The Example – CPU’s point of view

Explanation

- Branch prediction
  - Virtual function call is an indirect branch
  - Before determining the branch target, instruction can not be decoded
  - CPU tries to predict the target
    - Based on previous passes through the same instruction (at the same address)
      - Associative memory (address of branch instruction -> target address)
      - Heuristics, hardwired neural networks (AMD)
    - Call-return pairs
  - If prediction was wrong
    - Decoded and partially executed instructions are dropped
    - Decoding and execution of the correct instructions started from scratch
    - Delay: in the magnitude of 10 CPU cycles

Critical part of the code - Intel C++ 64-bit

- Source code

```
std::for_each( b, e, [&](A * p){ s += p->f(); });
```

- After procedure integration

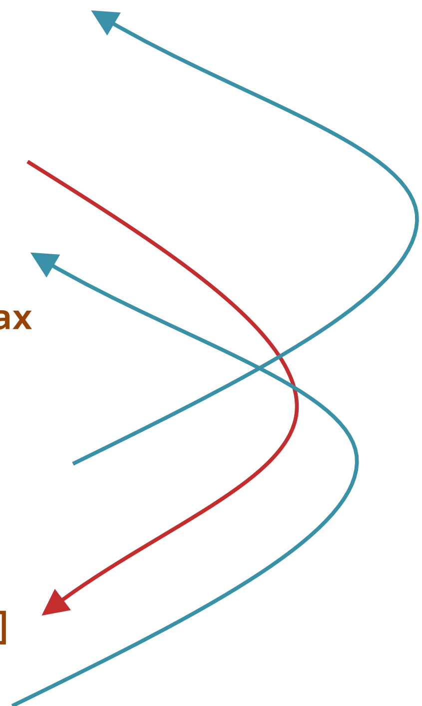
```
for(; b != e; ++ b) s += (*b)->f();
```

- Machine code of the loop

```
.lp:
mov        rcx, QWORD PTR [r14]
mov        rax, QWORD PTR [rcx]
call       QWORD PTR [8+rax]
add        r14, 8
add        DWORD PTR [88+rbp], eax
cmp        r14, r13
jne        .lp        ; Prob 82%
```

- Virtual function body

```
mov        eax, DWORD PTR [8+rcx]
ret
```



- Registers
  - r14 = b
  - r13 = e
  - rcx = * b
  - rax = VTable
  - eax = value of f()
  - rbp = stackframe
  - [88+rbp] = s
- Branches
  - well predicted
    - ret
    - jne
  - poorly predicted
    - call

Critical part of the code - Intel C++ 64-bit

- Source code

```
std::for_each( b, e, [&](A * p){ s += p->f(); });
```

- After procedure integration

```
for(; b != e; ++ b) s += (*b)->f();
```

- Machine code of the loop

```
.lp:
mov     rcx, QWORD PTR [r14]
mov     rax, QWORD PTR [rcx]
call    QWORD PTR [8+rax]
add     r14, 8
add     DWORD PTR [88+rbp], eax
cmp     r14, r13
jne     .lp      ; Prob 82%
```

- Virtual function body

```
mov     eax, DWORD PTR [8+rcx]
ret
```

- Dependencies
  - write-read
  - read-write
  - test-access
- Pipeline restart
  - Wrongly predicted branch (into virtual function)

Critical part of the code - Intel C++ 64-bit

- Virtual function body

```
mov        eax, DWORD PTR [8+rcx]
ret
```

- End of the loop

```
add        r14, 8
add        DWORD PTR [88+rbp], eax
cmp        r14, r13
jne        .lp       ; Prob 82%
```

- Beginning of the loop

```
.lp:
mov        rcx, QWORD PTR [r14]
mov        rax, QWORD PTR [rcx]
call       QWORD PTR [8+rax]
```

- Mispredicted branch target

```
mov        eax, DWORD PTR [4+rcx]
ret
```

- Microops (guess)

```
load       eax,[8+rcx]
load       t1,[rsp++]
jmp        t1
add        r14,8
load       t2,[88+rbp]
add        t2,eax
store      [88+rbp],t2
cmp        r14,r13,eflags
jne        .lp,eflags
load       rcx,[r14]
load       rax,[rcx]
load       t3,[8+rax]
store      [--rsp],rip
jmp        t3
load       eax',[4+rcx]
load       t4,[rsp++]
jmp        t4
```

Executing code with misprediction

	fetch+decode	load	ALU	retire+store
0	1..3			
1	4			
2	5..7			
3	8..9			
4	10..14			
5	1'..3'			
6		1		
7		. 2	4	
8		. . 5		
9		. .	8	1
10		. 10		2..4
11		. 16	6	5
12		. .		6..9
13		. 11		10
14		. 1'		
15		. .		
16		. 12		11
17		. 2'		
18		. .		
19		.		12..14

```
1:  load   eax,[8+rcx]
2:  load   t1,[rsp++]
3:  jmp    t1
4:  add    r14,8
5:  load   t2,[88+rbp]
6:  add    t2,eax
7:  store  [88+rbp],t2
8:  cmp    r14,r13,eflags
9:  jne    .lp,eflags
10: load rcx,[r14]
11: load rax,[rcx]
12: load t3,[8+rax]
13: store[--rsp],rip
14: jmp  t3
1': load eax',[4+rcx]
2': load t4,[rsp++]
3': jmp  t4
```

The same code with successful prediction

	fetch+decode	load	ALU	retire+store
0		1	4	
1		. 2	8	
2		. . 5		
3		. . 10		1
4		. .		2..4
5		.	6	5
6		11		6..9
0		. 1’	4’	10
1		. . 2’	8’	
2		. .12		11
3		. . 5’		
4		. . 10’		
5		..		12..14
6		.	6’	1’..4’
7		11’		5’..8’
0		. 1’’	4’’	9’-10’
1		. . 2’’	8’’	
2		. . 12’		11’
3		. . 5’’		
4		. . 10’’		

```
1:  load   eax,[8+rcx]
2:  load   t1,[rsp++]
3:  jmp    t1
4:  add    r14,8
5:  load   t2,[88+rbp]
6:  add    t2,eax
7:  store  [88+rbp],t2
8:  cmp    r14,r13,eflags
9:  jne    .lp,eflags
10: load   rcx,[r14]
11: load   rax,[rcx]
12: load   t3,[8+rax]
13: store  [--rsp],rip
14: jmp    t3
```

Column-oriented implementation - example

```
vector< int> data3d1;
vector< int> data3d2;
vector< int> data3d3;

vector< int> data5d1;
vector< int> data5d2;
vector< int> data5d3;
vector< int> data5d4;
vector< int> data5d5;

int s = 0;
s += std::reduce( data3d2.begin(),
  data3d2.end(), 0, std::plus<int>());
s += std::reduce( data5d3.begin(),
  data5d3.end(), 0, std::plus<int>());
```

- No classes at all
- Just arrays of elementary types
  - Extremely difficult and error-prone
  - No support for generic programming in this style
- For each column, data are dense
  - Improves cache behavior for column-oriented access
  - Degrades cache behavior for row-oriented access
  - Allows vectorization
    - std::reduce [C++17] allows compilers to vectorize automatically
    - The experiments were done with manual vectorization

Column-organized data – problems of vectorization

- Column-stored data

```
vector< int> data3d2;
```

- Non-vectorized naive code

```
int s = 0;
for ( auto && x : data3d2 )
  s += x;
```

- Good compilers can vectorize this code automatically
- There is no guarantee that vectorization is actually done
- Manual vectorization may be required to ensure performance

- SIMD instructions
  - Intel/AMD: MMX/SSE/AVX2/AVX512
- Automatic vectorization
  - Compilers generate vector instructions automatically
  - Hints from programmers needed
- Manual vectorization
  - Library functions corresponding to individual vector instructions
    - Knowledge of vector instruction set required
- Problem: Alignment
  - SIMD operations prefer/require aligned memory operands
  - Alignment to 16B or more
- Problem: Remainders at the end of the arrays
  - Enlarged loop overhead
  - Inefficient for small arrays

Column-organized data – SSE3

```
#include <emmintrin.h>
```

- Column-stored data in SSE types

```
vector< __m128i> data3d2;
```

  - Size of the unused space at the end

```
std::size_t reserve3;
```

- Sum the vectors

```
const __m128i * p = data3d2.data();
const __m128i * e1 = p + data3d2.size() - 1;
__m128i ss = 0;
for (; p < e1; ++ p)
    ss = _mm_add_epi32( ss, *p);
ss = _mm_hadd_epi32(ss,ss);
ss = _mm_hadd_epi32(ss,ss);
int s = mm_extract_epi32(ss, 0);
```

- Add the remaining scalars

```
const int * q = (const int*)p;
const int * e = q + 4 – reserve3,
for (; q < e; ++ q)
    s += *q;
```

- Intel intrinsics

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

  - Header files + support inside compilers
    - Available in all major C/C++ compiler suites
  - Intrinsic functions
    - A function for every instruction
      - Compiler replaces function calls with the respective instructions
    - Compiler handles register allocation, control-flow, scheduling etc.
      - Using intrinsics is far easier than assembly language programming
- Example
  - __m128i is a 128-bit type corresponding to a SSE register
  - _mm_add_epi32 is the SIMD instruction PADDQ
    - 4 additions of 32-bit integers
  - _mm_hadd_epi32 = PHADDQ
    - horizontal addition
  - _mm_extract_epi32 = PEXTRD
    - extract scalar from vector

Column-organized data – SSE3

```
#include <emmintrin.h>
```

- Column-stored data in SSE types

```
vector< __m128i> data3d2;
```

- Size of the unused space at the end

```
std::size_t reserve3;
```

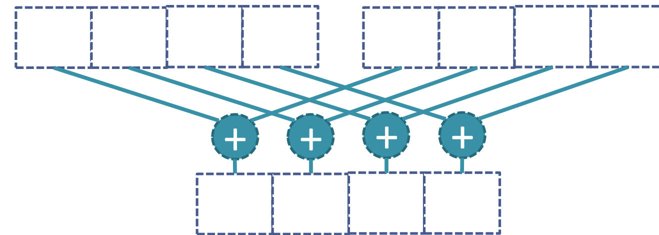
- Sum the vectors

```
const __m128i * p = data3d2.data();
const __m128i * e1 = p + data3d2.size() - 1;
__m128i ss = 0;
for (; p != e1; ++ p)
    ss = _mm_add_epi32( ss, *p);
ss = _mm_hadd_epi32(ss,ss);
ss = _mm_hadd_epi32(ss,ss);
int s = mm_extract_epi32(ss, 0);
```

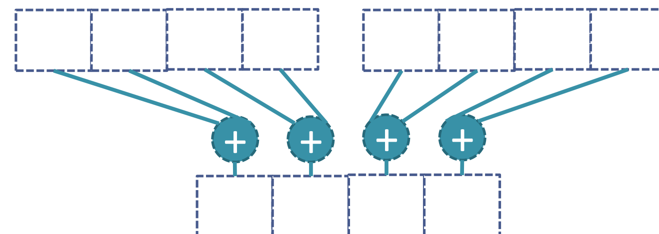
- Add the remaining scalars

```
const int * q = (const int*)p;
const int * e = q + 4 - reserve3,
for (; q != e; ++ q)
    s += *q;
```

- _mm_add_epi32 = PADDQ
  - 4 additions of 32-bit integers



- _mm_hadd_epi32 = PHADDQ
  - “horizontal addition”
  - significantly slower than PADDQ



- _mm_extract_epi32 = PEXTRD
  - extract scalar from vector