Dynamic Instrumentation with Pin

Robert Cohn

Intel

Pin

- Fine-grained dynamic instrumentation of user mode programs
- Instrumentation
  - Inserting extra code to observe/change program
  - Profilers, trace collectors, …
- Dynamic
  - Done at run-time, no special compilation or linking
  - Adapt instrumentation during execution
- Fine-grained
  - Observe the execution of every instruction
  - Request instrumentation before or after any instruction execution
- Transparent
  - Instrumentation observes original program

Instruction Trace

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

VOID traceInst(VOID *ip) {
    fprintf(trace, "%p\n", ip);
}

VOID Instruction(INS ins, VOID *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)traceInst, IARG_INST_PTR,
   IARG_END);
}

int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
    return 0;
}
```

Example: Malloc Trace

SimpleExamples/malloctrace.C

```
VOID Image(IMG img, VOID *v) {
    RTN mallocRtn = RTN_FindByName(img, "malloc");

    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn); // fetch insts in mallocRtn

        RTN_InsertCall(mallocRtn, IPOINT_BEFORE,
               (AFUNPTR)Arg1Before,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);

        RTN_InsertCall(mallocRtn, IPOINT_AFTER,
               (AFUNPTR)MallocAfter,
               IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);

        RTN_Close(mallocRtn);
    }
}
```

before malloc's entry

1st argument to malloc (#bytes wanted)

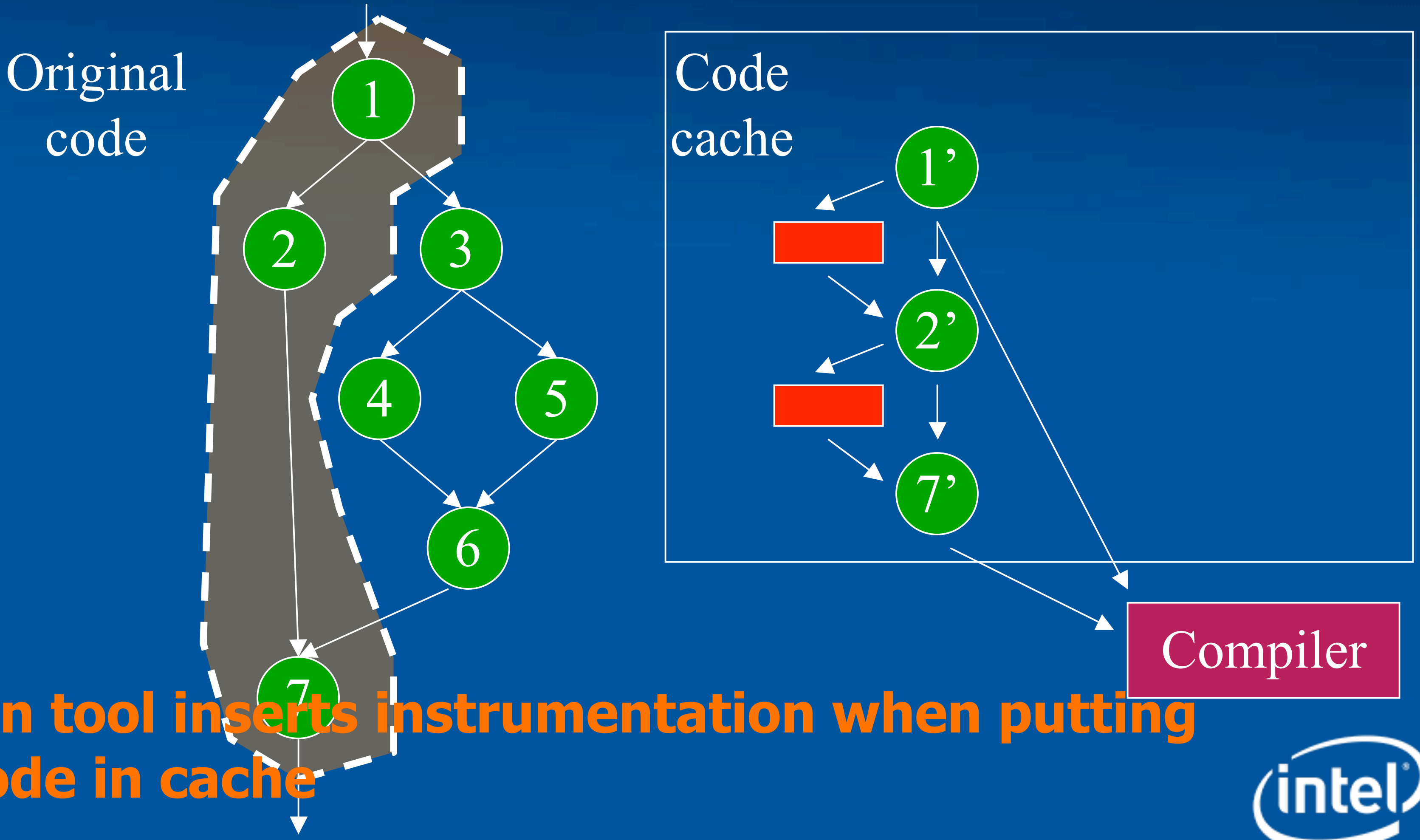
before malloc's return

1st return value (address allocated)

Instrumentation Philosophy

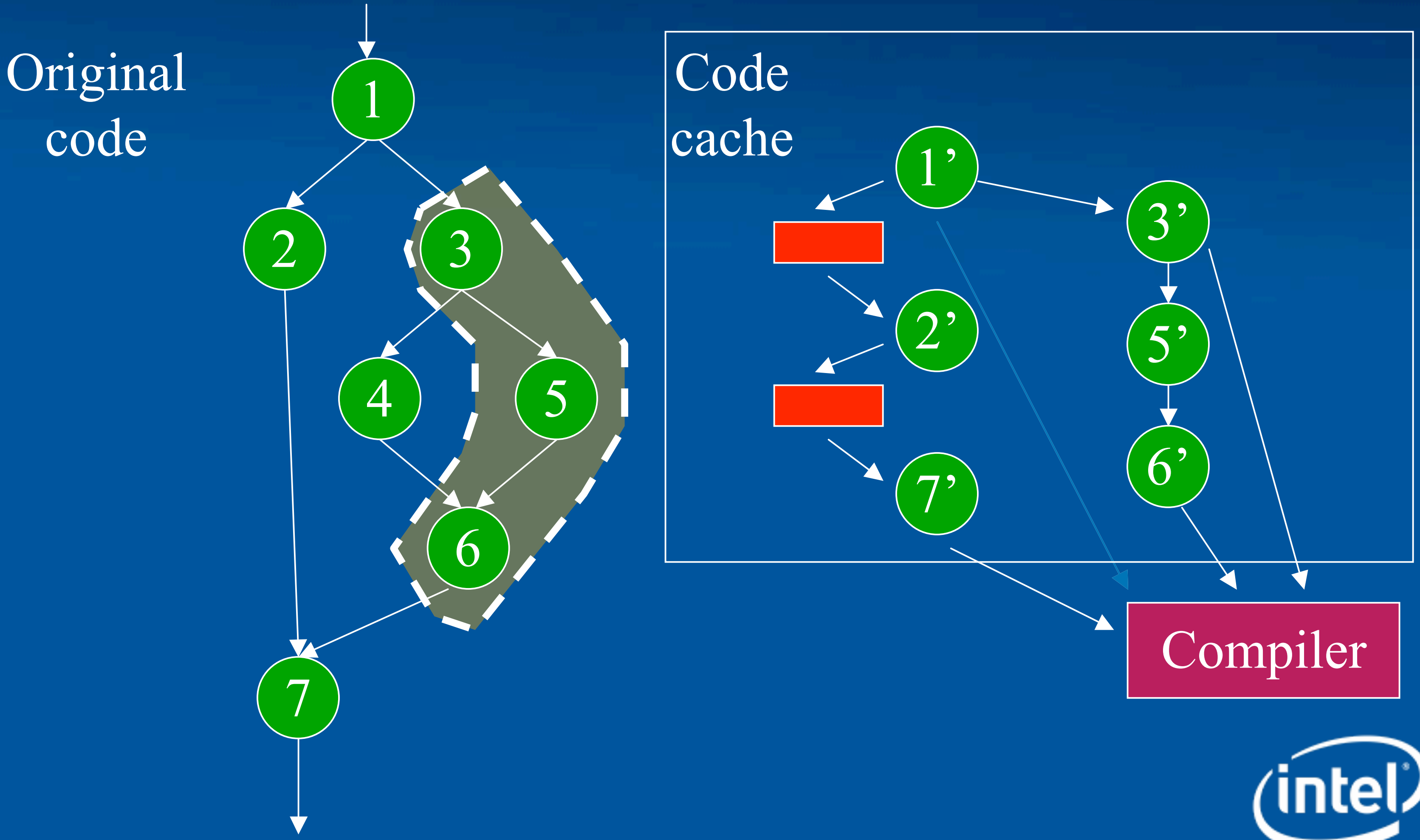
- Tools view instruction list of application instructions
- Users only insert function calls
- No general code modification ability for tools
- Try to close gap by inlining and optimization of instrumentation
- Still a gap in altering control flow

Just-in-time Instrumentation



Pin tool inserts instrumentation when putting code in cache

Just-in-time Instrumentation



JIT-based Instrumentation Features

- Handles mixed code and data, variable alignment, variable size instructions with 100% accuracy
- Maintains control at all times, change/observe anything
- Only instrument executed code
  - Database server code is 60Meg + shared libraries
- No special handling for shared libraries
- Handles dynamically generated code
- No dependence on compiler or binary format
  - applies to instrumentation engine, but tools may need to access symbol information
- Trace based optimization of instrumentation

JIT Based Instrumentation Drawbacks

- Time overhead 0% - 300%

- Hardware counters may not be give useful information

Probe-based instrumentation

- Overwrite original program with probe (branch) to reach dyninst-style trampolines
- Mostly execute original program, enter instrumentation via probes
- Subset of API
  - Started with very general capability
  - Replaced with straight jacket - wrap function calls to observe or alter behavior
- Near zero overhead and perturbation
- Relevant timing, hardware counter data
- Weak code and CFG discovery
  - Sufficient for tools that watch API usage
    - MPI trace analysis
    - Memory allocation errors
- Shares compiler & injector with JIT based instrumentation

Details

- IA32, Intel 64, IA64
- Linux, Windows, MacOs
- No charge
  - But not open source
- BSD-like license
  - no restrictions on use or redistribution
  - Instrumentation vm distributed as binary
  - Sample tools are open source
- Download it at http://rogue.colorado.edu/Pin
  - 600 downloads/month

Pin Users

- Microprocessor development
  - Fast & easy to extend emulator
- Intel Software Quality and Performance Analysis Products
  - Emphasis on parallelism
- ISV
  - Software quality & performance tools
- University
  - research & education

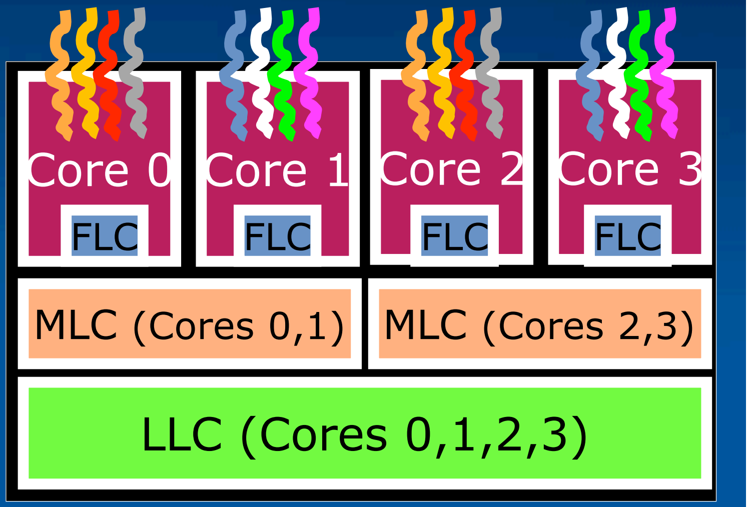
Microprocessor Development

- Model performance of hardware that does not exist
- Instrumentation based tools are fast and easy to develop

- CMP$im – memory system performance modeling
- EMX – instruction emulation
- PinPoints, PinPlay – workload capture

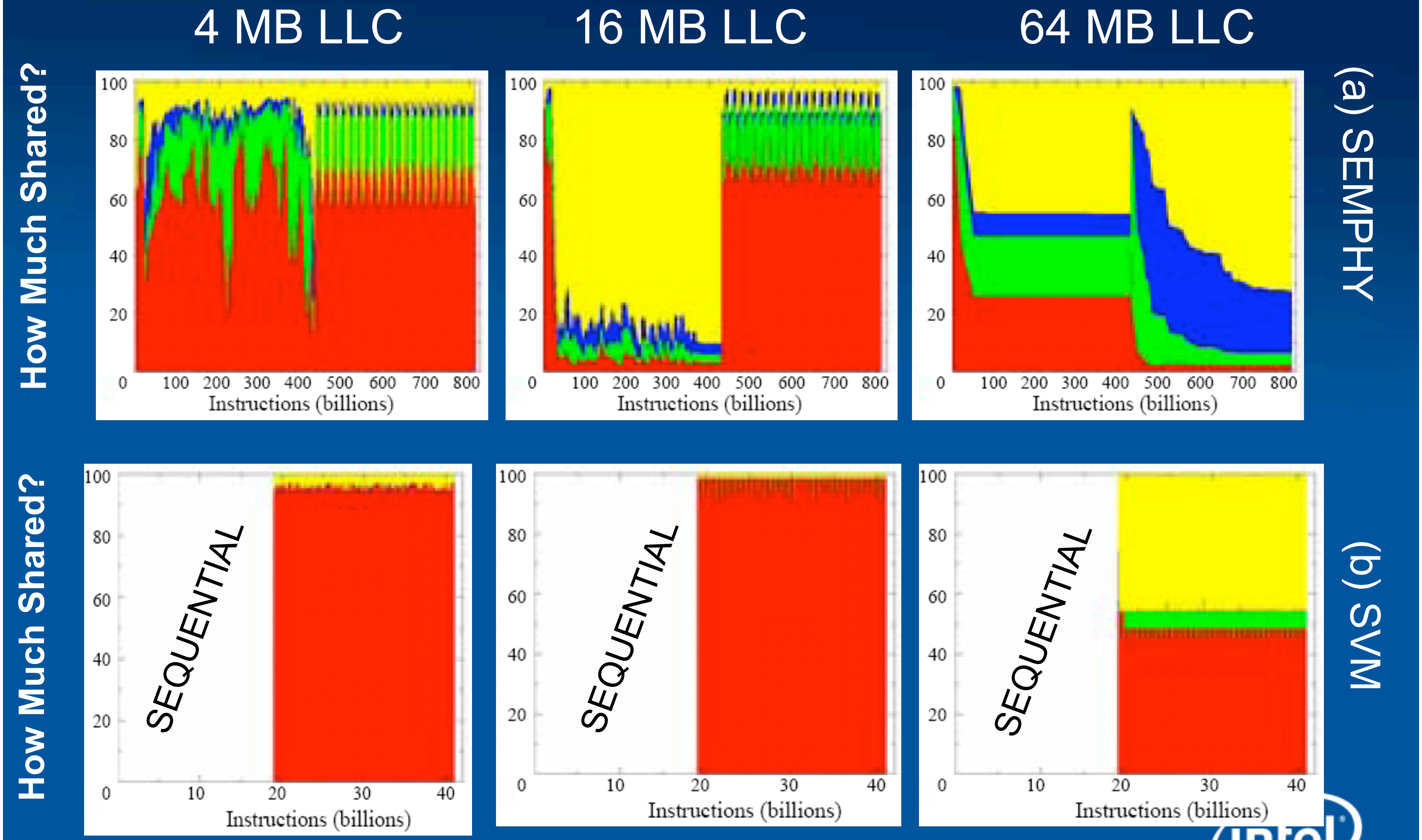
CMP$im Features

- Use Pin to instrument all loads and store
- Fast Memory Characterization:
  - Single/multi-threaded workloads
  - 4-25 MIPS (100x-800x slow)
- Memory System Configurations:
  - Model private/shared caches
  - Model single/hyper-threaded cores
  - Model inclusive or non-inclusive caches
- Statistics:
  - Detailed instruction/cache statistics
  - View phase behavior of workloads



4-threaded, 4-core, CMP
Multi-Level Cache Sharing:
4-threads per FLC
8-threads per MLC
16-threads per LLC

Sharing Phase Dependent & *f* (cache size)



4 Threaded Run: 🟥 1 Thread 🟩 2 Thread 🟦 3 Thread 🟨 4 Thread

EMX - Instruction Emulation

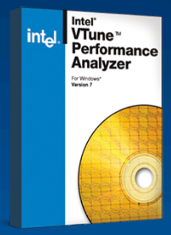
- Architectural evaluation requires extensions to existing instruction sets
  - Debug compiler & libraries
  - Performance evaluation
- EMX pintool
  - Instrumentation replaces new instructions with emulation functions
  - Near native speed for everything else
  - Can use instrumentation to study programs that use extensions

Software Quality & Performance Analysis

- Observe
  - Pin instruments loads and stores, control flow
- Analyze data
- Present information
  - Pin provides symbol/debug info

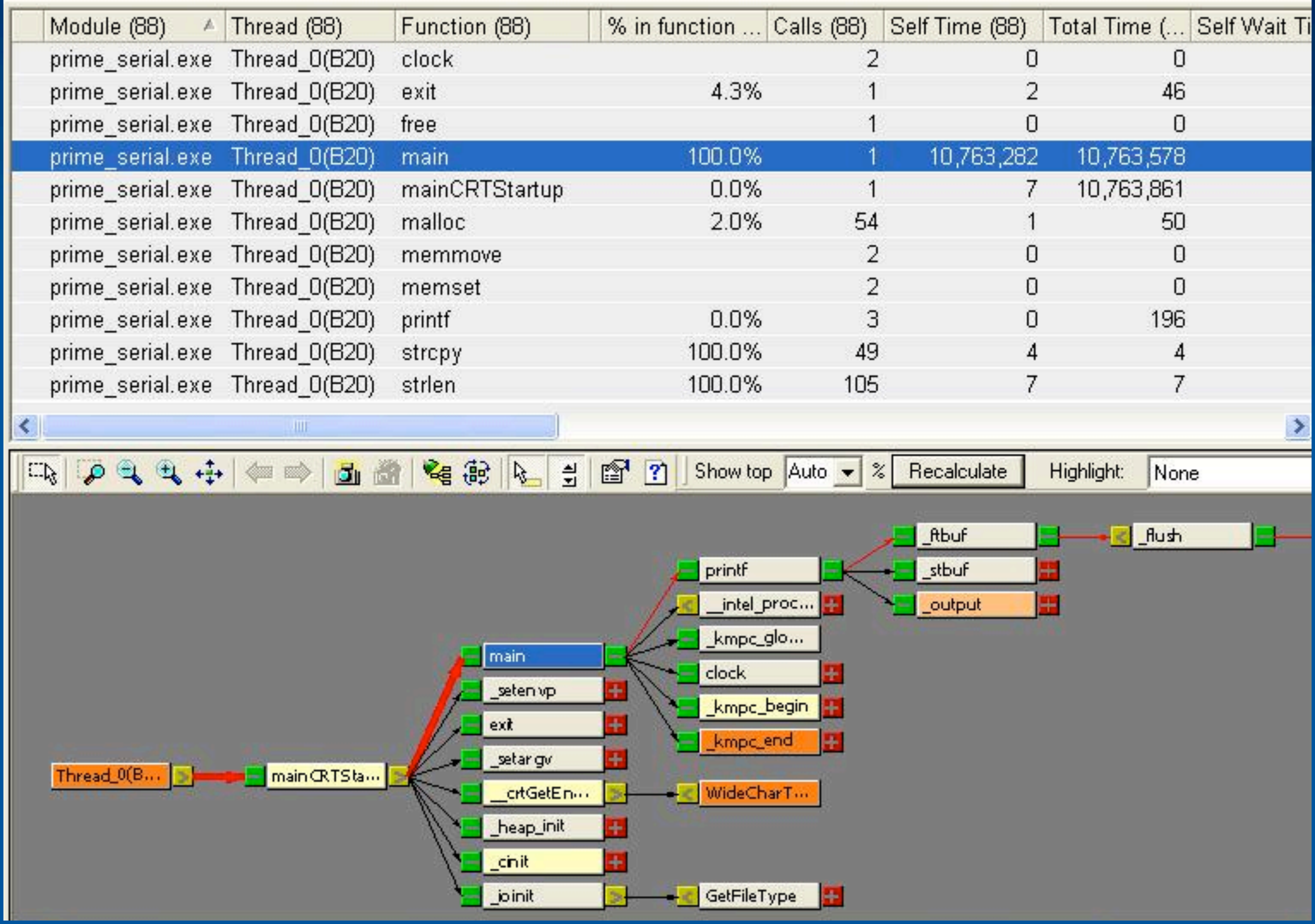
- Many tools use both probe-based instrumentation for speed and JIT-based instrumentation for detailed analysis

Intel® VTune™ Performance Analyzer



- Call graph
- Other profilers

Vtune™ Call Graph Profile

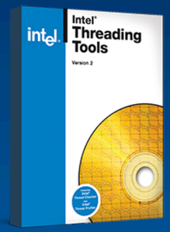


Module (88)	Thread (88)	Function (88)	% in function ...	Calls (88)	Self Time (88)	Total Time (...	Self Wait Ti
prime_serial.exe	Thread_0(B20)	clock		2	0	0	
prime_serial.exe	Thread_0(B20)	exit	4.3%	1	2	46	
prime_serial.exe	Thread_0(B20)	free		1	0	0	
prime_serial.exe	Thread_0(B20)	main	100.0%	1	10,763,282	10,763,578	
prime_serial.exe	Thread_0(B20)	mainCRTStartup	0.0%	1	7	10,763,861	
prime_serial.exe	Thread_0(B20)	malloc	2.0%	54	1	50	
prime_serial.exe	Thread_0(B20)	memmove		2	0	0	
prime_serial.exe	Thread_0(B20)	memset		2	0	0	
prime_serial.exe	Thread_0(B20)	printf	0.0%	3	0	196	
prime_serial.exe	Thread_0(B20)	strcpy	100.0%	49	4	4	
prime_serial.exe	Thread_0(B20)	strlen	100.0%	105	7	7	

Show top Auto %

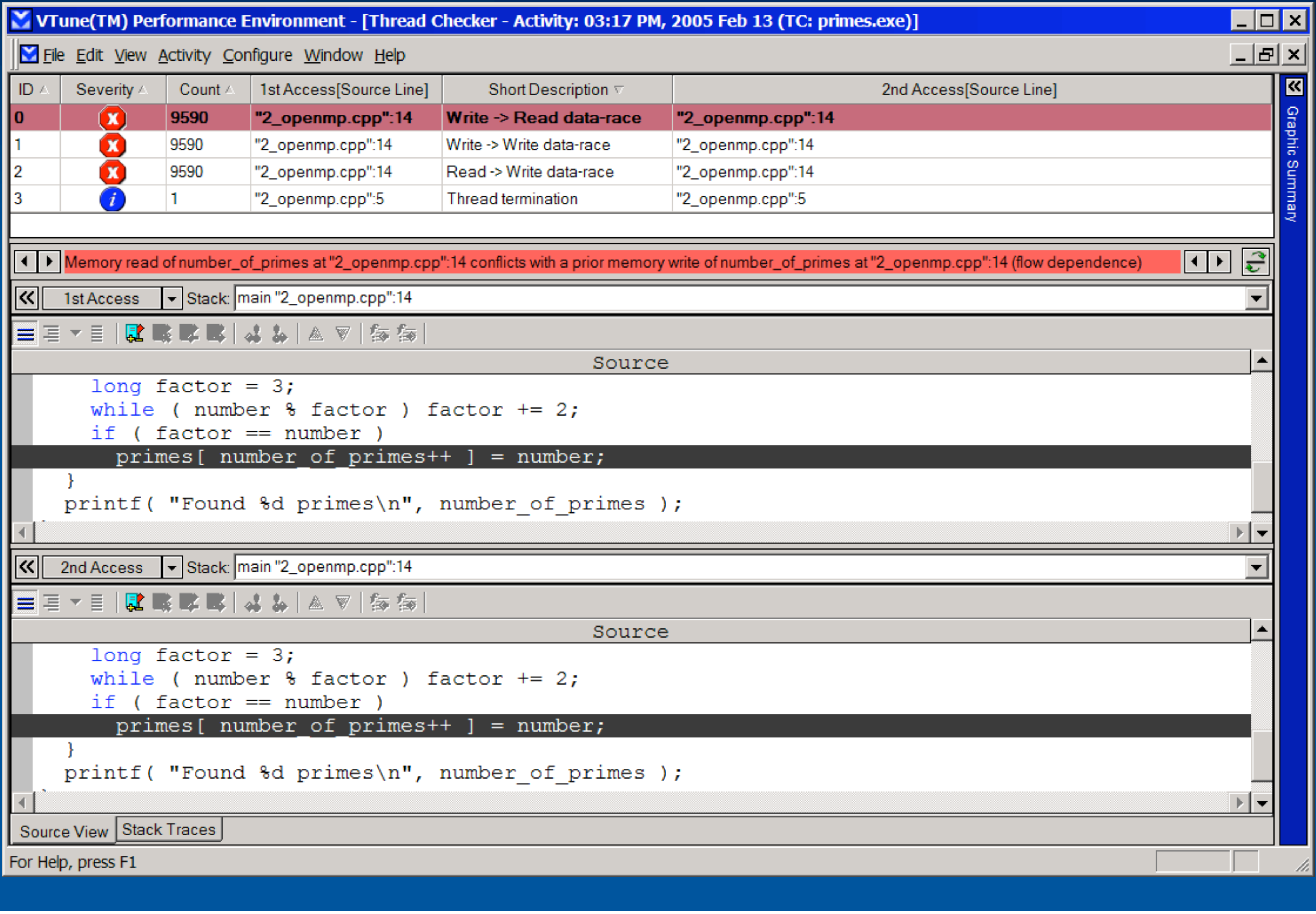
Highlight: None

Intel® Thread Checker



- Detects threading bugs
  - Data Races
  - Deadlocks
- Instruments loads, stores, threading API

Intel® Thread Checker



VTune(TM) Performance Environment - [Thread Checker - Activity: 03:17 PM, 2005 Feb 13 (TC: primes.exe)]
File Edit View Activity Configure Window Help
Graphic Summary

ID	Severity	Count	1st Access[Source Line]	Short Description	2nd Access[Source Line]
0	X	9590	"2_openmp.cpp":14	Write -> Read data-race	"2_openmp.cpp":14
1	X	9590	"2_openmp.cpp":14	Write -> Write data-race	"2_openmp.cpp":14
2	X	9590	"2_openmp.cpp":14	Read -> Write data-race	"2_openmp.cpp":14
3	i	1	"2_openmp.cpp":5	Thread termination	"2_openmp.cpp":5

Memory read of number_of_primes at "2_openmp.cpp":14 conflicts with a prior memory write of number_of_primes at "2_openmp.cpp":14 (flow dependence)

1st Access Stack: main "2_openmp.cpp":14

Source

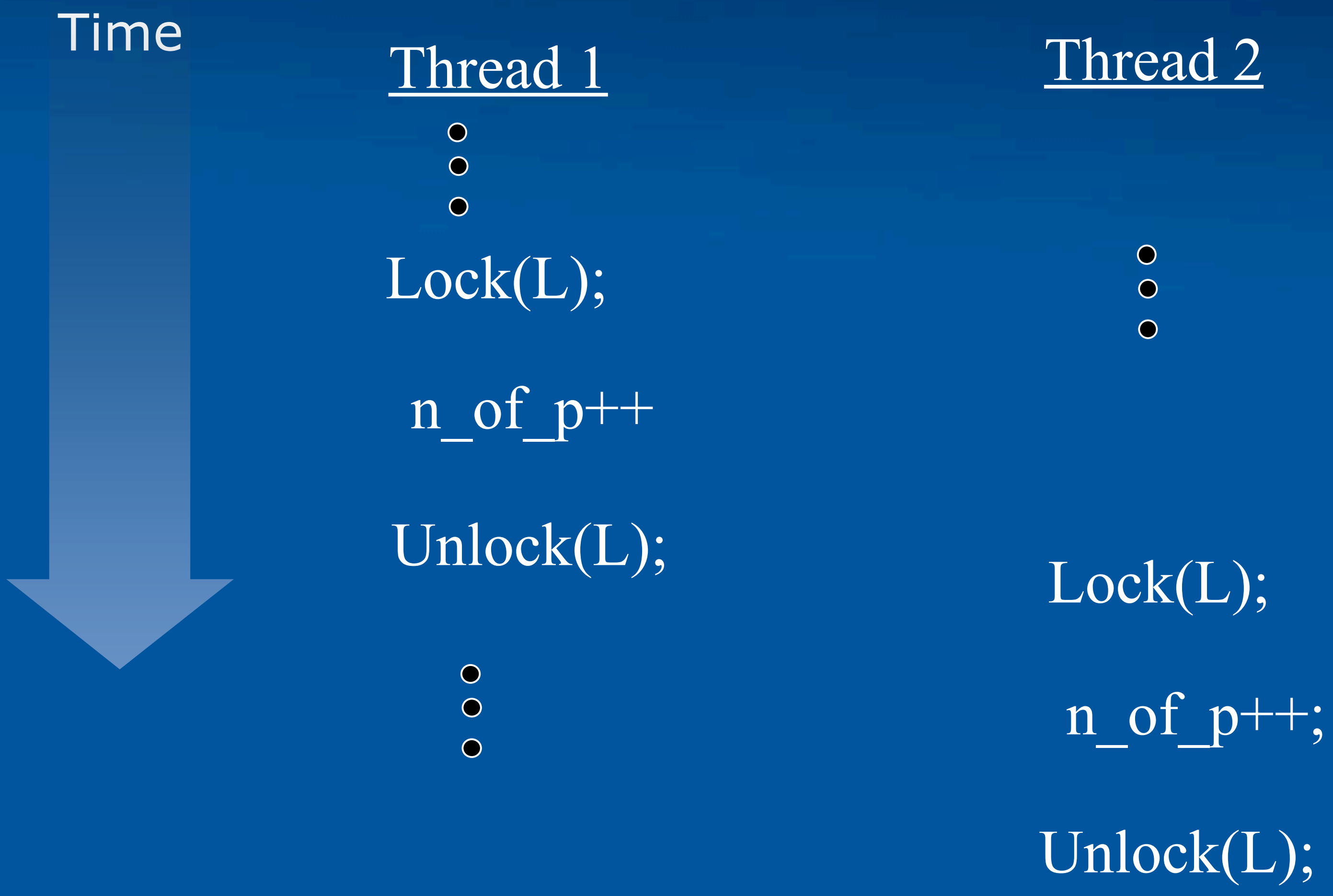
```
    long factor = 3;
    while ( number % factor ) factor += 2;
    if ( factor == number )
      primes[ number_of_primes++ ] = number;
  }
  printf( "Found %d primes\n", number_of_primes );
```

2nd Access Stack: main "2_openmp.cpp":14

Source

```
    long factor = 3;
    while ( number % factor ) factor += 2;
    if ( factor == number )
      primes[ number_of_primes++ ] = number;
  }
  printf( "Found %d primes\n", number_of_primes );
```

How Does Thread Checker Work?



Thread 1

⋮

Lock(L);

n_of_p++

Unlock(L);

⋮

Thread 2

⋮

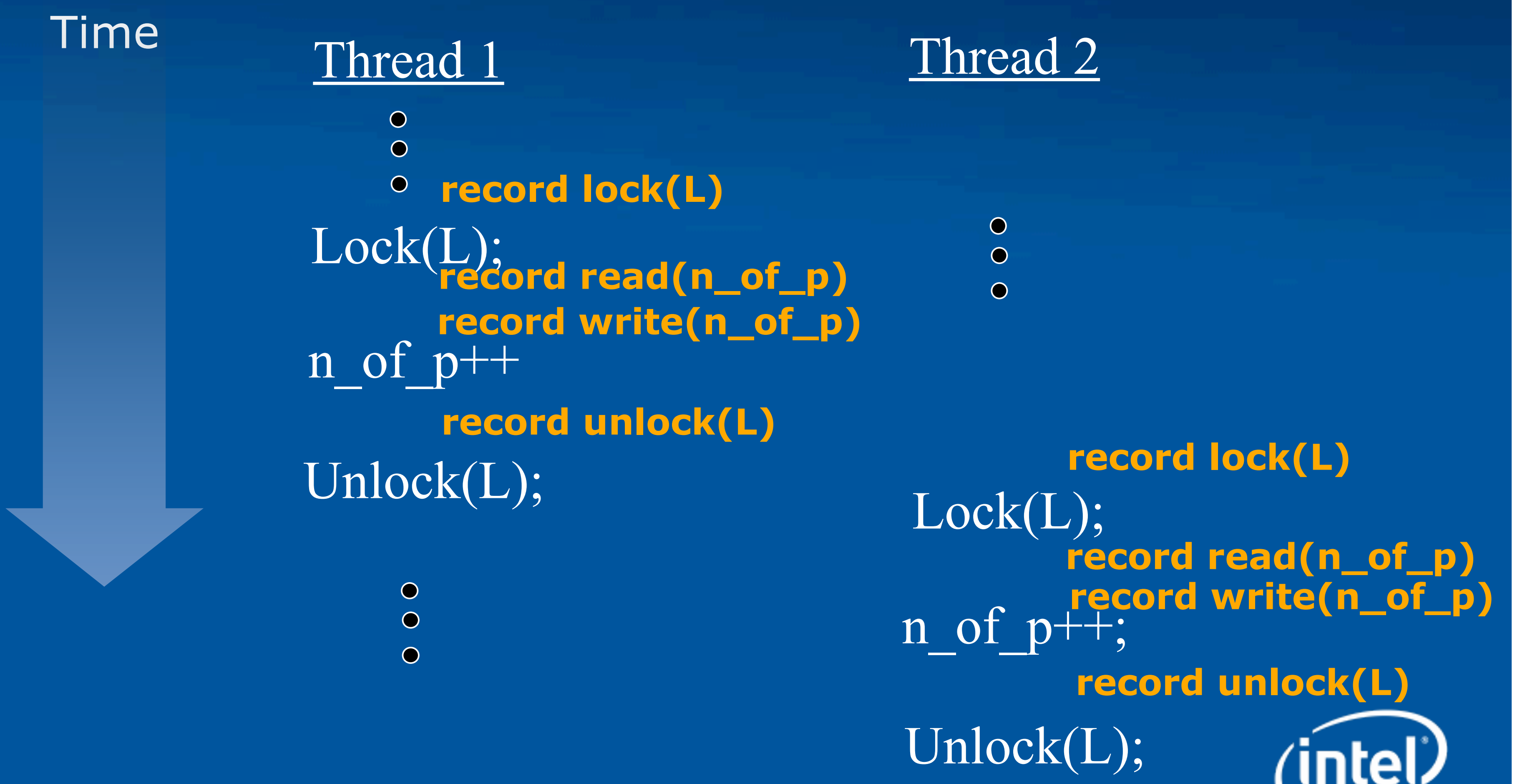
Lock(L);

n_of_p++;

Unlock(L);

How Does Thread Checker Work?

Use binary instrumentation



Time

Thread 1

●
●
●

record lock(L)

Lock(L);

record read(n_of_p)

record write(n_of_p)

n_of_p++

record unlock(L)

Unlock(L);

●
●
●

Thread 2

●
●
●

record lock(L)

Lock(L);

record read(n_of_p)

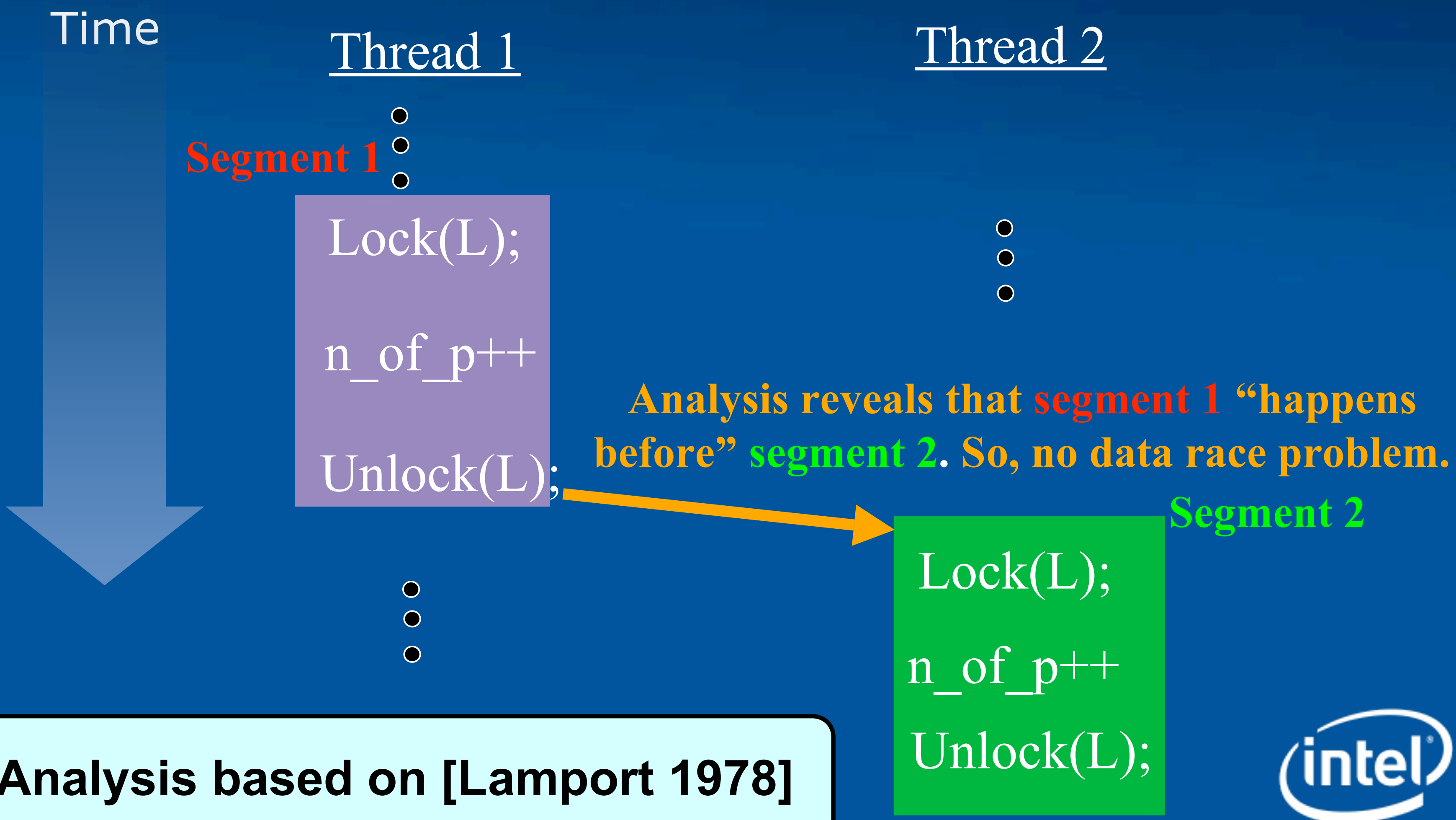
record write(n_of_p)

n_of_p++;

record unlock(L)

Unlock(L);

How Does Thread Checker Work?



Time

Thread 1

Segment 1

```
Lock(L);
n_of_p++
Unlock(L);
```

Thread 2

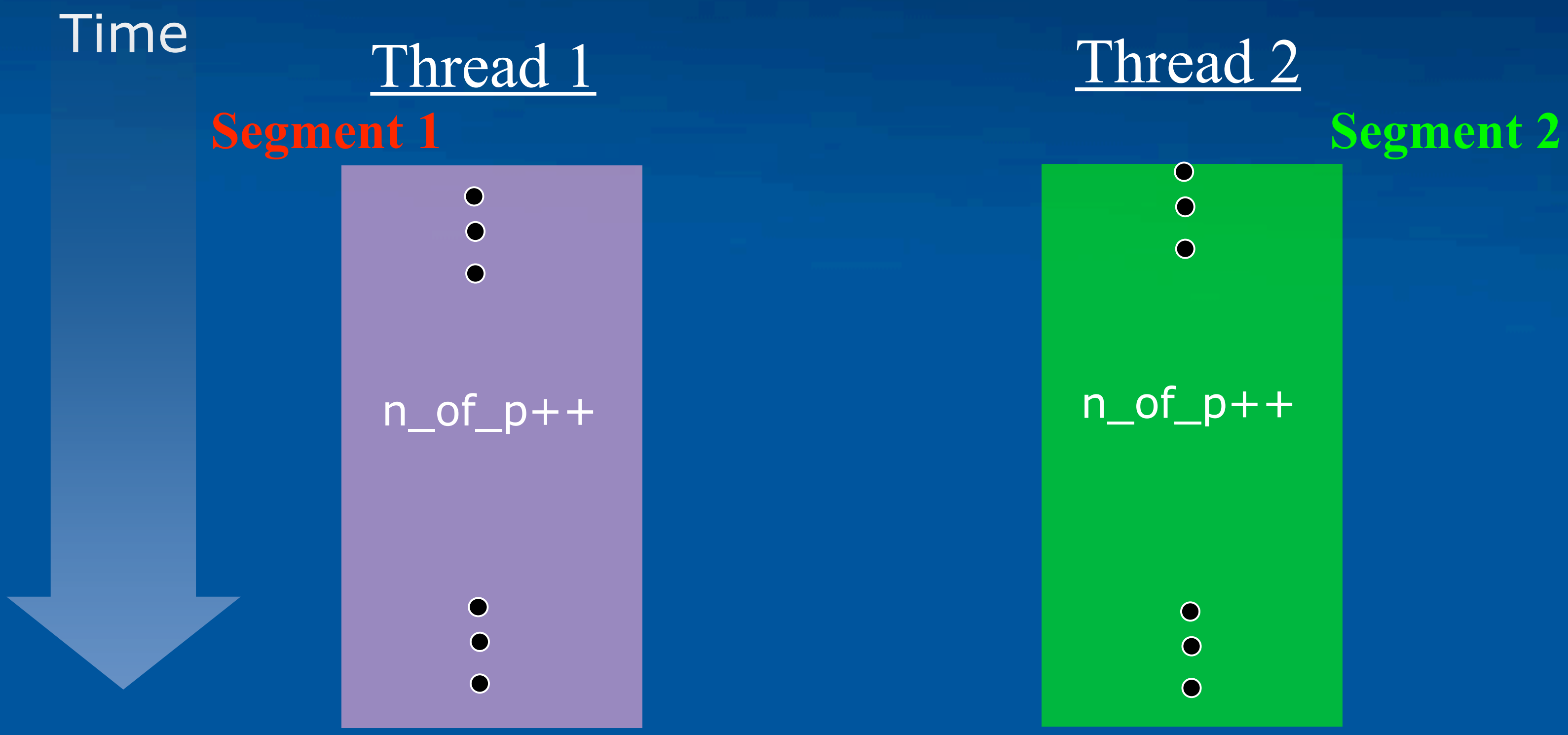
Analysis reveals that segment 1 “happens before” segment 2. So, no data race problem.

Segment 2

```
Lock(L);
n_of_p++
Unlock(L);
```

Analysis based on [Lamport 1978]

How Does Thread Checker Work?



Time

Thread 1

Segment 1

n_of_p++

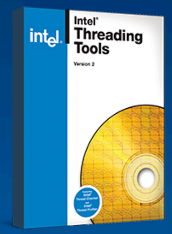
Thread 2

Segment 2

n_of_p++

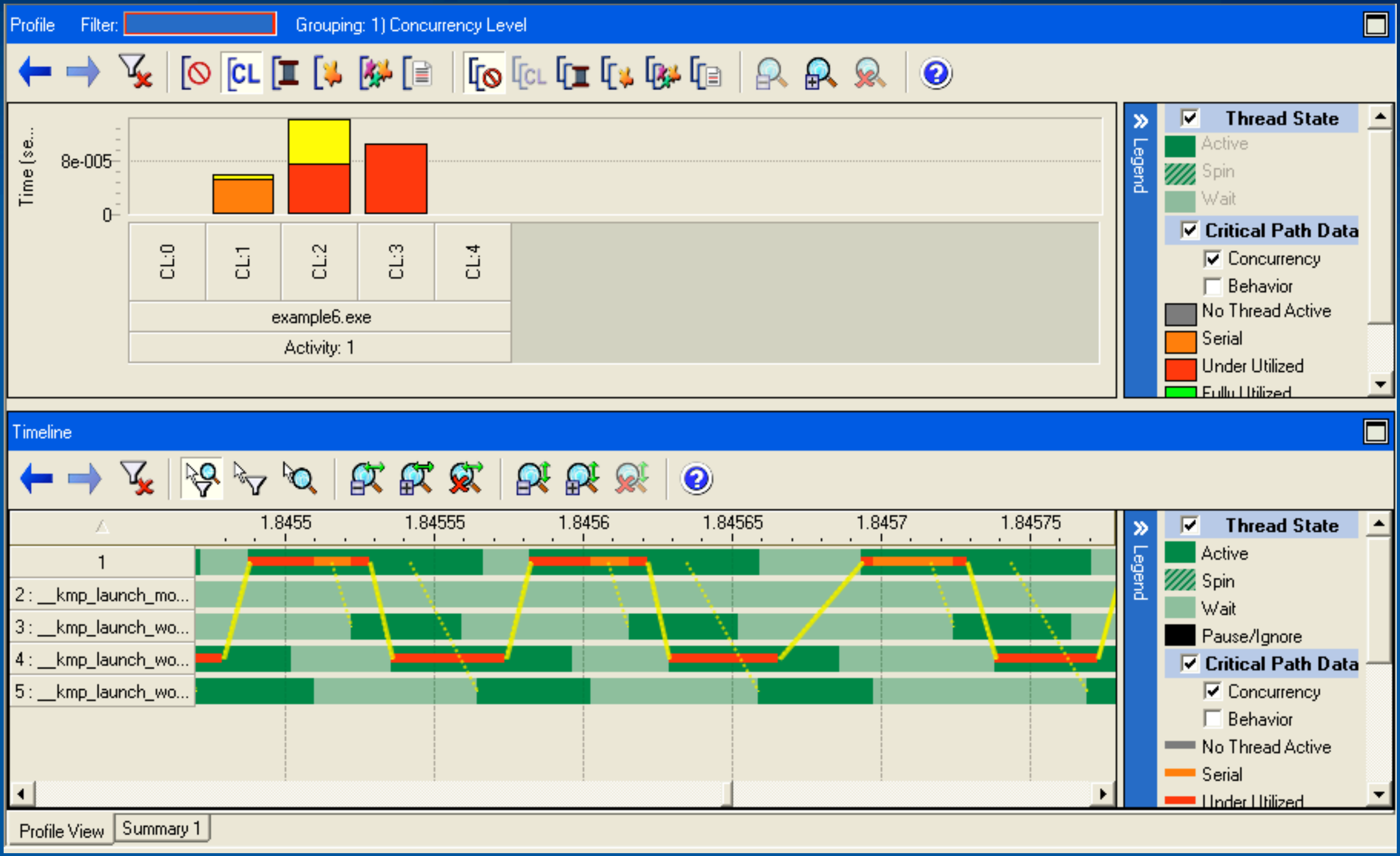
With no locks, neither segment "happens before" the other. So there is a data race!

Intel® Thread Profiler

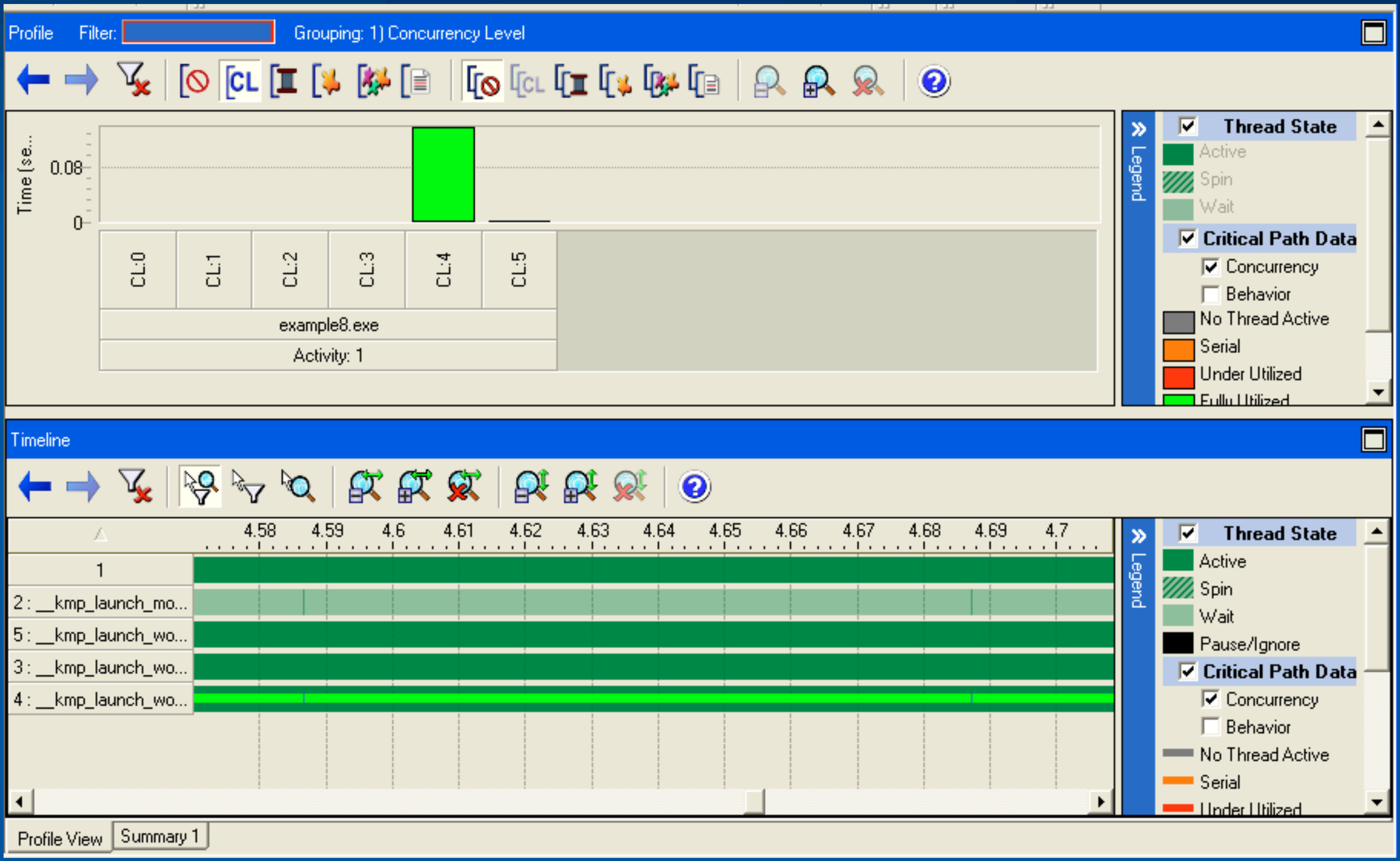


- Find Contended Locks
  - Most Overhead
  - Largest Reduction in Parallelism
- Probe-based instrumentation of threading API's

Using Thread Profiler



Using Thread Profiler

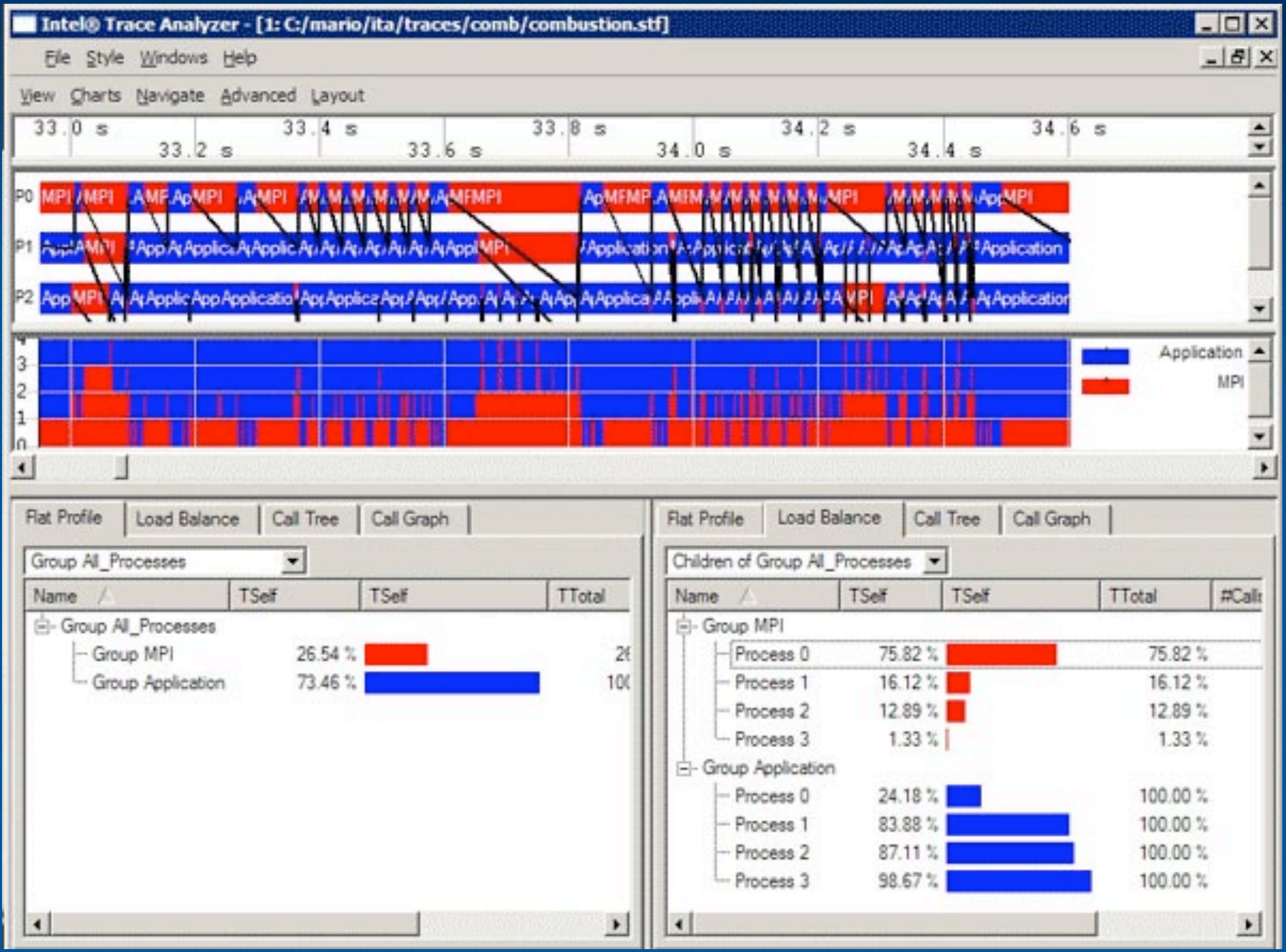


Grouping: 1) Concurrency Level

Intel Trace Analyzer and Collector

- Collects MPI trace
- Correctness checking of usage
- Analysis for optimization
- Probe-based to instrument MPI calls
- JIT-based for precise call stacks

Intel Trace Analyzer and Collector



Group All_Processes

Name	TSelf	TSelf	TTotal
Group All_Processes			
Group MPI	26.54 %	🟥	26
Group Application	73.46 %	🟦	10(

Children of Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls
Group MPI				
Process 0	75.82 %	🟥	75.82 %	
Process 1	16.12 %	🟥	16.12 %	
Process 2	12.89 %	🟥	12.89 %	
Process 3	1.33 %	🟥	1.33 %	
Group Application				
Process 0	24.18 %	🟦	100.00 %	
Process 1	83.88 %	🟦	100.00 %	
Process 2	87.11 %	🟦	100.00 %	
Process 3	98.67 %	🟦	100.00 %	

Cooperation

- Pin use of external components:
  - Cannot use GPL libraries, but LGPL OK
  - Symtab
  - Unwind
  - Code and CFG discovery (e.g. bloop)
  - Thread safe instrumentation tool runtime (libc)
    - No dependencies on system
- Intel contributions:
  - Pin is binary-only distribution
  - Difficult to provide binaries for non Intel ISA
  - XED IA32, Intel 64, AMD64 encoder/decoder/disassembler
  - Open source components? E.g. injector

XED encoder/decoder/disassembler

- Used in projects other than Pin
- Includes all public ISA extensions
- Correct
  - Only decode what really is an instruction
  - Get all the operands correct
  - read/write/conditional, size, register type, ...
  - Only encode well formed encode requests
- Fast, Small
- Thread safe
- Distributed as library in pin kit with manual

University Relations

- Close interaction with:
  - Harvard
  - MIT
  - U Colorado Boulder
  - U Virginia
- Internships
  - Self contained projects
    - Reliability, persistence
  - Adds support in pin for projects continued at university
    - Checkpoint/restart to support simulation

Summary

- Some instrumentation tools need full control and ability to instrument every instruction
  - JIT-based instrumentation
- Other tools need low overhead
  - Native execution with probes
- Both styles of instrumentation share functionality