Compiler-Based Timing in Nautilus (and Elsewhere)

Souradip Ghosh

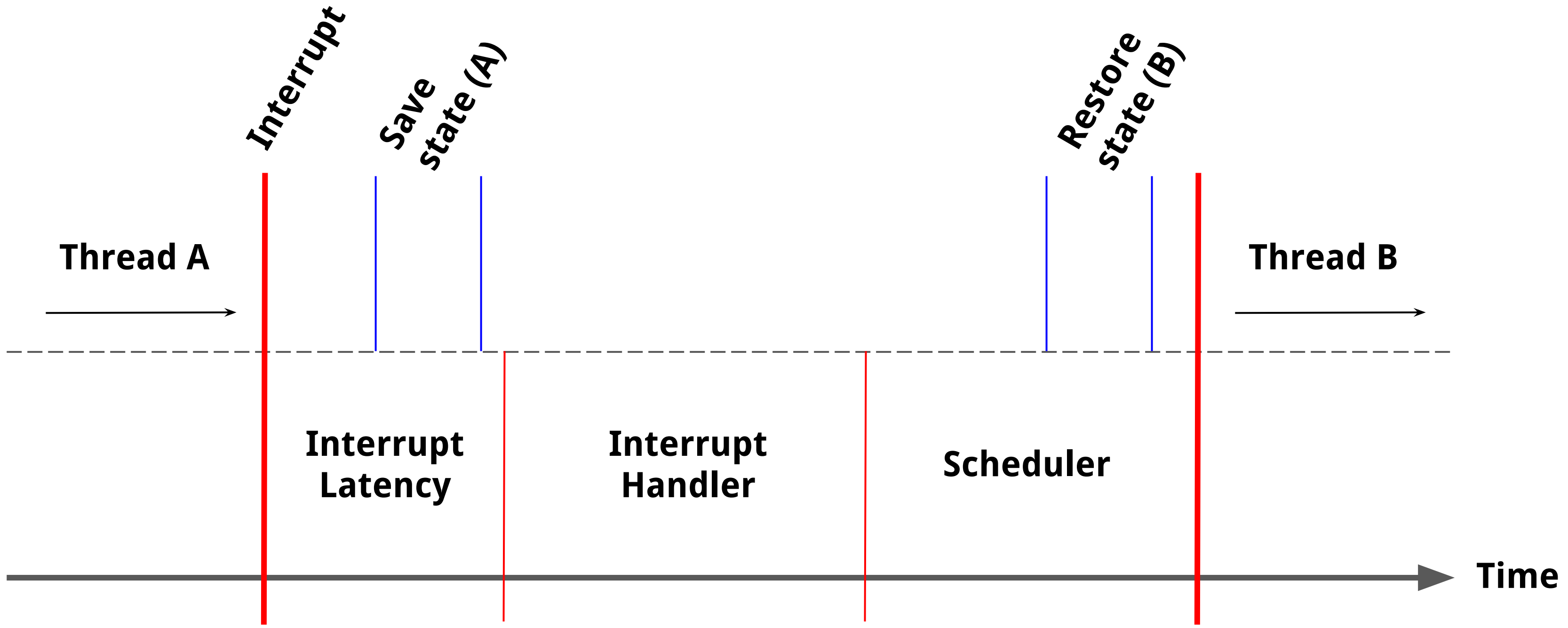
Agenda

Background: Preemptive multitasking

- Preemption:
  - Yielding control is *involuntary* --- triggered by hardware or software housed in the kernel
  - Context-switching between processes/threads
  - Scheduler decides which process/thread to switch to
- Provides a degree of fairness, completing I/O, important for real time systems

Background: Interrupts

- *Timer interrupts* are the main tool for starting context-switches



Background: Cooperative multitasking

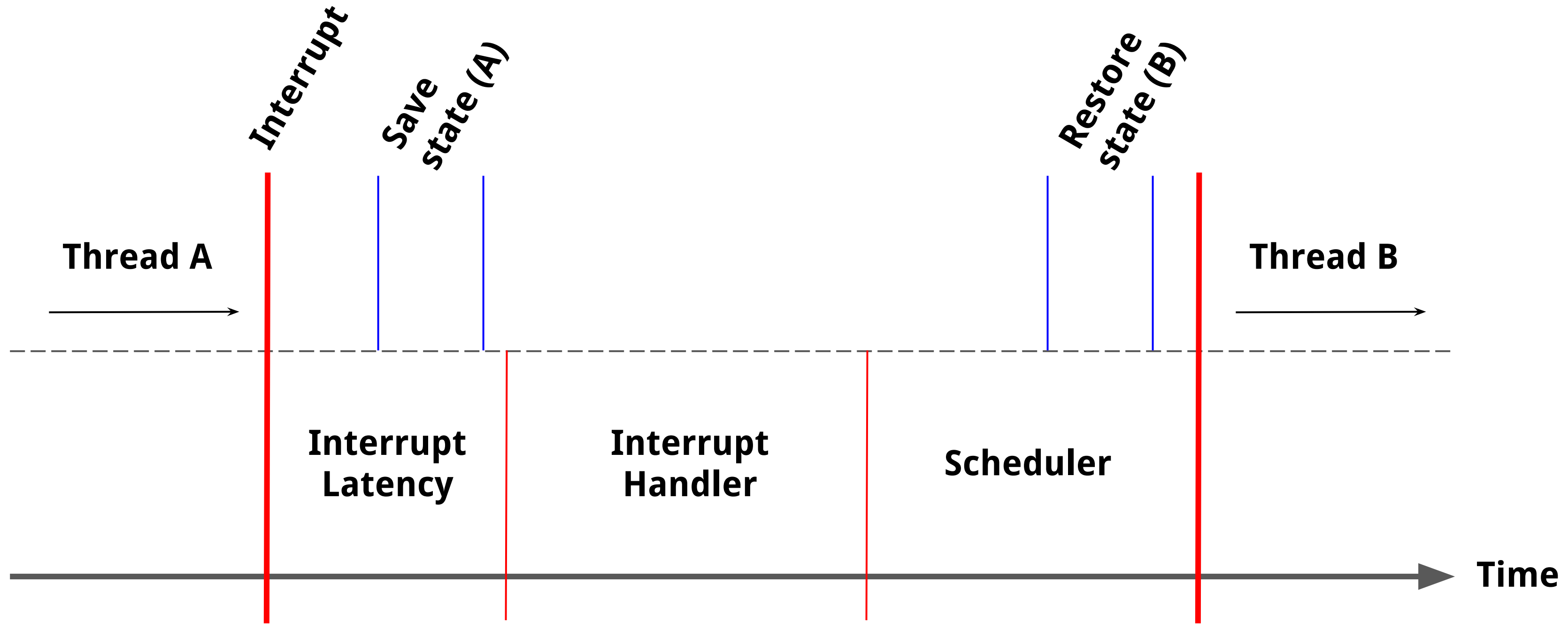
- Alternative to preemption
- Yielding control is *voluntary* --- the cooperative task decides when to yield control of the CPU
- Simplifies context-switching and scheduling
- Michael’s fibers implementation in Nautilus

Agenda

1. Background
2. **Motivation**
3. Scope: *Fibers in NK*
4. Compiler-based approach
5. Mechanics of the timing transform
6. Preliminary results
7. Further research

Motivation: Preemption overheads

- Large overheads in saving/restoring state, interrupt processing



Motivation: Finer granularity and parallelism

- *Granularity* --- ‘execution’ intervals for a thread/process
- Finer granularity → maximizes parallelism[1]
  - Works for thread/process-parallel written code
  - Facilitates load balancing across processors
- Given large context switch overhead:
  - Finer granularity not efficient
  - Interrupt-driven timing guarantees bottom out at 100 mHz
- Finer granularity easier with smaller context-switch overheads

[1] Moreira et al., *The Performance Impact of Granularity Control and Functional Parallelism* (2005). https://bit.ly/2k7frdQ

Motivation: Cooperating is difficult

- For developers --- programming headaches when deciding to yield control
- Inaccuracies in yielding are not practical for certain systems (real time systems)
- Easy scenarios for starving processes/threads, unfairness
- There’s no concept of *timing guarantees*

Motivation

Can we achieve a system that simplifies context-switching (to the extent seen cooperative multitasking) while maintaining timing guarantees (seen with preemption)?

Agenda

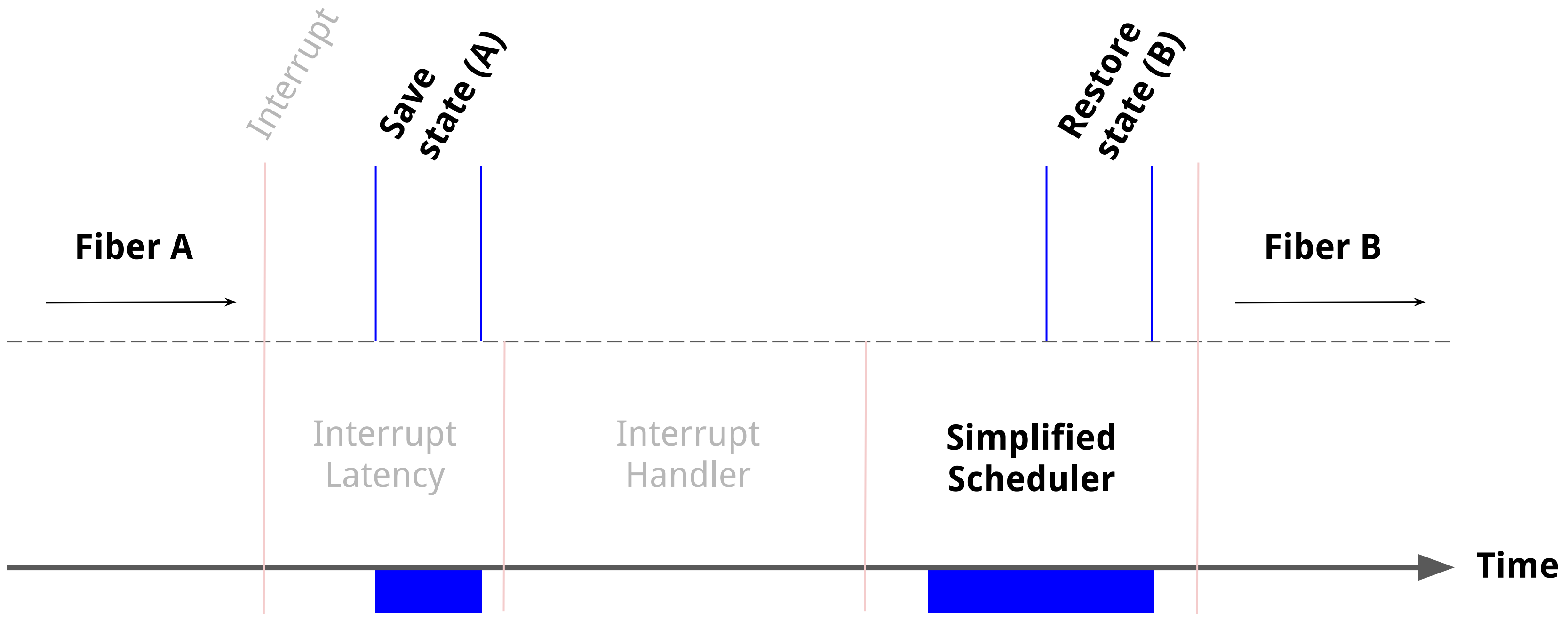
1. Background
2. Motivation
3. **Scope: *Fibers in NK***
4. Compiler-based approach
5. Mechanics of the timing transform
6. Preliminary results
7. Further research

Scope: Fibers in NK

- Fibers in NK designed with cooperative multitasking in mind:
  - No use of interrupts
  - Faster context-switches
  - Simplified scheduler
- The developer chooses when to yield (`nk_fiber_yield`)
- Current research focused on Michael’s fibers implementation

Scope: Fibers in NK --- Simplified approach

- Michael’s work --- fibers have lightweight context-switches



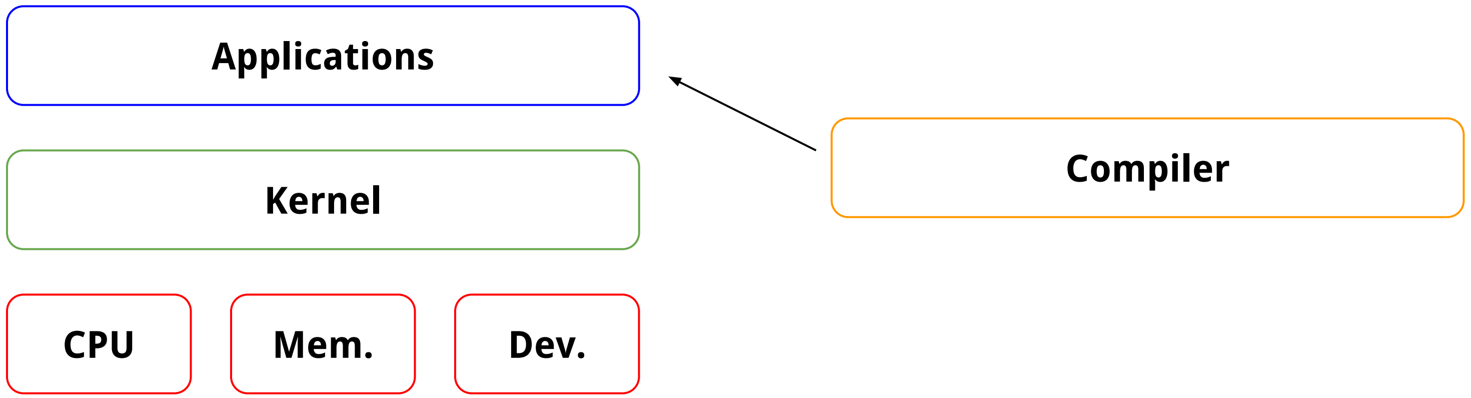
Scope: Fibers in NK --- Rethinking the question

Can we reintroduce timing guarantees for the fibers implementation in Nautilus?

Agenda

Compiler-based approach: Measuring time

- Determine execution “time” for all code paths at *compile-time*
- Fibers --- the compiler injects calls to **`nk_fiber_yield`** for a specified granularity



Agenda

1. Background
2. Motivation
3. Scope: *Fibers in NK*
4. Compiler-based approach
5. **Mechanics of the timing transform**
6. Preliminary results
7. Further research

Mechanics: Instruction latency

- Timing transform analyzes *instruction latencies* at the *bitcode* level (middle-end)
  - Sequential analysis --- in order of a routine’s CFG
- *Instruction latency* --- Time (clock cycles) before the data/output from an instruction is available to the module

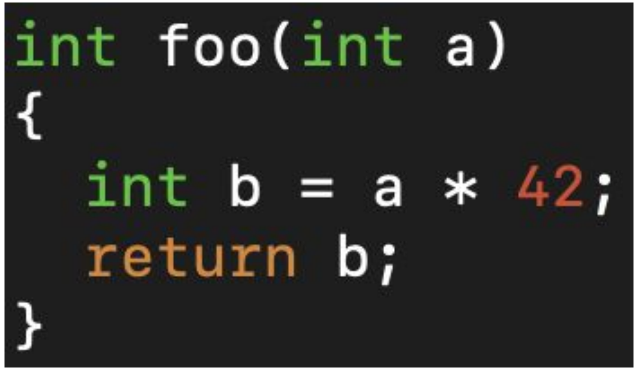
Mechanics: Simplified DFA

- Data flow analysis --- calculates accumulated latency up until a certain bitcode instruction
  - Utilizes function’s CFG
  - Iterate CFG *breadth-first*

Mechanics: Simplified DFA

- Data flow analysis --- calculates accumulated latency up until a certain bitcode instruction
    - Utilizes function's CFG
    - Iterate CFG *breadth-first*
- How do we calculate accumulated latencies? --- Use *data flow equations*
    - **GEN**[**I**] = **`getLatency`**(**I**)
    - **IN**[**I**] = **OUT**[**P**]
    - **OUT**[**I**] = **IN**[**P**] + **GEN**[**I**]

Function 'foo'



```
int foo(int a)
{
   int b = a * 42;
   return b;
}
```

CFG for 'foo' --- bitcode

```
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 %0, i32* %2, align 4
%4 = load i32, i32* %2, align 4
%5 = mul nsw i32 %4, 42
store i32 %5, i32* %3, align 4
%6 = load i32, i32* %3, align 4
ret i32 %6
```

	GEN[I]	IN[I]	OUT[I]
%2	3	-	3
%3	3	3	6
store ...	5	6	11
%4	5	11	16
%5	4	16	20
store ...	5	20	25
%6	5	25	30
ret	2	30	**32**

Mechanics: Expected and maximum “settings”

- Important case to handle --- *branching*
  - Analyze by basic blocks
- First instruction of a block may have *many* predecessors
  - What’s the IN set of that instruction?

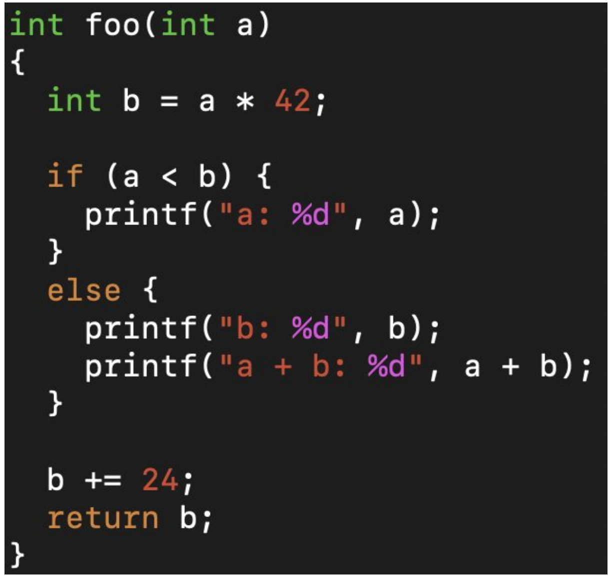
Mechanics: Expected and maximum “settings”

- Important case to handle --- *branching*
  - Analyze by basic blocks
- First instruction of a block may have *many* predecessors
  - What’s the IN set of that instruction?
- Transform introduces two approaches:
  - *Expected* latency --- $\mathbf{IN}[\mathbf{I}] = \mathbf{OUT}[\mathbf{P}_{k}] \cdot \mathbf{Pr}[\mathbf{P}_{k}]$; $\forall$ k, (k $\in$ P)
  - *Maximum* latency --- $\mathbf{IN}[\mathbf{I}] = \max\{ \mathbf{OUT}[\mathbf{P}_{k}] \}$; $\forall$ k, (k $\in$ P)

Mechanics: Expected and maximum “settings”

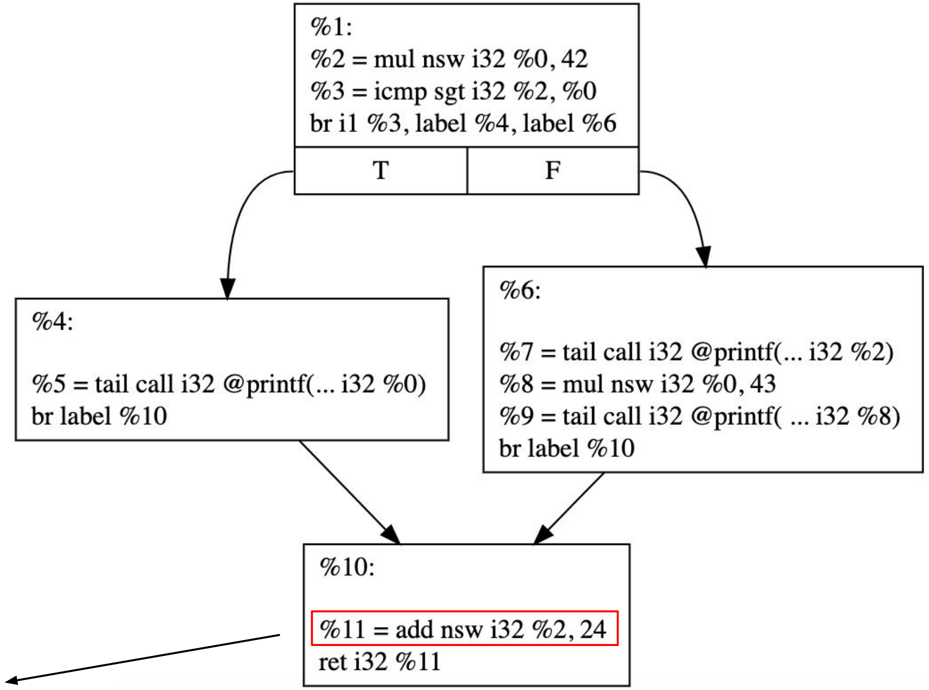
- Important case to handle --- *branching*
  - Analyze by basic blocks
- First instruction of a block may have *many* predecessors
  - What’s the IN set of that instruction?
- Transform introduces two approaches:
  - Expected latency (assuming no branch weights) ---
    $\mathbf{IN}[\mathbf{I}] = (\boldsymbol{\Sigma}\,\mathbf{OUT}[\mathbf{P}_k]) \,/\, |\mathbf{P}|$; $\forall$ k, (k $\in$ P)
  - Maximum latency --- $\mathbf{IN}[\mathbf{I}] = \max\{\, \mathbf{OUT}[\mathbf{P}_k] \,\}$; $\forall$ k, (k $\in$ P)

Function ‘foo’



```
int foo(int a)
{
  int b = a * 42;

  if (a < b) {
    printf("a: %d", a);
  }
  else {
    printf("b: %d", b);
    printf("a + b: %d", a + b);
  }

  b += 24;
  return b;
}
```

CFG for ‘foo’ --- bitcode



```
%1:
%2 = mul nsw i32 %0, 42
%3 = icmp sgt i32 %2, %0
br i1 %3, label %4, label %6
```

```
%4:

%5 = tail call i32 @printf(... i32 %0)
br label %10
```

```
%6:

%7 = tail call i32 @printf(... i32 %2)
%8 = mul nsw i32 %0, 43
%9 = tail call i32 @printf( ... i32 %8)
br label %10
```

```
%10:

%11 = add nsw i32 %2, 24
ret i32 %11
```

Expected: (20 + 42) / 2 = **31** cycles

Maximum: max{20, 42} = **42** cycles

Accumulated latencies:

- [br label %10].%4 = 20
- [br label %10].%6 = 42

Mechanics: Why have two “settings”?

- Analyzing by expected latency:
    - Broad estimate of latency given all predecessors of a block
    - Less likely to induce ‘conflicts’
    - More likely to ‘miss’ deadline

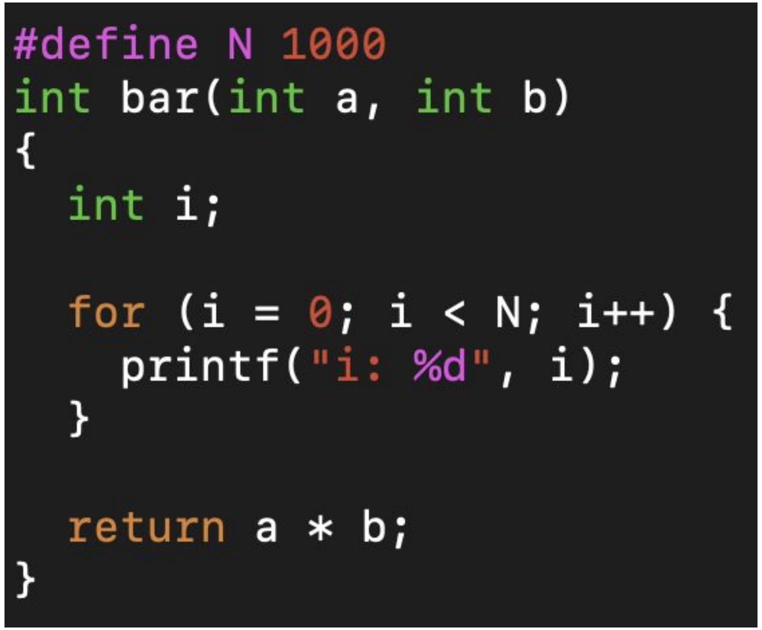
Mechanics: Why have two “settings”?

- Analyzing by expected latency:
  - Broad estimate of latency given all predecessors of a block
  - Less likely to induce ‘conflicts’
  - More likely to ‘miss’ deadline
- Analyzing by maximum latency:
  - Worst case latency given the predecessors of a block
  - Calculate worst case ‘latency size’ of a function
  - Likelier to inject calls to **`nk_fiber_yield`** on each code path
  - Incur overhead from ‘conflicts’

Mechanics: Loop “extension” policy

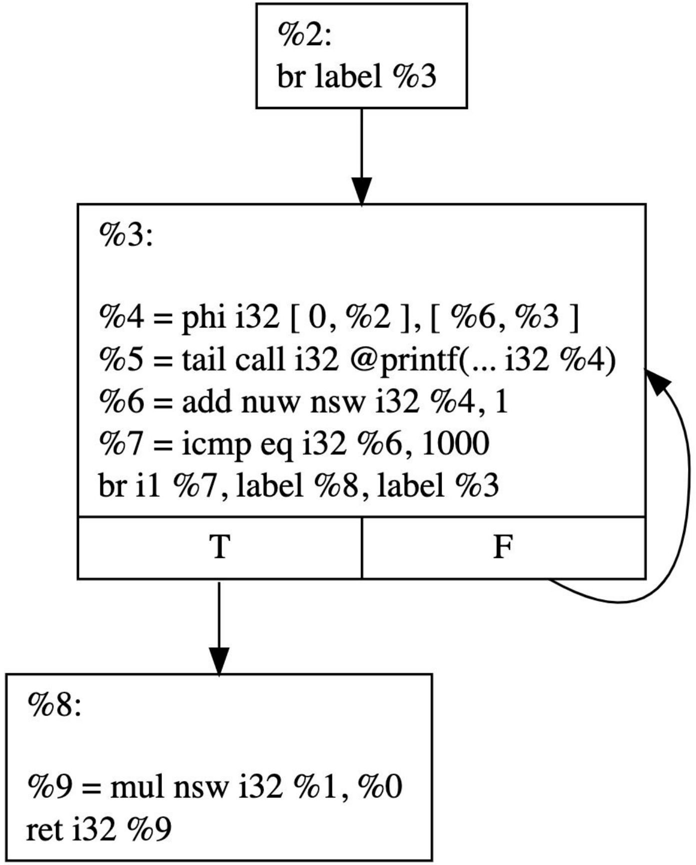
- Important case to handle --- *back edges*
- Analyzing predecessors for loops is difficult
- Possible solution --- “extend” the loop first
    - Calculate loop “latency size,” unroll to a multiple of the granularity
    - Ignore the back edge

Function ‘bar’



```
#define N 1000
int bar(int a, int b)
{
  int i;

  for (i = 0; i < N; i++) {
    printf("i: %d", i);
  }

  return a * b;
}
```

CFG for ‘bar’ --- bitcode

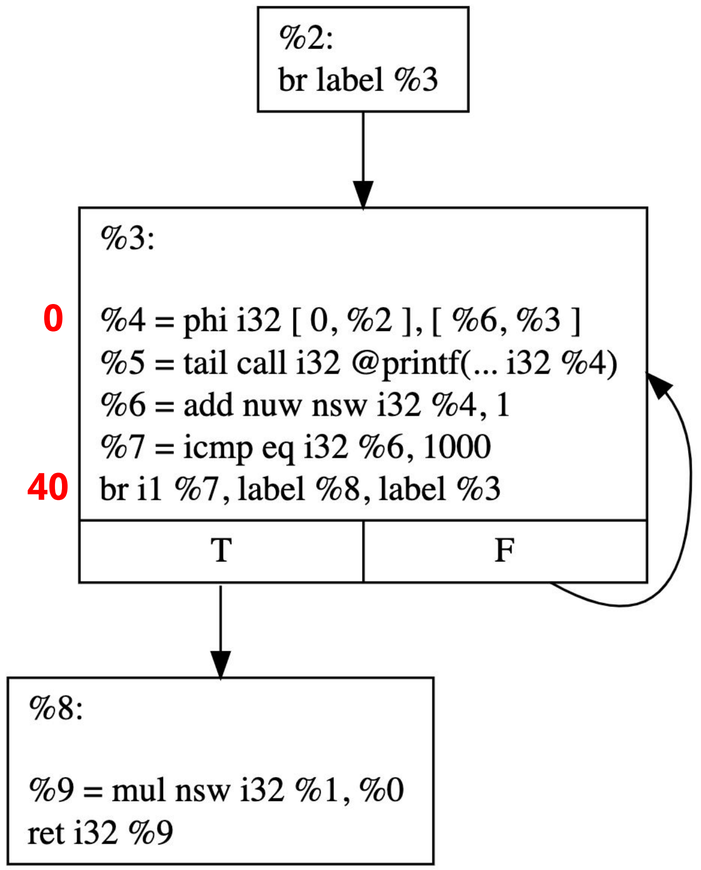


```
%2:
br label %3
```

```
%3:

%4 = phi i32 [ 0, %2 ], [ %6, %3 ]
%5 = tail call i32 @printf(... i32 %4)
%6 = add nuw nsw i32 %4, 1
%7 = icmp eq i32 %6, 1000
br i1 %7, label %8, label %3
```

```
%8:

%9 = mul nsw i32 %1, %0
ret i32 %9
```

CFG for ‘bar’ --- bitcode

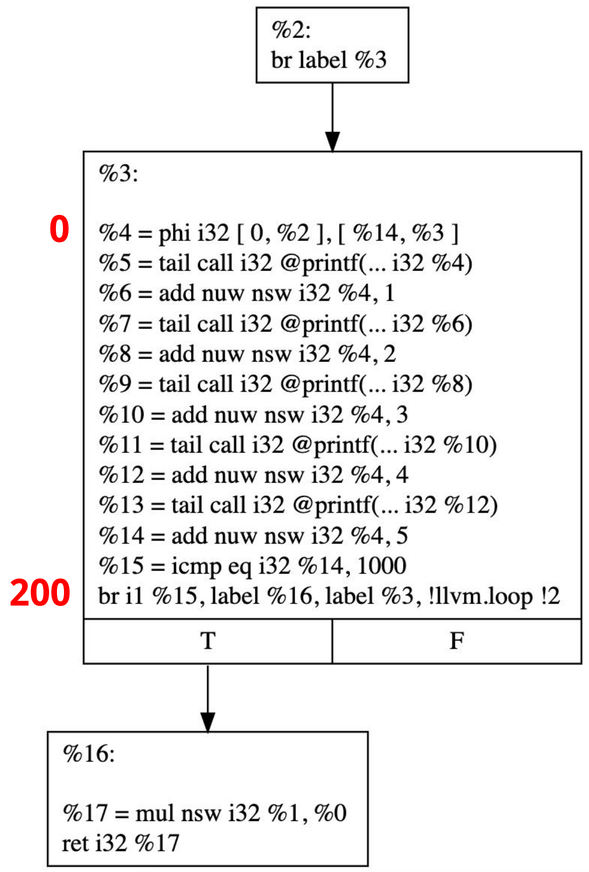


```
%2:
br label %3
```

```
%3:

%4 = phi i32 [ 0, %2 ], [ %6, %3 ]
%5 = tail call i32 @printf(... i32 %4)
%6 = add nuw nsw i32 %4, 1
%7 = icmp eq i32 %6, 1000
br i1 %7, label %8, label %3
```

```
%8:

%9 = mul nsw i32 %1, %0
ret i32 %9
```

Adjusted CFG for ‘bar’ --- bitcode



```
%2:
br label %3
```

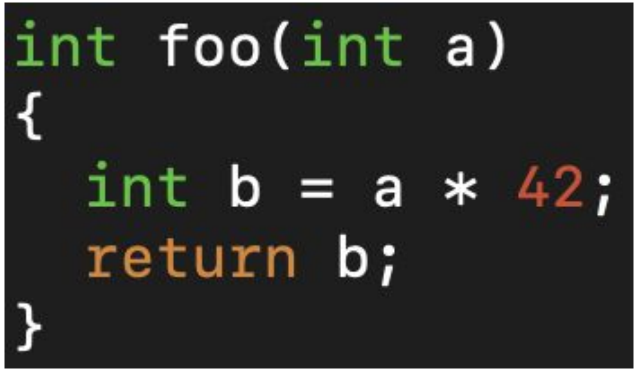
```
%3:

%4 = phi i32 [ 0, %2 ], [ %14, %3 ]
%5 = tail call i32 @printf(... i32 %4)
%6 = add nuw nsw i32 %4, 1
%7 = tail call i32 @printf(... i32 %6)
%8 = add nuw nsw i32 %4, 2
%9 = tail call i32 @printf(... i32 %8)
%10 = add nuw nsw i32 %4, 3
%11 = tail call i32 @printf(... i32 %10)
%12 = add nuw nsw i32 %4, 4
%13 = tail call i32 @printf(... i32 %12)
%14 = add nuw nsw i32 %4, 5
%15 = icmp eq i32 %14, 1000
br i1 %15, label %16, label %3, !llvm.loop !2
```

```
%16:

%17 = mul nsw i32 %1, %0
ret i32 %17
```

Mechanics: Determining injection locations

- Iterate over CFG --- traversing same as DFA (*breadth-first*)
  - Analyze each code path (no back edges)
- Mark instructions that reach the granularity/deadline
  - Reset a variable in the transform
  - Continue iterating until an instruction passes the deadline again
- Inject a call instruction to **`nk_fiber_yield`** before marked instructions

Function ‘foo’



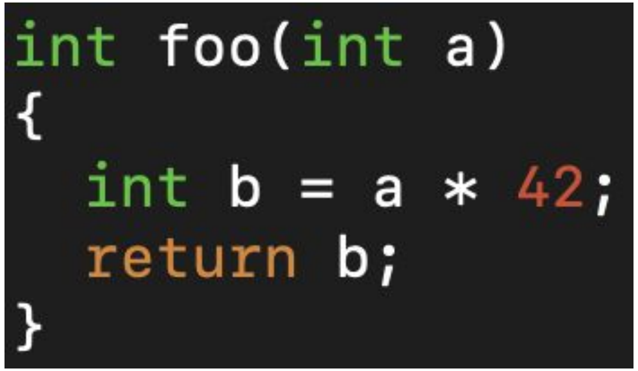
```
int foo(int a)
{
   int b = a * 42;
   return b;
}
```

CFG for ‘foo’ --- bitcode

```
3  %2 = alloca i32, align 4
6  %3 = alloca i32, align 4
11 store i32 %0, i32* %2, align 4
16 %4 = load i32, i32* %2, align 4
20 %5 = mul nsw i32 %4, 42
25 store i32 %5, i32* %3, align 4
30 %6 = load i32, i32* %3, align 4
32 ret i32 %6
```

Last injection location’s latency: 0

Function ‘foo’



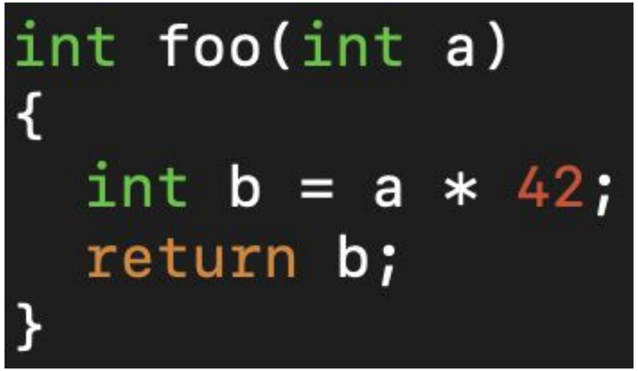
```c
int foo(int a)
{
   int b = a * 42;
   return b;
}
```

CFG for ‘foo’ --- bitcode

```
3  %2 = alloca i32, align 4
6  %3 = alloca i32, align 4
11 store i32 %0, i32* %2, align 4
16 %4 = load i32, i32* %2, align 4
20 %5 = mul nsw i32 %4, 42
25 store i32 %5, i32* %3, align 4
30 %6 = load i32, i32* %3, align 4
32 ret i32 %6
```

Last injection location’s latency: 11

Function 'foo'



```c
int foo(int a)
{
   int b = a * 42;
   return b;
}
```

CFG for 'foo' --- bitcode

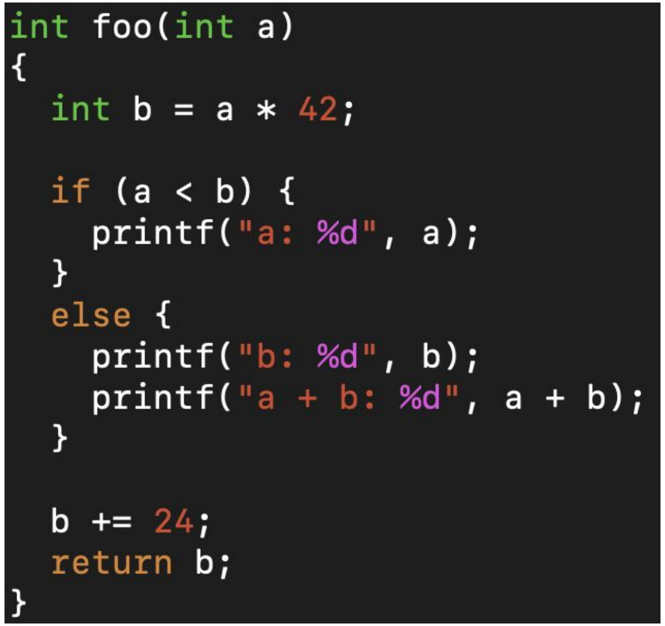
```
3  %2 = alloca i32, align 4
6  %3 = alloca i32, align 4
11 store i32 %0, i32* %2, align 4
16 %4 = load i32, i32* %2, align 4
20 %5 = mul nsw i32 %4, 42
25 store i32 %5, i32* %3, align 4
30 %6 = load i32, i32* %3, align 4
32 ret i32 %6
```

Last injection location's latency: 25

Mechanics: High and low “settings”

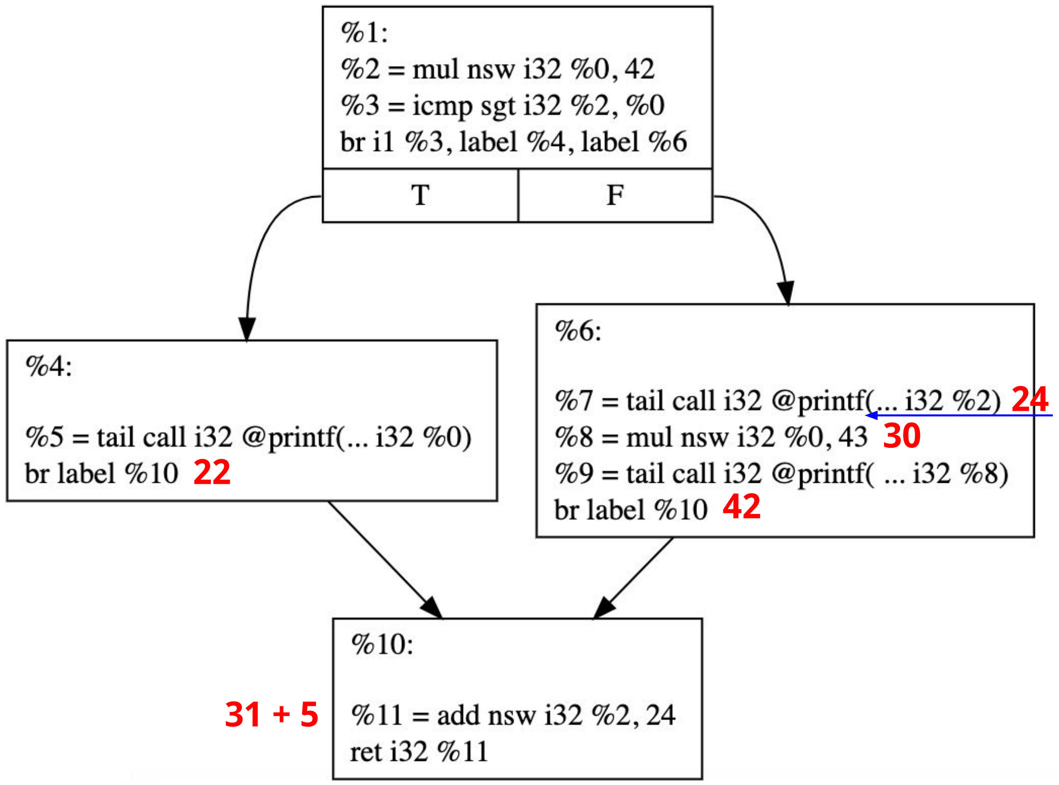
- Issue of multiple predecessors (again):
  - Which “last reset point” do we choose?
  - A second layer of “settings” --- *conservativeness*
- High conservativeness:
  - Min{ **k** } ; ∀ k, (k ∈ [Incoming reset points])
- Low conservativeness:
  - Max{ **k** } ; ∀ k, (k ∈ [Incoming reset points])

Function ‘foo’



```
int foo(int a)
{
  int b = a * 42;

  if (a < b) {
    printf("a: %d", a);
  }
  else {
    printf("b: %d", b);
    printf("a + b: %d", a + b);
  }

  b += 24;
  return b;
}
```

CFG for ‘foo’ --- bitcode --- Expected measurements



```
%1:
%2 = mul nsw i32 %0, 42
%3 = icmp sgt i32 %2, %0
br i1 %3, label %4, label %6
```

```
%4:

%5 = tail call i32 @printf(... i32 %0)
br label %10
```

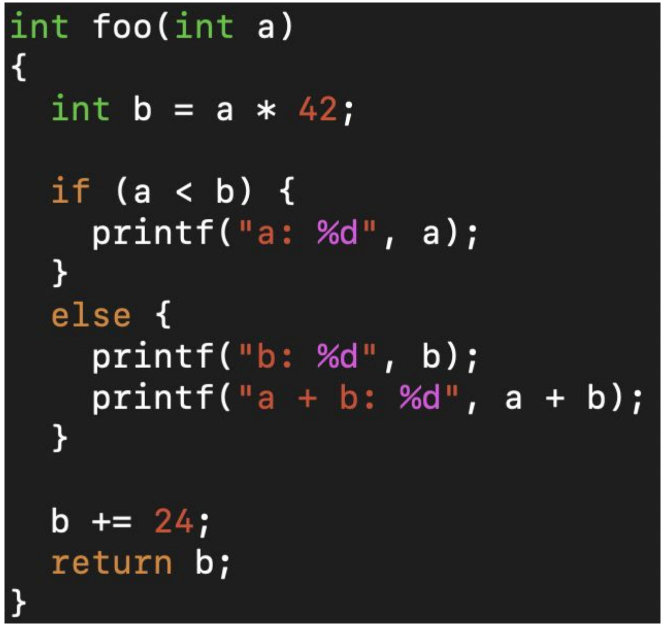
```
%6:

%7 = tail call i32 @printf(... i32 %2)
%8 = mul nsw i32 %0, 43
%9 = tail call i32 @printf( ... i32 %8)
br label %10
```

```
%10:

%11 = add nsw i32 %2, 24
ret i32 %11
```

Last injection location’s latency:

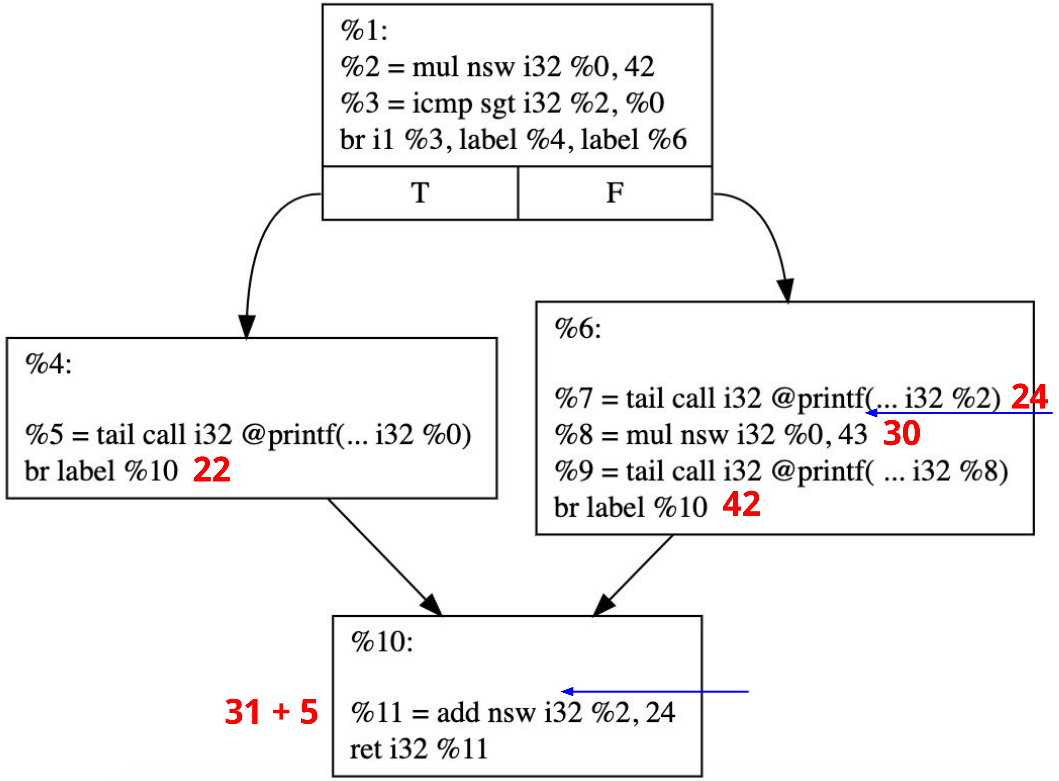
- [br label %10].%4 = **0**
- [br label %10].%6 = **30**

Function ‘foo’



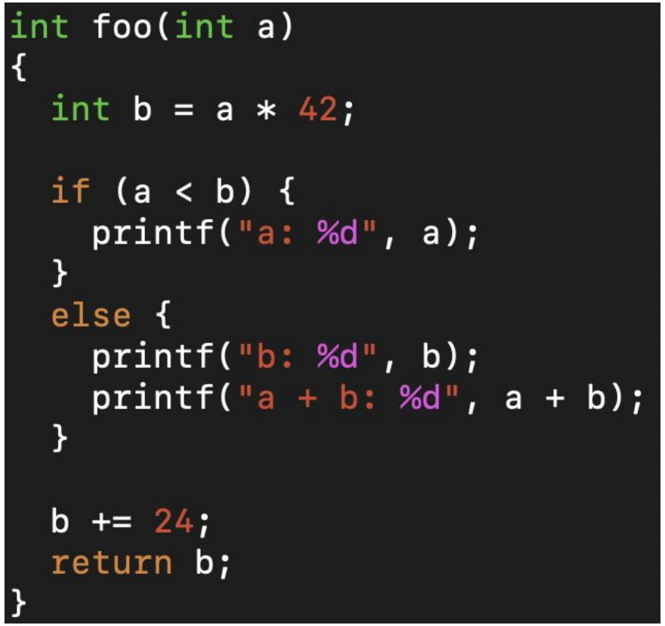
```
int foo(int a)
{
  int b = a * 42;

  if (a < b) {
    printf("a: %d", a);
  }
  else {
    printf("b: %d", b);
    printf("a + b: %d", a + b);
  }

  b += 24;
  return b;
}
```

CFG for ‘foo’ --- bitcode --- Expected measurements



Last injection location’s latency: 36

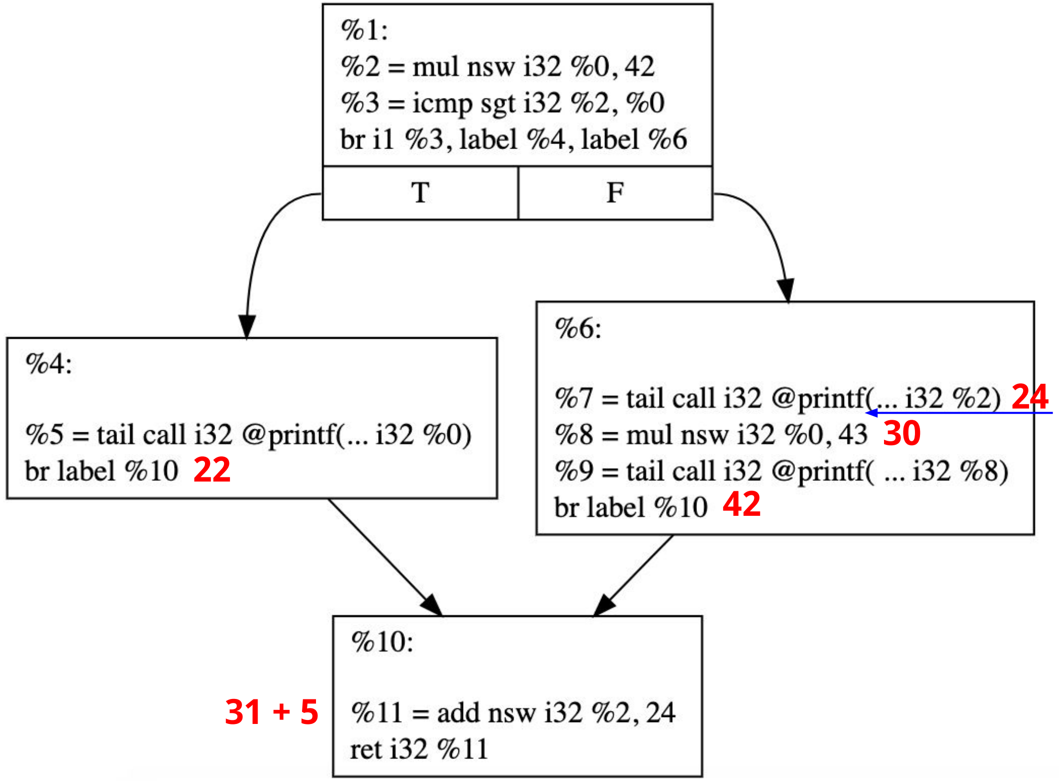
Function 'foo'



```
int foo(int a)
{
  int b = a * 42;

  if (a < b) {
    printf("a: %d", a);
  }
  else {
    printf("b: %d", b);
    printf("a + b: %d", a + b);
  }

  b += 24;
  return b;
}
```

Last injection location's latency: 30

CFG for 'foo' --- bitcode --- Expected measurements



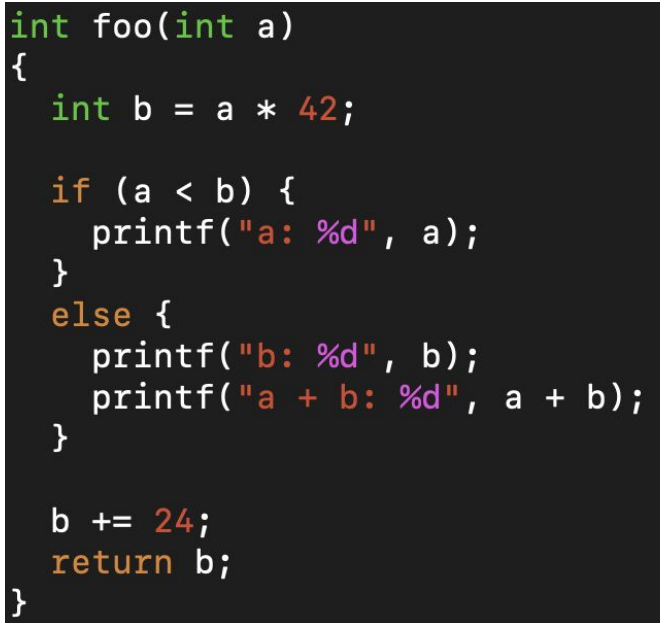
```
%1:
%2 = mul nsw i32 %0, 42
%3 = icmp sgt i32 %2, %0
br i1 %3, label %4, label %6
```

```
%4:

%5 = tail call i32 @printf(... i32 %0)
br label %10  22
```

```
%6:

%7 = tail call i32 @printf(... i32 %2)  24
%8 = mul nsw i32 %0, 43  30
%9 = tail call i32 @printf( ... i32 %8)
br label %10  42
```

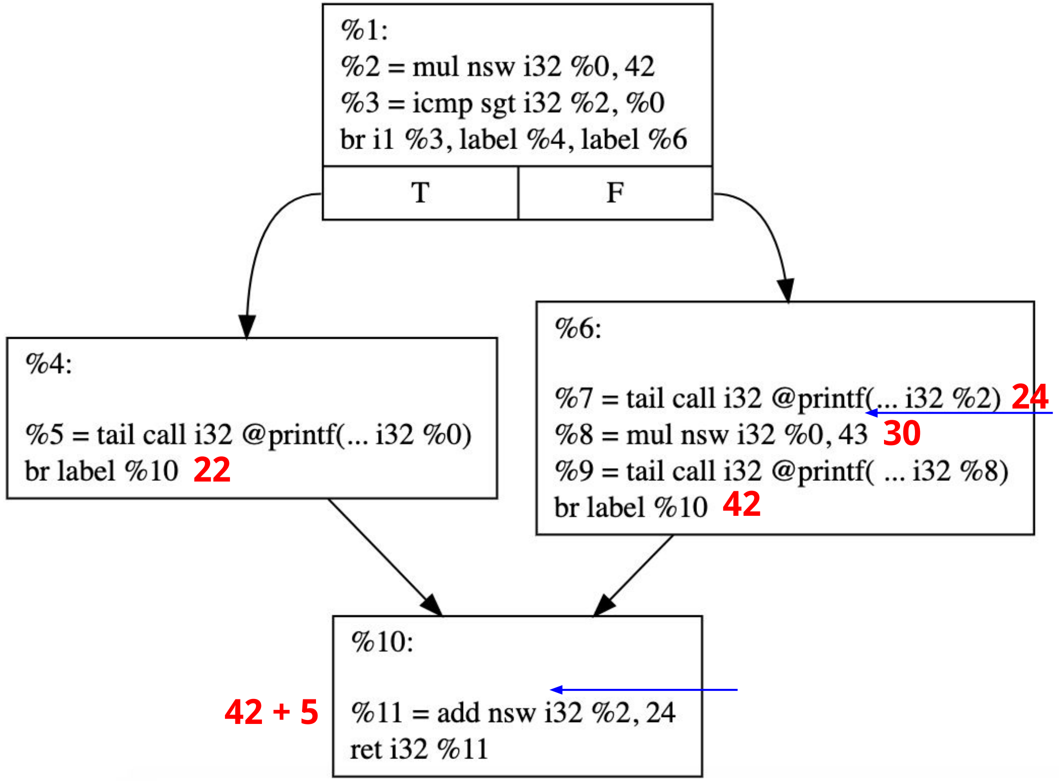
```
%10:

%11 = add nsw i32 %2, 24
ret i32 %11
```

Function ‘foo’



```
int foo(int a)
{
  int b = a * 42;

  if (a < b) {
    printf("a: %d", a);
  }
  else {
    printf("b: %d", b);
    printf("a + b: %d", a + b);
  }

  b += 24;
  return b;
}
```

CFG for ‘foo’ --- bitcode --- Maximum measurements



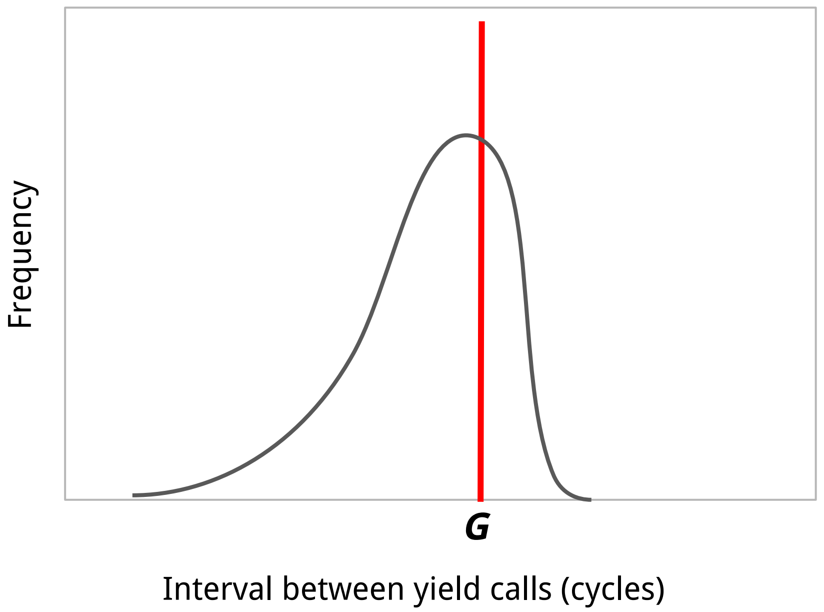
```
%1:
%2 = mul nsw i32 %0, 42
%3 = icmp sgt i32 %2, %0
br i1 %3, label %4, label %6
        T        F

%4:

%5 = tail call i32 @printf(... i32 %0)
br label %10  22

%6:

%7 = tail call i32 @printf(... i32 %2) 24
%8 = mul nsw i32 %0, 43  30
%9 = tail call i32 @printf( ... i32 %8)
br label %10  42

%10:

42 + 5   %11 = add nsw i32 %2, 24
         ret i32 %11
```

Last injection location’s latency:

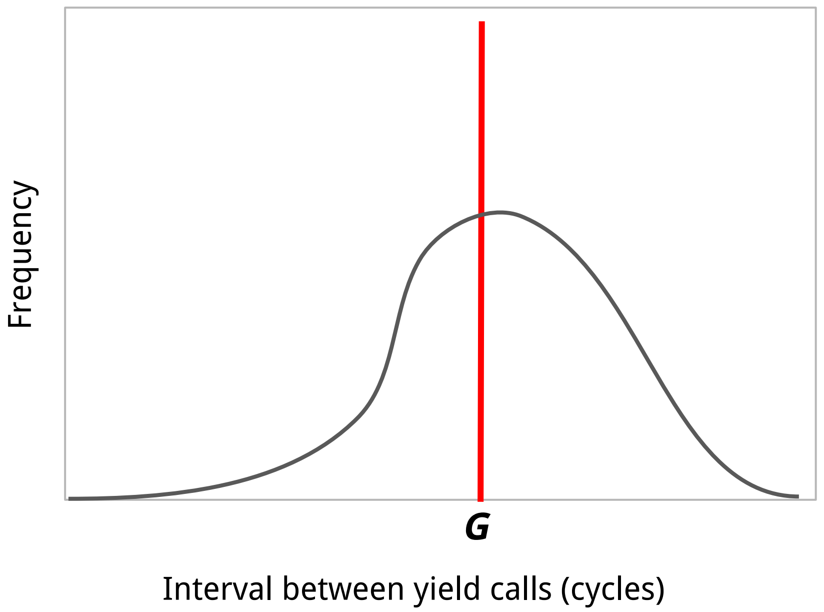
- [br label %10].%4 = 0
- [br label %10].%6 = 30

Mechanics: Why have two “settings” (again)?

Expected (DFA), **High** (Con.)

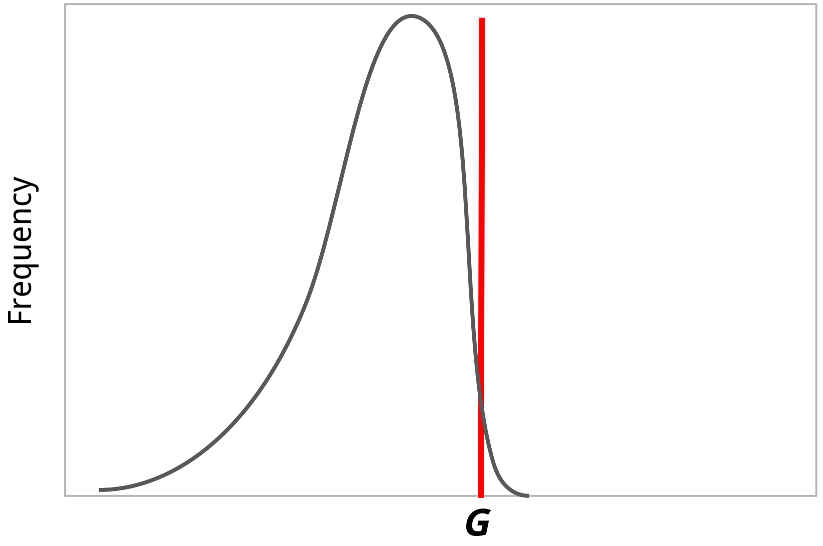


Expected (DFA), **Low** (Con.)



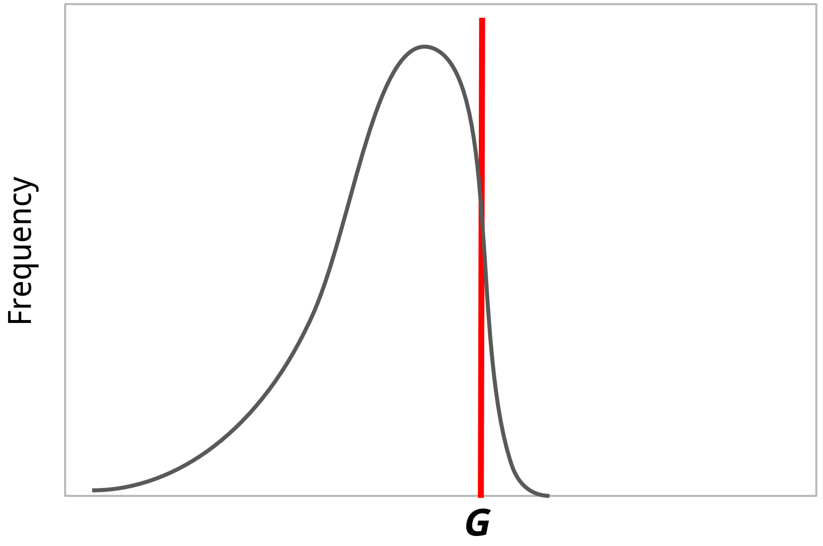
Mechanics: Why have two “settings” (again)?

Maximum (DFA), **High** (Con.)



Interval between yield calls (cycles)

Maximum (DFA), **Low** (Con.)

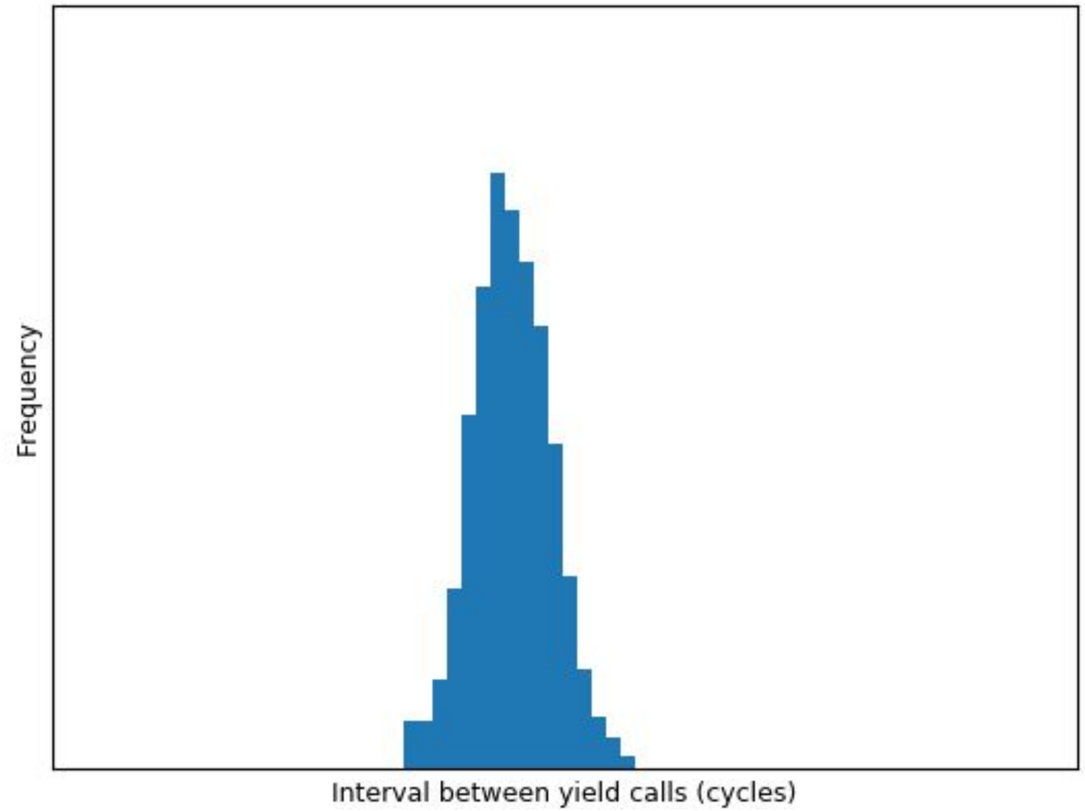


Interval between yield calls (cycles)

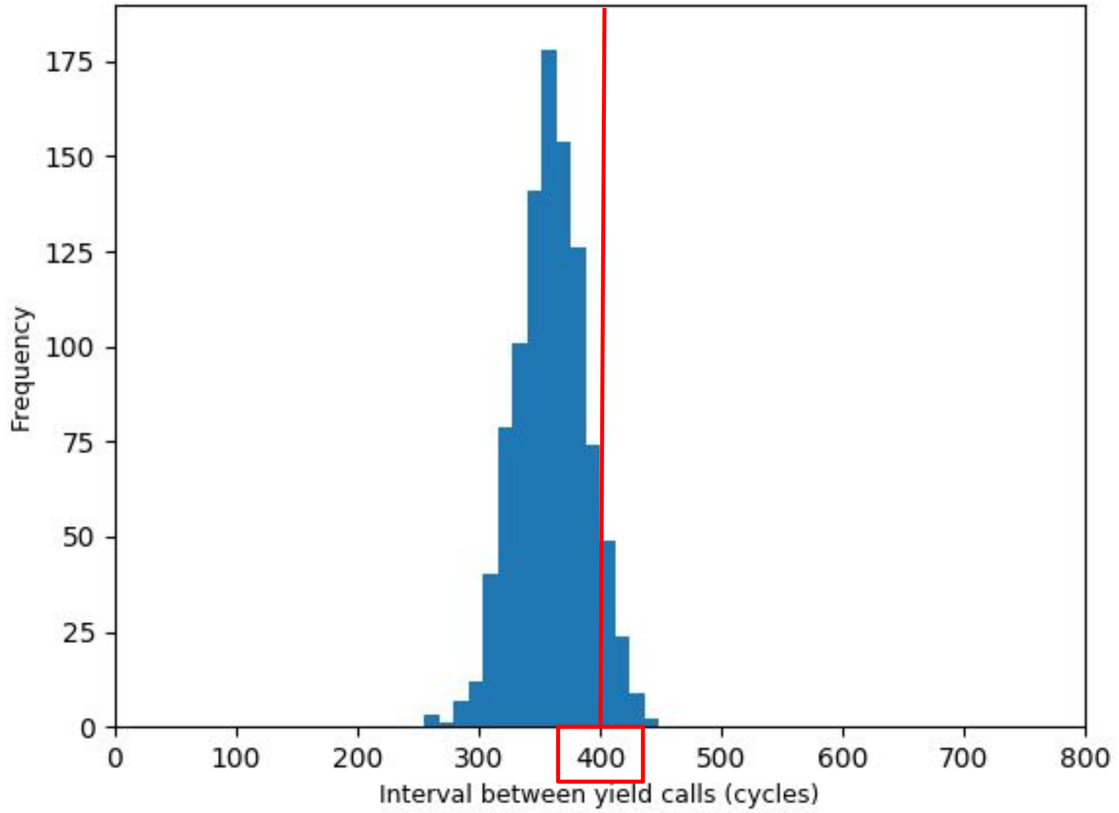
Agenda

1. Background
2. Motivation
3. Use case: *Fibers in NK*
4. Mechanics of the timing transform
5. **Preliminary results**
6. Further research

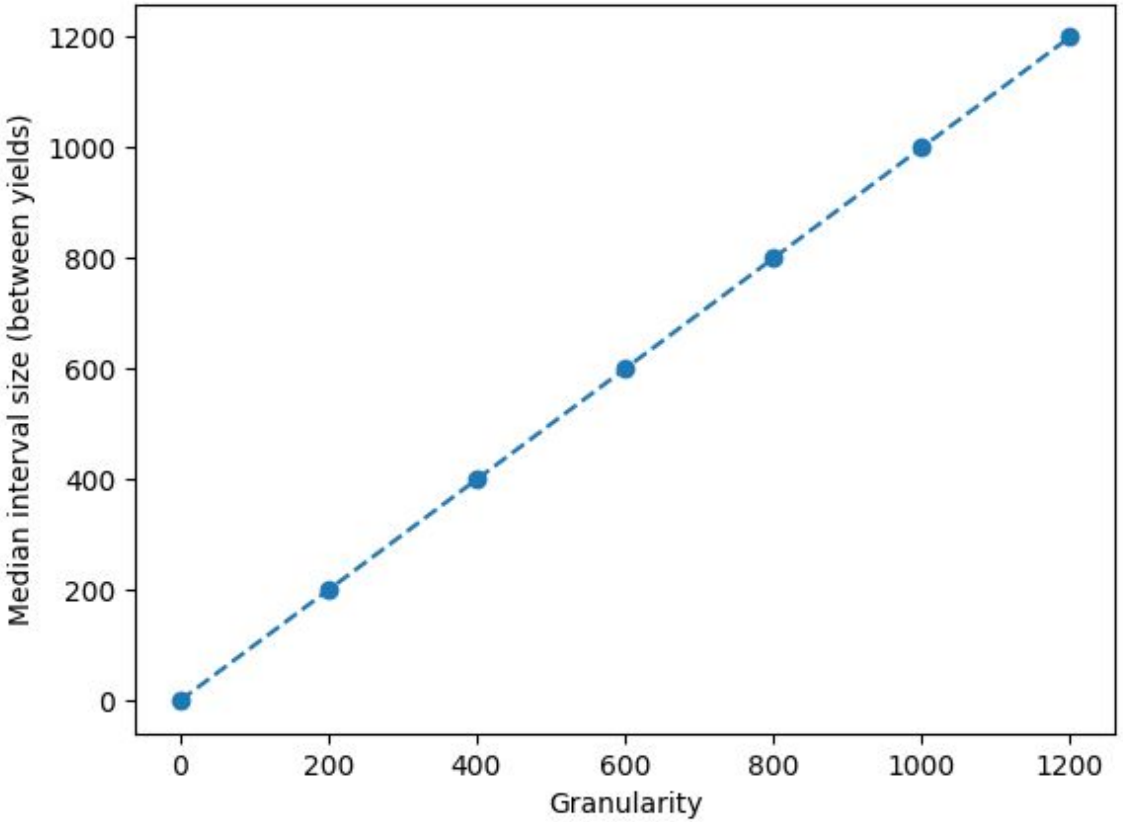
Preliminary results: Goals



Preliminary results: Goals



Preliminary results: Goals



Preliminary results: Testing environment

- Transform written using LLVM
- Testing conducted with Nautilus Aerokernel on Peroni/Zythos cluster
- Nautilus compiled with Clang 8.0, under O2
- Nautilus run with QEMU:
    - Enabled KVM --- more accurate results
    - Enabled instruction sets through AVX2

Preliminary results: Methodology

- Determining latency measurements:
  - Based on data of instruction latencies --- *bitcode* level
  - CMU data set, ~10 years old
- Measured time intervals between calls to **`nk_fiber_yield`**
  - Timing measured in cycle counts
  - Cycle counts collected via **`rdtsc`** (built into Nautilus)

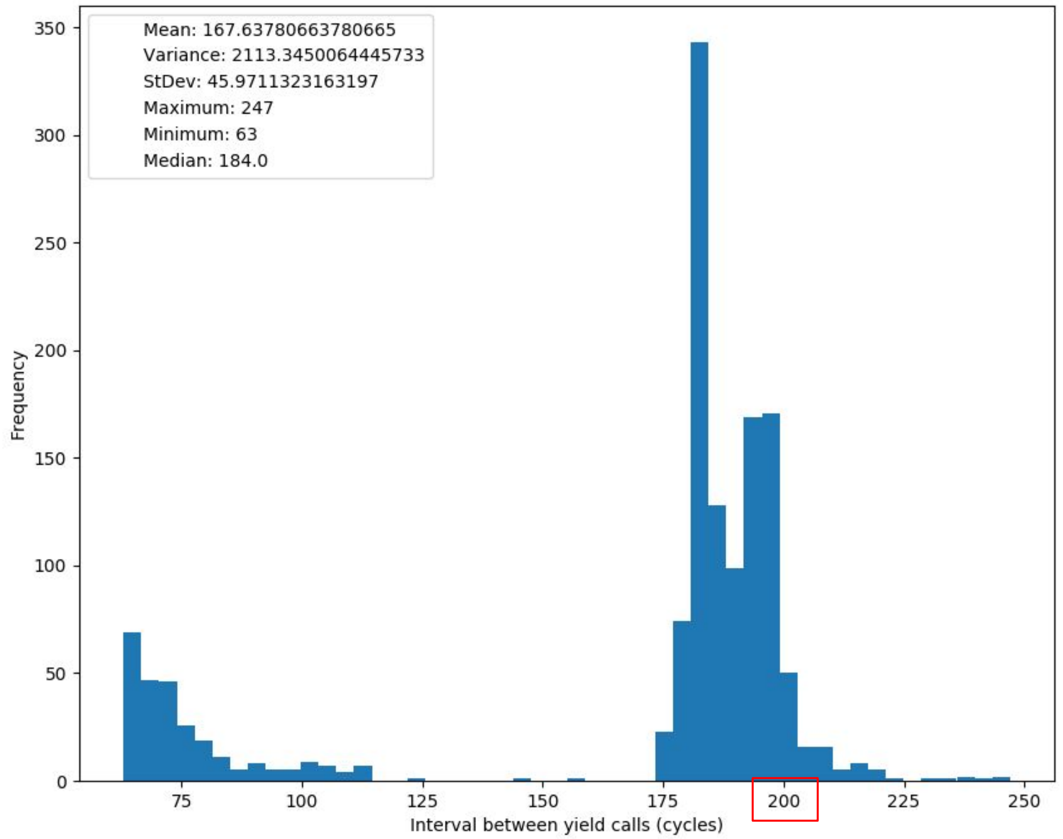
Preliminary results: Methodology

- Benchmarks written as fibers
  - Two fibers yielding back and forth
  - Each fiber has relatively equal execution time
- Benchmarks include --- simple algorithms, pointer indirection, floating point operations, nested loop structures
- Each benchmark executed 10 times, outliers discarded

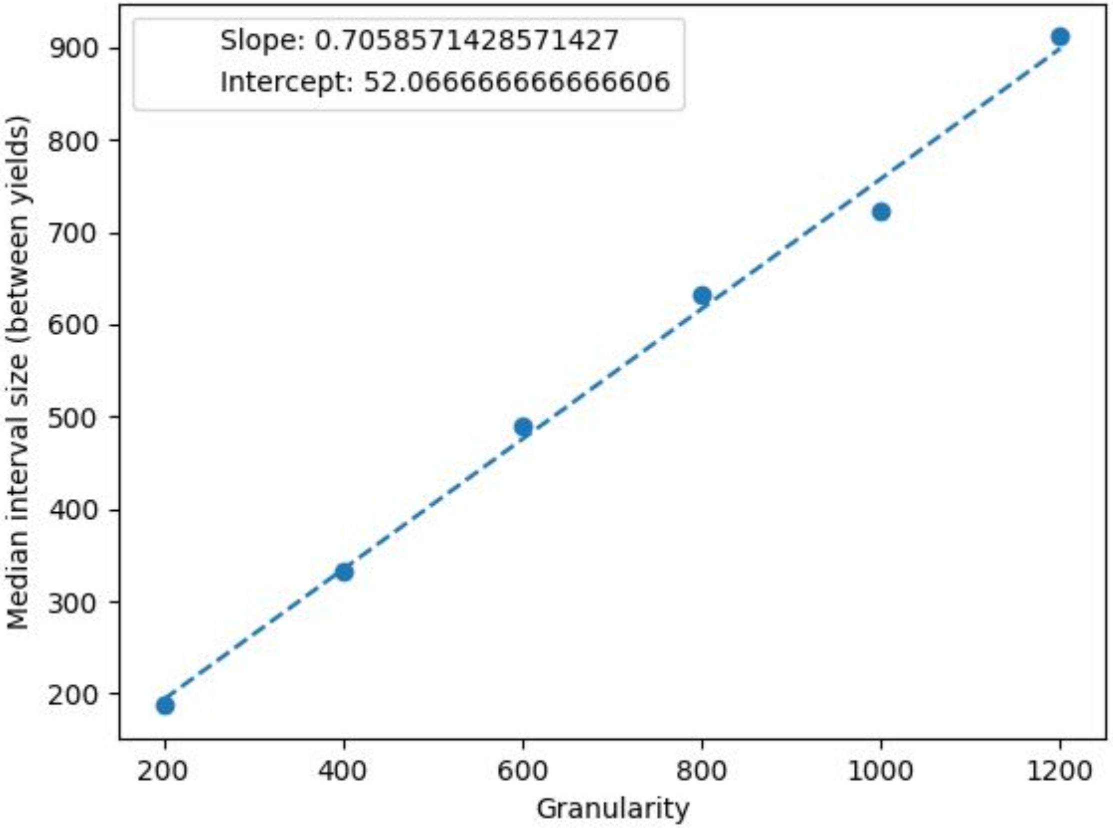
Preliminary results: Current constraints

- CMU data set --- for instruction latencies:
  - Out of date, inaccurate
  - Incomplete --- many LLVM bitcode instructions unhandled
- Intraprocedural
- Loop “extension” based on estimates of loop “latency size”
- QEMU not entirely accurate

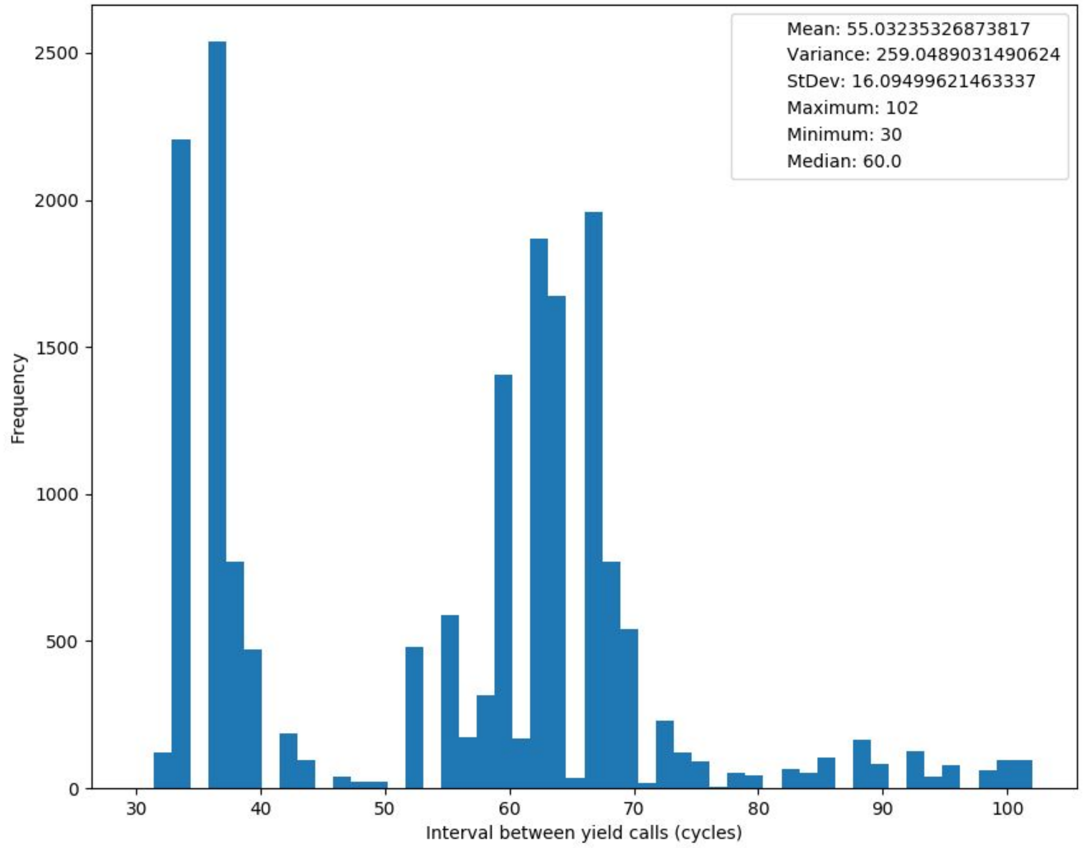
Preliminary results: Benchmark --- FP Operations



Preliminary results: Benchmark --- FP Operations

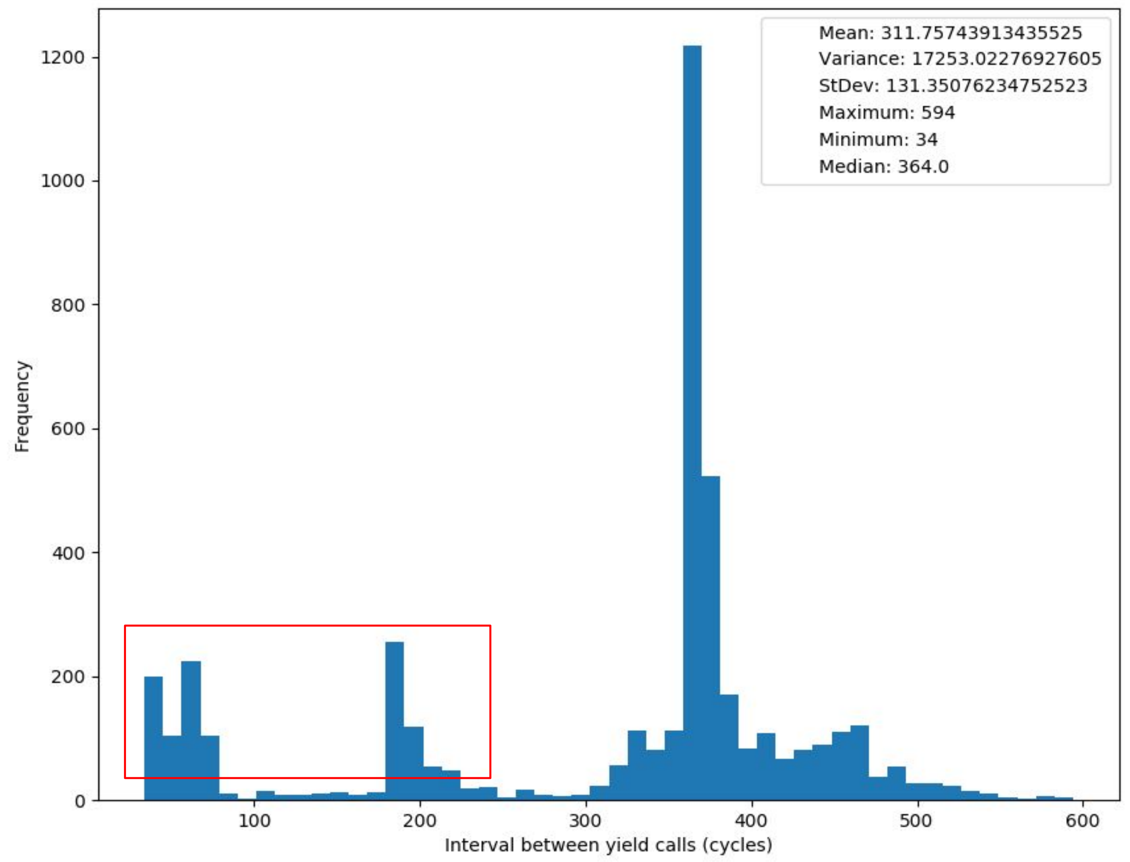


Preliminary results: Benchmark --- BST Lookup



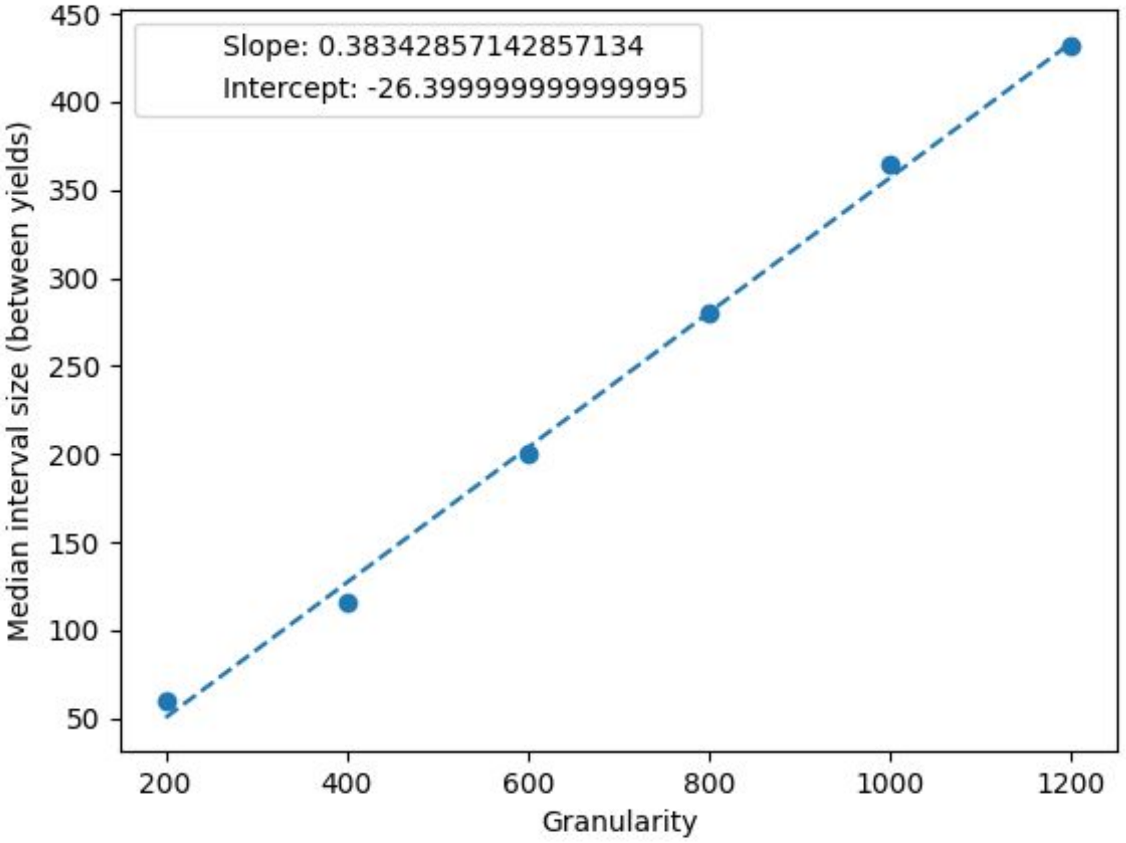
***G* = 200**

Preliminary results: Benchmark --- BST Lookup

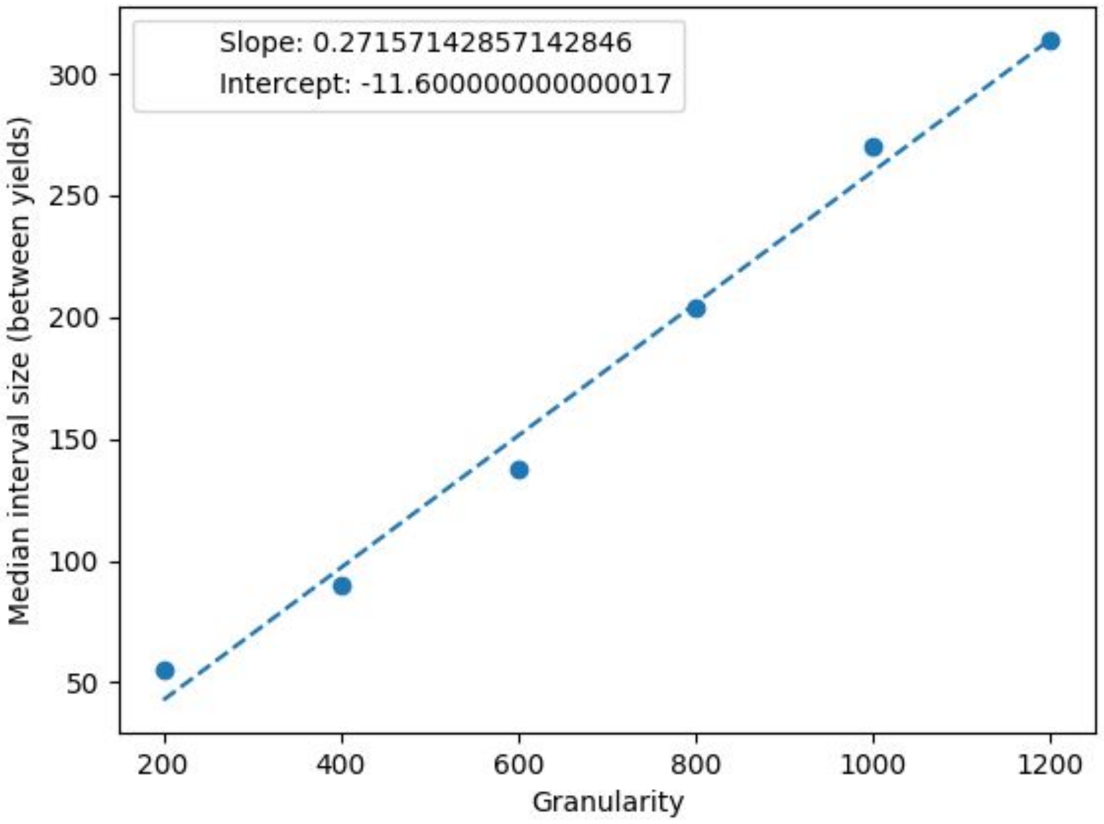


***G* = 1000**

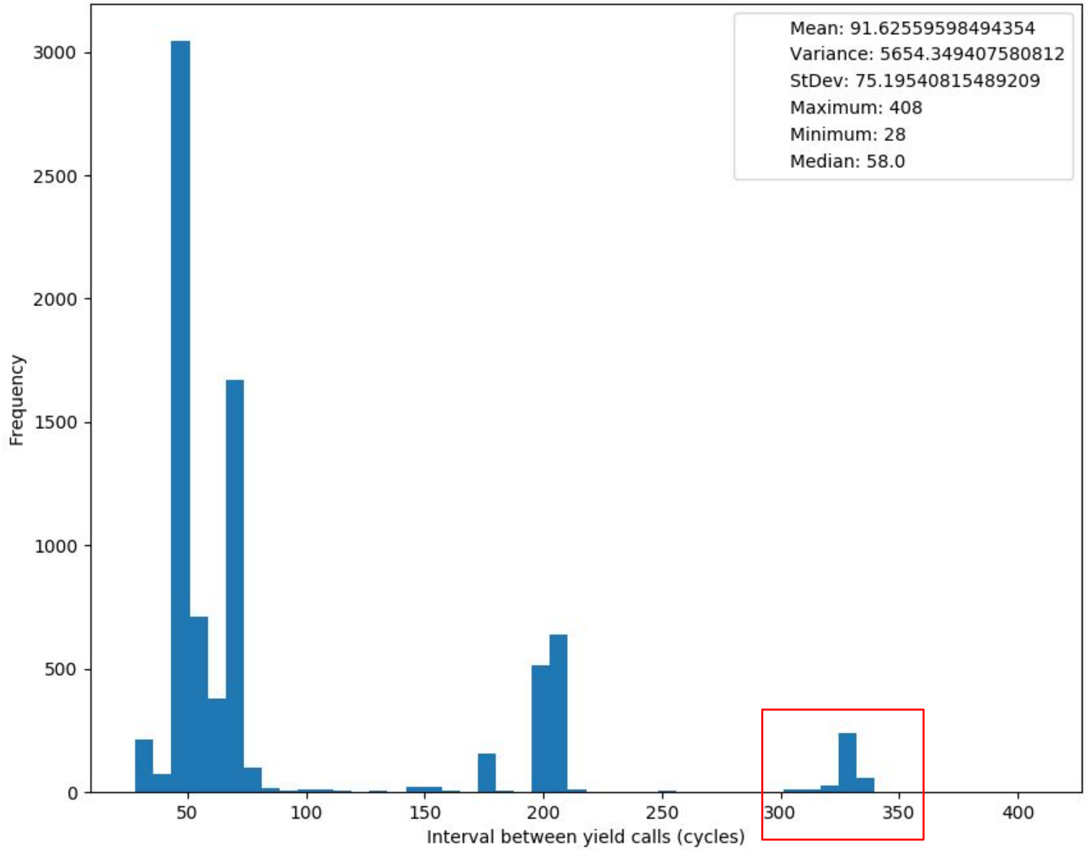
Preliminary results: Benchmark --- BST Lookup



Preliminary results: Benchmark --- LO Tree Traversal



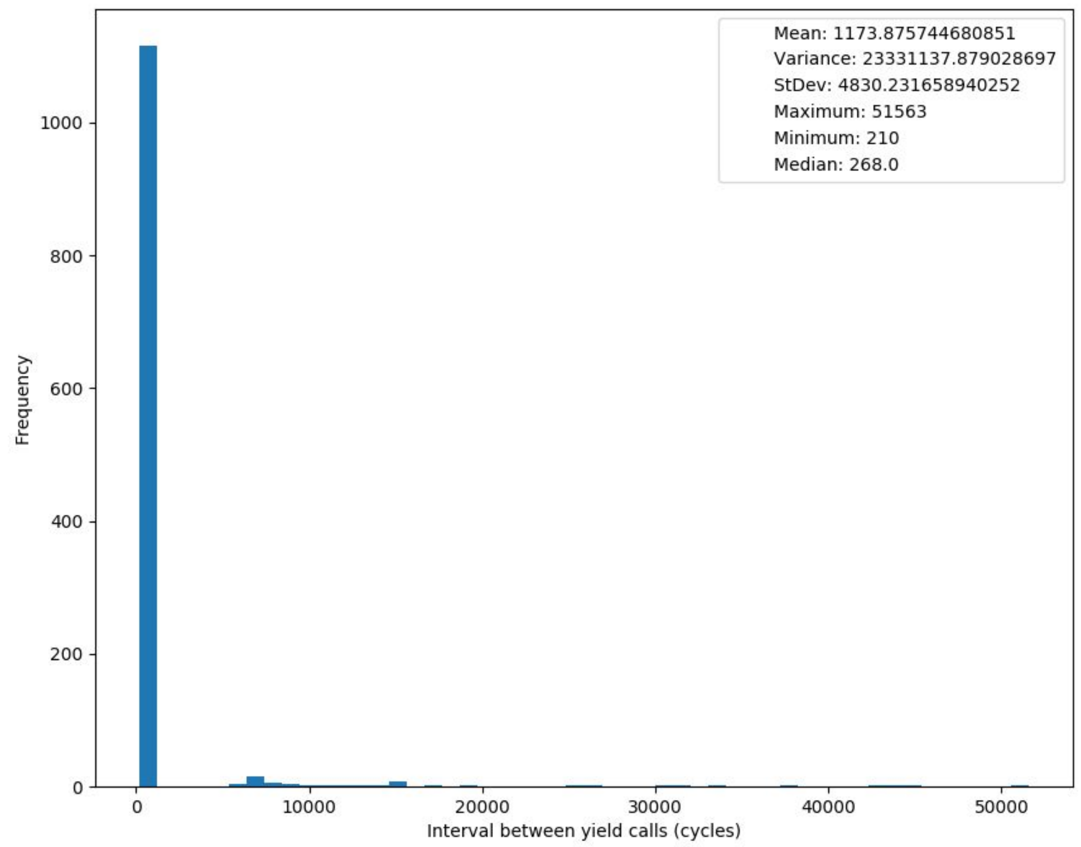
Preliminary results: Benchmark --- LO Tree Traversal



Mean: 91.62559598494354
Variance: 5654.349407580812
StDev: 75.19540815489209
Maximum: 408
Minimum: 28
Median: 58.0

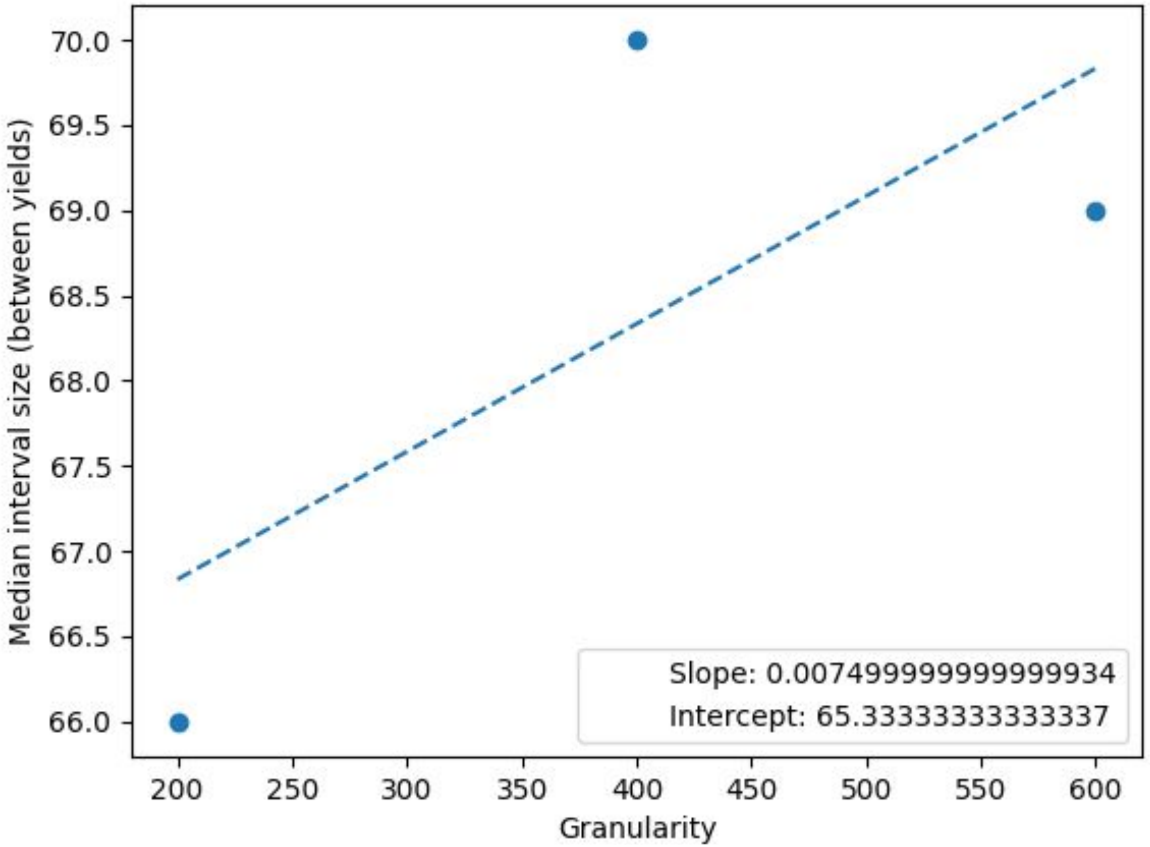
G **= 200**

Preliminary results: Benchmark --- LO Tree Traversal



***G* = 1000**

Preliminary results: Benchmark --- Matrix Multiply



Agenda

Further research: Improving on constraints

- CMU data set
- Loop “extension”

→

- LLVM profilers
- Better estimation

- Intraprocedural

→

- Interprocedural

- QEMU / KVM

→

- Hard metal, testing between settings

Further research: Expanding compiler-based timing

- Refactor the transform --- especially loop transformations
- Achieve timing guarantees at a finer granularity than standard timer interrupts --- *without misses*
- Widely integrate into Nautilus (not just fibers)
- Broader goal --- testing compiler-based timing with robust benchmark suites