Taming the Jaguar

Andreas Fredriksson
Lead Engine Programmer
Insomniac Games

Hi, I'm Andreas!

- Today's topic: the Jaguar CPU architecture
- Microarchitecture matters!
  - Code doesn't run in a vacuum
  - Low-level knowledge improves high-level designs
  - x86 doesn't mean "stop caring"

The Jaguar: What we already know

- Well rounded
- Not many crazy pitfalls
- Out of order execution
- Sounds easy!

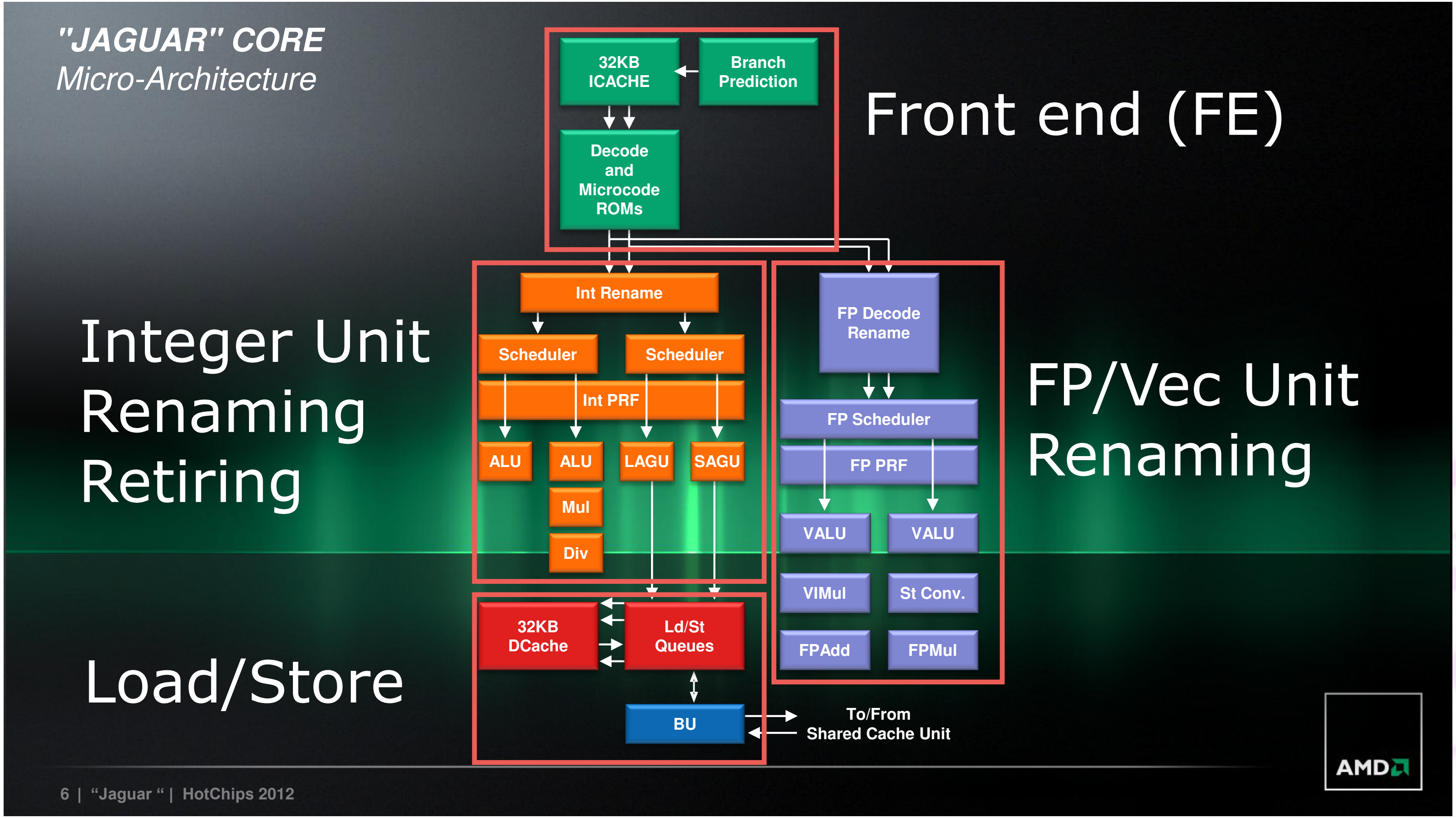
ROD STEWART
OUT OF ORDER
OUT OF
Rod Stewart
ORDER
DOLBY SYSTEM

Anonymous programmers on OOO

- "Memory access isn't a problem with OOO"
- "Branches aren't a problem with OOO"
- "SIMD isn't necessary on OOO"
- What's true? False?
  - We (optimizers) need OOO intuition, badly

Disclaimers

- This talk contains micro-optimization material
- Don't start here and expect results
  - Take a deep breath
  - Step back, consider the whole problem
  - Re-organize data before resorting to micro-opts
- Micro-optimize only where it makes sense
  - Special sauce for special circumstances



"JAGUAR" CORE

Micro-Architecture

Front end (FE)

32KB ICACHE

Branch Prediction

Decode and Microcode ROMs

Integer Unit
Renaming
Retiring

Int Rename

Scheduler

Scheduler

Int PRF

ALU

ALU

LAGU

SAGU

Mul

Div

FP/Vec Unit
Renaming

FP Decode Rename

FP Scheduler

FP PRF

VALU

VALU

VIMul

St Conv.

FPAdd

FPMul

Load/Store

32KB DCache

Ld/St Queues

BU

To/From Shared Cache Unit

AMD

ABF
ALWAYS BE FETCHING
memegenerator.net

OOO and instruction fetching

- A Jaguar core is always* fetching instructions
- It can decode up to 2 macro-ops / cycle
  - Most instructions decode to one macro-op
  - AVX instructions most notably take 2
  - Macro-ops split into micro-ops

* Assuming space in all relevant buffers

* Assuming no I1 or ITLB misses

Macro vs micro ops

- `add eax, ebx` => 1 macro-op, 1 micro-op
- `add eax, [m]` => 1 macro-op, 2 micro-ops
- `add [m], eax` => 1 macro-op, 2 (!) micro-ops

But where does it fetch from?

- Branch prediction and OOO are intertwined
- If the core doesn't know for sure, it'll guess
  - This is called *speculative execution*
- If it guesses incorrectly, things suck
- Fortunately, it's a pretty good guesser*

* Assuming programmer doesn't ignore microarchitecture

Will it execute speculatively into

- ...branches?
  - Yes
- ...direct function calls?
  - Yes
- ...indirect (virtual/pointer) function calls?
  - Yes, scarily

Getting fetching wrong

- Illusion of correctness is maintained, of course
- But the errant instructions *will* affect the cache
  - Loads and line reservations for writes
  - No way to "undo" - visible on other cores too
- Especially bad for "branchy" data structures
  - Tree-like data with pointers

Branchy data structure example

```
struct Node
{
  Node *left;
  Node *right;
  BigData bigData;
};
```

Branchy data structure example

```
void DoSomethingExpensiveToNodes(Node* n, int f)
{
  int decide = SomehowDecideChild(n, f); // high latency
  if (decide < 0) {
    DoSomethingExpensiveToNodes(n->left);
  } else if (decide > 0) {
    DoSomethingExpensiveToNodes(n->right);
  } else {
    // Do something expensive to n->bigData
  }
}
```

Misprediction central
= Bad guesses galore

Retiring

- All instructions *retire* (commit) in program order
  - That is, their effects are visible from outside the core
  - Retirement happens at a max rate of 2/cycle
- They can also be killed instead of retired
  - For example due to branch mispredictions as we saw

The Battle of North Bridge

D1 Hit

L2 Hit

Memory

3 c

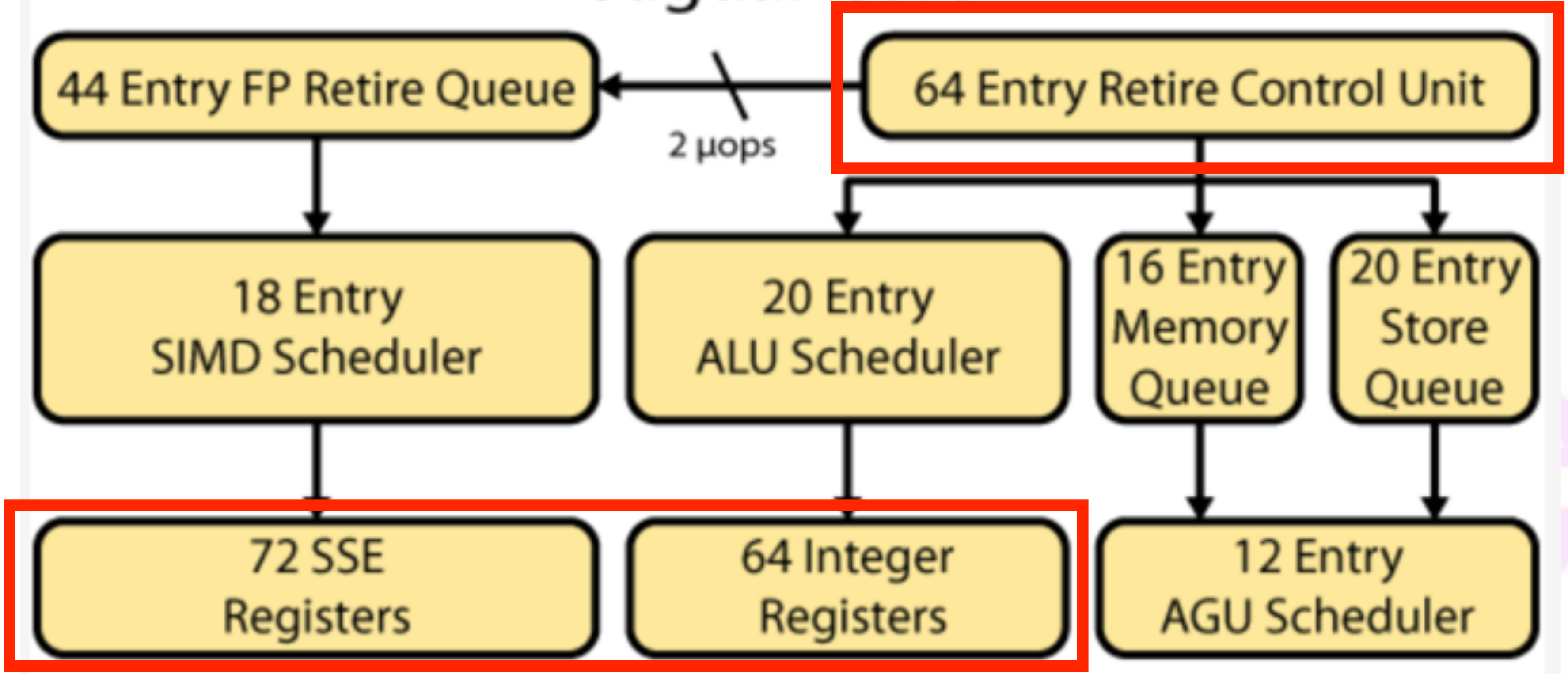
>= 25 c

at least 200 c

L2 misses, still a thing?

- A load from RAM will not retire for 200+ cycles
- So what?
  - OOO can reorder around long latencies, right?
- Sure, but: Always Be Fetching
  - The frontend issues 2 instructions / cycle..

Jaguar Core



44 Entry FP Retire Queue

2 μops

64 Entry Retire Control Unit

18 Entry SIMD Scheduler

20 Entry ALU Scheduler

16 Entry Memory Queue

20 Entry Store Queue

72 SSE Registers

64 Integer Registers

12 Entry AGU Scheduler

http://www.realworldtech.com/jaguar/4/

L2 misses & RCU tag team #fail

- L2 miss followed by low-latency instructions
  - Cache hits, simple/vector ALU etc etc
  - Remember, 2 per cycle!
- RCU fills up in < 32 cycles, and we're wedged
  - In practice less, because macro ops != instructions
- Result: ~150+ cycles wasted stalling
  - Only the L2 miss retiring will free up RCU space

Micro-optimizing L2 misses

- Re-schedule instructions
  - Move independent instructions with long latencies to right after a load that is likely to miss
  - The longer the latencies, the more it softens the blow
- Square roots, divides and reciprocals
- And other (independent) loads!

Poor load organization

```
void MyRoutine(A* ap, B* bp)
{
  float a = ap->A;    // L2 miss
  < prep work >       // RCU stall risk

  float b = bp->B;    // L2 miss!
  < prep work >       // Moar RCU stall

  < rest of routine >
}
```

Better load organization

```
void MyRoutineAlt(A* ap, B* bp)
{
  float a = ap->A;   // L2 miss
  float b = bp->B;   // L2 miss (probably "free")

  < prep work >
  < prep work >

  < rest of routine >
}
```

L2 misses on Jaguar in practice

- OOO doesn't fundamentally solve RAM latency
  - The window is way too small
  - Making it bigger has other problems
- Try to issue loads together to overlap misses
  - Hedging our bets in case more than one miss
  - Can overlap up to 8 L2 misses on single core
  - Key improvement over IOE, with some effort

Warming up: Unrolling

- Classical optimization technique
  - Idea: reduce loop management overhead
  - Very important on PS3/X360 in-order CPUs
- Heavily employed by compilers on x86 too
  - Clang *loves* unrolling, as we'll see
- Let's add some integers from an array
  - Does unrolling help?

Simple Unrolling, Scalar Base Version

```
.loop:            add      eax, [rdi]
                  lea      rdi, [rdi +  4]
                  dec      esi
                  jnz      .loop
```

Simple Unrolling, Scalar 2x unroll

```
.loop:          add     eax, [rdi +  0]
                add     eax, [rdi +  4]
                lea     rdi, [rdi +  8]
                dec     esi
                jnz     .loop
```

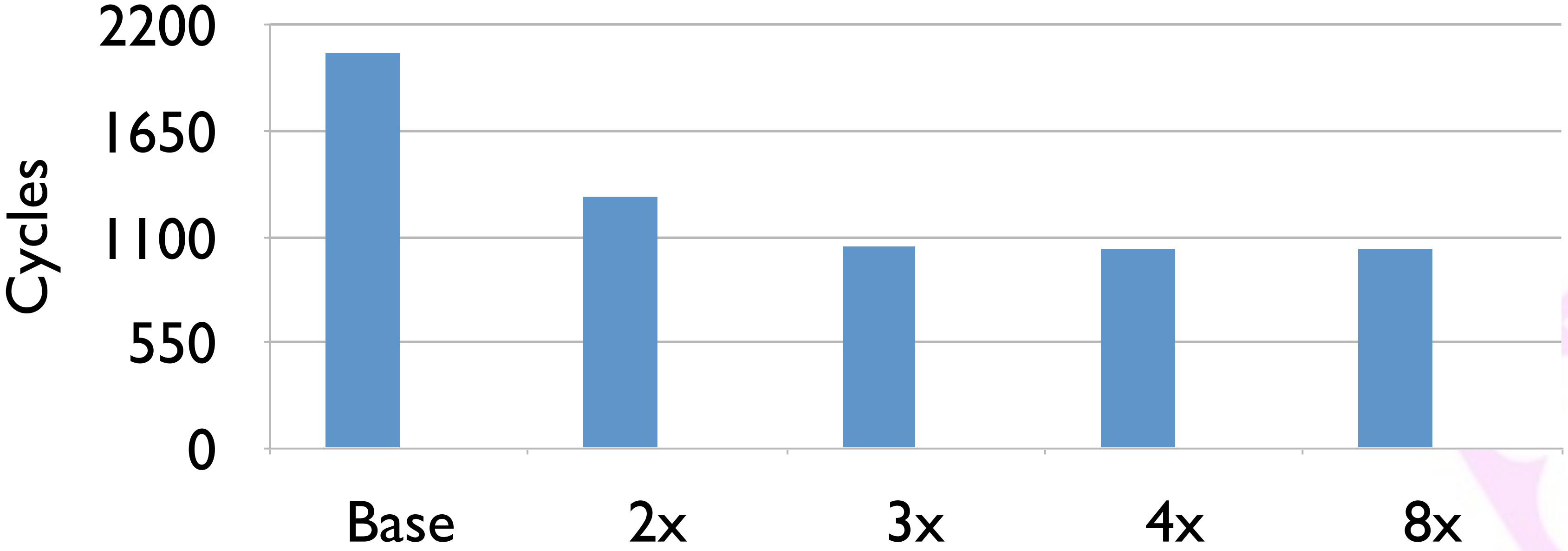
Simple Unrolling, Scalar 4x unroll

```
.loop:          add     eax, [rdi +  0]
                add     eax, [rdi +  4]
                add     eax, [rdi +  8]
                add     eax, [rdi + 12]
                lea     rdi, [rdi + 16]
                dec     esi
                jnz     .loop
```

Simple Unrolling, Scalar 8x unroll

```
.loop:          add     eax, [rdi +  0]
                add     eax, [rdi +  4]
                add     eax, [rdi +  8]
                add     eax, [rdi + 12]
                add     eax, [rdi + 16]
                add     eax, [rdi + 20]
                add     eax, [rdi + 24]
                add     eax, [rdi + 28]
                lea     rdi, [rdi + 32]
                dec     esi
                jnz     .loop
```

Scalar Unroll Performance, 1024 elems



Scalar Loop Performance Analysis

- You get to talk to cache once per cycle
  - (Once for reading, once for writing)
  - Each add needs one cache transaction to read
- 1024 x 32-bit read
  - Each will have 3 cycles D1$ latency
  - Fully overlaps to 1026 cycles of pure cache latency
  - 1026 = best possible latency this loop can ever have

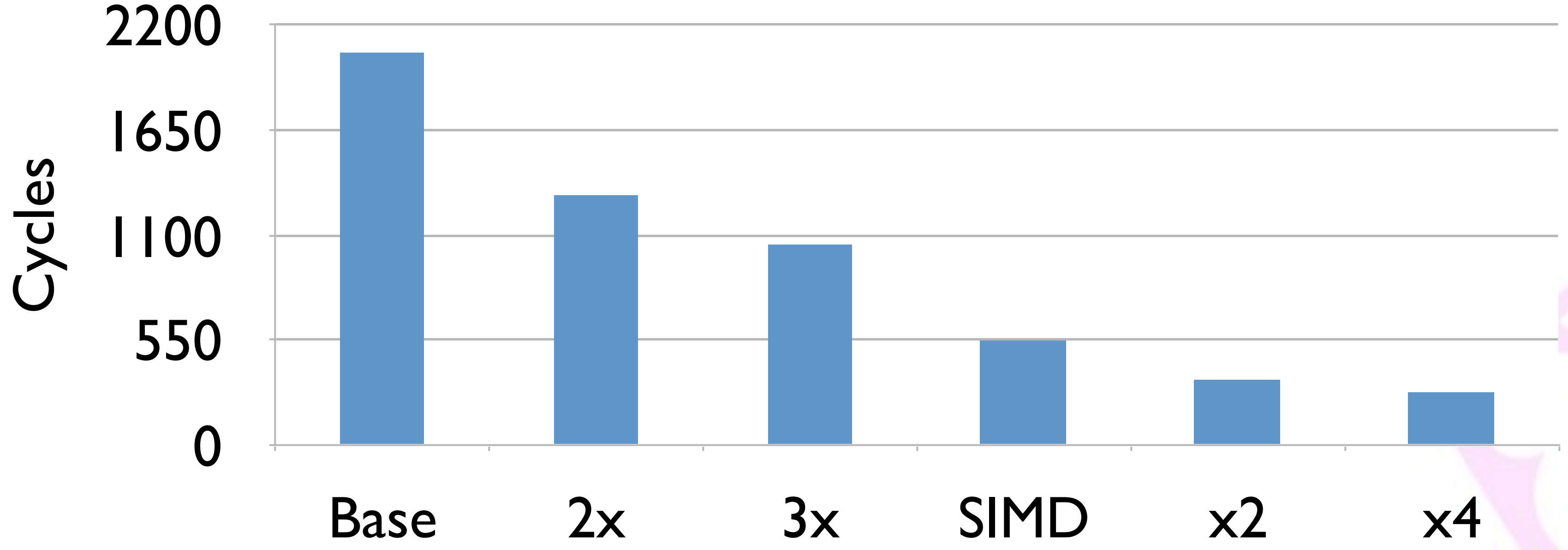
Scalar bottleneck

- The base loop bottlenecks on frontend
  - 2 instructions/cycle means 50% ALU utilization
  - We have 4 instructions in the loop, only one add
- As we unroll we shift the bottleneck to load unit
  - Getting closer and closer to the 1026 best case
- At 3x unroll we have an ideal steady state
  - Any more is a waste of .text bytes

What about SIMD?

```
.loop:          vpaddd   xmm0, xmm0, [rdi]
                ; unroll more vpaddd here..
                lea      rdi, [rdi + 0x10]
                dec      rsi
                jnz      .loop
```

Results



SIMD analysis

- Uses full 128 bit/cycle D1$ bandwidth
  - 4x improvement over scalar code
  - Not primarily because the adds are parallel!
- Unrolling helps the same way as for scalar case
  - Shifts emphasis from FE to LS

Unrolling Takeaways

- Unrolling can help very simple loops
  - By shifting emphasis from frontend to other ports
  - Frontend is relatively weak at 2 insns/cycle
- The cache can deliver 128 bits per cycle
  - Scalar code uses only a fraction of that bandwidth
  - SIMD code has a natural edge scalar can't touch

What if we do it in C?

```
uint32_t UnrollTestC(const uint32_t* nums, size_t count)
{
  uint32_t sum = 0;
  while (count--)
  {
    sum += *nums++;
  }
  return sum;
}
```

Meet clang, unroller extraordinaire

```
UnrollTestC(unsigned int const*, unsigned long):
  xor           eax,eax
  test          rsi,rsi
  je            00000000000000ABh
  mov           rax,rsi
  and           rax,0FFFFFFFFFFFFFFF0h
  mov           r8,rsi
  vpxor         xmm0,xmm0,xmm0
  mov           rcx,rsi
  and           r8,0FFFFFFFFFFFFFFF0h
  je            000000000000005Fh
  sub           rcx,rax
  lea           rdx,[rdi+r8*4]
  add           rdi,30h
  vpxor         xmm0,xmm0,xmm0
  mov           rax,r8
  vpxor         xmm1,xmm1,xmm1
  vpxor         xmm2,xmm2,xmm2
  vpxor         xmm3,xmm3,xmm3
  vpaddd        xmm0,xmm0,xmmword ptr [rdi-30h]
  vpaddd        xmm1,xmm1,xmmword ptr [rdi-20h]
  vpaddd        xmm2,xmm2,xmmword ptr [rdi-10h]
  vpaddd        xmm3,xmm3,xmmword ptr [rdi]
  add           rdi,40h
  add           rax,0FFFFFFFFFFFFFFF0h
```

```
jne           0000000000000040h
jmp           0000000000000071h
mov           rdx,rdi
xor           r8d,r8d
vpxor         xmm1,xmm1,xmm1
vpxor         xmm2,xmm2,xmm2
vpxor         xmm3,xmm3,xmm3
vpaddd        xmm0,xmm1,xmm0
vpaddd        xmm0,xmm2,xmm0
vpaddd        xmm0,xmm3,xmm0
vmovhlps      xmm1,xmm0,xmm0
vpaddd        xmm0,xmm0,xmm1
vphaddd       xmm0,xmm0,xmm0
vmovd         eax,xmm0
cmp           r8,rsi
je            00000000000000ABh
nop           word ptr cs:[rax+rax+0]
add           eax,dword ptr [rdx]
add           rdx,4
dec           rcx
jne           00000000000000A0h
ret
```

Clang output analysis

- Clang unrolls to 4x SIMD
  - Achieves theoretical best case in this case
- So compilers are great at this stuff, right?
  - Sometimes.. One sample is not enough.

Unrolling in General

- Typically doesn't help more complicated loops
  - Any added latency anywhere shifts the balance
- OOO is a hardware loop unroller!
  - The hardware will run head into "future" iterations of the loop, issuing them speculatively
  - Only if everything is in cache and all ops are simple will FE dominate the loop performance

Jaguar Unrolling Guidelines

- Turn to SIMD before you unroll scalar code
  - Use SSE (with VEX encoding), but not AVX
- Unroll to gather data to run at full SIMD width
  - E.g. Unroll 32-bit fetching gather loop 4 times
  - Then process in 128-bit SIMD registers

Prefetching

- Required on PPC console era chips
  - Sprinkle in loops and reap benefits!
- x86 also offers prefetch instructions
  - PREFETCHT0/1/2 - Vanilla prefetches
  - PREFETCHNTA - Non-temporal prefetch
  - Use _mm_prefetch(addr, _MM_HINT_xxx)
- So, should we use prefetches on Jaguar?

Linked List Example

```
struct GameObject {
  int          m_Health;
  GameObject *m_Next;
  // other members ...
};
```

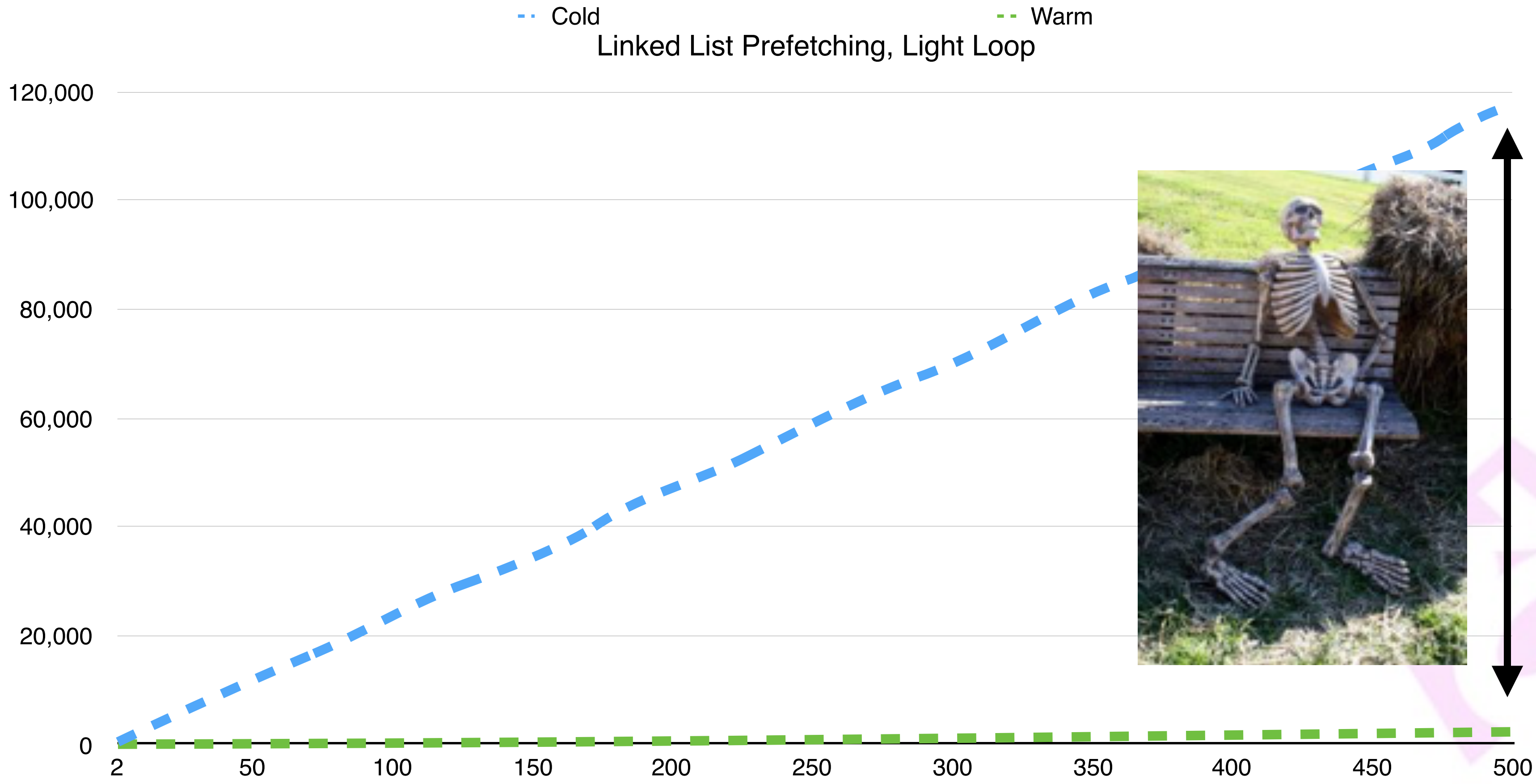
```
int CountDeadObjects(GameObject* head)
{
  int dead_count = 0;
  while (head) {
    GameObject *next = head->m_Next;
    _mm_prefetch(next, _MM_HINT_T0);
    dead_count += head->m_Health == 0 ? 1 : 0;
    head = next;
  }
  return dead_count;
}
```

Linked List Asm, clang

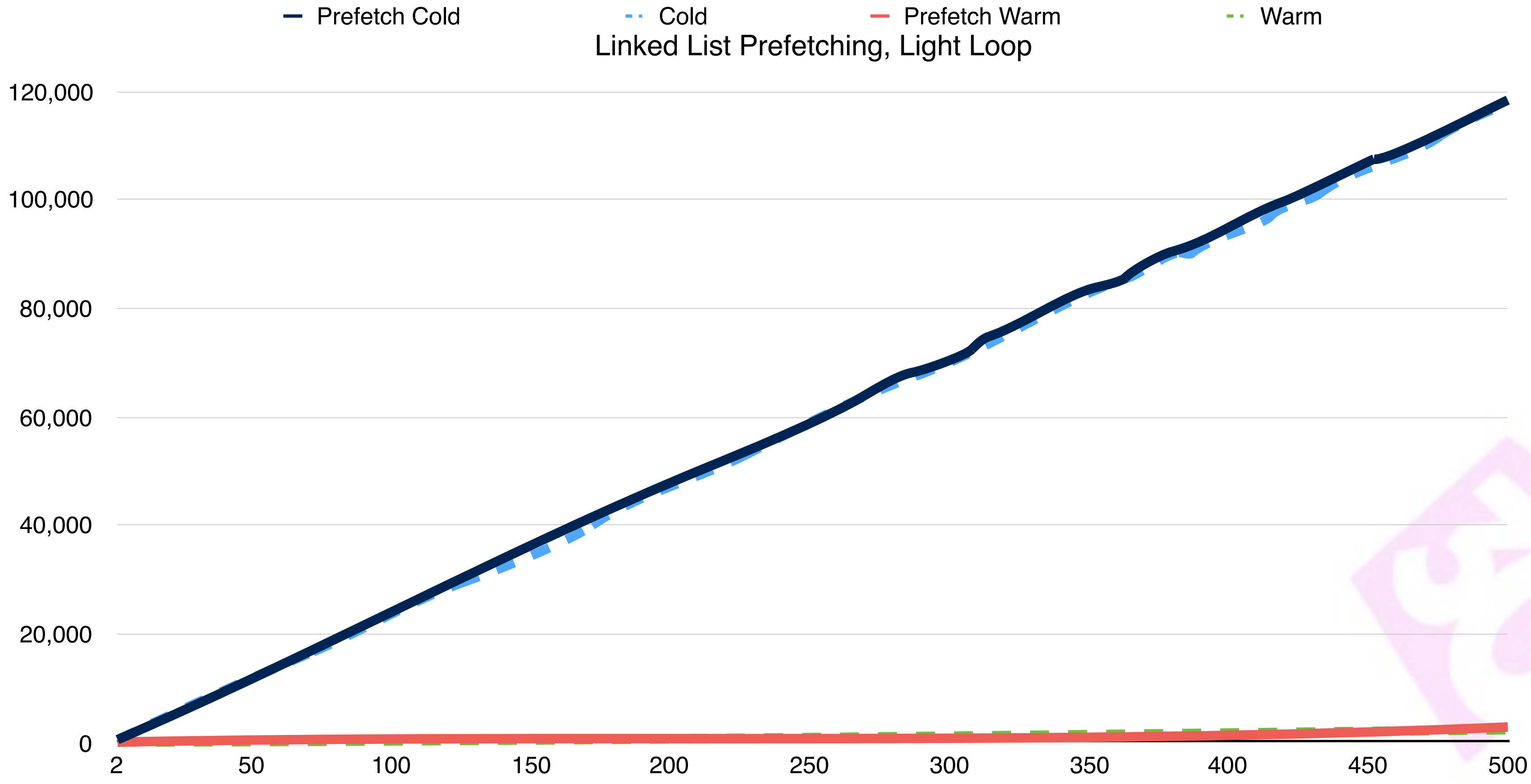
```
CountDeadObjects:
        push        rbp
        mov         rbp, rsp
        xor         eax, eax
        test        rdi, rdi
        .align      4, 0x90
.loop:  mov         rcx, qword ptr [rdi + 8]
        prefetcht0  byte ptr [rcx]
        cmp         dword ptr [rdi], 1
        adc         eax, 0
        test        rcx, rcx
        mov         rdi, rcx
        jne         .loop
.done   pop         rbp
        ret
```

Neat!

```
dead_count += head->m_Health == 0 ? 1 : 0;
```



Linked List Prefetching, Light Loop


Prefetch Cold
Cold
Prefetch Warm
Warm
Linked List Prefetching, Light Loop
120,000
100,000
80,000
60,000
40,000
20,000
0
2
50
100
150
200
250
300
350
400
450
500

"We chased pointers, and I *helped*!"

Linked List Results

- This type of prefetching is useless
  - No time for prefetch to actually help
- Linked lists turn OOO into in-order
  - 100% bound by memory latency
  - Next pointer to fetch is hidden in memory
  - No way for CPU to run ahead and get data early
  - Also renders hardware array prefetchers useless

Basic Array Example

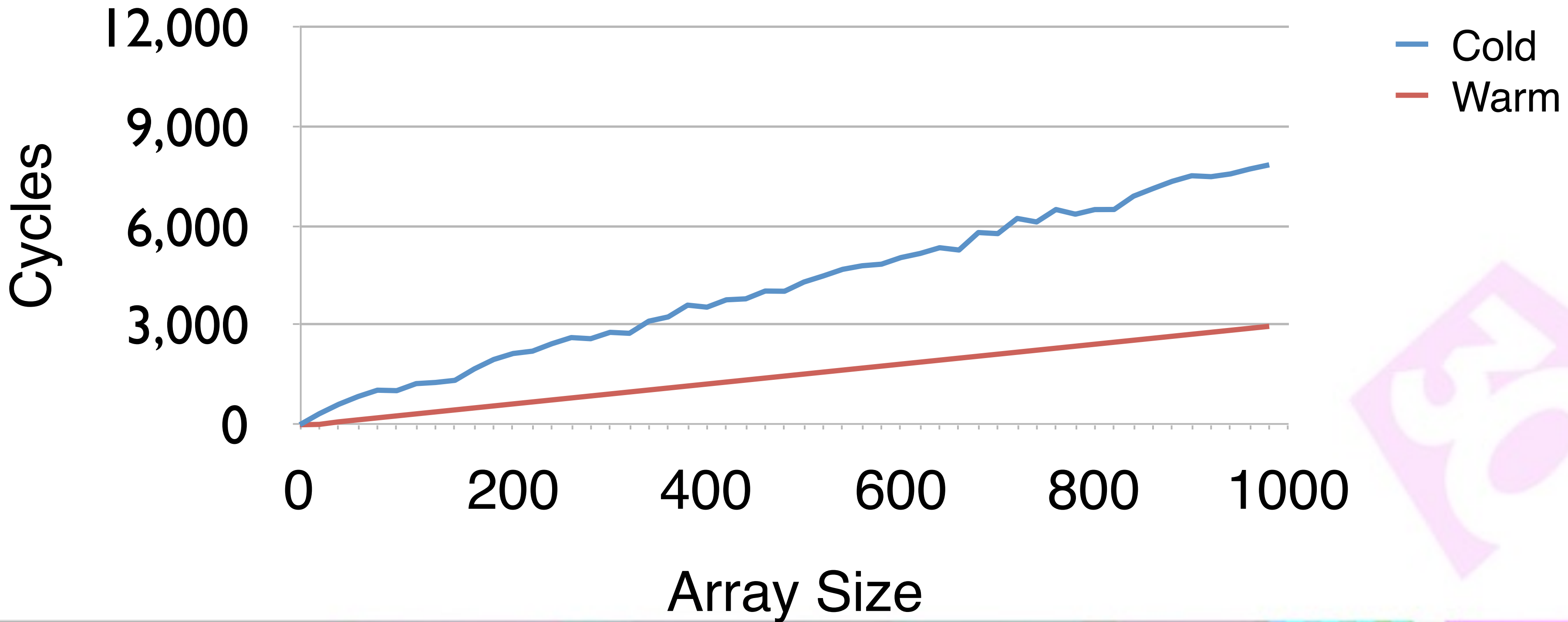
- Consuming data linearly from RAM
- No dependent pointers involved
- Does prefetching help?

Basic Array Example

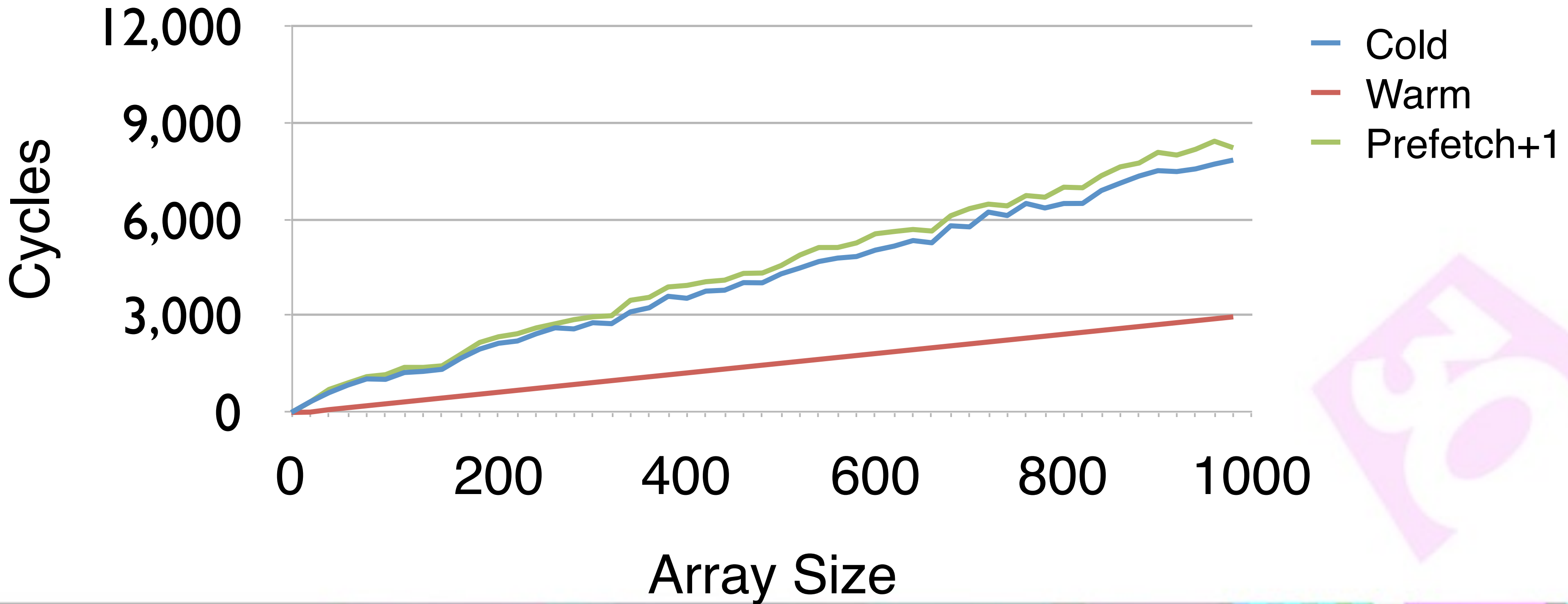
```
struct Node {
  // data (24 bytes)
};
```

```
Node* base = ...;
for (size_t i = 0; i < count; ++i) {
  // Compute based on base[i];
}
```

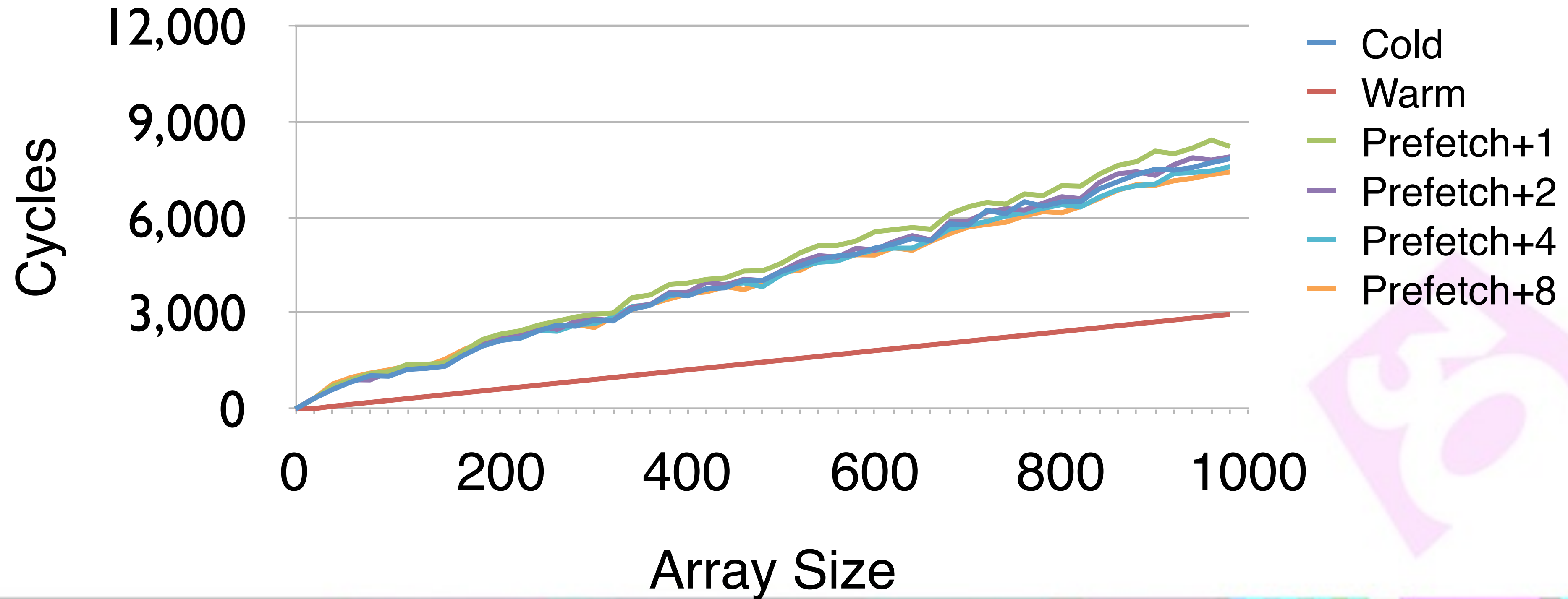
Basic Array, Light Workload



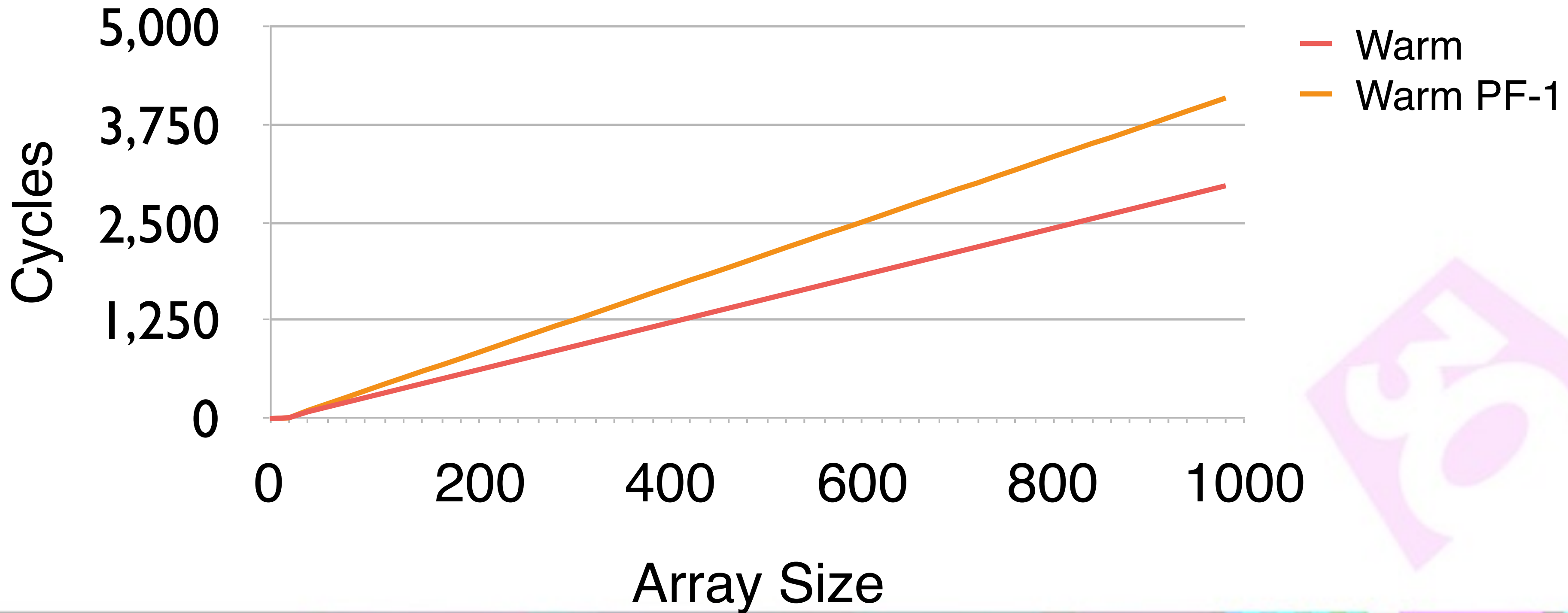
Basic Array, Light Workload



Basic Array, Light Workload



Basic Array, Light Workload, Warm



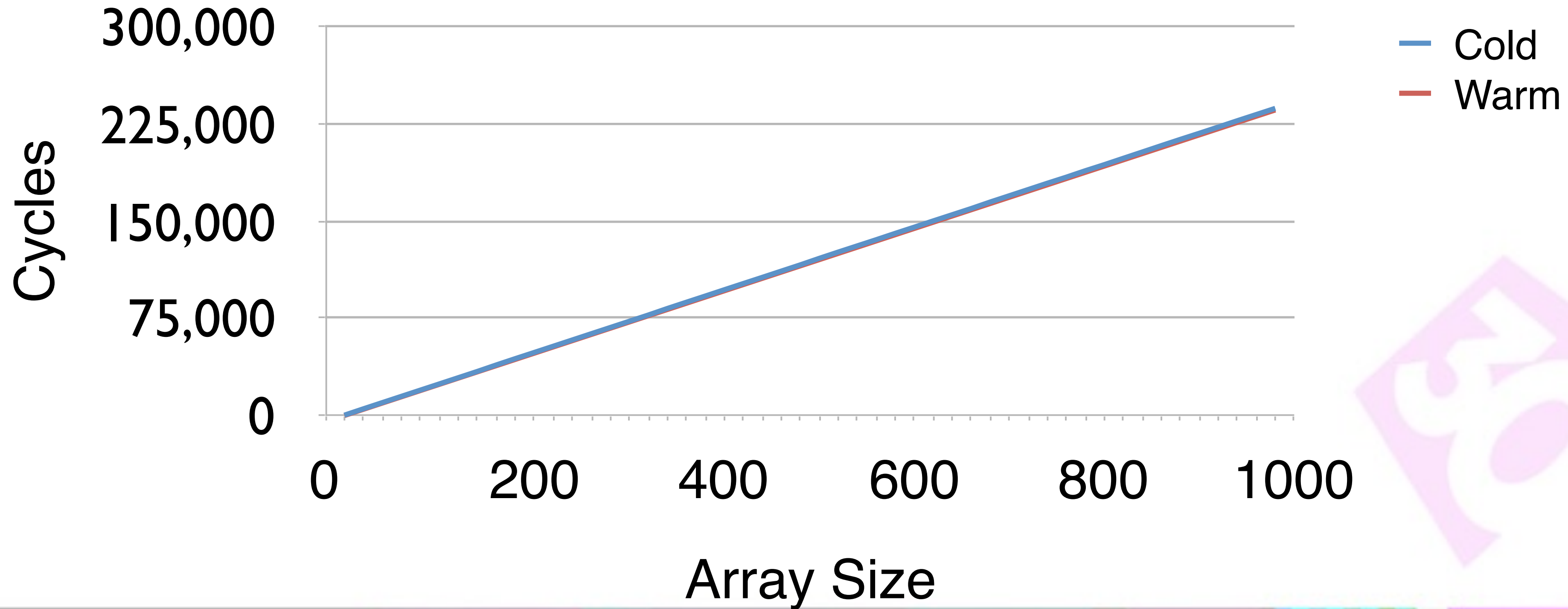
Light Array Workload Analysis

- Prefetching in the cold case doesn't help
  - OOO does it better, more cheaply than we can
  - Short loops will be running 4+ unrolls ahead
- Prefetching in the warm case actually hurts
  - Adds useless ops for the FE to decode
  - Adds load unit traffic that limit OOO "unrolling"
- Hardware figures this out itself without "help"

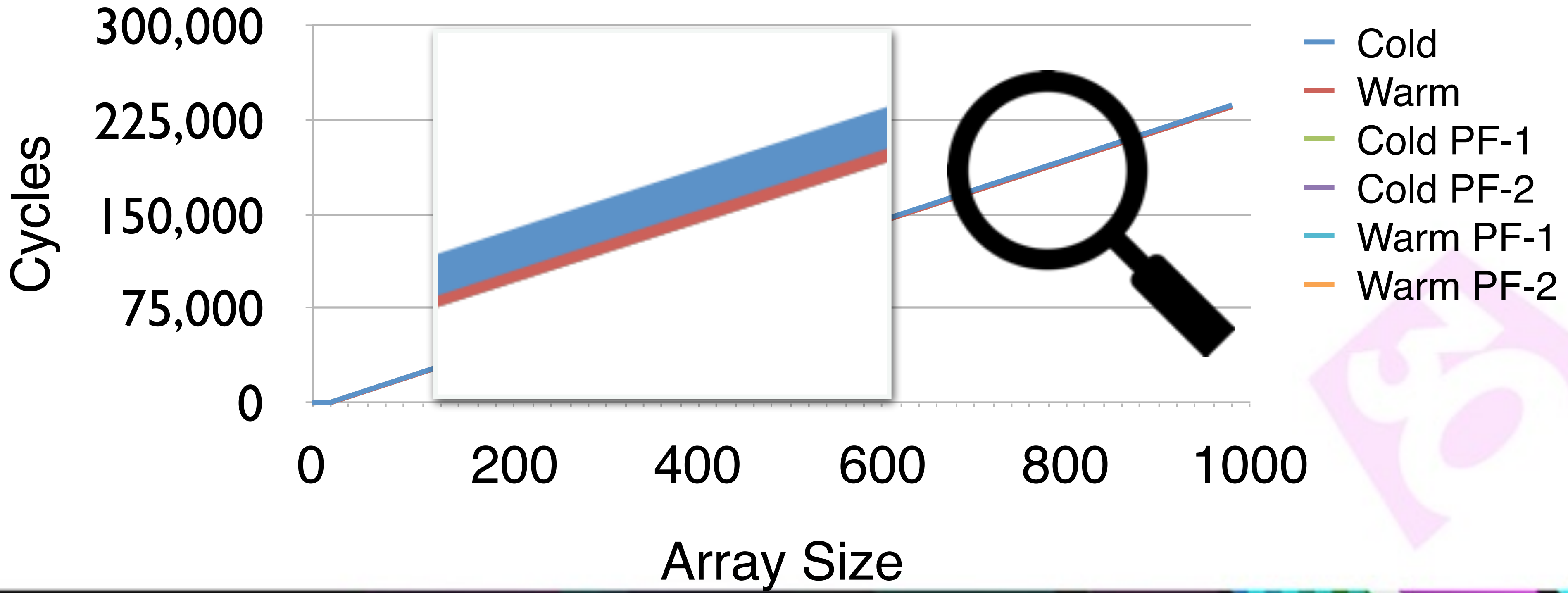
Heavy Array Workload

- Let's do some more number crunching
- Enough that we're compute bound in theory
- Does prefetching help in this case?

Basic Array, Heavy Workload



Basic Array, Heavy Workload



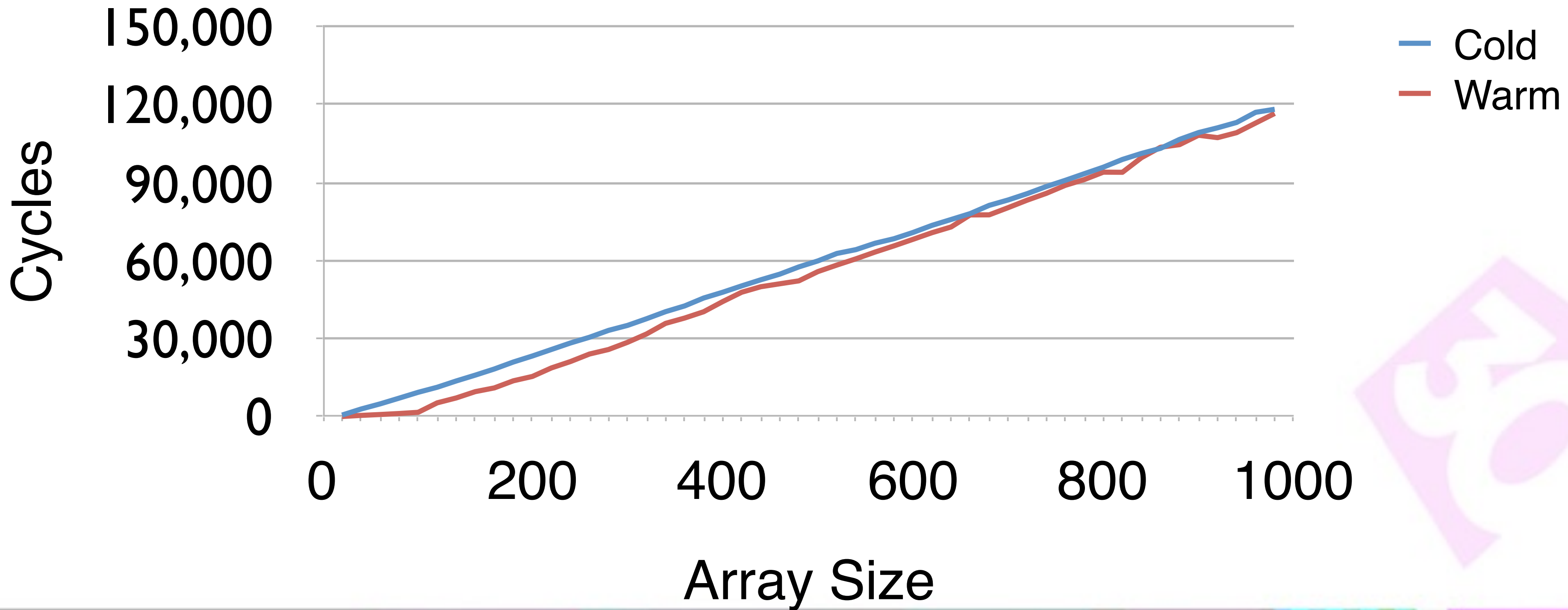
Basic Array Prefetching

- Don't waste time on this
- Jaguar loves arrays
  - The CPU has dedicated prefetchers (Both D1$ + L2!)
  - OOO will execute ahead and issue loads too
- It's very hard to improve on basic array performance using prefetches
  - But you can definitely hurt it!

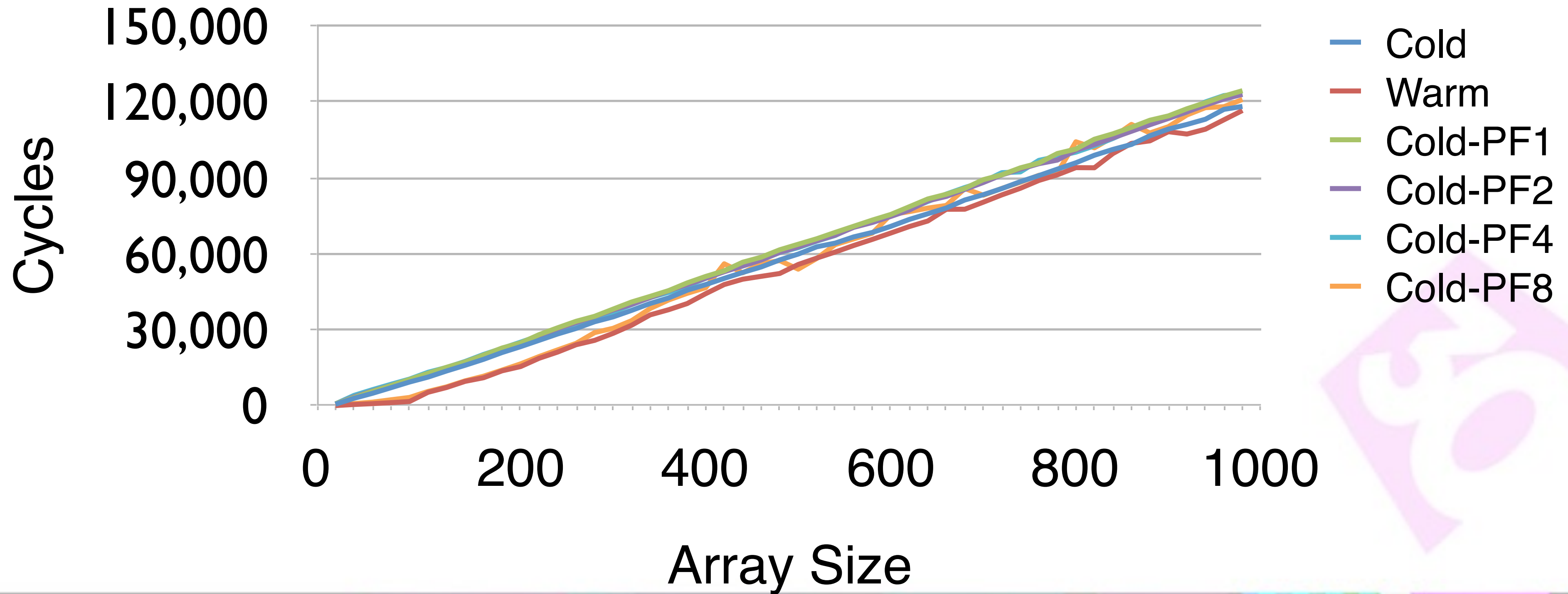
Mixed Workload Example

- Walking array elements with two pointers
  - `struct Node { Secondary *p1, *p2; }`
- Compute based on data fetched from both
- Does prefetching help?
  - Light workload - a couple of ALU instructions
  - Heavy workload - 100s of cycles of ALU latency

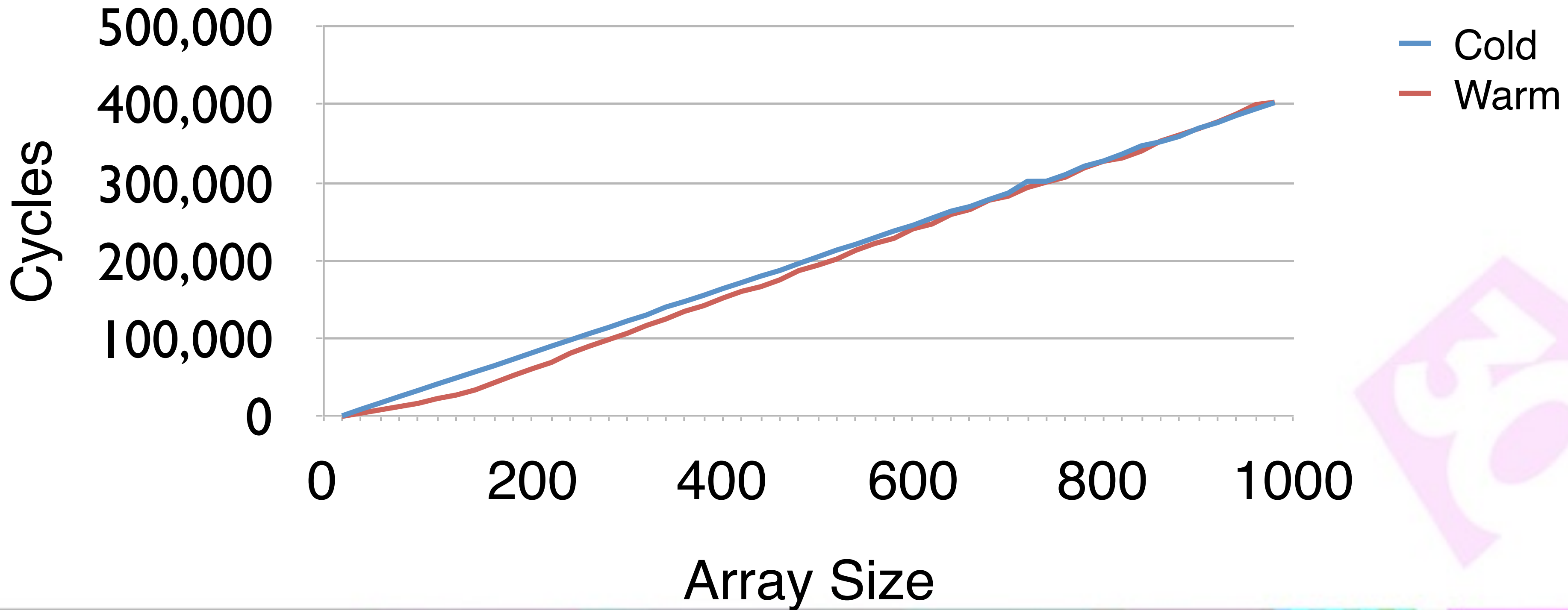
Light workload, Pointer Chasing



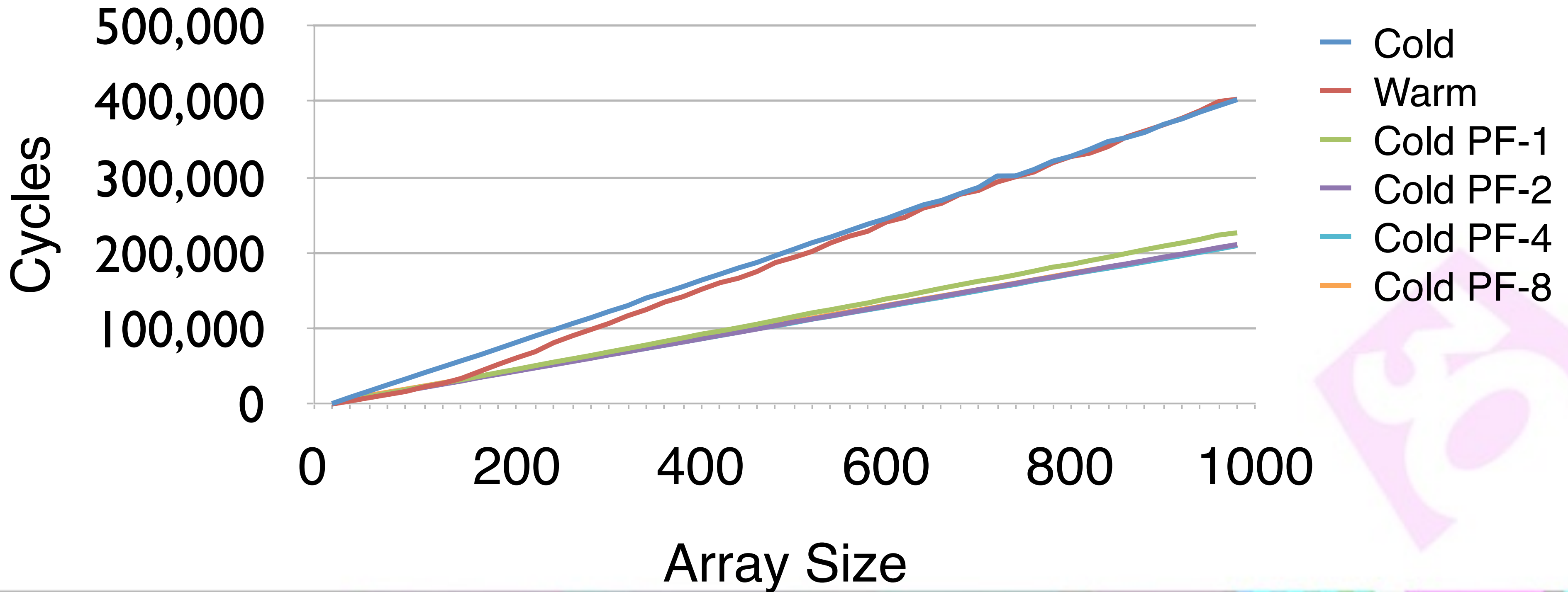
Light workload, Pointer Chasing



Heavy workload, Pointer Chasing



Heavy workload, Pointer Chasing



Mixed Workload Results

- Prefetch can win when there is a lot of ALU
  - Preventing OOO scheduler from fetching ahead
  - Prefetching helps as in the "good old days"
- In practice this isn't a super common setup
  - More bang for the buck to minimize pointers

Jaguar Prefetching Guidelines

- Never prefetch basic arrays
  - Actually hurts warm cache case with short loops
- Prefetch only heavy array/pointer workloads
  - Need work to overlap the latency of the prefetch
- Non-intuitive to reason about
  - Best to add close to gold when things are stable
  - Always measure, never assume!

Practical: Linear Searching

- How to best search an unsorted array?
- Jaguar micro-optimization exercise
  - Assume everything is in D1 cache
  - Assume searching unsorted 32-bit numbers
  - Assume we just need found/not found result
  - Assume we expect to find something 99% of the time
    - Need to scan about half the array if early outing

The Naive Approach

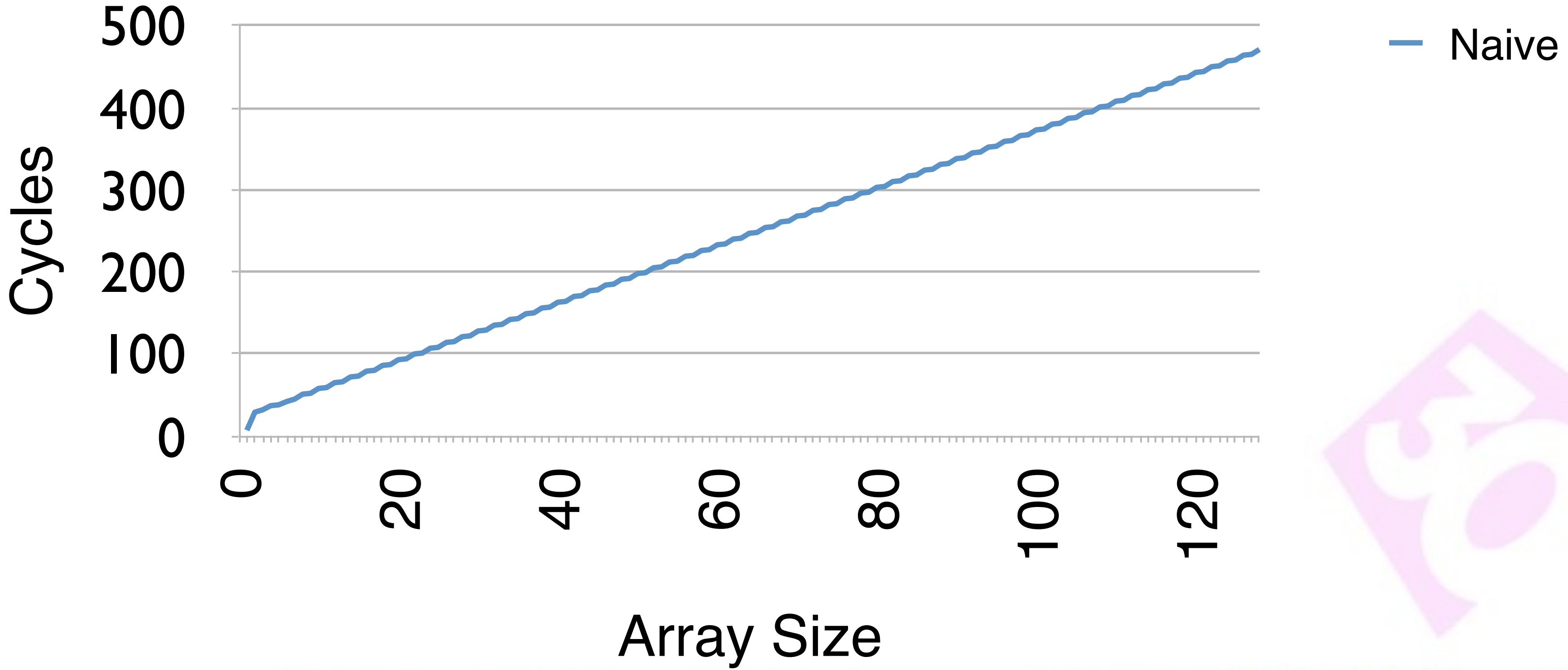
```
bool ArraySearchNaiveC(uint32_t needle, const uint32_t haystack[], int count)
{
  for (int i = 0; i < count; ++i)
  {
    if (needle == haystack[i])
    {
      return true;
    }
}
  return false;
}
```

clang output

```
ArraySearchNaiveC:
          xor          ecx,ecx
          mov          eax,0
          test         edx,edx
          jle          .fail
          nop          dword ptr [rax+rax+0]
.loop:    mov          al,1
          cmp          dword ptr [rsi+rcx*4],edi
          je           .success
          inc          rcx
          cmp          ecx,edx
          jl           .loop
.fail:    xor          eax,eax
.success: ret
```

Wat

Naive performance

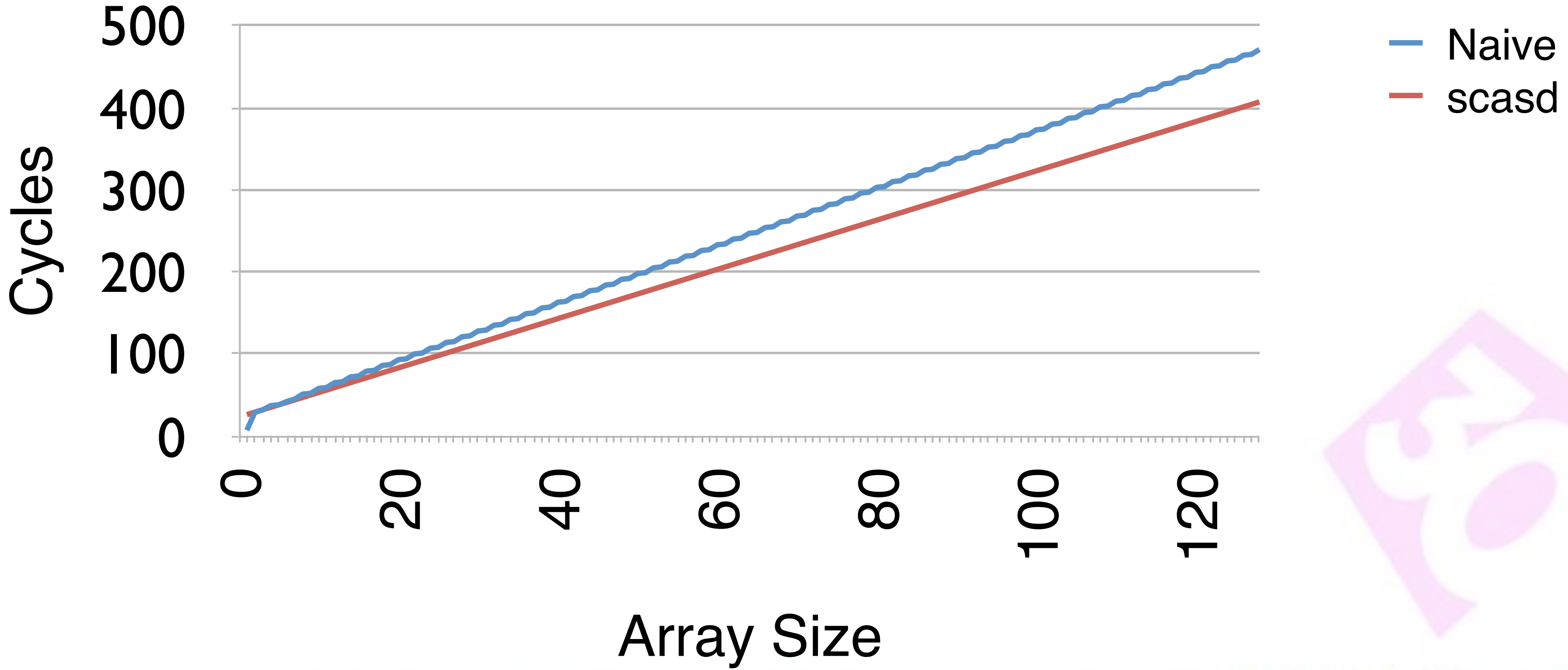


The naive approach, 1980s style

```
repne scasd
```

Isn't x86 something else?

Naive performance vs REPNE SCASD



Wat loses

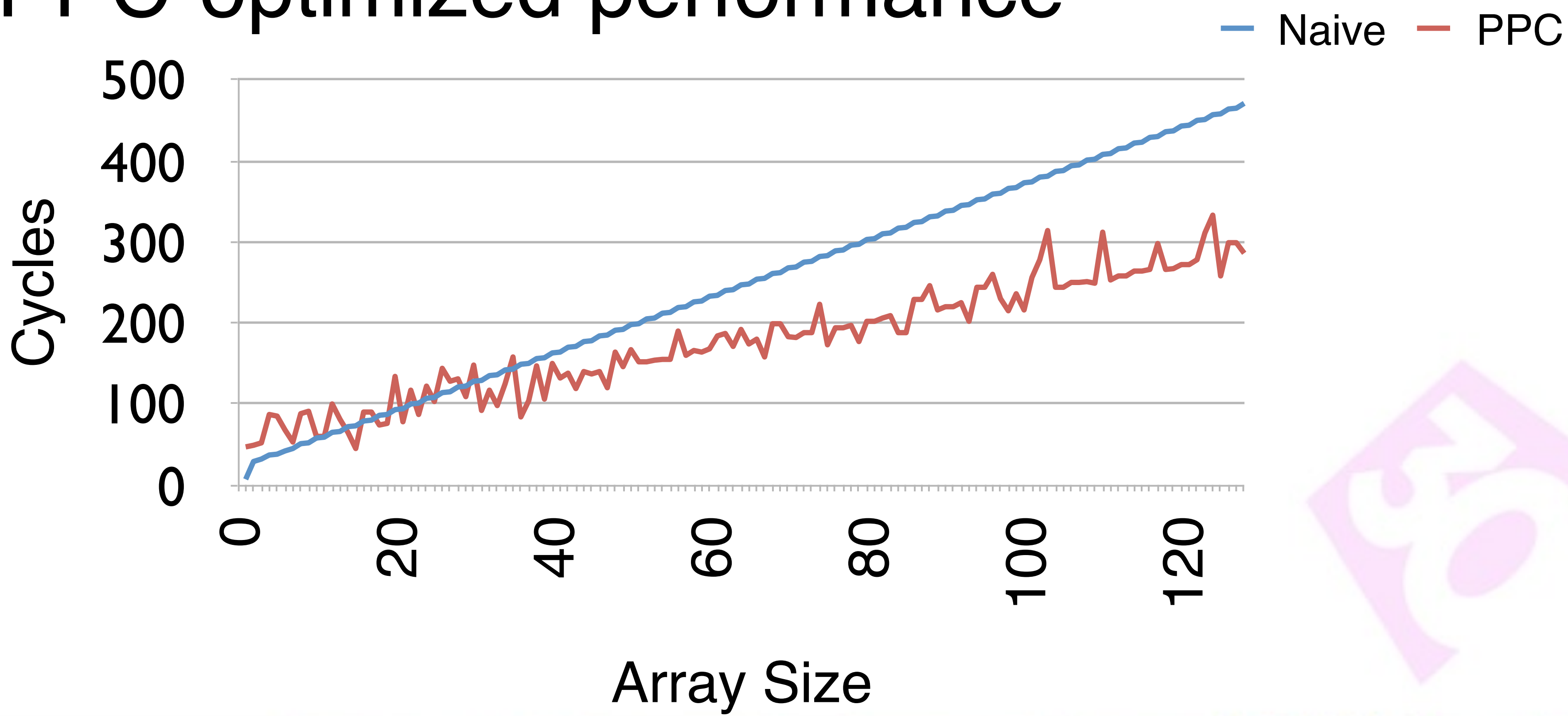
- That redundant mov cost clang the win
  - In the "naive" category
- Loops this tight are extremely heavy on FE
  - Remember: max 2 decodes / cycle
  - Additional instructions cause significant perf drops
- String instructions can easily be beat though

PPC-optimized approach

- We were using a remnant from our PS3 engine
  - Unroll cluster of 4 compares
  - Merge and branch once per cluster
  - Way better on PPU
- How does it perform on Jaguar?

```
FV32_Loop:
  v0     = list[0];
  v1     = list[1];
  v2     = list[2];
  v3     = list[3];
  list += 4;
  v0     = v0 ^ value;
  v1     = v1 ^ value;
  v2     = v2 ^ value;
  v3     = v3 ^ value;
  v0     = v0 | (-v0);
  v1     = v1 | (-v1);
  v2     = v2 | (-v2);
  v3     = v3 | (-v3);
  v0     = v0 & v1;
  v2     = v2 & v3;
  if ((v0 & v2) == 0) goto FV32_Found;
  if (list !=loop_term) goto FV32_Loop;
```

PPC-optimized performance



PPC-optimized aftermath

- Old in-order optimizations not always clear wins
- Watch out for trading ALU for less branching
  - Can remove OOO "unrolling" in tight loops
  - Latency chains become longer in general
- The 4 cluster branching wins after ~32 elements
- Should be able to do better..

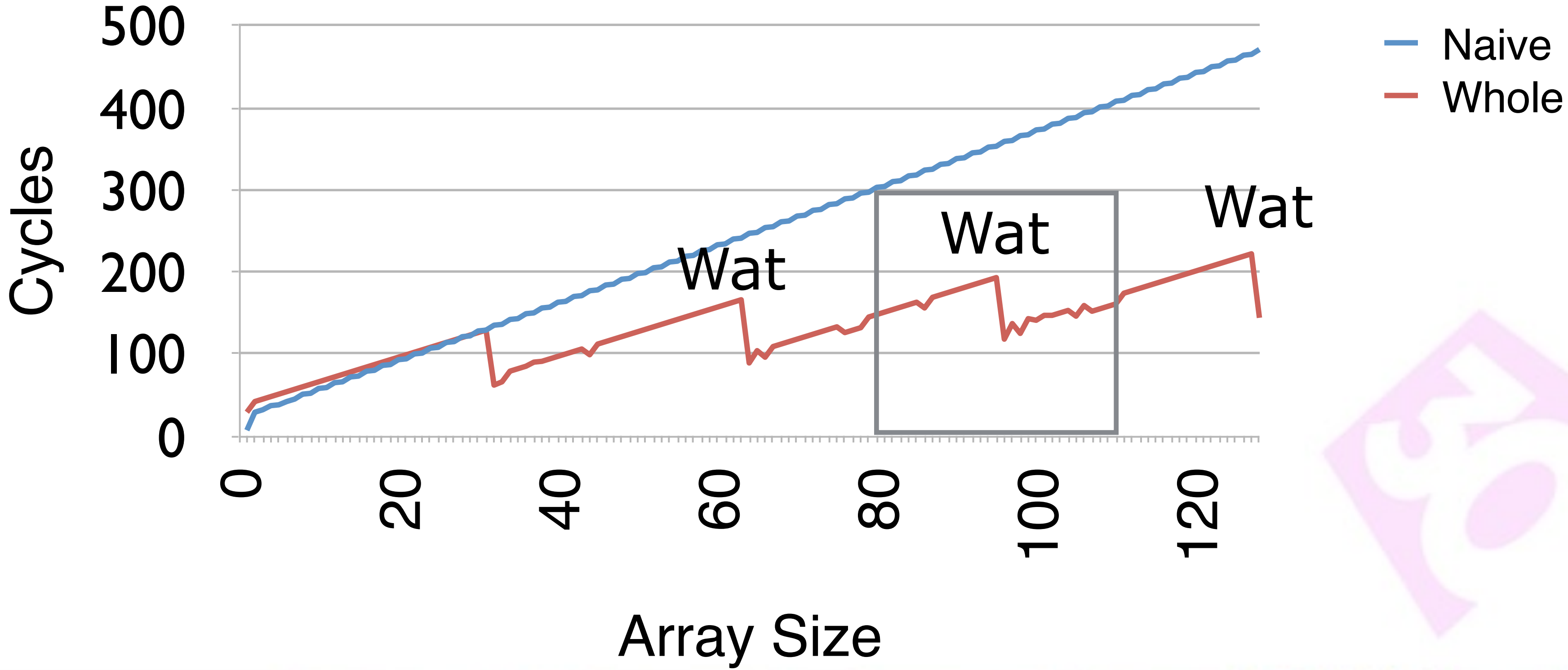
Let's search the whole array!

- Idea: Make it more predictable
    - Always the same work for a certain array size
- Should be simpler to reason about?

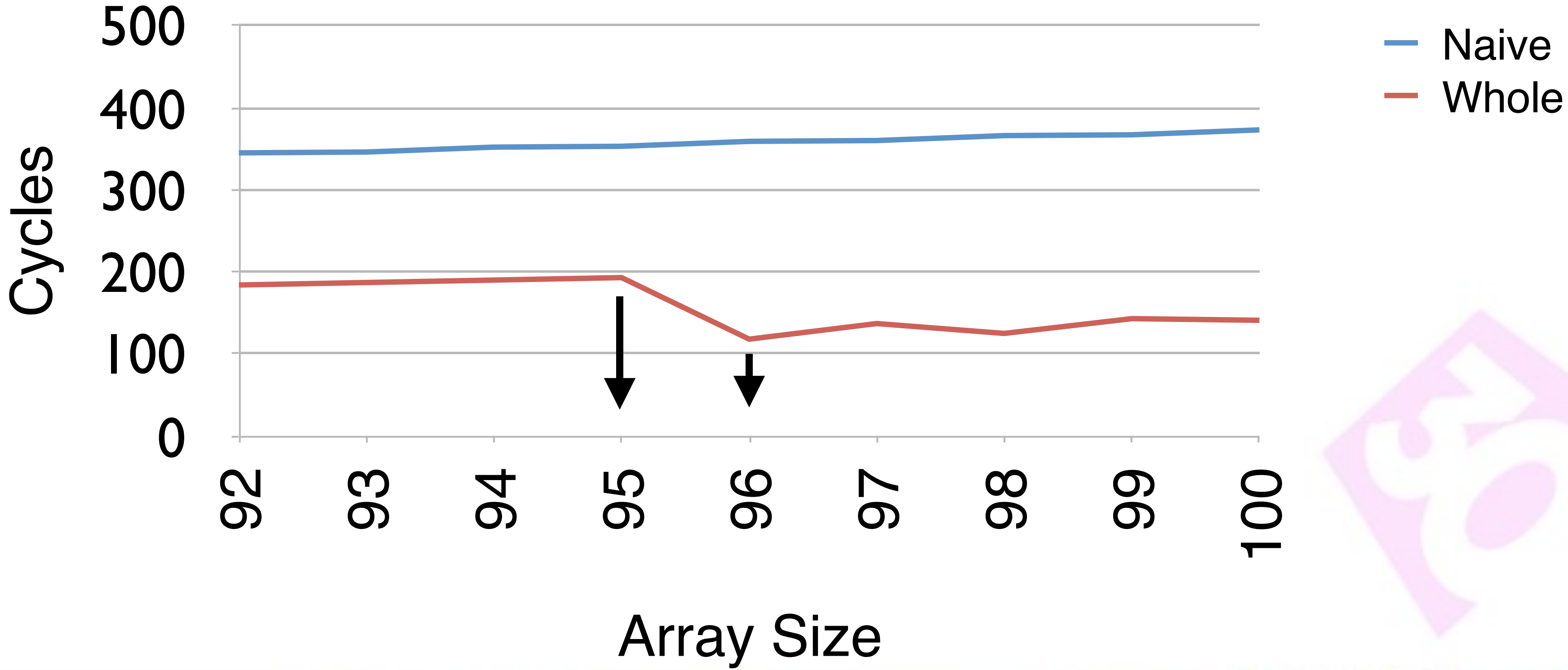
Whole Array Search

```
bool ArraySearchWholeArray(uint32_t needle, const uint32_t haystack[], int count)
{
  bool found = false;
  for (int i = 0; i < count; ++i)
  {
    found |= needle == haystack[i];
  }
  return found;
}
```

Whole Array Search Performance



Whole Array Search Performance



clang: "Let me unroll that for you..."

```
ArraySearchWholeArray(unsigned int, unsigned int
const*, int):
  test          edx,edx
  jle           0000000000000171h
  lea           eax,[rdx-1]
  lea           r8,[rax+1]
  xor           ecx,ecx
  mov           r9,1FFFFFE0h
  vpxor         xmm0,xmm0,xmm0
  vxorps        xmm1,xmm1,xmm1
  vxorps        xmm2,xmm2,xmm2
  vxorps        xmm3,xmm3,xmm3
  and           r9,r8
  je            000000000000011Dh
  vmovd         xmm0,edi
  vpshufd       xmm0,xmm0,0
  vinsertf128   ymm8,ymm0,xmm0,1
  lea           rcx,[rsi+60h]
  inc           rax
  and           rax,0FFFFFFFFFFFFFFE0h
  vpxor         xmm0,xmm0,xmm0
  vextractf128 xmm10,ymm8,1
  vmovdqa       xmm11,xmmword ptr [...]
  vxorps        xmm1,xmm1,xmm1
  vxorps        xmm2,xmm2,xmm2
  vxorps        xmm3,xmm3,xmm3
  nop           dword ptr [rax+0]
  vpcmpeqd      xmm7,xmm10,xmmword ptr [rcx-50h]
  vpshufb       xmm7,xmm7,xmm11
  vpcmpeqd      xmm4,xmm8,xmmword ptr [rcx-60h]
  vpshufb       xmm4,xmm4,xmm11
  vmovlhps      xmm9,xmm4,xmm7
  vpcmpeqd      xmm7,xmm10,xmmword ptr [rcx-30h]
  vpshufb       xmm7,xmm7,xmm11
```

```
vpcmpeqd      xmm4,xmm8,xmmword ptr [rcx-40h]
vpshufb       xmm4,xmm4,xmm11
vmovlhps      xmm4,xmm4,xmm7
vpcmpeqd      xmm7,xmm10,xmmword ptr [rcx-10h]
vpshufb       xmm7,xmm7,xmm11
vpcmpeqd      xmm5,xmm8,xmmword ptr [rcx-20h]
vpshufb       xmm5,xmm5,xmm11
vmovlhps      xmm5,xmm5,xmm7
vpcmpeqd      xmm7,xmm10,xmmword ptr [rcx+10h]
vpshufb       xmm7,xmm7,xmm11
vpcmpeqd      xmm6,xmm8,xmmword ptr [rcx]
vpshufb       xmm6,xmm6,xmm11
vmovlhps      xmm6,xmm6,xmm7
vpor          xmm0,xmm0,xmm9
vorps         xmm1,xmm1,xmm4
vorps         xmm2,xmm2,xmm5
vorps         xmm3,xmm3,xmm6
sub           rcx,0FFFFFFFFFFFFFF80h
add           rax,0FFFFFFFFFFFFFFE0h
jne           00000000000000A0h
mov           rcx,r9
vorps         xmm0,xmm1,xmm0
vorps         xmm0,xmm2,xmm0
vorps         xmm0,xmm3,xmm0
vmovhlps      xmm1,xmm0,xmm0
vorps         xmm0,xmm0,xmm1
vpshufd       xmm1,xmm0,1
vpor          xmm0,xmm0,xmm1
vpalignr      xmm1,xmm0,xmm0,2
vpor          xmm0,xmm0,xmm1
vpextrb       rax,xmm0,0
cmp           r8,rcx
je            0000000000000173h
lea           rsi,[rsi+rcx*4]
```

```
sub           edx,ecx
nop           word ptr cs:[rax+rax+0]
cmp           dword ptr [rsi],edi
sete          cl
or            al,cl
add           rsi,4
dec           edx
jne           0000000000000160h
jmp           0000000000000173h
xor           eax,eax
and           al,1
ret
```

So, compilers are great at this?

- Not always...
- Highly variable performance in this version
  - Long scalar fixup loop at the end
- We can easily do better ourselves

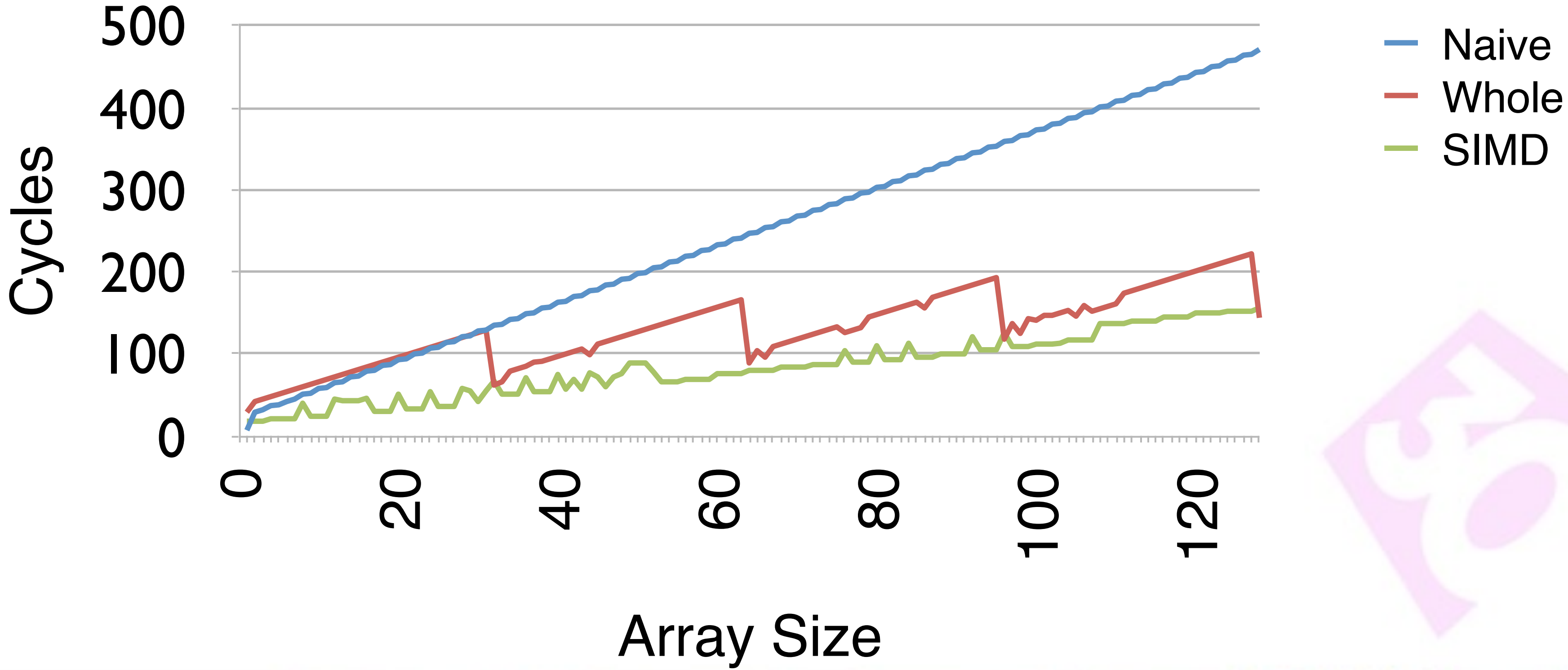
Let's try that again

```
bool ArraySearchSimd(uint32_t needle, const uint32_t haystack[], int count)
{
  __m128i n = _mm_set1_epi32(needle);
  __m128i mask = _mm_setzero_si128();
  int aligned_count = count & ~3;
  int straggler_count = count & 3;
  int i;

  for (i = 0; i < aligned_count; i += 4) {
    __m128i val = _mm_loadu_si128((const __m128i*)(haystack + i));
    __m128i cmpmask = _mm_cmpeq_epi32(val, n);
    mask = _mm_or_si128(mask, cmpmask);
  }

  // Stragglers
  uint32_t straggler_mask_int = straggler_count ? ~0u << (4 - straggler_count) : 0;
  __m128i sm0 = _mm_cvtsi32_si128(straggler_mask_int);
  __m128i sm1 = _mm_unpacklo_epi8(sm0, sm0);
  __m128i sm2 = _mm_unpacklo_epi16(sm1, sm1);
  __m128i val = _mm_loadu_si128((const __m128i*)(haystack + count - 4));
  __m128i cmpmask = _mm_and_si128(_mm_cmpeq_epi32(val, n), sm2);
  mask = _mm_or_si128(mask, cmpmask);
  return _mm_movemask_ps(_mm_castsi128_ps(mask));
}
```

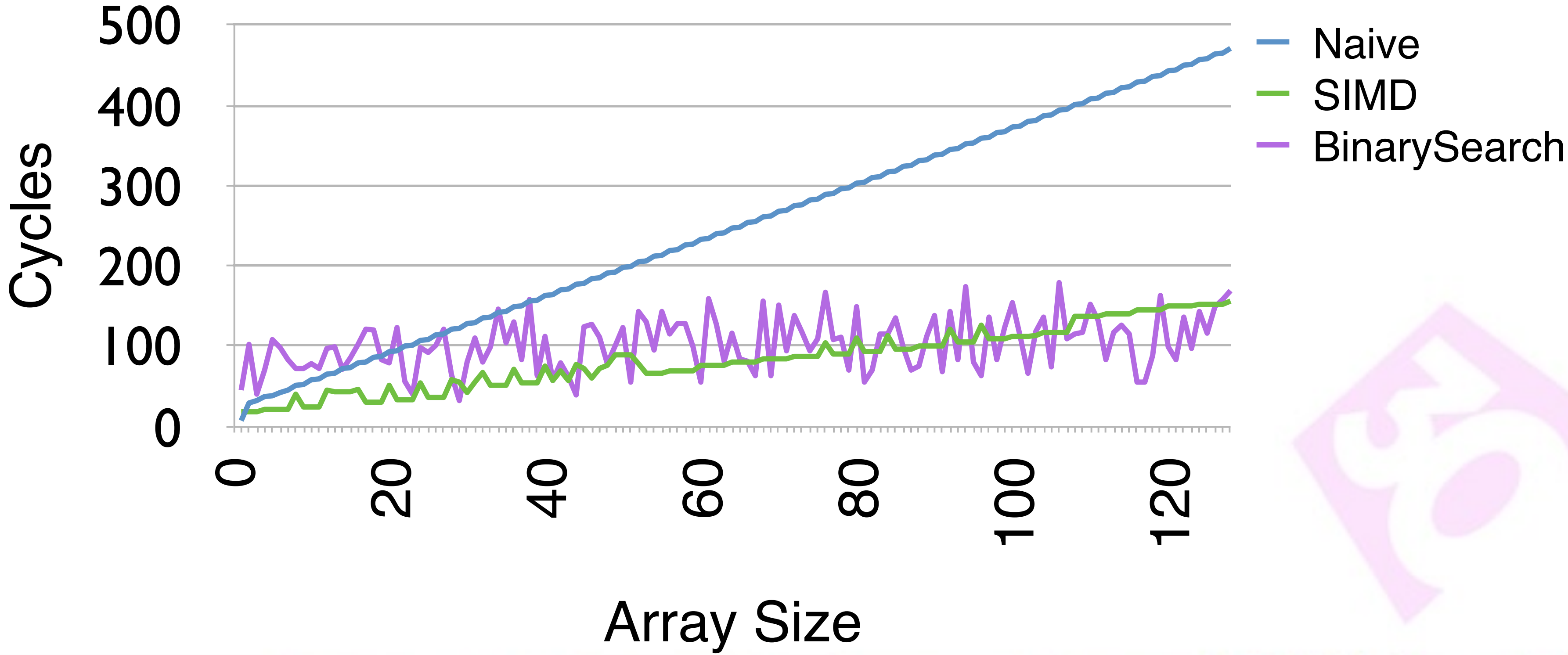
Let's try that again

```
ArraySearchSimd(unsigned int, unsigned int const*, int):
00000000000001C0 C5 F9 6E C7          vmovd        xmm0,edi
00000000000001C4 C5 F9 70 C0 00       vpshufd      xmm0,xmm0,0
00000000000001C9 89 D1                mov          ecx,edx
00000000000001CB 83 E1 FC             and          ecx,0FFFFFFFCh
00000000000001CE C5 F1 EF C9          vpxor        xmm1,xmm1,xmm1
00000000000001D2 31 C0                xor          eax,eax
00000000000001D4 85 C9                test         ecx,ecx
00000000000001D6 7E 19                jle          00000000000001F1h
00000000000001D8 31 FF                xor          edi,edi
00000000000001DA 66 0F 1F 44 00 00    nop          word ptr [rax+rax+0]
00000000000001E0 C5 F9 76 14 BE       vpcmpeqd     xmm2,xmm0,xmmword ptr [rsi+rdi*4]
00000000000001E5 C5 F1 EB CA          vpor         xmm1,xmm1,xmm2
00000000000001E9 48 83 C7 04          add          rdi,4
00000000000001ED 39 CF                cmp          edi,ecx
00000000000001EF 7C EF                jl           00000000000001E0h
00000000000001F1 89 D7                mov          edi,edx
00000000000001F3 83 E7 03             and          edi,3
00000000000001F6 74 0E                je           0000000000000206h
00000000000001F8 B9 04 00 00 00       mov          ecx,4
00000000000001FD 29 F9                sub          ecx,edi
00000000000001FF B8 FF FF FF FF       mov          eax,0FFFFFFFFh
0000000000000204 D3 E0                shl          eax,cl
0000000000000206 C5 F9 6E D0          vmovd        xmm2,eax
000000000000020A C5 E9 60 D2          vpunpcklbw   xmm2,xmm2,xmm2
000000000000020E C5 E9 61 D2          vpunpcklwd   xmm2,xmm2,xmm2
0000000000000212 48 63 C2             movsxd       rax,edx
0000000000000215 C5 F9 76 44 86 F0    vpcmpeqd     xmm0,xmm0,xmmword ptr [rsi+rax*4-10h]
000000000000021B C5 F9 DB C2          vpand        xmm0,xmm0,xmm2
000000000000021F C5 F1 EB C0          vpor         xmm0,xmm1,xmm0
0000000000000223 C5 F8 50 C0          vmovmskps    rax,xmm0
0000000000000227 85 C0                test         eax,eax
0000000000000229 0F 95 C0             setne        al
000000000000022C C3                   ret
```

SIMD performance



What about binary search?



Small Search Guidelines

- Naive code is reasonable for small counts
  - Because OOO runs Excel faster!
- Prefer SIMD for predictable <100 elem searches
  - Binary search competitive >100 32-bit elements
- Scrutinize older micro-optimization closely
- Make sure the compiler is playing for your team
  - Auto-vectorization generates terrible code sometimes

Measuring latency in cycles

- Need a way to synchronize OOO machinery
    - Retire all pending instructions, prevent scheduling
    - CPUID fits the bill - has fixed cost
- Use RDTSC to read time stamp counter
    - RDTSCP doesn't actually retire all pending instructions, can't use it. (See AMD errata.)
- Assumes platform has cycle TSCs (check yours)

Measuring, code

- Use CPUID/RDTSC/CPUID sandwich
- Subtract fixed cost later during reporting

```
xor   eax, eax
cpuid                     ; retire + prevent issues
rdtsc                     ; read TSC into edx:eax
shl   rdx, 32
lea   r15, [rax + rdx]    ; combine to 64-bit quantity, save in r15

xor   eax, eax
cpuid                     ; retire + prevent issues
```

Measuring latency

- Warm up I1 by calling the code first
- Run multiple tests to avoid interference
  - Even consoles have interrupts, OS shenanigans
- Clear cache by using _mm_clflush() in a loop

OOO Intuition

- Jaguar OOO is a loop unroller
  - Up to 64-or-so instructions
- Jaguar OOO is a prefetcher
  - And even fetches loads speculatively down branches you haven't taken yet!
- Jaguar OOO doesn't solve memory latency
  - But overlapping L2 misses is a big deal

Anonymous programmers corrected

- "Memory access isn't a problem with OOO"

Anonymous programmers corrected

- ~~"Memory access isn't a problem with OOO"~~
  - It still is. Overlap your loads!

Anonymous programmers corrected

- ~~"Memory access isn't a problem with OOO"~~
  - It still is. Overlap your loads!
- "Branches aren't a problem with OOO"

Anonymous programmers corrected

- ~~"Memory access isn't a problem with OOO"~~
  - It still is. Overlap your loads!
- ~~"Branches aren't a problem with OOO"~~
  - They still are. Avoid trees & speculative cache pollution.

Anonymous programmers corrected

- ~~"Memory access isn't a problem with OOO"~~
  - It still is. Overlap your loads!
- ~~"Branches aren't a problem with OOO"~~
  - They still are. Avoid trees & speculative cache pollution.
- "SIMD isn't necessary on OOO"

Anonymous programmers corrected

- ~~"Memory access isn't a problem with OOO"~~
  - It still is. Overlap your loads!
- ~~"Branches aren't a problem with OOO"~~
  - They still are. Avoid trees & speculative cache pollution.
- ~~"SIMD isn't necessary on OOO"~~
  - It's the only way to get the full cache bandwidth!

Takeaways for Jaguar Perf

- Unrolling, prefetching are of limited use
  - Measure carefully, consider maintenance aspects
- Use arrays
  - Really, really, really consider using an array
  - Linked lists turns OOO into in-order disaster
- Use SIMD
  - See my talk from last year for more meat

Resources

- Software Optimization Guide for AMD Family 16h Processors (AMD, pdf)
- http://www.agner.org/optimize/#manuals
- "JAGUAR" AMD’s Next Generation Low Power x86 Core, Jeff Rupley, AMD Fellow

Thank you! - Q & A

email: afredriksson@insomniacgames.com

twitter: @deplinenoise

Special thanks to:

Mark Cerny

Fabian Giesen

Jonathan Adamczewski

Mike Acton & the rest of the Insomniac Core team

Bonus: Hot D1, Cold L2

- Jaguar has an inclusive cache hierarchy
    - All D1/I1 lines must also be in L2
- L2 hears about all D1 misses
- L2 hears nothing about D1 *hits*
- ...
- So what if you have a routine that does nothing but HIT D1?

Bonus: Hot D1, Cold L2

- Net effect: White hot D1 data can be evicted
  - L2 assoc = 16 lines, they WILL be reused
  - Our data looks old in the LRU order and the L2 hasn't heard about it for a while..
- End game: Inner loop has to L2 miss all the way to main memory randomly to get back its really hot data
- In practice not a big deal, but can definitely show up