UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

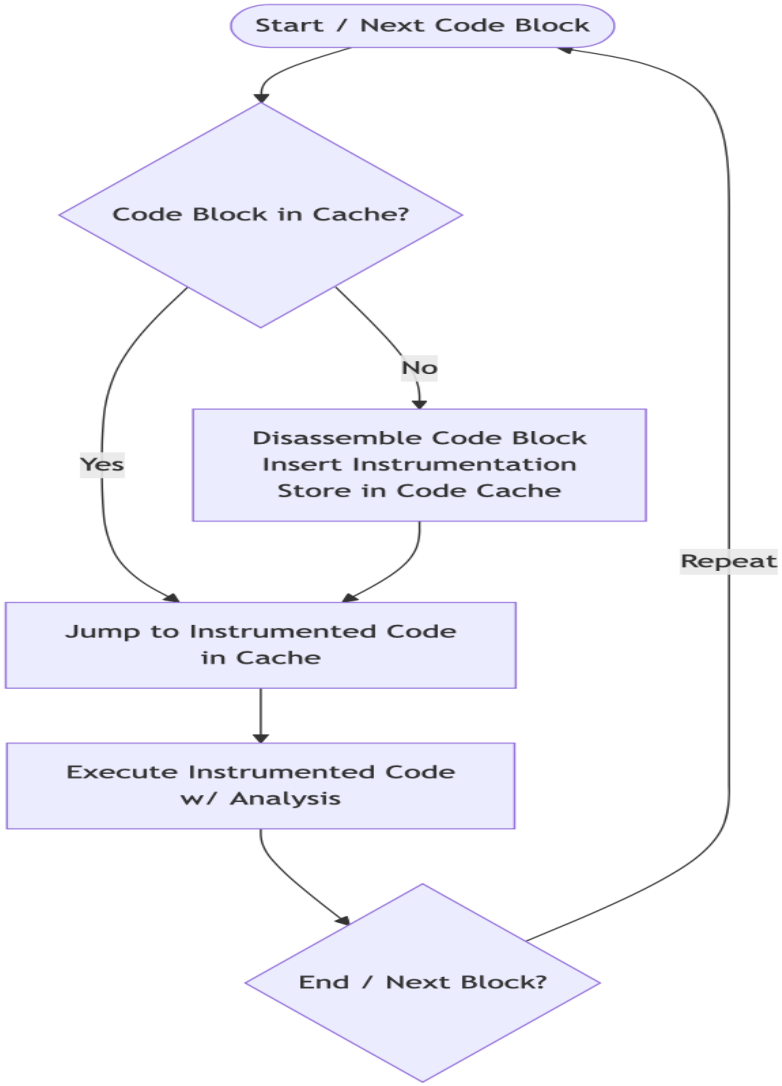
CS 250: Software Security

Dynamic Binary Translation & Instrumentation

What is Binary Instrumentation

- Insert analysis and monitoring code into a binary program
- Different from source code instrumentation
  - Add code during compilation process
- Applications
  - Collect runtime information for perf. optimization, greybox fuzzing, concolic execution, etc.
  - Protect the binary
- Two approaches
  - Static Instrumentation
    - Statically rewrite the binary
    - Hard, because correct disassembly is difficult
  - Dynamic Instrumentation
    - Insert code at runtime
    - No need to statically disassemble the code
    - More overhead

Dynamic Binary Instrumentation



DBI Tools

- Pin: A user-mode DBI tool from Intel
  - JIT-based rewriting of x86/x64 instructions
  - Provides an API to insert callbacks around instructions, basic blocks, functions, and system calls
  - Closed-source
- Valgrind: A framework best known for Memcheck
  - Translates binary code into an intermediate representation (VEX IR)
  - Insert analysis code (in form of IR)
  - Built for heavyweight analysis
  - Open-source
- QEMU: An emulator/virtualizer
  - Full-system mode and user mode
  - Can emulate different architectures
  - Translates binary code into an IR (TCG)
  - Open-source: not analysis tool, but many tools are built on top of it

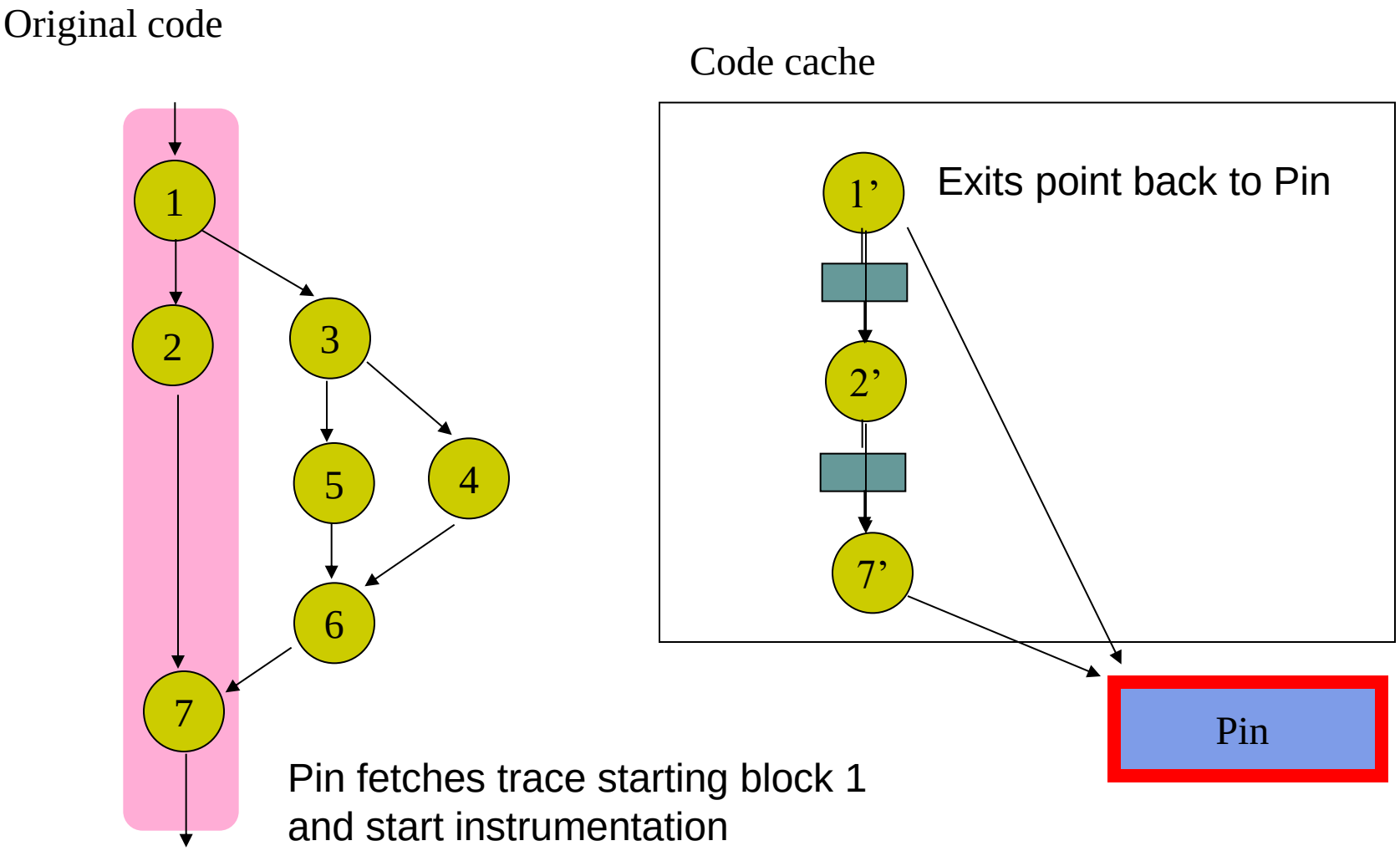
A Pintool for Tracing Memory Writes

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

FILE* trace;

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

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

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

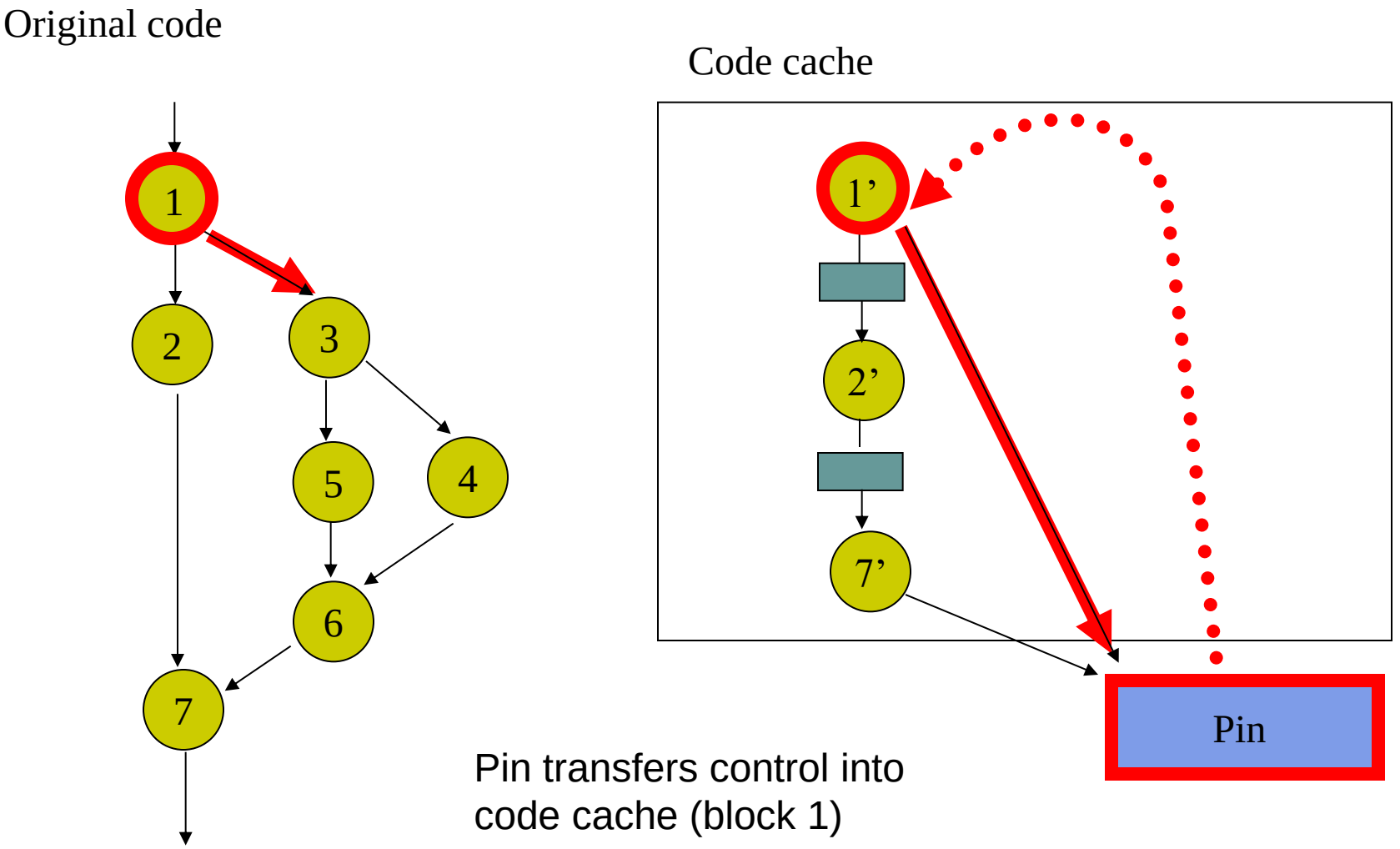
executed immediately before a write is executed

executed when an instruction is dynamically compiled

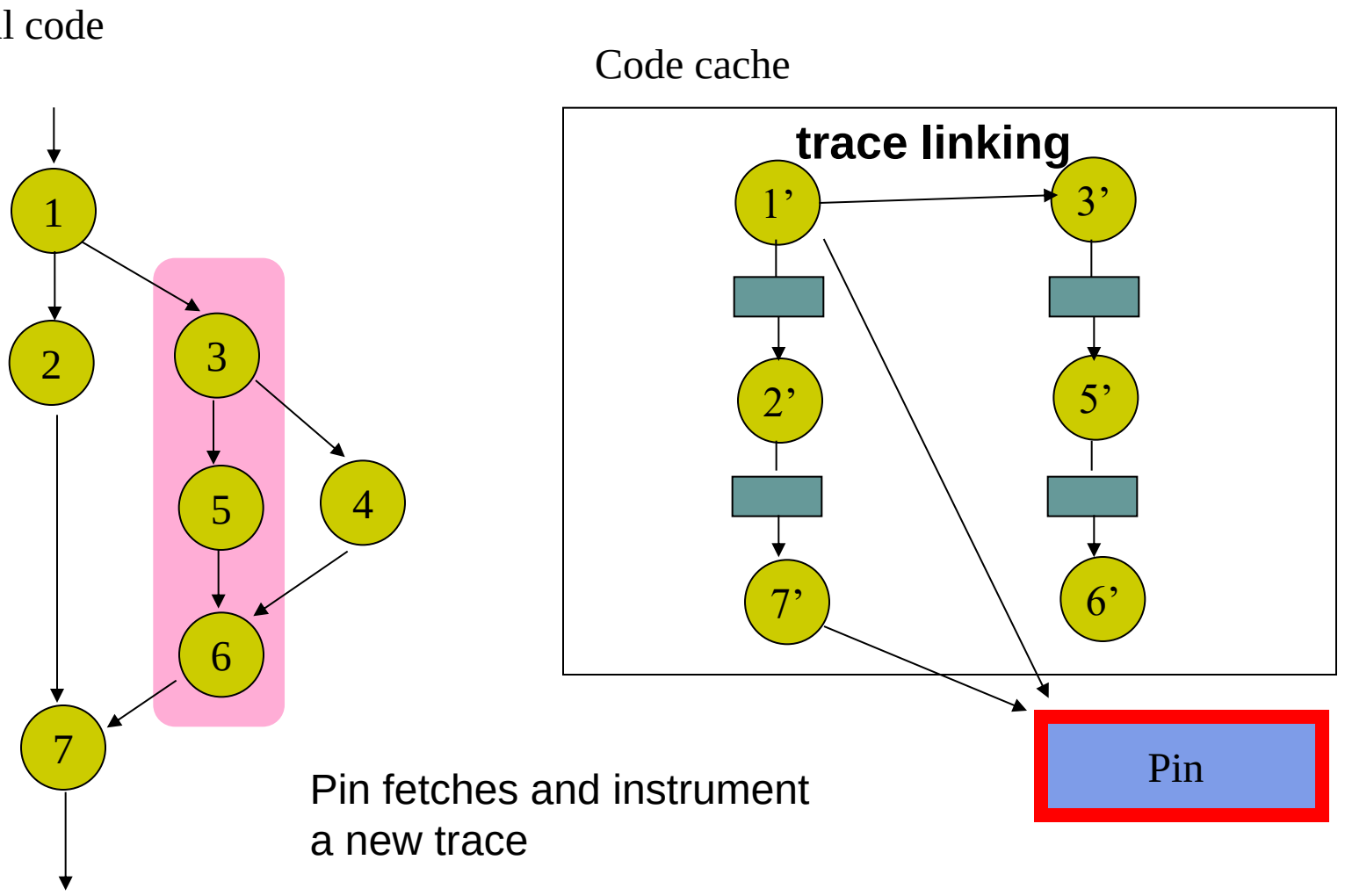
Dynamic Instrumentation



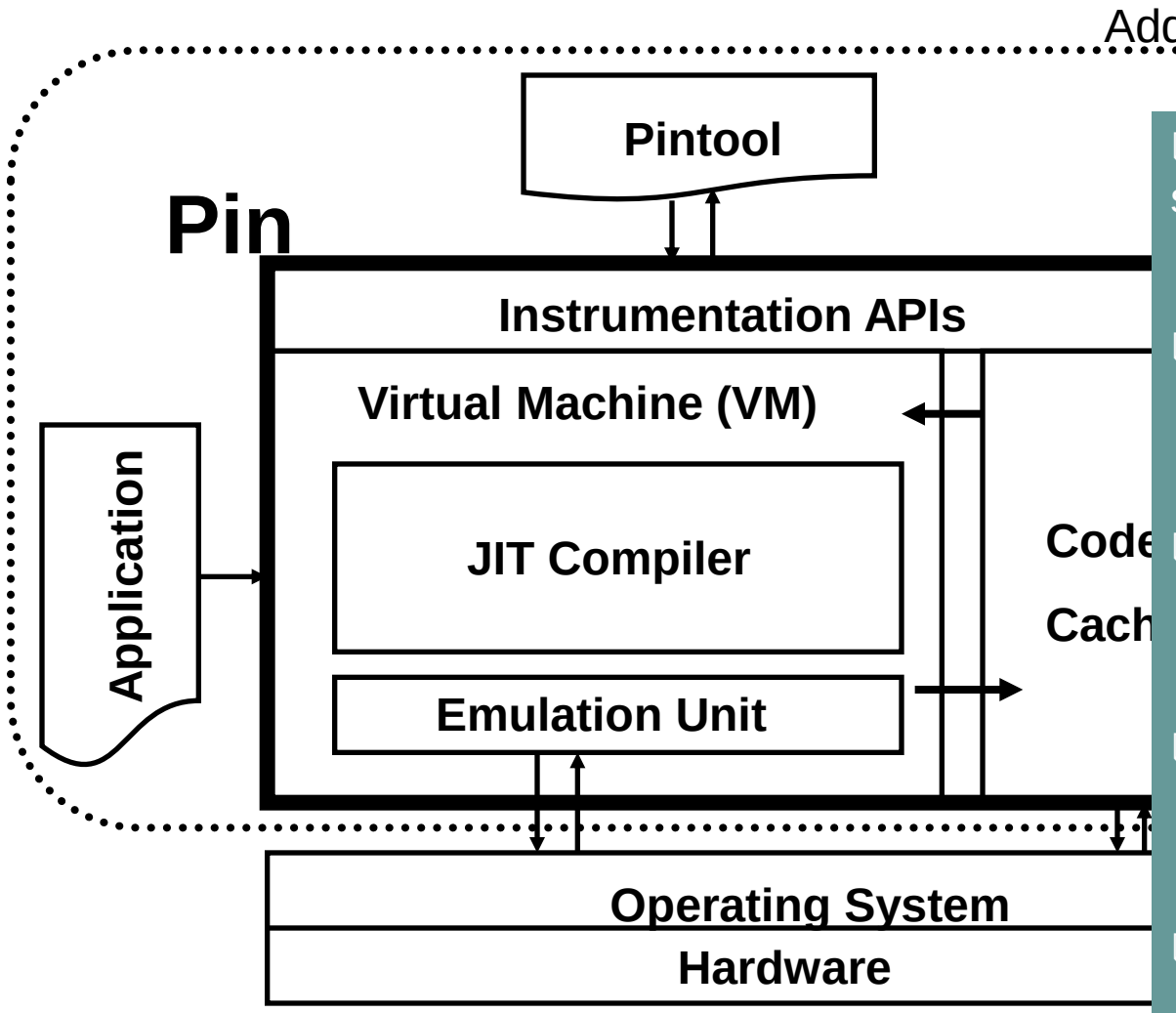
Dynamic Instrumentation



Dynamic Instrumentation



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

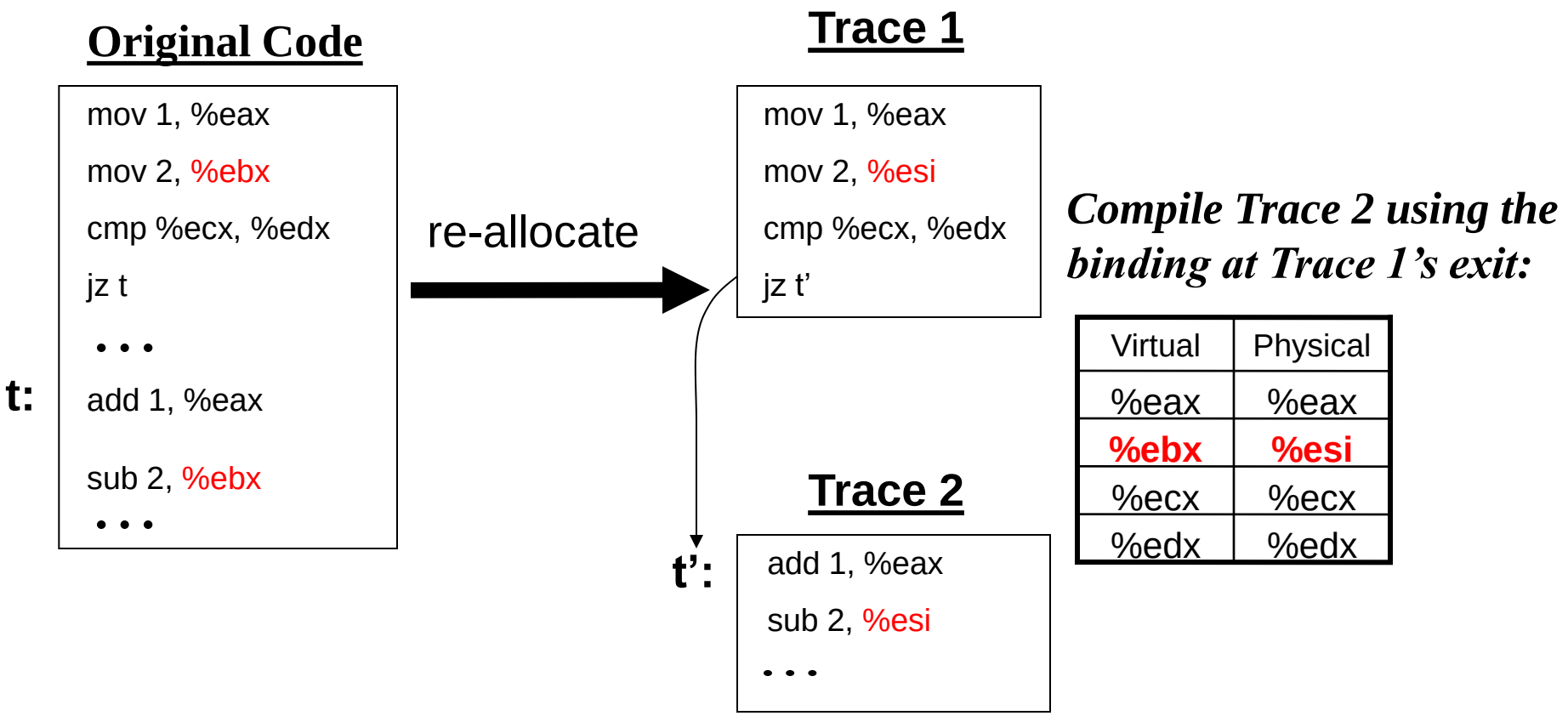
- Loading of Pin, Pintool, & Application
- An Improved Trace Linking Technique
- **Register Re-allocation**
- **Instrumentation Optimizations**
- Multithreading Support

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 …
- **Global allocation (Pin)**
  - **Allocate registers across traces (can be inter-procedural)**

Pin's Register Re-allocation

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



☞ *No spilling/filling needed across traces*

Pin's Register Re-allocation

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

Original Code

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

Original code

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

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

Inlining

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'>
```

```
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

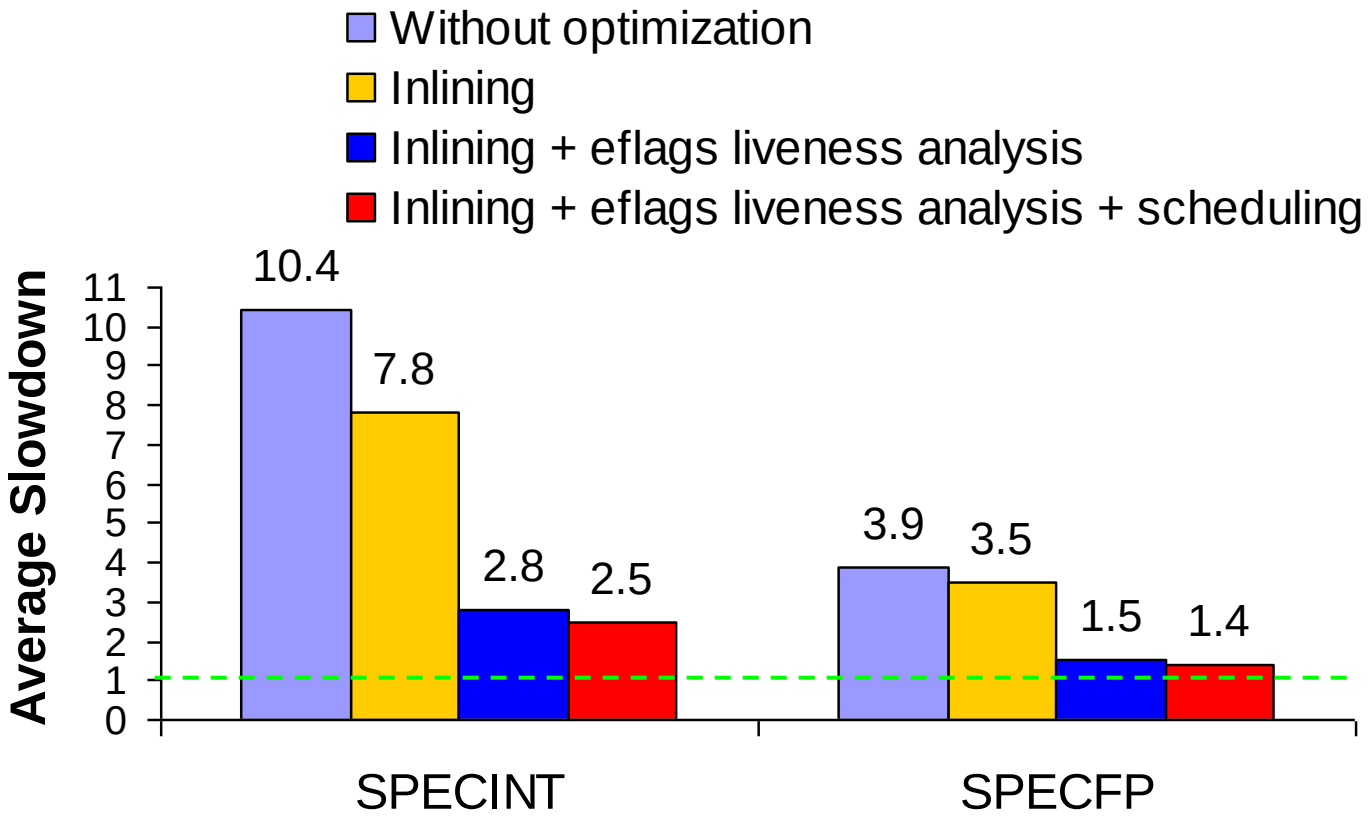
```
cmov %esi, %edi
add 0x3, icount
cmp %edi, (%esp)
jle <target1'>
```

```
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)

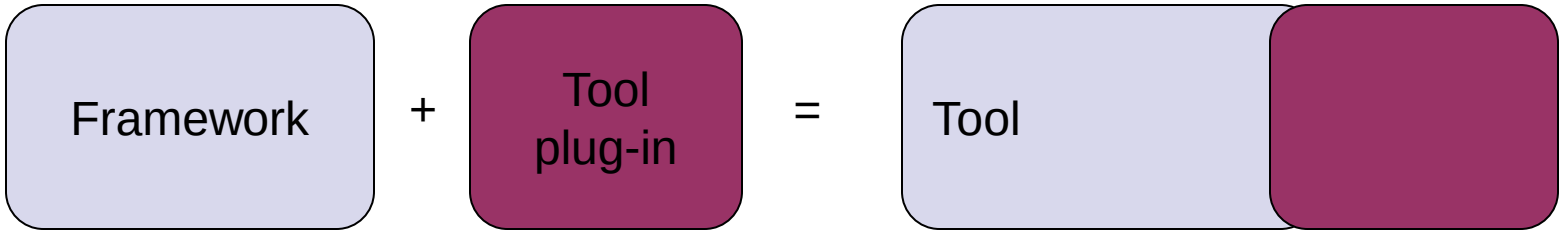
Valgrind
A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote — National ICT Australia

Julian Seward — OpenWorks LLP

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 (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
	Hobbes	Run-time type errors
security	TaintCheck, LIFT, TaintTrace	Uses of untrusted values
	“Secret tracker”	Leaked secrets
properties	DynCompB	Invariants
	Redux	Dynamic dataflow graphs

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)

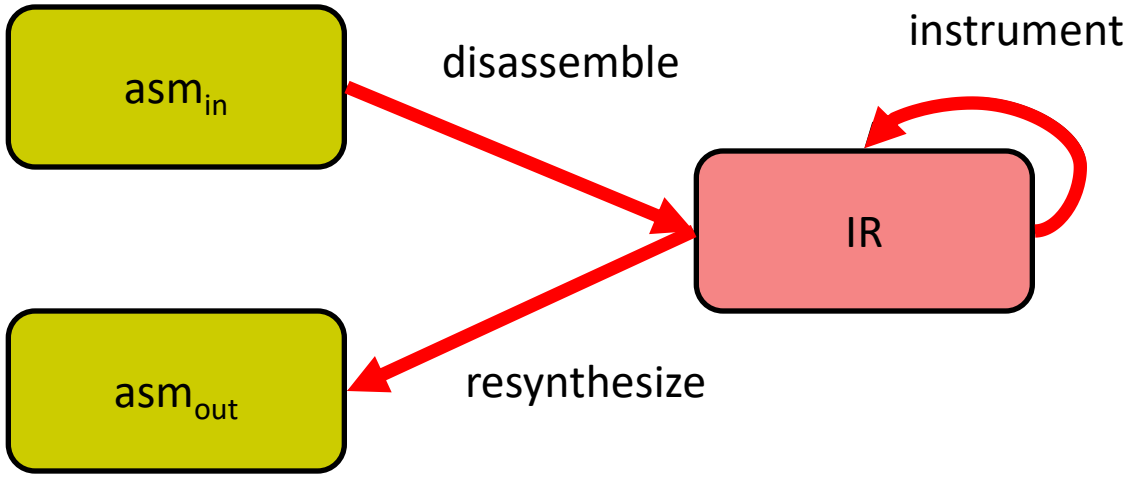
- All shadow value tools work in the same basic way
- Shadow value tools are **heavyweight** tools
  - Tool's data + ops are as complex as the original programs's
- Shadow value tools are hard to implement
  - Multiplex real and shadow registers onto register file
  - Squeeze real and shadow memory into address space
  - Instrument most instructions and system calls

Two unusual features of Valgrind

#1: Code Representation

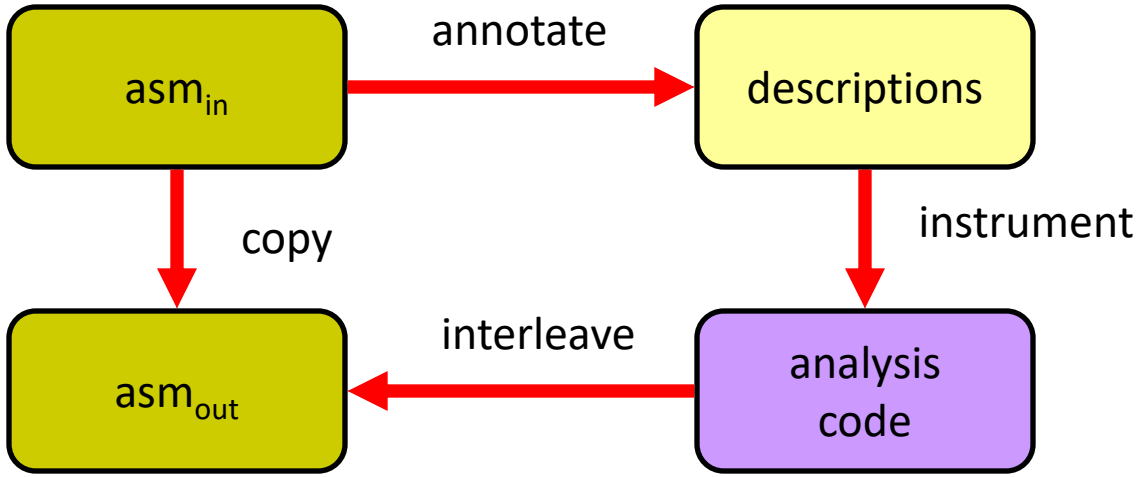
D&R

Disassemble-and-resynthesize (Valgrind)



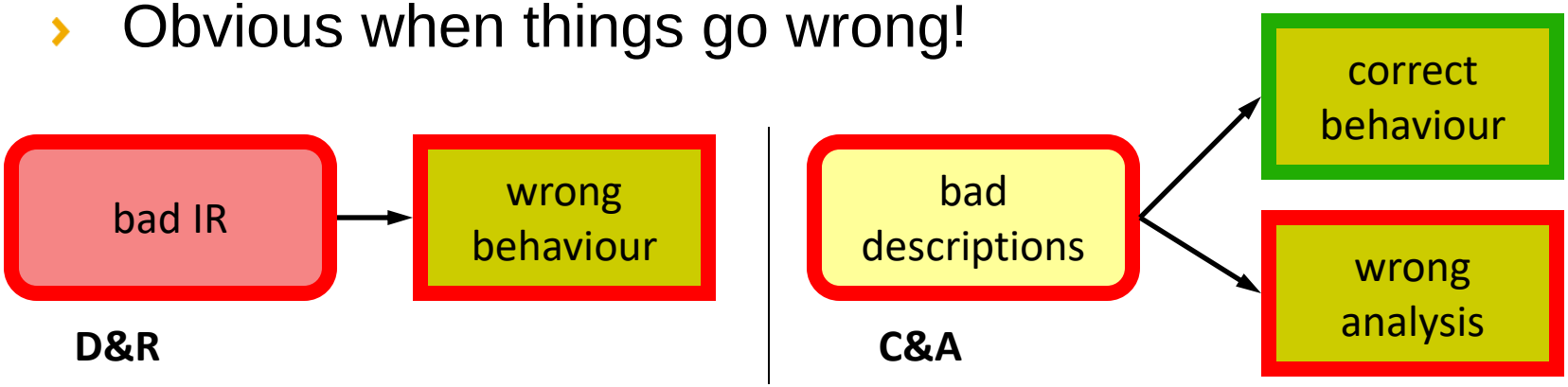
C&A

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 Serialization

- 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...

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