Hardware-Software Codesign for SoC

Part of the SoC Design Flow and Tools Course
Department of Computer Science & Information Engineering
National Chung Cheng University
Chiayi, Taiwan

Outline

- Introduction to Hardware-Software Codesign
- System Modeling, Architectures, Languages
- Partitioning Methods
- Design Quality Estimation
- Specification Refinement
- Co-synthesis Techniques
- Function-Architecture Codesign Paradigm
- Coverification Methodology & Tools
- Codesign Case Studies
  - ATM Virtual Private Network
  - Digital Camera and JPEG
**Classic Hardware/Software Design Process**

- Basic features of current process:
  - System immediately partitioned into hardware and software components
  - Hardware and software developed separately
  - “Hardware first” approach often adopted
- Implications of these features:
  - HW/SW trade-offs restricted
    - Impact of HW and SW on each other cannot be assessed easily
  - Late system integration
- Consequences of these features:
  - Poor quality designs
  - Costly modifications
  - Schedule slippages

**Misconceptions in Classic Hardware/Software Design Process**

- Hardware and software can be acquired separately and independently, with successful and easy integration of the two later
- Hardware problems can be fixed with simple software modifications
- Once operational, software rarely needs modification or maintenance
- Valid and complete software requirements are easy to state and implement in code
Codesign Definition and Key Concepts

- Co-design
  - The meeting of system-level objectives by exploiting the trade-offs between hardware and software in a system through their concurrent design

- Key concepts
  - Concurrent: hardware and software developed at the same time on parallel paths
  - Integrated: interaction between hardware and software developments to produce designs that meet performance criteria and functional specifications

Classic Design & Codesign

![Diagram showing classic design and co-design](image-url)
**Motivations for Codesign**

- Co-design helps meet time-to-market because developed software can be verified much earlier.
- Co-design improves overall system performance, reliability, and cost effectiveness because defects found in hardware can be corrected before tape-out.
- Co-design benefits the design of embedded systems and SoCs, which need HW/SW tailored for a particular application.
  - Faster integration: reduced design time and cost
  - Better integration: lower cost and better performance
  - Verified integration: lesser errors and re-spins

**Driving Factors for Codesign**

- Reusable Components
  - Instruction Set Processors
  - Embedded Software Components
  - Silicon Intellectual Properties
- Hardware-software trade-offs more feasible
  - Reconfigurable hardware (FPGA, CPLD)
  - Configurable processors (Tensilica, ARM, etc.)
- Transaction-Level Design and Verification
  - Peripheral and Bus Transactors (Bus Interface Models)
  - Transaction-level synthesis and verification tools
- Multi-million gate capacity in a single chip
- Software-rich chip systems
- Growing functional complexity
- Advances in computer-aided tools and technologies
  - Efficient C compilers for embedded processors
  - Efficient hardware synthesis capability
Codesign Applications

- Embedded Systems & SoC
  - Consumer electronics
  - Telecommunications
  - Manufacturing control
  - Vehicles

- Instruction Set Architectures
  - Application-specific instruction set processors.
    - Instructions of the CPU are also the target of design.
  - Dynamic compiler provides the tradeoff of HW/SW.

- Reconfigurable Systems

Categories of Codesign Problems

- Codesign of embedded systems
  - Usually consist of sensors, controller, and actuators
  - Are reactive systems
  - Usually have real-time constraints
  - Usually have dependability constraints

- Codesign of ISAs
  - Application-specific instruction set processors (ASIPs)
  - Compiler and hardware optimization and trade-offs

- Codesign of Reconfigurable Systems
  - Systems that can be personalized after manufacture for a specific application
  - Reconfiguration can be accomplished before execution or concurrent with execution (called evolvable systems)
The next design challenge: SoC

System-on-chip
- prefabricated components: IP cores
- great importance of software

How do you design such systems?

Typical Codesign Process
Codesign Process

- System specification
- HW/SW partitioning
  - Architectural assumptions:
    - Type of processor, interface style, etc.
  - Partitioning objectives:
    - Speedup, latency requirement, silicon size, cost, etc.
  - Partitioning strategies:
    - High-level partitioning by hand, computer-aided partitioning technique, etc.
- HW/SW synthesis
  - Operation scheduling in hardware
  - Instruction scheduling in compiler
  - Process scheduling in operating systems

Requirements for the Ideal Codesign Environment

- Unified, unbiased hardware/software representation
  - Supports uniform design and analysis techniques for hardware and software
  - Permits system evaluation in an integrated design environment
  - Allows easy migration of system tasks to either hardware or software
- Iterative partitioning techniques
  - Allow several different designs (HW/SW partitions) to be evaluated
  - Aid in determining best implementation for a system
  - Partitioning applied to modules to best meet design criteria (functionality and performance goals)
Requirements for the Ideal Codesign Environment (cont.)

- Integrated modeling substrate
  - Supports evaluation at several stages of the design process
  - Supports step-wise development and integration of hardware and software
- Validation Methodology
  - Insures that system implemented meets initial system requirements

Cross-fertilization Between Hardware and Software Design

- Fast growth in both VLSI design and software engineering has raised awareness of similarities between the two
  - Hardware synthesis
  - Programmable logic
  - Description languages
- Explicit attempts have been made to “transfer technology” between the domains
Conventional Codesign Methodology

Embedded System Design Process

Embedded System Design
Process
Key issues during the codesign process

- Unified representation
  - Models
  - Architectures
  - Languages

- HW/SW partitioning
  - Partition Algorithm
  - HW/SW Estimation Methods.

- HW/SW co-synthesis
  - Interface Synthesis
  - Refinement of Specification

Models, Architectures, Languages

- Introduction
- Models
  - State, Activity, Structure, Data, Heterogeneous
- Architectures
  - Function-Architecture, Platform-Based
- Languages
  - Hardware: VHDL / Verilog / SystemVerilog
  - Software: C / C++ / Java
  - System: SystemC / SLDL / SDL
  - Verification: PSL (Sugar, OVL)
Models, Architectures, Languages
INTRODUCTION

Design Methodologies

- Capture-and-Simulate
  - Schematic Capture
  - Simulation
- Describe-and-Synthesize
  - Hardware Description Language
  - Behavioral Synthesis
  - Logic Synthesis
- Specify-Explore-Refine
  - Executable Specification
  - Hardware-Software Partitioning
  - Estimation and Exploration
  - Specification Refinement
**Motivation**

Models are conceptual views of the system’s functionality
Architectures are abstract views of the system’s implementation

**Models & Architectures**

Models (Specification)
Architectures (Implementation)
Behavior Vs. Architecture

Performance models:
Emb. SW, comm. and comp. resources

Models of Computation

System Behavior
Behavior Simulation

System Architect

Mapping

Performance Simulation

Communication Refinement

Flow To Implementation

HW/SW partitioning, Scheduling

SW estimation

Synthesis

Models of an Elevator Controller

"If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle. If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor. If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor."

(a) English description

(b) Algorithmic model

(c) State-machine model
Architectures Implementing the Elevator Controller

Current Abstraction Mechanisms in Hardware Systems

**Abstraction**

The level of detail contained within the system model

- A system can be modeled at
  - System Level,
  - Algorithm or Instruction Set Level,
  - Register-Transfer Level (RTL),
  - Logic or Gate Level,
  - Circuit or Schematic Level.

- A model can describe a system in the
  - Behavioral domain,
  - Structural domain,
  - Physical domain.
Abstractions in Modeling: Hardware Systems

<table>
<thead>
<tr>
<th>Level</th>
<th>Behavior</th>
<th>Structure</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS (System)</td>
<td>Communicating</td>
<td>Processors</td>
<td>Cabinets, Cables</td>
</tr>
<tr>
<td></td>
<td>Processes</td>
<td>Memories</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Switches (PMS)</td>
<td></td>
</tr>
<tr>
<td>Instruction Set (Algorithm)</td>
<td>Input-Output</td>
<td>Memory, Ports</td>
<td>Board Floorplan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processors</td>
<td></td>
</tr>
<tr>
<td>Register-Transfer</td>
<td>Register Transfers</td>
<td>ALUs, Regs, Muxes, Bus</td>
<td>ICs Macro Cells</td>
</tr>
<tr>
<td>Logic</td>
<td>Logic Eqns.</td>
<td>Gates, Flip-flops</td>
<td>Std. cell layout</td>
</tr>
<tr>
<td>Circuit</td>
<td>Network Eqns.</td>
<td>Trans., Connections</td>
<td>Transistor layout</td>
</tr>
</tbody>
</table>

Current Abstraction Mechanisms for Software Systems

Virtual Machine
A software layer very close to the hardware that hides the hardware’s details and provides an abstract and portable view to the application programmer

Attributes
- Developer can treat it as the real machine
- A convenient set of instructions can be used by developer to model system
- Certain design decisions are hidden from the programmer
- Operating systems are often viewed as virtual machines
Abstractions for Software Systems

Virtual Machine Hierarchy
- Application Programs
- Utility Programs
- Operating System
- Monitor
- Machine Language
- Microcode
- Logic Devices

MODELS
**Unified HW/SW Representation**

- Unified Representation –
  - High-level system (SoC) architecture description
  - Implementation (hardware or software) independent
  - Delayed hardware/software partitioning
  - Cross-fertilization between hardware and software
  - Co-simulation environment for communication
  - System-level functional verification

---

**Abstract Hardware-Software Model**

Unified representation of system allows early performance analysis

[Diagram showing the flow of General Performance Evaluation, Identification of Bottlenecks, Abstract HW/SW Model, Evaluation of Design Alternatives, and Evaluation of HW/SW Trade-offs]
**HW/SW System Models**

- **State-Oriented Models**
  - Finite-State Machines (FSM), Petri-Nets (PN), Hierarchical Concurrent FSM
- **Activity-Oriented Models**
  - Data Flow Graph, Flow-Chart
- **Structure-Oriented Models**
  - Block Diagram, RT netlist, Gate netlist
- **Data-Oriented Models**
  - Entity-Relationship Diagram, Jackson’s Diagram
- **Heterogeneous Models**
  - UML (OO), CDFG, PSM, Queuing Model, Programming Language Paradigm, Structure Chart

**State-Oriented: Finite-State Machine (Mealy Model)**

![State-Oriented: Finite-State Machine (Mealy Model)](image)
State-Oriented: Finite State Machine (Moore Model)

State-Oriented: Finite State Machine with Datapath
Finite State Machines

- **Merits**
  - Represent system's temporal behavior explicitly
  - Suitable for control-dominated systems
  - Suitable for formal verification

- **Demerits**
  - Lack of hierarchy and concurrency
  - State or arc explosion when representing complex systems

State-Oriented: Petri Nets

- System model consisting of places, tokens, Petri Nets: transitions, arcs, and a marking
  - Places - equivalent to conditions and hold tokens
  - Tokens - represent information flow through system
  - Transitions - associated with events, a "firing" of a transition indicates that some event has occurred
  - Marking - a particular placement of tokens within places of a Petri net, representing the state of the net

Example:
State-Oriented: Petri Nets

- **Merits**
  - Good at modeling and analyzing concurrent systems
  - Extensive theoretical and experimental works
  - Used extensively for protocol engineering and control system modeling

- **Demerits**
  - “Flat Model” that becomes incomprehensible when system complexity increases

Petri Nets
**State-Oriented: Hierarchical Concurrent FSM**

- **Merits**
  - Support both hierarchy and concurrency
  - Good for representing complex systems

- **Demerits**
  - Concentrate only on modeling control aspects and not data and activities
**Activity-Oriented: Data Flow Graphs (DFG)**

![Diagram of Data Flow Graphs](image)

**Data Flow Graphs**

- **Merits**
  - Support hierarchy
  - Suitable for specifying complex transformational systems
  - Represent problem-inherent data dependencies

- **Demerits**
  - Do not express control sequencing or temporal behaviors
  - Weak for modeling embedded systems
Activity-Oriented: Flow Charts

Flow Charts

- Merits
  - Useful to represent tasks governed by control flows
  - Can impose an order to supersede natural data dependencies

- Demerits
  - Used only when the system’s computation is well known
Structure-Oriented: Component Connectivity Diagrams

- Merits
  - Good at representing system's structure

- Demerits
  - Behavior is not explicit

- Characteristics
  - Used in later phases of design
Data-Oriented: Entity-Relationship Diagram

Entity-Relationship Diagrams

- **Merits**
  - Provide a good view of the data in a system
  - Suitable for representing complex relationships among various kinds of data

- **Demerits**
  - Do not describe any functional or temporal behavior of a system
Data-Oriented: Jackson’s Diagram

Jackson’s Diagrams

- **Merits**
  - Suitable for representing data having a complex composite structure

- **Demerits**
  - Do not describe any functional or temporal behavior of the system
**Heterogeneous: Control/Data Flow Graphs (CDFG)**

- Graphs contain nodes corresponding to operations in either hardware or software
- Often used in high-level hardware synthesis
- Can easily model data flow, control steps, and concurrent operations because of its graphical nature

Example:

```
5 + X 4 + Y
   +  --->
     +  
```

**Control/Data Flow Graphs**

- **Merits**
  - Correct the inability to represent control dependencies in DFG
  - Correct the inability to represent data dependencies in CFG

- **Demerits**
  - Low level specification (behavior not evident)
Heterogeneous: Structure Chart

Structure Charts

- Merits
  - Represent both data and control

- Demerits
  - Used in the preliminary stages of system design
Heterogeneous: Object-Oriented Paradigms (UML, …)

- Use techniques previously applied to software to manage complexity and change in hardware modeling
- Use OO concepts such as
  - Data abstraction
  - Information hiding
  - Inheritance
- Use building block approach to gain OO benefits
  - Higher component reuse
  - Lower design cost
  - Faster system design process
  - Increased reliability

Object-Oriented Representation

Example:

3 Levels of abstraction:
Object-Oriented Paradigms

- **Merits**
  - Support information hiding
  - Support inheritance
  - Support natural concurrency

- **Demerits**
  - Not suitable for systems with complicated transformation functions
**Heterogeneous:**
*Program State Machine (PSM)*

**Program State Machine**

- **Merits**
  - Represent a system's state, data, control, and activities in a single model
  - Overcome the limitations of programming languages and HCFSM models
Heterogeneous: Queuing Model

Queuing Models

- Characteristics
  - Used for analyzing system’s performance
  - Can find utilization, queuing length, throughput, etc.
**Codesign Finite State Machine (CFSM)**

- CFSM is FSM extended with
  - Support for data handling
  - Asynchronous communication

- CFSM has:
  - FSM part
    - Inputs, outputs, states, transition and output relation
  - Data computation part
    - External, instantaneous functions

**Codesign Finite State Machine (CFSM)**

- CFSM has:
  - Locally synchronous behavior
    - CFSM executes based on snapshot input assignment
    - Synchronous from its own perspective
  - Globally asynchronous behavior
    - CFSM executes in non-zero, finite amount of time
    - Asynchronous from system perspective

- GALS model
  - Globally: Scheduling mechanism
  - Locally: CFSMs
Network of CFSMs: Depth-1 Buffers

- Globally Asynchronous, Locally Synchronous (GALS) model

Typical DSP Algorithm

- Traditional DSP
  - Convolution/Correlation
  - Filtering (FIR, IIR)
  \[
  y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k]
  \]
  - Adaptive Filtering (Varying Coefficient)
  \[
  y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M-1} b_k x[n]
  \]
  - DCT
  \[
  x[k] = c(k) \sum_{n=0}^{N-1} x[n] \cos\left(\frac{(2n+1)k\pi}{2N}\right)
  \]
Specification of DSP Algorithms

- Example
  \[ y(n) = a \cdot x(n) + b \cdot x(n-1) + c \cdot x(n-2) \]

- Graphical Representation Method 1: Block Diagram (Data-path architecture)
  - Consists of functional blocks connected with directed edges, which represent data flow from its input block to its output block

Graphical Representation Method 2: Signal-Flow Graph

- SFG: a collection of nodes and directed edges
- Nodes: represent computations and/or task, sum all incoming signals
- Directed edge (j, k): denotes a linear transformation from the input signal at node j to the output signal at node k
- Linear SFGs can be transformed into different forms without changing the system functions.
  - Flow graph reversal or transposition is one of these transformations (Note: only applicable to single-input-single-output systems)
**Signal-Flow Graph**

- Usually used for linear time-invariant DSP systems representation
- Example:

```
<table>
<thead>
<tr>
<th>x(n)</th>
<th>z^{-1}</th>
<th>y(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
```

**Graphical Representation Method 3: Data-Flow Graph**

- DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.
- DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.

```
x(n)   |
       |
D      |
       |
       |
D      |
       |
       |
y(n)   |
```

a   b   c
**Data-Flow Graph**

- Each edge describes a precedence constraint between two nodes in DFG:
  - Intra-iteration precedence constraint: if the edge has zero delays
  - Inter-iteration precedence constraint: if the edge has one or more delays (Delay here represents iteration delays.)
- DFGs and Block Diagrams can be used to describe both linear single-rate and nonlinear multi-rate DSP systems

**Fine-Grain DFG**

![Fine-Grain DFG Diagram]

**Examples of DFG**

- Nodes are complex blocks (in Coarse-Grain DFGs)

![Examples of DFG Diagram]

- Nodes can describe expanders/decimators in Multi-Rate DFGs
Graphical Representation Method 4: Dependence Graph

- DG contains computations for all iterations in an algorithm.
- DG does not contain delay elements.
- Edges represent precedence constraints among nodes.
- Mainly used for systolic array design.
"You hear the planned possibilities, but it is nothing like the visual concept of a model. You get the impact of the complete vision."

Galyn Fish
Director of PR, Southwest MedicalCenter

**System Level Design Science**

- **Design Methodology:**
  - **Top Down Aspect:**
    - Orthogonalization of Concerns:
      - Separate Implementation from Conceptual Aspects
      - Separate computation from communication
    - Formalization: precise unambiguous semantics
    - Abstraction: capture the desired system details (do not overspecify)
    - Decomposition: partitioning the system behavior into simpler behaviors
    - Successive Refinements: refine the abstraction level down to the implementation by filling in details and passing constraints
  - **Bottom Up Aspect:**
    - IP Re-use (even at the algorithmic and functional level)
    - Components of architecture from pre-existing library
Separate Behavior from Micro-architecture

- System Behavior
  - Functional specification of system
  - No notion of hardware or software!

- Implementation Architecture
  - Hardware and Software
  - Optimized Computer

Example of System Behavior
IP-Based Design of the System Behavior

The next level of Abstraction ...
IP-Based Design of the Implementation

Which Bus? PI? AMBA? Dedicated Bus for DSP?

Which DSP Processor? C50? Can DSP be done on Microcontroller?

Which Microcontroller? ARM? HC11?

How fast will my User Interface Software run? How Much can I fit on my Microcontroller?

Can I Buy an MPEG2 Processor? Which One?

Do I need a dedicated Audio Decoder? Can decode be done on Microcontroller?

Architectural Choices

Flexible Logic

Dedicated Hardware

General Purpose Processor

Programmable DSP

1/Efficiency (power, speed)
Map Between Behavior and Architecture

Transport Decode Implemented as Software Task Running on Microcontroller

Audio Decode Behavior Implemented on Dedicated Hardware

Classic A/D, HW/SW tradeoff

- RF Front End
- Can trade custom analog for hardware, even for software
  - Power, area critical criteria, or easy functional modification
Example: Voice Mail Pager

- Design considerations cross design layers
- Trade-offs require systematic methodology and constraint-based hierarchical approach for clear justification

Where All is Going

◆ Create paradigm shift - not just link methods
  ● New levels of abstraction to fluidly tradeoff HW/SW, A/D, HF/IF, interfaces, etc. to exploit heterogeneous nature of components
  ● Links already being forged
**Deep Submicron Paradigm Shift**

- **2M Transistors**
- **100M Metal**
- **100 MHz**
- **Wire RC 1 ns/cm**

- **40M Transistors**
- **2,000M Metal**
- **600 MHz**
- **Wire RC 6 ns/cm**

**Cell Based Design**
- Minimize Area
- Maximize Performance
- Optimize Gate Level

**Virtual Component Based Design**
- Minimize Design Time
- Maximize IP Reuse
- Optimize System Level

**Implementation Design Trends**

- **Flat ASIC**
- **Flat ASIC+**
- **Virtual Component Based Design**
- **Platform Based**
  - Consumer
  - Wireless
  - Automotive
- **Hierarchical**
  - Microprocessors
  - High end servers & W/S
- **Flat Layout**
  - Net & Compute
  - Servers
  - Base stations
**Platform-Based System Architecture Exploration**

**Function**
- HW/SW
- Architecture
- RTL - SW
- Mask - ASM

**Effort/Value**
- Today
- Tomorrow

**Application Space**
- System Platform
- Architectural Space

**Digital Wireless Platform**

**Source:** Berkeley Wireless Research Center
**Will the system solution match the original system spec?**

- Limited synergies between HW & SW teams
- Long complex flows in which teams do not reconcile efforts until the end
- High degree of risk that devices will be fully functional

**EDA Challenge to Close the Gap**

- Industry averaging 2-3 iterations SoC design
- Need to identify design issues earlier
- Gap between concept and logical / Physical implementation

**Source:** GSRC
AMBA-Based SoC Architecture

LANGUAGES
Languages

- Hardware Description Languages
  - VHDL / Verilog / SystemVerilog
- Software Programming Languages
  - C / C++ / Java
- Architecture Description Languages
  - EXPRESSION / MIMOLA / LISA
- System Specification Languages
  - SystemC / SLDL / SDL / Esterel
- Verification Languages
  - PSL (Sugar, OVL) / OpenVERA

Hardware Description Languages (HDL)

- VHDL (IEEE 1076)
- Verilog 1.0 (IEEE 1364)
- Verilog 2.0 (IEEE 1364)
- SystemVerilog 3.0 (to be sent for IEEE review)
- SystemVerilog 3.1 (to be sent for IEEE review)
SystemVerilog 3.0

- Built-in C types
- Synthesis improvements (avoid simulation and synthesis mismatches)
- Enhanced design verification
  - procedural assertions
- Improved modeling
  - communication interfaces

SystemVerilog 3.1

- Testbench automation
  - Data structures
  - Classes
  - Inter-process communication
  - Randomization
- Temporal assertions
  - Monitors
  - Coverage Analysis
SystemVerilog Environment for Ethernet MAC

Architecture Description Language in SOC Codesign Flow
Objectives for Embedded SOC

- Support automated SW toolkit generation
  - exploration quality SW tools (performance estimator, profiler, …)
  - production quality SW tools (cycle-accurate simulator, memory-aware compiler..)
- Specify a variety of architecture classes (VLIWs, DSP, RISC, ASIPs…)
- Specify novel memory organizations
- Specify pipelining and resource constraints

Architecture Description Languages

- Behavior-Centric ADLs *(primarily capture Instruction Set (IS))*
  - ISPS, nML, ISDL, SCP/ValenC, ...
  - good for regular architectures, provides programmer’s view
  - tedious for irregular architectures, hard to specify pipelining, implicit arch model
- Structure-Centric ADLs *(primarily capture architectural structure)*
  - MIMOLA, ...
  - can drive code generation and architecture synthesis, can specify detailed pipelining
  - hard to extract IS view
- Mixed-Level ADLs *(combine benefits of both)*
  - LISA, RADL, FLEXWARE, MDes, ...
  - contains detailed pipelining information
  - most are specific to single processor class and/or memory architecture
  - most generate either simulator or compiler but not both
**EXPRESSION ADL**

**SystemC History**

1991

**Frontier Design A/RT Library**

**Synopsys “Fridge”**

**SystemC v1.0**

Apr. 00

1997

**CoWare “N2C”**

Abstract Protocols

Jun. 00

VSIA SLD

Data Types

Spec (draft)

1996

**Synopsys ATG**

**“Scenic”**

SystemC v0.90

Sep. 99

1992

**imec**

1991

1997

1990

**UC Irvine**

**Synopsys “Fridge”**

**SystemC v1.1**
**SystemC Highlights**

- Features as a codesign language
  - Modules
  - Processes
  - Ports
  - Signals
  - Rich set of port and signal types
  - Rich set of data types
  - Clocks
  - Cycle-based simulation
  - Multiple abstraction levels
  - Communication protocols
  - Debugging support
  - Waveform tracing

---

**Current System Design Methodology**

1. **C/C++ System Level Model**
2. **Analysis**
3. **Results**
4. **Manual Conversion**
5. **VHDL/Verilog**
6. **Simulation**
7. **Synthesis**
8. **Rest of Process**
Current System Design Methodology (cont’d)

- Problems
  - Errors in manual conversion from C to HDL
  - Disconnect between system model and HDL model
  - Multiple system tests

SystemC Design Methodology

- SystemC Model
  - Simulation
    - Refinement
    - Synthesis
      - Rest of Process
SystemC Design Methodology (cont’d)

- Advantages
  - Refinement methodology
  - Written in a single language
  - Higher productivity
  - Reusable testbenches

SystemC Programming Model

- A set of modules interacting through signals.
- Module functionality is described by processes.
SystemC programming model (cont’d)

- System (program) debug/validation
  - Testbench
    - Simulation, Waveform view of signals
  - Normal C++ IDE facilities
    - Watch, Evaluate, Breakpoint, ...

- `sc_main()` function
  - instantiates all modules
  - initializes clocks
  - initializes output waveform files
  - starts simulation kernel

A Simple Example: Defining a Module

- Complex-number Multiplier
  \[(a+bi)(c+di) = (ac-bd)+(ad+bc)i\]

```cpp
SC_MODULE(cmplx_mult) {
    sc_in<int> a,b;
    sc_in<int> c,d;
    sc_out<int> e,f;
    ...  
}
```
A Simple Example: Defining a Module (cont’d)

```cpp
SC_MODULE(cmplx_mult) {
    sc_in<int> a, b;
    sc_in<int> c, d;
    sc_out<int> e, f;
    void calc();
    SC_CTOR(cmplx_mult) {
        SC_METHOD(calc);
        sensitive<<a<<b<<c<<d;
    }
}

void cmplx_mult::calc() {
    e = a*c - b*d;
    f = a*d + b*c;
}
```

Completing the Design
Completing the Design: input_gen module

```cpp
SC_MODULE(input_gen) {
  sc_in<bool> clk;
  sc_out<int> a,b;
  sc_out<int> c,d;
  void generate();
  SC_CTOR(input_gen) {
    SC_THREAD(generate);
    sensitive_pos(clk);
  }
}

void input_gen::generate()
{
  int a_val=0, c_val=0;
  while (true) {
    a = a_val++;
    wait();
    c = (c_val+=2);
    wait();
  }
}
```

Completing the Design: display module

```cpp
SC_MODULE(display) {
  sc_in<int> e,f;
  void show();
  SC_CTOR(display) {
    SC_METHOD(show);
    sensitive<<e<<f;
  }
}

void display::show()
{
  cout<<e<<'+'<f<<"\n"
};
```
## Putting it all together:
### sc_main function

```c
#include <systemc.h>
int sc_main()
{
    input_gen M1("I_G");
    cmplx_mult M2("C_M");
    display M3("D");
    sc_signal<int>
        a,b,c,d,e,f;
    sc_clock
        clk("clk",20,0.5);
    M1.clk(clk.signal());
    M1.a(a); M1.b(b);
    M1.c(c); M1.d(d);
    M2.a(a); M2.b(b);
    M2.c(c); M2.d(d);
    M2.e(e); M2.f(f);
    M3.e(e); M3.f(f);
    sc_start(100);
    return 0;
}
```

### Vertical Testbench Reuse

- Use Case Model
- Algorithmic Modelling
- SystemC Modelling
- Implementation
- Prototype Tests

A Design Methodology for the Development of a Complex SoC: Aiding UML and Executable System Models, © 2004 – ST Microelectronics
Front-end system design flow

- Specification document
  - UML model (analysis and implementation)
  - SystemC specification for synthesis
  - Design space exploration
  - SystemC hw spec
  - C++ sw spec
  - CoCentric System Studio

Architecture

- Module A
  - Adapter
  - Slave Comm. class
  - Slave Comm. class
  - Slave Comm. class
  - Master Comm. class
  - Shared memory
    - Driver
    - Libraries
  - SystemC world

- Module B
  - Adapter
  - Slave Comm. class
  - Slave Comm. class
  - Slave Comm. class
  - Master Comm. class
  - Shared memory
    - Driver
    - Libraries
  - HDL RTL world

Other Modules

- VHDL
  - Verilog Simulator
  - Cadence Toolshed™ Library
  - ORLibrary
  - Slave TVM
  - Slave TVM
  - Slave TVM
  - Master Comm. class
  - UNX external memory
  - Shared memory

- HDL RTL world
**Property Specification Language (PSL)**

- Accellera: a non-profit organization for standardization of design & verification languages
- PSL = IBM Sugar + Verplex OVL
- System Properties
  - Temporal Logic for Formal Verification
- Design Assertions
  - Procedural (like SystemVerilog assertions)
  - Declarative (like OVL assertion monitors)
- For:
  - Simulation-based Verification
  - Static Formal Verification
  - Dynamic Formal Verification

**System Partitioning**

- System functionality is implemented on system components
  - ASICs, processors, memories, buses
- Two design tasks:
  - Allocate system components or ASIC constraints
  - Partition functionality among components
- Constraints
  - Cost, performance, size, power
- Partitioning is a central system design task
Hardware/Software Partitioning

• Informal Definition
  – The process of deciding, for each subsystem, whether the required functionality is more advantageously implemented in hardware or software

• Goal
  – To achieve a partition that will give us the required performance within the overall system requirements (in size, weight, power, cost, etc.)

• This is a multivariate optimization problem that when automated, is an NP-hard problem

HW/SW Partitioning
Formal Definition

• A hardware/software partition is defined using two sets $H$ and $S$, where $H \subset O$, $S \subset O$, $H \cup S = O$, $H \cap S = \emptyset$

• Associated metrics:
  – $\text{Hsize}(H)$ is the size of the hardware needed to implement the functions in $H$ (e.g., number of transistors)
  – $\text{Performance}(G)$ is the total execution time for the group of functions in $G$ for a given partition $\{H, S\}$
  – Set of performance constraints, $\text{Cons} = (C_1, \ldots, C_m)$, where $C_j = \{G, \text{timecon}\}$, indicates the maximum execution time allowed for all the functions in group $G$ and $G \subset O$
**Performance Satisfying Partition**

- A performance satisfying partition is one for which performance($C_j . G$) ≤ $C_j . timecon$, for all $j=1…m$

- Given $O$ and Cons, the hardware/software partitioning problem is to find a performance satisfying partition $\{H,S\}$ such that Hsize($H$) is minimized

- The *all-hardware* size of $O$ is defined as the size of an all hardware partition (i.e., Hsize($O$))

---

**HW/SW Partitioning Issues**

- Partitioning into hardware and software affects overall system cost and performance

- Hardware implementation
  - Provides higher performance via hardware speeds and parallel execution of operations
  - Incurs additional expense of fabricating ASICs

- Software implementation
  - May run on high-performance processors at low cost (due to high-volume production)
  - Incurs high cost of developing and maintaining (complex) software
Structural vs. Functional Partitioning

- Structural: Implement structure, then partition
  - Good for the hardware (size & pin) estimation.
  - Size/performance tradeoffs are difficult.
  - Suffer for large possible number of objects.
  - Difficult for HW/SW tradeoff.

- Functional: Partition function, then implement
  - Enables better size/performance tradeoffs
  - Uses fewer objects, better for algorithms/humans
  - Permits hardware/software solutions
  - But, it’s harder than graph partitioning

Partitioning Approaches

- Start with all functionality in software and move portions into hardware which are time-critical and can not be allocated to software
  (software-oriented partitioning)

- Start with all functionality in hardware and move portions into software implementation
  (hardware-oriented partitioning)
System Partitioning (Functional Partitioning)

- System partitioning in the context of hardware/software codesign is also referred to as functional partitioning.
- Partitioning functional objects among system components is done as follows:
  - The system’s functionality is described as a collection of indivisible functional objects.
  - Each system component’s functionality is implemented in either hardware or software.
- An important advantage of functional partitioning is that it allows hardware/software solutions.

Binding Software to Hardware

- Binding: assigning software to hardware components.
- After parallel implementation of assigned modules, all design threads are joined for system integration:
  - Early binding commits a design process to a certain course.
  - Late binding, on the other hand, provides greater flexibility for last minute changes.
Hardware/Software System Architecture Trends

- Some operations in special-purpose hardware
  - Generally take the form of a coprocessor communicating with the CPU over its bus
    - Computation must be long enough to compensate for the communication overhead
  - May be implemented totally in hardware to avoid instruction interpretation overhead
    - Utilize high-level synthesis algorithms to generate a register transfer implementation from a behavior description
- Partitioning algorithms are closely related to the process scheduling model used for the software side of the implementation

Basic partitioning issues

- Specification-abstraction level: input definition
  - Executable languages becoming a requirement
    - Although natural languages common in practice.
  - Just indicating the language is insufficient
  - Abstraction-level indicates amount of design already done
    - e.g. task DFG, tasks, CDFG, FSMD
- Granularity: specification size in each object
  - Fine granularity yields more possible designs
  - Coarse granularity better for computation, designer interaction
    - e.g. tasks, procedures, statement blocks, statements
- Component allocation: types and numbers
  - e.g. ASICs, processors, memories, buses
Basic partitioning issues (cont.)

- Metrics and estimations: "good" partition attributes
  - e.g. cost, speed, power, size, pins, testability, reliability
  - Estimates derived from quick, rough implementation
  - Speed and accuracy are competing goals of estimation

- Objective and closeness functions
  - Combines multiple metric values
  - Closeness used for grouping before complete partition
  - Weighted sum common
  - e.g. \( k_1F(\text{area}, c) + k_2F(\text{delay}, c) + k_3F(\text{power}, c) \)

- Output: format and uses
  - e.g. new specification, hints to synthesis tool

- Flow of control and designer interaction

---

Issues in Partitioning (Cont.)

- High Level Abstraction
- Decomposition of functional objects
  - Metrics and estimations
  - Partitioning algorithms
  - Objective and closeness functions
- Component allocation
- Output
Specification Abstraction Levels

- Task-level dataflow graph
  - A Dataflow graph where each operation represents a task
- Task
  - Each task is described as a sequential program
- Arithmetic-level dataflow graph
  - A Dataflow graph of arithmetic operations along with some control operations
  - The most common model used in the partitioning techniques
- Finite state machine (FSM) with datapath
  - A finite state machine, with possibly complex expressions being computed in a state or during a transition

Specification Abstraction Levels (Cont.)

- Register transfers
  - The transfers between registers for each machine state are described
- Structure
  - A structural interconnection of physical components
  - Often called a net-list
**Granularity Issues in Partitioning**

- The granularity of the decomposition is a measure of the size of the specification in each object.
- The specification is first decomposed into functional objects, which are then partitioned among system components.
  - Coarse granularity means that each object contains a large amount of the specification.
  - Fine granularity means that each object contains only a small amount of the specification.
    - Many more objects
    - More possible partitions
      - Better optimizations can be achieved

**System Component Allocation**

- The process of choosing system component types from among those allowed, and selecting a number of each to use in a given design.
- The set of selected components is called an allocation.
  - Various allocations can be used to implement a specification, each differing primarily in monetary cost and performance.
  - Allocation is typically done manually or in conjunction with a partitioning algorithm.
- A partitioning technique must designate the types of system components to which functional objects can be mapped.
  - ASICs, memories, etc.
Metrics and Estimations Issues

- A technique must define the attributes of a partition that determine its quality
  - Such attributes are called metrics
    - Examples include monetary cost, execution time, communication bit-rates, power consumption, area, pins, testability, reliability, program size, data size, and memory size
    - Closeness metrics are used to predict the benefit of grouping any two objects

- Need to compute a metric’s value
  - Because all metrics are defined in terms of the structure (or software) that implements the functional objects, it is difficult to compute costs as no such implementation exists during partitioning

Metrics in HW/SW Partitioning

- Two key metrics are used in hardware/software partitioning
  - Performance: Generally improved by moving objects to hardware
  - Hardware size: Hardware size is generally improved by moving objects out of hardware
Computation of Metrics

- Two approaches to computing metrics
  - Creating a detailed implementation
    - Produces accurate metric values
    - Impractical as it requires too much time
  - Creating a rough implementation
    - Includes the major register transfer components of a design
    - Skips details such as precise routing or optimized logic, which require much design time
    - Determining metric values from a rough implementation is called estimation

Estimation of Partitioning Metrics

- Deterministic estimation techniques
  - Can be used only with a fully specified model with all data dependencies removed and all component costs known
  - Result in very good partitions
- Statistical estimation techniques
  - Used when the model is not fully specified
  - Based on the analysis of similar systems and certain design parameters
- Profiling techniques
  - Examine control flow and data flow within an architecture to determine computationally expensive parts which are better realized in hardware
**Objective and Closeness Functions**

- Multiple metrics, such as cost, power, and performance are weighed against one another
  - An expression combining multiple metric values into a single value that defines the quality of a partition is called an **Objective Function**
  - The value returned by such a function is called **cost**
  - Because many metrics may be of varying importance, a weighted sum objective function is used
    - e.g., \( \text{Objfct} = k_1 \cdot \text{area} + k_2 \cdot \text{delay} + k_3 \cdot \text{power} \)
  - Because constraints always exist on each design, they must be taken into account
    - e.g. \( \text{Objfct} = k_1 \cdot F(\text{area, area_constr}) + k_2 \cdot F(\text{delay, delay_constr}) + k_3 \cdot F(\text{power, power_constr}) \)

**Partitioning Algorithm Issues**

- Given a set of functional objects and a set of system components, a partitioning algorithm searches for the best partition, which is the one with the lowest cost, as computed by an objective function
- While the best partition can be found through exhaustive search, this method is impractical because of the inordinate amount of computation and time required
- The essence of a partitioning algorithm is the manner in which it chooses the subset of all possible partitions to examine
Partitioning Algorithm Classes

- **Constructive algorithms**
  - Group objects into a complete partition
  - Use closeness metrics to group objects, hoping for a good partition
  - Spend computation time constructing a small number of partitions
- **Iterative algorithms**
  - Modify a complete partition in the hope that such modifications will improve the partition
  - Use an objective function to evaluate each partition
  - Yield more accurate evaluations than closeness functions used by constructive algorithms
- In practice, a combination of constructive and iterative algorithms is often employed

Iterative Partitioning Algorithms

- The computation time in an iterative algorithm is spent evaluating large numbers of partitions
- Iterative algorithms differ from one another primarily in the ways in which they modify the partition and in which they accept or reject bad modifications
- The goal is to find global minimum while performing as little computation as possible
Iterative Partitioning Algorithms (Cont.)

- Greedy algorithms
  - Only accept moves that decrease cost
  - Can get trapped in local minima

- Hill-climbing algorithms
  - Allow moves in directions increasing cost (retracing)
    - Through use of stochastic functions
  - Can escape local minima
  - E.g., simulated annealing

Typical partitioning-system configuration
Basic partitioning algorithms

- Random mapping
  - Only used for the creation of the initial partition.
- Clustering and multi-stage clustering
- Group migration (a.k.a. min-cut or Kernighan/Lin)
- Ratio cut
- Simulated annealing
- Genetic evolution
- Integer linear programming

Hierarchical clustering

- One of constructive algorithm based on closeness metrics to group objects

- Fundamental steps:
  - Groups closest objects
  - Recompute closenesses
  - Repeat until termination condition met

- Cluster tree maintains history of merges
  - Cutline across the tree defines a partition
Hierarchical clustering algorithm

/* Initialize each object as a group */
for each cl loop
  pi=oi
  P=P∪pi
end loop

/* Compute closenesses between objects */
for each pi loop
  for each pj loop
    ci,j=ComputeCloseness(pi,pj)
  end loop
end loop

/* Merge closest objects and recompute closenesses */
While not Terminate(P) loop
  pi,pj=FindClosestObjects(P,C)
  P=P−pi−pj∪pij
  for each pk loop
    ci,j,k=ComputeCloseness(pij,pk)
  end loop
end loop
return P

Hierarchical clustering example
**Greedy partitioning for HW/SW partition**

- Two-way partition algorithm between the groups of HW and SW.
- Suffer from local minimum problem.

Repeat
  \[ P_{\text{orig}} = P \]
  for \( i \) in 1 to \( n \) loop
    if \( \text{Objfct}(\text{Move}(P,o)) < \text{Objfct}(P) \) then
      \( P = \text{Move}(P,o) \)
    end if
  end loop
Until \( P = P_{\text{orig}} \)

**Multi-stage clustering**

- Start hierarchical clustering with one metric and then continue with another metric.
- Each clustering with a particular metric is called a stage.
**Group migration**

- Another iteration improvement algorithm extended from two-way partitioning algorithm that suffer from local minimum problem.
- The movement of objects between groups depends on if it produces the greatest decrease or the smallest increase in cost.
  - To prevent an infinite loop in the algorithm, each object can only be moved once.

**Group migration’s Algorithm**

```
P=P_in
Loop
  /*Initialize*/
  prev_P=P
  prev_cost=Objfct(P)
  bestpart_cost=∞
  for each o, loop
    o.moved=false
  end loop
  /*create a sequence of n moves*/
  for i in 1 to n loop
    bestmove_cost=∞
    for each o not o.moved loop
      cost=Objfct(Move(P, o))
      if cost<bestmove_cost then
        bestmove_cost=cost
        bestmove_obj= o
      end if
    end loop
    P=Move(P,bestmove_obj)
    bestmove_obj.moved=true
  end loop
  /*Save the best partition during the sequence*/
  if bestmove_cost<bestpart_cost then
    bestpart_P=P
    bestpart_cost=bestmove_cost
  end if
end loop
/*Update P if a better cost was found, else exit*/
If bestpart_cost<prev_cost then
  P=bestpart_P
else return prev_P
end if
```
**Ratio Cut**

- A constructive algorithm that groups objects until a terminal condition has been met.
- A new metric \( \text{ratio} \) is defined as
  \[
  \text{ratio} = \frac{\text{cut}(P)}{\text{size}(p_1) \times \text{size}(p_2)}
  \]
  - \( \text{Cut}(P) \): sum of the weights of the edges that cross \( p_1 \) and \( p_2 \).
  - \( \text{Size}(p_i) \): size of \( p_i \).
- The ratio metric balances the competing goals of grouping objects to reduce the cutsize without grouping distance objects.
- Based on this new metric, the partition algorithms try to group objects to reduce the cutsizes without grouping objects that are not close.

**Simulated annealing**

- Iterative algorithm modeled after physical annealing process that to avoid local minimum problem.
- Overview
  - Starts with initial partition and temperature
  - Slowly decreases temperature
  - For each temperature, generates random moves
  - Accepts any move that improves cost
  - Accepts some bad moves, less likely at low temperatures
- Results and complexity depend on temperature decrease rate
Simulated annealing algorithm

Temp=initial temperature  
Cost=Objfct(P)  
While not Frozen loop  
    while not Equilibrium loop  
        P_tentative=Move(P)  
        cost_tentative=Objfct(P_tentative)  
        cost=cost_tentative-cost  
        if(Accept(cost,temp)>Random(0,1)) then  
            P=P_tentative  
            cost=cost_tentative  
        end if  
    end loop  
    temp=DecreaseTemp(temp)  
End loop  

where: Accept(cost,temp)=min(1,e^(-cost/temp))

Genetic evolution

- Genetic algorithms treat a set of partitions as a generation, and create a new generation from a current one by imitating three evolution methods found in nature.
- Three evolution methods
  - Selection: random selected partition.
  - Crossover: randomly selected from two strong partitions.
  - Mutation: randomly selected partition after some randomly modification.
- Produce good result but suffer from long run times.
**Genetic evolution’s algorithm**

/*Create first generation with gen_size random partitions*/
G = ∅
for i in 1 to gen_size loop
    G = G ∪ CreateRandomPart(O)
end loop
P_best = BestPart(G)

/*Evolve generation*/
While not Terminate loop
    G = Select*(G,num_sel) U Cross(G,num_cross)
    Mutate(G,num_mutations)
    If Objfct(BestPart(G)) < Objfct(P_best) then
        P_best = BestPart(G)
    end if
end loop

/*Return best partition in final generation*/
return P_best

---

**Integer Linear Programming**

- A linear program formulation consists of a set of variables, a set of linear inequalities, and a single linear function of the variables that serves as an objective function.
  - A integer linear program is a linear program in which the variables can only hold integers.
- For partition purpose, the variables are used to represent partitioning decision or metric estimations.
- Still a NP-hard problem that requires some heuristics.
**Partition example**

- The Yorktown Silicon compiler uses a hierarchical clustering algorithm with the following closeness as the terminal conditions:

\[
Closeness(p_i, p_j) = \left( \frac{Conn_{i,j}}{MaxConn(P)} \right)^{k_2} \cdot \left( \frac{size_{\text{max}}}{\text{Min}(size_i, size_j)} \right)^{k_3} \cdot \left( \frac{size_{\text{max}}}{size_i + size_j} \right)
\]

\[Conn_{i,j} = k_1 \cdot \text{inputs}_{i,j} + \text{wire}_{i,j} \]

\[\text{inputs}_{i,j} = \# \text{ of common inputs shared} \]

\[\text{wires}_{i,j} = \# \text{ of op to ip and ip to op} \]

\[MaxConn(P) = \text{maximum Conn over all pairs} \]

\[size_i = \text{estimated size of group } p_i \]

\[size_{\text{max}} = \text{maximum group size allowed} \]

\[k_1, k_2, k_3 = \text{constants} \]

---

**Ysc partitioning example**

YSC partitioning example: (a) input (b) operation (c) operation closeness values (d) Clusters formed with 0.5 threshold

- Closeness(\(+, =\)) = \[\frac{8+0}{8} \times \frac{300}{120} \times \frac{300}{120 + 140} = 2.9\]

- Closeness(\(-, <\)) = \[\frac{0+4}{8} \times \frac{300}{160} \times \frac{300}{160 + 180} = 0.8\]

All other operation pairs have a closeness value of 0. The closeness values between all operations are shown in figure 6.6(c).

Figure 6.6(d) shows the results of hierarchical clustering with a closeness threshold of 0.5. The + and = operations form one cluster, and the < and – operations form a second cluster.
Output Issues in Partitioning

- Any partitioning technique must define the representation format and potential use of its output
  - E.g., the format may be a list indicating which functional object is mapped to which system component
  - E.g., the output may be a revised specification
    - Containing structural objects for the system components
    - Defining a component’s functionality using the functional objects mapped to it
Flow of Control and Designer Interaction

- Sequence in making decisions is variable, and any partitioning technique must specify the appropriate sequences
  - E.g., selection of granularity, closeness metrics, closeness functions
- Two classes of interaction
  - Directives
    - Include possible actions the designer can perform manually, such as allocation, overriding estimations, etc.
  - Feedback
    - Describe the current design information available to the designer (e.g., graphs of wires between objects, histograms, etc.)

Comparing Partitions Using Cost Functions

- A cost function is a function Cost(H, S, Cons, I) which returns a natural number that summarizes the overall quality of a given partition
  - I contains any additional information that is not contained in H or S or Cons
  - A smaller cost function value is desired
- An iterative improvement partitioning algorithm is defined as a procedure
  Part_Alg(H, S, Cons, I, Cost( ) )
  which returns a partition H', S' such that
  Cost(H', S', Cons, I) ≤ Cost(H, S, Cons, I)
Estimation

- Estimates allow
  - Evaluation of design quality
  - Design space exploration
- Design model
  - Represents degree of design detail computed
  - Simple vs. complex models
- Issues for estimation
  - Accuracy
  - Speed
  - Fidelity

Typical estimation model example

<table>
<thead>
<tr>
<th>Design model</th>
<th>Additional tasks</th>
<th>Accuracy</th>
<th>Fidelity</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Low</td>
<td>Low</td>
<td>fast</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU</td>
<td>FU allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg</td>
<td>Lifetime analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux</td>
<td>FU binding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux+Wiring</td>
<td>Floorplanning</td>
<td>high</td>
<td>high</td>
<td>slow</td>
</tr>
</tbody>
</table>
Accuracy vs. Speed

- Accuracy: difference between estimated and actual value
  \[
  A = 1 - \frac{|E(D) - M(D)|}{M(D)}
  \]
  \[|E(D)|, |M(D)|: \text{estimated & measured value}\]
- Speed: computation time spent to obtain estimate
- Simplified estimation models yield fast estimator but result in greater estimation error and less accuracy.

Fidelity

- Estimates must predict quality metrics for different design alternatives
- Fidelity: % of correct predictions for pairs of design implementations
- The higher fidelity of the estimation, the more likely that correct decisions will be made based on estimates.
- Definition of fidelity:
  \[
  F = 100 \times \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} u_{i,j}
  \]
  \[(A, B) = E(A) > E(B), M(A) < M(B) \quad \times\]
  \[(B, C) = E(B) < E(C), M(B) > M(C) \quad \times\]
  \[(A, C) = E(A) < E(C), M(A) < M(C) \quad \bigcirc \]
  Fidelity = 33 %
Quality metrics

- **Performance Metrics**
  - Clock cycle, control steps, execution time, communication rates

- **Cost Metrics**
  - **Hardware**: manufacturing cost (area), packaging cost (pin)
  - **Software**: program size, data memory size

- **Other metrics**
  - Power
  - Design for testability: Controllability and Observability
  - Design time
  - Time to market

Hardware design model
Clock cycles metric

- Selection of a clock cycle before synthesis will affect the practical execution time and the hardware resources.
- Simple estimation of clock cycle is based on maximum-operator-delay method.

\[ clk(MOD) = \max_{i} \left( \text{delay}(t_i) \right) \]

- The estimation is simple but may lead to underutilization of the faster functional units.
- Clock slack represents the portion of the clock cycle for which the functional unit is idle.

Clock cycle estimation

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Exec. Time</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>383 ns</td>
<td>383 ns</td>
<td>2 x 4 +</td>
</tr>
<tr>
<td>150 ns</td>
<td>630 ns</td>
<td>1 x 1+</td>
</tr>
<tr>
<td>80 ns</td>
<td>400 ns</td>
<td>1 x 1+</td>
</tr>
</tbody>
</table>
Clock slack and utilization

- **Slack**: portion of clock cycle for which FU is idle
  \[
  \text{slack}(\text{clk}, \text{ti}) = (\frac{\text{delay}(\text{ti})}{\text{dk}} \cdot \text{dk}) - \text{delay}(\text{ti})
  \]

- **Average slack**: FU slack averaged over all operations
  \[
  \text{ave_slack} = \frac{\sum_i \text{occur}(\text{ti}) \cdot \text{slack}(	ext{clk},\text{ti})}{\sum_i \text{occur}(\text{ti})}
  \]

- **Clock utilization**: % of clock cycle utilized for computations
  \[
  \text{utilization} = 1 - \frac{\text{ave_slack}(	ext{clk})}{\text{clk}}
  \]

Clock utilization

ave_slack(65 ns) = 6 + 2 + 2 = 24.4 ns

utilization(65 ns) = 1 - (24.4 / 65.0) = 62%
**Control steps estimation**

- Operations in the specification assigned to control step

- **Number of control steps** reflects:
  - Execution time of design
  - Complexity of control unit

- Techniques used to estimate the number of control steps in a behavior specified as straight-line code
  - Operator-use method.
  - Scheduling

**Operator-use method**

- Easy to estimate the number of control steps given the resources of its implementation.
- Number of control steps for each node can be calculated:

\[
csteps(n_j) = \max_{t_i \in T} \left[ \frac{\text{occur}(t_j)}{\text{num}(t_j)} \times \text{clocks}(t_i) \right]
\]

- The total number of control steps for behavior B is

\[
csteps(B) = \max_{n_j \in N} csteps(n_j)
\]
Operator-use method Example

- Differential-equation example:

Scheduling

- A scheduling technique is applied to the behavior description in order to determine the number of controls steps.

- It’s quite expensive to obtain the estimate based on scheduling.

- Resource-constrained vs time-constrained scheduling.
Scheduling for DSP algorithms

- Scheduling: assigning nodes of DFG to control times
- Resource allocations: assigning nodes of DFG to hardware (functional units)
- High-level synthesis
  - Resource-constrained synthesis
  - Time-constrained synthesis

Classification of scheduling algorithms

- Iterative/Constructive Scheduling Algorithms
  - As Soon As Possible Scheduling Algorithm (ASAP)
  - As Late As Possible Scheduling Algorithm (ALAP)
  - List-Scheduling Algorithms

- Transformational Scheduling Algorithms
  - Force Directed Scheduling Algorithm
  - Iterative Loop Based Scheduling Algorithm
  - Other Heuristic Scheduling Algorithms
The DFG in the Example

As Soon As Possible (ASAP) Scheduling Algorithm
- Find minimum start times of each node

Input: DFG $G = (N, E)$.
Output: ASAP Schedule.
1. $TS_0 = 1; \text{/* Set initial time step */}$
2. While (Unscheduled nodes exist) {
   2.1 Select a node $n_j$ whose predecessors have already been scheduled;
   2.2 Schedule node $n_j$ to time step $TS_j = \max \{ TS_i + (C_i) \}$ $orall n_i \rightarrow n_j$;
As Soon As Possible (ASAP) Scheduling Algorithm
- The ASAP schedule for the 2nd-order differential equation

As Late As Possible (ALAP) Scheduling Algorithm
- Find maximum start times of each node

Input: DFG $G = (N, E)$, Iteration Period = $T$.
Output: ALAP Schedule.
1. $T_{S0} = T$; /* Set initial time step */
2. While (Unscheduled nodes exist) {
   2.1 Select a node $n_i$ whose successors have already been scheduled;
   2.2 Schedule node $n_i$ to time step $T_{S_i} = \min \{ T_{S_j} - (C_i) \} \forall n_i \rightarrow n_j$;
As Late As Possible (ALAP) Scheduling Algorithm
- The ALAP schedule for the 2nd-order differential equation

List-Scheduling Algorithm (resource-constrained)
- A simple list-scheduling algorithm that prioritizes nodes by decreasing criticalness (e.g., scheduling range)
Force Directed Scheduling Algorithm (time-constrained)

- Transformation algorithm

Input: $DGF \ G = (N, E)$, $Iteration\ Period = T$.  
Output: Final FDS Schedule.  
1. While (Unscheduled nodes exist) {  
  1.1 Compute the time frames for each node;  
  1.2 Build the distribution graph;  
  1.3 Compute the self-forces;  
  1.4 Compute the predecessor and successor forces;  
  1.5 Schedule the node into the time step that minimizes the total force;  
}

Force Directed Scheduling Algorithm

- Figure (a) shows the time frame of the example DFG and the associated probabilities (obtained using ALAP and ASAP).  
- Figure (b) shows the DGs for the 2nd-order differential equation.
Force Directed Scheduling Algorithm

\[ Self\_Force(j) = \sum_{i=Sj}^{Lj} [DG(i) \times x(i)] \]

- Example:
  \[ Self\_Force_4(1) = Force_4(1) + Force_4(2) \]
  \[ = (DG_4(1) \times x_4(1)) + (DG_4(2) \times x_4(2)) \]
  \[ = (2.833 \times (-0.5)) + (2.333 \times (0.5)) \]
  \[ = -0.25 \]

- Example (con’d.):
  \[ Self\_Force_4(2) = Force_4(1) + Force_4(2) \]
  \[ = (DG_4(1) \times x_4(1)) + (DG_4(2) \times x_4(2)) \]
  \[ = (2.833 \times (-0.5)) + (2.333 \times (+0.5)) \]
  \[ = -0.25 \]

  \[ Succ\_Force_4(2) = Self\_Force_4(2) + Self\_Force_4(3) \]
  \[ = (DG_4(2) \times x_4(2)) + (DG_4(3) \times x_4(3)) \]
  \[ = (2.333 \times (0.5)) + (0.833 \times (-0.5)) \]
  \[ = -0.75 \]

  \[ Force_4(2) = Self\_Force_4(2) + Succ\_Force_4(2) \]
  \[ = -0.25 - 0.75 \]
  \[ = -1.00 \]
**Force Directed Scheduling Algorithm (another example)**

A1: \[ F_{a1}(0) = 0; \]
A2: \[ F_{a2}(1) = 0; \]
A3: T(1): Self\_F_{a3}(1)=1.5\times0.5-0.5\times0.5=0.5
Pred\_F_{m2}(1)=0.5\times0.5-0.5\times0.5=0
\[ F_{a3}(1) = 0.5 \]
T(2): Self\_F_{a3}(2)=-1.5\times0.5+0.5\times0.5=-0.5
\[ F_{a3}(2) = -0.5 \]
M1: \[ F_{m1}(2) = 0; \]
M2: T(0): Self\_F_{m2}(0)=0.5\times0.5-0.5\times0.5=0
\[ F_{m2}(0) = 0 \]
T(1): Self\_F_{m2}(1)=-0.5\times0.5+0.5\times0.5=0
Succ\_F_{a3}(2)=-1.5\times0.5+0.5\times0.5=-0.5
\[ F_{m2}(0) = -0.5 \]

**Scheduler**

- Critical path scheduler
  - Based on precedence graph (intra-iteration precedence constraints)
- Overlapping scheduler
  - Allow iterations to overlap
- Block schedule
  - Allow iterations to overlap
  - Allow different iterations to be assigned to different processors.
Overlapping schedule

- Example:
  - Minimum iteration period obtained from critical path scheduler is 8 t.u

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>A</td>
<td>1</td>
<td>A</td>
<td>1</td>
<td>A</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>P2</td>
<td>B</td>
<td>0</td>
<td>B</td>
<td>0</td>
<td>B</td>
<td>1</td>
<td>B</td>
<td>1</td>
<td>B</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td>0</td>
<td>C</td>
<td>0</td>
<td>E</td>
<td>0</td>
<td>C</td>
<td>1</td>
<td>C</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>P4</td>
<td>D</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

Block schedule

- Example:
  - Minimum iteration period obtained from critical path scheduler is 20 t.u

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A0</td>
<td>A0</td>
<td>B0</td>
<td>B0</td>
<td>C0</td>
<td>C0</td>
<td>A2</td>
<td>A2</td>
<td>B2</td>
<td>B2</td>
<td>C2</td>
<td>C2</td>
</tr>
<tr>
<td>P2</td>
<td>A1</td>
<td>A1</td>
<td>B1</td>
<td>B1</td>
<td>C1</td>
<td>C1</td>
<td>A3</td>
<td>A3</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>D0</td>
<td>D0</td>
<td>E0</td>
<td>D1</td>
<td>D1</td>
<td>E1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branching in behaviors

- Control steps maybe shared across exclusive branches
  - sharing schedule: fewer states, status register
  - non-sharing schedule: more states, no status registers

Execution time estimation

- Average start to finish time of behavior
- **Straight-line code behaviors**
  \[ \text{execution}(B) = \text{csteps}(B) \times \text{clk} \]
- **Behavior with branching**
  - Estimate execution time for each basic block
  - Create control flow graph from basic blocks
  - Determine branching probabilities
  - Formulate equations for node frequencies
  - Solve set of equations
  \[ \text{execution}(B) = \sum_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i) \]
Probability-based flow analysis

Flow equations:
- $\text{freq}(S) = 1.0$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(S)$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_1) + 0.9 > \text{freq}(v_5)$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_2)$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_2)$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_3) + 1.0 > \text{freq}(v_4)$
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_5)$

Node execution frequencies:
- $\text{freq}(v_1) = 1.0 \times \text{freq}(v_2) = 10.0$
- $\text{freq}(v_3) = 5.0 \times \text{freq}(v_4) = 5.0$
- $\text{freq}(v_5) = 10.0 \times \text{freq}(v_6) = 1.0$

Can be used to estimate number of accesses to variables, channels or procedures
Communication rate

- Communication between concurrent behaviors (or processes) is usually represented as messages sent over an abstract channel.
- Communication channel may be either explicitly specified in the description or created after system partitioning.
- Average rate of a channel $C$, $\text{avgrate}(C)$, is defined as the rate at which data is sent during the entire channel's lifetime.
- Peak rate of a channel, $\text{peakrate}(C)$, is defined as the rate at which data is sent in a single message transfer.

Communication rate estimation

- Total behavior execution time consists of
  - Computation time $\text{comptime}(B)$
    - Time required for behavior $B$ to perform its internal computation.
    - Obtained by the flow-analysis method.
  - Communication time $\text{commtime}(B, C)$
    - Time spent by behavior to transfer data over the channel $\text{commtime}(B, C) = \text{access}(B, C) \times \text{portdelay}(C)$
- Total bits transferred by the channel, $\text{total_bits}(B, C) = \text{access}(B, C) \times \text{bits}(C)$
- Channel average rate
  $$\text{average}(C) = \frac{\text{Total_bits}(B, C)}{\text{comptime}(B) + \text{commtime}(B, C)}$$
- Channel peak rate
  $$\text{peakrate}(C) = \frac{\text{bits}(C)}{\text{protocol_delay}(C)}$$
Communication rates

- **Average channel rate**
  rate of data transfer over lifetime of behavior
  \[
  \text{average rate (C)} = \frac{56\text{bits}}{1000\text{ns}} = 56\text{Mb/s}
  \]

- **Peak channel rate**
  rate of data transfer of single message
  \[
  \text{peak rate (C)} = \frac{8\text{bits}}{100\text{ns}} = 80\text{Mb/s}
  \]

Area estimation

- **Two tasks:**
  - Determining number and type of components required
  - Estimating component size for a specific technology (FSMD, gate arrays etc.)

- **Behavior implemented as a FSMD (finite state machine with datapath):**
  - Datapath components: registers, functional units, multiplexers/buses
  - Control unit: state register, control logic, next-state logic

- **Area can be accessed from the following aspects:**
  - Datapath component estimation
  - Control unit estimation
  - Layout area for a custom implementation
**Clique-partitioning**

- Commonly used for determining datapath components
- Let $G = (V,E)$ be a graph, $V$ and $E$ are set of vertices and edges
- Clique is a complete subgraph of $G$
- Clique-partitioning
  - divides the vertices into a minimal number of cliques
  - each vertex in exactly one clique
- One heuristic: maximum number of common neighbors
  - Two nodes with maximum number of common neighbors are merged
  - Edges to two nodes replaced by edges to merged node
  - Process repeated till no more nodes can be merged
**Storage unit estimation**

- Variables used in the behavior are mapped to storage units like registers or memory.
- Variables not used concurrently may be mapped to the same storage unit.
- Variables with non-overlapping lifetimes have an edge between their vertices.
- Lifetime analysis is popularly used in DSP synthesis in order to reduce number of registers required.

**Register Minimization Technique**

- Lifetime analysis is used for register minimization techniques in a DSP hardware.
- A ‘data sample or variable’ is live from the time it is produced through the time it is consumed. After that it is dead.
- Linear lifetime chart: Represents the lifetime of the variables in a linear fashion.
  - Note: Linear lifetime chart uses the convention that the variable is not live during the clock cycle when it is produced but live during the clock cycle when it is consumed.
- Due to the periodic nature of DSP programs the lifetime chart can be drawn for only one iteration to give an indication of the # of registers that are needed.
**Lifetime Chart**

- For DSP programs with iteration period $N$
  - Let the # of live variables at time partitions $n \geq N$ be the # of live variables due to 0-th iteration at cycles $n-kN$ for $k \geq 0$. In the example, # of live variables at cycle $7 \geq N$ ($=6$) is the sum of the # of live variables due to the 0-th iteration at cycles $7$ and $(7 - 1 \times 6) = 1$, which is $2 + 1 = 3$.

![Cycle Diagram](image1)

**Matrix transpose example**

<table>
<thead>
<tr>
<th>Sample</th>
<th>$T_a$</th>
<th>$T_{out}$</th>
<th>$T_{diff}$</th>
<th>$T_{out}$</th>
<th>Life</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0-94</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>1-97</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>2-10</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>1</td>
<td>-2</td>
<td>5</td>
<td>3-95</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>4-98</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>11</td>
<td>5-11</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>2</td>
<td>-4</td>
<td>6</td>
<td>6-96</td>
</tr>
<tr>
<td>h</td>
<td>7</td>
<td>5</td>
<td>-2</td>
<td>9</td>
<td>7-99</td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>12</td>
<td>8-12</td>
</tr>
</tbody>
</table>

To make the system causal a latency of 4 is added to the difference so that $T_{out}$ is the actual output time.
Circular Lifetime Chart

- Useful to represent the periodic nature of the DSP programs.
- In a circular lifetime chart of periodicity N, the point marked \( 0 \leq i \leq N - 1 \) represents the time partition \( i \) and all time instances \( ((N+i)) \) where \( i \) is any non-negative integer.
- For example: If \( N = 8 \), then time partition \( i = 3 \) represents time instances \( \{3, 11, 19, \ldots\} \).
- Note: Variable produced during time unit \( j \) and consumed during time unit \( k \) is shown to be alive from \( j + 1 \) to \( k \).
- The numbers in the bracket in the adjacent figure correspond to the \# of live variables at each time partition.

Forward-Backward Register Allocation Technique:

Note: Hashing is done to avoid conflict during backward allocation.
Steps of Register Allocation

- Determine the minimum number of registers using lifetime analysis.
- Input each variable at the time step corresponding to the beginning of its lifetime. If multiple variables are input in a given cycle, these are allocated to multiple registers with preference given to the variable with the longest lifetime.
- Each variable is allocated in a forward manner until it is dead or it reaches the last register. In forward allocation, if the register i holds the variable in the current cycle, then register i + 1 holds the same variable in the next cycle. If (i + 1)-th register is not free then use the first available forward register.
- Being periodic the allocation repeats in each iteration, so hash out the register Rj for the cycle /+ N if it holds a variable during cycle /.
- For variables that reach the last register and are still alive, they are allocated in a backward manner on a first come first serve basis.
- Repeat previous two steps until the allocation is complete.

Functional-unit and interconnect-unit estimation

- Clique-partitioning can be applied
- For determining the number of FU’s required, construct a graph where
  - Each operation in behavior represented by a vertex
  - Edge connects two vertices if corresponding operations assigned different control steps there exists an FU that can implement both operations
- For determining the number of interconnect units, construct a graph where
  - Each connection between two units is represented by a vertex
  - Edge connects two vertices if corresponding connections are not used in same control step
Computing datapath area

- Bit-sliced datapath

\[ L_{\text{bit}} = \alpha \times tr(DP) \]
\[ H_{rt} = \frac{\text{nets}}{\text{nets\_per\_track}} \times \beta \]
\[ area(\text{bit}) = L_{\text{bit}} \times (H_{\text{cell}} + H_{rt}) \]
\[ area(DP) = \text{bitwidth}(DP) \times area(\text{bit}) \]

Pin estimation

- Number of wires at behavior's boundary depends on
  - Global data
  - Port accessed
  - Communication channels used
  - Procedure calls
Software estimation model

- Processor-specific estimation model
  - Exact value of a metric is computed by compiling each behavior into the instruction set of the targeted processor using a specific compiler.
  - Estimation can be made accurately from the timing and size information reported.
  - Bad side is hard to adapt an existing estimator for a new processor.

- Generic estimation model
  - Behavior will be mapped to some generic instructions first.
  - Processor-specific technology files will then be used to estimate the performance for the targeted processors.
Deriving processor technology files

<table>
<thead>
<tr>
<th>Instruction</th>
<th>clock</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax,word ptr[bp+offset1]</td>
<td>(10)</td>
<td>3</td>
</tr>
<tr>
<td>add ax,word ptr[bp+offset2]</td>
<td>(9+EA1)</td>
<td>4</td>
</tr>
<tr>
<td>mov word ptr[bp+offset3],ax)</td>
<td>(10)</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>clock</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov a6@(offset1),do</td>
<td>(7)</td>
<td>2</td>
</tr>
<tr>
<td>add a6@(offset2),do</td>
<td>(2+EA2)</td>
<td>2</td>
</tr>
<tr>
<td>mov d0,a6@(offset3)</td>
<td>(5)</td>
<td>2</td>
</tr>
</tbody>
</table>

8086 instructions: 6820 instructions

Software estimation

- **Program execution time**
  - Create basic blocks and compile into generic instructions
  - Estimate execution time of basic blocks
  - Perform probability-based flow analysis
  - Compute execution time of the entire behavior:
    \[
    \text{exectime}(B) = \delta \times \left( \sum \text{exectime}(bi) \times \text{freq}(bi) \right)
    \]
    \(\delta\) accounts for compiler optimizations
  - accounts for compiler optimizations

- **Program memory size**
  \[
  \text{progsizze}(B) = \sum_{g} \text{instr\_size}(g)
  \]

- **Data memory size**
  \[
  \text{datasize}(B) = \sum_{d} \text{datasize}(d)
  \]
Refinement

- Refinement is used to reflect the condition after the partitioning and the interface between HW/SW is built
  - Refinement is the update of specification to reflect the mapping of variables.
- Functional objects are grouped and mapped to system components
  - Functional objects: variables, behaviors, and channels
  - System components: memories, chips or processors, and buses
- Specification refinement is very important
  - Makes specification consistent
  - Enables simulation of specification
  - Generate input for synthesis, compilation and verification tools

Refining variable groups

- The memory to which the group of variables are reflected and refined in specification.
- Variable folding:
  - Implementing each variable in a memory with a fixed word size
- Memory address translation
  - Assignment of addresses to each variable in group
  - Update references to variable by accesses to memory
Variable folding

```
variable J, K : integer := 0;
variable V : IntArray (63 downto 0);
....
V(K) := 3;
X := V(36);
V(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + V(J);
end loop;
....
```

```
V (63 downto 0)
MEM(163 downto 100)
```

Assigning addresses to V

```
variable J : integer := 0;
variable K : integer := 0;
variable MEM : IntArray (255 downto 0);
....
MEM(K + 100) := 3;
X := MEM(136);
MEM(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + MEM(J);
end loop;
....
```

Refined specification

```
variable J, K : integer := 0;
variable MEM : IntArray (255 downto 0);
....
MEM(K +100) := 3;
X := MEM(136);
MEM(J+100) := X;
....
for J in 0 to 63 loop
SUM := SUM + MEM(J+100);
end loop;
....
```

Refined specification without offsets for index J

Memory address translation
Channel refinement

- Channels: virtual entities over which messages are transferred
- Bus: physical medium that implements groups of channels
- Bus consists of:
  - wires representing data and control lines
  - protocol defining sequence of assignments to data and control lines
- Two refinement tasks
  - Bus generation: determining bus width
    - number of data lines
  - Protocol generation: specifying mechanism of transfer over bus

Communication

- Shared-memory communication model
  - Persistent shared medium
  - Non-persistent shared medium
- Message-passing communication model
  - Channel
    - uni-directional
    - bi-directional
    - Point-to-point
    - Multi-way
  - Blocking
  - Non-blocking
- Standard interface scheme
  - Memory-mapped, serial port, parallel port, self-timed, synchronous, blocking
**Communication (cont)**

Shared memory $M$

Process P
begin
variable $x$
... $M := x$;
... end

Process Q
begin
variable $y$
... $y := M$;
... end

Process P
begin
variable $x$
... send($x$);
... end

Process Q
begin
variable $y$
... receive($y$);
... end

Channel $C$

(a) (b)

Inter-process communication paradigms: (a) shared memory, (b) message passing

**Characterizing communication channels**

- For a given behavior that sends data over channel $C$,
  - **Message size** $\text{bits}(C)$
    - number of bits in each message
  - **Accesses**: $\text{accesses}(B,C)$
    - number of times $P$ transfers data over $C$
  - **Average rate** $\text{averate}(C)$
    - rate of data transfer of $C$ over lifetime of behavior
  - **Peak rate** $\text{peakrate}(C)$
    - rate of transfer of single message

```
\text{bits}(C) = 8 \text{ bits}
\text{averate}(C) = \frac{24 \text{ bits}}{400 \text{ ns}} = 60 \text{ Mbits/s}
\text{peakrate}(C) = \frac{8 \text{ bits}}{100 \text{ ns}} = 80 \text{ Mbits/s}
```
Characterizing buses

For a given bus \( B \)
- **Buswidth** \( \text{buswidth}(B) \)
  - number of data lines in \( B \)
- **Protocol delay** \( \text{protdelay}(B) \)
  - delay for single message transfer over bus
- **Average rate** \( \text{averate}(B) \)
  - rate of data transfer over lifetime of system
- **Peak rate** \( \text{peakrate}(B) \)
  - maximum rate of transfer of data on bus

\[
\text{peakrate}(C) = \frac{\text{buswidth}(B)}{\text{protdelay}(B)}
\]

Determining bus rates

- Idle slots of a channel used for messages of other channels
- To ensure that channel average rates are unaffected by bus
  \[
  \text{averate}(B) = \sum_{C \in B} \text{averate}(C)
  \]
- Goal: to synthesize a bus that constantly transfers data for channel
  \( \text{peakrate}(B) = \text{averate}(C) \)
Constraints for bus generation

- **Bus-width**: affects number of pins on chip boundaries
- **Channel average rates**: affects execution time of behaviors
- **Channel peak rates**: affects time required for single message transfer

![Diagram showing channel and bus structures]

**Bus generation algorithm**

- **Compute buswidth range**: minwidth=1, maxwidth = Max(bit(C))
- **For** minwidth: curwidth = maxwidth **loop**
  - **Compute bus peak rate**:
    
    \[
    \text{peakrate}(B) = \text{curwidth} \div \text{protdelay}(B)
    \]
  - **Compute channel average rates**
    
    \[
    \text{commtime}(B) = \text{access}(B, C) \times \left[ \text{curwidth} \right] \times \text{protdelay}(B)
    \]
    
    \[
    \text{average}(C) = \frac{\text{access}(B, C) \times \text{bits}(C)}{\text{commtime}(B) + \text{commtime}(B)}
    \]
  - **If** peakrate(B) ≥ \( \Sigma \) averate(C) **then**
    
    if bestcost > ComputeCost(curwidth) then
    
    bestcost = ComputeCost(curwidth)
    
    bestwidth = curwidth
Bus generation example

- Assume
  - 2 behavior accessing 16 bit data over two channels
  - Constraints specified for channel peak rates

<table>
<thead>
<tr>
<th>Channel C</th>
<th>Behavior B</th>
<th>Variable accessed</th>
<th>Bits(C)</th>
<th>Access(B, C)</th>
<th>Comptime(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH1</td>
<td>P1</td>
<td>V1</td>
<td>16 data + 7 addr</td>
<td>128</td>
<td>515</td>
</tr>
<tr>
<td>CH2</td>
<td>P2</td>
<td>V2</td>
<td>16 data + 7 addr</td>
<td>128</td>
<td>129</td>
</tr>
</tbody>
</table>

Protocol generation

- Bus consists of several sets of wires:
  - Data lines, used for transferring message bits
  - Control lines, used for synchronization between behaviors
  - ID lines, used for identifying the channel active on the bus
- All channels mapped to bus share these lines
- Number of data lines determined by bus generation algorithm
- Protocol generation consists of six steps
Protocol generation steps

1. Protocol selection
   - full handshake, half-handshake etc.

2. ID assignment
   - N channels require log2(N) ID lines

3. Bus structure and procedure definition
   - The structure of bus (the data, control, ID lines) is defined in the specification.

4. Update variable-reference
   - References to a variable that has been assigned to another component must be updated.

5. Generate processes for variables
   - Extra behavior should be created for those variables that have been sent across a channel.
Protocol generation example

```vhdl
type HandShakeBus is record
  START, DONE : bit;
  ID : bit_vector(1 downto 0);
  DATA : bit_vector(7 downto 0);
end record;

signal B : HandShakeBus;

procedure ReceiveCH0( rxdata : out bit_vector) is
begin
  for J in 1 to 2 loop
    wait until (B.START = '1') and (B.ID = "00")
    rxdata (8*J-1 downto 8*(J-1)) <= B.DATA;
    B.DONE <= '1';
  end loop;
end ReceiveCH0;

procedure SendCH0( txdata : in bit_vector) is
begin
  bus B.ID <= "00";
  for J in 1 to 2 loop
    B.data <= txdata(8*J-1 downto 8*(J-1))
    B.START <= '1';
    B.START <= '0';
    B.DONE <= '1';
  end loop;
end SendCH0;
```

Refined specification after protocol generation
Resolving access conflicts

- System partitioning may result in concurrent accesses to a resource
  - Channels mapped to a bus may attempt data transfer simultaneously
  - Variables mapped to a memory may be accessed by behaviors simultaneously
- Arbiter needs to be generated to resolve such access conflicts
- Three tasks
  - Arbitration model selection
  - Arbitration scheme selection
  - Arbiter generation

Arbitration models

STATIC

Dynamic
**Arbitration schemes**

- Arbitration schemes determines the priorities of the group of behaviors’ access to solve the access conflicts.
- Fixed-priority scheme statically assigns a priority to each behavior, and the relative priorities for all behaviors are not changed throughout the system's lifetime.
  - Fixed priority can be also pre-emptive.
  - It may lead to higher mean waiting time.
- Dynamic-priority scheme determines the priority of a behavior at the run-time.
  - Round-robin
  - First-come-first-served

**Refinement of incompatible interfaces**

- Three situation may arise if we bind functional objects to standard components:
- Neither behavior is bound to a standard component.
  - Communication between two can be established by generating the bus and inserting the protocol into these objects.
- One behavior is bound to a standard component
  - The behavior that is not associated with standard component has to use dual protocol to the other behavior.
- Both behaviors are bound to standard components.
  - An interface process has to be inserted between the two standard components to make the communication compatible.
Effect of binding on interfaces

Protocol operations

- Protocols usually consist of five atomic operations
  - waiting for an event on input control line
  - assigning value to output control line
  - reading value from input data port
  - assigning value to output data port
  - waiting for fixed time interval
- Protocol operations may be specified in one of three ways
  - Finite state machines (FSMs)
  - Timing diagrams
  - Hardware description languages (HDLs)
Protocol specification: FSMs

- Protocol operations ordered by sequencing between states
- Constraints between events may be specified using timing arcs
- Conditional & repetitive event sequences require extra states, transitions

Protocol specification: Timing diagrams

- Advantages:
  - Ease of comprehension, representation of timing constraints
- Disadvantages:
  - Lack of action language, not simulatable
  - Difficult to specify conditional and repetitive event sequences
Protocol specification: HDLs

- **Advantages:**
  - Functionality can be verified by simulation
  - Easy to specify conditional and repetitive event sequences

- **Disadvantages:**
  - Cumbersome to represent timing constraints between events

### Interface process generation

- **Input:** HDL description of two fixed, but incompatible protocols
- **Output:** HDL process that translates one protocol to the other
  - i.e. responds to their control signals and sequence their data transfers
- **Four steps required for generating interface process (IP):**
  - Creating relations
  - Partitioning relations into groups
  - Generating interface process statements
  - Interconnect optimization
IP generation: creating relations

- Protocol represented as an ordered set of relations
- Relations are sequences of events/actions

Protocol Pa

ADDRp <= AddrVar(7 downto 0);
ARDyp <= '1';
wait until (ARCVp = '1');
ADDRp <= AddrVar(15 downto 8);
DREQp <= '1';
wait until (DRDYp = '1');
DataVar <= DATAp;

Relations

A1[ (true) :
    ADDRp <= AddrVar(7 downto 0)
    ARDYp <= '1' ]
A2[ (ARCVp = '1') :
    ADDRp <= AddrVar(15 downto 8)
    DREQp <= '1' ]
A3 [ (DRDYp = '1') :
    DataVar <= DATAp ]

IP generation: partitioning relations

- Partition the set of relations from both protocols into groups.
- Group represents a unit of data transfer

<table>
<thead>
<tr>
<th>Protocol Pa</th>
<th>Protocol Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 (8 bits out)</td>
<td>B1 (16 bits in)</td>
</tr>
<tr>
<td>A2 (8 bits out)</td>
<td></td>
</tr>
<tr>
<td>A3 (16 bits in)</td>
<td>B2 (16 bits out)</td>
</tr>
<tr>
<td>G1=(A1 A2 B1)</td>
<td>G2=(B1 A3)</td>
</tr>
</tbody>
</table>
IP generation: inverting protocol operations

- For each operation in a group, add its dual to interface process
- Dual of an operation represents the complementary operation
- Temporary variable may be required to hold data values

<table>
<thead>
<tr>
<th>Atomic operation</th>
<th>Dual operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait until (Cp = '1')</td>
<td>Cp = '1' wait until (Cp = '1')</td>
</tr>
<tr>
<td>var &lt;= Dp</td>
<td>Dp &lt;= TempVar TempVar := Dp wait for 100 ns</td>
</tr>
<tr>
<td>Dp &lt;= var</td>
<td>wait for 100 ns</td>
</tr>
</tbody>
</table>

IP generation: interconnect optimization

- Certain ports of both protocols may be directly connected
- Advantages:
  - Bypassing interface process reduces interconnect cost
  - Operations related to these ports can be eliminated from interface process
Transducer synthesis

- Input: Timing diagram description of two fixed protocols
- Output: Logic circuit description of transducer
- Steps for generating logic circuit from timing diagrams:
  - Create event graphs for both protocols
  - Connect graphs based on data dependencies or explicitly specified ordering
  - Add templates for each output node in combined graph
  - Merge and connect templates
  - Satisfy min/max timing constraints
  - Optimize skeletal circuit

Generating event graphs from timing diagrams

[Diagram of event graphs for FIFO stack control cell]
Deriving skeletal circuit from event graph

- Advantages:
  - Synthesizes logic for transducer circuit directly
  - Accounts for min/max timing constraints between events
- Disadvantages:
  - Cannot interface protocols with different data port sizes
  - Transducer not simulatable with timing diagram description of protocols

Hardware/Software interface refinement
Tasks of hardware/software interfacing

- Data access (e.g., behavior accessing variable) refinement
- Control access (e.g., behavior starting behavior) refinement
- Select bus to satisfy data transfer rate and reduce interfacing cost
- Interface software/hardware components to standard buses
- Schedule software behaviors to satisfy data input/output rate
- Distribute variables to reduce ASIC cost and satisfy performance

Cosynthesis

- Methodical approach to system implementations using automated synthesis-oriented techniques
- Methodology and performance constraints determine partitioning into hardware and software implementations
- The result is "optimal" system that benefits from analysis of hardware/software design trade-off analysis
Cosynthesis Approach to System Implementation

Co-synthesis

- Implementation of hardware and software components after partitioning
- Constraints and optimization criteria similar to those for partitioning
- Area and size traded-off against performance
- Cost considerations
**Synthesis Flow**

- HW synthesis of dedicated units
  - Based on research or commercial standard synthesis tools
- SW synthesis of dedicated units (processors)
  - Based on specialized compiling techniques
- Interface synthesis
  - Definition of HW/SW interface and synchronization
  - Drivers of peripheral devices

**Co-Synthesis - The POLIS Flow**

“ESTEREL” for functional specification language
Hardware Design Methodology

Hardware Design Process:
Waterfall Model

Hardware Design Methodology (Cont.)

- Use of HDLs for modeling and simulation
- Use of lower-level synthesis tools to derive register transfer and lower-level designs
- Use of high-level hardware synthesis tools
  - Behavioral descriptions
  - System design constraints
- Introduction of synthesis for testability at all levels
Hardware Synthesis

- **Definition**
  - The automatic design and implementation of hardware from a specification written in a hardware description language

- **Goals/benefits**
  - To quickly create and modify designs
  - To support a methodology that allows for multiple design alternative consideration
  - To remove from the designer the handling of the tedious details of VLSI design
  - To support the development of correct designs

Hardware Synthesis Categories

- **Algorithm synthesis**
  - Synthesis from design requirements to control-flow behavior or abstract behavior
  - Largely a manual process

- **Register-transfer synthesis**
  - Also referred to as “high-level” or “behavioral” synthesis
  - Synthesis from abstract behavior, control-flow behavior, or register-transfer behavior (on one hand) to register-transfer structure (on the other)
  - Logic synthesis
  - Synthesis from register-transfer structures or Boolean equations to gate-level logic (or physical implementations using a predefined cell or IC library)
**Hardware Synthesis Process Overview**

1. **Specification**
   - Behavioral Simulation
   - Optional RTL Simulation

2. **Implementation**
   - Behavioral Synthesis
   - Synthesis & Test Synthesis

3. **Verification**
   - Gate-level Simulation
   - Gate-level Analysis

4. **Silicon Vendor**
   - Place and Route
   - Silicon

**HW Synthesis**

1. Specification Analysis
2. Concurrent Design
3. System Integration
Software Design Methodology

Software Design Process:
Waterfall Model

Software Design Methodology (Cont.)

- Software requirements includes both
  - Analysis
  - Specification
- Design: 2 levels:
  - System level - module specs.
  - Detailed level - process design language (PDL) used
- Coding - in high-level language
  - C/C++
- Maintenance - several levels
  - Unit testing
  - Integration testing
  - System testing
  - Regression testing
  - Acceptance testing
Software Synthesis

- Definition: the automatic development of correct and efficient software from specifications and reusable components

- Goals/benefits
  - To increase software productivity
  - To lower development costs
  - To increase confidence that software implementation satisfies specification
  - To support the development of correct programs

Why Use Software Synthesis?

- Software development is becoming the major cost driver in fielding a system

- To significantly improve both the design cycle time and life-cycle cost of embedded systems, a new software design methodology, including automated code generation, is necessary

- Synthesis supports a correct-by-construction philosophy

- Techniques support software reuse
Why Software Synthesis?

- More software ➔ high complexity ➔ need for automatic design (synthesis)
- Eliminate human and logical errors
- Relatively immature synthesis techniques for software
- Code optimizations
  - size
  - efficiency
- Automatic code generation

Software Synthesis Flow Diagram for Embedded System with Time Petri-Net

Automatically Generate CODE

- Real-Time Embedded System Model?
  Set of concurrent tasks with memory and timing constraints!
- How to execute in an embedded system (e.g. 1 CPU, 100 KB Mem)?
  Task Scheduling!
- How to generate code?
  Map schedules to software code!
- Code optimizations?
  Minimize size, maximize efficiency!

Software Synthesis Categories

- Language compilers
  - ADA and C compilers
  - YACC - yet another compiler compiler
  - Visual Basic
- Domain-specific synthesis
  - Application generators from software libraries
Software Synthesis Examples

- Mentor Graphics Concurrent Design Environment System
  - Uses object-oriented programming (written in C++)
  - Allows communication between hardware and software synthesis tools
- Index Technologies Excelerator and Cadre’s Teamwork Toolsets
  - Provide an interface with COBOL and PL/1 code generators
- KnowledgeWare’s IEW Gamma
  - Used in MIS applications
  - Can generate COBOL source code for system designers
- MCCI’s Graph Translation Tool (GrTT)
  - Used by Lockheed Martin ATL
  - Can generate ADA from Processing Graph Method (PGM) graphs

Function Architecture Co-design Methodology

- System Level design methodology
- Top-down (synthesis)
- Bottom-up (constraint-driven)
Co-design Process Methodology

System Level Design Vision
**Main Concepts**

- Decomposition
- Abstraction and successive refinement
- Target architectural exploration and estimation

**Decomposition**

- Top-down flow
- Find an optimal match between the application function and architectural application constraints (size, power, performance).
- Use separation of concerns approach to decompose a function into architectural units.
Abstraction & Successive Refinement

- Function/Architecture formal trade-off is applied for mapping function onto architecture
- Co-design and trade-off evaluation from the highest level down to the lower levels
- Successive refinement to add details to the earlier abstraction level

Target Architectural Exploration and Estimation

- Synthesized target architecture is analyzed and estimated
- Architecture constraints are derived
- An adequate model of target architecture is built
Architectural Exploration in POLIS

Main Steps in Co-design and Synthesis

- Function architecture co-design and trade-off
  - Fully synthesize the architecture?
  - Co-simulation in trade-off evaluation
    - Functional debugging
    - Constraint satisfaction and missed deadlines
    - Processor utilization and task scheduling charts
    - Cost of implementation
- Mapping function on the architecture
  - Architecture organization can be a pre-designed collection of components with various degrees of flexibilities
  - Matching the optimal function to the best architecture
**Function/ Architecture Co-design vs. HW/SW Co-design**

- Design problem over-simplified
- Must use Fun./Arch. Optimization & Co-design to match the optimal Function to the best Architecture
  1. Fun./Arch. Co-design and Trade-off
  2. Mapping Function Onto Architecture

---

**Reactive System Co-synthesis\(^{(1)}\)**

- Control-dominated Design
- EFSM Representation
- CDFG Representation
- HW/SW

EFSM: Extended Finite State Machines
CDFG: Control Data Flow directed acyclic Graph
Reactive System Co-synthesis

CDFG is suitable for describing EFSM reactive behavior but:
- Some of the control flow is hidden
- Data cannot be propagated

Data Flow Optimization
Optimization and Co-design Approach

- **Architecture-independent phase**
  - Task function is considered solely and control data flow analysis is performed
  - Removing redundant information and computations

- **Architecture-dependent phase**
  - Rely on architectural information to perform additional guided optimizations tuned to the target platform

Concrete Co-design Flow
Function/Architecture Co-Design

Design Representation

Abstract Co-design Flow

- Design
- Application
- Decomposition
- Function/Architecture Optimization and Co-design
- Mapping
- Hardware/Software Co-synthesis
**Unifying Intermediate Design Representation for Co-design**

**Platform-Based Design**

Source: ASV
Models and System

- Models of computation
  - Petri-net model (graphical language for system design)
  - FSM (Finite-State Machine) models
  - Hierarchical Concurrent FSM models
- POLIS system
  - CFSM (Co-design FSM)
  - EFSM (Extended FSM): support for data handling and asynchronous communication

CFSM

- Includes
  - Finite state machine
  - Data computation
  - Locally synchronous behavior
  - Globally asynchronous behavior
- Semantics: GALS (Globally Asynchronous and Locally Synchronous communication model)
CFSM Network MOC

Communication between CFSMs by means of events

System Specification Language

- "Esterel"
  - as "front-end" for functional specification
  - Synchronous programming language for specifying reactive real-time systems

- Reactive VHDL

- Graphical EFSM
**Intermediate Design Representation (IDR)**

- Most current optimization and synthesis are performed at the low abstraction level of a DAG (Direct Acyclic Graph).
- Function Flow Graph (FFG) is an IDR having the notion of I/O semantics.
- Textual interchange format of FFG is called C-Like Intermediate Format (CLIF).
- FFG is generated from an EFSM description and can be in a Tree Form or a DAG Form.

**Diagram:**

```
Refinement
Architecture Independent   Architecture Dependent

FFG

Constraints

Design

I/O Semantics

EFSM Semantics

SW

HW
```

**Architecture Function Flow Graph**
**FFG/CLIF**

- Develop Function Flow Graph (FFG) / C-Like Intermediate Format (CLIF)
  - Able to capture EFSM
  - Suitable for control and data flow analysis

**Function Flow Graph (FFG)**

- FFG is a triple $G = (V, E, N_0)$ where
  - $V$ is a finite set of nodes
  - $E = \{(x,y)\}$, a subset of $V \times V$; $(x,y)$ is an edge from $x$ to $y$ where $x \in \text{Pred}(y)$, the set of predecessor nodes of $y$.
  - $N_0 \in V$ is the start node corresponding to the EFSM initial state.
  - An unordered set of operations is associated with each node $N$.
  - Operations consist of TESTs performed on the EFSM inputs and internal variables, and ASSIGNs of computations on the input alphabet (inputs/internal variables) to the EFSM output alphabet (outputs and internal (state) variables)
C-Like Intermediate Format (CLIF)

- Import/Export Function Flow Graph (FFG)
- "Un-ordered" list of TEST and ASSIGN operations
  - \([\text{if } (\text{condition})] \text{ goto label}\)
  - \(\text{dest} = \text{op}(\text{src})\)
    - \(\text{op} = \{\text{not, minus, ...}\}\)
  - \(\text{dest} = \text{src}1 \text{ op src}2\)
    - \(\text{op} = \{+, *, /, ||, &&, |, &,...\}\)
  - \(\text{dest} = \text{func} (\text{arg}1, \text{arg}2, ...)\)

Preserving I/O Semantics

input inp;
output outp;
int a = 0;
int CONST_0 = 0;
int T11 = 0;
int T13 = 0;

S1: goto S2;
S2:
a = inp;
\(T13 = a + 1 \text{ CONST}_0;\)
\(T11 = a + a;\)
outp = T11;
goto S3;
S3:
outp = T13;
goto S3;
**FFG / CLIF Example**

*Legend: constant, output flow, dead operation*

\( S# = \text{State}, \ S#L# = \text{Label in State} S# \)

**Function**

Flow Graph

- \((\text{cond2} == 1) / \text{output(b)}\)
- \((\text{cond2} == 0) / \text{output(a)}\)

CLIF

Textual Representation

- \(x \leftarrow x + y\)
- \(x \leftarrow x + y\)
- \(a \leftarrow b + c\)
- \(a \leftarrow x\)
- \(\text{cond1} = (y == \text{cst1})\)
- \(\text{cond2} = !\text{cond1}\)
- if (\text{cond2}) goto S1L0
- \(\text{output} = a\); goto S1; /* Loop */
- \(S1L0: \text{output} = b; \)
- \(\text{goto} \ S1;\)

**Tree-Form FFG**

Figure 3.7  EFSM in FFG Tree Form: A Simple Example
Function/Architecture Co-Design

Function/Architecture Optimizations

Function Optimization

- Architecture-Independent optimization objective:
  - Eliminate redundant information in the FFG.
  - Represent the information in an optimized FFG that has a minimal number of nodes and associated operations.
FFG Optimization Algorithm

- FFG Optimization algorithm (G)

\[
\text{begin}\\
\text{while changes to FFG do}\\
\text{Variable Definition and Uses}\\
\text{FFG Build}\\
\text{Reachability Analysis}\\
\text{Normalization}\\
\text{Available Elimination}\\
\text{False Branch Pruning}\\
\text{Copy Propagation}\\
\text{Dead Operation Elimination}\\
\text{end while}\\
\text{end}
\]

Optimization Approach

- Develop optimizer for FFG (CLIF) intermediate design representation
- Goal: Optimize for speed, and size by reducing
  - ASSIGN operations
  - TEST operations
  - variables
- Reach goal by solving sequence of data flow problems for analysis and information gathering using an underlying Data Flow Analysis (DFA) framework
- Optimize by information redundancy elimination
Sample DFA Problem
Available Expressions Example

- Goal is to eliminate re-computations
  - Formulate Available Expressions Problem
    - Forward Flow (meet) Problem

- $AE = \emptyset$
- $t := a + 1$
- $AE = \{a+1\}$
- $AE = \emptyset$
- $S_2$
  - $t_1 := a + 1$
  - $t_2 := b + 2$
- $AE = \emptyset$

$AE = \{a+1\}$

- $S_3$
  - $a := a \cdot 5$
  - $t_3 := a + 2$
- $AE = \{a+2\}$

Data Flow Problem Instance

- A particular (problem) instance of a monotone data flow analysis framework is a pair $I = (G, M)$ where $M: N \rightarrow F$ is a function that maps each node $N$ in $V$ of FFG $G$ to a function in $F$ on the node label semilattice $L$ of the framework $D$. 
Data Flow Analysis Framework

- A monotone data flow analysis framework $D = (L, \land, F)$ is used to manipulate the data flow information by interpreting the node labels on $N$ in $V$ of the FFG $G$ as elements of an algebraic structure where
  - $L$ is a bounded semilattice with meet $\land$, and
  - $F$ is a monotone function space associated with $L$.

Solving Data Flow Problems

Data Flow Equations

\[
\text{In}(S3) = \bigcap_{P \in \{S1, S2\}} \text{Out}(P)
\]

\[
\text{Out}(S3) = (\text{In}(S3) - \text{Kill}(S3)) \cup \text{Gen}(S3)
\]
Solving Data Flow Problems

- Solve data flow problems using the *iterative method*
  - General: does not depend on the flow graph
  - Optimal for a class of data flow problems Reaches fixpoint in polynomial time (O(n²))

FFG Optimization Algorithm

- Solve following problems in order to improve design:
  - Reaching Definitions and Uses
  - Normalization
  - Available Expression Computation
  - Copy Propagation, and Constant Folding
  - Reachability Analysis
  - False Branch Pruning

- Code Improvement techniques
  - Dead Operation Elimination
  - Computation sharing through normalization
**Function/Architecture Co-design**

![Diagram](image)

**Function Architecture Optimizations**

- Fun./Arch. Representation:
  - Attributed Function Flow Graph (AFFG) is used to represent architectural constraints impressed upon the functional behavior of an EFSM task.
Architecture Dependent Optimizations

![Diagram](image)

EFSM in AFFG (State Tree) Form

![Diagram](image)
**Architecture Dependent Optimization Objective**

- Optimize the AFFG task representation for speed of execution and size given a set of architectural constrains
- Size: area of hardware, code size of software

---

**Motivating Example**

Eliminate the redundant needless runtime re-evaluation of the a+b operation
Cost-guided Relaxed Operation Motion (ROM)

- For performing safe and operation from heavily executed portions of a design task to less visited segments
- Relaxed-Operation-Motion (ROM):
  ```
  begin
  Data Flow and Control Optimization
  Reverse Sweep (dead operation addition, Normalization and available operation elimination, dead operation elimination)
  Forward Sweep (optional, minimize the lifetime)
  Final Optimization Pass
  end
  ```

Cost-Guided Operation Motion
Function Architecture Co-design in the Micro-Architecture

System Constraints
Decomposition

System Specs
Decomposition

t1 = 3*b

emit x(t2)

t2 = t1 + a

Operator Strength Reduction

Instruction Selection

Operator Strength Reduction

Reducing the multiplication operator

expr1 = b + b;
t1 = expr1 + b;
t2 = t1 + a;
x = t2;
**Architectural Optimization**

- Abstract Target Platform
  - Macro-architectures of the HW or SW system design tasks
- CFSM (Co-design FSM): FSM with reactive behavior
  - A reactive block
  - A set of combinational data-low functions
- Software Hardware Intermediate Format (SHIFT)
  - SHIFT = CFSMs + Functions

**Macro-Architectural Organization**

[Diagram showing SW Partition and HW Partition with interfaces and processors]
Architectural Organization of a Single CFSM Task

CFSM

Task Level Control and Data Flow Organization
CFSM Network Architecture

- Software Hardware Intermediate FormaT (SHIFT) for describing a network of CFSMs
- It is a hierarchical netlist of
  - Co-design finite state machine
  - Functions: state-less arithmetic, Boolean, or user-defined operations

SHIFT: CFSMs + Functions
Architectural Modeling

- Using an AUXiliary specification (AUX)
- AUX can describe the following information
  - Signal and variable type-related information
  - Definition of the value of constants
  - Creation of hierarchical netlist, instantiating and interconnecting the CFSMs described in SHIFT

Mapping AFFG onto SHIFT

- Synthesis through mapping AFFG onto SHIFT and AUX (Auxiliary Specification)
- Decompose each AFFG task behavior into a single reactive control part, and a set of data-path functions.

Mapping AFFG onto SHIFT Algorithm (G, AUX)
begin
  foreach state s belong to G do
    build_trel (s.trel, s, s.start_node, G, AUX);
  end foreach
end
**Architecture Dependent Optimizations**

- Additional architecture Information leads to an increased level of macro- (or micro-) architectural optimization
- Examples of macro-arch. Optimization
  - Multiplexing computation Inputs
  - Function sharing
- Example of micro-arch. Optimization
  - Data Type Optimization

**Distributing the Reactive Controller**

Move some of the control into data path as an ITE assign expression

ITE: if-then-else
**Multiplexing Inputs**

\[ T = b + c \]

\[ c = a \]

**Micro-Architectural Optimization**

- Available Expressions cannot eliminate \( T_2 \)
- But if variables are registered (additional architectural information) we can share \( T_1 \) and \( T_2 \)
Function/Architecture Co-Design

Hardware/Software Co-Synthesis and Estimation

Co-Synthesis Flow

FFG Interpreter (Simulation)

 FFG  AFFG  SHIFT

Or

Object Code (.o)  Netlist
**POLIS Co-design Environment**

- Specification: FSM-based languages (Esterel, ...)
- Internal representation: CFSM network
- Validation:
  - High-level co-simulation
  - FSM-based formal verification
  - Rapid prototyping
- Partitioning: based on co-simulation estimates
- Scheduling
- Synthesis:
  - S-graph (based on a CDFG) based code synthesis for software
  - Logic synthesis for hardware
- Main emphasis on unbiased verifiable specification
Hardware/Software Co-Synthesis

- Functional GALS CFSM model for hardware and software
  - initially unbounded delays refined after architecture mapping
- Automatic synthesis of:
  - Hardware
  - Software
  - Interfaces
  - RTOS

RTOS Synthesis and Evaluation in Polis

1. Provide communication mechanisms among CFSMs implemented in SW and between the OS is running on and HW partitions.
2. Schedule the execution of the SW tasks.
### Estimation on the Synthesis CDFG

```
BEGIN
detect(c)
40  a < b
26  F
41  F
63  T
a := 0
9
14
emit(y)
END
```

### Architecture Evaluation Problem

```
System Behavior
'''
System Architecture
Out of Spec
High Cost
HDL

Time and Money

```

---

**Estimation on the Synthesis CDFG**

**Architecture Evaluation Problem**
Proper Architectural Evaluation

Estimation-Based Co-simulation

Network of EFSMs

HW/SW Partitioning

SW Estimation
HW Estimation

HW/SW Co-Simulation Performance/trade-off Evaluation

In Spec Low Cost
Refine
Time and Money
System Behavior
System Architecture
System Architecture
System Architecture
Co-simulation Approach (1)

- Fills the “validation gap” between fast and slow models
  - Performs performance simulation based on software and hardware timing estimates
- Outputs behavioral VHDL code
  - Generated from CDFG describing EFSM reactive function
  - Annotated with clock cycles required on target processors
- Can incorporate VHDL models of pre-existing components

Co-simulation Approach (2)

- Models of mixed hardware, software, RTOS and interfaces
- Mimics the RTOS I/O monitoring and scheduling
  - Hardware CFSMs are concurrent
  - Only one software CFSM can be active at a time

Future Work

- Architectural view instead of component view
Research Directions in F-A Codesign

- Functional decomposition, cross-“block” optimization ~ hardware/software partitioning techniques
- Task and system level algorithm manipulations ~ performing user-guided algorithmic manipulations

Coverification Methodology & Tools

- Cosimulation
  - Motivations
  - Essentials of Environment
  - Examples: IPC-based, Verilog-C, VHDL-C, DSP
  - Model Continuity Problem
- Main Issues
- Current State-of-Art
  - Solutions & Real Cosimulation Examples
- Coverification Tools
  - Mentor Graphics Seamless CVE
**Cosimulation**

- Motivations
  - Early integration of hardware and software
  - Ease of debugging hardware before fabrication so that software does not have to compensate for hardware deficiencies
  - Hardware and software engineers can work together, rather than against each other
  - Better systems: higher performance, lower cost
  - Reduced design time and efforts: bugs detected early so time is reduced

**Prototype Availability Gap**
Essentials of a Cosimulation Env.

- A hardware simulator
- A software simulator
- Communication interface modeling
  - Memory models
  - Bus models
  - Protocols
- Abstraction mechanisms
  - HW: bus function models
  - SW: instruction set models
- Refinement mechanisms
  - HW: different abstraction levels
  - SW: stub functions
- A user-interface for debugging both HW and SW

Hardware & Software Simulations

- Hardware Simulation
  - FPGA prototype or emulation system
  - RTL level HDL and a logic simulator
  - behavioral models in an HDL and a simulator
  - high level procedural language: C, Java
  - modeling language: UML
- Software Simulation
  - Instruction Set Simulator (ISS)
  - Compiled/executed on host
  - Real CPU
Performance/Accuracy Tradeoffs

- Most widely used:
  - RTL HDL and ISS
  - Very accurate, but very slow

IPC-based Cosimulation

- An HDL (VHDL or Verilog) simulation environment is used to perform behavioral simulation of the system hardware processes

- A Software environment (C or C++) is used to develop the code

- SW and HW execute as separate processes linked through UNIX IPC (interprocessor communications) mechanisms (sockets)
**Verilog Cosimulation Example**

Software processes communicate with hardware simulator via UNIX sockets

Verilog PLI (programming language interface) serves as translator, allowing hardware simulation models to communicate with software processes.

**VHDL Cosimulation Example**

Software processes communicate with hardware simulator via foreign language interface

Allowing hardware simulation models to "cosimulate" with software processes.
VHDL-C Based HW/SW Cosimulation for DSP Multicomputer Application

Algorithm - C

Scheduler - C

Mapping Function (e.g.):
- Round Robin
- Computational Requirements Based
- Communications Requirements Based

Architecture - VHDL

CPU 1
CPU 2
CPU 3
CPU 4

Communications Network

VHDL-C Based HW/SW Cosimulation for DSP Multicomputer Application

Unix C Program

System State (e.g.):
- CPU: Time to instruction completion
- Comm Agent: Messages in Send Queue, Messages in Recv Queue
- Network: Communications Channels Busy

Algorithm/Scheduler

Next Instruction for CPU to Execute (e.g.):
- Send(destination, message_length)
-Recv(source, message_length)
-Compute(time)

VHDL Simulator

Architecture Model
**Model Continuity Problem**

Model Continuity Problem

- Inability to gradually define a system-level model into a hardware/software implementation
- Model continuity problems exist in both hardware and software systems
- Model continuity can help address several system design problems
  - Allows validation of system level models with corresponding HW/SW implementation
  - Addresses subsystem integration

**Main Issues in Cosimulation**

- Logic simulation is significantly slower than software simulation
  - HDL simulator v/s ISS: 100 ~ 1000 times
- Accuracy v/s speed trade-off
  - event-accurate: very fast
  - bus-accurate: fast
  - instruction-accurate: slow
  - cycle-accurate: very slow
- HW-SW debugging
  - microprocessor register values
  - hardware signal values
  - memory contents
- Memory modeling
State-of-Art in Cosimulation

- Imbalance in simulation speeds
  - memory coherency technology
  - cycle hiding: hide sw actions from hw logic simulator
- Accuracy/speed tradeoff
  - Bus function modeling
  - transaction-level modeling, design, verification
- HW-SW debugging
  - Software debugger (UI)
  - In-Circuit emulator (ICE)
  - Hardware simulator
- Memory modeling
  - different kinds of memory models

Real Cosimulation Examples (1)

- Nortel
  - Frequency matching algorithm
  - Incoming signal compared to a reference frequency
  - Hardware:
    - phase comparison
    - pass phase difference to software
  - Software:
    - Munter low pass filter (rate of frequency change)
    - compute new value for VCO (voltage-controlled oscillator)
  - Verification:
    - determine the collaboration of hw-sw is working
    - signals brought back into phase within time limits
    - does not overshoot the target frequency
    - does not oscillate trying to find a match
Real Cosimulation Examples (2)

- Packet Video
  - Effect of hw configurations on MPEG4 encoder/decoder
  - System simulation
    - cache sizes
    - bus clock rate
    - memory speed
    - external bus traffic
  - Verification results
    - optimal instruction size
    - optimal cache size
    - optimal ratio between speed of bus clock & processor clock for a given bus loading

Real Cosimulation Examples (3)

- In-Systems Design (now a division of Cypress Semiconductor)
  - SoC based Ink Jet printer design with VxWorks OS
  - To check through simulation if BSP is working
    - 18 minutes to boot OS: 16 minutes in logic simulator
    - 16 task swaps / sec of simulation time
  - Results:
    - better than expected
Real Cosimulation Examples (4)

- ARM
  - Development of ARM920
  - To validate Windows CE OS runs correctly on ARM920
  - Backward compatibility critical
  - ISS model of Windows CE from Microsoft
  - RTL level HDL model of ARM920
  - Behavioral model of ARM920

Real Cosimulation Examples (5)

- Qualcomm
  - Cellular phone chip set
  - Compatibility to existing cell-phone protocols
  - Modeling mobile station modem chip
    - ISS with IKOS hardware accelerator
    - Behavioral model of a cell base station
    - Test base station acquisition
      - 1 sec of real time ➔ 8 hours of simulation time
  - device drivers, BSP, entire application
  - found a number of hw and sw bugs
  - design cycle shortened
Real Cosimulation Examples (6)

- Hyperchip
  - Optimal communication line cards
  - cards plug into highly parallel switch fabric
  - core routing of optical networks (10^15 bits/s)
  - forwarding & traffic management: several FPGAs
  - PCI interface, CPU with VxWorks
  - host code execution for simulating CPU
  - RTOS VxWorks emulator on Sun workstation
  - Found several bugs in software

Coverification Tools

- Mentor Graphics Seamless Co-Verification Environment (CVE)
Seamless CVE Performance Analysis

References

Codesign Case Studies

- ATM Virtual Private Network
  - CSELT, Italy
  - Virtual IP library reuse
- Digital Camera Design with JPEG
  - Frank Vahid, *Embedded System Design*, Chapter 7
  - Typical SoC

 Codesign Case Studies

INTELLECTUAL PROPERTY RE-USE IN EMBEDDED SYSTEM CO-DESIGN: AN INDUSTRIAL CASE STUDY

E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, A. Sangiovanni-Vincentelli

Centro Studi e Laboratori Telecomunicazioni S.p.A., Italy
Dipartimento di Elettronica - Politecnico di Torino, Italy
EECS Dept. - University of California, Berkeley, USA
ATM EXAMPLE OUTLINE

- INTRODUCTION
- THE POLIS CO-DESIGN METHODOLOGY
- IP INTEGRATION INTO THE CO-DESIGN FLOW
- THE TARGET DESIGN: THE ATM VIRTUAL PRIVATE NETWORK SERVER
- RESULTS
- CONCLUSIONS

NEEDS FOR EMBEDDED SYSTEM DESIGN

- CODESIGN METHODOLOGY AND TOOLS
- EASY DESIGN SPACE EXPLORATION
- EARLY DESIGN VALIDATION
- INTELLECTUAL PROPERTY LIBRARY
- HIGH DESIGN PRODUCTIVITY
- HIGH DESIGN RELIABILITY
THE POLIS EMBEDDED SYSTEM CO-DESIGN ENVIRONMENT

- HW-SW CO-DESIGN FOR CONTROL-DOMINATED REAL-TIME REACTIVE SYSTEMS
  - AUTOMOTIVE ENGINE CONTROL, COMMUNICATION PROTOCOLS, APPLIANCES, ...
- DESIGN METHODOLOGY
  - FORMAL SPECIFICATION: ESTEREL, FSMS
  - TRADE-OFF ANALYSIS, PROCESSOR SELECTION, DELAYED PARTITIONING
  - VERIFY PROPERTIES OF THE DESIGN
  - APPLY HW AND SW SYNTHESIS FOR FINAL IMPLEMENTATION
  - MAP INTO FLEXIBLE EMULATION BOARD FOR EMBEDDED VERIFICATION

THE POLIS CODESIGN FLOW

GRAPHICAL EFSM
ESTEREL
...
FORMAL VERIFICATION
COMPILERS
CFSMs
SW SYNTHESIS
PARTITIONING
HW SYNTHESIS
HW/SW CO-SIMULATION PERFORMACE / TRADE-OFF EVALUATION
SW ESTIMATION
HW ESTIMATION
LOGIC NETLIST
PHYSICAL PROTOTYPING
SW CODE + RTOS CODE
GOALS

➔ ASSESSMENT OF POLIS ON A TELECOM SYSTEM DESIGN
   ➜ CASE STUDY: ATM VIRTUAL PRIVATE NETWORK SERVER

➔ INTEGRATION OF THE POLIS DESIGN FLOW
CASE STUDY: AN ATM VIRTUAL PRIVATE NETWORK SERVER

CRITICAL DESIGN ISSUES

- **TIGHT TIMING CONSTRAINTS**
  - FUNCTIONS TO BE PERFORMED WITHIN A CELL TIME SLOT (2.72 \(\mu\)s FOR A 155 Mbps FLOW) ARE:
    - PROCESS ONE INPUT CELL
    - PROCESS ONE OUTPUT CELL
    - PERFORM MANAGEMENT TASKS (IF ANY)

- **FREQUENT ACCESS TO MEMORY TABLES**
  - THAT STORE ROUTING INFORMATION FOR EACH CONNECTION AND STATE INFORMATION FOR EACH QUEUE
DESIGN IMPLEMENTATION

DATA PATH:
- 7 VIP LIBRARY™ MODULES
- 2 COMMERCIAL MEMORIES
- SOME CUSTOM LOGIC (PROTOCOL TRANSLATORS)

CONTROL UNIT:
- 25 CFSMs

VIPTM LIBRARY MODULES
HW/SW CODESIGN MODULES
COMMERCIAL MEMORIES

ALGORITHM MODULE ARCHITECTURE
### DESIGN SPACE EXPLORATION

- **CONTROL UNIT**
  - Source code: 1151 ESTEREL lines
  - Target processor family: MIPS 3000 (RISC)

- **FUNCTIONAL VERIFICATION**
  - Simulation (PTOLEMY)

- **SW PERFORMANCE ESTIMATION**
  - Co-simulation (POLIS VHDL model generator)

- **RESULTS**
  - 544 CPU clock cycles per time slot
  - $\downarrow$ 200 MHz clock frequency

### DESIGN VALIDATION

- **VHDL co-simulation of the complete design**
  - Co-design module code generated by POLIS

- **Server code:** ~ 14,000 lines
  - VIP LIBRARY™ modules: ~ 7,000 lines
  - HW/SW co-design modules: ~ 6,700 lines
  - IP integration modules: ~ 300 lines

- **Test bench code:** ~ 2,000 lines
  - ATM cell flow generation
  - ATM cell flow analysis
  - Co-design protocol adapters
### CONTROL UNIT MAPPING RESULTS

<table>
<thead>
<tr>
<th>MODULE</th>
<th>FFs</th>
<th>CLBs</th>
<th>I/Os</th>
<th>GATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD TECHNIQUE</td>
<td>66</td>
<td>106</td>
<td>114</td>
<td>1,600</td>
</tr>
<tr>
<td>CELL EXTRACTION</td>
<td>26</td>
<td>35</td>
<td>68</td>
<td>384</td>
</tr>
<tr>
<td>VIRTUAL CLOCK SCHEDULER</td>
<td>77</td>
<td>71</td>
<td>95</td>
<td>1,280</td>
</tr>
<tr>
<td>REAL TIME SORTER</td>
<td>281</td>
<td>731</td>
<td>52</td>
<td>10,504</td>
</tr>
<tr>
<td>ARBITER #1</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>114</td>
</tr>
<tr>
<td>ARBITER #2</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>127</td>
</tr>
<tr>
<td>ARBITER #3</td>
<td>18</td>
<td>9</td>
<td>17</td>
<td>150</td>
</tr>
<tr>
<td>LQM INTERFACE</td>
<td>20</td>
<td>39</td>
<td>27</td>
<td>603</td>
</tr>
<tr>
<td>PARTITION I</td>
<td>409</td>
<td>892</td>
<td>120</td>
<td>13,224</td>
</tr>
<tr>
<td>PARTITION II</td>
<td>443</td>
<td>961</td>
<td>256</td>
<td>14,228</td>
</tr>
</tbody>
</table>

### DATA PATH MAPPING RESULTS

<table>
<thead>
<tr>
<th>MODULE</th>
<th>FFs</th>
<th>CLBs</th>
<th>I/Os</th>
<th>GATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTOPIA RX INTERFACE</td>
<td>120</td>
<td>251</td>
<td>37</td>
<td>16,300</td>
</tr>
<tr>
<td>UTOPIA TX INTERFACE</td>
<td>140</td>
<td>265</td>
<td>43</td>
<td>16,700</td>
</tr>
<tr>
<td>LOGIC QUEUE MANAGER</td>
<td>247</td>
<td>322</td>
<td>31</td>
<td>5,380</td>
</tr>
<tr>
<td>ADDRESS LOOKUP</td>
<td>87</td>
<td>96</td>
<td>32</td>
<td>1,700</td>
</tr>
<tr>
<td>ADDRESS CONVERTER</td>
<td>14</td>
<td>13</td>
<td>17</td>
<td>240</td>
</tr>
<tr>
<td>PARALLELISM CONVERTER</td>
<td>46</td>
<td>31</td>
<td>47</td>
<td>480</td>
</tr>
<tr>
<td>DATAPATH TOTAL</td>
<td>658</td>
<td>1001</td>
<td>47</td>
<td>42,000</td>
</tr>
</tbody>
</table>
WHAT DO WE NEED FROM POLIS?

➡ IMPROVED RTOS SCHEDULING POLICIES
Available now:
- Round Robin
- Static Priority

Needed:
- Quasi-Static Scheduling Policy

➡ BETTER MEMORY INTERFACE MECHANISMS
Available now:
- Event Based (Return to the RTOS on events generated by memory read/write operations)

Needed:
- Function Based (No return to the RTOS on events generated by memory read/write operations)

WHAT ELSE DO WE NEED FROM POLIS?

➡ MOST WANTED: EVENT OPTIMIZATION
Event definition ↓ Inter-module communication primitive
But:
- Not all of the above primitives are actually necessary
- Unnecessary inter-module communication lowers performance

➡ SYNTHESIZABLE RTL OUTPUT
Synthesizable output format used: XNF
Problem: Complex operators are translated into equations
- Difficult to optimize
- Cannot use specialized HW (adders, comparators...)


ATM EXAMPLE CONCLUSIONS

- HW/SW CODESIGN TOOLS PROVED HELPFUL IN REDUCING
  DESIGN TIME AND ERRORS
  
  CODESIGN TIME = 8 MAN MONTHS
  STANDARD DESIGN TIME = 3 MAN YEARS

- POLIS REQUIRES IMPROVEMENTS TO FIT INDUSTRIAL
  TELECOM DESIGN NEEDS
  
  EVENT OPTIMIZATION + MEMORY ACCESS + SCHEDULING POLICY

- EASY IP INTEGRATION IN THE POLIS DESIGN FLOW
  
  FURTHER IMPROVEMENTS IN DESIGN TIME AND RELIABILITY