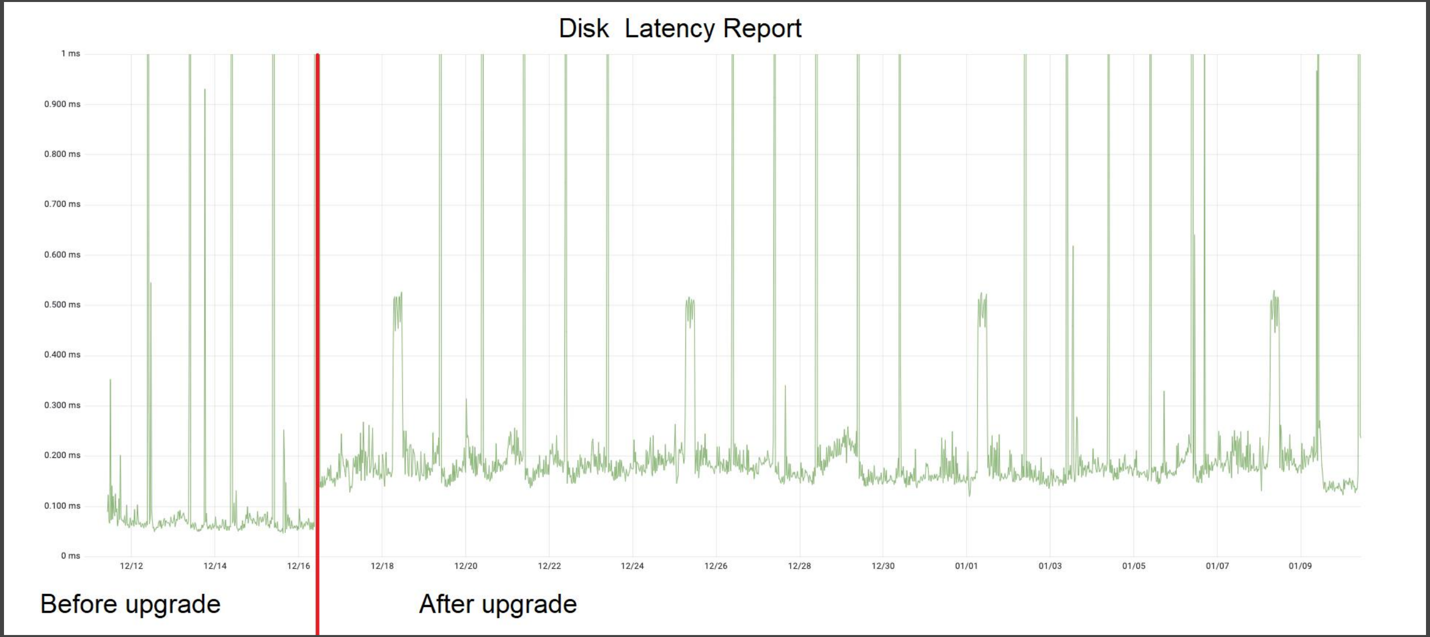
Logs Told Us It Was Kernel
It Felt Like Kernel
It Had To Be Kernel
It Wasn’t Kernel

Valery Sigalov, Bloomberg

Logs Told Us It Was Kernel



- The disk latency issues have been resolved after reducing the number of cgroups
- The latest Linux kernel includes the cgroup fixes for these and the other issues

Logs Told Us It Was Kernel
It Felt Like Kernel

Linux Kernel 3.10.0-957.35.2, Glibc 2.17

```
./funccount -i 5 'c:__memcpy*'
Tracing 9 functions for "c:__memcpy*"...


FUNC                                    COUNT
__memcpy_ssse3_back                    986330
__memcpy_sse2                         1866318
```

Linux Kernel 4.18.0-372.9.1, Glibc 2.28

```
./funccount -i 5 'c:__memcpy*'
Tracing 21 functions for "b'c:__memcpy*'"...


FUNC                                    COUNT
b'__memcpy_avx_unaligned_erms'        1224262
b'__memcpy_sse2_unaligned'            8025791
```

- __memcpy_ssse3_back is most optimal for small and large buffers
- __memcpy_avx_unaligned doesn’t perform well for small buffers (less than 10 bytes)
- GLIBC_TUNABLES glibc.cpu.hwcaps can be used to disable AVX and enable SSE
- This issue was fixed in the latest Glibc 2.28 build

Logs Told Us It Was Kernel
It Felt Like Kernel
It Had To Be Kernel

Linux Kernel 3.10.0-957.35.2	**Linux Kernel 4.18.0-372.9.1**
# timertest -t timertest 1.4.0 Time Call Tests: clock_gettime(CLOCK_MONOTONIC): Diff: 0.000017363 sec Avg 17 nsec	# timertest -t timertest 1.4.0 Time Call Tests: clock_gettime(CLOCK_MONOTONIC): Diff: 0.000047152 sec Avg 47 nsec

- The intel_pstate=disable intel_idle.max_cstate=0 processor.max_cstate=1 boot parameters didn’t work with the new Linux kernel
- The vendor recommended boot parameters reduced clock_gettime overhead to 23 nsec

Logs Told Us It Was Kernel
It Felt Like Kernel
It Had To Be Kernel

Was it Kernel?

Simplified version of benchmark test

```
void foo(void) {
}
int main(int argc, char *argv[]) {
    int64_t sum = 0;
    for (int64_t i = 0; i < 1000000000LL; i++) {
        foo();
        sum += i;
    }
}
```

- Exclude all memory accesses
- Isolate from any other performance issues
- Long loop, empty function call, sum of two local variables

Compare execution time between old and new Linux kernel

```
(3.10.0-957.35.2) $ time ./loop

real    0m2.06s
user    0m2.04s
sys     0m0.01s
```

```
(4.18.0-372.9.1) $ time ./loop

real    0m2.65s
user    0m2.61s
sys     0m0.01s
```

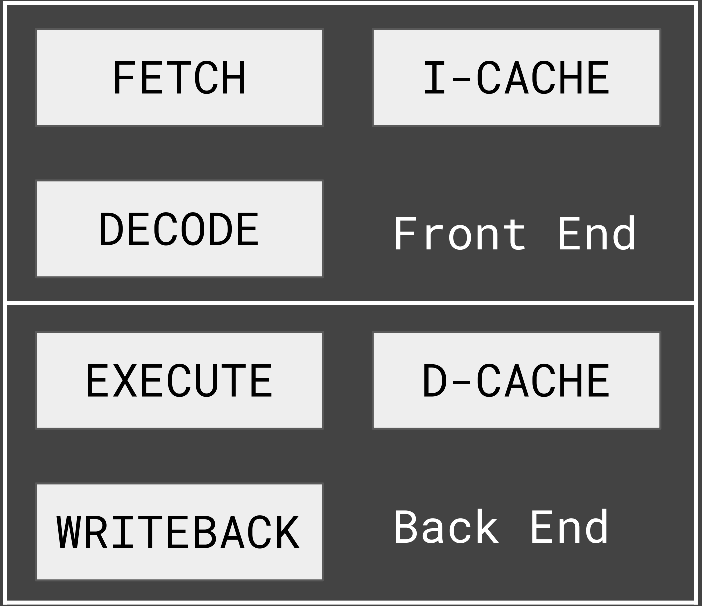
- The application performance degraded by about 30% compared to the old Linux kernel

The techniques used during the investigation

- Profile the additional “nop” instruction in the code
- Profile placing the local variables into the registers
- Run the Intel VTune profiler and analyze the performance
- Profile the hot code block alignment in the instruction cache
- Research the compilation flags to optimize the performance
- Research profile-guided compilation for better optimization

Basic concepts of CPU architecture

CPU Pipeline



Intel Core i7-9xx

Memory	Size	Latency
Register	64 bit	1 cycle
L1 cache	64 KB	4 cycles
L2 cache	256 KB	11 cycles
L3 cache	8 MB	39 cycles
Main memory	4+ GB	107 cycles

Disassemble code to check the differences

Linux Kernel 3.10.0-957.35.2,GCC 4.8.5	Linux Kernel 4.18.0-372.9.1,GCC 8.5.0

```
void foo(void)
 4004ed:   55         push   %rbp
 4004ee:   48 89 e5   mov    %rsp,%rbp
 4004f1:   5d         pop    %rbp
 4004f2:   c3         retq
```
	```
	void foo(void)
	 400536:   55         push   %rbp
	 400537:   48 89 e5   mov    %rsp,%rbp
	 40053a:   90         nop
	 40053b:   5d         pop    %rbp
	 40053c:   c3         retq
	```

- The new GCC 8.5.0 generates an additional '**nop**' instruction
- It doesn't emit any microcode, but must be fetched and decoded
- The additional '**nop**' instructions contribute to large code size

Profile ‘**nop**’ instruction with GCC 4.8.5

```
void foo(void) {

}

6,765,712,500      cycles
10,009,141,973     instructions
3,001,637,870      branches
31,451             branch-misses

2.275540333        sec time elapsed
```

```
void foo(void) {
        __asm__("nop");
}

6,798,423,185      cycles
11,008,906,212     instructions
3,001,595,573      branches
31,056             branch-misses

2.222114193        sec time elapsed
```

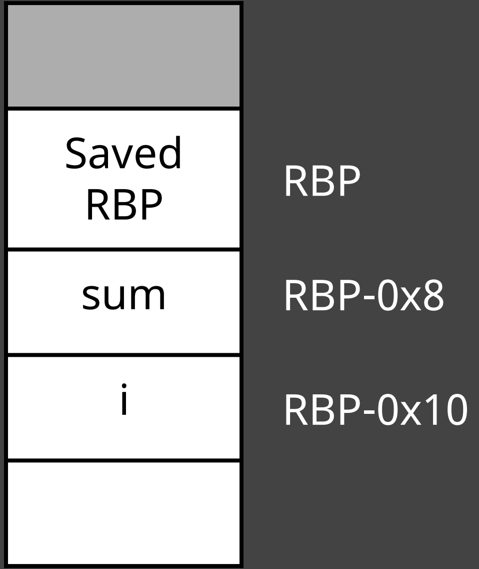
- Number of instructions increased
- Execution time remained the same

Local variables storage

```
  int64_t sum = 0;
40054c: 48 c7 45 f8 00 00 00  movq  $0x0,-0x8(%rbp)
  for (int64_t i = 0; i < 1000000000LL; i++) {
400554: 48 c7 45 f0 00 00 00  movq  $0x0,-0x10(%rbp)

. . . . . omitted text . . . . .

400563: 48 8b 45 f0           mov   -0x10(%rbp),%rax
400567: 48 01 45 f8           add   %rax,-0x8(%rbp)
  for (int64_t i = 0; i < 1000000000LL; i++) {
40056b: 48 83 45 f0 01        addq  $0x1,-0x10(%rbp)
```



- Compiler puts the local variables on the stack
- The access to the memory is much slower than the access to the register

Profile ‘**register**’ keyword with GCC 8.5.0

```
  register int64_t i, sum = 0;
40054f:   41 bc 00 00 00 00   mov    $0x0,%r12d
  for (i = 0; i < 1000000000LL; i++) {
400555:   bb 00 00 00 00      mov    $0x0,%rbx

  . . . . . omitted text . . . . .

400561:   49 01 dc            add    %rbx,%r12d
  for (i = 0; i < 1000000000LL; i++) {
400564:   48 83 c3 01         add    $0x1,%rbx
```

Without ‘register’ keyword:

```
8,294,200,306   cycles
11,012,279,315  instructions
3.456486927     seconds
```

With ‘register’ keyword:

```
7,147,275,887   cycles
10,010,622,649  instructions
2.978650290     seconds
```

- The ‘**register**’ keyword suggests compiler to use register for local variable
- The access to the register is much faster than the access to the memory
- The ‘**add**’ instruction can work with two registers directly

Profile with Intel VTune

Linux Kernel 3.10.0-957.35.2, GCC 4.8.5	**Linux Kernel 4.18.0-372.9.1, GCC 8.5.0**
Front-End Bound: **14.4%** of Pipeline Slots	Front-End Bound: **45.2%** of Pipeline Slots *Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.Tips: Make sure the code working size is not too large, the code layout does not require too many memory accesses per cycle to get enough instructions for filling four pipeline slots*

- The new Linux kernel showed much lower front-end pipeline utilization
- The problem is most likely the layout of the generated loop block

Loop block layout with GCC 8.5.0

```
4005be: call 400596 <foo>
  sum+=i;
4005c3: mov   -0x8(%rbp),%rax
4005c7: add   %rax,-0x10(%rbp)
  for(;i<10000000000LL;i++)
4005cb: add   $0x1,-0x8(%rbp)
4005d0: mov
$0x2540be3ff,%rax
4005da: cmp   %rax,-0x8(%rbp)
4005de: jle   4005be <main>
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80																
90																
A0																
B0															CALL	CALL
C0	CALL	CALL	CALL	MOV	MOV	MOV	MOV	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
D0	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	CMP	CMP	CMP	CMP	JLE	JLE
E0																
F0																

- CPU reads from the address aligned to the cache line size **0x400580** and **0x4005C0**
- The loop code block generated by GCC 8.5.0 spans across the two instruction cache lines
- This is the main reason of the poor front-end pipeline performance

Loop block layout with GCC 4.8.5

```
400554: callq 40052d <foo>
  sum+=i;
400559: mov    -0x10(%rbp),%rax
40055d: add    %rax,-0x8(%rbp)
  for(;i<10000000000LL;i++)
400561: addq   $0x1,-0x10(%rbp)
400566: mov    0x2540be3ff,%rax
400570: cmp    %rax,-0x10(%rbp)
400574: jle    400554 <main>
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40																
50					CALL	CALL	CALL	CALL	CALL	MOV	MOV	MOV	MOV	ADD	ADD	ADD
60	ADD	ADD	ADD	ADD	ADD	ADD	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
70	CMP	CMP	CMP	CMP	JLE	JLE										

- CPU reads from the address aligned to the cache line size **0x400540**
- The loop code block generated by GCC 4.8.5 fits into the one instruction cache line
- The loop code block that fits into the one instruction cache line reduces the number of Decoded Stream Buffer (DSB) cache misses

Align loop block with GCC 8.5.0

```
  __asm__("nop");
4005be: 90      nop
  __asm__("nop");
4005bf: 90      nop
4005c0: callq  400596 <foo>
  sum+=i;
4005c5: mov   -0x8(%rbp),%rax
4005c9: add   %rax,-0x10(%rbp)
  for(;i<10000000000LL;i++)
4005cd: addq $0x1,-0x8(%rbp)
4005d2: mov   $0x2540be3ff,%rax
4005dc: cmp   %rax,-0x8(%rbp)
4005e0: jle   4005c0 <main>
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80																
90																
A0																
B0															NOP	NOP
C0	CALL	CALL	CALL	CALL	CALL	MOV	MOV	MOV	MOV	ADD	ADD	ADD	ADD	ADD	ADD	ADD
D0	ADD	ADD	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	CMP	CMP	CMP	CMP
E0	JLE	JLE														
F0																

- By adding the two '**nop**' instructions the loop block was shifted two bytes forward to the address aligned to the cache line size **0x4005C0** and fits one instruction cache line
- The application performance was significantly improved

Use -falign-functions with GCC 8.5.0

```
0000000000400540 <foo>:

0000000000400550 <main>:

4005d1: callq 400540 <foo>
  sum+=i;
4005d6: mov    -0x8(%rbp),%rax
4005da: add    %rax,-0x10(%rbp)
  for(;i<10000000000LL;i++)
4005de: addq   $0x1,-0x8(%rbp)
4005e3: mov
$0x2540be3ff,%rax
4005ed: cmp    %rax,-0x8(%rbp)
4005f1: jle    4005d1 <main>
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0																
D0		CALL	CALL	CALL	CALL	CALL	MOV	MOV	MOV	MOV	ADD	ADD	ADD	ADD	ADD	ADD
E0	ADD	ADD	ADD	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	CMP	CMP	CMP
F0	CMP	JLE	JLE													

- The functions are aligned by 16 bytes, which benefits the execution speed
- The code block was shifted to fit the one instruction cache line

Performance report analysis

GCC 4.8.5 default	GCC 8.5.0 default	GCC 8.5.0 aligned
Elapsed Time: 18.597s Efficient fetching and decoding. Front-End Bound: 14.4% of Pipeline Slots Higher number of remote accesses don't affect code efficiency. NUMA: % of Remote Accesses: 16.1%	Elapsed Time: 21.404s Pipeline slots are mostly empty. Front-End Bound: 50.3% of Pipeline Slots Lower number of remote accesses don't improve code efficiency. NUMA: % of Remote Accesses: 4.0%	Elapsed Time: 18.517s Efficient fetching and decoding. Front-End Bound: 15.4% of Pipeline Slots Higher number of remote accesses don't affect code efficiency. NUMA: % of Remote Accesses: 6.3%

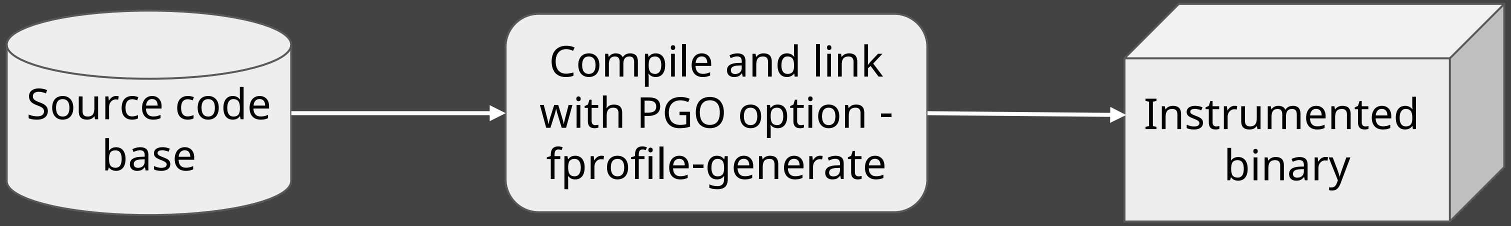
Compiler optimization options

- -falign-loops align loop code block to the beginning of the instruction cache line
- -funroll-loops remove shorter loops from generated code and mitigate the negative effect of the loop code block alignment
- -O2 optimization level includes all alignment options along with many other optimization flags
- -Os optimization level includes all optimizations from -O2 without alignment options
- -O3 optimization level turns on more expensive optimizations such as function inlining and various loop optimizations
- Higher levels of optimization can restrict debugging visibility
- The performance optimization options increase the time and the memory consumption during the compilation

Profile guided optimization

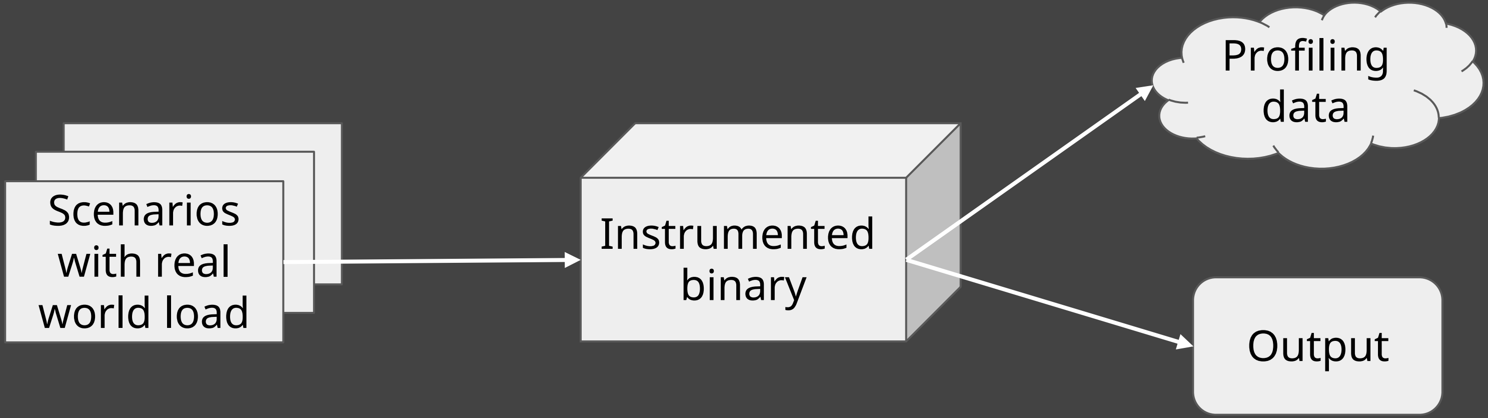
- PGO is a method used by GCC to produce optimal code by using the runtime data
- Since the data comes from the application, GCC can make more accurate guesses
- PGO workflow:
    - Instrumented compilation
    - Profiled execution
    - Optimized compilation

PGO instrumented compilation phase



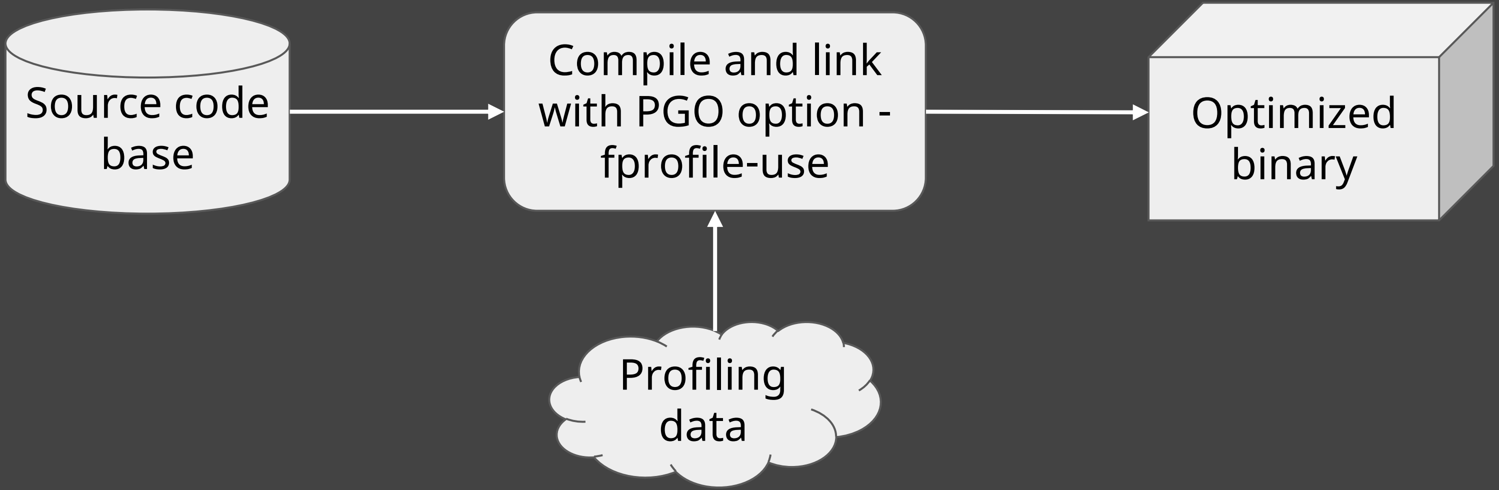
- Produces an executable with probes in each of the basic blocks of the program
- Each probe counts the number of times a basic block runs and records the direction taken by the branch

PGO profiled execution phase



- Instrumented binary generates a profiling data file that contains the counts from the program execution

PGO optimization phase



- Information from the profiled execution of the program is fed back to GCC
- GCC uses the profiling data to produce an optimized binary

Performance boost by using PGO

Test Suite	Without PGO	With PGO	Improvement
python_startup	17.1 ms	13.7 ms	1.25x faster
json_dumps	13.9 ms	11.4 ms	1.22x faster
json_loads	30.4 us	25.3 us	1.20x faster
xml_etree_generate	129 ms	109 ms	1.18x faster
xml_etree_parse	199 ms	175 ms	1.13x faster

- The Specs: Python 3.12.0, Linux kernel 4.18.0-372.9.1, GCC 8.5.0
- Benchmarking tests: py-performance benchmark suite

Logs Told Us It Was Kernel
It Felt Like Kernel
It Had To Be Kernel

It wasn’t Kernel

References

- The mystery of an unstable performance
- Performance Analysis and Tuning on Modern CPUs
- Using the GNU Compiler Collection (GCC) - Optimization Options
- Intel(R) 64 and IA-32 Architectures Optimization Reference Manual
- CPU Caches and Why You Care
- Profile guided optimization benchmarking
- Code alignment issues

Thank you!

Q & A