
intel
Leap ahead™
12th EMEA Academic Forum
EMEA ACADEMIC
FORUM

Dynamic Binary Instrumentation and Tools for Supporting Multi-threaded Applications

Moshe Bach
Intel Corporation
Israel Design Center
Software Solutions Group

June 13, 2007

12th EMEA Academic Forum

EMEA ACADEMIC FORUM

Outline

- Basic description of Pin
- Writing Pin tools
  - Simple examples of Pin use
- Intel software products and their use of Pin

What is Pin?

- Easy-to-use Instrumentation:
  - Tool for dynamic instrumentation
    - Inject C/C++ code into an executing program at arbitrary places to collect run-time information
    - Do not need source code, recompilation, post-linking
- Why instrumentation?
  - Observe program behavior
    - Eg Application profilers, memory leak detector, trace generator, cache simulators
- Programmable Instrumentation:
  - Rich APIs to write your own instrumentation tools (called Pintools) in C/C++
- Multiplatform:
  - Supports IA-32, Intel64, IA-64, supports Linux, Windows, MacOS
- Robust:
  - Instruments real-life, multithreaded applications
    - Database, search engines, web browsers, …
- Efficient:
  - Applies compiler optimizations to instrumentation code

How to use Pin?

- Launch and instrument an application

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

Instrumentation engine (provided in our kit)

Instrumentation tool (write your own, or use one provided in our kit)

- Attach to and instrument an application

```
$ pin –t pintool –pid 1234
```

Outline

- Basic description of Pin
- Writing Pin tools
  - Simple examples of Pin use
- Intel software products and their use of Pin

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 IA-32
- Call-based APIs:
  - Instrumentation routines
    - Define where instrumentation is inserted
    - Occurs first time an instruction is executed
  - Analysis routines
    - Defines what to do when instrumentation is activated
    - Occurs every time instrumentation is activated

Pintool 1: Instruction Count

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

Pintool 1: 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;

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)
{ 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;
}
```

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
```

Pintool 2: 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); }
```

analysis routine

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

instrumentation routine

```
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;
}
```

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

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

Outline

- Basic description of Pin
- Writing Pin tools
  - Simple examples of Pin use
- Intel software technology and use of Pin

How to Implement a Simple Call Graph

- Instrument the entry to each function and its return
  - Track tree of functions
  - Track time of entry, return from each function
- Result
  - Call graph of program
  - Time spent in subtrees

How to Implement a Simple Memory Checker

- Replace all heap-oriented calls
  - Malloc, free, reallloc etc
  - Save memory attributes:
    - Allocated
    - Freed (to check access after memory was freed)
    - Init'ed or uninit'ed addresses
- Instrument all non-stack memory instructions
  - Analysis: check address, size, read/write of memory access
  - Is address legal? Read from uninitialized memory?
- Fini – memory leaks

How to Implement a Simple Thread Checker

- Intercept thread creation, sync functions
  - Enter critical section, increment a Lamport-style time cookie
  - Check mutex to verify memory accesses are synced
- Memory instructions
  - Analysis: address, size, data type, thread id
  - Check if there is memory contention between different threads
    - "Simultaneous" unprotected writes to memory
    - Use of memory before write
  - Check for deadlocks in mutex's

How to Implement a Simple Thread Profiler

- Instrument sync primitives (enter/exit critical section)
  - Record thread ID when entering a critical section
  - Record time of entry/exit of critical section
    - Identify heavily used critical sections
  - Build time graph when thread is active, when it waits.
    - Show bottlenecks
    - Stack walk to see how you got there

How to Implement a Simple MPI Message Tracer

- Redirection of MPI functions to a private library
  - Trace MPI calls
- Invoke Pin on child processes
  - Follow messages between processes.
- Check to which nodes you send info, time to get return message, trip counts

Pin Usage @ Universities

University	*PinTool / Pin Enhancement*
Maryland	Multi-processor Cache Simulator
MIT	Driver for a memory simulator
UCSD	Deterministic re-player for multi-threaded programs
Colorado	Post-link analyzer
Princeton	Software reliability enhancer
Colorado	Persistence support in Pin
Harvard	Code transformer for dealing with power emergencies

Conclusions

Pin

- Build your own Pin tools with ease
- Run on multiple platforms:
  - IA-32, Intel64, IA-64
  - Linux, Windows, MacOS
- Work on real-life applications
- Efficient instrumentation

Call For Action

Try it out!

Free download from:

http://rogue.colorado.edu/pin

* User manual, many example tools, tutorials
* >10,000 downloads since 2004 July

Group: **http://groups.yahoo.com/group/pinheads/**

Email: ***pinheads@yahoogroups.com***

Backup

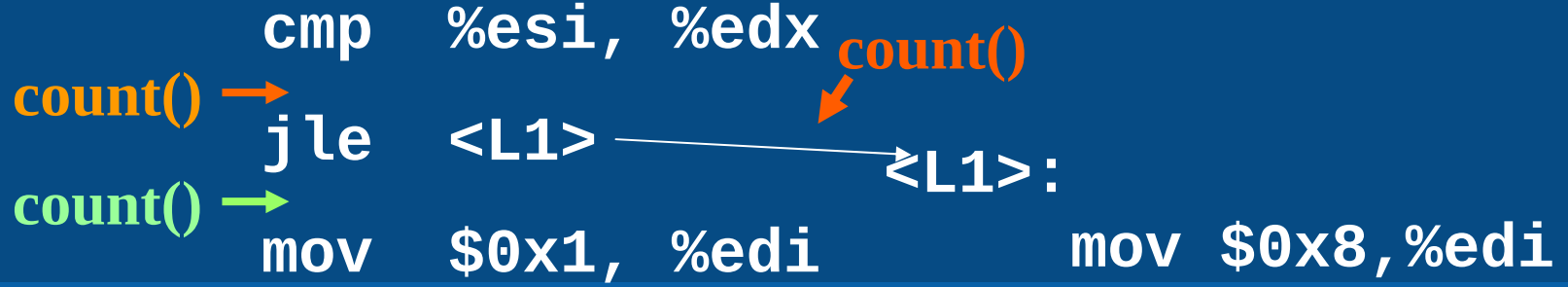
Examples of Arguments to Analysis Routine

- **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

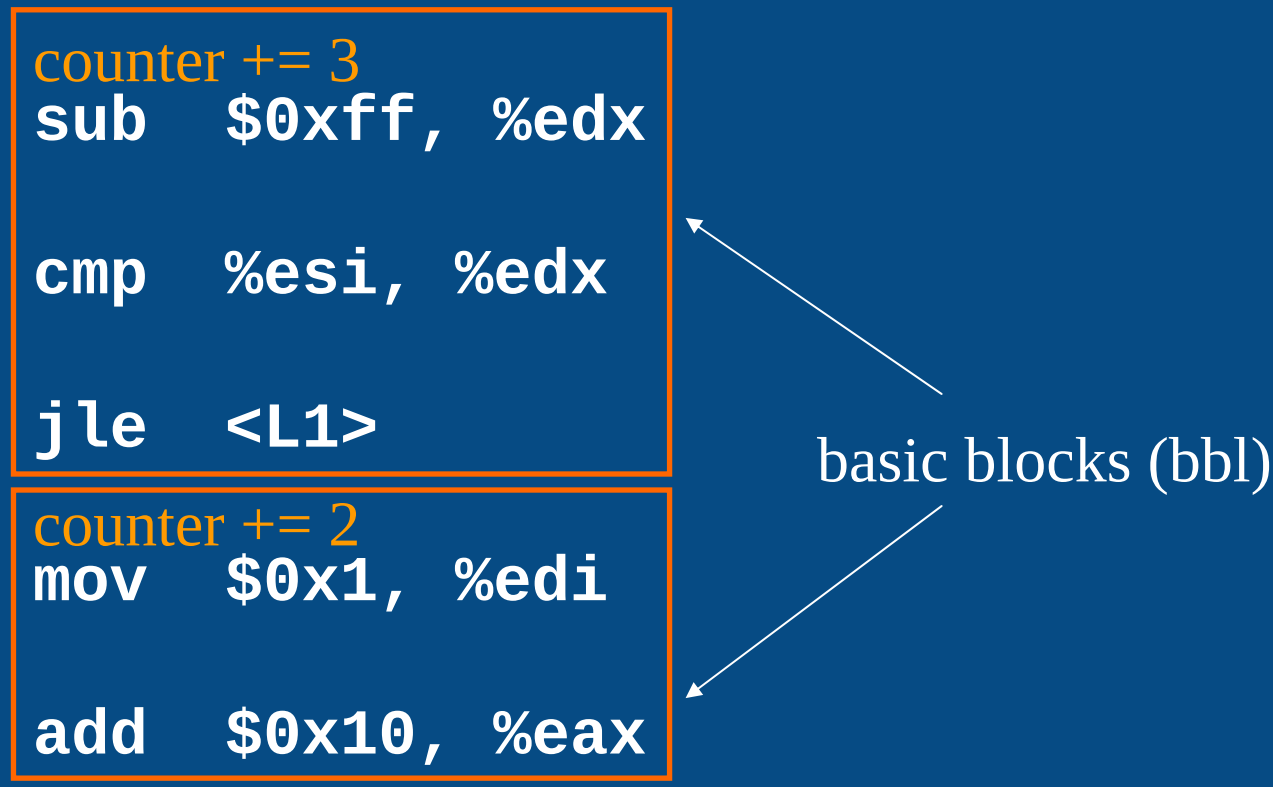
And many more … (refer to the Pin manual for details)

Instrumentation Points

- Instrument points relative to an instruction:
  - *Before (IPOINT_BEFORE)*
  - After:
    - Fall-through edge (IPOINT_AFTER)
    - **Taken edge (IPOINT_TAKEN)**



Pintool 3: Faster Instruction Count



ManualExamples/inscount1.C

```
#include <stdio.h>
#include "pin.H“
UINT64 icount = 0;

void docount(INT32 c) { icount += c; }          analysis routine
void Trace(TRACE trace, void *v) {          instrumentation routine
  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;
}
```

Example of Instrumenting 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
```

Tests/mtest.C

```
FILE * out;
PIN_LOCK lock;

VOID TraceBegin(VOID * ip, UINT32 threadid) {
    GetLock(&lock, threadid+1);
    fprintf(out, "%p: %d\n", ip, threadid);
    ReleaseLock(&lock);
}
VOID Trace(TRACE trace, VOID *v) {
    TRACE_InsertCall(trace, IPOINT_BEFORE, AFUNPTR(TraceBegin),
          IARG_INST_PTR, IARG_THREAD_ID, IARG_END);
}
VOID ThreadBegin(UINT32 threadid, VOID * sp, int flags, VOID *v) {
    GetLock(&lock, threadid+1);
    fprintf(out, "thread begin %d sp %p flags %x\n",threadid, sp, flags);
    ReleaseLock(&lock);
}
VOID ThreadEnd(UINT32 threadid, INT32 code, VOID *v) {
    GetLock(&lock, threadid+1);
    fprintf(out, "thread end %d code %d\n",threadid, code);
    ReleaseLock(&lock);
}
VOID Fini(INT32 code, VOID *v) {
    fprintf(out, "Fini: code %d\n", code);
}
int main(INT32 argc, CHAR **argv) {
    InitLock(&lock);
    out = fopen("mtest.out", "w");
    PIN_Init(argc, argv);
    PIN_AddThreadBeginFunction(ThreadBegin, 0);
    PIN_AddThreadEndFunction(ThreadEnd, 0);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

analysis routine

instrumentation routines

12th EMEA Academic Forum