Dynamic Binary Translation & Instrumentation

Pin

Building Customized Program Analysis Tools with Dynamic Instrumentation

CK Luk, Robert Cohn, Robert Muth, Harish Patil,

Artur Klauser, Geoff Lowney, Steven Wallace, Kim Hazelwood

Intel

Vijay Janapa Reddi

University of Colorado

http://rogue.colorado.edu/Pin

Instrumentation

- Insert extra code into programs to collect information about execution
  - Program analysis:
    - Code coverage, call-graph generation, memory-leak detection
  - Architectural study:
    - Processor simulation, fault injection
- Existing binary-level instrumentation systems:
  - Static:
    - ATOM, EEL, Etch, Morph
  - Dynamic:
    - Dyninst, Vulcan, DTrace, Valgrind, Strata, DynamoRIO

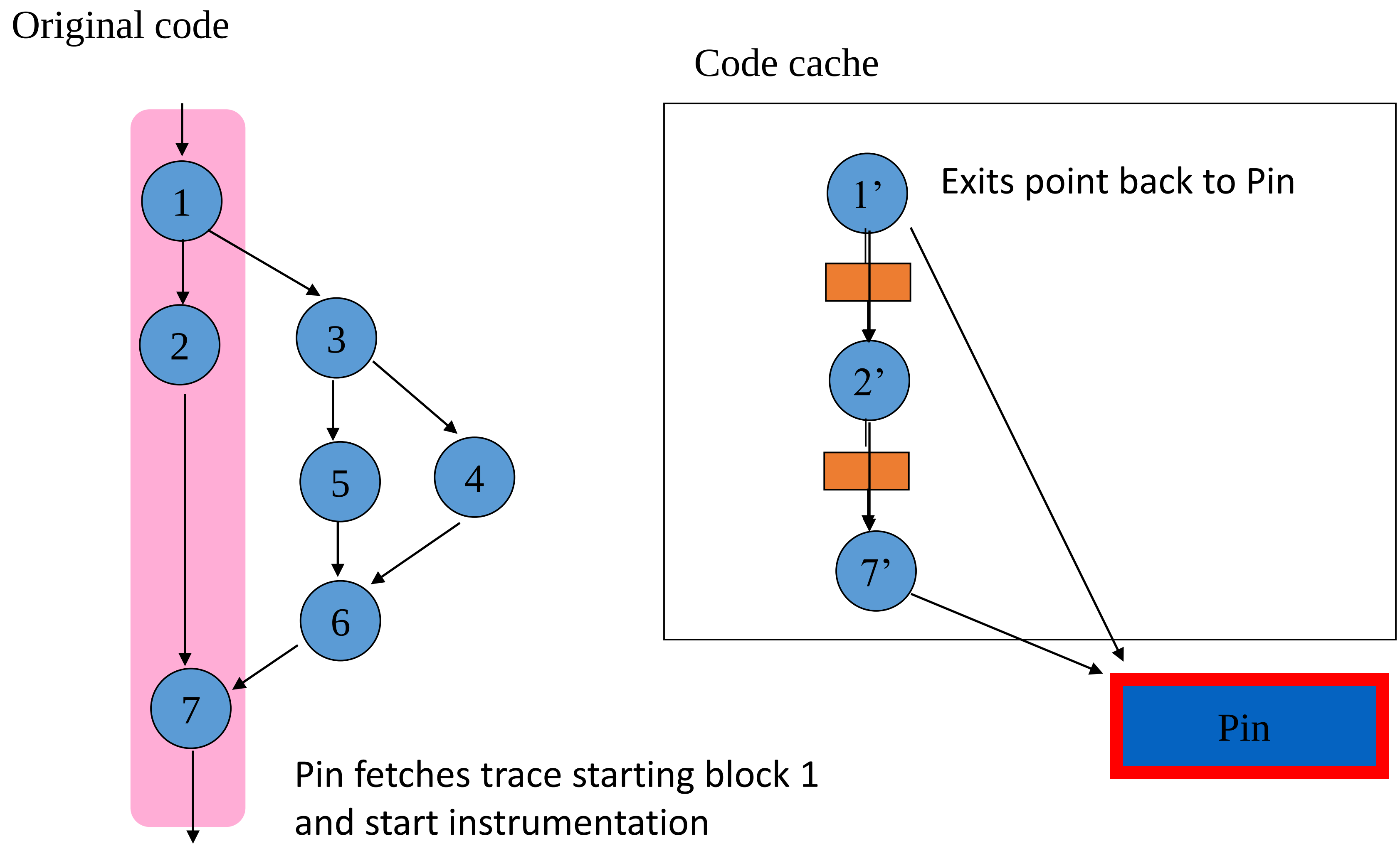
☞ ***Pin is a new dynamic binary instrumentation system***

A Pintool for Tracing Memory Writes

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

FILE*

VOID
    fp
}

VOID
    if

IARG

}

int m
    PI
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
    return 0;
}
```

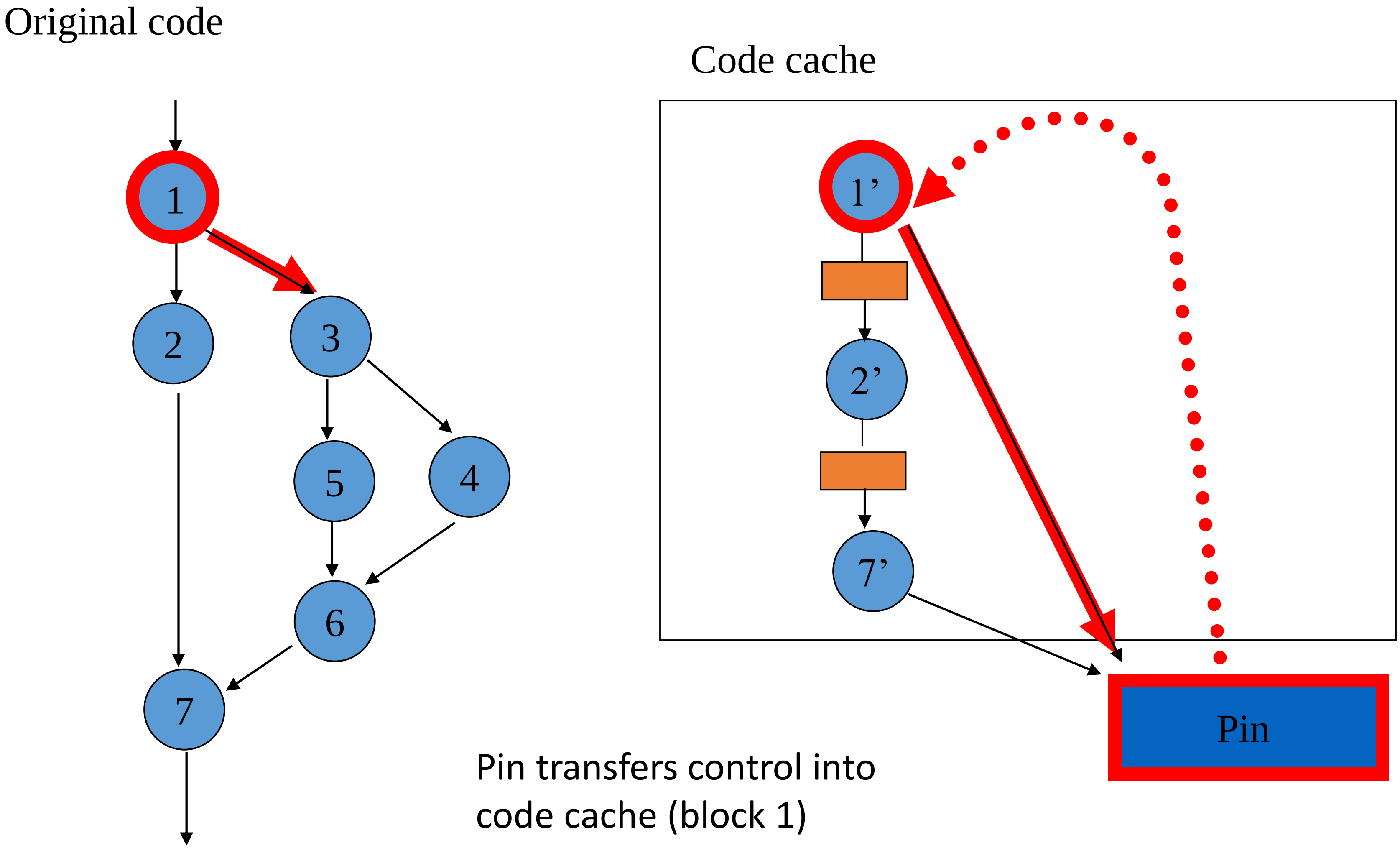
- Same source code works on the 4 architectures
  => Pin takes care of different addressing modes
- No need to manually save/restore application state
  => Pin does it for you automatically and efficiently

Dynamic Instrumentation



Pin fetches trace starting block 1
and start instrumentation

Dynamic Instrumentation

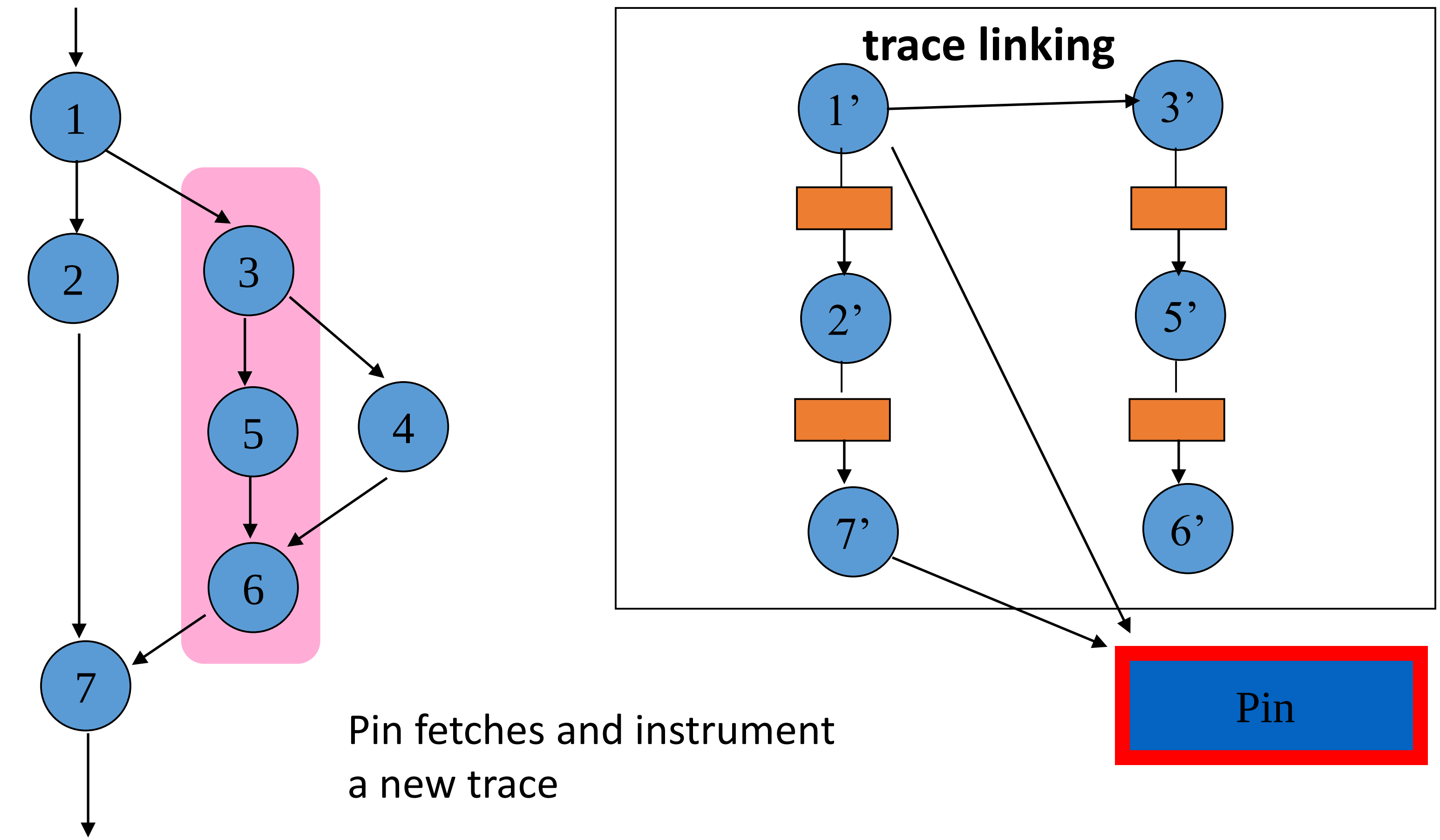


Pin transfers control into code cache (block 1)

Dynamic Instrumentation

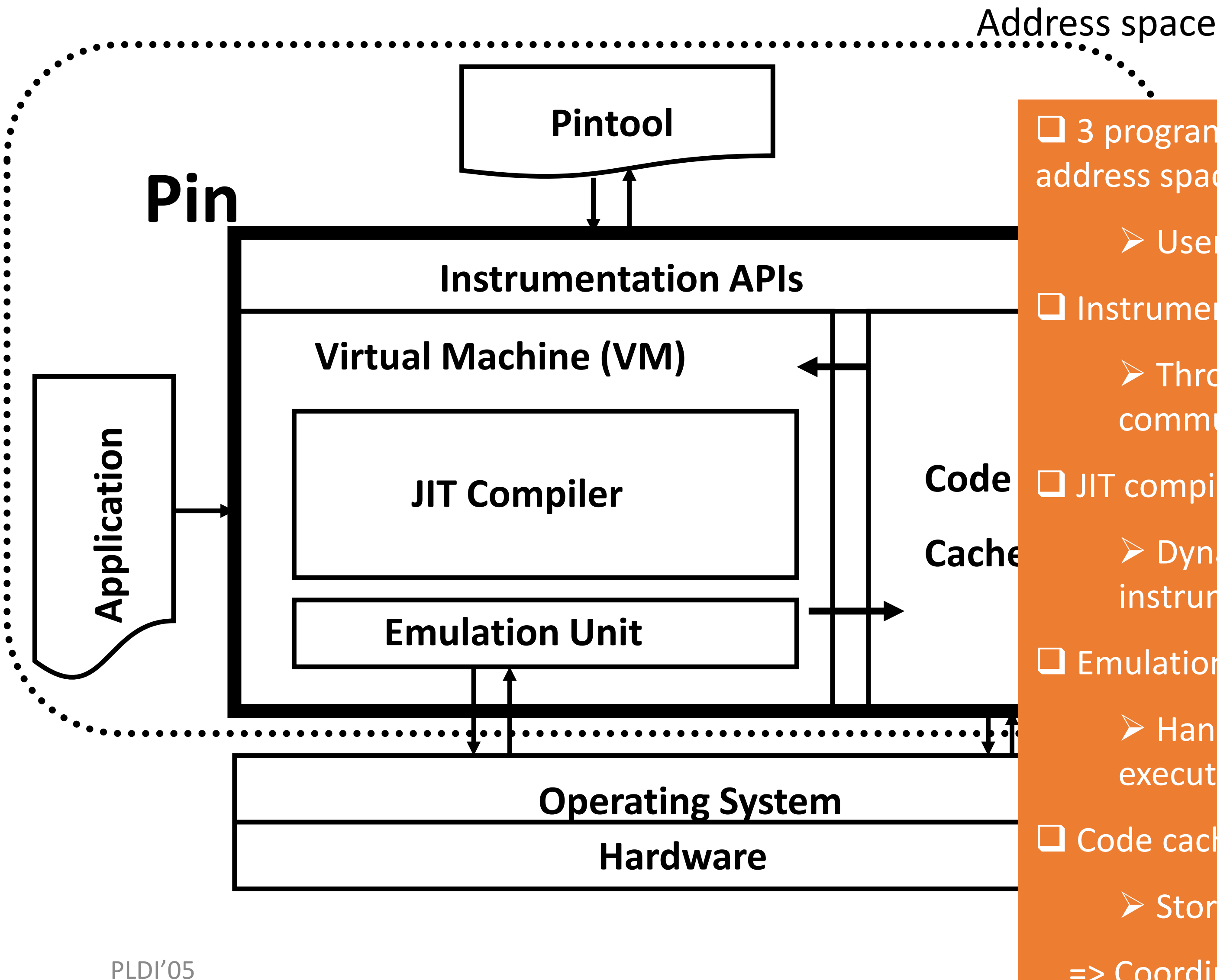
Original code

Code cache



Pin fetches and instrument a new trace

Pin’s Software Architecture



- ❑ 3 programs (Pin, Pintool, App) in same address space:
  - ➢ User-level only
- ❑ Instrumentation APIs:
  - ➢ Through which Pintool communicates with Pin
- ❑ JIT compiler:
  - ➢ Dynamically compile and instrument
- ❑ Emulation unit:
  - ➢ Handle insts that can’t be directly executed (e.g., syscalls)
- ❑ Code cache:
  - ➢ Store compiled code

=> Coordinated by VM

Pin Internal Details

- Loading of Pin, Pintool, & Application
- An Improved Trace Linking Technique
- **Register Re-allocation**
- **Instrumentation Optimizations**
- Multithreading Support

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)**

Valgrind's Register Re-allocation

Original Code

```
    mov 1, %eax
    mov 2, %ebx
    cmp %ecx, %edx
    jz t
    . . .
t:  add 1, %eax
    sub 2, %ebx
    . . .
```

re-allocate →

Trace 1

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
mov %eax, SPILL_eax
mov %esi, SPILL_ebx
jz t'
```

Virtual	Physical
%eax	%eax
%ebx	**%esi**
%ecx	%ecx
%edx	%edx

Trace 2

```
t': mov SPILL_eax, %eax
    mov SPILL_ebx ,%edi
    add 1, %eax
    sub 2, %edi
    . . .
```

Virtual	Physical
%eax	%eax
%ebx	**%edi**
%ecx	%ecx
%edx	%edx

☞ ***Simple but inefficient***

- All modified registers are spilled at a trace's end
- Refill registers at a trace's beginning

Pin's Register Re-allocation

Scenario (1): Compiling a new trace at a trace exit

Original Code

```
    mov 1, %eax
    mov 2, %ebx
    cmp %ecx, %edx
    jz t
    • • •
t:  add 1, %eax
    sub 2, %ebx
    • • •
```

re-allocate

Trace 1

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
jz t'
```

Trace 2

```
t': add 1, %eax
    sub 2, %esi
    • • •
```

Compile Trace 2 using the binding at Trace 1's exit:

Virtual	Physical
%eax	%eax
%ebx	**%esi**
%ecx	%ecx
%edx	%edx

☞ ***No spilling/filling needed across traces***

Pin’s Register Re-allocation

Scenario (2): Targeting an already generated trace at a trace exit

Original Code

```
    mov 1, %eax
    mov 2, %ebx
    cmp %ecx, %edx
    jz t
    • • •
t:  add 1, %eax
    sub 2, %ebx
    • • •
```

re-allocate →

Trace 1 (being compiled)

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
mov %esi, SPILL_ebx
mov SPILL_ebx, %edi
jz t’
```

Virtual	Physical
%eax	%eax
%ebx	**%esi**
%ecx	%ecx
%edx	%edx

Trace 2 (in code cache)

```
t’: add 1, %eax
    sub 2, %edi
    • • •
```

Virtual	Physical
%eax	%eax
%ebx	**%edi**
%ecx	%ecx
%edx	%edx

☞ ***Minimal spilling/filling code***

Instrumentation Optimizations

1. Inline instrumentation code into the application

2. Avoid saving/restoring eflags with liveness analysis

3. Schedule inlined instrumentation code

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
```

Example: Instruction Counting

<u>Original code</u>

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

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

Inlining →

<u>Trace</u>

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

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

☞ ***11 extra instructions executed***

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

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

☞ ***7 extra instructions executed***

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

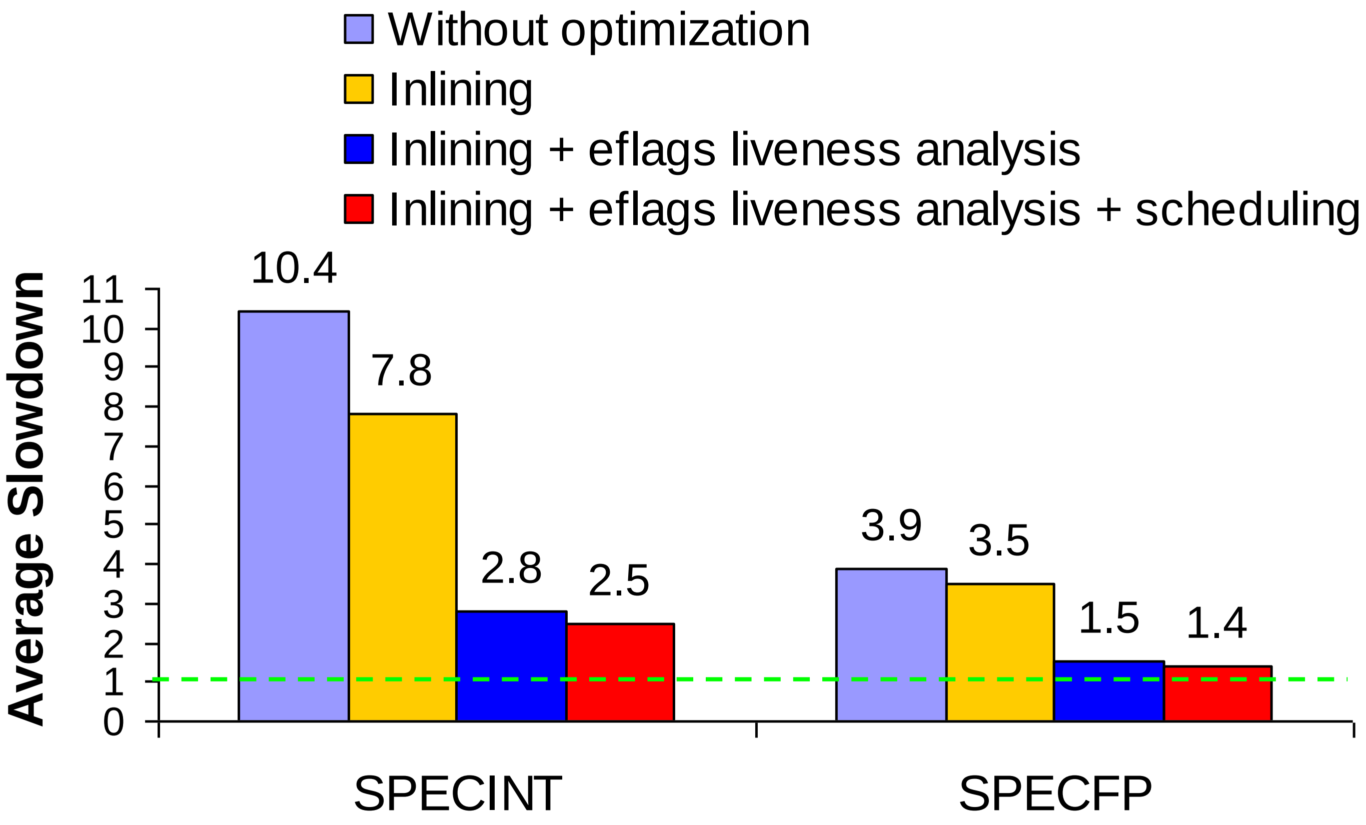
```
cmov %esi, %edi
add 0x3, icount
cmp %edi, (%esp)
jle <target1'>
```

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

☞ ***2 extra instructions executed***

Pin Instrumentation Performance

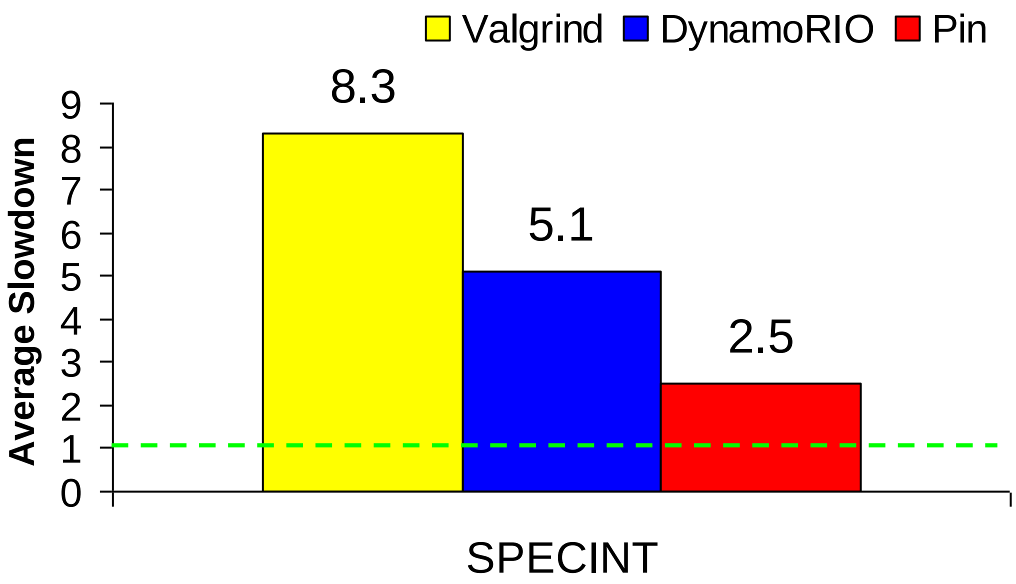
Runtime overhead of basic-block counting with Pin on IA32



(SPEC2K using reference data sets)

Comparison among Dynamic Instrumentation Tools

<u>Runtime overhead of basic-block counting with three different tools</u>



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

☞ ***Pin automatically provides efficient instrumentation***

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, …

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

Valgrind
A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote — National ICT Australia
Julian Seward — OpenWorks LLP

FAQ #1

- How do you pronounce “Valgrind”?

- “**Val-grinned**”, not “Val-grined”

- Don’t feel bad: almost everyone gets it wrong at first

DBA tools

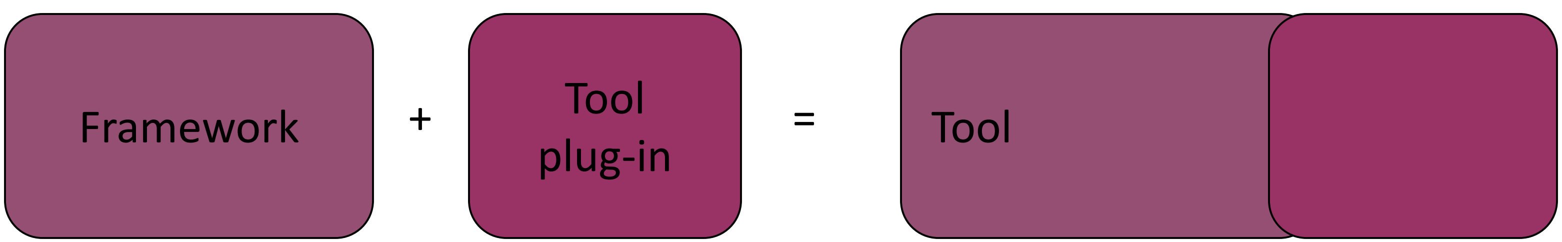
- Program analysis tools are useful
  - Bug detectors
  - Profilers
  - Visualizers

- **Dynamic binary analysis** (DBA) tools
  - Analyse a program's machine code at run-time
  - Augment original code with **analysis code**

Building DBA tools

- **Dynamic binary instrumentation** (DBI)
  - Add analysis code to the original machine code at run-time
  - No preparation, 100% coverage

- DBI frameworks
  - Pin, DynamoRIO, Valgrind, etc.



Prior work

Well-studied	Not well-studied
Framework performance	Instrumentation capabilities
Simple tools	Complex tools

- **Potential of DBI has not been fully exploited**
  - Tools get less attention than frameworks
  - Complex tools are more interesting than simple tools

Shadow value tools

Shadow value tools (I)

- Shadow every value with another value that describes it
  - Tool stores and propagates shadow values in parallel

	Tool(s)	Shadow values help find...
bugs	**Memcheck**	**Uses of undefined values**
	Annelid	Array bounds violations
security	Hobbes	Run-time type errors
	TaintCheck, LIFT, TaintTrace	Uses of untrusted values
propertie s	“Secret tracker”	Leaked secrets
	DynCompB	Invariants
	Redux	Dynamic dataflow

Memcheck

- Shadow values: defined or undefined

Original operation	Shadow operation
int* p = malloc(4)	sh(p) = undefined
R1 = 0x12345678	sh(R1) = defined
R1 = R2	sh(R1) = sh(R2)
R1 = R2 + R3	sh(R1) = add$_{sh}$(R2, R3)
if R1==0 then goto L	complain if sh(R1) is
	undefined

- 30 undefined value bugs found in OpenOffice

Shadow value tools (II)

- All shadow value tools work in the same basic way

- Shadow value tools are **heavyweight** tools
  - Tool's data + ops are as complex as the original programs's

- Shadow value tools are hard to implement
  - Multiplex real and shadow registers onto register file
  - Squeeze real and shadow memory into address space
  - Instrument most instructions and system calls

Valgrind basics

Valgrind

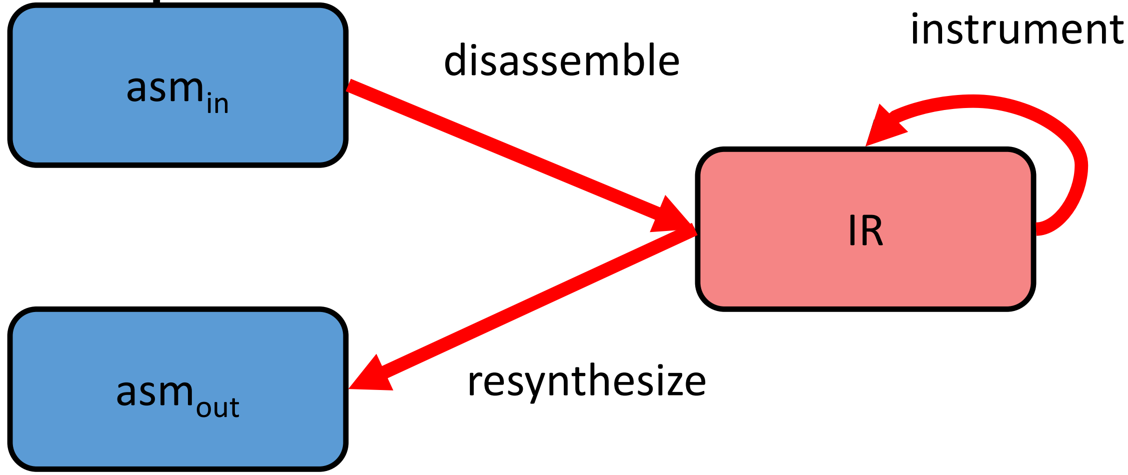
- Software
  - Free software (GPL)
  - {x86, x86-64, PPC}/Linux, PPC/AIX
- Users
  - Development: Firefox, OpenOffice, KDE, GNOME, MySQL, Perl, Python, PHP, Samba, RenderMan, Unreal Tournament, NASA, CERN
  - Research: Cambridge, MIT, Berkeley, CMU, Cornell, UNM, ANU, Melbourne, TU Muenchen, TU Graz
- Design
  - Heavyweight tools are well supported
  - Lightweight tools are slow

Two unusual features of Valgrind

#1: Code representation

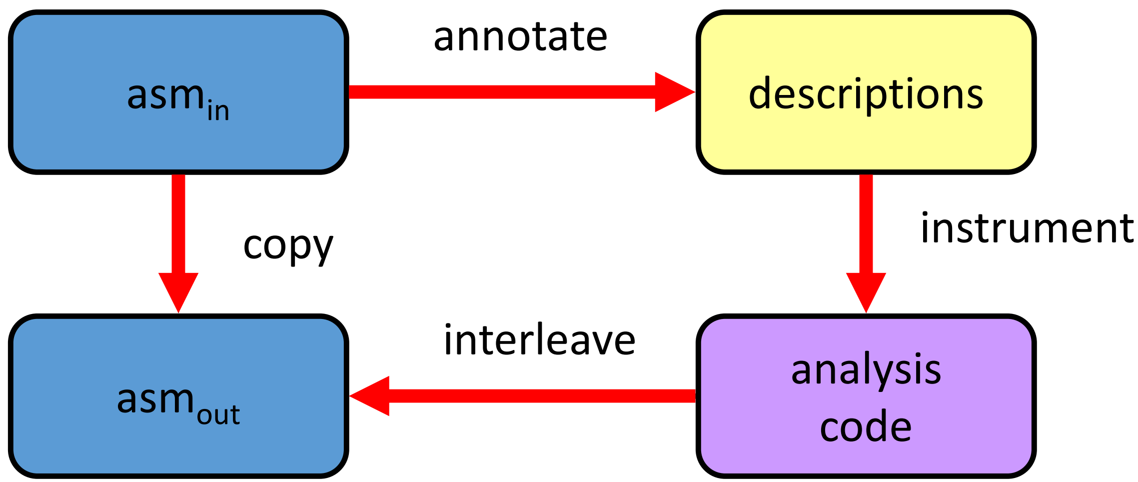
D&R

Disassemble-and-resynthesize (Valgrind)



C&A

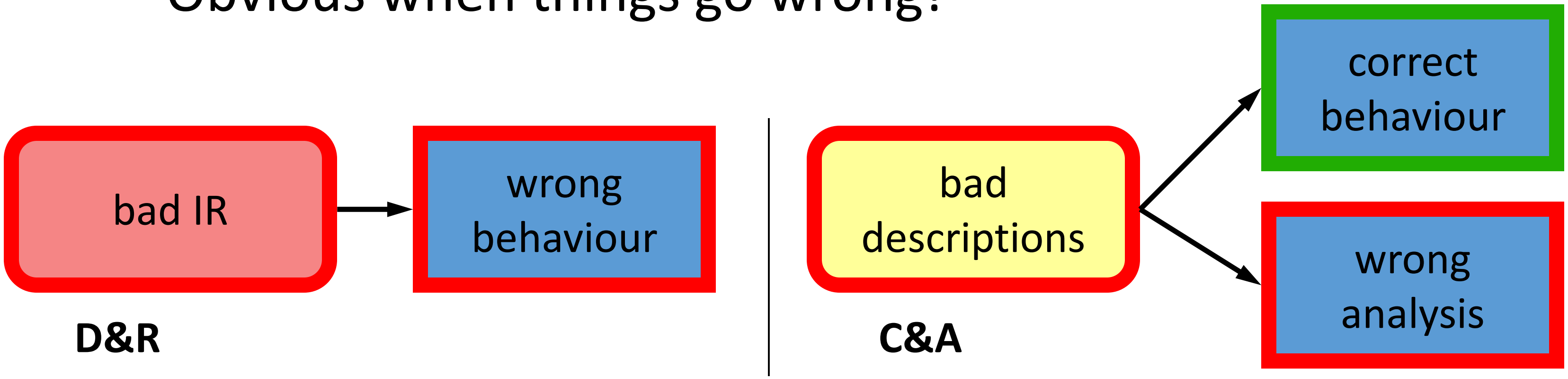
Copy-and-annotate



Pros and cons of D&R

- Cons: Lightweight tools
  - Framework design and implementation effort
  - Code translation cost, code quality

- Pros: Heavyweight tools
  - Analysis code as expressive as original code
  - Tight interleaving of original code and analysis code
  - Obvious when things go wrong!



Other IR features

Feature	Benefit
First-class shadow registers	As expressive as normal registers
Typed, SSA	Catches instrumentation errors
RISC-like	Fewer cases to handle
Infinitely many temporaries	Never have to find a spare register

- Writing complex inline analysis code is easy

#2: Thread serialisation

- Shadow memory: memory accesses no longer atomic
  - Uni-processors: thread switches may intervene
  - Multi-processors: real/shadow accesses may be reordered

- Simple solution: serialise thread execution!
  - Tools can ignore the issue
  - Great for uni-processors, slow for multi-processors...

Performance

SPEC2000 Performance

Valgrind, no-instrumentation	4.3x
Pin/DynRIO, no-instrumentation	~1.5x

Memcheck	22.1x (7--58x)
Most other shadow value tools	10--180x
LIFT	3.6x (*)

(*) LIFT limitations:

- No FP or SIMD programs
- No multi-threaded programs
- 32-bit x86 code on 64-bit x86 machines only

Post-performance

- Only Valgrind allows robust shadow value tools
  - All robust ones built with Valgrind or from scratch

- Perception: “Valgrind is slow”
  - Too simplistic
  - Beware apples-to-oranges comparisons
  - Different frameworks have different strengths

Future of DBI

The future

- Interesting tools!
    - Memcheck changed many C/C++ programmer's lives
    - Tools don't arise in a vacuum

- What do you want to know about program execution?
    - Think big!
    - Don't worry about being practical at first

If you remember nothing else...

Take-home messages

- Heavyweight tools are interesting
- Each DBI framework has its pros and cons
- Valgrind supports heavyweight tools well

www.valgrind.org