ECEN 5003-001: Evaluating Processor Microarchitectures

Manish Vachharajani
Fall 2004
Contribution of Microarchitecture

Graph courtesy Sanjay Patel, data courtesy Intel
The Hardware Design Process

- Microarchitecture level design
  - Microarchitecture difficult to design
  - Design decisions have major implications
  - Expensive to correct shortcomings later

Cost of microarchitecture level changes: Cheap to Expensive

Design Criteria

Microarchitecture → RTL → Gates → Transistors → Layout

Layout for Fabrication

4-6 yrs
Microarchitecture Design Process

- Only 1 major design concept is tried
  - Only minor variations such as cache size and branch prediction explored
- Simulator is inaccurate [Black ‘98, Gibson ‘00, Intel ‘04]
  - Predictions off by 10%-30%
The Liberty Simulation Environment: Rapid High-level Modeling of Microarchitecture
Managing Hardware Complexity During Design

- Designers reason about the design through decomposition
- Design is refined by adding and improving components
- Components are complex, thus time-consuming to specify
A Natural Specification Language

- Modularize the model like HW
- Basic components
  - Concurrent computation
  - Communication through ports
- Called structural modeling

- Compare to C/C++ approach
  - Function encapsulation ≠ hardware block encapsulation
  - Requires re-divide and re-conquer
  - Leads to [MICRO-35]
    - Inaccuracies
    - Long development times
Accelerate Modeling with Reuse

- Reuse components to amortize costs [Charest ’02, Emer ’02, Koegst ’98]
- e.g., Itanium 2 Model, 3% error, 11 person-weeks
  - 80% of components from library, similar for other models
- Low-overhead customizability critical for this reuse [Radetzki ’98]
Realities of Reuse

- Cannot reuse blocks that cannot be modularized
  - E.g. timing control has global pipeline knowledge
- Reuse requires highly flexible components, but ...
  - Flexible components add too much specification overhead [Radetzki '98]
  - Often arises due to parameterization overhead
- Addresses issues via
  - Modularization strategy for timing control
  - Techniques that simultaneously allow
    - Flexible components
    - Statically analyzable model structure
    - Inference of component parameters to reduce overhead
- Liberty Simulation Environment (LSE)
  - Input language called Liberty Structural Specification Language (LSS)
  - LSS and LSE incorporate the above techniques
Component Flexibility
Polymorphism

- Many state and routing components
  - Share identical functionality
  - Store different data types
- Polymorphism allows components to adapt
- But, polymorphism forces over 101 type instantiations for the Itanium 2
Polymorphism specified via type variables
- response:'a is a polymorphic port with type variable 'a

Structure can be used to infer types
- response's connection to icache_line resolves type to bytes[linesize]
- Common 'a type variable resolves request as well

With inference Itanium 2 needs 38 type instantiations vs. 101
Component Flexibility
Parametric Customization of Structure

- Different levels model refinement require different number of ports
- Variation in ports requires varying structure
  - One register read port requires one MUX
  - Two register read ports requires two MUXes
Component Flexibility
Parametric Customization of Structure

module register_file {
    parameter num_reads:int;
    inport read:read_req;
    inport write:write_req;

    instance data_array:...;
    instance muxes:
        mux[num_reads];
    for(i=0;i<num_reads;i++){
        data_array.out[i] ->
        muxes[i].in[0];
    }
    ...
};

Flexible interface add parameterization overhead
- The Itanium 2 model needs 523 interface sizing parameters
Lowering Overhead
Use-based Specialization

- Infer parameters from usage (i.e. use-based specialization)
- Other customizations based on usage
  - Types – if auxiliary data ports connected, output type is a struct
  - Semantics – If branch target port is connected a BTB is instantiated
- Eliminates 523 interface size size parameters for I2 model
Modularization for Timing Control
Timing Control Example

- Timing control handles stalling
- Timing control is logically centralized
  - Controller has global knowledge
Even simple data path changes require updating the controller
Controller cannot be reused!
Timing Control and Existing Modeling Systems

- Existing work treats timing-control as a global entity
- Control neutrality
  - SystemC, Objective VHDL, UPFAST, HASE, etc.
  - Approaches focus on other problems, not timing-control
- Specification of global timing controller
  - Generality versus complexity tradeoff
  - Template-based
    - LISA [Pees ’99], RADL [Siska ’98]
    - Very limited architecture class
  - Alternative representations
    - Expression [Mishra ’01], MADL [Qin ’02]
    - Generality vs. complexity
- Modularize control instead
The Components of Timing Control

Stall Detection
- Structural stall conditions
- Semantic stall conditions

Stall Distribution
- Back-pressure stall distribution
- Peer stall distribution

Timing Control Tasks
- Stall detection – when to stall
  - Structural stall conditions
  - Semantic stall conditions
- Stall distribution – what to stall
  - Back-pressure stall distribution
  - Peer stall distribution
Stall Detection

- **Semantic stalls**
  - Stall condition varies as semantics change
  - Data hazards, control hazards, etc.

- **Structural stalls** –
  - Stall condition invariant across different semantics
  - No buffer slots, bus arbitration loss, etc.
  - Should be reusable!

![Diagram showing stall detection processes](image-url)
Stall Distribution

- Stalls propagate along the datapath
- Back pressure
  - Stall earlier blocks in the pipeline
  - Follows opposite direction of datapath
- Coordination
  - Stall peers in the pipeline
  - Usually follows datapath at fanout nodes
Modularizing Backpressure Stall Distribution

- Reverse control signal for backpressure stalls
Modularizing Structural Stall Detection

- Structural stalls encapsulated in reusable components
Modularizing Peer Stall Distribution & Coordination

- Forward control signal handles peer stalls
- Coordination controlled at fanout
  - Reasonable default semantics for almost all components
  - User over-ridable

Coordination Logic

Diagram showing the flow of signals between elements such as source, 3 slot buffer, block 2, block 3, comp, and comp2.
Timing Control Modularization Summary

- Control abstraction makes 3 of 4 portions reusable
- Other approaches good at semantic stalls
  - Expression, MADL, etc. can be leveraged!
End