

Synthesis of Application-Specific Counterflow Pipelines

Bruce R. Childers
 Jack W. Davidson
 Wm. A Wulf

Department of Computer Science
 University of Virginia

{ brc2m, jwd, wulf }@virginia.edu

Workshop on the Interaction between Compilers and Computer Architecture

February 4, 1996, San Jose, CA

Problem Overview

Embedded microprocessors (systems) are very sensitive to cost-performance

- Pennies matter in high-volume applications
- Many dimensions (e.g., power drain, code size, chip area, heat dissipation, etc.)

Improve cost-performance

- Include only necessary features

Current solutions

- Microprocessor cores with peripherals
- Domain specific processors
- Full-custom processors

The Counterflow Pipeline Processor

Developed by Sproull R. F., Sutherland I. E., and Molnar C. E.

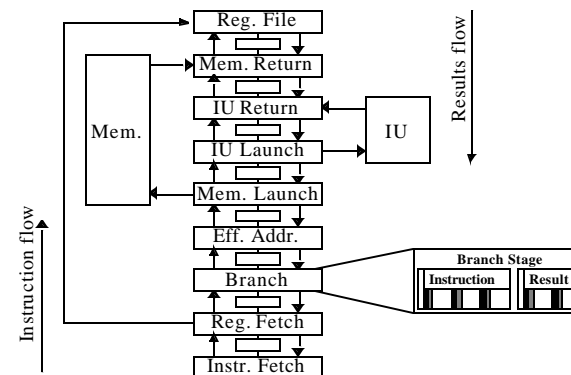
Performance potential of asynchronous application-specific CFPPs

- Microprocessor core is part of the design
- Customize core for an application

Two key issues

- CFPP micro-architecture synthesis
- Organization

The Counterflow Pipeline Processor



- Two pipelines: instruction and result
- Devices: stages, sidings, and arbitration units

Application-Specific Counterflow Pipelines

Characteristics

- Simple & regular interconnection
- Modular
 - Device libraries
- Local control
 - No global pipeline synchronization
- Inherently handles complex structures
 - Out-of-order completion
 - Register renaming
 - Result forwarding
 - Speculative execution

slide(5)

Counterflow Pipeline Synthesis

Couple software and hardware

- Application program is specification
- Execution trace is a deep Counterflow Pipeline

Advantages

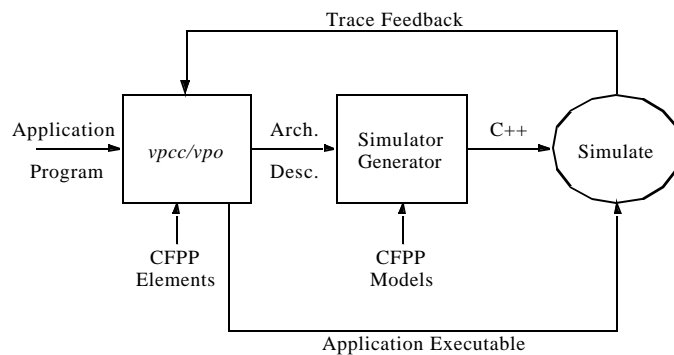
- Cost-performance
- Abstraction
- Design cost & turn-around time

Achieve high throughput

- Exploit concurrency
- Avoid pipeline delays

slide(6)

CFPP Synthesis Framework



slide(7)

A Preliminary CFPP Synthesis Methodology

Methodology Overview

- 1) Perform dataflow analysis to build dependency graph
- 2) Partition functionality among pipeline stages and sidings
- 3) Arrange interconnection network
- 4) Simulate and collect execution trace to refine organization

slide(8)

A Preliminary CFPP Synthesis Methodology

Dataflow Analysis

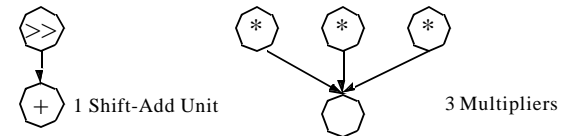
- Most optimizing compilers collect dataflow information
- Integrate steps 2 and 3 into compiler at dataflow analysis phase

slide(9)

A Preliminary CFPP Synthesis Methodology

Functionality Partitioning

- Graph structure
- Type and number of sidings or stages



- Apply heuristics
 - High latency operations > Siding
 - High latency operations with no available concurrency > Pipeline stage
 - Low latency operations > Pipeline stage

slide(10)

A Preliminary CFPP Synthesis Methodology

Pipeline Interconnection

- Stage ordering defines interconnection network
- Two stage types
 - Fixed location stages (e.g., instruction fetch, register file, register fetch)
 - Dependency ordered stages (e.g., operational, launch, return stages)
- Dependency Ordered Stages
 - Critical path order
 - Minimize result flow distance
 - Operational latency and concurrency

Execution Trace Feedback

- Refine stage placement

slide(11)

A Preliminary CFPP Synthesis Methodology

In Practice, a Preliminary Approach

- 1) Compile the application and build the dataflow graph.
- 2) Partition pipeline functionality based on latency and concurrency.
- 3) Arrange stages initially based only on the critical path.
- 4) Generate simulator and execute program.
- 5) Examine instruction trace by hand.
- 6) Refine arrangement.
- 7) Repeat from 4 until execution latency is near the ideal critical path latency.
- 8) Add instructions off the critical path one at a time.
- 9) Repeat from 2 until all instructions are added.

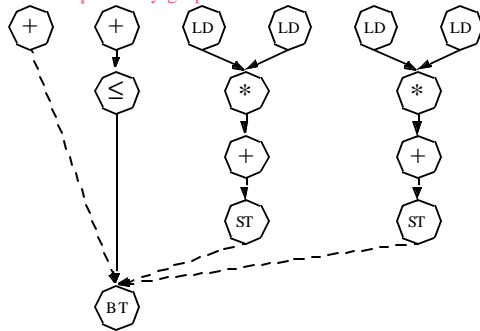
slide(12)

A CFPP Synthesis Example

Consider a loop with a series of Multiply-And-Accumulates:

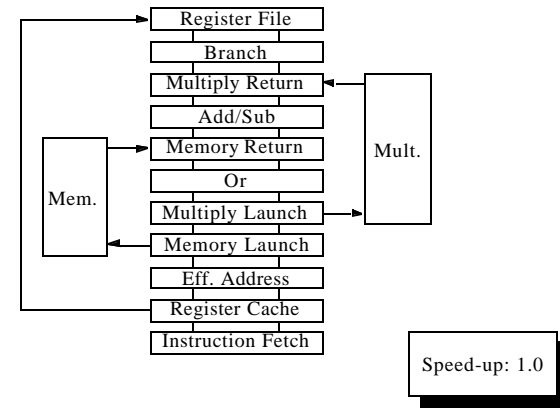
```
for (i = 0; i < 10; i += 2) {
  c[i] = a[i] * b[i] + k;
  c[i+1] = a[i+1] * b[i+1] + k;
}
```

And its instruction dependency graph:



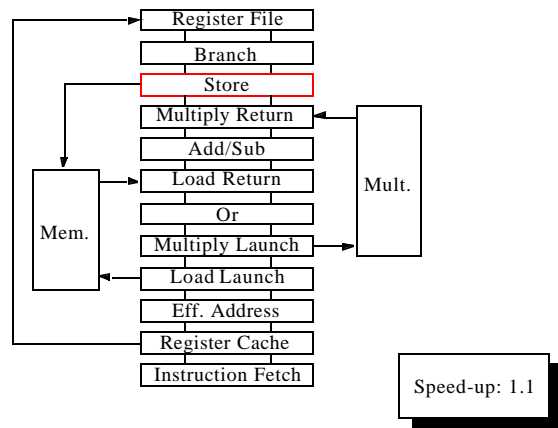
slide(13)

Multiply-And-Accumulate Pipeline 1



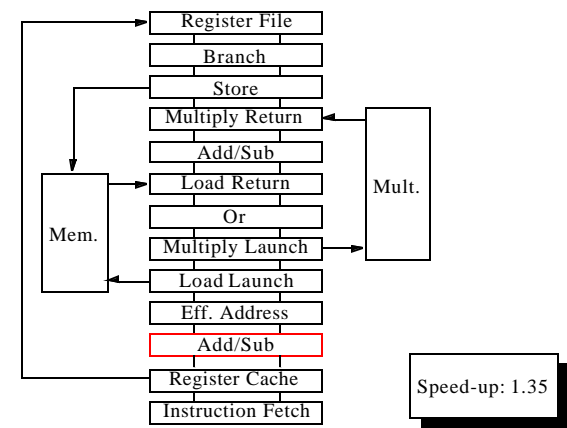
slide(14)

Multiply-And-Accumulate Pipeline 2



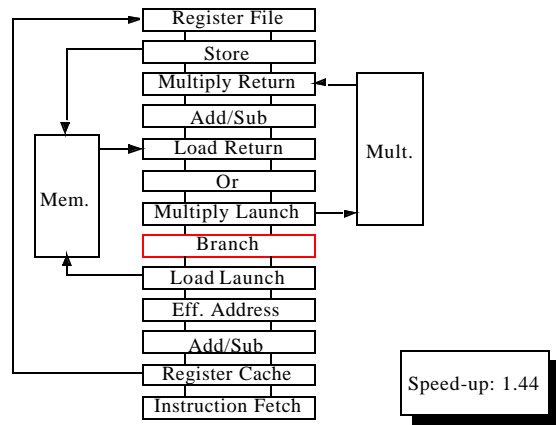
slide(15)

Multiply-And-Accumulate Pipeline 3



slide(16)

Multiply-And-Accumulate Pipeline 4



slide(17)

Preliminary Results

	<i>memcpy_{cjpp}</i>	<i>iir_{cjpp}</i>	<i>mac_{cjpp}</i>	<i>mac-i_{cjpp}</i>
memcpy	1.68	1.49	1.49	0.91
dot-product	—	—	1.47	1.17
iir	—	1.5	—	—
mac-i	—	—	1.39	1.73
mac-d	—	—	1.8	1.62
mac	—	—	1.43	1.04

Values are speed-up over the *generic_{cjpp}* structure.
Some structures do not support full functionality required for all benchmarks.

slide(18)

Summary and Future Work

Summary

- Flexible target for micro-architecture synthesis
- Application program serves as a specification
- Dataflow graph indicates pipeline functionality partitioning and interconnect

Future Work

- Organizational enhancements and structures
- Partitioning and arrangement heuristics
- Automate synthesis methodology
- Try larger and different types of applications

slide(19)