CS294-7: Reconfigurable Computing

Day 28: Thursday, May 1

Everything Else...

- Admin
- Programming Style Survey and Discussion
- HKN Evaluations

Admin

- Larus/Profiling — Today at 4pm, Soda 306
- Architectural Retiming talk — Friday 11am, Hogan Room
- Project: T -5 days
  - URLs by class time Tuesday
  - All presentations in 505 (even T afternoon)
  - 10 Minute talk slots (2-3 minutes questions)
- HKN at end of class

Programming

Goals: (what want out of a programming style?)

- Easy to develop applications (modify, maintain)
- Portable
- Adapt to varying performance requirements and resource availability
- Yield good performance (high fraction of raw computational density)
- ? Fast translation ?
- ? Predictable performance ? (simple "abstract" performance model)
- Machine invisible to programmer?
- Development Environment (debugging, etc.)

Review: Know

What do we know how to do (from course):

- LUT mapping
- Retiming – minimize cycle time, c-slow, multicontext
- Partition
- Place and Route (static communication patterns)
- *Operator Area/Time tradeoffs ?*
- (simple) specialization

Things that make life harder

Overspecification: Specify too much of implementation—limit transformations which can produce better mapping.

- code bubble sort

Underspecification: Specify too little such that tools cannot find a good solution (in reasonable time).

- L, R such that $L \cup R = X$, $0.95|R| <= |L| <= 1.05|R|$ $L_{outs} \cap R_{ins} \cup R_{outs} \cap L_{ins}$ is minimized
- Retime such that C minimized.

Hard?

Things that seem hard today:

1. undoing *overspecification*
   - sequentialization
   - segregating memory pools
   - changing visible timing
   - excessive data width
2. finding good mappings when problem *underspecified*
   - picking adequate data size (?)
   - locality clustering
   - discover structure/commonality
   - FSM partitioning

Translation Tasks

If program at a different level than final implementation, will need to perform some translations to bridge gap.

Dependency identification	Placement
Exclusivity identification	Allocation
Parallelism extraction	Specialization
Sequentialization	Scheduling
Sharing and generalization	Time-space selection
Regularity identification	

Programming Styles

1. Structural netlist / RTL (schematic, structure HDL)
2. Behavioral Verilog/VHDL
3. Sequential Programming Language
4. x86 assembly code
5. Synchronous Data Flow
6. Dynamic Dataflow
7. Multithreaded / Communicating Sequential Processes
8. Data Parallel (SPMD)
9. Direct mapping from domain specific language
10. Mixed Style
11. Evolve/Genetic Programming
12. DWIM?

Structural netlist Styles

User specifies structural composition of primitive operators (gates/LUTs) (and library built from primitives).

netlist $\rightarrow$ LUT Mapping $\rightarrow$ scheduling $\rightarrow$ place $\rightarrow$ route

Structural netlist Styles

Good

- know how to do
- Parallelism exposed
- easier than raw bit-stream
- Clear how to optimize
- Rewards wizards?

Bad

- difficult for user to construct
- hard to rediscover structure (area-time mapping)
- hard to rediscover exclusivity
- ? layout quality ?

Behavioral HDL

User specifies behavior (Verilog, VHDL, Silage,...)

behavioral spec $\rightarrow$ RTL architecture

Behavioral HDL

Good

- Specify less details
- easier
- more mapping freedom
- parallelism exposed
- supports successive refinement

Bad

- easy to overspecify timing
- difficult for SW enineers
- hard to train user
- parallelism exposed
- tools not good at open architectural space exploration
- Quality = Designer $\cap$ Tool

Sequential Language

User writes code in favorite programming language (*e.g.* C, Fortran, Cobol, Scheme, Java, x86 assembly)

C $\rightarrow$ control/dataflow graph $\rightarrow$ RTL architecture

Sequential Language

Good

- Everyone knows how
- Dusty Deck Code
- little improvements v/s pure sequential execution

Bad

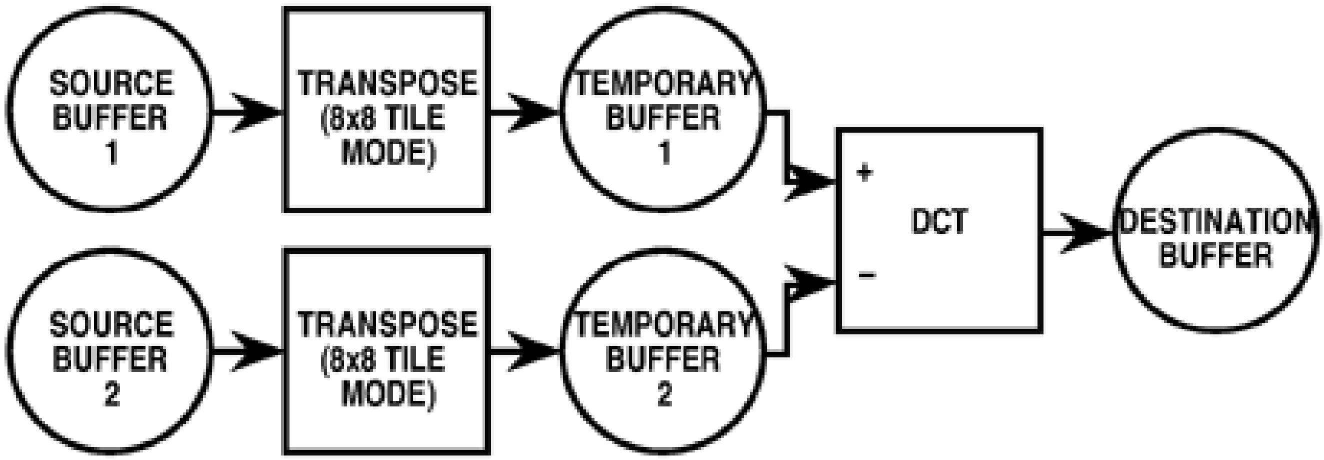
- overspecified sequentialization
- overspecified memory sharing
- dependency extraction hard
- parallelism extraction hard
- limited/mismatched datatypes
- wrong cost model
- hard to discover regularity

Synchronous Data Flow

User strings together dataflow between high-level operators; i/o ratio constant

n inputs $\rightarrow$ m outputs

SDF $\rightarrow$ scheduling $\rightarrow$ RTL architecture



Synchronous Data Flow

Good

- scheduling/buffering easy
- parallelism exposed
- select operators tailored to target
- flow/regularity natural match RC (spatial computing)
- ______________________________

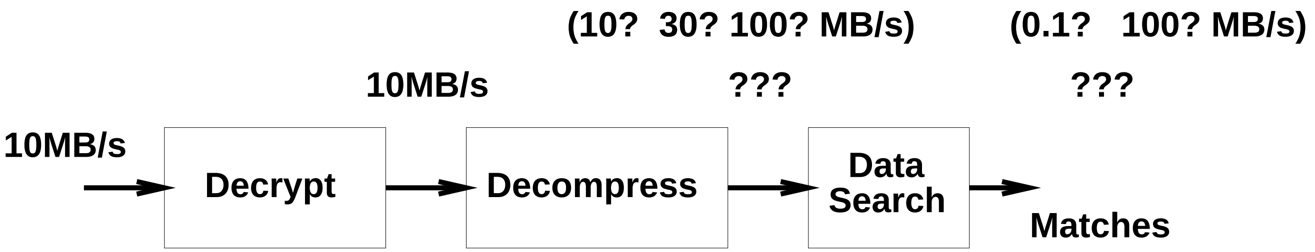
Bad

- Limited model
- How specify operators?
- ______________________________
- ______________________________
- ______________________________

Dynamic Dataflow

User strings together dataflow between operators.
Relative activation rates unknown.

DDF $\rightarrow$ schedule activations $\rightarrow$ RTL architecture
...including swapping control



Dynamic Dataflow

Good

- less restrictive model
- parallelism exposed
- select operators tailored to target
- ________________________
- ________________________

Bad

- unpredictable flow
- need online scheduling
- (profiling help?)
- presence/scheduling overhead
- online allocation/area-time select

Multithreaded / Communicating Sequential Processes (FSMD's)

Task composed of a collection of communicating computation blocks.
User specifies processing and i/o in each compute block.

process collection $\rightarrow$ group processes to FSMD
$\rightarrow$ synthesize each FSMD $\rightarrow$ RTL architecture

process collection $\rightarrow$ synthesize each as FSMD
$\rightarrow$ RTL architecture $\rightarrow$
...with online allocation/placement/scheduling

Multithreaded / Communicating Sequential Processes (FSMD's)

Good

- potentially even less restrictive
- gross parallelism exposed
- ______________________
- ______________________
- ______________________

Bad

- unpredictable flow/activation
- need online scheduling and allocation
- how specify process?
- (profiling help?)
- online communication scheduling overhead
- online allocation/area-time select

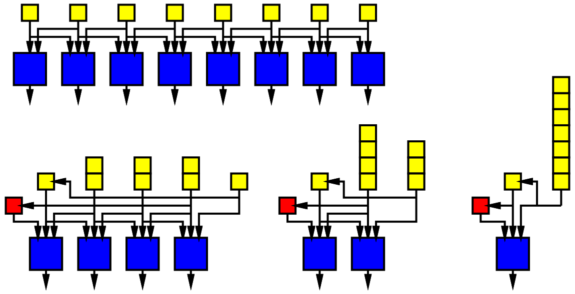
Data Parallel (SPMD)

SPMD: Single Program Multiple Data

User specifies common operations on aggregates.

SPMD program → aggregate dataflow
→ RTL architecture

(probably rotate SPMD data parallel operations into sequential streams;
pick parallel width based on relative throughput requirement — *e.g.* image processing folding example)



SPMD

Good

- captures data parallelism
- enable aggregate area-time tradeoffs
- ____________________
- ____________________
- ____________________

Bad

- overspecified sequentialization
- overspecified memory sharing
- inter-aggregate dependency extraction hard
- inter-operator parallelism extraction hard
- wrong cost model
- ____________________

Direct Problem Mapping

User specifies an instance of a problem domain; Instance mapped directly to implementation.

Problem Instance $\rightarrow$ RTL architecture

Translator written specificly for problem/architecture.

e.g.

- FIR Generator: k, $\{C_0, C_1, \ldots C_k\} \rightarrow$ circuit
- SPICE circuit simulator
- n-body problem (Supercomputer toolkit)
- protocol compiler
- FSM compiler?

Direct Problem Mapping

Good

- Minimal architecture bias
- Easy for user to specify
- Mapping tailored to device
- Can provide general framework?

Bad

- Someone must write problem mapper
- Each mapper solve only particular problem

Specialized Mappers via Partial Evaluation

User specifies how a computation is done, taking both instance definition and data as input.

User Spec + Problem Instance $\rightarrow$ RTL architecture

User Spec specialized to problem instance

e.g.

- SPICE evaluator + circuit $\rightarrow$ simulator
- n-body simulator + instance $\rightarrow$ simulator

Specialized Mappers via Partial Evaluation

Good

- More general technique
- Don't have to have an expert to handle new problem domain
- can expose much parallelism
- ________________
- ________________

Bad

- Easily run into semantic limitations of source programming language (may need to restrict)
- general behavioral mapping problems
- ________________
- ________________

Programming Styles

Programming styles are a balance of:

- richness to express problem (efficiently)
- restrictions
  - to support automated reasoning/optimization
  - to make mapping tractable
- minimize what programmer has to express
  - ease programming task
  - avoid overspecification
- allow programmer to specify enough to allow mapping
  - avoid underspecification