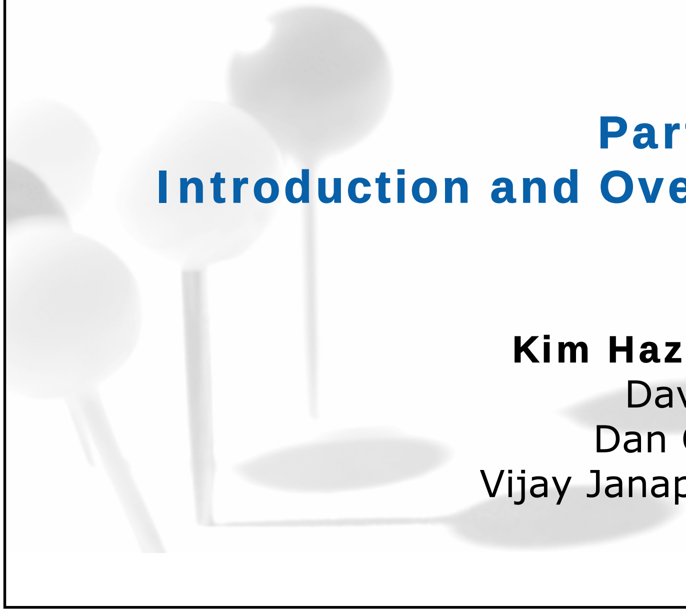
Using Pin for Compiler and Computer Architecture Research and Education

PLDI Tutorial
Sunday, June 10, 2007

Kim Hazelwood
David Kaeli
Dan Connors
Vijay Janapa Reddi



Part One: Introduction and Overview

Kim Hazelwood
David Kaeli
Dan Connors
Vijay Janapa Reddi

What is Instrumentation?

A technique that inserts extra code into a program to collect runtime information

Instrumentation approaches:

- Source instrumentation:
  - Instrument source programs
- **Binary instrumentation**:
  - Instrument executables directly

Why use Dynamic Instrumentation?

- ✓ No need to recompile or relink
- ✓ Discover code at runtime
- ✓ Handle dynamically-generated code
- ✓ Attach to running processes

How is Instrumentation used in Compiler Research?

Program analysis

- Code coverage
- Call-graph generation
- Memory-leak detection
- Instruction profiling

Thread analysis

- Thread profiling
- Race detection

How is Instrumentation used in Computer Architecture Research?

- Trace Generation
- Branch Predictor and Cache Modeling
- Fault Tolerance Studies
- Emulating Speculation
- Emulating New Instructions

Advantages of Pin Instrumentation

Easy-to-use Instrumentation:

- Uses dynamic instrumentation
  - Do not need source code, recompilation, post-linking

Programmable Instrumentation:

- Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)

Multiplatform:

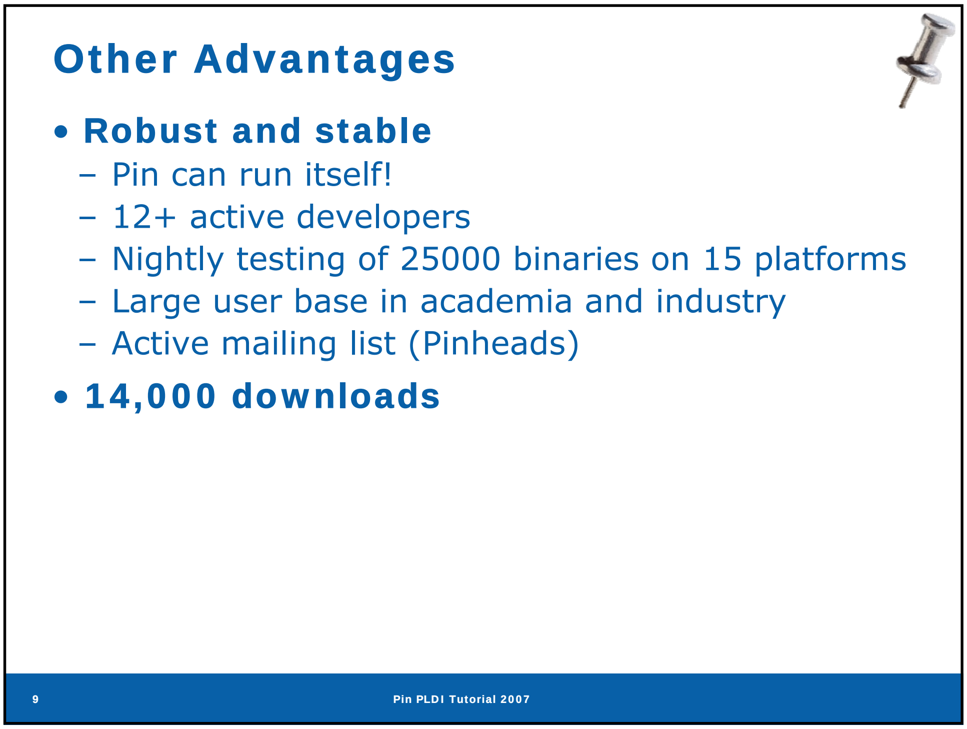
- Supports x86, x86-64, Itanium, Xscale
- Supports Linux, Windows, MacOS

Robust:

- Instruments real-life applications: Database, web browsers, …
- Instruments multithreaded applications
- Supports signals

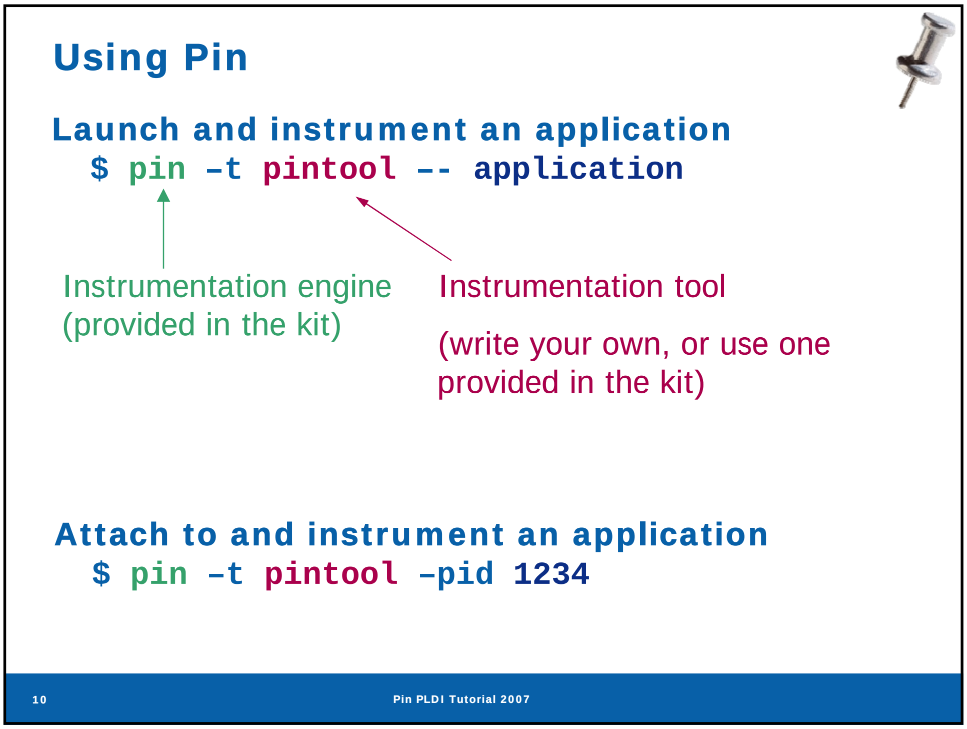
Efficient:

- Applies compiler optimizations on instrumentation code



Other Advantages

- **Robust and stable**
  - Pin can run itself!
  - 12+ active developers
  - Nightly testing of 25000 binaries on 15 platforms
  - Large user base in academia and industry
  - Active mailing list (Pinheads)
- **14,000 downloads**



Using Pin

Launch and instrument an application

```
$ pin –t pintool –- application
```

Instrumentation engine (provided in the kit)

Instrumentation tool

(write your own, or use one provided in the kit)

Attach to and instrument an application

```
$ pin –t pintool –pid 1234
```

Pin Instrumentation APIs

Basic APIs are architecture independent:

- Provide common functionalities like determining:
  - Control-flow changes
  - Memory accesses

Architecture-specific APIs

- e.g., Info about segmentation registers on IA32

Call-based APIs:

- Instrumentation routines
- Analysis routines

Instrumentation vs. Analysis

Concepts borrowed from the ATOM tool:

Instrumentation routines define where instrumentation is **inserted**

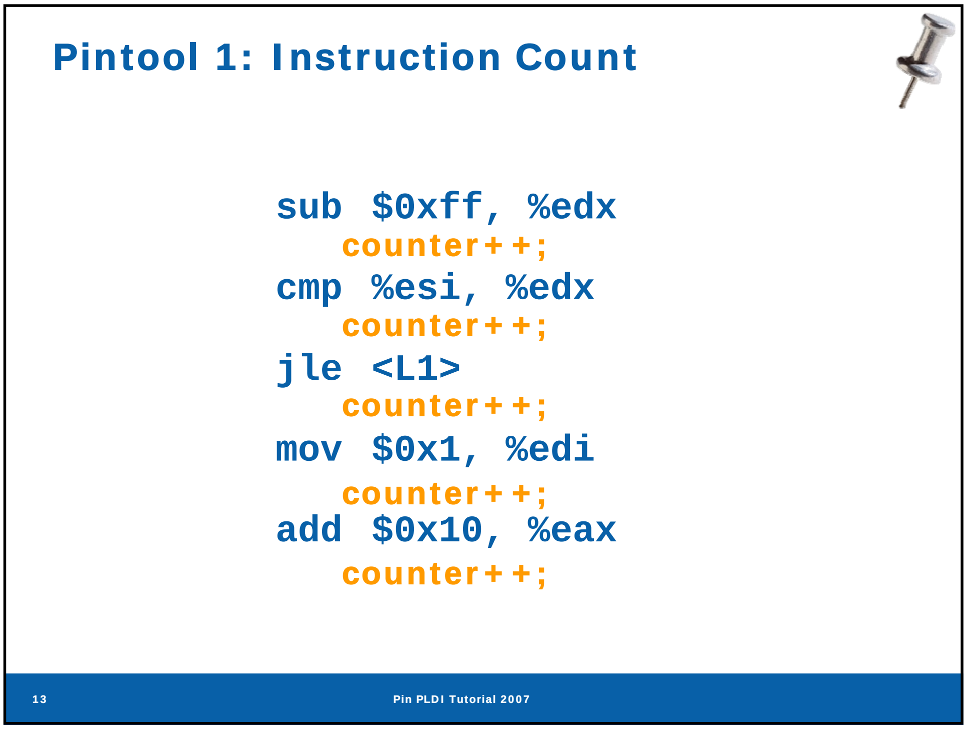
- e.g., before instruction

☞ **Occurs *first time* an instruction is executed**

Analysis routines define what to do when instrumentation is **activated**

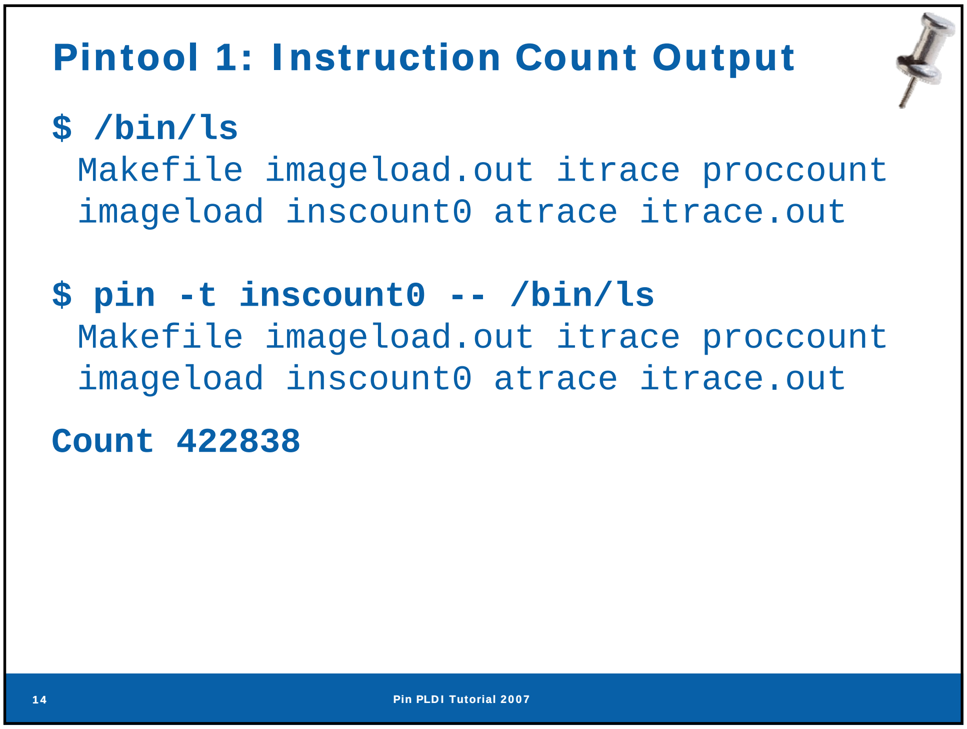
- e.g., increment counter

☞ **Occurs *every time* an instruction is executed**



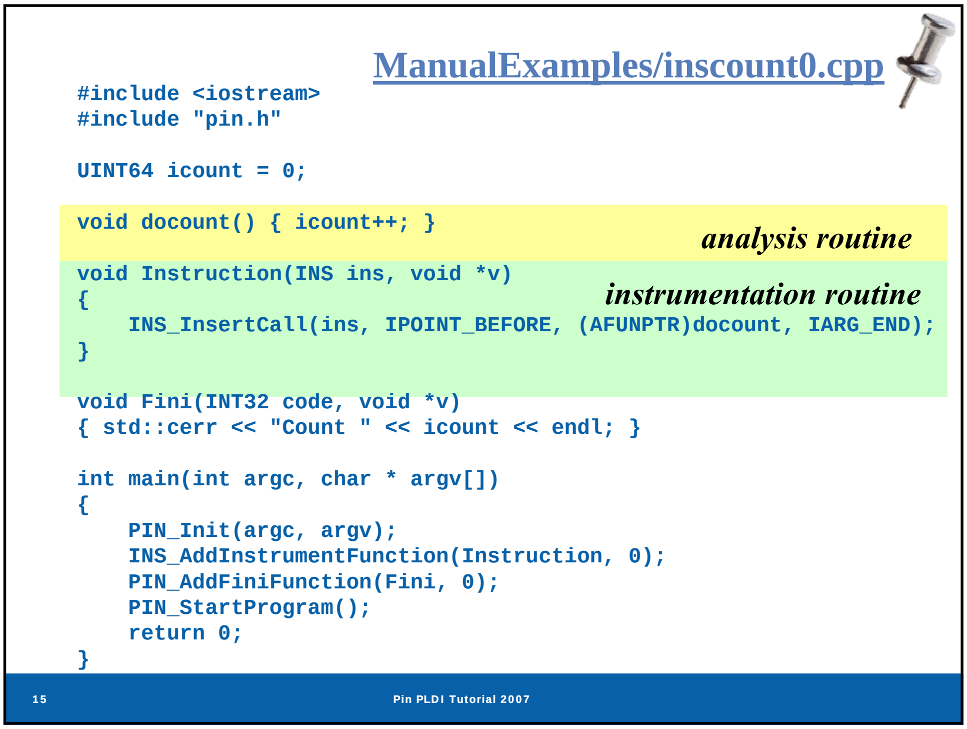
Pintool 1: Instruction Count

```
sub  $0xff, %edx
   counter++;
cmp  %esi, %edx
   counter++;
jle  <L1>
   counter++;
mov  $0x1, %edi
   counter++;
add  $0x10, %eax
   counter++;
```



Pintool 1: Instruction Count Output

```
$ /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out

$ pin -t inscount0 -- /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out

Count 422838
```

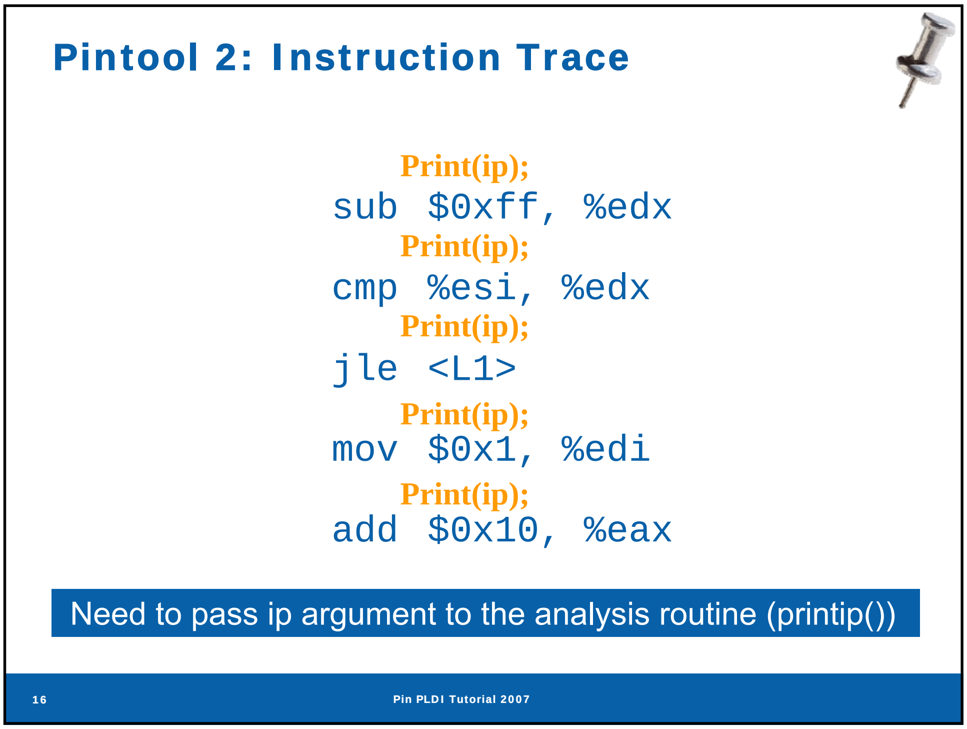


ManualExamples/inscount0.cpp

```
#include <iostream>
#include "pin.h"

UINT64 icount = 0;

void docount() { icount++; }

void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

analysis routine

instrumentation routine



Pintool 2: Instruction Trace

```
      Print(ip);
sub  $0xff, %edx
      Print(ip);
cmp  %esi, %edx
      Print(ip);
jle  <L1>
      Print(ip);
mov  $0x1, %edi
      Print(ip);
add  $0x10, %eax
```

Need to pass ip argument to the analysis routine (printip())

Pintool 2: Instruction Trace Output

```
$ pin -t itrace -- /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out

$ head -4 itrace.out
 0x40001e90
 0x40001e91
 0x40001ee4
 0x40001ee5
```

ManualExamples/itrace.cpp

argument to analysis routine

```
#include <stdio.h>
#include "pin.H"
FILE * trace;
void printip(void *ip) { fprintf(trace, "%p\n", ip); }
```

analysis routine

instrumentation routine

```
void Instruction(INS ins, void *v) {
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip,
                                IARG_INST_PTR, IARG_END);
}
void Fini(INT32 code, void *v) { fclose(trace); }
int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Examples of Arguments to Analysis Routine

`IARG_INST_PTR`

- Instruction pointer (program counter) value

`IARG_UINT32 <value>`

- An integer value

`IARG_REG_VALUE <register name>`

- Value of the register specified

`IARG_BRANCH_TARGET_ADDR`

- Target address of the branch instrumented

`IARG_MEMORY_READ_EA`

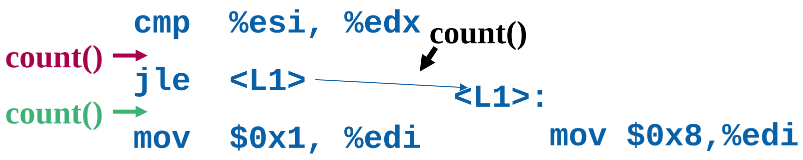
- Effective address of a memory read

And many more … (refer to the Pin manual for details)

Instrumentation Points

Instrument points relative to an instruction:

- *Before (IPOINT_BEFORE)*
- After:
  - Fall-through edge (IPOINT_AFTER)
  - Taken edge (IPOINT_TAKEN_BRANCH)



```
cmp  %esi, %edx
jle  <L1>
mov  $0x1, %edi
```

```
<L1>:
mov $0x8,%edi
```

Instrumentation Granularity

Instrumentation can be done at three different granularities:

- Instruction
- Basic block
  - A sequence of instructions terminated at a control-flow changing instruction
  - Single entry, single exit
- Trace
  - A sequence of basic blocks terminated at an unconditional control-flow changing instruction
  - Single entry, multiple exits

```
sub   $0xff, %edx
cmp   %esi, %edx
jle   <L1>
```

```
mov   $0x1, %edi
add   $0x10, %eax
jmp   <L2>
```

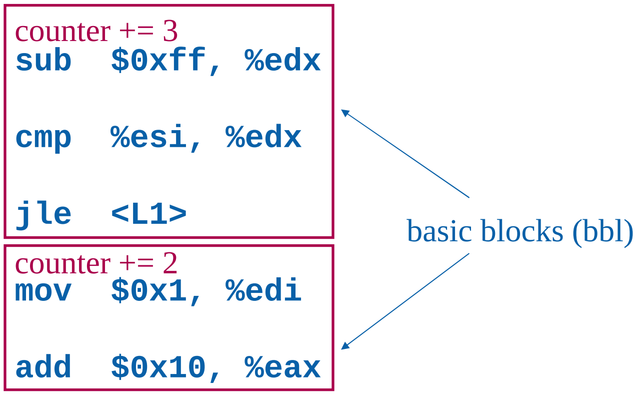
1 Trace, 2 BBs, 6 insts

Recap of Pintool 1: Instruction Count

```
counter++;
sub  $0xff, %edx
counter++;
cmp  %esi, %edx
counter++;
jle  <L1>
counter++;
mov  $0x1, %edi
counter++;
add  $0x10, %eax
```

Straightforward, but the counting can be more efficient

Pintool 3: Faster Instruction Count


counter += 3

```
sub  $0xff, %edx

cmp  %esi, %edx

jle  <L1>
```

counter += 2

```
mov  $0x1, %edi

add  $0x10, %eax
```

basic blocks (bbl)

ManualExamples/inscount1.cpp

```
#include <stdio.h>
#include "pin.H“
UINT64 icount = 0;
void docount(INT32 c) { icount += c; }
void Trace(TRACE trace, void *v) {
  for (BBL bbl = TRACE_BblHead(trace);
    BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
    BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
          IARG_UINT32, BBL_NumIns(bbl), IARG_END);
  }
}
void Fini(INT32 code, void *v) {
  fprintf(stderr, "Count %lld\n", icount);
}
int main(int argc, char * argv[]) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

analysis routine

instrumentation routine

Modifying Program Behavior

Pin allows you not only to observe but also change program behavior

Ways to change program behavior:

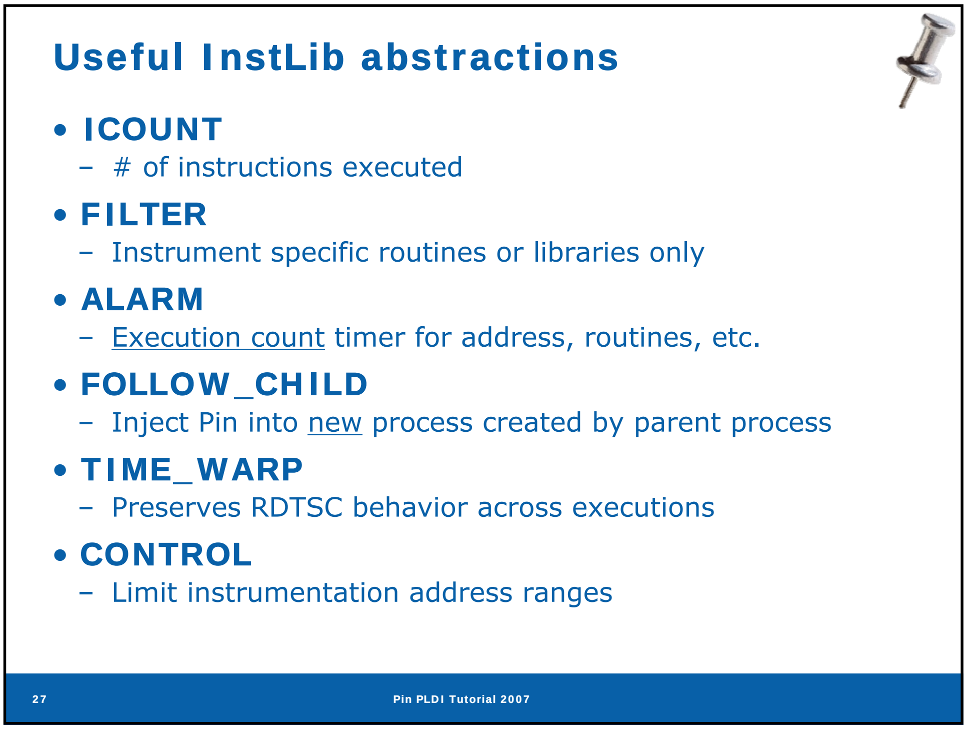
- Add/delete instructions
- Change register values
- Change memory values
- Change control flow

Instrumentation Library

```
#include <iostream>
#include "pin.H"

UINT64 icount = 0;

VOID Fini(INT32 code, VOID *v) {
  std::cerr << "Count " << icount <
}
VOID docount() {
  icount++;
}
VOID Instruction(INS ins, VOID *v)
  INS_InsertCall(ins, IPOINT_BEFORE
}
int main(int argc, char * argv[]) {
  PIN_Init(argc, argv);
  INS_AddInstrumentFunction(Instruc
  PIN_AddFiniFunction(Fini, 0);
  PIN_StartProgram();
  return 0;
}
```

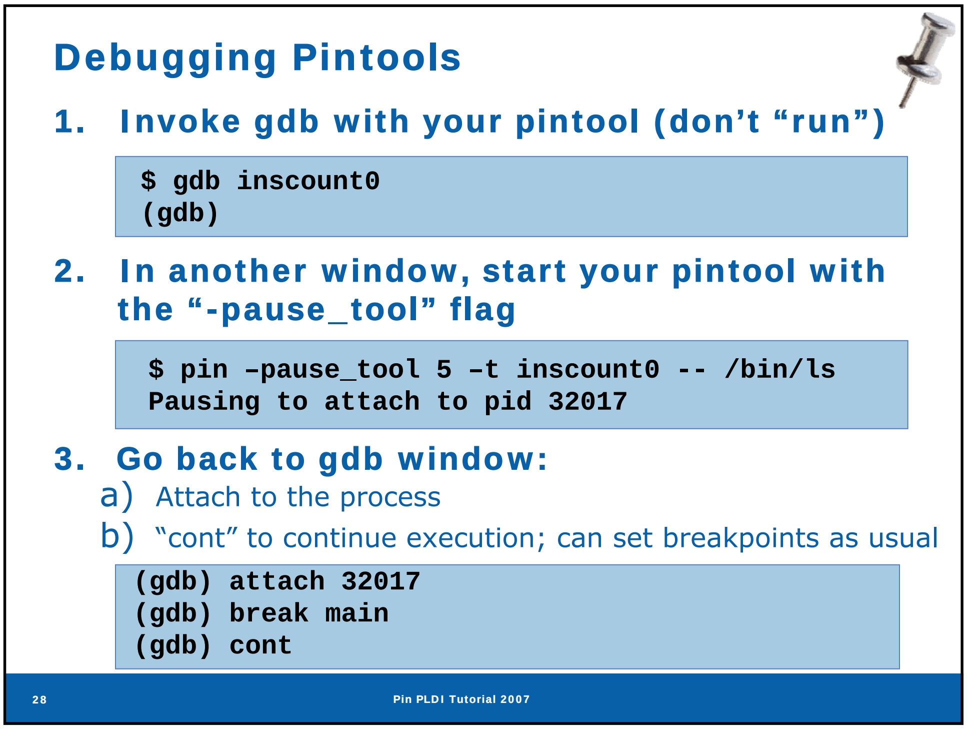
Instruction counting Pin Tool

```
#include <iostream>
#include "pin.H"
#include "instlib.H"

INSTLIB::ICOUNT icount;

VOID Fini(INT32 code, VOID *v) {
   cout << "Count" << icount.Count() << endl;
}

int main(int argc, char * argv[]) {
   PIN_Init(argc, argv);
   PIN_AddFiniFunction(Fini, 0);
   icount.Activate();
   PIN_StartProgram();
   return 0;
}
```



Useful InstLib abstractions

- **ICOUNT**
  - # of instructions executed
- **FILTER**
  - Instrument specific routines or libraries only
- **ALARM**
  - Execution count timer for address, routines, etc.
- **FOLLOW_CHILD**
  - Inject Pin into new process created by parent process
- **TIME_WARP**
  - Preserves RDTSC behavior across executions
- **CONTROL**
  - Limit instrumentation address ranges



Debugging Pintools

1. **Invoke gdb with your pintool (don’t “run”)**

```
$ gdb inscount0
(gdb)
```

2. **In another window, start your pintool with the “-pause_tool” flag**

```
$ pin –pause_tool 5 –t inscount0 -- /bin/ls
Pausing to attach to pid 32017
```

3. **Go back to gdb window:**
   a) Attach to the process
   b) “cont” to continue execution; can set breakpoints as usual

```
(gdb) attach 32017
(gdb) break main
(gdb) cont
```

Pin Internals

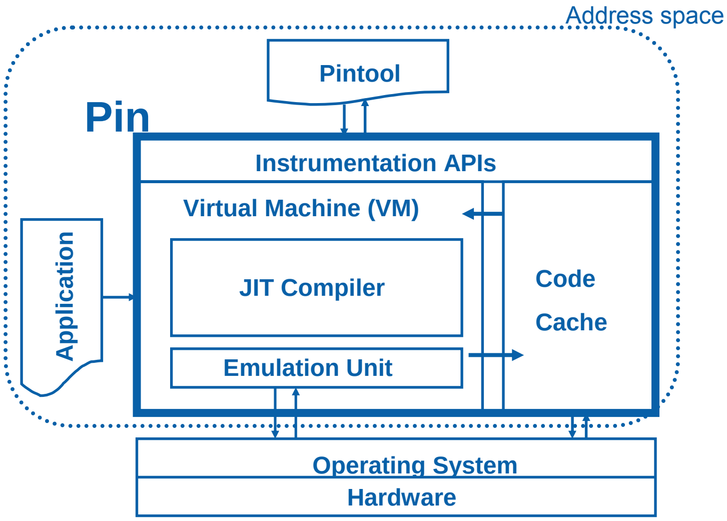
Pin Source Code Organization

Pin source organized into generic, architecture-dependent, OS-dependent modules:

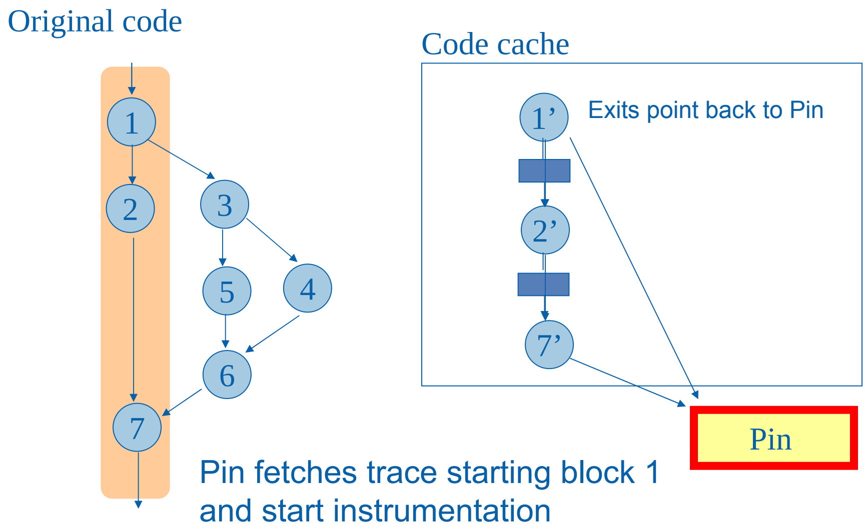
Architecture	#source files	#source lines
Generic	87 (48%)	53595 (47%)
x86 (32-bit+ 64-bit)	34 (19%)	22794 (20%)
Itanium	34 (19%)	20474 (18%)
ARM	27 (14%)	17933 (15%)
TOTAL	182 (100%)	114796 (100%)

☞ *~50% code shared among architectures*

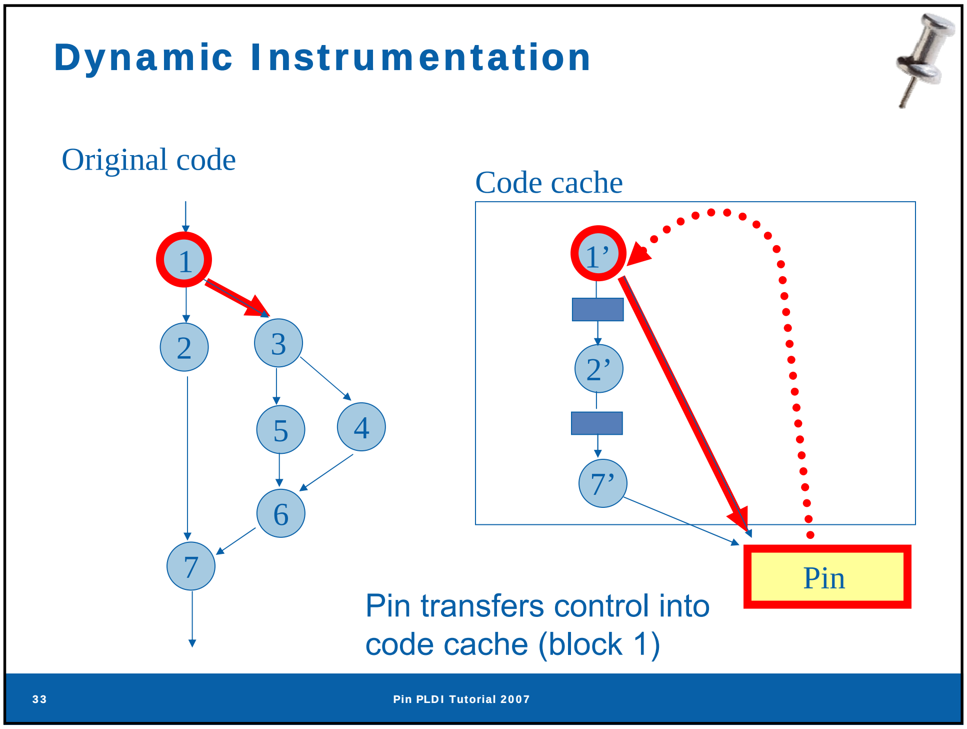
Pin’s Software Architecture



Dynamic Instrumentation

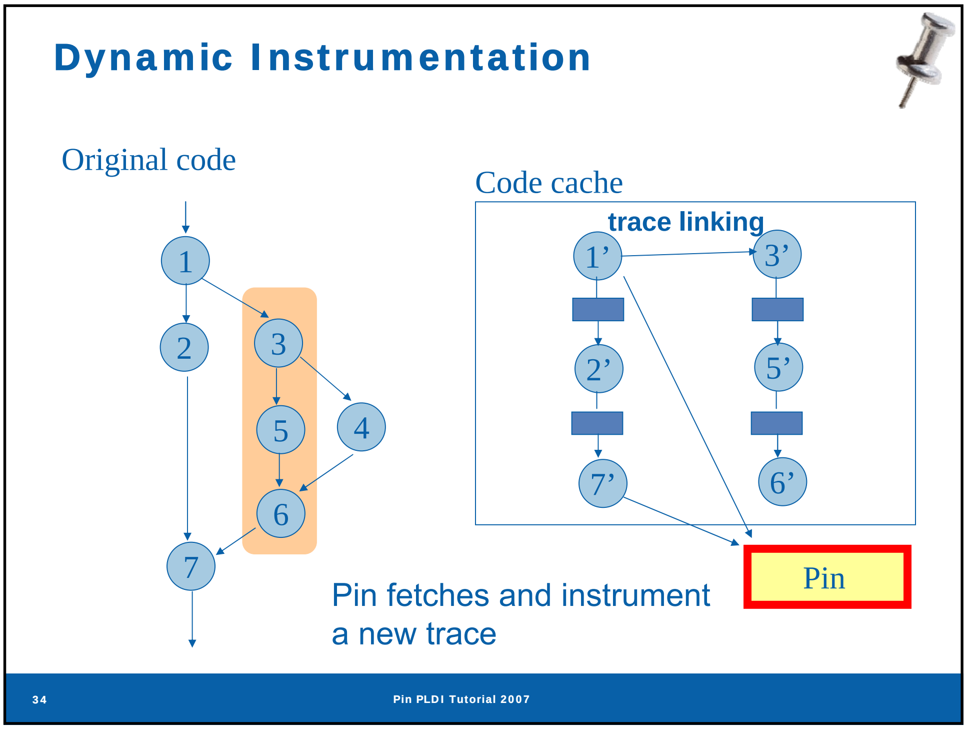


Pin fetches trace starting block 1 and start instrumentation



Dynamic Instrumentation

Pin transfers control into code cache (block 1)



Dynamic Instrumentation

Pin fetches and instrument a new trace

Implementation Challenges

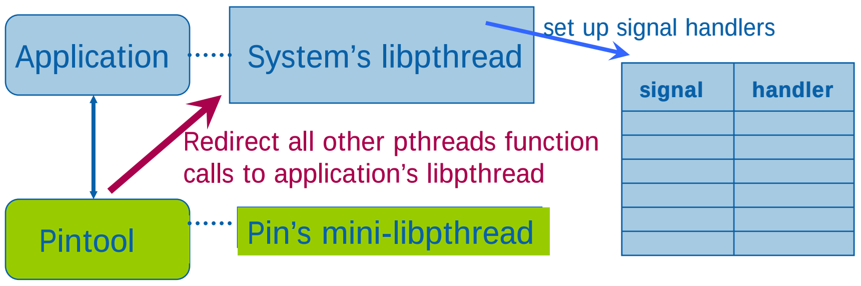
- **Linking**
  - Straightforward for direct branches
  - Tricky for indirects, invalidations
- **Re-allocating registers**
- **Maintaining transparency**
- **Self-modifying code**

- **Supporting MT applications…**

Pin's Multithreading Support

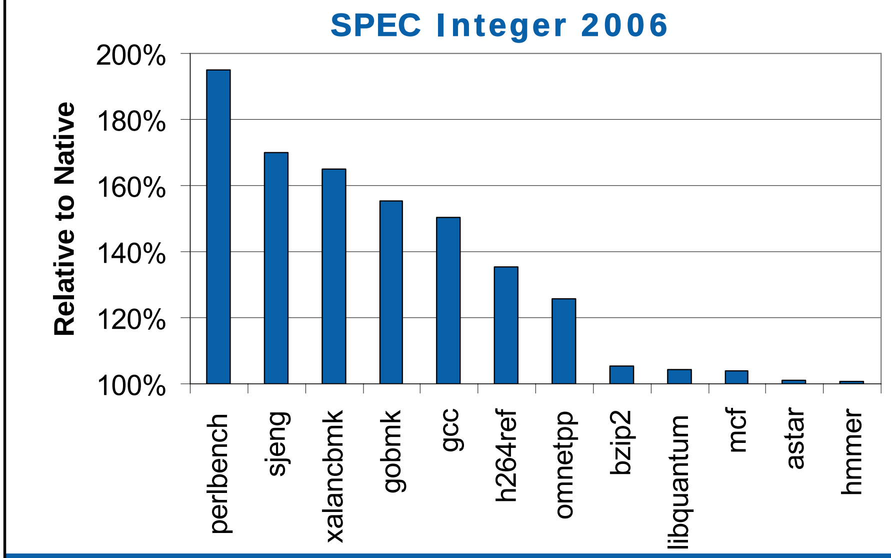
Thread-safe accesses Pin, Pintool, and App
- Pin: One thread in the VM at a time
- Pintool: Locks, ThreadID, event notification
- App: Thread-local spill area

Providing pthreads functions to instrumentation tools

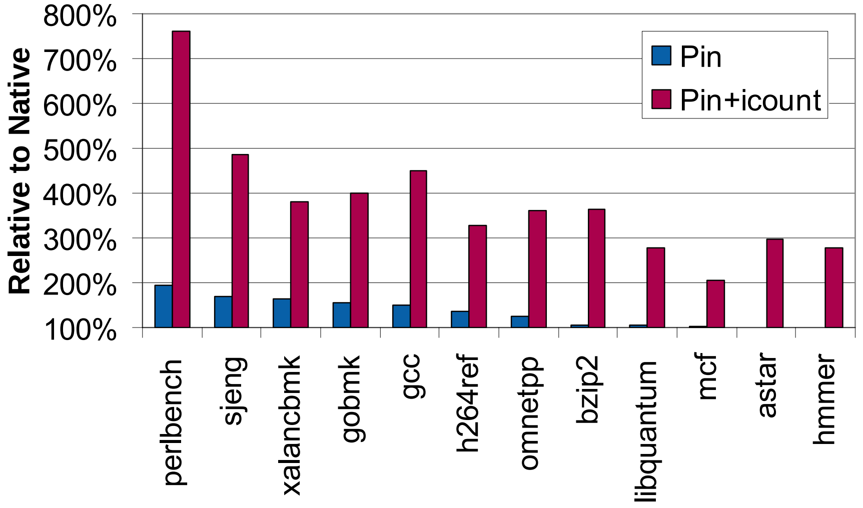


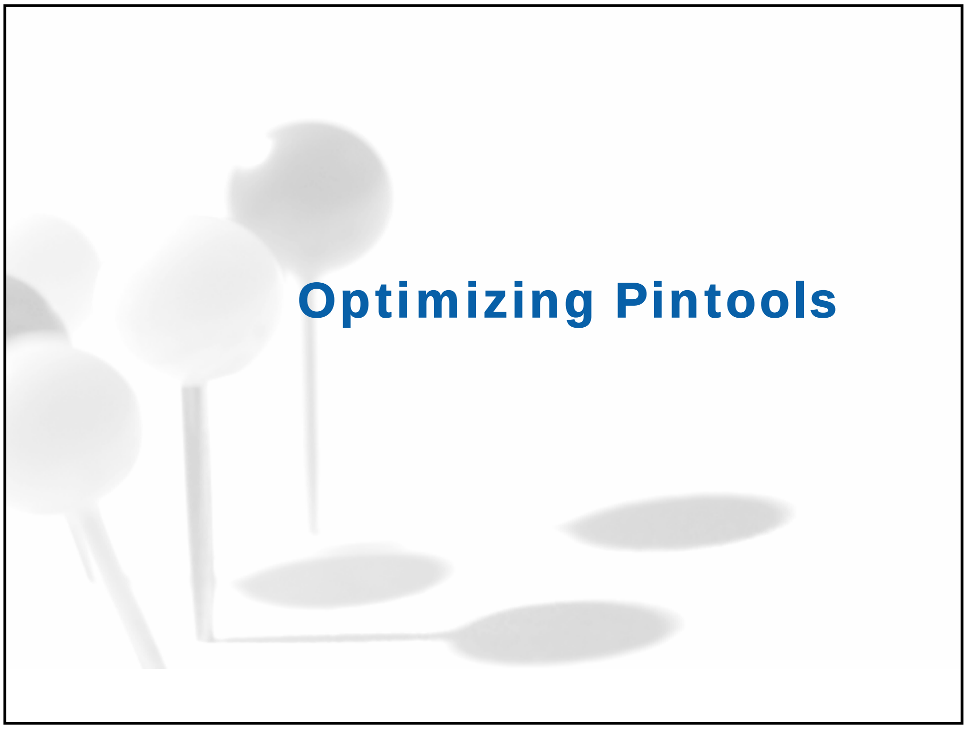
signal	handler

Pin Overhead

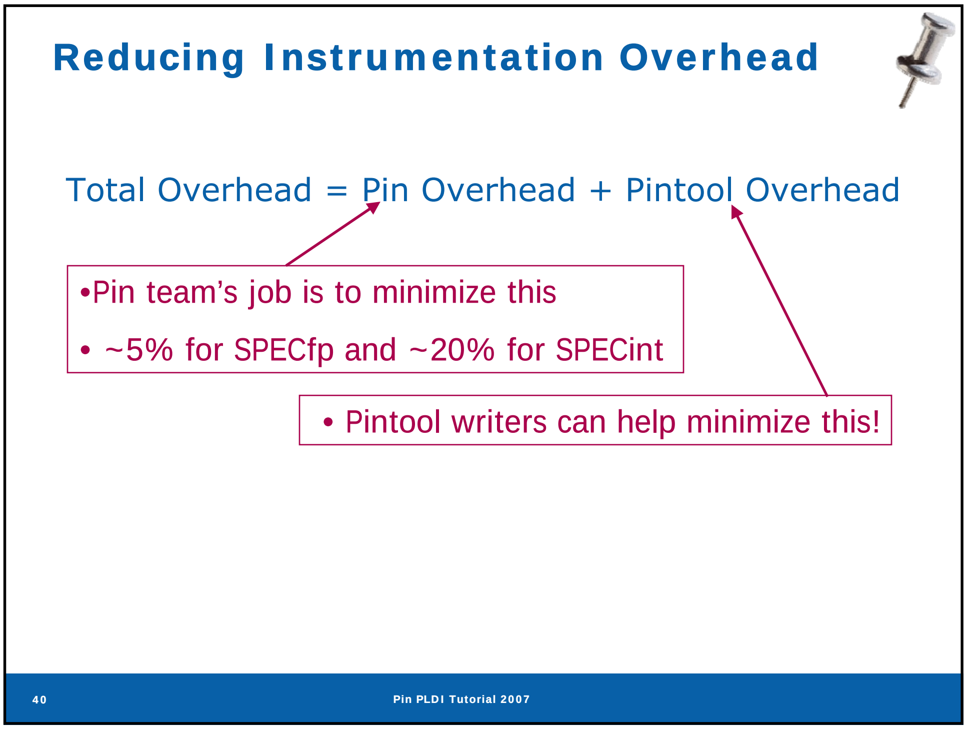


Adding User Instrumentation





Optimizing Pintools



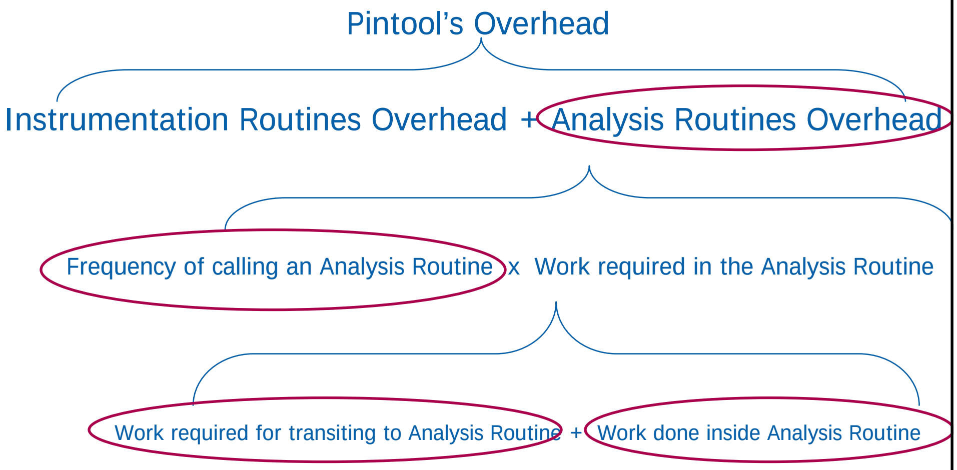
Reducing Instrumentation Overhead

Total Overhead = Pin Overhead + Pintool Overhead

- Pin team's job is to minimize this
- ~5% for SPECfp and ~20% for SPECint

- Pintool writers can help minimize this!

Reducing the Pintool’s Overhead



Pintool’s Overhead

$$\underbrace{\text{Instrumentation Routines Overhead} + \text{Analysis Routines Overhead}}_{\text{Pintool's Overhead}}$$

$$\underbrace{\text{Frequency of calling an Analysis Routine} \times \text{Work required in the Analysis Routine}}_{\text{Analysis Routines Overhead}}$$

$$\underbrace{\text{Work required for transiting to Analysis Routine} + \text{Work done inside Analysis Routine}}_{\text{Work required in the Analysis Routine}}$$

Analysis Routines: Reduce Call Frequency

Key: Instrument at the largest granularity whenever possible

Trace > Basic Block > Instruction

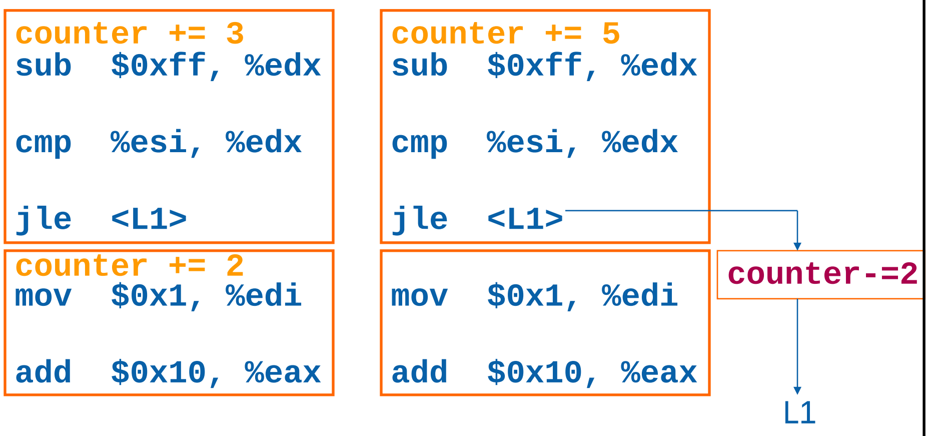
Slower Instruction Counting

```
counter++;
sub  $0xff, %edx
counter++;
cmp  %esi, %edx
counter++;
jle  <L1>
counter++;
mov  $0x1, %edi
counter++;
add  $0x10, %eax
```

Faster Instruction Counting

Counting at BBL level

Counting at Trace level



Counting at BBL level:

```
counter += 3
sub  $0xff, %edx

cmp  %esi, %edx

jle  <L1>
counter += 2
mov  $0x1, %edi

add  $0x10, %eax
```

Counting at Trace level:

```
counter += 5
sub  $0xff, %edx

cmp  %esi, %edx

jle  <L1>

mov  $0x1, %edi

add  $0x10, %eax
```

```
counter-=2
```

Reducing Work in Analysis Routines

Key: Shift computation from analysis routines to instrumentation routines whenever possible

Edge Counting: a Slower Version

```
...
void docount2(ADDRINT src, ADDRINT dst, INT32 taken)
{
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}
void Instruction(INS ins, void *v) {
    if (INS_IsBranchOrCall(ins))
    {
       INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount2,
                      IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                      IARG_BRANCH_TAKEN, IARG_END);
    }
}
...
```

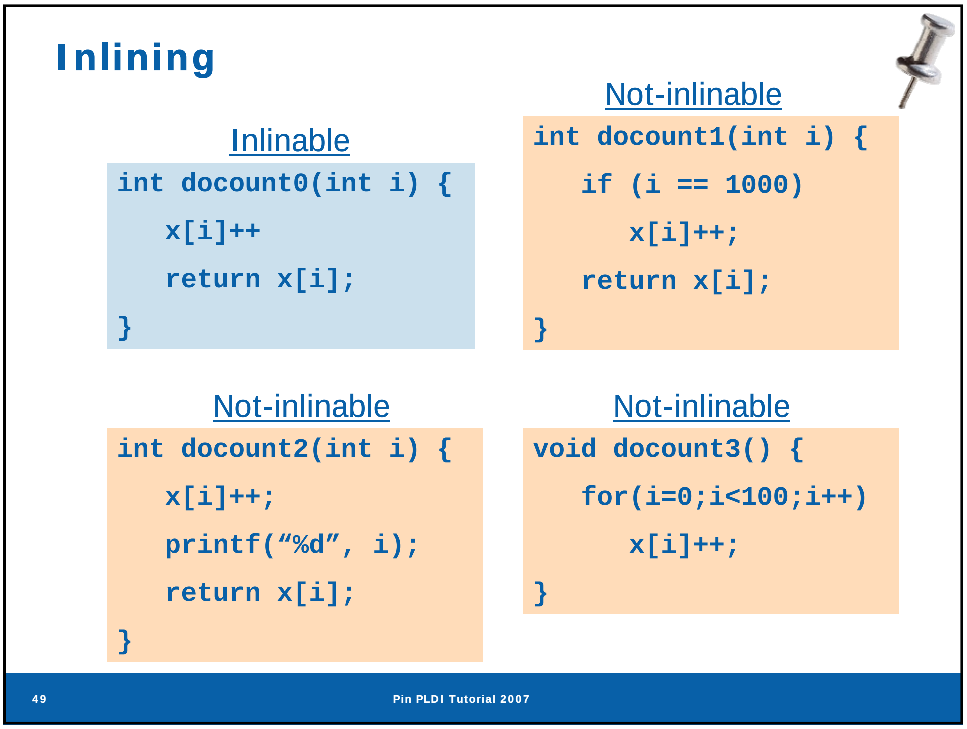
Edge Counting: a Faster Version

```
void docount(COUNTER* pedge, INT32 taken) {
    pedg->count += taken;
}
void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}
void Instruction(INS ins, void *v) {
    if (INS_IsDirectBranchOrCall(ins)) {
       COUNTER *pedg = Lookup(INS_Address(ins),
              INS_DirectBranchOrCallTargetAddress(ins));
       INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount,
              IARG_ADDRINT, pedg, IARG_BRANCH_TAKEN, IARG_END);
    } else
       INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                     IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                     IARG_BRANCH_TAKEN, IARG_END);
}
…
```

Reducing Work for Analysis Transitions

Key: Help Pin's optimizations apply to your analysis routines:

- Inlining
- Scheduling



Inlining

Inlinable

```
int docount0(int i) {
   x[i]++
   return x[i];
}
```

Not-inlinable

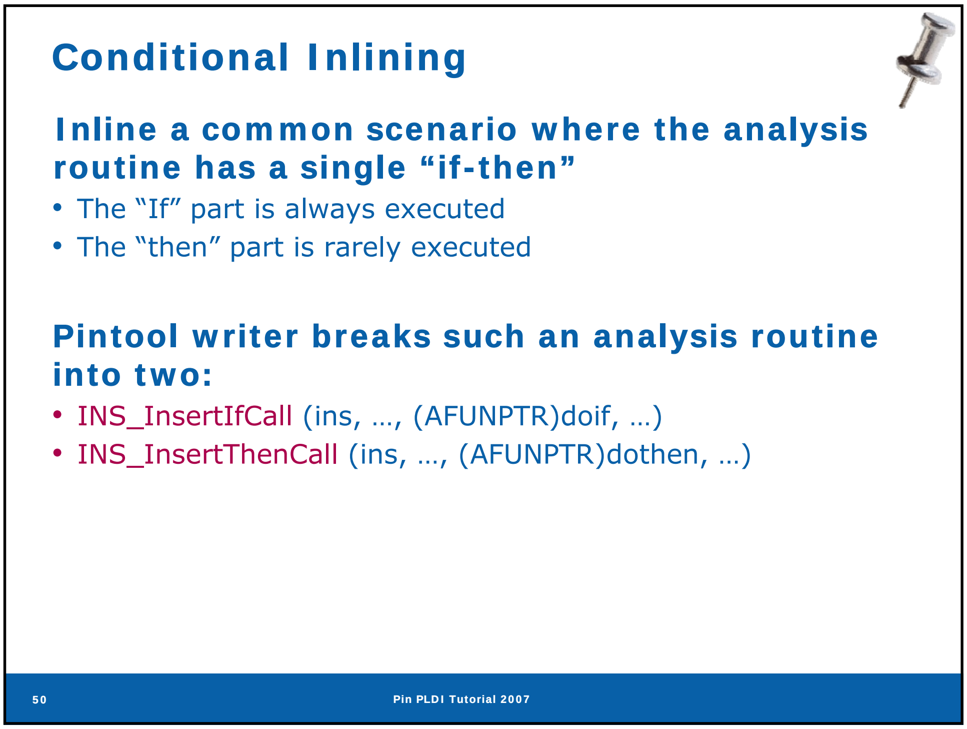
```
int docount1(int i) {
   if (i == 1000)
      x[i]++;
   return x[i];
}
```

Not-inlinable

```
int docount2(int i) {
   x[i]++;
   printf("%d", i);
   return x[i];
}
```

Not-inlinable

```
void docount3() {
   for(i=0;i<100;i++)
      x[i]++;
}
```



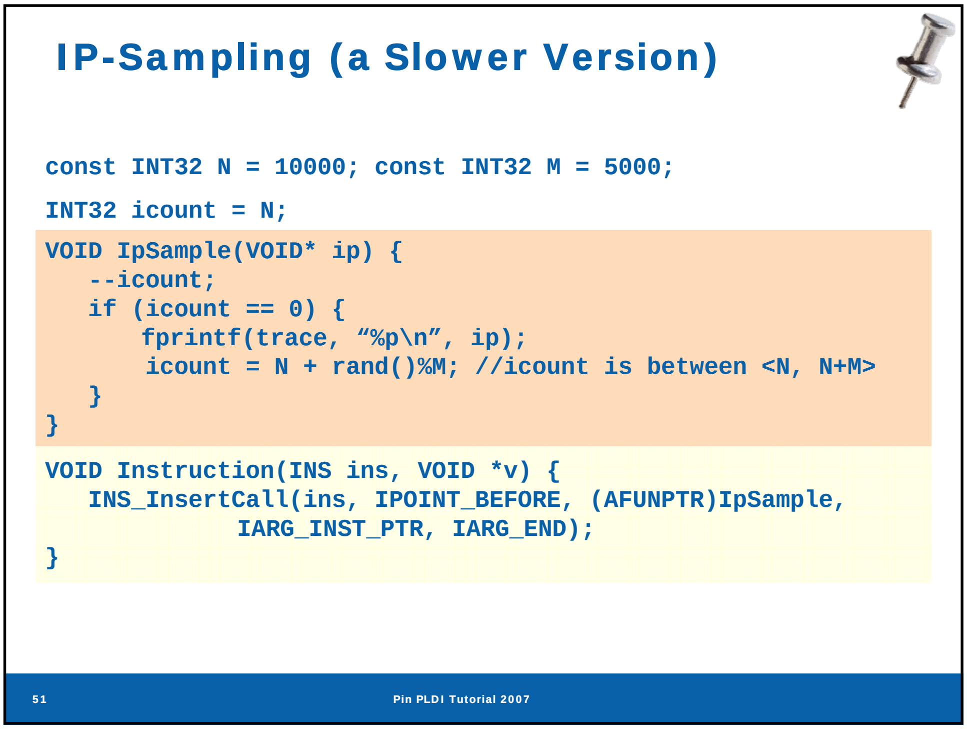
Conditional Inlining

Inline a common scenario where the analysis routine has a single "if-then"

- The "If" part is always executed
- The "then" part is rarely executed

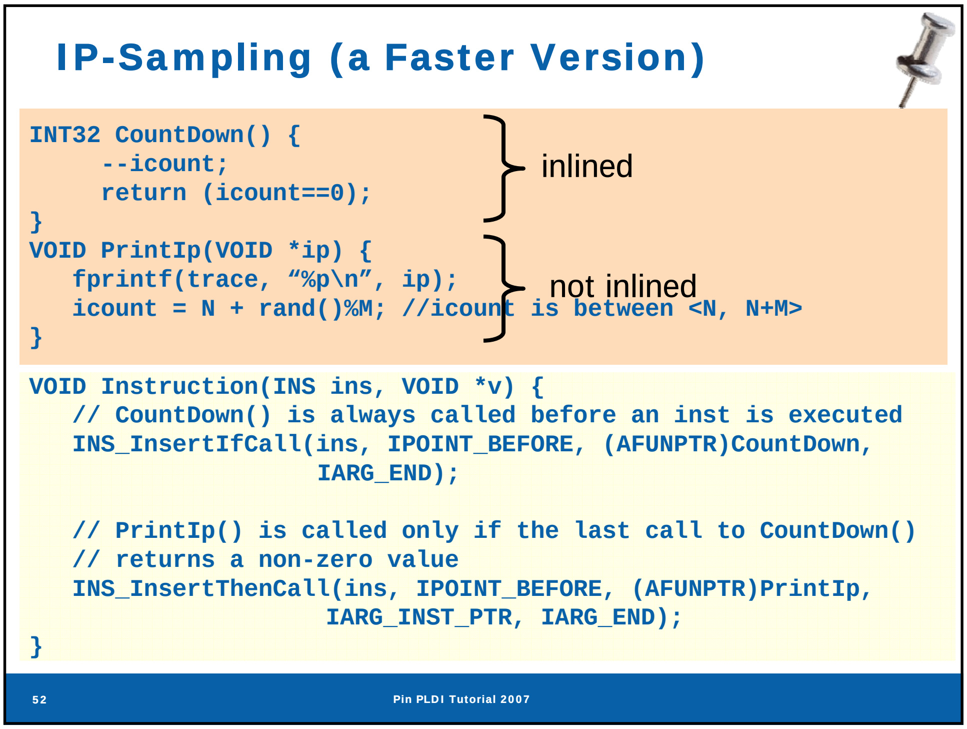
Pintool writer breaks such an analysis routine into two:

- INS_InsertIfCall (ins, …, (AFUNPTR)doif, …)
- INS_InsertThenCall (ins, …, (AFUNPTR)dothen, …)



IP-Sampling (a Slower Version)

```
const INT32 N = 10000; const INT32 M = 5000;

INT32 icount = N;
VOID IpSample(VOID* ip) {
   --icount;
   if (icount == 0) {
       fprintf(trace, “%p\n”, ip);
       icount = N + rand()%M; //icount is between <N, N+M>
   }
}
VOID Instruction(INS ins, VOID *v) {
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)IpSample,
                  IARG_INST_PTR, IARG_END);
}
```



IP-Sampling (a Faster Version)

```
INT32 CountDown() {
     --icount;
     return (icount==0);
}
VOID PrintIp(VOID *ip) {
   fprintf(trace, “%p\n”, ip);
   icount = N + rand()%M; //icount is between <N, N+M>
}
```

inlined

not inlined

```
VOID Instruction(INS ins, VOID *v) {
   // CountDown() is always called before an inst is executed
   INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)CountDown,
                    IARG_END);

   // PrintIp() is called only if the last call to CountDown()
   // returns a non-zero value
   INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR)PrintIp,
                      IARG_INST_PTR, IARG_END);
}
```

Instrumentation Scheduling

If an instrumentation can be inserted anywhere in a basic block:

- Let Pin know via IPOINT_ANYWHERE
- Pin will find the best point to insert the instrumentation to minimize register spilling

ManualExamples/inscount1.cpp

```
#include <stdio.h>
#include "pin.H“
UINT64 icount = 0;
void docount(INT32 c) { icount += c; }
void Trace(TRACE trace, void *v) {
  for (BBL bbl = TRACE_BblHead(trace);
    BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
    BBL_InsertCall(bbl,IPOINT_ANYWHERE,(AFUNPTR)docount,
          IARG_UINT32, BBL_NumIns(bbl), IARG_END);
  }
}
void Fini(INT32 code, void *v) {
  fprintf(stderr, "Count %lld\n", icount);
}
int main(int argc, char * argv[]) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

analysis routine

instrumentation routine

Conclusions

A dynamic instrumentation system for building your own program analysis tools

Runs on multiple platforms:

- IA-32, Intel64, Itanium, and XScale
- Linux, Windows, MacOS

Works on real-life applications

Efficient instrumentation (especially with your help!)

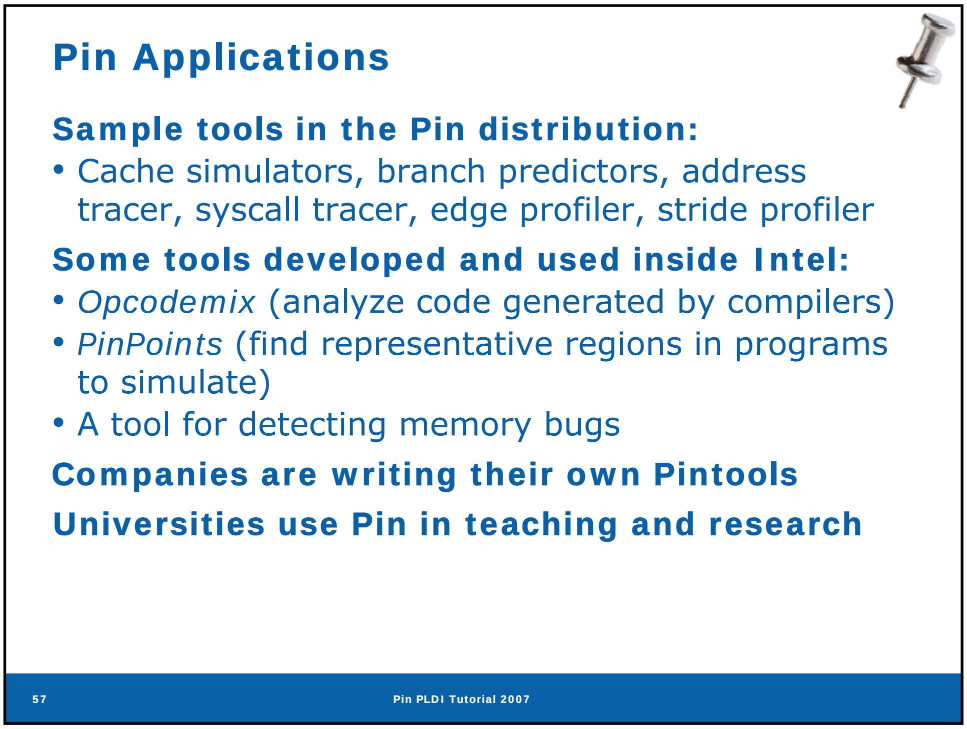
Part Two: Fundamental Concepts in Compilers and Architecture

Kim Hazelwood

David Kaeli

Dan Connors

Vijay Janapa Reddi



Pin Applications

Sample tools in the Pin distribution:

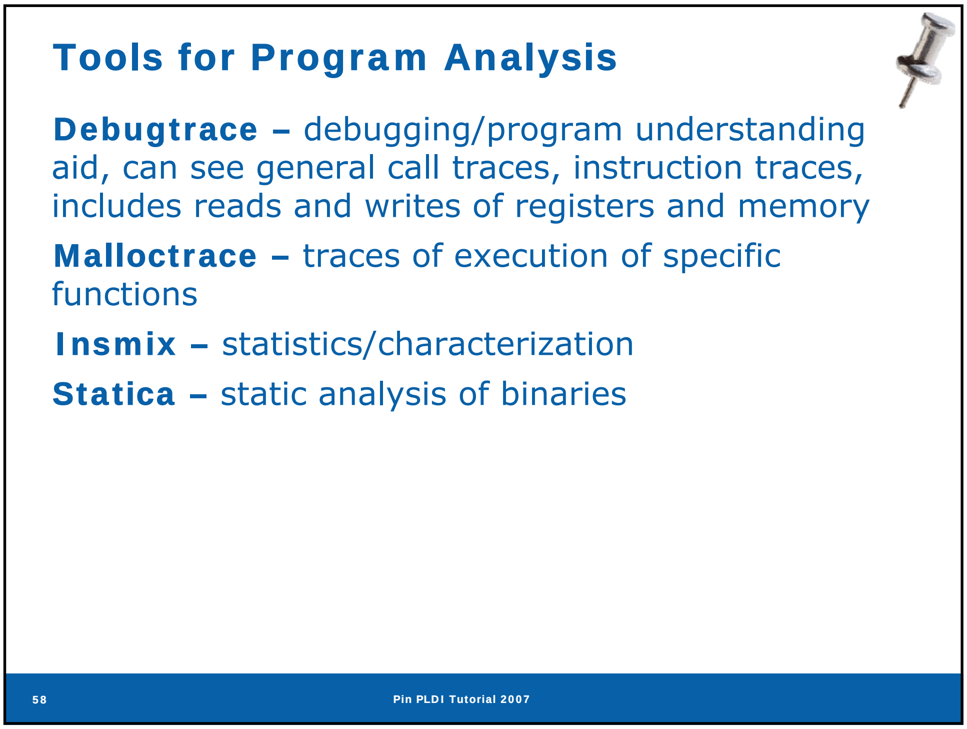
- Cache simulators, branch predictors, address tracer, syscall tracer, edge profiler, stride profiler

Some tools developed and used inside Intel:

- *Opcodemix* (analyze code generated by compilers)
- *PinPoints* (find representative regions in programs to simulate)
- A tool for detecting memory bugs

Companies are writing their own Pintools

Universities use Pin in teaching and research



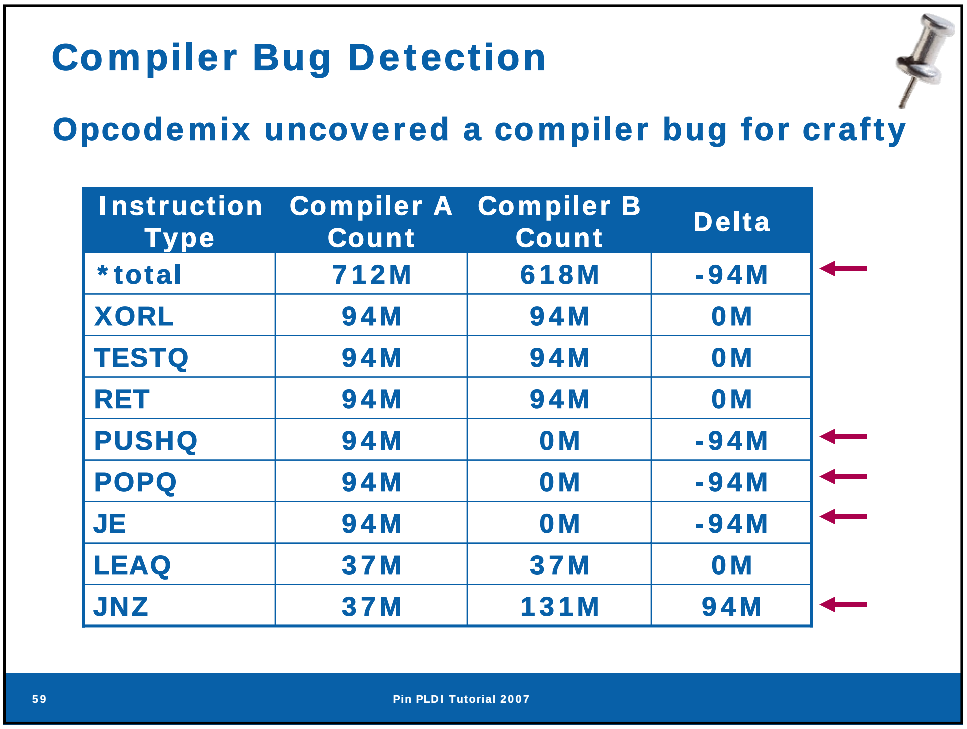
Tools for Program Analysis

Debugtrace – debugging/program understanding aid, can see general call traces, instruction traces, includes reads and writes of registers and memory

Malloctrace – traces of execution of specific functions

Insmix – statistics/characterization

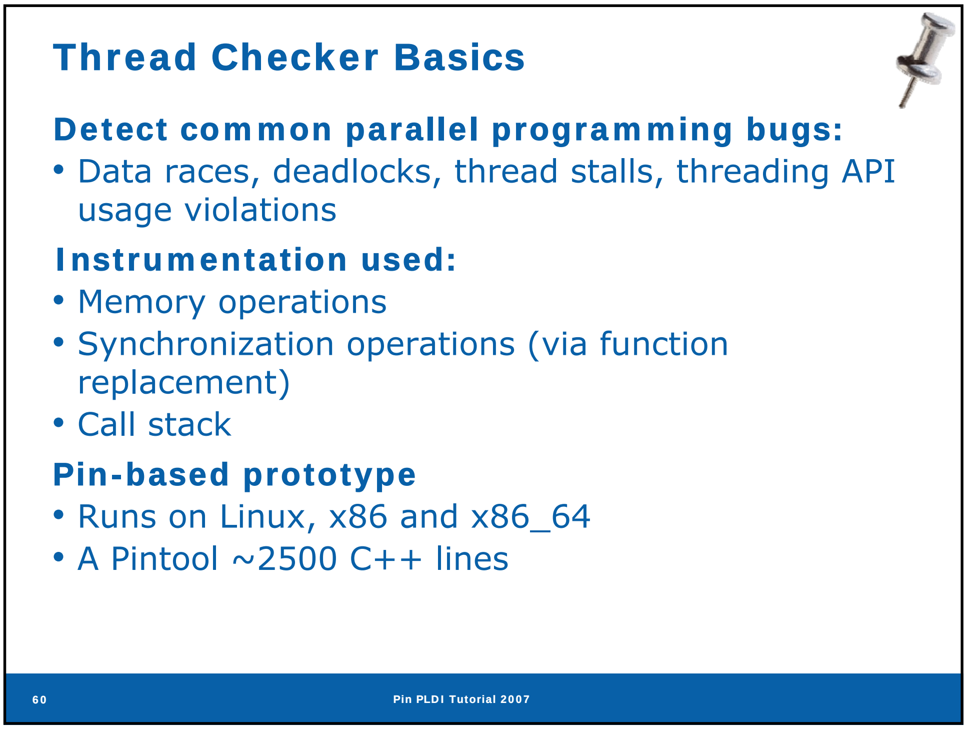
Statica – static analysis of binaries



Compiler Bug Detection

Opcodemix uncovered a compiler bug for crafty

Instruction Type	Compiler A Count	Compiler B Count	Delta
*total	712M	618M	-94M
XORL	94M	94M	0M
TESTQ	94M	94M	0M
RET	94M	94M	0M
PUSHQ	94M	0M	-94M
POPQ	94M	0M	-94M
JE	94M	0M	-94M
LEAQ	37M	37M	0M
JNZ	37M	131M	94M



Thread Checker Basics

Detect common parallel programming bugs:

- Data races, deadlocks, thread stalls, threading API usage violations

Instrumentation used:

- Memory operations
- Synchronization operations (via function replacement)
- Call stack

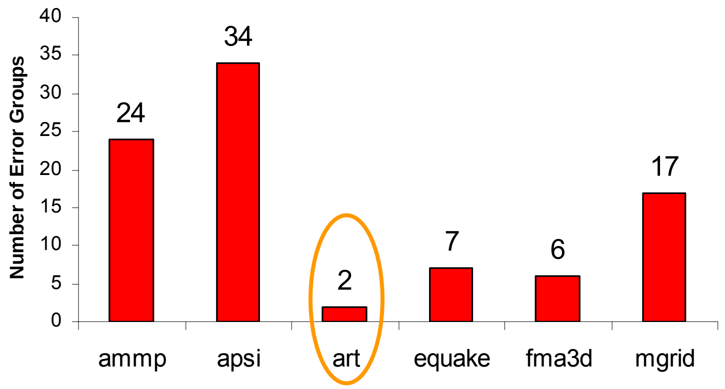
Pin-based prototype

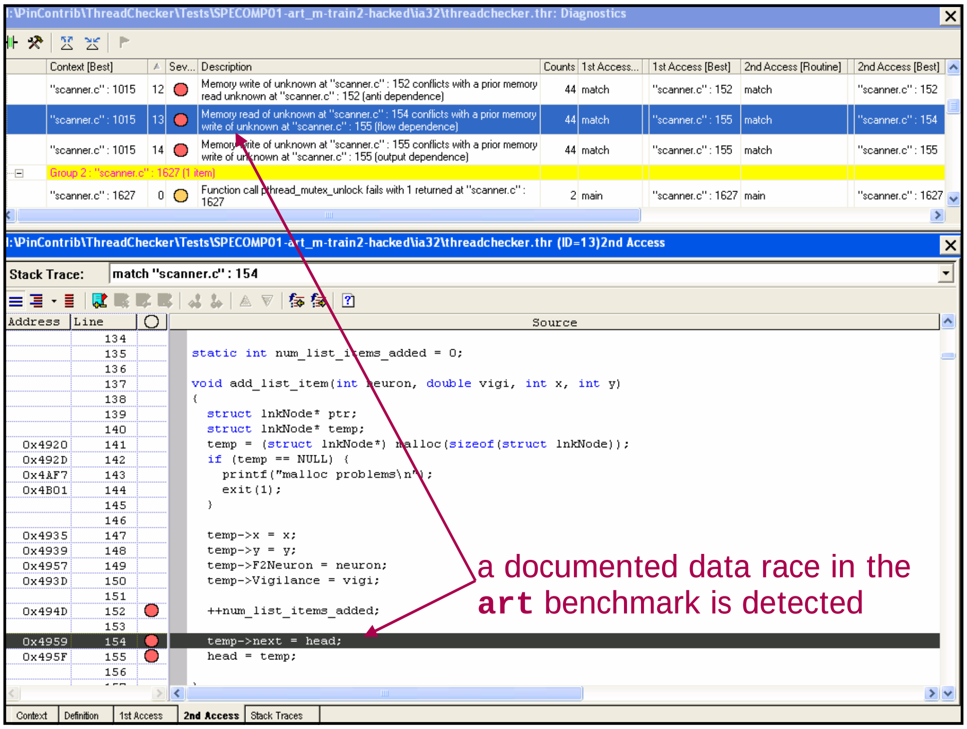
- Runs on Linux, x86 and x86_64
- A Pintool ~2500 C++ lines

Thread Checker Results

Potential errors in SPECOMP01 reported by Thread Checker

(4 threads were used)





I:\PinContrib\ThreadChecker\Tests\SPECOMP01-art_m-train2-hacked\ia32\threadchecker.thr: Diagnostics

Context [Best]		Sev...	Description	Counts	1st Access...	1st Access [Best]	2nd Access [Routine]	2nd Access [Best]
"scanner.c" : 1015	12	🔴	Memory write of unknown at "scanner.c" : 152 conflicts with a prior memory read unknown at "scanner.c" : 152 (anti dependence)	44	match	"scanner.c" : 152	match	"scanner.c" : 152
"scanner.c" : 1015	13	🔴	Memory read of unknown at "scanner.c" : 154 conflicts with a prior memory write of unknown at "scanner.c" : 155 (flow dependence)	44	match	"scanner.c" : 155	match	"scanner.c" : 154
"scanner.c" : 1015	14	🔴	Memory write of unknown at "scanner.c" : 155 conflicts with a prior memory write of unknown at "scanner.c" : 155 (output dependence)	44	match	"scanner.c" : 155	match	"scanner.c" : 155
Group 2 : "scanner.c" : 1627 (1 item)								
"scanner.c" : 1627	0	🟡	Function call pthread_mutex_unlock fails with 1 returned at "scanner.c" : 1627	2	main	"scanner.c" : 1627	main	"scanner.c" : 1627

I:\PinContrib\ThreadChecker\Tests\SPECOMP01-art_m-train2-hacked\ia32\threadchecker.thr (ID=13)2nd Access

Stack Trace: match "scanner.c" : 154

```
Address  Line  ○   Source
         134
         135       static int num_list_items_added = 0;
         136
         137       void add_list_item(int neuron, double vigi, int x, int y)
         138       {
         139         struct lnkNode* ptr;
         140         struct lnkNode* temp;
0x4920   141         temp = (struct lnkNode*) malloc(sizeof(struct lnkNode));
0x492D   142         if (temp == NULL) {
0x4AF7   143           printf("malloc problems\n");
0x4B01   144           exit(1);
         145         }
         146
0x4935   147         temp->x = x;
0x4939   148         temp->y = y;
0x4957   149         temp->F2Neuron = neuron;
0x493D   150         temp->Vigilance = vigi;
         151
0x494D   152  ●      ++num_list_items_added;
         153
0x4959   154  ●      temp->next = head;
0x495F   155  ●      head = temp;
         156
```

a documented data race in the **art** benchmark is detected

Instrumentation Driven Simulation

Fast exploratory studies

- Instrumentation ~= native execution
- Simulation speeds at MIPS

Characterize complex applications

- E.g. Oracle, Java, parallel data-mining apps

Simple to build instrumentation tools

- Tools can feed simulation models in real time
- Tools can gather instruction traces for later use

Performance Models

Branch Predictor Models:

- PC of conditional instructions
- Direction Predictor: Taken/not-taken information
- Target Predictor: PC of target instruction if taken

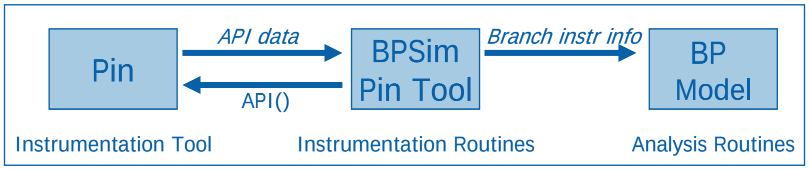
Cache Models:

- Thread ID (if multi-threaded workload)
- Memory address
- Size of memory operation
- Type of memory operation (Read/Write)

Simple Timing Models:

- Latency information

Branch Predictor Model



Pin

API data

API()

BPSim Pin Tool

Branch instr info

BP Model

Instrumentation Tool

Instrumentation Routines

Analysis Routines

BPSim Pin Tool

- Instruments all branches
- Uses API to set up call backs to analysis routines

Branch Predictor Model:

- Detailed branch predictor simulator

BP Implementation

```
BranchPredictor myBPU;
```

ANALYSIS

```
VOID ProcessBranch(ADDRINT PC, ADDRINT targetPC, bool BrTaken) {
  BP_Info pred = myBPU.GetPrediction( PC );
  if( pred.Taken != BrTaken ) {
    // Direction Mispredicted
  }
  if( pred.predTarget != targetPC ) {
    // Target Mispredicted
  }
  myBPU.Update( PC, BrTaken, targetPC);
}
```

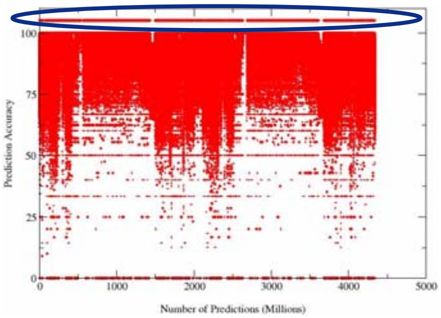
INSTRUMENT

```
VOID Instruction(INS ins, VOID *v)
{
  if( INS_IsDirectBranchOrCall(ins) || INS_HasFallThrough(ins) )
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) ProcessBranch,
      ADDRINT, INS_Address(ins),
      IARG_UINT32, INS_DirectBranchOrCallTargetAddress(ins),
      IARG_BRANCH_TAKEN, IARG_END);
}
```

MAIN

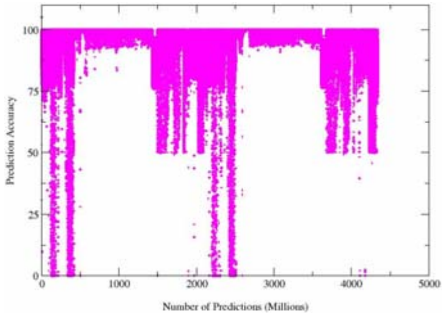
```
int main() {
  PIN_Init();
  INS_AddInstrumentationFunction(Instruction, 0);
  PIN_StartProgram();
}
```

Branch Predictor Performance - GCC



Bimodal not chosen

Bimodal In McFarling Predictor



McFarling Predictor

Branch prediction accuracies range from 0-100%

Branches are hard to predict in some phases

- Can simulate these regions alone by fast forwarding to them in real time

Performance Model Inputs

Branch Predictor Models:

- PC of conditional instructions
- Direction Predictor: Taken/not-taken information
- Target Predictor: PC of target instruction if taken

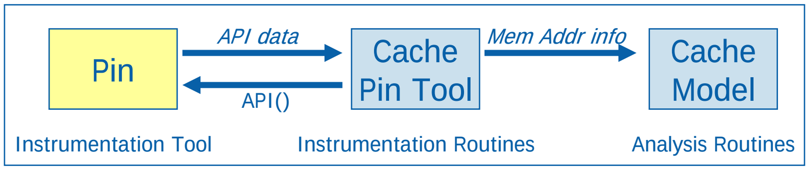
Cache Models:

- Thread ID (if multi-threaded workload)
- Memory address
- Size of memory operation
- Type of memory operation (Read/Write)

Simple Timing Models:

- Latency information

Cache Simulators



Pin

API data

API()

Cache Pin Tool

Mem Addr info

Cache Model

Instrumentation Tool

Instrumentation Routines

Analysis Routines

Cache Pin Tool

- Instruments all instructions that reference memory
- Use API to set up call backs to analysis routines

Cache Model:

- Detailed cache simulator

Cache Implementation

```
CACHE_t CacheHierarchy[MAX_NUM_THREADS][MAX_NUM_LEVELS];
```

ANALYSIS

```
VOID MemRef(int tid, ADDRINT addrStart, int size, int type) {
  for(addr=addrStart; addr<(addrStart+size); addr+=LINE_SIZE)
    LookupHierarchy( tid, FIRST_LEVEL_CACHE, addr, type);
}
VOID LookupHierarchy(int tid, int level, ADDRINT addr, int accessType){
  result = cacheHier[tid][cacheLevel]->Lookup(addr, accessType );
  if( result == CACHE_MISS ) {
    if( level == LAST_LEVEL_CACHE ) return;
    LookupHierarchy(tid, level+1, addr, accessType);
  }
}
```

INSTRUMENT

```
VOID Instruction(INS ins, VOID *v)
{
  if( INS_IsMemoryRead(ins) )
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) MemRef,
      IARG_THREAD_ID, IARG_MEMORYREAD_EA, IARG_MEMORYREAD_SIZE,
      IARG_UINT32, ACCESS_TYPE_LOAD, IARG_END);
  if( INS_IsMemoryWrite(ins) )
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) MemRef,
      IARG_THREAD_ID, IARG_MEMORYWRITE_EA, IARG_MEMORYWRITE_SIZE,
      IARG_UINT32, ACCESS_TYPE_STORE, IARG_END);
}
```

MAIN

```
int main() {
  PIN_Init();
  INS_AddInstrumentationFunction(Instruction, 0);
  PIN_StartProgram();
}
```

Performance Models

Branch Predictor Models:

- PC of conditional instructions
- Direction Predictor: Taken/not-taken information
- Target Predictor: PC of target instruction if taken

Cache Models:

- Thread ID (if multi-threaded workload)
- Memory address
- Size of memory operation
- Type of memory operation (Read/Write)

Simple Timing Models:

- Latency information

Simple Timing Model

Assume 1-stage pipeline

- T_i cycles for instruction execution

Assume branch misprediction penalty

- T_b cycles penalty for branch misprediction

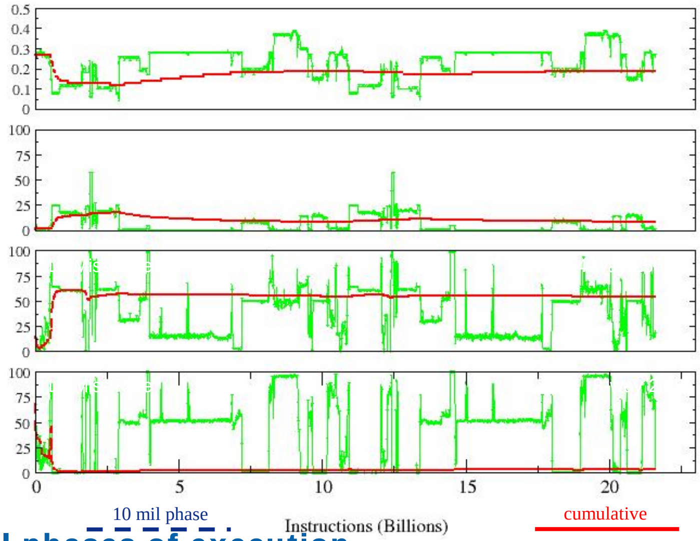
Assume cache access & miss penalty

- T_l cycles for demand reference to cache level l
- T_m cycles for demand reference to memory

$$\textbf{Total cycles} = \alpha T_i + \beta T_b + \sum_{l=1}^{LLC} A_l T_l + \eta T_m$$

α = instruction count; β = # branch mispredicts ;
A_l = # accesses to cache level l ; η = # last level cache (LLC) misses

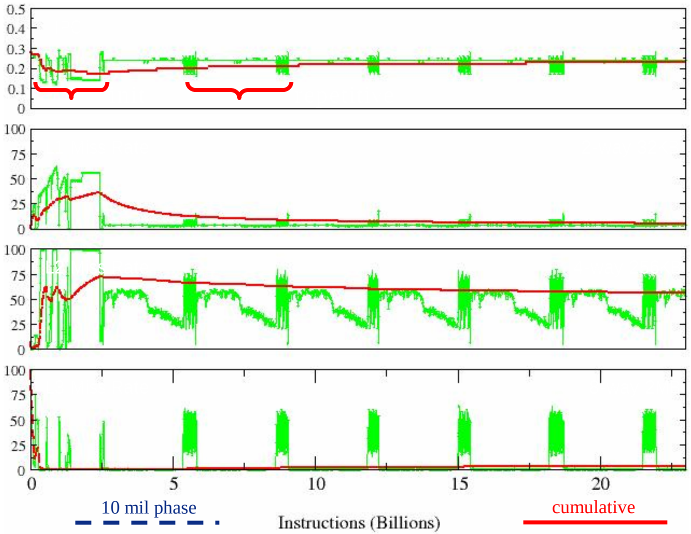
Performance - GCC



Several phases of execution

- Important to pick the correct phase of exeuction

Performance – AMMP



One loop (3 billion instructions) is representative

- High miss rate at beginning; exploits locality at end

More Fundamental Concepts using Pin

Kim Hazelwood

David Kaeli

Dan Connors

Vijay Janapa Reddi

Moving from 32-bit to 64-bit Applications

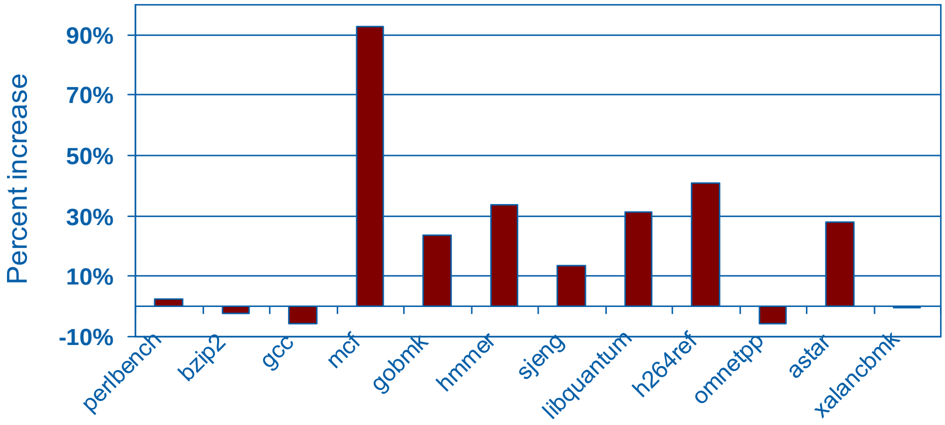
- Intuition would tell us that code expansion will occur
- How can compiler writers exploit the features of a 64-bit ISA?
- What type of programs will benefit from this migration?
- What data types make use of the move to 64 bits?
- How do we begin to identify the reasons for the performance results shown in this table?

➢ **Profiling with Pin!**

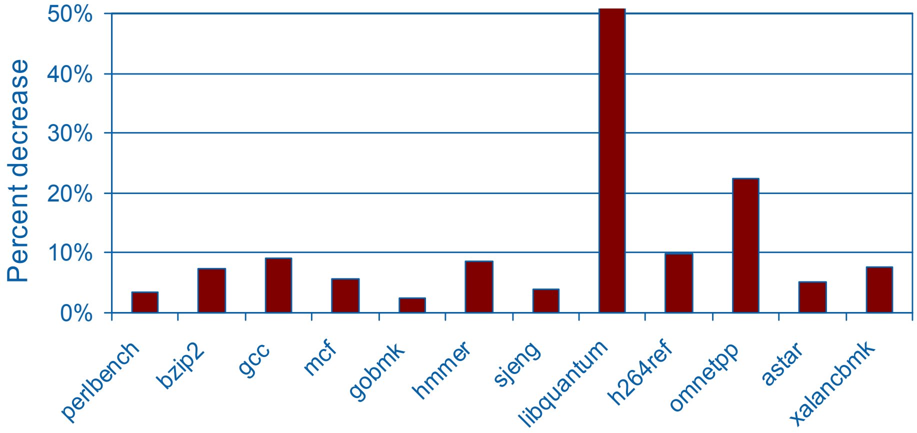
Benchmark	Language	64-bit vs. 32-bit speedup
perlbench	C	3.42%
bzip2	C	15.77%
gcc	C	-18.09%
mcf	C	-26.35%
gobmk	C	4.97%
hmmer	C	34.34%
sjeng	C	14.21%
libquantum	C	35.38%
h264ref	C	35.35%
omnetpp	C++	-7.83%
astar	C++	8.46%
xalancbmk	C++	-13.65%
Average		7.16%

Ye06, IISWC2006

Code Size *Increases* in 64-bit Mode



Dynamic Instruction Count *Decreases* in 64-bit Mode



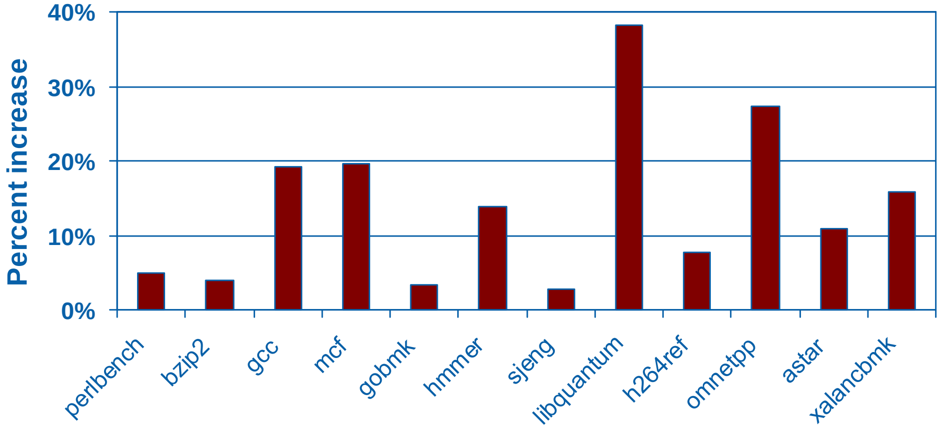
Observations

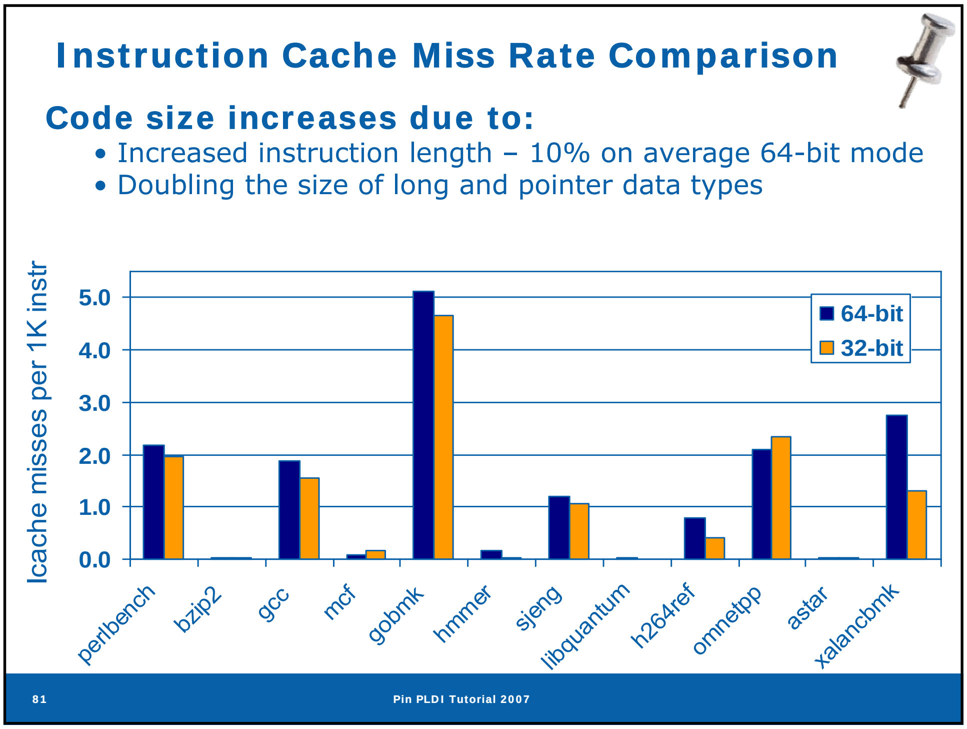
Code density increases in 64-bit mode

- More registers help
- 64-bit integer arithmetic helps a lot in the case of libquantum

Concern over the reduction of decoding efficiency in 64-bit is not substantiated

Instruction Cache (L1) Request Rate *Increases* in 64-bit Mode

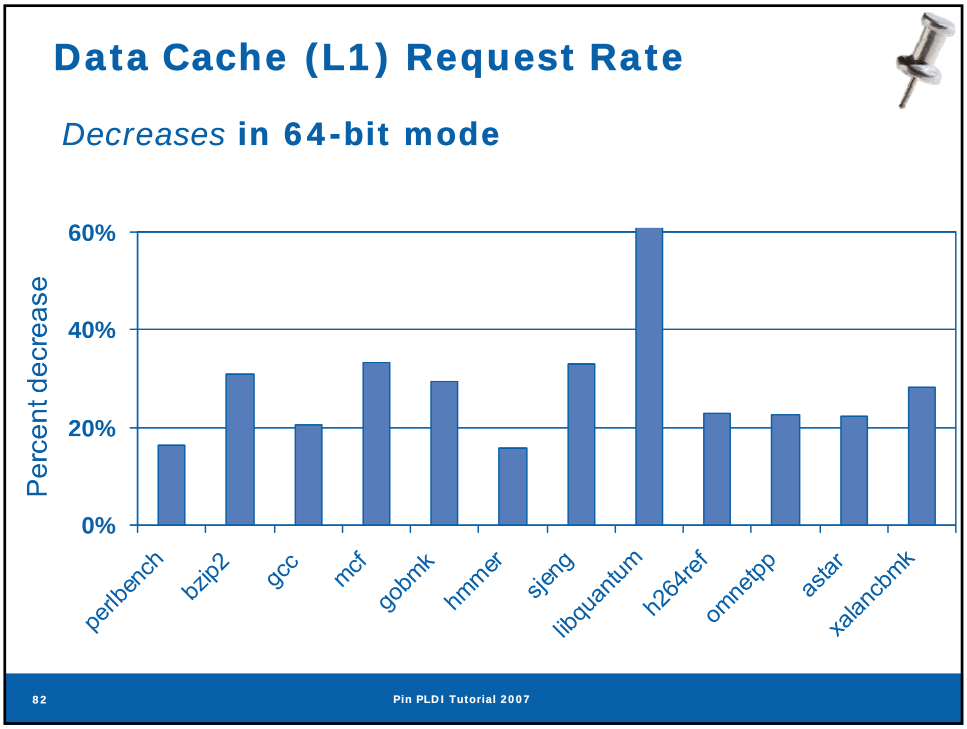




Instruction Cache Miss Rate Comparison

Code size increases due to:

- Increased instruction length – 10% on average 64-bit mode
- Doubling the size of long and pointer data types

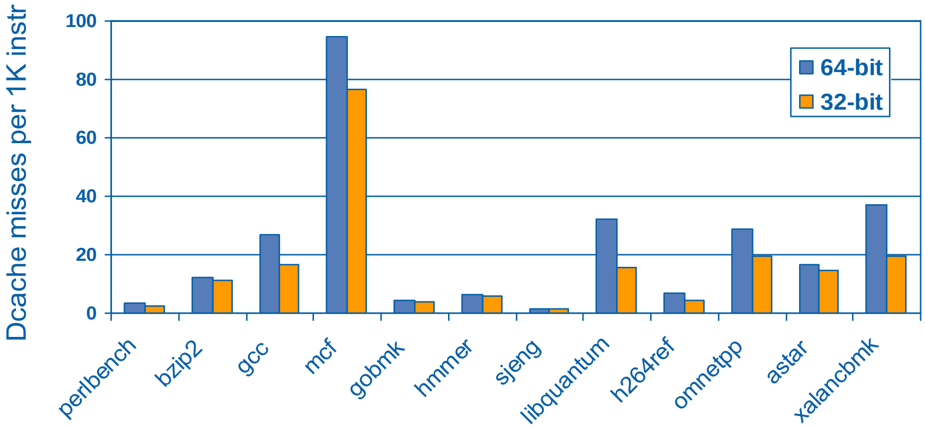


Data Cache (L1) Request Rate

Decreases **in 64-bit mode**

Data Cache Miss Rate Comparison

Data cache miss rate increases in 64-bit



Observations

The instruction cache miss rate is very low in both 64-bit and 32-bit modes

The data cache request rate decreases significantly in 64-bit mode

- Extra registers help

The data cache miss rate increases in 64-bit mode

- The increased size of long and pointer data types has an adverse impact on data cache performance

Moving from 32-bit to 64-bit Applications

- **Common assumptions associated with changes in architecture word sizes need to be studied carefully**
- **All of these analyses were done with slightly modified versions of the Pintools in the SimpleExamples directory shipped with Pin**
  - icount.cpp
  - ilenmix.cpp
  - opcodemix.cpp
  - icache.cpp
  - dcache.cpp

Conclusions

Instrumentation based simulation:

- Simple compared to detailed models
- Can easily run complex applications
- Provides insight on workload behavior over their entire runs in a reasonable amount of time

Illustrated the use of Pin for:

- Compilers
  – Bug detection, thread analysis
- Computer architecture
  – Branch predictors, cache simulators, timing models, architecture width
- Architecture changes
  – Move from 32-bit to 64-bit

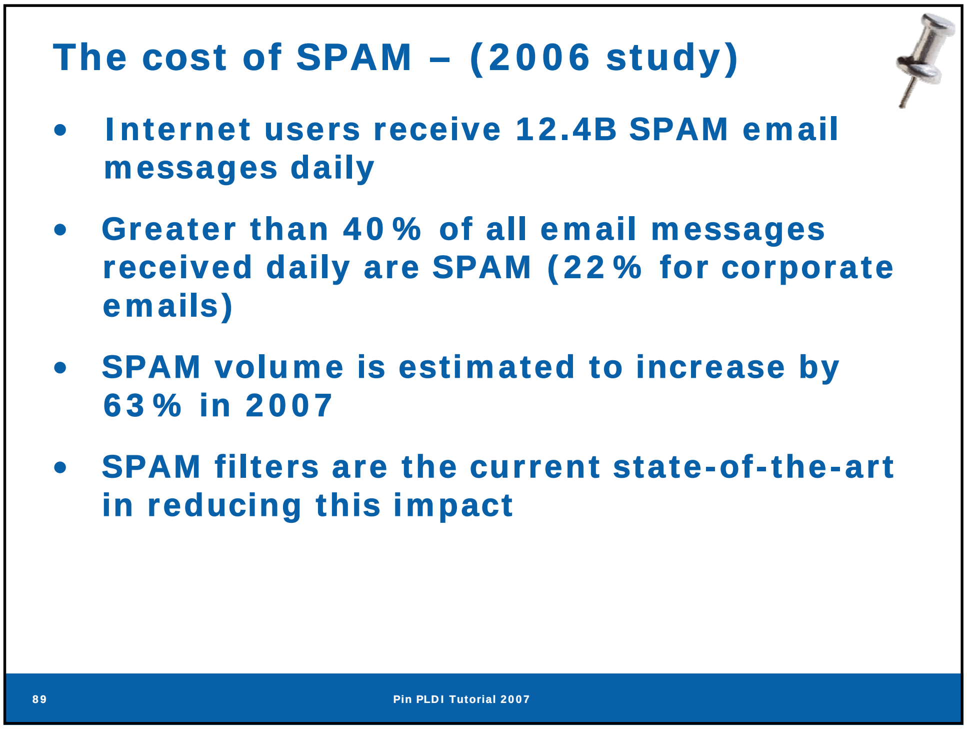
Part Three: Advanced Concepts in Compilers and Architecture

Kim Hazelwood
David Kaeli
Dan Connors
Vijay Janapa Reddi

Using Pin in Security Research

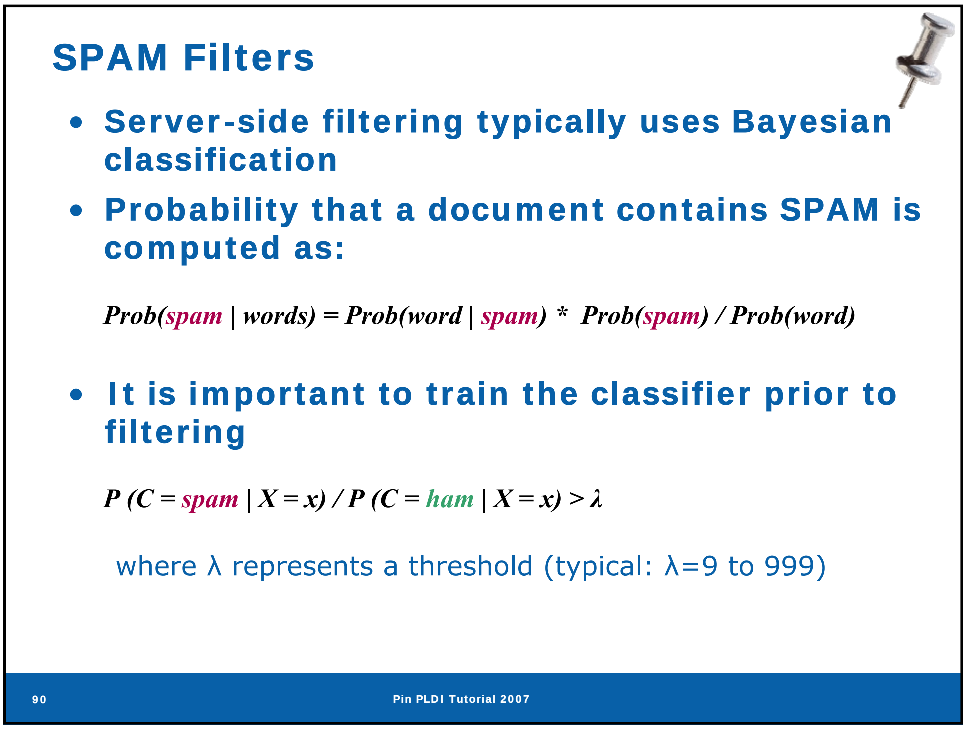
- **How do we design architectural extensions to accelerate SPAM filtering and anti-virus scanning workloads?**
- **How do we track dynamic information flow to detect zero-day attack intrusions?**

➢ Pin!!



The cost of SPAM – (2006 study)

- Internet users receive 12.4B SPAM email messages daily
- Greater than 40% of all email messages received daily are SPAM (22% for corporate emails)
- SPAM volume is estimated to increase by 63% in 2007
- SPAM filters are the current state-of-the-art in reducing this impact



SPAM Filters

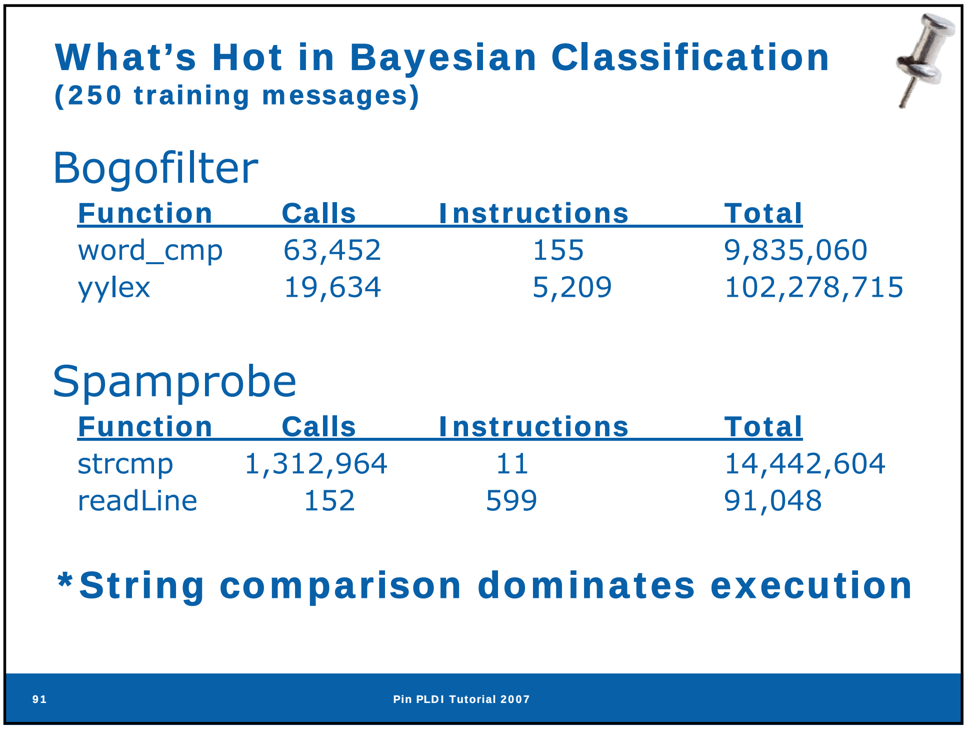
- Server-side filtering typically uses Bayesian classification
- Probability that a document contains SPAM is computed as:

$$Prob(spam \mid words) = Prob(word \mid spam) * Prob(spam) / Prob(word)$$

- It is important to train the classifier prior to filtering

$$P\,(C = spam \mid X = x) / P\,(C = ham \mid X = x) > \lambda$$

where λ represents a threshold (typical: λ=9 to 999)



What’s Hot in Bayesian Classification
(250 training messages)

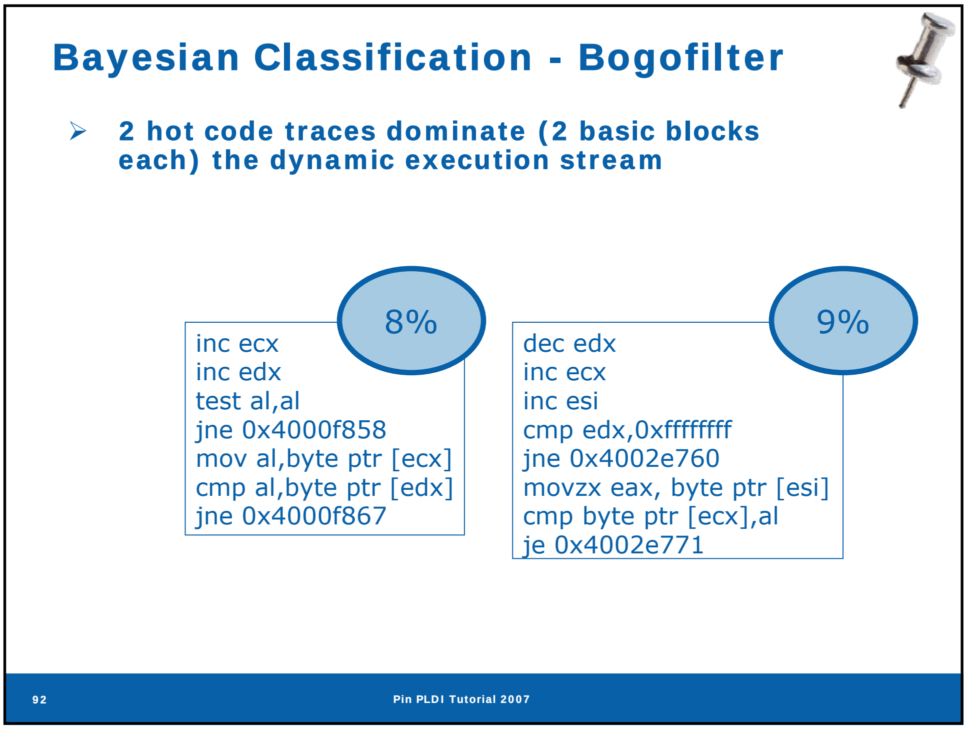
Bogofilter

Function	Calls	Instructions	Total
word_cmp	63,452	155	9,835,060
yylex	19,634	5,209	102,278,715

Spamprobe

Function	Calls	Instructions	Total
strcmp	1,312,964	11	14,442,604
readLine	152	599	91,048

***String comparison dominates execution**



Bayesian Classification - Bogofilter

- **2 hot code traces dominate (2 basic blocks each) the dynamic execution stream**

8%

```
inc ecx
inc edx
test al,al
jne 0x4000f858
mov al,byte ptr [ecx]
cmp al,byte ptr [edx]
jne 0x4000f867
```

9%

```
dec edx
inc ecx
inc esi
cmp edx,0xffffffff
jne 0x4002e760
movzx eax, byte ptr [esi]
cmp byte ptr [ecx],al
je 0x4002e771
```

Bayesian Classification - Spamprobe

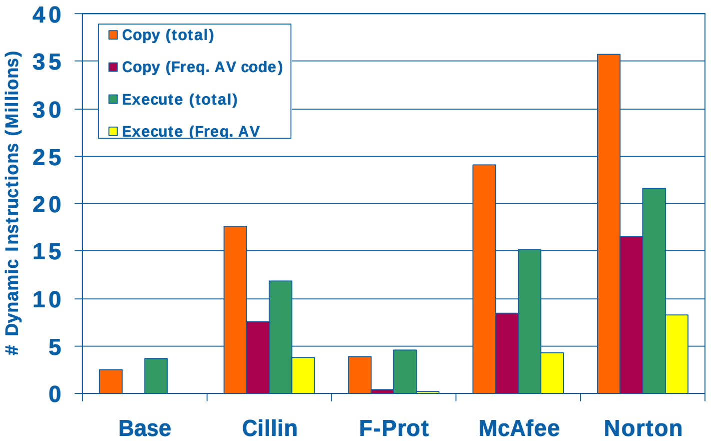
1 hot code
trace makes up 17%
of the total execution

```
mov esi,dword ptr 0x14[ebp]
test esi,esi
mov esi,dword ptr 0xf0[ebp]
mov edx,edi
movzx eax, dl
add eax,esi
movzx eax, byte ptr [eax]
cmp dword ptr 0xec[ebp],eax
ja 0x420cd84f
```

- ➢A hot comparison block dominates execution

The Cost of Anti-virus Execution

Copy/Execution of "Hello, world" Application



Overhead Causes in Anti-Virus Security Mechanisms

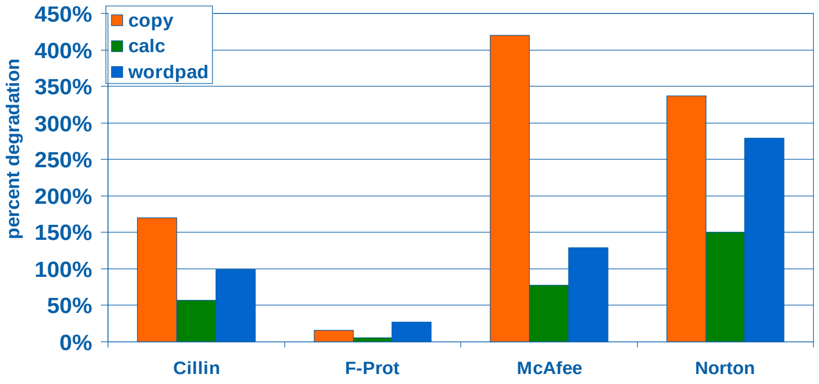
Signature Matching

- Program will refer to a dictionary of “signatures” or sequences of code known to be part of a malicious file
- If a signature is found in the file in question, it is marked as a virus
- Disadvantages:
  - Requires continuous updates
  - Cannot detect “zero-day attacks”

Heuristics

- Set of rules that the AV software will apply
  - For example, if the file contains self-modifying code
- If the file in question violates any of the given rules, it is marked as a virus
- Advantages:
  - May find virus variants
- Disadvantages:
  - Generates false positives

Antivirus Software Overhead



What’s hot in AV applications?

Frequent “Hot” Code Examples

PC-Cillin

```
mov edx, dword ptr 0xb0[ebp]
inc ecx
add eax, 0xc
cmp ecx, edx
mov dword ptr 0xd4[ebp], ecx
jl 0xf45cc81a
```

F-Prot

```
mov exc, dword ptr 0x8[ebp]
mov cl, byte ptr[ecx]
cmp cl, byte ptr 0xc[ebp]
je 0xf76a713
```

McAfee

```
xor edi, edi
mov ecx dword ptr 0x8[ebp]
mov al, byte ptr 0x1[ebx]
lea edx,[edi][ecx]
mov cl, byte ptr [edi][ecx]
cmp al, cl
jne 0x1203c028
```

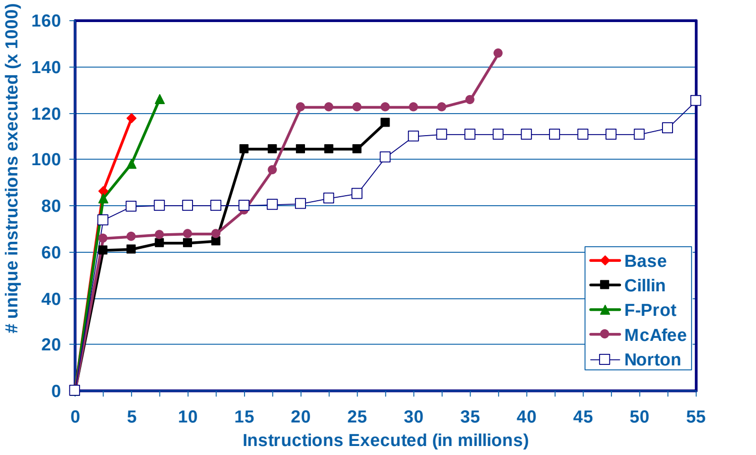
Norton

```
movzx edi, ax
imul edi,dword ptr 0xcfc[edx]
mov ebx,dword ptr 0x10[ebp]
add edi,ecx
cmp ebx,dword ptr [edi]
je 0xf6a13e02
```

- **Frequent code exhibits similar structure**

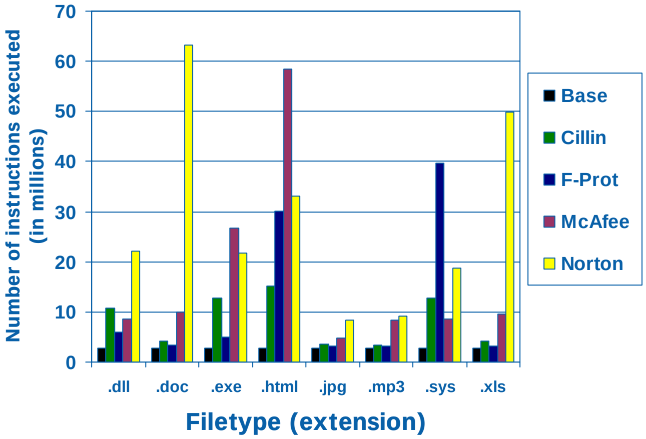
Instruction Footprint

Scenario: Copying a file from CDROM to the C drive



Workload Characterization

Copy file, 128 KB



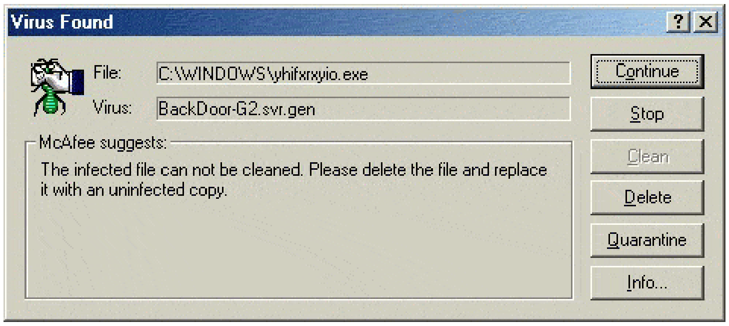
Architectural Extensions for SPAM Filtering and AV Scanning

- Can we exploit the characteristics of hot blocks and develop ISA extensions to accelerate these operations?

- How will we address these issues when running in a virtualized environment? – Recent project with VMware

☺ Pin can help us identify answers to these challenging questions….

Pin for Information Flow Tracking



Virus Found

File: C:\WINDOWS\yhifxrxyio.exe

Virus: BackDoor-G2.svr.gen

McAfee suggests:

The infected file can not be cleaned. Please delete the file and replace it with an uninfected copy.

Zero-Day Attack Trends

In 2005, 74% of the top 50 code samples exposed confidential information

- ➢ Effects of Trojan Horses and Backdoors are more subtle

Symantec reported that 6 of the top 10 spyware were bundled with other programs

- ➢ Malware are executed without explicit consent

Zero-day attacks are increasing and are sold on black market

- ➢ Freshly authored malicious code can go undetected by even the most up-to-date virus scanners

We need a behavior-tracking mechanism that does not rely on known signatures

Example 1: PWSteal.Tarno.Q

Password-stealing Trojan Horse

1. Arrives in an email with a downloader in the attachment
   Subject: Payment Receipt
   Message: Dear Customer …
   Attachment: FILE.965658.exe
2. Downloads main part of the Trojan from a fixed location
   [http://]dimmers.phpwebhosting.com/msupdate.exe?r=[UNIQUE_ID]
3. Creates a browser help object that runs every time Internet Explorer starts
4. Monitors windows and web pages with specific strings
   gold, cash, bank, log, user, pin, memorable, secret
5. Periodically sends gather information using the following url:
   [http://]neverdays.com/reporter.php

Example 2: Trojan.Lodeight.A

Trojan Horse that installs a Beagle and a backdoor

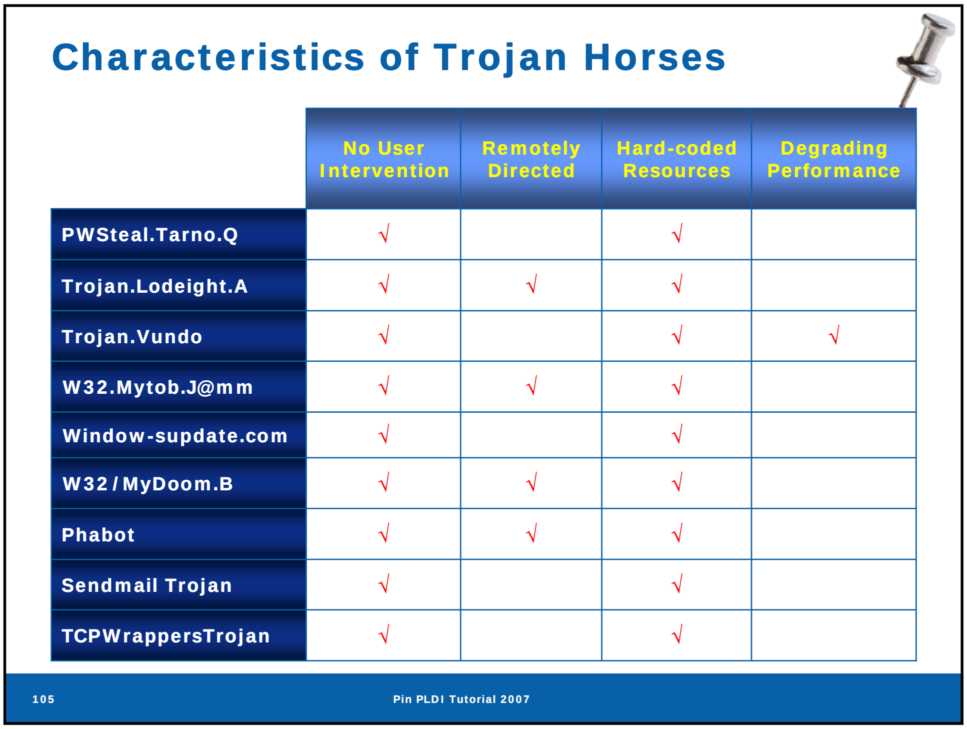
1. Contacts one of the following web sites using TCP port 80

   [http://]www.tsaa.net/[REMOVED]
   [http://]www.aureaorodeley.com/[REMOVED]

2. Downloads a remote file into the following folder and executes it. This remote file may be a mass-mailing worm such as W32.Beagle.CZ@mm.

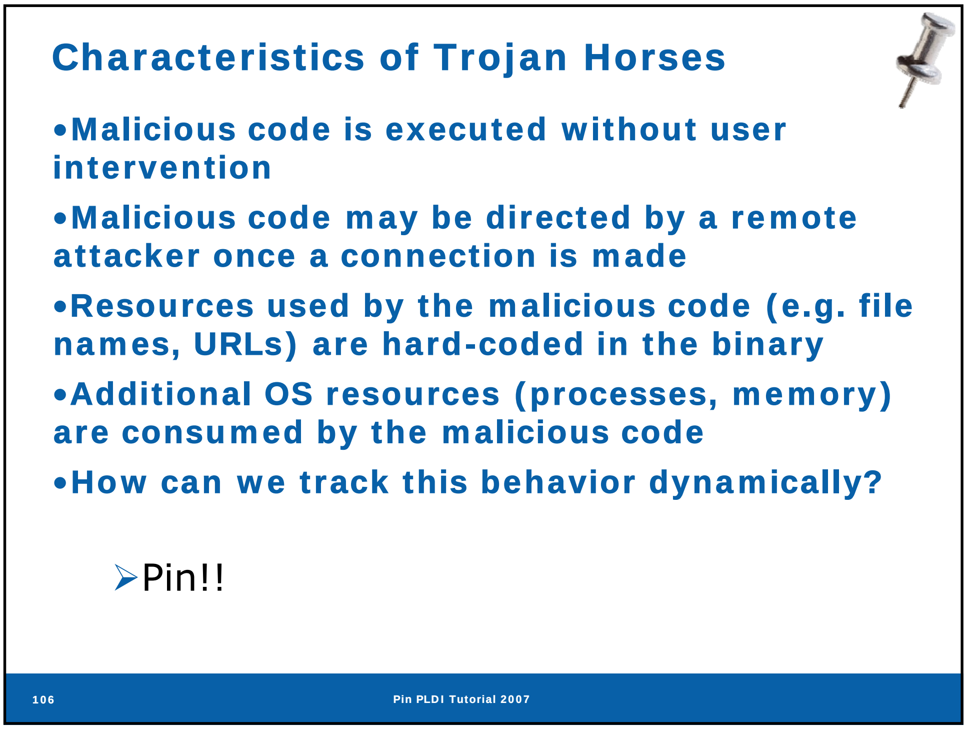
   %Windir%\[RANDON NAME].exe

3. Opens a back door on TCP port 1084



Characteristics of Trojan Horses

	No User Intervention	Remotely Directed	Hard-coded Resources	Degrading Performance
PWSteal.Tarno.Q	√		√	
Trojan.Lodeight.A	√	√	√	
Trojan.Vundo	√		√	√
W32.Mytob.J@mm	√	√	√	
Window-supdate.com	√		√	
W32/MyDoom.B	√	√	√	
Phabot	√	√	√	
Sendmail Trojan	√		√	
TCPWrappersTrojan	√		√	



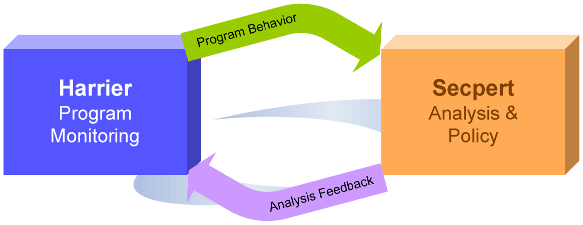
Characteristics of Trojan Horses

- Malicious code is executed without user intervention
- Malicious code may be directed by a remote attacker once a connection is made
- Resources used by the malicious code (e.g. file names, URLs) are hard-coded in the binary
- Additional OS resources (processes, memory) are consumed by the malicious code
- How can we track this behavior dynamically?
  - ➢Pin!!

Information Flow Tracking

Approach

- ➢Track data sources and monitor information flow using Pin
- ➢Send program behavior to back end whenever suspicious program behavior is suspected
- ➢Provide analysis and policies to decide classify program behavior



Information Flow Tracking using Pin

- **Pin tracks information flow in the program and identifies exact source of data**
  - ➢**USER_INPUT**: data is retrieved via user interaction
  - ➢**FILE**: data is read from a file
  - ➢**SOCKET**: data is retrieved from socket interface
  - ➢**BINARY**: data is part of the program binary image
  - ➢**HARDWARE**: data originated from hardware
- **Pin maintains data source information for all memory locations and registers**
- **Propagates flow information by taking union of data sources of all operands**

Example – Register Tracking

- We track flow from source to destination operands

. . .

%ebx - {}

%ecx - {BINARY1} /* ecx contains information from BINARY1 */

%edx - {SOCKET1} /* edx contains information from SOCKET1 */

%esi - {FILE2} /* esi contains information from FILE2 */

%edi - {}

. . .

- Assume the following XOR instruction:

xor %edx,%esi

which has the following semantics:

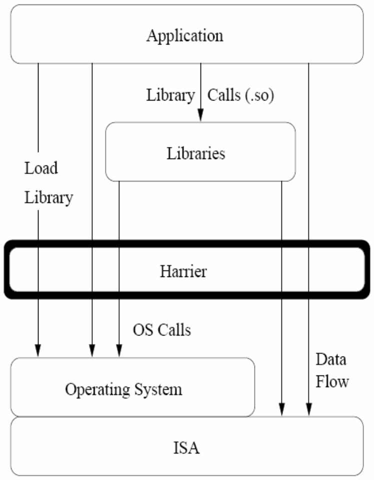
dst(%esi) := dst(%esi) XOR src(%edx)

- Pin will instrument this instruction and will insert an analysis routine to merge the source and destination operand information

%edx - {SOCKET1} /* edx contains information from SOCKET1 */ %esi - {SOCKET1, FILE2} /* esi contains information from FILE2 */

Information Flow Tracking using Pin

- **Different levels of abstraction**
- **Event Monitoring**
  - **Architectural Events**
    - Instructions executed
  - **OS Events**
    - System calls
  - **Library Events**
    - Library routines



Information Flow Tracking Prototype

System Calls

- – Instrument selected system calls (12 in prototype)

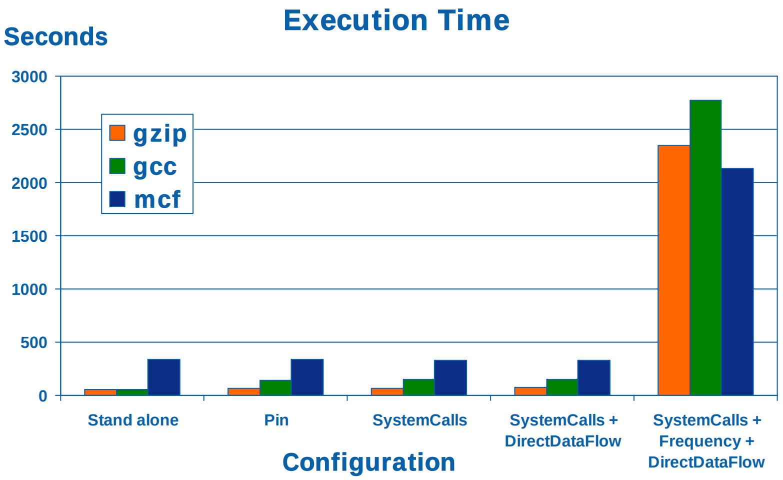
Code Frequency

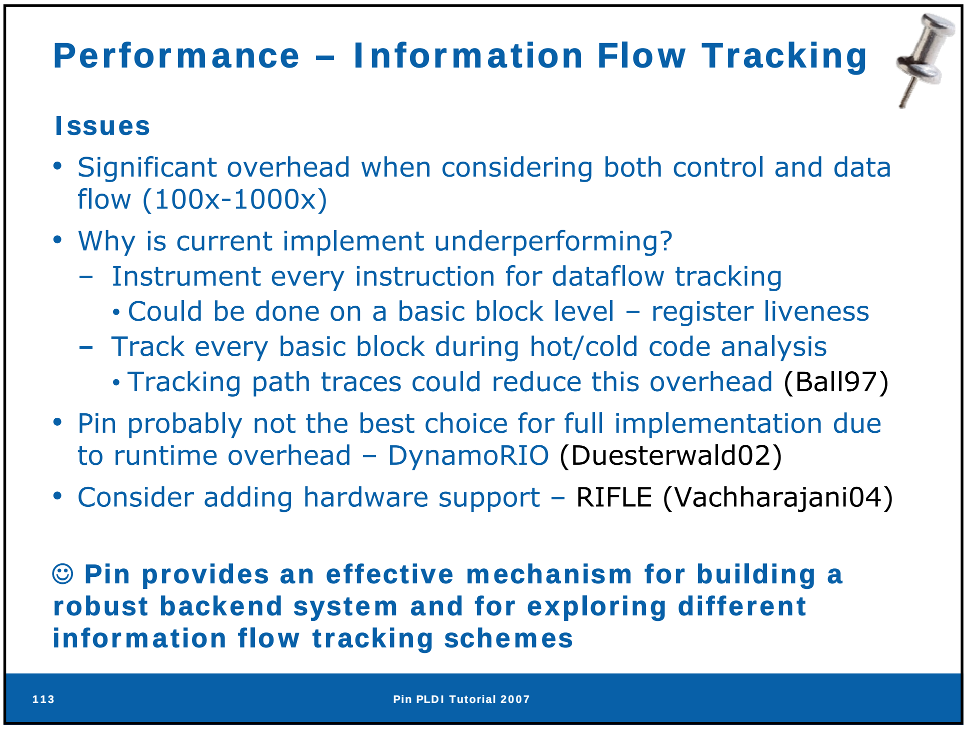
- – Instrument every basic block
- – Determine code “hotness”
- – Application binary vs. shared object

Program Data Flow

- System call specific data flow
  - – Tracking file loads, mapping memory to files ..
- Application data flow
  - – Instrument memory access instructions
  - – Instrument ALU instructions

Performance – Information Flow Tracking



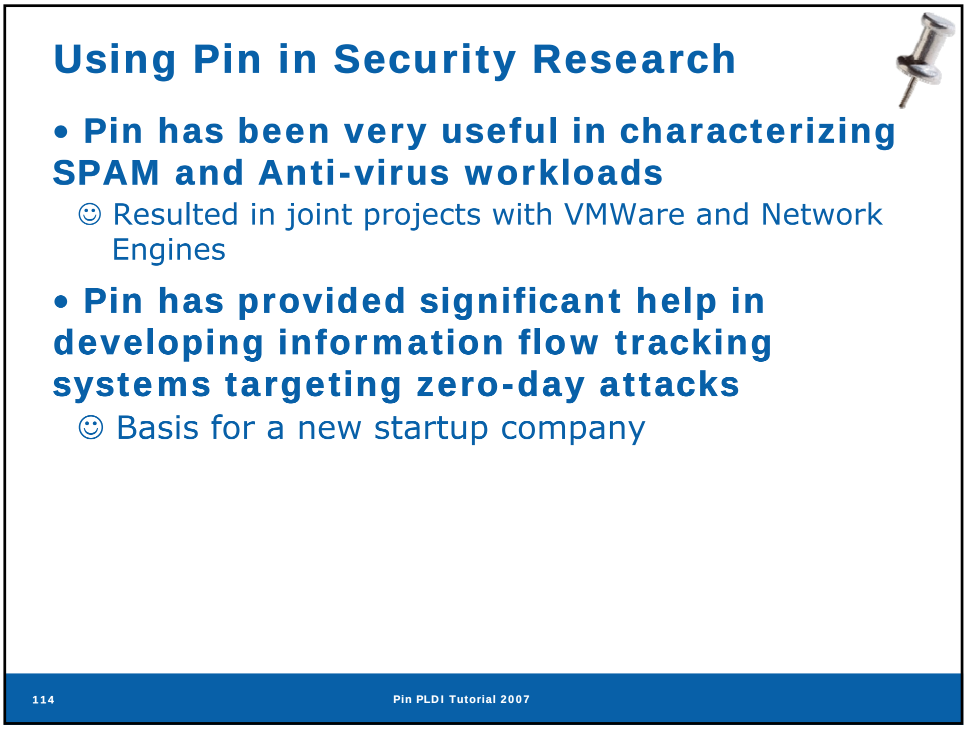


Performance – Information Flow Tracking

Issues

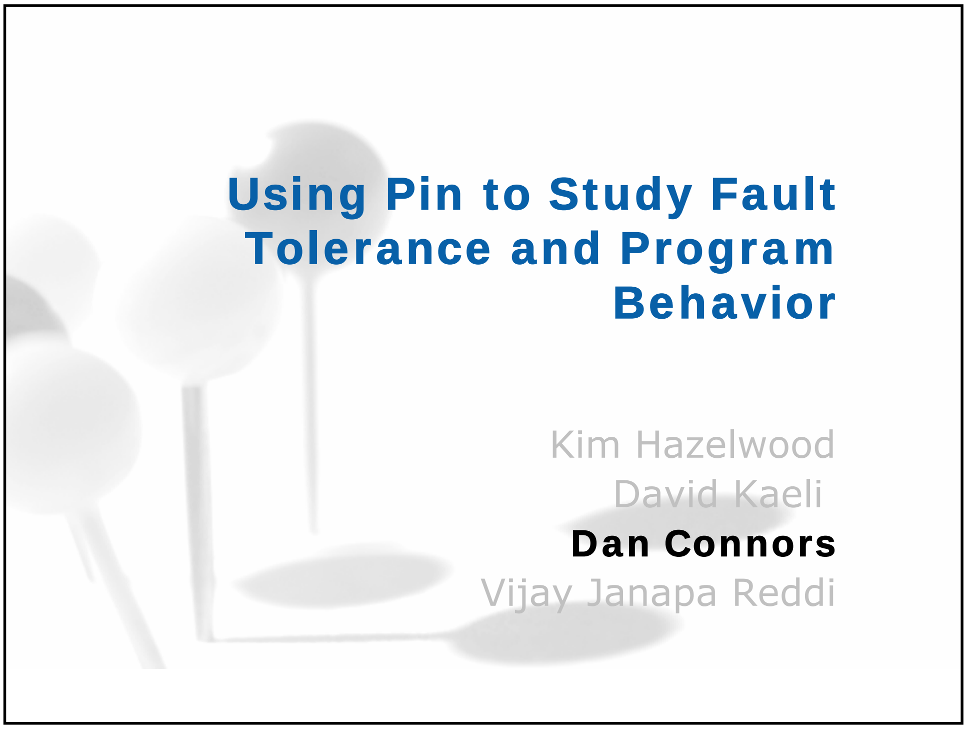
- Significant overhead when considering both control and data flow (100x-1000x)
- Why is current implement underperforming?
  - Instrument every instruction for dataflow tracking
    - Could be done on a basic block level – register liveness
  - Track every basic block during hot/cold code analysis
    - Tracking path traces could reduce this overhead (Ball97)
- Pin probably not the best choice for full implementation due to runtime overhead – DynamoRIO (Duesterwald02)
- Consider adding hardware support – RIFLE (Vachharajani04)

☺ Pin provides an effective mechanism for building a robust backend system and for exploring different information flow tracking schemes



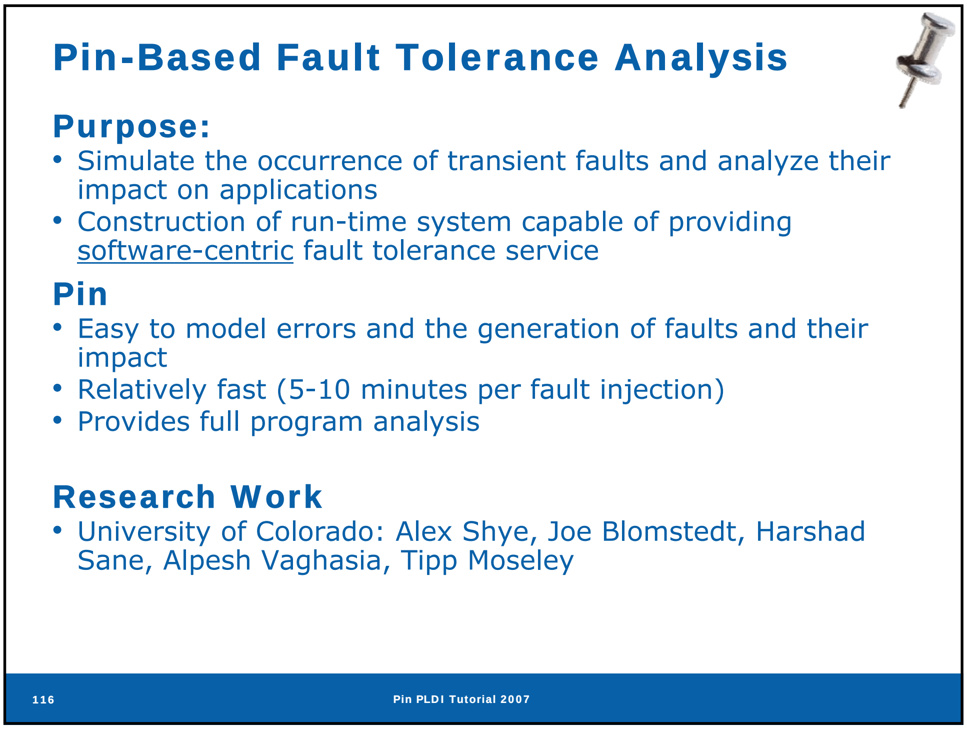
Using Pin in Security Research

- **Pin has been very useful in characterizing SPAM and Anti-virus workloads**
  - ☺ Resulted in joint projects with VMWare and Network Engines
- **Pin has provided significant help in developing information flow tracking systems targeting zero-day attacks**
  - ☺ Basis for a new startup company



Using Pin to Study Fault Tolerance and Program Behavior

Kim Hazelwood
David Kaeli
Dan Connors
Vijay Janapa Reddi



Pin-Based Fault Tolerance Analysis

Purpose:

- Simulate the occurrence of transient faults and analyze their impact on applications
- Construction of run-time system capable of providing software-centric fault tolerance service

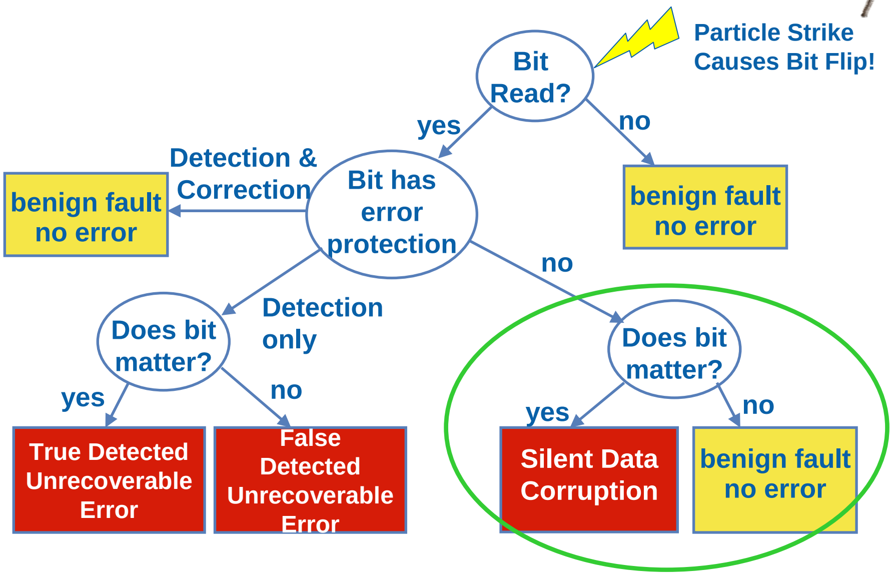
Pin

- Easy to model errors and the generation of faults and their impact
- Relatively fast (5-10 minutes per fault injection)
- Provides full program analysis

Research Work

- University of Colorado: Alex Shye, Joe Blomstedt, Harshad Sane, Alpesh Vaghasia, Tipp Moseley

Division of Transient Faults Analysis



Particle Strike Causes Bit Flip!

Bit Read?

yes

no

benign fault no error

Bit has error protection

Detection & Correction

benign fault no error

Detection only

Does bit matter?

yes

True Detected Unrecoverable Error

no

False Detected Unrecoverable Error

no

Does bit matter?

yes

Silent Data Corruption

no

benign fault no error

Modeling Microarchitectural Faults in Pin

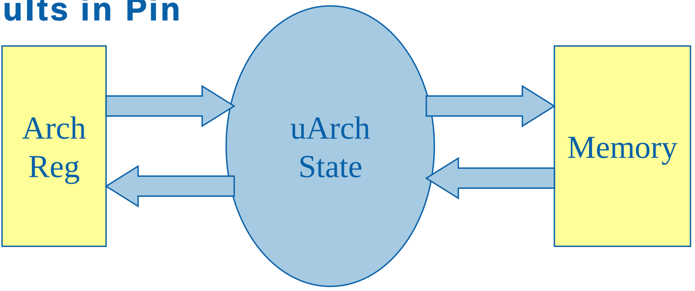
Accuracy of fault methodology depends on the complexity of the underlying system

- Microarchitecture, RTL, physical silicon

Build a microarchitectural model into Pin

- A low fidelity model may suffice
- Adds complexity and slows down simulation time

Emulate certain types of microarchitectural faults in Pin

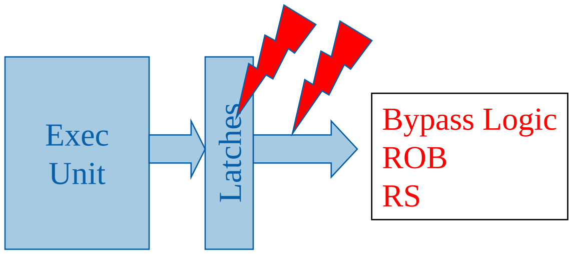


Example: Destination/Source Register Transmission Fault

Fault occurs in latches when forwarding instruction output

Change architectural value of destination register at the instruction where fault occurs

NOTE: This is different than inserting fault into register file because the destination is selected based on the instruction where fault occurs



Example: Load Data Transmission Faults

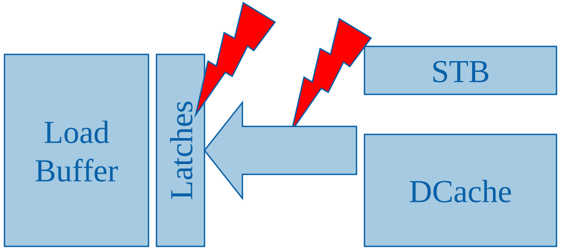
Fault occurs when loading data from the memory system

Before load instruction, insert fault into memory

Execute load instruction

After load instruction, remove fault from memory (Cleanup)

NOTE: This models a fault occurring in the transmission of data from the STB or L1 Cache



Steps for Fault Analysis

Determine 'WHEN' the error occurs

Determine 'WHERE' the error occurs

Inject Error

Determine/Analyze Outcome

Step: WHEN

Sample Pin Tool: InstCount.C

- Purpose: Efficiently determines the number of dynamic instances of each static instruction

Output: For each static instruction

- Function name
- Dynamic instructions per static instruction

```
IP: 135000941 Count: 492714322 Func: propagate_block.104
IP: 135000939 Count: 492714322 Func: propagate_block.104
IP: 135000961 Count: 492701800 Func: propagate_block.104
IP: 135000959 Count: 492701800 Func: propagate_block.104
IP: 135000956 Count: 492701800 Func: propagate_block.104
IP: 135000950 Count: 492701800 Func: propagate_block.104
```

Step: WHEN

InstProf.C

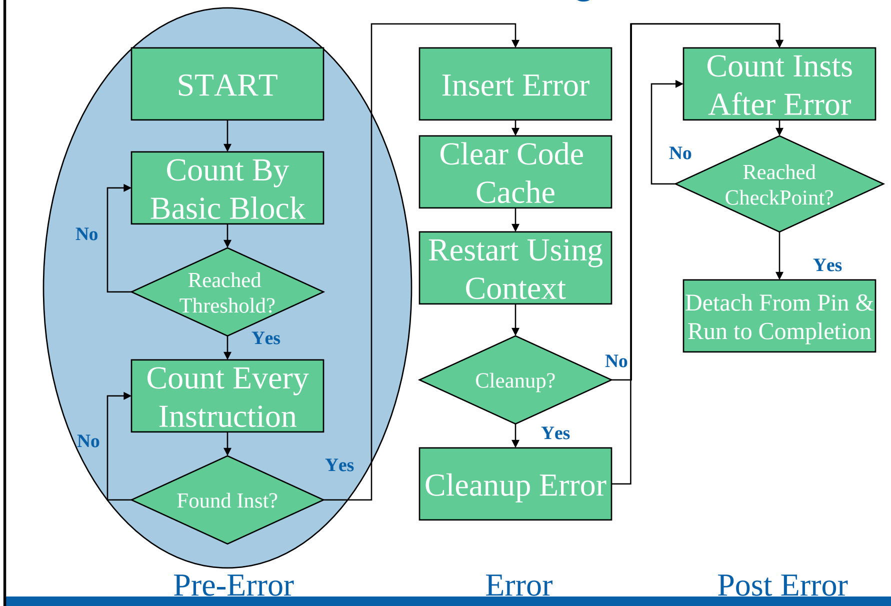
- Purpose: Traces basic blocks for contents and execution count

Output: For a program input

- Listing of dynamic block executions
- Used to generate a profile to select error injection point (opcode, function, etc)

```
BBL NumIns: 6     Count: 13356     Func: build_tree
804cb88  BINARY  ADD      [Dest: ax]  [Src: ax edx]  MR: 1 MW: 0
804cb90  SHIFT    SHL     [Dest: eax]  [Src: eax]  MR: 0 MW: 0
804cb92  DATAXFER MOV  [Dest:]  [Src: esp edx ax] MR: 0 MW: 1
804cb97  BINARY   INC     [Dest: edx]  [Src: edx]  MR: 0  MW: 0
804cb98  BINARY  CMP     [Dest:]  [Src: edx]  MR: 0  MW: 0
804cb9b  COND_BR JLE    [Dest:]  [Src:]  MR: 0  MW: 0
```

Error Insertion State Diagram



START

Count By Basic Block

Reached Threshold?

No

Yes

Count Every Instruction

Found Inst?

No

Yes

Insert Error

Clear Code Cache

Restart Using Context

Cleanup?

No

Yes

Cleanup Error

Count Insts After Error

Reached CheckPoint?

No

Yes

Detach From Pin & Run to Completion

Pre-Error

Error

Post Error

Step: WHERE

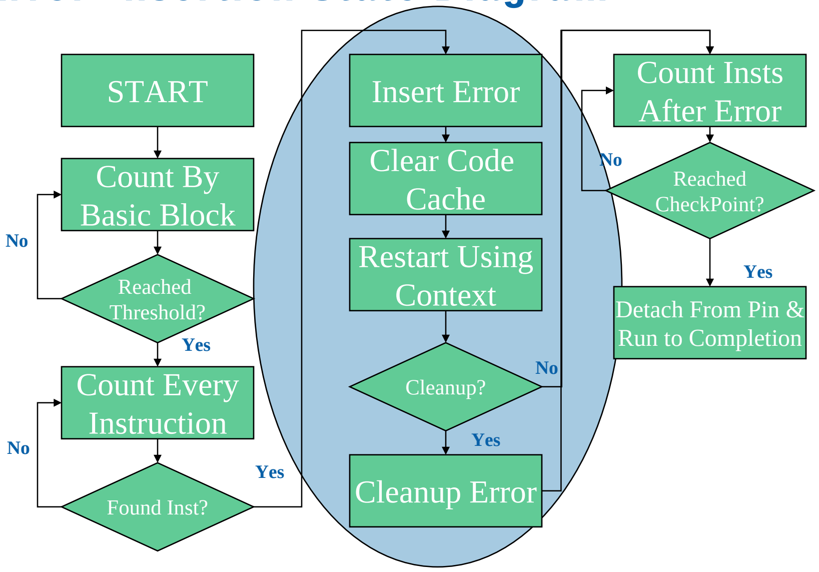
Reality:

- Where the transient fault occurs is a function of the size of the structure on the chip
- Faults can occur in both architectural and microarchitectural state

Approximation:

- Pin only provides architectural state, not microarchitectural state (no uops, for instance)
  - Either inject faults only into architectural state
  - Build an approximation for some microarchitectural state

Error Insertion State Diagram



START

Count By Basic Block

Reached Threshold?

No

Yes

Count Every Instruction

Found Inst?

No

Yes

Insert Error

Clear Code Cache

Restart Using Context

Cleanup?

No

Yes

Cleanup Error

Count Insts After Error

Reached CheckPoint?

No

Yes

Detach From Pin & Run to Completion

Pre-Error Error Post Error

Step: Injecting Error

Error Insertion Routine

```
VOID InsertFault(CONTEXT* _ctxt) {
   srand(curDynInst);
   GetFaultyBit(_ctxt, &faultReg, &faultBit);

   UINT32 old_val;   UINT32 new_val;
   old_val = PIN_GetContextReg(_ctxt, faultReg);
   faultMask = (1 << faultBit);
   new_val = old_val ^ faultMask;
   PIN_SetContextReg(_ctxt, faultReg, new_val);

   PIN_RemoveInstrumentation();
   faultDone = 1;
   PIN_ExecuteAt(_ctxt);
}
```

Step: Determining Outcome

Outcomes that can be tracked:

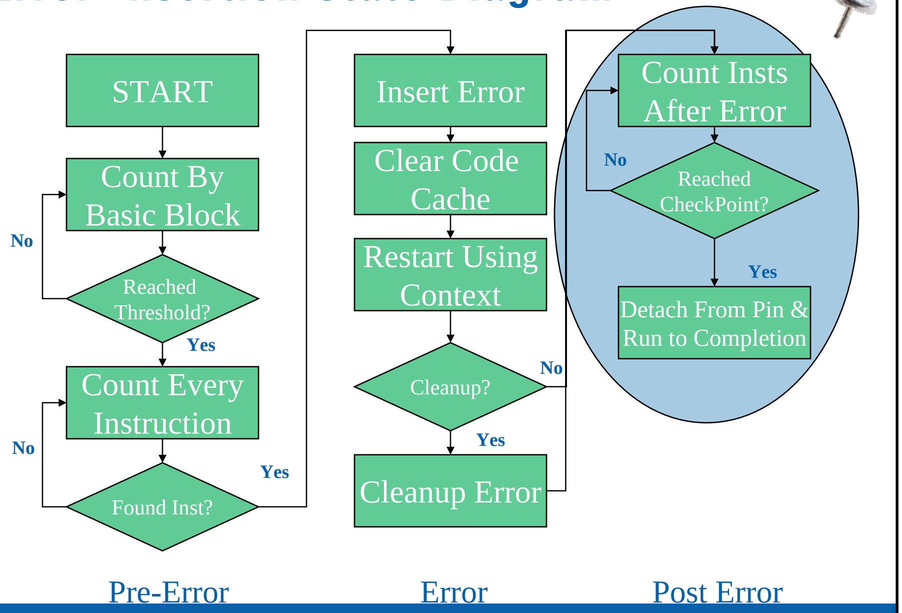
- Did the program complete?
- Did the program complete and have the correct IO result?
- If the program crashed, how many instructions were executed after fault injection before program crashed?
- If the program crashed, why did it crash (trapping signals)?

Register Fault Pin Tool: RegFault.C

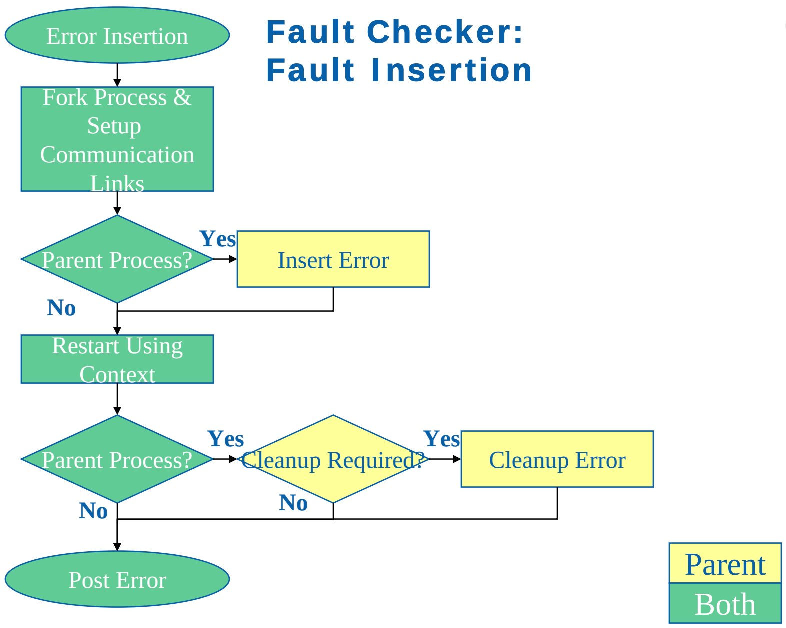
MAIN

```
main(int argc, char * argv[]) {
   if (PIN_Init(argc, argv))
      return Usage();
   out_file.open(KnobOutputFile.Value().c_str());
   faultInst = KnobFaultInst.Value();

   TRACE_AddInstrumentFunction (Trace, 0);
   INS_AddInstrumentFunction(Instruction, 0);

   PIN_AddFiniFunction(Fini, 0);

   PIN_AddSignalInterceptFunction(SIGSEGV, SigFunc, 0);
   PIN_AddSignalInterceptFunction(SIGFPE, SigFunc, 0);
   PIN_AddSignalInterceptFunction(SIGILL, SigFunc, 0);
   PIN_AddSignalInterceptFunction(SIGSYS, SigFunc, 0);

   PIN_StartProgram();
   return 0;
}
```

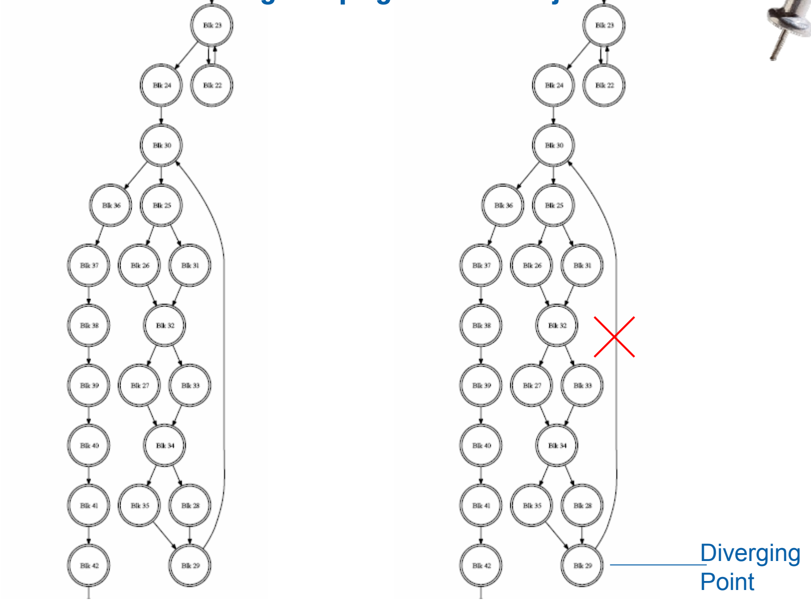
Error Insertion State Diagram



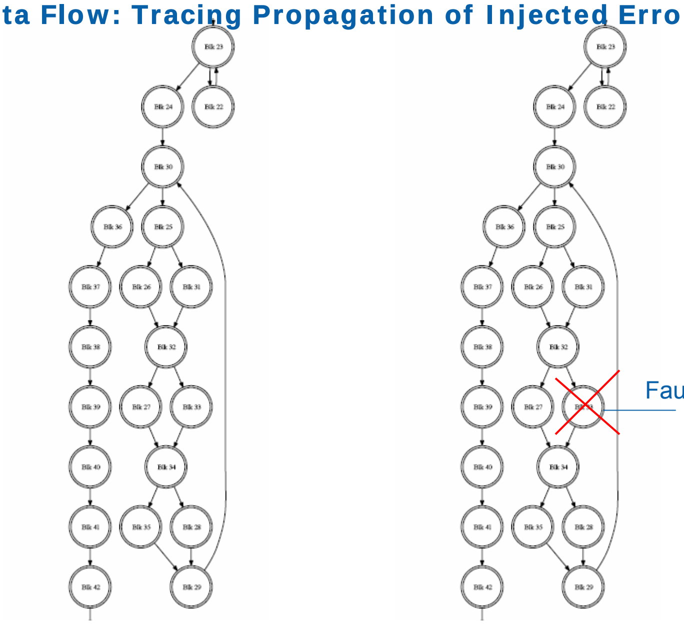
Fault Checker: Fault Insertion



Control Flow: Tracing Propagation of Injected Errors

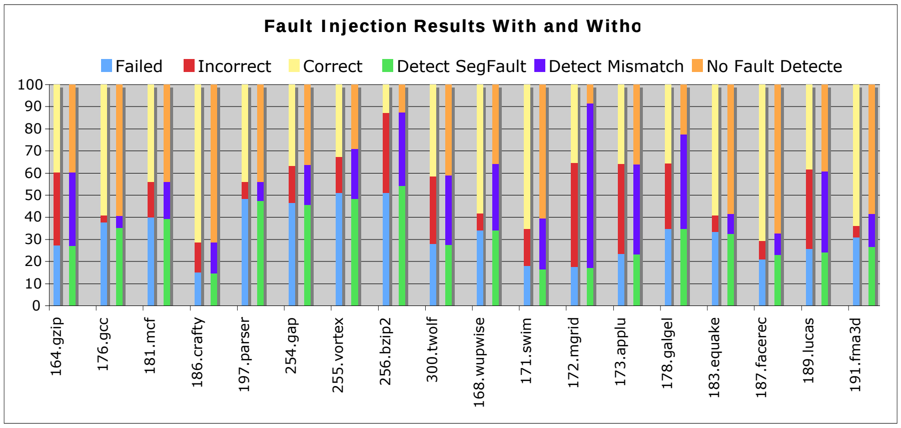


Data Flow: Tracing Propagation of Injected Errors



Fault Detection

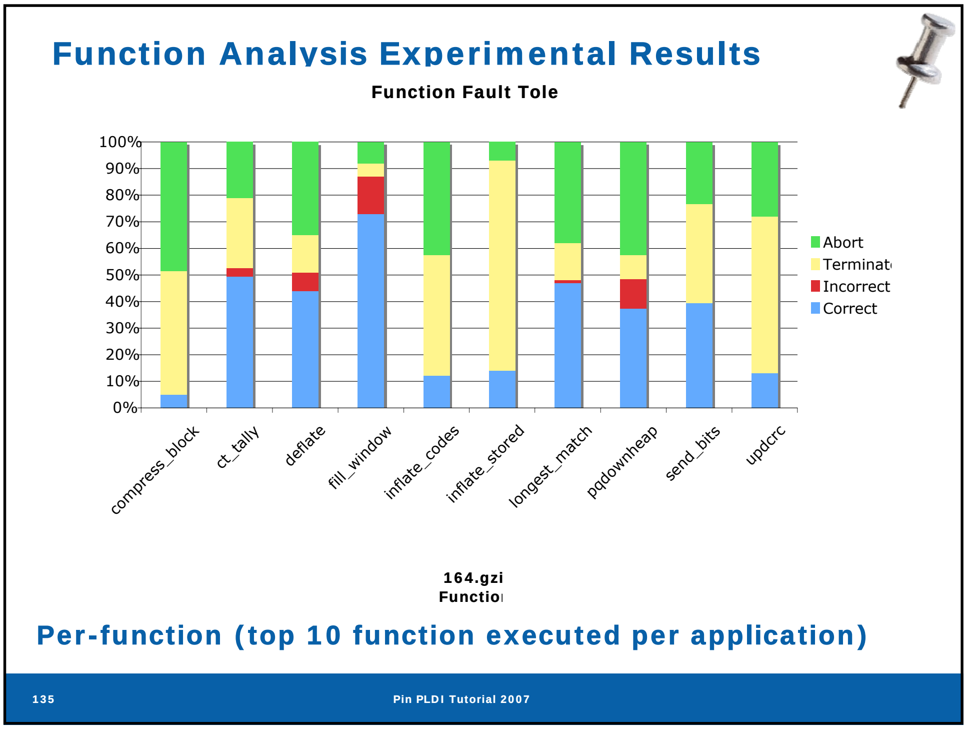
Fault Coverage Experimental Results



Watchdog timeout very rare so not shown

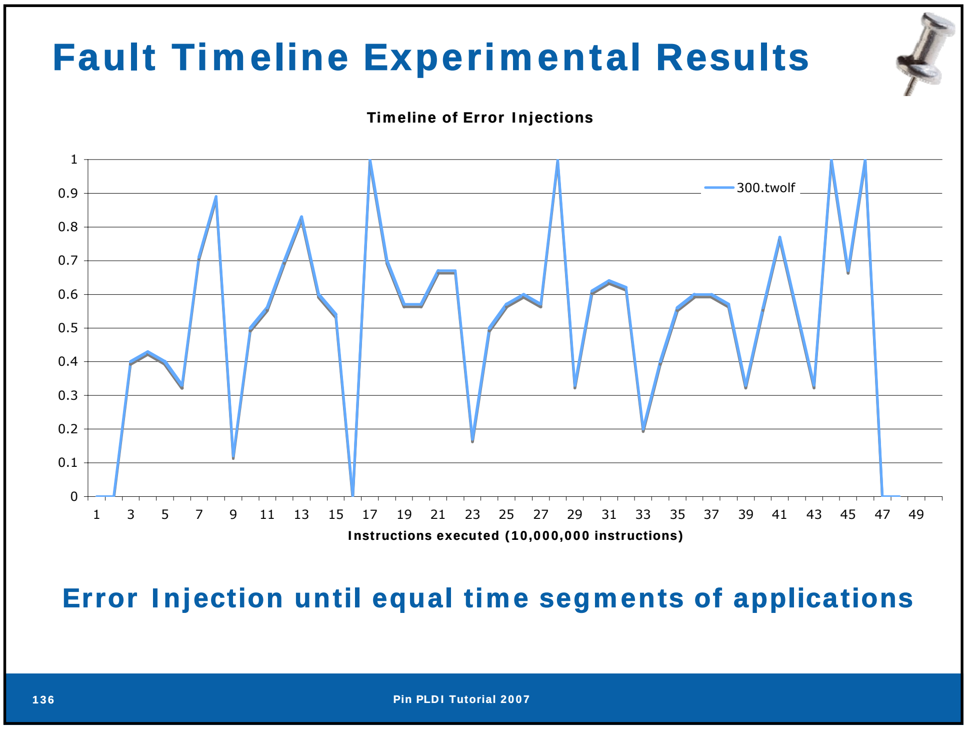
PLR detects all *Incorrect* and *Failed* cases

Effectively detects relevant faults and ignores benign faults



Function Analysis Experimental Results

Per-function (top 10 function executed per application)



Fault Timeline Experimental Results

Error Injection until equal time segments of applications

Run-time System for Fault Tolerance

Process technology trends

- Single transistor error rate is expected to stay close to constant
- Number of transistors is increasing exponentially with each generation

Transient faults will be a problem for microprocessors!

Hardware Approaches

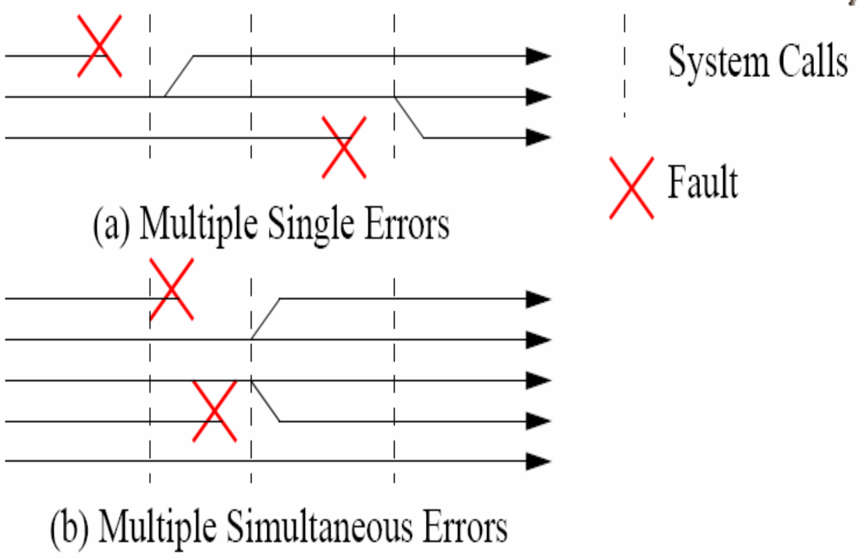
- Specialized redundant hardware, redundant multi-threading

Software Approaches

- Compiler solutions: instruction duplication, control flow checking
- Low-cost, flexible alternative but higher overhead

Goal:* *Leverage available hardware parallelism in multi-core architectures to improve the performance of software-based transient fault tolerance

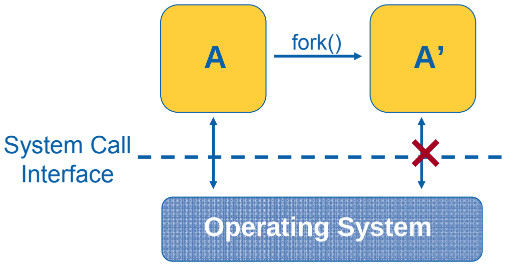
Process-level Redundancy



(a) Multiple Single Errors

(b) Multiple Simultaneous Errors

Replicating Processes



Straight-forward and fast

- fork()
- Let OS schedule to cores

Maintain transparency in replica

✗ System calls
✗ Shared memory R/W

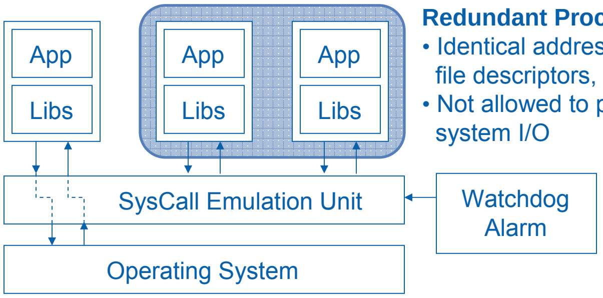
Replicas provide an extra copy of the program+input

What can we do with this?

- Software transient fault tolerance
- Low-overhead program instrumentation
- More?

Process-Level Redundancy (PLR)

Master Process
- Only process allowed to perform system I/O



Redundant Processes
- Identical address space, file descriptors, etc.
- Not allowed to perform system I/O

System Call Emulation Unit

Creates redundant processes

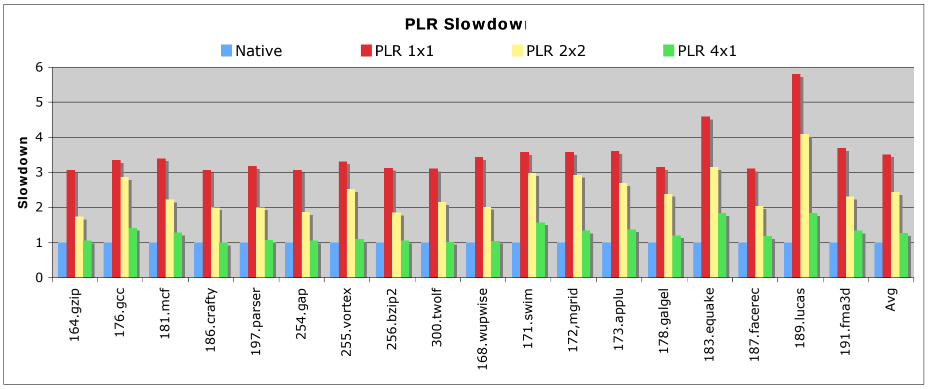
Barrier synchronize at all system calls

Emulates system calls to guarantee determinism among all processes

Detects and recovers from transient faults

Watchdog Alarm
- Occasionally a process will hang
- Set at beginning of barrier synchronization to ensure that all processes are alive

PLR Performance



Performance for single processor (PLR 1x1), 2 SMT processors (PLR 2x1) and 4 way SMP (PLR 4x1)

Slowdown for 4-way SMP only 1.26x

Conclusion

Fault insertion using Pin is a great way to determine the impacts faults have within an application

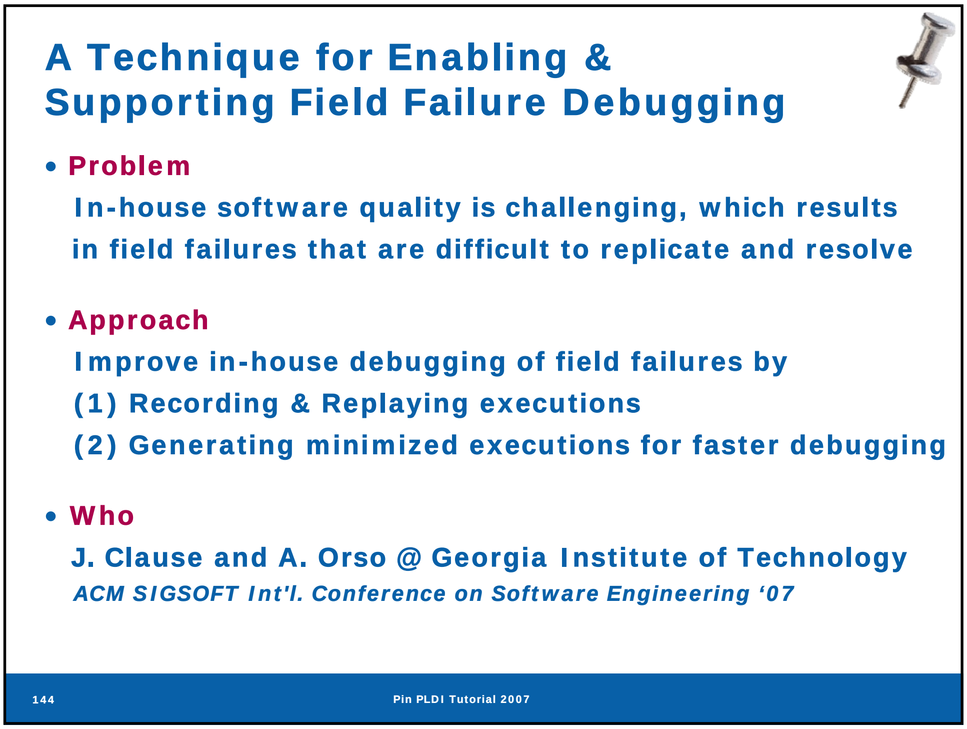
- Easy to use
- Enables full program analysis
- Accurately describes fault behavior once it has reached architectural state

Transient fault tolerance at 30% overhead

- Future work
  - Support non-determinism (shared memory, interrupts, multi-threading)
  - Fault coverage-performance trade-off in switching on/off

Pin-based Projects in Academia

Kim Hazelwood
David Kaeli
Dan Connors
Vijay Janapa Reddi



A Technique for Enabling & Supporting Field Failure Debugging

- **Problem**

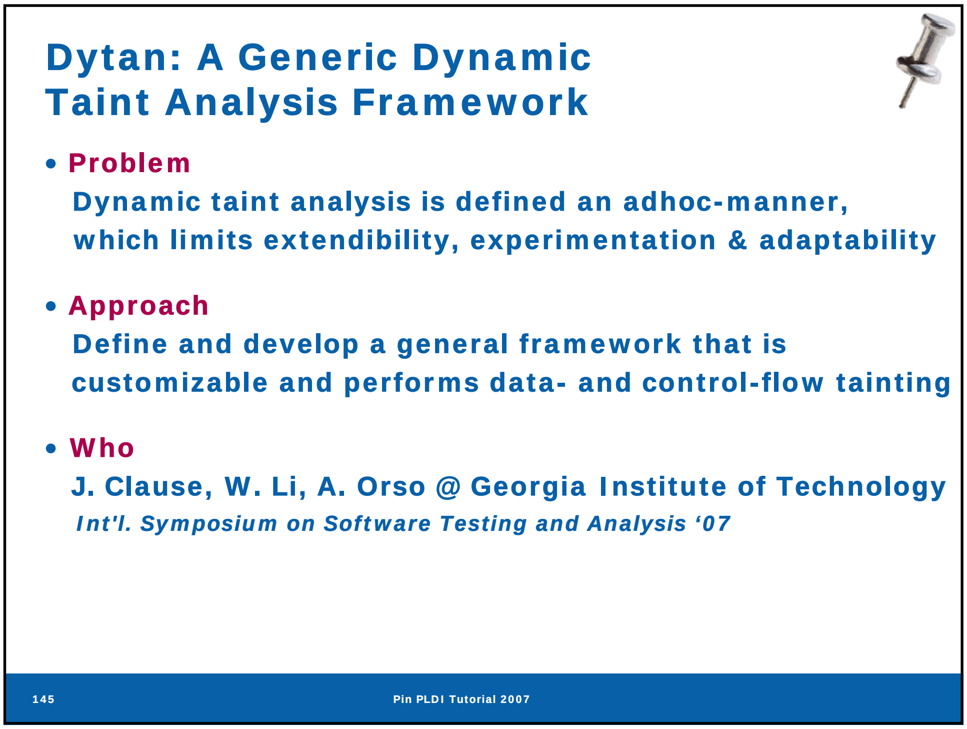
  In-house software quality is challenging, which results in field failures that are difficult to replicate and resolve

- **Approach**

  Improve in-house debugging of field failures by
  (1) Recording & Replaying executions
  (2) Generating minimized executions for faster debugging

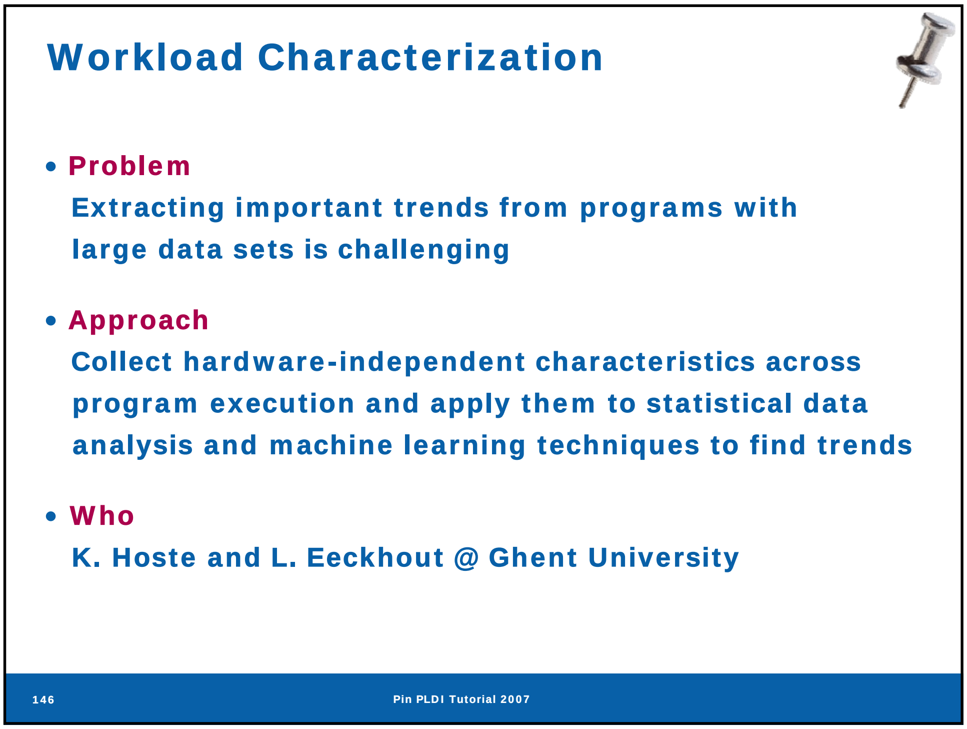
- **Who**

  J. Clause and A. Orso @ Georgia Institute of Technology
  ACM SIGSOFT Int'l. Conference on Software Engineering '07



Dytan: A Generic Dynamic Taint Analysis Framework

- **Problem**
  Dynamic taint analysis is defined an adhoc-manner, which limits extendibility, experimentation & adaptability
- **Approach**
  Define and develop a general framework that is customizable and performs data- and control-flow tainting
- **Who**
  J. Clause, W. Li, A. Orso @ Georgia Institute of Technology
  Int'l. Symposium on Software Testing and Analysis '07



Workload Characterization

- **Problem**
  Extracting important trends from programs with large data sets is challenging
- **Approach**
  Collect hardware-independent characteristics across program execution and apply them to statistical data analysis and machine learning techniques to find trends
- **Who**
  K. Hoste and L. Eeckhout @ Ghent University

Loop-Centric Profiling

- **Problem**

  Identifying parallelism is difficult

- **Approach**

  Provide a hierarchical view of how much time is spent in loops, and the loops nested within them using (1) instrumentation and (2) light-weight sampling to automatically identify opportunities of parallelism

- **Who**

  T. Moseley, D. Connors, D. Grunwald, R. Peri @
  University of Colorado, Boulder and Intel Corporation

  Int'l. Conference on Computing Frontiers (CF) '07

Shadow Profiling

- **Problem**

  Attaining accurate profile information results in large overheads for runtime & feedback-directed optimizers

- **Approach**

  `fork()` shadow copies of an application onto spare cores, which can be instrumented aggressively to collect accurate information without slowing the parent process

- **Who**

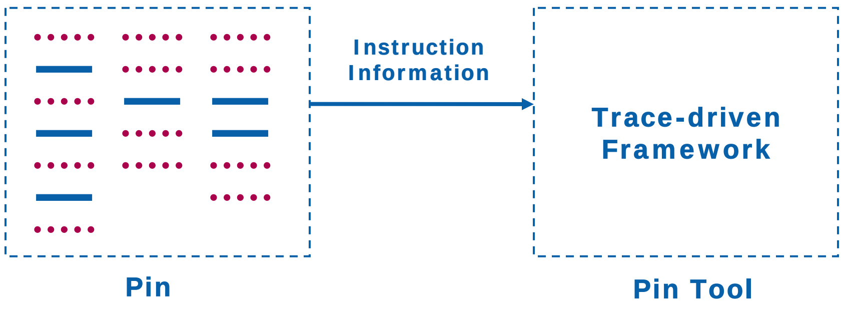
  T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, R. Peri
  University of Colorado, Boulder and Intel Corporation

  Int'l. Conference on Code Generation and Optimization (CGO) '07

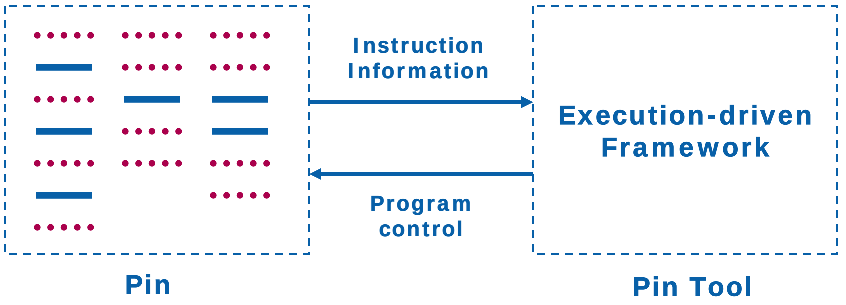
Part Four: Exploratory Extensions and Hands-On Workshop

Kim Hazelwood
Dan Connors
David Kaeli
Vijay Janapa Reddi

Common use of Pin



Driving execution using Pin



Session Objectives

- Building and Running Pin Tools
- Understanding program execution using Pin

Program Instruction Stream

Memory

Machine State

- Putting it all together: Transactional Memory

Structure of a Pin Tool

Pin Tool traces Virtual Addresses

```
FILE * trace;

VOID RecordMemWrite(VOID * ip, VOID * va, UINT32 size) {
  fprintf(trace,"%p: W %p %d\n", ip, va, size);
}
```

Analysis

```
VOID Instruction(INS ins, VOID *v) {
  if (INS_IsMemoryWrite(ins)) {
     INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
                    IARG_INST_PTR,
                    IARG_MEMORYWRITE_VA,
                    IARG_MEMORYWRITE_SIZE, IARG_END);
  }
}
```

Instrumentation

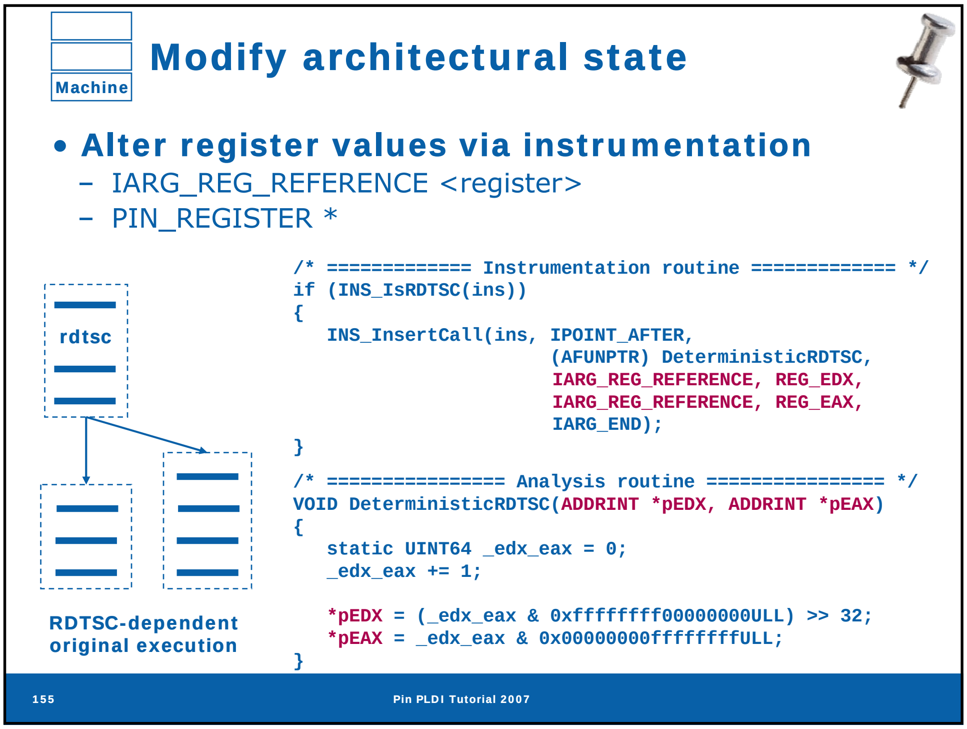
```
int main(int argc, char *argv[]) {
  PIN_Init(argc, argv);
  trace = fopen("atrace.out", "w");
  INS_AddInstrumentFunction(Instruction, 0);
  PIN_StartProgram();
  return 0;
}
```

Callback Registration

Machine

Architectural State Interposition

- **Observe instruction operands and their values**
  - IARG_BRANCH_TAKEN, IARG_REG_VALUE, IARG_CONTEXT, …
- **Modify register values**
- **Save and restore state**
- **Instruction emulation**



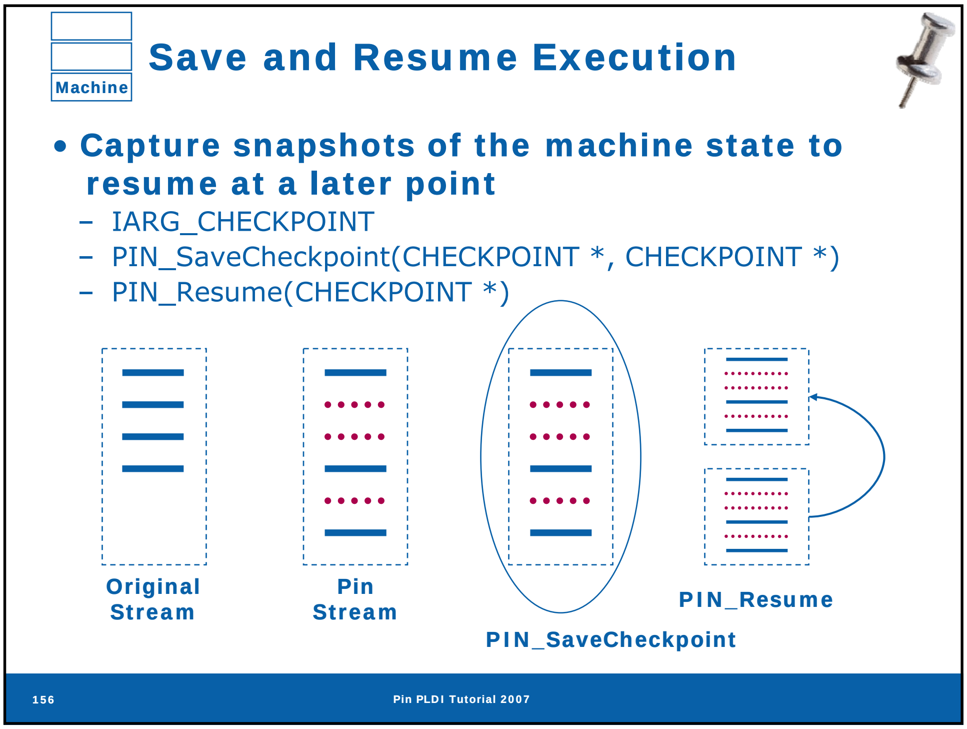
Machine

Modify architectural state

- **Alter register values via instrumentation**
  - IARG_REG_REFERENCE <register>
  - PIN_REGISTER *

```
/* ============= Instrumentation routine ============= */
if (INS_IsRDTSC(ins))
{
   INS_InsertCall(ins, IPOINT_AFTER,
                     (AFUNPTR) DeterministicRDTSC,
                     IARG_REG_REFERENCE, REG_EDX,
                     IARG_REG_REFERENCE, REG_EAX,
                     IARG_END);
}
/* ================ Analysis routine ================ */
VOID DeterministicRDTSC(ADDRINT *pEDX, ADDRINT *pEAX)
{
   static UINT64 _edx_eax = 0;
   _edx_eax += 1;

   *pEDX = (_edx_eax & 0xffffffff00000000ULL) >> 32;
   *pEAX = _edx_eax & 0x00000000ffffffffULL;
}
```

RDTSC-dependent original execution



Machine

Save and Resume Execution

- **Capture snapshots of the machine state to resume at a later point**
  - IARG_CHECKPOINT
  - PIN_SaveCheckpoint(CHECKPOINT *, CHECKPOINT *)
  - PIN_Resume(CHECKPOINT *)

Original Stream

Pin Stream

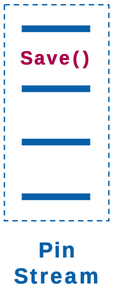
PIN_SaveCheckpoint

PIN_Resume

Machine

Save and Resume Execution (2)

- **IARG_CHECKPOINT**
  - Pin generates a snapshot (includes instrumented state)
- **PIN_SaveCheckpoint (CHECKPOINT *src, CHECKPOINT *dst)**
  - Extract and copy state from handle(src) to local buffer(dst)

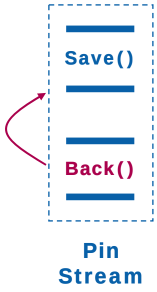


```
/* ========== Instrumentation routine ========== */
INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) Save,
                    IARG_CHECKPOINT,
                    IARG_END);


/* ============= Analysis routine ============= */
CHECKPOINT ckpt;

VOID Save(CHECKPOINT* _ckpt)
{
   PIN_SaveCheckpoint(_ckpt, &ckpt);
}
```

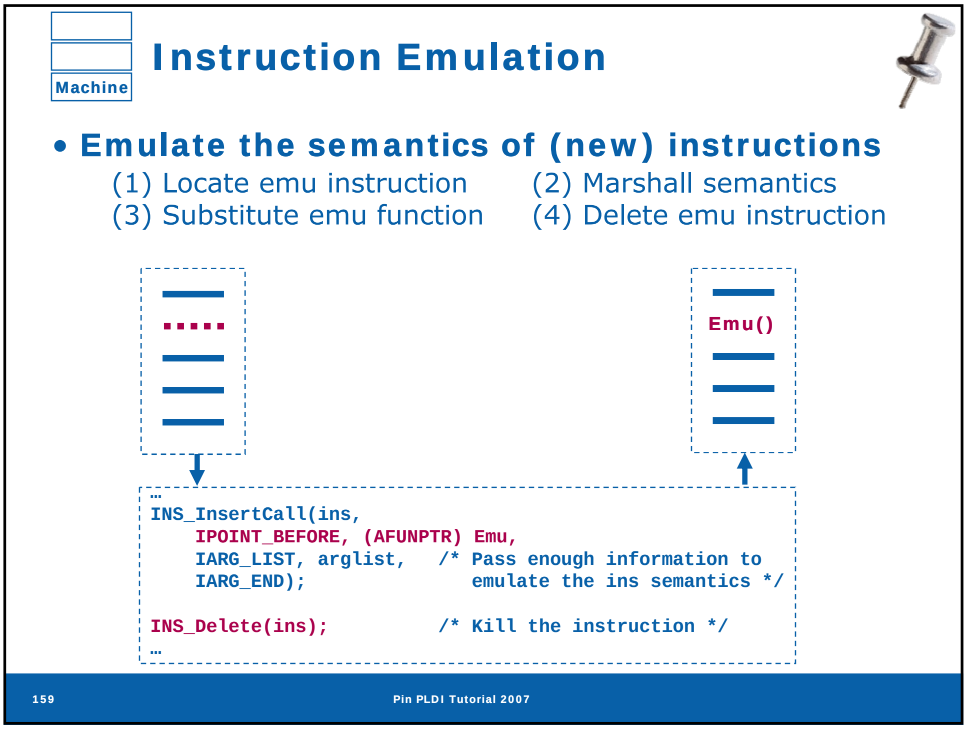
Machine

Save and Resume Execution (3)

- **PIN_Resume(CHECKPOINT *)**
  - Restore processor state to saved checkpoint
  - Continue execution



```
/* =========== Instrumentation routine =========== */
INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) Back,
                    IARG_END);


/* ============== Analysis routine ============== */
CHECKPOINT ckpt;

VOID Back()
{
   PIN_Resume(&ckpt);

   assert(false); /* PIN_Resume does not return! */
}
```



Machine

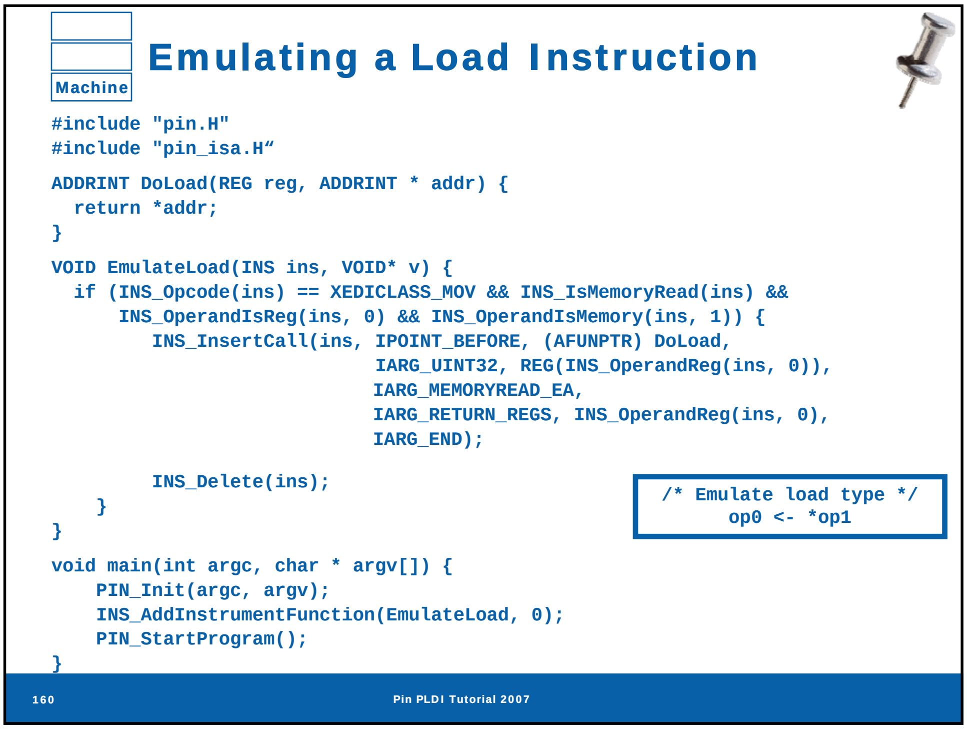
Instruction Emulation

- **Emulate the semantics of (new) instructions**

  (1) Locate emu instruction (2) Marshall semantics
  (3) Substitute emu function (4) Delete emu instruction

Emu()

```
…
INS_InsertCall(ins,
    IPOINT_BEFORE, (AFUNPTR) Emu,
    IARG_LIST, arglist,   /* Pass enough information to
    IARG_END);               emulate the ins semantics */

INS_Delete(ins);          /* Kill the instruction */
…
```



Machine

Emulating a Load Instruction

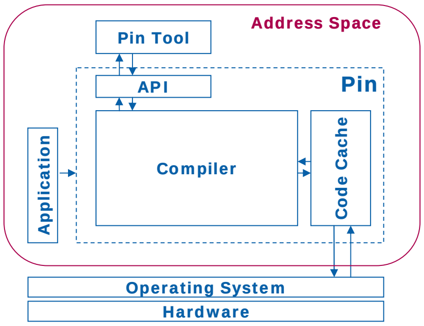
```
#include "pin.H"
#include "pin_isa.H“

ADDRINT DoLoad(REG reg, ADDRINT * addr) {
  return *addr;
}

VOID EmulateLoad(INS ins, VOID* v) {
  if (INS_Opcode(ins) == XEDICLASS_MOV && INS_IsMemoryRead(ins) &&
      INS_OperandIsReg(ins, 0) && INS_OperandIsMemory(ins, 1)) {
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) DoLoad,
                             IARG_UINT32, REG(INS_OperandReg(ins, 0)),
                             IARG_MEMORYREAD_EA,
                             IARG_RETURN_REGS, INS_OperandReg(ins, 0),
                             IARG_END);

         INS_Delete(ins);
    }
}

void main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(EmulateLoad, 0);
    PIN_StartProgram();
}
```

```
/* Emulate load type */
     op0 <- *op1
```

Memory

Memory Behavior

- **Memory access tracing**
  - IARG_MEMORYREAD_EA, IARG_MEMORYWRITE_EA, …
- **Modify program memory**
  - Pin Tool resides in the process’ address space

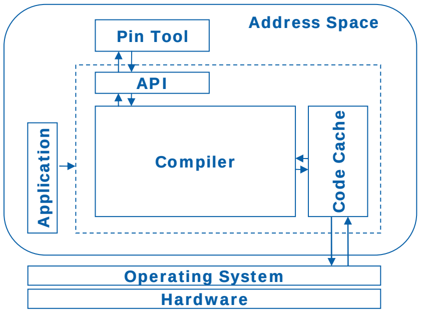


Change memory directly
(*addr = 0x123 **)**

Program

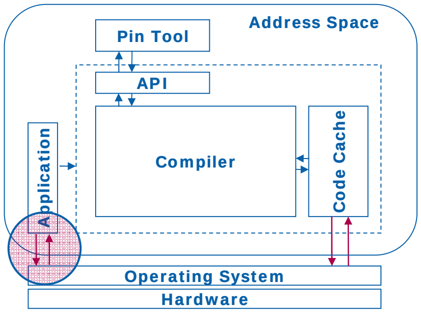
Controlling Program Execution

Pin (JIT)



Only translated code cached in the *Code Cache* is executed
Pros : Complete coverage
Cons: Slow

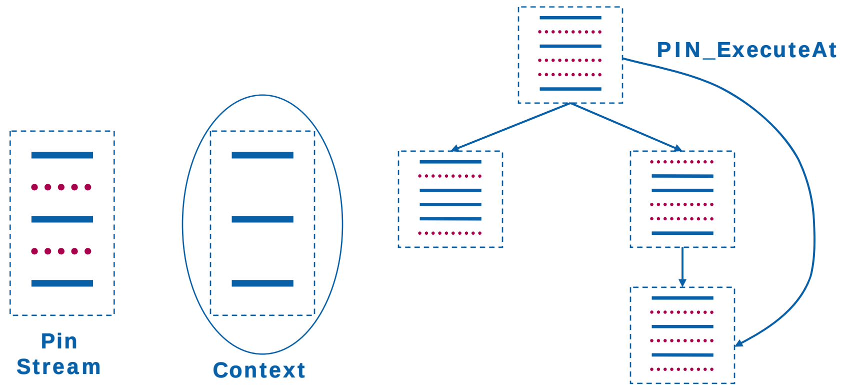
Pin (Probes)



Original code, and translated code are executed intermixed with one another
Pros : Fast
Cons: Limited coverage

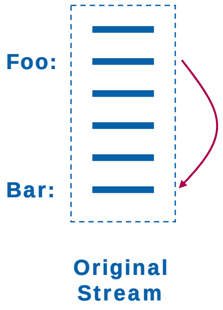
Executing @ Arbitrary Locations

- **JIT-mode** **(execute only translated code)**
  - IARG_CONTEXT
  - PIN_ExecuteAt (CONTEXT *)

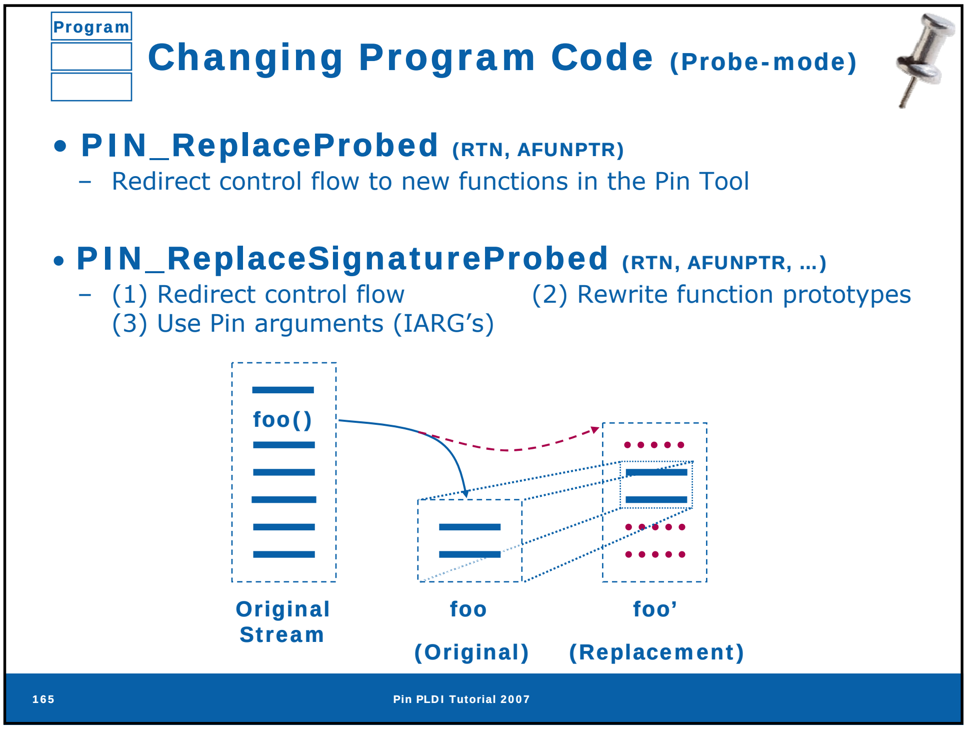


Executing @ Arbitrary Locations (2)

- **IARG_CONTEXT**
  - Pin generates program’s perception of machine state
- **Pin_ExceuteAt** **(CONTEXT *)**
  - Continue executing at context state



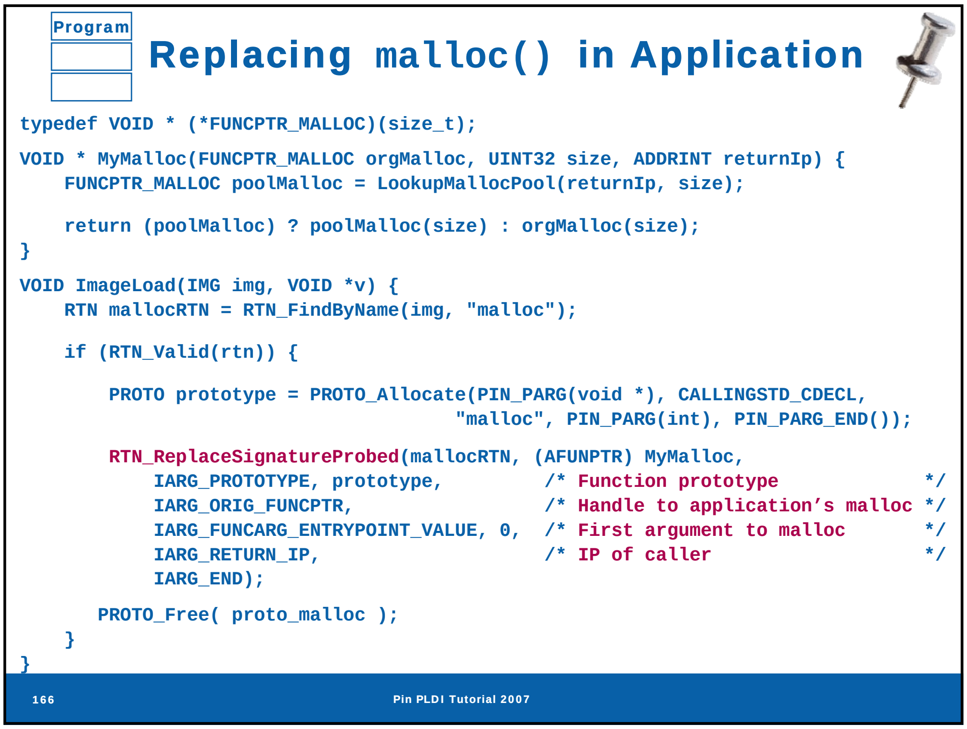
```
/* ========== Instrumentation routine ========== */
if (INS_Address(ins) == 0x40000000 /* Foo: */)
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) Jmp2Bar,
                      IARG_CONTEXT,
                      IARG_END);


/* ============= Analysis routine ============= */
VOID Jmp2Bar(CONTEXT *ctxt)
{
   PIN_SetContextReg(ctxt, REG_INST_PTR, Bar);
   PIN_ExecuteAt(ctxt);

   assert(false); /* PIN_ExecuteAt does not return! */
}
```



Program

Changing Program Code (Probe-mode)

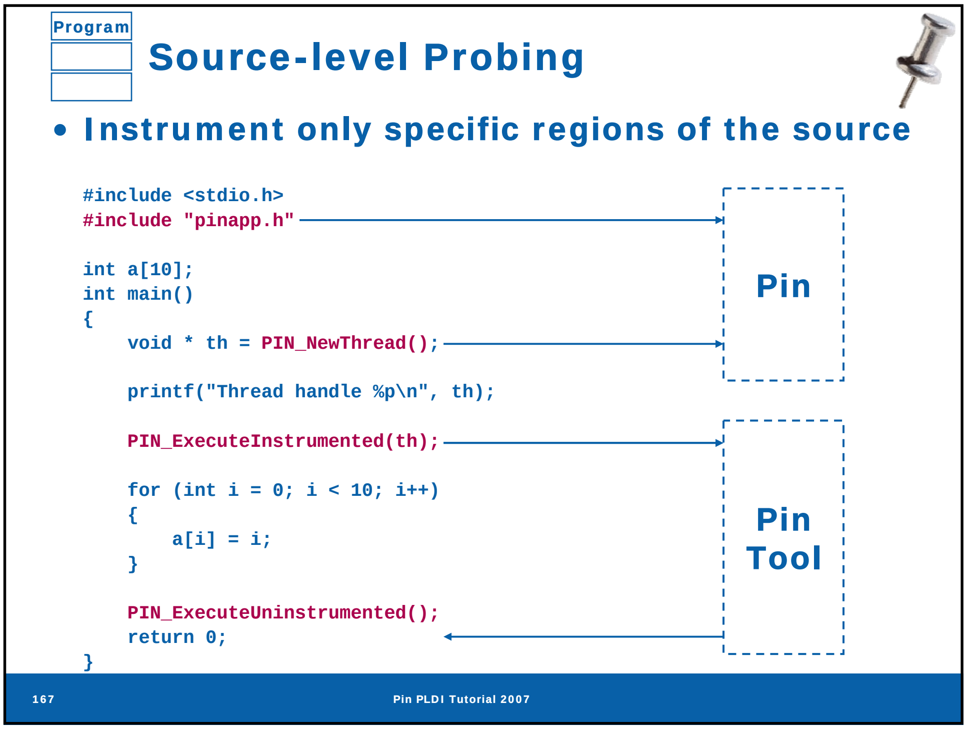
- **PIN_ReplaceProbed** **(RTN, AFUNPTR)**
  - Redirect control flow to new functions in the Pin Tool
- **PIN_ReplaceSignatureProbed** **(RTN, AFUNPTR, …)**
  - (1) Redirect control flow (2) Rewrite function prototypes
    (3) Use Pin arguments (IARG’s)



Program

Replacing malloc() in Application

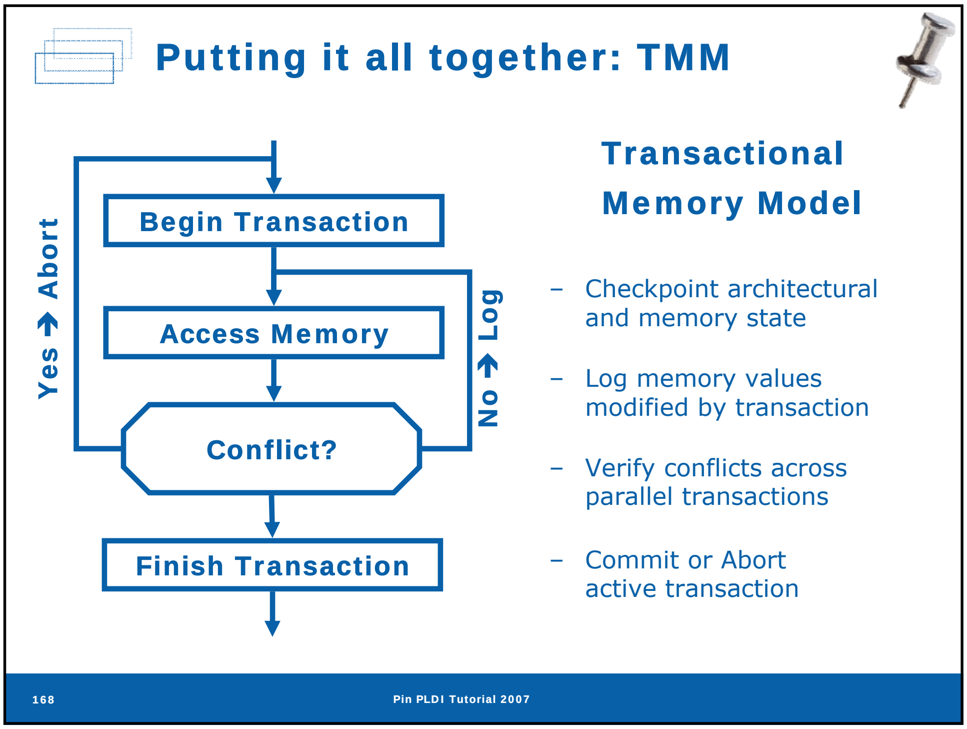
```
typedef VOID * (*FUNCPTR_MALLOC)(size_t);

VOID * MyMalloc(FUNCPTR_MALLOC orgMalloc, UINT32 size, ADDRINT returnIp) {
    FUNCPTR_MALLOC poolMalloc = LookupMallocPool(returnIp, size);

    return (poolMalloc) ? poolMalloc(size) : orgMalloc(size);
}

VOID ImageLoad(IMG img, VOID *v) {
    RTN mallocRTN = RTN_FindByName(img, "malloc");

    if (RTN_Valid(rtn)) {

        PROTO prototype = PROTO_Allocate(PIN_PARG(void *), CALLINGSTD_CDECL,
                                         "malloc", PIN_PARG(int), PIN_PARG_END());

        RTN_ReplaceSignatureProbed(mallocRTN, (AFUNPTR) MyMalloc,
            IARG_PROTOTYPE, prototype,          /* Function prototype           */
            IARG_ORIG_FUNCPTR,                  /* Handle to application’s malloc */
            IARG_FUNCARG_ENTRYPOINT_VALUE, 0,   /* First argument to malloc      */
            IARG_RETURN_IP,                     /* IP of caller                 */
            IARG_END);

       PROTO_Free( proto_malloc );
    }
}
```



Source-level Probing

- Instrument only specific regions of the source

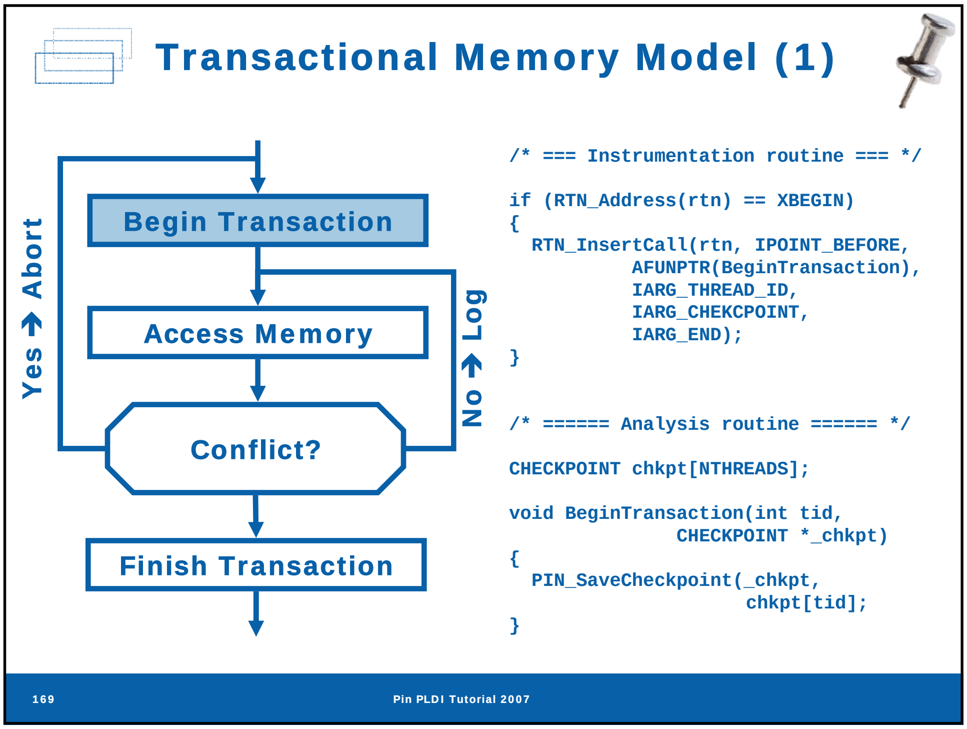
```
#include <stdio.h>
#include "pinapp.h"

int a[10];
int main()
{
    void * th = PIN_NewThread();

    printf("Thread handle %p\n", th);

    PIN_ExecuteInstrumented(th);

    for (int i = 0; i < 10; i++)
    {
        a[i] = i;
    }

    PIN_ExecuteUninstrumented();
    return 0;
}
```



Putting it all together: TMM

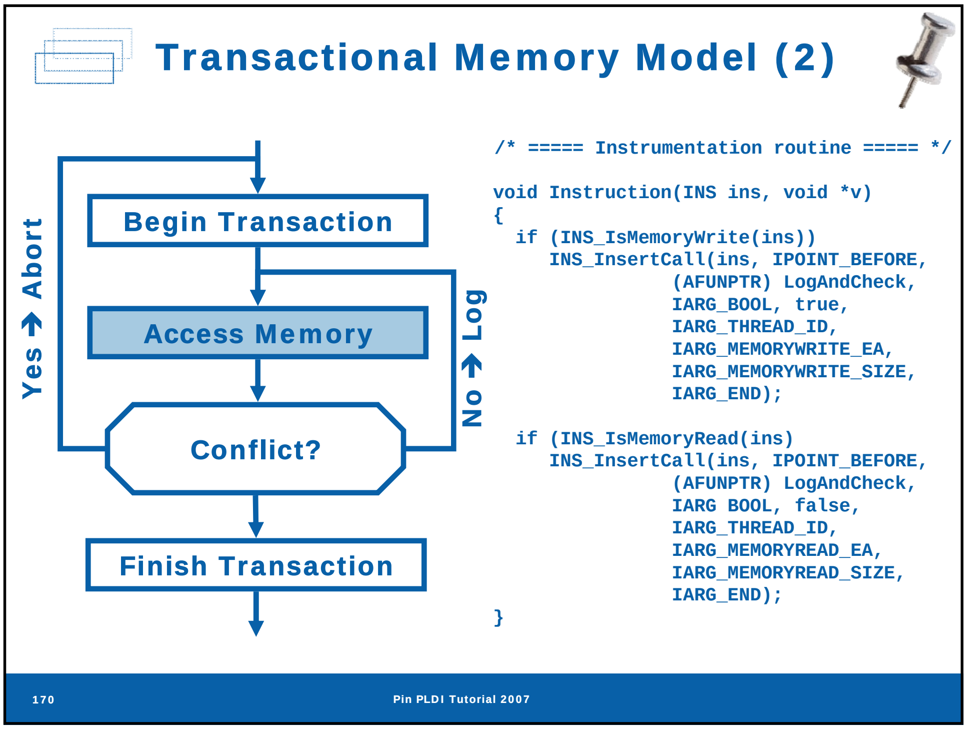
Transactional Memory Model

- Checkpoint architectural and memory state
- Log memory values modified by transaction
- Verify conflicts across parallel transactions
- Commit or Abort active transaction



Transactional Memory Model (1)

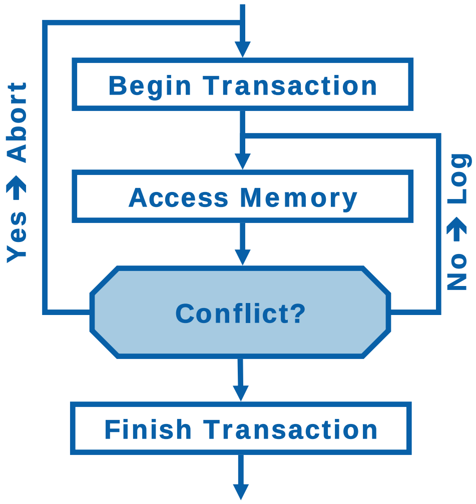
```
/* === Instrumentation routine === */

if (RTN_Address(rtn) == XBEGIN)
{
  RTN_InsertCall(rtn, IPOINT_BEFORE,
           AFUNPTR(BeginTransaction),
           IARG_THREAD_ID,
           IARG_CHEKCPOINT,
           IARG_END);
}

/* ====== Analysis routine ====== */

CHECKPOINT chkpt[NTHREADS];

void BeginTransaction(int tid,
               CHECKPOINT *_chkpt)
{
  PIN_SaveCheckpoint(_chkpt,
                     chkpt[tid];
}
```



Transactional Memory Model (2)

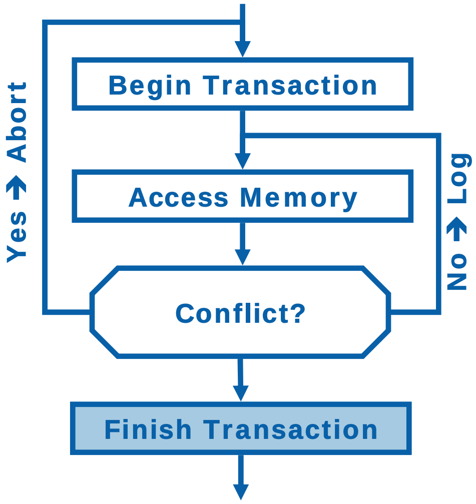
```
/* ===== Instrumentation routine ===== */

void Instruction(INS ins, void *v)
{
  if (INS_IsMemoryWrite(ins))
     INS_InsertCall(ins, IPOINT_BEFORE,
                (AFUNPTR) LogAndCheck,
                IARG_BOOL, true,
                IARG_THREAD_ID,
                IARG_MEMORYWRITE_EA,
                IARG_MEMORYWRITE_SIZE,
                IARG_END);

  if (INS_IsMemoryRead(ins)
     INS_InsertCall(ins, IPOINT_BEFORE,
                (AFUNPTR) LogAndCheck,
                IARG BOOL, false,
                IARG_THREAD_ID,
                IARG_MEMORYREAD_EA,
                IARG_MEMORYREAD_SIZE,
                IARG_END);
}
```

Transactional Memory Model (3)



```
/* ======== Analysis routine ======== */

void LogAndCheck(BOOL iswrite,
                 ADDRINT tid,
                 ADDRINT addr,
                 ADDRINT len)
{
   if ( /* in transaction */ )
   {
      if ( /* is conflict */ )
      {
        /* restore mem with log[tid] */
        PIN_Resume(&chkpt[th]);
      }
      else {
         /* record access in log[tid] */
      }
   }
}
```

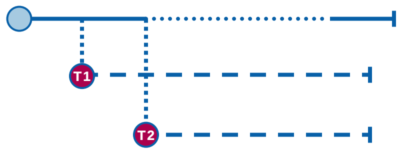
Transactional Memory Model (4)



```
/* === Instrumentation routine === */

if (RTN_Address(rtn) == XEND)
{
  RTN_InsertCall(rtn, IPOINT_BEFORE,
           AFUNPTR(CommitTransaction),
           IARG_THREAD_ID,
           IARG_END);
}

/* ====== Analysis routine ====== */

void CommitTransaction(ADDRINT th)
{
   /*
    * free thread’s checkpoint
    * and memory access log
    */
}
```

Demo of Transactional Memory

Multi-threaded Application



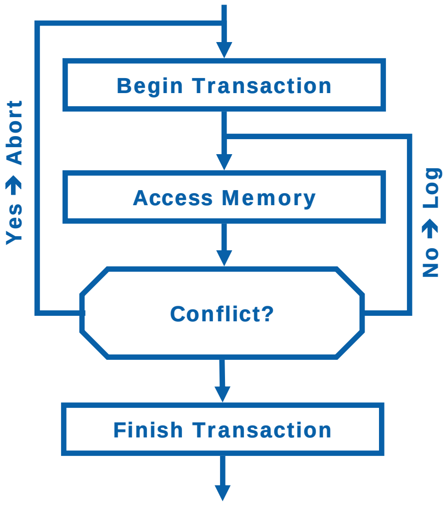
T1

```
XBEGIN();
for (uint32_t i = 0; i < MAX; i++)
{
   myarray[i] = 1;
}
XEND();
```

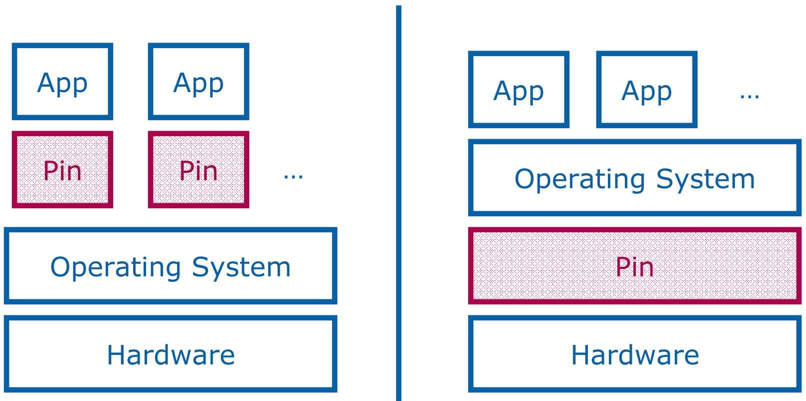
T2

```
XBEGIN();
for (int32_t i = MAX-1; i >= 0; i++)
{
   myarray[i] = 2;
}
XEND();
```

Transactional Memory Pin Tool



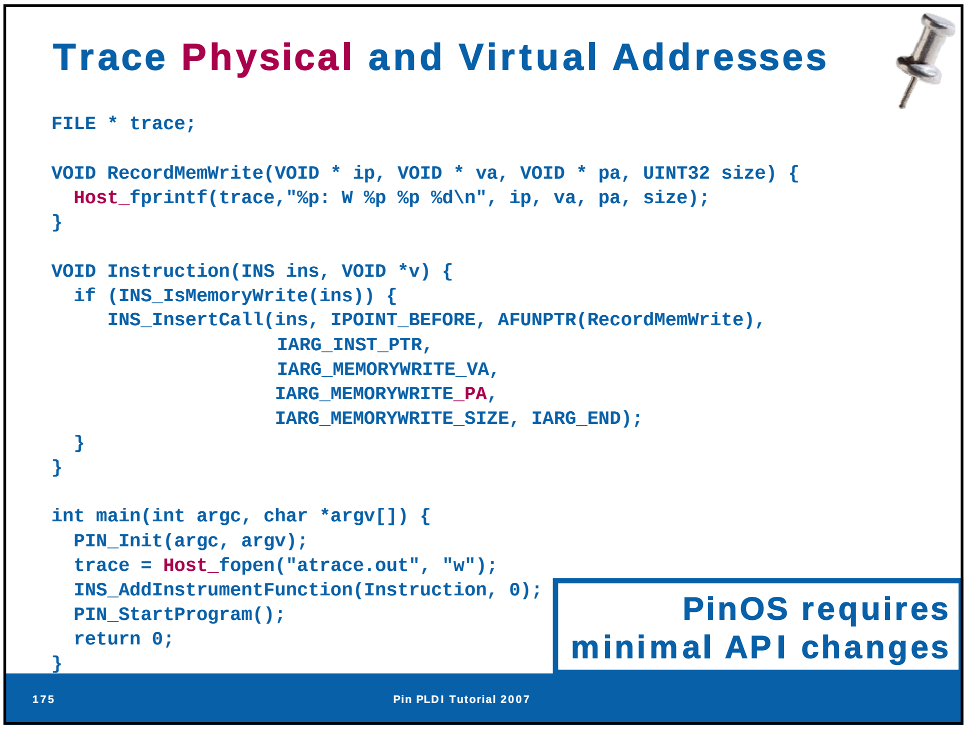
Pin (user-level) → PinOS (system-level)



Pin

Pin → Pin the OS!

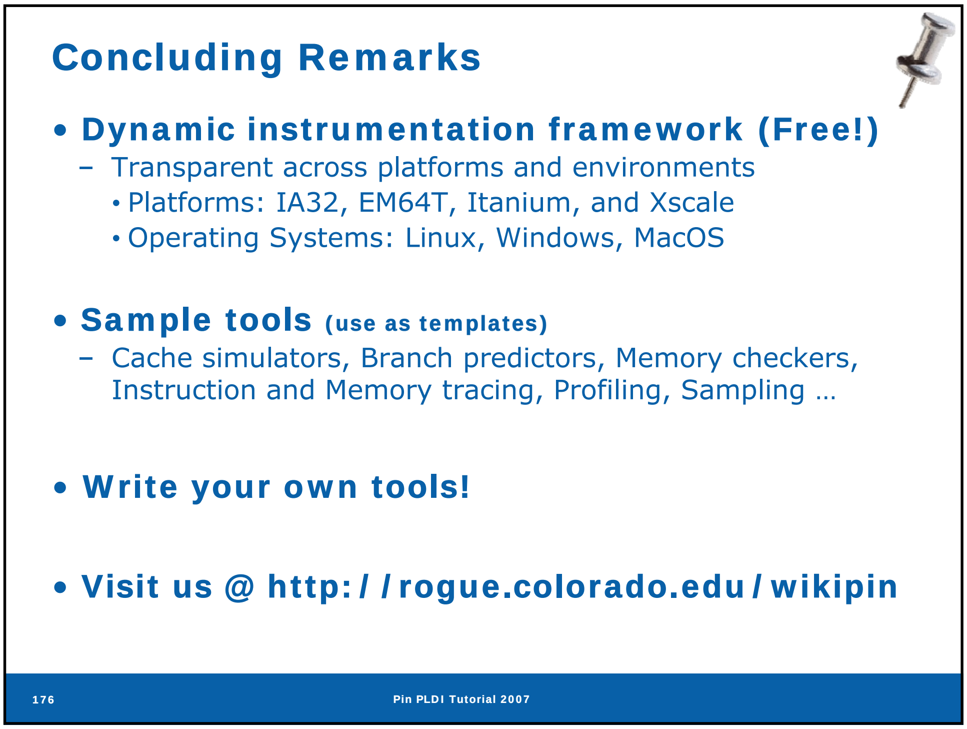
PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation.
Prashanth P. Bungale, C. K. Luk. *Proceedings of Virtual Execution Environments (VEE 2007)*



Trace Physical and Virtual Addresses

```
FILE * trace;

VOID RecordMemWrite(VOID * ip, VOID * va, VOID * pa, UINT32 size) {
  Host_fprintf(trace,"%p: W %p %p %d\n", ip, va, pa, size);
}

VOID Instruction(INS ins, VOID *v) {
  if (INS_IsMemoryWrite(ins)) {
     INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
                    IARG_INST_PTR,
                    IARG_MEMORYWRITE_VA,
                    IARG_MEMORYWRITE_PA,
                    IARG_MEMORYWRITE_SIZE, IARG_END);
  }
}

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

PinOS requires minimal API changes



Concluding Remarks

- **Dynamic instrumentation framework (Free!)**
  - Transparent across platforms and environments
    - Platforms: IA32, EM64T, Itanium, and Xscale
    - Operating Systems: Linux, Windows, MacOS
- **Sample tools (use as templates)**
  - Cache simulators, Branch predictors, Memory checkers, Instruction and Memory tracing, Profiling, Sampling …
- **Write your own tools!**
- **Visit us @ http://rogue.colorado.edu/wikipin**

NOTES

NOTES