

Spatial Computation

Mihai Budiu
CMU CS

Thesis committee:

Seth Goldstein

Peter Lee

Todd Mowry

Babak Falsafi

Nevin Heintze



Ph.D. Thesis defense, December 8, 2003

sky image courtesy of NASA

Spatial Computation

A model of general-purpose computation
based on Application-Specific Hardware.

Definition of Spatial Computation.

Thesis Statement

Application-Specific Hardware (ASH):

- can be synthesized by **adapting software compilation** for predicated architectures,
- provides high-performance for programs with **high ILP**, with **very low power** consumption,
- is a **more scalable and efficient** computation substrate than **multicore** processors.

Outline

- Introduction

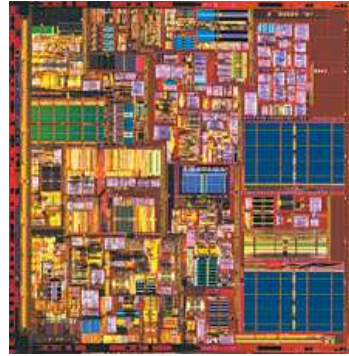


- Compiling for ASH
- Media processing on ASH
- ASH vs. superscalar processors
- Conclusions

Handshake image from www.247mail.com/why.html

CPU Problems

- Complexity
- Power
- Global Signals
- Limited ILP

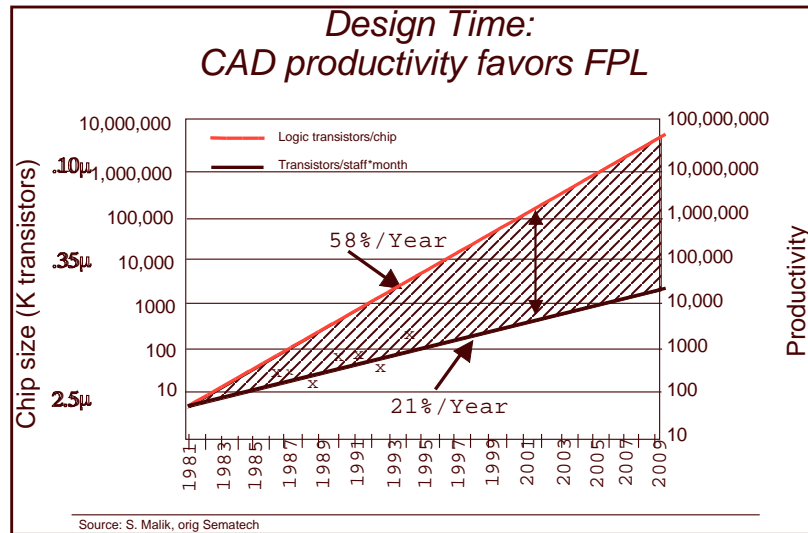


5

This research addresses some limitations of the (superscalar) microprocessor structure.

Complexity = of design, verification, manufacturing.

Design Complexity

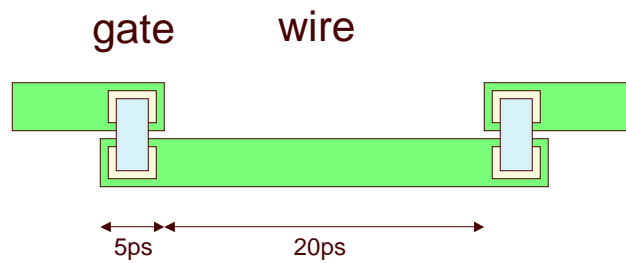


from Michael Flynn's FCRC 2003 talk

6

Design complexity outpaces productivity increase. A slide from Michael Flynn's FCRC plenary presentation.

Communication vs. Computation



Power consumption on wires is also dominant

7

Gates are cheap and wires expensive at very high clock speeds.

Our Approach: ASH



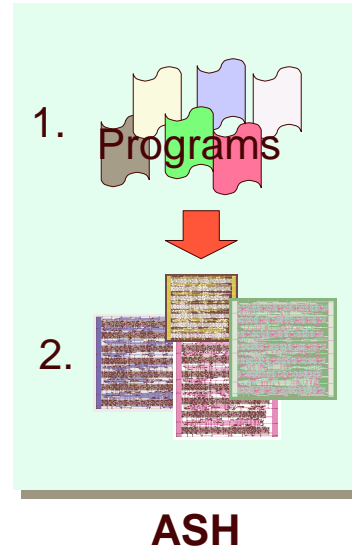
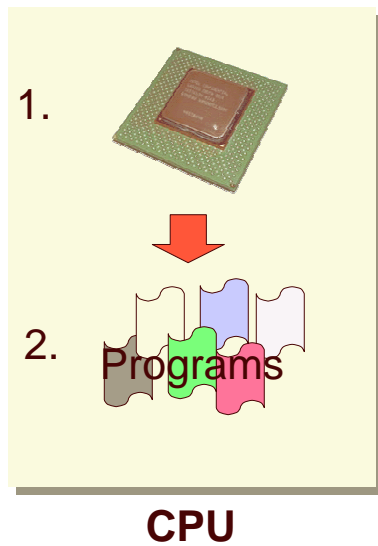
Application-Specific Hardware

8

Volcanic ash Image from

http://www.geology.sdsu.edu/how_volcanoes_work/Tephra.html

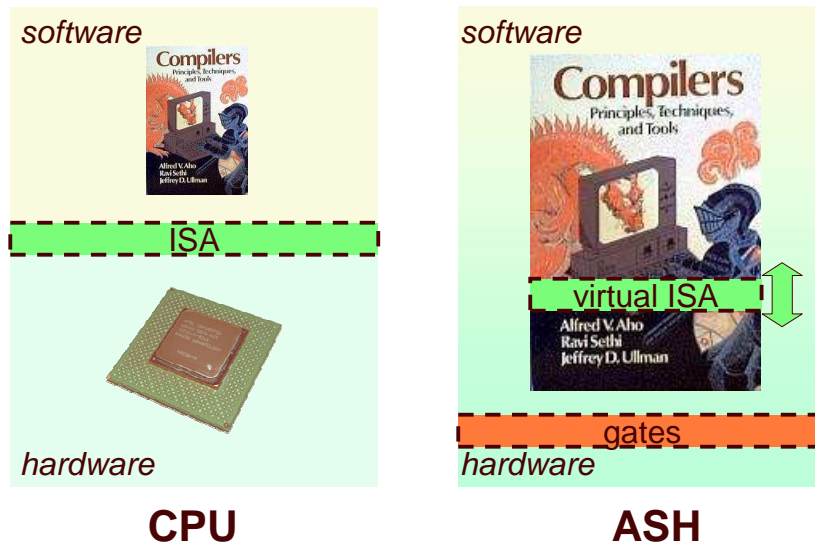
Resource Binding Time



9

We take advantage of the fact that we create hardware **after** the program is known.

Hardware Interface



10

We escape the “tyranny of the ISA”. This makes compilation more difficult.

Application-Specific Hardware

C program



Dataflow IR

Compiler



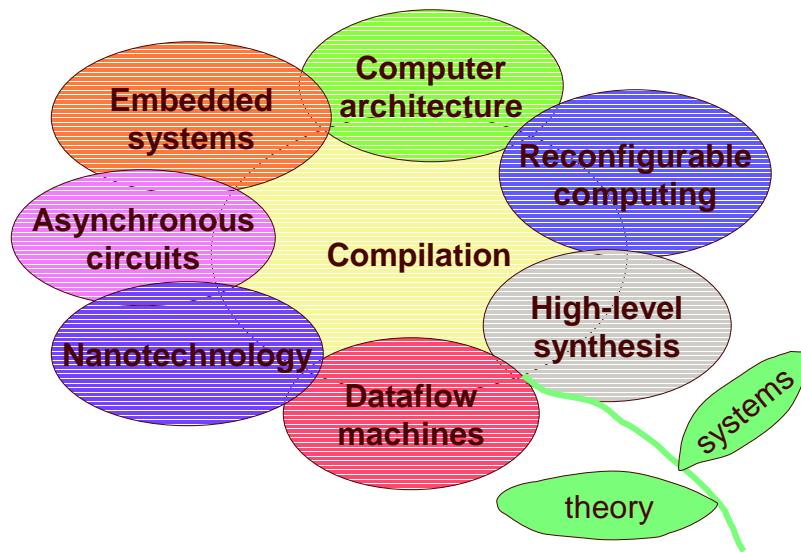
dataflow machine

Reconfigurable/custom hw

11

The dataflow machine generated is very close semantically to the internal compiler representation.

Contributions



12

This research makes most contributions in the area of optimizing compilation, but brings new insights into other areas as well.

Outline

- Introduction
- **CASH: Compiling for ASH**



- Media processing on ASH
- ASH vs. superscalar processors
- Conclusions

13

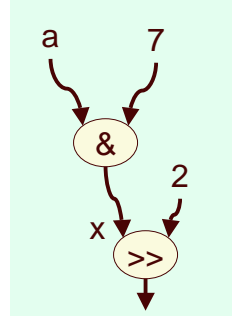
Computation = Dataflow

Programs

```
x = a & 7;  
...  
y = x >> 2;
```



Circuits

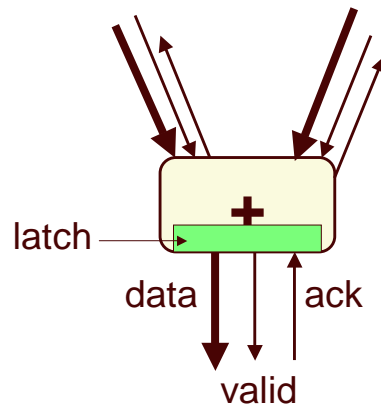


- Operations \Rightarrow functional units
- Variables \Rightarrow wires
- No interpretation

14

We do not have the disadvantages of the traditional dataflow machine model, because we implement the machines directly in hardware.

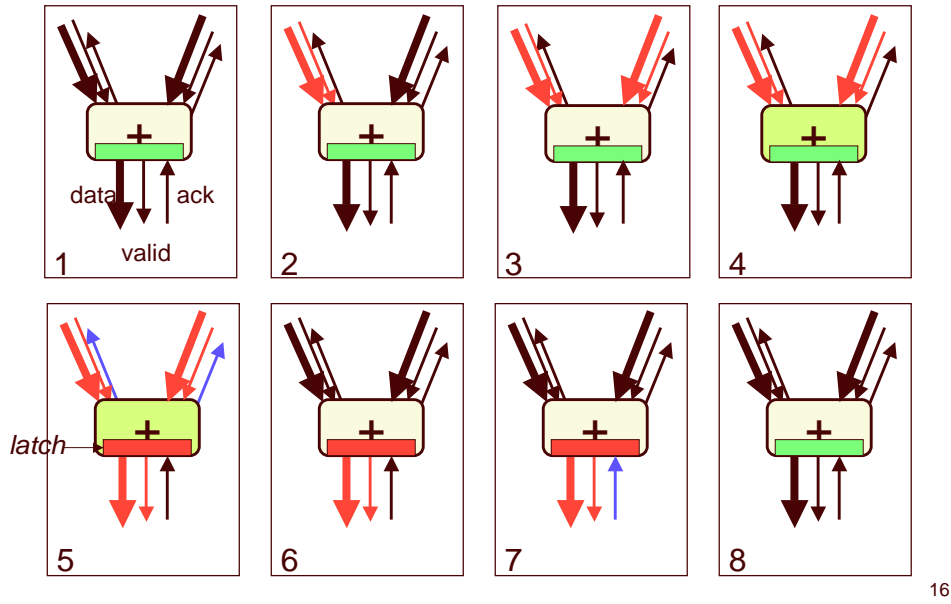
Basic Operation



15

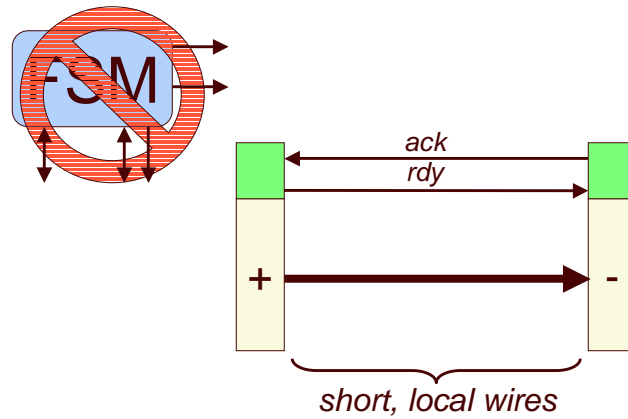
Both the internal representation and the synthesized circuit are built from such constructs.

Asynchronous Computation



Asynchronous computation is triggered by data availability; it is completely dynamically scheduled.

Distributed Control Logic



[asynchronous control](#)

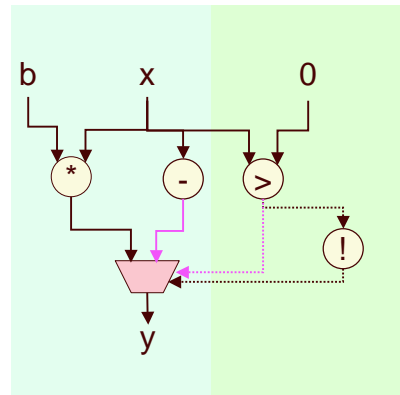
17

There is no centralized FSM controlling the behavior of the circuit; each producer-consumer pair has some control logic for handshaking, and that is all.

(There is however also logic for memory access, which features some global structures and arbitration.)

Forward Branches

```
if (x > 0)
    y = -x;
else
    y = b*x;
```



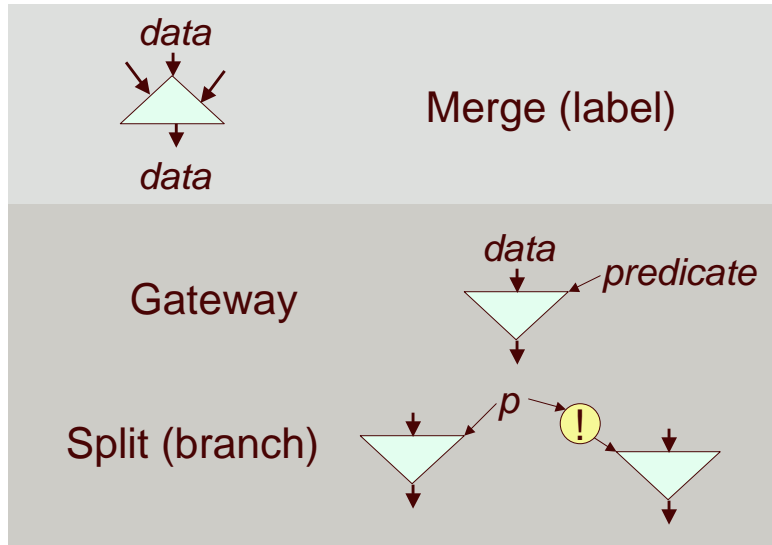
Conditionals \Rightarrow Speculation

[critical path](#)

18

Conditionals are compiled into circuits executing simultaneously all branches.

Control Flow \Rightarrow Data Flow



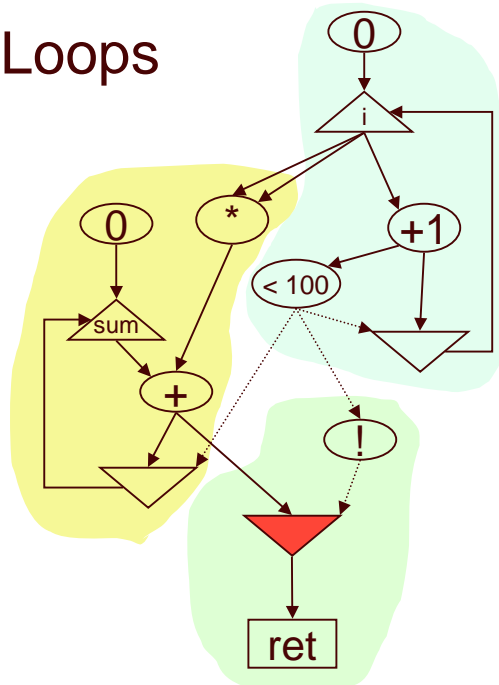
19

Note: in reality the merge operator is slightly more complicated than described here, since multiple merge operators sometimes need to cooperate in order to correctly execute the program.

The merge is also called “mu”, while the gateway is also called “eta”.

Loops

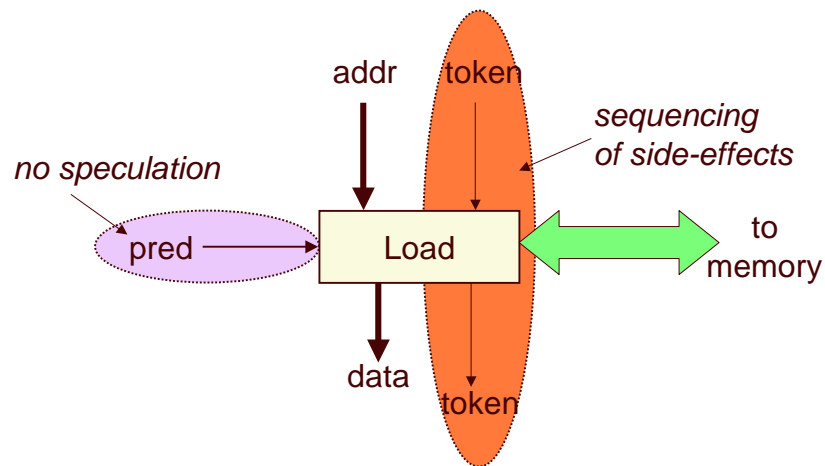
```
int sum=0, i;  
for (i=0; i < 100; i++)  
    sum += i*i;  
return sum;
```



20

Loops can be built using merges and gateways as building blocks. Some gateways direct the data to the beginning of the loop while other direct the data out of the loop (red). They are controlled by complementary predicates.

Predication and Side-Effects



21

Side-effect operations are predicated (never speculated).

The lack of a PC, i.e., program order, mandates the use of another mechanism to preserve correctness; this is the flow of tokens.

Tokens are both a compile-time and run-time construct. This is one instance where we do things an ISA would not allow.

Thesis Statement

Application-Specific Hardware:

- can be synthesized by **adapting software compilation** for predicated architectures,



Outline

- Introduction
- CASH: Compiling for ASH
 - An optimization on the SIDE



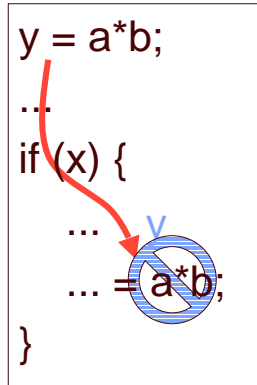
- Media processing on ASH
- ASH vs. superscalar processors
- Conclusions

[skip to](#)

23

Salad image from www.amgmedia.com/freephotos/salad.jpg

Availability Dataflow Analysis

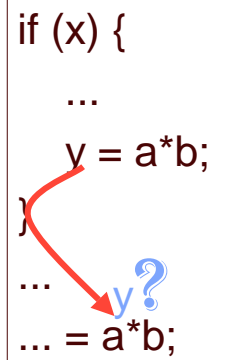


24

Replace expressions already computed with scalar values.

Dataflow Analysis Is Conservative

```
if (x) {  
    ...  
    y = a*b;  
}  
...  
... = a*b;
```



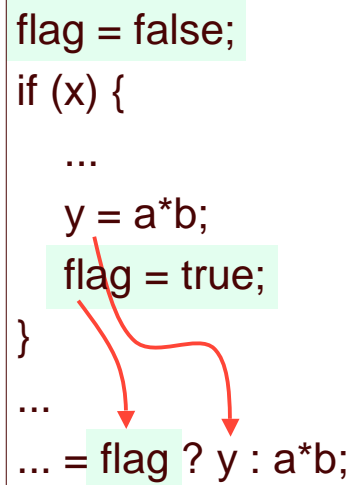
The diagram illustrates a conservative assumption in dataflow analysis. A red arrow points from the variable 'y' in the assignment 'y = a*b;' inside an 'if' block to a blue 'y?' in a subsequent assignment '... = a*b;'. This indicates that the analysis conservatively assumes the value of 'y' is available for the second assignment, even though it might not be if the first assignment is not executed.

25

Conservativeness must assume that the expression is not available.

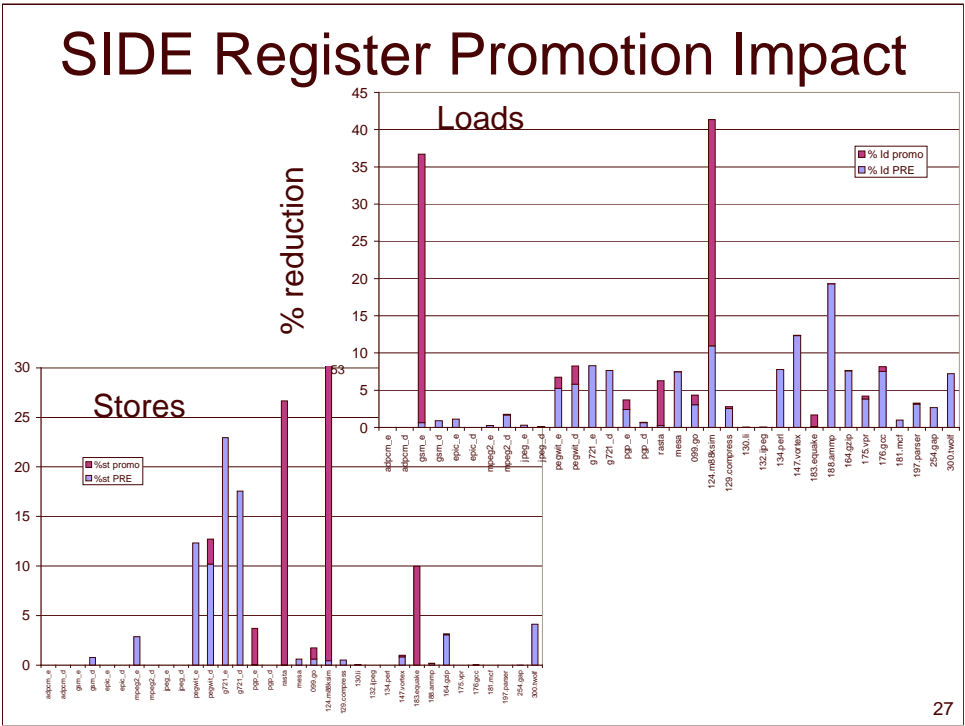
Static Instantiation, Dynamic Evaluation

```
flag = false;  
if (x) {  
    ...  
    y = a*b;  
    flag = true;  
}  
...  
... = flag ? y : a*b;
```

The diagram illustrates a control flow graph. It starts with a line 'flag = false;'. Below it is an 'if (x) {' block. Inside the block, there are three lines: '...', 'y = a*b;', and 'flag = true;'. The 'if' block is closed with a '}' line. Below the 'if' block is another '...' line, followed by a conditional expression '... = flag ? y : a*b;'. Two red arrows originate from the 'if' block: one from the 'y = a*b;' line pointing to the 'y' in the conditional expression, and another from the 'flag = true;' line pointing to the 'flag' in the conditional expression. The lines 'flag = false;', '...', 'y = a*b;', 'flag = true;', and the conditional expression are highlighted with light green backgrounds.

26

Solution: maintain dynamically the availability state of an expression;
predicate its re-computation.



We applied SIDE to memory access removal through register promotion; it can be very effective.

Outline

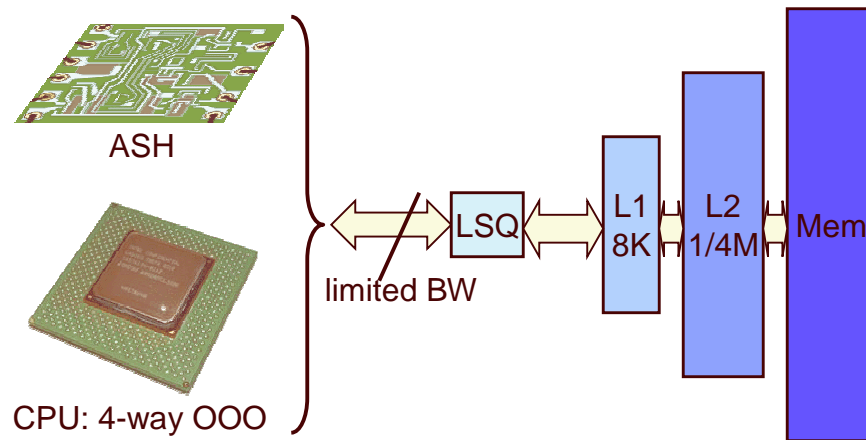
- Introduction
- CASH: Compiling for ASH
- **Media processing on ASH**



- ASH vs. superscalar processors
- Conclusions

Photo taken and processed by me.

Performance Evaluation



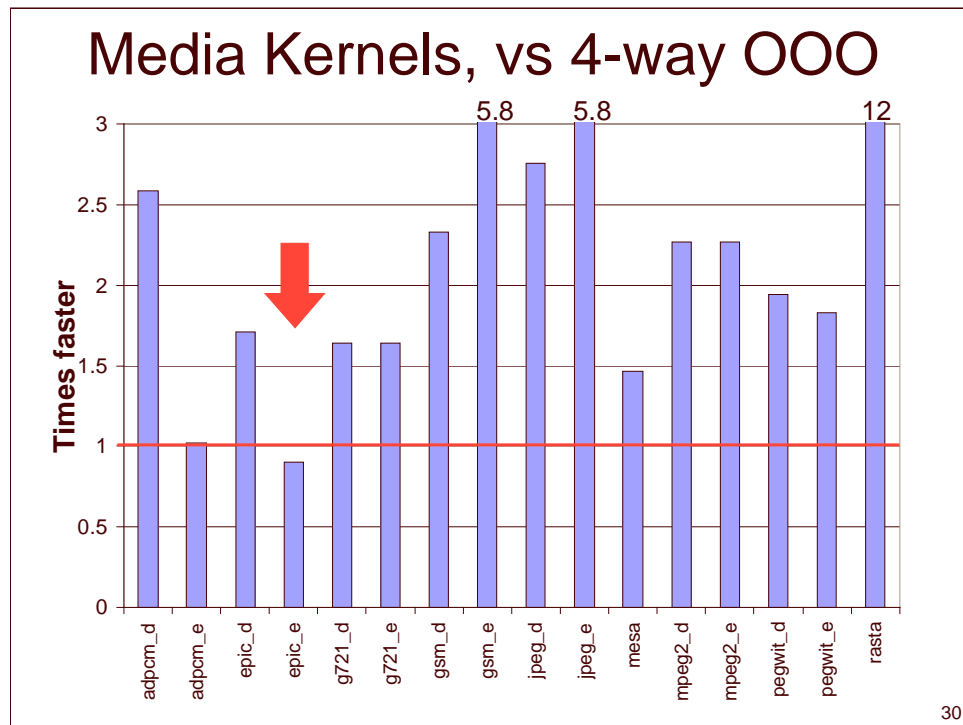
Assumption: all operations have the same latency.

29

We compare ASH and a 4-way superscalar by connecting them to the same memory hierarchy.

We also assume (for most of the evaluations) that the operation latencies are identical.

Ideally we'd like to compare ASH and a VLIW, but we didn't have access handy to a VLIW simulator.

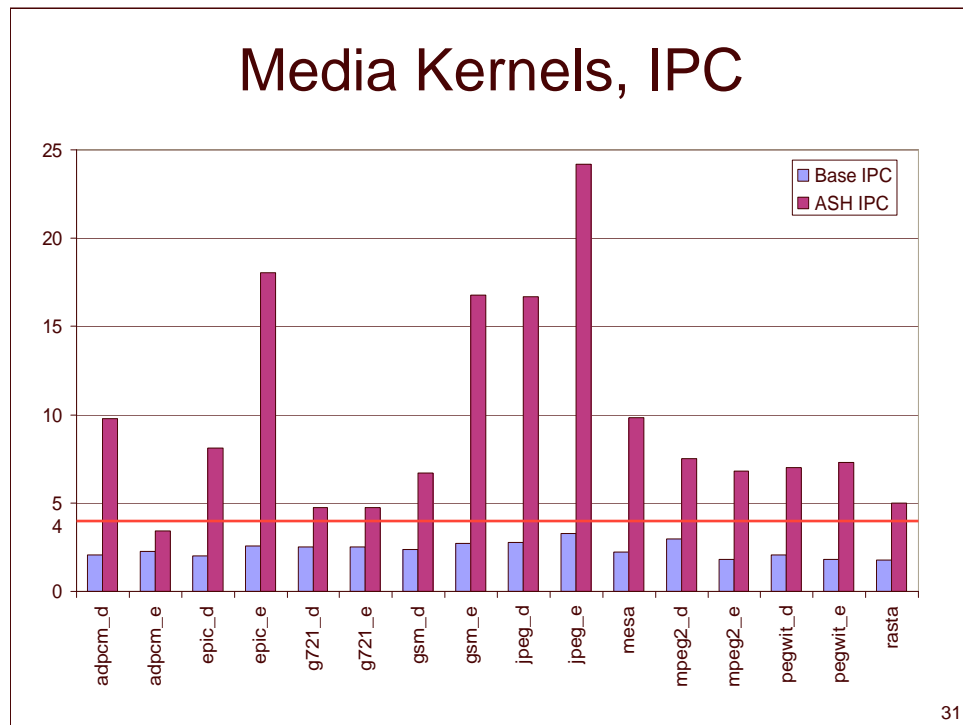


This data is just for the kernels.

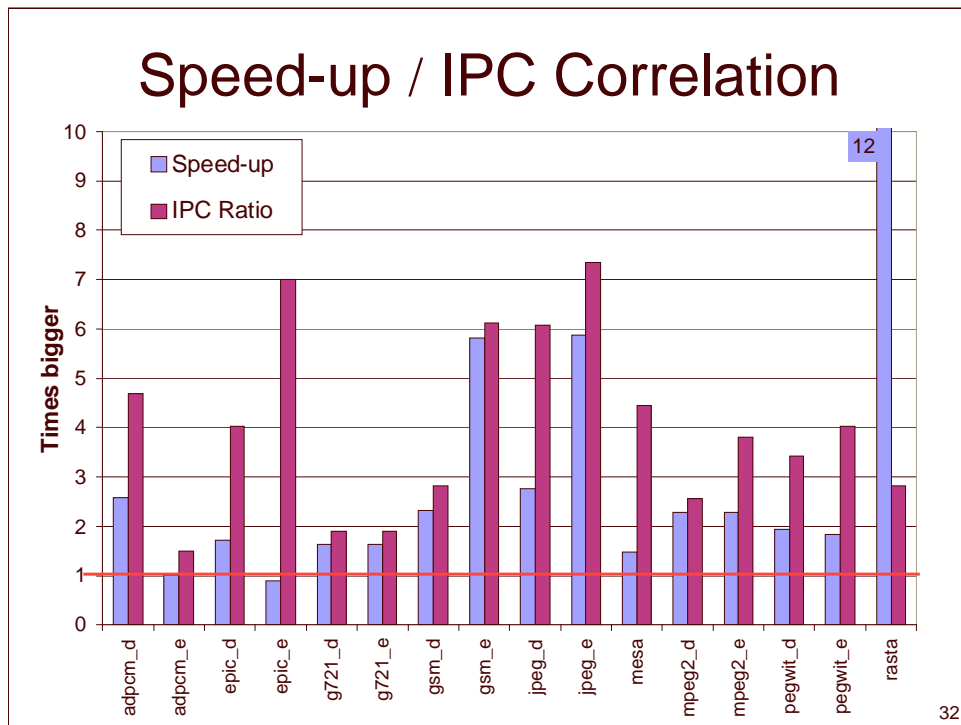
Kernels were selected manually to be the hottest 1, 2, or 3 functions in each application.

See the thesis for details on these kernels.

Only one kernel exhibits a slowdown.



ASH can sustain very high parallelism for these kernels.



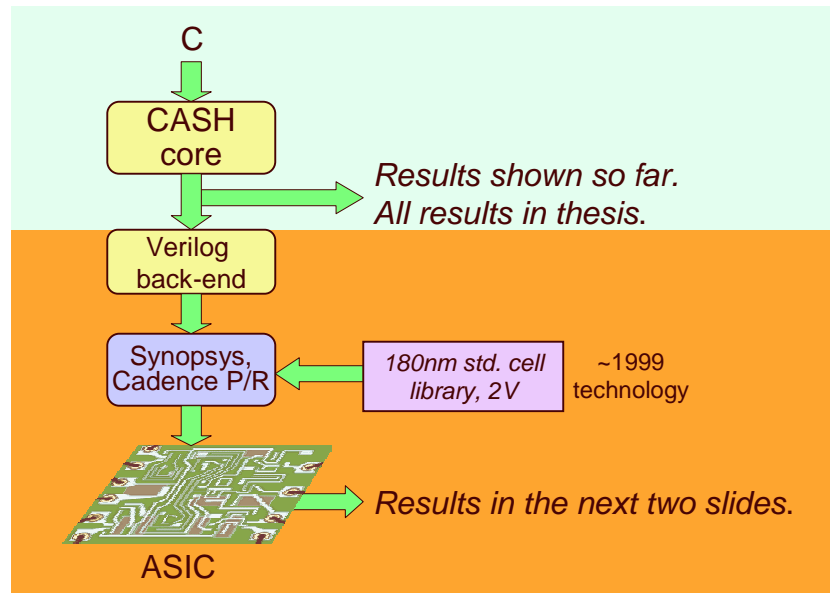
This graph compares speed-up and parallelism increase; i.e., how much parallelism we waste for getting the speed-up.

Note that the parallelism includes speculative execution.

epic_e is a very bad case: lots of useless speculation.

rasta is a great case; it results from the register promotion generating much better code than for the processor.

Low-Level Evaluation

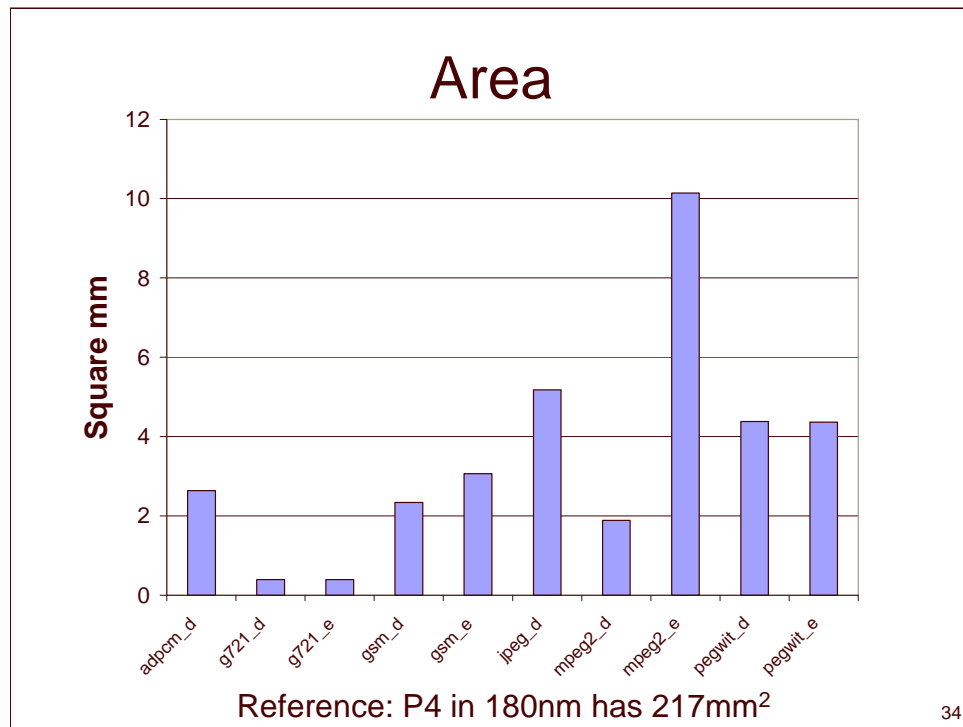


33

We compare the truly asynchronous circuits generated with a 600Mhz 4-way OOO superscalar based on SimpleScalar + Wattch.

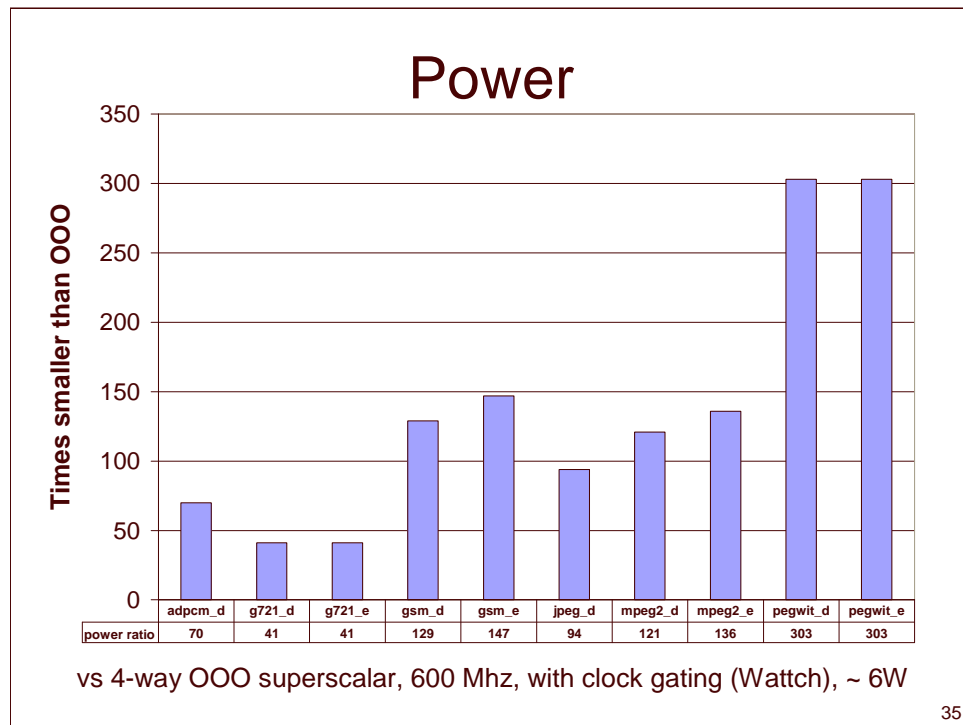
This is preliminary data; this back-end is still under development.

The Verilog back-end was written entirely by Girish Venkataramani.



These values are for slightly different kernels: for each program exactly one function was selected.

This data includes aggressive loop unrolling.



35

These values are not percents!

Bigger is better.

Thesis Statement

Application-Specific Hardware:

- provides high-performance for programs with **high ILP**, with **very low power** consumption,



Outline

- Introduction
- CASH: Compiling for ASH
- Media processing on ASH
 - dataflow pipelining



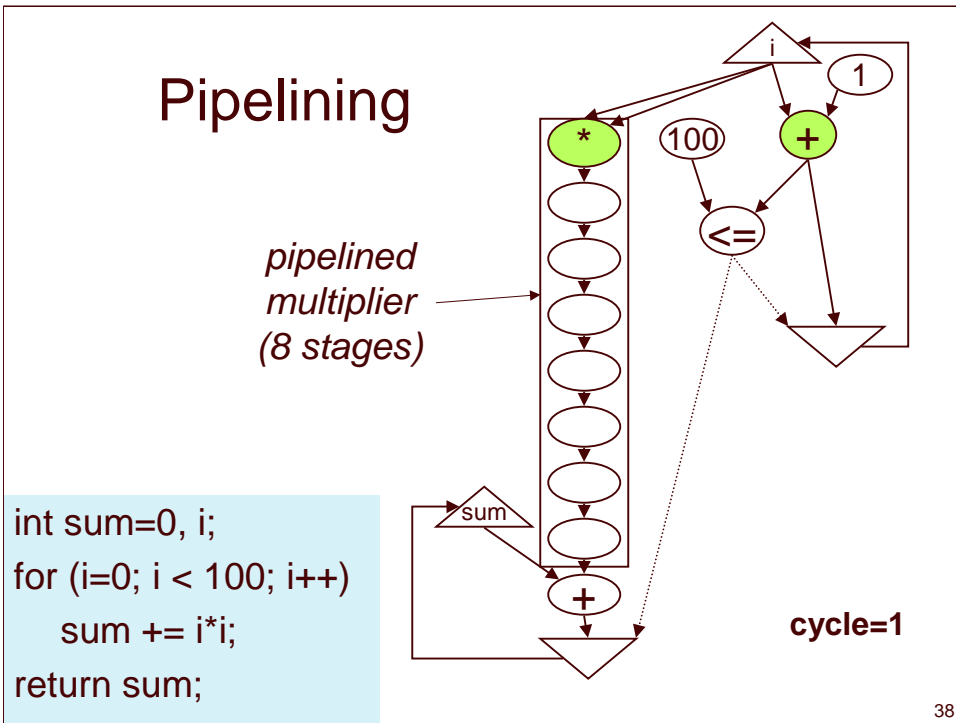
- ASH vs. superscalar processors [skip to](#)
- Conclusions

37

What is the source of parallelism in ASH?

Pipeline image from www.bfi.org/Trimtab/winter02/brittleTimes.htm

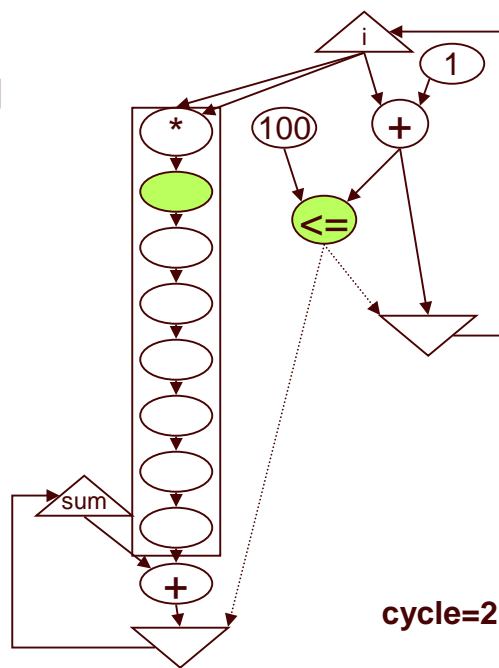
Pipelining



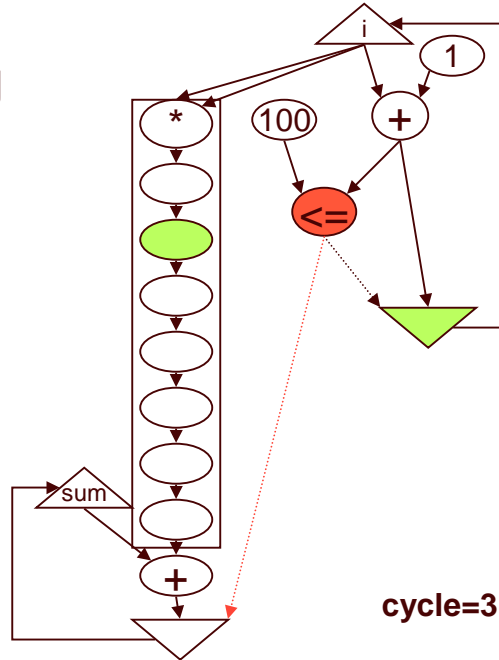
38

Let us look how the sum-of-squares program behaves at run-time.

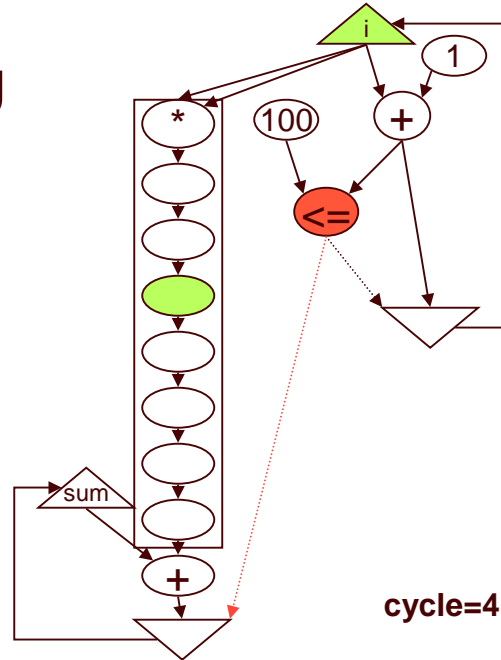
Pipelining

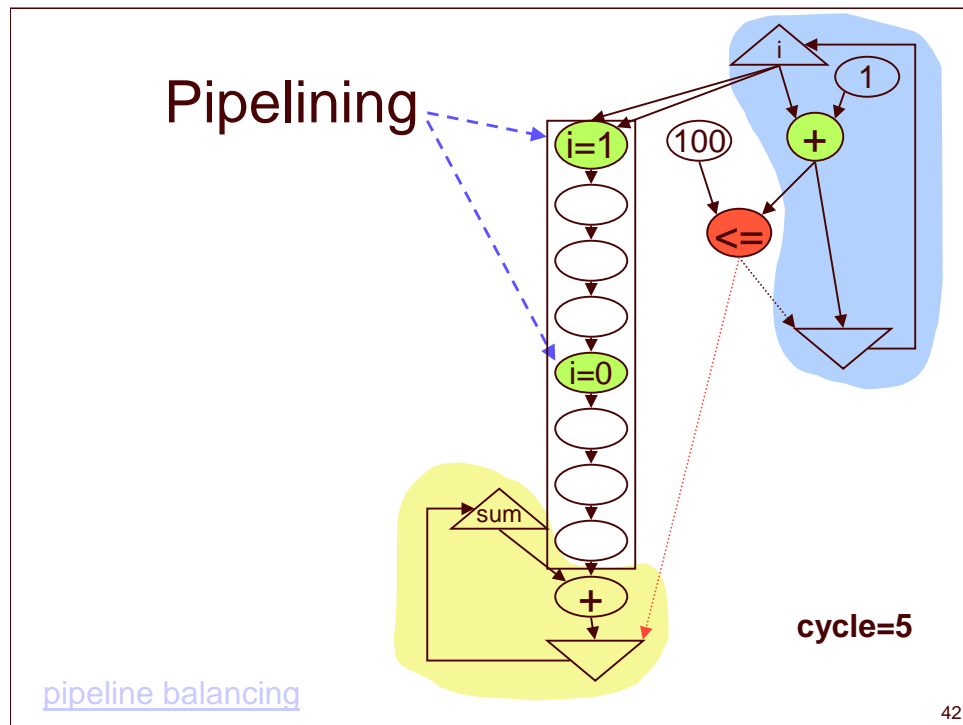


Pipelining



Pipelining





Pipelining occurs naturally, due to the asynchronous nature of the computation.

This phenomenon was exploited by the dataflow machines

Outline

- Introduction
- CASH: Compiling for ASH
- Media processing on ASH
- **ASH vs. superscalar processors**



- Conclusions

43

Now we perform a limit-study to see the potential of ASH on whole-programs.

Karate fighters image from <http://www.enighet.se/karate.htm>

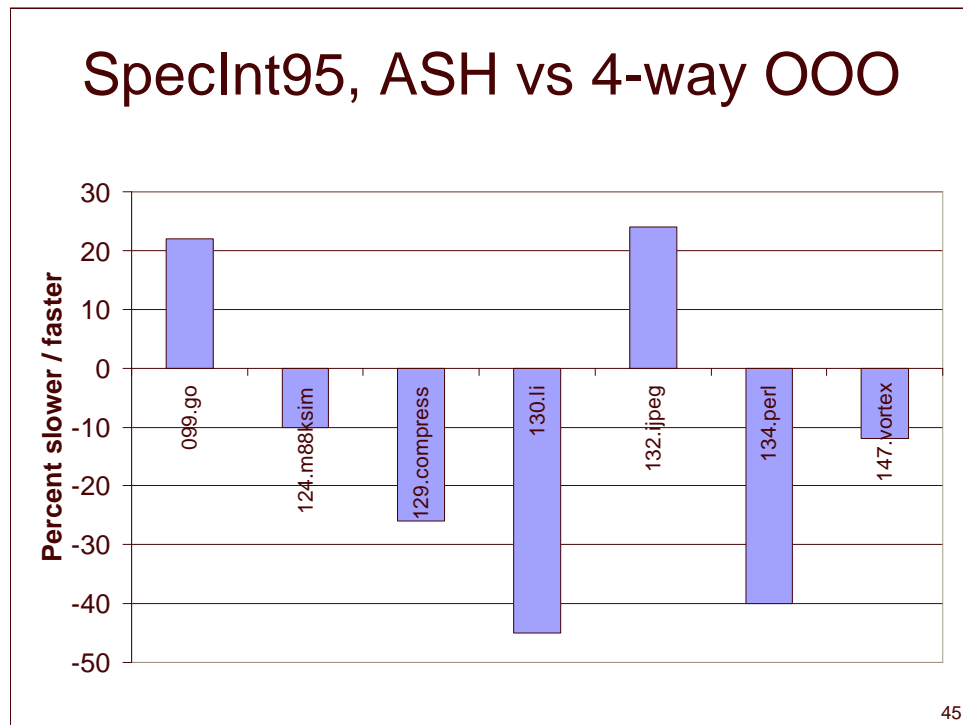
This Is Obvious!

wrong!

ASIC runs at full dataflow speed,
so CPU cannot do any better
(if compilers equally good).

44

There was already a hint in the epic_e kernel which was doing worse than a 4-way OOO.



For most of SpecInt95 ASH underperforms OOO.

This data is again with the high-level simulation model in which everything is assumed equal.

Branch Prediction

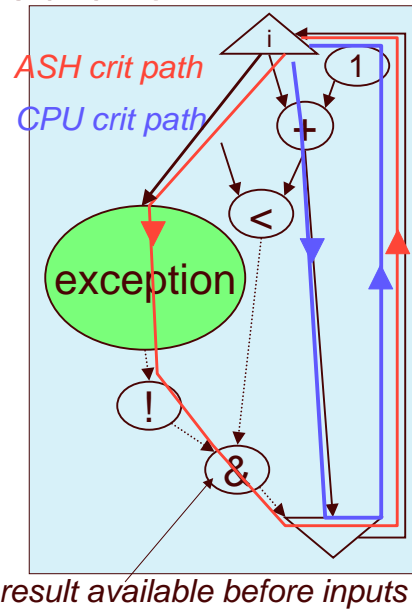
```
for (i=0; i < N; i++) {
```

■ ■ ■

```
if (exception) break;
```

↑ *Predicted not taken*
Effectively a noop for CPU!

Predicted taken.



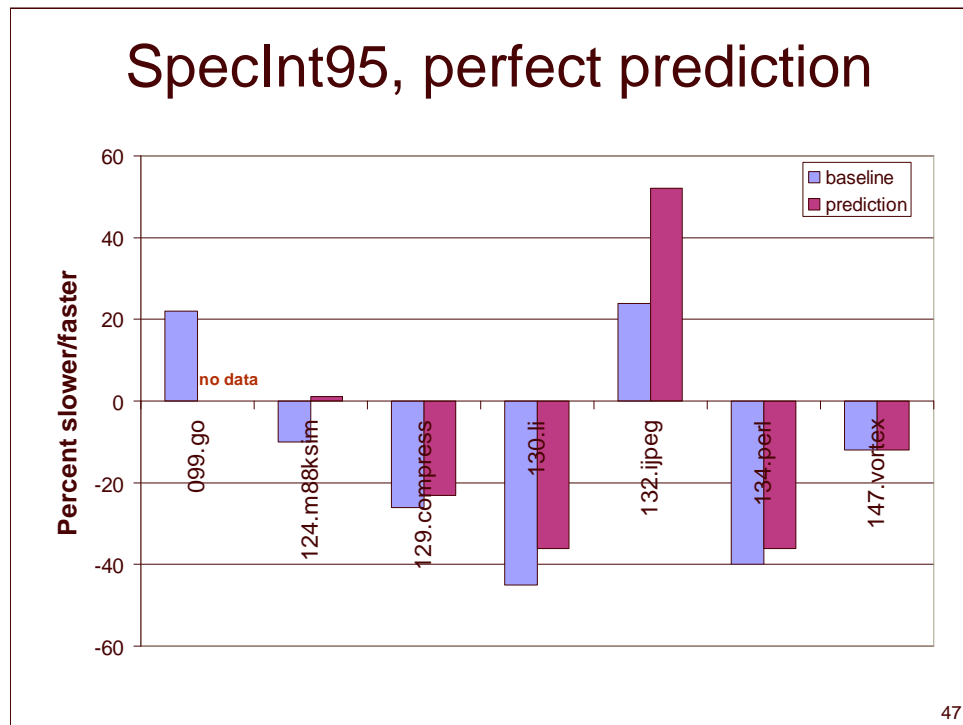
46

(Successful) branch prediction in processor renders some computations irrelevant.

Since we transformed control-flow into dataflow, the gateway predicate is on the critical path for ASH, but not for processor!

In the thesis we suggest a way to add a limited form of branch-prediction to ASH without destroying its distributed structure.

The hard part is not prediction, but stopping runaway speculation which crosses hyperblock boundaries.



Unfortunately even branch prediction does not redeem ASH.

ASH Problems



- Both ~~branch~~ and ~~join~~ not free
- Static dataflow
(no re-issue of same instr)
- Memory is “far”
- Fully static
 - No branch prediction
 - No dynamic unrolling
 - No register renaming
- Calls/returns not lenient
- ...

48

Photo of volcanic ash from <http://photo2.si.edu/earthquakes/sakurajima.html>

Other problems plague ASH. Some are inherent in the distributed structure, and some could be fixed with changes in the model/compiler.

Thesis Statement

Application-Specific Hardware:

- is a **more scalable and efficient** computation substrate than microarchitectural processors.



Outline

Introduction

+ CASH: Compiling for ASH

+ Media processing on ASH

+ ASH vs. superscalar processors

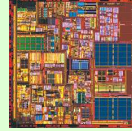
= Conclusions

50

Strengths



- low power
- simple verification?
- specialized to app.
- unlimited ILP
- simple hardware
- no fixed window



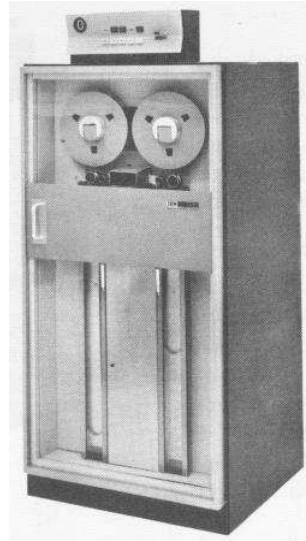
- economies of scale
- highly optimized
- branch prediction
- control speculation
- full-dataflow
- global signals/decision

Conclusions

- Compiling “around the ISA” is a fruitful research approach.
- Distributed computation structures require more synchronization overhead.
- Spatial Computation efficiently implements high-ILP computation with very low power.

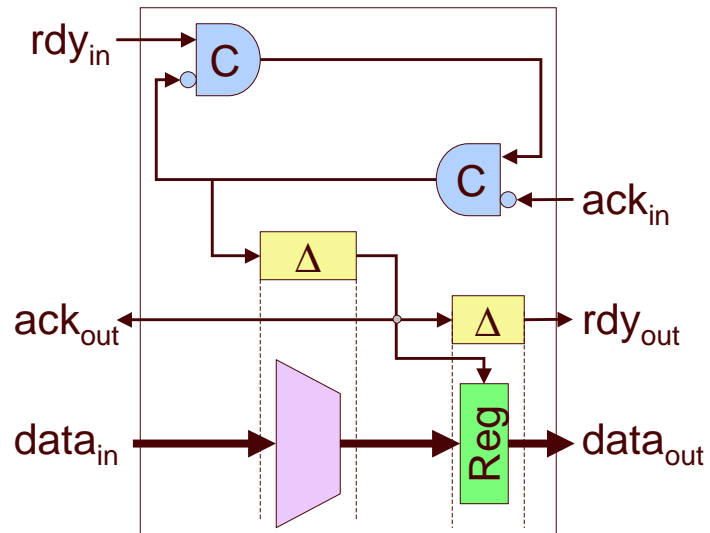
Backup Slides

- [Control logic](#)
- [Pipeline balancing](#)
- [Lenient execution](#)
- [Dynamic Critical Path](#)
- [Memory PRE](#)
- [Critical path analysis](#)
- [CPU + ASH](#)



53

Control Logic



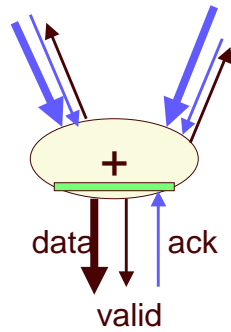
[back](#) [back to talk](#)

54

Typical micropipelines.

The rdy and ack signals may have fan-out/fan-in.

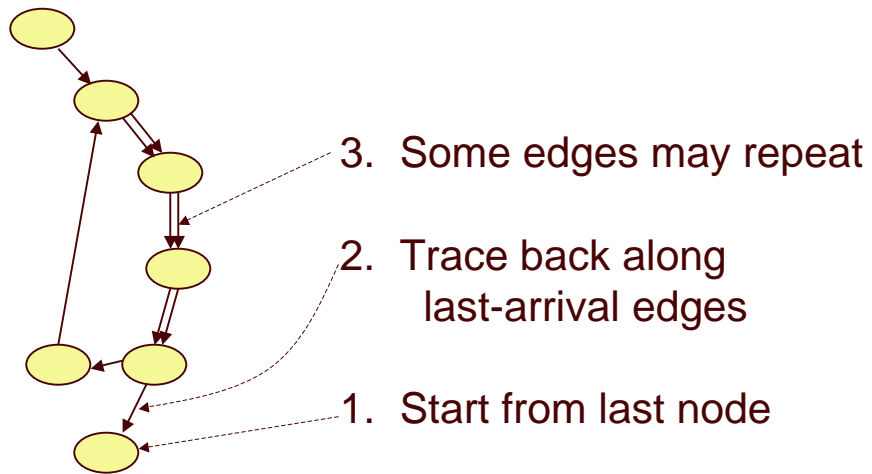
Last-Arrival Events



- Event enabling the generation of a result
- May be an ack
- Critical path=collection of last-arrival edges

55

Dynamic Critical Path



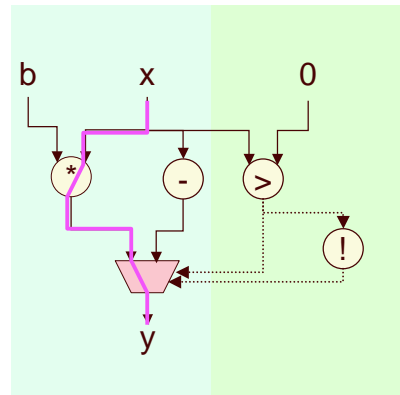
[back](#) [back to analysis](#)

56

See Fields & Bodik's seminal ISCA 2001 paper.

Critical Paths

```
if (x > 0)
    y = -x;
else
    y = b*x;
```

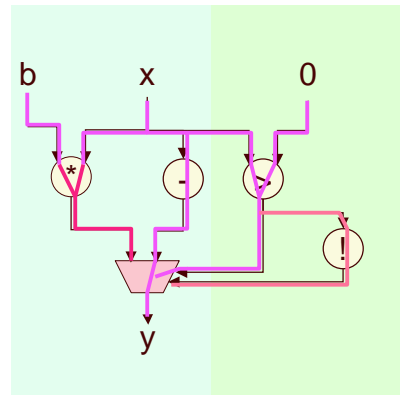


57

Speculation introduces the problem of long critical paths. The solution is on the next slide.

Lenient Operations

```
if (x > 0)
    y = -x;
else
    y = b*x;
```



Solve the problem of unbalanced paths

[back](#) [back to talk](#)

58

A lenient operation can generate the result before all of its inputs are known.
Leniency reduces the dynamic critical path.

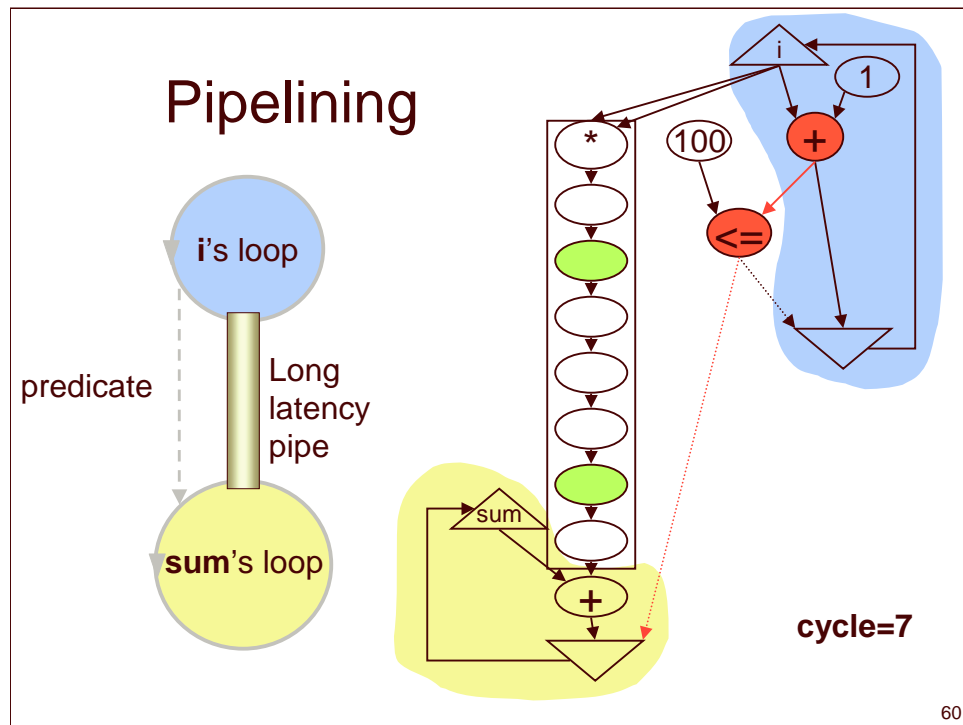
Pipelining

The diagram illustrates a 6-stage processor pipeline. The stages are represented by a vertical column of six ovals. The first stage is a multiplier (*). The second stage is a register (i=1). The third stage is an empty register. The fourth stage is an empty register. The fifth stage is a register (i=0). The sixth stage is an empty register. The output of the sixth stage is connected to a branch adder (+) and a branch comparator (<=). The branch adder takes the output of the fifth stage and a constant value of 1. The branch comparator takes the output of the fourth stage and a constant value of 100. The branch adder's output is connected to the input of the branch comparator. The branch comparator's output is connected to a multiplexer (triangle) that selects between the output of the sixth stage and the output of the branch adder. The output of the multiplexer is connected to a register (sum) and a branch adder (+). The branch adder takes the output of the register (sum) and the output of the multiplexer. The output of the branch adder is connected to the input of the multiplexer. The text "cycle=6" is shown in the bottom right corner.

cycle=6

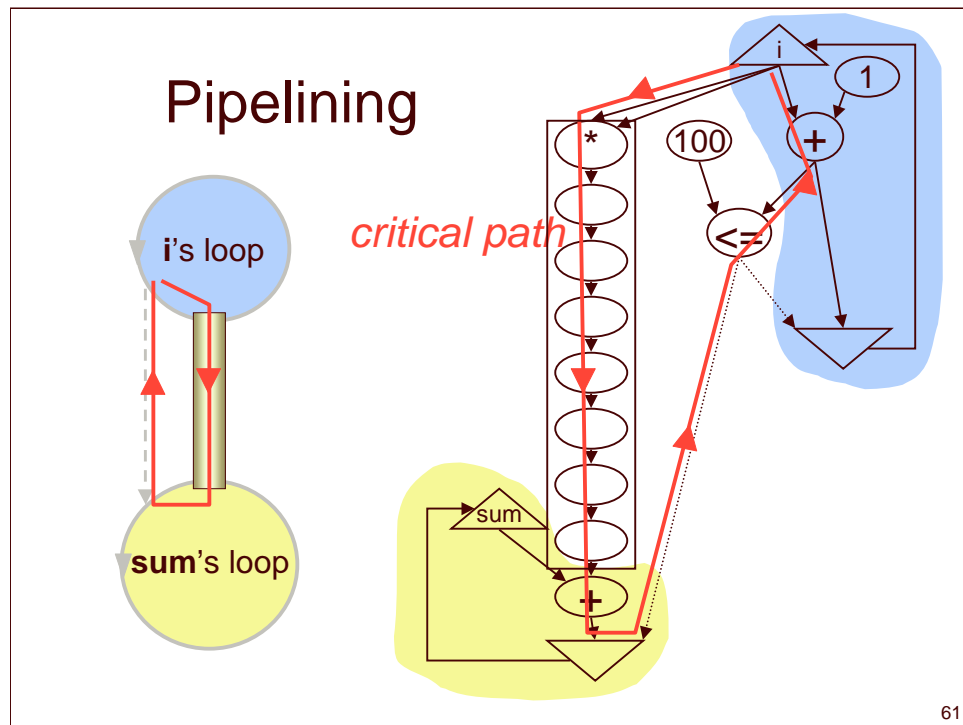
59

The thesis presents a general solution called “pipeline balancing”.



60

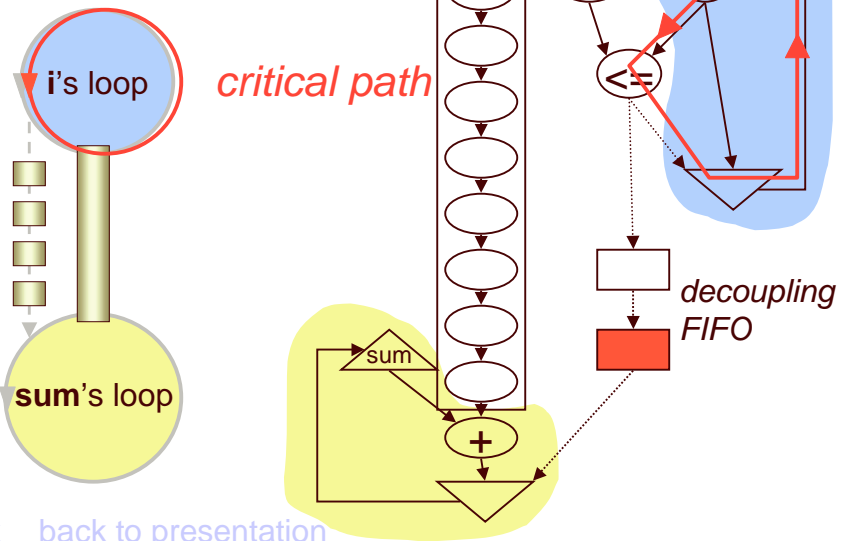
The i loop is coupled to the sum loop through the predicate. The acknowledgment for the predicate prevents i's loop from making progress.



61

The critical path of this circuit actually goes through an acknowledgment edge.

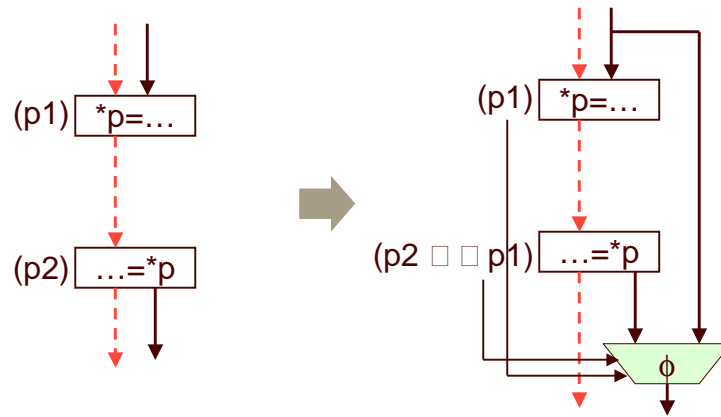
Pipeline balancing



63

The critical path is minimal for the resulting circuit, implying maximal performance.

Register Promotion

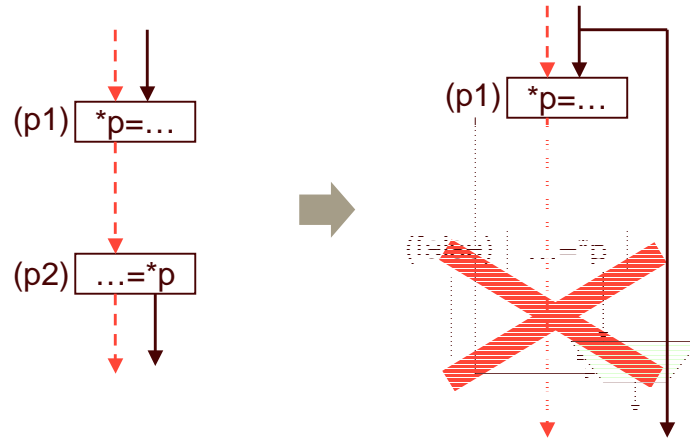


Load is executed only if store is not

64

The load predicate is weakened.

Register Promotion (2)



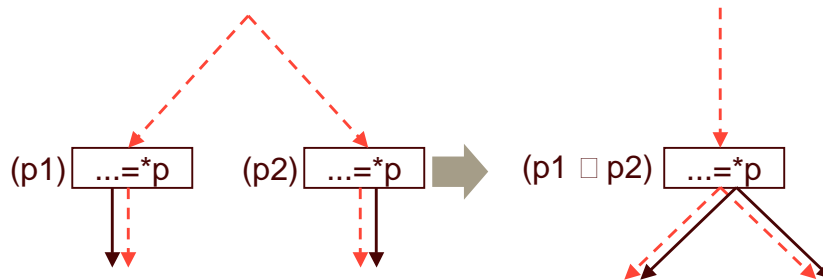
- When $p2 \Rightarrow p1$ the load becomes dead...
- ...i.e., when store dominates load in CFG

[back](#)

65

As a particular case of the previous optimization, if whenever the load predicate is true the store predicate is as well, the load can be completely removed. It is removed by the dead-code optimization, since its predicate becomes false. This optimization can be generalized for the case of multiple stores preceding the load.

□ PRE



This corresponds in the CFG to **lifting** the load to a basic block dominating the original loads

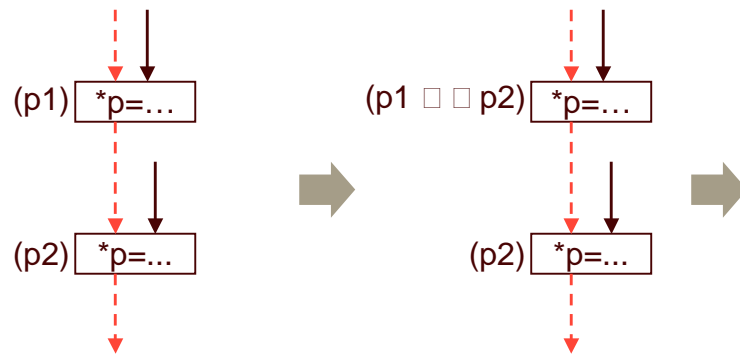
66

The same optimization can be applied to loads as well.

The code is basically the same as for common-subexpression elimination in an SSA form.

(Technicality: to enable this transformation no predicate must depend on the other operation. Easily checked with a reachability computation.)

Store-store (1)

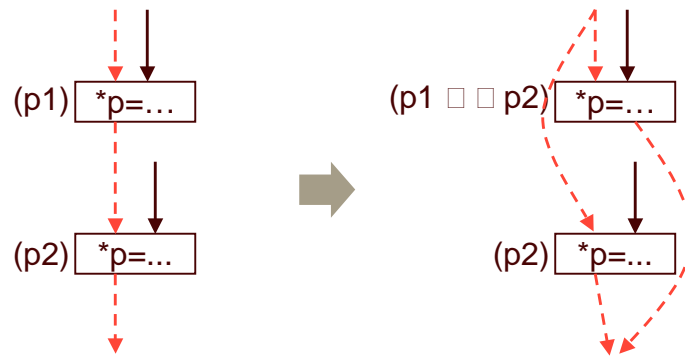


- When $p1 \Rightarrow p2$ the first store becomes dead...
- ...i.e., when second store post-dominates first in CFG

67

In this case the predicate of the first store is weakened, so that it is never executed when the second one will overwrite it.

Store-store (2)



- Token edge eliminated, but...
- ...transitive closure of tokens preserved

[back](#)

68

Since the two predicates are now disjoint, the two stores can never occur simultaneously, so there is no need for a token edge between them.

Notice that the removal of the token edge is not trivial.

A Code Fragment

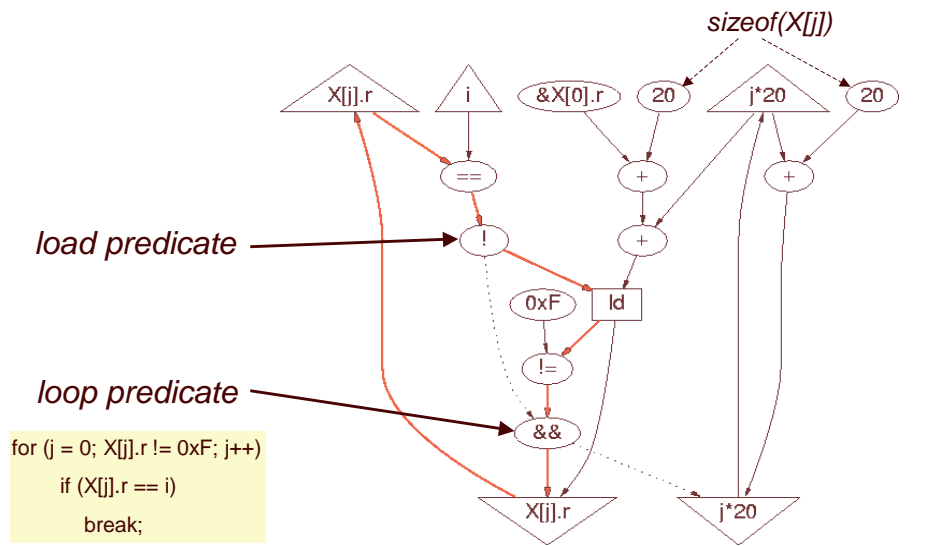
```
for(i = 0; i < 64; i++)    {  
    for (j = 0; X[j].r != 0xF; j++)  
        if (X[j].r == i)  
            break;  
  
    Y[i] = X[j].q;  
}
```

SpecINT95:124.m88ksim:init_processor, stylized

69

A sample analysis of bottlenecks in ASH.

Dynamic Critical Path [definition](#)



70

Critical path contains control-flow (i.e., eta) predicate.

MIPS gcc Code

LOOP:

L1: beq \$v0,\$a1,EXIT ; X[j].r == i

L2: addiu \$v1,\$v1,20 ; &X[j+1].r

L3: lw \$v0,0(\$v1) ; X[j+1].r

L4: addiu \$a0,\$a0,1 ; j++

L5: bne \$v0,\$a3,LOOP ; X[j+1].r == 0xF

EXIT:

```
for (j = 0; X[j].r != 0xF; j++)  
    if (X[j].r == i)  
        break;
```

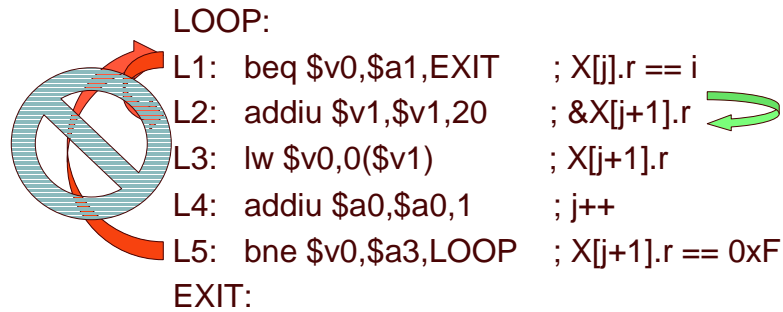
L1 □ L2 □ L3 □ L5 □ L1

4-instructions loop-carried dependence

71

Processor critical path also spans two branches.

If Branch Prediction Correct



```

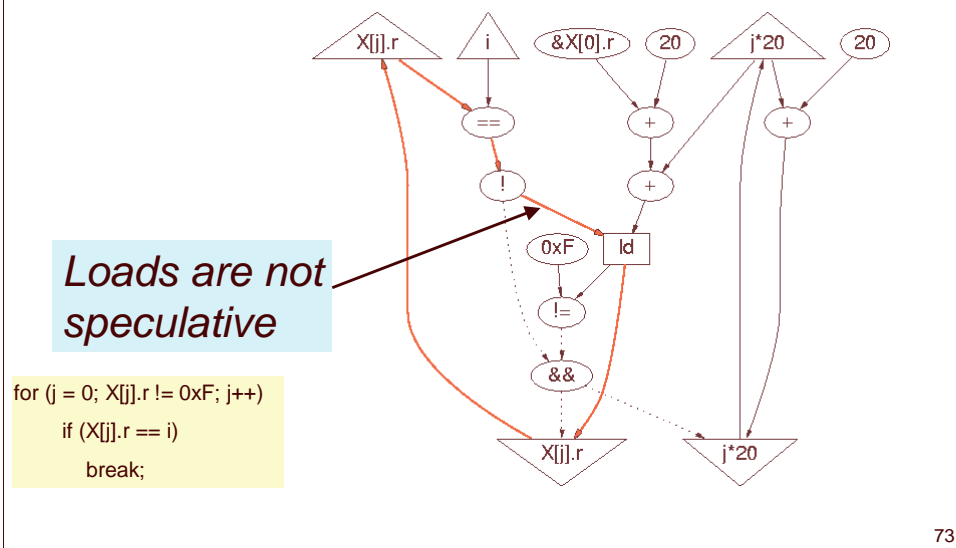
for (j = 0; X[j].r != 0xF; j++)
    if (X[j].r == i)
        break;
    
```

L1 L2 L3 L5 L1
Superscalar is issue-limited!
2 cycles/iteration sustained

72

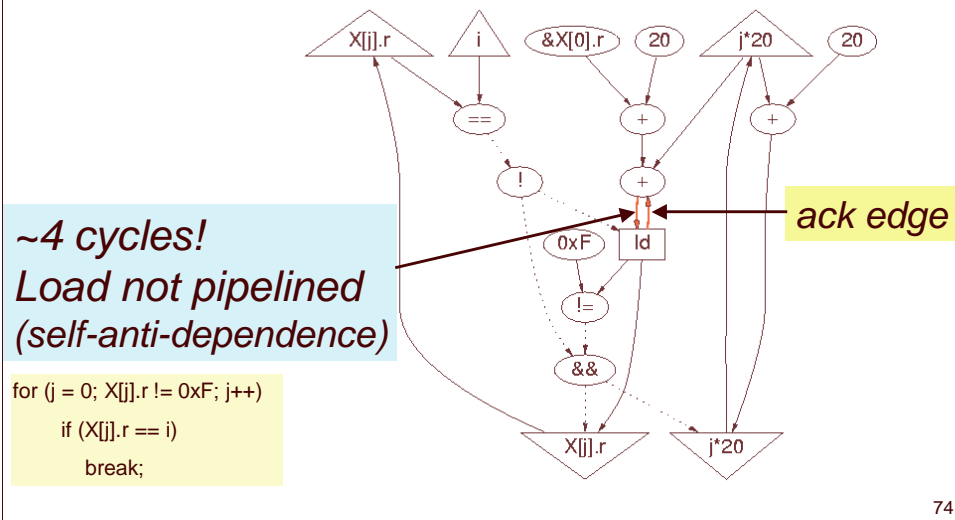
But not if branch prediction guesses their outcome. If it does, the critical path is 1 instruction long!

Critical Path with Prediction



When predicting the load-controlling predicate is on the critical path.

Prediction + Load Speculation



74

When we allow loads to issue speculatively (as a processor also does), the critical path contains an ack edge, because the latency of the load is 3 cycles. There is a dependence between the output register of the load in different iterations.

OOO Pipe Snapshot

LOOP:

L1: beq \$v0,\$a1,EXIT ; X[j].r == i

L2: addiu \$v1,\$v1,20 ; &X[j+1].r

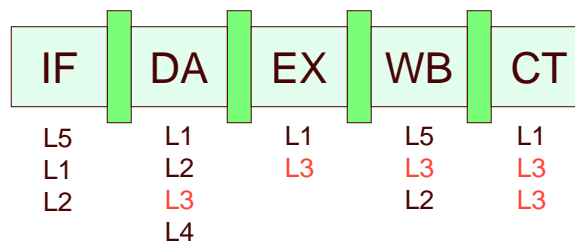
L3: lw \$v0,0(\$v1) ; X[j+1].r

L4: addiu \$a0,\$a0,1 ; j++

L5: bne \$v0,\$a3,LOOP ; X[j+1].r == 0xF

EXIT:

*register
renaming*



75

A processor however can have multiple instances of the load simultaneously active, since each uses a different physical register, due to register renaming.

Unrolling?

```
for(i = 0; i < 64; i++)    {  
    for (j = 0; X[j].r != 0xF; j+=2) {  
        if (X[j].r == i)  
            break;  
        if (X[j+1].r == 0xF)  
            break;  
        if (X[j+1].r == i)  
            break;  
    }  
    Y[i] = X[j].q;  
}
```

← *when 1 iteration*



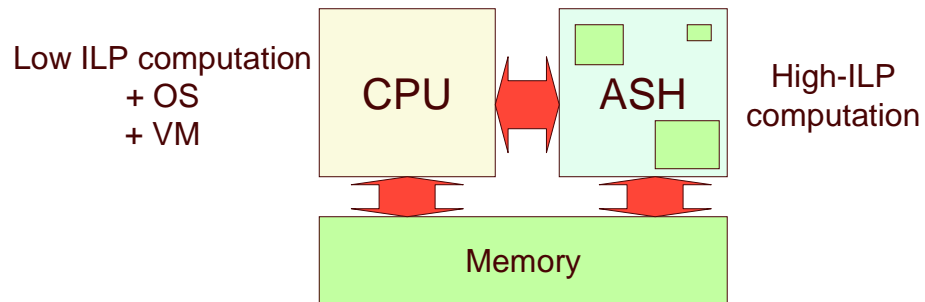
[back](#) [back to talk](#)

76

Unrolling does not help when the number of iterations may be small.

Overhead projector image from staples.com

Ideal Architecture



[back](#)

77

CPUs are very convenient; they will probably be around forever, but they will not evolve as fast as in the past.

We can envision delegating to them “fringe” jobs, while we run the core computation on the ASH fabrics.