

# **Program Instrumentation & Profiling**

CS 699/CS-IT 803

**Dan Ramey**

1

## **Outline**

- ATOM – Mid '90's instrumentation framework.
- Pin – Current tool targeted to Linux infrastructure
- PinPoints – Simulation tool based on Pin and SimPoints.

2

# Acknowledgments

- Portions of this presentation taken from:
  - CK Luk et al, “Pin Building Customized Program Analysis Tools with Dynamic Instrumentation”, *PLD’05* presentation.
  - Kim Hazelwood, Robert Muth, “Pin Tutorial”,  
<http://rogue.colorado.edu/pin/>
  - Harish Patil et al, “Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation”,  
<http://rogue.colorado.edu/Pin/PinPoints/documentation.html>

3

# Uses for Instrumentation

- **Profiling for optimization**
  - Basic block counts, edge counts
  - Value profiles, stride profiling, load latencies
- **Micro-architectural studies**
  - Branch predictor simulation
  - Cache simulation
  - Trace generation
- **Bug checking**
  - Find uninitialized or unallocated data references

4

## Motivation --ATOM

- Classes of tools
  - Basic block counting
  - Address tracing tools
  - Simulators
- Limitations of existing tools:
  - Designed to perform a specific type of instrumentation; difficult to modify
  - Generate too much detailed information
  - Add large overhead
  - Change the target program's heap addresses
  - Communicate with data analysis routines via inter-process communication or by writing to disks.

5

## A Common Infrastructure: ATOM

- A tool-building system: User creation/re-use of tools.
- Common infrastructure: User-specification of tool details.
- Selective instrumentation: User-specification of instrumentation points, procedures called, data passed.
- Communication through procedure calls: Elimination of IPC.
- Analysis of object models: Compiler and language independence.

6

# ATOM Design

- Must be able to instrument a program at a few selected points.
- A program is a linear collection of procedures; a procedure, a linear collection of basic blocks; a basic block, a linear collection of instructions.
- Common infrastructure for object-code manipulation.
- User-specification of:
  - Instrumentation routines.
  - Analysis routines.
- Application and analysis routines run in same address space – eliminates IPCs.
- Built on top of OM, a link-time code modification system.

7

## ATOM Design (continued)

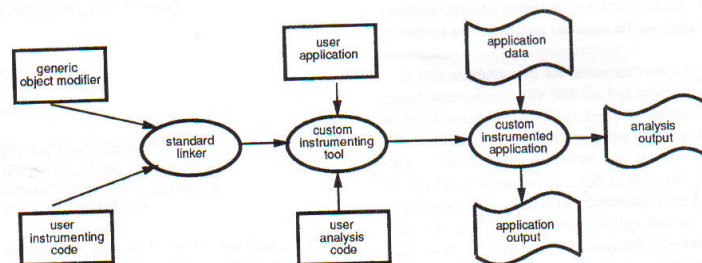


Figure 1: The ATOM Process

This figure taken from Srivastava & Eustace

8

## Using ATOM – Instrumentation

```
Instrument(int iargc, char **iargv)
{
    Proc *p;
    Block *b;
    Inst *inst;
    int nbranch = 0;

    AddCallProto("OpenFile(int)");
    AddCallProto("CondBranch(int, VALUE)");
    AddCallProto("PrintBranch(int, long)");
    AddCallProto("CloseFile()");

    for(p=GetFirstProc(); p!=NULL;p=GetNextProc(p)){
        for(b=GetFirstBlock(p);b!=NULL;b=GetNextBlock(b)){
            inst = GetLastInst(b);
            if(IsInstType(inst, InstTypeCondBr)){
                AddCallInst(inst, InstBefore, "CondBranch",
                    nbranch, BrCondValue);
                AddCallProgram(ProgramAfter, "PrintBranch",
                    nbranch, InstPC(inst));
                nbranch++;
            }
        }
    }

    AddCallProgram(ProgramBefore, "OpenFile", nbranch);
    AddCallProgram(ProgramAfter, "CloseFile");
}
```

Figure taken from Srivastava & Eustace

9

## Using ATOM – Analysis

```
#include <stdio.h>
File *file

struct BranchInfo{
    long taken;
    long notTaken;
} *bstats;

void OpenFile(int n){
    bstats = (structBranchInfo *)
        malloc (n * sizeof(struct BranchInfo));
    file = fopen("btaken.out", "w");
    fprintf(file, "PC \t Taken \t Not Taken \n");
}

void CondBranch(int n, long taken){
    if (taken)
        bstats[n].taken++;
    else
        bstats[n].notTaken++;
}

void PrintBranch(int n, long pc){
    fprintf(file, "0x%x \t %d \t %d \n",
        pc, bstats[n].taken, bstats[n].notTaken);
}

void CloseFile(){
    fclose(file);
}
```

10

## So What's the Problem?

- ATOM performs static instrumentation:
  - May not be able to distinguish between code and data
  - Difficulties with indirect branches, shared libraries, and dynamically generated code
- Can change program behavior
  - C-library shares data with application.
  - Some programs have behavior that depends on code addresses – ATOM changes these addresses
  - Puts code in the gap between data and code segments.
- Can't be turned on-and-off.

11

## Pin Motivation

- Provide run-time instrumentation
- Provide an instrumentation platform vice an instrumentation tool – Similar to ATOM
- Provide efficient instrumentation
- Provide facility to attach to and detach from a process.
- Defer code discovery to execution time
- Provide instrumentation transparency
- Multiple architectures running Linux applications

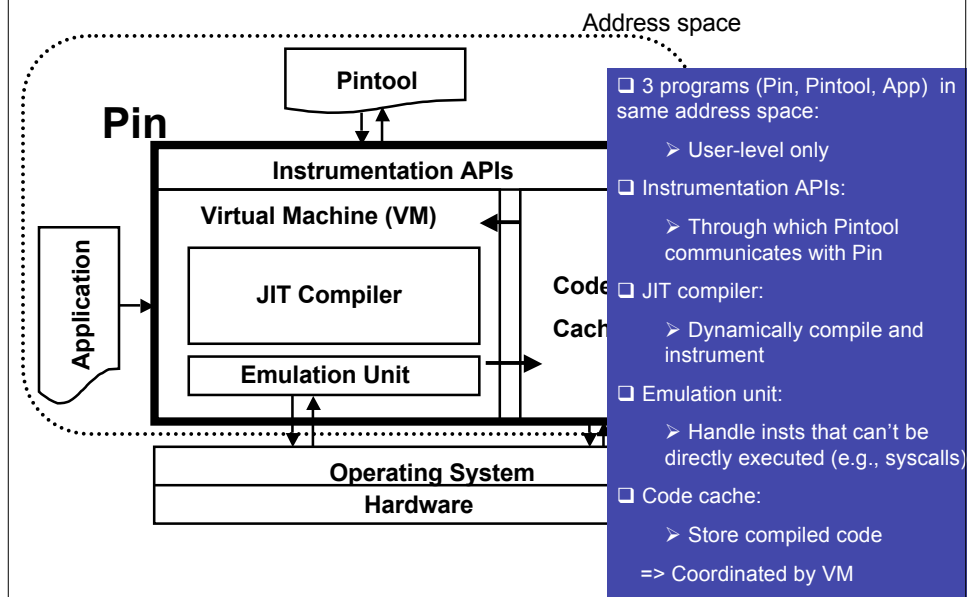
12

## Dynamic Instrumentation ("Pin Style")

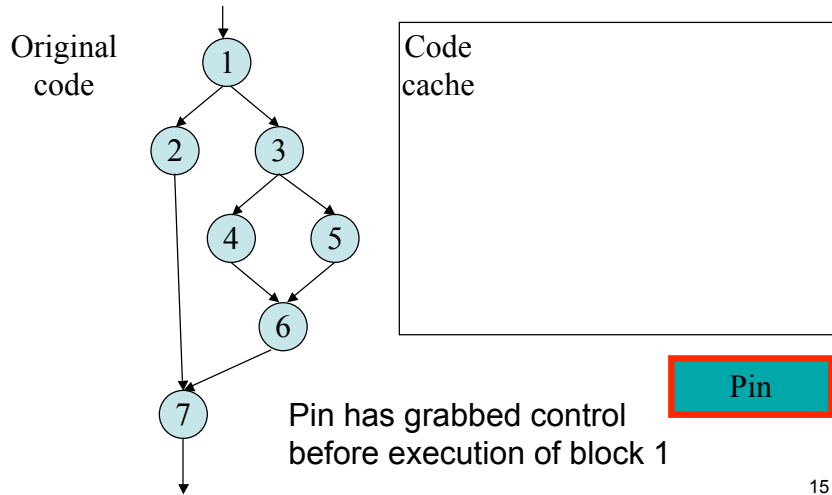
- Execution driven
  - Occurs when code is executed
- Original program is NOT modified
  - Code is "copied" into code cache
  - Only code in code cache is executed
- Instrumentation is not persistent
- Can also instrument libraries

13

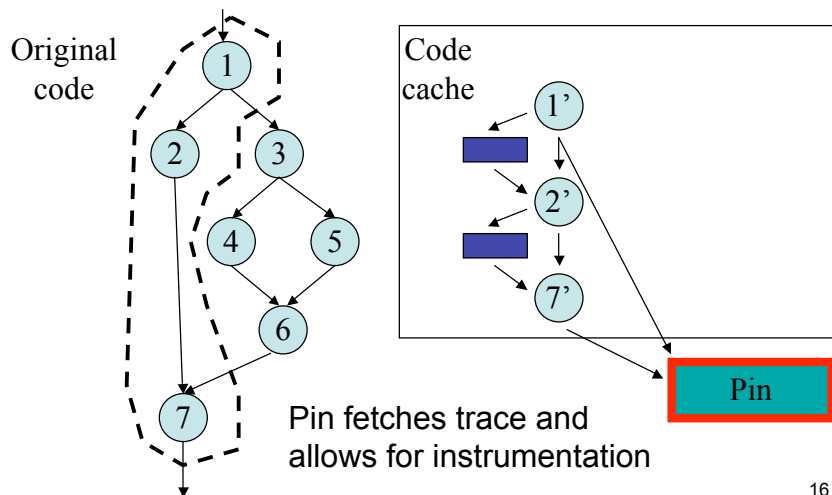
## Pin's Software Architecture



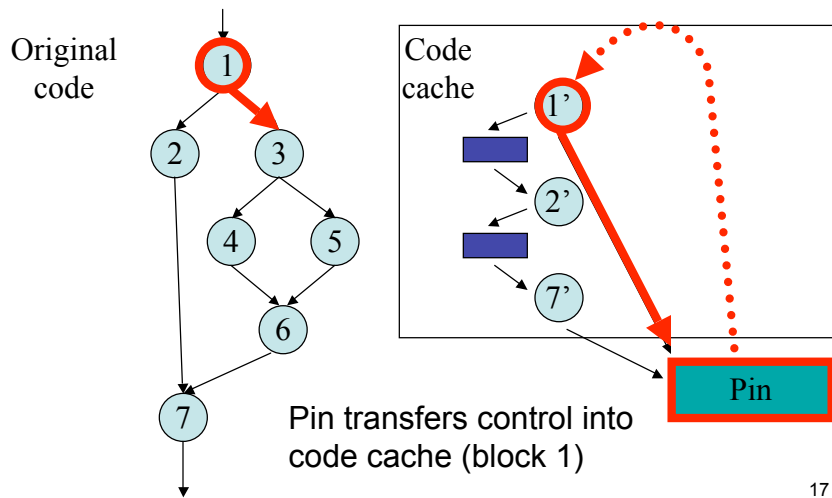
# Dynamic Instrumentation



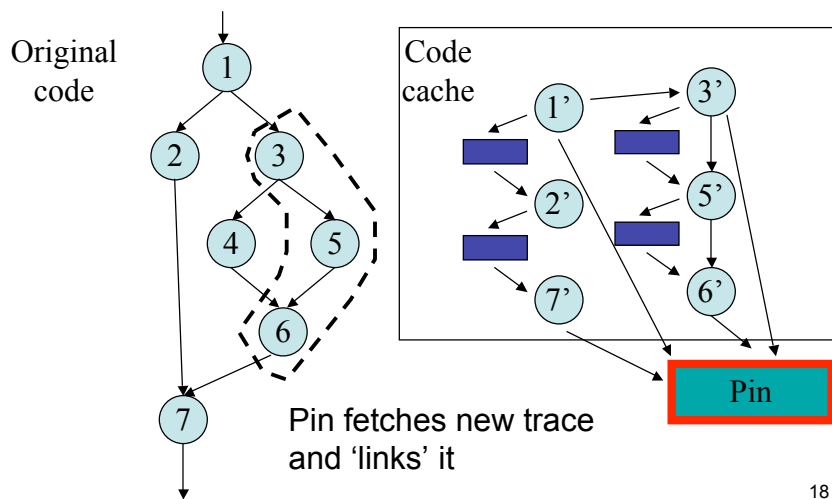
# Dynamic Instrumentation



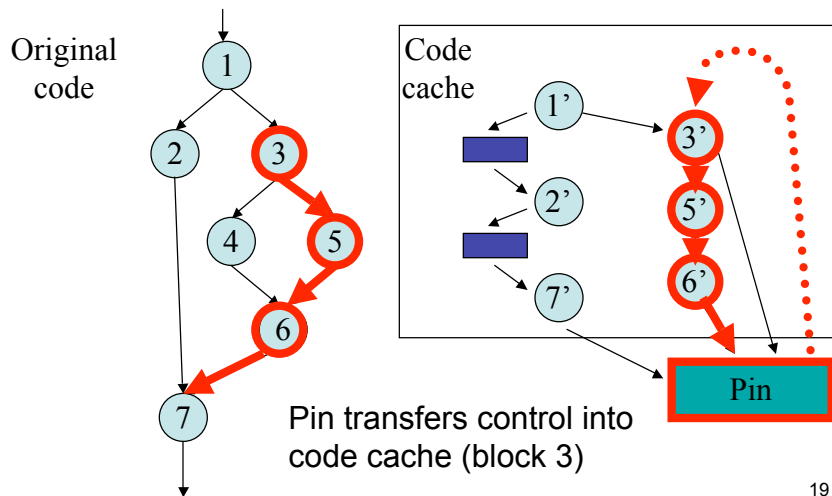
# Dynamic Instrumentation



# Dynamic Instrumentation



# Dynamic Instrumentation



# Register Re-allocation

- Instrumented code needs extra registers. E.g.:
  - Virtual registers available to the tool
  - A virtual stack pointer pointing to the instrumentation stack
  - Many more ...
- Approaches to get extra registers:
  1. Ad-hoc (e.g., DynamoRIO, Strata, DynInst)
    - Whenever you need a register, spill one and fill it afterward
  2. Re-allocate all registers during compilation
    - a. Local allocation (e.g., Valgrind)
      - Allocate registers independently within each trace
    - b. **Global allocation (Pin)**
      - **Allocate registers across traces (can be inter-procedural)**

20

# Instrumentation Optimizations

1. Inline instrumentation code into the application
2. Avoid saving/restoring eflags with liveness analysis
3. Schedule inlined instrumentation code

21

## Example: Instruction Counting

### Original code

```

cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>

add %ecx, %edx
cmp %edx, 0
je <target2>
    
```

BBL\_InsertCall(bbl, IPOINT\_BEFORE, docount(),  
IARG\_UINT32, BBL\_NumIns(bbl),  
IARG\_END)

33 extra instructions executed altogether

Instrument without applying any optimization

Trace

```

mov %esp, SPILL_appsp
mov SPILL_pinsp, %esp
call <bridge>
cmov %esi, %edi
mov SPILL_appsp, %esp
cmp %edi, (%esp)
jle <target1'>
    
```

```

mov %esp, SPILL_appsp
mov SPILL_pinsp, %esp
call <bridge>
add %ecx, %edx
cmp %edx, 0
je <target2'>
    
```

bridge()

```

pushf
push %edx
push %ecx
push %eax
movl 0x3, %eax
call docount
pop %eax
pop %ecx
pop %edx
popf
ret
    
```

docount()

```

add %eax, icount
ret
    
```

22

# Example: Instruction Counting

## Original code

```
cmov %esi, %edi  
cmp %edi, (%esp)  
jle <target1>
```

```
add %ecx, %edx  
cmp %edx, 0  
je <target2>
```

Inlining

## Trace

```
mov %esp, SPILLappsp  
mov SPILLpinsp, %esp  
pushf  
add 0x3, icount  
popf  
cmov %esi, %edi  
mov SPILLappsp, %esp  
cmp %edi, (%esp)  
jle <target1'>
```

```
mov %esp, SPILLappsp  
mov SPILLpinsp, %esp  
pushf  
add 0x3, icount  
popf  
add %ecx, %edx  
cmp %edx, 0  
je <target2'>
```

☞ 11 extra instructions executed

23

# Example: Instruction Counting

## Original code

```
cmov %esi, %edi  
cmp %edi, (%esp)  
jle <target1>
```

```
add %ecx, %edx  
cmp %edx, 0  
je <target2>
```

Inlining + eflags liveness analysis

## Trace

```
mov %esp, SPILLappsp  
mov SPILLpinsp, %esp  
pushf  
add 0x3, icount  
popf  
cmov %esi, %edi  
mov SPILLappsp, %esp  
cmp %edi, (%esp)  
jle <target1'>
```

```
add 0x3, icount  
add %ecx, %edx  
cmp %edx, 0  
je <target2'>
```

☞ 7 extra instructions executed

24

## Example: Instruction Counting

### Original code

```
cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>
```

```
add %ecx, %edx
cmp %edx, 0
je <target2>
```

Inlining + eflags liveness analysis + **scheduling**

### Trace

```
cmov %esi, %edi
add 0x3, icount
cmp %edi, (%esp)
jle <target1'>
```

```
add 0x3, icount
add %ecx, %edx
cmp %edx, 0
je <target2'>
```

☞ *2 extra instructions executed*

25

## Trace Linking

- Trace linking is a very effective optimization
  - Bypass VM when transferring from one trace to another
  - Slowdown without trace linking as much as 100x
- Linking direct branches/calls
  - Straightforward as targets are unique
- Linking indirect branches/calls & returns
  - More challenging because the target can be different each time
  - Our approach:
    - For all indirect control transfers, use **chaining**
    - For returns, further optimizes with **function cloning**

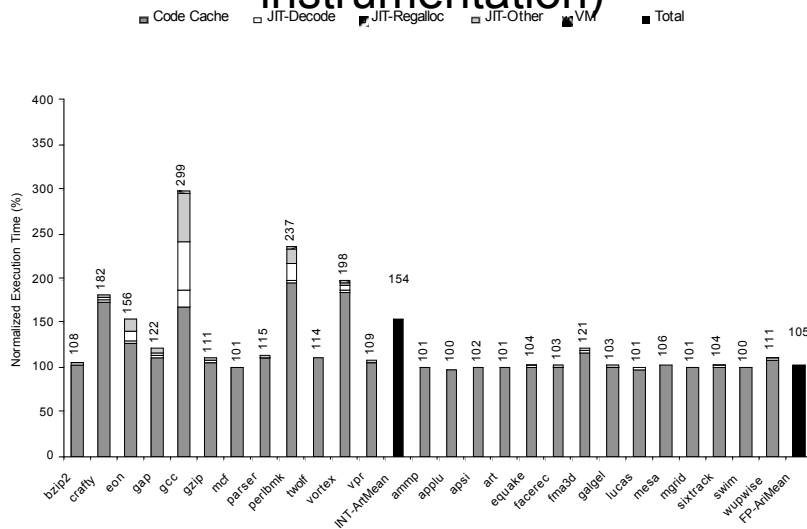
26

# Pin Evaluation

- Performance measured against standard benchmarks.
- No instrumentation – measures Pin’s dynamic instrumentation overhead.
- Basic block counting – measures impact of JIT on overhead.
- Comparison with other dynamic instrumentation tools.

27

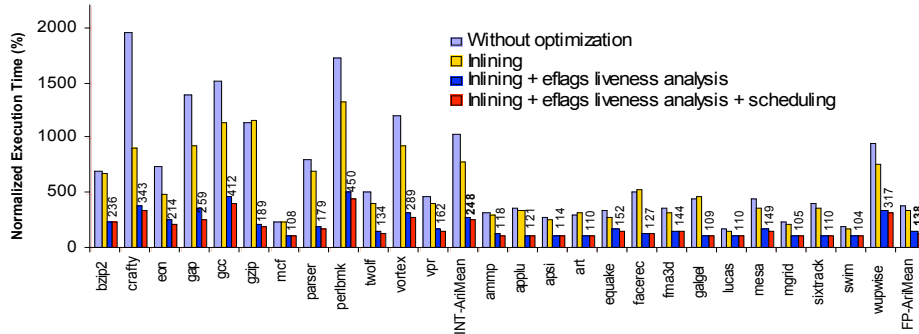
## Pin/IA32 Performance (no instrumentation)



28

# Pin Instrumentation

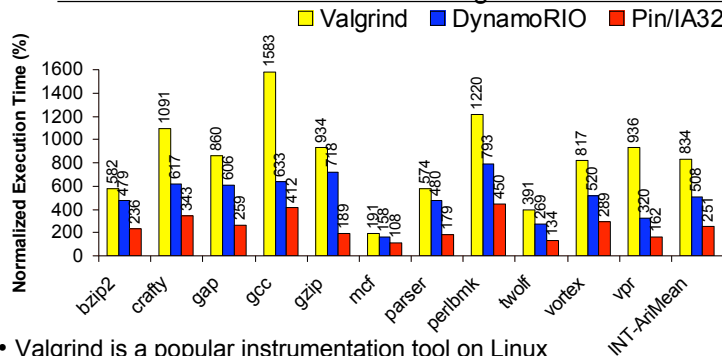
Performance of basic-block counting with Pin/IA32



	Average slowdown	
	INT	FP
Without optimization	10.4x	3.9x
Inlining	7.8x	3.5x
Inlining + eflags analysis	2.8x	1.5x
Inlining + eflags analysis + scheduling	2.5x	1.4x

# Comparison among Dynamic Instrumentation Tools

Performance of basic-block counting with three different tools



- Valgrind is a popular instrumentation tool on Linux
  - Call-based instrumentation, no inlining
- DynamoRIO is the performance leader in dynamic optimization
  - Manually inline, no eflags liveness analysis and scheduling

*Pin automatically provides efficient instrumentation*

## Optimizing Instrumentation Performance

### Observations:

- Slowdown largely due to executing instrumentation code rather than dynamic compilation
  - ⇒ Make sense to spend more time to optimize
- Focus on optimizing simple instrumentation tools:
  - Performance depends on how fast we can transit between the application and the tool
  - Simple yet commonly used (e.g., basic-block profiling)

31

## Pin Evaluation – Concluded

- Pin overhead – largest impact on short-running programs such as gcc; small for longer-running programs.
- With various optimizations basic instrumentation doubles or triples execution time.
- Better performance than two other dynamic instrumentation packages considered.

32

## 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
- Some companies are writing their own Pintools:
  - A major database vendor, a major search engine provider
- Some universities using Pin in teaching and research:
  - U. of Colorado, MIT, Harvard, Princeton, U of Minnesota, Northeastern, Tufts, University of Rochester, ...

33

## Conclusions

- Pin
  - A dynamic instrumentation system for building your own program analysis tools
  - Easy to use, robust, transparent, efficient
  - Tool source compatible on IA32, EM64T, Itanium, ARM
  - Works on large applications
    - database, search engine, web browsers, ...
  - Available on Linux; Windows version coming soon
- Downloadable from <http://rogue.colorado.edu/Pin>
  - User manual, many example tools, tutorials
  - 3300 downloads since 2004 July

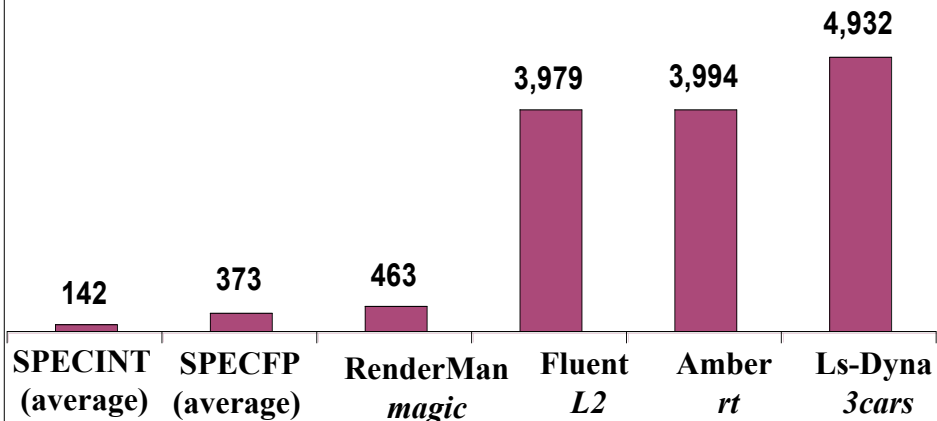
34

## PinPoints Overview

- A simulation tool, not an instrumentation tool.  
Uses:
  - Pin to generate program traces
  - Sim to characterize program phases
- Develop accurate simulations of very LARGE applications by selecting representative program phases for simulation:
  - With little/no manual intervention
  - Within reasonable time

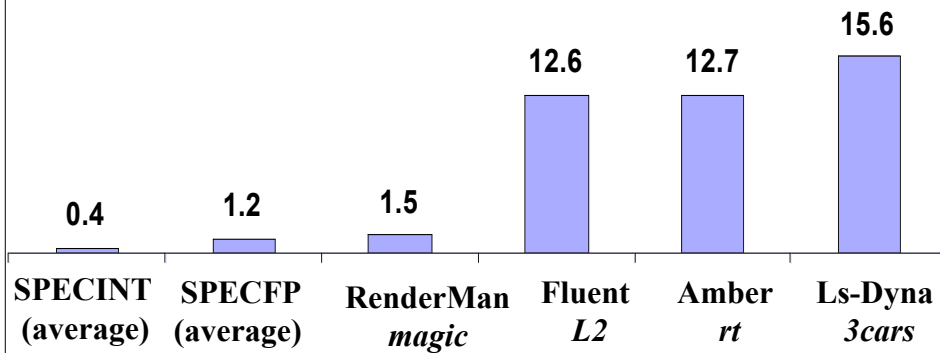
35

## Instruction Counts : Some Itanium Applications # Instructions ( billions )



36

## Whole-Program Simulation is Slow Simulation Time in YEARS @ 10,000 Instructions/Second



37

## Solution: Select Simulation Points

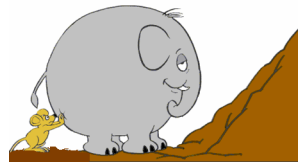
- Manually
- Randomly
  - Anywhere
  - From uniform regions
- Fine-grain sampling (SMARTS: CMU)
- **By program-phase analysis**  
(SimPoint:UCSD, *iPart*: Intel/MRL)

38

## Running Commercial Applications on Simulators is Hard

- **Resource Requirements:** Disks etc.
  - Need to modify/re-configure the simulator
- **OS dependencies**
  - Need support for specific kernel and device drivers
- **License checking**
  - Need special action

*It's an uphill task!*



## Solution: Native Execution with Instrumentation

Use **PIN** to select simulation points (*PinPoints*) and generate traces

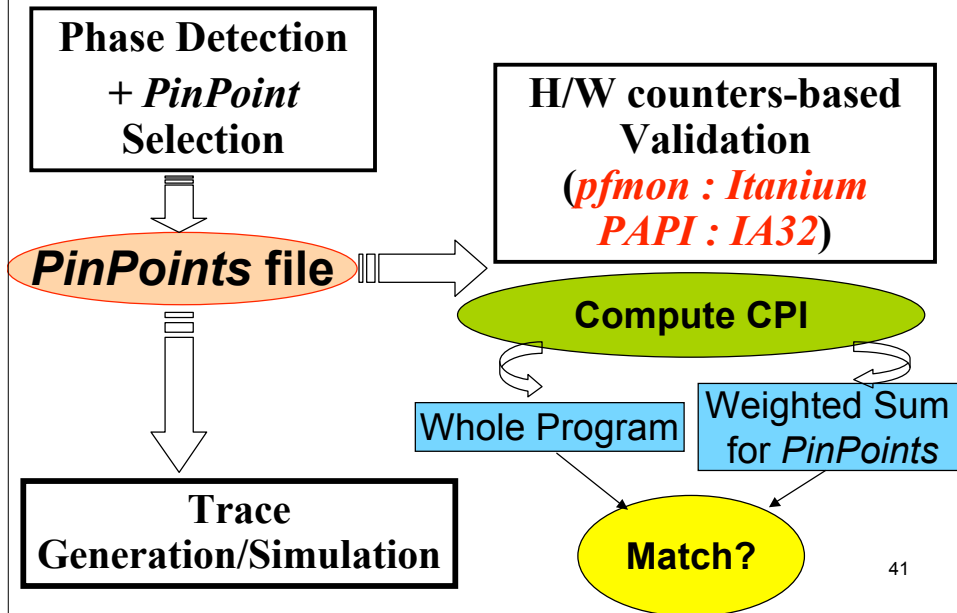
**PIN: A dynamic-instrumentation system**

+ A tool for writing tools

+ No special compiler/linker flags required



## The *PinPoints* Toolkit



## Evaluations

**Applications:** Built w/ Intel's compilers (*high opt*)

**HPC:** Fluent, AMBER, LS-Dyna, RenderMan

**SPEC2000:** Processed 8-9 times

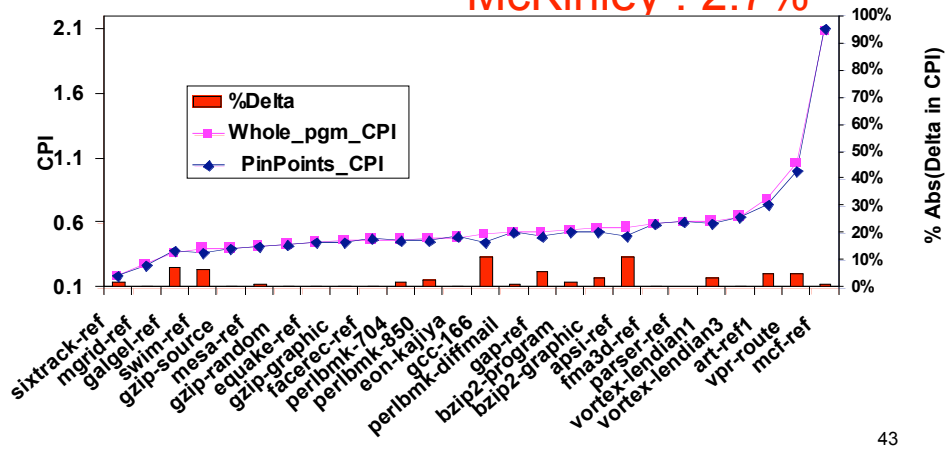
**Test Configurations:** Linux (RedHat)

<b>Merced</b>	<b>Itanium (1)</b>	<b>800 MHz</b>	<b>L3: 2MB</b>
<b>McKinley</b>	<b>Itanium-2</b>	<b>900 MHz</b>	<b>L3: 1.5MB</b>
<b>Madison</b>	<b>Itanium-2</b>	<b>1.3 GHz</b>	<b>L3: 3-6 MB</b>

42

# SPEC2000 CPI Prediction

**Average Error: Madison : 2.8%**  
**Merced : 3.2%**  
**McKinley : 2.7%**



43

# CPI – Commercial Programs

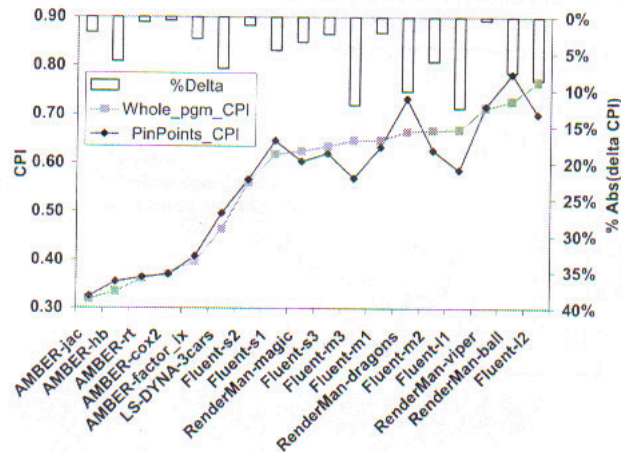


Figure 4 Actual vs. predicted CPI and % delta for commercial programs

44



## Summary

- ATOM – Mid 90's program instrumentation tool:
  - Early framework for program instrumentation
  - User- created C modules for instrumentation and analysis
  - Static – instrumentation at compile time
  - Mostly does not affect program behavior
  - Self-modifying code an issue

47

## Summary (continued)

- Pin – Mid 2000's program instrumentation tool:
  - ATOM-style framework
  - User- created C and C++ modules for instrumentation and analysis
  - Dynamic – instrumentation at run time
  - Claim that does not affect program behavior
  - Able to handle self-modifying code
  - Linux OS plus Windows beta for Intel architectures

48

## Summary (concluded)

- PinPoints – Pin-based simulation tool:
  - Select representative program phases for program simulation
  - Enables simulation of very long running programs by simulating only representative phases: Simulation run time drops from years to days with reasonable errors

49

## References

- Srivastava & Eustace, “ATOM A System for Building Customized Program Analysis Tools”, *PLDI'94*
- Luk *et al*, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”, *PLID'05*
- Patil *et al*, “Pinpointing Representative Portions of Large Intel® Programs with Dynamic Instrumentation”, *MICRO'04*
- Pin website: <http://rogue.colorado.edu/Pin/index.html>

50

# Open Discussion

51