hp
invent

IA-64 Architecture and Its Performance

Hsin-Ying Lin hsin-ying_lin@hp.com

Kevin Wadleigh kevin_wadleigh@hp.com

So, What is IA-64?

Next-Generation Microprocessors

Most Significant Architecture Since 80386

- 64-Bit Architecture (Post RISC, 32-Bit)
- Explicitly Parallel Instruction Computing (EPIC)
- Comprehensive Predication
- Enhanced Speculation

The IA-64 Advantages

Performance Optimized

- Breakthrough Performance for Workstation and Server Applications
- Multi-Platform Support
- Delivering Next-Generation Computing Today

High-End Application Support

- E-services
- Technical Computing
- Business Intelligence

Today's Architecture Challenges:

- Memory Latency
- Branch Misdirects

- Too Few Registers
- Hardware-Based Instruction Scheduling

- Memory Addressing Efficiency
- Hardware, I/O Capacity

HP's IA-64 Target Customers

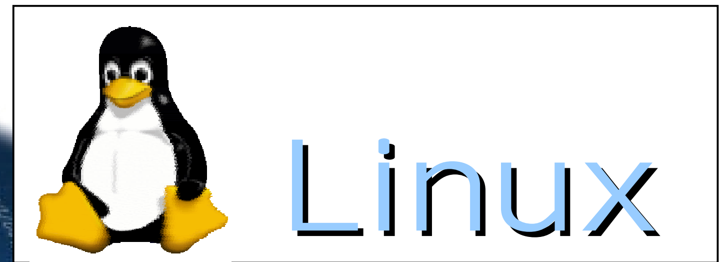
- Memory Addressing Efficiency
- Breakthrough OLTP Performance

- Network Security

- Floating-Point Mathematics

What are the IA-64 Customer Benefits?

HP-UX:
UNIX
For the Enterprise



HP's Binary Compatibility Advantage:

Itanium & HP-UX, Windows NT, and Linux

Tens of Thousands of Applications

(32-bit and 64-bit)

HP-UX

Linux

NT

IA-64

PA Apps

IA-32 Apps

Pervasive Technology

PA-RISC

Binary Compatibility

IA-32

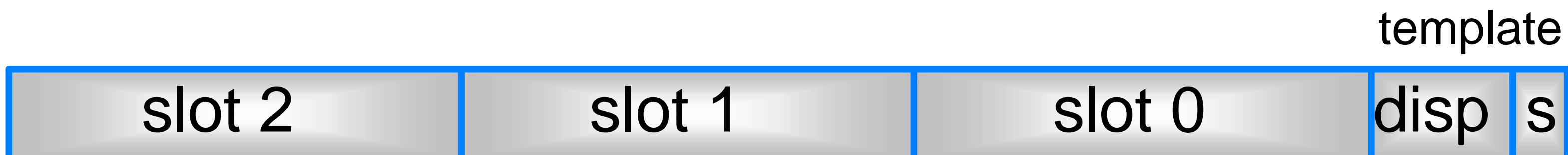
IA-64

- Architecture resources
- Predication
- Register rotation
- Speculation
- Processors
- Performance Tuning for ISV

Machine Resources

Instruction Bundling

- 128-bit aligned instruction bundles contain
  - three 41-bit instructions
  - 5-bit template consisting of 4-bit dispersal template + 1 stop bit
- Branches are to bundle boundaries
- Implementations are allowed to have any number of functional units
- Template controls dispersal to functional units: Memory, Integer, Floating-point, Branch, Long immediate



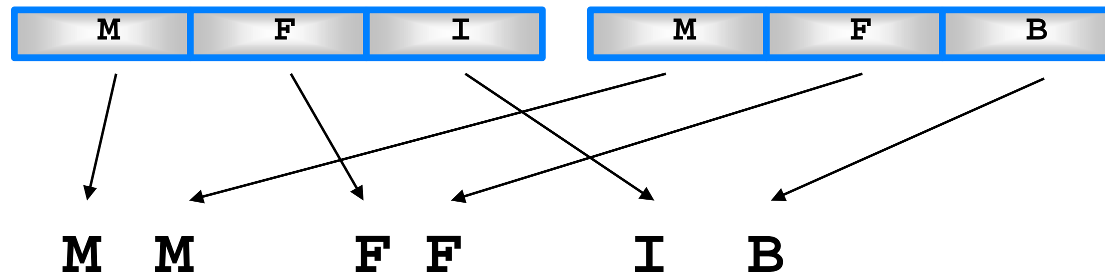
Templates and Dispersal

Templates:

0 1 2
M L X
M I I
M I / I
M M I
M/ M I
M M F
M M B
M F I
M F B
M I B
M B B
B B B

/ = stop bit

Each template is available with stop bit at end



Dispersal maps the instructions to functional units. This example shows a CPU that can perform at least two M units, two F units, an I unit and a B unit in one cycle. (Itanium can perform 2M, 2I, 2F, 3B)

Parallelism – Code example y = x + y

- Instruction stream

```
ldfd      f10 = [r21]     //load x
ldfd      f11 = [r24]     //load y
;;                        //break in parallelism
fadd.d    f10 = f10,f11   //y = x + y
;;                        //break in parallelism
stfd      [r24]= f10      //store y
```

- Maps to

```
M         M       nop.I ;;  //MMI
nop.m     F       nop.I ;;  //MFI
M         nop.I   nop.I     //MII
```

Predication – removes branches

- Converts a control dependence to a data dependence
  - Compare instructions set predicate bit
  - Predicated instructions are either normally executed or they do not affect the architectural state – example code below

```
if (ix .eq. iy) then
  a = 0
else
  c = 0
endif
```

- Becomes

```
        cmp.eq   p6,p7 = r16,r17 ;;
(p6)    fadd.d   f4 = f0,f0
(p7)    fadd.d   f5 = f0,f0
```

- Maps to

```
nop.M    I    nop.I ;;
nop.M    F    nop.I
nop.M    F    nop.I
```

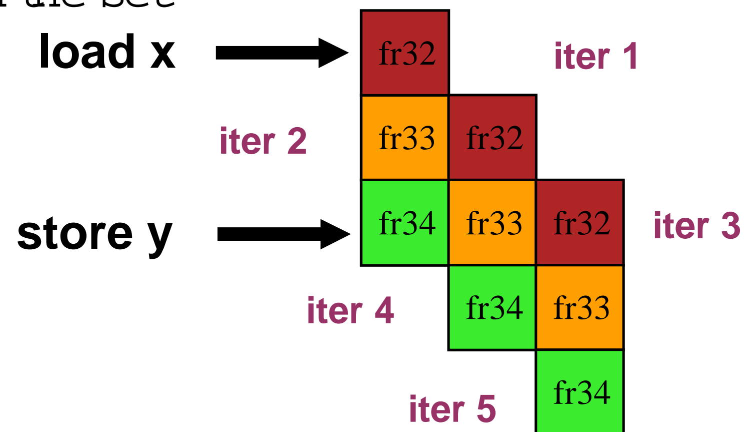
Software Pipelining

- Traditional architectures use loop unrolling to hide latencies
  - High overhead: extra code for loop body, prologue, and epilogue
- Synergistic use of IA-64 features allows efficient pipelining
  - Special branches cause registers to rotate
  - Register rotation removes need for explicit unrolling
  - Predicate rotation removes prologue & epilogue

Register Rotation

- Key to good loop performance
  - software pipelining uses register rotation
  - acts like short vectors
  - with each iteration of a loop, data in rotation registers moves to the next register in the set
- Code example

```
do i = 1,n
  y(i) = x(i)
enddo
```



- Becomes

```
loop:
  ldfd        fr32 = [r26],8   // load x and incr address
  stfd        [r22] = fr34,8   // store y (2 iter after load of x)
  br.ctop.sptk loop ;;          // loop instruction - reg. rotate
```

- This example omits predication necessary for prologue and epilogue

Software Pipelining using Rotation and Predication

- DAXPY inner loop

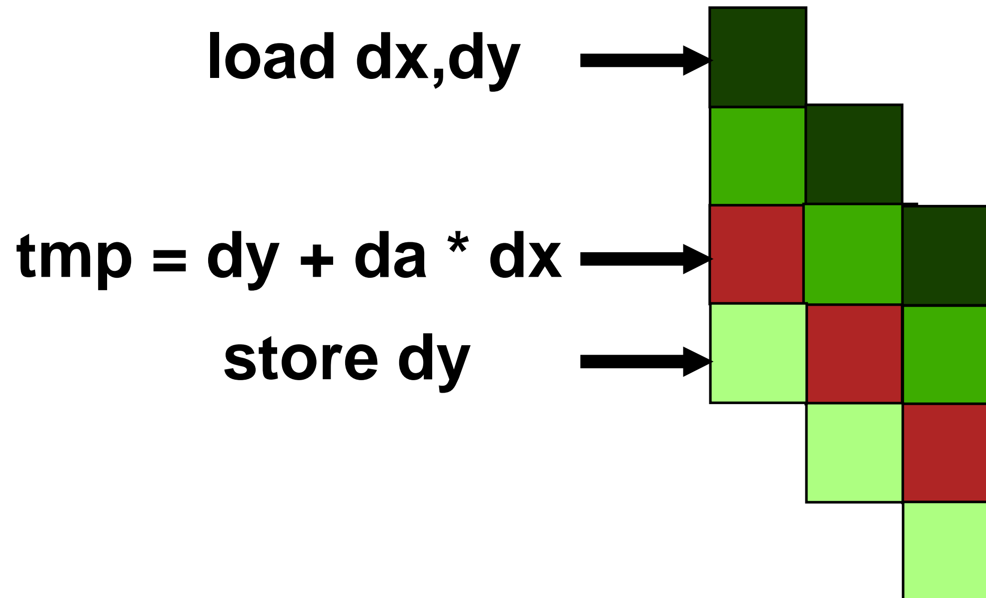
```
for (i= 0;i< 3;i++)
  dy[i] = dy[i] + (da * dx[i]);
```

(2 loads, 1 fma, 1 store per iteration)

- Consider a hypothetical processor than can perform
  - 2 loads, 1 fma, 1 store per iteration
  - load latency of 2 cycles
  - fma latency of 1 cycle
- (Itanium can perform: 2M, 2I, 2F, 3B per cycle)

Example: Pipeline

Each column represents 1 source iteration



Example Code

```
.rotf dx[3], dy[3], tmp[2]          // short vectors
      mov    ar.lc = 2              // lc = loop count
                                    //    = #iterations-1
      mov    ar.ec = 4              // epilogue count
                                    // #stages (or # pred)
      mov    pr.rot = 0x10000       // p16=1, p17=p18=…=0
      ;;
looptop:
 (p16) ldfd   dx[0] = [dxsp],8
 (p16) ldfd   dy[0] = [dysp],8
 (p18) fma.d  tmp[0] = da, dx[2], dy[2]
 (p19) stfd  [dydp] = tmp[1],8
      br.ctop looptop
      ;;
```

Loop Execution

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st

Predicates	
16:	1
17:	0
18:	0
19:	0

Loop 1

LC=2 EC=4

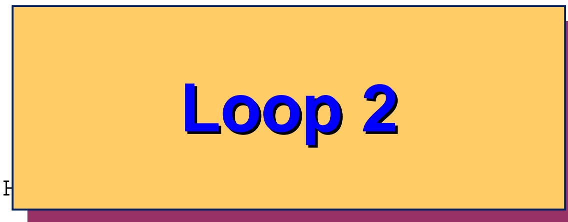
Loop Execution

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

Predicates	
16:	1
17:	1
18:	0
19:	0

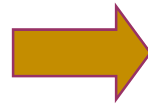


LC=1 EC=4

Loop Execution

Execution Sequence

(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st



Predicates	
16:	1
17:	1
18:	1
19:	0

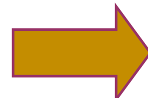
Loop 3

LC=0 EC=4

Loop Execution

Execution Sequence

(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st

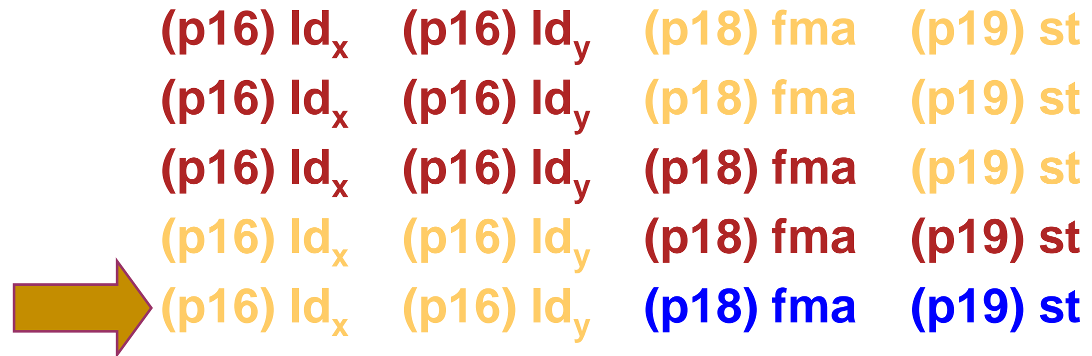


Predicates	
16:	0
17:	1
18:	1
19:	1

Epilogue 1

LC=0 EC=3

Loop Execution



Predicates	
16:	0
17:	0
18:	1
19:	1

Epilogue 2

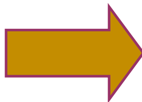
LC=0 EC=2

Loop Execution

Execution Sequence

(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st

Predicates	
16:	0
17:	0
18:	0
19:	1



Epilogue 3

LC=0 EC=1

Loop Execution

Execution Sequence

(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st
(p16) ld_x	(p16) ld_y	(p18) fma	(p19) st

fall through

Predicates	
16:	0
17:	0
18:	0
19:	0

Done

LC=0 EC=0

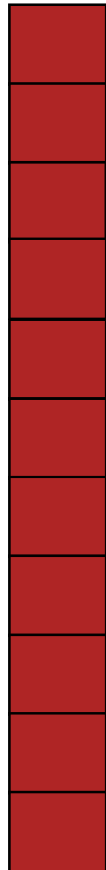
Pipeline and Latency

load dx,dy

- Suppose we change to latencies to
  - load latency of 6 cycles
  - fma latency of 4 cycles
- Each column represents 1 source iteration

tmp = dy + da * dx

store dy



Updated Loop

```
.rotf dx[7], dy[7], tmp[5]

       mov    ar.lc = 2             // #iterations-1
       mov    ar.ec = 11            // #stages
       mov    pr.rot = 0x10000
       ;;
looptop:
 (p16) ldfd   dx[0] = [dxsp],8
 (p16) ldfd   dy[0] = [dysp],8
 (p22) fma.d  tmp[0] = da, dx[6], dy[6]
 (p26) stfd  [dydp] = tmp[4],8
       br.ctop looptop
       ;;
```

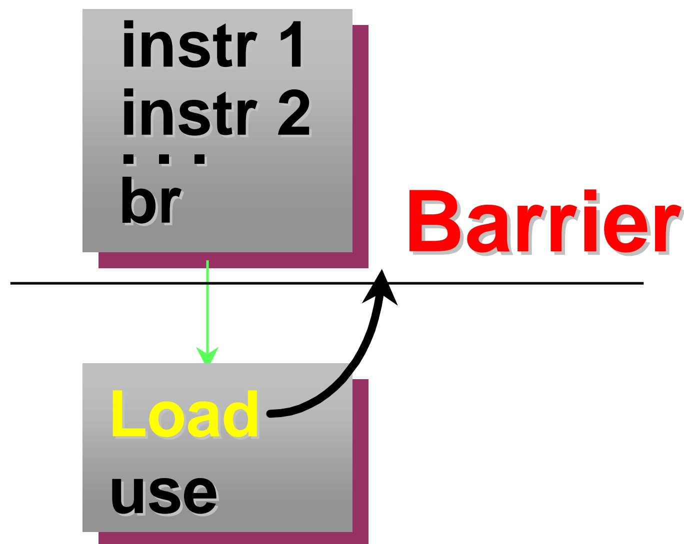
Rotation: Summary

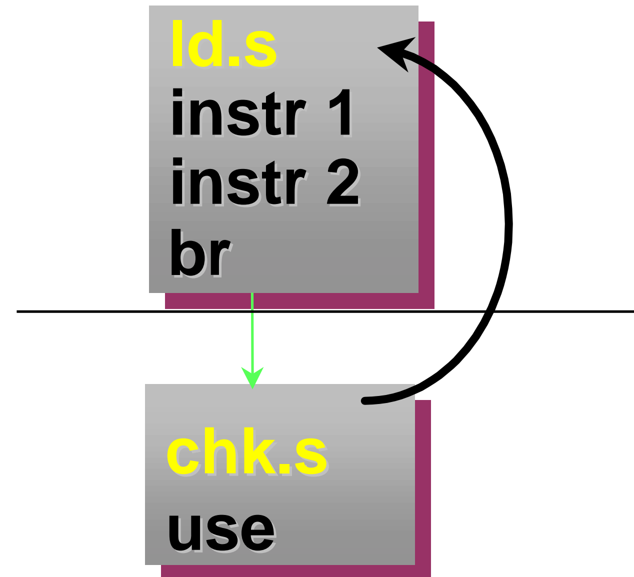
- Loop pipelining maximizes performance; minimizes overhead
  - Avoids code expansion of unrolling and code explosion of prologue and epilogue
  - Smaller code means fewer cache misses
  - Greater performance improvements in higher latency conditions
- Reduced overhead allows S/W pipelining of small loops with unknown trip counts
  - Typical of integer scalar codes

Speculation

- Memory is very far away, so we would like to load data well before its use
- Prefetch instructions will not prefetch pages that have not been mapped by the TLB
- Prefetch instruction will not prefetch data from invalid addresses
- Speculative loads allow users to try to load data from addresses regardless of whether or not the data will be used, the address will be written to in the meantime, or the address is known to be valid. What could go wrong?
- Control speculation versus data speculation
  - Control - moves loads around branches
  - Data - moves loads around stores

Control Speculation
Move Loads before Branches

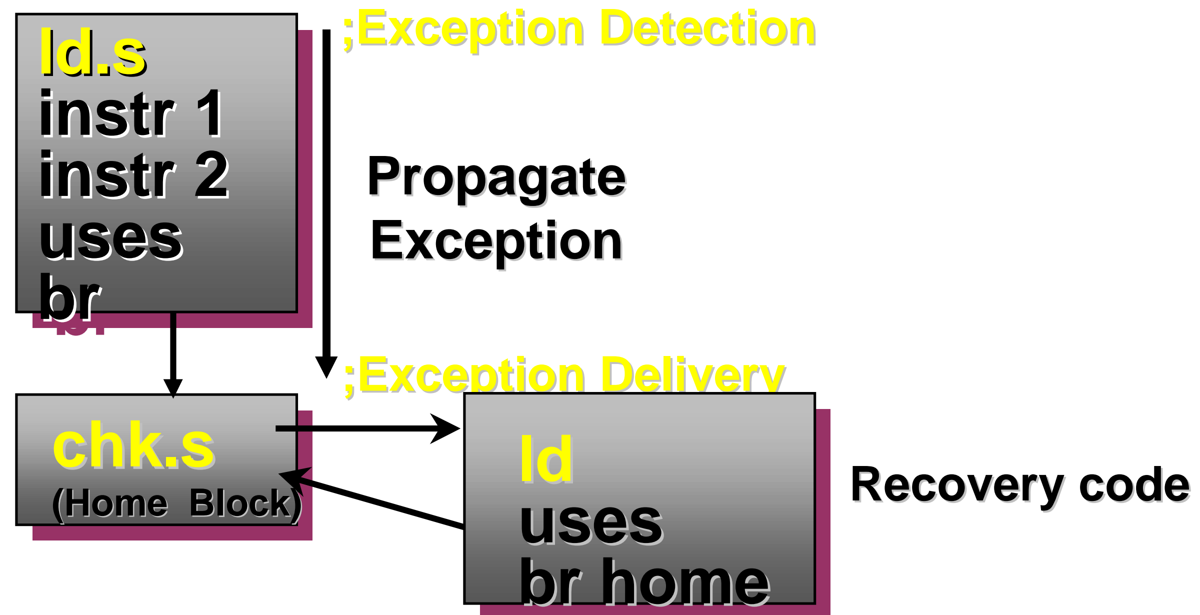




Control Speculation

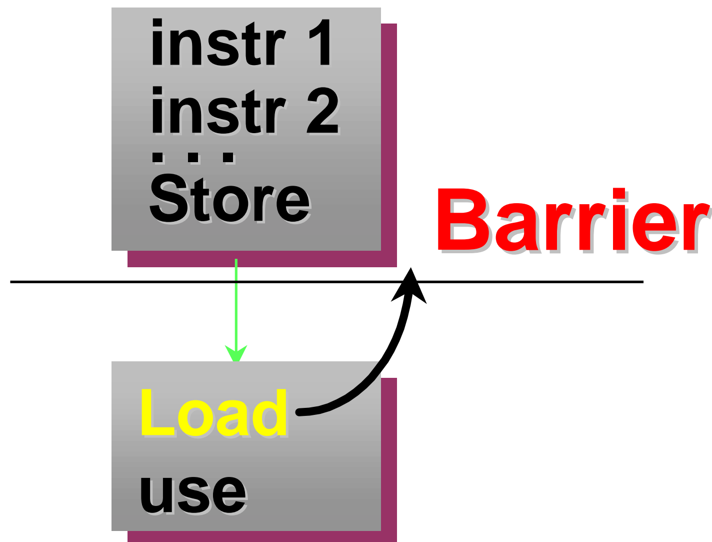
- Regular loads are replaced with speculative load, followed by speculative chk instruction
  - ld is replaced by ld.s, chk.s
  - ldf is replaced by ldf.s, chk.s
- For safety, special values are used for illegal returns
  - Integer loads set the Not a Thing (NaT) bit associated with the target general register
  - Floating-point loads set the target floating-point register to a special value: NaTVal = 0, 0x1FFFE, 0... 0

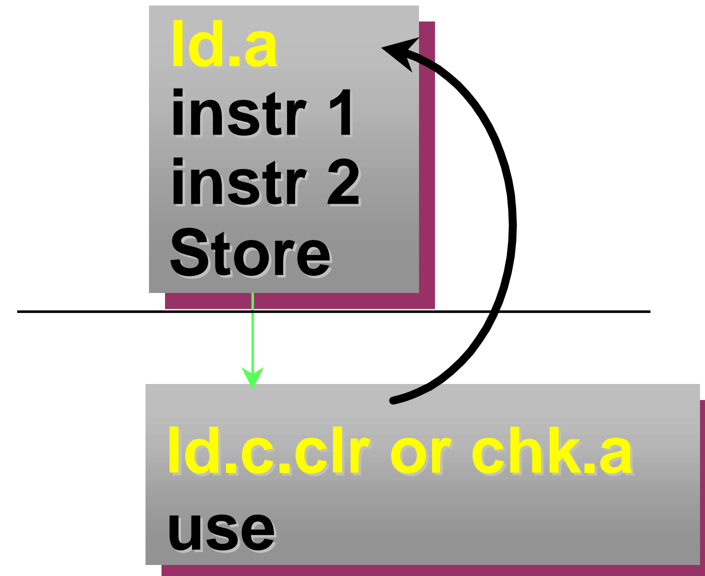
NaT ("Not a Thing") and NaTVal



- NaT (or NaTVal) indicates:
  - whether or not an exception has occurred
- If NaT (or NaTVal) set during ld.s (ldf.s), it is checked by the instruction chk.s (usage: chk.s reg, target), then branch to target
  - code at target can redo the bad and take the normal exception

Data Speculation
Move Loads before Stores





Data Speculation

- Moves loads around possibly conflicting stores
- Regular loads are replaced with advanced loads, followed by either a check load or advanced chk instruction
- If the only instruction that was ambiguous is the load, then a check load can be performed after the load
  - ld is replaced by ld.a, ld.c.clr
  - ldf is replaced by ldf.a, ldf.c.clr
- If there are several instructions that depend on the advanced load, then a chk.a can be used to branch to fix up code
  - ld is replaced by ld.a, chk.a
  - ldf is replaced by ldf.a, chk.a

Data Speculation - example

- If the only instruction that was ambiguous is the load, then a check load can be performed after the load

```
st        [r4] = r12
ld        r3 = [r5] ;;
```

- Becomes

```
ld.a      r3 = [r5] ;; //advanced load - note a suffix
…
st        [r4] = r12
ld.c.clr  r3= [r5] // if the addr has been modified,
                   // redo it
```

Processor Evolution

Performance

Next generation

EPIC, IA-64
6 instructions/cycle

HEWLETT PACKARD

intel

Superscalar RISC
≤4 instructions / cycle

RISC
≤1 instruction / cycle

CISC

1 micron - > .5 micron --> .35 micron --> .25 micron --> .18 micron --> .13 micron

20-30% increase per year due to advances in underlying semiconductor technology

.3 ins / cycle

Time

HP Microprocessor Roadmap

Madison Performance

Deerfield Price/Perf

McKinley

PA-8800

PA-8700

Itanium

PA-8600 550MHz

PA-8500 360-440MHz

PA-8200 240MHz

PA-8000 180MHz

'96 '98 '00 '02 Future

Performance Tuning for ISV

Hsin-Ying Lin and Kevin Wadleigh MSW, TCD

Characteristics of ISV Application

One of our commercial ISV application involves a lot of floating computation. On their benchmark suite, over 50% of the computation time was concentrated in about 25% of the routines. Furthermore, about 40% of computation time actually spend in two kernels, WAXPY and DOTPRODUCT.

WVAXPY's C Code

```
wvaxpy(w, x, y, n, alpha)
double *w, *x, *y, alpha;
int n;
{
  while( n-- > 0 ) *(w++) = *(x++) +alpha*
*(y++);
}
```

IA-64 Compiler Generate Code for WAXPY – Ideally

Instructions		Template		Clocks on
				Merced
				L1
L L x+	S L x+	MMF	MMF	2
L S -	L L x+	MMI	MMF	2
S L x+	L S -	MMF	MMI	2
L L x+	S L x+	MMF	MMF	2
L S -	L L x+	MMI	MMF	2
S L x+	LF S -	MMF	MMI	2
L LF -	LF - B	MMI	MIB	2
Total clocks for 8 iterations				14
Clocks per iteration				1.75

Note: LF indicates for prefetch instruction

IA-64 Compiler Generate Code for WAXPY

Instructions	Template	Clocks on Itanium L1
L L - L L -	MMI MMI	2
L L - L L -	MMI MMI	2
LF - x+ - - x+	MMF MMF	2
LF - x+ - - x+	MMF MMF	2
LF LF - S S -	MMI MMI	2
LF LF -	MMI	1
LF LF - S S B	MMI MMB	2
Total clocks for 4 iterations		13
Clocks per iteration		3.25

Note: LF indicates prefetch instruction

Assembly Code Generated by Compiler for WAXPY

```
..L11:
(p16)  ldfd         f44 = [r11], 64        // M
(p16)  ldfd         f47 = [r10], 32        // M
       nop.i        0                      // I
(p16)  ldfd         f34 = [r9], 32         // M
(p16)  ldfd         f32 = [r8], 32         // M
       nop.i        0                   ;; // I
(p16)  ldfd         f40 = [r17], 64        // M
(p16)  ldfd         f42 = [r16], 32        // M
       nop.i        0                      // I
(p16)  ldfd         f38 = [r15], 32        // M
(p16)  ldfd         f36 = [r14], 32        // M
       nop.i        0                   ;; // I
(p16)  lfetch.nt1   [r11], 8               // M
       nop.m        0                      // M
(p17)  fma.d.s0     f50 = f8, f45, f41     // F
       nop.m        0                      // M
       nop.m        0                      // M
(p17)  fma.d.s0     f51 = f8, f48, f43  ;; // F
(p16)  lfetch.nt1   [r17], 8               // M
       nop.m        0                      // M
(p17)  fma.d.s0     f45 = f8, f35, f39     // F
       nop.m        0                      // M
       nop.m        0                      // M
(p17)  fma.d.s0     f48 = f8, f33, f37  ;; // F
(p16)  lfetch.nt1   [r11], 8               // M
(p16)  lfetch.nt1   [r17], 8               // M
       nop.i        0                      // I
(p18)  stfd         [r19] = f46, 16        // M
(p18)  stfd         [r18] = f49, 16        // M
       nop.i        0                   ;; // I
(p16)  lfetch.nt1   [r11], 8               // M
(p16)  lfetch.nt1   [r17], 8               // M
       nop.i        0                   ;; // I
(p16)  lfetch.nt1   [r11], -56             // M
(p16)  lfetch.nt1   [r17], -56             // M
       nop.i        0                      // I
(p17)  stfd         [r19] = f50, 16        // M
(p17)  stfd         [r18] = f51, 16        // M
       br.ctop.dptk.few ..L11           ;; // B
```

Hand Tuned IA-64 WAXPY Assembly Code

Instructions	Template	Clocks on Itanium L1
LP S x+ LP S x+	MMF MMF	2
LP S x+ LP S x+	MMF MMF	2
LP S x+ LP S x+	MMF MMF	2
LP S x+ LP S x+	MMF MMF	2
LF LF - LF - B	MMI MIB	2
Total clocks for 8 iterations		10
Clocks per iteration		1.25
Speedup ratio of HLL/Assembly		1.4
Speedup ratio of Compiler/Assembly		2.6

Note: LF indicates prefetch instruction; LP means quad word load

WVAXPY Assembly vs C Code's Performance

on IA-64 Itanium 499 MHz

Mflops: 900, 800, 700, 600, 500, 400, 300, 200, 100, 0

tccp
assem

N: 10, 20, 100, 200, 1000, 2000, 10000, 20000, 100000, 200000, 1000000

tccp -- +O2 +Onoparmsoverlap +Odataprefetch

WVAXPY Assembly vs C Codes' Speedup

on Itanium 499 MHz CPU

Speedup Ratio

3.5
3
2.5
2
1.5
1
0.5
0

assem/tccp

10 20 100 200 1000 2000 10000 20000 100000 200000 1000000

N

tccp -- +O2 +Onoparmsoverlap +Odataprefetch

DOTPRODUCT 's C Code

```
double dotproduct(a, b, n)
double *a, *b;
int n;
{
    double dot;
    dot = 0.;
    while( n-- > 0) dot += *(a++) * *(b++);
    return(dot);
 }
```

IA-64 Compiler Generate Code for DOTPRODUCT

Instructions	Template	Clocks on Itanium
L L x+ LF LF I	MMI MMI	2
M M B	MMB	1
Total clocks for 1 iterations		4
Clocks per iteration		4

Note: The floating point latency determines the rate.

IA-64 DOTPRODUCT Assembly Code Generated by Compiler

```
.L5:
(p16) ldfd          f32 = [r8],8          //M  [line/col 7/20]
(p16) ldfd          f35 = [r14],8         //M  [line/col 7/20]
(p18) fma.d.s0      f8 = f37,f34,f8       //F
(p16) lfetch.nt1    [r9],8                //M
(p16) lfetch.nt1    [r10],8               //M
      nop.i         0                   ;; //I
      nop.m         0                     //M
      nop.m         0                     //M
      br.ctop.dptk.few  .L5             ;; //B [line/col 7/11]
```

IA-64 Compiler Generated Code for DOTPRODUCT – Ideally

Instructions	Template	Clocks on Itanium
L L x+ L L x+	MMF MMF	2
L L x+ L L x+	MMF MMF	2
L L x+ L L x+	MMF MMF	2
L L x+ L L x+	MMF MMF	2
LF LF B	MMB	1
Total clocks for 8 iterations		9
Clocks per iteration		1.13

Hand Tuned IA-64 DOTPRODUCT Assembly Code

Instructions	Template	Clocks on Itanium
		L1
LP x+ I LP x+ I	MFI MFI	1
LP x+ I LP x+ I	MFI MFI	1
LP x+ I LP x+ I	MFI MFI	1
LP x+ I LP x+ I	MFI MFI	1
LF LF B	MMB	1
Total clocks for 8 iterations		5
Clocks per iteration		0.6
Speedup ratio of HLL/Assembly		1.8
Speedup ratio of Compiler/Assembly		6.4

DOTPRODUCT Assembly Codes vs C Codes

on Itanium 499 MHz

Mflops

1400
1200
1000
800
600
400
200
0

10 20 100 200 1000 2000 10000 20000 100000 200000

N

assem
cc_d

cc_d: compiled with +O2 +Onoparmsoverlap +Odataprefetch

DOTPRODUCT's Assembly Code vs C Code

on Itanium 499 MHz CPU

Speedup

9
8
7
6
5
4
3
2
1
0

Speedup

10 20 100 200 1000 2000 10000 20000 100000 200000

N

C code compiled with +O2 +Onoparmsoverlap +Odataprefetch

IA-64 Assembly code Vs C code on Itanium

Speedup	In-Cache	Out-of-Cache
WAXPY	2.8x	2.2x
DOTPRODUCT	8.0x	6.5x

Overall speedup on ISV application suite = 1.3x (Estimation)

Note: C code is compiled with +O2 +Onoparmsoverlap +Odataprefetch

Performance Tuning Summary

- We estimate that we will improve this ISV applications performance on IA-64 platforms by 30%
- We will work closely with the ISV R&D team to ensure that the ISV's customers will enjoy performance improvements on HP platforms in the near future

invent

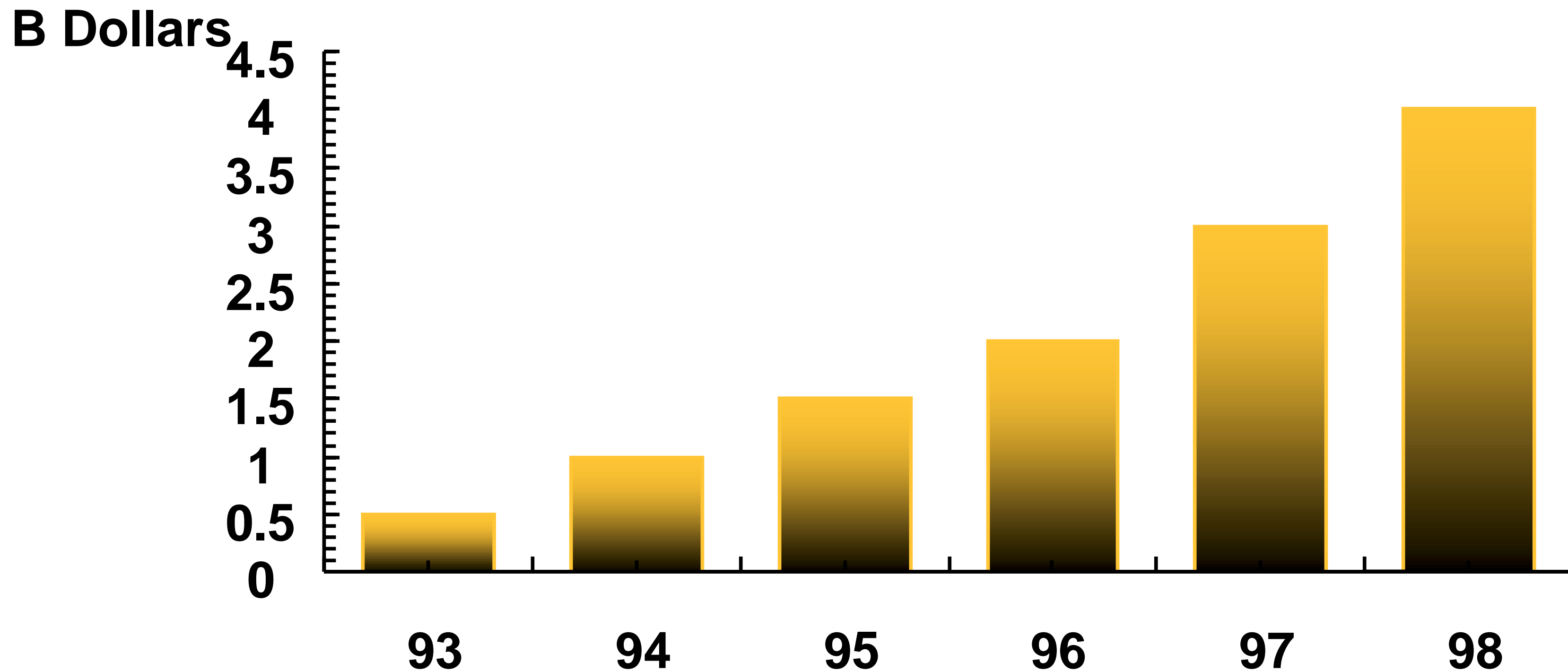
Backup Slides

Where we're going – IA64

- An *EPIC* story, years in the making
  - HP and Intel jointly designed instruction set
- *Now it can be told*
  - Intel IA-64 home page – http://developer.intel.com/design/ia-64
  - Recommended articles:
  - Next Generation Instruction Set Architecture' (Crawford, Huck) – http://developer.intel.com/design/ia-64/next/index.htm
  - Itanium Processor Microarchitecture Overview' (Sharangpani) – http://developer.intel.com/design/ia-64/microarch_ovw
  - The complete (>500 pages) 4 volume 'The IA-64 Architecture Software Developer's Manual' – http://developer.intel.com/design/ia-64/manuals

Chip production costs per year

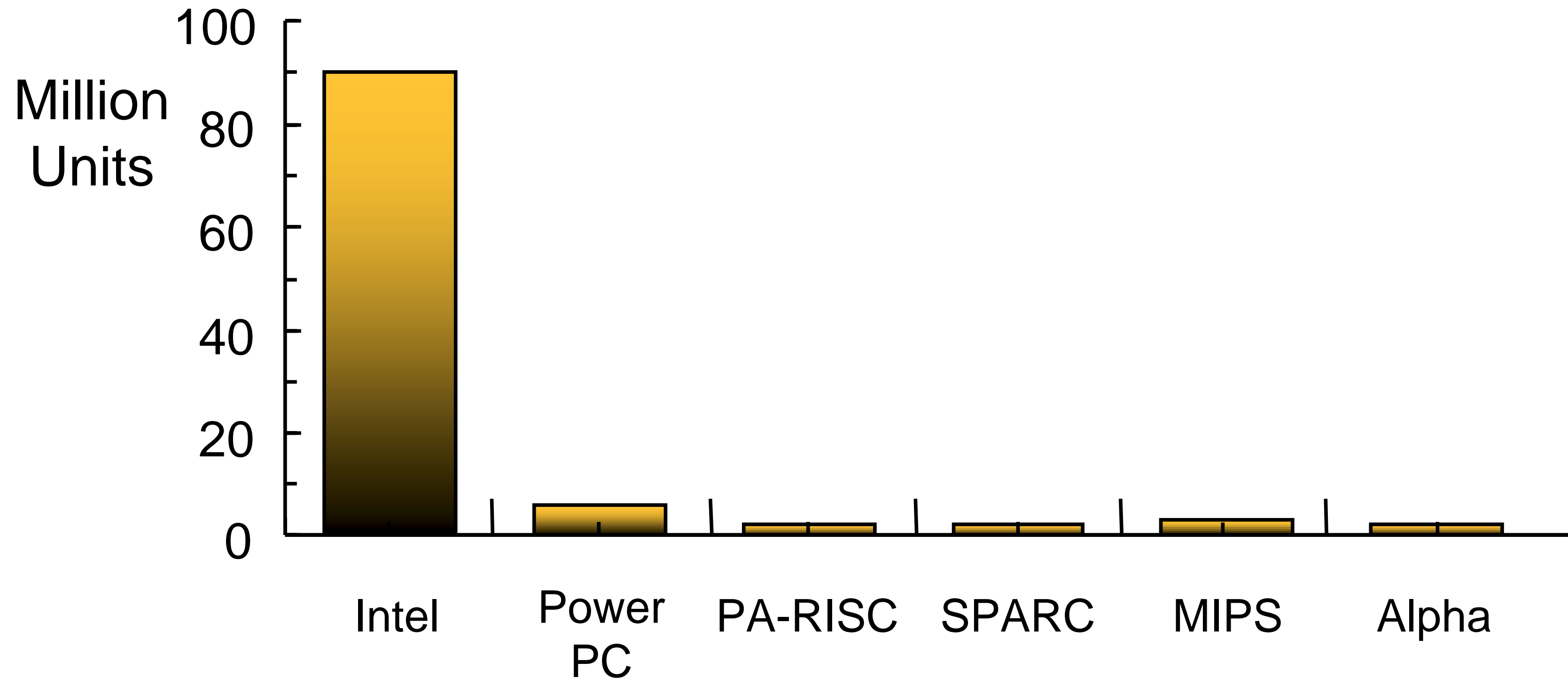
How many proprietary RISC vendors can continue to invest?



(...or, alternatively, face higher chip costs for low volumes with a third party fab?)

Microprocessor Production Capacity

Especially when fabrication and design costs must be recouped against relatively small unit volumes compared with merchants...



Hsin-Ying Lin and Kevin Wadleigh MSW, TCD

Where we've been

Processor families	CISC (Complex Instruction Set Computing)	Vector	RISC (Reduced Instruction Set Computing)	LIW (Long Instruction Word) Example: EPIC - Explicitly Parallel Instruction Computing
Architecture (Instruction Set)	IA-32	C-Series	PA-RISC, MIPS, Alpha	IA-64
Processors	Pentium III Xeon	C-4	PA-8500, R12000, Alpha21264	Itanium, McKinley

IA-64 Public Information

- Itanium
  - Multiple configurations for servers and w/s
  - Production in mid-2000
  - 0.18m CMOS technology
  - 4 DP Flops/cycle – 3 Gflop/s peak
  - Three level cache hierarchy (64-byte line size)
    - –L0: separate instruction and data
    - –L1: unified cache on die
    - –L2: off die, 2 or 4 MB
- McKinley
  - Clock > 1GHz, increased number of execution units, on die L2 cache
  - Increased bus bandwidth
  - Target production: late '01
- Madison
  - McKinley follow-on
  - Performance optimized on 0.13m technology
- Deerfield
  - McKinley follow-on
  - Price/performance optimized on 0.13m technology