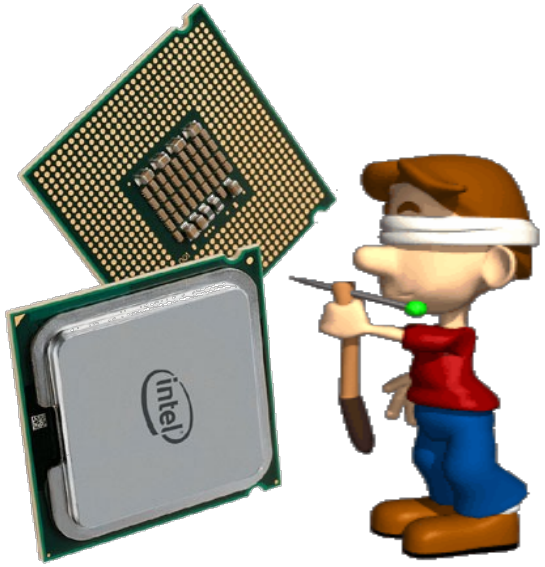
CSE 520
Computer Architecture II

Intel Pin Introduction*

Prof. Michel A. Kinsy

Why Instrumentation?

- Inspect the micro-architecture states of the chip without physically opening it up



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

Example: Instruction Count

Logically →

Add counter, 0x1
Actually →

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

How Pin Works – High Level

- What is modified
  - New instructions are added at user defined points
  - Static addresses and references
  - Register allocation
  - Pin stack
- What is executed
  - Instrumented traces
  - Code cache

How Pin Works – High Level

- When does the modification occur
  - At run time
  - Can attach to running process

Example: Instruction Trace

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

Instrumentation vs. Simulation

- Advantages of Simulation:
  - Detailed modeling of processors
  - Can model non-existing hardware
- Advantages of Instrumentation:
  - Easy to prototype
  - Fast to run (allowing complete runs)

Usage in Architecture

- How is Instrumentation used in Computer Architecture?
  - Trace Generation
  - Branch Predictor and Cache Modeling
  - Fault Tolerance Study
  - Emulating Speculation
  - Emulating New Instructions
  - Cache Coherence Protocols

What is Pin?

- 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 IA-32, EM64T, Itanium, Xscale
  - Supports Linux, Windows, MacOS

What is Pin?

- Robust:
  - Instruments real-life applications
    - Database, search engines, web browsers, …
  - Instruments multithreaded applications
- Efficient:
  - Applies compiler optimizations on instrumentation code

How to use Pin?

- Launch and instrument an application

$ pin –t pintool – application

Instrumentation engine (provided in kit)

Instrumentation tool

(write your own, or use one provided in 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 ++
```

Instruction Count Output

$ /bin/ls

```
Makefile atrace.o imageload.out itrace proccount Makefile.example
imageload inscount0 itrace.o proccount.o atrace imageload.o
inscount0.o itrace.out
```

$ pin -t inscount0 -- /bin/ls

```
Makefile atrace.o imageload.out itrace proccount Makefile.example
imageload inscount0 itrace.o proccount.o atrace imageload.o
inscount0.o itrace.out
```

Count 422838

ManualExamples/inscount0.C

```
#include <iostream>
#include "pin.h"
UINT64 icount = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
                            "results.out", "specify output file");
void docount() { icount++; }
```

analysis routine

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

instrumentation routine

```
void Fini(INT32 code, void *v)
{
     FILE* outfile = fopen(KnobOutputFile.Value().c_str(),"w");
     fprintf(outfile, "Count %d\n", icount);
}

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

ManualExamples/inscount0.C

- Same source code works on the 4 architectures
- Pin automatically and efficiently saves/restores application state

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 an argument (ip) to the analysis routine (printip())

Instruction Trace Output

$ pin -t itrace -- /bin/ls

```
Makefile atrace.o imageload.out itrace proccount Makefile.example
imageload inscount0 itrace.o proccount.o atrace imageload.o
inscount0.o itrace.out
```

$ head -4 itrace.out

```
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
```

ManualExamples/itrace.C

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

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;
}
```

Argument to analysis routine

analysis routine

instrumentation routine

Arguments to Analysis Routine

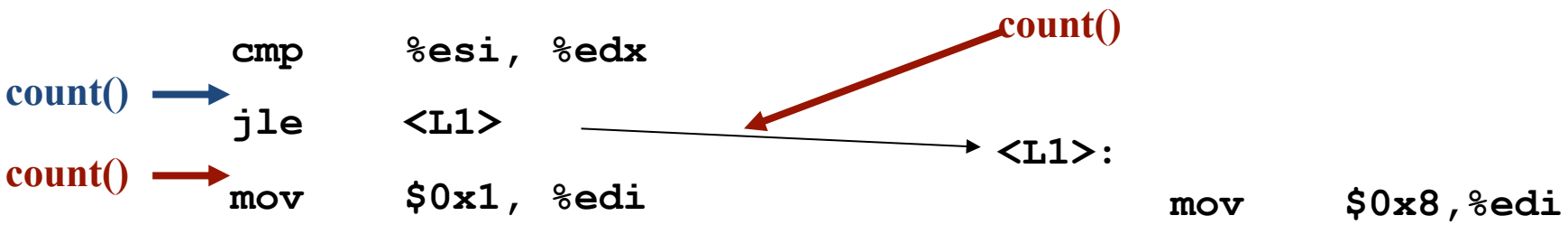
- IARG_INST_PTR
  - Instruction pointer (program counter) value
- IARG_PTR <pointer>
  - A pointer to some data
- IARG_REG_VALUE <register name>
  - Value of the register specified
- IARG_BRANCH_TARGET_ADDR
  - Target address of the branch instrumented

Arguments to Analysis Routine

- 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)



Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:
  - Instruction
  - Basic block
    - A sequence of instructions terminated at a (conditional or unconditional) 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

Instrumentation Granularity

- 1 Trace, 2 basic blocks, 6 instructions

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

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

Instruction Count

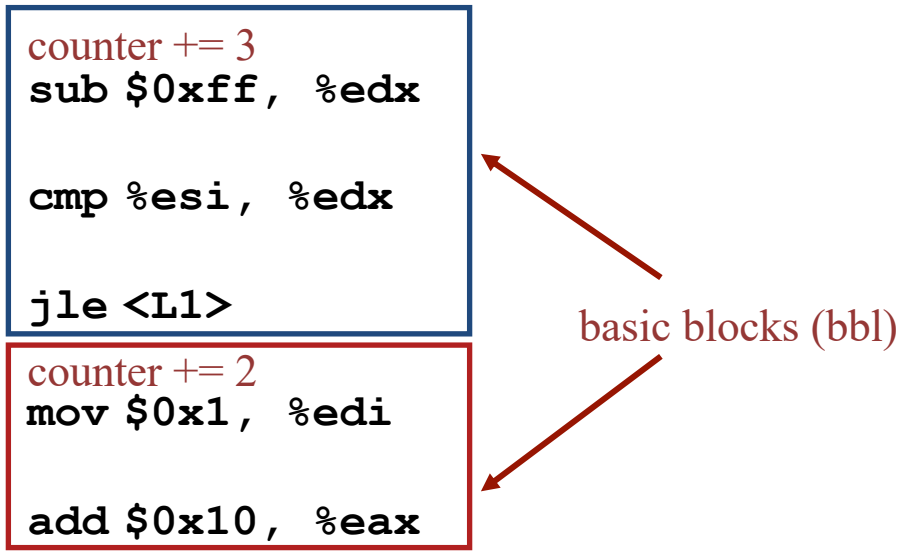
- Recap of Pintool 1: Instruction Count

```
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

Faster Instruction Count

- Reduce the number of calls made to analysis routine



ManualExamples/inscount1.C

```
#include <iostream>
#include "pin.h"
UINT64 icount = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
                            "results.out", "specify output file");
void docount(INT32 c) {icount += c; }
```

analysis routine

```
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);}
}
```

instrumentation routine

```
void Fini(INT32 code, void *v)
{
     FILE* outfile = fopen(KnobOutputFile.Value().c_str(),"w");
     fprintf(outfile, "Count %d\n", icount);
}
int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Modifying Program Behavior

- Pin allows you not only observing but also changing program behavior
- Ways to change program behavior:
  - Add/delete instructions
  - Change register values
  - Change memory values
  - Change control flow
  - Inject errors

Example: Emulation of Loads

```
sub     $0x11c,%esp
mov     0xc(%ebp),%eax
add     $0x128, %eax
mov     0x8(%ebp),%edi
xor     %eax, %edi
```

Multithreading Support

- Notify the pintool when a thread is created or exited
- Provide a “thread id” for pintools to identify a thread
- Provide locks for pintools to access shared data structures

Multithreaded Programs

```
$ pin -mt -t mtest -- thread
Creating thread
Creating thread
Joined 0
Joined 1
$ cat mtest.out
0x400109a8: 0
thread begin 1 sp 0x80acc00 flags f00
0x40001d38: 1
thread begin 3 sp 0x43305bd8 flags f21
0x40011220: 3
thread begin 2 sp 0x42302bd8 flags f21
0x40010e15: 2
0x40005cdc: 2
thread end 3 code 0
0x40005e90: 0
0x40005e90: 0
thread end 2 code 0
thread end 1 code 0
```

Debugging Pintools

- Invoke gdb with your pintool (but don't use "run")

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

- On another window, start your pintool with "-pause_tool"

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

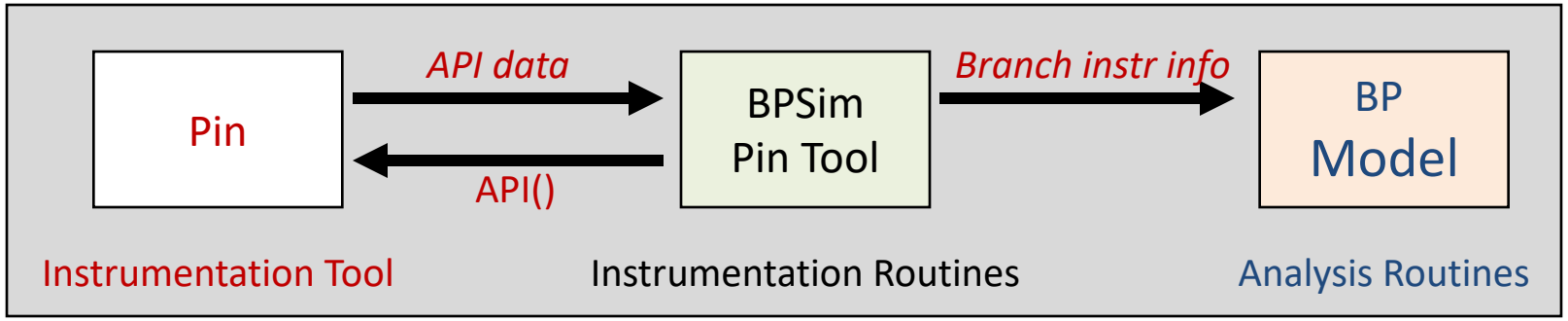
- Go back to gdb:
  - Attach to the process
  - Use "cont" to continue execution; can set breakpoints as usual

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

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)

Branch Predictor Model



- BPSim Pin Tool
  - Instruments all branches
  - Uses API to set up call backs to analysis routines
- Branch Predictor Model:
  - Detailed branch predictor simulator

Branch Predictor Implementation

ANALYSIS

```
BranchPredictor myBPU;

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
  }
}
```

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 Implementation

ANALYSIS

```
BranchPredictor myBPU;

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
  }
}
```

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 Implementation

ANALYSIS

```
BranchPredictor myBPU;

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
  }
}
```

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);
}
```

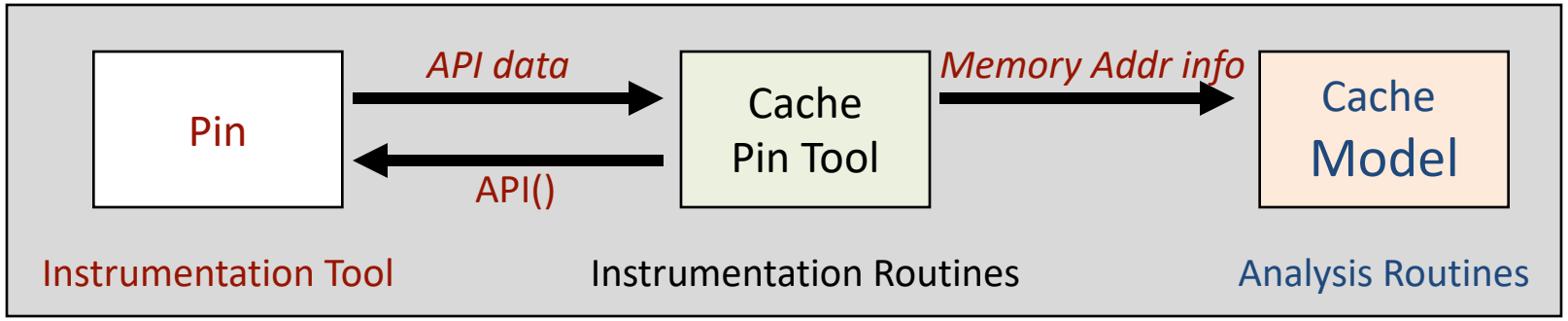
MAIN

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

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)

Cache Simulators



- 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

ANALYSIS

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

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();
}
```

Cache Implementation

ANALYSIS

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

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();
}
```

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);
  }
}
```

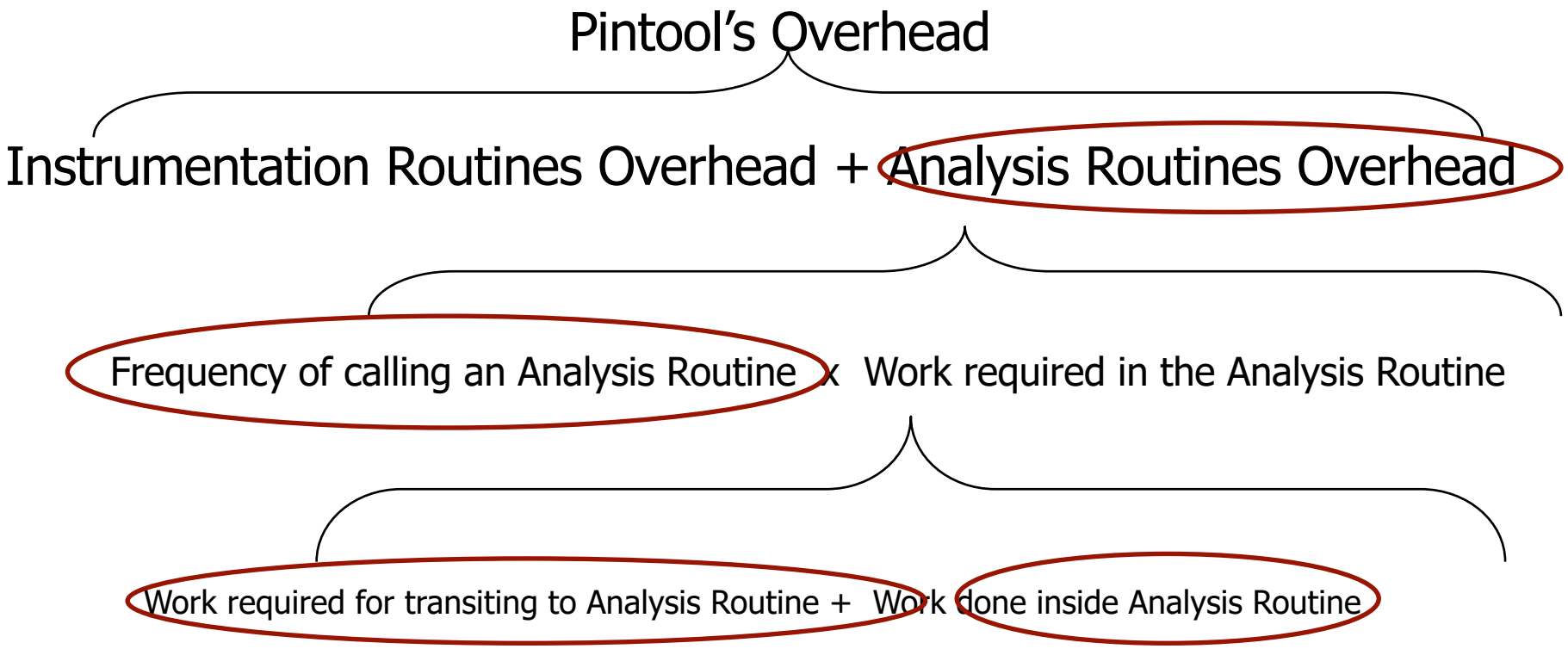
INSTRUMENT

```
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();
}
```

Reducing Pintool's Overhead



Optimization

- Reducing Frequency of Calling Analysis Routines
  - Key:
    - Instrument at the largest granularity whenever possible:
      - Trace > Basic Block > Instruction

Conclusions

- Pin
  - Build your own architectural tools with ease
  - Run on multiple platforms:
    - IA-32, EM64T, Itanium, and XScale
    - Linux, Windows, MacOS
  - Work on real-life applications
  - Efficient instrumentation

Next Lecture Module

- Single-cycle & Multi-cycle Architectures