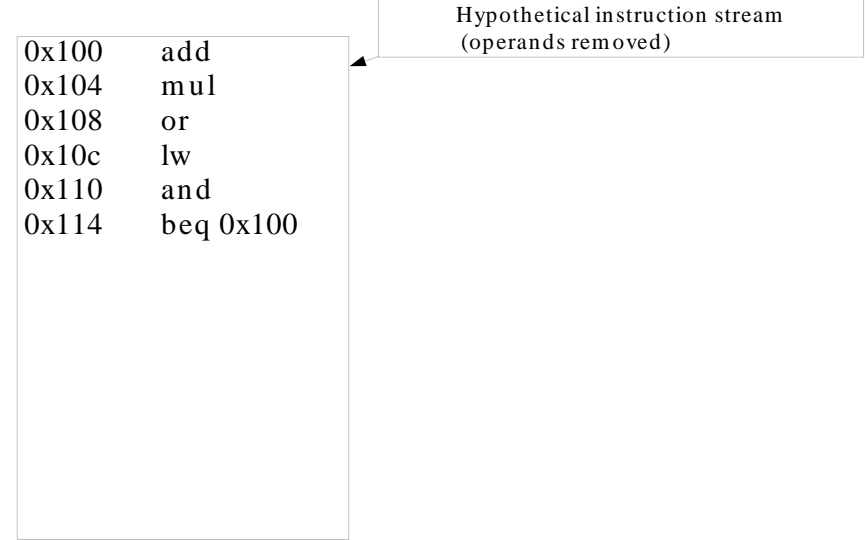


# Advanced Computer Architecture: s1/2005

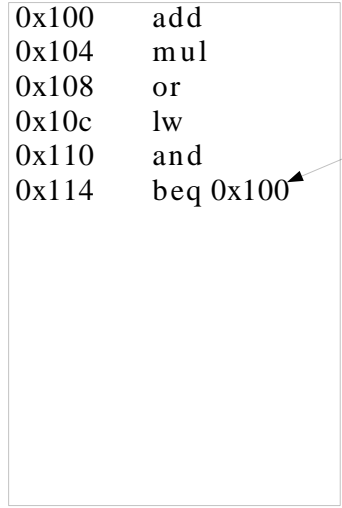
Project Presentation – David Mirabito

Handling branches through context forking

Currently:

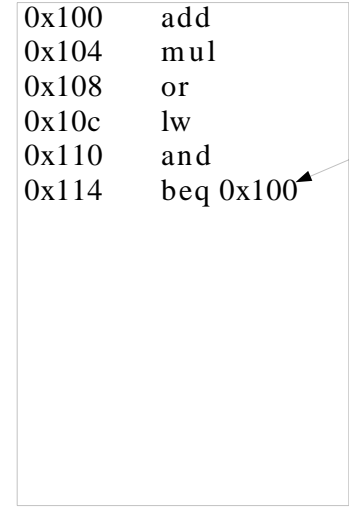


Currently:



What now? The operands of this branch won't be fetched, compared and have the result known until end of the EX stage... 3 more cycles!

Currently:



Currently handled by large, complex, power consuming branch prediction logic. In test-printf this is found to be 91.9% accurate for dir prediction, and 90.3% accurate with target address.

Currently:

```
0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
0x100  add
0x104  mul
0x108  or
```

Assumue: Branch SHOULD be taken  
Predicted correctly

At this stage (3 cycles later) we can be sure we  
predicted correctly.

Currently:

```
0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
0x100  add
0x104  mul
0x108  or
```

Assumue: Branch SHOULD be taken  
Predicted correctly

At this stage (3 cycles later) we can be sure we  
predicted correctly.

**BUT, 10% of the time...**

Currently:

```
0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
```

Assumue: Branch SHOULD be taken  
Predicted incorrectly

Currently:

```
0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
0x118  ori
0x11c  sub
0x120  sw
```

Assumue: Branch SHOULD be taken  
Predicted incorrectly

Here we realise we were wrong.  
Have to nullify incorrect insts and start again.

The amount of nullified instructions will only  
increace as fetch, dispatch and execute widths  
grow. On this simpliscalar model, this can be up  
to 4 instructions per cycle: 12 potential  
instructions wasted.

Currently:

```

0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
0x118  ori
0x11e  sub
0x120  sw
0x100  add
0x104  mul
0x108  or

```

Assumue: Branch SHOULD be taken  
Predicted incorrectly

Here we realise we were wrong.  
Have to nullify incorrect insts and start again.

The amount of nullified instructions will only  
increase as fetch, dispatch and execute widths  
grow. On this simplescalar model, this can be up  
to 4 instructions per cycle: 12 potential  
instructions wasted.

Currently:

```

0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and
0x114  beq 0x100
0x118  ori
0x11e  sub
0x120  sw
0x100  add
0x104  mul
0x108  or

```

Assumue: Branch SHOULD be taken  
Predicted incorrectly

These represent wasted fetch bandwidth,  
computation cycles and instigate fetching  
unneded data/ insts from  
system memory.

8.1% x 254825 branches comitted  
= 20640 mispredicted branches  
= 61922 wasted cycles  
= 4.9% of execution time.

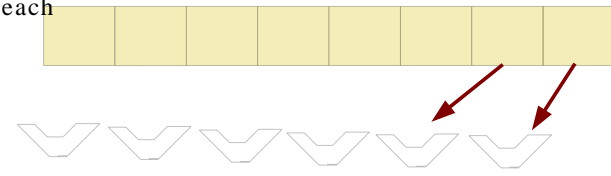
Elsewhere...

Instruction  
streams from  
2 independent  
threads

0x100	add	0x400	sw
0x104	mul	0x404	sdd
0x108	or	0x408	mov
0x10c	lw	0x40c	sll
0x110	and	0x410	addi
0x114	beq 0x100	0x414	lui
0x118	ori	0x418	subu
0x11c	sub	0x41c	sub
0x120	sw	0x420	sb

Lookahead  
window.  
Split 50/50 for each  
thread

Multiple  
execute units  
in a  
superscalar  
arch



HyperThreading allows two threads to be run concurrently, with one  
using the execution units that the other doesn't need. Backend of cpu is  
the similar, only instructions need to writeback to correct register file.

Combining the two...

```

0x100  add
0x104  mul
0x108  or
0x10c  lw
0x110  and

```

Initially, things proceed as normal.

### Combining the two...

```

0x100    add
0x104    mul
0x108    or
0x10c    lw
0x110    and
0x114    beq 0x100
    
```

Initially, things proceed as normal.

Until a branch is hit, in which case the single stream becomes two logical threads, one following each path of execution (taken / not taken)

```

0x118    ori
    
```

```

0x100    add
    
```

### Combining the two...

```

0x100    add
0x104    mul
0x108    or
0x10c    lw
0x110    and
0x114    beq 0x100
    
```

Now, the fetch bandwidth is shared between each of the new 'forked contexts' (2 insts/cycle each, instead of 4)

Beyond the frontend things remain similar, as in HT. Only we must ensure instructions only retire to the appropriate context

```

0x118    ori
0x11c    sub
    
```

```

0x100    add
0x104    mul
    
```

### Combining the two...

```

0x100    add
0x104    mul
0x108    or
0x10c    lw
0x110    and
0x114    beq 0x100
    
```

At this stage, the result of the comparison is made known.

```

0x118    ori
0x11c    sub
0x120    sw
    
```

```

0x100    add
0x104    mul
0x108    or
    
```

### Combining the two...

```

0x100    add
0x104    mul
0x108    or
0x10c    lw
0x110    and
0x114    beq 0x100
0x100    add
0x104    mul
0x108    or
    
```

At this stage, the result of the comparison is made known.

We can now take the correct context and merge any changes to its register file / memory back with the parent context

```

    
```

```

    
```

Unfortunately...

Implementing this functionality on top of sim-outorder.c within the simplescalar test suite was a **much** larger undertaking than originally anticipated.

Currently:

Can fork context upon a branch instruction and split incoming instructions between these 50/50. When the branch reaches writeback the appropriate context is selected and the modified registers are written back to the parent.

But:

Execution does not run to completion, memory reads/writes across contexts are being corrupted, this leads to an incorrect address being loaded and an attempted read from 0x00000000, crashing the app.

However:

This is after 4043 cycles, or 3626 instructions, so I will attempt to make what conclusions I can.

Stats...

Num branches encountered:	781
% cycles in forked state:	64.3% (2603 / 4043)
avg num insss in context[0]:	
avd num insts in context[1]:	

% time stalled context[0]:	
% time stalled context[1]:	
% time stalled context[2]:	

avg amount of registers / mem locations writtenback during context:

Observations...

Some things I noticed whilst stepping through traces:

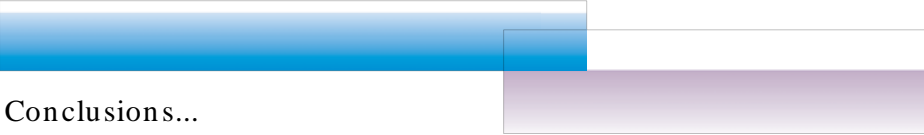
- \* This will only ever be worthwhile if we only fork the times we mis-predict. Perhaps not necessary to do this every branch.
- \* Still quite useful during compulsory misses in the branch predictor
- \* Can aid performance by prematurely warming cache for the exit code of a loop. We can brace against the cost of tlb/cache miss on this code during the 2<sup>nd</sup> and other iterations of a loop.
- \* It might be beneficial to take advantage of known compiler quirks: eg: beq r0 r0 XXX should be considered a non-conditional branch and not be forked. It is advantageous that this isn't currently done for J insts.
- \* It is allowable in the PISA architecture to have 2 adjacent branch insts. Quite often one or both child contexts stall when they too come across a branch and cannot fork. This indicates that more contexts would allow increased performance (and troubles)

Wishlist...

Other things to implement: (in increasing order of need):

- \* Varying priorities to each context (eg: 27/75), based on confidence level of the branch predictor.
- \* Support for more than 1 level of forking, so if a forked context encounters another branch it no longer needs to stall.
- \* Smarter handling of JAL / JR combinations. Currently can only be done in root context, to save corruption of the return addr stack in the branch predictor
- \* Better reporting / accounting.
- \* Complete program correctness

Some of these can/ will be achieved before the report is due.



## Conclusions...

- \* In all likelihood, this idea is not worth being implemented, considering cost:benefit ratio.
- \* Have read other papers doing similar things that concluded the same thing.
  
- \* Implementing a new idea and seeing how it affects the program trace++
- \* Yet still immensely useful as a learning exercise: Actually seeing register, control and data dependencies work themselves out in an out of order environment perfectly brings home ideas learned in class
  
- \* Also skills involved in working on a large, foreign codebase built upon