Behavioral Synthesis and the Generalized Reconfigurable Architecture Compilation Engine (GRACE)

Rick Copeland

Center for Research in Embedded Systems and Technology

rick.copeland@ece.gatech.edu

Outline

- **Overview of Behavioral Synthesis**
  - Motivation
  - Problem Definition
  - Behavioral Optimization
  - Commercial Examples
- Exploiting Parallelism
  - Thread & loop level parallelism
  - Data parallelism
  - Iteration level parallelism
  - Instruction Level Parallelism
- Introduction to GRACE
  - Overall Architecture
  - Implementation Mapping
  - Scheduling
  - Pipelining
  - Control synthesis & code generation

Behavioral Synthesis Motivation: Productivity

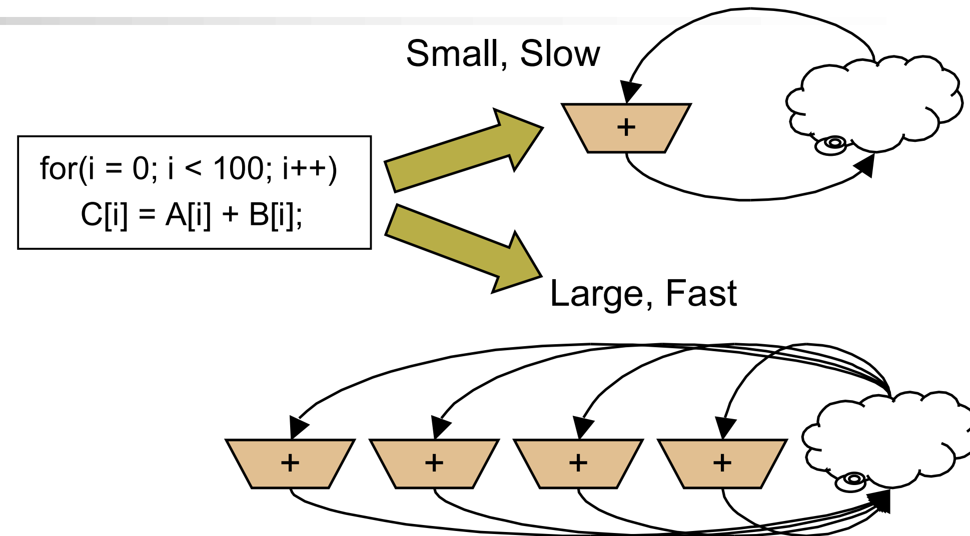
```
int mult32(int a, int b) {
    return a * b;
}
```

- Productivity
  - Replace verbose HDL description…
  - With concise high-level language description

Benefits

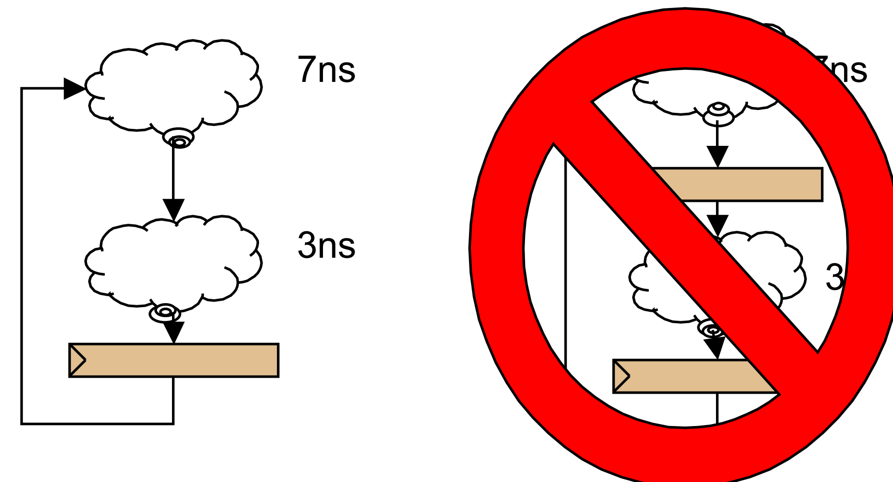
- Faster implementation
- Fewer lines of code to introduce bugs

Behavioral Synthesis Motivation: Explore Architectural Trade-offs



- Area vs. Speed

- Clock frequency versus throughput



Behavioral synthesis can explore in minutes what would take an RTL designer weeks

Other Behavioral Synthesis Motivations

- Consistency of high-level specification with low-level implementation
  - Most designers today use extensive test suites to guarantee consistency, even in “system level” (ESL) design flows
  - Synthesis guarantees consistency of implementation with specification (“correct by construction”)
- Better integration with application designers
  - Application-level designers know and use behavioral languages: C/C++, Matlab
  - Functional designers today manually map behavior to RTL: VHDL/Verilog which application designers *cannot read*
  - Behavioral synthesis allows application designers greater visibility into implementation

Behavioral Synthesis Problem Definition

Translate a ***behavioral model*** into an ***RTL implementation*** which

- preserves the semantics of the behavioral model and
- meets specified performance, power, and area constraints

- ***Behavioral model*** – an untimed, functional description of a circuit
  - Examples: C/C++ programs, Matlab functions
  - Higher-level language → more productive
  - Assumes standard architecture model (CPU+memory)
- ***RTL implementation*** – a cycle-accurate model of a circuit expressed in terms of registers and the data flowing between them
  - Examples: VHDL/Verilog concurrent assignments
  - Lower level → closely tied to actual implementation
  - Wide range of architectures supported

Behavioral Synthesis: Problems to Solve

- Synthesize architecture
  - Datapath (functional units)
  - Memory blocks
  - Interfaces (busses, FIFOs, etc)
- Map application to architecture
- Schedule mapped application & generate control
  - Pipelining
  - Retiming
  - Control generation
  - Loops and conditional code blocks present special problems

Behavioral Optimizations: Tree Height Reduction

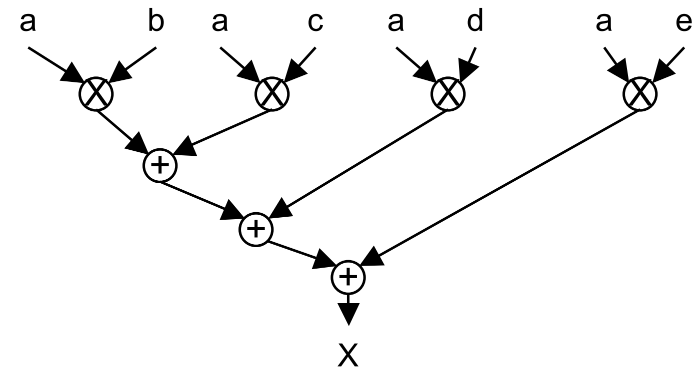
- Use commutative, distributive, and associative properties to exploit hardware parallelism
- Commutative: a • b = b • a
- Distributive: a * (b+c) = a*b + a*c
  - Used to reduce # of operations (and therefore resource requirements)
  - Tree height remains the same
- Associative: (a+b) + c = a + (b+c)
  - Minimize the number of expressions that must be evaluated in series (tree height decreases)
  - Total # of operations is unaffected

- Example:

X = a*b + a*c + a*d + a*e

really means ➔ X = ((a*b + a*c) + a*d) + a*e

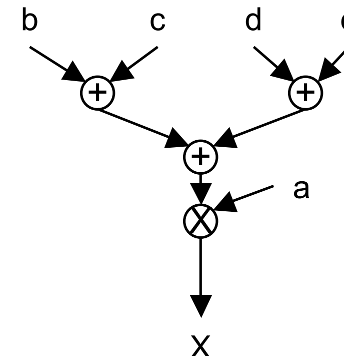
with the graph ➔



but we can factor ➔ X = a * (((b+c) + d) + e)

and re-parenthesize ➔ X = a * ((b+c) + (d+e))

to get the graph ➔



Behavioral Optimization: Copy Prop, Const. Fold, Dead Code Elim

- Constant folding – evaluate as many expressions at *compile time* as possible
  - Example: c = 5+9 ➔ c = 14
- Copy propagation – propagate values through copies
  - Example: a=5; b = a; c = b + d ➔ a = 5; b = 5; c = 5 + d
- Constant folding & copy propagation often work together!
  - Example
    - a = 5; d = 9; b = a; c = b+d ➔ (copy propagate) ➔
    - a = 5; d = 9; b = 5; c = 5+9 ➔ (constant fold) ➔
    - a = 5; d = 9; b = 5; c = 14
- Dead code elimination – remove code that has no effect
  - Example: a=5; d = 9; b = 5; c = 14 ➔ c = 14 (if a,d,b are unused elsewhere)
  - Can remove calculations for unused expressions
  - Can remove if-then or else- blocks, zero-iteration loops, uncalled functions, etc.

Behavioral Optimization: Other standard optimizations

- Common subexpression elimination
  - Identify redundant computation of the same value & remove
  - Example
    - a = x+y; b = a + 1; c = x+y ➔
    - a = x+y; b = a + 1; c = a
- Strength reduction
  - Identify places where an “expensive” operation can be performed by “cheap” operator(s)
  - Example
    - a = x^2; b = 3 * x ➔
    - a = x * x; b = (x<<1) + x;
- Array scalarization
  - Identify arrays that are accessed only with constant indices & replace with scalars
  - Example
    - x[0] = a+b; x[1] = c+d; x[2] = x[0] + x[1] ➔
    - x_0 = a+b; x_1 = c+d; x_2 = x_0 + x_1

Behavioral Optimization: Loop Transformations

- Loop unrolling
  - Execute loop body more than once each iteration, lowering the number of loop iterations
  - Example
    - for(i = 0; i < 100; i++) c += a[i]; ➔
    - for(i = 0; i < 100; i+=4) c = c + a[i] + a[i+1] + a[i+2] + a[i+3];
- Loop fission/fusion
  - Split/combine loops which have no dependencies
  - Example
    - for(i = 0; i<100; i++) { c += a[i]; b *= d[i] } ⇔
    - for(i=0;i<100;i++) { c += a[i]; }
      for(i=0;i<100;i++) { b += d[i]; }
- Others
  - Loop interchange – switch inner and outer loops
  - Loop tiling – reduce/regularize memory footprint of loop
  - Loop skewing – reduce inter-iteration dependencies
  - Outer unrolling (“unroll-and-jam”), reduction unrolling

Behavioral Optimization Example

```
for j in 0 to num_blocks:
   local w:array{unsigned}[17]
   @unroll()
   for ii in 0 to 17:
      w[ii] = 0
```

unroll

```
for j in 0 to num_blocks:
   local w:array{unsigned}[17]
   ii=0
   w[ii] = 0
   ii=1
   w[ii] = 0
   ii=2
   w[ii] = 0
   …
```

Constant Fold / Copy Propagate

```
for j in 0 to num_blocks:
   local w:array{unsigned}[17]
   w[0] = 0
   w[1] = 0
   w[2] = 0
   …
```

Scalarize

```
for j in 0 to num_blocks:
   w_0 = 0
   w_1 = 0
   w_2 = 0
   …
```

Behavioral Synthesis: Commercial Examples

- ASIC-targeted
  - Synopsys Behavioral Compiler (no longer supported)
    - Translate “behavioral” VHDL to RTL
  - Forte Design
    - Translate C (now SystemC) to RTL
  - Tensilica
    - Translate C to CPU + custom functional units
  - Synfora
    - Translate C to CPU + custom loop accelerators
- FPGA-targeted
  - Proceler (now defunct)
    - Translate C to FPGA netlist, focus on loop transformations
  - Celoxica
    - Translate C to HandelC to FPGA netlist

Outline

- Overview of Behavioral Synthesis
  - Motivation
  - Problem Definition
  - Behavioral Optimization
  - Commercial Examples
- **Exploiting Parallelism**
  - Thread & loop level parallelism
  - Data parallelism
  - Iteration level parallelism
  - Instruction Level Parallelism
- Introduction to GRACE
  - Overall Architecture
  - Implementation Mapping
  - Scheduling
  - Pipelining
  - Control synthesis & code generation

Exploiting Parallelism: Motivation

- Speed is constrained at the lowest level by
  - Clock rate
  - Voltage
  - Functional unit architecture (fast CLA vs. ripple-carry)
    - “bit-level parallelism”
- Behavioral synthesis seeks to use the silicon more *efficiently*
  - Larger percentage of transistors doing useful work per cycle
- This is accomplished by the exploitation of various types of *parallelism*
  - Thread level
  - Instruction and data level
  - Iteration level

Exploiting Parallelism: Thread Level Parallelism

- Control split into multiple “threads”, which execute in parallel
- Usually exposed by programmer (e.g. pthreads)
- Can be inferred by compiler in some cases
  - No dependencies exist between program regions
    ➔ regions can execute in parallel
  - Inter-thread dependencies can sometimes be effectively managed by FIFOs
- Example

```
for(i = 0; i<100; i++)
    c += a[i];
for(i = 0; i < 100; i++)
    d += b[i];
➔
T0 = thread {for(i = 0; i<100; i++) c += a[i]; };
T1 = thread {for(i = 0; i<100; i++) d += b[i]; };
T0.start(); T1.start();
T0.join(); T1.join();
```

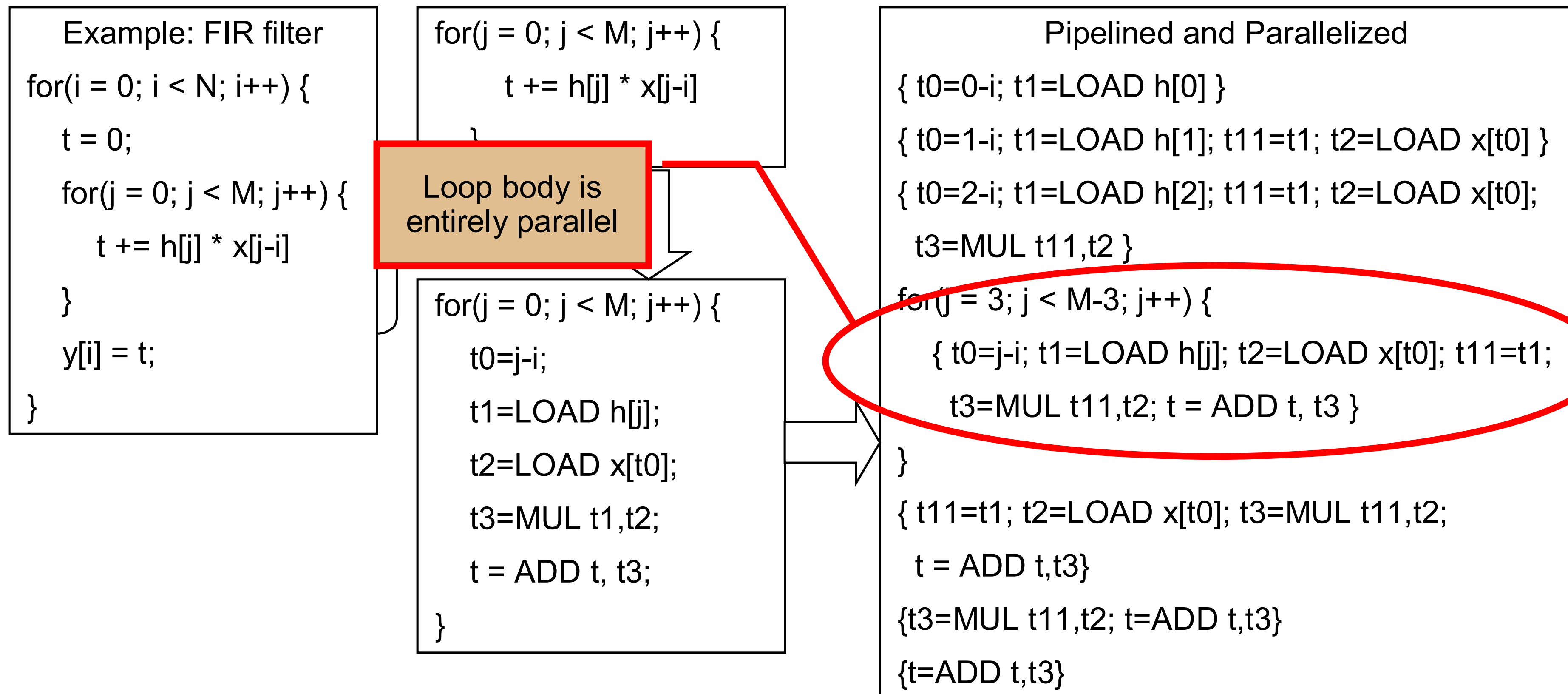
- Often useful in conjunction with loop distribution

Exploiting Parallelism: Instruction Level Parallelism

- When instructions have no dependencies between them, they can run in parallel
  - Example: a = b + c; d = e * f
- Loop unrolling can be used to increase instruction level parallelism:
  - for(i = 0; i < 100; i++) { A[i] = B[i] + C[i]; } ➔
  - for(i = 0; i < 100; i+=2) { A[i] = B[i] + C[i]; A[i+1] = B[i+1] + C[i+1]; }
- ***Data-level parallelism*** is a special case of instruction level parallelism
  - ***Same*** instruction repeated with different operands
    - Maybe spread over loop iterations: for(…) { c = A[i] * 4; }
  - Efficiently executed on SIMD or vector hardware in CPUs

Exploiting Parallelism: Iteration Level Parallelism

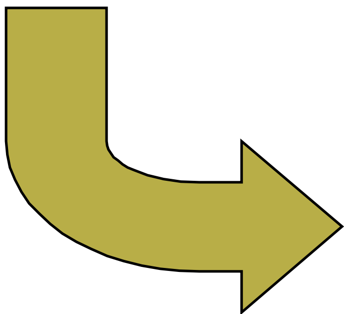
- Exploit parallelism between iterations of loops
  - Loop unrolling converts iteration-level parallelism to instruction-level parallelism, at the expense of additional resources
  - Pipelining exploits iteration-level parallelism without unrolling

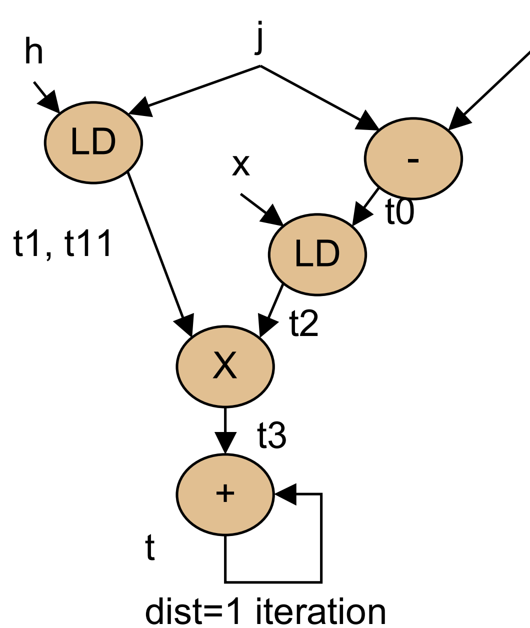


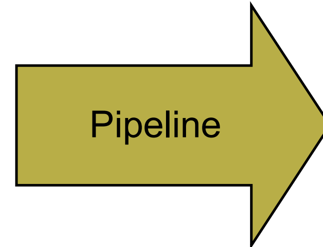
Exploiting Parallelism: Iteration Level Parallelism in Hardware

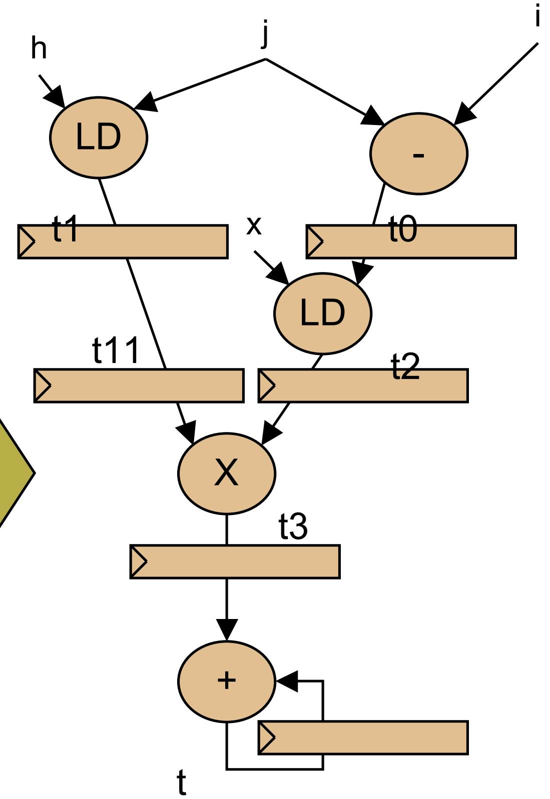
```
for(j = 0; j < M; j++) {
    t0=j-i;
    t1=LOAD h[j];
    t2=LOAD x[t0];
    t3=MUL t1,t2;
    t = ADD t, t3;
}
```

Hardware pipelining is significantly less complex to implement than software pipelining



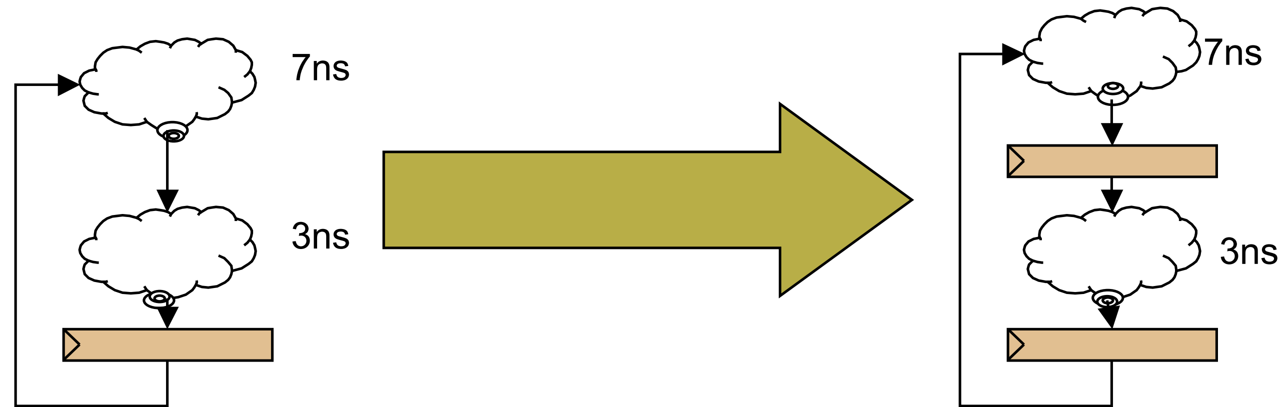


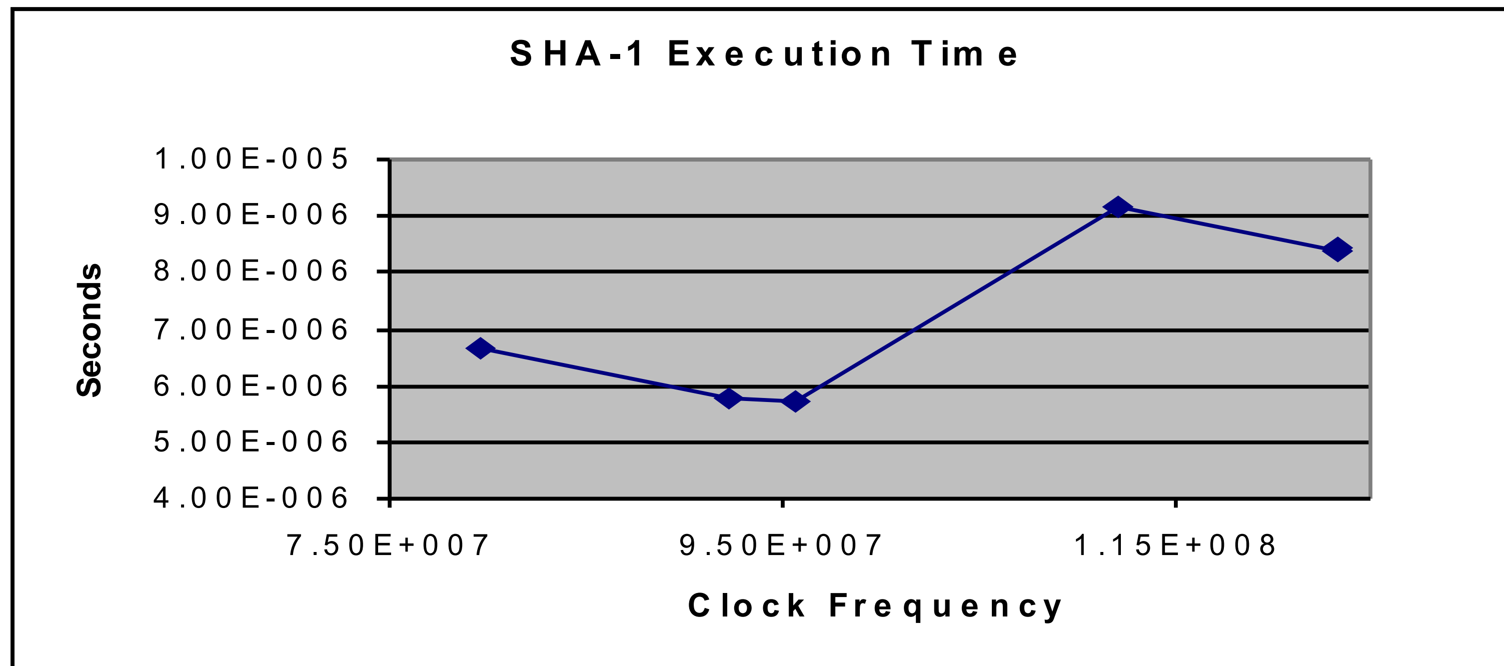




Exploiting Parallelism: Pipelining Caveat

- Sometimes too-deep pipelining can slow down a design!



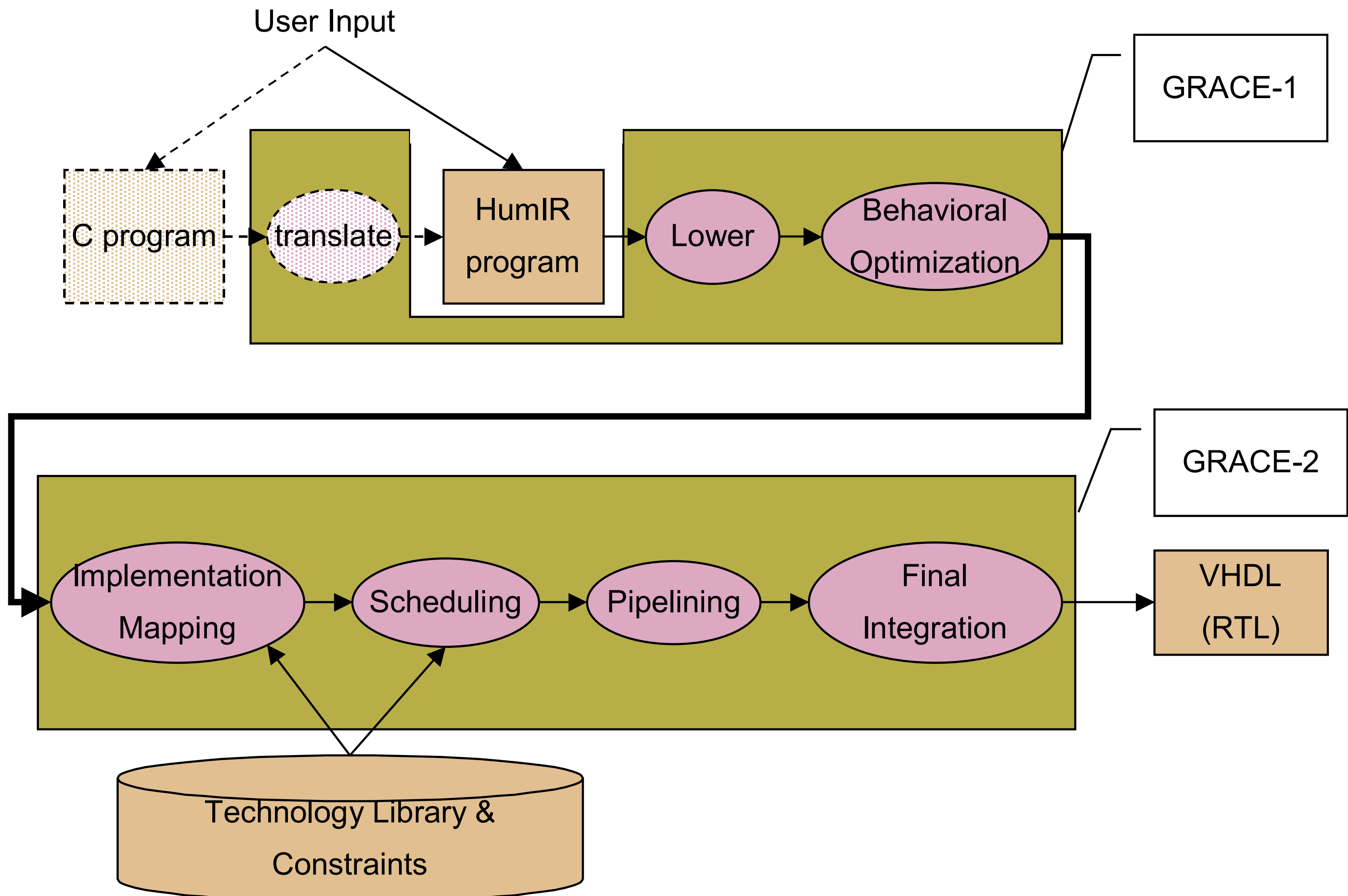


Outline

- Overview of Behavioral Synthesis
  - Motivation
  - Problem Definition
  - Behavioral Optimization
  - Commercial Examples
- Exploiting Parallelism
  - Thread & loop level parallelism
  - Data parallelism
  - Iteration level parallelism
  - Instruction Level Parallelism
- **Introduction to GRACE**
  - Overall Architecture
  - Implementation Mapping
  - Scheduling
  - Pipelining
  - Control synthesis & code generation

GRACE Architecture

Generalized Reconfigurable Architecture Compilation Environment



GRACE Input Language

- Main language is Human-readable Internal Representation
- Syntax similar to Python
  - Block structure denoted by indentation
  - Basic type inference ➔ some type declarations can be omitted
    - (Still statically typed!)
  - Standard arithmetic, logic, and comparison operators
  - Control structures
    - Function – all function calls are currently inlined
    - Thread – like a function+function call, but caller does not wait for thread to return
    - While loop – general looping construct
    - For loop – specialized loop for loops with known bounds
      - for ***ctr*** in ***min*** to ***max*** by ***step***: …
    - If/elif/else – conditional code execution
      - implemented in hardware by predication
- Prototype C implementation
  - *Not* ready for public use yet

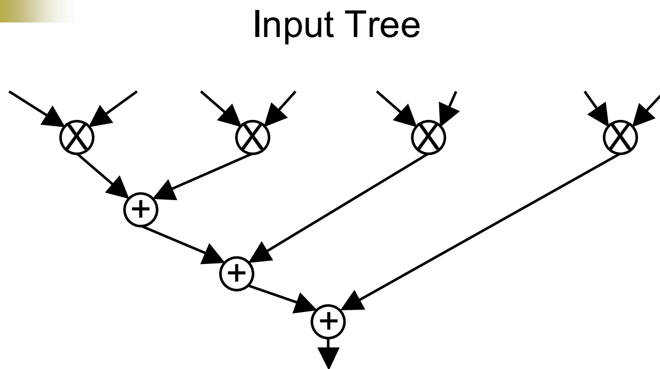
HumIR Example

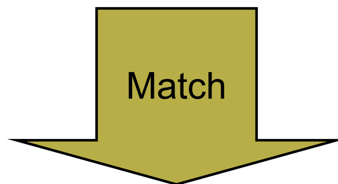
```
fifo_pc:fifo{unsigned}
fifo_out:fifo{unsigned}
() = function producer_consumer():
    thread:
        for i in 0 to 100:
            push{‘fifo_pc’}(i,0)
        push{‘fifo_pc’}(100,1)
    while not eos{‘fifo_pc’}():
        value = pop{‘fifo_pc’}()
        stream_eos = eos{‘fifo_pc’}()
        push{‘fifo_out’}(value, stream_eos)
```

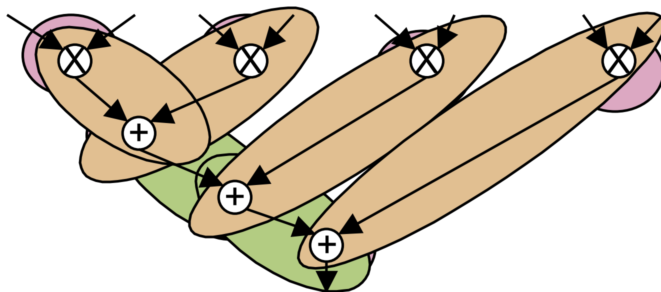
GRACE Implementation Mapping

- Problem definition
  - Given the technology-neutral IR of a program, map the IR onto functional units from a technology-specific library such that
    - The semantics of the input IR are preserved
    - Some cost function is minimized
- GRACE approach: dynamic programming and tree matching
  - Represent IR as graph
  - Break feedback edges in graph (loop carried values) to convert into a DAG
  - Split graph on multiple-fanout vertices to create a forest of trees
  - Find every template in the technology library that matches any subtree → O(N) operation due to use of tree-matching FSM
  - Choose mappings based on cost function
    - Current approach minimizes area

GRACE Implementation Mapping: Example



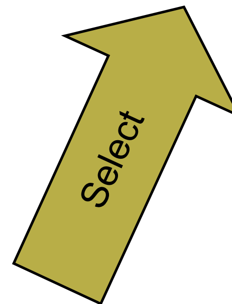


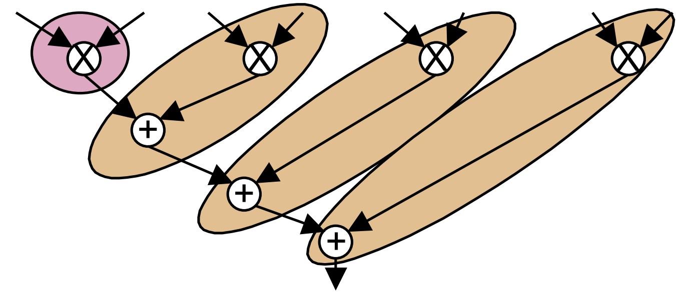


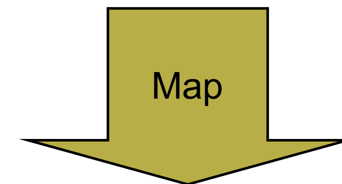
Single-operator matches

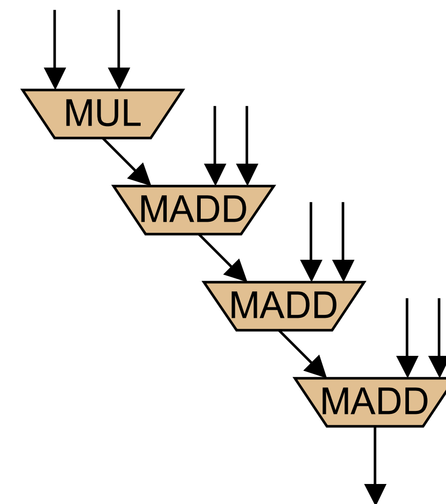
3-input adder

Multiply-add









GRACE Scheduling & Resource Allocation

- Problem Definition
  - Given a technology-specific ***IR block***, generate an assignment of operations to physical resources and times such that
    - all dependencies are satisfied, and
    - some cost function is minimized
- ***IR block*** – the IR representation of a function, loop, or thread, with any nested functions, loops, or threads represented as single vertices
- GRACE Approach
  - Assume 1-1 mapping of IR nodes to physical resources
    - Future work includes multiplexing physical resources between IR nodes
  - Use modulo scheduling (from software pipelining) to generate a schedule the the block (list scheduling would be OK for functions and threads)
  - Initiation interval is minimized, followed by latency
  - NP-hard problem ➔ use heuristics and budget time to do “good” job in O(N) steps

Modulo Scheduling

- IR has loop carried dependencies annotated with their loop carry distance
- For each vertex type, the technology library specifies, for each port:
  - A required/available time (“delay”) relative to the “schedule” time for its vertex
  - A latency (for pipelined nodes) relative to the “schedule” time
  - A distance (for nodes that carry values across loop iterations)
- For each edge, a delay, latency, and distance value is calculated based on the source (output) port and the destination (input) port
  - delay = src.delay + dst.delay
  - latency = src.latency – dst.latency
  - distance = src.distance + dst.distance

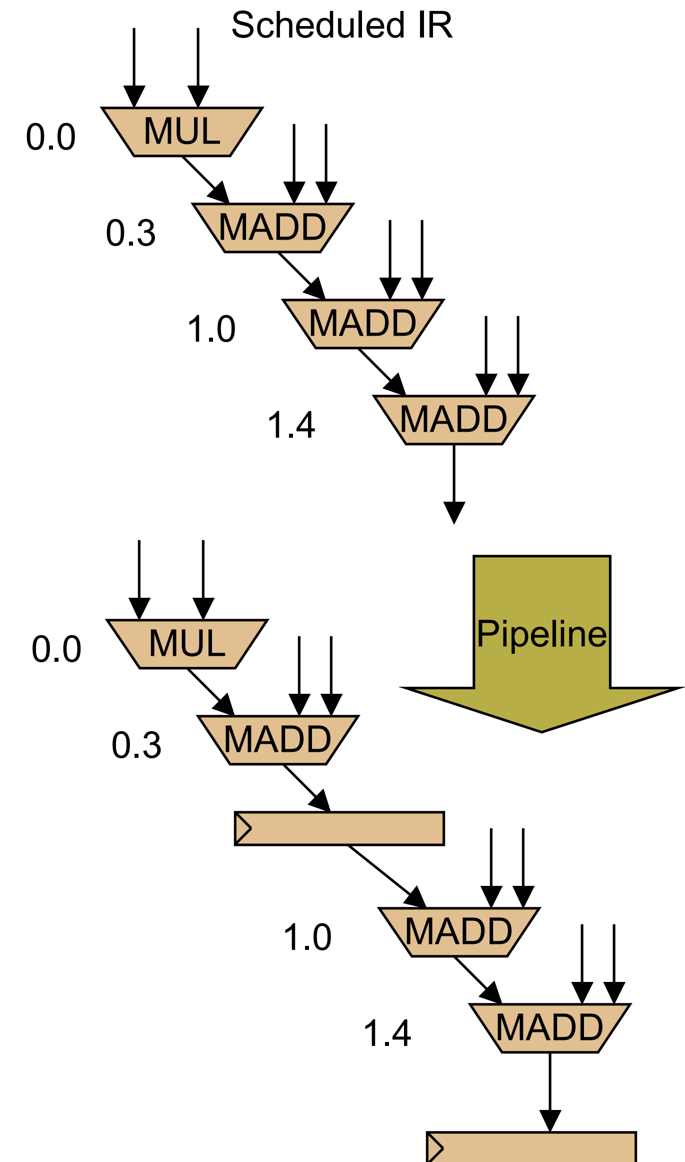
Iterative Modulo Scheduling (simplified)

- Pick a target II
- Calculate effective delay for each edge based on
  - eff_dly = latency + delay * frequency – II * distance
  - This denotes a lower bound on the difference in schedule times for the source and destination nodes
- While there are unscheduled nodes and the time budget > 0
  - Pick a schedule time for a vertex such that all its predecessor constraints are satisfied.
  - Unschedule all vertices that have unsatisfied dependences with the newly-scheduled node
- If the time budget is exhausted, increase the II by 1 and try again

- *Note –*
  - *Schedule times may be non-integers*
  - *Combinational nodes are not permitted to be scheduled with their inputs in one clock cycle and their outputs in another*

GRACE Pipelining

- Goal is to ensure values arrive on input ports exactly when they are due
- GRACE Approach:
  - Based on schedule times, insert registers along edges produced in one clock cycle and consumed in another
  - When multiple operations are in series in a single clock cycle, this is called "chaining"

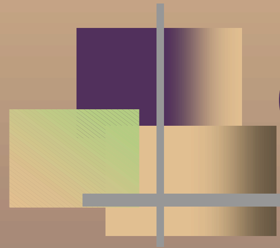


GRACE Control Synthesis & RTL Generation

- Specialized VHDL operators hand-optimized for controlling blocks:
  - Function controller
  - Thread controller
  - While loop controller
  - For loop controller in various bitwidths
- Controller generates an “enable” signal which is pipelined (just like “data” signals) to arrive on correct cycle to enable operator
- Resulting graph is emitted in structural VHDL + RTL format suitable for RTL synthesis

GRACE Availability

- Not yet publicly released – still has a few bugs
- Will appear at http://users.ece.gatech.edu/rick446/ along with
  - Tutorials
  - Technical Report
  - This presentation



Questions?