Using PinPlay for Reproducible Analysis and Replay Debugging

Harish Patil, Cristiano Pereira, Charles(Chuck) Yount

Intel Corporation

Milind Chabbi, Rice University

With contributions from:

Mack Stallcup, Greg Lueck, Tevi Devor, David Wootton (Intel Corporation)

PLDI 2015 Tutorial, Portland, June 14th , 2015

Tutorial Objective

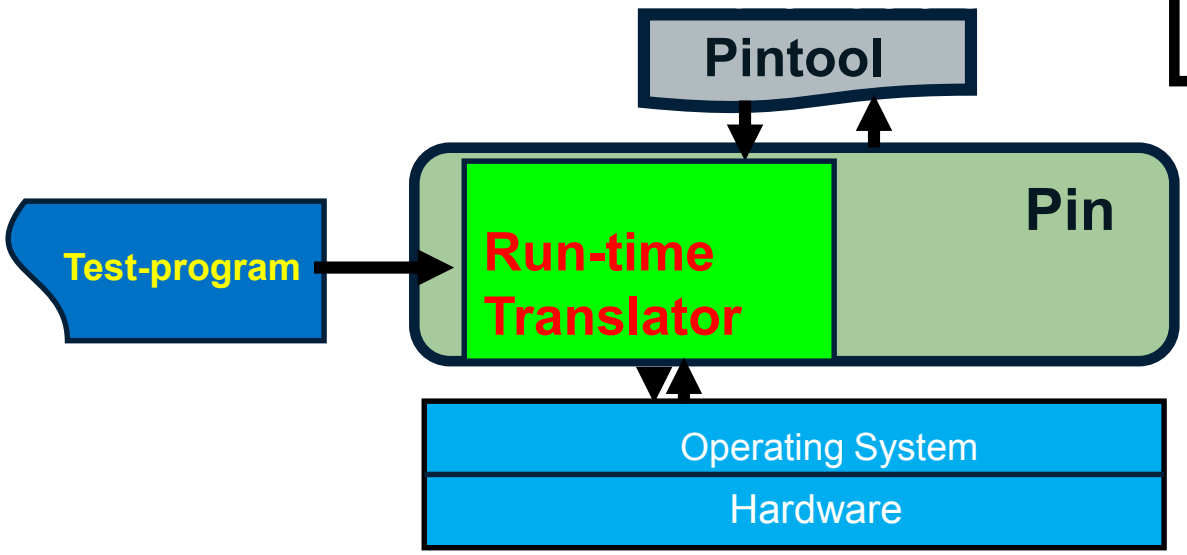
*To show that PinPlay is an **easy-to-use**, **flexible**, and **effective** framework for reproducible analysis of parallel programs.*

1. How PinPlay works
2. How to use it for reproducible analysis (e.g. debugging, dynamic program slicing, DCFG)

Pin: A Tool for Writing Program Analysis Tools

```
sub     $0xff, %edx
movl    0x8(%ebp), %eax
jle     <L1>
```

```
counter++; print(IP)
sub     $0xff, %edx
counter++; print(EA)
movl 0x8(%ebp), %eax
counter++;print(br_taken)
jle     <L1>
```




```
$ pin -t pintool -- test-program
```

Normal output + *Analysis output*

Pin: A Dynamic Instrumentation Framework from Intel
http://www.pintool.org

Schedule

9:00 -- 9:45 PinPlay and PinADX

9:45 – 10:15 DrDebug demo

10:15 – 10:30 Testimonial (Milind)

10:30 – 11:00 Slicing demo

<11 – 11:20 Break>

11:20 – 11:50 DCFG (Chuck)

11:50 – 12:15 Maple (Cristiano)

12:15 – 12:30 Wrap-up + Q&A (all)

The Need : Easier Analysis of Parallel Programs

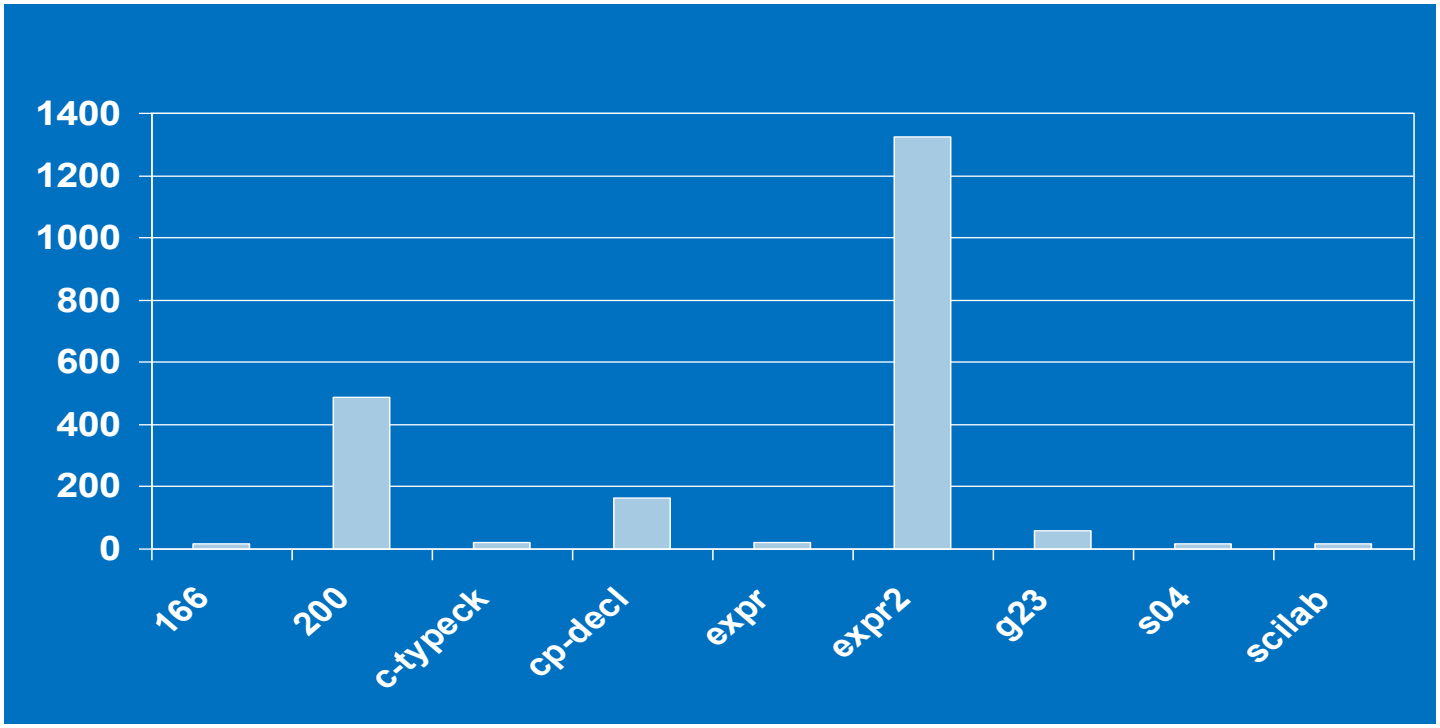
Programmers need a way to deterministically analyze and debug parallel programs

Why?

Run-to-run variation ➔ Chasing a moving target

Run-to-run Control-flow Divergence in 403.gcc (SPEC2006)

# of divergences between two runs

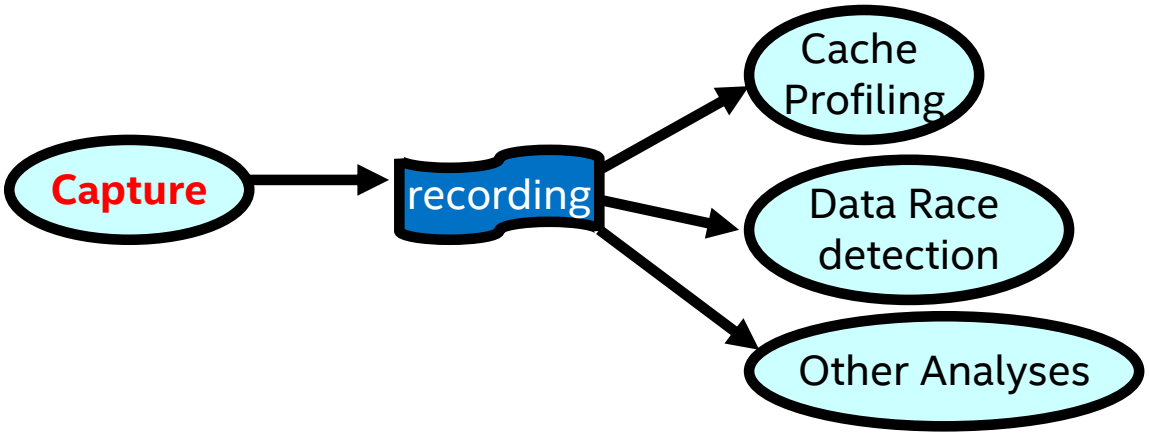


8 'ref' inputs for gcc

Measured using an execution differencing tool based on PinPlay [Bert Maher]

Even Single-threaded program runs are not repeatable!

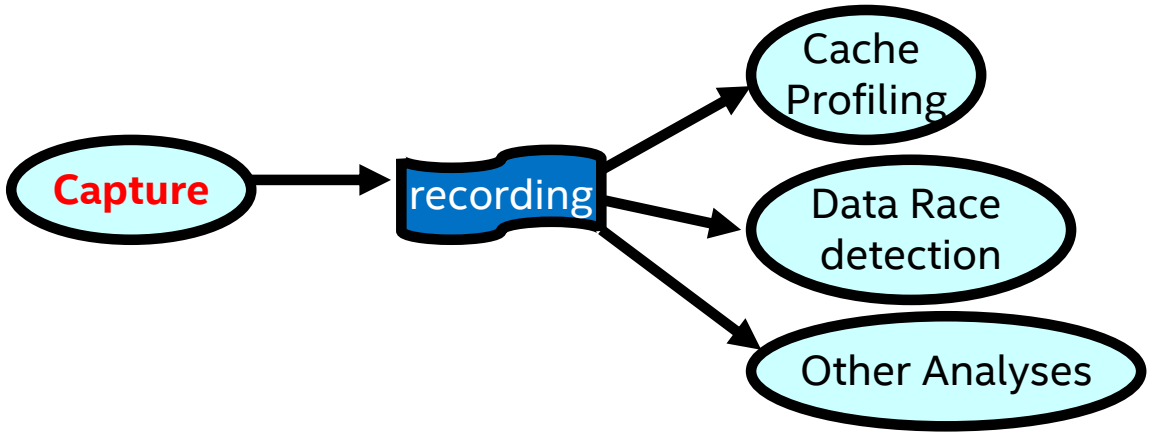
The Solution :
Record → Replay & Analyze



Capture an execution and replay it deterministically with analysis

Grand Vision Capture once Analyze Multiple times! **Anywhere!**

Expensive analyses can be delayed till replay time with guaranteed repeatability



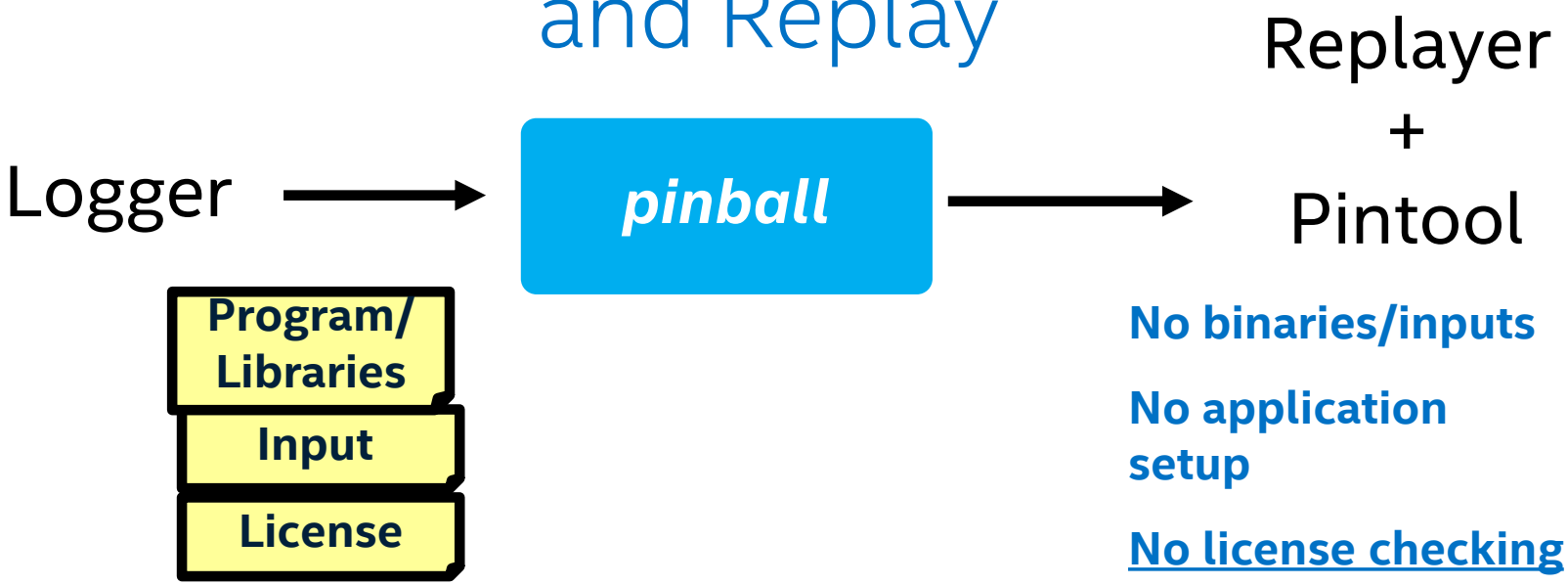
Single Machine → NetBatch

2.4 Kernel → 2.6 Kernel

Windows → Linux

Customer site → Developer site

PinPlay : Software-based User-level Capture and Replay

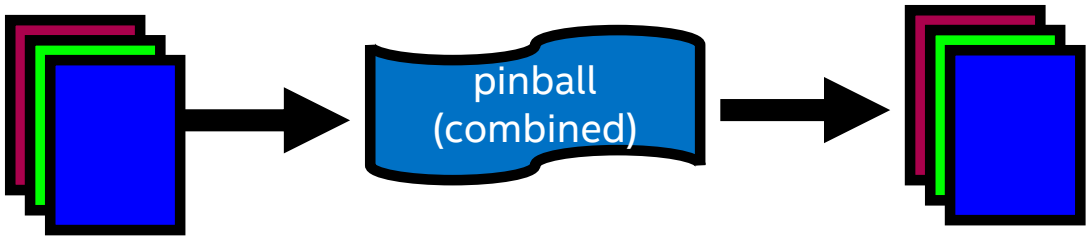


Platforms : Linux, Windows, Android, MacOS

Upside : It works! Large OpenMP / MPI programs, Oracle

Downside : High run-time overhead: ~100-200X for capture ➔ Cannot be turned on all the time

Model 1: Parallel Capture : Parallel Replay
For Multi-threaded Programs

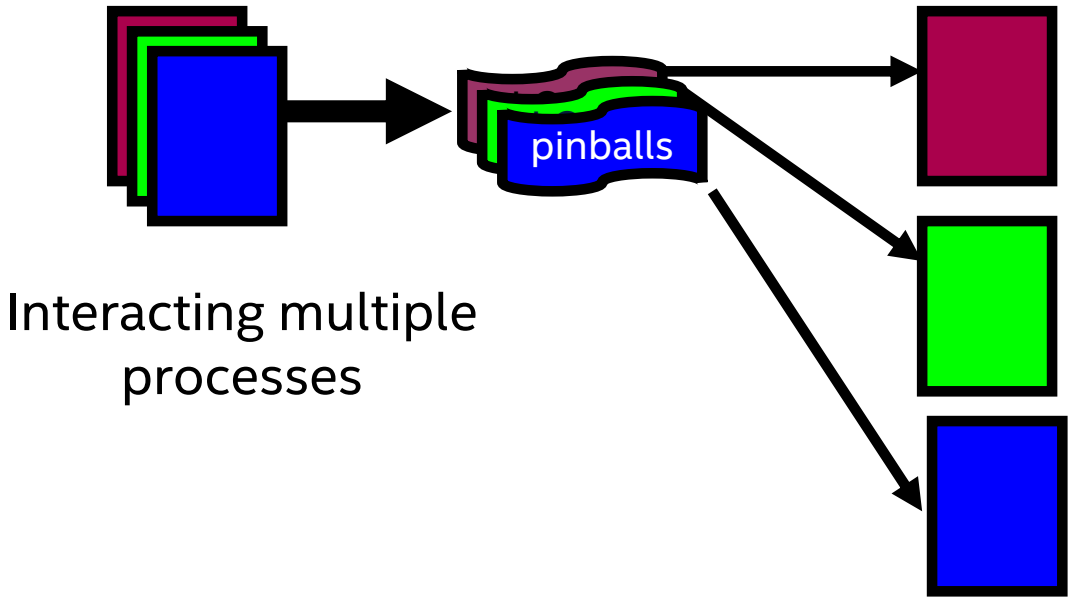


Interacting multiple threads

Replay: Interacting multiple threads

Useful for parallel analysis/debugging

Model 2: Parallel Capture : Isolated Replay For Multi-process Programs



- **multi-process ➔ multi-programmed**

Useful for tracing/simulation : can focus on one process

PinPlay : History & Acknowledgements

Motivation : Repeatable ***PinPoints***

Inspiration: ***BugNet*** work from **UC San Diego**

Break-through : Automatic system call side-effect analysis

Initial implementation for deterministic multi-threaded simulation

Further development + Support : Windows port, Android port, multi-threaded region recording, tracing…

Why? "*PinPoints out of order*" 27/55 SPEC2006 benchmarks!

Satish Narayanasamy, Giles Pokam, Prof. Brad Calder [ISCA 2005]

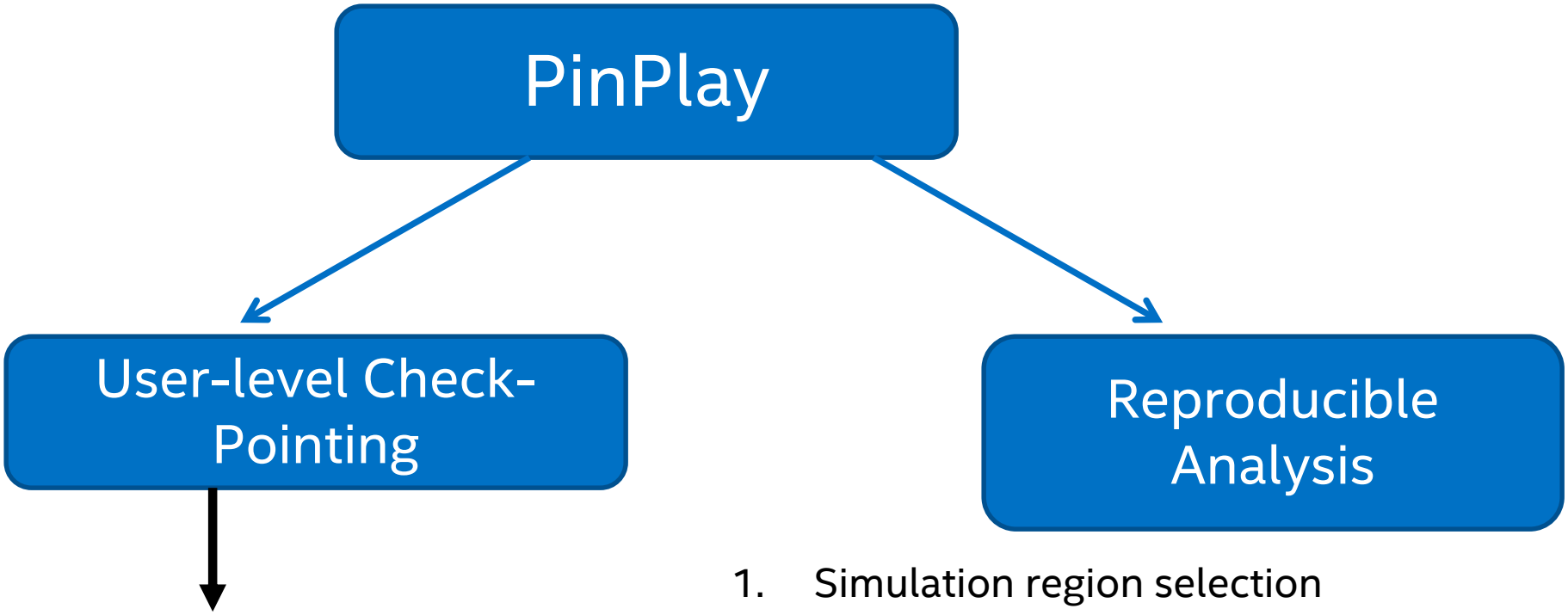
Satish Narayanasamy, Cristiano Pereira… [SIGMETRICS 2006]

Cristiano Pereira (Ph. D. thesis)

Jim Cownie, Ady Tal, Ariel Slonim, Michael Gorin, Michael Berezalsky, Tevi Devor, Omer Mor, Roman Zendel, Mack Stallcup, Cristiano Pereira, Harish Patil, **Pin team**

Sponsors: Geoff Lowney, Robert Cohn, Moshe Bach, Sion Berkowits, Nafta Shalev

PinPlay Applications at a glance



Traces for architecture simulation

1. pinballs 2. LIT

→ Intel simulators

→ Sniper (**U Ghent**) www.snipersim.org

1. Simulation region selection (*PinPoints*)
2. Dynamic program slicing (**UC Riverside**)
3. Replay-based debugging (*DrDebug*)
4. Dynamic control-flow graph generation
5. **< Your analysis here>**

PinPlay basics

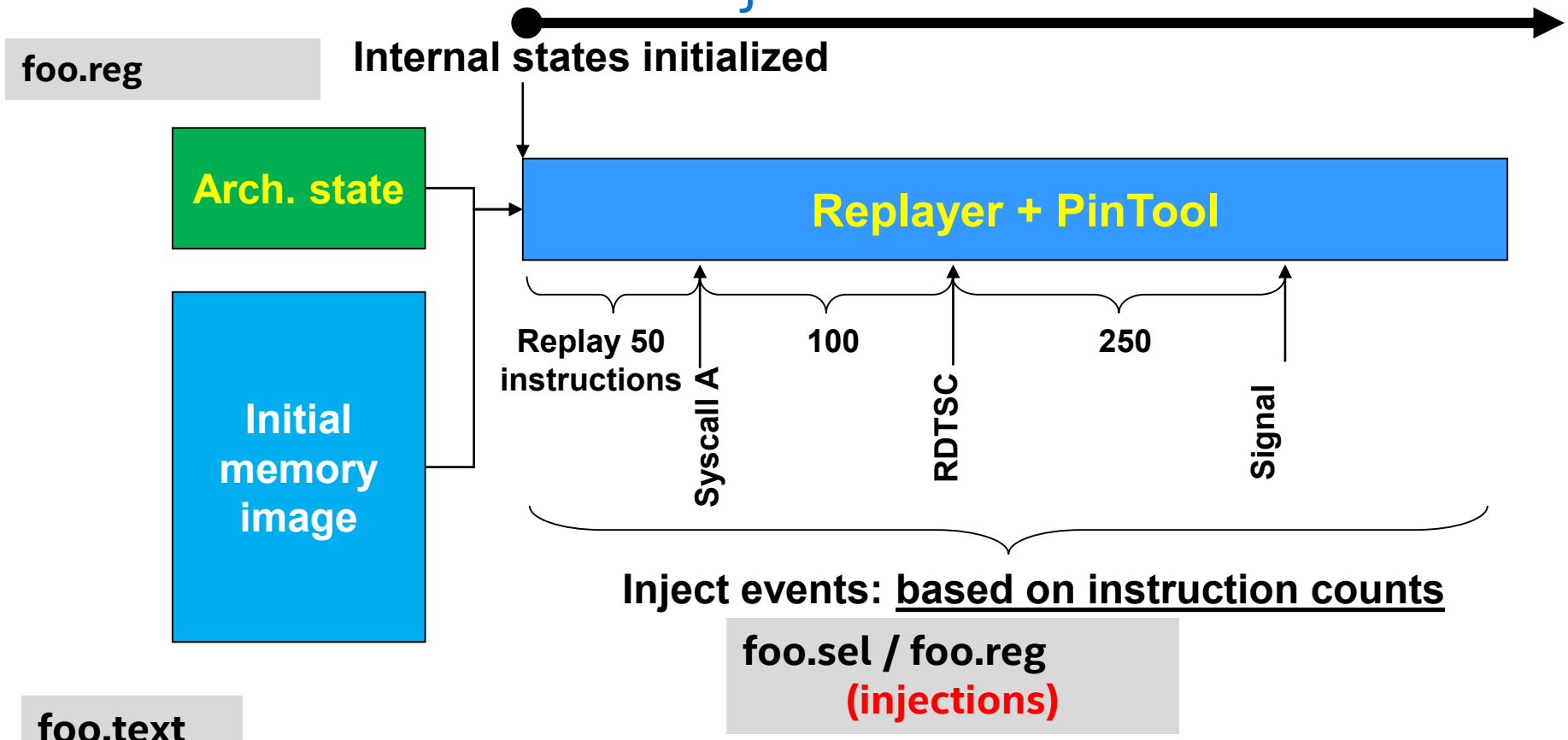
What makes single-threaded program runs non-repeatable?

1. Initial **stack** location changes (by kernel); changes to the location of **dynamically allocated memory** (heap)
2. **Shared library load location** changes (randomized address space);
3. Program binary/shared-library **code** changes (change over time or from machine to machine)
4. Processor-specific instruction behavior changes: **CPUID, RDTSC**
5. **Signals:** arbitrary memory/register/control-flow changes
6. **Un-initialized memory** location reads
7. **System call** behavior changes (depending on workdirectory, environment variables. e.g: ***gettimeofday() uname()***)

Memory logging

PinPlay efficiently handles all these

Pinball (ST) = Initial memory/register + injections



- **System calls** : skipped by injecting next rip/ memory changed
- **CPUID, RDTSC** : affected registers injected
- **Signals/Callbacks** : New register state injected

Multi-threaded programs: Additional source of non-repeatability

- Access order to shared-memory locations may change from run-to-run
  - Un-protected shared memory accesses
  - Accesses to locks/mutexes

Why: Threads may have different rates of progress across runs

- OS scheduling, machine load, machine configuration

Multi-threaded programs: Repeating Shared Memory Access Order

Logging: Record (a subset of) RAW, WAW, WAR dependences

Replaying: Satisfy dependences; make threads wait if needed

How to detect RAW/WAW/WAR dependences?

Simulate a directory-based cached coherence protocol in software

Netzer optimization : avoids logging implied dependencies
SPECOMP :Only 0.22% of all dependencies actually logged!

Pinball (MT) : Pinball (ST) + Thread-dependencies

foo.reg (per-thread)

foo.text

Application Memory (common)

foo.reg (per-thread)

foo.sel (per-thread)

Event injection works only if same behavior (same instruction counts) is guaranteed during replay

[T1] 2 T2 2

[T1] 3 T2 3

[T2] 5 T4 1

Thread T2 cannot execute instruction 5 until T4 executes instruction 1

foo.race (per-thread)

Thread T1 cannot execute instruction 2 until T2 executes instruction 2

Sources of Slow-down

1. **All programs (single-threaded or parallel)** :
   Logger: Instrument loads/stores (memory logging)
   Replayer: Instrument loads : Restore memory at the right "time"
2. **Parallel programs:**
   Logger: All memory instruction analyses guarded by locks to prevent changes to memory during analysis

If parallel replay desired:

Logger: Emulate directory-based cache-coherence protocol in software
Replayer: Obey logged shared-memory dependences; make threads wait

Logging more expensive than replay

Parallel replay more expensive than isolated replay

Special Case : Single-threaded Programs Fast Logging and Replay possible

Fast Logging (not implemented yet):

- Ad-hoc system-call side-effect analysis and logging (only practical for Linux)

Fast Replay (implemented):

- Restore memory locations “lump-sum” at system calls

➔ Replay at Pin-only speed!
(for single-threaded programs without signals: e.g. SPEC2006)

Summary: Slowdown & Disk Usage

Application	Instruction Count (avg)	Pinball size (average)	Logger Slowdown : X native (avg)	Replayer Slowdown: X-native (avg)
SPEC2006 (55 runs) single-threaded	924 billion	39MB	98 X	11 X [*1.4 X with "-replay:fast"*]
PARSEC (8 runs) : multi –threaded: 4T—25T	844 billion	3.9GB	197 X	37 X
SPECOMP 2001 : 4T	307 billion	91 MB	117X	25X

- Measured with PinPlay 2.0 'pin –t pinplay-driver.so'
- Some PARSEC benchmarks read large files which increases pinball size substantially

The PinPlay kit

Download from http://www.pinplay.org

Enabling a Pintool for PinPlay

```
#include "pinplay.H"
```

```
PINPLAY_ENGINE pinplay_engine;
```

```
KNOB<BOOL>KnobReplayer(KNOB_MODE_WRITEONCE, KNOB_FAMILY,
                       KNOB_REPLAY_NAME, "0", "Replay a pinball");
KNOB<BOOL>KnobLogger(KNOB_MODE_WRITEONCE,  KNOB_FAMILY,
                     KNOB_LOG_NAME, "0", "Create a pinball");
```

```
  pinplay_engine.Activate(argc, argv, KnobLogger, KnobReplayer);
```

Link in ***libpinplay.a, libzlib.a, libbz2.a, $(CONTROLLERLIB)***

Restriction:

- **PinTool shouldn't change application control flow**

Example: pinplay-branch-predictor.cpp

```
#define KNOB_LOG_NAME  "log"
#define KNOB_REPLAY_NAME "replay"
#define KNOB_FAMILY "pintool:pinplay-driver"


PINPLAY_ENGINE pinplay_engine;

KNOB_COMMENT pinplay_driver_knob_family(KNOB_FAMILY, "PinPlay Driver Knobs");

KNOB<BOOL>KnobReplayer(KNOB_MODE_WRITEONCE, KNOB_FAMILY,
                       KNOB_REPLAY_NAME, "0", "Replay a pinball");
KNOB<BOOL>KnobLogger(KNOB_MODE_WRITEONCE,  KNOB_FAMILY,
                     KNOB_LOG_NAME, "0", "Create a pinball");

int main(int argc, char *argv[])
{
    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }

    outfile = new ofstream(KnobStatFileName.Value().c_str());
    bimodal.Activate(KnobPhases, outfile);

    pinplay_engine.Activate(argc, argv, KnobLogger, KnobReplayer);

    PIN_AddThreadStartFunction(threadCreated, reinterpret_cast<void *>(0));


    PIN_StartProgram();
}
```

PinPlay-enabled PinTools : 3 Modes

1. **Regular Analysis mode**

```
$ pin -t pintool -- test-program
```

Normal output + *Analysis output*

2. **Logging Mode**

```
$ pin -t pintool –log –log:basename pinball/foo - - test-program
```

3. **Replay Mode**

```
$ pin -t pintool –replay –replay:basename pinball/foo - - nullapp
```

Example: pinplay-branch-predictor.so

% pin -t $PIN_ROOT/extras/pinplay/bin/intel64/pinplay-branch-predictor.so -- hello

Creates **"bimodal.out"**

%pin –t pinplay-branch-predictor.so **–log –log:basename pinball/foo** hello

Creates **"bimodal.out"** and **"pinball/foo*"**

%pin **–xyzzy –reserve_memory pinball/foo.address –replay –replay:basename pinball/foo** –- $PIN_KIT/extras/pinplay/intel64/bin/**nullapp**

Creates **"bimodal.out"**

Using $PIN_ROOT/extras/pinplay/scripts :Recording (uses pinplay-driver.so):

```
pinplay-VirtualBox:~/tests/hello> which record
/home/pinplay/PinPlay/latest/extras/pinplay/scripts//record
```

```
% record --help
Usage: record.py [options] -- binary args
               or
       record.py [options] --pid PID

Create a recording (pinball).  There are two modes:
  1) Give command line of a binary to record
  2) Give the PID of a running process
```

```
pinplay-VirtualBox:~/tests/hello> record --pintool $PIN_ROOT/ex
tras/pinplay/bin/intel64/pinplay-branch-predictor.so --pinball
pinball/foo -- hello
```

** Developed by Mack Stallcup*

Using $PIN_ROOT/extras/pinplay/scripts: Replaying (uses pinplay-driver.so)

```
pinplay-VirtualBox:~/tests/hello> which replay
/home/pinplay/PinPlay/latest/extras/pinplay/scripts//replay
```

```
% replay --help
Usage: replay.py [options] -- pinball

Replay a recording (pinball).
```

```
pinplay-VirtualBox:~/tests/hello> replay --pintool $PIN_ROOT/ex
tras/pinplay/bin/intel64/pinplay-branch-predictor.so -- pinball
/foo_0
```

0 added by 'record'

Adding your own file to a pinball

pinplay.H

```
//register to start/stop logging callbacks
VOID RegisterRegionStart(PINPLAY_HANDLER handler,VOID* args);
VOID RegisterRegionStop(PINPLAY_HANDLER handler,VOID* args);
```

PinPlay-enabled Pin-tool

Register callbacks

```
// Region start/stop callbacks from PinPlay when logging.
pinplay_engine.RegisterRegionStart(RegionStart, 0);
pinplay_engine.RegisterRegionStop(RegionStop, 0);
```

START handler

```
// Start a region when logging.
VOID RegionStart(VOID* v)
{
    THREADID pin_tid = PIN_ThreadId();
    string region_basename = pinplay_engine->GetRegionBaseName(pin_tid);
    // Open file you are adding -- use 'region_basename' as prefix.
    // $region_basname.myext
```

STOP handler

```
// Stop a region when logging.
VOID RegionStop(VOID* v)
{
    // close your file $region_basname.myext
```

Replay Debugging Foundation: PinADX

Developed by Greg Lueck (Intel Corporation**)**

Slides from Tevi Devor (Intel Corporation**)**

Transparent debugging, and extending the debugger

Transparently debug the application while it is running on Pin + Pin Tool

- **PinADX**: Customizable Debugging with Dynamic Instrumentation (Presented at CGO 2012)

Use Pin Tool to enhance/extend the debugger capabilities

- Watchpoint: Is order of magnitude faster when implemented using Pin Tool

- Which branch is branching to address 0
  - Easy to write a Pin Tool that implements this

Debug Application while Running Pin

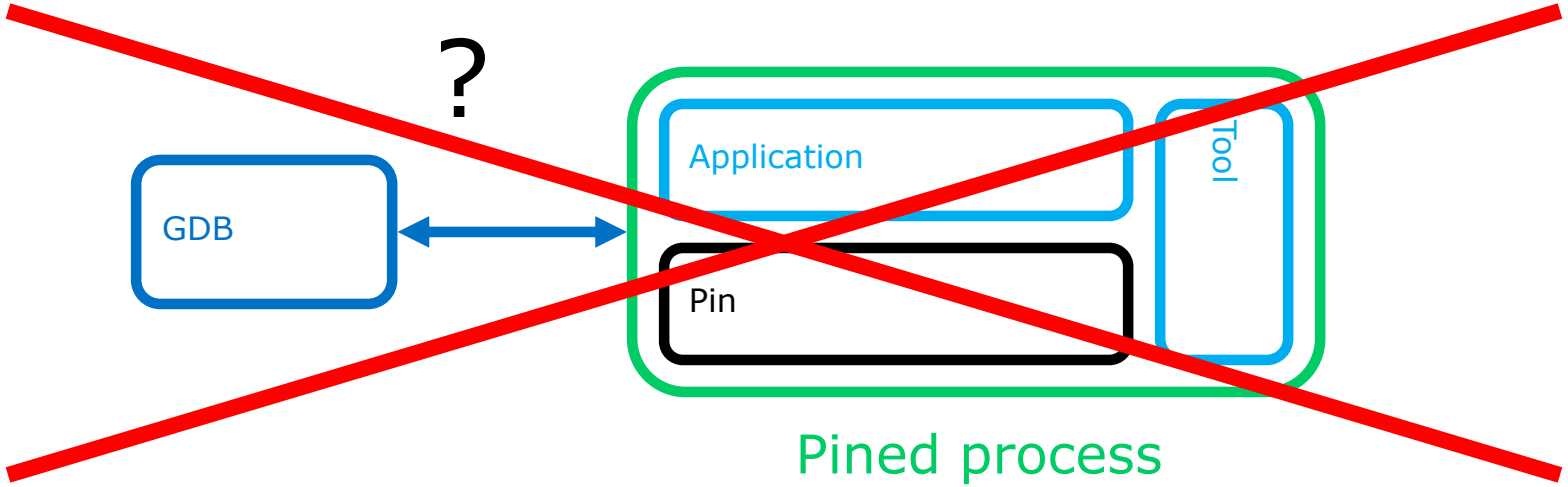
Useful for Pin-based emulators

- User can debug application while emulating

Provide advanced debugging features with Pin:

- Stack monitoring features
- Buffer overrun detection
- Reverse debugging
- Write your own debugger extension via Pin

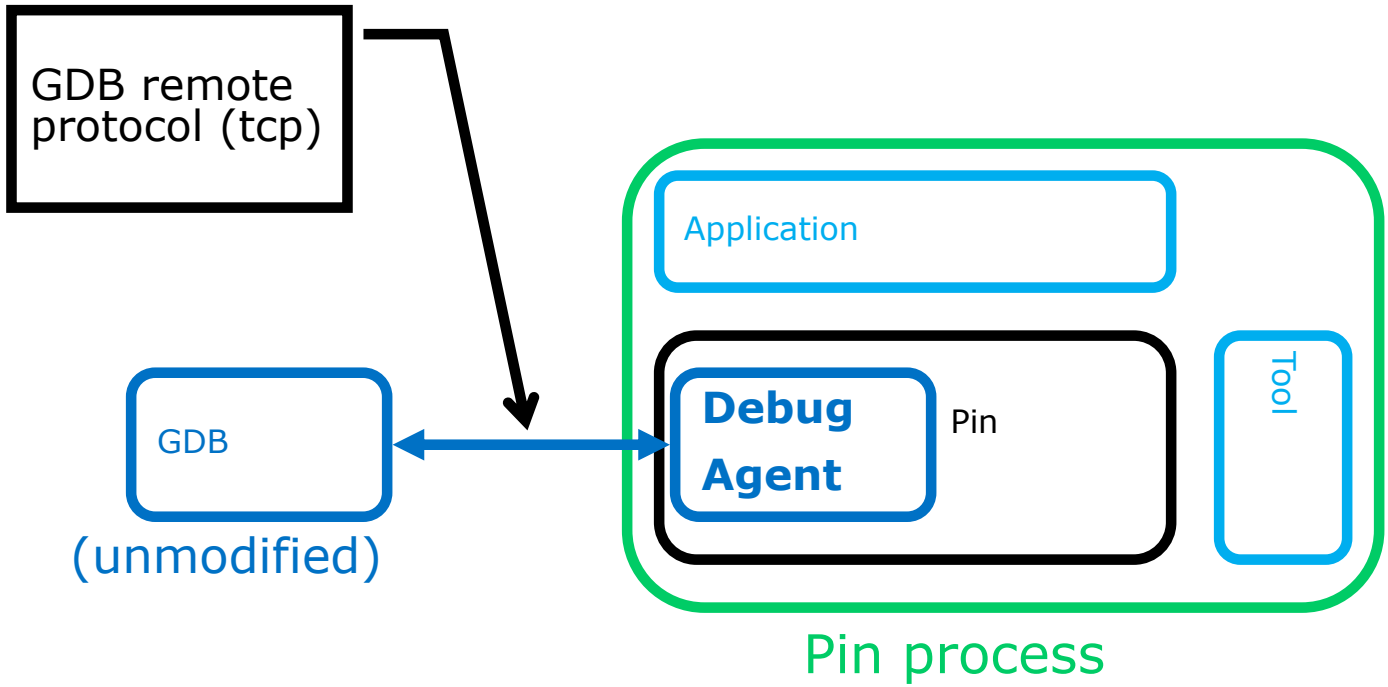
Naïve Solution Won't Work



Why can't we just debug normally?

- Debugger sees Pin state, not application state
- Pin recompiles application code
- Instructions wrong, registers wrong, PC wrong, …

Pin Debugger Interface



GDB debugs application (not Pin itself)

Leverage GDB remote protocol ABI

Debug the Application with Pin

1. Run Pin with -appdebug

```
$ pin -appdebug -t tool.so -- ./application
Application stopped until continued from debugger.
Start GDB, then issue this command at the (gdb) prompt:
  target remote :1234
```

2. Start GDB, enter “target remote …”

```
$ gdb ./application
(gdb) target remote :1234
```

3. Set breakpoints, etc. Continue with “cont”

```
(gdb) break main
(gdb) cont
```

[DrDebug scripts : gdb_record and gdb_replay : Single window solution]

Extending the Debugger

Normal debugging with Pin useful but limited

Extending the debugger:

- Add GDB commands via a Pin tool
- Stop at “semantic breakpoint” via instrumentation

Use the “monitor” keyword for implementing custom commands

```
// Debugger interpreter, to process debugger commands.
//
PIN_AddDebugInterpreter(DebugInterpreter, this);
```

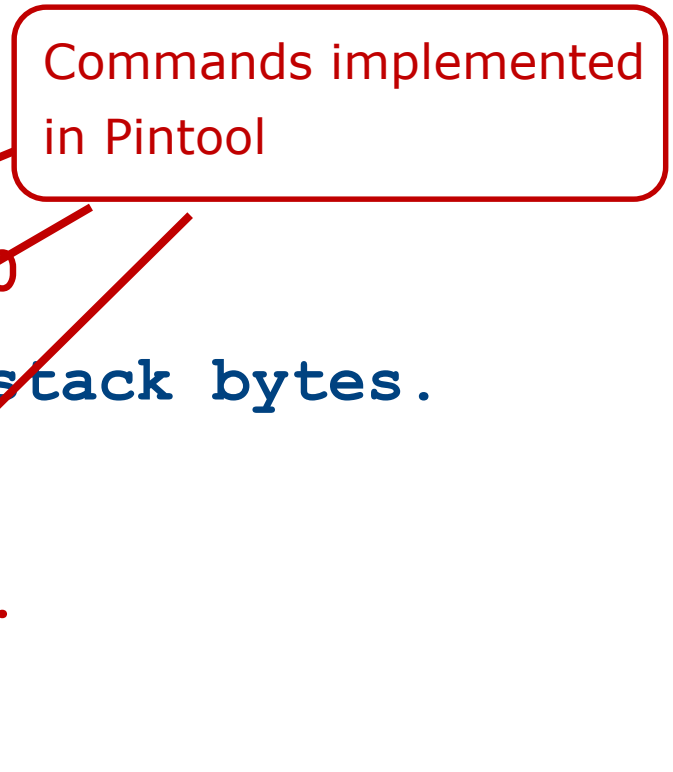
*[DrDebug : See extras/pinplay/examples/p**inplay-debugger-shell.cpp**]*

[DrDebug: uses “pin.py” to provide “pin <extended-command>”]

Stack Debugger Pintool

```
$ pin -appdebug -t stack-debugger.so -- ./app
```

```
$ gdb ./app
(gdb) target remote :1234
(gdb) monitor stackbreak 4000
Break when thread uses 4000 stack bytes.
(gdb) cont
Thread uses 4004 stack bytes.
[…]
(gdb) monitor stats
Maximum stack usage: 8560 bytes.
```



Stack-Debugger Instrumentation

```
Thread Start:
   StackBase = %esp;
   MaxStack = 0;

[…]
sub $0x60, %esp
   size = StackBase - %esp;
   if (size > MaxStack) MaxStack = size;
   if (size > StackLimit) TriggerBreakpoint();
cmp %esi, %edx
jle <L1>
```

After each stack-changing instruction

ManualExamples/stack-debugger.cpp

instrumentation routine

```
VOID Instruction(INS ins, VOID *)
{
    if (INS_RegWContain(ins, REG_STACK_PTR))
    {
        IPOINT where = (INS_HasFallThrough(ins)) ?
            IPOINT_AFTER : IPOINT_TAKEN_BRANCH;
        INS_InsertCall(ins, where, (AFUNPTR)OnStackChange,
            IARG_REG_VALUE, REG_STACK_PTR,
            IARG_THREAD_ID, IARG_CONST_CONTEXT, IARG_END);
    }
}
```

analysis routine

```
VOID OnStackChange(ADDRINT sp, THREADID tid, CONTEXT *ctxt)
{
    size_t size = StackBase - sp;
    if (size > StackMax) StackMax = size;
    if (size > StackLimit) {
        ostringstream os;
        os << "Thread uses " << size << " stack bytes.";
        PIN_ApplicationBreakpoint(ctxt, tid, FALSE, os.str());
    }
}
```

Insert only after instructions that write to %esp

ManualExamples/stack-debugger.cpp

instrumentation routine

```
VOID Instruction(INS ins, VOID *)
{
    if (INS_RegWContain(ins, REG_STACK_PTR))
    {
        IPOINT where = (INS_HasFallThrough(ins)) ?
            IPOINT_AFTER : IPOINT_TAKEN_BRANCH;
        INS_InsertCall(ins, where, (AFUNPTR)OnStackChange,
            IARG_REG_VALUE, REG_STACK_PTR,
            IARG_THREAD_ID, IARG_CONST_CONTEXT, IARG_END);
    }
}
```

analysis routine

```
VOID OnStackChange(ADDRINT sp, THREADID tid, CONTEXT *ctxt)
{
    size_t size = StackBase - sp;
    if (size > StackMax) StackMax = size;
    if (size > StackLimit) {
        ostringstream os;
        os << "Thread uses " << size << " stack bytes.";
        PIN_ApplicationBreakpoint(ctxt, tid, FALSE, os.str());
    }
}
```

Triggers debugger breakpoint

stack-debugger.cpp

Hooks the GDB "monitor" command. E.g.:
(gdb) monitor stats
(gdb) monitor stackbreak 4000

```
int main() {
    […]
    PIN_AddDebugInterpreter(HandleDebugCommand, 0);
}
BOOL HandleDebugCommand(const string &cmd, string *result) {
    if (cmd == "stats")
    {
        ostringstream os;
        os << "Maximum stack usage: " << StackMax << " bytes.\n";
        *result = os.str();
        return TRUE;
    }
    else if (cmd.find("stackbreak ") == 0)
    {
        StackLimit = /* parse limit */;
        ostringstream os;
        os << "Break when thread uses " << StackLimit << " stack bytes.";
        *result = os.str();
        return TRUE;
    }
    return FALSE;  // Unknown command
}
```

[DrDebug : See extras/pinplay/examples/pinplay-debugger-shell.cpp]

ManualExamples/stack-debugger.cpp

```
int main() {
    […]
    PIN_AddDebugInterpreter(HandleDebugCommand, 0);
}
```

Receives text after "monitor"

```
BOOL HandleDebugCommand(const string &cmd, string *result) {
    if (cmd == "stats")
    {
        ostringstream os;
        os << "Maximum stack usage: "
        *result = os.str();
        return TRUE;
    }
    else if (cmd.find("stackbreak ") == 0)
    {
        StackLimit = /* parse limit */;
        ostringstream os;
        os << "Break when thread uses " << StackLimit << " stack bytes.";
        *result = os.str();
        return TRUE;
    }
    return FALSE;  // Unknown command
}
```

This string written to GDB session

[DrDebug : See extras/pinplay/examples/pinplay-debugger-shell.cpp]

Other Debugger Tools

Breakpoint on buffer overrun

Debug from a recorded log file

Reverse debugging from a recording

Design your own custom debugger tool

DrDebug : Debugging, Analysis, and Replay Toolkit

See extras/pinplay/examples/pinplay-debugger-shell.cpp

Replay Debugging: DrDebug kit

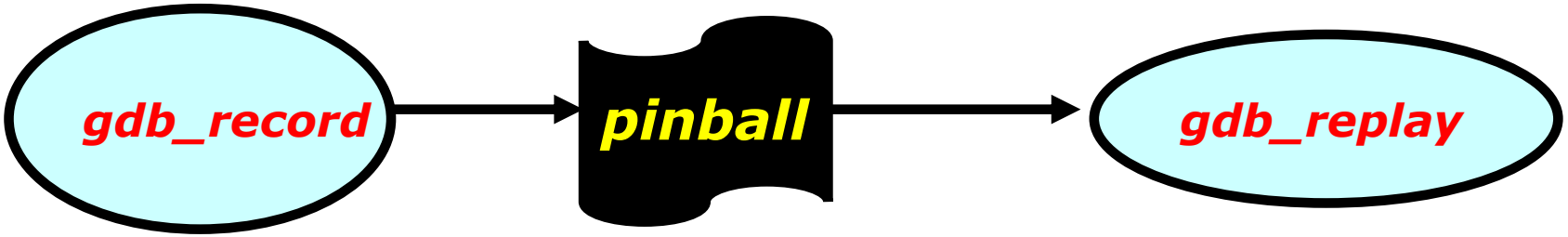
1. What is DrDebug?
2. Why DrDebug?
3. How DrDebug works?
4. Demo

What is DrDebug

- **Pin-based GDB extensions**

  gdb_record and ***gdb_replay***

- **Supports replay-based debugging**



- **Brings the power of Pin's dynamic analysis to GDB**

  pintools → Extended GDB commands
  (fast conditional breakpoints, ***printf*** debugging..)

DrDebug/PinPlay kit @ www.drdebug.org
Eclipse GUI: CDT plugin PinPlay→Dynamic-Slicing

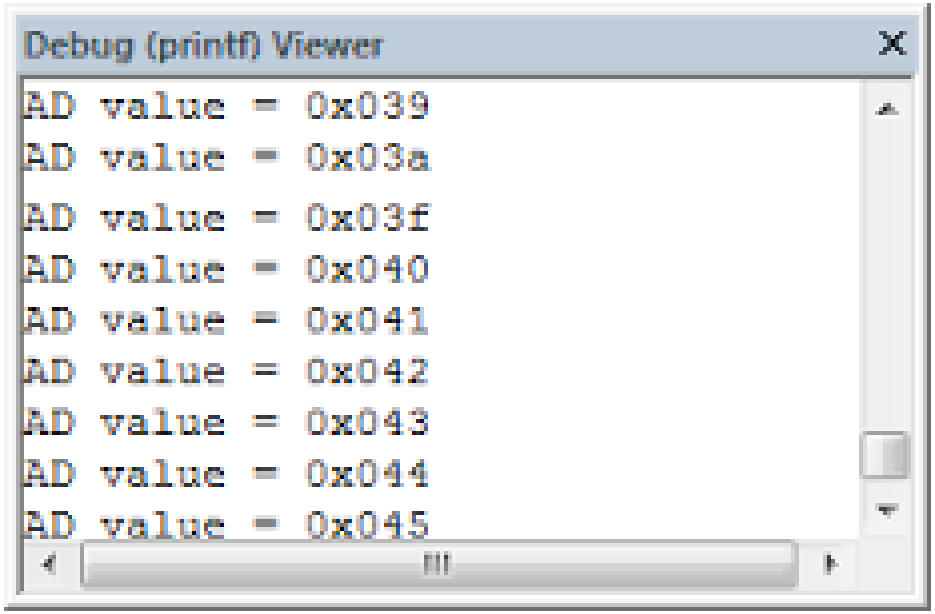
Why DrDebug?

"Because programmers are people too!"

How people debug?

1. printf debugging

What: Print values/messages at points of interest



Challenges:

1. Requires source/re-compilation
2. Values/control-flow may change across runs

Tracing Variables with DrDebug

- Use Pin to instrument points of interest and print values
- No source changes/re-compilation necessary
- Debugging with replay ➔ Values do not change across sessions

```
% gdb_replay -- failing.pinball/log_0 bread-demo
```

```
(gdb) pin trace order at 189
monitor trace [ %rbp + -8 ] 8 at 0x40171d #order:<189>
Tracepoint #1:  trace memory [%rbp offset -8 ] length 8
 at 0x40171d #order:<189>
```

```
Breakpoint 1, 0x00002aaac0884160 in _exit ()
(gdb) pin trace print to order.txt
```

```
% head order.txt
0x000000000040171d: [$rbp + -8] = 0x551bb0 #order:<189>
0x000000000040171d: [$rbp + -8] = 0x550550 #order:<189>
0x000000000040171d: [$rbp + -8] = 0x5492b0 #order:<189>
0x000000000040171d: [$rbp + -8] = 0x5486f0 #order:<189>
0x000000000040171d: [$rbp + -8] = 0x547350 #order:<189>
```

How people debug?

2. Cyclic debugging


```
0. Make a hypothesis about the bug
while (not (bug-root-caused))
do
   1. Fast forward till the buggy
      region
   2. Proceed slowly, examine state
   3. Hit the bug; more state
      examination
done
```

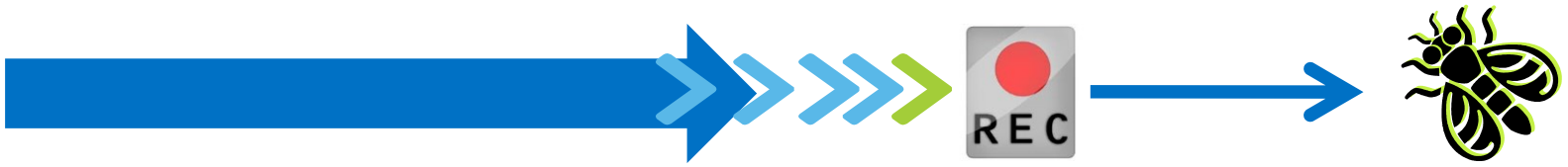
Cyclic Debugging Challenges

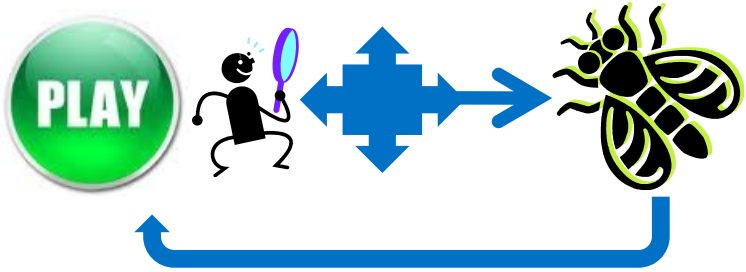
1. **Takes too long to reach bugs at times** : every debug cycle is long
2. **Many things change on each debug cycle iteration** : stack, heap, signals, interaction with outside world (interactive programs), network IO, thread schedule….
3. **Some bugs are hard to repeat in general and also under debugger**

1. Takes too long to reach bugs at times : Each debug cycle iteration is long

Once: Go to buggy region, hit record ➔ capture bug

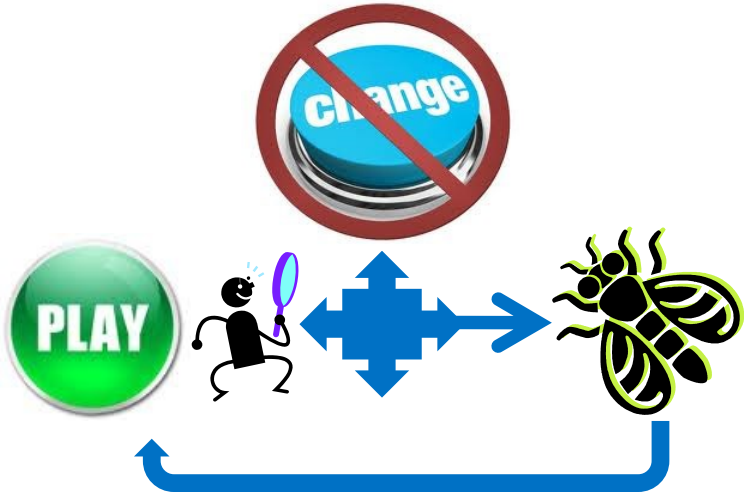
Iteratively : Start at the buggy region not the beginning of the program





2. Many things change on each debug cycle iteration : stack, heap, thread schedule, system call results…

DrDebug guarantees stack/heap/other non-deterministic events are repeatable



3. Some bugs are hard to repeat in general and also under debugger

Once captured, the bug never escapes

How do I get the bug to occur under the debugger?

Expose the bug by changing thread schedules

What we did: Interfaced with *Maple* bug exposing tool

(**See Maple presentation later**)

So, what is the cost?
Record/Replay Everything ➔ High Overhead

Why: No HW/OS dependencies; Guaranteed repeatability

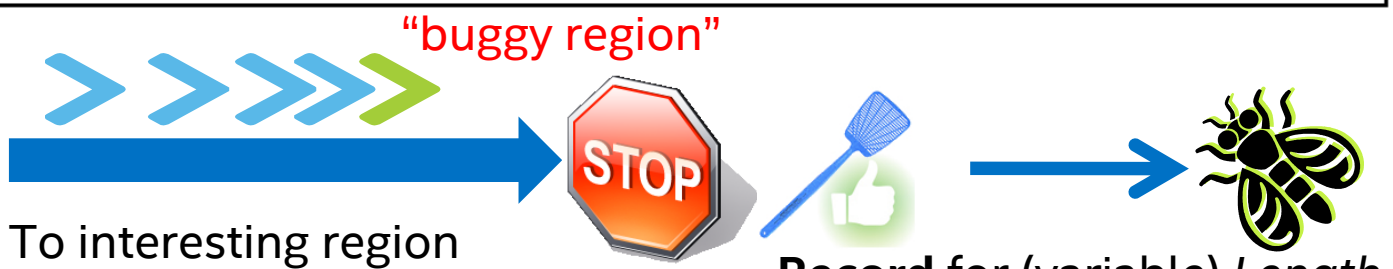
Program/input	Logging Slowdown (X native)	Replayer Slowdown (X native)
SPEC2006/ref (ST)	98X	11X
PARSEC/'native'(4-25T)	197X	37X

Not needed for debugging!

Slide from Cristian Zamfir: USENIX HotOS XIII

Debug Determinism

To be useful for debugging, should at a minimum reproduce the original failure and root cause.

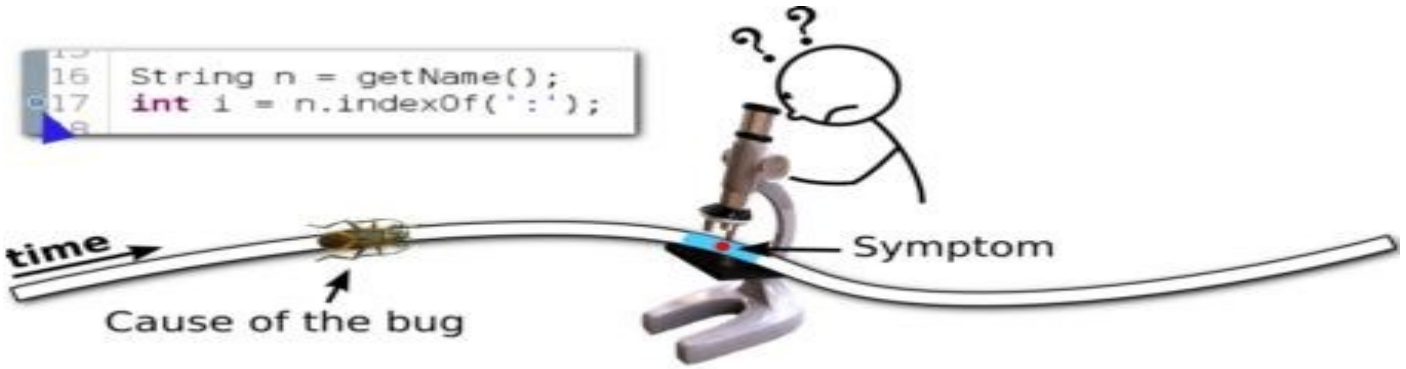


Skip: To interesting region (before cause)

Record for (variable) *Length*

Cause → Symptom

So, what's the cost, again?
Depends on "Buggy Region" length



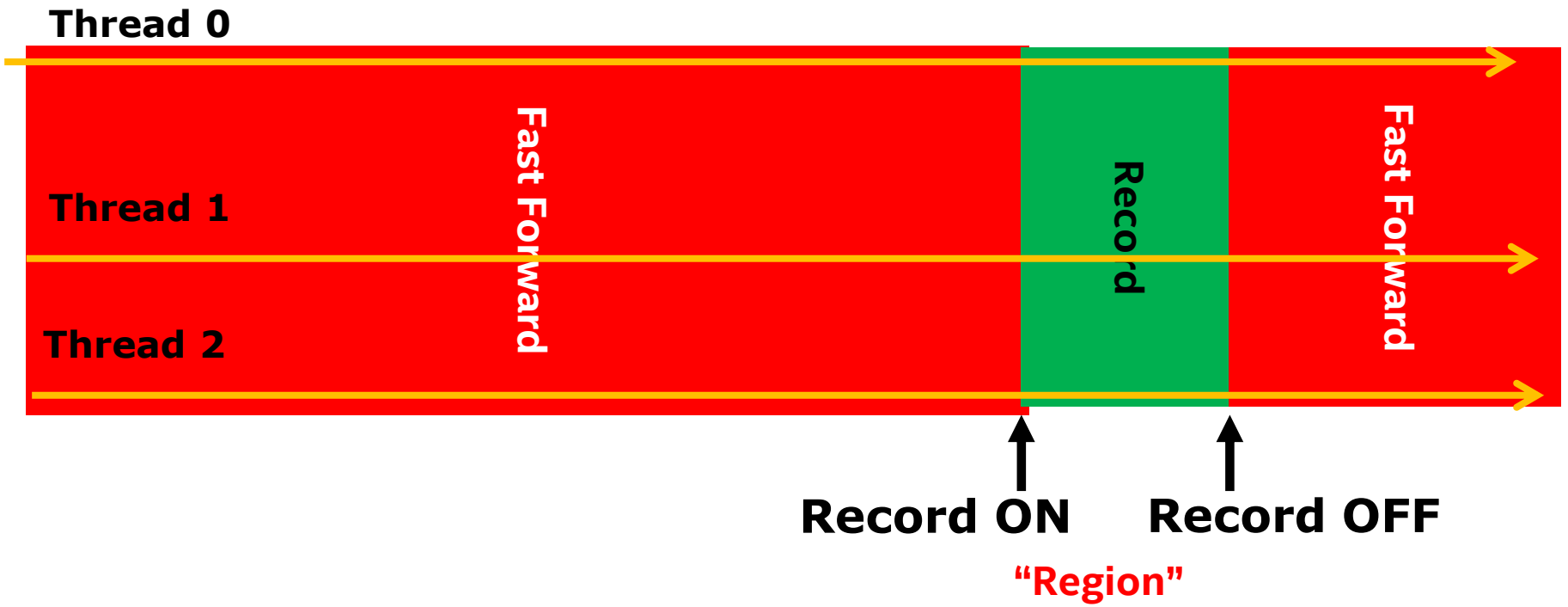
Cause → Symptom

Typical "buggy region" lengths:

1. ~10 million instructions *(study of 13 buggy open source programs)*
2. 8 million(*Mozilla*), 76K(*Aget*) : (our testing CGO2014)

Terminology: What is a “region”?

Points in execution where recording “starts” and “ends”.



Buggy Region: captures **Root Cause → Symptom**

Example region record/replay times

PARSEC (native input)– 4 Threaded runs

Average times in seconds for 8 programs

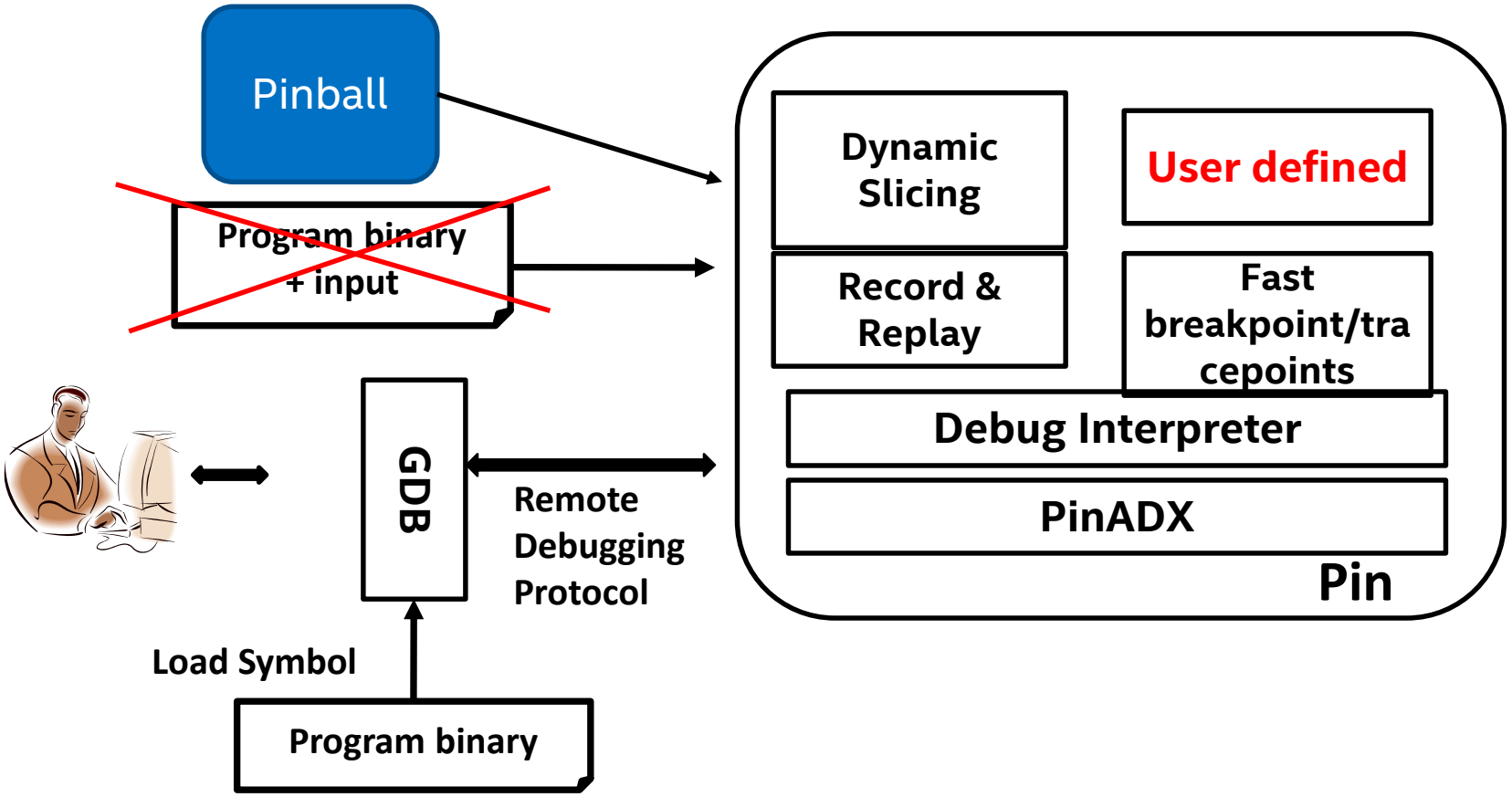
Region length in 'instruction count'

(region triggered in one thread : all threads captured)

Cause → Symptom

Region length	Region record time	Region replay time
4 x 10 million	10 seconds	10 seconds
4 x 100 million	29 seconds	27 seconds
4 x 500 million	75 seconds	43 seconds
4 x 1 billion	125 seconds	55 seconds

DrDebug: Linux Setup

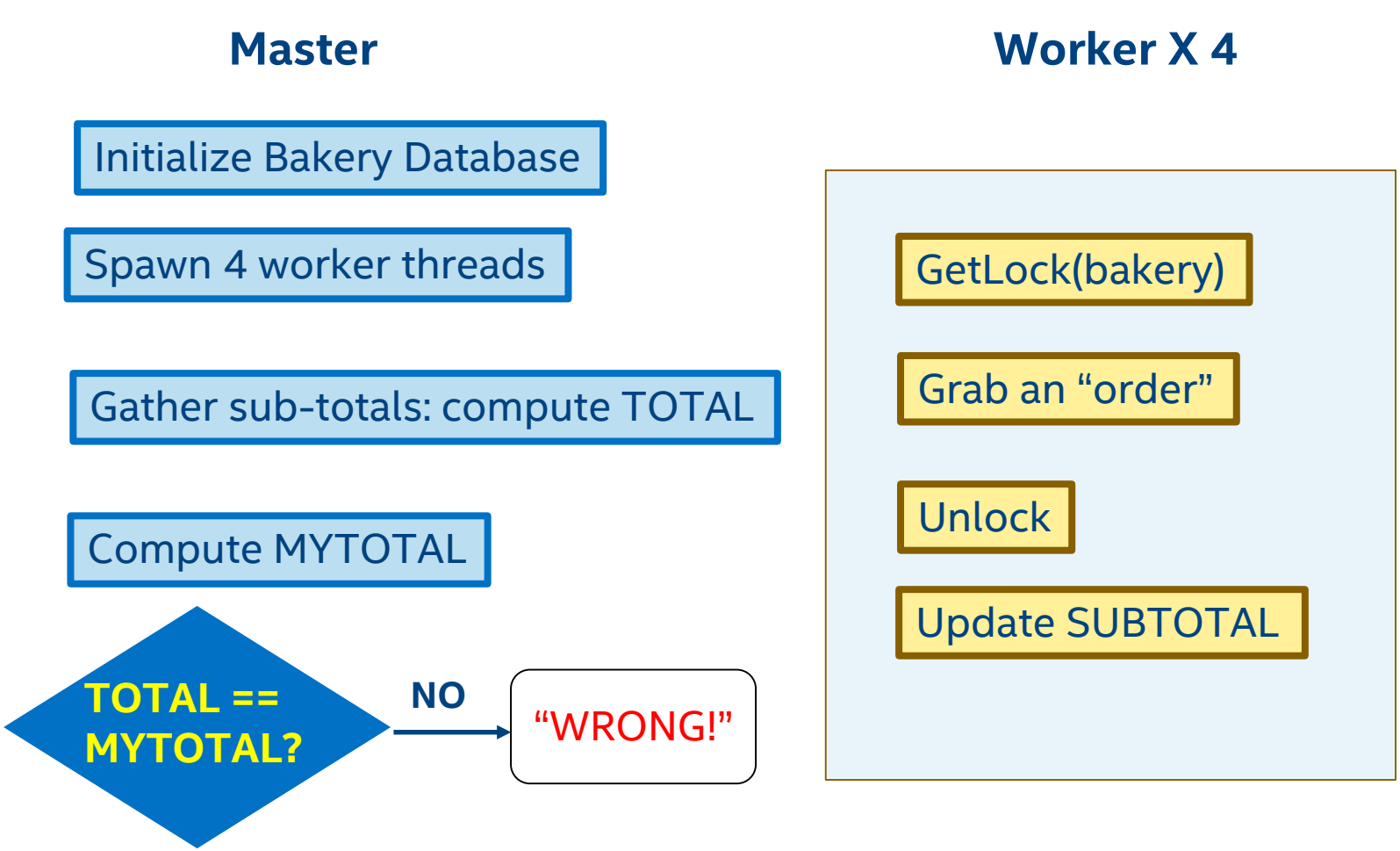


http://www.drdebug.org

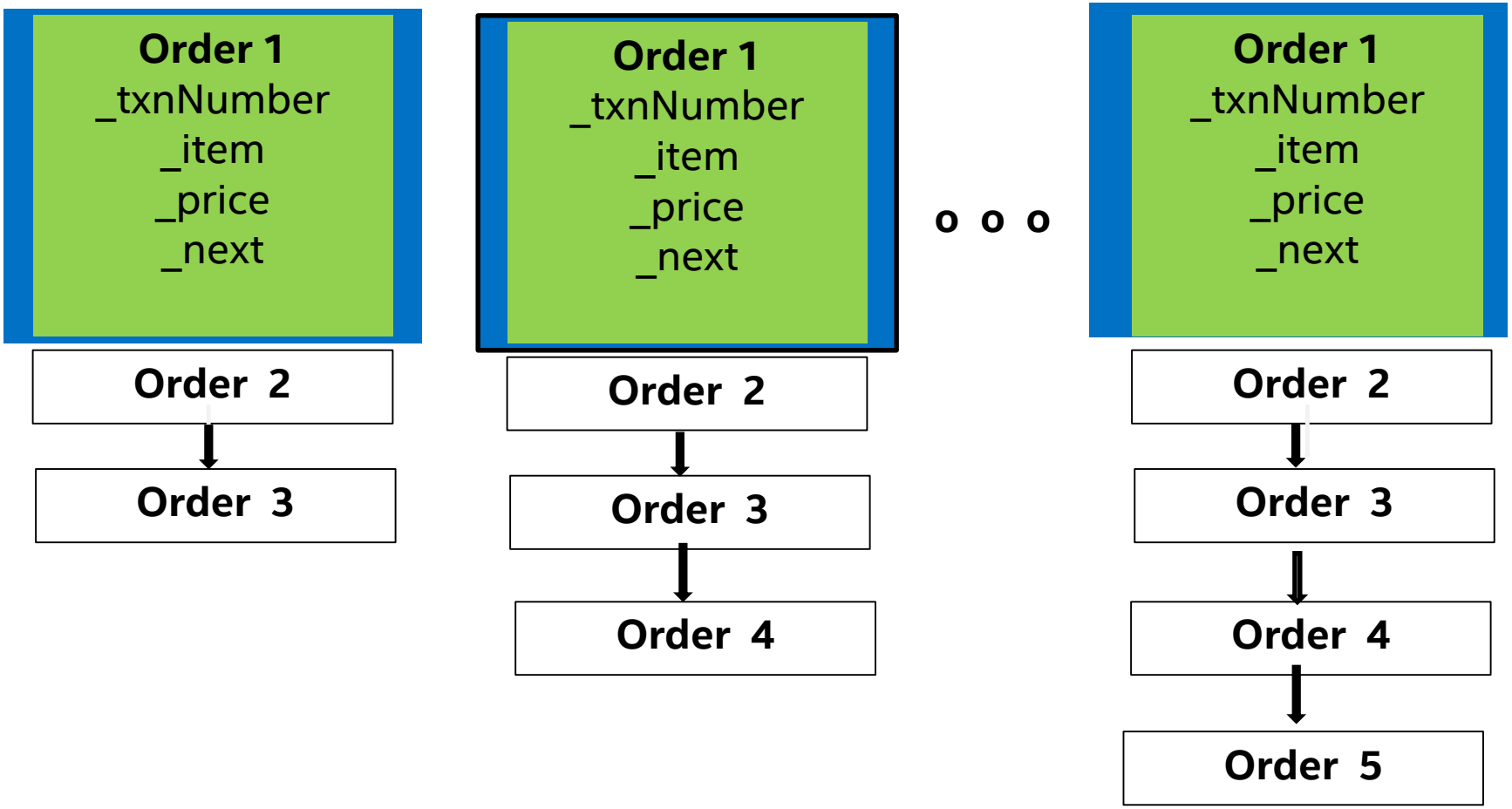
Test Program : bread-demo

Processes **10000 orders** with a pool of **4 threads**

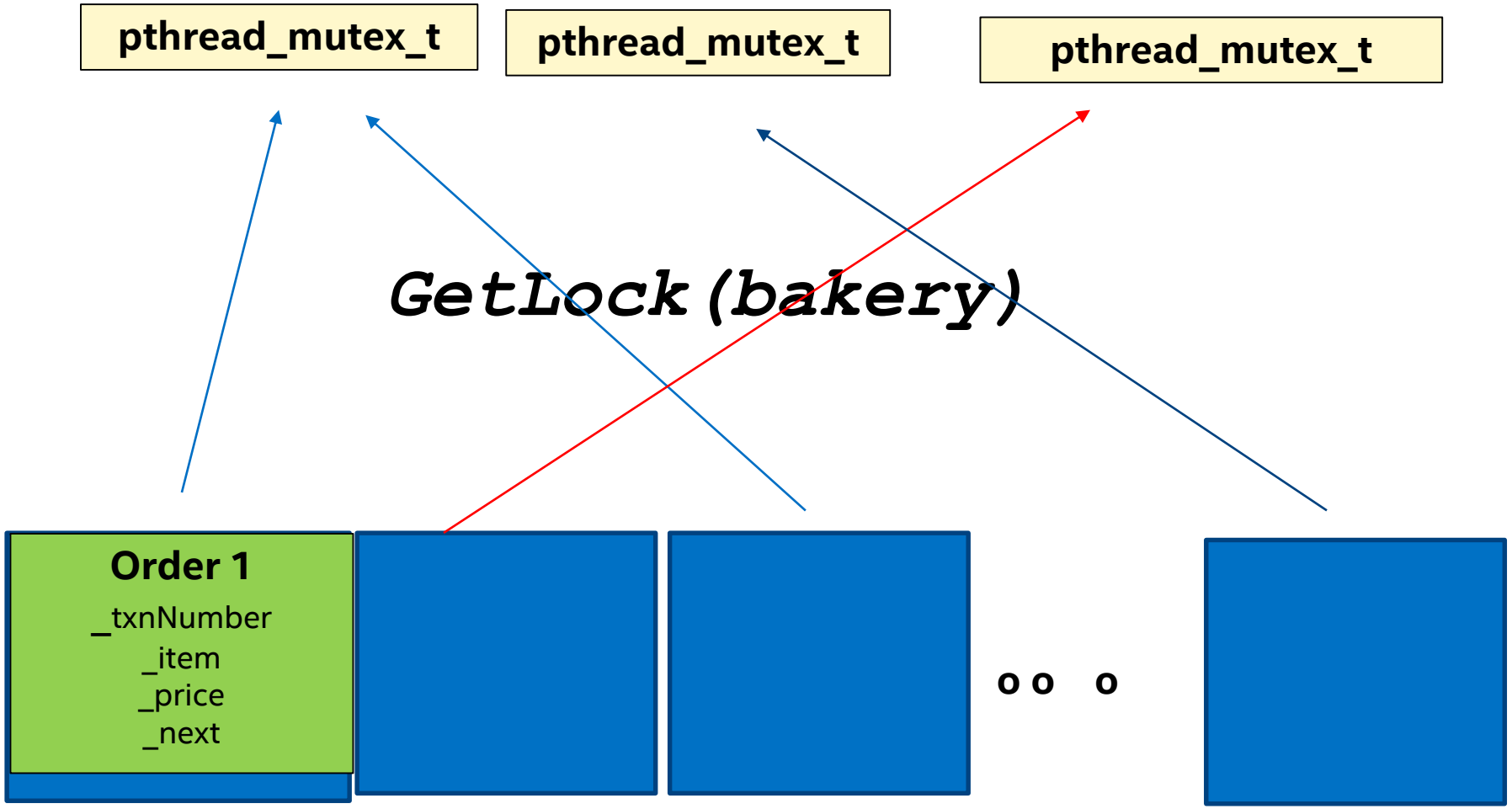
Prints "WRONG!" if check fails



ORDER *Bakeries[NUMBER_BAKERIES (= 500)]



Pthread_mutex_t Locks[NUMBER_LOCKS (= 8)]



Need a unique lock per bakery

```
static void *WorkerMain(void *v)
{
    // Loop over each bakery's queue, processing one transaction from
    // each queue before moving to the next queue.
    //
    for (;;)
    {
        bakery = (bakery + 1) % NUMBER_BAKERIES;

        // Get the lock for this bakery's queue.
        //
        pthread_mutex_t *lock = GetLock(bakery);
        pthread_mutex_lock(lock);

            // Get one transaction from this queue.
            //
            ORDER *order = Bakeries[bakery];

Line 189    Bakeries[bakery] = order->_next;

        pthread_mutex_unlock(lock);

        // Process the transaction.
        //
        worker->_breadOrders[order->_item]++;
        worker->_totalRevenue += order->_price;
    }
    // We're done when there are no transactions left on any bakery's queue.
    return 0;
```

Demo: Instructions

```
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> cat HOWTO.txt
Record a bug:
------------
 gdb_record bread-demo
 b 113
 c
 pin record on
 c
 q
# if the session above shows a "WRONG" message, you have captured a buggy
#  execution; else repeat the session.
Replay a buggy execution:
------------------------
   replay pinball/log_0

gdb+replay a buggy execution:
----------------------------
   gdb_replay pinball/log_0 bread-demo
   pin trace order at 189
   b _exit
   c
   pin trace print to order.txt
   q

Find order being processed twice/multiple times:
-----------------------------------------------
% sort.sh order.txt
 capture order value XXXX that appears multiple times
```

Demo: Instructions (continued)

```
gdb+replay a buggy execution:
-----------------------------
   gdb_replay pinball/log_0 bread-demo
   pin break 189 if order == XXXX
   c
   p order
   info threads
   # see which thread is processing order XXXX
   c
   # second processing of the same order
   p order
   info threads
   # see which thread is processing order XXXX
   # at this point we know which two threads are processing the same order.
   # if both the threads are still at line 189, you can switch between threads
   thread m
   print lock
   thread j
   print lock
   #two threads have different locks, why?
Continue exploring....
```

Recording a "*buggy region*" with gdb_record

```
Terminal
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> gdb_record bread-demo
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
```

```
(gdb) b 113
Breakpoint 1 at 0x4011fd: file bread-demo.cpp, line 113.
(gdb) c
```

```
Breakpoint 1, main (argc=1, argv=0x7fff812181a8) at bread-demo.cpp:113
113         Go = true;
(gdb) pin record on
monitor record on
Started recording region number 0
```

```
(gdb) c
Continuing.
```

```
Total revenue: $21948.42 (WRONG!)
[Inferior 1 (Remote target) exited with code 01]
(gdb)
```

Replay + debugging with gdb_replay

```
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> replay pinball/log_0

Bread sales for WW04
--------------------
  white     : 1720
  wheat     : 1644
  bagel     : 1728
  baguette  : 1674
  croissant: 1606
  challah   : 1630

Total revenue: $21948.42 (WRONG!)
```

```
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> gdb_replay pinball/log_0 bread-demo

GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
```

```
118             double revenue = 0.0;
(gdb) pin trace order at 189
```

```
(gdb) b _exit
Breakpoint 1 at 0x7f74587242d0: _exit. (2 locations)
(gdb) c
```

Replay + debugging with gdb_replay (contd.)

```
Total revenue: $21948.42 (WRONG!)

Breakpoint 1, __GI__exit (status=status@entry=1)
    at ../sysdeps/unix/sysv/linux/_exit.c:28
28      ../sysdeps/unix/sysv/linux/_exit.c: No such file or directory.
(gdb) pin trace print to order.txt
monitor trace print to order.txt
(gdb) c
Continuing.
[Inferior 1 (Remote target) exited normally]
(gdb)
```

```
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> wc -l order.txt
10002 order.txt
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> sort.sh order.txt
      2 0x0000000000401533: [$rbp + -8] = 0x1f45550 #order:<189>
      2 0x0000000000401533: [$rbp + -8] = 0x1f28e90 #order:<189>
      1 0x0000000000401533: [$rbp + -8] = 0x1f4c1f0 #order:<189>
      1 0x0000000000401533: [$rbp + -8] = 0x1f4c1d0 #order:<189>
      1 0x0000000000401533: [$rbp + -8] = 0x1f4c1b0 #order:<189>
```

Replay + debugging with gdb_replay (contd.)

```
pinplay-VirtualBox:~/DrDebug/BreadDemo.pldi> gdb_replay pinball/log_0 bread-demo

GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
```

```
(gdb) pin break 189 if order == 0x1f45550
monitor break at 0x401533 if [ %rbp + -8 ] 8 == 0x1f45550 #order:<189>
Breakpoint #1:  break  at 0x401533 if  [$rbp offset -8 ] length 8 == 0x1f45550 #
order:<189>
(gdb) c
Continuing.
```

```
(gdb) p order
$1 = (ORDER *) 0x1f45550
(gdb) p bakery
$2 = 4
(gdb) info threads
[New Thread 9011]
[New Thread 9010]
[New Thread 9009]
  Id   Target Id         Frame
  5    Thread 9009       0x000000000040153e in WorkerMain (v=0x1f4c210)
    at bread-demo.cpp:189
  4    Thread 9010       WorkerMain (v=0x1f4c238) at bread-demo.cpp:171
  3    Thread 9011       WorkerMain (v=0x1f4c288) at bread-demo.cpp:171
* 2    Thread 9012       WorkerMain (v=0x1f4c260) at bread-demo.cpp:189
  1    Thread 9001       0x00007f7458f7966b in pthread_join (
    threadid=140137651791616, thread_return=0x0) at pthread_join.c:92
(gdb) p lock
$3 = (pthread_mutex_t *) 0x6043f0 <Locks+240>
```

Replay + debugging with gdb_replay (contd.)

```
(gdb) c
Continuing.
Triggered breakpoint #1: break  at 0x401533 if  [$rbp offset -8 ] length 8 == 0x
1f45550 #order:<189>
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 9010]
WorkerMain (v=0x1f4c238) at bread-demo.cpp:189
189             Bakeries[bakery] = order->_next;
(gdb) p order
$4 = (ORDER *) 0x1f45550
(gdb) p bakery
$5 = 4
(gdb) info threads
  Id   Target Id         Frame
  5    Thread 9009       0x000000000040153e in WorkerMain (v=0x1f4c210)
    at bread-demo.cpp:189
* 4    Thread 9010       WorkerMain (v=0x1f4c238) at bread-demo.cpp:189
  3    Thread 9011       0x0000000000400f00 in pthread_mutex_lock@plt ()
  2    Thread 9012       WorkerMain (v=0x1f4c260) at bread-demo.cpp:189
  1    Thread 9001       0x00007f7458f7966b in pthread_join (
    threadid=140137651791616, thread_return=0x0) at pthread_join.c:92
```

Demo : Takeaway

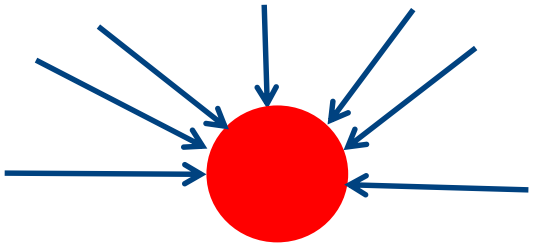
- We could do multiple `gdb_replay` sessions and gather more information each time
- This was feasible because
  1. the same buggy schedule was reproduced in each session and
  2. pointer values ('order' values) remained the same across sessions
- That is the power of replay debugging –
  bug once captured never escapes!

Replay Debugging: User Testimonial

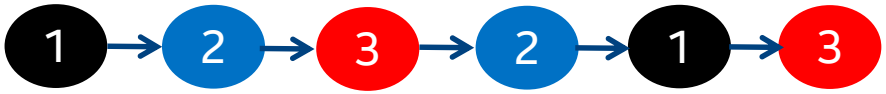
Milind Chabbi : Rice University

A Quick Review of Locks

- [BAD] Centralized locks (Test&Set, Test&Test&Set, Backoff)
  - ✕ Remote traffic
  - ✕ Poor scaling
  - ✕ Unfair

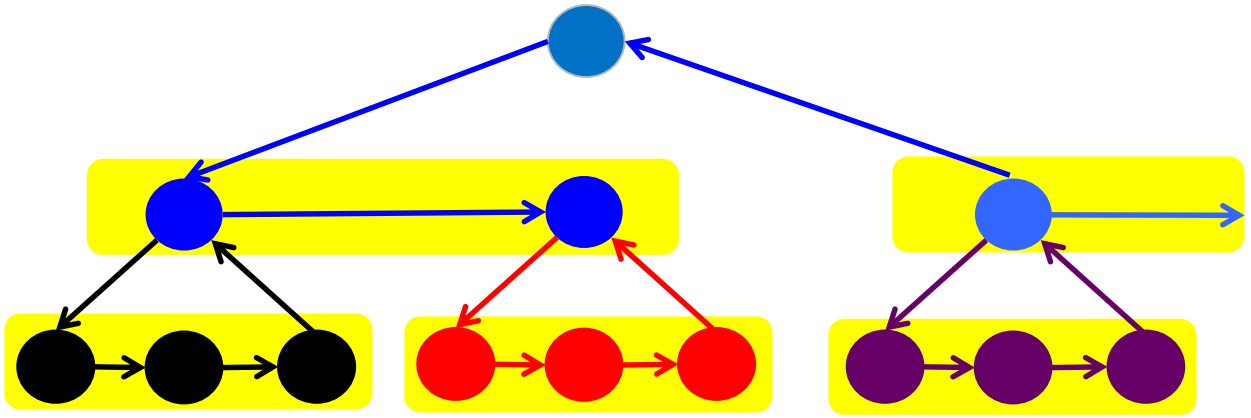


- [GOOD] Queuing locks (MCS, CLH)
  - ✓ Local spinning
  - ✓ Superior scaling
  - ✓ FIFO fairness
  - ✕ **Indiscriminate data movement on NUMA architectures**



Hierarchical MCS (HMCS): A NUMA-Aware Lock

- Always pass the lock to a thread in the nearest NUMA domain
  - Reuse the temporal locality of reference
- Bound the local passing to ensure starvation freedom
- Result: ~70x higher throughput than MCS lock on 4096-core SGI UV 1000

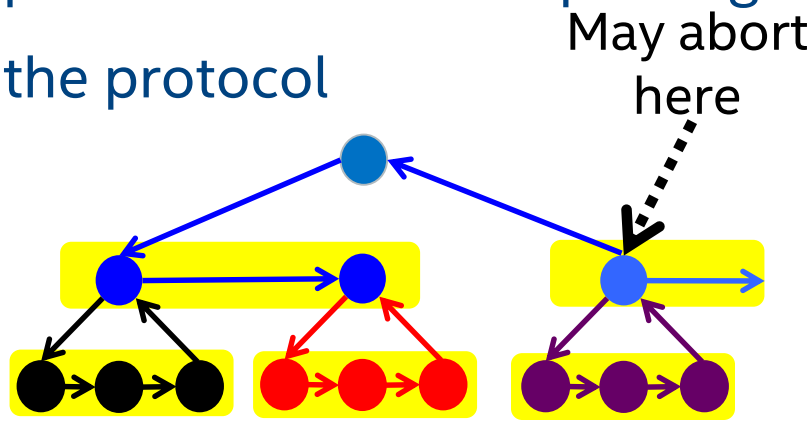


[PPoPP'15] High performance locks for multi-level NUMA systems. Chabbi et al.

A Sophisticated Abortable HMCS Lock

Allow a waiting thread to **timeout** and leave the protocol

- Complications due to
  - State-full nature of the protocol and massive asynchrony
  - Complex flow of control, mutual recursion, and atomic operations
  - Dependence of a successor on its predecessor for lock passing
  - Reentry of an aborted thread into the protocol
- Result
  - T&S lock: **3 LOC**
  - MCS lock: **11 LOC**
  - Abortable HMCS lock: **300+ LOC**



Challenges Developing the Abortable HMCS Lock

- Ensuring the correctness of the protocol
  - Algorithm and implementation
- What happened?
  - A live lock in a simple test
- Nature of the concurrency bug:
  - Non-deterministic
  - Sporadic
  - Heisenbug
  - Many threads needed to reproduce (typically)

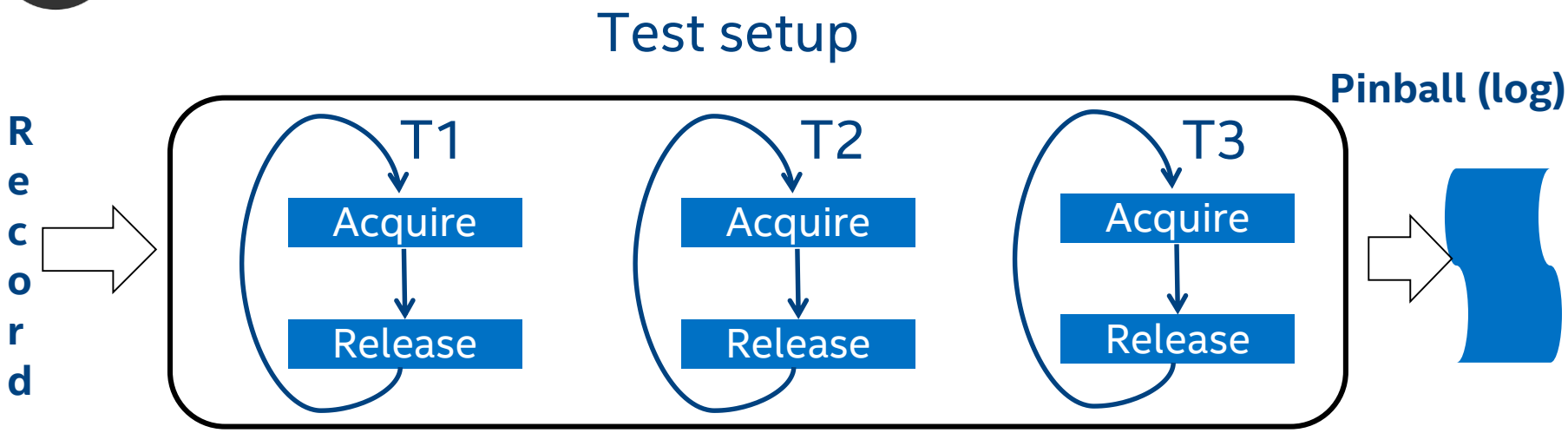
Failed Efforts to Pinpoint the Bug

First instinct: “My code must be wrong!”

- Low level techniques
  - Debugging: single stepping and watch points
  - Extensive set of assertions
  - Page protection to catch stray accesses
- High level strategies
  - Protocol/design review with synchronization experts
  - Reasoning about the memory model and instruction reordering

PinPlay to Pinpoint the Bug

Record the buggy execution with PinPlay



- Launch the locking test under PinPlay recorder
- If the assertion is hit then done
- Else, delete the pinball and try again

- Pinball with buggy execution ready overnight

PinPlay Makes Execution Recoding Really Simple

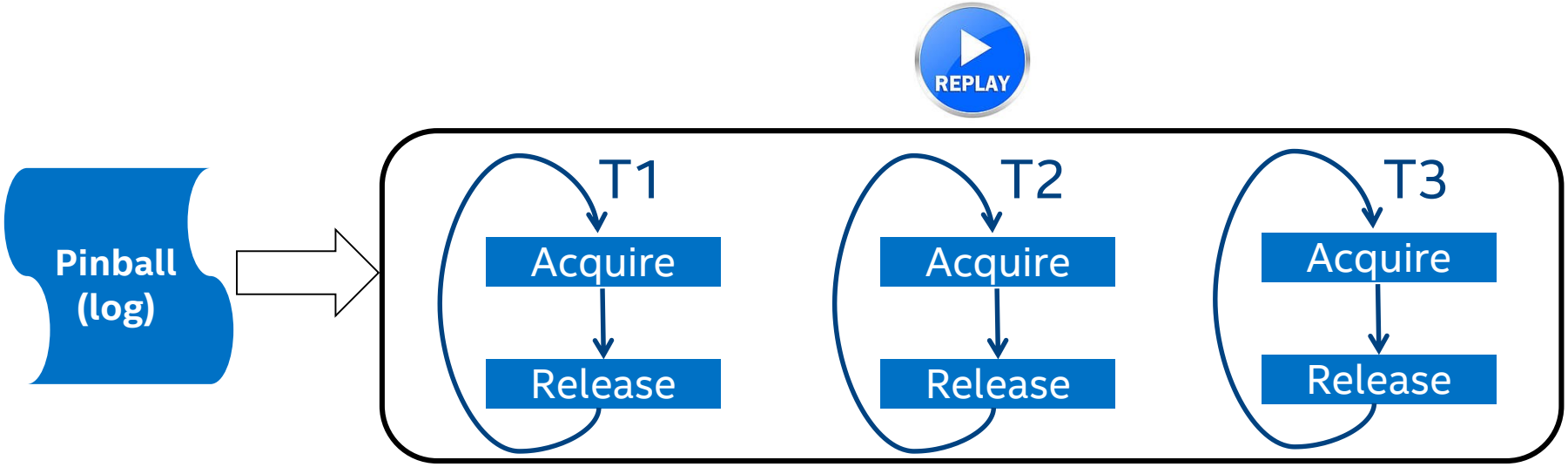
```
i=0
while record ./buggy_app
do
   if test -e core*
   then
      echo "Core dumped"
      exit 1
   fi
   i=`expr $i + 1`
   echo "=== $i ==="
   rm -rf pinball
done
```

Record the execution

Stop if we hit the bug

Delete the pinball
and
Keep trying

Deterministic Debugging with Replay

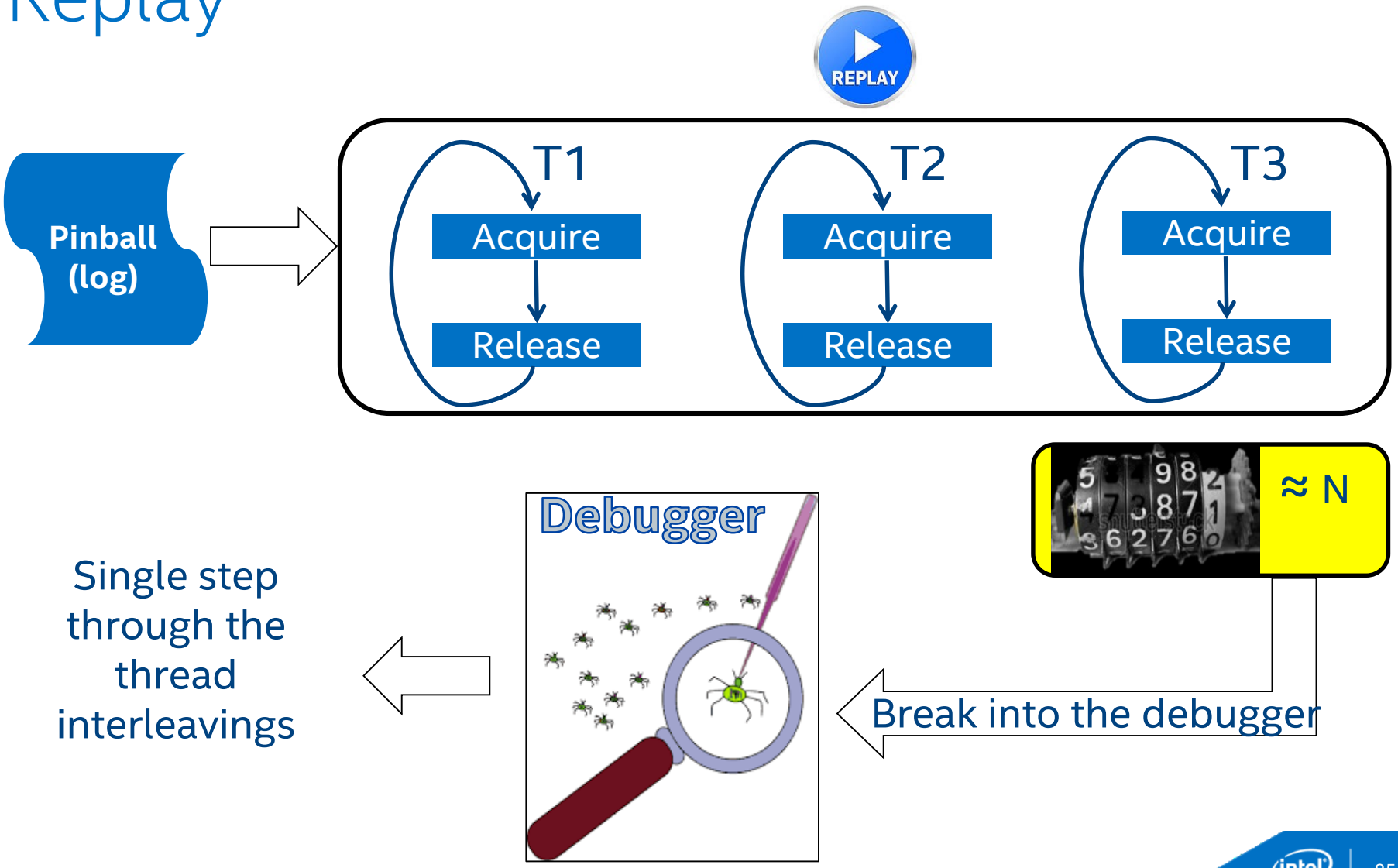


- **Challenge**: in which loop iteration did the bug happen?
  - Each thread had executed > 500K iterations
  - GDB conditional break points: too slow for remote debugging

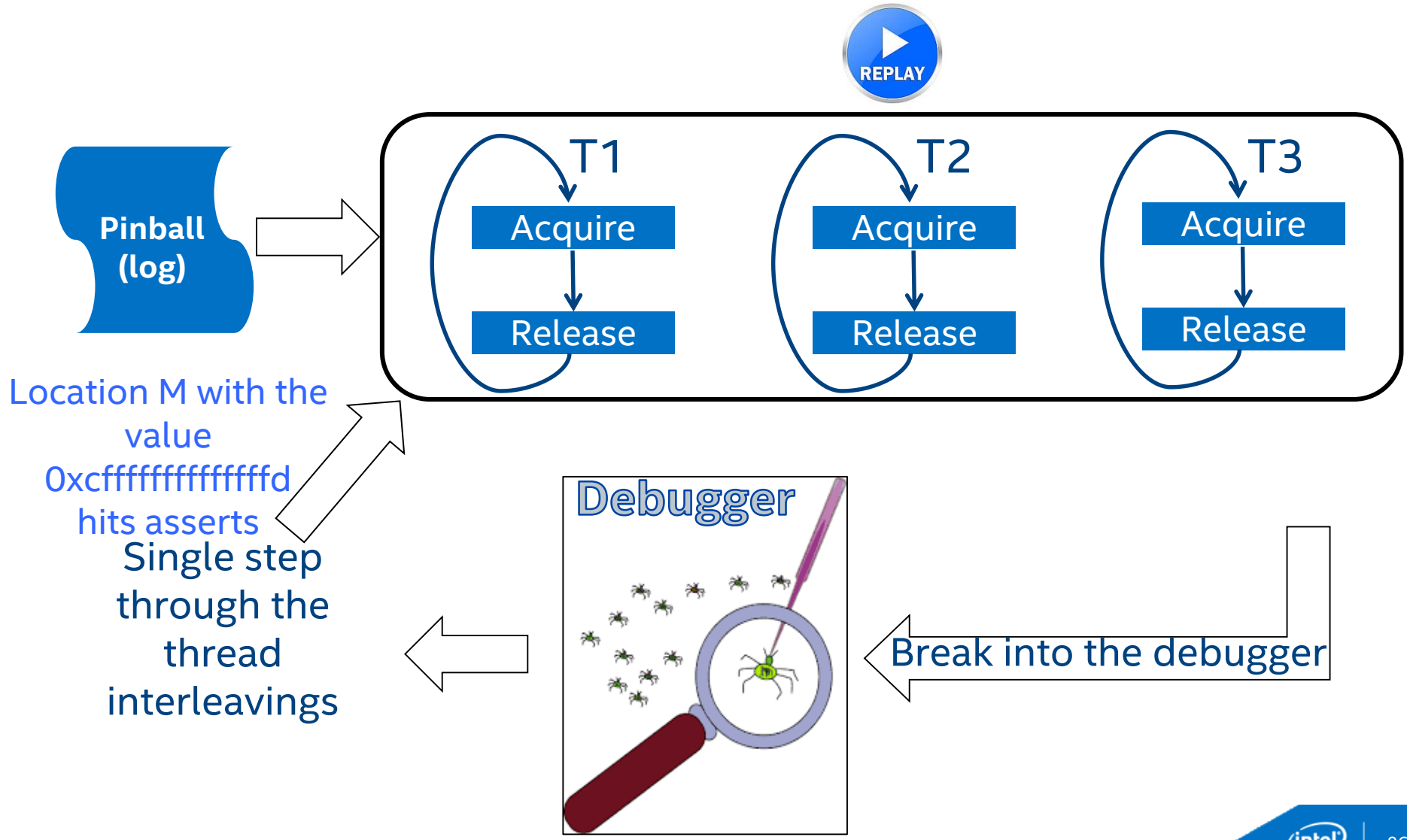
Deterministic Debugging with Replay

- **Challenge**: in which loop iteration did the bug happen?
  - Each thread had executed > 500K iterations
  - GDB conditional break points: too slow for remote debugging
- **Solution**: Pin tools to count/intercept/monitor interesting events

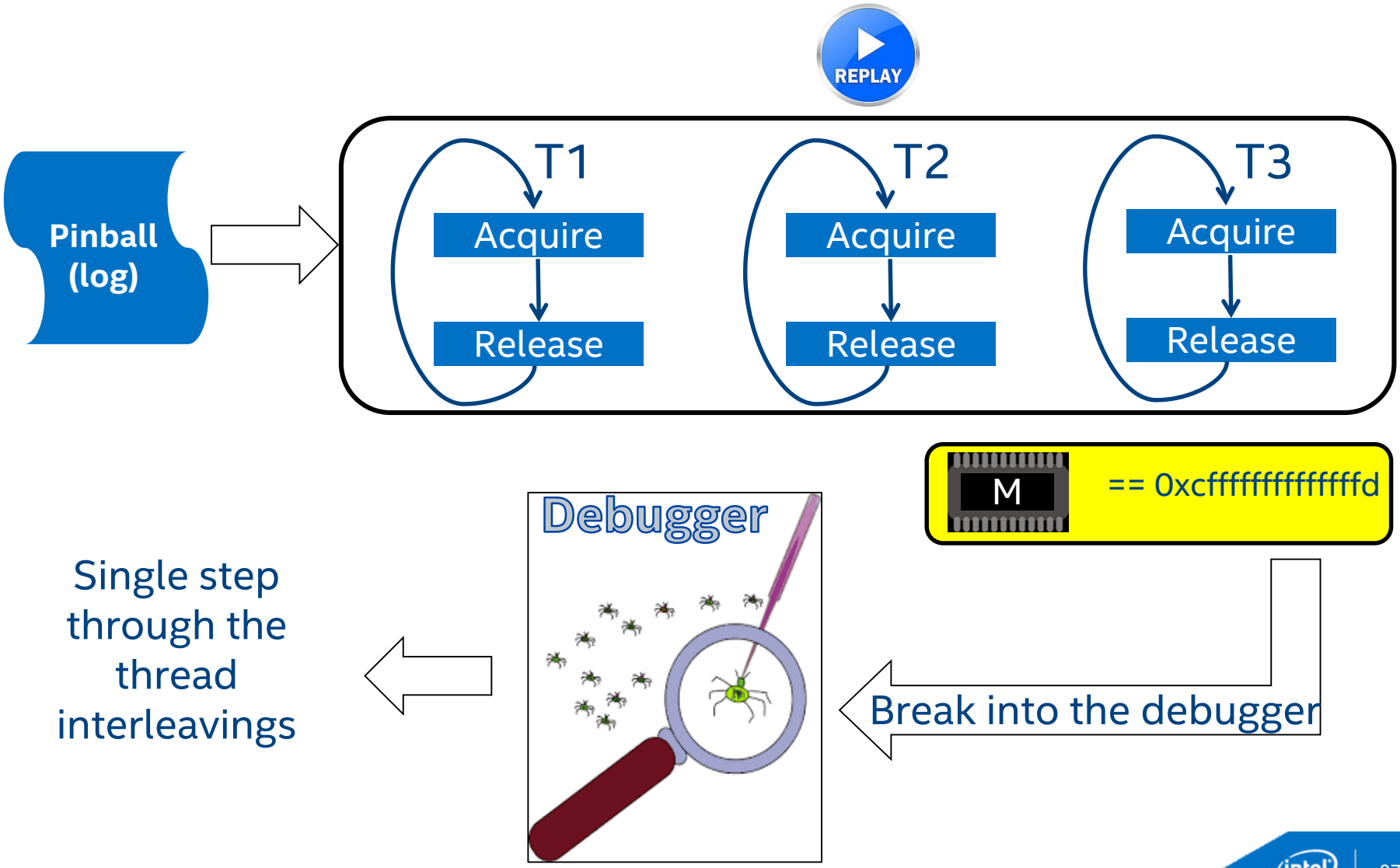
Access to Rich Set of Pin APIs During Replay



Effective Debugging with Faithful Replay



Breaking into the Debugger Just In Time



Dubious g++ Code Generation Corrupts Algorithm!

- Source code

```
#define WAIT        (0xffffffffffffffff)
#define READY       (0xdfffffffffffffff)
#define ACQUIRE     (0xcfffffffffffffff)
L->flag = READY // flag is 64-bit and cache line aligned
```

Start with L->flag: 0xcfffffffffffffc

T1

```
movl   $0xffffffffd,(%rax)
//L->flag: 0xcfffffffffffffd

movl   $0xdfffffff,0x4(%rax)
//L->flag: 0xdfffffffffffffff
```

T2

```
val = SWAP(&L->flag, WAIT)
//L->flag: 0xffffffffffffffff

switch(val) {
 case READY: /* do something */ break;
 . . .
 default:
  while (L->flag != READY);
  // T1 will never make L->flag READY
}
```

PinPlay was the Savior

- **Problem**: a non-deterministic concurrency bug
  - Not noticeable at the source (the point of write) or sync (read a clobbered, yet valid, 64-bit value)
  - Not reproducible with single-stepping the source
- **Reason**: a code generation defect that split a 64-bit write into two 32-bit writes, creating a small window of inconsistent state
- **Takeaway**: a reliable record/replay tool is invaluable in identifying concurrency bugs

Dynamic Program Slicing with Replay

Joint work with Yan Wang, Rajiv Gupta, and Iulian Neamtiu
University of California, Riverside
Eclipse GUI by David Wootton (Intel Corporation) sponsored by DARPA

Program Slicing

Definition: Slice(v@S)

Slice of v at S is the set of statements involved in computing v 's value at S. *[Mark Weiser, 1982]*

Static slice is the set of statements that **COULD** influence the value of a variable for **ANY** input

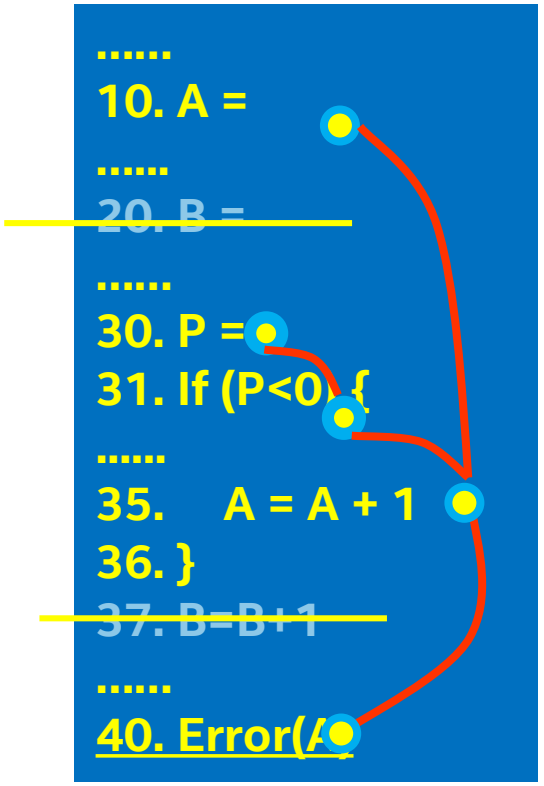
- Construct **static** dependence graph
  - Control dependences
  - Data dependences
- Traverse dependence graph to compute slice
  - Transitive closure over control and data dependences

Problem : Too many statements in a slice

Dynamic Slicing

Dynamic slice is the set of statements that **DID** affect the value of a variable at a program point for **ONE** specific execution.

[Korel and Laski, 1988]



Dynamic Slice (A@40) = {10, 30, 31, 35, 40)

Dynamic Slicing: Requirements

1. **Collect Execution trace**
   - **control flow trace** -- dynamic control dependences
   - **memory reference trace** -- dynamic data dependences

2. **Construct** a **dynamic** dependence graph

3. **Traverse** dynamic dependence graph to **compute slices**

Challenges : Run-time and Space overhead

Goals of Slicing Research

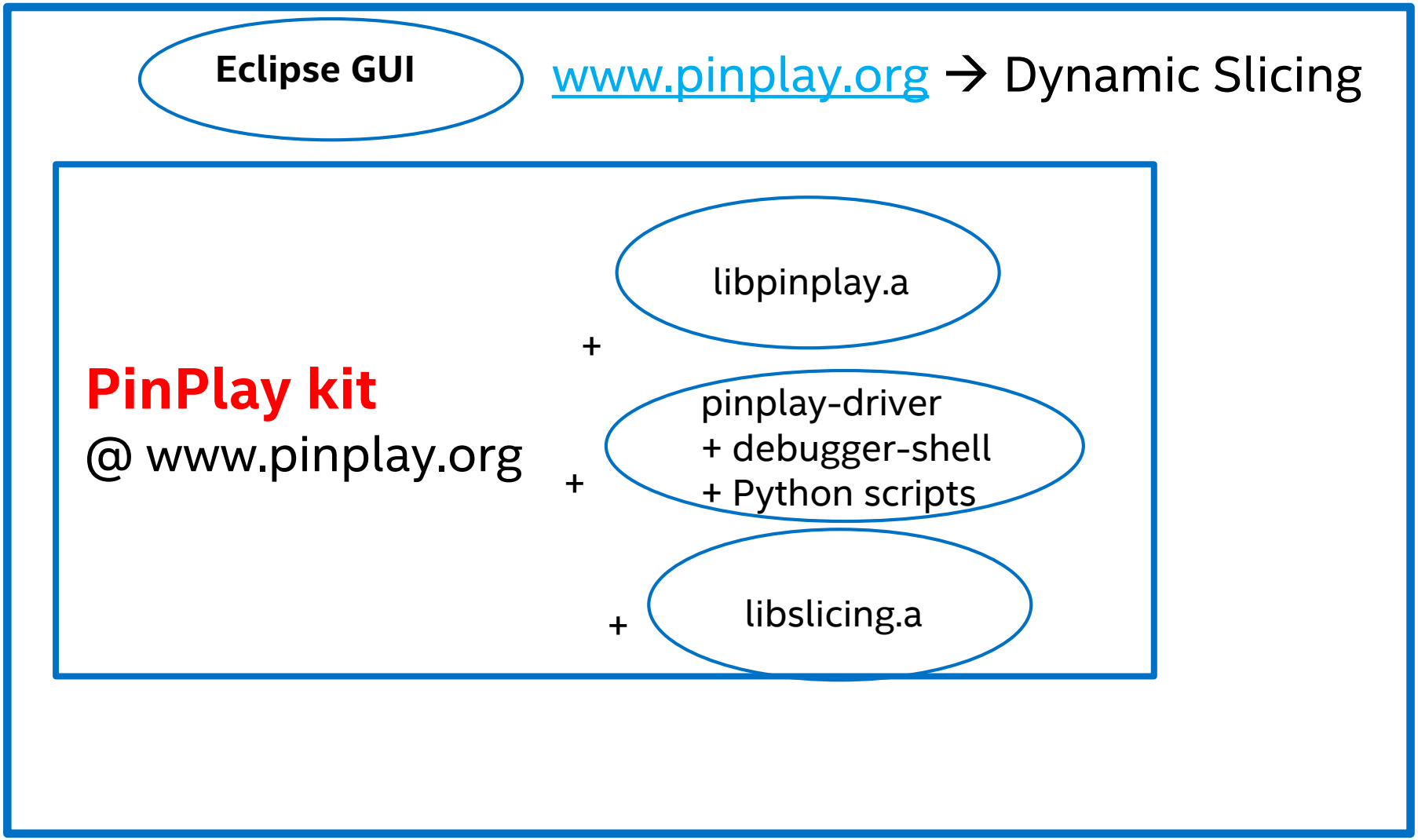
Practical Dynamic Slicing for Replay-based Interactive Debugging

Focus on a recording of a 'buggy region':

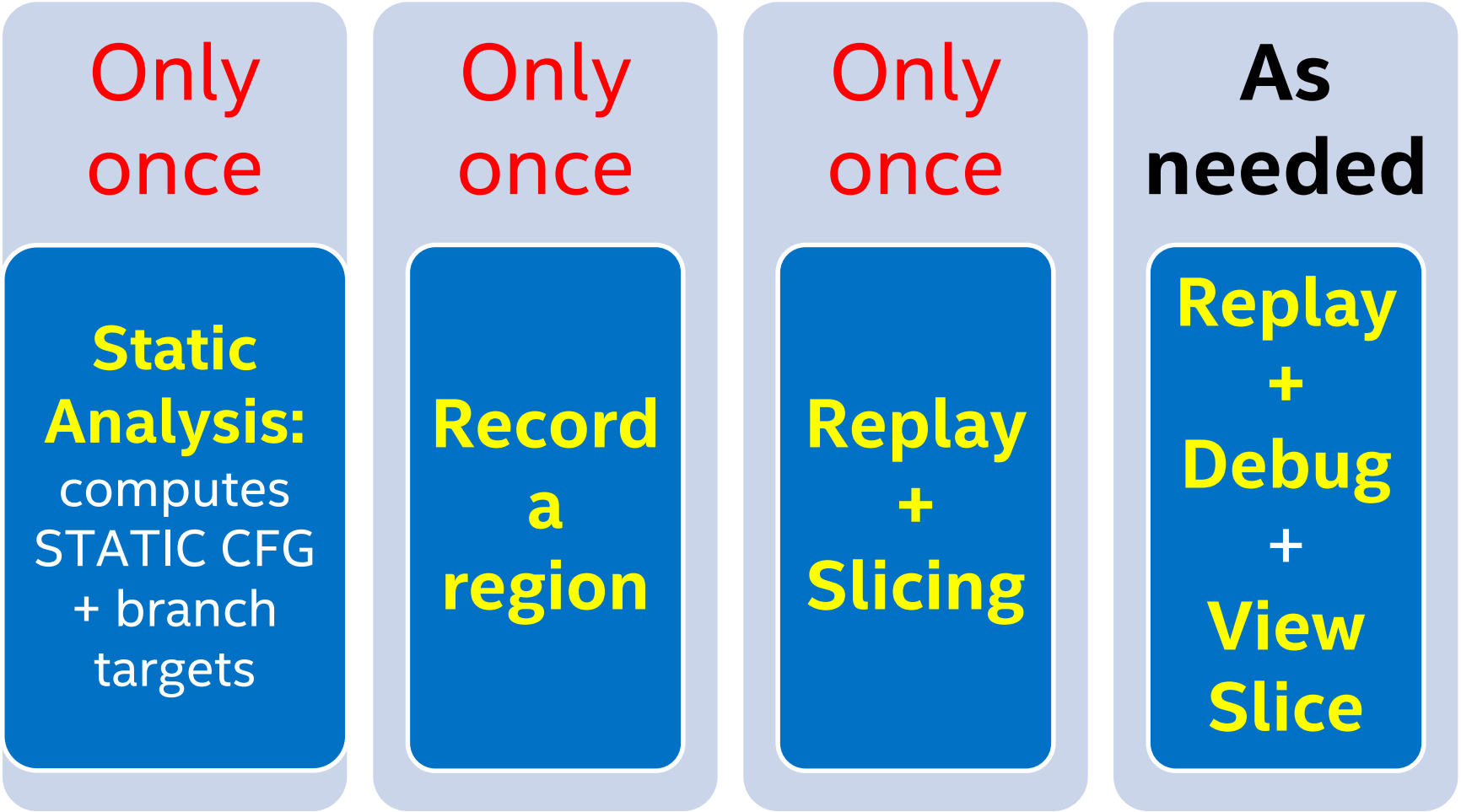
- ❑ *Manageable overhead*
- ❑ *Repeatability : Slice once → use across multiple debug sessions*

- Develop a set of commands that integrate dynamic slicing with interactive debugging

Distribution model : PinPlay kit + Eclipse GUI



Slicing with PinPlay: Steps



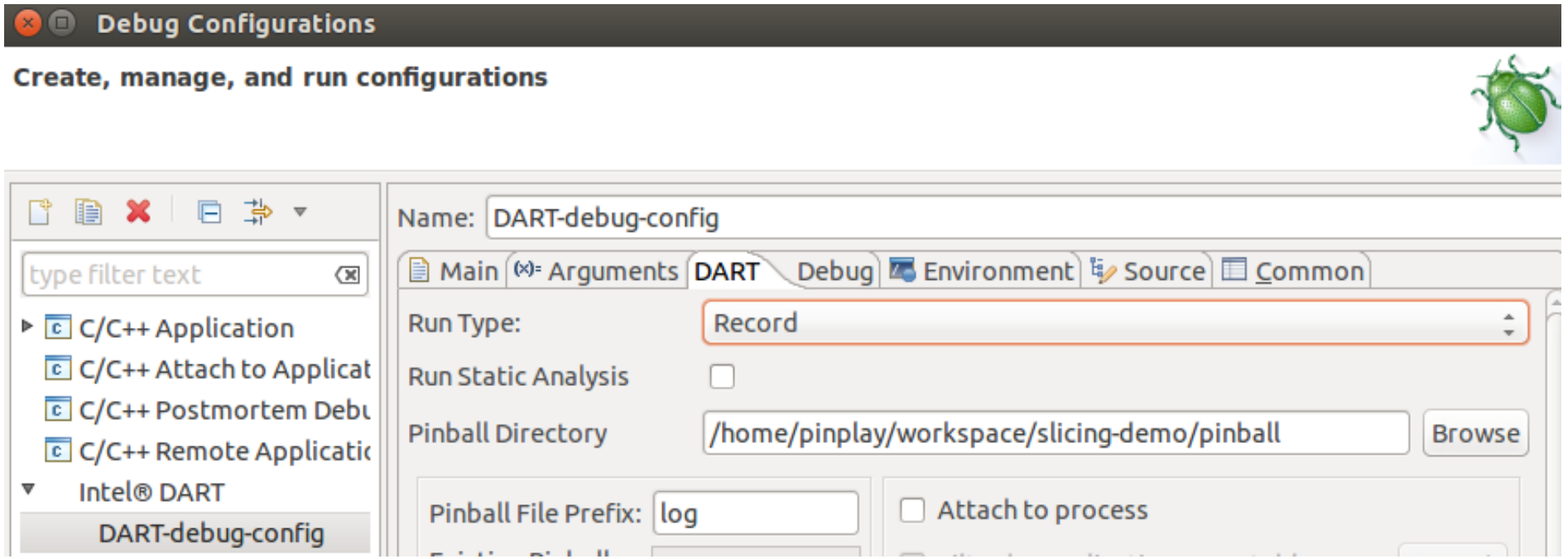
Time Overhead for PARSEC

Slicing time overhead:

- 10 slices for the last 10 different read instructions, spread across five threads, for region length **1 million instructions (main thread)**
- Average dynamic information tracing time: 51 seconds
- Average size of slice: 218K dynamic instructions
- Average slicing time: 585 seconds

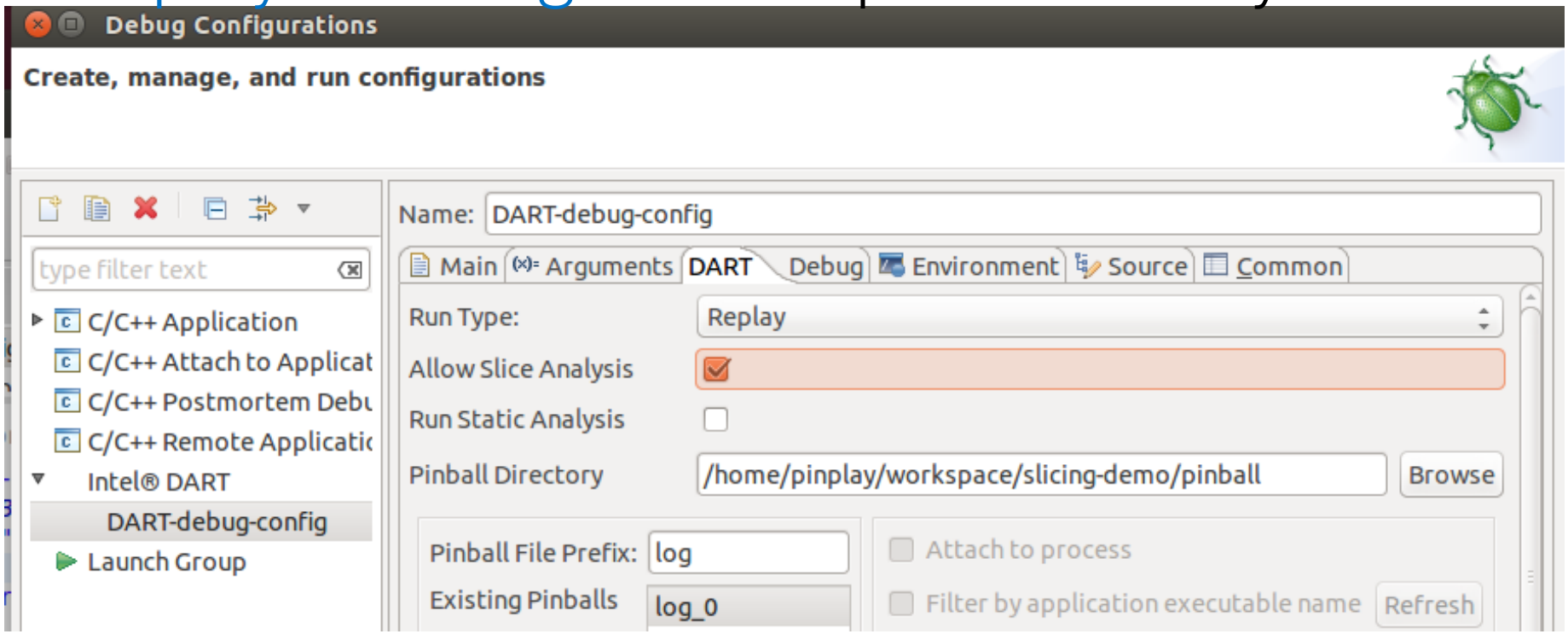
Record with eclipse + PinPlay

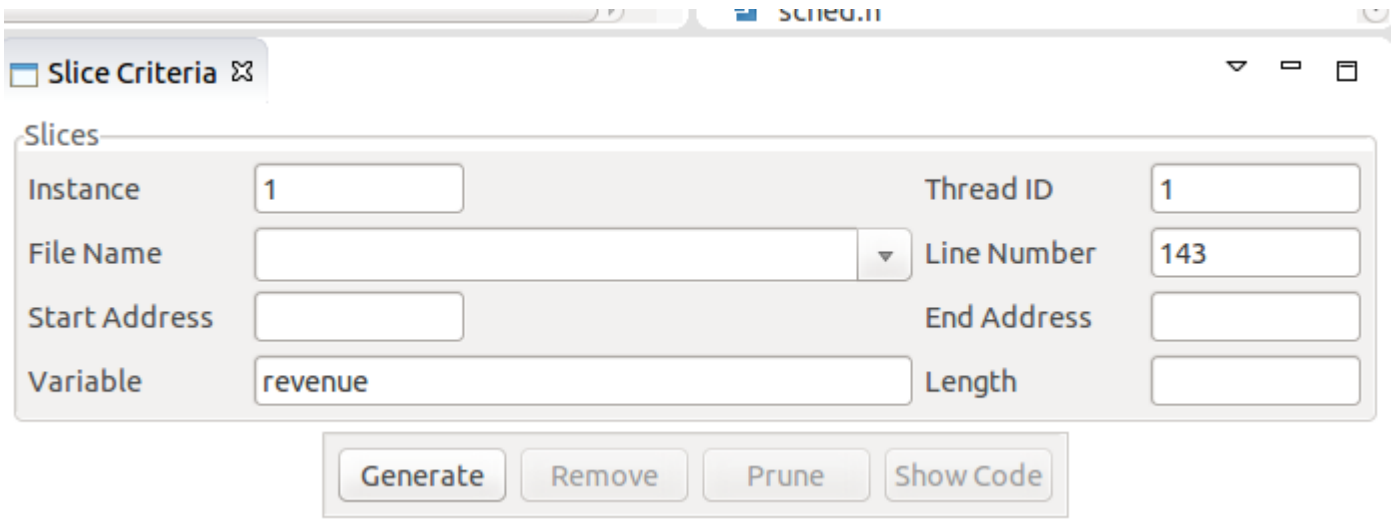
```
pinplay-VirtualBox:~/DrDebug/Slicing> EclipseDART/eclipse
```



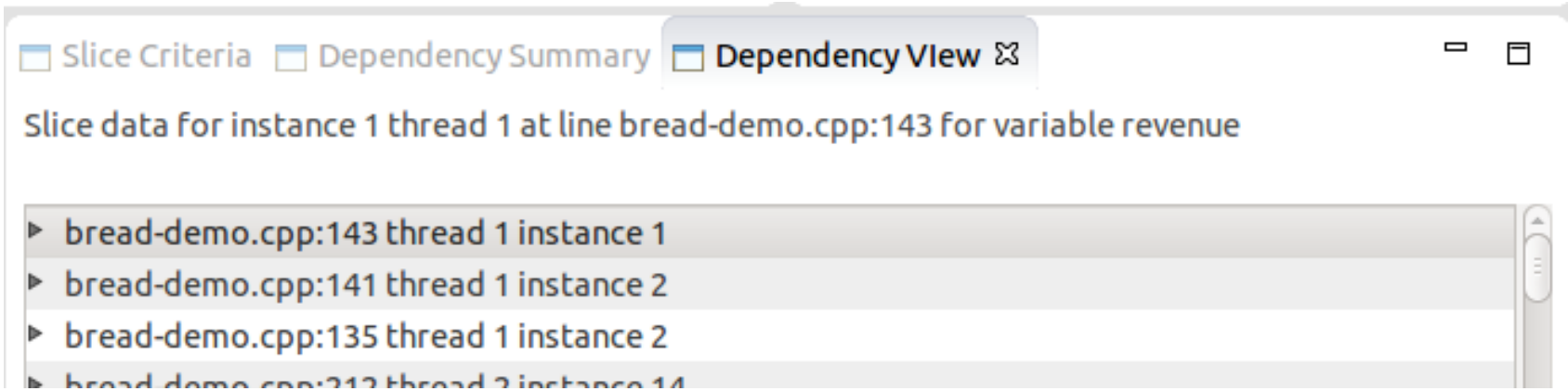
Record ON

Replay + Slicing with eclipse + PinPlay





Replay + Slicing with eclipse + PinPlay (contd.)




```
135     for (w = 0;  w < NumWorkers;  w++)
136     {
137         pthread_join(workers[w]._tid, 0);
138
139         for (i = 0;  i < NUMBER_BREADS;  i++)
140             breadOrders[i] += workers[w]._breadOrders[i];
141         revenue += workers[w]._totalRevenue;
142     }
```

Schedule

~~9:00 – 9:45 PinPlay and PinADX~~

~~9:45 – 10:15 DrDebug demo~~

~~10:15 – 10:30 Testimonial (Milind)~~

~~10:30 – 11:00 Slicing demo~~

<11 – 11:20 Break>

11:20 – 11:50 DCFG (Chuck)

11:50 – 12:15 Maple (Cristiano)

12:15 – 12:30 Wrap-up + Q&A (all)

Dynamic Control-Flow Graph (DCFG)

Chuck Yount
Intel Corporation

Overview

Tutorial goals

- Show how to create a Dynamic Control-Flow Graph (DCFG) from a PinPlay-enabled tool
- Illustrate additional control-flow analysis enabled by PinPlay's ability to replay execution

Agenda

- DCFG Definition
- How to make a DCFG
- How to use a DCFG
- Optional DCFG-Trace creation and usage

DCFG web site for documentation and download

- https://software.intel.com/en-us/articles/pintool-dcfg
- Or, http://pinplay.org and follow the DCFG link

DCFG definition

Control-Flow Graph [Allen 1970] (CFG)

- Directed graph in which nodes represent basic blocks and edges represent control-flow paths
- Basic block: linear sequence of instructions having one entry point and one exit point
- Used in many compiler and analysis algorithms

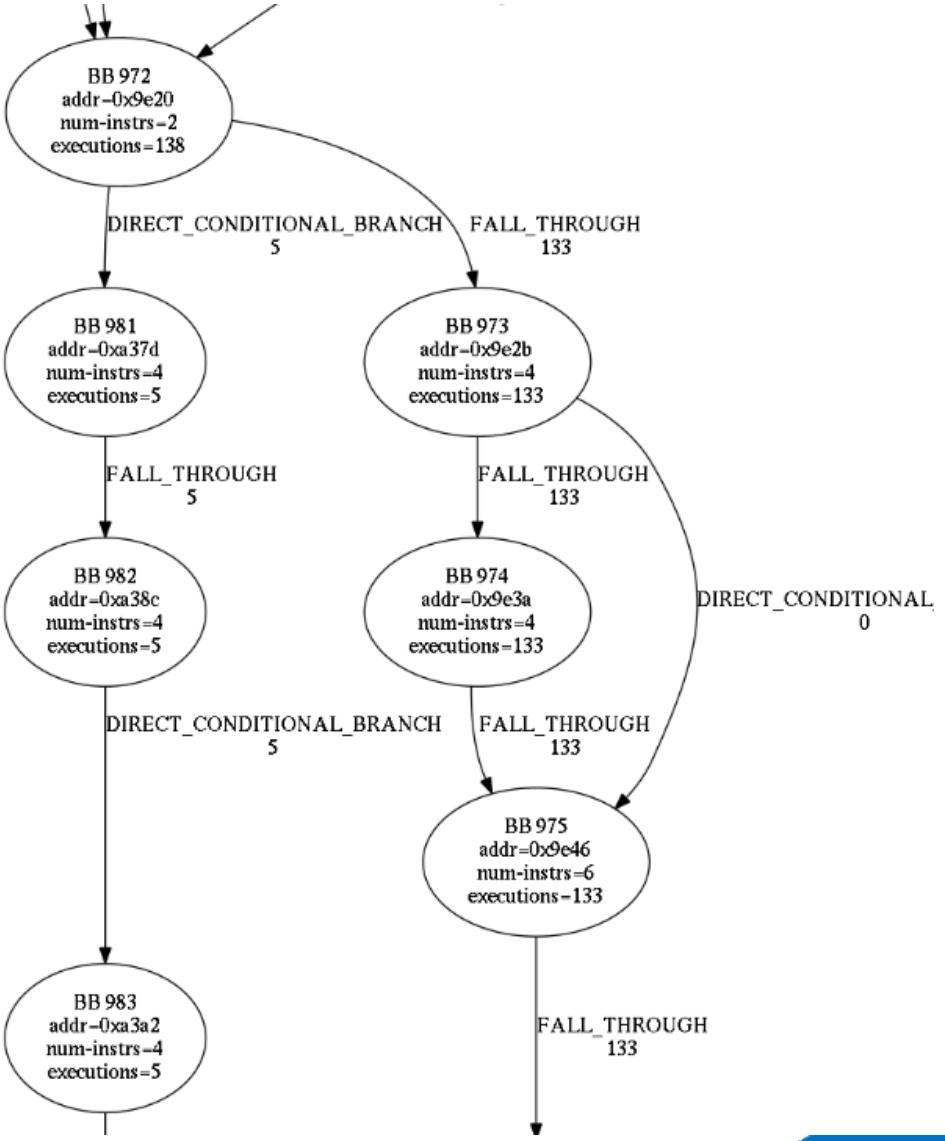
Dynamic Control-Flow Graph (DCFG)

- Defined by and extracted from a particular execution of a program
- Edges augmented with per-thread execution count; basic-block and other counts can be derived from these
- Need not contain non-executed edges or blocks
- May include paths due to exceptions, etc.

Example DCFG snippet

Shows a nested conditional

- BB (basic block) 972 entered 138 times
  - If-then-else construct
  - Conditional branch to BB 981 (left side) taken 5 times
  - Fall-through to BB 973 remaining 133 times
    - If-then construct
    - Fall-through to BB 974 always taken
- This image was created using the 'dcfg-to-dot' utility program included in the package



Prerequisite tutorial setup

Virtual machine

- Install Oracle VM VirtualBox or equivalent VM
- Download the PinPlay virtual appliance and import into the VM

Or, local Linux install

- Download the PinPlay package from http://pinplay.org
- Extract the tarball
- Set PIN_ROOT and PIN_HOME environment vars to the install directory

How to create a DCFG

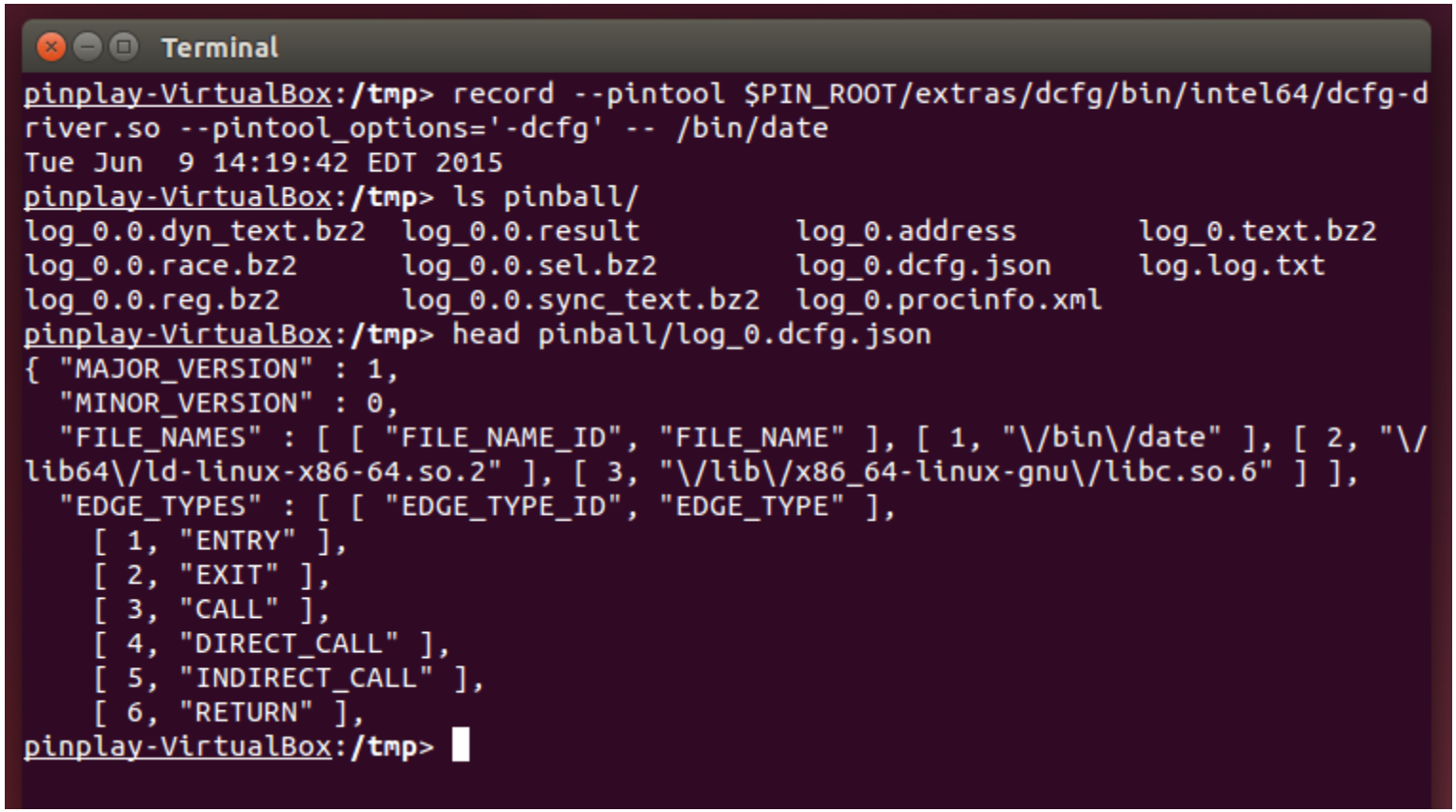
Run the ‘record’ script with the DCFG driver tool

- `record --pintool $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-driver.so --pintool_options='-dcfg' -- /bin/date`

Creates new file in ‘pinball’ directory: log_0.dcfg.json

- Contains data on logged process in JSON format
  - Meta-data: Images, symbols, and debug info
  - Fundamental CFG elements: basic blocks and edges
  - Derived structures: routines and loops
  - See “DCFG format description” on website for full documentation

DCFG creation example



DCFG-creation code

In $PIN_ROOT/extras/dcfg

- See examples/makefile.rules
- examples/dcfg-driver.cpp provides minimal DCFG functionality

```
#include "dcfg_pin_api.H" …
DCFG_PIN_MANAGER* dcfgMgr =
DCFG_PIN_MANAGER::new_manager();
    if (dcfgMgr->dcfg_enable_knob())
        dcfgMgr->activate(&pinplay_engine);
```

- Link with lib/*arch*/libdcfg-pinplay.a
  - Provides '-dcfg' and other DCFG command-line options

Reading a DCFG file (standalone tool)

C++ API usable from standalone program or from a PinPlay tool

- Documentation at DCFG web site
  - "Hierarchical Index" is a good starting point
- Example standalone code in examples/dcfg-reader.cpp
  - Link with lib/*arch*/libdcfg.a
  - Create a DCFG_DATA object and read contents from a file

```
#include "dcfg_api.H " …
DCFG_DATA* dcfg = DCFG_DATA::new_dcfg();
dcfg->read(filename, errMsg);
```

  - Most data is accessed by getting one or more IDs, e.g.,

```
dcfg->get_process_ids(proc_ids);
```

  - Then, use an ID to get a pointer to detailed data, e.g.,

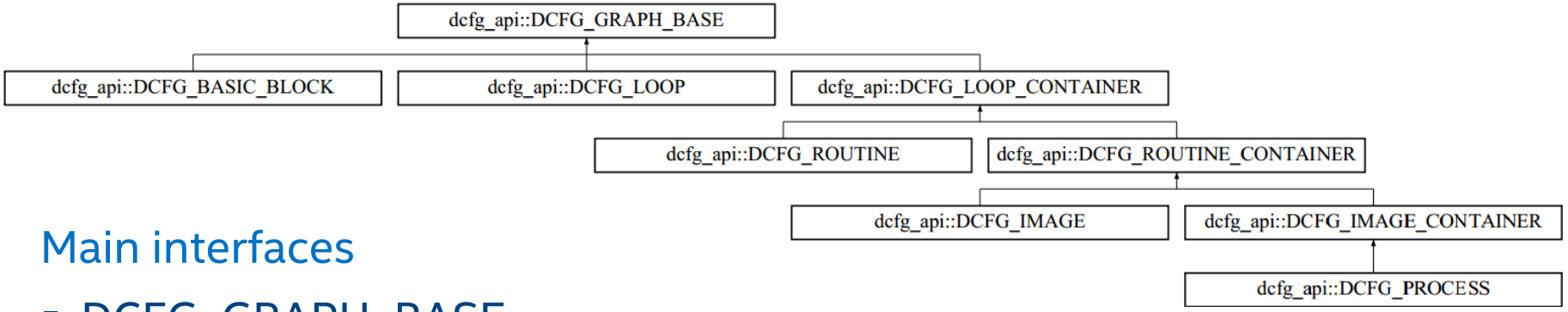
```
DCFG_PROCESS_CPTR pinfo =
  dcfg->get_process_info(proc_ids[i]);
```

  - Similar code to get data on images, routines, loops, basic blocks, edges, etc.
- Run example code to print high-level statistics
  - `$PIN_ROOT/extras/dcfg/bin/intel64/dcfg-reader pinball/log_0.dcfg.json`

Standalone-code example

```
Terminal
pinplay-VirtualBox:/tmp> $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-reader pinball/log_0.dcfg.json
| head -25
Reading DCFG from 'pinball/log_0.dcfg.json'...
Summary of DCFG:
 Num processes           = 1
 Process 2266
  Num threads = 1
  Instr count = 222435
  Num edges   = 4238
  Num images  = 3
  Image 1
   Load addr         = 0x400000
   Size              = 2155712
   File              = '/bin/date'
   Num basic blocks = 220
   Num routines      = 41
   Num loops         = 4
  Image 2
   Load addr         = 0x7ff1f0dff000
   Size              = 2245064
   File              = '/lib64/ld-linux-x86-64.so.2'
   Num basic blocks = 1278
   Num routines      = 74
   Num loops         = 82
  Image 3
   Load addr         = 0x7ff1d8c07000
   Size              = 3949248
   File              = '/lib/x86_64-linux-gnu/libc.so.6'
```

API interface inheritance hierarchy



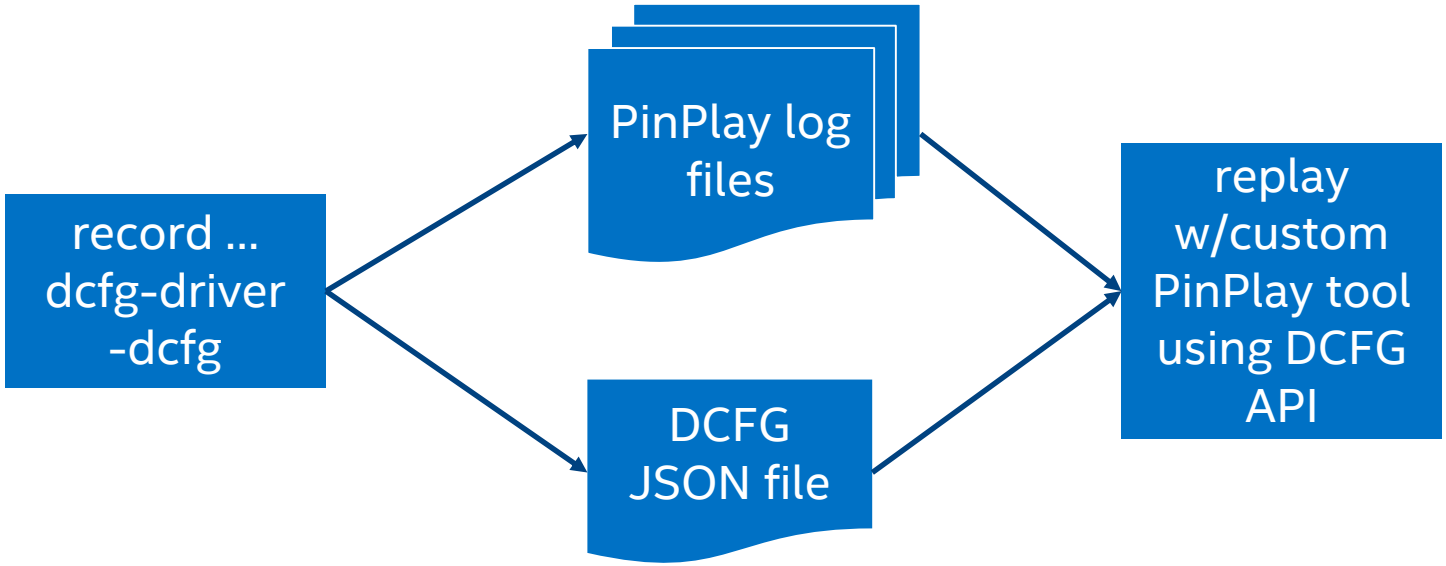
Main interfaces

- DCFG_GRAPH_BASE
  - Base interface for anything with basic blocks and edges
- DCFG_{LOOP,ROUTINE,IMAGE}_CONTAINER
  - Base interface for something containing the given part
  - Example: a process and an image are both routine containers
- DCFG_{BASIC_BLOCK,LOOP,ROUTINE,IMAGE,PROCESS}
  - Main hierarchical structural components
- DCFG_EDGE (not shown)
  - Control-flow path between any two nodes (usually BBs)

Using a DCFG during replay

Provides access to DCFG structures during replay

- Can instrument code based on analysis of edges, loops, etc.

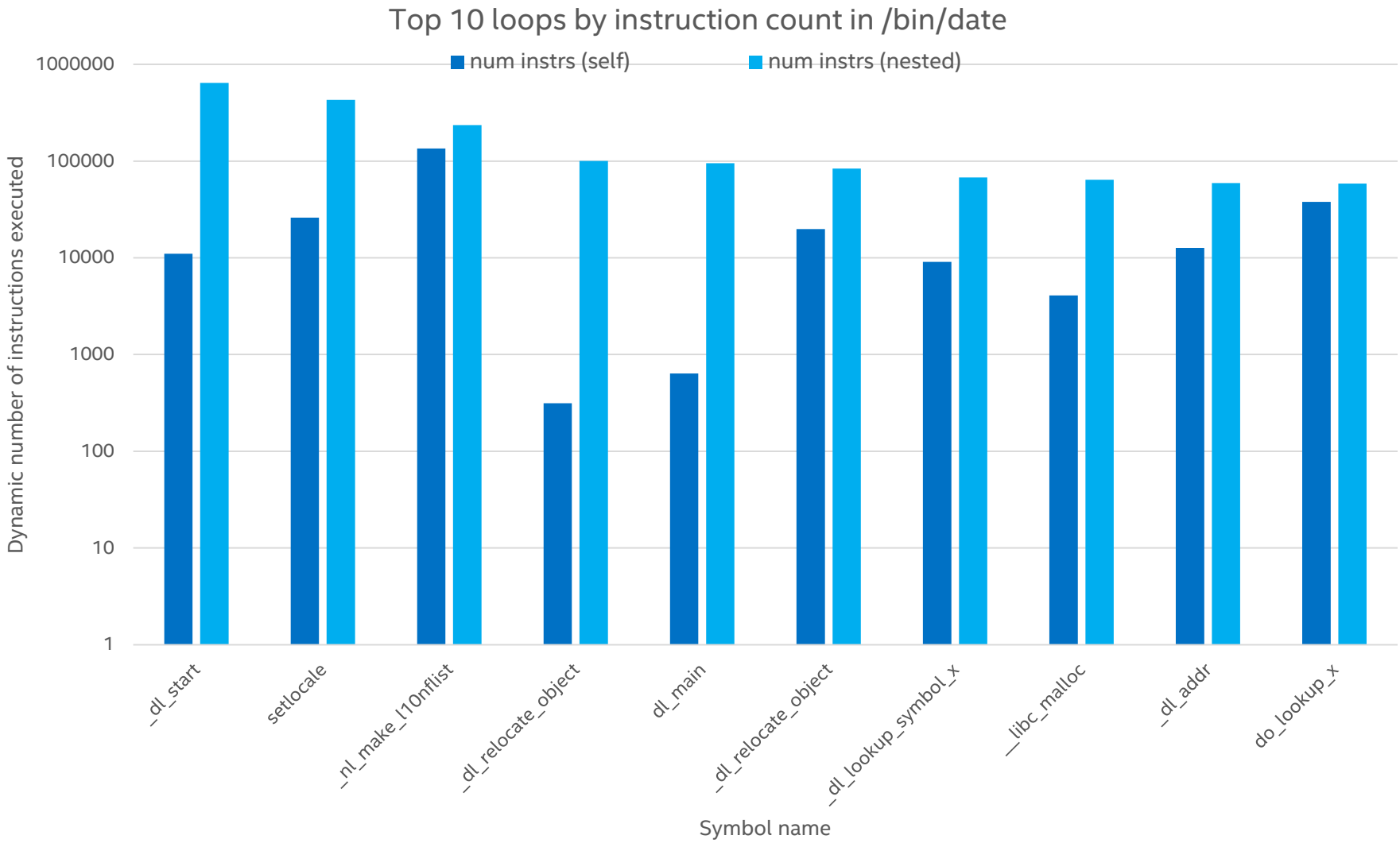


Reading a DCFG file (PinPlay tool)

Same C++ API can be used in a PinPlay tool

- Example code in examples/loop-tracker.{cpp,H}
  - Collects statistics from loops
  - Link with lib/*arch*/libdcfg-pinplay.a as before
  - Noteworthy methods in LOOP_TRACKER class
    - activate(): reads the DCFG file and adds instrumentation functions
    - processDcfg(): collects data from the DCFG data to be used later
    - handleTrace(): adds analysis routine at DCFG basic block heads
    - enterBb(): analysis code that is run at each basic block
      - Remembers previous BB to determine DCFG edge
      - Uses data from processDcfg() to track loop entry and exit
      - Collects statistics on each loop
    - printData(): outputs statistics for each loop in tabular format
- Run loop tracker
  - `replay --pintool $PIN_ROOT/extras/dcfg/bin/intel64/loop-tracker.so --pintool_options='-loop-tracker:dcfg-file pinball/log_0.dcfg.json' -- pinball/log_0`
  - Creates loop-stats.csv by default

Data from example loop-tracker run



DCFG PinPlay tool example

Terminal

```
pinplay-VirtualBox:/tmp> replay --pintool $PIN_ROOT/extras/dcfg/bin/intel64/loop-tracker.so --p
intool_options='-loop-tracker:dcfg-file pinball/log_0.dcfg.json' -- pinball/log_0
Tue Jun  9 14:19:42 EDT 2015
pinplay-VirtualBox:/tmp> head loop-stats.csv
loop id,thread id,image name,symbol name,source file,source line number,loop addr,num loop entr
ies,num iterations,ave iterations/entry,ave loop-nesting depth,num instrs (self),num instrs (ne
sted),ave num instrs/iteration (self),ave num instrs/iteration (nested)
0,0,"/lib64/ld-linux-x86-64.so.2",".text","unknown",0,0x7ff1f0e03a70,1,1,1.00,0.00,9379,222467,
9379.00,222467.00
100,0,"/bin/date","**entry**","unknown",0,0x402e79,1,15,15.00,1.00,3658,6464,243.87,430.93
144,0,"/bin/date","**entry**","unknown",0,0x4034c3,5,11,2.20,2.00,182,182,16.55,16.55
180,0,"/bin/date","**entry**","unknown",0,0x4041d0,1,1,1.00,2.00,86,86,86.00,86.00
218,0,"/bin/date","**entry**","unknown",0,0x409390,1,1,1.00,1.00,22,22,22.00,22.00
231,0,"/lib64/ld-linux-x86-64.so.2",".text","unknown",0,0x7ff1f0dfff45,1,4,4.00,1.00,24,24,6.00
,6.00
274,0,"/lib64/ld-linux-x86-64.so.2",".text","unknown",0,0x7ff1f0e01239,1,9,9.00,1.00,156,156,17
.33,17.33
296,0,"/lib64/ld-linux-x86-64.so.2",".text","unknown",0,0x7ff1f0e01409,1,23,23.00,1.00,201,201,
8.74,8.74
336,0,"/lib64/ld-linux-x86-64.so.2",".text","unknown",0,0x7ff1f0e02032,1,1,1.00,1.00,8,8,8.00,8
.00
```

Utility to create visual DCFG

Sample tool to output the DCFG of one routine in 'DOT' format

- Usage information
  - `$PIN_ROOT/extras/dcfg/bin/intel64/dcfg-to-dot`
    `Writes graph of routine containing node-id to a DOT-language file.`
    `usage: dcfg-to-dot <input-dcfg-file> <output-dot-file> <node-id>`
- The 'node-id' can be any node in the routine for which you want to create the dot-file
- Use 'dot' program to convert to PDF, PNG, etc.
  - Download and install from http://www.graphviz.org/

DCFG-Trace

Additional optional record of entire sequence of edges

- Enabled by using -dcfg:write_trace option
- Allows standalone-tool to analyze sequence
  - Use DCFG_TRACE_READER API
  - See summarizeTrace() in examples/dcfg-reader.cpp
    - Inputs DCFG-Trace file
    - Outputs table of edges and basic-blocks at edge targets
- PinPlay tool provides many more possibilities (all Pin APIs) and does not require disk-space for large edge-trace file

DCFG-Trace example

```
Terminal
pinplay-VirtualBox:/tmp> record --pintool $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-driver.so --pi
ntool_options='-dcfg -dcfg:write_trace' -- /bin/date
Tue Jun  9 20:13:35 EDT 2015
pinplay-VirtualBox:/tmp> $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-reader pinball/log_0.dcfg.json
pinball/log_0.trace.json | grep -A10 'edge id'
Reading DCFG from 'pinball/log_0.dcfg.json'...
Reading DCFG trace for PID 2921 and TID 0 from 'pinball/log_0.trace.json'...
edge id,basic-block id,basic-block addr,basic-block symbol,num instrs in BB
2549,410,0x7fbe0bf6cabf,".text",1
2727,411,0x7fbe0bf6cac5,".text",11
2285,414,0x7fbe0bf6cb1f,".text",2
307,412,0x7fbe0bf6cb08,".text",2
309,413,0x7fbe0bf6cb13,".text",4
300,414,0x7fbe0bf6cb1f,".text",2
307,412,0x7fbe0bf6cb08,".text",2
309,413,0x7fbe0bf6cb13,".text",4
300,414,0x7fbe0bf6cb1f,".text",2
637,415,0x7fbe0bf6cb25,".text",4
Done reading 56096 edges.
```

Summary

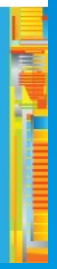
DCFG creation

- Minimal dcfg-driver.so PinPlay tool creates DCFG JSON file
  - Contains structure of basic-blocks, edges, loops, and more
- Optionally, DCFG-Trace file can be created
  - Contains sequence of edges

DCFG usage

- Example dcfg-reader standalone program inputs DCFG (and optionally DCFG-Trace) and prints summary data
- Example loop-tracker.so PinPlay tool inputs DCFG and instruments basic-blocks to track edges and loops

Triggering Bugs for Repeatable Analysis



Cristiano Pereira
Intel Corporation

Overview

- Tutorial goals
  - Describe a tool to trigger an existing concurrency error in a multi-threaded program execution
  - Demonstrate how PinPlay can be used to capture an execution that contains an exposed concurrency error
- Agenda
  - Maple: A tool and technique to trigger concurrency errors in a multi-threaded program
  - Brief overview of how to use Maple
  - How to capture a bug exposed by Maple with PinPlay and replay-debug it

Triggering Bugs for Repeatable Analysis

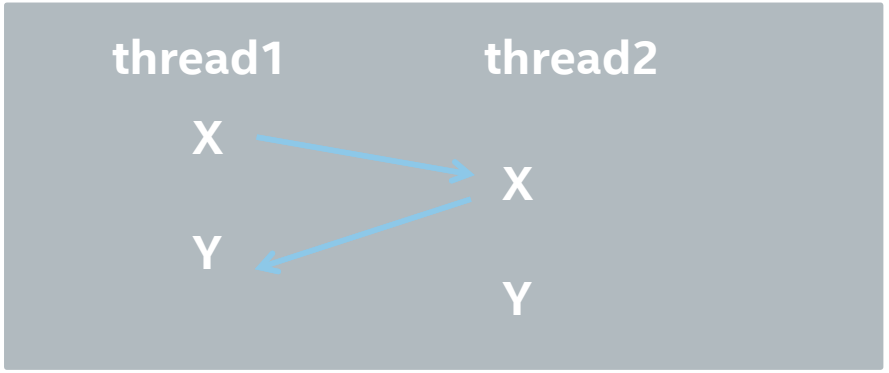
- Replay is a powerful technique for analyzing the root cause of concurrency bugs
- However, the bug must manifest during recording for replay to be applicable, and this may not be easy
- A mechanism for triggering concurrency bugs is desirable
  - Single-threaded: mostly driven by program input
  - Multi-threaded programs: driven by both program input and by thread-interleaving
  - Thread-interleaving depends on # of cores, cache sizes, memory latencies, OS scheduler, machine load, position of stars in the sky…

Motivation

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>

unsigned NUM_THREADS = 1;
unsigned global_count = 0;
void *thread(void *);

int main(int argc, char *argv[]) {
  pthread_t *pthread_id[100];
  NUM_THREADS = atoi(argv[1]);

  for(long i = 0; i < NUM_THREADS; i++)
    pthread_create(&pthread_id[i],
                   NULL, thread, (void *) i);

  for(long i = 0; i < NUM_THREADS; i++)
    pthread_join(pthread_id[i], NULL);

  assert(global_count==NUM_THREADS);
  exit(0);
}
```

```
void * thread(void * num)
{
X: unsigned temp = global_count;
   temp++;
Y: global_count = temp;
   return NULL;
}
```

What is the buggy interleaving?



How frequently?

 # crashes/10,000 runs (8 cores):

0 with 8 threads/run (0%)
2 with 32 threads/run (0.02%)

Maple Tool

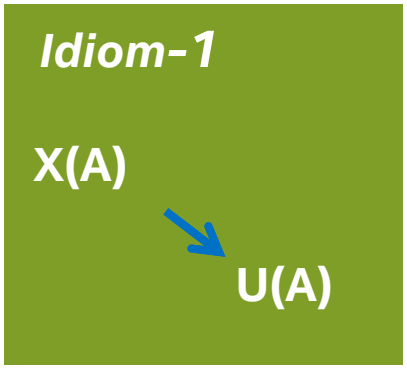
- Goal:
  - Expose executions that exhibit buggy behavior
- How:
  - Finds memory operations likely to lead to bugs
  - Produces a buggy run by manipulating thread schedule using Linux real time priorities
  - Implemented in PIN, easy to integrate with other pintools

Triggering Concurrency Bugs with Maple

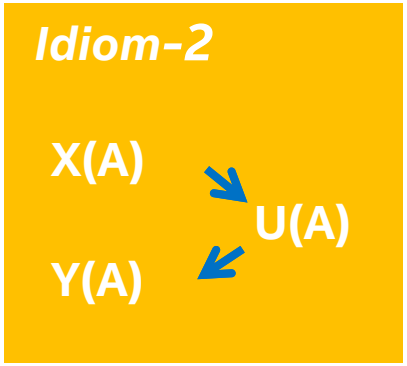
Maple: a coverage-driven testing tool for multithreaded programs, Yu et al. OOPSLA'12
https://github.com/jieyu/maple

- Maple uses a "best effort" approach by profiling the execution before hand and looking for specific bug patterns or idioms
  - An idiom is a sequence of memory operations and their dependencies across threads
  - Limits the search by observing that bugs are not caused by random memory operations
  - No completeness guarantees like systematic (formal) testing, but a practical technique that scales to relatively large programs
  - Details in the OOPSLA'12 paper, out of scope for this tutorial

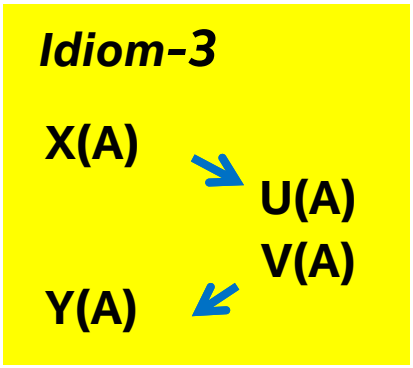
Idioms of bugs with 2 threads, ≤2 variables



Data race or ordering

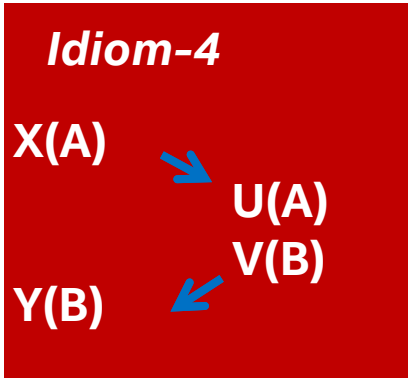


Low level atomicity

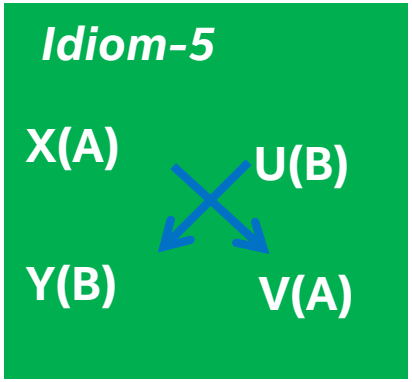


High level atomicity

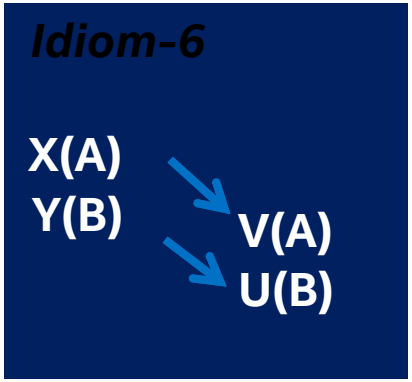
Single variable



Multi-variable atomicity



Deadlock



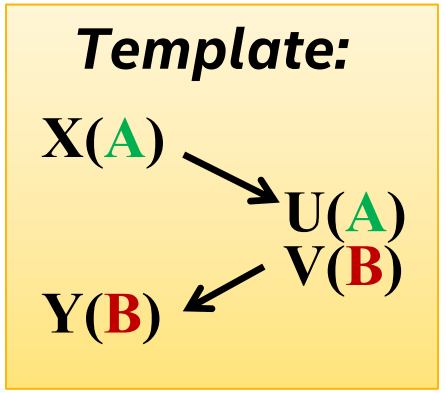
Unknown

Double variable

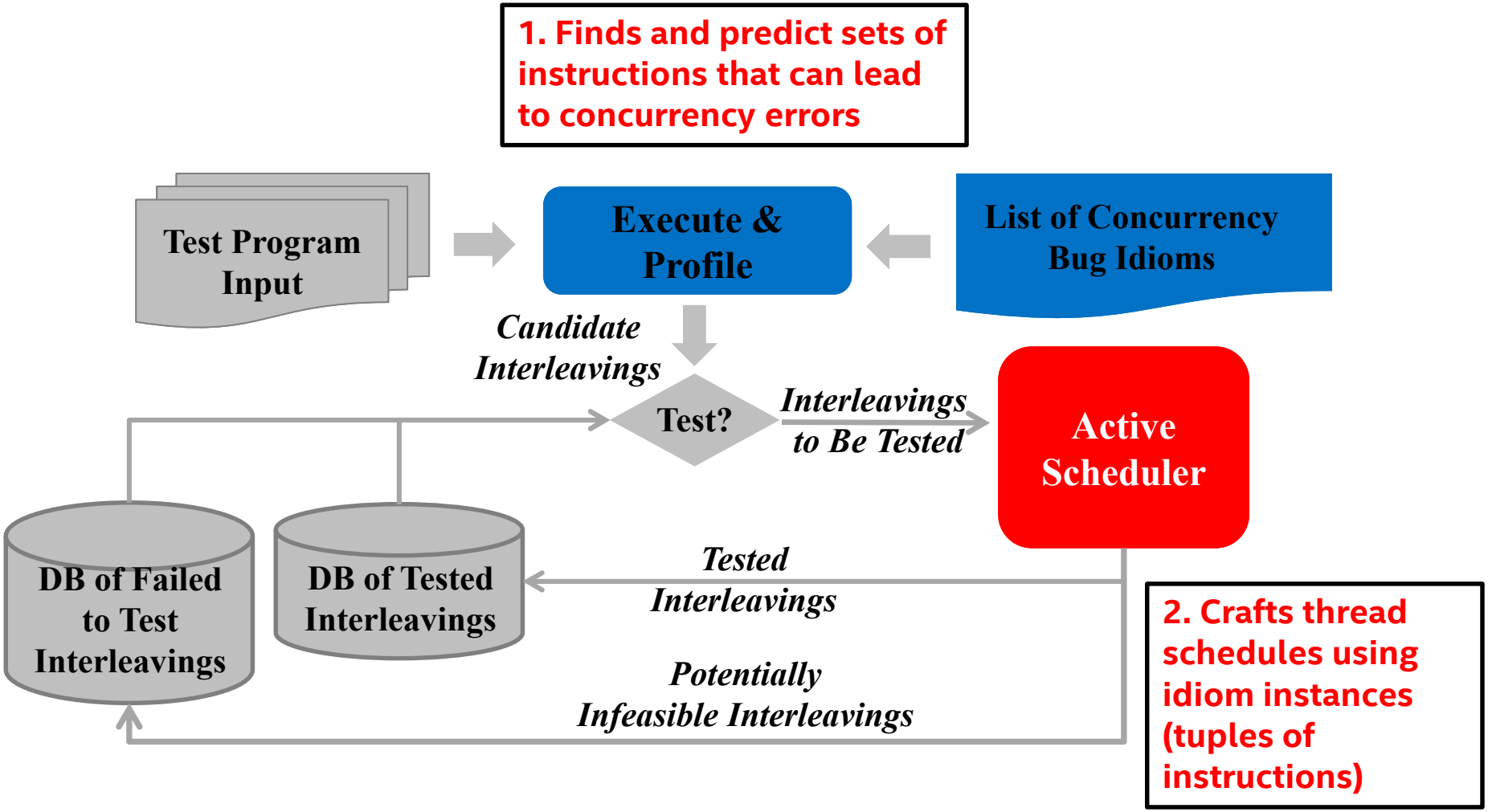
Idiom Example: MySQL Bug #169

```
int generate_table(...)
{
  lock(&LOCK_open);
  // delete entries
X: unlock(&LOCK_open);

Y: lock(&LOCK_log);
  log.write(...);
  unlock(&LOCK_log);

}
```

```
int mysql_insert(...)
{

U: lock($LOCK_open);
  // insert entry
  unlock($LOCK_open);

  lock(&LOCK_log);
  log.write(...);
V: unlock(&LOCK_log);

}
```



Maple Bug Triggering Overview



Exposing Idiom Instances

- An *idiom* instance, called an *iroot*, *consists of the* PCs of memory operations that fit within the idiom

```
2     IDIOM_1
        e0: WRITE   [79    ld          0x4c9f                ]
        e1: READ    [1212  ld          0x9f7e                ]
```

- A data base of *iroots* is built during profiling
- Maple then executes the program multiple times forcing each iroot to be exercised during the scheduling phase
- If an execution exposes bug, it is reproduced by repeating the iroot inter-thread dependencies

Maple and PinPlay Usage

- Maple exercises all interleavings in the iroot database
- When maple reproduces a thread interleaving, it only reproduces the interleaving involved in the idiom that exposed the bug
- All other sources of non-determinism are not controlled
  - Memory addresses may differ
  - Control-flow may differ
  - Interleaving is not guaranteed to repeat 100% of the time
- Combining Maple and PinPlay guarantees that an execution exposing the idiom instance can be repeated 100% deterministically and replay-debugged that way

Modified Shared Counter Example

Change lines as highlighted in the example program under:
/home/pinplay/maple/example/shared_counter/main.cc

```
17 #include <stdio.h>
18 #include <stdlib.h>
19 #include <pthread.h>
20 #include <assert.h>
21
22 unsigned LOOP_COUNT = 100;
23 unsigned NUM_THREADS = 1;
24 unsigned global_count = 0;
25 void *thread(void *);
26
27 int main(int argc, char *argv[]) {
28   long i;
29   pthread_t pthread_id[200];
30   NUM_THREADS = atoi(argv[1]);
31
32   for(i = 0; i < NUM_THREADS; i++)
33     pthread_create(&pthread_id[i], NULL, thread, (void *) i);
34   for(i = 0; i < NUM_THREADS; i++)
35     pthread_join(pthread_id[i], NULL);
36
37   printf("global_count: %d\n", global_count);
38   assert(global_count==(NUM_THREADS*LOOP_COUNT));
39   return 0;
40 }
41
42 void *thread(void * num) {
43   for (int i=0; i < LOOP_COUNT; i++)
44      global_count++;
45 }
46
```

Running Maple

```
Terminal
pinplay-VirtualBox:~/maple/example/shared_counter> ../../maple --profile_single_var_idioms --- ./main
2
global_count: 200
[MAPLE] === profile iteration 1 done === (1.412212) (/home/pinplay/maple/example/shared_counter)
global_count: 200
[MAPLE] === profile iteration 2 done === (1.609778) (/home/pinplay/maple/example/shared_counter)
global_count: 200
[MAPLE] === profile iteration 3 done === (1.409103) (/home/pinplay/maple/example/shared_counter)
global_count: 200
[MAPLE] === profile iteration 4 done === (1.509463) (/home/pinplay/maple/example/shared_counter)
global_count: 200
[MAPLE] === profile iteration 5 done === (1.713700) (/home/pinplay/maple/example/shared_counter)
[MAPLE] profile threshold reached
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
[MAPLE] === active iteration 1 done === (1.109534) (/home/pinplay/maple/example/shared_counter)
[MAPLE] active fatal error detected
[MAPLE]
[MAPLE] --------------------------
[MAPLE] profile_runs    5
[MAPLE] profile_time    7.854002
[MAPLE] active_runs     1
[MAPLE] active_time     1.125862
pinplay-VirtualBox:~/maple/example/shared_counter>
```

Profiling and building iroots from predefined idioms

Actively scheduling threads to reproduced the iroots found

Visualizing the IRoot database

```
pinplay-VirtualBox:~/maple/example/shared_counter> ../../script/idiom display iroot_db
1     IDIOM_1
        e0: WRITE   [1206  ld          0xa8cf                  ]
        e1: READ    [1206  ld          0xa8cf                  ]
2     IDIOM_1
        e0: WRITE   [79    ld          0x4c9f                  ]
        e1: READ    [1212  ld          0x9f7e                  ]
3     IDIOM_1
        e0: WRITE   [1365  ld          0x100c1                 ]
        e1: READ    [1214  ld          0x9fa4                  ]
4     IDIOM_1
        e0: WRITE   [70    ld          0xb972                  ]
        e1: READ    [1216  ld          0x9fbe                  ]
5     IDIOM_1
        e0: WRITE   [8     ld          0x4b0f                  ]
        e1: READ    [1217  ld          0x9fcc                  ]
6     IDIOM_1
        e0: WRITE   [8     ld          0x4b0f                  ]
        e1: READ    [1219  ld          0x9fd4                  ]
7     IDIOM_1
        e0: WRITE   [75    ld          0xb995                  ]
        e1: READ    [1221  ld          0x9fe1                  ]
8     IDIOM_1
        e0: WRITE   [73    ld          0xb986                  ]
        e1: READ    [1222  ld          0x9ff4                  ]
9     IDIOM_1
        e0: WRITE   [76    ld          0xb99c                  ]
        e1: READ    [1224  ld          0x9ffe                  ]
10    IDIOM_1
        e0: WRITE   [77    ld          0xb9ad                  ]
        e1: READ    [1226  ld          0xa794                  ]
```

```
28    IDIOM_2
        e0: WRITE   [1206  ld          0xa8cf                  ]
        e1: WRITE   [1206  ld          0xa8cf                  ]
        e2: READ    [1206  ld          0xa8cf                  ]
29    IDIOM_1
        e0: READ    [1754  main        0x859    main.cc +44    ]
        e1: WRITE   [1755  main        0x862    main.cc +44    ]
30    IDIOM_1
        e0: WRITE   [1755  main        0x862    main.cc +44    ]
        e1: WRITE   [1755  main        0x862    main.cc +44    ]
```

Reproducing the Bug with Maple

```
pinplay-VirtualBox:~/maple/example/shared_counter> ../../script/idiom display test_history
29    IDIOM_1 Success 1433996132
```

Last iroot tested, ID=29
and random seed=1433996132

```
pinplay-VirtualBox:~/maple/example/shared_counter> ../../script/idiom active --target_iroot=29
  --random_seed=1433996132 --- ./main 2
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
[MAPLE] === active iteration 1 done === (1.113106) (/home/pinplay/maple/example/shared_counter)
[MAPLE] active fatal error detected
```

Bug reproduced by
Maple by repeating same
interleaving

Capturing the Bug With PinPlay

```
pinplay-VirtualBox:~/maple/example/shared_counter> ../../script/idiom active --target_iroot=29 --rando
m_seed=1433996132 --pinplay --pinplay_options="-log:basename,failing.pinball/log" --- ./main 2
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
[MAPLE] === active iteration 1 done === (6.938914) (/home/pinplay/maple/example/shared_counter)
[MAPLE] active fatal error detected
```

Execution captured in a log file

```
pinplay-VirtualBox:~/maple/example/shared_counter> replay failing.pinball/log
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
pinplay-VirtualBox:~/maple/example/shared_counter> replay failing.pinball/log
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
pinplay-VirtualBox:~/maple/example/shared_counter> replay failing.pinball/log
global_count: 100
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
pinplay-VirtualBox:~/maple/example/shared_counter>
```

Execution replayed deterministically

Replaying within GDB

```
pinplay-VirtualBox:~/maple/example/shared_counter> gdb_replay failing.pinball/log main
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...done.
_dl_start (arg=0x0) at rtld.c:362
362     rtld.c: No such file or directory.
(gdb) b main.cc:38
Breakpoint 1 at 0x40080f: file main.cc, line 38.
(gdb) c
Continuing.
global_count: 100

Breakpoint 1, main (argc=2, argv=0x7fffefa3e188) at main.cc:38
38        assert(global_count==(NUM_THREADS*LOOP_COUNT));
(gdb) print global_count
$1 = 100
(gdb) c
Continuing.
main: main.cc:38: int main(int, char**): Assertion `global_count==(NUM_THREADS*LOOP_COUNT)' failed.
[Inferior 1 (Remote target) exited normally]
(gdb)
```

Using GDB + PinPlay to Debug

- PinPlay interfaces with PinADX and is enabled with a set of new GDB commands

```
(gdb) monitor help
general                 - General commands.
breakpoints             - Breakpoint commands.
tracepoints             - Tracepoint commands.
registers               - Register names.
PinPlay                 - PinPlay commands.

type "help <category>" for help on commands  in that category.
```

- Among the new commands, PinPlay is enabled with tracepoints, allowing the capture of a trace during execution (triggered by PC value, load or store value, etc)
- In this tutorial, we will use tracepoints to inspect the value of the global_counter and how it is updated by various threads

Modifying Tracepoint Implementation

- Tracepoint implementation can be found at: */home/pinplay/PinPlay/pinplay-drdebug-2.2-pldi2015-pin-2.14-71313-gcc.4.4.7-linux/extras/pinplay/examples/pinplay-debugger-shell.cpp*
- We will use the trace point to capture memory addresses executed at a given PC (the read and the write to global_count)

```
trace <addr> <length> at <pc>
                            Record trace entry at <pc> from memory value from
                            'addr'  of <length> bytes.
```

- However, the default implementation does not output the thread id for each trace, so we modify it to do so

```
struct TRACEREC
{
    unsigned _id;        // Index of EVENT in '_events'.
    ADDRINT _pc;         // PC where tracepoint triggered.
    ADDRINT  reg_mem_Value; // If tracepoints traces a register, it's value.
    THREADID _tid;       // Thread ID added for PLDI 2015 tutorial
};
```

*** Analysis functions to collect tracepoints are also changed to take the thread ID**

Creating a Tracepoint

```
(gdb) b main
Breakpoint 1 at 0x400745: file main.cc, line 30.
(gdb) c
Continuing.

Breakpoint 1, main (argc=2, argv=0x7fff3e67a2e8) at main.cc:30
30          NUM_THREADS = atoi(argv[1]);
(gdb) pin trace global_count at 44
monitor trace 0x60106c 4 at 0x400859 #global_count:<44>
Tracepoint #1:  trace memory address 0x00060106c length 4
 at 0x400859 #global_count:<44>
(gdb) b 38
Breakpoint 2 at 0x40080f: file main.cc, line 38.
(gdb) c
Continuing.
global_count: 100

Breakpoint 2, main (argc=2, argv=0x7fff3e67a2e8) at main.cc:38
38          assert(global_count==(NUM_THREADS*LOOP_COUNT));
(gdb) pin trace print to gcount.txt
monitor trace print to gcount.txt
```

- File gcount.txt contains the traces collected at runtime
- Now we need to inspect it!

Inspecting the Tracepoint File

```
2 0x0000000000400859: 0x00060106c = 0x0
3 0x0000000000400859: 0x00060106c = 0x0
3 0x0000000000400868: 0x00060106c = 0x1
3 0x0000000000400859: 0x00060106c = 0x1
3 0x0000000000400868: 0x00060106c = 0x2
3 0x0000000000400859: 0x00060106c = 0x2
3 0x0000000000400868: 0x00060106c = 0x3
3 0x0000000000400859: 0x00060106c = 0x3
3 0x0000000000400868: 0x00060106c = 0x4
3 0x0000000000400859: 0x00060106c = 0x4
3 0x0000000000400868: 0x00060106c = 0x5
3 0x0000000000400859: 0x00060106c = 0x5
3 0x0000000000400868: 0x00060106c = 0x6
3 0x0000000000400859: 0x00060106c = 0x6
3 0x0000000000400868: 0x00060106c = 0x7
3 0x0000000000400859: 0x00060106c = 0x7
3 0x0000000000400868: 0x00060106c = 0x8
```

Thread 2 reads initial counter value, context switches out

Thread 3 also read initial value before Thread 2 updates it

Thread 3 then executes all its loop iterations

```
3 0x0000000000400868: 0x00060106c = 0x61
3 0x0000000000400859: 0x00060106c = 0x61
3 0x0000000000400868: 0x00060106c = 0x62
3 0x0000000000400859: 0x00060106c = 0x62
3 0x0000000000400868: 0x00060106c = 0x63
3 0x0000000000400859: 0x00060106c = 0x63
3 0x0000000000400868: 0x00060106c = 0x64
2 0x0000000000400868: 0x00060106c = 0x1
2 0x0000000000400859: 0x00060106c = 0x1
2 0x0000000000400868: 0x00060106c = 0x2
2 0x0000000000400859: 0x00060106c = 0x2
2 0x0000000000400868: 0x00060106c = 0x3
2 0x0000000000400859: 0x00060106c = 0x3
```

global_count incremented up to 100, thread 3 finishes

Thread 2 context switches back in, overwrites the last update by Thread 3 and then increments count to 100

```
2 0x0000000000400868: 0x00060106c = 0x62
2 0x0000000000400859: 0x00060106c = 0x62
2 0x0000000000400868: 0x00060106c = 0x63
2 0x0000000000400859: 0x00060106c = 0x63
2 0x0000000000400868: 0x00060106c = 0x64
```

Interactive analysis of a particular run is a lot easier with deterministic replay because addresses and thread interleaving never changes

Summary

- Deterministic replay is a great tool for root causing concurrency errors
- However capturing a buggy execution may take a long time depending on the bug
- Active testing via Maple addresses the bug triggering issue and, combined with PinPlay, provides a framework to expose and analyze concurrency errors

PinPlay: A framework for Deterministic Replay and Reproducible Analysis

*PinPlay is an **easy-to-use**, **flexible**, and **effective** framework for reproducible analysis of multi-threaded programs.*

- ***easy-to-use** : the Pin advantage (download and use, no re-compile…)*
- ***Flexible:** combine with any Pintool, combine with debuggers*
- ***Effective**: it works (deterministic replay → reproducible analysis)*

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel