{set of variables}; \text{Similar for functions } I$

Integer programming formulation:

Maximize $\sum_{k \in K} x(var_k) E(var_k) + \sum_{i \in I} x(F_i) E(F_i)$

Subject to the constraint

$\sum_{k \in K} S(var_k) x(var_k) + \sum_{i \in I} S(F_i) x(F_i) \leq SSP$
Reduction in energy and average runtime

Feasible with standard compiler & postpass optimization

Measured processor / external memory energy + CACTI values for SPM (combined model)

Numbers will change with technology, algorithms remain unchanged.

Allocation of basic blocks

Fine-grained granularity smoothens dependency on the size of the scratch pad.

Requires additional jump instructions to return to "main" memory.
Allocation of basic blocks, sets of adjacent basic blocks and the stack

Requires generation of additional jumps (special compiler)

Savings for memory system energy alone

Combined model for memories
Multiple scratch pads

Small is beautiful:
One small SPM is beautiful (😊).
May be, several smaller SPMs are even more beautiful (😊😊😊)?

scratch pad 0, 256 entries
scratch pad 1, 2 k entries
scratch pad 2, 16 k entries
background memory

Optimization for multiple scratch pads

Minimize \[ C = \sum_j e_j \cdot \sum_i x_{j,i} \cdot n_i \]

With \( e_j \): energy per access to memory \( j \),
and \( x_{j,i} = 1 \) if object \( i \) is mapped to memory \( j \), =0 otherwise,
and \( n_i \): number of accesses to memory object \( i \),
subject to the constraints:

\[ \forall j : \sum_i x_{j,i} \cdot S_i \leq SSP_j \]

\[ \forall i : \sum_j x_{j,i} = 1 \]

With \( S_i \): size of memory object \( i \),
\( SSP_j \): size of memory \( j \).
Considered partitions

Example of considered memory partitions for a total capacity of 4096 bytes

<table>
<thead>
<tr>
<th># of partitions</th>
<th>number of partitions of size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4k</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Results for parts of GSM coder/decoder

A key advantage of partitioned scratchpads for multiple applications is their ability to adapt to the size of the current working set.
Worst/best case execution times (WCET/BCET)

Requirements on WCET estimates:
- **Safeness**: WCET \( \leq \text{WCET}_{\text{EST}} \)
- **Tightness**: \( \text{WCET}_{\text{EST}} - \text{WCET} \rightarrow \text{minimal} \)

**Scratch-pad/tightly coupled memory based predictability**

*Pre run-time scheduling* is often the only practical means of providing predictability in a complex system [Xu, Parnas].

- Time-triggered, statically scheduled operating systems
- Let's do the same for the memory system
  - Are SPMs really more timing predictable?
  - Analysis using the aiT timing analyzer
Architectures considered

ARM7TDI with 3 different memory architectures:

1. **Main memory**
   LDR-cycles: \((\text{CPU,IF,DF}) = (3,2,2)\)
   STR-cycles: \((2,2,2)\)
   \(* = (1,2,0)\)

2. **Main memory + unified cache**
   LDR-cycles: \((\text{CPU,IF,DF}) = (3,12,6)\)
   STR-cycles: \((2,12,3)\)
   \(* = (1,12,0)\)

3. **Main memory + scratch pad**
   LDR-cycles: \((\text{CPU,IF,DF}) = (3,0,2)\)
   STR-cycles: \((2,0,0)\)
   \(* = (1,0,0)\)

Results for G.721

Using Scratchpad:

Using Unified Cache:

References:

- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005
Tight integration of compilation and timing analysis

- Computation of the WCET after compilation does not give us optimum results
- Let’s optimize for the WCET during compilation
- Tight integration of aiT WCET analyzer from AbsInt into experimental WCET aware compiler WCC

\[ \text{WCET}_{\text{EST}} \] for g721 encoder

Steady WCET\text{\textsubscript{EST}} decreases for increasing SPM sizes
WCET\text{\textsubscript{EST}} reductions from 29\% – 48\%

X-Axis: SPM size = x\% of benchmark’s code size
Y-Axis: 100\% = WCET\text{\textsubscript{EST}} when not using SPM at all

H. Falk, J. Kleinsorge: Optimal Static WCET-aware Scratch-pad Allocation of Program Code, 46th Design Automation Conference (DAC), 2009
Average $\text{WCET}_{\text{EST}}$ for 73 Benchmarks

Steady $\text{WCET}_{\text{EST}}$ decreases for increasing SPM sizes
$\text{WCET}_{\text{EST}}$ reductions from 7% – 40%

X-Axis: SPM size = $x\%$ of benchmark’s code size
Y-Axis: 100% = $\text{WCET}_{\text{EST}}$ when not using SPM at all

Dynamic replacement within scratch pad

- Effectively results in a kind of compiler-controlled segmentation/paging for SPM
- Address assignment within SPM required (paging or segmentation-like)

Dynamic replacement of data within scratch pad: based on liveness analysis

MO = \{A, T1, T2, T3, T4\}
SP Size = |A| = |T1| \ldots = |T4|

SPILL_STORE(A);
SPILL_LOAD(T3);

Solution:
A \rightarrow SP & T3 \rightarrow SP

Dynamic replacement within scratch pad
- Results for edge detection relative to static allocation -

- Processor Energy
- Memory Energy
- Total Energy
- Execution Time

Scratchpad Size (Bytes)

- 147 -

- 148 -
References to large arrays (1)
- Regular accesses -

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      U[i][j]=U[i][j] + V[i][k] * W[k][j]

Tiling

for (it=0; it<n; it=it+Sb)
  {read_tile V[it:it+Sb-1, 1:n]}
  for (jt=0; jt<n; jt=jt+Sb)
    {read_tile U[it:it+Sb-1, jt:jt+Sb-1];
     read_tile W[1:n, jt:jt+Sb-1];
     U[it:it+Sb-1, jt:jt+Sb-1]=U[it:it+Sb-1, jt:jt+Sb-1]
      + V[it:it+Sb-1, 1:n]
      * W[1:n, it:it+Sb-1];
     write_tile U[it:it+Sb-1, jt:jt+Sb-1]}

References to large arrays
- Irregular accesses -

for each loop nest L in program P {
  apply loop tiling to L based on the access patterns of
  regular array references;
  for each assignment to index array X
    update the block minimum and maximum values of X;
    compute the set of array elements that are irregularly
    referenced in the current inter-tile iteration;
    compare the memory access costs for using
    and not using SPM;
  if (using SPM is beneficial)
    execute the intra-tile loop iterations by using the SPM
  else
    execute the intra-tile loop iterations by not
    using the SPM
}
Results for irregular approach

Cache

Dynamic  Static  Hybrid

Saving/Restoring Context Switch

Process P2

Scratchpad

Saving Context Switch (Saving)

- Utilizes SPM as a common region shared all processes
- Contents of processes are copied on/off the SPM at context switch
- Good for small scratchpads
Non-Saving Context Switch

- Partitions SPM into disjoint regions
- Each process is assigned a SPM region
- Copies contents during initialization
- Good for large scratchpads

Hybrid Context Switch

- Disjoint + Shared SPM regions
- Good for all scratchpads
- Analysis is similar to Non-Saving Approach
- Runtime: $O(nM^3)$
Multi-process Scratchpad Allocation: Results

- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

Dynamic set of multiple applications

Compile-time partitioning of SPM no longer feasible
- Introduction of SPM-manager
  - Runtime decisions, but compile-time supported

Approach overview

- 2 steps: compile-time analysis & runtime decisions
- No need to know all applications at compile-time
- Capable of managing runtime allocated memory objects
- Integrates into an embedded operating system

Using MPArm simulator from U. Bologna

Comparison of SPMM to Caches for SORT

- Baseline: Main memory only
- SPMM peak energy reduction by 83% at 4k Bytes scratchpad
- Cache peak: 75% at 2k 2-way cache
- SPMM outperforming caches
- OS and libraries are not considered yet

Chunk allocation results:

<table>
<thead>
<tr>
<th>SPM Size</th>
<th>Δ 4-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>74.81%</td>
</tr>
<tr>
<td>2048</td>
<td>65.35%</td>
</tr>
<tr>
<td>4096</td>
<td>64.39%</td>
</tr>
<tr>
<td>8192</td>
<td>65.64%</td>
</tr>
<tr>
<td>16384</td>
<td>63.73%</td>
</tr>
</tbody>
</table>
Research monographs


Textbook(s)

Several Editions:

- 1st English edition
  - Original hardcover version, Kluwer, 2003, >100 $/€
  - Reprint, lighter cover borders;
- 2nd English edition, 2010
- 1st German edition 29€
  - March 2007
  - Reprint, 2008
- Chinese edition, April 2007, only preface in Chinese, not for sale outside China
- Plans for Russian, Portuguese, Macedonian and Greek edition
- Video recordings will also be made available

Next short courses (simplified view)

- Specification
- Design repository
- Application mapping
- Optimization
- Evaluation (energy, cost, performance, ...)
- Validation & test
- Design

- ES-hardware
- System software (RTOS, middleware, ...)
- Almeida: Real-Time Communication in ES

- Buttazzo: Real-Time Operating Systems and Task Scheduling

- Nielsen: Modeling, verification and testing in embedded systems

*Not just hardware, but also protocols, timing analysis*
Overall Summary

- Introduction, Motivation and Overview
  - Motivation
  - Common characteristics
- Specifications and Modeling
  - Models of computation
  - Early phases
  - FSM-based models, Data flow, Petri nets, discrete event-based models, Von-Neumann models
  - Comparison
- Exploitation of the memory hierarchy
  - Scratch pad memories
    - Non-overlaying allocation
    - Overlaying allocation

Good night!