Using PinPlay for Reproducible Analysis and Replay Debugging

Harish Patil, Charles(Chuck) Yount, Intel Corporation

Vineet Singh, University of California, Riverside

With contributions from:

Ady Tal, Ariel Slonim, Michael Gorin, Tevi Devor (Intel Corporation)

PLDI 2016 Tutorial, Santa Barbara, June 14th , 2016

PinPlay : History & Acknowledgements

Trigger : 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 2006)

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

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

Tutorial Objective

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

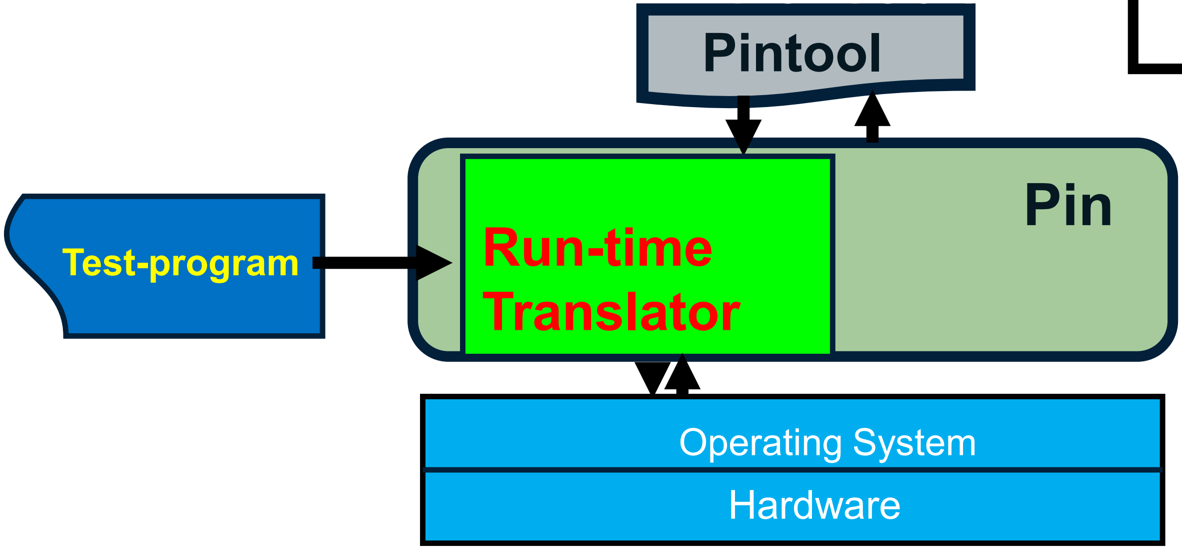
Goal: Complement earlier tutorial (PLDI2015)

1. Cross-OS replay + analysis
2. Tool-chain for loop detection, tracking, dependence anlysis

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

Agenda

PinPlay basics & internals [till 9:30]

Intel SDE [9:30—10]

<10 – 10:30 Break>

DCFG (Chuck) [10:30—10:50]

Dynamic Slicing (Vineet) [10:50—11:10]

Example tool-chain : Replay + DCFG + Slicing
[11:10 – 11:30]

PinADX [till 11:30 -11:40]

DrDebug [11:40 – noon]

PinPlay basics

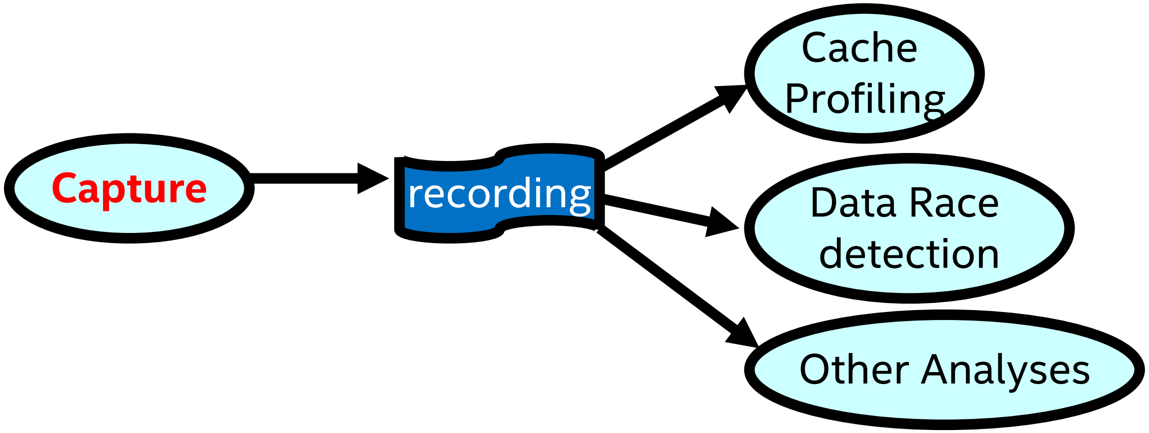
The Need : Easier Analysis of Parallel Programs

Programmers need a way to deterministically analyze and debug parallel programs

Why?

Run-to-run variation ➔ Chasing a moving target

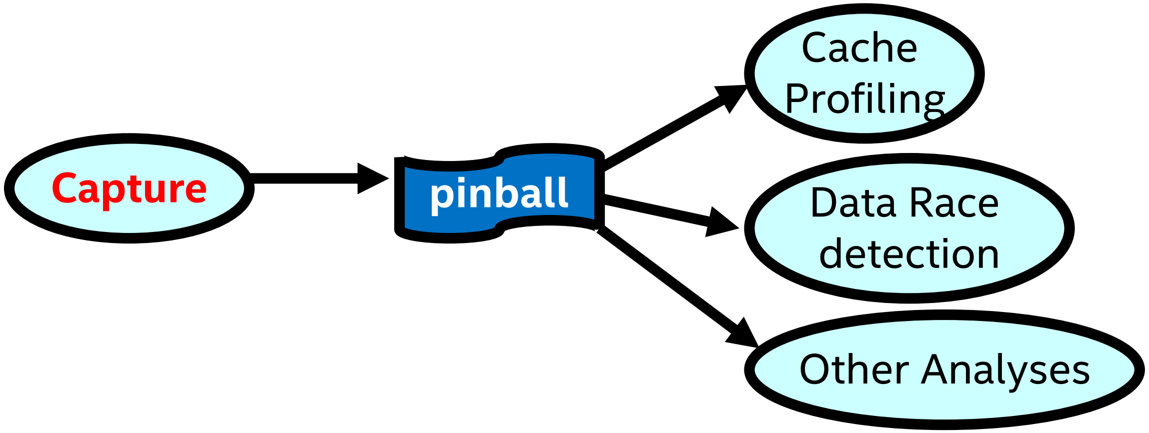
The Solution :
Record → Replay & Analyze



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

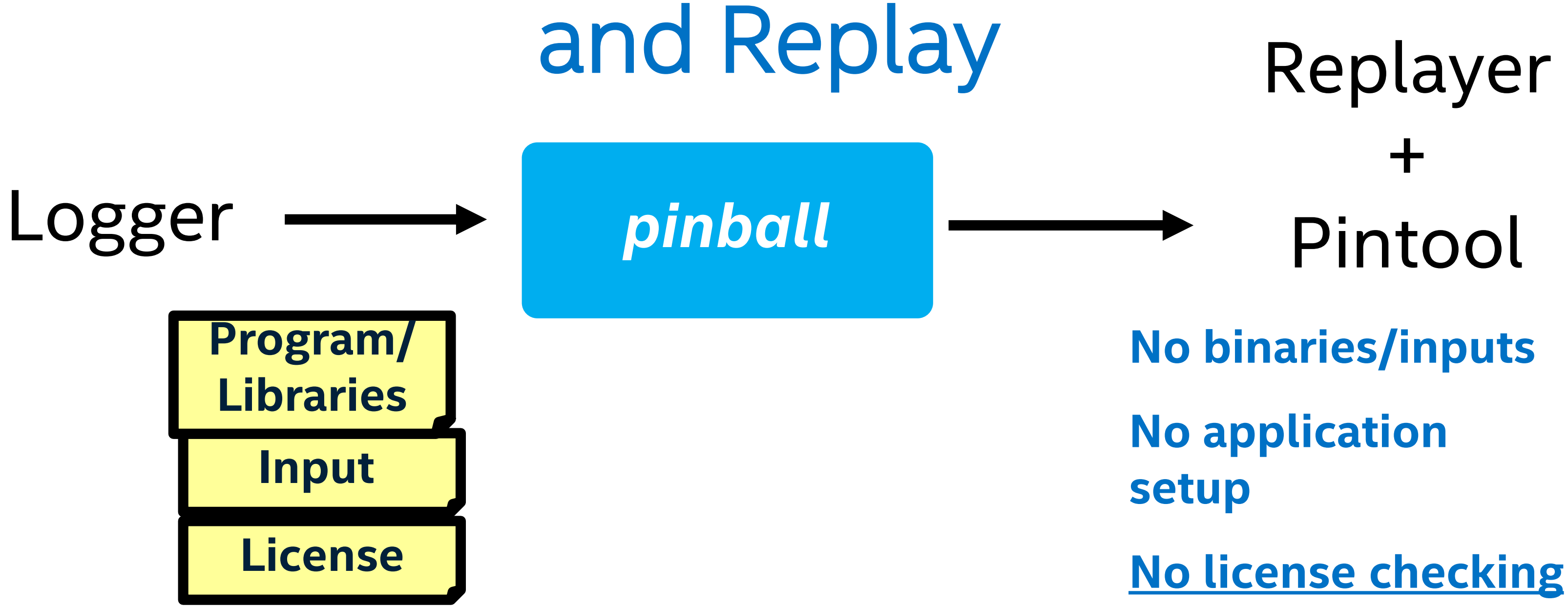


Kernel A → Kernel B

Windows → Linux

Customer site → Developer site

PinPlay : Software-based User-level Capture and Replay

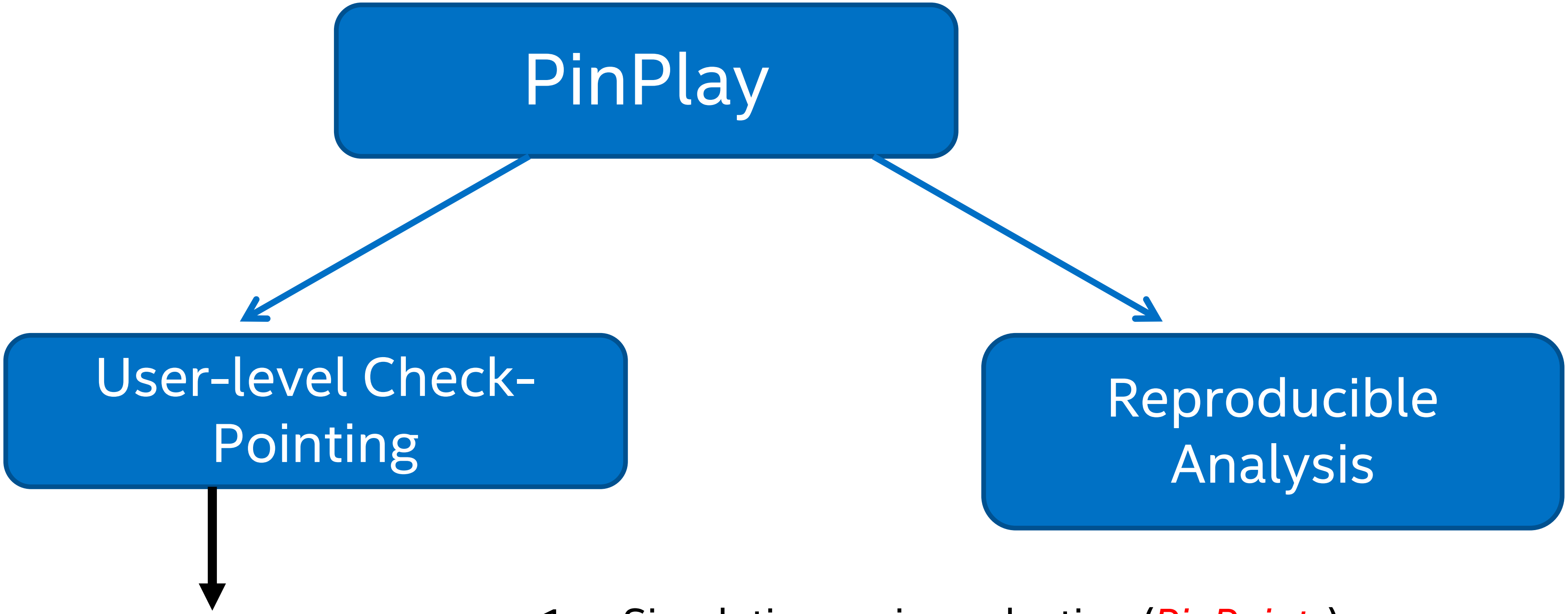


Platforms : Linux, Windows, Android, MacOS

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

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

PinPlay Applications at a glance



PinPlay

User-level Check-Pointing

Reproducible Analysis

Traces for architecture simulation

1. pinballs 2. LIT

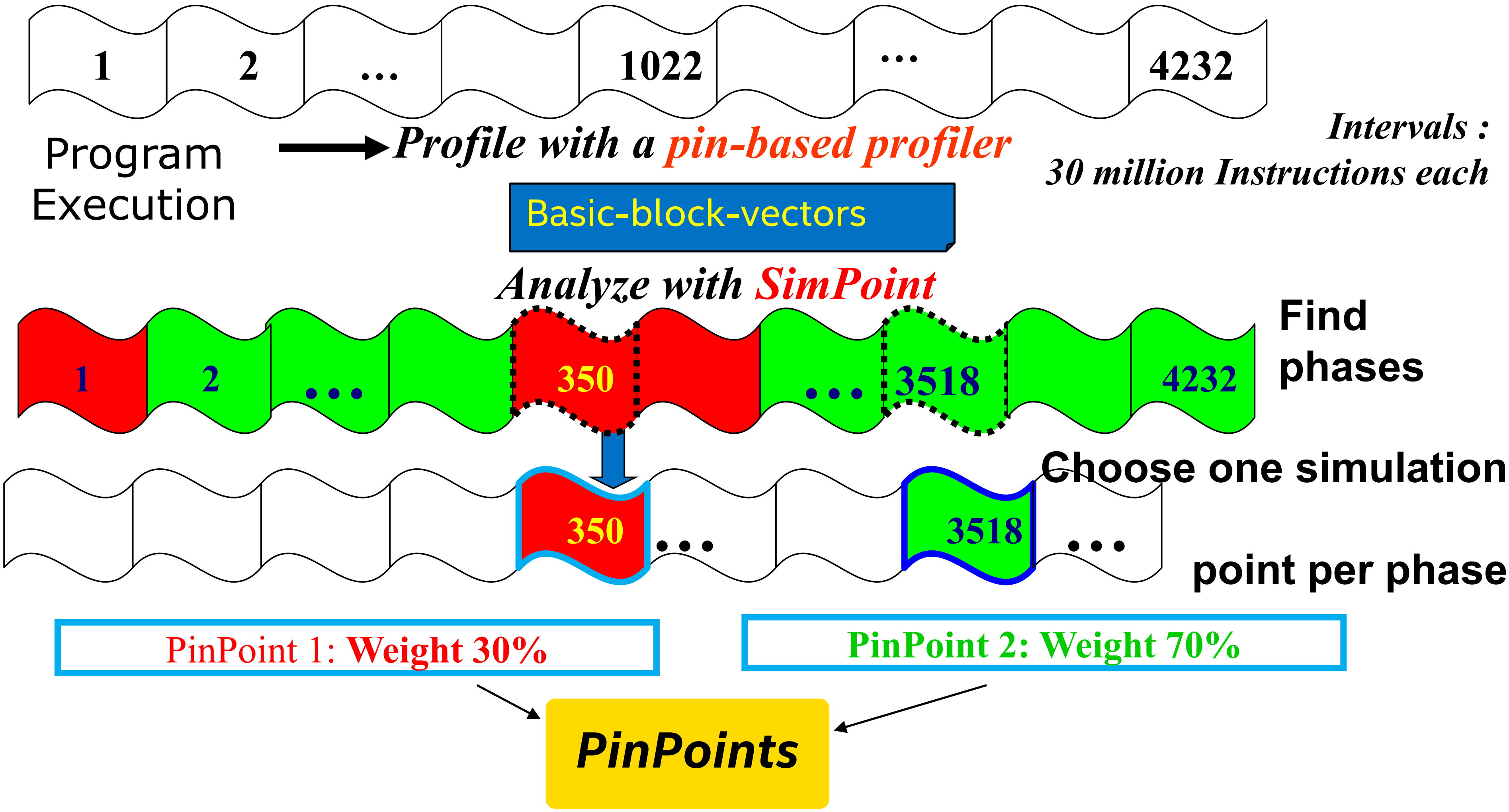
→ Intel simulators

→ Sniper (**U Ghent**)

www.snipersim.org

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

PinPoints = Pin + SimPoint



Program Execution

Profile with a pin-based profiler

Intervals : 30 million Instructions each

Basic-block-vectors

Analyze with SimPoint

Find phases

Choose one simulation point per phase

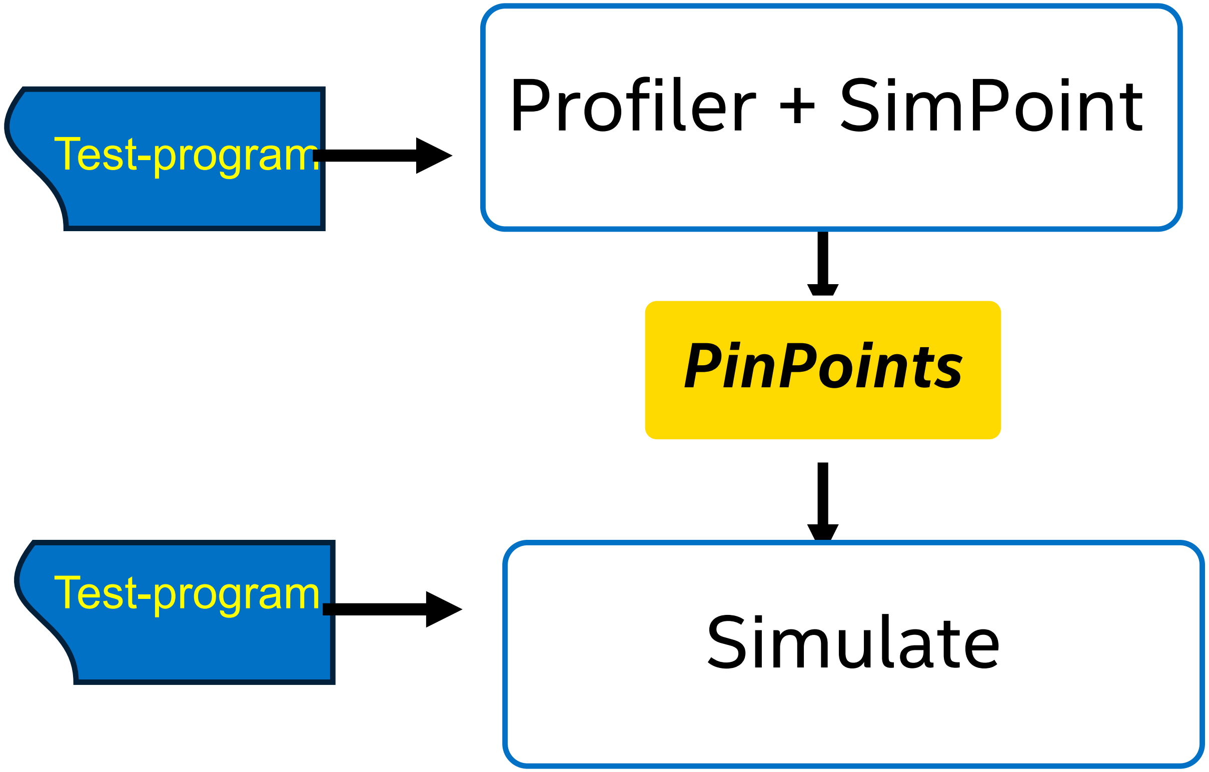
PinPoint 1: **Weight 30%**

PinPoint 2: **Weight 70%**

PinPoints

Two Phases => Two PinPoints

PinPoints : The repeatability challenge

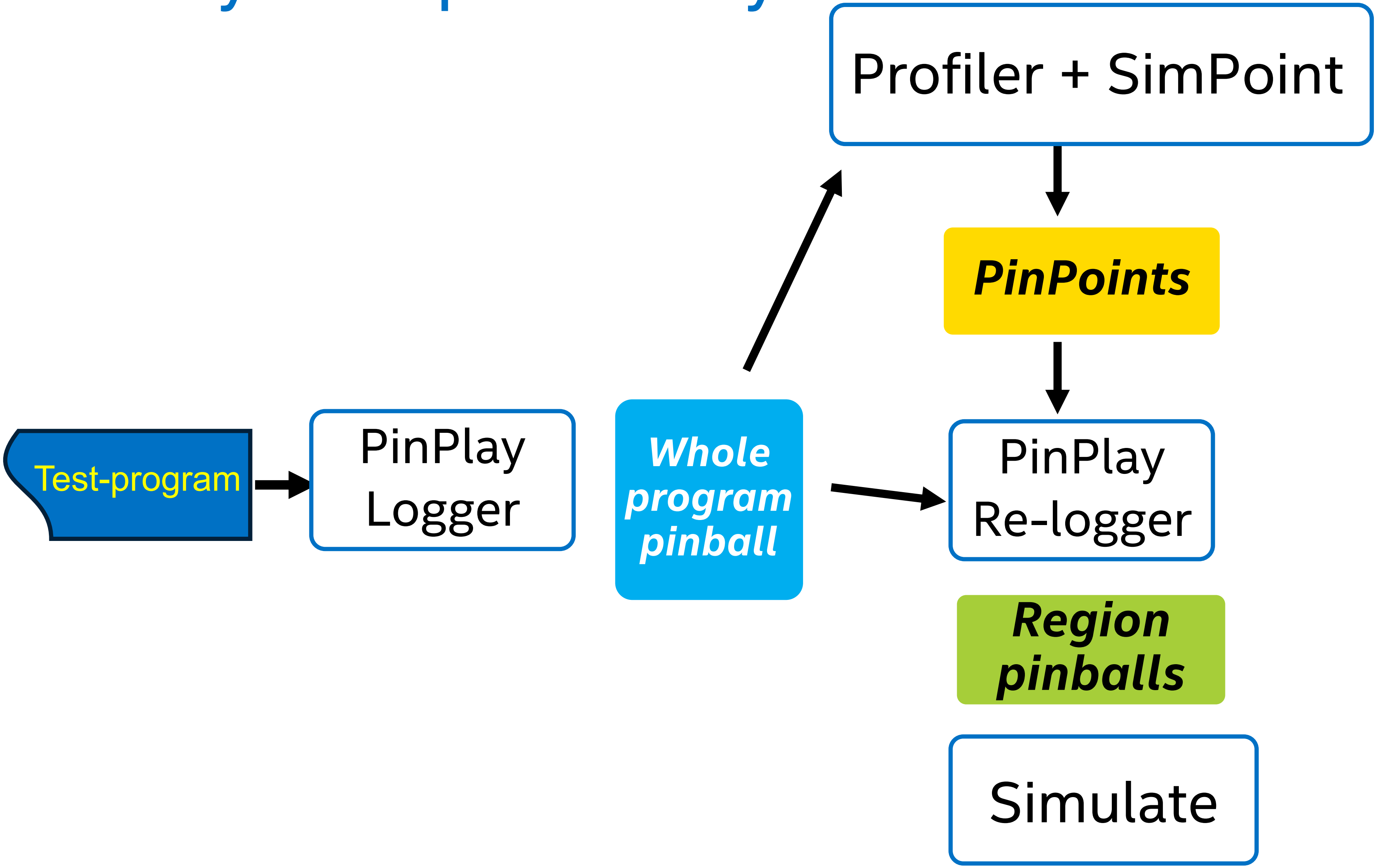


Problem: Two runs are not exactly same → PinPoints missed

Found this for 25/54 SPEC2006 runs!

[***"PinPoints out of order" "PinPoint End seen before Start"***]

PinPlay → Repeatability


Profiler + SimPoint

PinPoints

Test-program

PinPlay Logger

Whole program pinball

PinPlay Re-logger

Region pinballs

Simulate

Pinballs: Portable, OS independent, provide determinism

Why use PinPlay?

- No special OS, hardware support, source changes, re-linking required

  If you can run it, you can Pin it

  If you can Pin it, you can replay it

- Close integration with Pin

  Pintool + PinPlay ➔ Reproducible analysis

- Available on all major operating systems (Windows/MacOS/Linux)

  Record anywhere (once) → Replay anywhere (multiple times)

- Completely faithful replay (all thread inter-leaving reproduced)

- Developed and supported by Intel

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 (all threads/processes) 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 (per process/thread) replay

Dealing with slow-downs : 3 Mantras

1) "Attach"ment is good

```
% $PIN_ROOT/extras/pinplay/scripts record --pid PID
% $PIN_ROOT/pin -pid PID -t $PIN_ROOT/…/pinplay-driver.so ..
% $SDEHOME -attach-pid PID
```

2) Be Selective

- ❑ Specify "region of interest" : See https://software.intel.com/en-us/articles/pintool-regions
- ❑ Focus on a specific 'thread of interest'

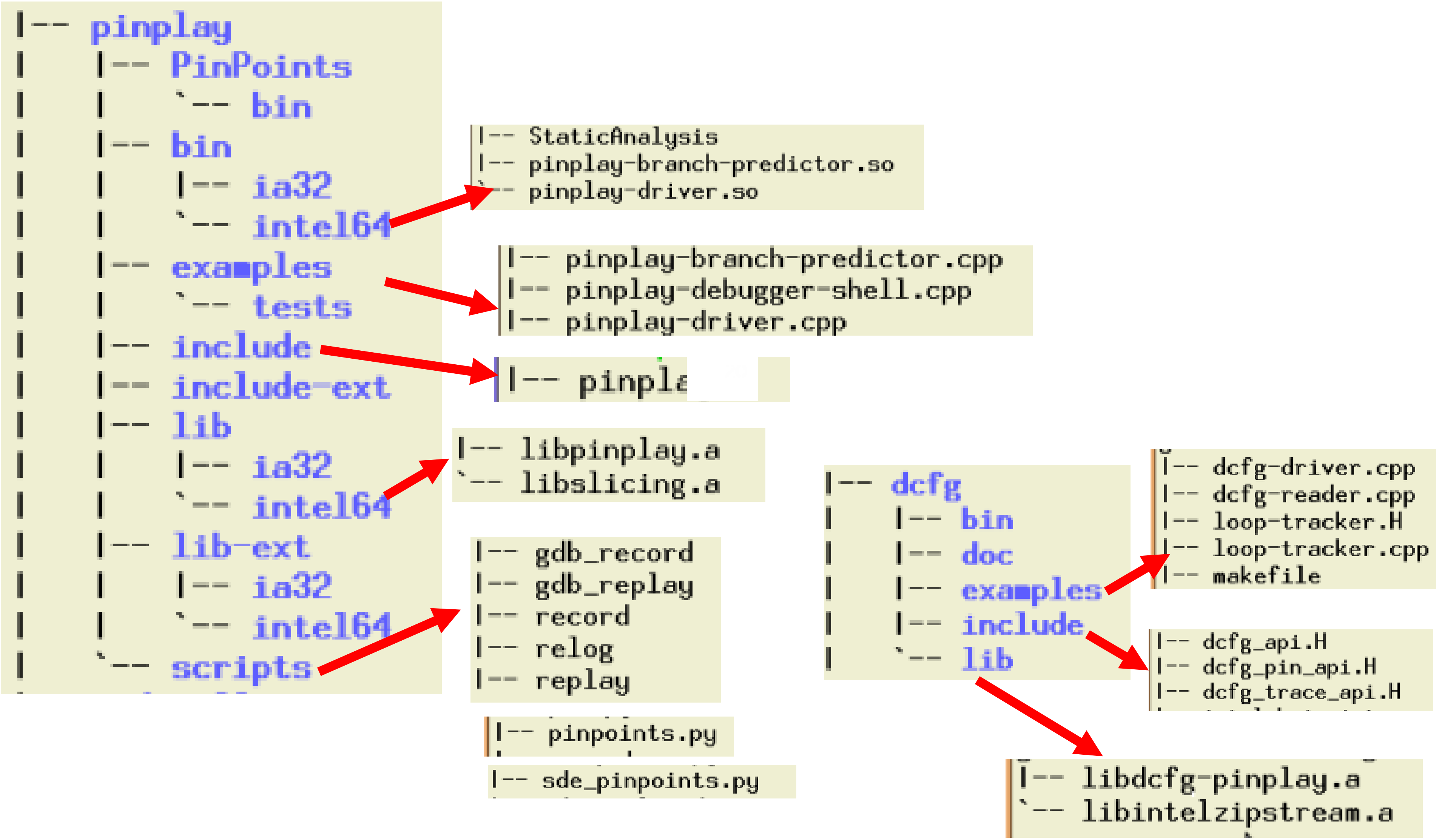
```
% record --pintool_options "-log:focus_thread TID"
```

3) Practice exclusion

```
 % record -pintool_options "-log:exclude_image IMG"
 % record -pintool_options "-log:exclude_code -
log:exclude:control REGION-SPECIFICATION"
```

The PinPlay kit

Download from http://www.pinplay.org



```
|-- pinplay
|    |-- PinPoints
|    |    `-- bin
|    |-- bin
|    |    |-- ia32
|    |    `-- intel64
|    |-- examples
|    |    `-- tests
|    |-- include
|    |-- include-ext
|    |-- lib
|    |    |-- ia32
|    |    `-- intel64
|    |-- lib-ext
|    |    |-- ia32
|    |    `-- intel64
|    `-- scripts
```

```
|-- StaticAnalysis
|-- pinplay-branch-predictor.so
`-- pinplay-driver.so
```

```
|-- pinplay-branch-predictor.cpp
|-- pinplay-debugger-shell.cpp
|-- pinplay-driver.cpp
```

```
|-- pinpla
```

```
|-- libpinplay.a
`-- libslicing.a
```

```
|-- gdb_record
|-- gdb_replay
|-- record
|-- relog
|-- replay
```

```
|-- pinpoints.py
```

```
|-- sde_pinpoints.py
```

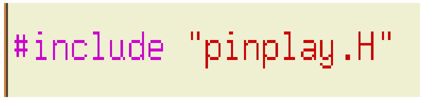
```
|-- dcfg
|    |-- bin
|    |-- doc
|    |-- examples
|    |-- include
|    `-- lib
```

```
|-- dcfg-driver.cpp
|-- dcfg-reader.cpp
|-- loop-tracker.H
|-- loop-tracker.cpp
|-- makefile
```

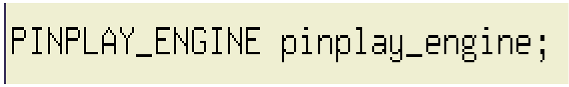
```
|-- dcfg_api.H
|-- dcfg_pin_api.H
|-- dcfg_trace_api.H
```

```
|-- libdcfg-pinplay.a
`-- libintelzipstream.a
```

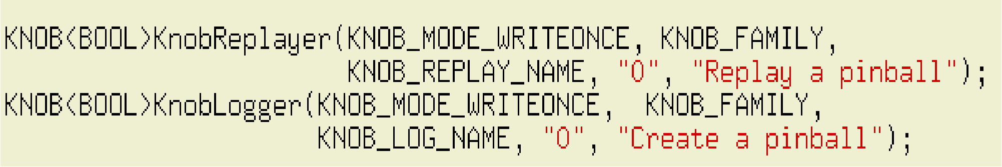
Enabling a Pintool for PinPlay



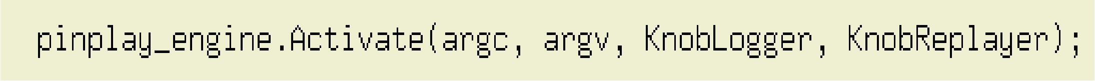
```
#include "pinplay.H"
```



```
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);
```

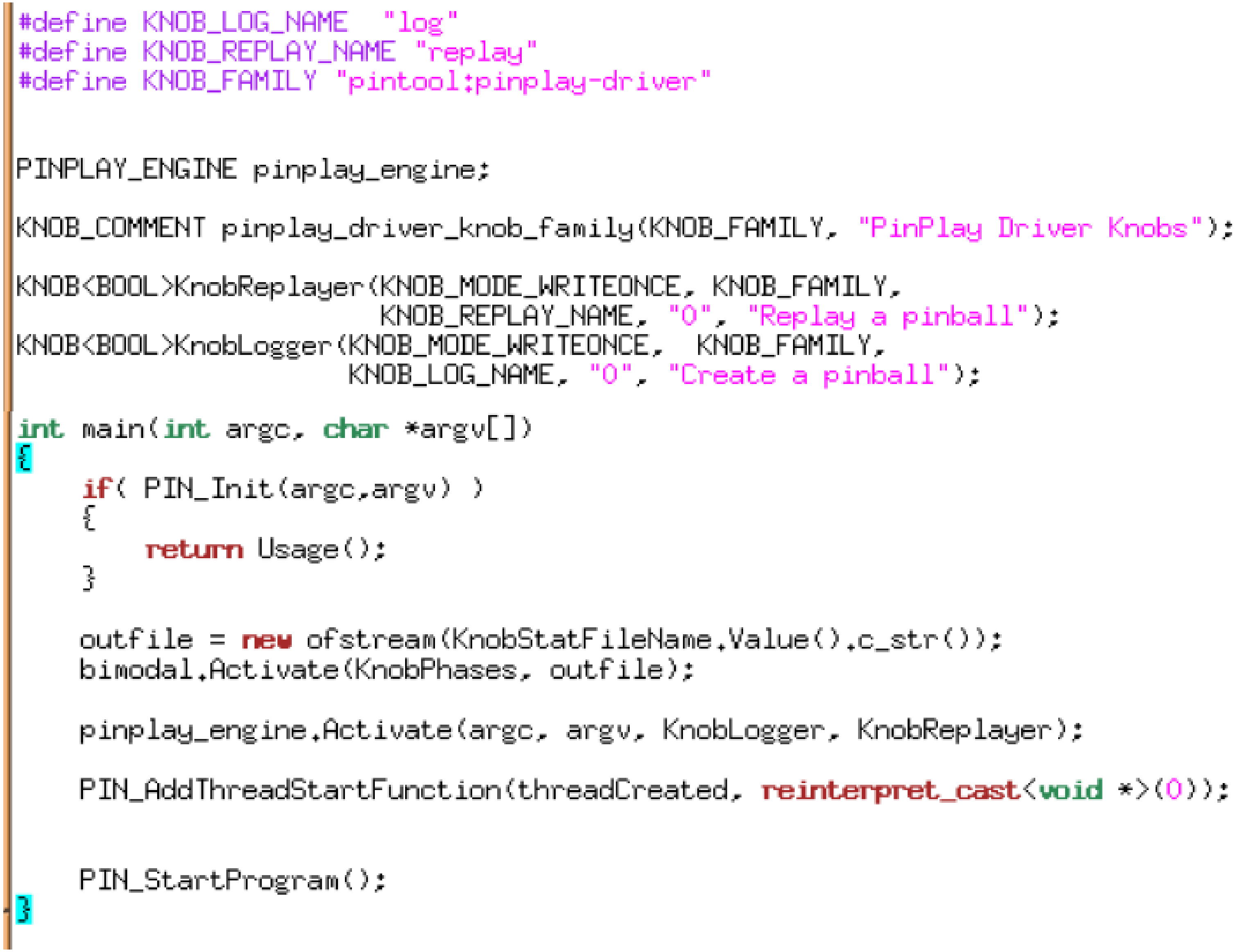
Makefile changes:

Include Path : ***$PIN_ROOT/extras/pinplay/include***

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**

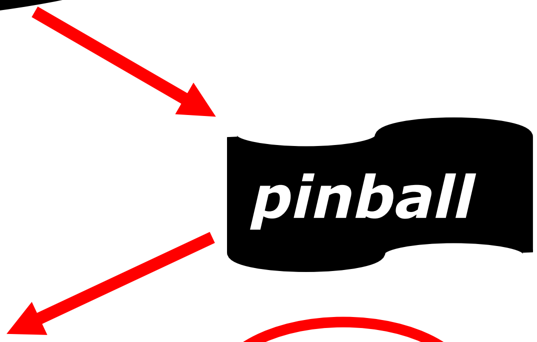
```
% pin -t pintool -- test-program
```

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**”

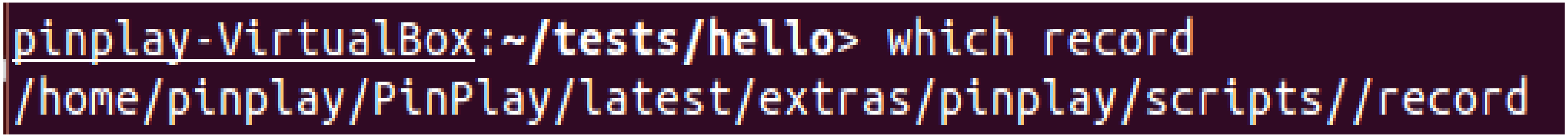
```
% pin -t pinplay-branch-predictor.so -log -
log:basename pinball/foo hello
```

Creates “**bimodal.out**” and “**pinball/foo***”

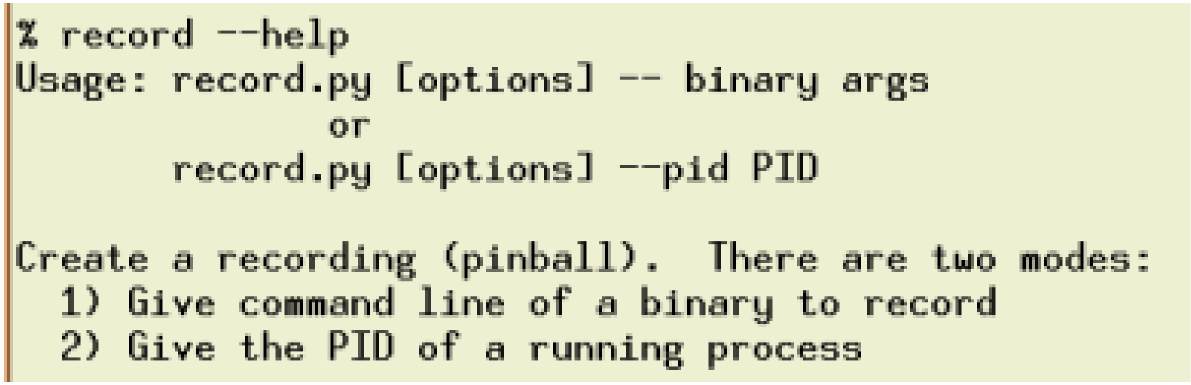
```
% pin -xyzzy -reserve_memory
pinball/foo.address -replay
    -replay:basename pinball/foo --
   $PIN_KIT/extras/pinplay/intel64/bin/nullapp
```

Creates “**bimodal.out**”

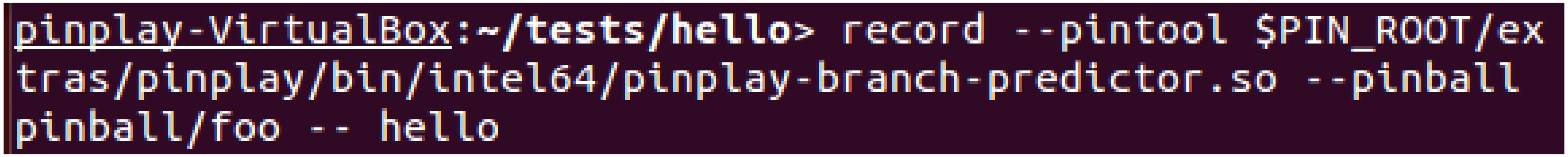
Using $PIN_ROOT/extras/pinplay/scripts: Recording (uses pinplay-driver.so)



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



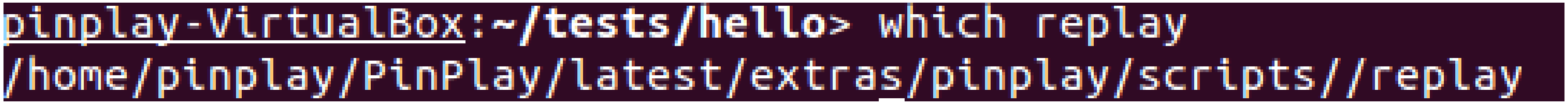
```
% record --help
Usage: record.py [options] -- binary args
              or
       record.py [options] --pid PID

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
```



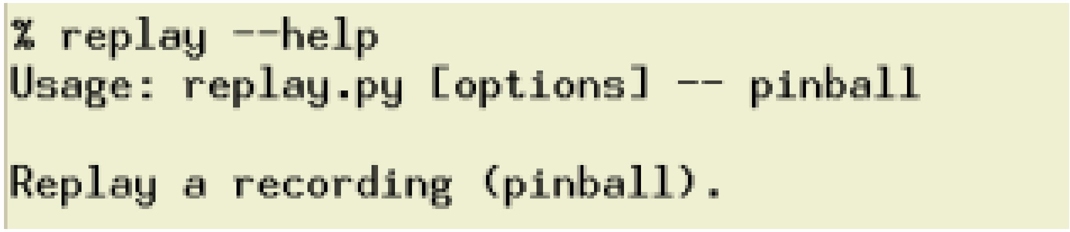
```
pinplay-VirtualBox:~/tests/hello> record --pintool $PIN_ROOT/ex
tras/pinplay/bin/intel64/pinplay-branch-predictor.so --pinball
pinball/foo -- hello
```

** Developed by Mack Stallcup*

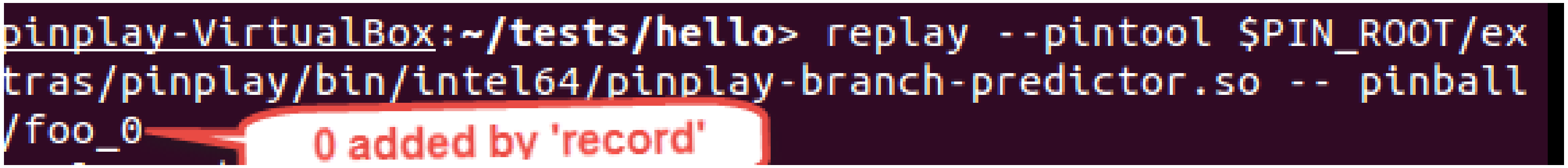
Using $PIN_ROOT/extras/pinplay/scripts: Replaying (uses pinplay-driver.so)



```
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'

Intel® Software Development Emulator: *SDE*

With contributions from Ady Tal, Ariel Slonim, Michael Gorin(Intel Corporation)
Original developer: **Mark Charney(**Intel Corporation)

What is SDE

The Intel® Software Development Emulator is **a functional user-level (ring 3) emulator** for x86 (32b and 64b) new instructions built upon Pin and XED (X86 encoder/decoder)

Goal: New instruction/register emulation between the time when they are designed and when the hardware is available.

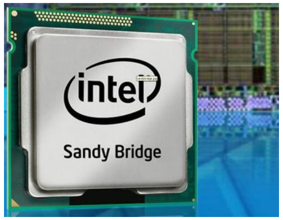
Used for compiler development, architecture and workload analysis, and tracing for architecture simulators

http://www.intel.com/software/sde

Currently Supported ISA Extensions (as of sde-external-7.45.0-2016-05-09)

Public: Everything up to CNL

- NHM
- WSM
- SNB
- IVB
- SKL
- SKX
- KNL
- CNL (default)



Available externally for : Linux, Windows, and OS X (MacOS)

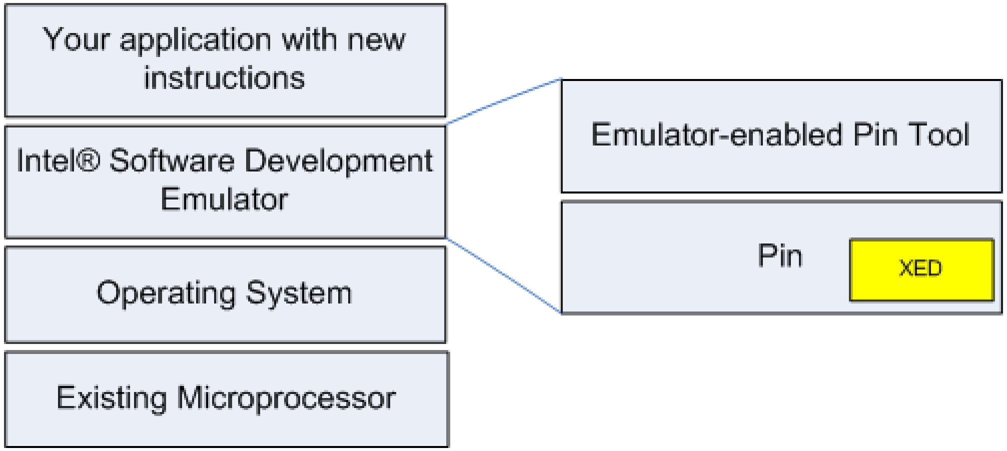
How Does SDE Works

Based on Pin/XED

XED decode/encode

Existing OS

Existing CPU

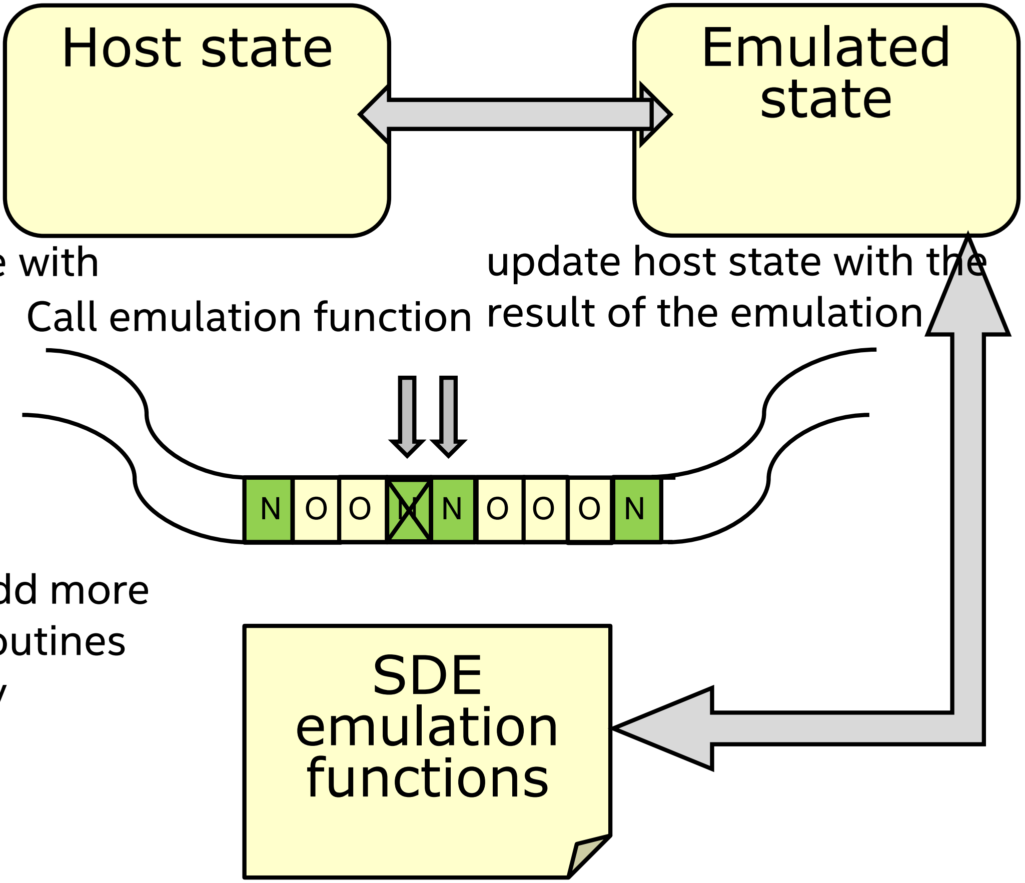

Your application with new instructions
Intel® Software Development Emulator
Operating System
Existing Microprocessor
Emulator-enabled Pin Tool
Pin
XED

No special compilation required

Supported on Windows/Linux/Mac OS/Android(internal)

Runs only in user space (ring 3)

Getting Deeper



update emulated state with data from host state

Call emulation function

update host state with the result of the emulation

Remove new instruction and go to next one

Additional tools add more instrumentation routines before or after any instructions

🟩 New instruction
Legacy instruction

Basic Emulation

For each instruction check:

- Is this a “new” instruction
  - Figure out what its operands and side effects are
  - Add a call to a marshal-and-emulate function
  - Skip the original instruction to avoid faulting

Scan each instruction approximately once

- Keep translated code in a software cache

Performance depends on amount of emulation

- Normally: 100mips – 2bips

How to Use SDE

SDEHOME →<path-to-kit>

General format:

```
% $SDEHOME/sde [options] -- userapp [app options]
```

```
% $SDEHOME/sde -attach-pid <pid>
```

- No application necessary
- Seems to quit right away
  - Output goes to process' work directory

Running the basic emulator

```
% $SDEHOME/sde -- foo.exe
```

Getting help:

```
% $SDEHOME/sde -help
```

http://www.intel.com/software/sde

SDE Emulation Based Features

SDE comes with emulation based features

- Mix histogram tool
- Debug-trace ASCII tracing tool
- Memory footprint tool
- Extended debugging (via **PinADX**)
- **PinPlay** support for generating/replaying pinballs
- Dynamic Control Flow Graph (**DCFG**) generation

Use –long-help to get detailed help on all features

Binary distribution : Does not allow writing your own Pin tools

Mix Histogram Tool

Calculates the application instruction mix statistics

Provides

- Static and dynamic instruction mix statistics
- Blocks with highest dynamic instruction count
  - Annotated with image/function/file/line
- Instruction count per function
- Histogram per instruction class and various categories
  - E.g. memory operation by size, scalar/vector
- Histogram per thread and total process summary
- DCFG feature : hot loop analysis : “-mix –mix_loops”

Mix Output

```
% $SDEHOME/sde -mix -mix_loops -- /bin/ls
```

```
# Mix output version 10
# Intel(R) SDE version: 7.45.0 external
....
# =========================================
# LOOPS_STATS_FROM_DCFG
# =========================================
LOOP: 0    NUM BLOCKS: 2   HEAD BLOCK_ID: 1638   ENTRIES: 1847   EXECUTIONS: 7431
EDGE FROM BLOCK_ID: 1638 TO BLOCK_ID: 1639   EXECUTIONS: 5923
EDGE FROM BLOCK_ID: 1639 TO BLOCK_ID: 1638   EXECUTIONS: 5584

BLOCK_ID:  1638   PC: 7ffff7df36a0   ICOUNT:    22293   EXECUTIONS:     7431   #BYTES:  6  FN:
strcmp  IMG: /lib64/ld-linux-x86-64.so.2
XDIS 00007ffff7df36a0: BASE 8A07              mov al, byte ptr [rdi]
XDIS 00007ffff7df36a2: BASE 3A06              cmp al, byte ptr [rsi]
XDIS 00007ffff7df36a4: BASE 750D              jnz 0x7ffff7df36b3
```

SDE + PinPlay

Recording:

```
% $SDEHOME/sde  -log –log:basename <dir>/<name>--
userapp [app options]
```

Replaying:

```
% $SDEHOME/sde  -replay –replay:addr_trans –
replay:basename <dir>/<name> -- $SDEHOME/intel64/nullapp
```

- `-replay:addr_trans` : Address translation (see next foil)

Can use SDE tools, e.g. mix with replay (sde –mix –replay …**)**

Scripts for Linux SDE :

```
% cp -r <PinPlay-kit>/extras/pinplay/scripts
$SDEHOME/pinplay-scripts
```

```
% $SDEHOME/pinplay-scripts/record userapp [app options]
```

```
% $SDEHOME/pinplay-scripts/replay <pinball dir/name>
```

Cross-OS Replay

- Memory ranges required may not be available on target OS
  Solution: use "-replay:addr_trans"
  - Relocates text/data at pinball load time
  - Builds a translation table
  - Text/data addresses re-written to use translation
- System calls
  - Most system calls are skipped hence not an issue
  - Semantics of certain system calls, e.g. thread creation, exit, are recorded in pinball
    - Replayer uses equivalent system calls on the target OS

Recording with SDE (any OS) → Replaying with PinPlay kit (Linux)

- Default emulated ISA is Cannonlake (CNL) : SDE 7.45 May 2016
- CNL pinballs use ZMM registers not supported in any currently available x86 machine
- Record for an existing/shipping ISA e.g. Ivy Bridge
  - Will work only if the application does not use newer, CNL, instructions

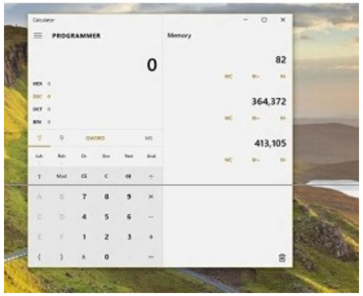
Recording with SDE (any OS):

```
% $SDEHOME/sde  -ivb -log -log:basename
<dir>/<name>-- userapp [app options]
```

Replay with PinPlay kit (Linux Ivy Bridge machine):

```
% $PIN_ROOT/extras/scripts/replay --cross_os
                                  < dir/name>
```

Example: Record “Calculator” on Windows 10



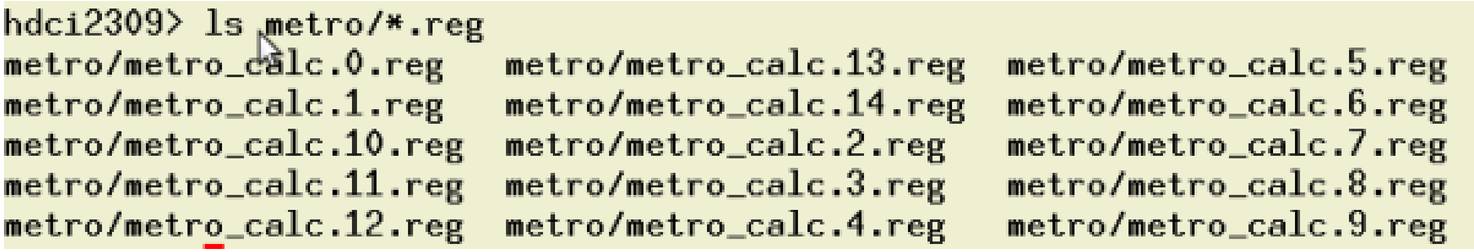
```
% $SDEHOME/sde  -ivb -log -log:basename <full-
path>metro/metro_calc  -attach-pid 9932  -log:mt -
log:early_out -length 100000000 -pinplay:max_threads
100
```

1) “Attach”ment is good

- Found out the PID for already running calculator process
- No “ `--app [args]` “ given
- Full path to pinball given (else will be written in Calculator’s working directory)
- The command prompt will return right away, but SDE attached to Calculator will be running
- **`-length 100000000`** : 100 million instructions in the active thread (at the time of attach)
  - Note: no “-log” prefix (unlike in PinPlay kit commands)
- **`-log:early_out`**: exit after tracing 100 million instructions

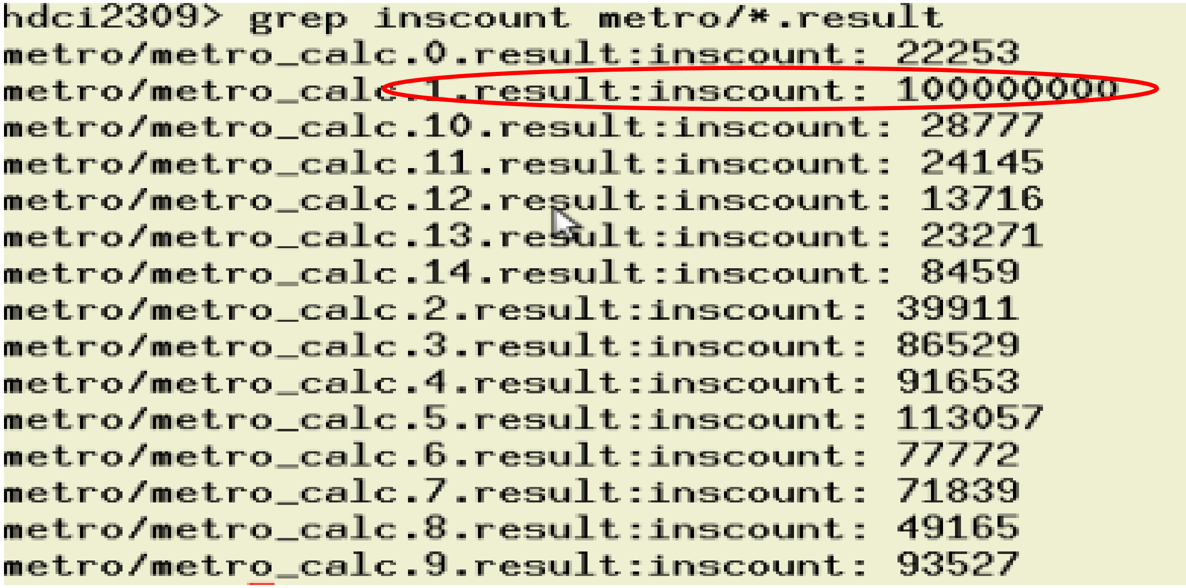
Copy ‘metro’ pinball to Linux (Ivy Bridge)

- 15 threads captured



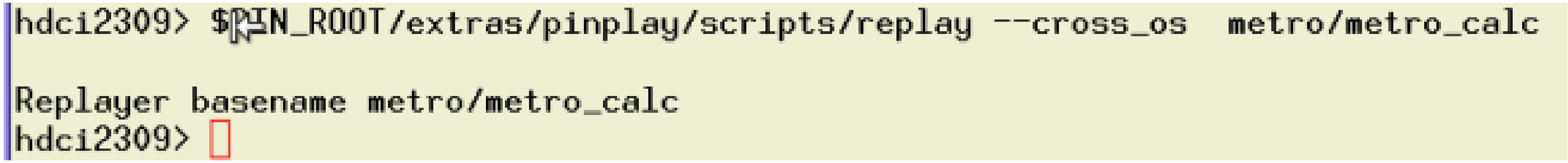
```
hdci2309> ls metro/*.reg
metro/metro_calc.0.reg    metro/metro_calc.13.reg  metro/metro_calc.5.reg
metro/metro_calc.1.reg    metro/metro_calc.14.reg  metro/metro_calc.6.reg
metro/metro_calc.10.reg   metro/metro_calc.2.reg   metro/metro_calc.7.reg
metro/metro_calc.11.reg   metro/metro_calc.3.reg   metro/metro_calc.8.reg
metro/metro_calc.12.reg   metro/metro_calc.4.reg   metro/metro_calc.9.reg
```

- Tid 1 was the active thread at attach time



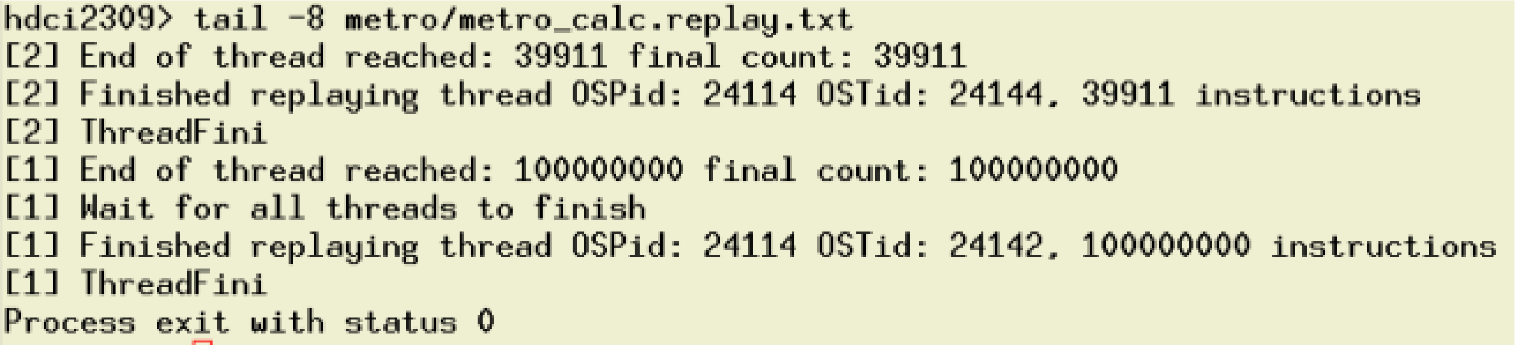
```
hdci2309> grep inscount metro/*.result
metro/metro_calc.0.result:inscount: 22253
metro/metro_calc.1.result:inscount: 100000000
metro/metro_calc.10.result:inscount: 28777
metro/metro_calc.11.result:inscount: 24145
metro/metro_calc.12.result:inscount: 13716
metro/metro_calc.13.result:inscount: 23271
metro/metro_calc.14.result:inscount: 8459
metro/metro_calc.2.result:inscount: 39911
metro/metro_calc.3.result:inscount: 86529
metro/metro_calc.4.result:inscount: 91653
metro/metro_calc.5.result:inscount: 113057
metro/metro_calc.6.result:inscount: 77772
metro/metro_calc.7.result:inscount: 71839
metro/metro_calc.8.result:inscount: 49165
metro/metro_calc.9.result:inscount: 93527
```

Cross-OS Replay on Linux (Ivy Bridge) with PinPlay kit

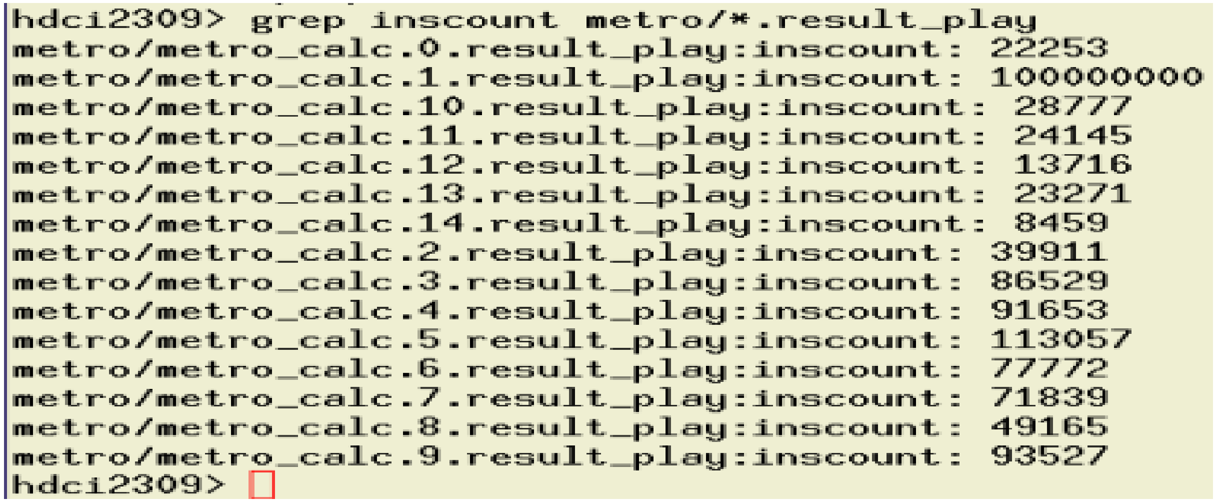


```
hdci2309> $PIN_ROOT/extras/pinplay/scripts/replay --cross_os  metro/metro_calc

Replayer basename metro/metro_calc
hdci2309> 
```

Replay successful!

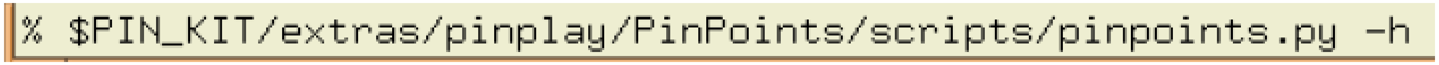


```
hdci2309> tail -8 metro/metro_calc.replay.txt
[2] End of thread reached: 39911 final count: 39911
[2] Finished replaying thread OSPid: 24114 OSTid: 24144, 39911 instructions
[2] ThreadFini
[1] End of thread reached: 100000000 final count: 100000000
[1] Wait for all threads to finish
[1] Finished replaying thread OSPid: 24114 OSTid: 24142, 100000000 instructions
[1] ThreadFini
Process exit with status 0
```

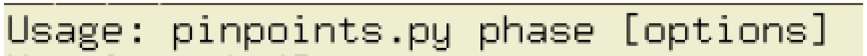


```
hdci2309> grep inscount metro/*.result_play
metro/metro_calc.0.result_play:inscount: 22253
metro/metro_calc.1.result_play:inscount: 100000000
metro/metro_calc.10.result_play:inscount: 28777
metro/metro_calc.11.result_play:inscount: 24145
metro/metro_calc.12.result_play:inscount: 13716
metro/metro_calc.13.result_play:inscount: 23271
metro/metro_calc.14.result_play:inscount: 8459
metro/metro_calc.2.result_play:inscount: 39911
metro/metro_calc.3.result_play:inscount: 86529
metro/metro_calc.4.result_play:inscount: 91653
metro/metro_calc.5.result_play:inscount: 113057
metro/metro_calc.6.result_play:inscount: 77772
metro/metro_calc.7.result_play:inscount: 71839
metro/metro_calc.8.result_play:inscount: 49165
metro/metro_calc.9.result_play:inscount: 93527
hdci2309> 
```

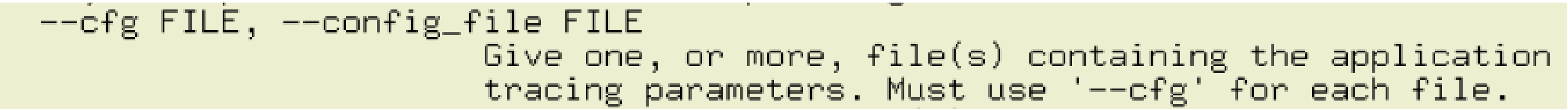
Example: Cross-OS PinPoints



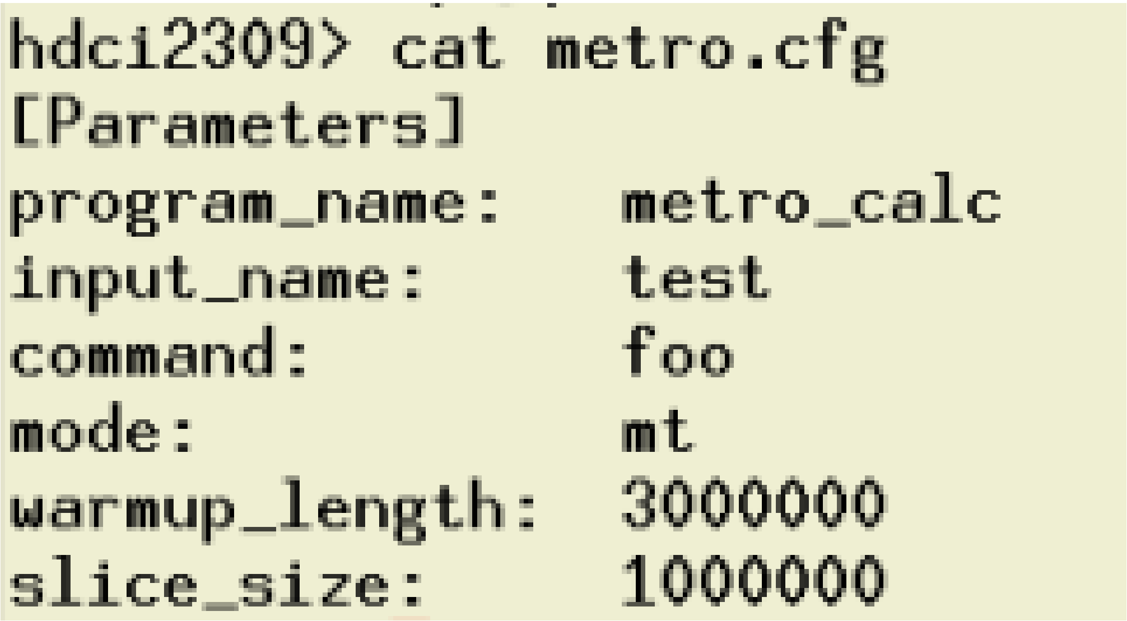
```
% $PIN_KIT/extras/pinplay/PinPoints/scripts/pinpoints.py -h
```



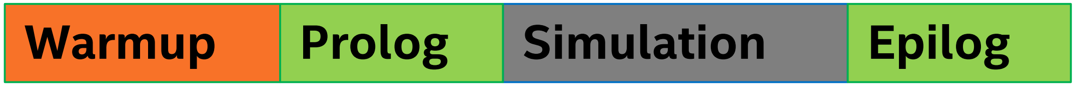
```
Usage: pinpoints.py phase [options]
```



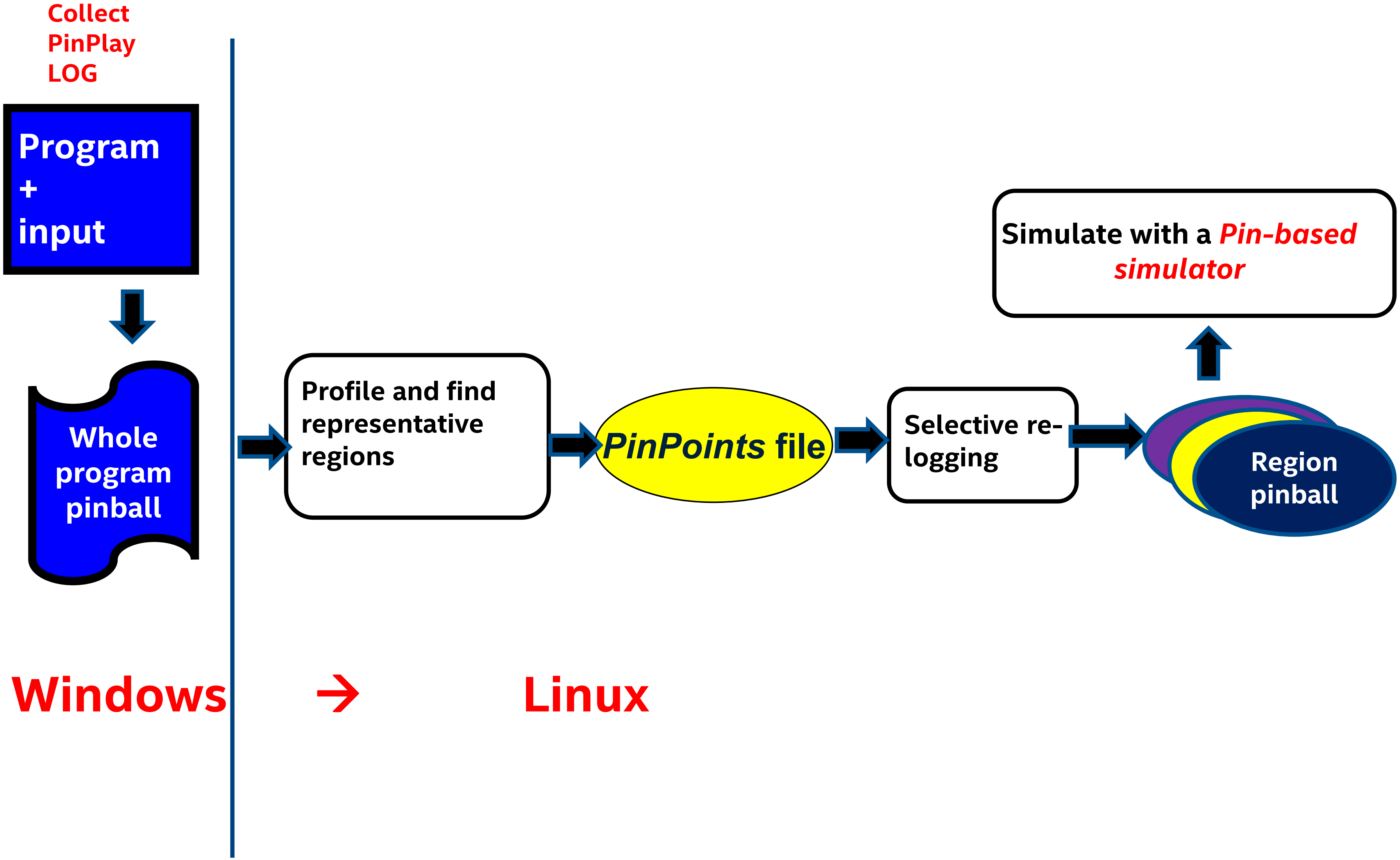
```
  --cfg FILE, --config_file FILE
                        Give one, or more, file(s) containing the application
                        tracing parameters. Must use '--cfg' for each file.
```



```
hdci2309> cat metro.cfg
[Parameters]
program_name:   metro_calc
input_name:     test
command:        foo
mode:           mt
warmup_length:  3000000
slice_size:     1000000
```



Cross-OS PinPlay + PinPoints



Collect PinPlay LOG

Program + input

Whole program pinball

Profile and find representative regions

PinPoints file

Selective re-logging

Region pinball

Simulate with a *Pin-based simulator*

Windows → **Linux**

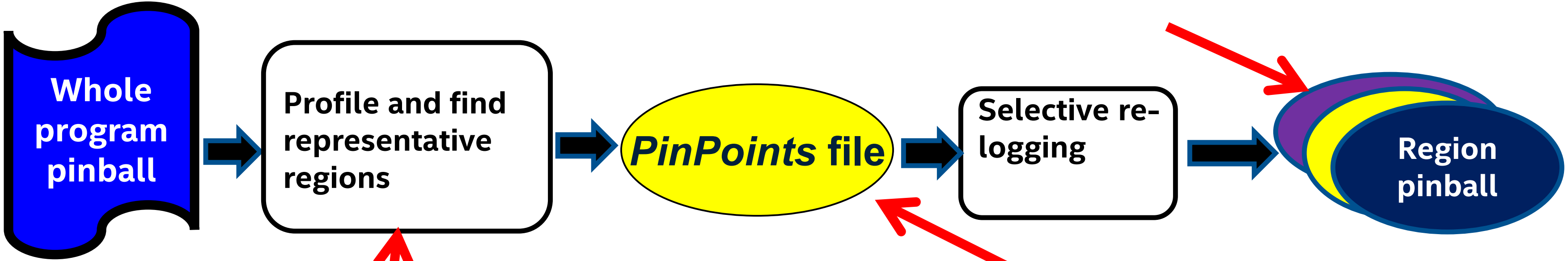
metro_calc PinPoints: Linux

```
% $PIN_ROOT/extras/pinplay/scripts/pinpoints.py --cross_os
--cfg metro.cfg --whole_pgm_dir metro -b -s -f 1 -p -R
```

`--whole_pgm_dir metro`

Use existing whole program pinballs

-p : Relog whole program pinballs using representative regions from Simpoint to generate region pinballs



-b : Generate basic block vectors for whole program pinball (all threads)

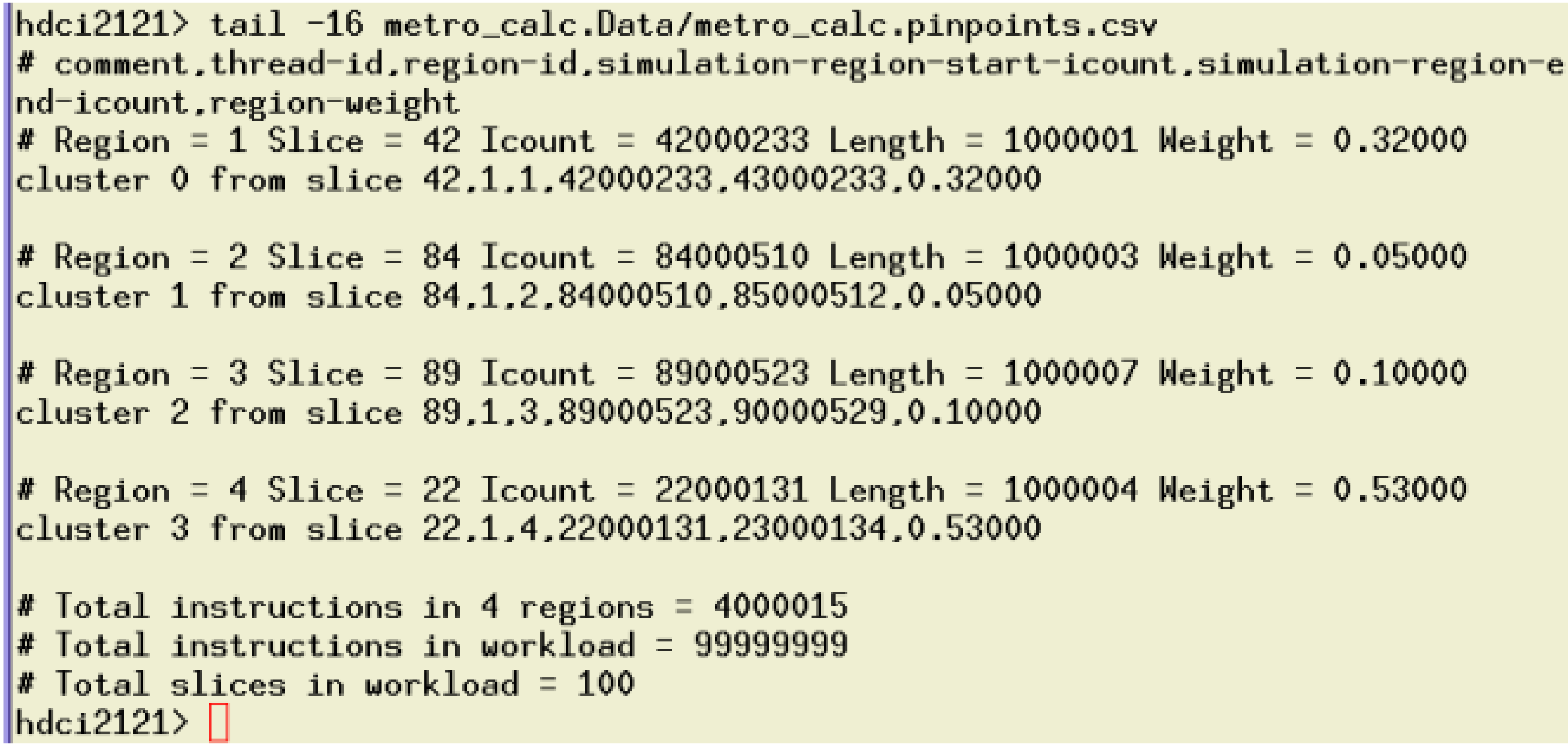
-s: Run SimPoint

-f 1 : Use basic block vectors from thread 1 only for running SimPoint

Generate (single-threaded) pinballs focusing on thread 1

-R : replay region pinballs

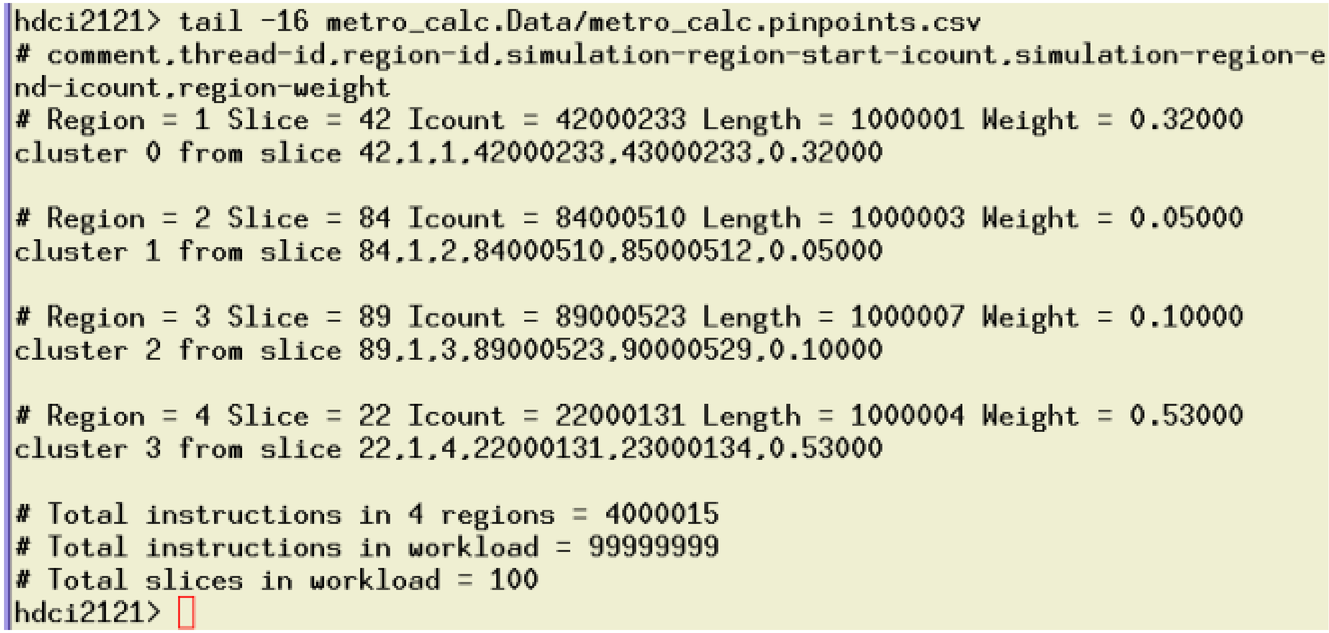
4 PinPoints ➔ 4 region pinballs



```
hdci2121> tail -16 metro_calc.Data/metro_calc.pinpoints.csv
# comment,thread-id,region-id,simulation-region-start-icount,simulation-region-e
nd-icount,region-weight
# Region = 1 Slice = 42 Icount = 42000233 Length = 1000001 Weight = 0.32000
cluster 0 from slice 42,1,1,42000233,43000233,0.32000

# Region = 2 Slice = 84 Icount = 84000510 Length = 1000003 Weight = 0.05000
cluster 1 from slice 84,1,2,84000510,85000512,0.05000

# Region = 3 Slice = 89 Icount = 89000523 Length = 1000007 Weight = 0.10000
cluster 2 from slice 89,1,3,89000523,90000529,0.10000

# Region = 4 Slice = 22 Icount = 22000131 Length = 1000004 Weight = 0.53000
cluster 3 from slice 22,1,4,22000131,23000134,0.53000

# Total instructions in 4 regions = 4000015
# Total instructions in workload = 99999999
# Total slices in workload = 100
hdci2121> 
```

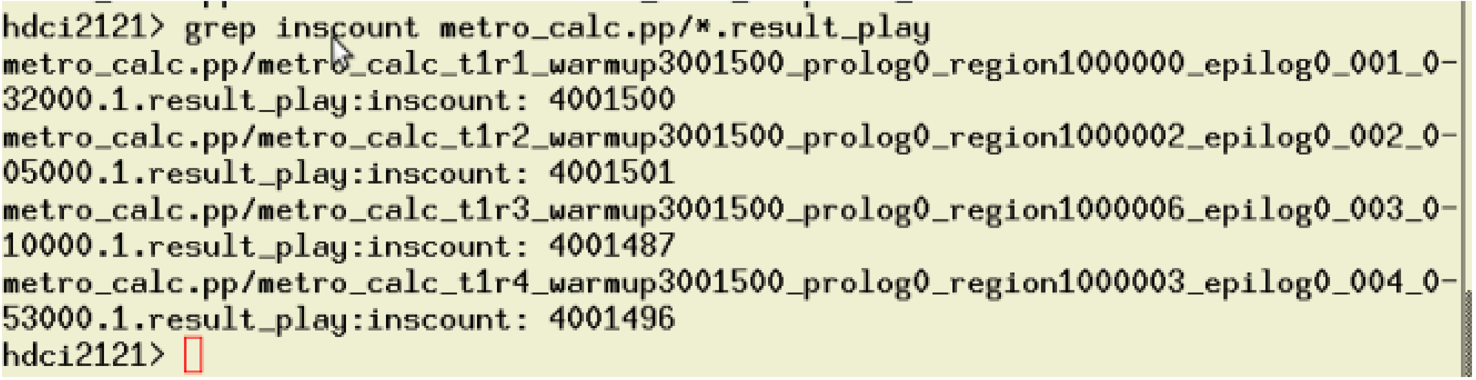
Cross-OS PinPoints results

Four PinPoints:



```
hdci2121> tail -16 metro_calc.Data/metro_calc.pinpoints.csv
# comment,thread-id,region-id,simulation-region-start-icount,simulation-region-e
nd-icount,region-weight
# Region = 1 Slice = 42 Icount = 42000233 Length = 1000001 Weight = 0.32000
cluster 0 from slice 42,1,1,42000233,43000233,0.32000

# Region = 2 Slice = 84 Icount = 84000510 Length = 1000003 Weight = 0.05000
cluster 1 from slice 84,1,2,84000510,85000512,0.05000

# Region = 3 Slice = 89 Icount = 89000523 Length = 1000007 Weight = 0.10000
cluster 2 from slice 89,1,3,89000523,90000529,0.10000

# Region = 4 Slice = 22 Icount = 22000131 Length = 1000004 Weight = 0.53000
cluster 3 from slice 22,1,4,22000131,23000134,0.53000

# Total instructions in 4 regions = 4000015
# Total instructions in workload = 99999999
# Total slices in workload = 100
hdci2121> 
```

Four (single-threaded) region pinballs :
(3 million warmup + 1 million simulation regions)



```
hdci2121> grep inscount metro_calc.pp/*.result_play
metro_calc.pp/metro_calc_t1r1_warmup3001500_prolog0_region1000000_epilog0_001_0-
32000.1.result_play:inscount: 4001500
metro_calc.pp/metro_calc_t1r2_warmup3001500_prolog0_region1000002_epilog0_002_0-
05000.1.result_play:inscount: 4001501
metro_calc.pp/metro_calc_t1r3_warmup3001500_prolog0_region1000006_epilog0_003_0-
10000.1.result_play:inscount: 4001487
metro_calc.pp/metro_calc_t1r4_warmup3001500_prolog0_region1000003_epilog0_004_0-
53000.1.result_play:inscount: 4001496
hdci2121> 
```

Optimization Notice

Dynamic Control-Flow Graph (DCFG) Library Tutorial

Programming Language Design and Implementation (PLDI)
June 14, 2016, Santa Barbara, CA, USA

Chuck Yount
Principal Engineer
Intel
Software & Services Group

Overview

Tutorial goals

- Show how to create a Dynamic Control-Flow Graph (DCFG) from a PinPlay-enabled tool
- Introduce API that can be used to read data from an existing DCFG during replay

Agenda

- DCFG Motivation & Definition
- How to make a DCFG
- How to use a DCFG

DCFG Motivation

General

- A control-flow graph (CFG) is a fundamental structure used in computer science and engineering for describing and analyzing the structure of an algorithm or program
- Used in many discovery, debugging, and performance-analysis tools

Why needed in Pin?

- A “BBL” in Pin does not follow the normal definition or expectations of a basic-block needed for CFG analysis
- Example: If Pin detects a jump to an instruction in the middle of an existing BBL, it will create a new, overlapping BBL beginning at the target instruction

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

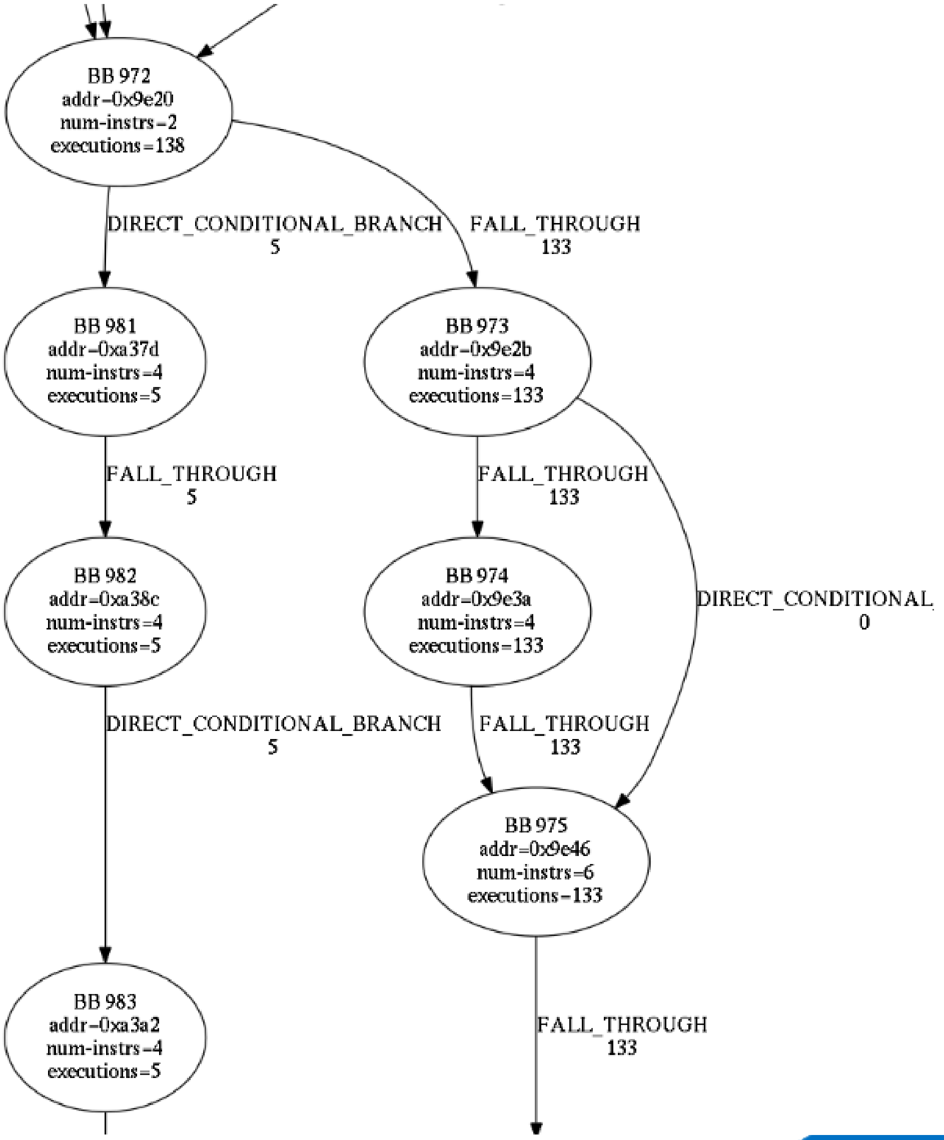
Dynamic Control-Flow Graph (DCFG)

- 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



How to create a DCFG

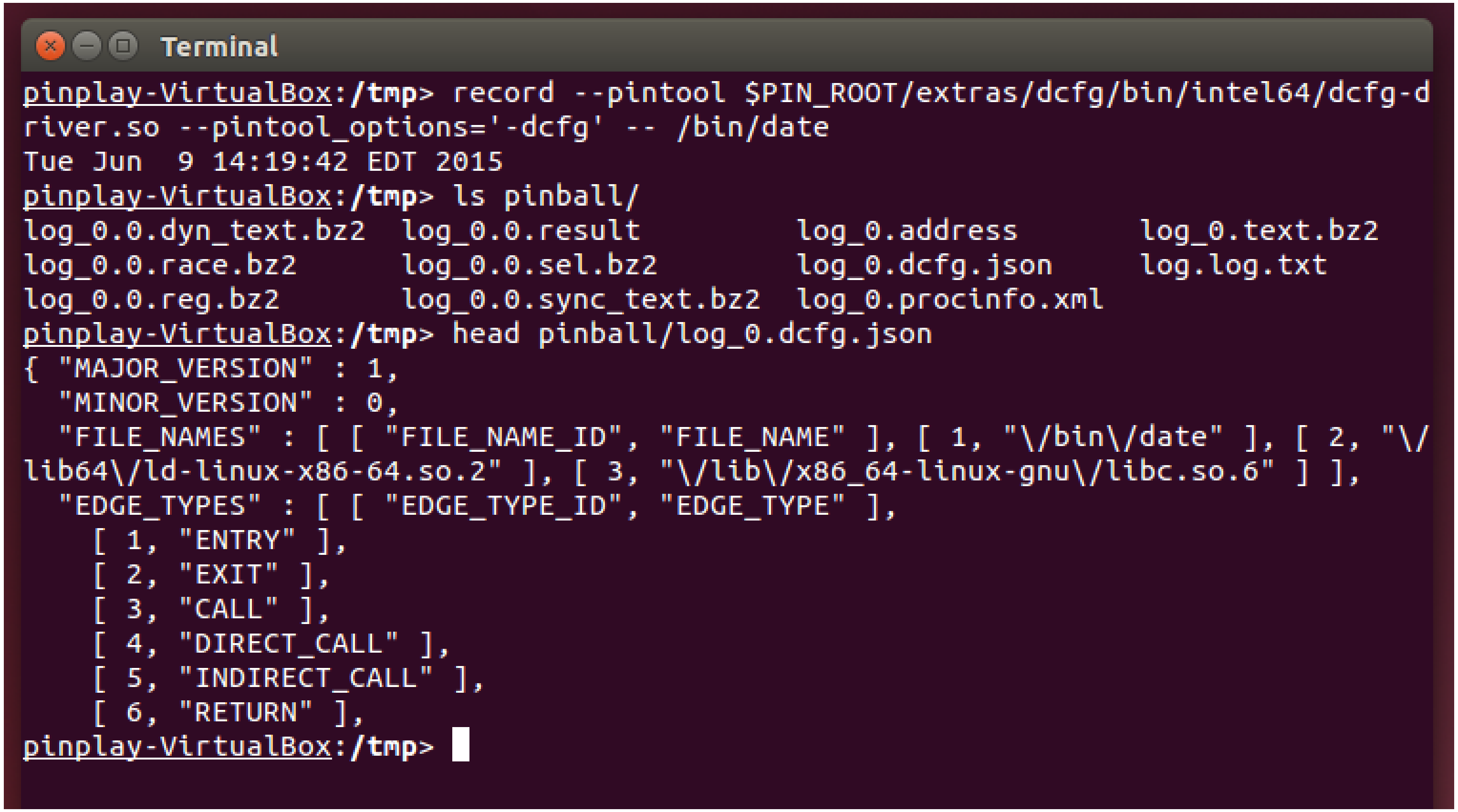
Run the ‘record’ script with the DCFG driver tool

```
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



Terminal

```
pinplay-VirtualBox:/tmp> record --pintool $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-d
river.so --pintool_options='-dcfg' -- /bin/date
Tue Jun  9 14:19:42 EDT 2015
pinplay-VirtualBox:/tmp> ls pinball/
log_0.0.dyn_text.bz2  log_0.0.result         log_0.address        log_0.text.bz2
log_0.0.race.bz2      log_0.0.sel.bz2        log_0.dcfg.json      log.log.txt
log_0.0.reg.bz2       log_0.0.sync_text.bz2  log_0.procinfo.xml
pinplay-VirtualBox:/tmp> head pinball/log_0.dcfg.json
{ "MAJOR_VERSION" : 1,
  "MINOR_VERSION" : 0,
  "FILE_NAMES" : [ [ "FILE_NAME_ID", "FILE_NAME" ], [ 1, "\/bin\/date" ], [ 2, "\/
lib64\/ld-linux-x86-64.so.2" ], [ 3, "\/lib\/x86_64-linux-gnu\/libc.so.6" ] ],
  "EDGE_TYPES" : [ [ "EDGE_TYPE_ID", "EDGE_TYPE" ],
    [ 1, "ENTRY" ],
    [ 2, "EXIT" ],
    [ 3, "CALL" ],
    [ 4, "DIRECT_CALL" ],
    [ 5, "INDIRECT_CALL" ],
    [ 6, "RETURN" ],
pinplay-VirtualBox:/tmp> 
```

Optimization Notice

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-pinplay.a and libintelzipstream.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);
    ```

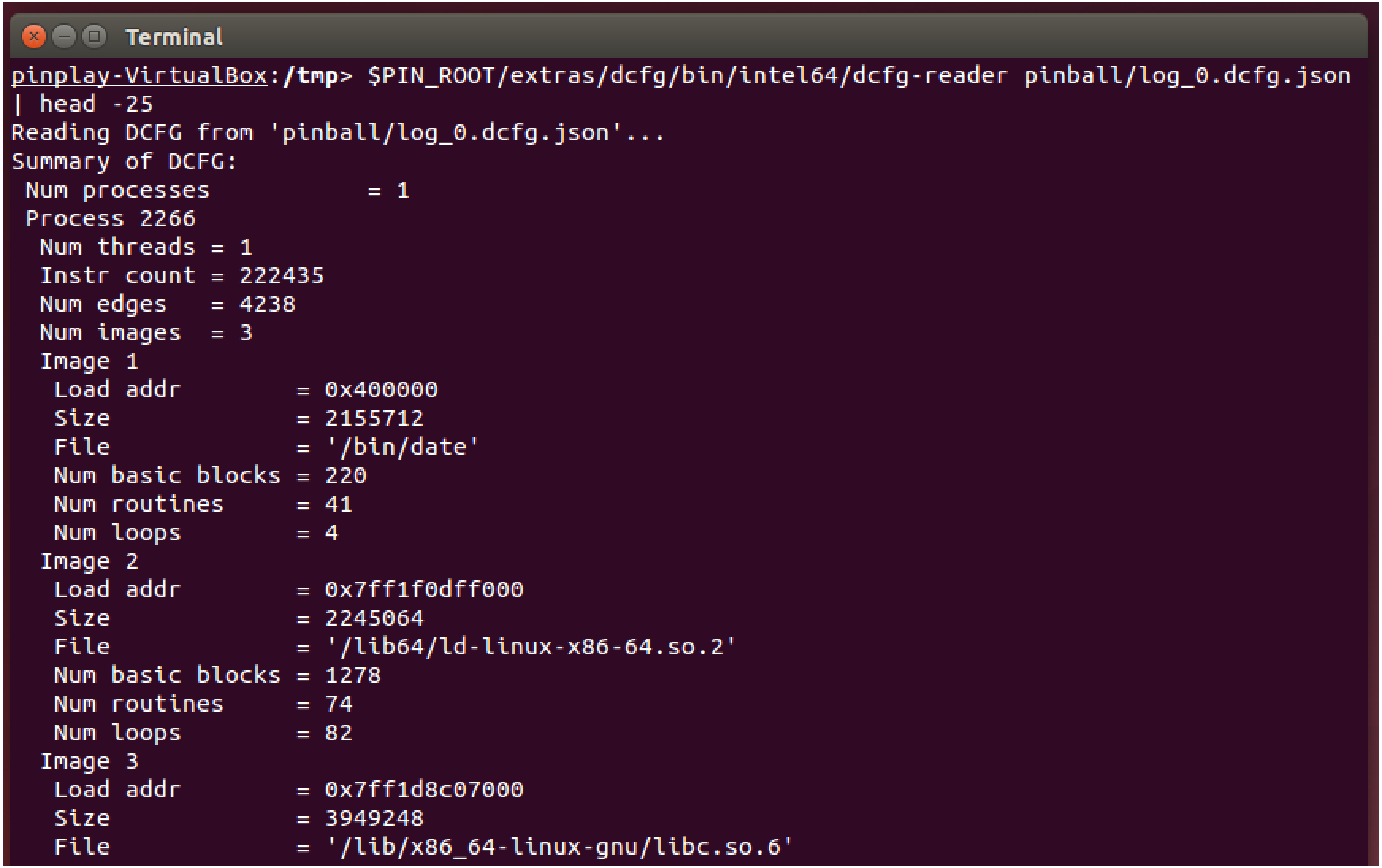
  - Then, use an ID to get a pointer to detailed data, e.g.,

    ```
    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

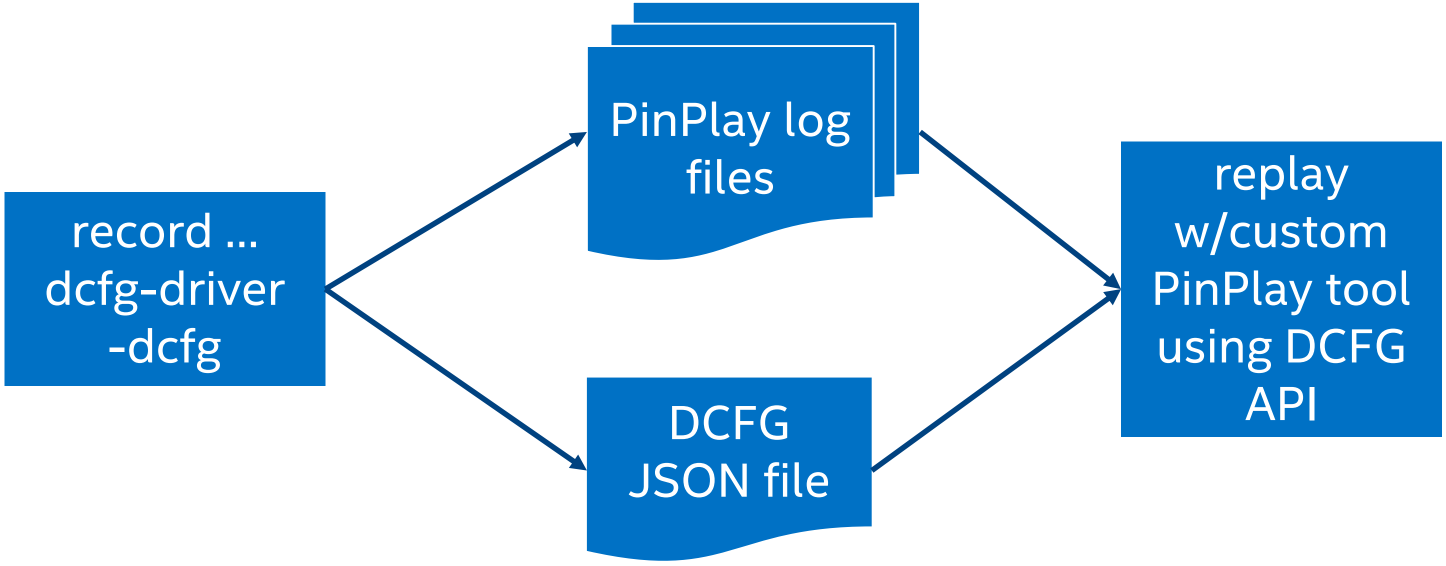


```
pinplay-VirtualBox:/tmp> $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-reader pinball/log_0.dcfg.json
| head -25
Reading DCFG from 'pinball/log_0.dcfg.json'...
Summary of DCFG:
 Num processes           = 1
 Process 2266
  Num threads = 1
  Instr count = 222435
  Num edges   = 4238
  Num images  = 3
  Image 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'
```

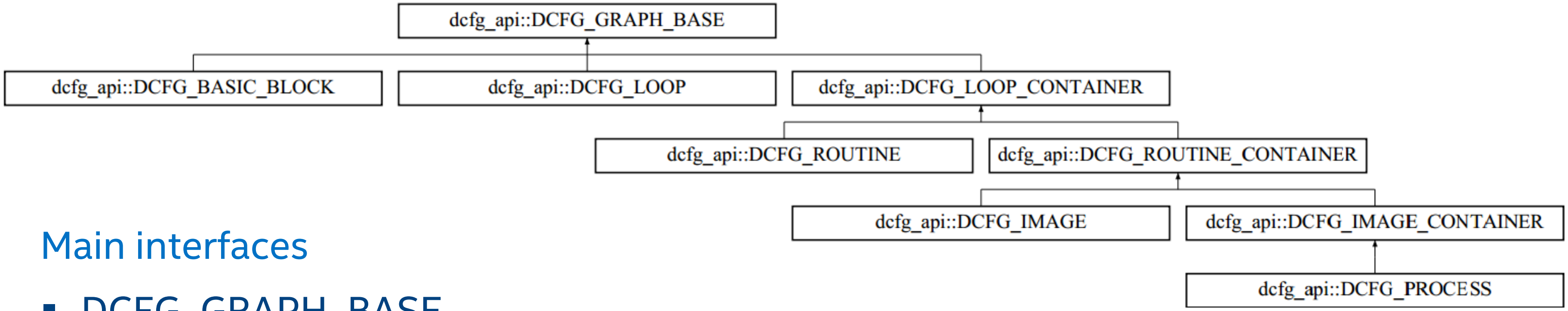
Using a DCFG during replay

Provides access to DCFG structures during replay

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



API to read DCFG


dcfg_api::DCFG_GRAPH_BASE
dcfg_api::DCFG_BASIC_BLOCK
dcfg_api::DCFG_LOOP
dcfg_api::DCFG_LOOP_CONTAINER
dcfg_api::DCFG_ROUTINE
dcfg_api::DCFG_ROUTINE_CONTAINER
dcfg_api::DCFG_IMAGE
dcfg_api::DCFG_IMAGE_CONTAINER
dcfg_api::DCFG_PROCESS

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)
- See “DCFG API description” on website for full documentation

Summary and Web Resources

DCFG creation

- Minimal dcfg-driver.so PinPlay tool creates DCFG JSON file
- Contains structure of basic-blocks, edges, loops, and more
- Contains dynamic counts of these control-flow elements

DCFG usage

- API can be used to read DCFG data during replay
- This allows a PinPlay-enabled tool to instrument code based on control-flow elements like loops

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

Dynamic Program Slicing with Replay

Joint work with Vineet Singh, Yan Wang, Rajiv Gupta, and Iulian Neamtiu University of California, Riverside

Program Slicing *[Mark Weiser, 1982]*

Definition: Slice(v@S)

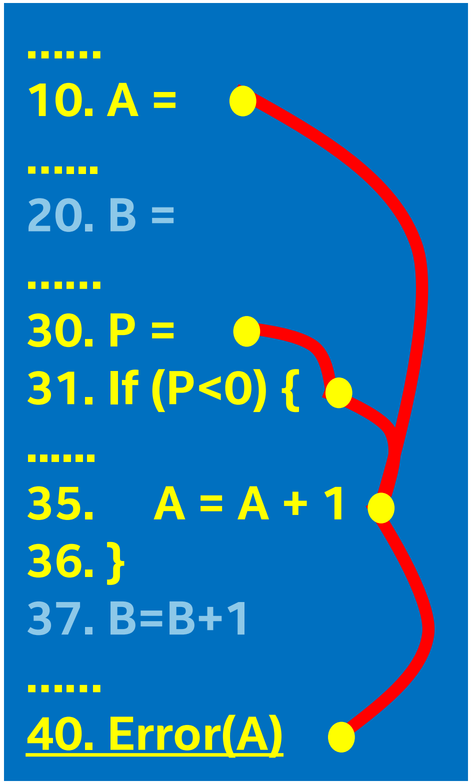
Backward slice of v at S is the set of statements involved in computing v 's value at S

Forward slice of v at S is the set of statements effected by value of v at S

Transitive closure over data dependencies and control dependencies starting from the slicing criterion (v@S)

Dynamic Slicing *[Korel and Laski, 1988]*

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



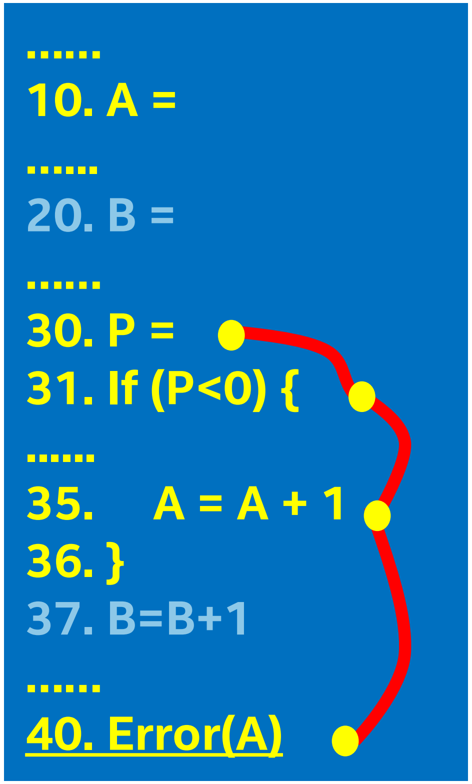
```
……
10. A =
……
20. B =
……
30. P =
31. If (P<0) {
……
35.    A = A + 1
36. }
37. B=B+1
……
40. Error(A)
```

Slicing Criterion (A@40)

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

Dynamic Slicing *[Korel and Laski, 1988]*

Dynamic forward slice is the set of statements that **were** affected by the value of a variable at a program point for ONE specific execution.



```
……
10. A =
……
20. B =
……
30. P =
31. If (P<0) {
……
35.    A = A + 1
36. }
37. B=B+1
……
40. Error(A)
```

Slicing Criterion (P@30)

Dynamic Forward Slice (P@30) = {30, 31, 35, 40}

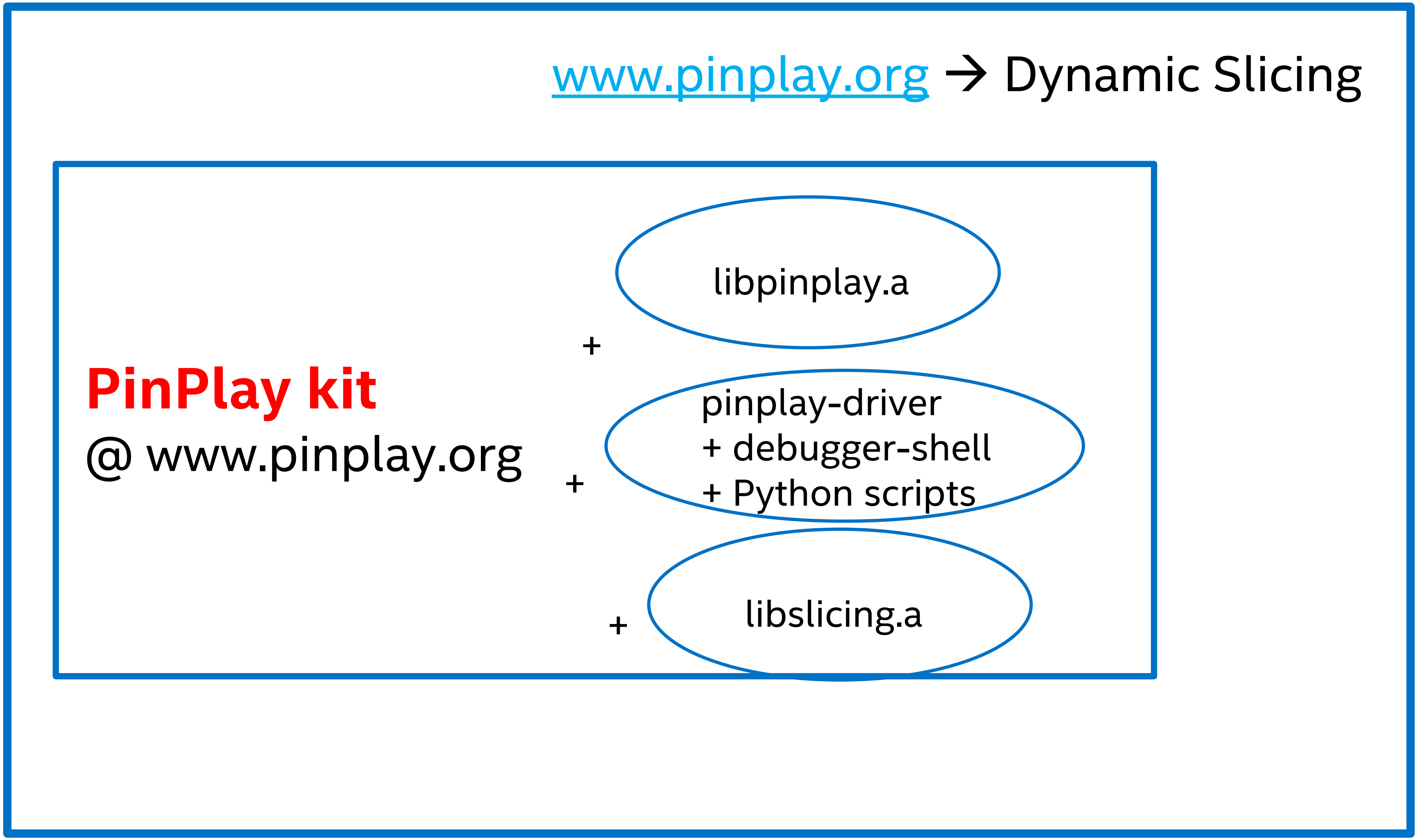
Dynamic Slicing with PinPlay

New for 2016: support for forward slicing

Based on deterministic replay: Produces the same slice for a given slicing criterion every time (even for multithreaded programs).

Supports both interactive and non-interactive modes.

Distribution model : PinPlay kit + Slicing library



www.pinplay.org → Dynamic Slicing

PinPlay kit

@ www.pinplay.org

libpinplay.a

+

pinplay-driver
+ debugger-shell
+ Python scripts

+

libslicing.a

+

Interactive Slicing Steps

Only once

Record a region

Only once

Static Analysis: computes STATIC CFG + branch targets

Only once

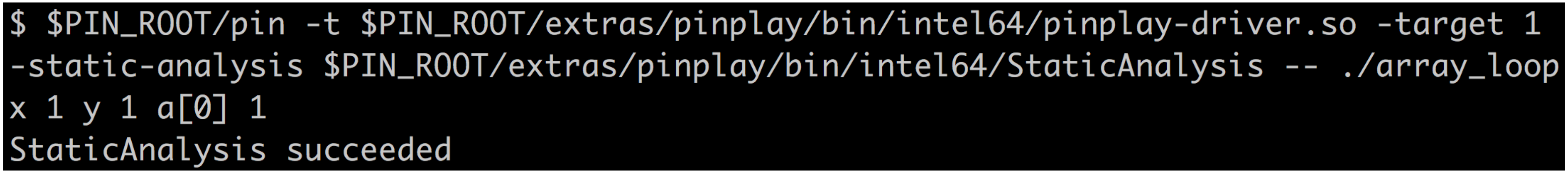
Replay with gdb

As needed

Debug + Slicing

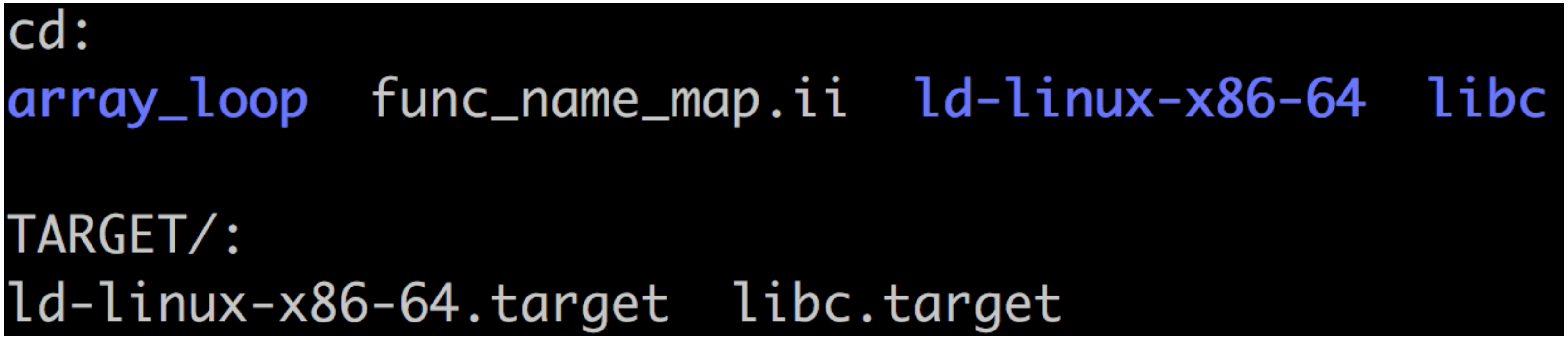
Static Analysis

Same for interactive and non-interactive modes



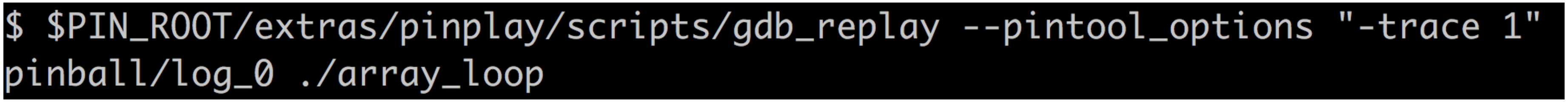
```
$ $PIN_ROOT/pin -t $PIN_ROOT/extras/pinplay/bin/intel64/pinplay-driver.so -target 1
-static-analysis $PIN_ROOT/extras/pinplay/bin/intel64/StaticAnalysis -- ./array_loop
x 1 y 1 a[0] 1
StaticAnalysis succeeded
```

Results



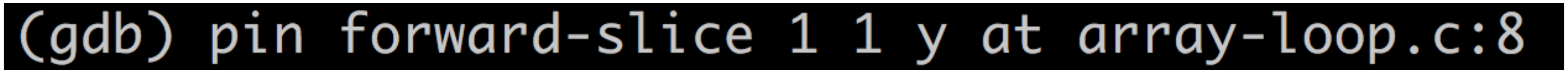
```
cd:
array_loop  func_name_map.ii  ld-linux-x86-64  libc

TARGET/:
ld-linux-x86-64.target  libc.target
```

Replay with gdb_replay



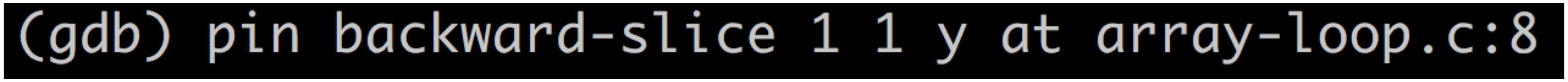
```
$ $PIN_ROOT/extras/pinplay/scripts/gdb_replay --pintool_options "-trace 1"
pinball/log_0 ./array_loop
```

Forward Slicing with gdb



```
(gdb) pin forward-slice 1 1 y at array-loop.c:8
```

Backward Slicing with gdb



```
(gdb) pin backward-slice 1 1 y at array-loop.c:8
```

Support for multiple slicing criteria in the same debug session

Non-Interactive Slicing Steps

Only once

Record a region

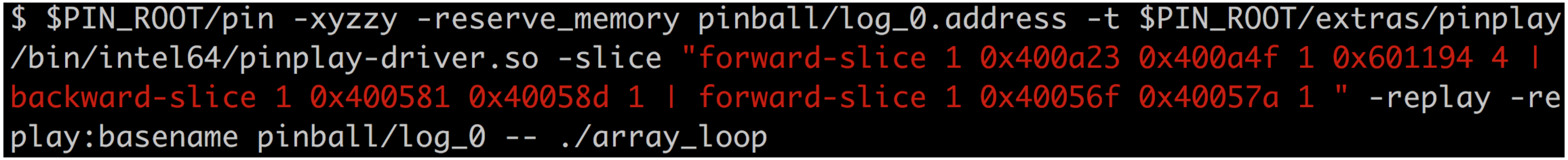
Only once

Static Analysis: computes STATIC CFG + branch targets

Only once

Replay + Slicing

Replay + Slicing



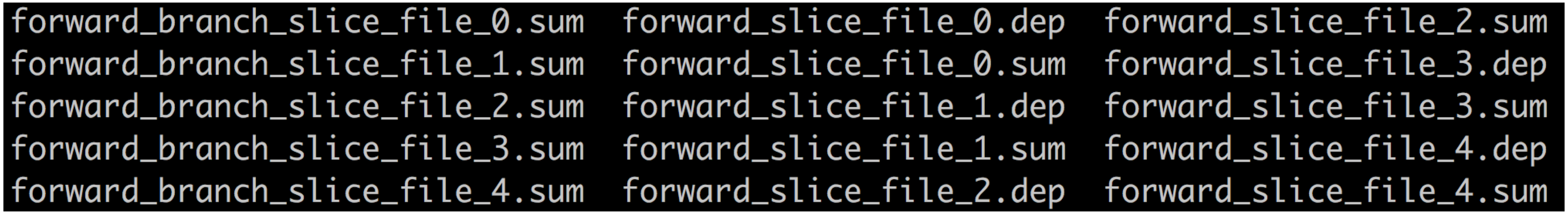
```
$ $PIN_ROOT/pin -xyzzy -reserve_memory pinball/log_0.address -t $PIN_ROOT/extras/pinplay
/bin/intel64/pinplay-driver.so -slice "forward-slice 1 0x400a23 0x400a4f 1 0x601194 4 |
backward-slice 1 0x400581 0x40058d 1 | forward-slice 1 0x40056f 0x40057a 1 " -replay -re
play:basename pinball/log_0 -- ./array_loop
```

Slicing criterion

Slice Direction Thread_Id StartPc EndPc Instance Memory_Address Size

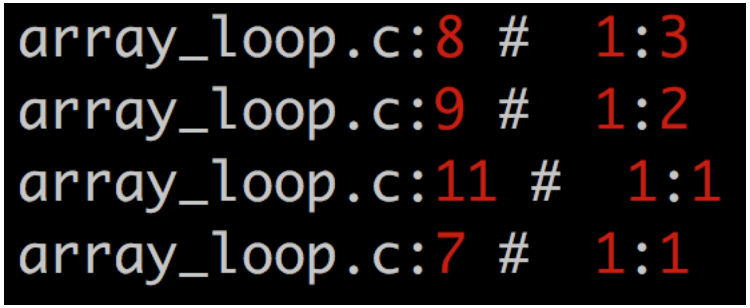
- Support for multiple slicing criteria
- Memory address specification is optional
- Useful for batch processing

Viewing Slicing Results



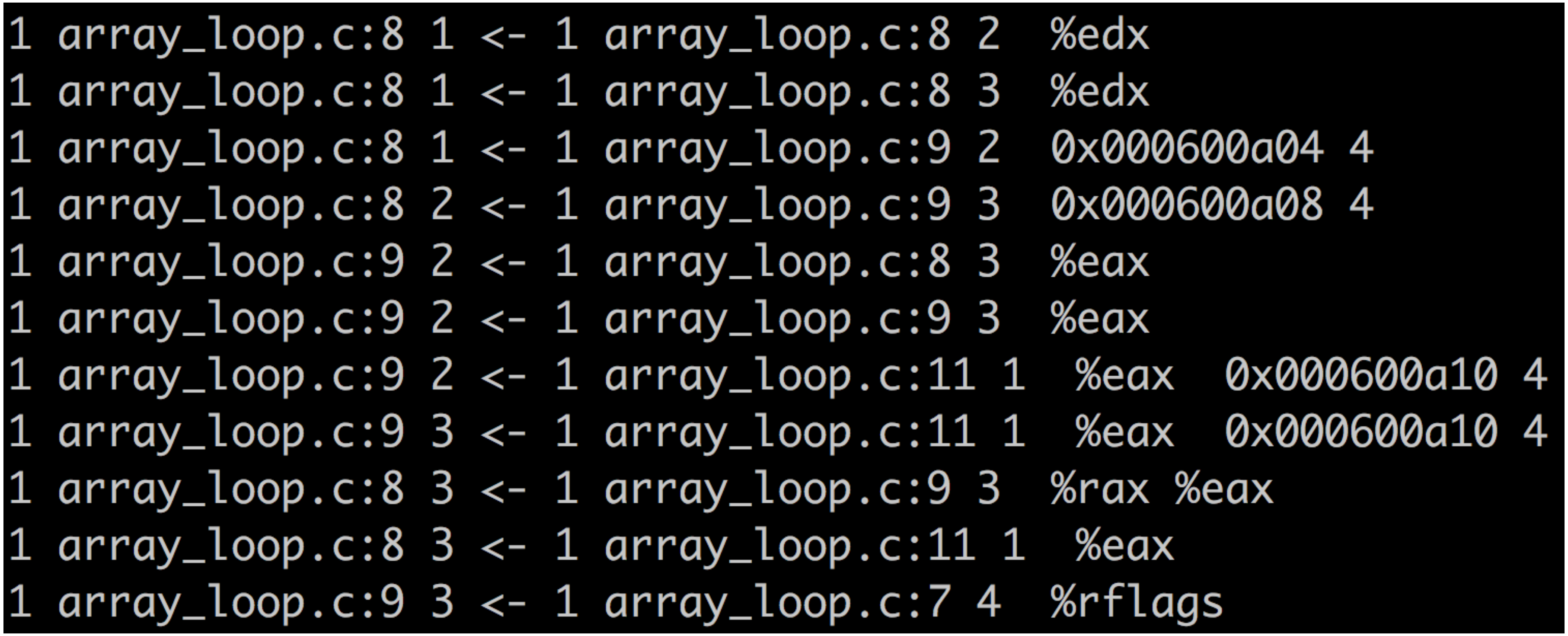
```
forward_branch_slice_file_0.sum  forward_slice_file_0.dep  forward_slice_file_2.sum
forward_branch_slice_file_1.sum  forward_slice_file_0.sum  forward_slice_file_3.dep
forward_branch_slice_file_2.sum  forward_slice_file_1.dep  forward_slice_file_3.sum
forward_branch_slice_file_3.sum  forward_slice_file_1.sum  forward_slice_file_4.dep
forward_branch_slice_file_4.sum  forward_slice_file_2.dep  forward_slice_file_4.sum
```

Slice Summary File



```
array_loop.c:8 #  1:3
array_loop.c:9 #  1:2
array_loop.c:11 #  1:1
array_loop.c:7 #  1:1
```

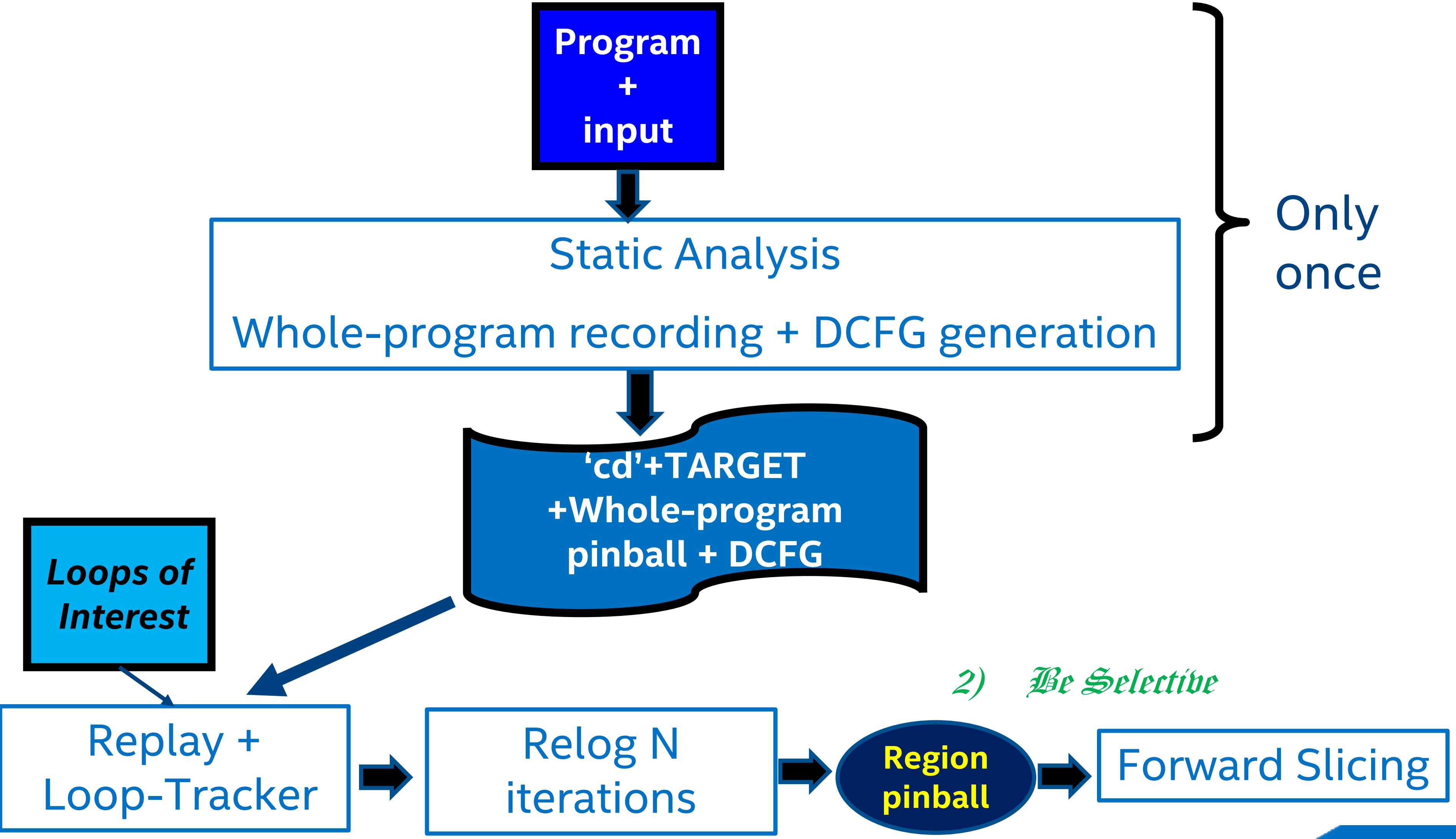
Dependency File



```
1 array_loop.c:8 1 <- 1 array_loop.c:8 2  %edx
1 array_loop.c:8 1 <- 1 array_loop.c:8 3  %edx
1 array_loop.c:8 1 <- 1 array_loop.c:9 2  0x000600a04 4
1 array_loop.c:8 2 <- 1 array_loop.c:9 3  0x000600a08 4
1 array_loop.c:9 2 <- 1 array_loop.c:8 3  %eax
1 array_loop.c:9 2 <- 1 array_loop.c:9 3  %eax
1 array_loop.c:9 2 <- 1 array_loop.c:11 1  %eax  0x000600a10 4
1 array_loop.c:9 3 <- 1 array_loop.c:11 1  %eax  0x000600a10 4
1 array_loop.c:8 3 <- 1 array_loop.c:9 3  %rax %eax
1 array_loop.c:8 3 <- 1 array_loop.c:11 1  %eax
1 array_loop.c:9 3 <- 1 array_loop.c:7 4  %rflags
```

Example Tool-chain
Loop Tracking with
Replay+DCFG+Slicing

Track Source-level Loops & Find Loop-carried Dependences



Program + input

Static Analysis

Whole-program recording + DCFG generation

Only once

'cd'+TARGET +Whole-program pinball + DCFG

Loops of Interest

Replay + Loop-Tracker

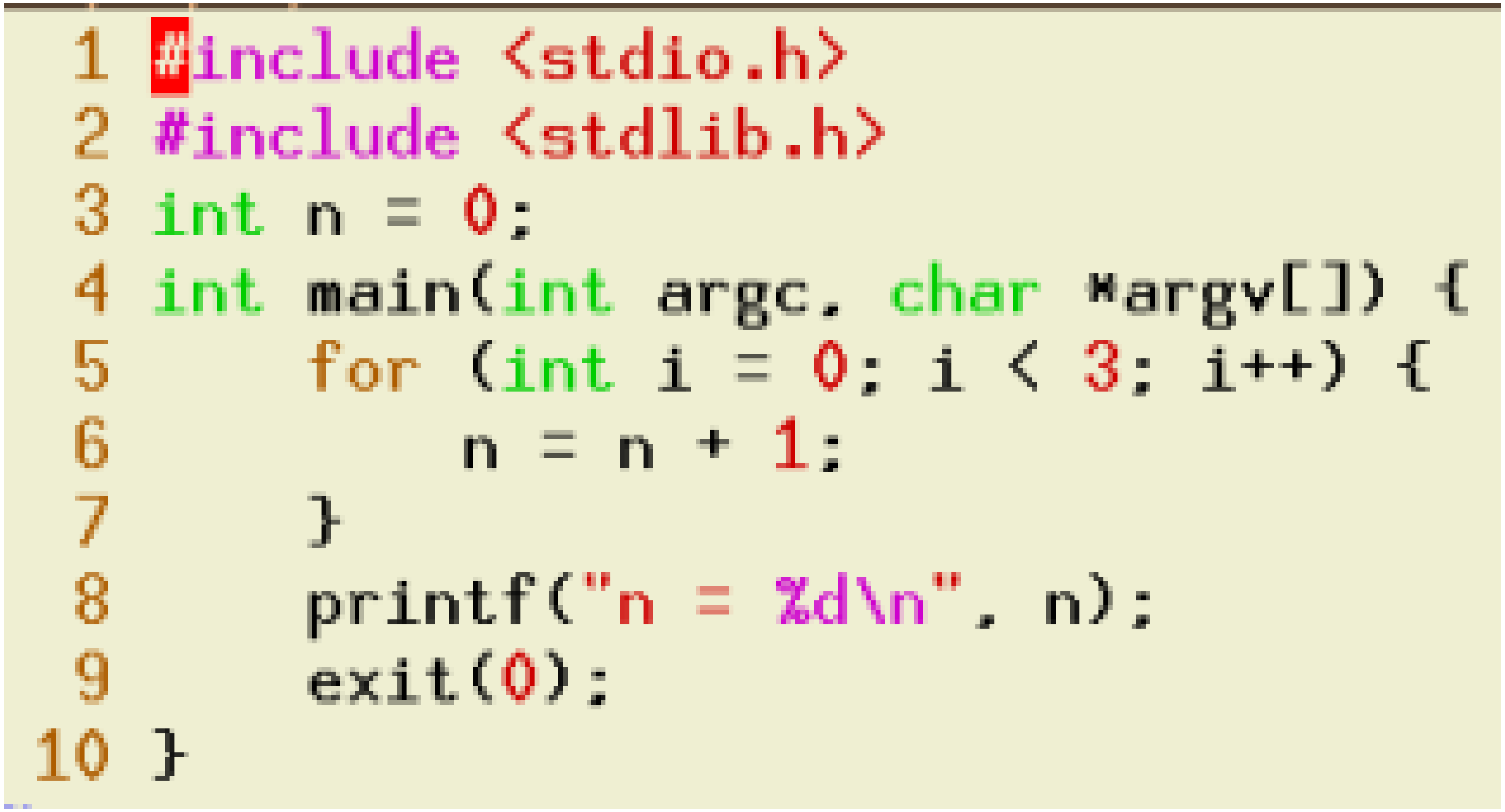
Relog N iterations

Region pinball

Forward Slicing

2) Be Selective

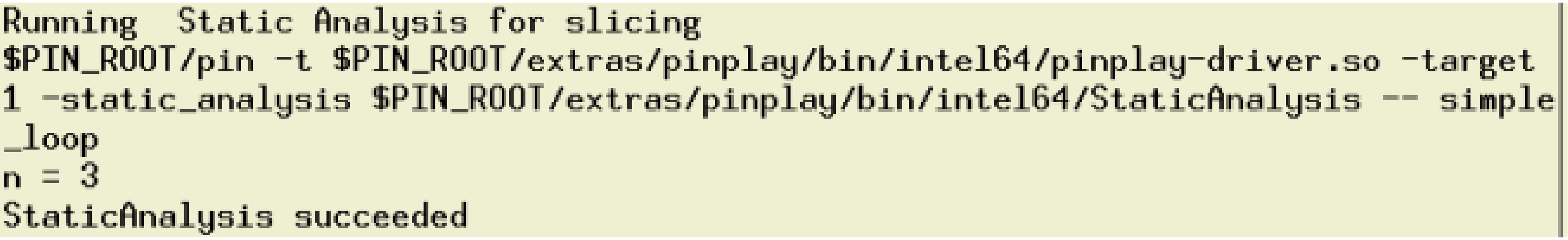
simple_loop.c



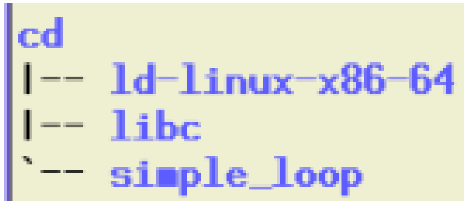
```c
#include <stdio.h>
#include <stdlib.h>
int n = 0;
int main(int argc, char *argv[]) {
    for (int i = 0; i < 3; i++) {
        n = n + 1;
    }
    printf("n = %d\n", n);
    exit(0);
}
```

Loop of Interest

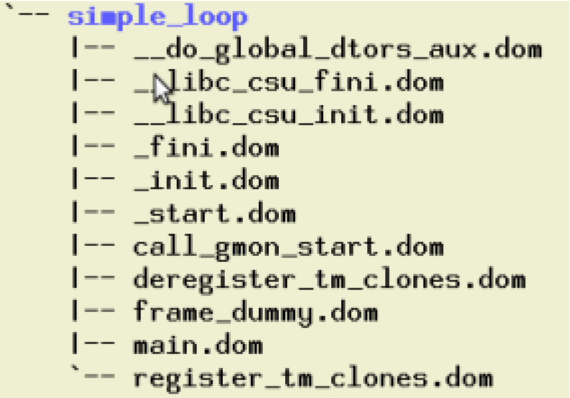
Static Analysis



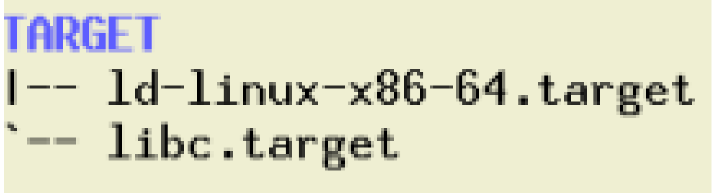
```
Running  Static Analysis for slicing
$PIN_ROOT/pin -t $PIN_ROOT/extras/pinplay/bin/intel64/pinplay-driver.so -target
1 -static_analysis $PIN_ROOT/extras/pinplay/bin/intel64/StaticAnalysis -- simple
_loop
n = 3
StaticAnalysis succeeded
```



```
cd
|-- ld-linux-x86-64
|-- libc
`-- simple_loop
```

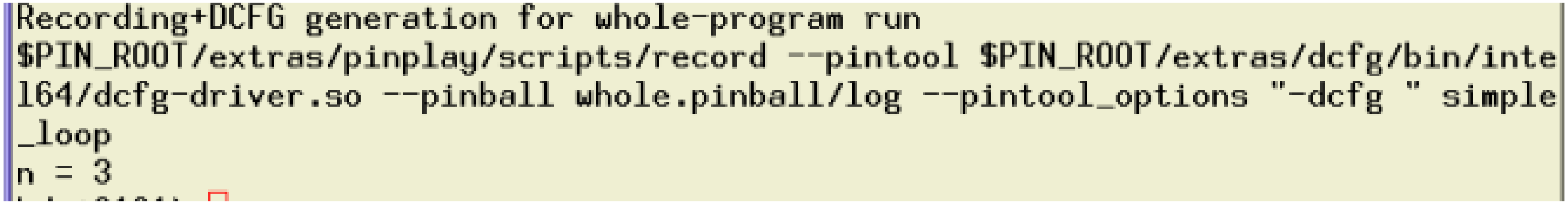


```
`-- simple_loop
    |-- __do_global_dtors_aux.dom
    |-- __libc_csu_fini.dom
    |-- __libc_csu_init.dom
    |-- _fini.dom
    |-- _init.dom
    |-- _start.dom
    |-- call_gmon_start.dom
    |-- deregister_tm_clones.dom
    |-- frame_dummy.dom
    |-- main.dom
    `-- register_tm_clones.dom
```

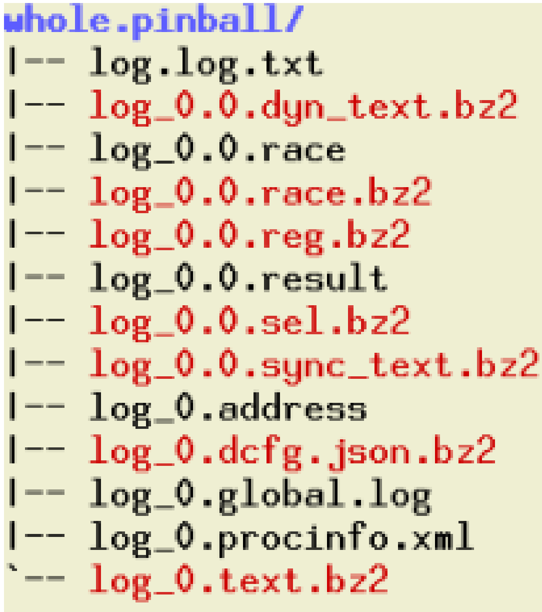


```
TARGET
|-- ld-linux-x86-64.target
`-- libc.target
```

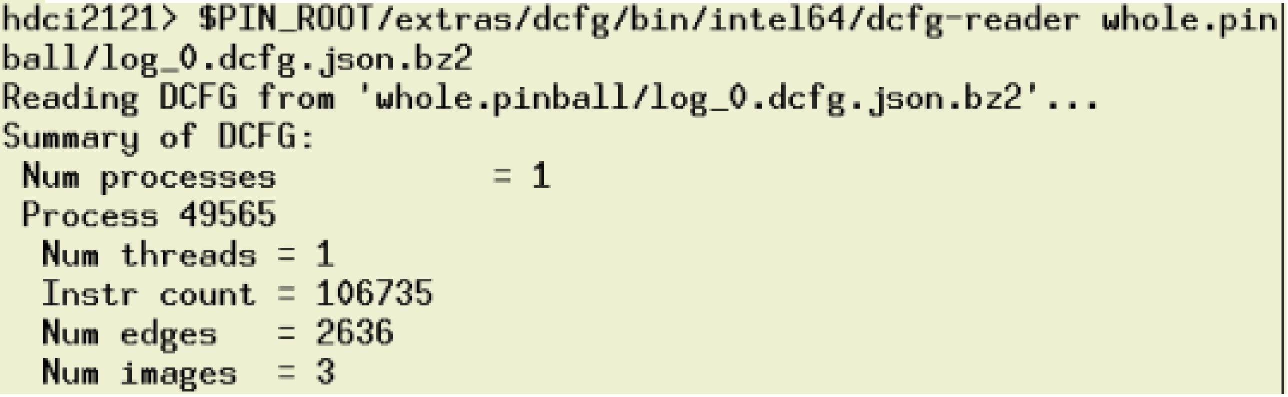
Whole-program recording + DCFG generation



```
Recording+DCFG generation for whole-program run
$PIN_ROOT/extras/pinplay/scripts/record --pintool $PIN_ROOT/extras/dcfg/bin/inte
l64/dcfg-driver.so --pinball whole.pinball/log --pintool_options "-dcfg " simple
_loop
n = 3
```

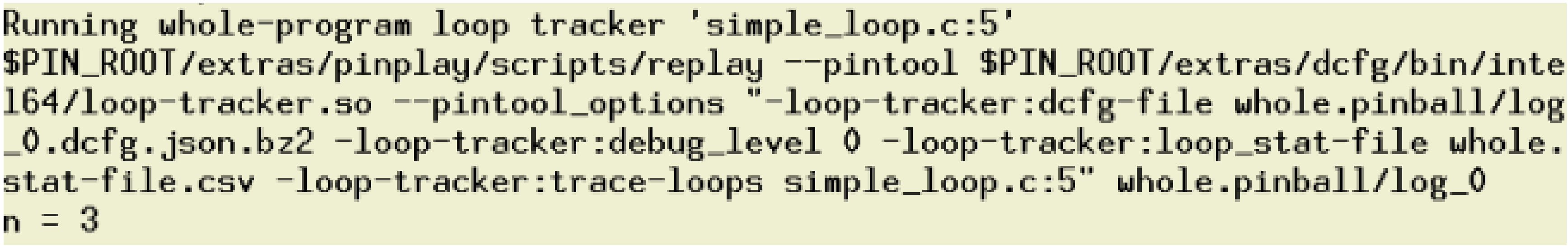


```
whole.pinball/
|-- log.log.txt
|-- log_0.0.dyn_text.bz2
|-- log_0.0.race
|-- log_0.0.race.bz2
|-- log_0.0.reg.bz2
|-- log_0.0.result
|-- log_0.0.sel.bz2
|-- log_0.0.sync_text.bz2
|-- log_0.address
|-- log_0.dcfg.json.bz2
|-- log_0.global.log
|-- log_0.procinfo.xml
`-- log_0.text.bz2
```



```
hdci2121> $PIN_ROOT/extras/dcfg/bin/intel64/dcfg-reader whole.pin
ball/log_0.dcfg.json.bz2
Reading DCFG from 'whole.pinball/log_0.dcfg.json.bz2'...
Summary of DCFG:
 Num processes            = 1
 Process 49565
  Num threads = 1
  Instr count = 106735
  Num edges   = 2636
  Num images  = 3
```

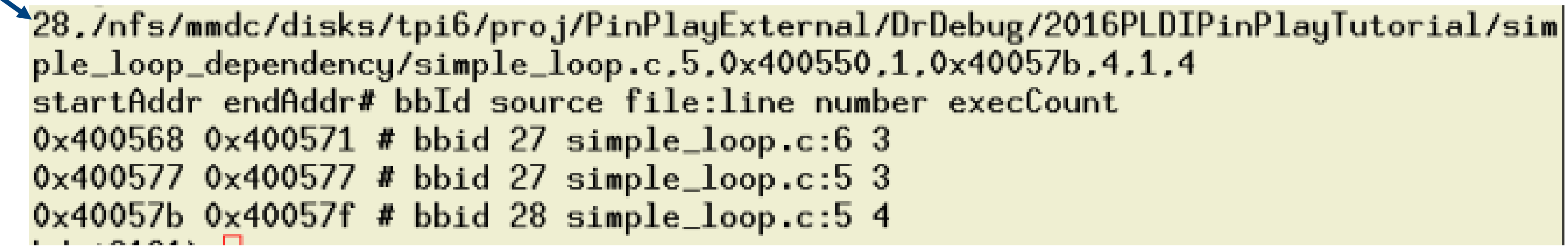
Replay + Loop-Tracker



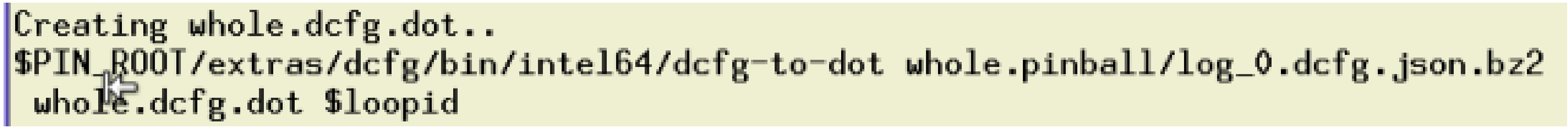
```
Running whole-program loop tracker 'simple_loop.c:5'
$PIN_ROOT/extras/pinplay/scripts/replay --pintool $PIN_ROOT/extras/dcfg/bin/inte
l64/loop-tracker.so --pintool_options "-loop-tracker:dcfg-file whole.pinball/log
_0.dcfg.json.bz2 -loop-tracker:debug_level 0 -loop-tracker:loop_stat-file whole.
stat-file.csv -loop-tracker:trace-loops simple_loop.c:5" whole.pinball/log_0
n = 3
```

Loop-id

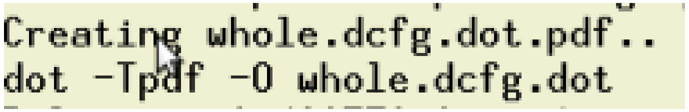
whole.loop-stats.csv



```
28,/nfs/mmdc/disks/tpi6/proj/PinPlayExternal/DrDebug/2016PLDIPinPlayTutorial/sim
ple_loop_dependency/simple_loop.c,5,0x400550,1,0x40057b,4,1,4
startAddr endAddr# bbId source file:line number execCount
0x400568 0x400571 # bbid 27 simple_loop.c:6 3
0x400577 0x400577 # bbid 27 simple_loop.c:5 3
0x40057b 0x40057f # bbid 28 simple_loop.c:5 4
```

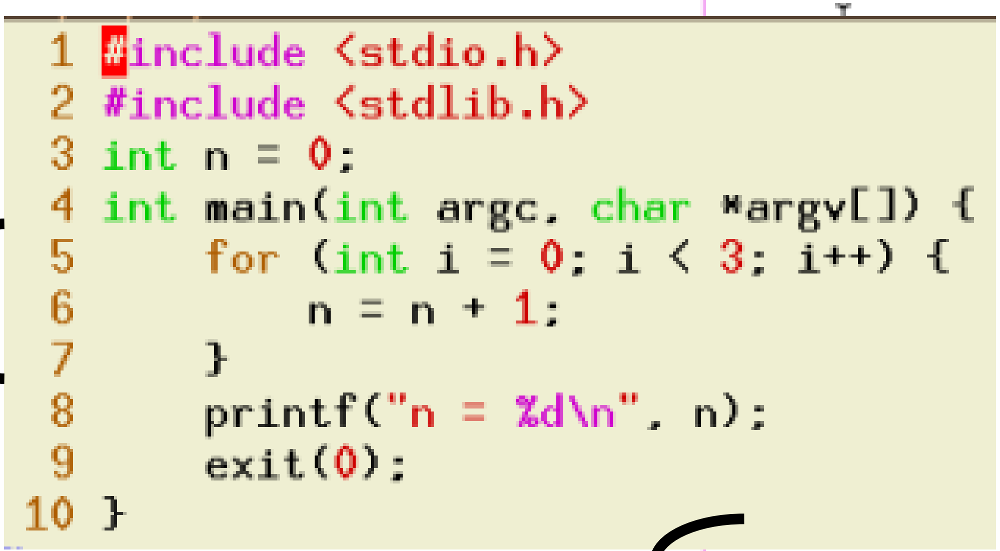


```
Creating whole.dcfg.dot..
$PIN_ROOT/extras/dcfg/bin/intel64/dcfg-to-dot whole.pinball/log_0.dcfg.json.bz2
 whole.dcfg.dot $loopid
```



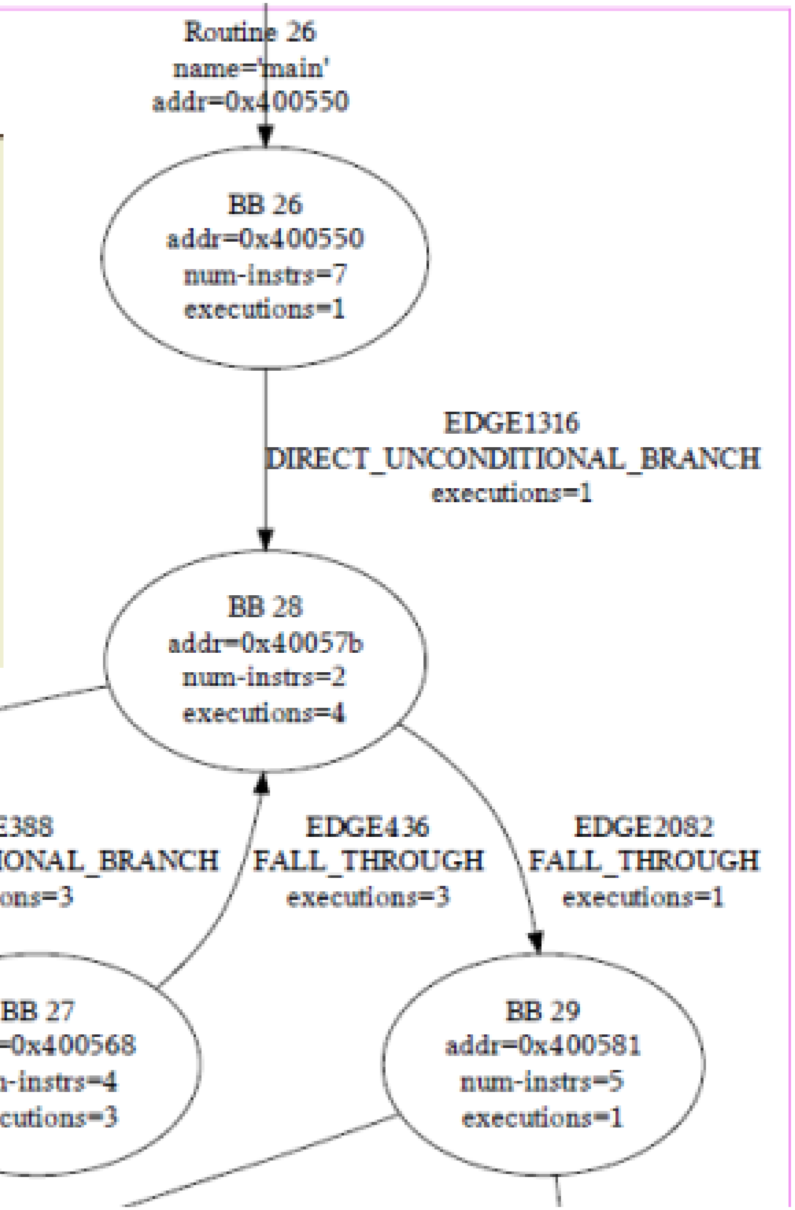
```
Creating whole.dcfg.dot.pdf..
dot -Tpdf -O whole.dcfg.dot
```

Visualizing DCFG



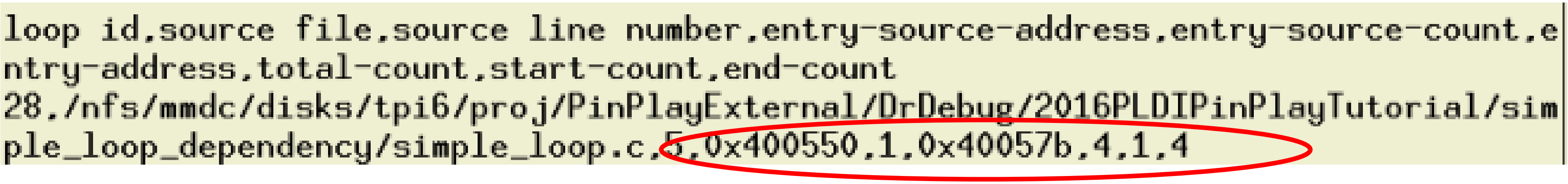
```
#include <stdio.h>
#include <stdlib.h>
int n = 0;
int main(int argc, char *argv[]) {
    for (int i = 0; i < 3; i++) {
        n = n + 1;
    }
    printf("n = %d\n", n);
    exit(0);
}
```

Loop of Interest

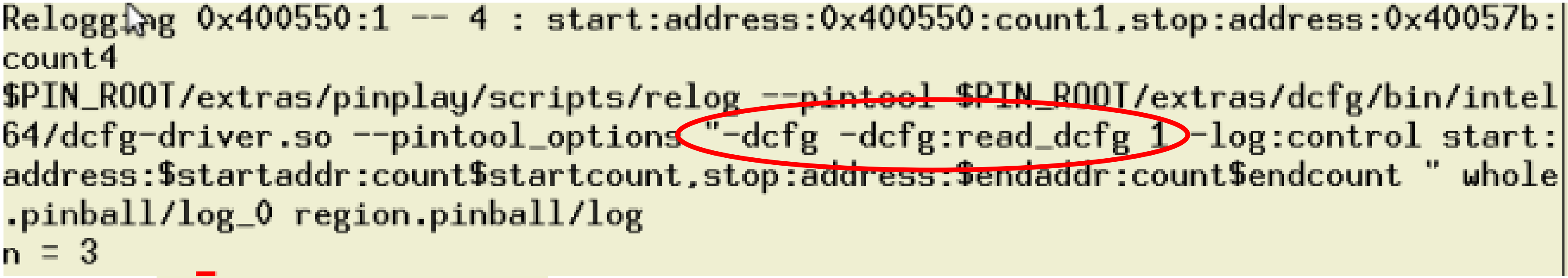


Relog N iterations

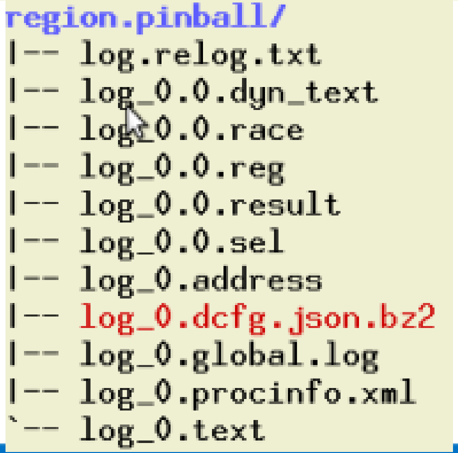
whole.loop-stats.csv



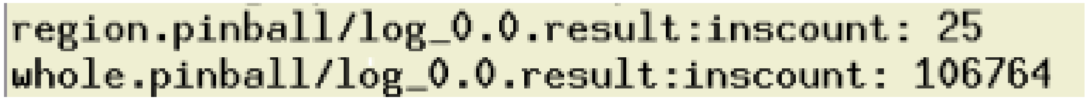
```
loop id,source file,source line number,entry-source-address,entry-source-count,e
ntry-address,total-count,start-count,end-count
28,/nfs/mmdc/disks/tpi6/proj/PinPlayExternal/DrDebug/2016PLDIPinPlayTutorial/sim
ple_loop_dependency/simple_loop.c,5,0x400550,1,0x40057b,4,1,4
```



```
Relogging 0x400550:1 -- 4 : start:address:0x400550:count1,stop:address:0x40057b:
count4
$PIN_ROOT/extras/pinplay/scripts/relog --pintool $PIN_ROOT/extras/dcfg/bin/intel
64/dcfg-driver.so --pintool_options "-dcfg -dcfg:read_dcfg 1 -log:control start:
address:$startaddr:count$startcount,stop:address:$endaddr:count$endcount " whole
.pinball/log_0 region.pinball/log
n = 3
```



```
region.pinball/
|-- log.relog.txt
|-- log_0.0.dyn_text
|-- log_0.0.race
|-- log_0.0.reg
|-- log_0.0.result
|-- log_0.0.sel
|-- log_0.address
|-- log_0.dcfg.json.bz2
|-- log_0.global.log
|-- log_0.procinfo.xml
`-- log_0.text
```



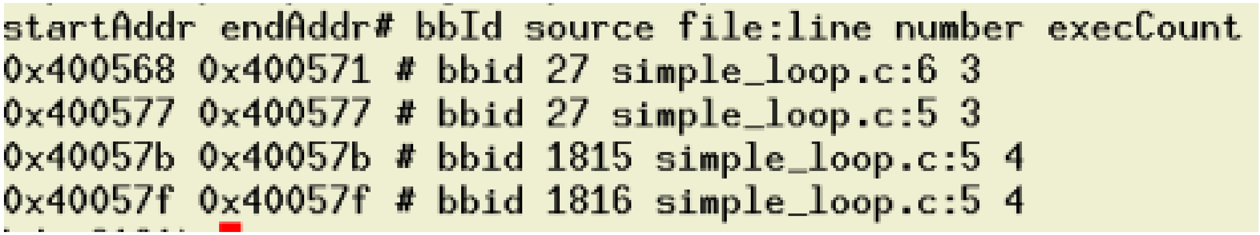
```
region.pinball/log_0.0.result:inscount: 25
whole.pinball/log_0.0.result:inscount: 106764
```

Region pinball

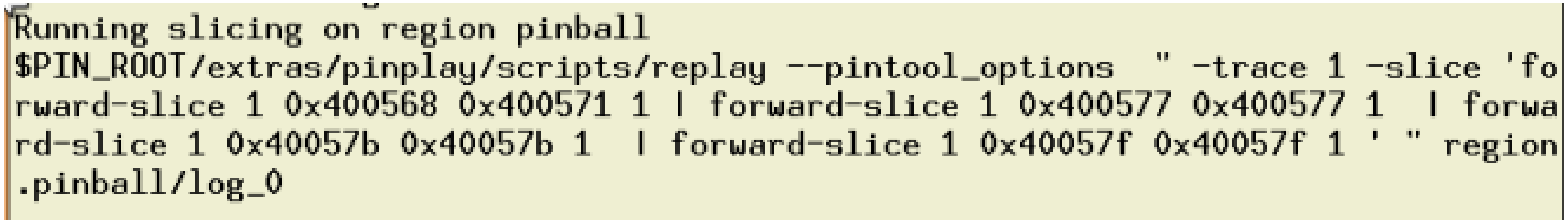
Forward Slicing

region.loop-stats.csv

(Run loop-tracker on region pinball)

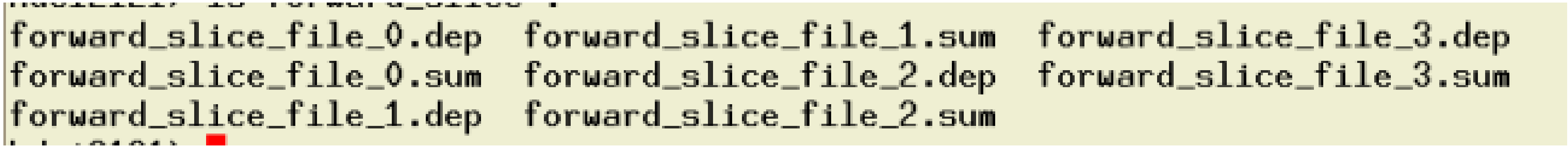


```
startAddr endAddr# bbId source file:line number execCount
0x400568 0x400571 # bbid 27 simple_loop.c:6 3
0x400577 0x400577 # bbid 27 simple_loop.c:5 3
0x40057b 0x40057b # bbid 1815 simple_loop.c:5 4
0x40057f 0x40057f # bbid 1816 simple_loop.c:5 4
```



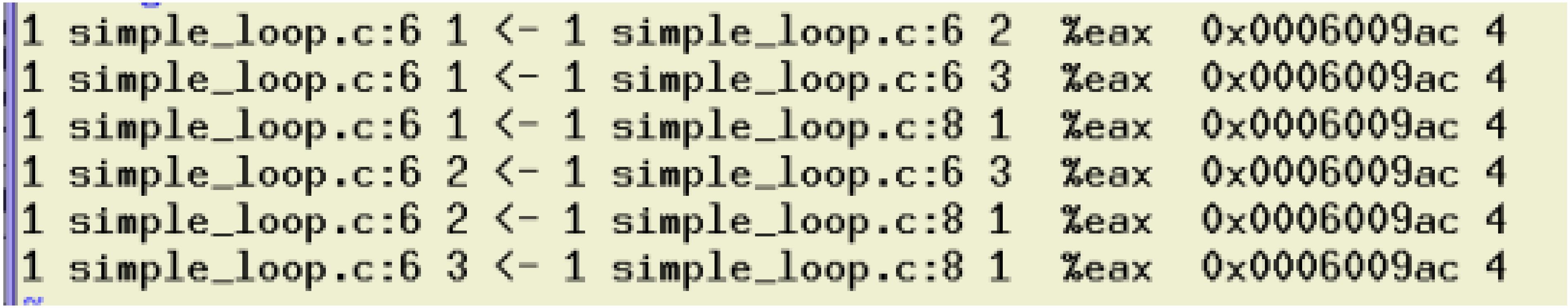
```
Running slicing on region pinball
$PIN_ROOT/extras/pinplay/scripts/replay --pintool_options  " -trace 1 -slice 'fo
rward-slice 1 0x400568 0x400571 1 | forward-slice 1 0x400577 0x400577 1  | forwa
rd-slice 1 0x40057b 0x40057b 1  | forward-slice 1 0x40057f 0x40057f 1 ' " region
.pinball/log_0
```

Results

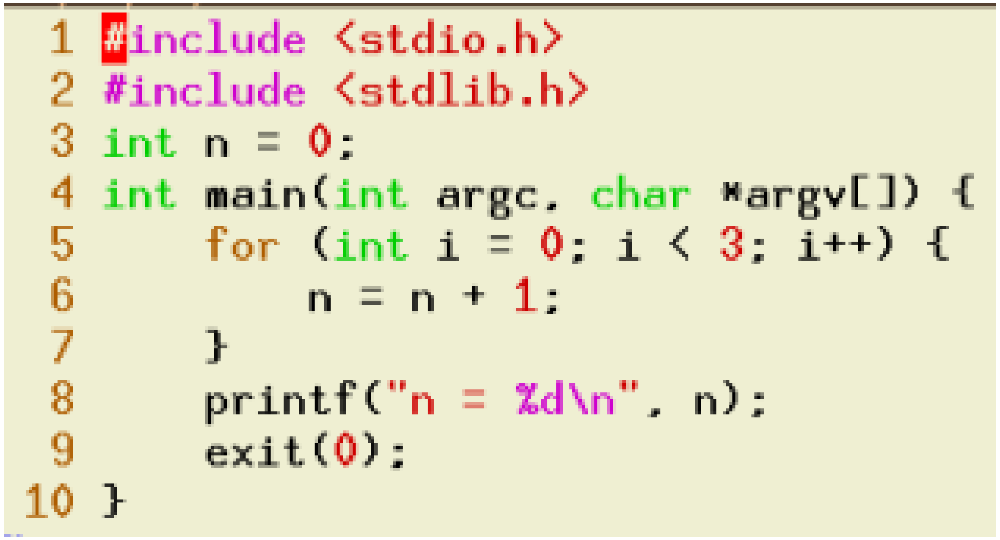


```
forward_slice_file_0.dep  forward_slice_file_1.sum  forward_slice_file_3.dep
forward_slice_file_0.sum  forward_slice_file_2.dep  forward_slice_file_3.sum
forward_slice_file_1.dep  forward_slice_file_2.sum
```

Viewing Slicing Results

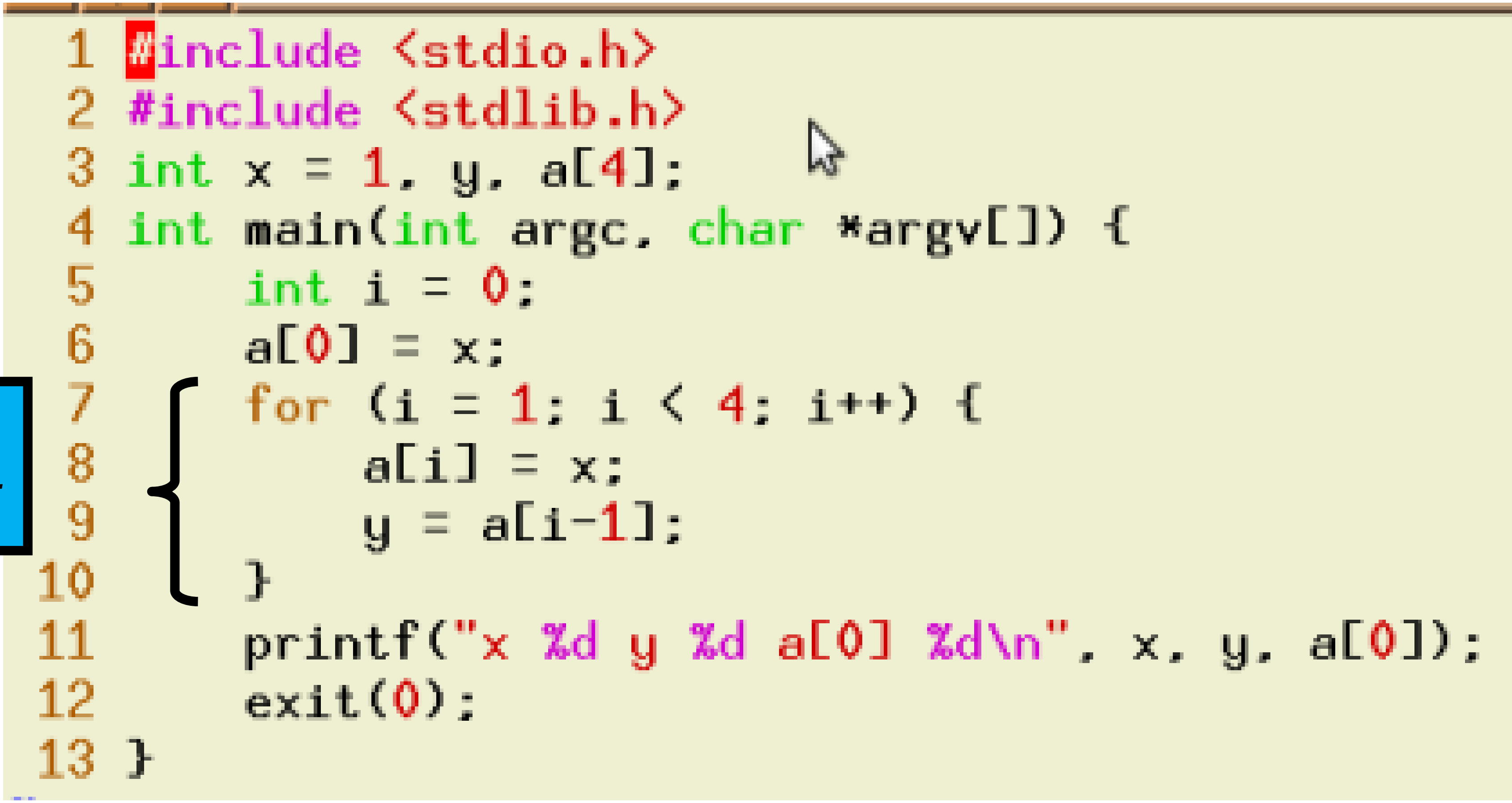


```
1 simple_loop.c:6 1 <- 1 simple_loop.c:6 2  %eax  0x0006009ac 4
1 simple_loop.c:6 1 <- 1 simple_loop.c:6 3  %eax  0x0006009ac 4
1 simple_loop.c:6 1 <- 1 simple_loop.c:8 1  %eax  0x0006009ac 4
1 simple_loop.c:6 2 <- 1 simple_loop.c:6 3  %eax  0x0006009ac 4
1 simple_loop.c:6 2 <- 1 simple_loop.c:8 1  %eax  0x0006009ac 4
1 simple_loop.c:6 3 <- 1 simple_loop.c:8 1  %eax  0x0006009ac 4
```



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int n = 0;
 4 int main(int argc, char *argv[]) {
 5     for (int i = 0; i < 3; i++) {
 6         n = n + 1;
 7     }
 8     printf("n = %d\n", n);
 9     exit(0);
10 }
```

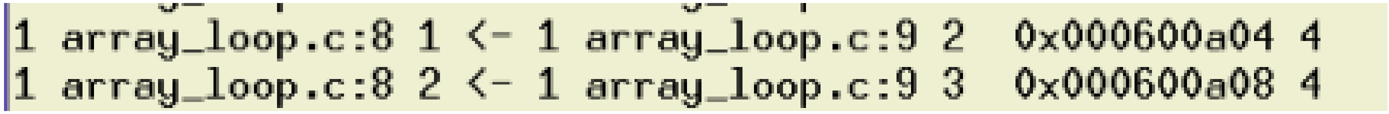
array_loop.c



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int x = 1, y, a[4];
 4 int main(int argc, char *argv[]) {
 5     int i = 0;
 6     a[0] = x;
 7     for (i = 1; i < 4; i++) {
 8         a[i] = x;
 9         y = a[i-1];
10     }
11     printf("x %d y %d a[0] %d\n", x, y, a[0]);
12     exit(0);
13 }
```

Loop of Interest

Slicing Results (excerpt)



```
1 array_loop.c:8 1 <- 1 array_loop.c:9 2   0x000600a04 4
1 array_loop.c:8 2 <- 1 array_loop.c:9 3   0x000600a08 4
```

Extending Debuggers with Pin and PinPlay

Debugging == Dynamic Program Analysis

Fact: Programmers spend majority of their time debugging

Need: better debugging tools

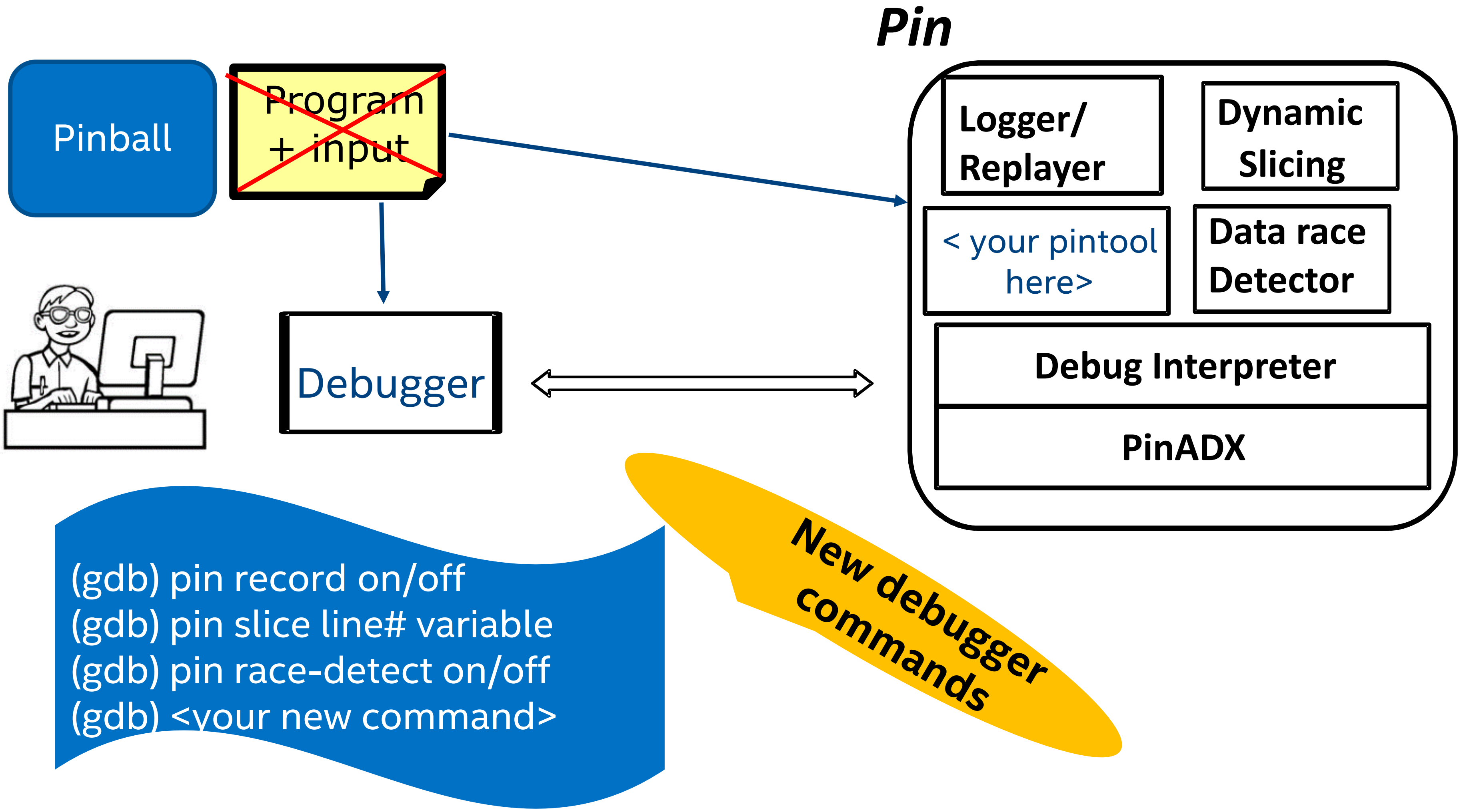
Opportunity : Enhance debugging with dynamic analysis (***PinTools/PinPlay***)

DrDebug :Dynamic analysis and **R**eplay based **Debug**ging

Goal : Debugger extensions for dynamic analysis with *Pin/PinPlay*

Not replacing debuggers but enhancing them

DrDebug: Bringing Pin's power to debugging



Pin

Pinball

~~Program + input~~

Debugger

Logger/ Replayer

Dynamic Slicing

< your pintool here>

Data race Detector

Debug Interpreter

PinADX

```
(gdb) pin record on/off
(gdb) pin slice line# variable
(gdb) pin race-detect on/off
(gdb) <your new command>
```

New debugger commands

DrDebug : Debugger interface to Pin-based analyses

Replay Debugging Foundation: PinADX

With contributions from Tevi Devor (Intel Corporation**)**

Original developer: **Greg Lueck (**Intel Corporation**)**

Transparent debugging, and extending the debugger

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

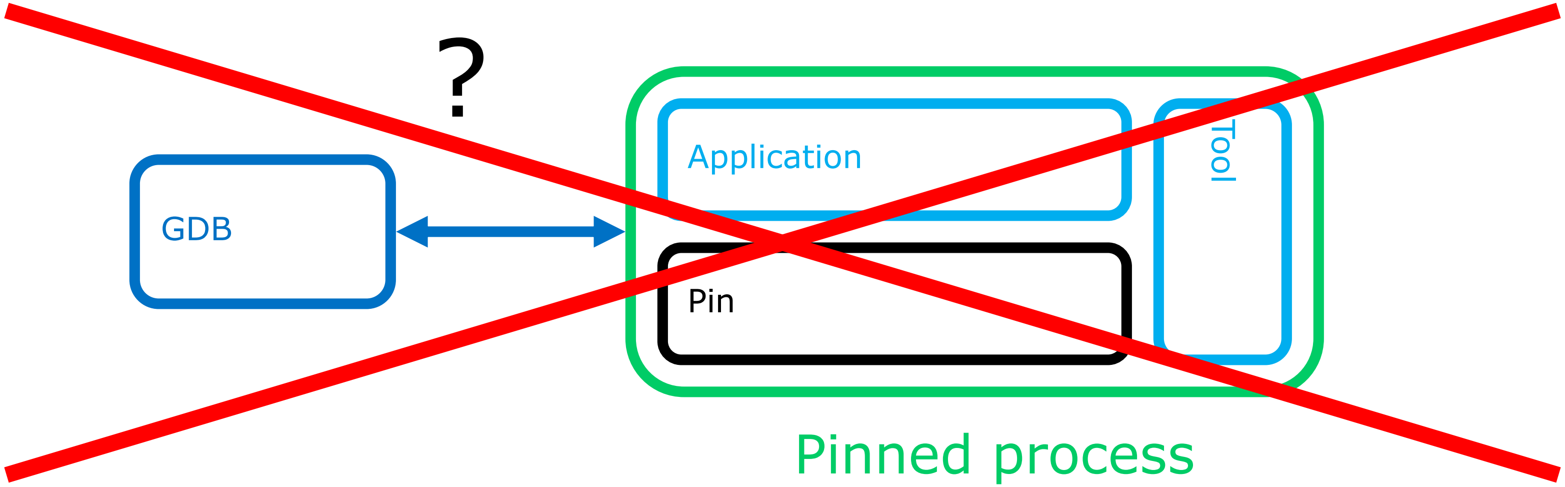
- **PinADX**: Customizable Debugging with Dynamic Instrumentation (Presented at CGO 2012)

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

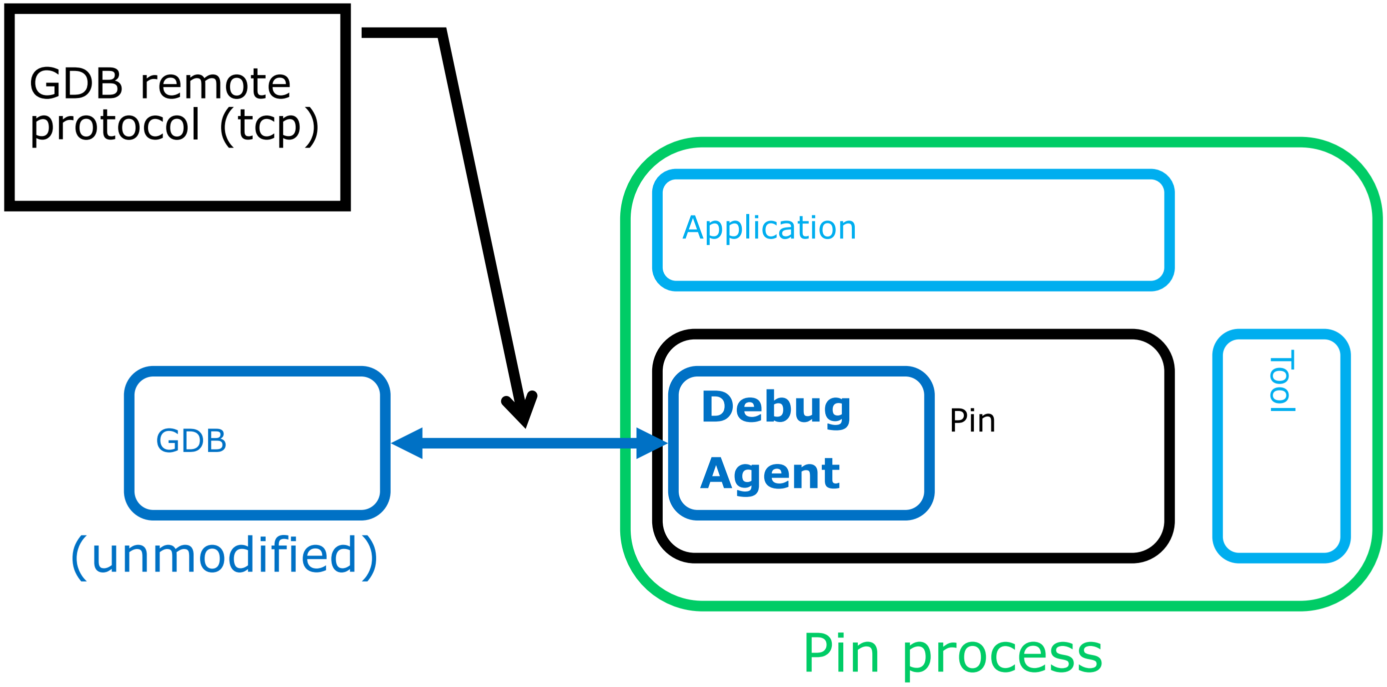
Naïve Solution Won’t Work



Why can’t we just debug normally?

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

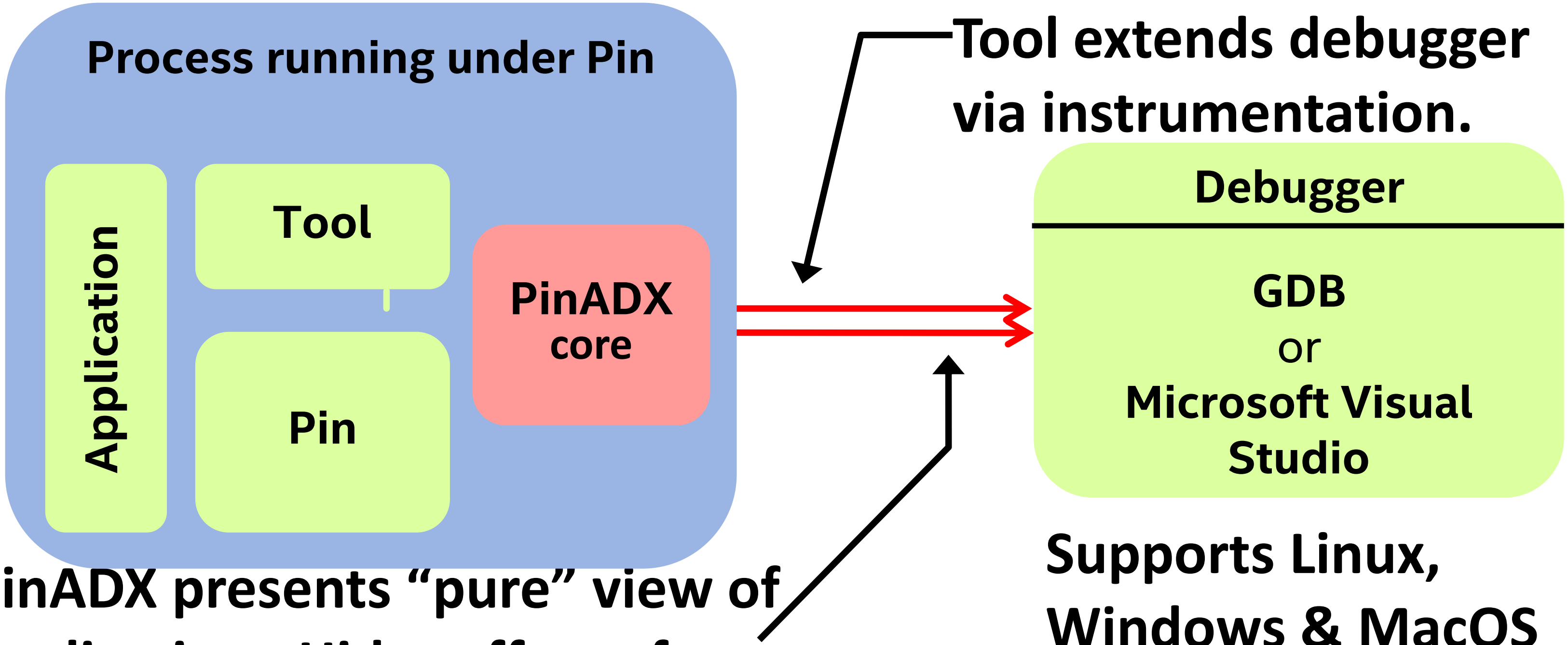
Pin Debugger Interface



GDB debugs application (not Pin itself)

Leverage GDB remote protocol API

PinADX: Pin’s Advanced Debugger Extensions



Tool extends debugger via instrumentation.

PinADX presents “pure” view of application. Hides effect of instrumentation and recompilation.

Supports Linux, Windows & MacOS

Included in Pin distributions (“debugger shell”)
Also in PinPlay kit:
extras/pinplay/examples/pinplay-debugger-shell.*

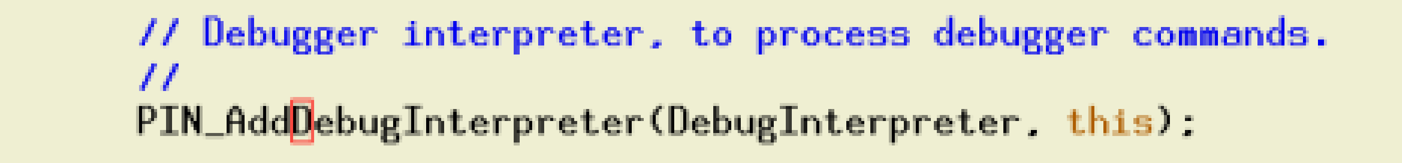
Extending the Debugger

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);
```

Pin + PinADX : Stack Debugger

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 …”

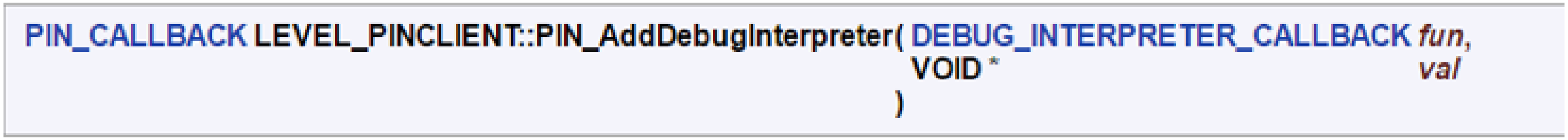
```
$ gdb ./application
(gdb) target remote :1234
```

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

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

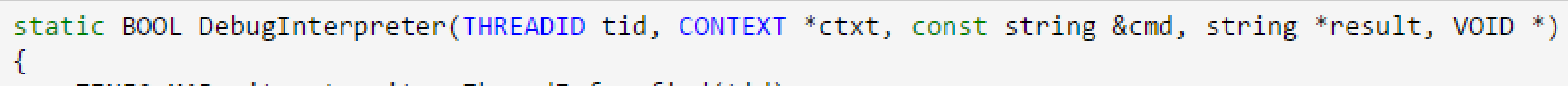
ManualExamples/stack-debugger.cpp

Relevant Pin API



```
PIN_CALLBACK LEVEL_PINCLIENT::PIN_AddDebugInterpreter( DEBUG_INTERPRETER_CALLBACK fun,
                                                       VOID *                     val
                                                     )
```

Where:



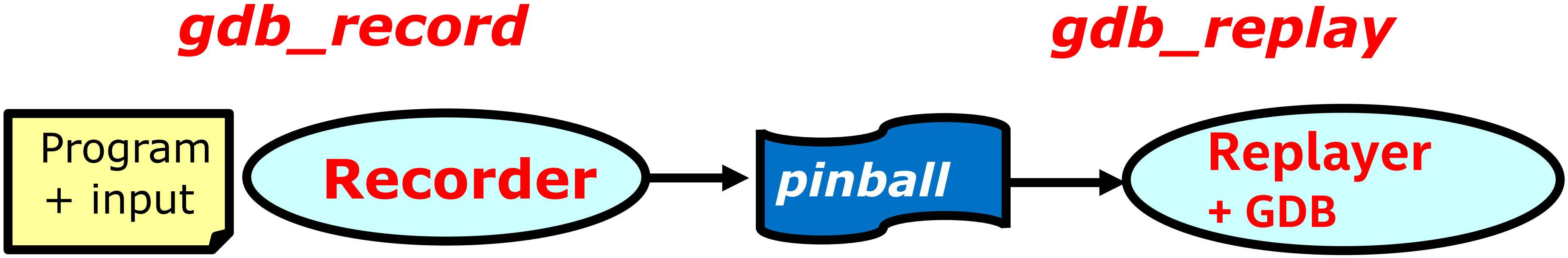
```
static BOOL DebugInterpreter(THREADID tid, CONTEXT *ctxt, const string &cmd, string *result, VOID *)
{
```

return TRUE if command is handled; FALSE otherwise

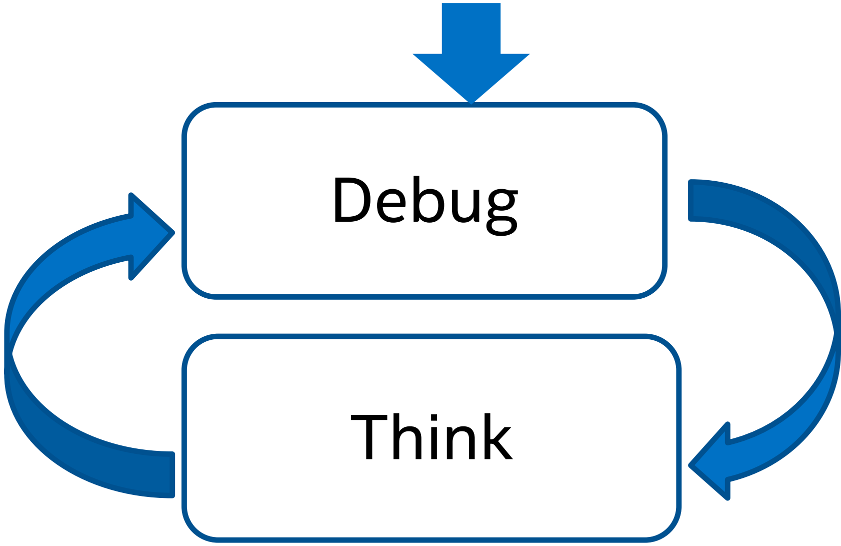
Multiple debug interpreters : called in order of registration

- Stop when TRUE is returned
- No debug interpreter handles the command : Error

PinPlay + PinADX: Reproducible Debugging with Record/Replay



Cyclic Debugging with *DrDebug*



PROVIDES

A guarantee of bug/behavior repeatability!

Bug once captured does not escape!

Inside gdb_record/gdb_replay

Python scripts : Single window interface to PinADX

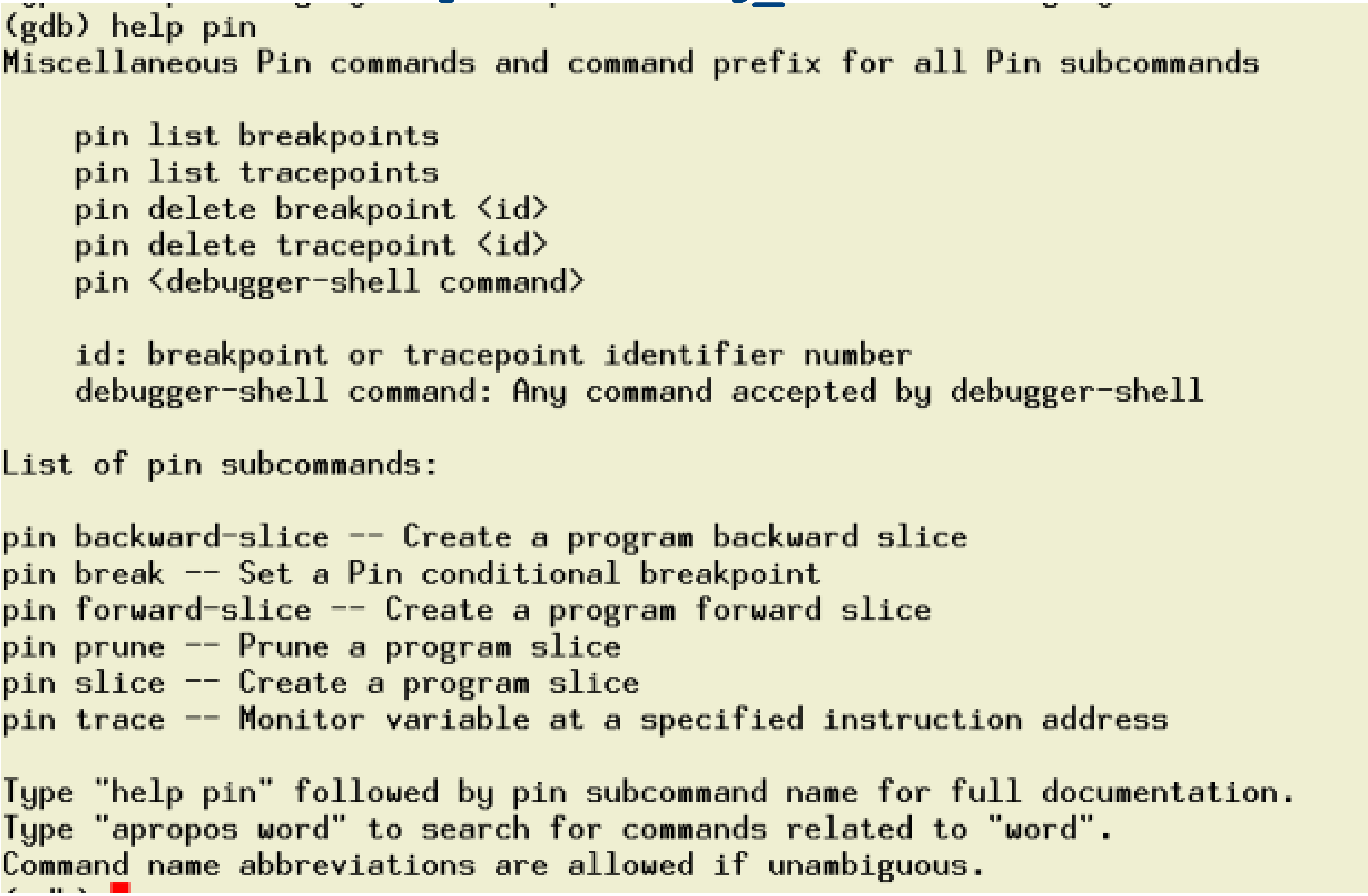
- Run "**`pin –appdebug`**" in the background
- Captures 'debug port'
- Start GDB and issue 'target remote :DEBUG_PORT'

Provide new "pin" commands to GDB using GDB-Python-extension mechanism

- **`(gdb) pin slice <high-level slice criterion>`**
  ➔ **`monitor slice <low-level slice criterion>`**

gdb_record/gdb_replay : “help pin” High-level commands supported

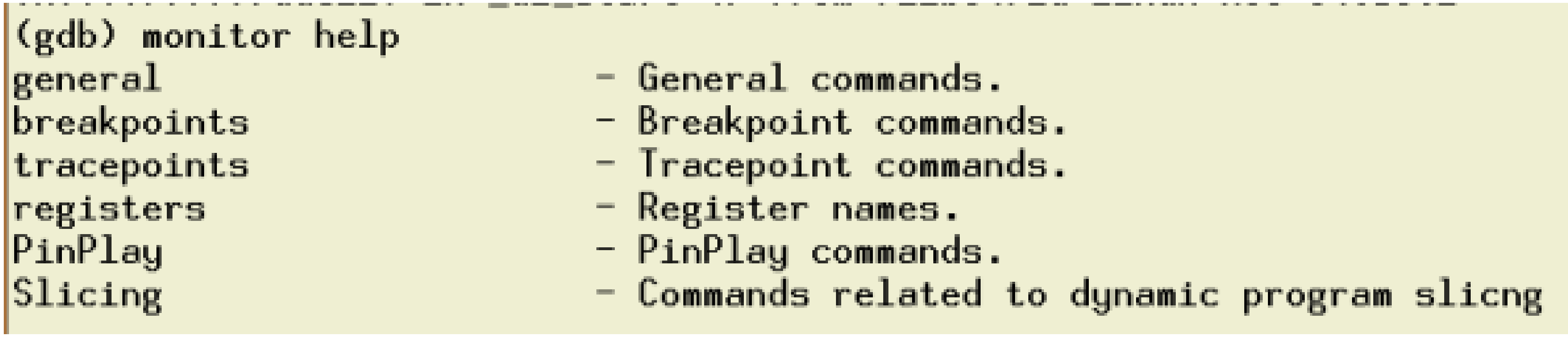
```
% $PIN_ROOT/extras/pinplay/scripts/gdb_replay
                pinball/log_0 /bin/ls
```



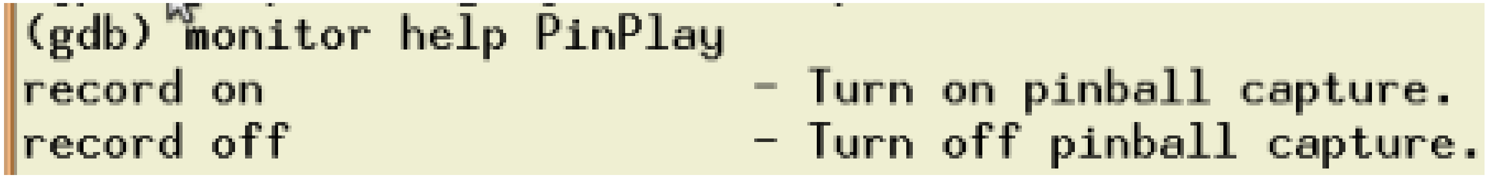
```
(gdb) help pin
Miscellaneous Pin commands and command prefix for all Pin subcommands

    pin list breakpoints
    pin list tracepoints
    pin delete breakpoint <id>
    pin delete tracepoint <id>
    pin <debugger-shell command>

    id: breakpoint or tracepoint identifier number
    debugger-shell command: Any command accepted by debugger-shell

List of pin subcommands:

pin backward-slice -- Create a program backward slice
pin break -- Set a Pin conditional breakpoint
pin forward-slice -- Create a program forward slice
pin prune -- Prune a program slice
pin slice -- Create a program slice
pin trace -- Monitor variable at a specified instruction address

Type "help pin" followed by pin subcommand name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

gdb_record/gdb_replay : “monitor help” Low-level commands supported

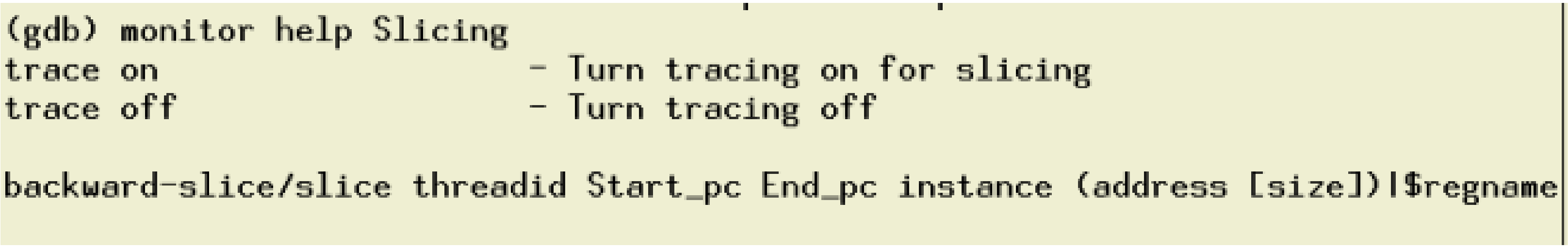
```
% $PIN_ROOT/extras/pinplay/scripts/gdb_replay
pinball/log_0 /bin/ls
```



```
(gdb) monitor help
general                       - General commands.
breakpoints                   - Breakpoint commands.
tracepoints                   - Tracepoint commands.
registers                     - Register names.
PinPlay                       - PinPlay commands.
Slicing                       - Commands related to dynamic program slicng
```



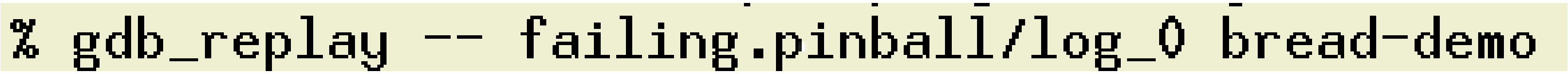
```
(gdb) monitor help PinPlay
record on                     - Turn on pinball capture.
record off                    - Turn off pinball capture.
```



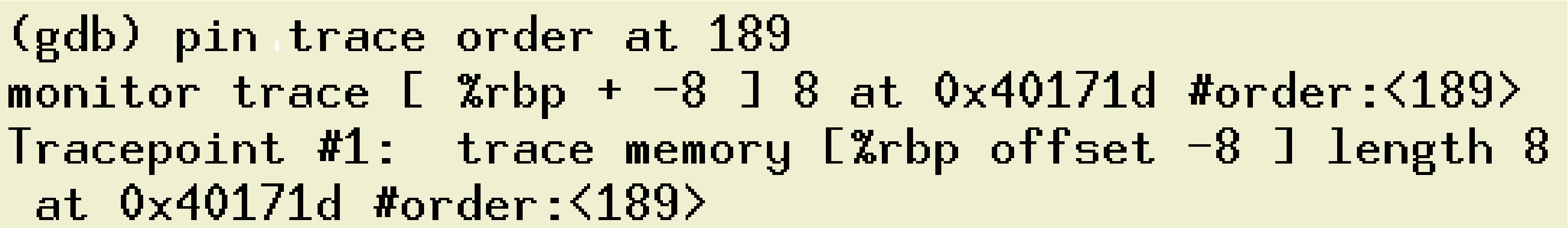
```
(gdb) monitor help Slicing
trace on                      - Turn tracing on for slicing
trace off                     - Turn tracing off

backward-slice/slice threadid Start_pc End_pc instance (address [size])|$regname
```

Printf debugging without printf()'s

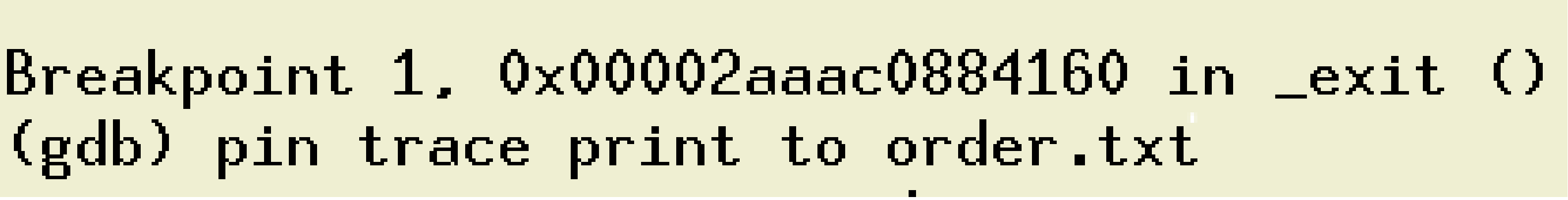
- Use Pin to instrument points of interest and print values
- No source changes/re-compilation necessary
- Debugging with replay ➔ Values do not change across sessions



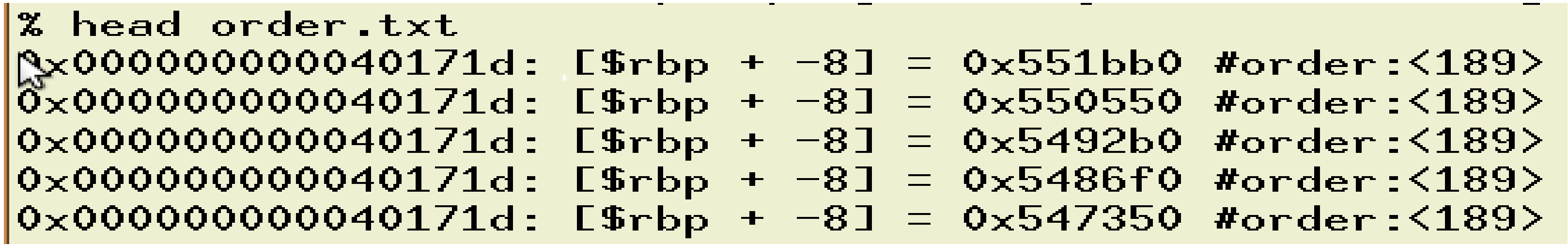
```
% gdb_replay -- failing.pinball/log_0 bread-demo
```



```
(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>
```



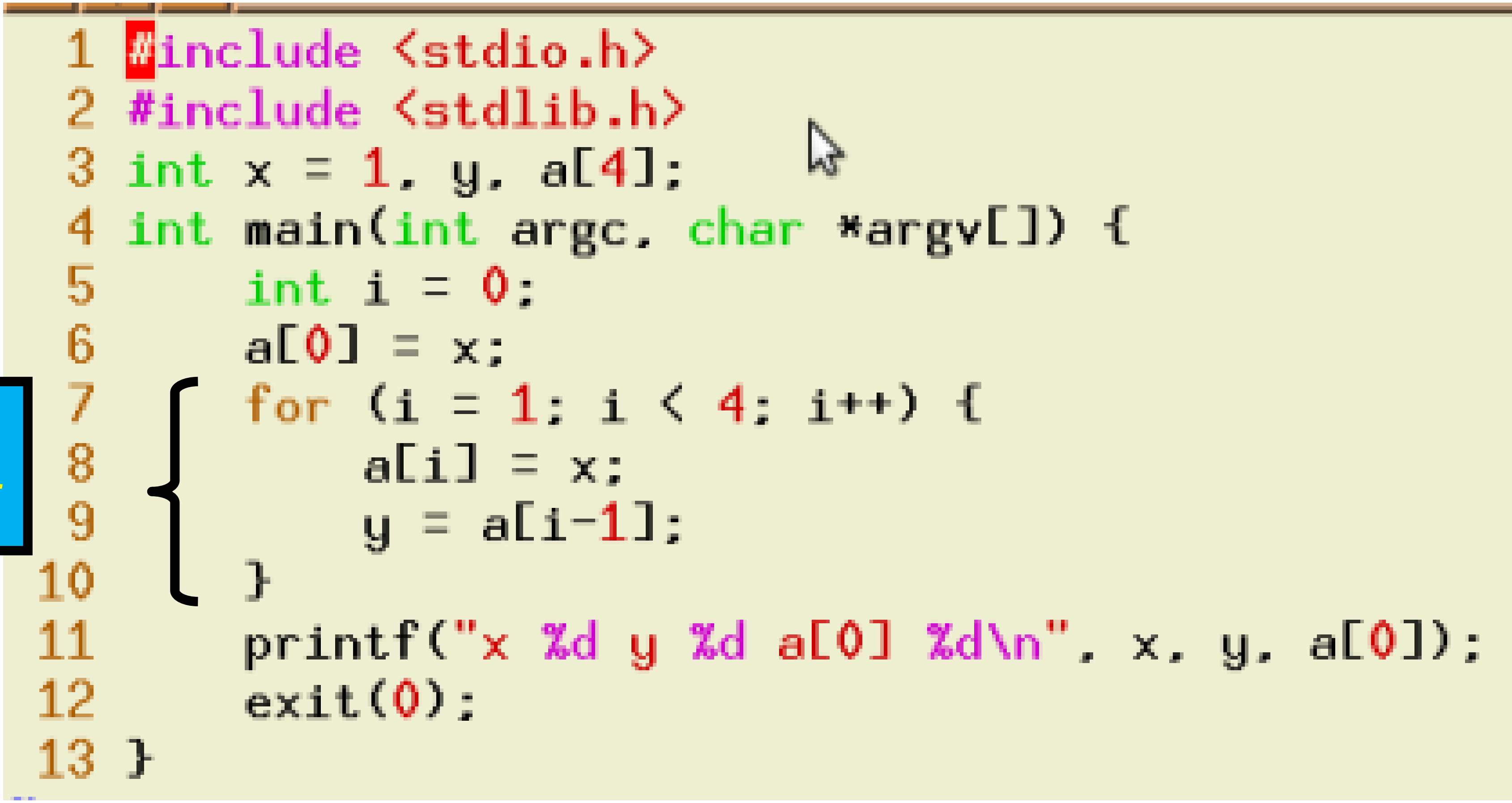
```
Breakpoint 1, 0x00002aaac0884160 in _exit ()
(gdb) pin trace print to order.txt
```



```
% 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>
```

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

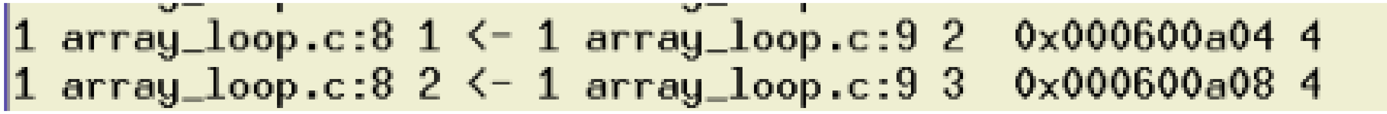
Recall: array_loop.c



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int x = 1, y, a[4];
 4 int main(int argc, char *argv[]) {
 5       int i = 0;
 6       a[0] = x;
 7       for (i = 1; i < 4; i++) {
 8             a[i] = x;
 9             y = a[i-1];
10       }
11       printf("x %d y %d a[0] %d\n", x, y, a[0]);
12       exit(0);
13 }
```

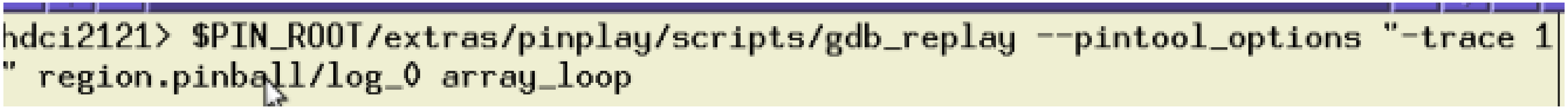
Loop of Interest

Slicing Results (excerpt)

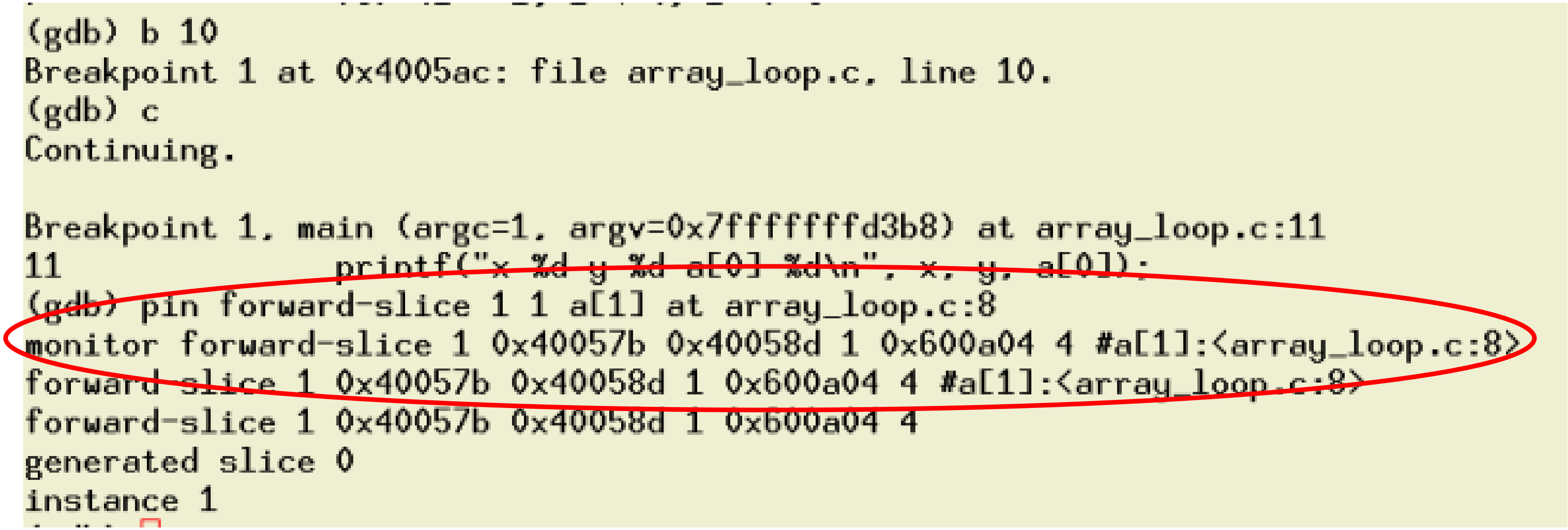


```
1 array_loop.c:8 1 <- 1 array_loop.c:9 2  0x000600a04 4
1 array_loop.c:8 2 <- 1 array_loop.c:9 3  0x000600a08 4
```

Interactive slicing : array_loop region pinball

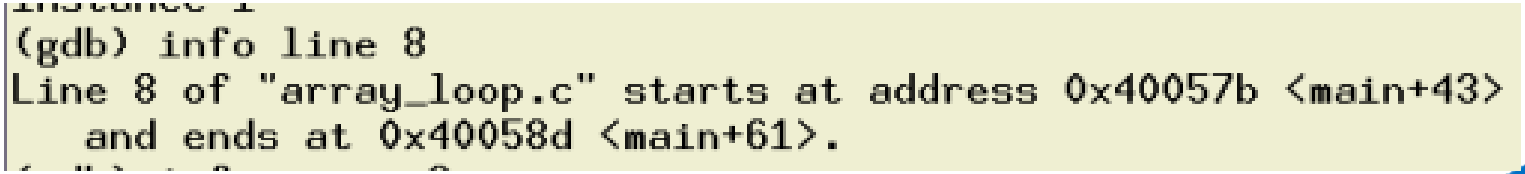


```
hdci2121> $PIN_ROOT/extras/pinplay/scripts/gdb_replay --pintool_options "-trace 1
" region.pinball/log_0 array_loop
```



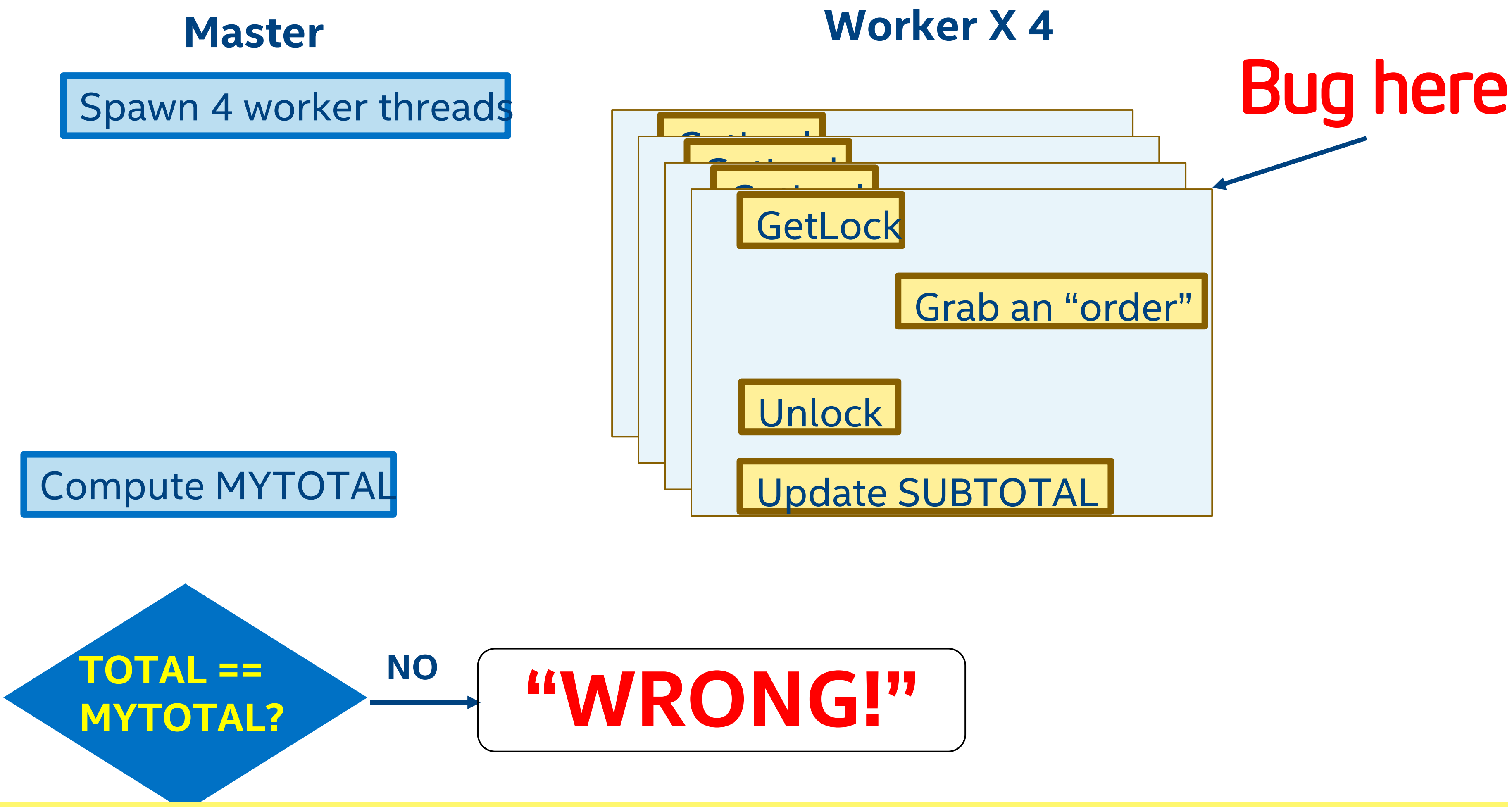
```
(gdb) b 10
Breakpoint 1 at 0x4005ac: file array_loop.c, line 10.
(gdb) c
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fffffffd3b8) at array_loop.c:11
11              printf("x %d y %d a[0] %d\n", x, y, a[0]);
(gdb) pin forward-slice 1 1 a[1] at array_loop.c:8
monitor forward-slice 1 0x40057b 0x40058d 1 0x600a04 4 #a[1]:<array_loop.c:8>
forward-slice 1 0x40057b 0x40058d 1 0x600a04 4 #a[1]:<array_loop.c:8>
forward-slice 1 0x40057b 0x40058d 1 0x600a04 4
generated slice 0
instance 1
```

Translation to low-level command uses GDB commands:



```
(gdb) info line 8
Line 8 of "array_loop.c" starts at address 0x40057b <main+43>
   and ends at 0x40058d <main+61>.
```

DEMO: A self-checking multi-threaded program, with a bug



Master

Spawn 4 worker threads

Compute MYTOTAL

TOTAL == MYTOTAL?

NO

“WRONG!”

Worker X 4

GetLock

Grab an “order”

Unlock

Update SUBTOTAL

Bug here

Problem: Shifting/disappearing bug!

<Demo Video Clip(s) Available Separately>

Debugging a multi-threaded bug with DrDebug: Takeaways

- ❑ We can do multiple `gdb_replay` sessions and gather more information each time
- ❑ This is feasible because
  1. the same buggy schedule is reproduced in each session and
  2. pointer values remain the same across sessions
- ❑ That is the power of replay debugging – **bug once captured never escapes!**

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*

Key Takeaways: 3 mantras:

1. 'Attach'ment is good
2. Be selective
3. Practice exclusion

References

Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation; Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. *PLDI 2005*. **Most influential paper of PLDI2005 : Awarded PLDI2015. www.pintool.org**

PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs; Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, James Cownie. CGO 2010. **CGO 2010 Best Paper Award Winner. www.pinplay.org**

PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation; Gregory Lueck, Harish Patil, and Cristiano Pereira, A. CGO 2012. **Nominated for CGO 2012 Best Paper Award.**

DrDebug/Slicing: Deterministic Replay based Cyclic Debugging with Dynamic Slicing; Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, Iulian Neamtiu. CGO 2014. **www.drdebug.org**

Pinballs: Portable and Shareable User-level Checkpoints for Reproducible Analysis and Simulation; Harish Patil and Trevor Carlson. REPRODUCE : Workshop on Reproducible Research Methodologies HPCA 2014.

DCFG: Graph-matching-based simulation-region selection for multiple binaries; Yount, C., Patil, H. Islam, M.S., Srikanth, A.. Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on , March 2015.

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