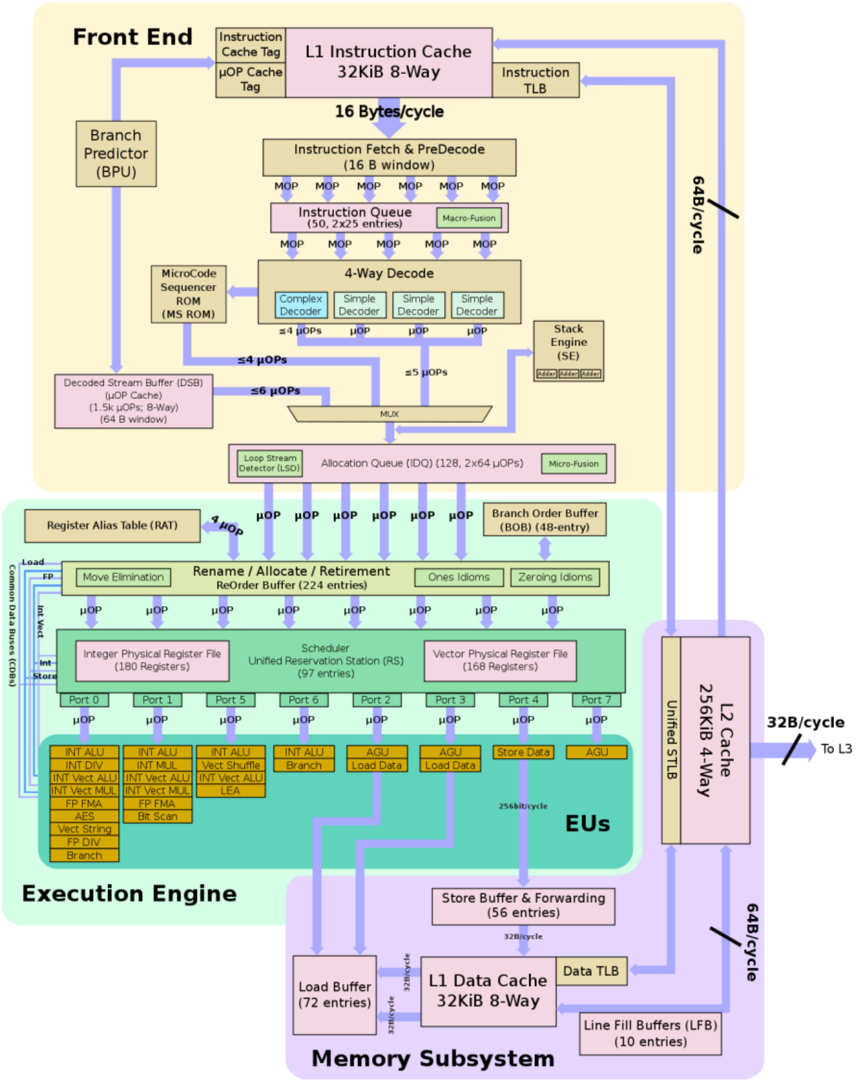
谢天

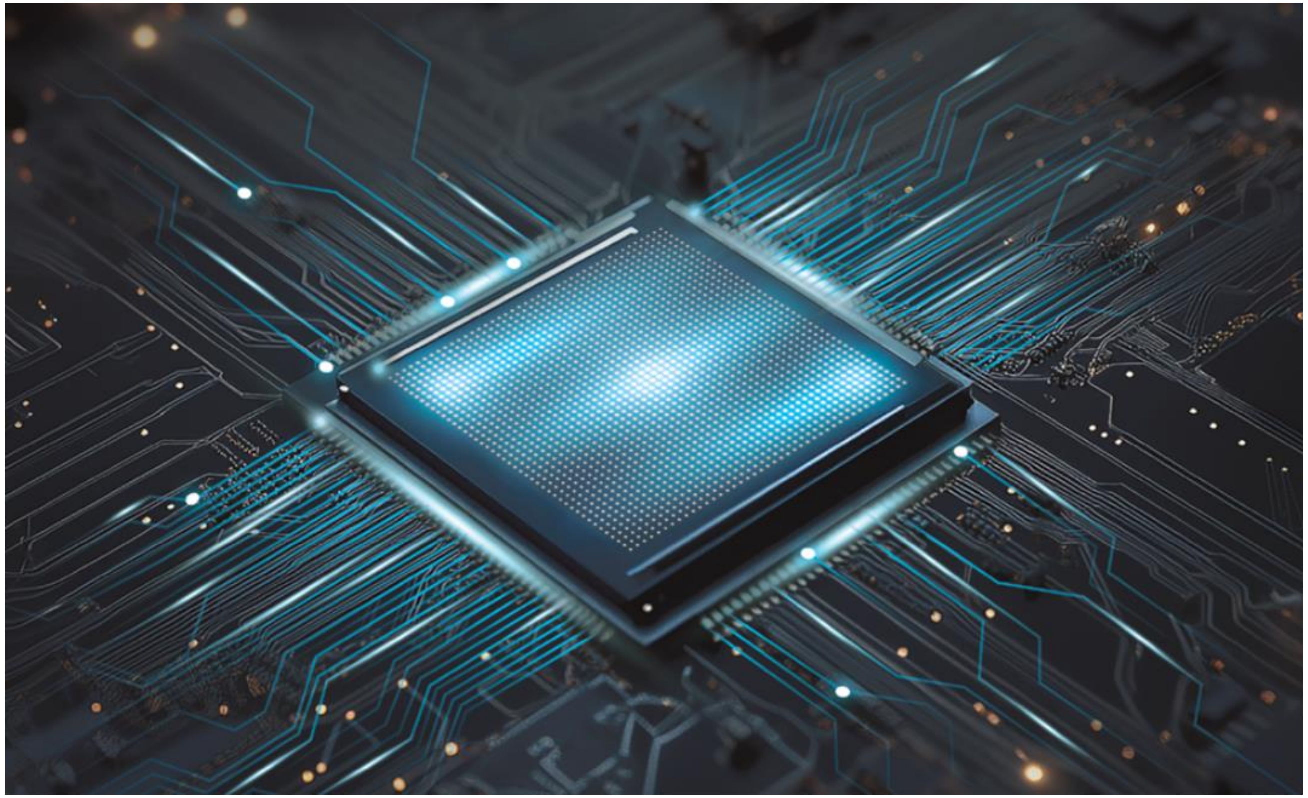
单核篇

CPU Performance



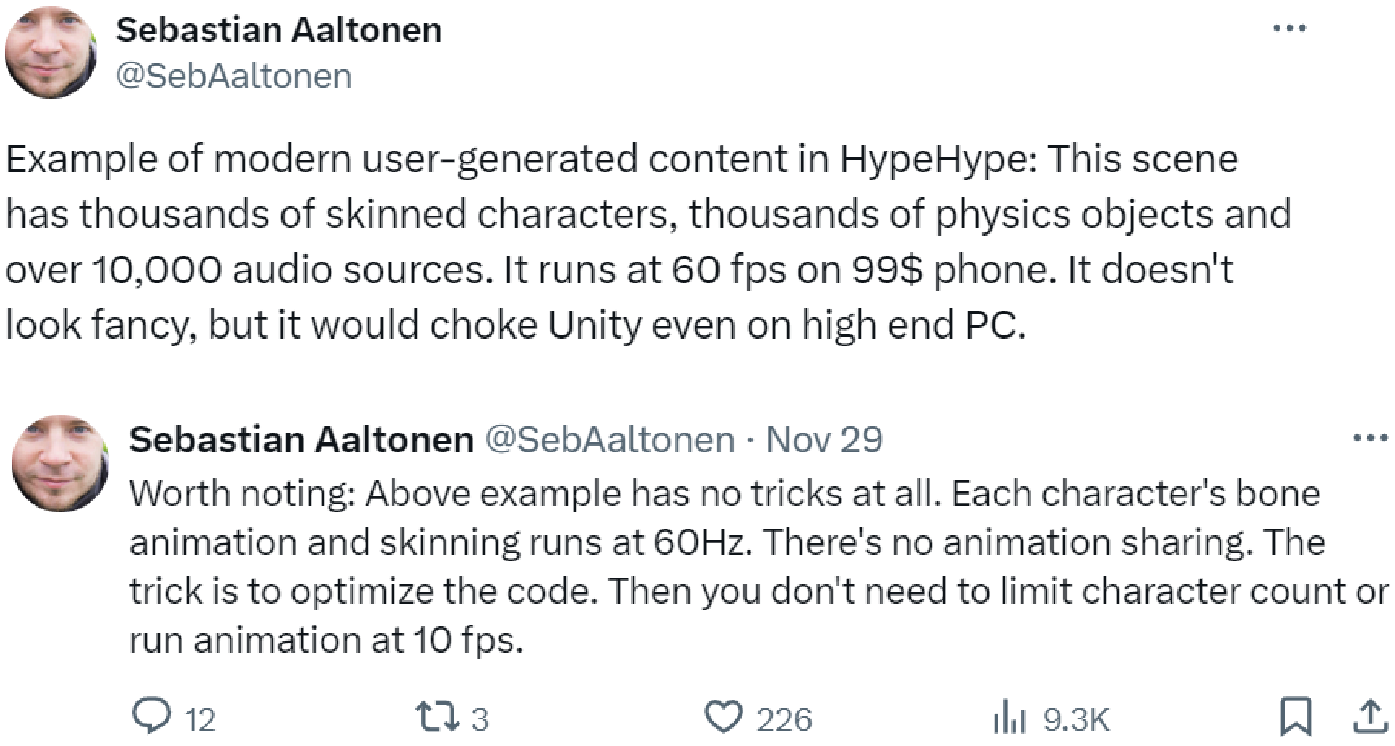
Disclaimer

声明



1. 分享内容基于个人的经验，测试数据都是特定情况下的数据，不具有通用性
2. Opinions are my own，我的观点不代表FF引擎组 or 公司的观点
3. 时间有限，只能分享一些我觉得有趣的方面，欢迎提问交流/会后交流

Performance Matters

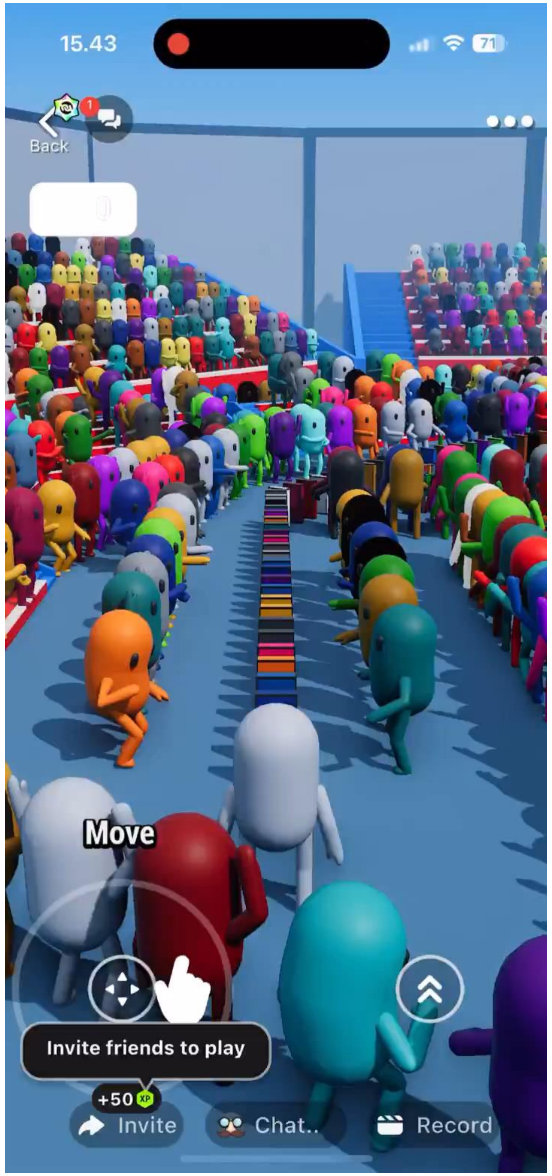


Sebastian Aaltonen
@SebAaltonen

Example of modern user-generated content in HypeHype: This scene has thousands of skinned characters, thousands of physics objects and over 10,000 audio sources. It runs at 60 fps on 99$ phone. It doesn't look fancy, but it would choke Unity even on high end PC.

Sebastian Aaltonen @SebAaltonen · Nov 29

Worth noting: Above example has no tricks at all. Each character's bone animation and skinning runs at 60Hz. There's no animation sharing. The trick is to optimize the code. Then you don't need to limit character count or run animation at 10 fps.



https://x.com/SebAaltonen/status/1862397887541387424

Performance Tuning

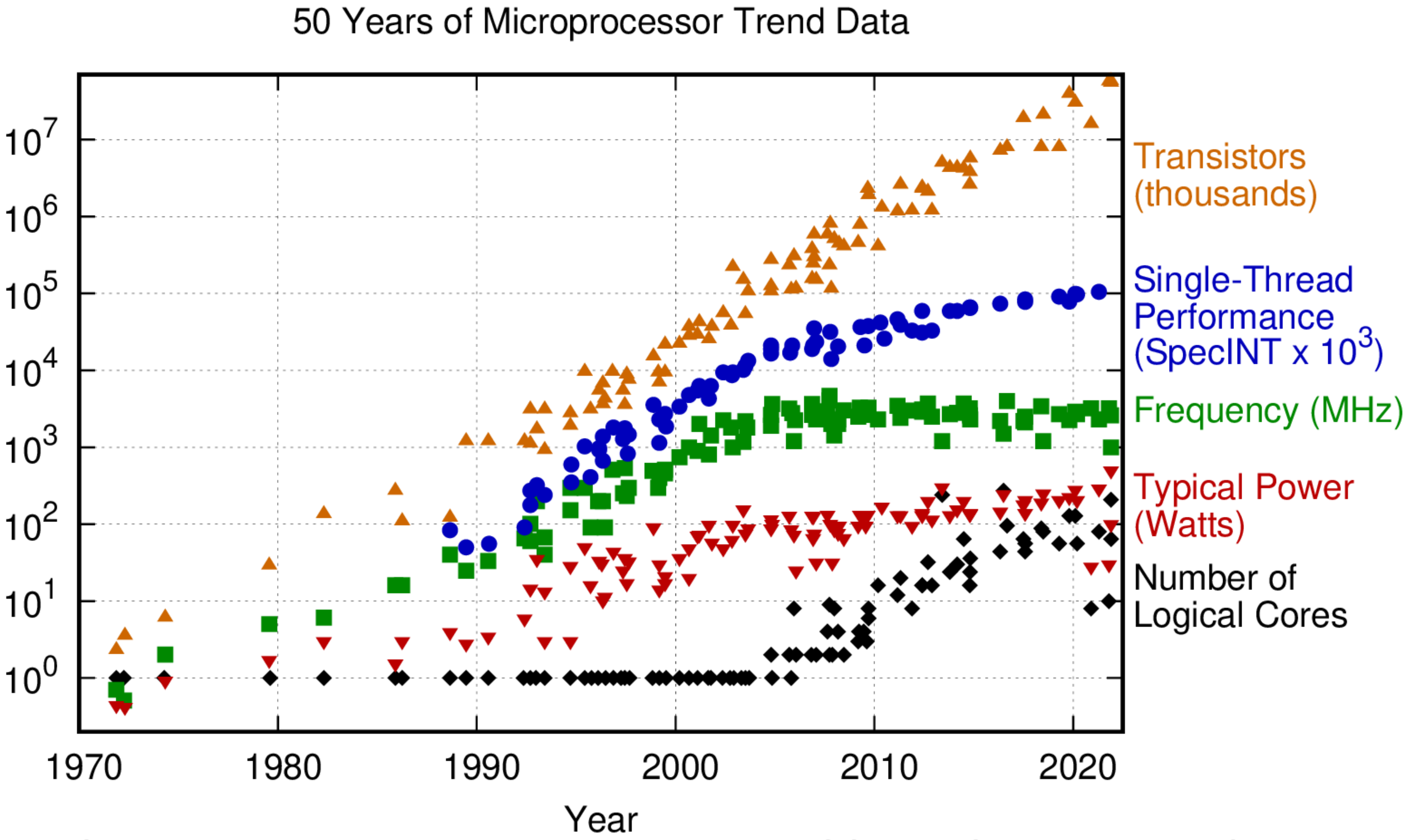
Low-level optimizations (On Single Core)

takes into account the details of the underlying hardware capabilities.

High-level optimizations

more about application-level logic, algorithms, and data structures.

The free lunch is over.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data?tab=readme-ov-file

性能分析方法

责怪他人法

~~1.找到一个不是你负责的系统或环境的组件~~

~~2.假定问题是与那个组件相关的~~

~~3.把问题扔给负责那个组件的团队~~

~~4.如果证明错了，返回步骤1~~

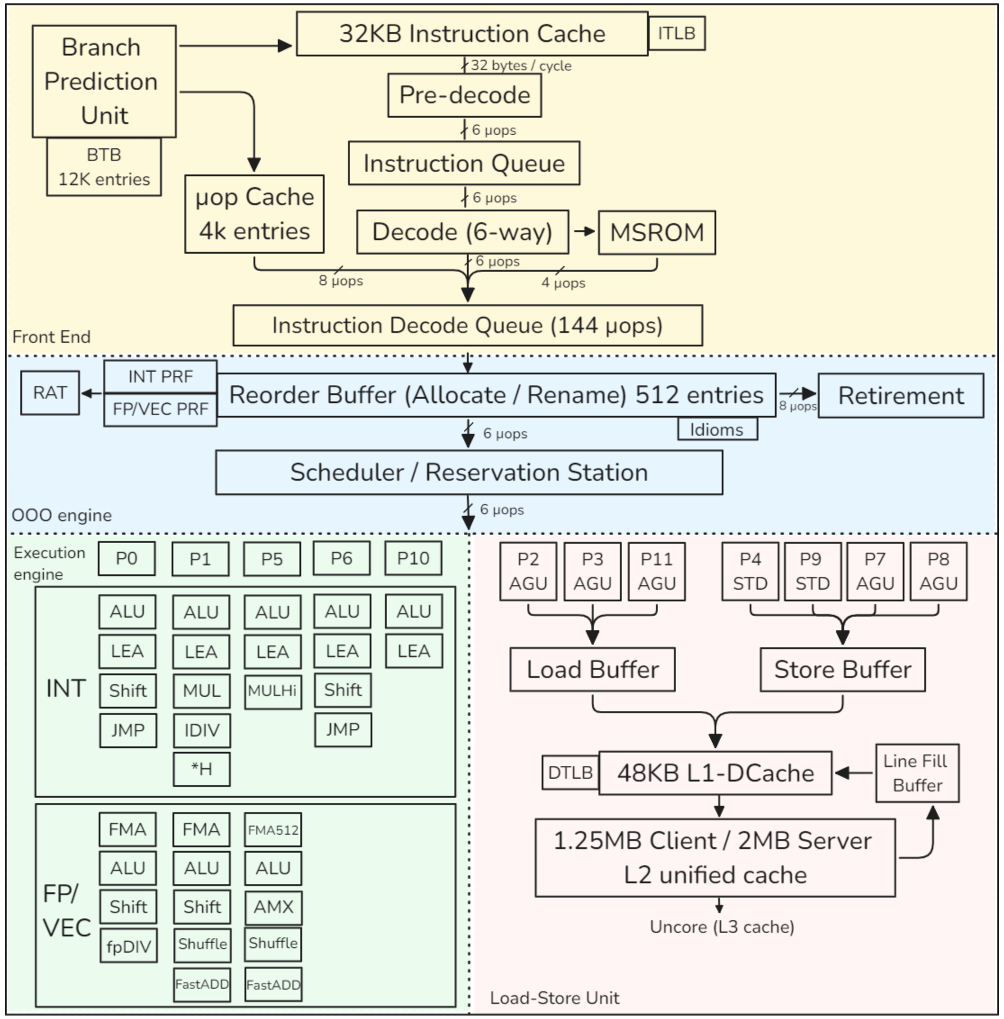
性能分析方法

Identifying CPU bottlenecks

Top-down Microarchitecture Analysis Method

https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html

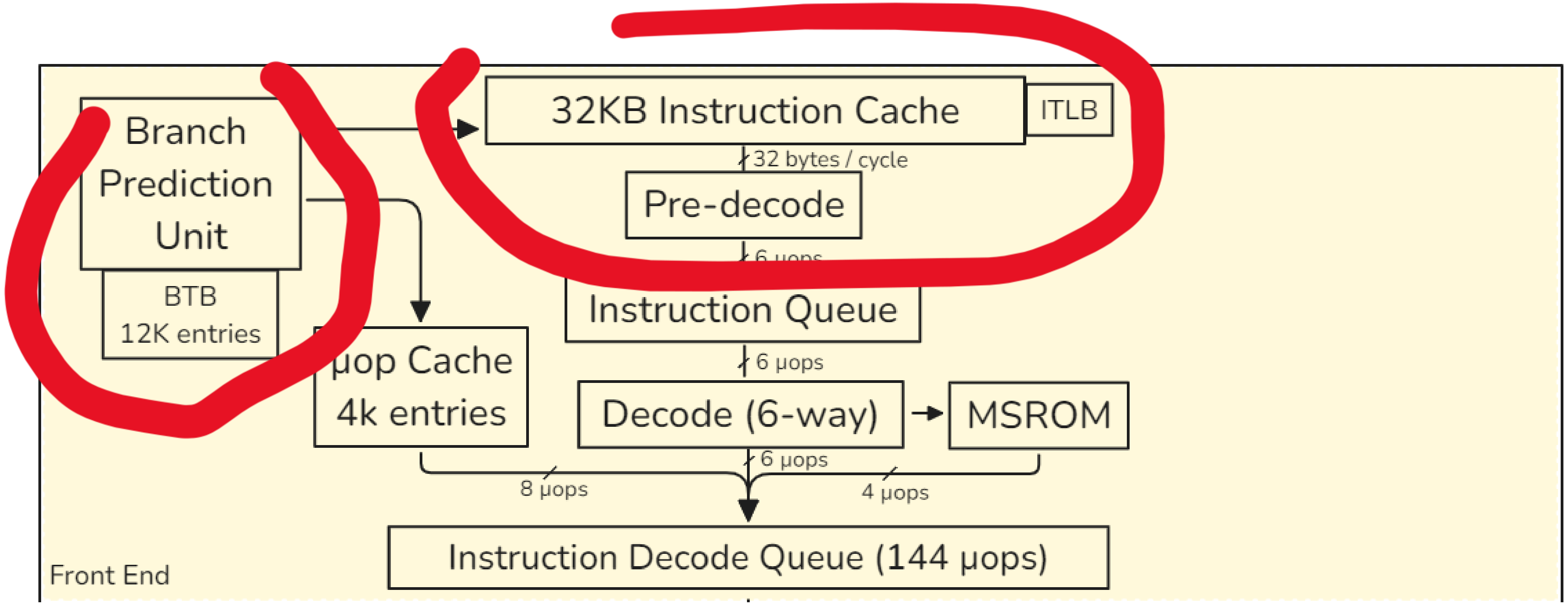
Microarchitecture: CPU simplified view



the implementation of Intel 12th-generation core, Golden Cove

Front End

Fetch and decode instructions from memory



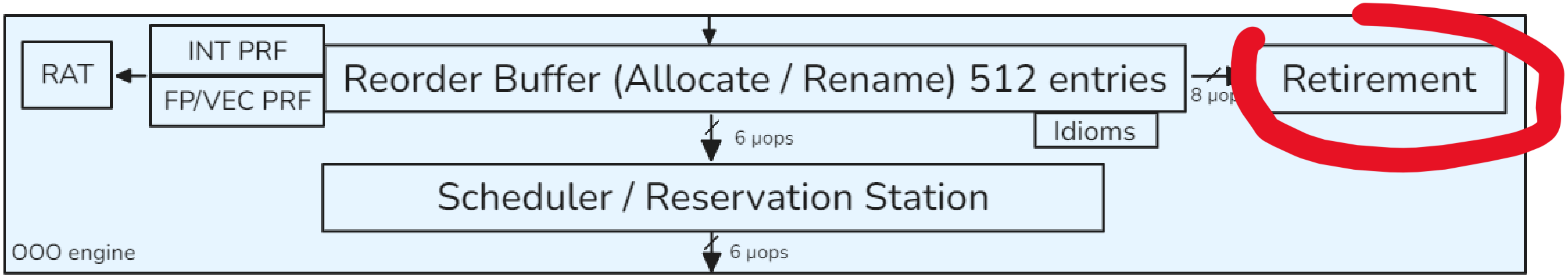
x86: translate CISC instructions into simple µops => TMAM based on **µops/uop**

The CPU Frontend **fetches** 32 bytes per cycle of x86 instructions from the L1 I-cache => **Potential stall!**

The **BPU** predicts the target of all branch instructions based on prediction => **Potential stall!**

Back End

an OOO engine that executes instructions and stores results

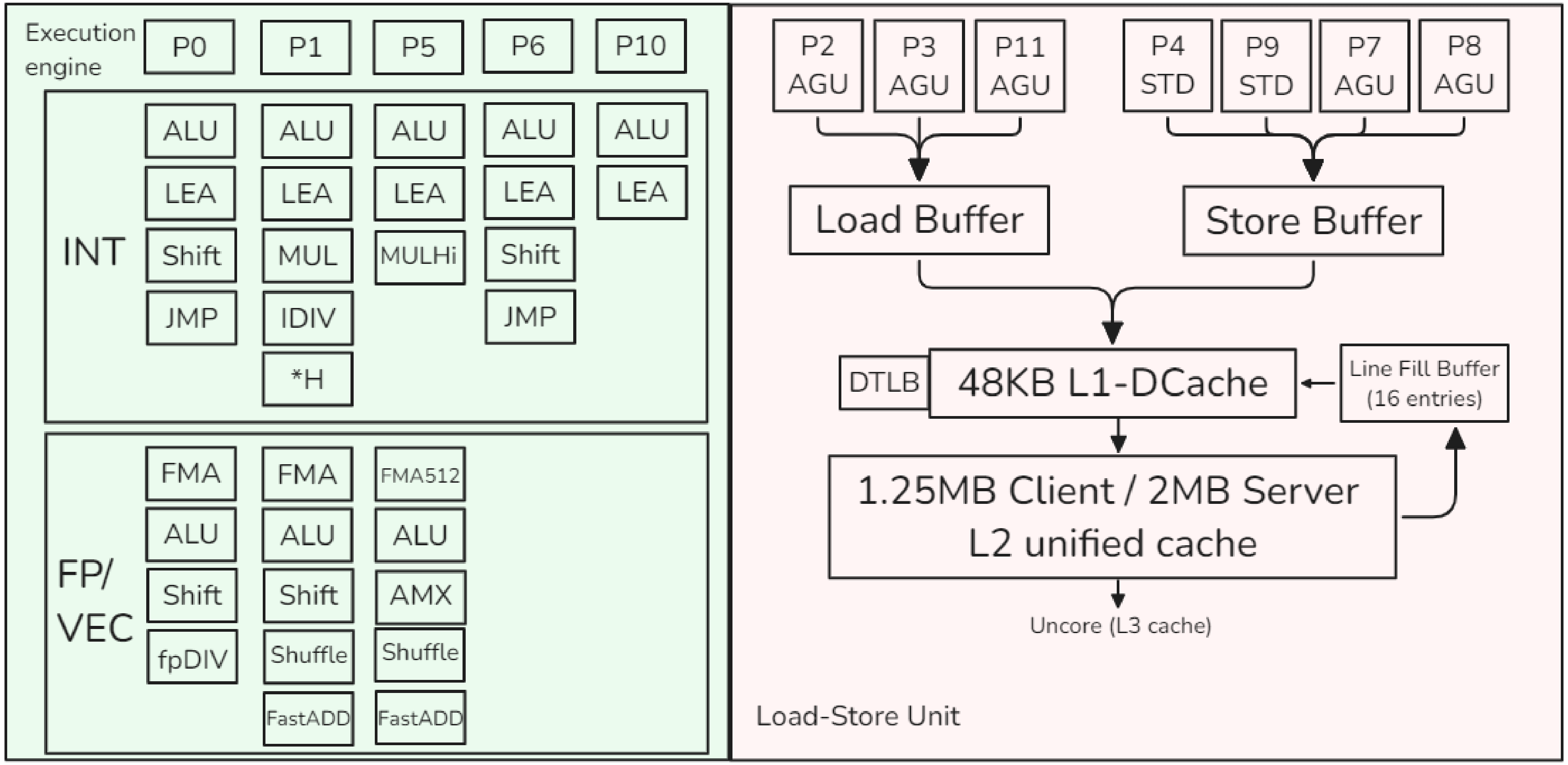

RAT
INT PRF
FP/VEC PRF
Reorder Buffer (Allocate / Rename) 512 entries
8 µops
Retirement
Idioms
6 µops
Scheduler / Reservation Station
6 µops
OOO engine

ReOrder Buffer: Register renaming, allocates execution resources, tracks speculative execution.

The Scheduler / Reservation Station: tracks the availability of all resources for a given µop and dispatches the µop to an execution port once it is ready.

Back End

The execution engine and the Load-Store unit



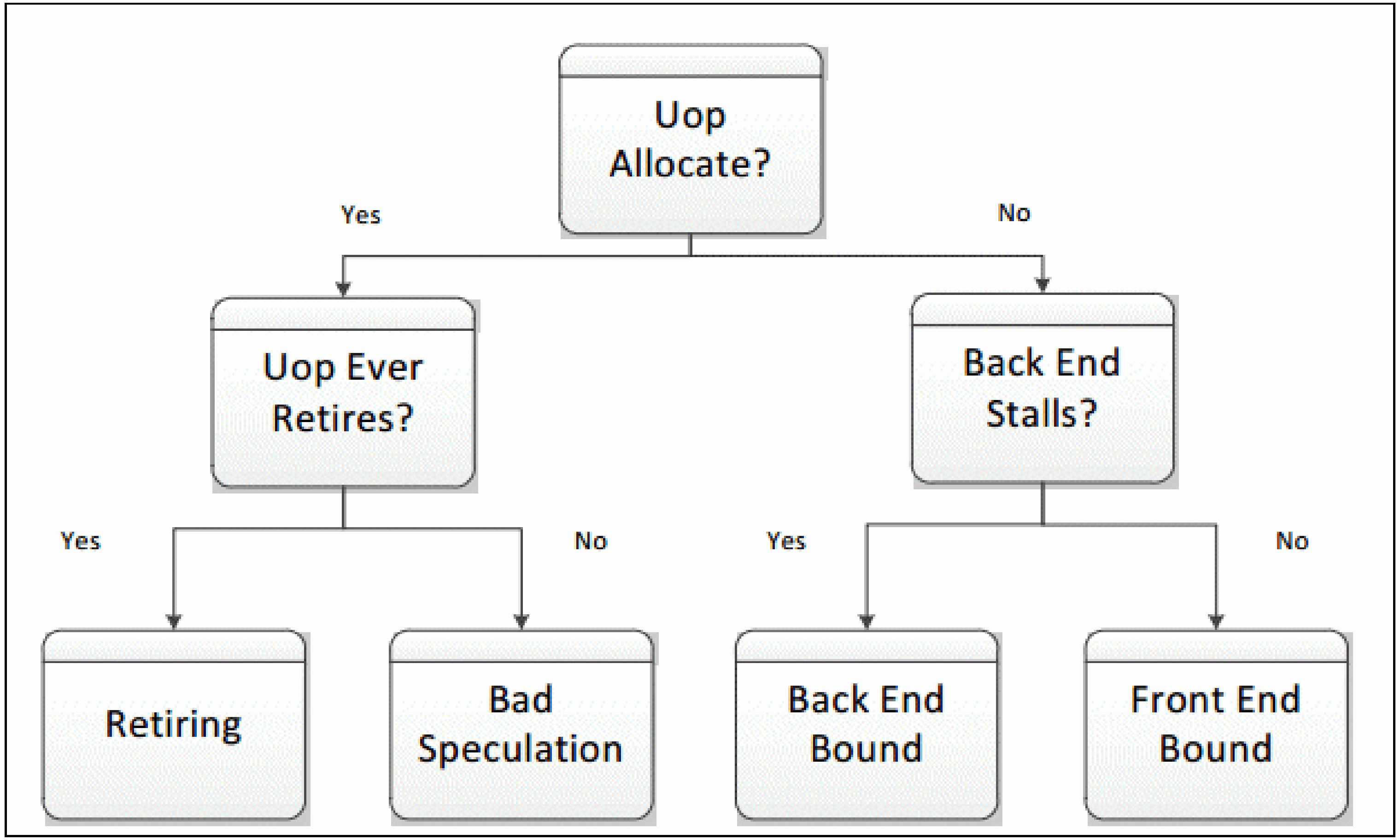
Potential stall!

Potential stall!

12 execution ports

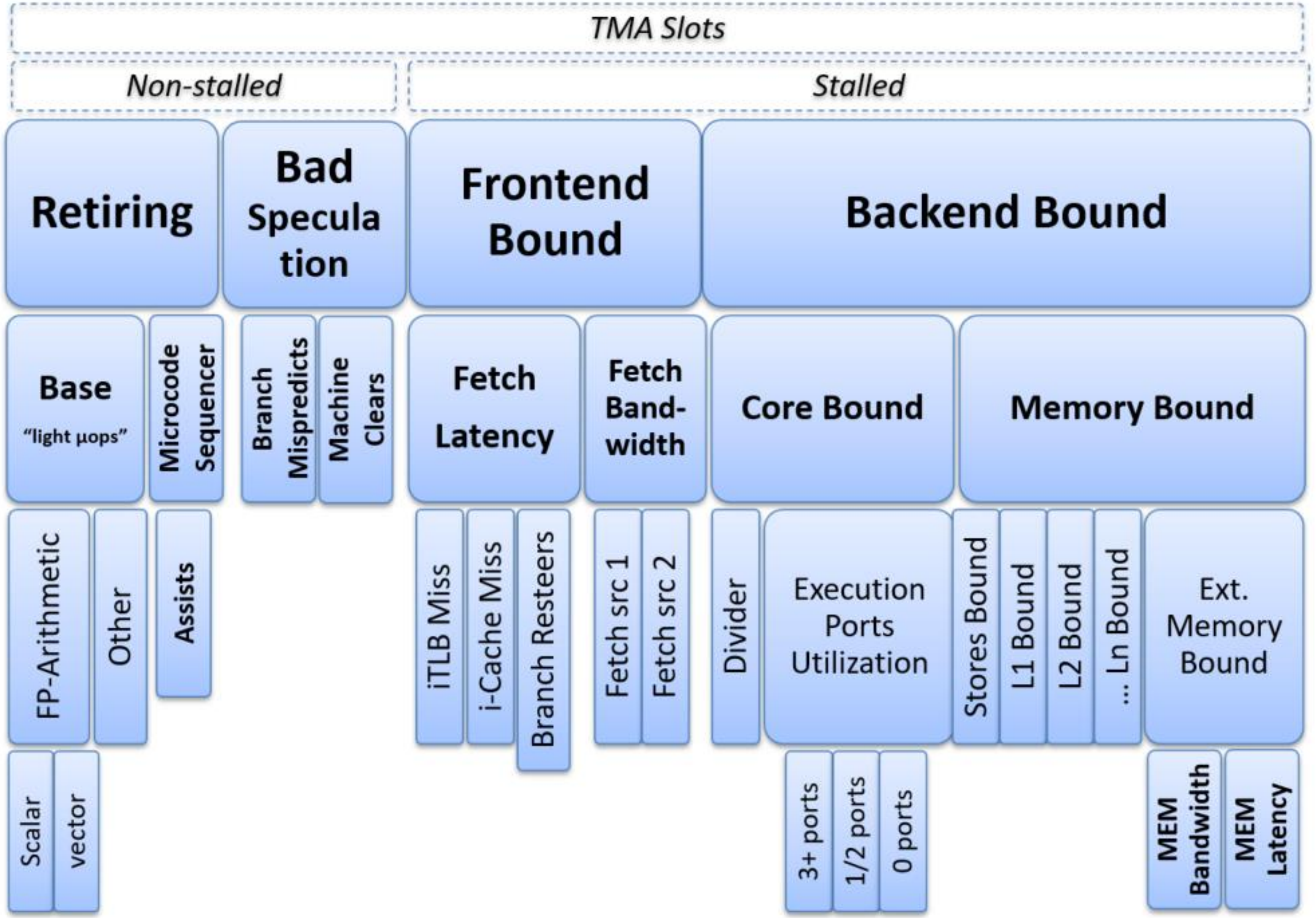
When a scheduler has to dispatch two operations that require the same execution port, one of them will have to be delayed.

TMAM: Pipeline slot



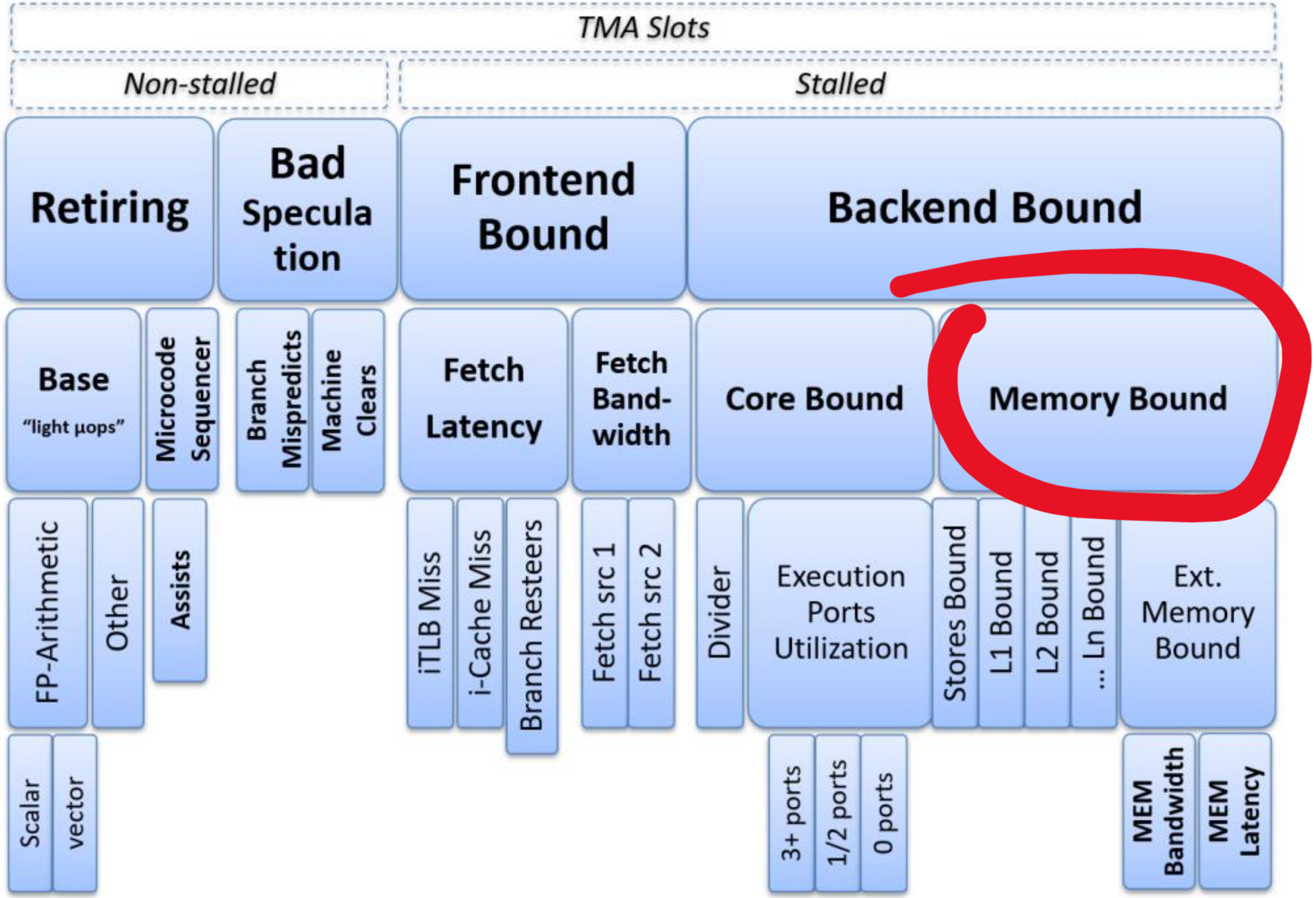
https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html

TMA hierarchy of performance bottlenecks

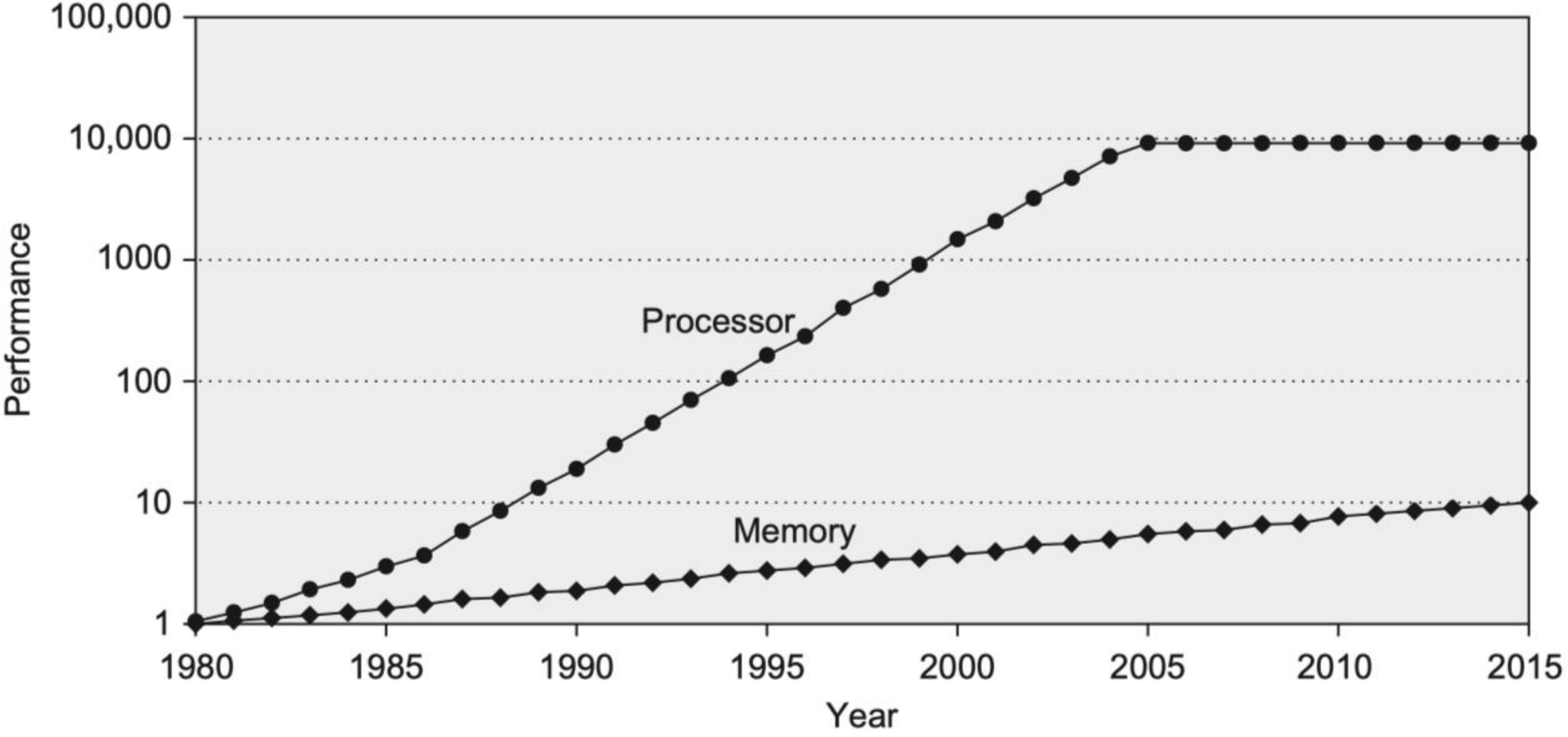


[Yasin, 2014]

Four Corners

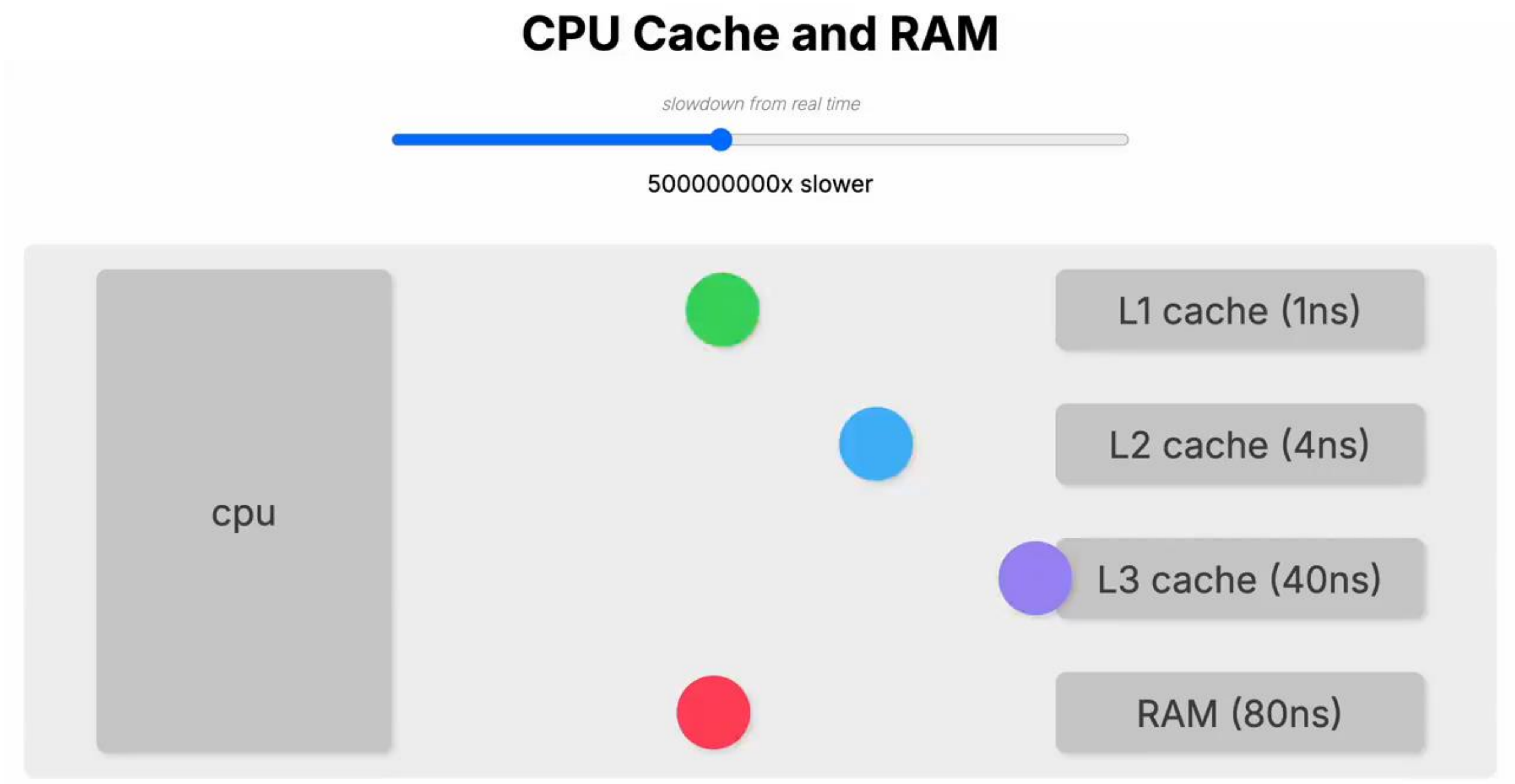


Processor Memory Gap



[Hennessy & Patterson, 2017]

内存层次



CPU Cache and RAM

slowdown from real time

500000000x slower

https://x.com/ptoboley/status/1847370345302593930

延迟

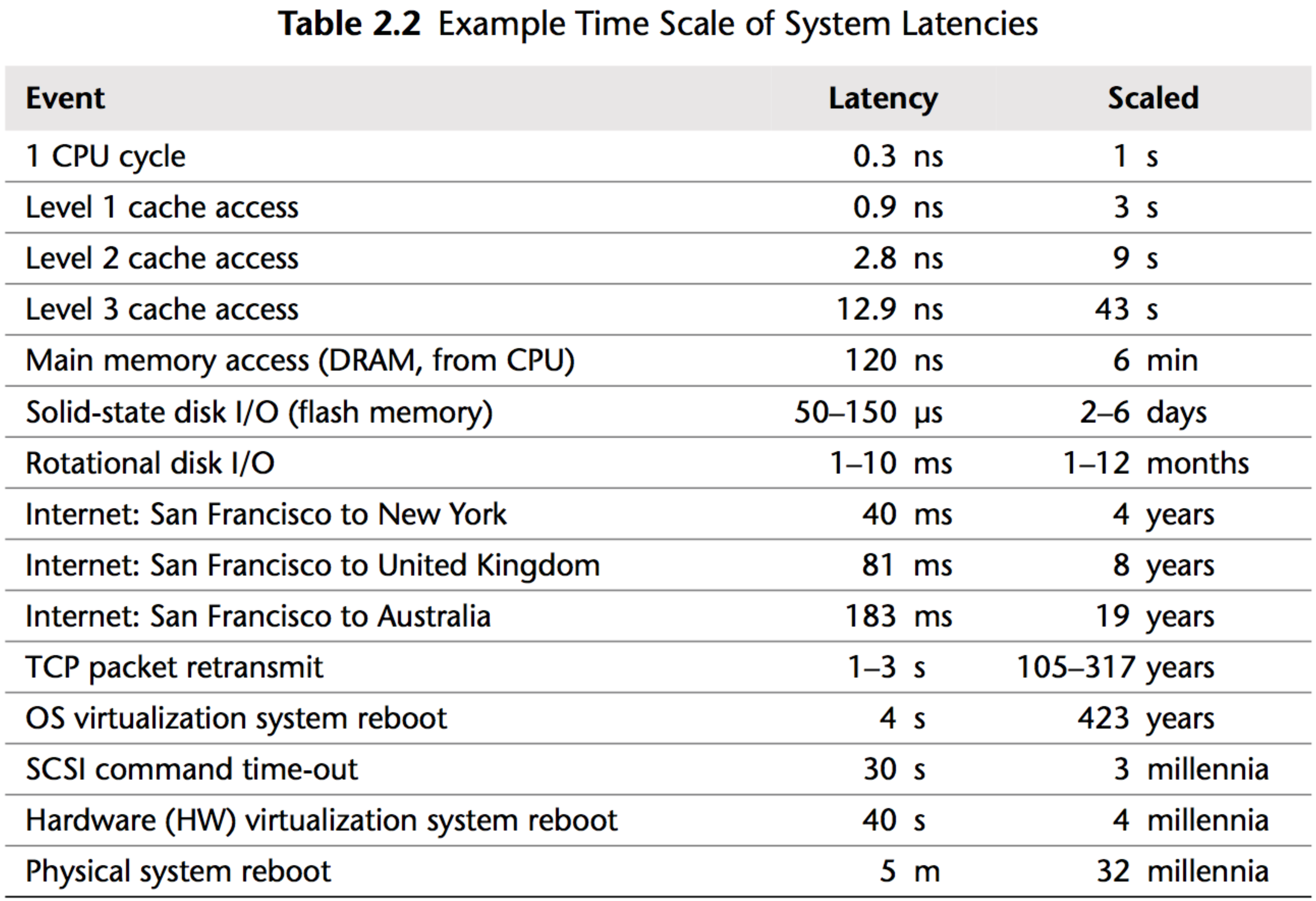
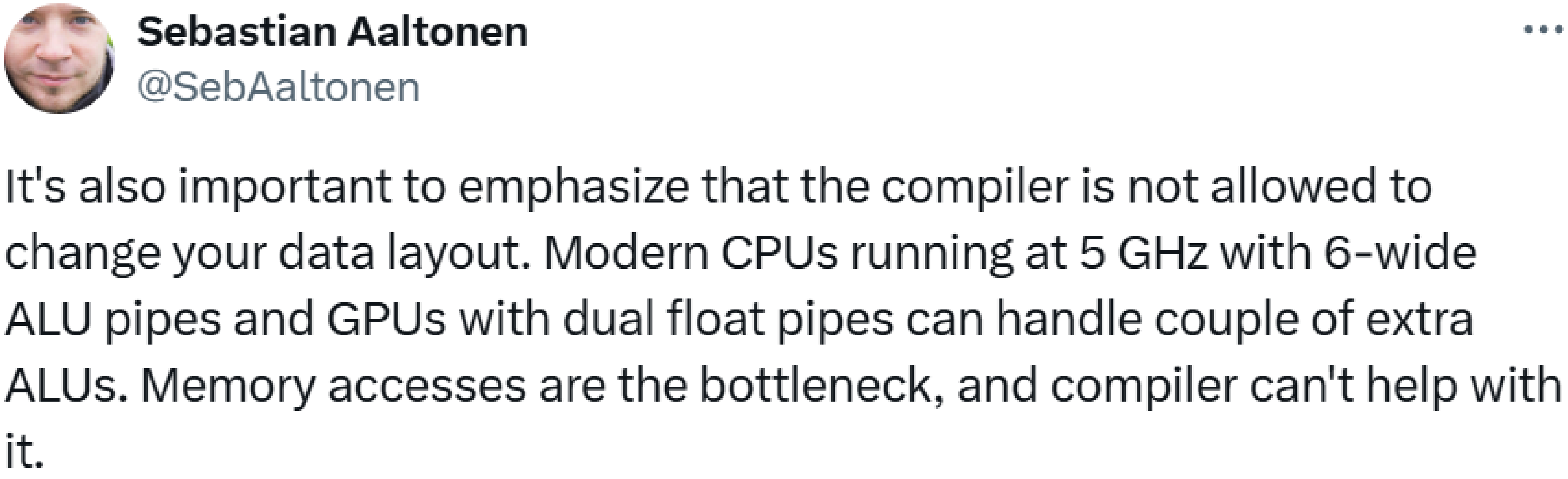


Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

What Compilers Cannot Do

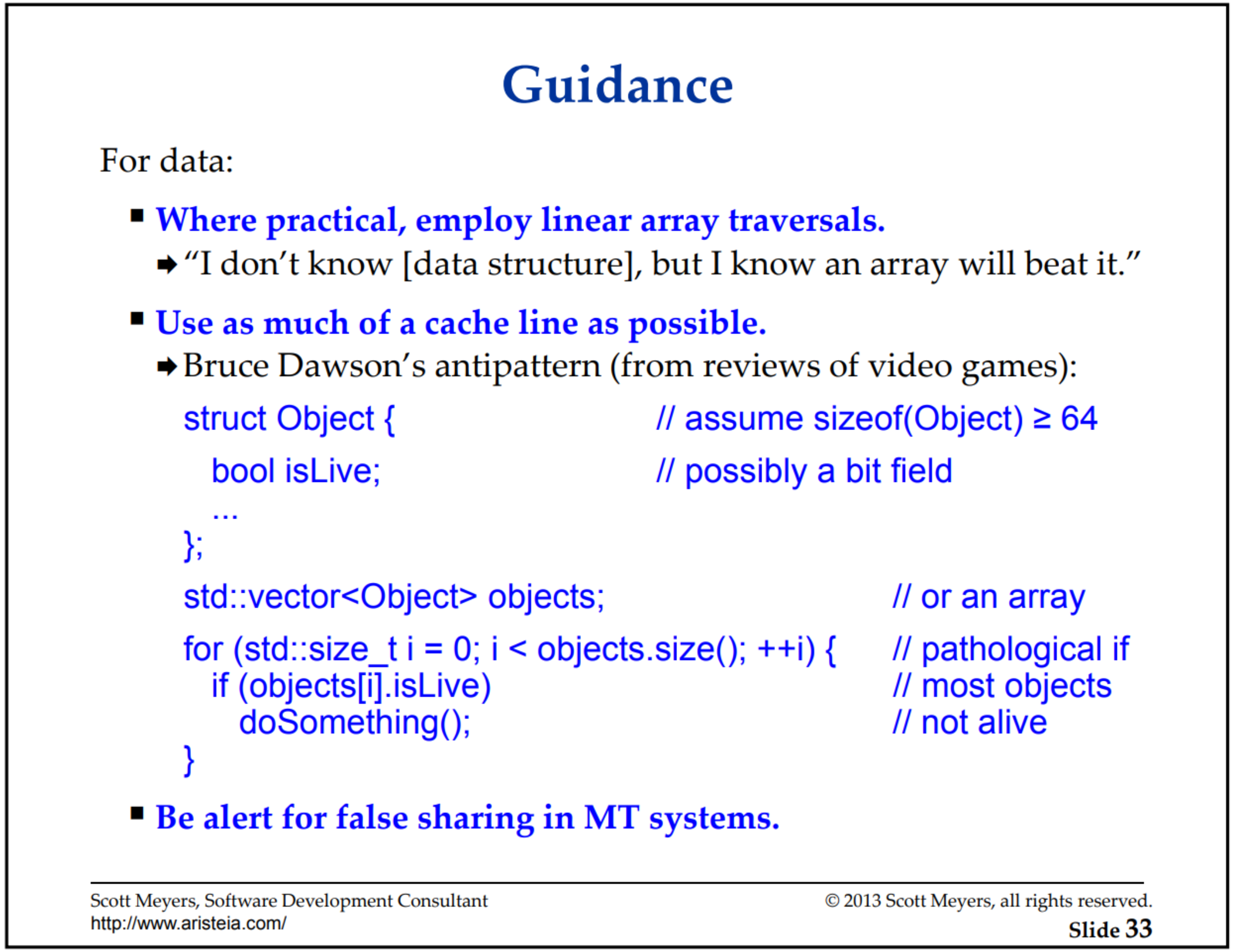


Sebastian Aaltonen
@SebAaltonen

It's also important to emphasize that the compiler is not allowed to change your data layout. Modern CPUs running at 5 GHz with 6-wide ALU pipes and GPUs with dual float pipes can handle couple of extra ALUs. Memory accesses are the bottleneck, and compiler can't help with it.

https://x.com/SebAaltonen/status/1861718633354756151

CPU Love Array



Guidance

For data:

- **Where practical, employ linear array traversals.**
  - ➡ "I don't know [data structure], but I know an array will beat it."
- **Use as much of a cache line as possible.**
  - ➡ Bruce Dawson's antipattern (from reviews of video games):

```
struct Object {                    // assume sizeof(Object) ≥ 64
  bool isLive;                     // possibly a bit field
  ...
};

std::vector<Object> objects;                         // or an array

for (std::size_t i = 0; i < objects.size(); ++i) {   // pathological if
  if (objects[i].isLive)                             // most objects
    doSomething();                                   // not alive
}
```

- **Be alert for false sharing in MT systems.**

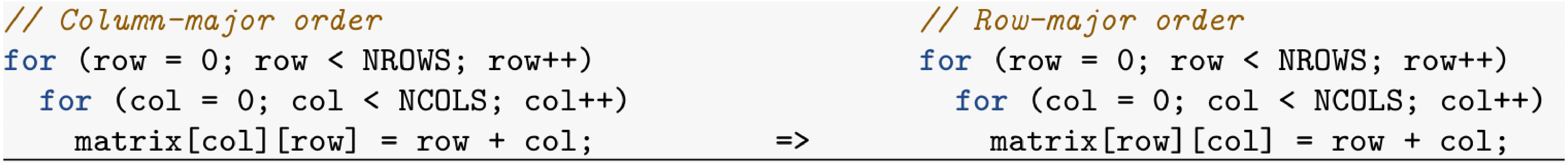
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Slide 33

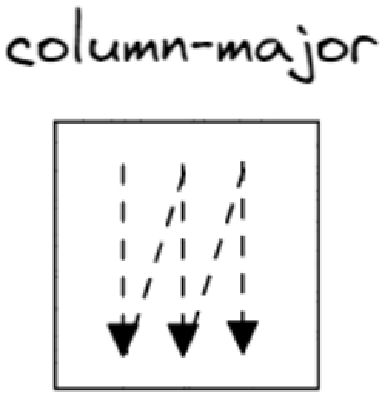
code::dive conference 2014 - Scott Meyers: Cpu Caches and Why You Care

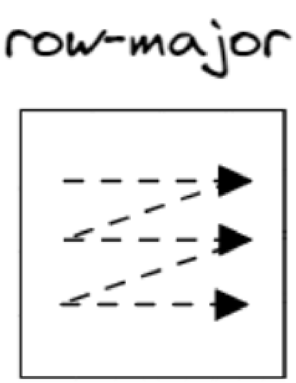
Example: Loop interchange

矩阵乘法



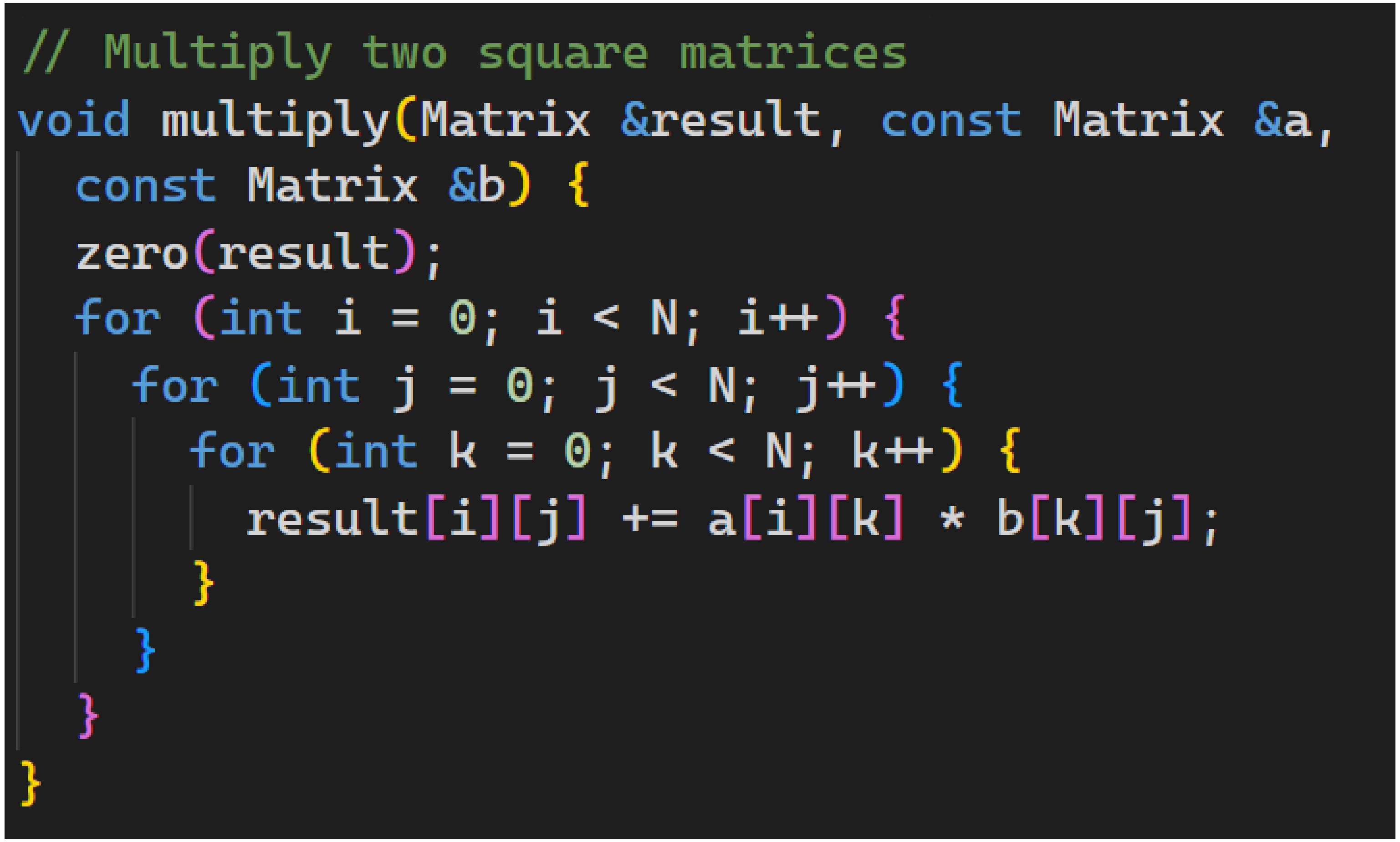
```
// Column-major order                              // Row-major order
for (row = 0; row < NROWS; row++)                  for (row = 0; row < NROWS; row++)
  for (col = 0; col < NCOLS; col++)                  for (col = 0; col < NCOLS; col++)
    matrix[col][row] = row + col;            =>        matrix[row][col] = row + col;
```





Example: Loop interchange

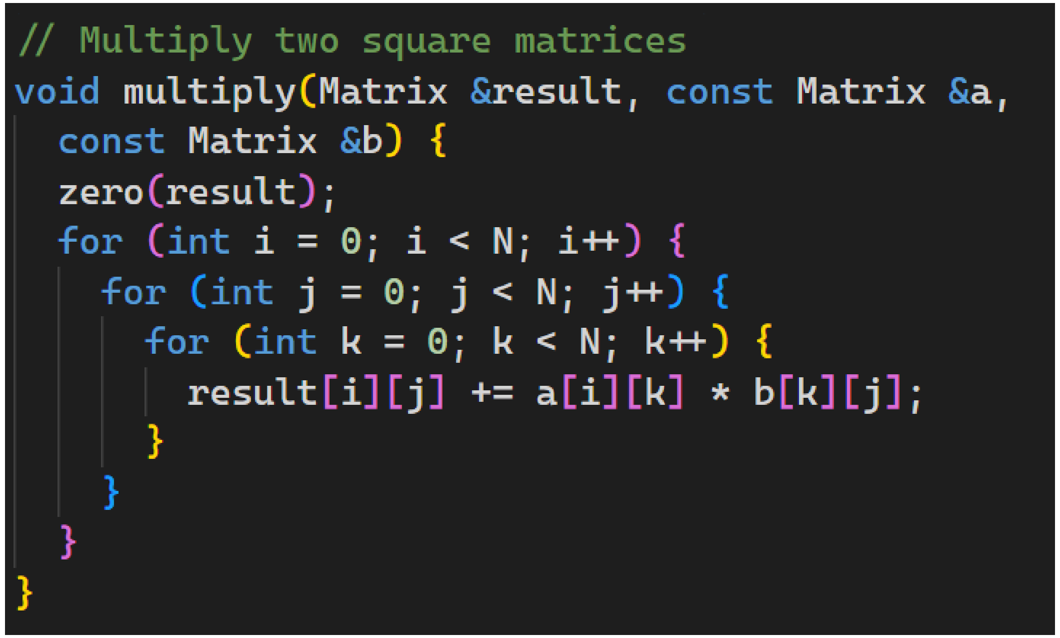
矩阵乘法



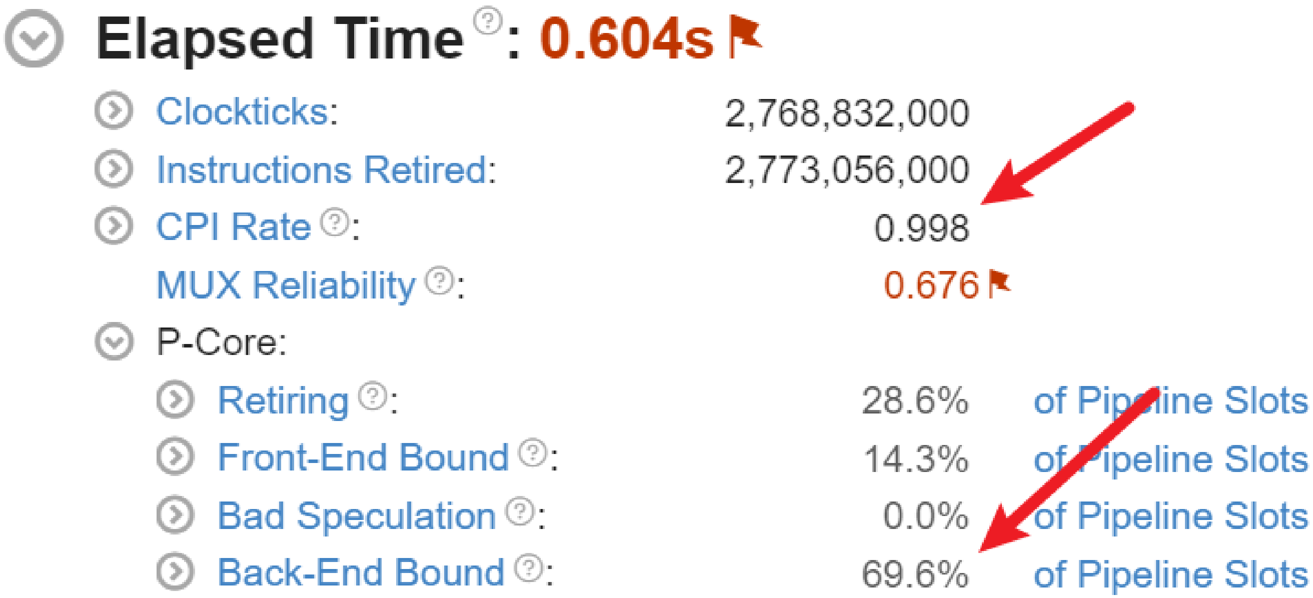
```cpp
// Multiply two square matrices
void multiply(Matrix &result, const Matrix &a,
  const Matrix &b) {
  zero(result);
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Loop interchange

矩阵乘法



```
// Multiply two square matrices
void multiply(Matrix &result, const Matrix &a,
  const Matrix &b) {
  zero(result);
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

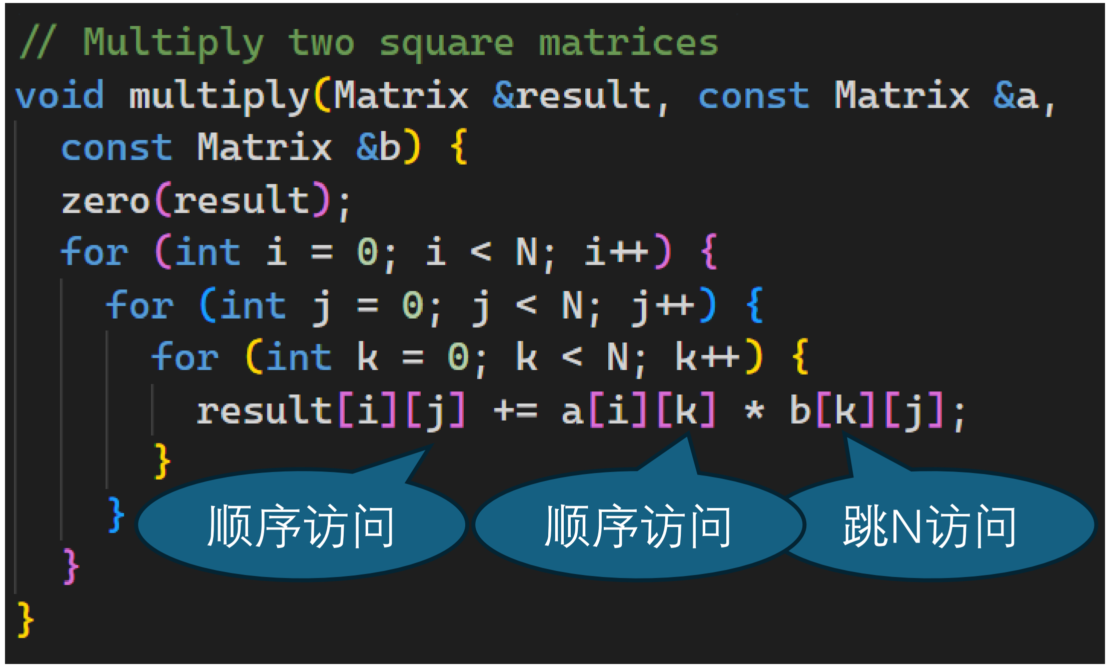


Elapsed Time: 0.604s

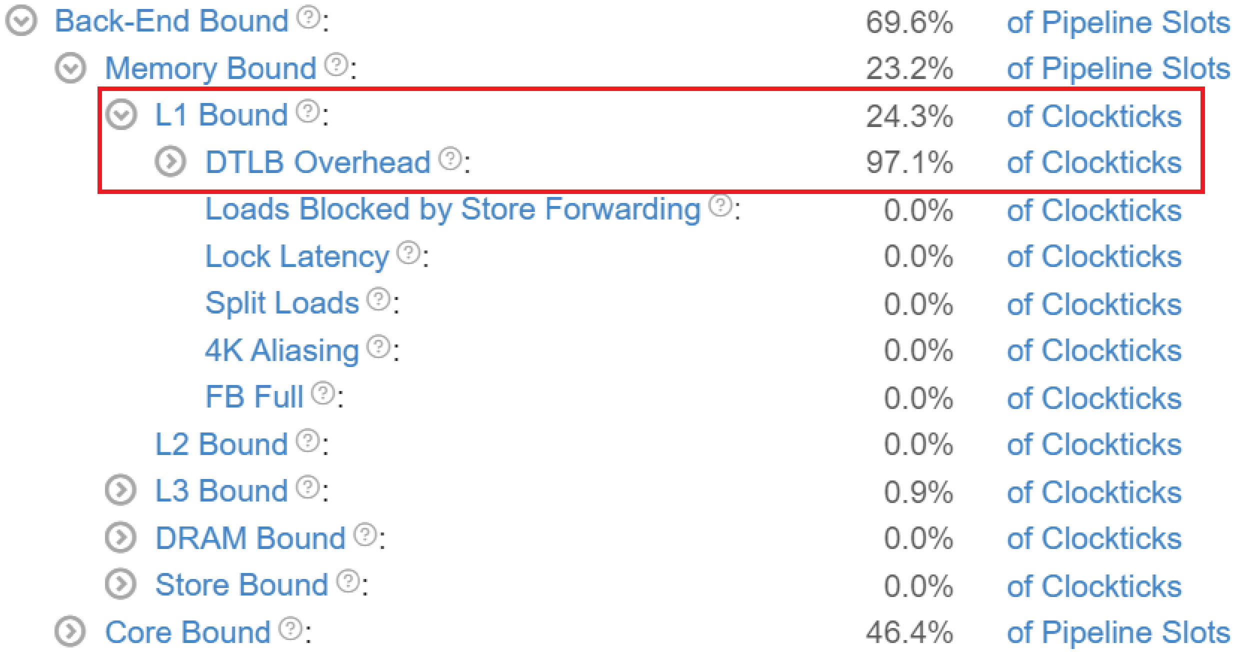
Clockticks:	2,768,832,000	
Instructions Retired:	2,773,056,000	
CPI Rate:	0.998	
MUX Reliability:	0.676	
P-Core:		
Retiring:	28.6%	of Pipeline Slots
Front-End Bound:	14.3%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	69.6%	of Pipeline Slots

Loop interchange

矩阵乘法



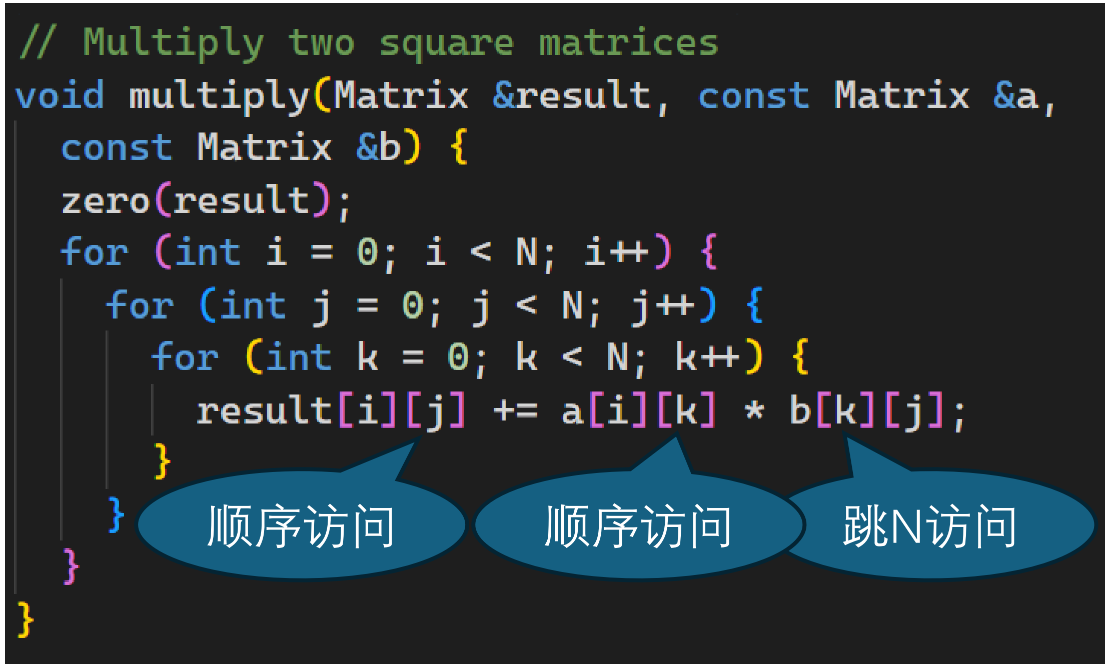
```
// Multiply two square matrices
void multiply(Matrix &result, const Matrix &a,
  const Matrix &b) {
  zero(result);
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```



Back-End Bound:	69.6%	of Pipeline Slots
Memory Bound:	23.2%	of Pipeline Slots
L1 Bound:	24.3%	of Clockticks
DTLB Overhead:	97.1%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.0%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.9%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	46.4%	of Pipeline Slots

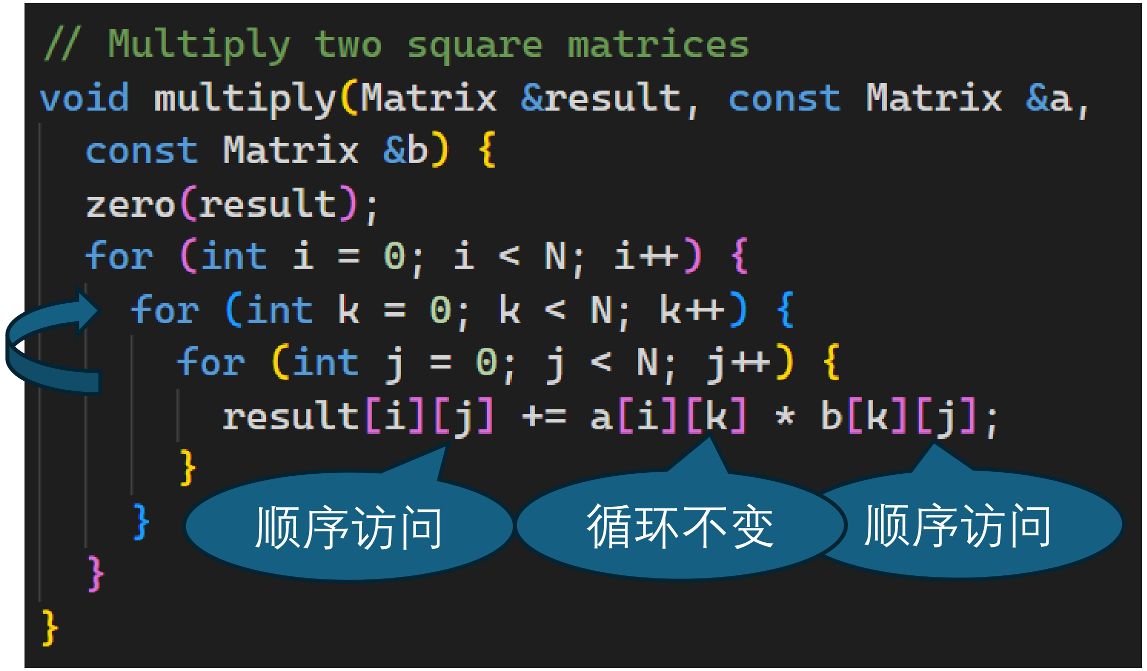
Loop interchange

优化前



```
// Multiply two square matrices
void multiply(Matrix &result, const Matrix &a,
  const Matrix &b) {
  zero(result);
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

优化后

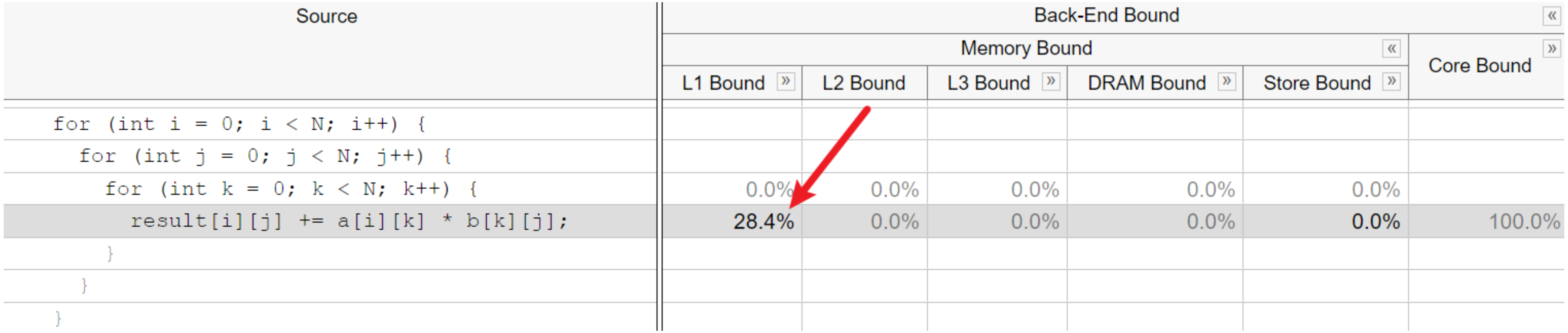


```
// Multiply two square matrices
void multiply(Matrix &result, const Matrix &a,
  const Matrix &b) {
  zero(result);
  for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
      for (int j = 0; j < N; j++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

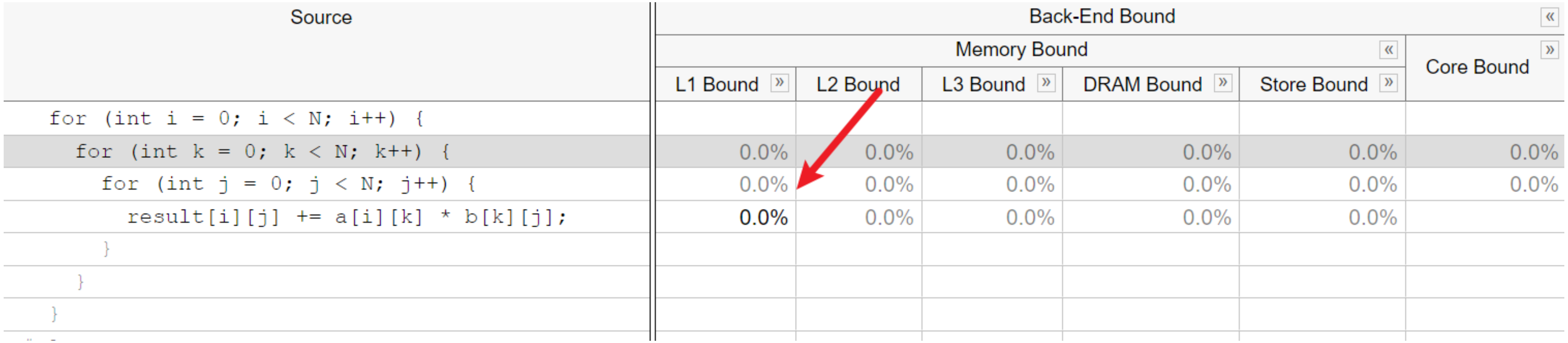
88% speed up

https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/memory_bound/loop_interchange_1

Loop interchange

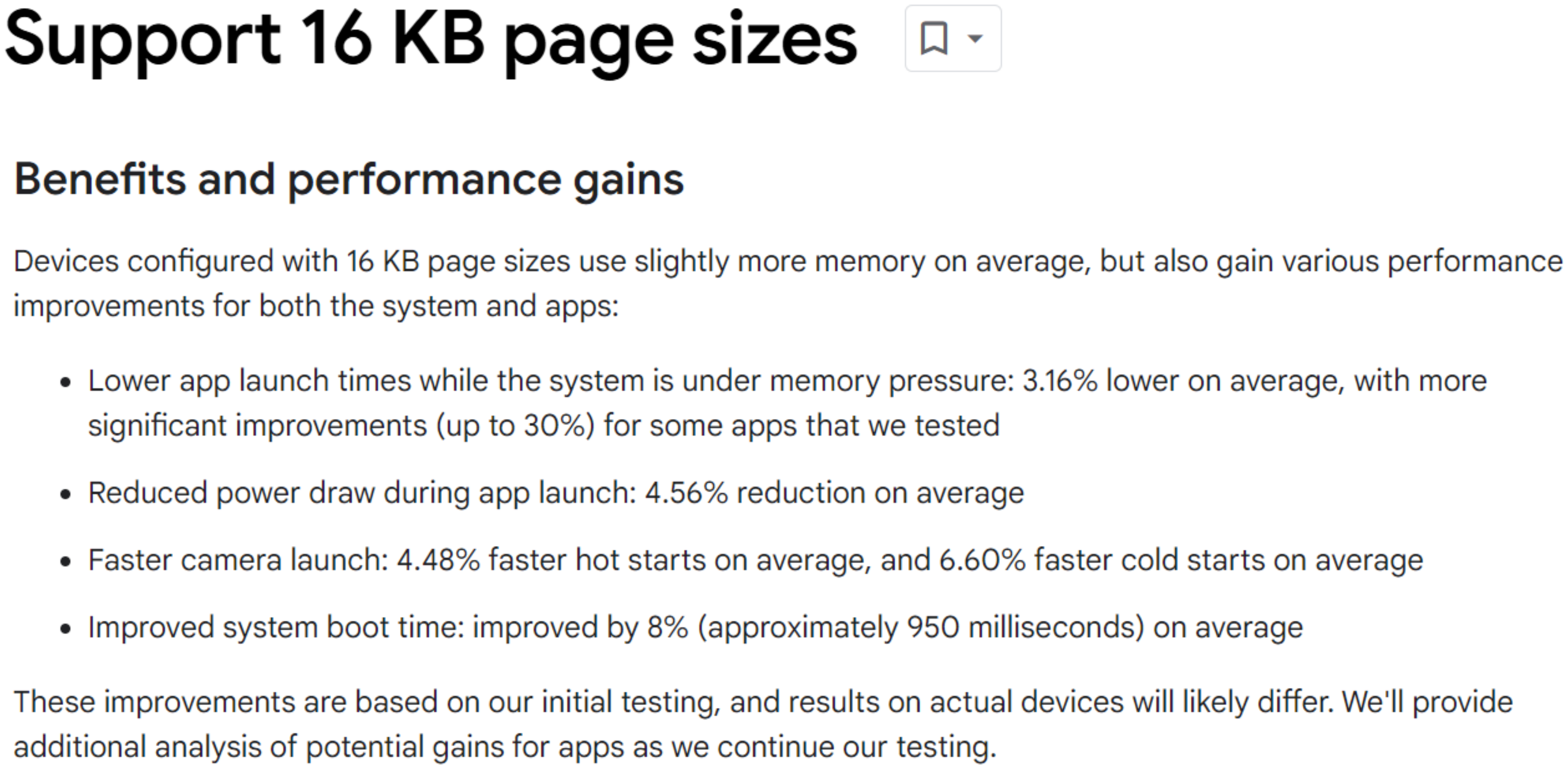


Source	Back-End Bound					
	Memory Bound					Core Bound
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
for (int i = 0; i < N; i++) {						
for (int j = 0; j < N; j++) {						
for (int k = 0; k < N; k++) {	0.0%	0.0%	0.0%	0.0%	0.0%	
result[i][j] += a[i][k] * b[k][j];	28.4%	0.0%	0.0%	0.0%	0.0%	100.0%
}						
}						
}						



Source	Back-End Bound					
	Memory Bound					Core Bound
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
for (int i = 0; i < N; i++) {						
for (int k = 0; k < N; k++) {	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
for (int j = 0; j < N; j++) {	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
result[i][j] += a[i][k] * b[k][j];	0.0%	0.0%	0.0%	0.0%	0.0%	
}						
}						
}						

Huge Page



Support 16 KB page sizes

Benefits and performance gains

Devices configured with 16 KB page sizes use slightly more memory on average, but also gain various performance improvements for both the system and apps:

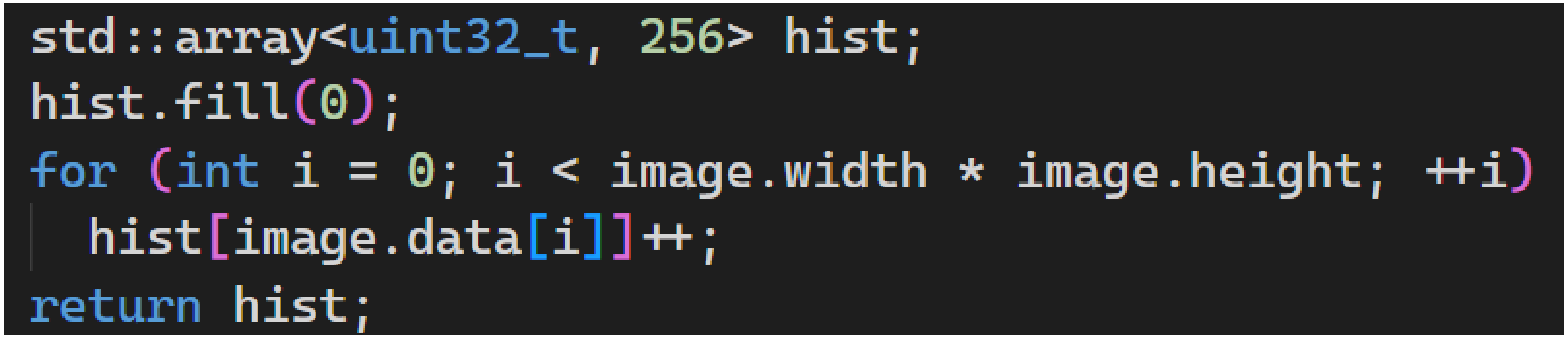
- Lower app launch times while the system is under memory pressure: 3.16% lower on average, with more significant improvements (up to 30%) for some apps that we tested
- Reduced power draw during app launch: 4.56% reduction on average
- Faster camera launch: 4.48% faster hot starts on average, and 6.60% faster cold starts on average
- Improved system boot time: improved by 8% (approximately 950 milliseconds) on average

These improvements are based on our initial testing, and results on actual devices will likely differ. We'll provide additional analysis of potential gains for apps as we continue our testing.

https://developer.android.com/guide/practices/page-sizes

Hardware Effects

Memory Order Violation



```
std::array<uint32_t, 256> hist;
hist.fill(0);
for (int i = 0; i < image.width * image.height; ++i)
  hist[image.data[i]]++;
return hist;
```

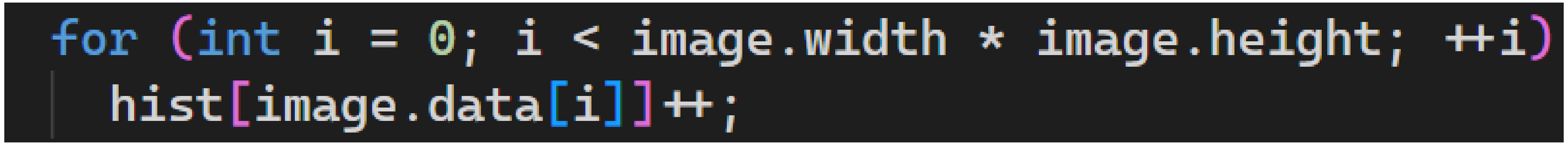
求灰度图的直方图

0xFF 0xFF 0xFF 0xFF 0xFF 0xFF …

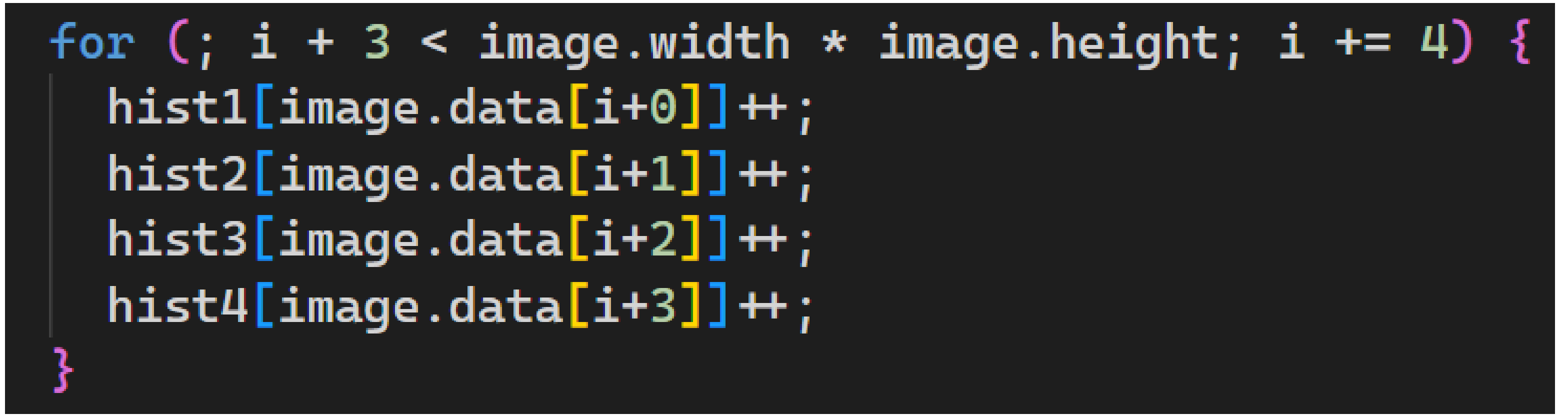
Then all updates to hist[0xFF] will be serialized.

Hardware Effects

Memory Order Violation



```
for (int i = 0; i < image.width * image.height; ++i)
  hist[image.data[i]]++;
```

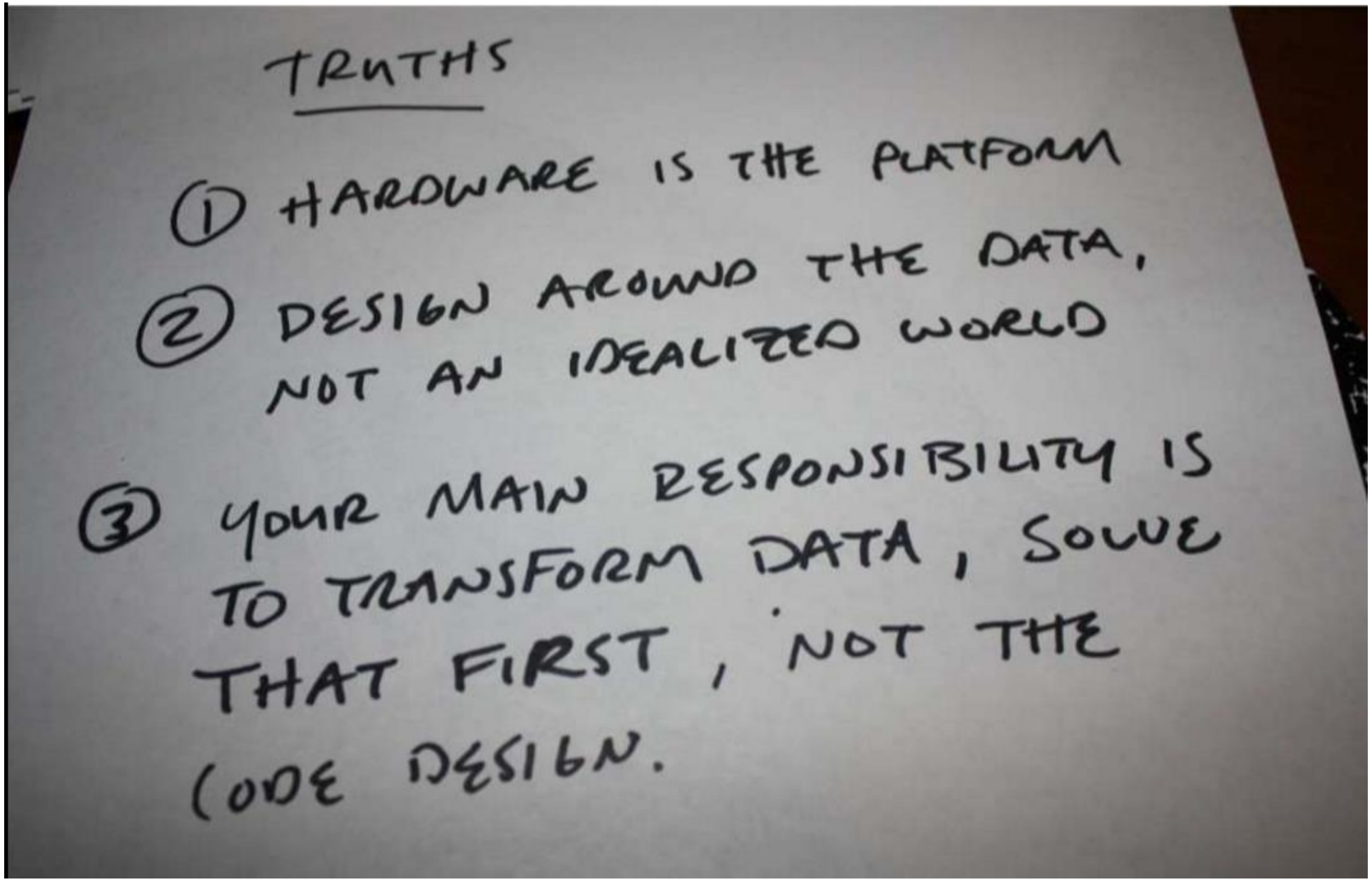


```
for (; i + 3 < image.width * image.height; i += 4) {
  hist1[image.data[i+0]]++;
  hist2[image.data[i+1]]++;
  hist3[image.data[i+2]]++;
  hist4[image.data[i+3]]++;
}
```

0-60% speed up

https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/memory_bound/mem_order_violation_1

Data Oriented Design



TRUTHS

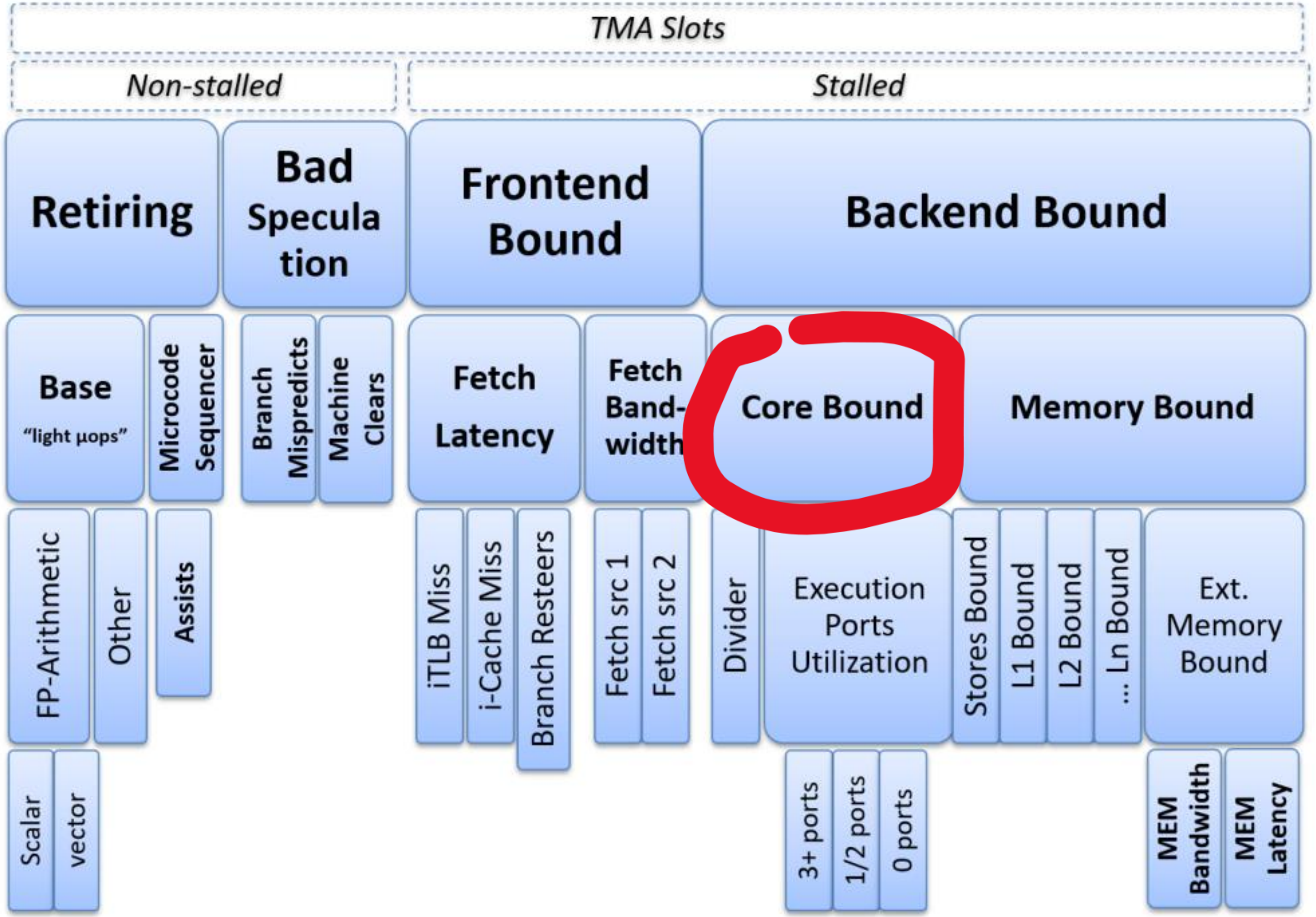
① HARDWARE IS THE PLATFORM

② DESIGN AROUND THE DATA, NOT AN IDEALIZED WORLD

③ YOUR MAIN RESPONSIBILITY IS TO TRANSFORM DATA, SOLVE THAT FIRST, NOT THE CODE DESIGN.

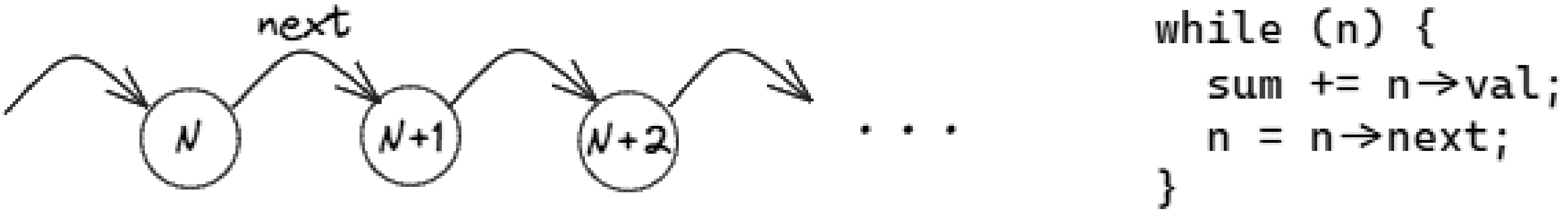
https://www.youtube.com/watch?v=rX0ItVEVjHc

Four Corners



Data Dependencies

Linked List



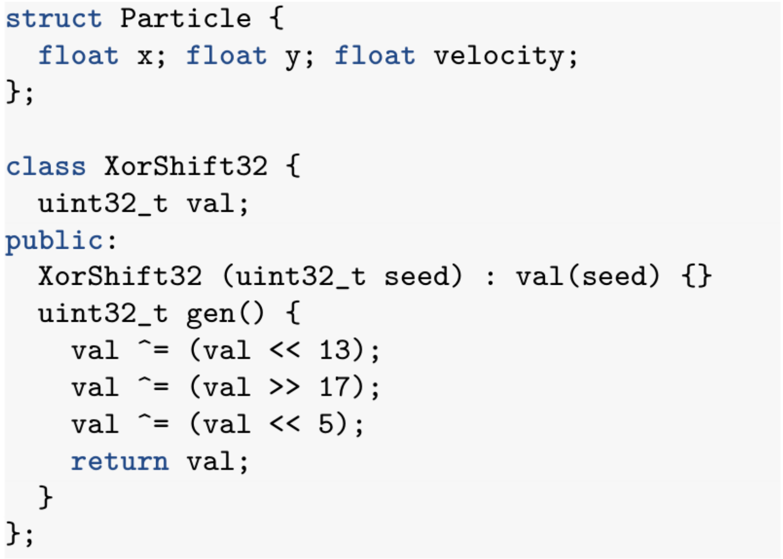
```
while (n) {
  sum += n->val;
  n = n->next;
}
```

Pointer chasing does not benefit from OOO execution and thus will run at the speed of an in-order CPU.

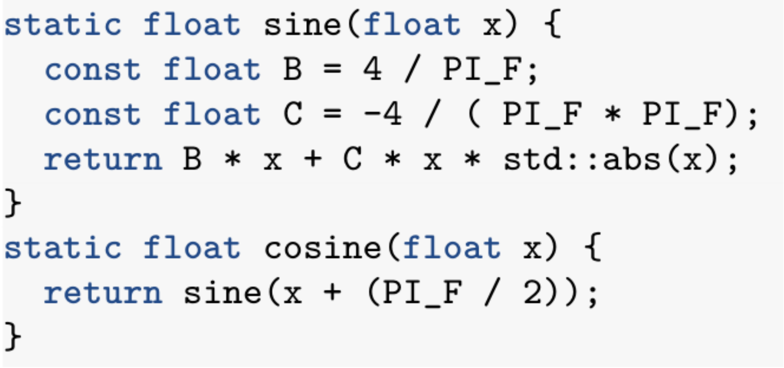
https://github.com/dendibakh/perf-book/blob/main/img/computation-opts/LinkedListChasing.png

Data Dependencies

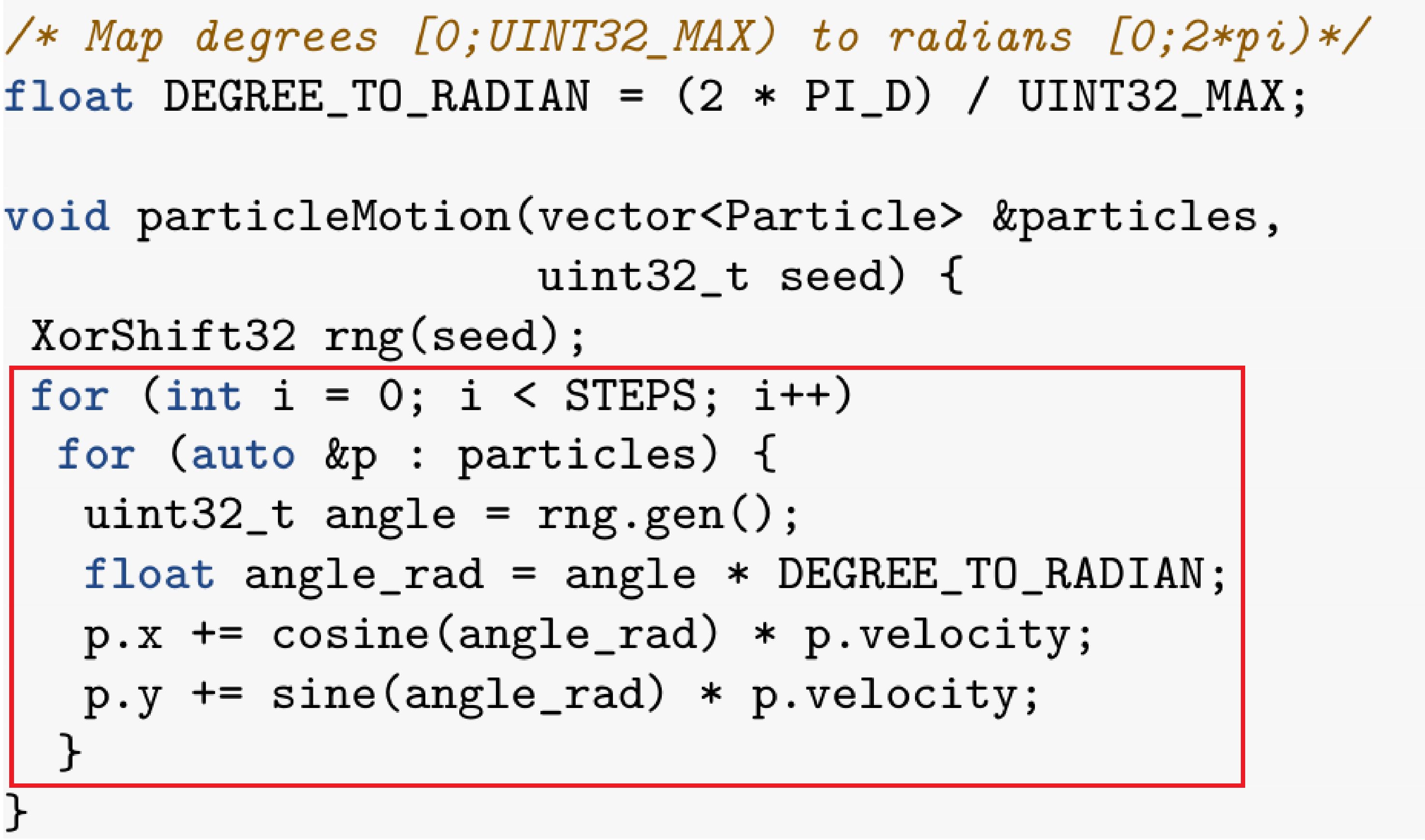
Simulate random particle movement



```cpp
struct Particle {
  float x; float y; float velocity;
};

class XorShift32 {
  uint32_t val;
public:
  XorShift32 (uint32_t seed) : val(seed) {}
  uint32_t gen() {
    val ^= (val << 13);
    val ^= (val >> 17);
    val ^= (val << 5);
    return val;
  }
};
```



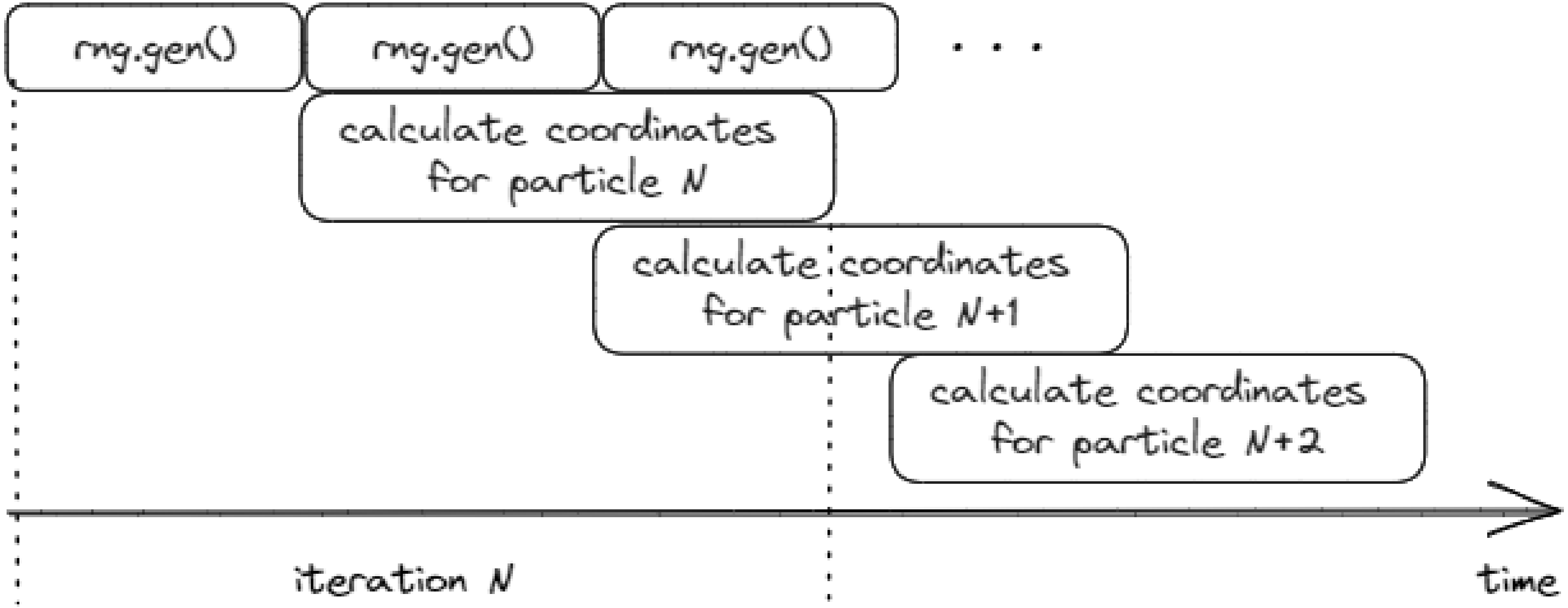
```cpp
static float sine(float x) {
  const float B = 4 / PI_F;
  const float C = -4 / ( PI_F * PI_F);
  return B * x + C * x * std::abs(x);
}
static float cosine(float x) {
  return sine(x + (PI_F / 2));
}
```



```cpp
/* Map degrees [0;UINT32_MAX) to radians [0;2*pi)*/
float DEGREE_TO_RADIAN = (2 * PI_D) / UINT32_MAX;

void particleMotion(vector<Particle> &particles,
                    uint32_t seed) {
 XorShift32 rng(seed);
 for (int i = 0; i < STEPS; i++)
  for (auto &p : particles) {
   uint32_t angle = rng.gen();
   float angle_rad = angle * DEGREE_TO_RADIAN;
   p.x += cosine(angle_rad) * p.velocity;
   p.y += sine(angle_rad) * p.velocity;
  }
}
```

Data Dependencies

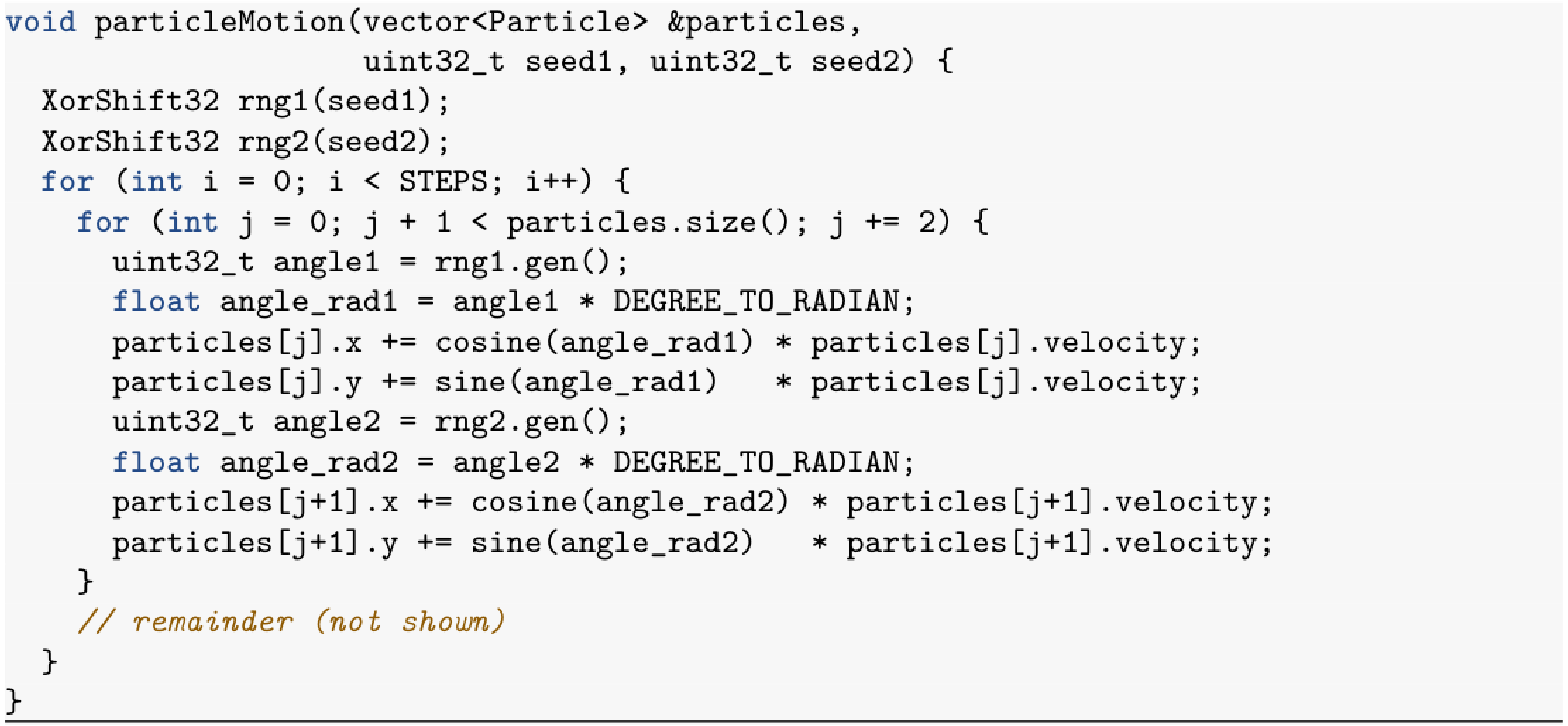
Visualization of dependent execution


rng.gen()
rng.gen()
rng.gen()
. . .
calculate coordinates for particle N
calculate coordinates for particle N+1
calculate coordinates for particle N+2
iteration N
time

https://github.com/dendibakh/perf-book/blob/main/img/computation-opts/DepChain.png

Data Dependencies

Solution



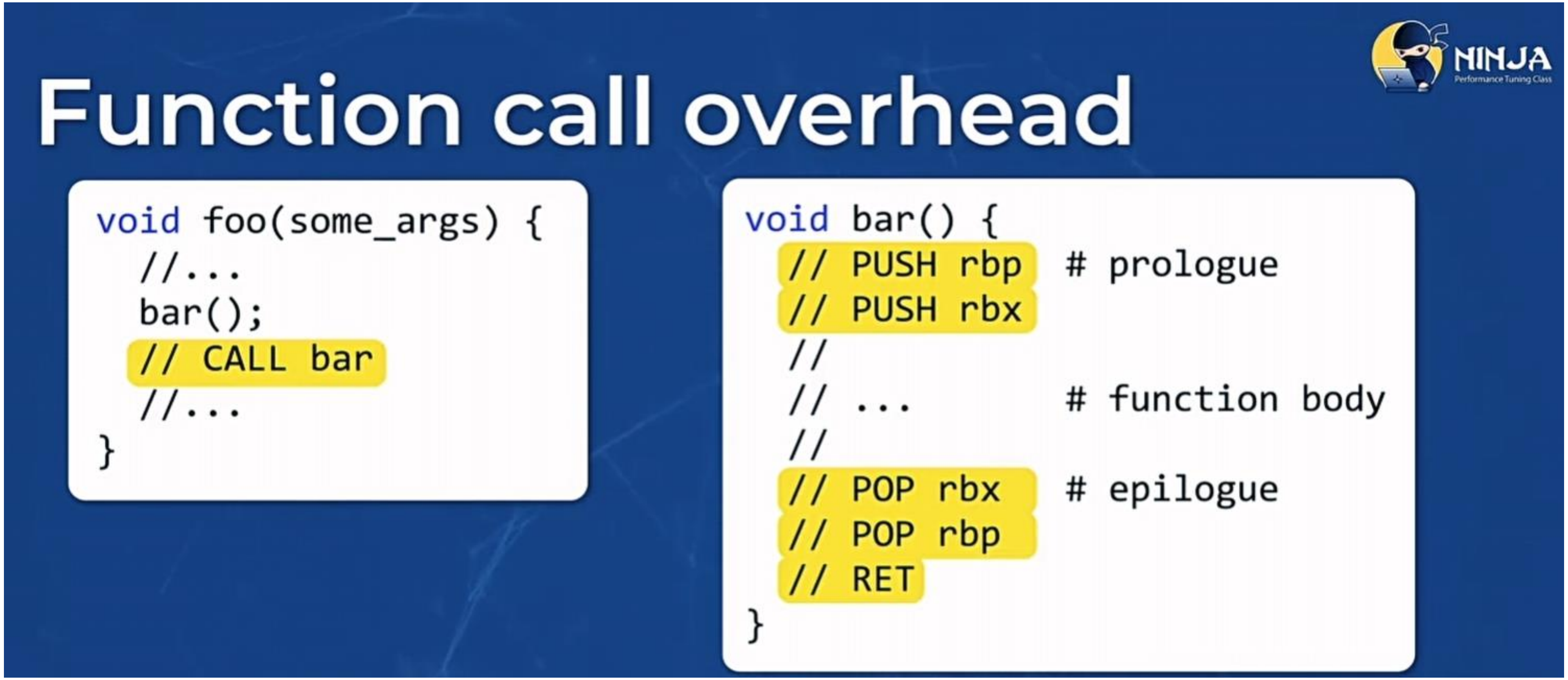
```
void particleMotion(vector<Particle> &particles,
                    uint32_t seed1, uint32_t seed2) {
  XorShift32 rng1(seed1);
  XorShift32 rng2(seed2);
  for (int i = 0; i < STEPS; i++) {
    for (int j = 0; j + 1 < particles.size(); j += 2) {
      uint32_t angle1 = rng1.gen();
      float angle_rad1 = angle1 * DEGREE_TO_RADIAN;
      particles[j].x += cosine(angle_rad1) * particles[j].velocity;
      particles[j].y += sine(angle_rad1)   * particles[j].velocity;
      uint32_t angle2 = rng2.gen();
      float angle_rad2 = angle2 * DEGREE_TO_RADIAN;
      particles[j+1].x += cosine(angle_rad2) * particles[j+1].velocity;
      particles[j+1].y += sine(angle_rad2)   * particles[j+1].velocity;
    }
    // remainder (not shown)
  }
}
```

Once you do this transformation, the compiler starts autovectorizing the body of the loop.

55% speed up

https://github.com/dendibakh/perf-ninja/tree/golden/labs/core_bound/dep_chains_2

Inline



NINJA Performance Tuning Class

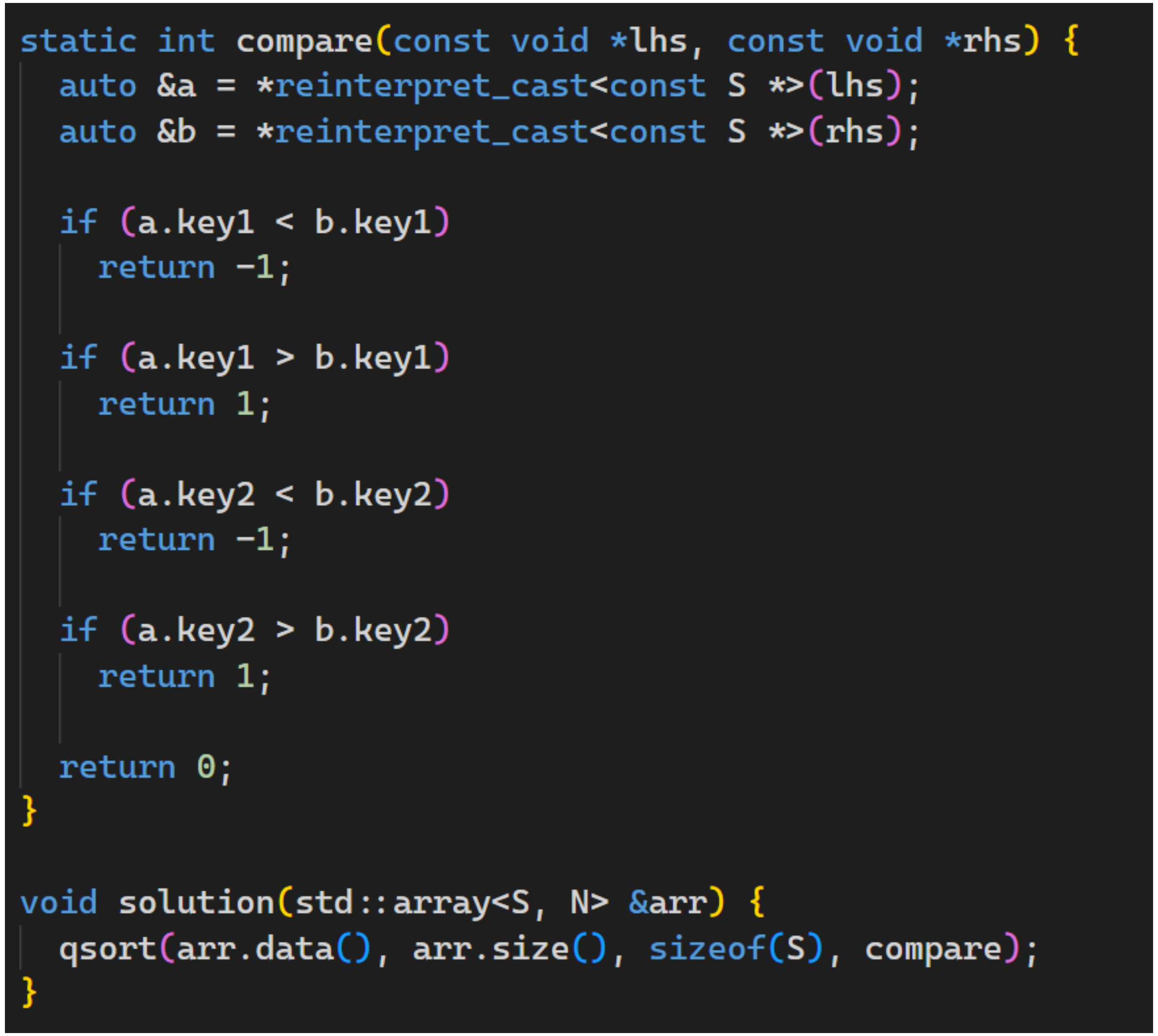
Function call overhead

```
void foo(some_args) {
  //...
  bar();
  // CALL bar
  //...
}
```

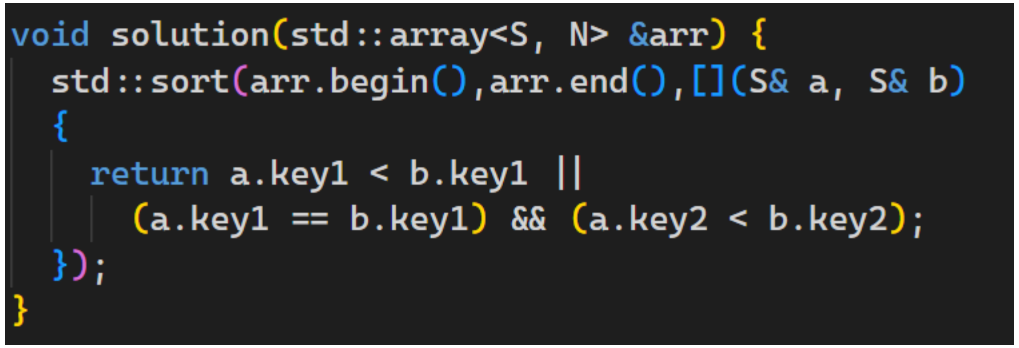
```
void bar() {
  // PUSH rbp   # prologue
  // PUSH rbx
  //
  // ...        # function body
  //
  // POP rbx    # epilogue
  // POP rbp
  // RET
}
```

https://www.youtube.com/watch?v=fp1_e3rjZQs&list=PLRWO2AL1QAV6bJAU2kgB4xfodGID43Y5d&index=14

Inline



```
static int compare(const void *lhs, const void *rhs) {
  auto &a = *reinterpret_cast<const S *>(lhs);
  auto &b = *reinterpret_cast<const S *>(rhs);

  if (a.key1 < b.key1)
    return -1;

  if (a.key1 > b.key1)
    return 1;

  if (a.key2 < b.key2)
    return -1;

  if (a.key2 > b.key2)
    return 1;

  return 0;
}

void solution(std::array<S, N> &arr) {
  qsort(arr.data(), arr.size(), sizeof(S), compare);
}
```



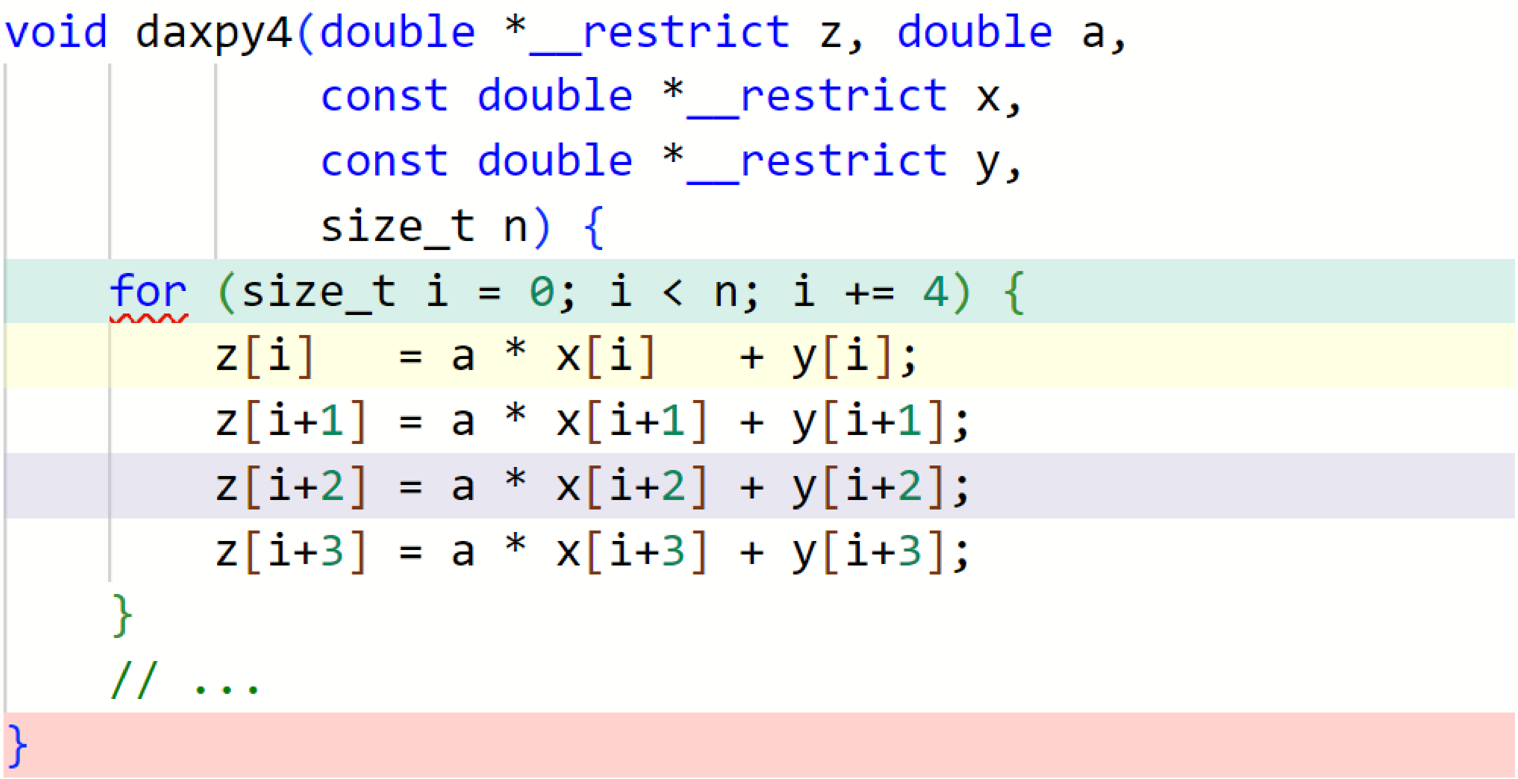
```
void solution(std::array<S, N> &arr) {
  std::sort(arr.begin(),arr.end(),[](S& a, S& b)
  {
    return a.key1 < b.key1 ||
      (a.key1 == b.key1) && (a.key2 < b.key2);
  });
}
```

47% speed up

https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/core_bound/function_inlining_1

Auto vectorization: Rely on Compiler

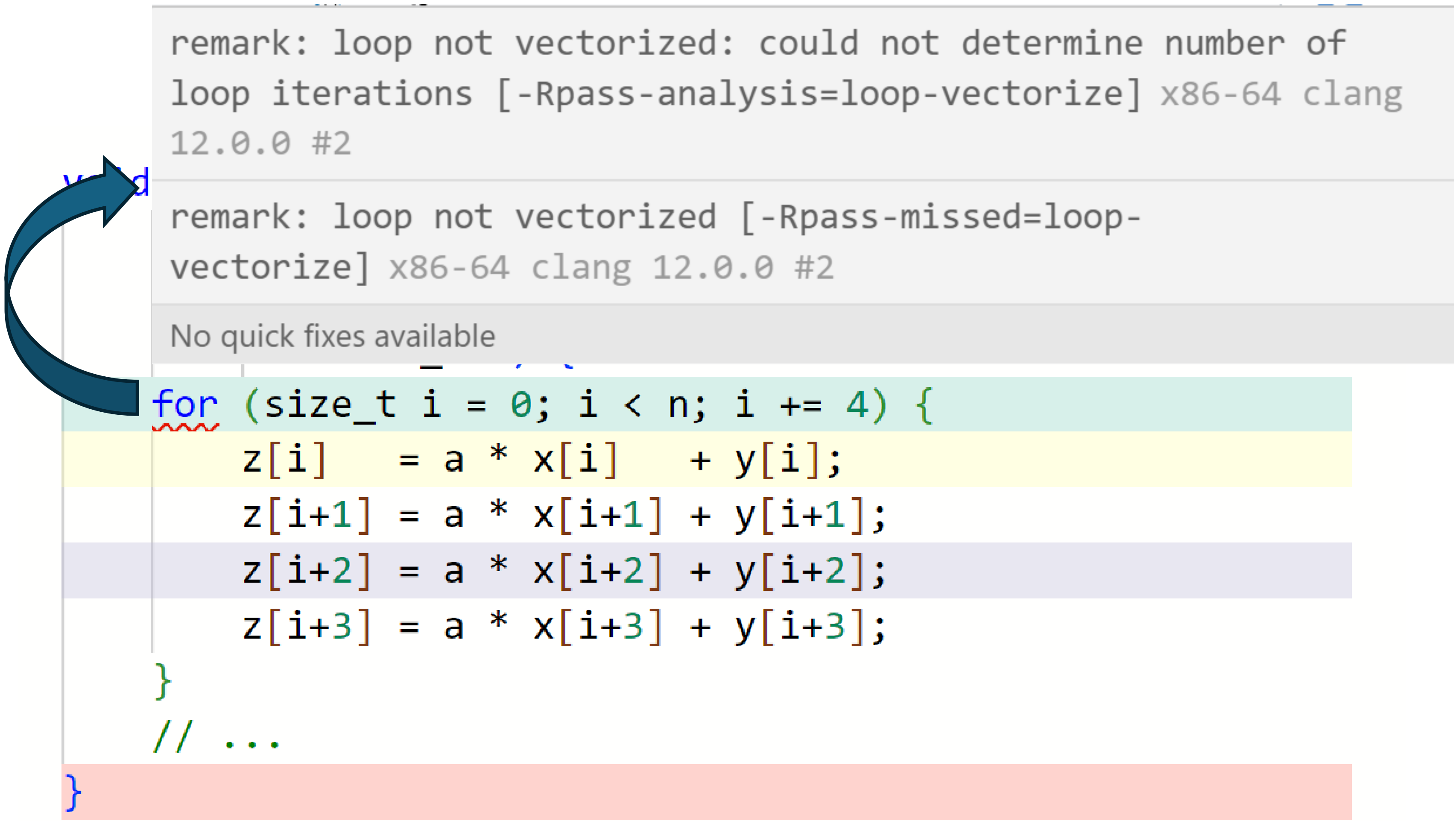
Does the following loop vectorize?



```
void daxpy4(double *__restrict z, double a,
            const double *__restrict x,
            const double *__restrict y,
            size_t n) {
    for (size_t i = 0; i < n; i += 4) {
        z[i]   = a * x[i]   + y[i];
        z[i+1] = a * x[i+1] + y[i+1];
        z[i+2] = a * x[i+2] + y[i+2];
        z[i+3] = a * x[i+3] + y[i+3];
    }
    // ...
}
```

What Compilers Cannot Do

自动向量化失败



```
remark: loop not vectorized: could not determine number of
loop iterations [-Rpass-analysis=loop-vectorize] x86-64 clang
12.0.0 #2
```

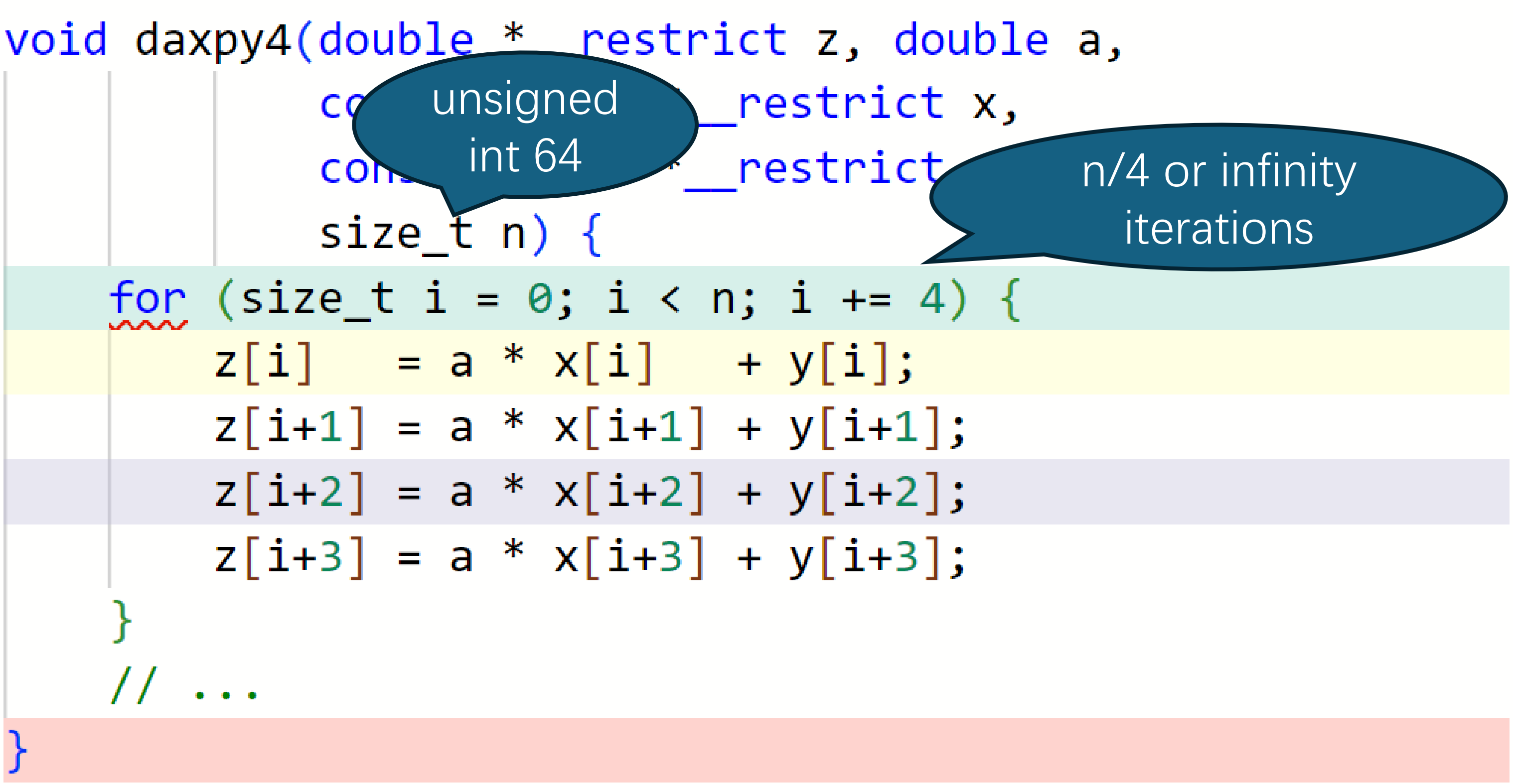
```
remark: loop not vectorized [-Rpass-missed=loop-
vectorize] x86-64 clang 12.0.0 #2
```

```
for (size_t i = 0; i < n; i += 4) {
    z[i]   = a * x[i]   + y[i];
    z[i+1] = a * x[i+1] + y[i+1];
    z[i+2] = a * x[i+2] + y[i+2];
    z[i+3] = a * x[i+3] + y[i+3];
}
// ...
}
```

clang12 -O3 -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize

What Compilers Cannot Do

编译器的限制(bug)

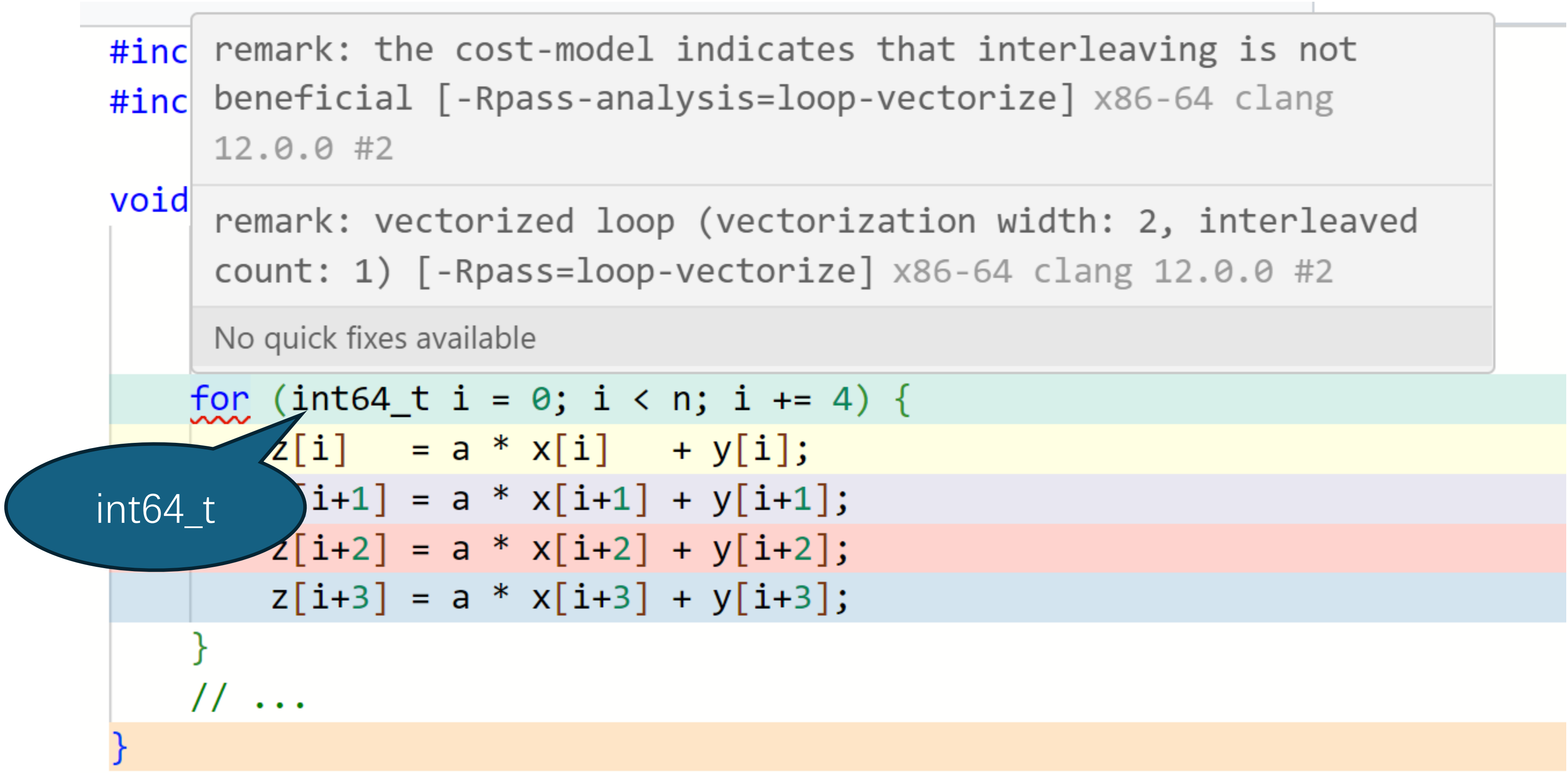


```
void daxpy4(double *  restrict z, double a,
            co               _restrict x,
            con             *__restrict
            size_t n) {
    for (size_t i = 0; i < n; i += 4) {
        z[i]   = a * x[i]   + y[i];
        z[i+1] = a * x[i+1] + y[i+1];
        z[i+2] = a * x[i+2] + y[i+2];
        z[i+3] = a * x[i+3] + y[i+3];
    }
    // ...
}
```

PROBLEM: In C, the behavior of unsigned-integer overflow is defined to wrap around.

What Compilers Cannot Do

自动向量化



```
remark: the cost-model indicates that interleaving is not
beneficial [-Rpass-analysis=loop-vectorize] x86-64 clang
12.0.0 #2

remark: vectorized loop (vectorization width: 2, interleaved
count: 1) [-Rpass=loop-vectorize] x86-64 clang 12.0.0 #2
```

No quick fixes available

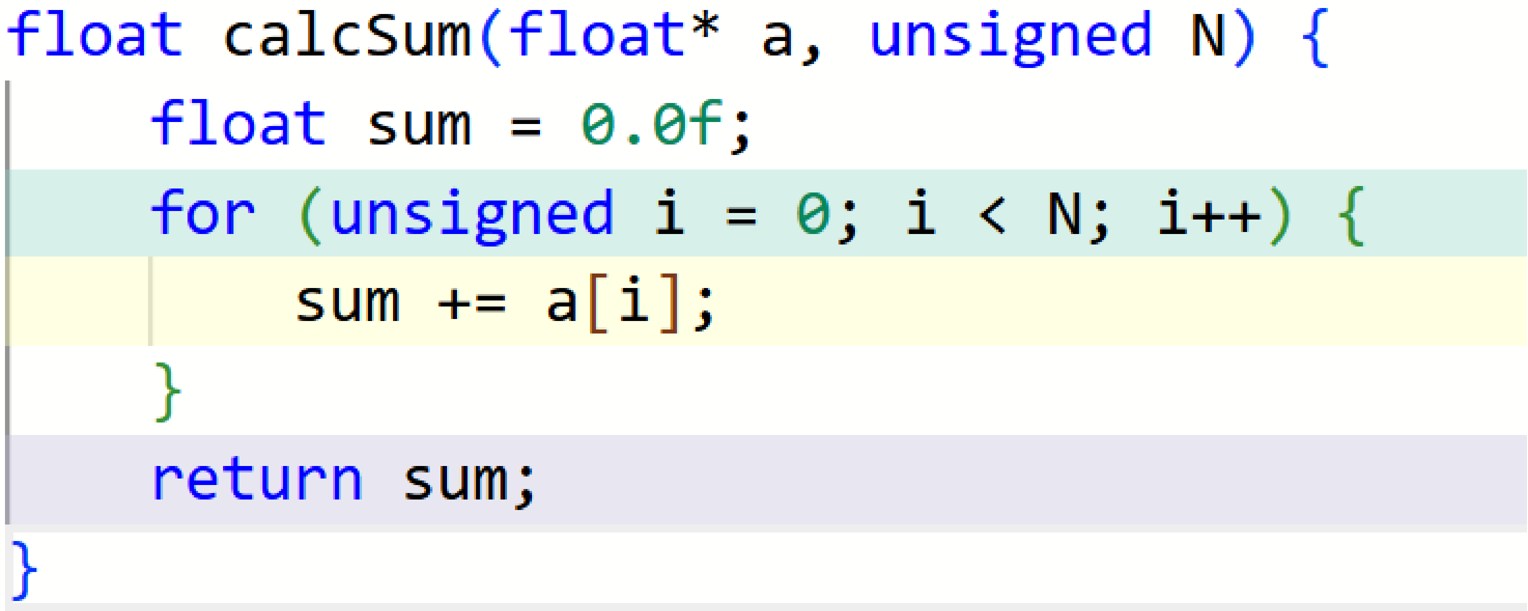
```
#inc
#inc

void

    for (int64_t i = 0; i < n; i += 4) {
        z[i]   = a * x[i]   + y[i];
         i+1] = a * x[i+1] + y[i+1];
        z[i+2] = a * x[i+2] + y[i+2];
        z[i+3] = a * x[i+3] + y[i+3];
    }
    // ...
}
```

SOLUTION: Use signed integer types, signed-integer overflow has undefined behavior.

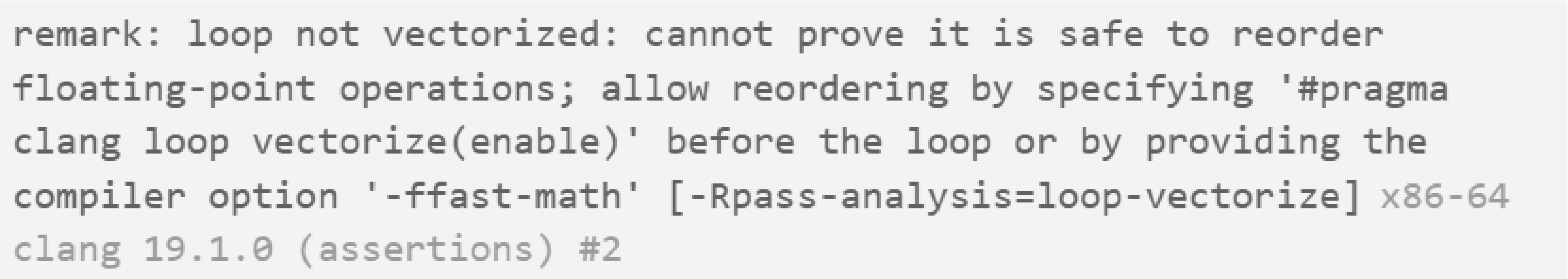
Or upgrade clang

What Compilers Cannot Do

float



```
float calcSum(float* a, unsigned N) {
    float sum = 0.0f;
    for (unsigned i = 0; i < N; i++) {
        sum += a[i];
    }
    return sum;
}
```



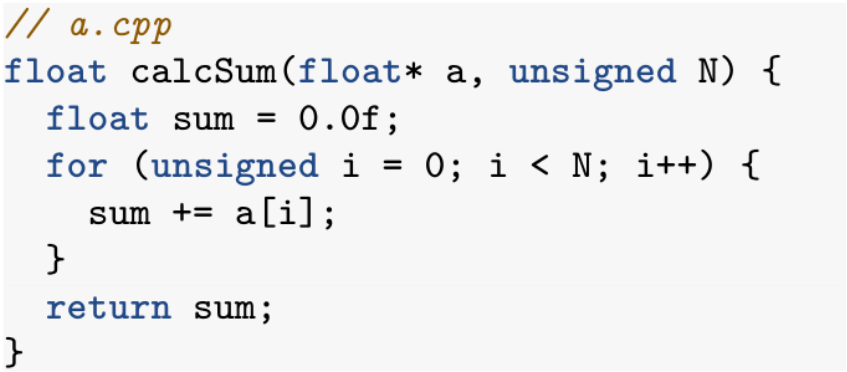
```
remark: loop not vectorized: cannot prove it is safe to reorder
floating-point operations; allow reordering by specifying '#pragma
clang loop vectorize(enable)' before the loop or by providing the
compiler option '-ffast-math' [-Rpass-analysis=loop-vectorize] x86-64
clang 19.1.0 (assertions) #2
```

SOLUTION:

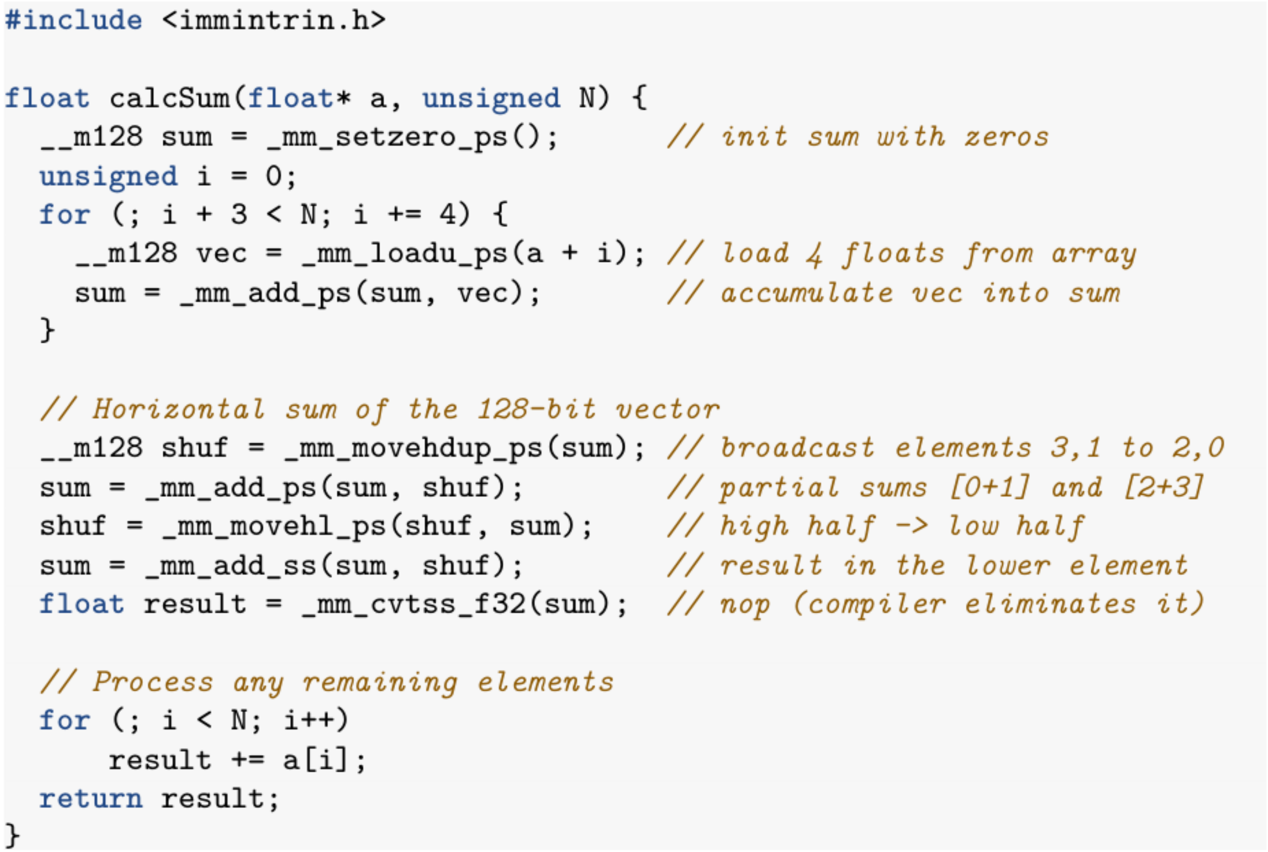
-ffast-math

Compiler Intrinsics

When the compiler fails

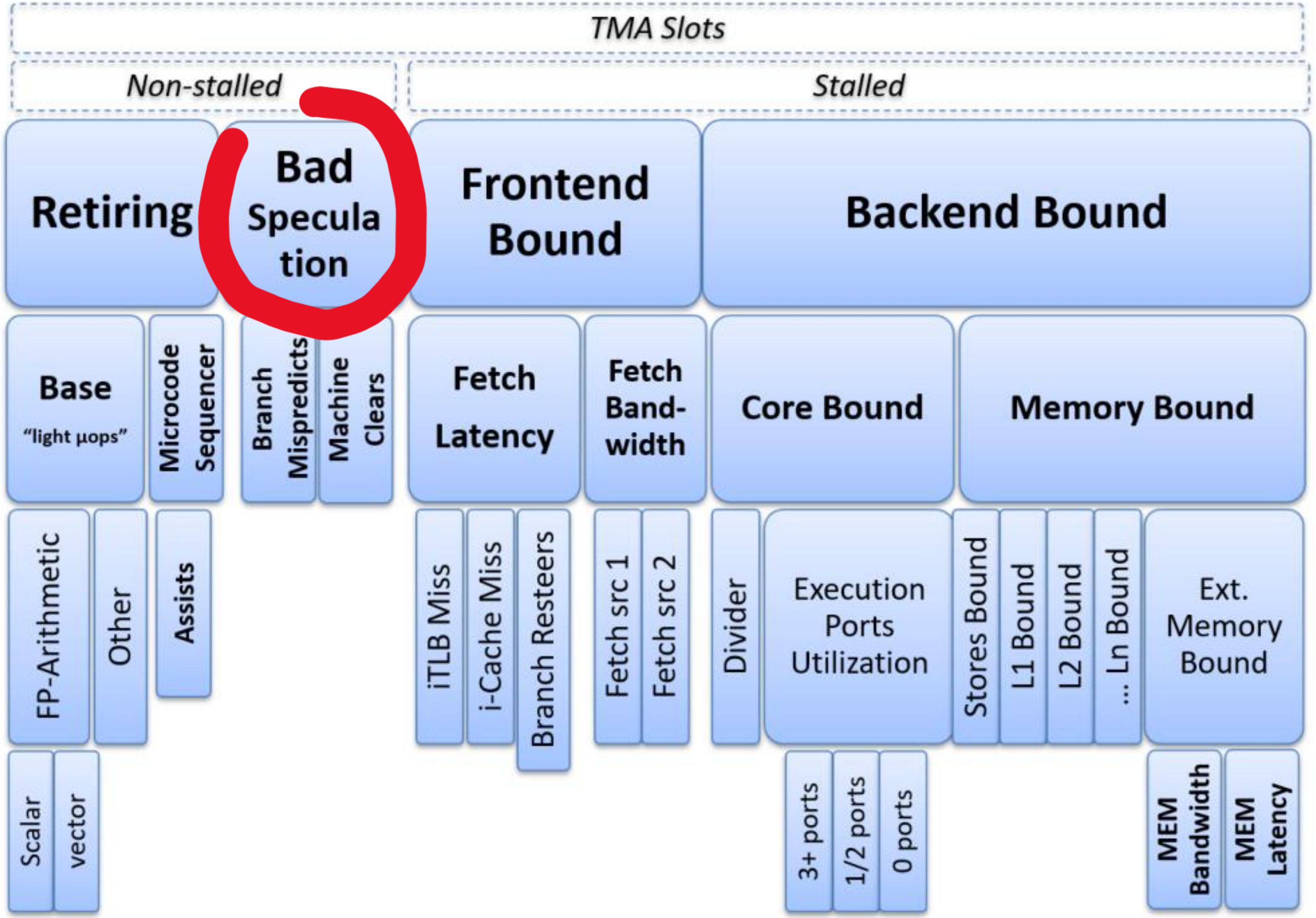


```
// a.cpp
float calcSum(float* a, unsigned N) {
  float sum = 0.0f;
  for (unsigned i = 0; i < N; i++) {
    sum += a[i];
  }
  return sum;
}
```



```
#include <immintrin.h>

float calcSum(float* a, unsigned N) {
  __m128 sum = _mm_setzero_ps();       // init sum with zeros
  unsigned i = 0;
  for (; i + 3 < N; i += 4) {
    __m128 vec = _mm_loadu_ps(a + i); // load 4 floats from array
    sum = _mm_add_ps(sum, vec);       // accumulate vec into sum
  }

  // Horizontal sum of the 128-bit vector
  __m128 shuf = _mm_movehdup_ps(sum); // broadcast elements 3,1 to 2,0
  sum = _mm_add_ps(sum, shuf);         // partial sums [0+1] and [2+3]
  shuf = _mm_movehl_ps(shuf, sum);     // high half -> low half
  sum = _mm_add_ss(sum, shuf);         // result in the lower element
  float result = _mm_cvtss_f32(sum);   // nop (compiler eliminates it)

  // Process any remaining elements
  for (; i < N; i++)
      result += a[i];
  return result;
}
```

Four Corners



TMA Slots

Non-stalled

Stalled

Retiring

Bad Speculation

Frontend Bound

Backend Bound

Base

"light μops"

Microcode Sequencer

Branch Mispredicts

Machine Clears

Fetch Latency

Fetch Bandwidth

Core Bound

Memory Bound

FP-Arithmetic

Other

Assists

iTLB Miss

i-Cache Miss

Branch Resteers

Fetch src 1

Fetch src 2

Divider

Execution Ports Utilization

Stores Bound

L1 Bound

L2 Bound

... Ln Bound

Ext. Memory Bound

Scalar

vector

3+ ports

1/2 ports

0 ports

MEM Bandwidth

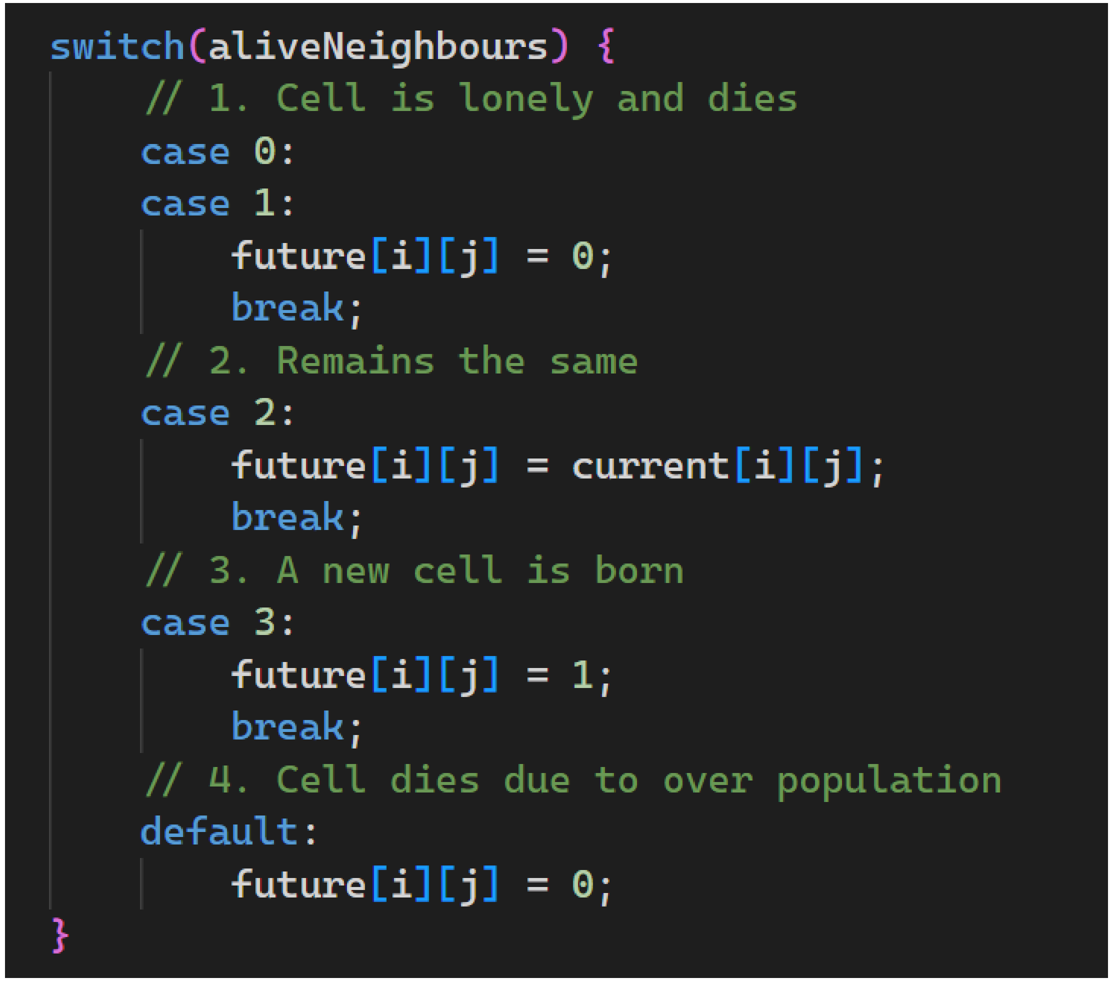
MEM Latency

生命游戏

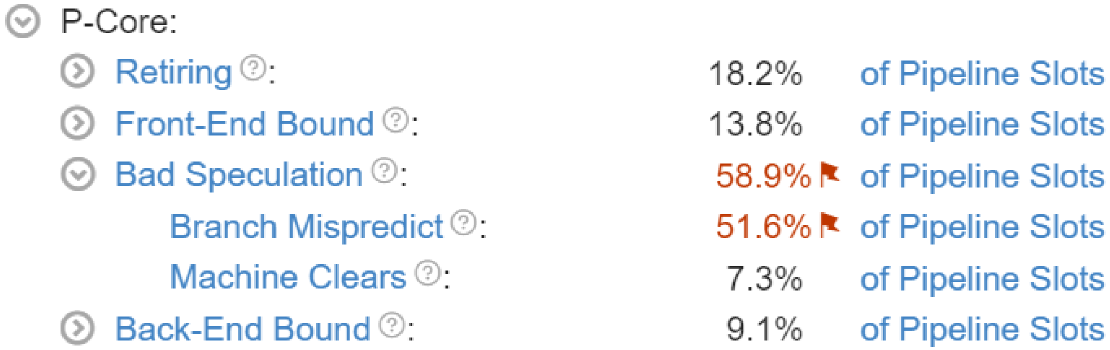
规则：

1. 每个细胞有两种状态 - 存活或死亡，每个细胞与以自身为中心的周围八格细胞产生互动
2. 当前细胞为存活状态时，当周围的存活细胞低于2个时（不包含2个），该细胞变成死亡状态。
3. 当前细胞为存活状态时，当周围有2个或3个存活细胞时，该细胞保持原样。
4. 当前细胞为存活状态时，当周围有超过3个存活细胞时，该细胞变成死亡状态。
5. 当前细胞为死亡状态时，当周围有3个存活细胞时，该细胞变成存活状态。

生命游戏



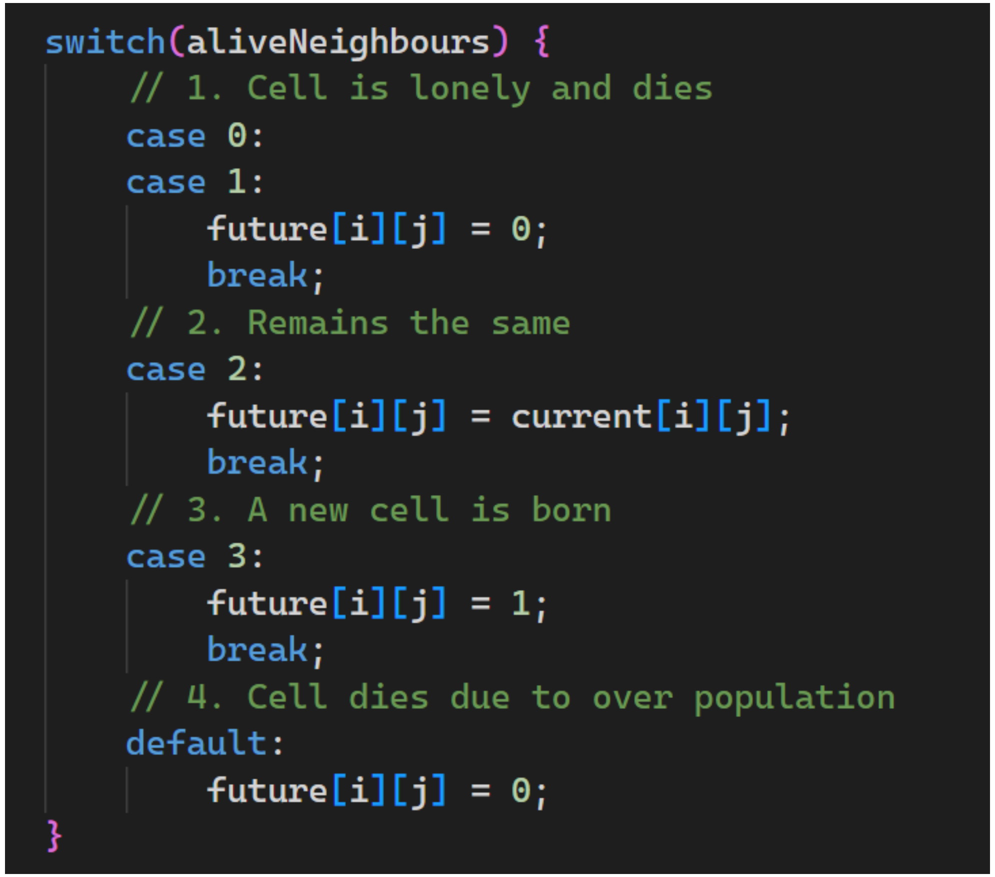
```
switch(aliveNeighbours) {
    // 1. Cell is lonely and dies
    case 0:
    case 1:
        future[i][j] = 0;
        break;
    // 2. Remains the same
    case 2:
        future[i][j] = current[i][j];
        break;
    // 3. A new cell is born
    case 3:
        future[i][j] = 1;
        break;
    // 4. Cell dies due to over population
    default:
        future[i][j] = 0;
}
```



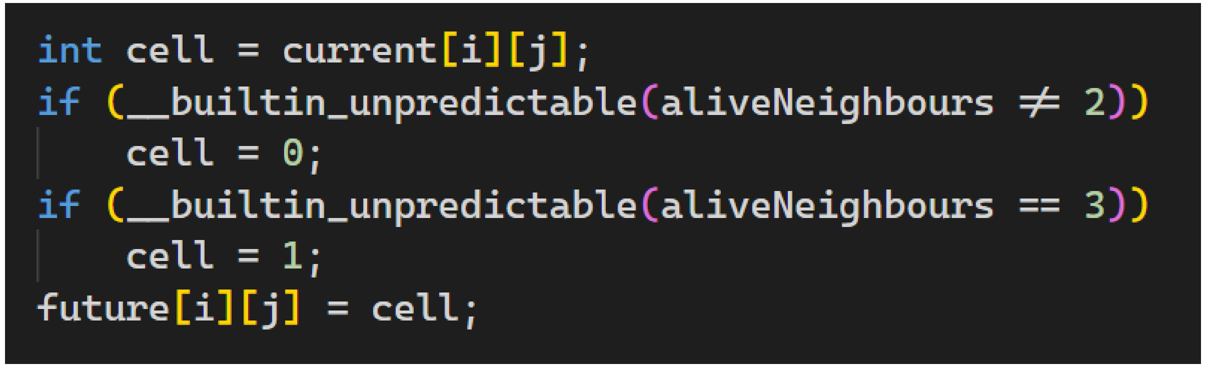
P-Core:

Retiring ⓘ:	18.2%	of Pipeline Slots
Front-End Bound ⓘ:	13.8%	of Pipeline Slots
Bad Speculation ⓘ:	58.9% ⚑	of Pipeline Slots
Branch Mispredict ⓘ:	51.6% ⚑	of Pipeline Slots
Machine Clears ⓘ:	7.3%	of Pipeline Slots
Back-End Bound ⓘ:	9.1%	of Pipeline Slots

Branch => cmove



```
switch(aliveNeighbours) {
    // 1. Cell is lonely and dies
    case 0:
    case 1:
        future[i][j] = 0;
        break;
    // 2. Remains the same
    case 2:
        future[i][j] = current[i][j];
        break;
    // 3. A new cell is born
    case 3:
        future[i][j] = 1;
        break;
    // 4. Cell dies due to over population
    default:
        future[i][j] = 0;
}
```



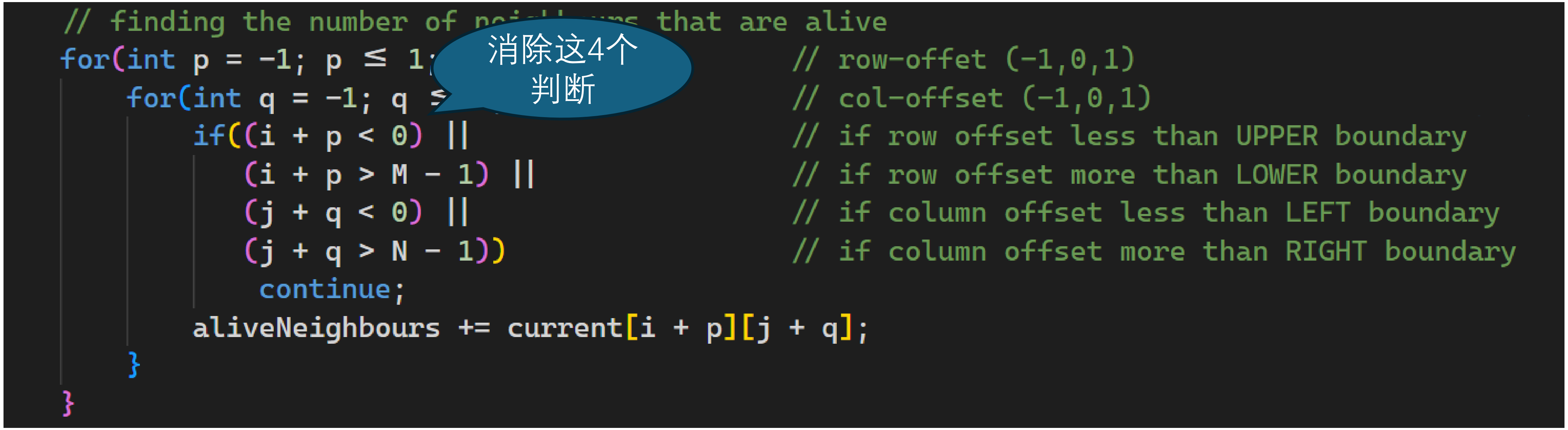
```
int cell = current[i][j];
if (__builtin_unpredictable(aliveNeighbours != 2))
    cell = 0;
if (__builtin_unpredictable(aliveNeighbours == 3))
    cell = 1;
future[i][j] = cell;
```

49% speed up

https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/bad_speculation/branches_to_cmov_1

另一个热点

边界判断



```
    // finding the number of neighbours that are alive
    for(int p = -1; p ≤ 1;                              // row-offet (-1,0,1)
        for(int q = -1; q ≤                             // col-offset (-1,0,1)
            if((i + p < 0) ||                           // if row offset less than UPPER boundary
               (i + p > M - 1) ||                       // if row offset more than LOWER boundary
               (j + q < 0) ||                           // if column offset less than LEFT boundary
               (j + q > N - 1))                         // if column offset more than RIGHT boundary
                continue;
            aliveNeighbours += current[i + p][j + q];
        }
    }
```

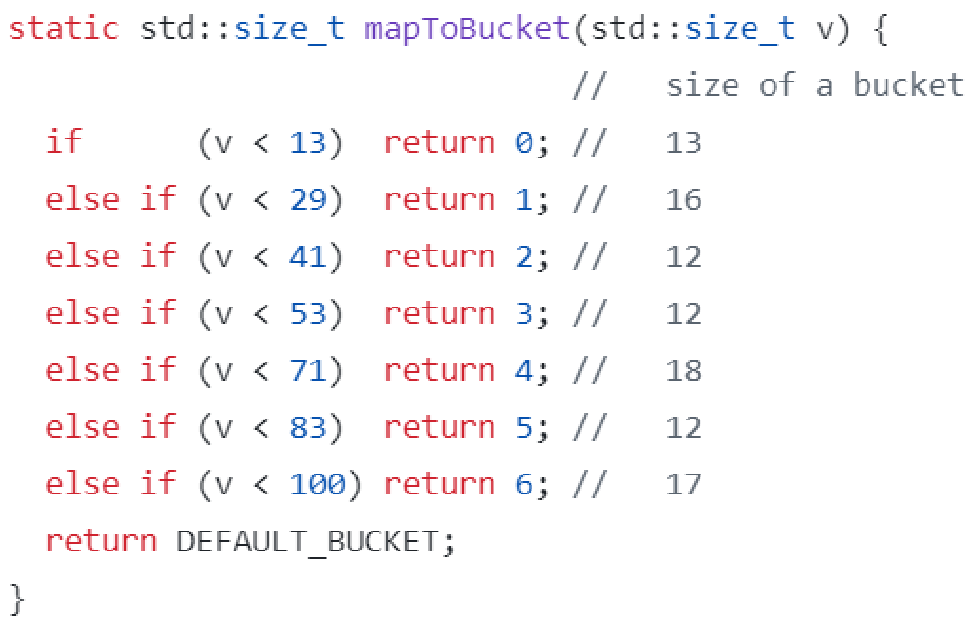
消除这4个判断

Solution：上下左右的边界各往外扩一格即可

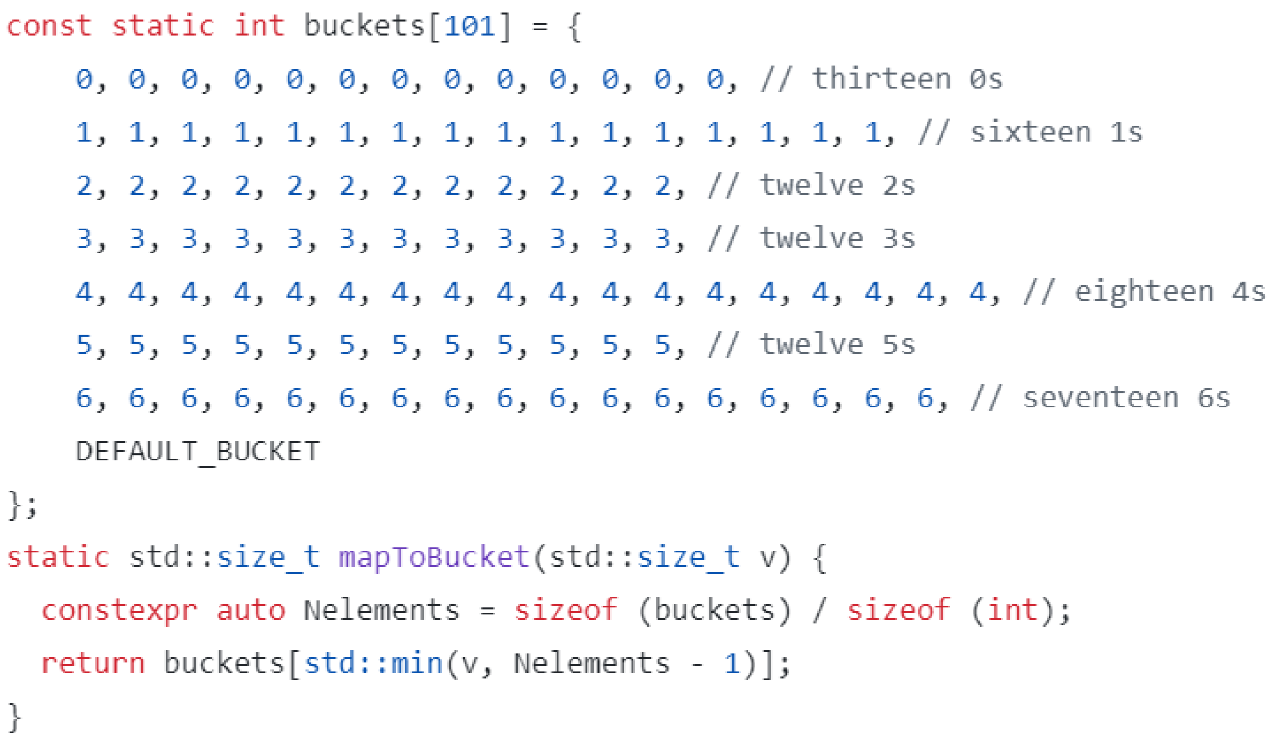
93% speed up

https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/bad_speculation/branches_to_cmov_1

Lookup table



```
static std::size_t mapToBucket(std::size_t v) {
                                              //   size of a bucket
  if      (v < 13)  return 0; //   13
  else if (v < 29)  return 1; //   16
  else if (v < 41)  return 2; //   12
  else if (v < 53)  return 3; //   12
  else if (v < 71)  return 4; //   18
  else if (v < 83)  return 5; //   12
  else if (v < 100) return 6; //   17
  return DEFAULT_BUCKET;
}
```



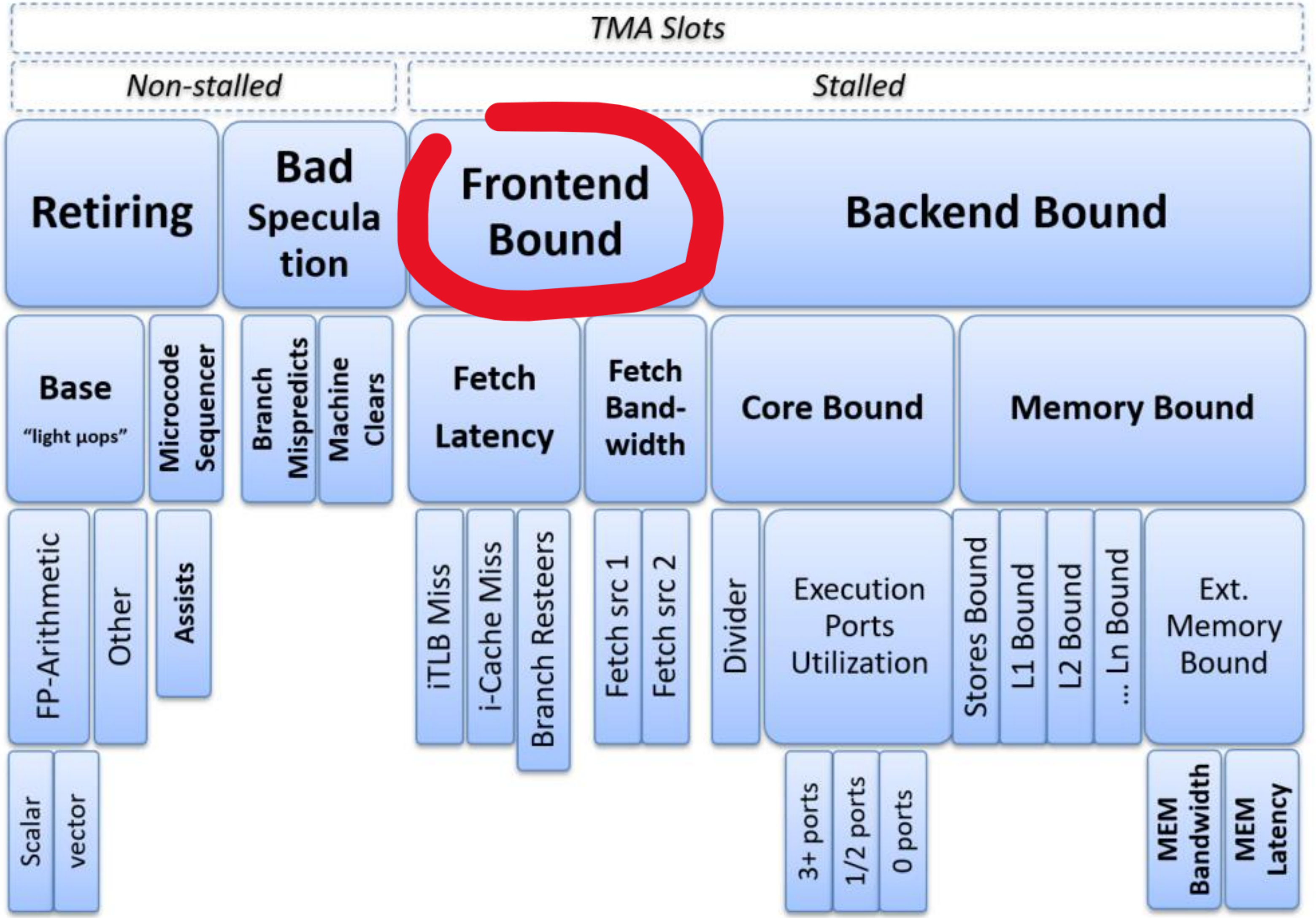
```
const static int buckets[101] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // thirteen 0s
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sixteen 1s
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // twelve 2s
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, // twelve 3s
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, // eighteen 4s
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, // twelve 5s
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, // seventeen 6s
    DEFAULT_BUCKET
};
static std::size_t mapToBucket(std::size_t v) {
  constexpr auto Nelements = sizeof (buckets) / sizeof (int);
  return buckets[std::min(v, Nelements - 1)];
}
```

88% speed up

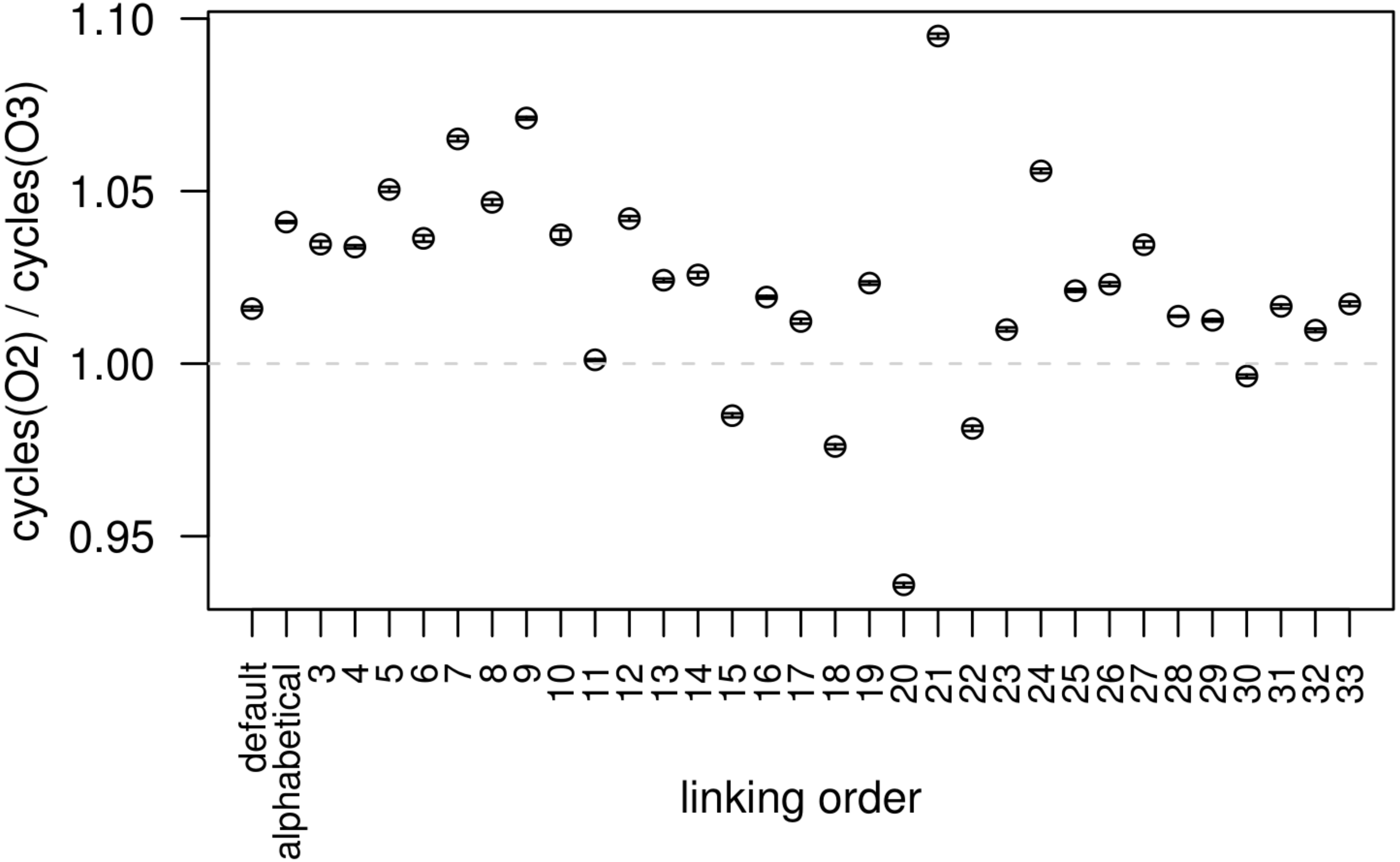
https://github.com/jsjtxietian/perf-ninja-solution/tree/master/labs/bad_speculation/lookup_tables_1

Note: always measure, 右边的增加了cache的压力

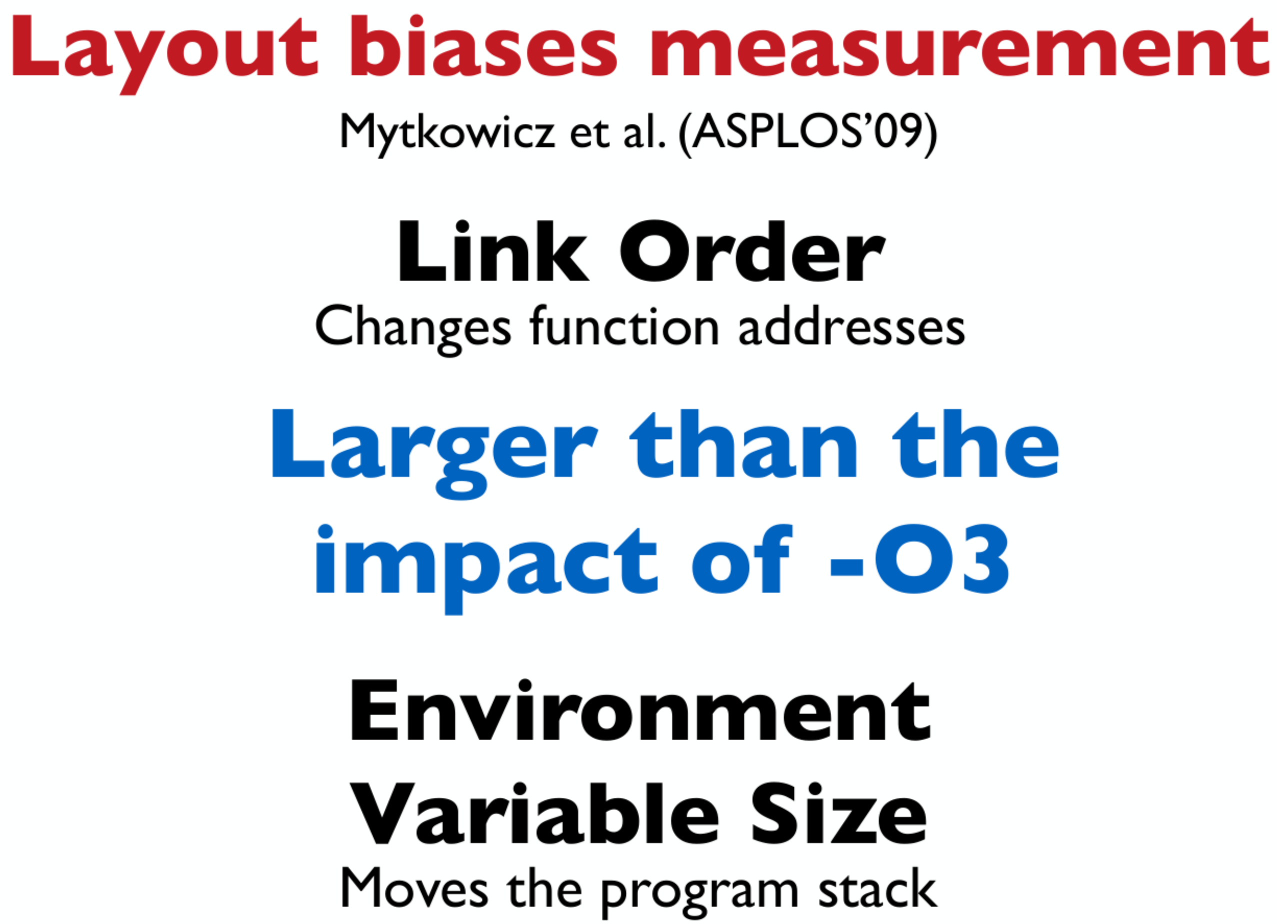
Four Corners



Machine Code Layout Causes Noise



Machine Code Layout Causes Noise



Layout biases measurement

Mytkowicz et al. (ASPLOS'09)

Link Order

Changes function addresses

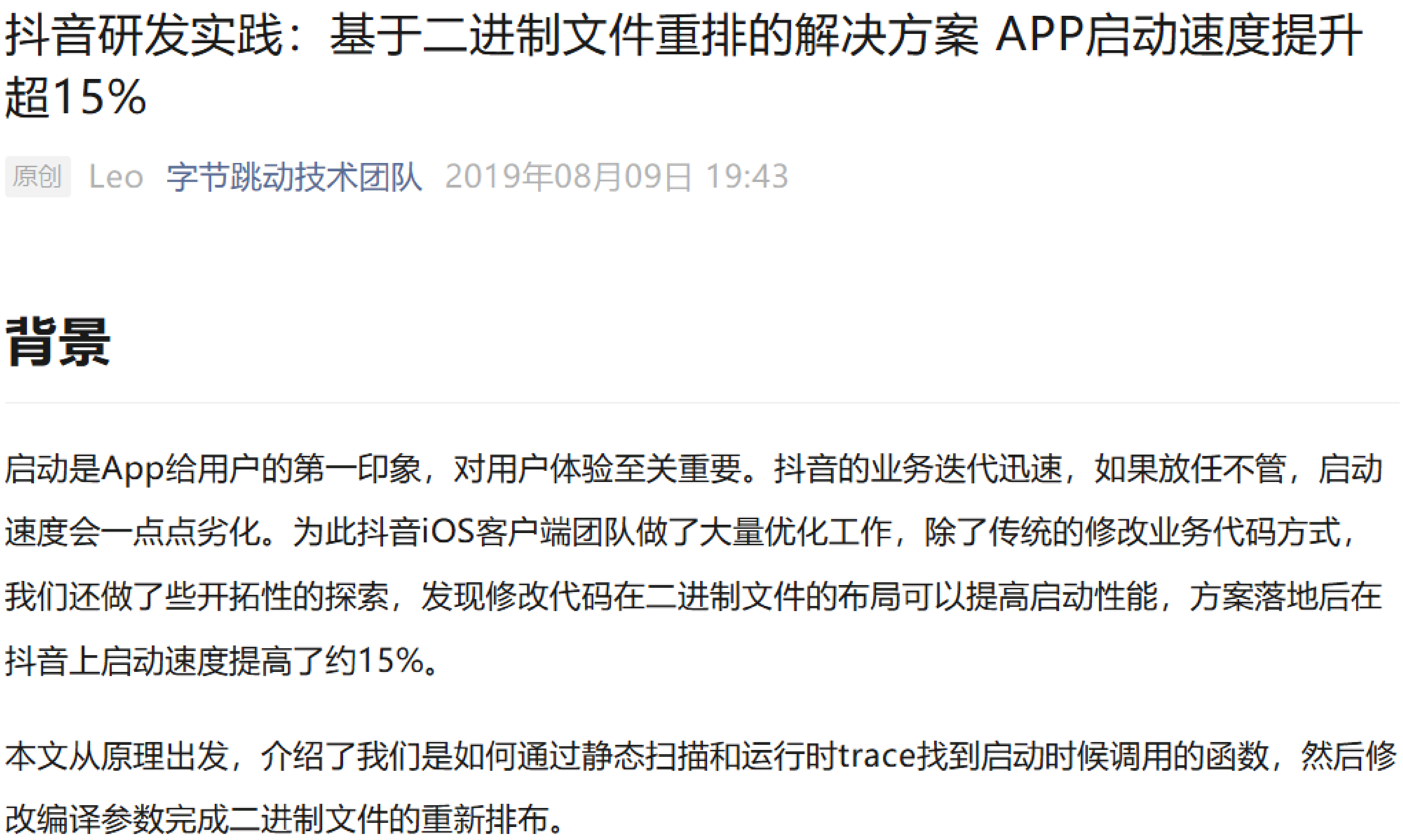
Larger than the impact of -O3

Environment Variable Size

Moves the program stack

https://www.youtube.com/watch?v=7g1Acy5eGbE

Machine Code Layout



抖音研发实践：基于二进制文件重排的解决方案 APP启动速度提升超15%

原创 Leo 字节跳动技术团队 2019年08月09日 19:43

背景

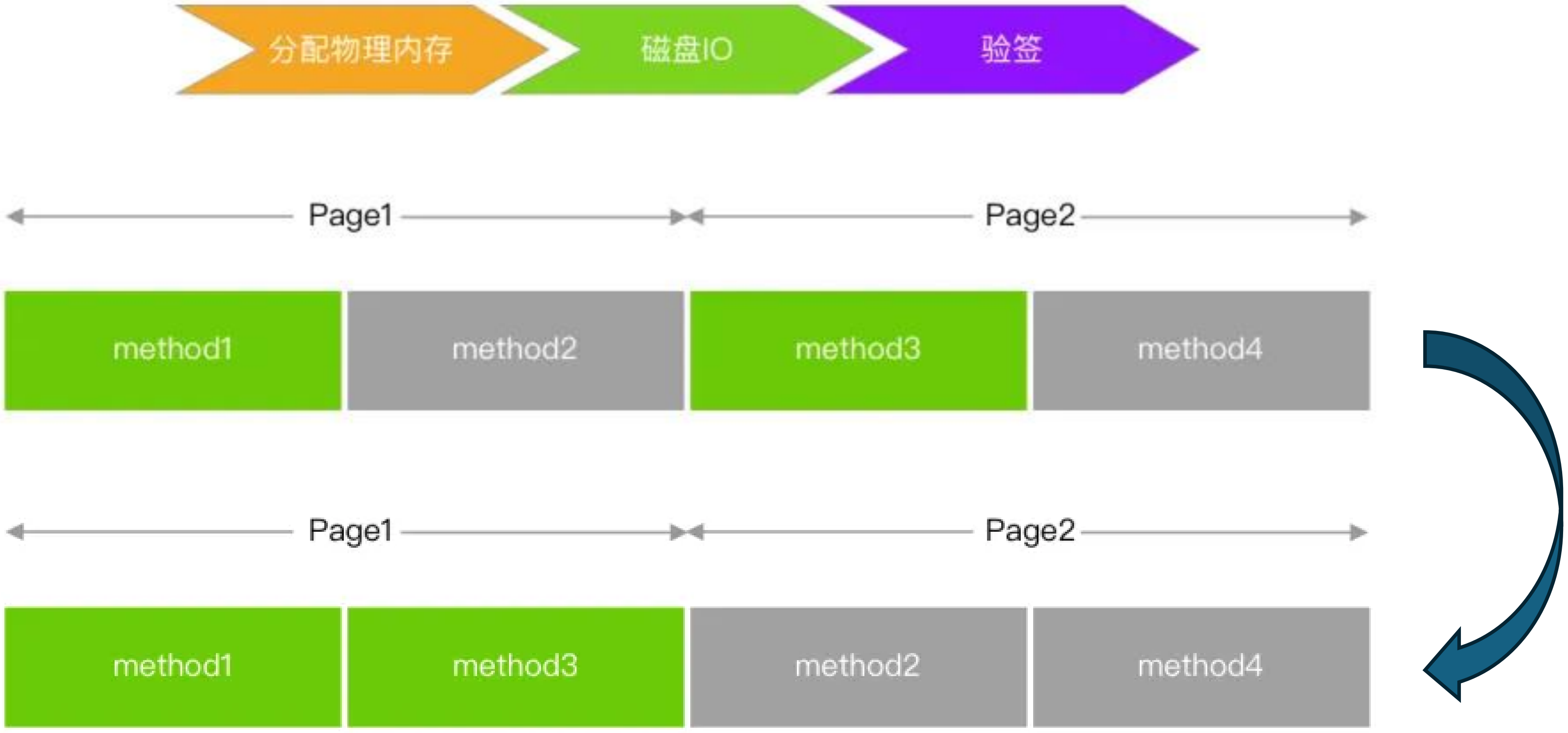
启动是App给用户的第一印象，对用户体验至关重要。抖音的业务迭代迅速，如果放任不管，启动速度会一点点劣化。为此抖音iOS客户端团队做了大量优化工作，除了传统的修改业务代码方式，我们还做了些开拓性的探索，发现修改代码在二进制文件的布局可以提高启动性能，方案落地后在抖音上启动速度提高了约15%。

本文从原理出发，介绍了我们是如何通过静态扫描和运行时trace找到启动时候调用的函数，然后修改编译参数完成二进制文件的重新排布。

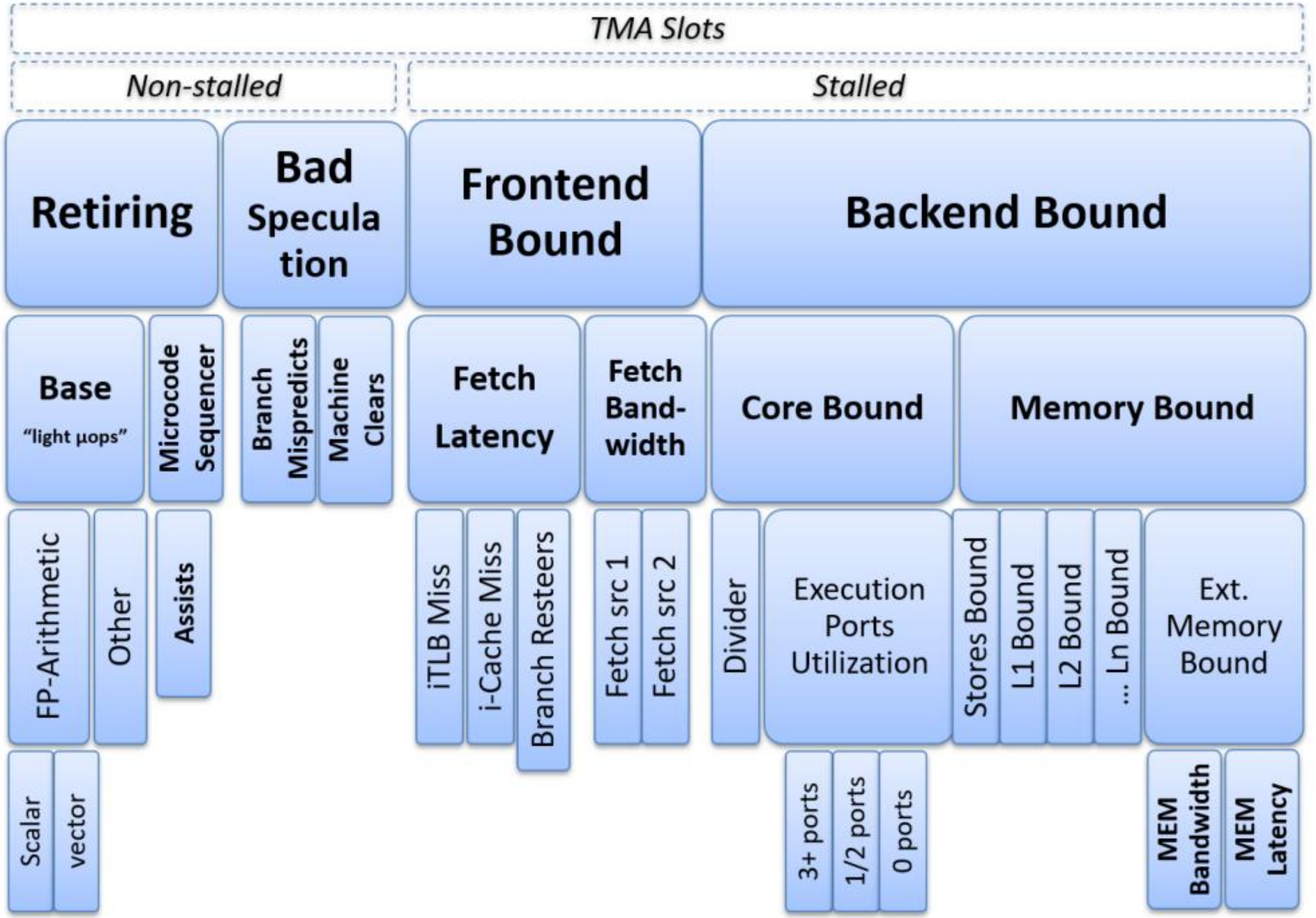
抖音研发实践：基于二进制文件重排的解决方案 APP启动速度提升超15%

Machine Code Layout

优化一个Page Fault，启动速度提升0.6~0.8ms



抖音研发实践：基于二进制文件重排的解决方案 APP启动速度提升超15%



TMA Slots

Non-stalled

Stalled

Retiring

Bad Speculation

Frontend Bound

Backend Bound

Base

"light μops"

Microcode Sequencer

Branch Mispredicts

Machine Clears

Fetch Latency

Fetch Bandwidth

Core Bound

Memory Bound

FP-Arithmetic

Other

Assists

iTLB Miss

i-Cache Miss

Branch Resteers

Fetch src 1

Fetch src 2

Divider

Execution Ports Utilization

Stores Bound

L1 Bound

L2 Bound

... Ln Bound

Ext. Memory Bound

Scalar

vector

3+ ports

1/2 ports

0 ports

MEM Bandwidth

MEM Latency

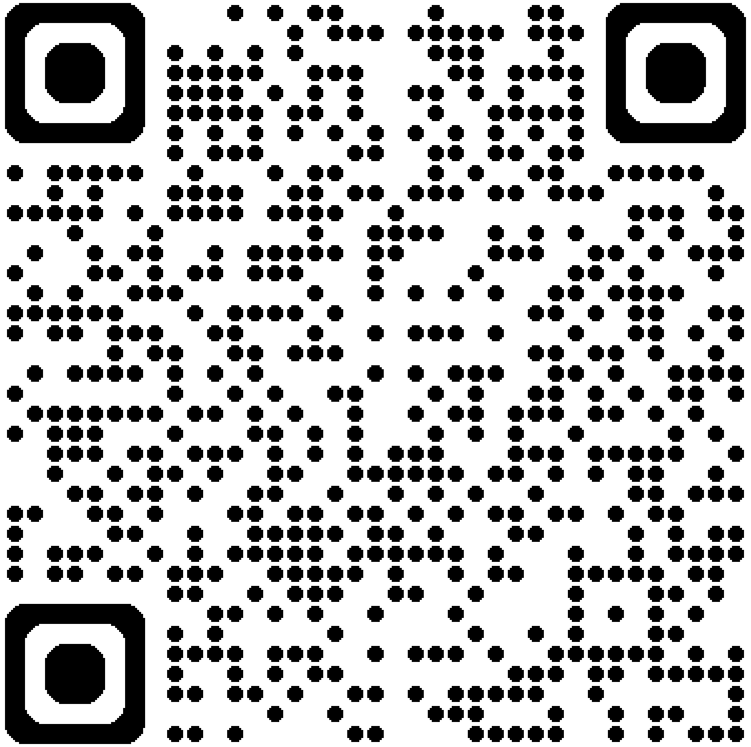
Final

Always Measure!

1. Mental model can never be as accurate as the actual microarchitecture design of a CPU. Always measure!
2. When measuring performance, understand the underlying technical reasons for the performance results .

More

Benchmark



Perf