

## Pin 2.14 User Guide

### Introduction

Pin is a tool for the instrumentation of programs. It supports the Android\*, Linux\*, OS X\* and Windows\* operating systems and executables for the IA-32, Intel(R) 64 and Intel(R) Many Integrated Core architectures.

Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

Pin provides a rich API that abstracts away the underlying instruction set idiosyncracies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. Limited access to symbol and debug information is available as well.

Pin includes the source code for a large number of example instrumentation tools like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new tools using the examples as a template.

#### Tutorial Sections

- [How to Instrument With Pin](#)
- [Examples](#)
- [Callbacks](#)
- [Modifying Application Instructions](#)
- [The Pin Advanced Debugging Extensions](#)
- [Applying a Pintool to an Application](#)
- [Debugging Tips for Debugging a Pintool](#)
- [Recording Messages from a Pintool](#)
- [Performance Considerations](#)
- [Memory management](#)
- [PinTools Information and Restrictions](#)
- [Building Tools on windows](#)
- [Libraries for Windows](#)
- [Libraries for Linux](#)
- [Installation](#)
- [Usage Instructions for Intel\(R\) Xeon Phi\(TM\)](#)
- [Building Your Own Tool](#)
- [Pin's makefile Infrastructure](#)
- [Feedback](#)
- [Disclaimer and Legal Information](#)

#### Reference Sections

- [Pin API reference](#)
- [Pin Command Line Switches](#)
- [Instrumentation Library](#)

### How to Instrument with Pin

#### Table of Contents

- [Pin](#)
- [Pintools](#)
- [Observations](#)
- [Instrumentation Granularity](#)
- [Managed platforms support](#)
- [Symbols](#)
- [Floating Point Support in Analysis Routines](#)
- [Instrumenting Multi-threaded Applications](#)
- [Avoiding Deadlocks in Multi-threaded Applications](#)

#### Pin

The best way to think about Pin is as a **"just in time" (JIT) compiler**. The input to this compiler is not bytecode, however, but a regular executable. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. **Pin makes this efficient by keeping all of the generated code in memory so it can be reused and directly branching from one sequence to another.**

**In JIT mode, the only code ever executed is the generated code. The original code is only used for reference. When generating code, Pin gives the user an opportunity to inject their own code (instrumentation).**

Pin instruments all instructions that are actually executed. It does not matter in what section they reside. Although there are some exceptions for conditional branches, generally speaking, if an instruction is never executed then it will not be instrumented.

#### Pintools

Conceptually, instrumentation consists of two components:

- **A mechanism that decides where and what code is inserted**
- **The code to execute at insertion points**

**These two components are *instrumentation* and *analysis* code.** Both components live in a single executable, a *Pintool*. **Pintools can be thought of as plugins that can modify the code generation process inside Pin.**

The Pintool registers instrumentation callback routines with Pin that are called from Pin whenever new code needs to be generated. This instrumentation callback routine represents the instrumentation component. It inspects the code to be generated, investigates its static properties, and decides if and where to inject calls to analysis functions.

The analysis function gathers data about the application. Pin makes sure that the **integer and floating point register state is saved** and restored as necessary and allow arguments to be passed to the functions.

**The Pintool can also register notification callback routines for events such as thread creation or forking.** These callbacks are generally used to gather data or tool initialization or clean up.

### Observations

**Since a Pintool works like a plugin, it must run in the same address space as Pin and the executable to be instrumented.** Hence the Pintool has access to all of the executable's data. It also shares file descriptors and other process information with the executable.

**Pin and the Pintool control a program starting with the very first instruction.** For executables compiled with shared libraries this implies that the execution of the dynamic loader and all shared libraries will be visible to the Pintool.

When writing tools, it is more important to tune the analysis code than the instrumentation code. This is because the instrumentation is executed once, but analysis code is called many times.

### Instrumentation Granularity

As described above, Pin's instrumentation is "just in time" (JIT). Instrumentation occurs immediately before a code sequence is executed for the first time. We call this mode of operation **trace instrumentation**.

Trace instrumentation lets the Pintool inspect and instrument an executable one trace at a time. Traces usually begin at the target of a taken branch and end with an unconditional branch, including calls and returns. Pin guarantees that a trace is only entered at the top, but it may contain multiple exits. If a branch joins the middle of a trace, Pin constructs a new trace that begins with the branch target. **Pin breaks the trace into basic blocks, BBLs. A BBL is a single entrance, single exit sequence of instructions.** Branches to the middle of a bbl begin a new trace and hence a new BBL. It is often possible to insert a single analysis call for a BBL, instead of one analysis call for every instruction. **Reducing the number of analysis calls makes instrumentation more efficient. Trace instrumentation utilizes the TRACE\_AddInstrumentFunction API call.**

Note, though, that since Pin is discovering the **control flow of the program dynamically as it executes**, Pin's BBL can be different from the classical definition of a BBL which you will find in a compiler textbook. For instance, consider the code generated for the body of a switch statement like this

```
switch(i)
{
    case 4: total++;
    case 3: total++;
    case 2: total++;
    case 1: total++;
    case 0:
        default: break;
}
```

It will generate instructions something like this (for the IA-32 architecture)

```
.L7:
    addl    $1, -4(%ebp)
.L6:
    addl    $1, -4(%ebp)
.L5:
    addl    $1, -4(%ebp)
.L4:
    addl    $1, -4(%ebp)
```

In terms of classical basic blocks, each addl instruction is in a single instruction basic block. However as the different switch cases are executed, Pin will generate BBLs which contain all four instructions (when the .L7 case is entered), three instructions (when the .L6 case is entered), and so on. This means that counting Pin BBLs is unlikely to give the count you would expect if you thought that Pin BBLs were the same as the basic blocks in the text book. Here, for instance, if the code branches to .L7 you will count one Pin BBL, but there are four classical basic blocks executed.

Pin also breaks BBLs on some other instructions which may be unexpected, for instance cpushd, popfd and REP prefixed instructions all end traces and therefore BBLs. Since REP prefixed instructions are treated as implicit loops, if a REP prefixed instruction iterates more than once, iterations after the first will cause a single instruction BBL to be generated, so in this case you would see more basic blocks executed than you might expect.

As a convenience for Pintool writers, Pin also offers an **instruction instrumentation mode** which lets the tool inspect and instrument an executable a single instruction at a time. This is essentially identical to trace instrumentation where the Pintool writer has been freed from the responsibility of iterating over the instructions inside a trace. As described under trace instrumentation, certain BBLs and the instructions inside of them may be generated (and hence instrumented) multiple times. **Instruction instrumentation utilizes the INS\_AddInstrumentFunction API call.**

Sometimes, however, it can be useful to look at different granularity than a trace. **For this purpose Pin offers two additional modes: image and routine instrumentation. These modes are implemented by "caching" instrumentation requests and hence incur a space overhead, these modes are also referred to as ahead-of-time instrumentation.**

**Image instrumentation lets the Pintool inspect and instrument an entire image, IMG, when it is first loaded.** A Pintool can walk the sections, SEC, of the image, the routines, RTN, of a section, and the instructions, INS, of a routine. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. Image instrumentation utilizes the **IMG\_AddInstrumentFunction** API call. **Image instrumentation depends on symbol information to determine routine boundaries hence PIN\_InitSymbols must be called before PIN\_Init.**

**Routine instrumentation lets the Pintool inspect and instrument an entire routine when the image it is contained in is first loaded.** A Pintool can walk the instructions of a routine. There is **not enough information available to break the instructions into BBLs. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed.** Routine instrumentation is provided as a convenience for Pintool writers, as an alternative to walking the sections and routines of the image during the Image instrumentation, as described in the previous paragraph.

Routine instrumentation utilizes the **RTN\_AddInstrumentFunction** API call. Instrumentation of routine exits does not work reliably in the presence of tail calls or when return instructions cannot reliably be detected.

Note that in both Image and Routine instrumentation, it is not possible to know whether or not a routine will actually be executed (since these instrumentations are done at image load time). **It is possible to walk the instructions only of routines that are executed, in the Trace or Instruction instrumentation routines,** by identifying instructions that are the start of routines. See the tool Tests/parse\_executed\_rtns.cpp.

### Managed platforms support

Pin supports all executables including the managed binaries. From Pin point of view managed binary is one more kind of a self-modifying program. There is a way to cause Pin to differentiate the just-in-time compiled code (Jitted code) from all other dynamically generated code and associate Jitted code with appropriate managed functions. To get this functionality, the just-in-time compiler (Jitter) of the running managed platform should support [Jit Profiling API](#)

The following capabilities are supported:

- **RTN\_IsDynamic() API is used to identify dynamically created code.** A routine can be marked as dynamically created using Jit Profiling API only.
- A Pin tool can instrument Jitted routines using **RTN\_AddInstrumentFunction** API See the examples [Managed platforms support](#) for more information.

Following conditions must be satisfied to get the managed platforms support:

- Set **INTEL\_JIT\_PROFILER32** and **INTEL\_JIT\_PROFILER64** environment variables to appropriate pinjitprofiling dynamic library
  - For Windows:

```
set INTEL_JIT_PROFILER32=<The Pin kit full path>\ia32\bin\pinjitprofiling.dll
set INTEL_JIT_PROFILER64=<The Pin kit full path>\intel64\bin\pinjitprofiling.dll
```

- For Linux:

```
setenv INTEL_JIT_PROFILER32 <The Pin kit full path>/ia32/bin/libpinjitprofiling.so
setenv INTEL_JIT_PROFILER64 <The Pin kit full path>/intel64/bin/libpinjitprofiling.so
```

- Add the knob support\_jit\_api to the Pin command line as Pin tool option:

```
<Pin executable> <pin options> -t <Pin tool> -support_jit_api <Other Pin tool options> -- <Test application> <Test application options>
```

## Symbols

**Pin provides access to function names using the symbol object (SYM).** Symbol objects only provide information about the function symbols in the application. Information about other types of symbols (e.g. data symbols), must be obtained independently by the tool.

On Windows, you can use dbghelp.dll for this. Note that using dbghelp.dll in an instrumented process is not safe and can cause dead-locks in some cases. A possible solution is to find symbols using a different non-instrumented process.

On Linux, libelf.so or libdwf.so can be used to access symbol information.

**PIN\_InitSymbols** must be called to access functions by name. See **SYM: Symbol Object** for more information.

## Floating Point Support in Analysis Routines

Pin takes care of maintaining the application's floating point state across analysis routines.

IARG\_REG\_VALUE cannot be used to pass floating point register values as arguments to analysis routines.

## Instrumenting Multi-threaded Applications

Instrumenting a multi-threaded program requires that the tool **be thread safe** - access to global storage must be coordinated with other threads. Pin tries to provide a conventional C++ program environment for tools, but it is not possible to use the standard library interfaces to manage threads in a Pintool. For example, Linux tools cannot use the pthreads library and Windows tools should not use the **Win32 API's to manage threads**. Instead, Pin provides its own locking and thread management API's, which the Pintool should use. (See **LOCK: Locking Primitives** and **Pin Thread API**.)

Pintools do not need to add explicit locking to instrumentation routines because Pin calls these routines while holding an internal lock called the VM lock. However, Pin does execute analysis and replacement functions in parallel, so Pintools may need to add locking to these routines if they access global data.

Pintools on Linux also need to take care when calling standard C or C++ library routines from analysis or replacement functions because the C and C++ libraries linked into Pintools are **not** thread-safe. Some simple C / C++ routines are safe to call without locking, because their implementations are inherently thread-safe, however, Pin does not attempt to provide a list of safe routines. If you are in doubt, you should add locking around calls to library functions. In particular, the "errno" value is not multi-thread safe, so tools that use this should provide their own locking. Note that these restrictions only exist on the Unix platforms, as the library routines on Windows are thread safe.

Pin provides call-backs when each thread starts and ends (see **PIN\_AddThreadStartFunction** and **PIN\_AddThreadFiniFunction**). These provide a convenient place for a Pintool to allocate and manipulate thread local data and store it on a thread's local storage.

Pin also provides an analysis routine argument (IARG\_THREAD\_ID), which passes a Pin-specific thread ID for the calling thread. This ID is different from the O/S system thread ID, and is a small number starting at 0, which can be used as an index to an array of thread data or as the locking value to Pin user locks. See the example **Instrumenting Threaded Applications** for more information.

In addition to the Pin thread ID, the Pin API provides an efficient thread local storage (TLS), with the option to allocate a new TLS key and associate it with a given data destruction function. Any thread of the process can store and retrieve values in its own slot, referenced by the allocated key. The initial value associated with the key in all threads is NULL. See the example **Using TLS** for more information.

False sharing occurs when multiple threads access different parts of the same cache line and at least one of them is a write. To maintain memory coherency, the computer must copy the memory from one CPU's cache to another, even though data is not truly shared. False sharing can usually be avoided by padding critical data structures to the size of a cache line, or by rearranging the data layout of structures. See the example **Using TLS** for more information.

## Avoiding Deadlocks in Multi-threaded Applications

Since Pin, the tool, and the application may each acquire and release locks, Pin tool developers must take care to avoid deadlocks with either the application or Pin. Deadlocks generally occur when two threads acquire the same locks in a different order. For example, thread A acquires lock L1 and then acquires lock L2, while thread B acquires lock L2 and then acquires lock L1. This will lead to a deadlock if thread A holds lock L1 and waits for L2 while thread B holds lock L2 and waits for L1. To avoid such deadlocks, Pin imposes a hierarchy on the order in which locks must be acquired. Pin generally acquires its own internal locks before the tool acquires any lock (e.g. via **PIN\_GetLock()**). Additionally, we assume that the application may acquire locks at the top of this hierarchy (i.e. before Pin acquires its internal locks). The following diagram illustrates the hierarchy:

```
Application locks -> Pin internal locks -> Tool locks
```

Pin tool developers should design their Pin tools such that they never break this lock hierarchy, and they can do so by following these basic guidelines:

- If the tool acquires any locks from within a Pin call-back, it must release those locks before returning from that call-back. Holding a lock across Pin call-backs violates the hierarchy with respect to the Pin internal locks.
- If the tool acquires any locks from within an analysis routine, it must release those locks before returning from the analysis routine. Holding a lock across Pin analysis routines violates the hierarchy with respect to Pin internal locks and other locks used by the instrumented application itself.
- If the tool calls a Pin API from within a Pin call-back or analysis routine, it should not hold any tool locks when calling the API. Some of the Pin APIs use the internal Pin locks so holding a tool lock before invoking these APIs violates the hierarchy with respect to the Pin internal locks.
- If the tool calls a Pin API from within an analysis routine, it may need to acquire the Pin client lock first by calling **PIN\_LockClient()**. This depends on the API, so check the documentation for the specific API for more information. Note that the tool should not hold any other locks when calling **PIN\_LockClient()**, as described in the previous item.

While these guidelines are sufficient in most cases, they may turn out to be too restrictive for certain use-cases. The next set of guidelines explains the conditions in which it is safe to relax the basic guidelines above:

- In JIT mode, the tool may acquire locks from within an analysis routine and not release them, providing it releases these locks before leaving the trace that contains the analysis routine. The tool must expect that the trace may exit "early" if an application instruction raises an exception. Any lock L, which the tool might hold when the application raises an exception, must obey the following sub-rules:
  - The tool must establish a call-back that executes when the application raises an exception and this call-back must release lock L if it was acquired at the time the exception occurred. Tools can use **PIN\_AddContextChangeFunction()** to establish this call-back.
  - The tool must not acquire lock L from within any Pin call-back, to avoid violating the hierarchy with respect to the Pin internal locks.
- If the tool calls a Pin API from an analysis routine, it may acquire and hold a lock L while calling the API providing that:
  - Lock L is not being acquired from any Pin call-back. This avoids the hierarchy violation with respect to the Pin internal locks.
  - The Pin API being invoked does not cause application code to execute (e.g., **PIN\_CallApplicationFunction()**). This avoids the hierarchy violation with

respect to the locks used by the application itself.

## Examples

### Table of Contents

- [Building the Example Tools](#)
- [Notes for Building Tools for Windows](#)
- [Simple Instruction Count \(Instruction Instrumentation\)](#)
- [Instruction Address Trace \(Instruction Instrumentation\)](#)
- [Memory Reference Trace \(Instruction Instrumentation\)](#)
- [Detecting the Loading and Unloading of Images \(Image Instrumentation\)](#)
- [More Efficient Instruction Counting \(Trace Instrumentation\)](#)
- [Procedure Instruction Count \(Routine Instrumentation\)](#)
- [Using PIN\\_SafeCopy\(\)](#)
- [Order of Instrumentation](#)
- [Finding the Value of Function Arguments](#)
- [Finding Functions By Name on Windows](#)
- [Instrumenting Threaded Applications](#)
- [Using TLS](#)
- [Using the Fast Buffering APIs](#)
- [Finding the Static Properties of an Image](#)
- [Detaching Pin from the Application](#)
- [Replacing a Routine in Probe Mode](#)
- [Instrumenting Child Processes](#)
- [Instrumenting Before and After Forks](#)
- [Managed platforms support](#)

To illustrate how to write Pintools, we present some simple examples. In the web based version of the manual, you can click on a function in the Pin API to see its documentation.

All the examples presented in the manual can be found in the source/tools/ManualExamples directory.

### Building the Example Tools

To build all examples in a directory for ia32 architecture:

```
$ cd source/tools/ManualExamples
$ make all TARGET=ia32
```

To build all examples in a directory for intel64 architecture:

```
$ cd source/tools/ManualExamples
$ make all TARGET=intel64
```

To build and run a specific example (e.g., inscount0):

```
$ cd source/tools/ManualExamples
$ make inscount0.test TARGET=intel64
```

To build a specific example without running it (e.g., inscount0):

```
$ cd source/tools/ManualExamples
$ make obj-intel64/inscount0.so TARGET=intel64
```

The above applies to the Intel(R) 64 architecture. For the IA-32 architecture, use TARGET=ia32 instead.

```
$ cd source/tools/ManualExamples
$ make obj-ia32/inscount0.so TARGET=ia32
```

### Notes for Building Tools for Windows

Since the tools are built using make, be sure to install cygwin make first.

Open the Visual Studio Command Prompt corresponding to your target architecture, i.e. x86 or x64, and follow the steps in the [Building the Example Tools](#) section.

### Simple Instruction Count (Instruction Instrumentation)

The example below instruments a program to count the total number of instructions executed. It inserts a call to `docount` before every instruction. When the program exits, it saves the count in the file `inscount.out`.

Here is how to run it and display its output (note that the file list is the `ls` output, so it may be different on your machine, similarly the instruction count will depend on the implementation of `ls`):

```
$ ../../pin -t obj-intel64/inscount0.so -- /bin/ls
Makefile      atrace.o      imageload.out itrace      proccount
Makefile.example  imageload    inscount0      itrace.o    proccount.o
atrace        imageload.o   inscount0.o    itrace.o
$ cat inscount.out
Count 422838
$
```

The KNOB exhibited in the example below overwrites the default name for the output file. To use this feature, add "-o <file\_name>" to the command line. Tool command line options should be inserted between the tool name and the double dash ("--"). For more information on how to add command line options to your tool, please see [KNOB: Commandline Option Handling](#).

```
$ ../../pin -t obj-intel64/inscount0.so -o inscount0.log -- /bin/ls
```

The example can be found in `source/tools/ManualExamples/inscount0.cpp`

```
#include <iostream>
#include <fstream>
```

```

#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */
/*  argc, argv are the entire command line: pin -t <toolname> -- ... */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

### Instruction Address Trace (Instruction Instrumentation)

In the previous example, we did not pass any arguments to `docount`, the analysis procedure. In this example, we show how to pass arguments. When calling an analysis procedure, Pin allows you to pass the **instruction pointer**, current value of registers, **effective address of memory operations**, constants, etc. For a complete list, see **IARG\_TYPE**.

With a small change, we can turn the instruction counting example into a Pintool that prints the address of every instruction that is executed. This tool is useful for understanding the control flow of a program for debugging, or in processor design when simulating an instruction cache.

We change the arguments to `INS_InsertCall` to pass the address of the instruction about to be executed. We replace `docount` with `printip`, which prints the instruction address. It writes its output to the file `itrace.out`.

This is how to run it and look at the output:

```

$ ../../pin -t obj-intel64/itrace.so -- /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 itrace.out
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
0x40001ee7
0x40001ee8
0x40001ee9
0x40001eea
0x40001ef0
0x40001ee0
$

```

The example can be found in `source/tools/ManualExamples/itrace.cpp`

```

#include <stdio.h>
#include "pin.H"

```

```

FILE * trace;

// This function is called before every instruction is executed
// and prints the IP
VOID printip(VOID *ip) { fprintf(trace, "%p\n", ip); }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to printip before every instruction, and pass it the IP
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace, "#eof\n");
    fclose(trace);
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    PIN_ERROR("This Pintool prints the IPs of every instruction executed\n"
              + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    trace = fopen("itrace.out", "w");

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

### Memory Reference Trace (Instruction Instrumentation)

The previous example instruments all instructions. Sometimes a tool may only want to instrument a class of instructions, like memory operations or branch instructions. A tool can do this by using the Pin API which includes functions that classify and examine instructions. The basic API is common to all instruction sets and is described [here](#). In addition, there is an instruction set specific API for the **IA-32 ISA**.

In this example, we show how to do more selective instrumentation by examining the instructions. This tool generates a trace of all memory addresses referenced by a program. This is also useful for debugging and for simulating a data cache in a processor.

We only instrument instructions that read or write memory. We also use **INS\_InsertPredicatedCall** instead of **INS\_InsertCall** to avoid generating references to instructions that are predicated when the predicate is false. On IA-32 and Intel(R) 64 architectures CMOVcc, FCMOVcc and REP prefixed string operations are treated as being predicated. For CMOVcc and FCMOVcc the predicate is the condition test implied by "cc", for REP prefixed string ops it is that the count register is non-zero.

Since the instrumentation functions are only called once and the analysis functions are called every time an instruction is executed, it is much faster to instrument only the memory operations, as compared to the previous instruction trace example that instruments every instruction.

Here is how to run it and the sample output:

```

$ ../../pin -t obj-intel64/pinatrace.so -- /bin/ls
Makefile      atrace.o      imageload.o   inscount0.o   itrace.out
Makefile.example atrace.out   imageload.out itrace         proccount
atrace        imageload    inscount0     itrace.o       proccount.o
$ head pinatrace.out
0x40001ee0: R 0xbffff798
0x40001efd: W 0xbffff7d4
0x40001f09: W 0xbffff7d8
0x40001f20: W 0xbffff864
0x40001f20: W 0xbffff868
0x40001f20: W 0xbffff86c
0x40001f20: W 0xbffff870
0x40001f20: W 0xbffff874
0x40001f20: W 0xbffff878
0x40001f20: W 0xbffff87c
$

```

The example can be found in `source/tools/ManualExamples/pinatrace.cpp`

```

/*
 * This file contains an ISA-portable PIN tool for tracing memory accesses.
 */

#include <stdio.h>
#include "pin.H"

FILE * trace;

```

```

// Print a memory read record
VOID RecordMemRead(VOID * ip, VOID * addr)
{
    fprintf(trace, "%p: R %p\n", ip, addr);
}

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr)
{
    fprintf(trace, "%p: W %p\n", ip, addr);
}

// Is called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v)
{
    // Instruments memory accesses using a predicated call, i.e.
    // the instrumentation is called iff the instruction will actually be executed.
    //
    // On the IA-32 and Intel(R) 64 architectures conditional moves and REP
    // prefixed instructions appear as predicated instructions in Pin.
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Iterate over each memory operand of the instruction.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        if (INS_MemoryOperandIsRead(ins, memOp))
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                IARG_INST_PTR,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }

        // Note that in some architectures a single memory operand can be
        // both read and written (for instance incl (%eax) on IA-32)
        // In that case we instrument it once for read and once for write.
        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                IARG_INST_PTR,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }
    }
}

VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace, "eof\n");
    fclose(trace);
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    PIN_ERROR( "This Pintool prints a trace of memory addresses\n"
               + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char *argv[])
{
    if (PIN_Init(argc, argv)) return Usage();

    trace = fopen("pinatrace.out", "w");

    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

```

### Detecting the Loading and Unloading of Images (Image Instrumentation)

The example below prints a message to a trace file every time an image is loaded or unloaded. It really abuses the image instrumentation mode as the Pintool neither inspects the image nor adds instrumentation code.

If you invoke it on `ls`, you would see this output:

```

$ ../../pin -t obj-intel64/imageload.so -- /bin/ls
Makefile      atrace.o      imageload.o    inscount0.o   proccount
Makefile.example atrace.out    imageload.out  itrace        proccount.o
atrace        imageload    inscount0      itrace.o      trace.out
$ cat imageload.out
Loading /bin/ls
Loading /lib/ld-linux.so.2
Loading /lib/libtermcap.so.2
Loading /lib/i686/libc.so.6
Unloading /bin/ls
Unloading /lib/ld-linux.so.2
Unloading /lib/libtermcap.so.2
Unloading /lib/i686/libc.so.6

```



\$

The example can be found in `source/tools/ManualExamples/imageload.cpp`

```
//
// This tool prints a trace of image load and unload events
//

#include "pin.H"
#include <iostream>
#include <fstream>
#include <stdlib.h>

using namespace std;

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "imageload.out", "specify file name");

ofstream TraceFile;

// Pin calls this function every time a new img is loaded
// It can instrument the image, but this example does not
// Note that imgs (including shared libraries) are loaded lazily
VOID ImageLoad(IMG img, VOID *v)
{
    TraceFile << "Loading " << IMG_Name(img) << ", Image id = " << IMG_Id(img) << endl;
}

// Pin calls this function every time a new img is unloaded
// You can't instrument an image that is about to be unloaded
VOID ImageUnload(IMG img, VOID *v)
{
    TraceFile << "Unloading " << IMG_Name(img) << endl;
}

// This function is called when the application exits
// It closes the output file.
VOID Fini(INT32 code, VOID *v)
{
    if (TraceFile.is_open()) { TraceFile.close(); }
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    PIN_ERROR("This tool prints a log of image load and unload events\n"
        + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    TraceFile.open(KnobOutputFile.Value().c_str());

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Register ImageUnload to be called when an image is unloaded
    IMG_AddUnloadFunction(ImageUnload, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

### More Efficient Instruction Counting (Trace Instrumentation)

The example **Simple Instruction Count (Instruction Instrumentation)** computed the number of executed instructions by inserting a call before every instruction. In this example, we make it more efficient by counting the number of instructions in a BBL at instrumentation time, and incrementing the counter once per BBL, instead of once per instruction.

The example can be found in `source/tools/ManualExamples/inscount1.cpp`

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

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every block
```



```

VOID docount(UINT32 c) { icount += c; }

// Pin calls this function every time a new basic block is encountered
// It inserts a call to docount
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount before every bbl, passing the number of instructions
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount, IARG_UINT32, BBL_NumIns(bbl), IARG_END);
    }
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    TRACE_AddInstrumentFunction(Trace, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

### Procedure Instruction Count (Routine Instrumentation)

The example below instruments a program to count the number of times a procedure is called, and the total number of instructions executed in each procedure. When it finishes, it prints a profile to proccount.out

Executing the tool and sample output:

```

$ ../../..pin -t obj-intel64/proccount.so -- /bin/grep proccount.cpp Makefile
proccount_SOURCES = proccount.cpp
$ head proccount.out

```

Procedure	Image	Address	Calls	Instructions
_fini	libc.so.6	0x40144d00	1	21
__deregister_frame_info	libc.so.6	0x40143f60	2	70
__register_frame_info	libc.so.6	0x40143df0	2	62
fde_merge	libc.so.6	0x40143870	0	8
__init_misc	libc.so.6	0x40115824	1	85
__getclktck	libc.so.6	0x401157f4	0	2
munmap	libc.so.6	0x40112ca0	1	9
mmap	libc.so.6	0x40112bb0	1	23
getpagesize	libc.so.6	0x4010f934	2	26

```

$

```

The example can be found in source/tools/ManualExamples/proccount.cpp

```

//
// This tool counts the number of times a routine is executed and
// the number of instructions executed in a routine
//

#include <fstream>
#include <iomanip>
#include <iostream>
#include <string.h>
#include "pin.H"

ofstream outFile;

// Holds instruction count for a single procedure
typedef struct RtnCount
{
    string _name;
    string _image;
    ADDRINT _address;
}

```

```

    RTN_rtn;
    UINT64 _rtnCount;
    UINT64 _icount;
    struct RtnCount * _next;
} RTN_COUNT;

// Linked list of instruction counts for each routine
RTN_COUNT * RtnList = 0;

// This function is called before every instruction is executed
VOID docount(UINT64 * counter)
{
    (*counter)++;
}

const char * StripPath(const char * path)
{
    const char * file = strrchr(path, '/');
    if (file)
        return file+1;
    else
        return path;
}

// Pin calls this function every time a new rtn is executed
VOID Routine(RTN rtn, VOID *v)
{
    // Allocate a counter for this routine
    RTN_COUNT * rc = new RTN_COUNT;

    // The RTN goes away when the image is unloaded, so save it now
    // because we need it in the fini
    rc->_name = RTN_Name(rtn);
    rc->_image = StripPath(IMG_Name(SEC_Img(RTN_Sec(rtn))).c_str());
    rc->_address = RTN_Address(rtn);
    rc->_icount = 0;
    rc->_rtnCount = 0;

    // Add to list of routines
    rc->_next = RtnList;
    RtnList = rc;

    RTN_Open(rtn);

    // Insert a call at the entry point of a routine to increment the call count
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)docount, IARG_PTR, &(rc->_rtnCount), IARG_END);

    // For each instruction of the routine
    for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins))
    {
        // Insert a call to docount to increment the instruction counter for this rtn
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_PTR, &(rc->_icount), IARG_END);
    }

    RTN_Close(rtn);
}

// This function is called when the application exits
// It prints the name and count for each procedure
VOID Fini(INT32 code, VOID *v)
{
    outFile << setw(23) << "Procedure" << " "
        << setw(15) << "Image" << " "
        << setw(18) << "Address" << " "
        << setw(12) << "Calls" << " "
        << setw(12) << "Instructions" << endl;

    for (RTN_COUNT * rc = RtnList; rc; rc = rc->_next)
    {
        if (rc->_icount > 0)
            outFile << setw(23) << rc->_name << " "
                << setw(15) << rc->_image << " "
                << setw(18) << hex << rc->_address << dec << " "
                << setw(12) << rc->_rtnCount << " "
                << setw(12) << rc->_icount << endl;
    }
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This Pintool counts the number of times a routine is executed" << endl;
    cerr << "and the number of instructions executed in a routine" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize symbol table code, needed for rtn instrumentation
    PIN_InitSymbols();

    outFile.open("proccount.out");

```

```

// Initialize pin
if (PIN_Init(argc, argv)) return Usage();

// Register Routine to be called to instrument rtn
RTN_AddInstrumentFunction(Routine, 0);

// Register Fini to be called when the application exits
PIN_AddFiniFunction(Fini, 0);

// Start the program, never returns
PIN_StartProgram();

return 0;
}

```

### Using PIN\_SafeCopy()

PIN\_SafeCopy is used to copy the specified number of bytes from a source memory region to a destination memory region. This function guarantees safe return to the caller even if the source or destination regions are inaccessible (entirely or partially).

Use of this function also guarantees that the tool reads or writes the values used by the application. For example, on Windows, Pin replaces certain TEB fields when running a tool's analysis code. If the tool accessed these fields directly, it would see the modified values rather than the original ones. Using [PIN\\_SafeCopy\(\)](#) allows the tool to read or write the application's values for these fields.

We recommend using this API any time a tool reads or writes application memory.

```

$ ../../../../pin -t obj-ia32/safecopy.so -- /bin/cp makefile obj-ia32/safecopy.so.makefile.copy
$ head safecopy.out
Emulate loading from addr 0xbff0057c to ebx
Emulate loading from addr 0x64ffd4 to eax
Emulate loading from addr 0xbff00598 to esi
Emulate loading from addr 0x6501c8 to edi
Emulate loading from addr 0x64ff14 to edx
Emulate loading from addr 0x64ff1c to edx
Emulate loading from addr 0x64ff24 to edx
Emulate loading from addr 0x64ff2c to edx
Emulate loading from addr 0x64ff34 to edx
Emulate loading from addr 0x64ff3c to edx

```

The example can be found in source/tools/ManualExamples/safecopy.cpp.

```

#include <stdio.h>
#include "pin.H"
#include "pin_isa.H"
#include <iostream>
#include <fstream>

std::ofstream out = 0;

//=====
// Analysis routines
//=====

// Move from memory to register
ADDRINT DoLoad(REG reg, ADDRINT * addr)
{
    *out << "Emulate loading from addr " << addr << " to " << REG_StringShort(reg) << endl;
    ADDRINT value;
    PIN_SafeCopy(&value, addr, sizeof(ADDRINT));
    return value;
}

//=====
// Instrumentation routines
//=====

VOID EmulateLoad(INS ins, VOID* v)
{
    // Find the instructions that move a value from memory to a register
    if (INS_Opcode(ins) == XED_ICLASS_MOV &&
        INS_IsMemoryRead(ins) &&
        INS_OperandIsReg(ins, 0) &&
        INS_OperandIsMemory(ins, 1))
    {
        // op0 <- *op1
        INS_InsertCall(ins,
            IPOINT_BEFORE,
            AFUNPTR(DoLoad),
            IARG_UINT32,
            REG(INS_OperandReg(ins, 0)),
            IARG_MEMORYREAD_EA,
            IARG_RETURN_REGS,
            INS_OperandReg(ins, 0),
            IARG_END);

        // Delete the instruction
        INS_Delete(ins);
    }
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool demonstrates the use of SafeCopy" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */

```

```

/* Main
/* ===== */

int main(int argc, char * argv[])
{
    // Write to a file since cout and cerr maybe closed by the application
    out = new std::ofstream("safecopy.out");

    // Initialize pin & symbol manager
    if (PIN_Init(argc, argv)) return Usage();
    PIN_InitSymbols();

    // Register EmulateLoad to be called to instrument instructions
    INS_AddInstrumentFunction(EmulateLoad, 0);

    // Never returns
    PIN_StartProgram();
    return 0;
}

```

### Order of Instrumentation

Pin provides tools with multiple ways to control the execution order of analysis calls. The execution order depends mainly on the insertion action (**IPOINT**) and call order (**CALL\_ORDER**). The example below illustrates this behavior by instrumenting all return instructions in three different ways. Additional examples can be found in `source/tools/InstrumentationOrderAndVersion`.

```

$ ../../..pin -t obj-ia32/invoke.so -- obj-ia32/little_malloc
$ head invocation.out
After: IP = 0x64bc5e
Before: IP = 0x64bc5e
Taken: IP = 0x63a12e
After: IP = 0x64bc5e
Before: IP = 0x64bc5e
Taken: IP = 0x641c76
After: IP = 0x641ca6
After: IP = 0x64bc5e
Before: IP = 0x64bc5e
Taken: IP = 0x648b02

```

The example can be found in `source/tools/ManualExamples/invoke.cpp`.

```

#include "pin.H"
#include <iostream>
#include <fstream>
using namespace std;

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "invocation.out", "specify output file name");

ofstream OutFile;

/*
 * Analysis routines
 */
VOID Taken( const CONTEXT * ctxt)
{
    ADDRINT TakenIP = (ADDRINT)PIN_GetContextReg( ctxt, REG_INST_PTR );
    OutFile << "Taken: IP = " << hex << TakenIP << dec << endl;
}

VOID Before(CONTEXT * ctxt)
{
    ADDRINT BeforeIP = (ADDRINT)PIN_GetContextReg( ctxt, REG_INST_PTR );
    OutFile << "Before: IP = " << hex << BeforeIP << dec << endl;
}

VOID After(CONTEXT * ctxt)
{
    ADDRINT AfterIP = (ADDRINT)PIN_GetContextReg( ctxt, REG_INST_PTR );
    OutFile << "After: IP = " << hex << AfterIP << dec << endl;
}

/*
 * Instrumentation routines
 */
VOID ImageLoad(IMG img, VOID *v)
{
    for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
    {
        // RTN_InsertCall() and INS_InsertCall() are executed in order of
        // appearance. In the code sequence below, the IPOINT_AFTER is
        // executed before the IPOINT_BEFORE.
        for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn))
        {
            // Open the RTN.
            RTN_Open( rtn );

            // IPOINT_AFTER is implemented by instrumenting each return
            // instruction in a routine. Pin tries to find all return
            // instructions, but success is not guaranteed.
            RTN_InsertCall( rtn, IPOINT_AFTER, (AFUNPTR)After,
                IARG_CONTEXT, IARG_END);

            // Examine each instruction in the routine.
            for( INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins) )
            {
                if( INS_IsRet( ins ) )
                {
                    // instrument each return instruction.
                    // IPOINT_TAKEN_BRANCH always occurs last.

```

```

        INS_InsertCall( ins, IPOINT_BEFORE, (AFUNPTR)Before,
                        IARG_CONTEXT, IARG_END);
        INS_InsertCall( ins, IPOINT_TAKEN_BRANCH, (AFUNPTR)Taken,
                        IARG_CONTEXT, IARG_END);
    }
}
// Close the RTN.
RTN_Close( rtn );
}
}
}

VOID Fini(INT32 code, VOID *v)
{
    OutFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This is the invocation pintool" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize pin & symbol manager
    if (PIN_Init(argc, argv)) return Usage();
    PIN_InitSymbols();

    // Register ImageLoad to be called to instrument instructions
    IMG_AddInstrumentFunction(ImageLoad, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Write to a file since cout and cerr maybe closed by the application
    OutFile.open(KnobOutputFile.Value().c_str());
    OutFile.setf(ios::showbase);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
/* ===== */

```

### Finding the Value of Function Arguments

Often one needs to know the value of the argument passed into a function, or the return value. You can use Pin to find this information. Using the **RTN\_InsertCall()** function, you can specify the arguments of interest.

The example below prints the input argument for malloc() and free(), and the return value from malloc().

```

$ ../../pin -t obj-ia32/malloctrace.so -- /bin/cp makefile obj-ia32/malloctrace.so.makefile.copy
$ head malloctrace.out
malloc(0x24d)
  returns 0x6504f8
malloc(0x57)
  returns 0x650748
malloc(0xc)
  returns 0x6507a0
malloc(0x3c0)
  returns 0x6507b0
malloc(0xc)
  returns 0x650b70

```

The example can be found in source/tools/ManualExamples/malloctrace.cpp.

```

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

/* ===== */
/* Names of malloc and free */
/* ===== */
#if defined(TARGET_MAC)
#define MALLOC "_malloc"
#define FREE "_free"
#else
#define MALLOC "malloc"
#define FREE "free"
#endif

/* ===== */
/* Global Variables */
/* ===== */

std::ofstream TraceFile;

/* ===== */
/* Commandline Switches */
/* ===== */

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "malloctrace.out", "specify trace file name");

```

```

/* ===== */

/* ===== */
/* Analysis routines */
/* ===== */

VOID Arg1Before(CHAR * name, ADDRINT size)
{
    TraceFile << name << "(" << size << ")" << endl;
}

VOID MallocAfter(ADDRINT ret)
{
    TraceFile << " returns " << ret << endl;
}

/* ===== */
/* Instrumentation routines */
/* ===== */

VOID Image(IMG img, VOID *v)
{
    // Instrument the malloc() and free() functions. Print the input argument
    // of each malloc() or free(), and the return value of malloc().
    //
    // Find the malloc() function.
    RTN mallocRtn = RTN_FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);

        // Instrument malloc() to print the input argument value and the return value.
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                      IARG_ADDRINT, MALLOC,
                      IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                      IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                      IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);

        RTN_Close(mallocRtn);
    }

    // Find the free() function.
    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        // Instrument free() to print the input argument value.
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                      IARG_ADDRINT, FREE,
                      IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                      IARG_END);
        RTN_Close(freeRtn);
    }
}

/* ===== */

VOID Fini(INT32 code, VOID *v)
{
    TraceFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool produces a trace of calls to malloc." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char *argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    if (PIN_Init(argc, argv) )
    {
        return Usage();
    }

    // Write to a file since cout and cerr maybe closed by the application
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);

    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

```

```

/* ===== */
/* eof */
/* ===== */

```

## Finding Functions By Name on Windows

Finding functions by name on Windows requires a different methodology. Several symbols could resolve to the same function address. It is important to check all symbol names.

The following example finds the function name in the symbol table, and uses the symbol address to find the appropriate RTN.

```

$ ..\..\..\pin -t obj-ia32\w_malloctrace.dll -- ..\Tests\obj-ia32\cp-pin.exe makefile w_malloctrace.makefile.copy
$ head *.out
Before: RtlAllocateHeap(00150000, 0, 0x94)
After: RtlAllocateHeap returns 0x153440
After: RtlAllocateHeap returns 0x153440
Before: RtlAllocateHeap(00150000, 0, 0x20)
After: RtlAllocateHeap returns 0
After: RtlAllocateHeap returns 0x1567c0
Before: RtlAllocateHeap(019E0000, 0x8, 0x1800)
After: RtlAllocateHeap returns 0x19e0688
Before: RtlAllocateHeap(00150000, 0, 0x1a)thread begin 0

After: RtlAllocateHeap returns 0

```

The example can be found in source/tools/ManualExamples/w\_malloctrace.cpp.

```

/* ===== */
/* This example demonstrates finding a function by name on Windows. */
/* ===== */

#include "pin.H"
namespace WINDOWS
{
#include<Windows.h>
}
#include <iostream>
#include <fstream>

/* ===== */
/* Global Variables */
/* ===== */

std::ofstream TraceFile;

/* ===== */
/* Commandline Switches */
/* ===== */

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "w_malloctrace.out", "specify trace file name");

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool produces a trace of calls to RtlAllocateHeap.";
    cerr << endl << endl;
    cerr << KNOB_BASE::StringKnobSummary();
    cerr << endl;
    return -1;
}

/* ===== */
/* Analysis routines */
/* ===== */

VOID Before(CHAR * name, WINDOWS::HANDLE hHeap,
    WINDOWS::DWORD dwFlags, WINDOWS::DWORD dwBytes)
{
    TraceFile << "Before: " << name << "(" << hex << hHeap << ", "
        << dwFlags << ", " << dwBytes << ")" << dec << endl;
}

VOID After(CHAR * name, ADDRINT ret)
{
    TraceFile << "After: " << name << " returns " << hex
        << ret << dec << endl;
}

/* ===== */
/* Instrumentation routines */
/* ===== */

VOID Image(IMG img, VOID *v)
{
    // Walk through the symbols in the symbol table.
    //
    for (SYM sym = IMG_RegsymHead(img); SYM_Valid(sym); sym = SYM_Next(sym))
    {
        string undFuncName = PIN_UndecorateSymbolName(SYM_Name(sym), UNDECORATION_NAME_ONLY);

        // Find the RtlAllocHeap() function.
        if (undFuncName == "RtlAllocateHeap")
        {
            RTN allocRtn = RTN_FindByAddress(IMG_LowAddress(img) + SYM_Value(sym));

            if (RTN_Valid(allocRtn))
            {

```



```

// Instrument to print the input argument value and the return value.
RTN_Open(allocRtn);

RTN_InsertCall(allocRtn, IPOINT_BEFORE, (AFUNPTR)Before,
               IARG_ADDRINT, "RtlAllocateHeap",
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 1,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 2,
               IARG_END);
RTN_InsertCall(allocRtn, IPOINT_AFTER, (AFUNPTR)After,
               IARG_ADDRINT, "RtlAllocateHeap",
               IARG_FUNCARG_ENTRYPOINT_VALUE,
               IARG_FUNCARG_ENTRYPOINT_VALUE,
               IARG_END);

RTN_Close(allocRtn);
    }
}
}

/* ===== */
VOID Fini(INT32 code, VOID *v)
{
    TraceFile.close();
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char *argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }

    // Write to a file since cout and cerr maybe closed by the application
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);

    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

/* ===== */
/* eof */
/* ===== */

```

## Instrumenting Threaded Applications

The following example demonstrates using the ThreadStart() and ThreadFini() notification callbacks. Although ThreadStart() and ThreadFini() are executed under the VM and client locks, they could still contend with resources that are shared by other analysis routines. Using [PIN\\_GetLock\(\)](#) prevents this.

Note that there is known isolation issue when using Pin on Windows. On Windows, a deadlock can occur if a tool opens a file in a callback when run on a multi-threaded application. To work around this problem, open one file in main, and tag the data with the thread ID. See [source/tools/ManualExamples/buffer\\_windows.cpp](#) as an example. This problem does not exist on Linux.

```

$ ../../../../pin -t obj-ia32/malloc_mt.so -- obj-ia32/thread_lin
$ head malloc_mt.out
thread begin 0
thread 0 entered malloc(24d)
thread 0 entered malloc(57)
thread 0 entered malloc(c)
thread 0 entered malloc(3c0)
thread 0 entered malloc(c)
thread 0 entered malloc(58)
thread 0 entered malloc(56)
thread 0 entered malloc(19)
thread 0 entered malloc(25c)

```

The example can be found in [source/tools/ManualExamples/malloc\\_mt.cpp](#)

```

#include <stdio.h>
#include "pin.H"

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "malloc_mt.out", "specify output file name");

//=====
// Analysis Routines
//=====
// Note: threadid+1 is used as an argument to the PIN_GetLock()
// routine as a debugging aid. This is the value that
// the lock is set to, so it must be non-zero.

// lock serializes access to the output file.
FILE * out;
PIN_LOCK lock;

// Note that opening a file in a callback is only supported on Linux systems.
// See buffer-win.cpp for how to work around this issue on Windows.

```

```

//
// This routine is executed every time a thread is created.
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    PIN_GetLock(&lock, threadid+1);
    fprintf(out, "thread begin %d\n", threadid);
    fflush(out);
    PIN_ReleaseLock(&lock);
}

// This routine is executed every time a thread is destroyed.
VOID ThreadFini(THREADID threadid, const CONTEXT *ctxt, INT32 code, VOID *v)
{
    PIN_GetLock(&lock, threadid+1);
    fprintf(out, "thread end %d code %d\n", threadid, code);
    fflush(out);
    PIN_ReleaseLock(&lock);
}

// This routine is executed each time malloc is called.
VOID BeforeMalloc( int size, THREADID threadid )
{
    PIN_GetLock(&lock, threadid+1);
    fprintf(out, "thread %d entered malloc(%d)\n", threadid, size);
    fflush(out);
    PIN_ReleaseLock(&lock);
}

//=====
// Instrumentation Routines
//=====

// This routine is executed for each image.
VOID ImageLoad(IMG img, VOID *)
{
    RTN rtn = RTN_FindByName(img, "malloc");

    if ( RTN_Valid( rtn ))
    {
        RTN_Open(rtn);

        RTN_InsertCall(rtn, IPOINT_BEFORE, AFUNPTR(BeforeMalloc),
                      IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                      IARG_THREAD_ID, IARG_END);

        RTN_Close(rtn);
    }
}

// This routine is executed once at the end.
VOID Fini(INT32 code, VOID *v)
{
    fclose(out);
}

/* ===== */
/* Print Help Message                                */
/* ===== */

INT32 Usage()
{
    PIN_ERROR("This Pintool prints a trace of malloc calls in the guest application\n"
              + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===== */
/* Main                                              */
/* ===== */

int main(INT32 argc, CHAR **argv)
{
    // Initialize the pin lock
    PIN_InitLock(&lock);

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();
    PIN_InitSymbols();

    out = fopen(KnobOutputFile.Value().c_str(), "w");

    // Register ImageLoad to be called when each image is loaded.
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Register Analysis routines to be called when a thread begins/ends
    PIN_AddThreadStartFunction(ThreadStart, 0);
    PIN_AddThreadFiniFunction(ThreadFini, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

```

## Using TLS

Pin provides efficient thread local storage (TLS) APIs. These APIs allow a tool to create thread-specific data. The example below demonstrates how to use these APIs.

```
$ ../../pin -t obj-ia32/inscount_tls.so -- obj-ia32/thread_lin
$ head
Count[0]= 237993
Count[1]= 213296
Count[2]= 209223
Count[3]= 209223
Count[4]= 209223
Count[5]= 209223
Count[6]= 209223
Count[7]= 209223
Count[8]= 209223
Count[9]= 209223
```

The example can be found in `source/tools/ManualExamples/inscount_tls.cpp`

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

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount_tls.out", "specify output file name");

PIN_LOCK lock;
INT32 numThreads = 0;
ofstream OutFile;

// Force each thread's data to be in its own data cache line so that
// multiple threads do not contend for the same data cache line.
// This avoids the false sharing problem.
#define PADSIZ 56 // 64 byte line size: 64-8

// a running count of the instructions
class thread_data_t
{
public:
    thread_data_t() : _count(0) {}
    UINT64 _count;
    UINT8 _pad[PADSIZ];
};

// key for accessing TLS storage in the threads. initialized once in main()
static TLS_KEY tls_key;

// function to access thread-specific data
thread_data_t* get_tls(THREADID threadid)
{
    thread_data_t* tdata =
        static_cast<thread_data_t*>(PIN_GetThreadData(tls_key, threadid));
    return tdata;
}

// This function is called before every block
VOID PIN_FAST_ANALYSIS_CALL docount(UINT32 c, THREADID threadid)
{
    thread_data_t* tdata = get_tls(threadid);
    tdata->_count += c;
}

VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    PIN_GetLock(&lock, threadid+1);
    numThreads++;
    PIN_ReleaseLock(&lock);

    thread_data_t* tdata = new thread_data_t;

    PIN_SetThreadData(tls_key, tdata, threadid);
}

// Pin calls this function every time a new basic block is encountered.
// It inserts a call to docount.
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount for every bbl, passing the number of instructions.

        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)docount, IARG_FAST_ANALYSIS_CALL,
            IARG_UINT32, BBL_NumIns(bbl), IARG_THREAD_ID, IARG_END);
    }
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile << "Total number of threads = " << numThreads << endl;

    for (INT32 t=0; t<numThreads; t++)
    {
        thread_data_t* tdata = get_tls(t);
        OutFile << "Count[" << decstr(t) << "] = " << tdata->_count << endl;
    }

    OutFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */
```

```

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main                                           */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize pin
    PIN_InitSymbols();
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Initialize the lock
    PIN_InitLock(&lock);

    // Obtain a key for TLS storage.
    tls_key = PIN_CreateThreadDataKey(0);

    // Register ThreadStart to be called when a thread starts.
    PIN_AddThreadStartFunction(ThreadStart, 0);

    // Register Instruction to be called to instrument instructions.
    TRACE_AddInstrumentFunction(Trace, 0);

    // Register Fini to be called when the application exits.
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

### Using the Fast Buffering APIs

Pin provides support for buffering data for processing. If all that your analysis callback does is to store its arguments into a buffer, then you should be able to use the buffering API instead, with some performance benefit. **PIN\_DefineTraceBuffer()** defines the buffer that will be used. The buffer is allocated by each thread when it starts up, and deallocated when the thread exits. **INS\_InsertFillBuffer()** writes the requested data directly to the given buffer. The callback delineated in the **PIN\_DefineTraceBuffer()** call is used to process the buffer when the buffer is nearly full, and when the thread exits. Pin does **not** serialize the calls to this callback, so it is the tool writers responsibility to make sure this function is thread safe. This example records the PC of all instructions that access memory, and the effective address accessed by the instruction. Note that IARG\_REG\_REFERENCE, IARG\_REG\_CONST\_REFERENCE and IARG\_CONTEXT can NOT be used in the Fast Buffering APIs

```

$ ../.././pin -t obj-ia32/buffer_linux.so -- obj-ia32/thread_lin
$ tail buffer.out.*
3263df 330108
3263df 330108
3263f1 a92f43fc
3263f7 a92f4d7d
326404 a92f43fc
32640a a92f4bf8
32640a a92f4bf8
32640f a92f4d94
32641b a92f43fc
326421 a92f4bf8

```

The example can be found in source/tools/ManualExamples/buffer\_linux.cpp. This example is appropriate for Linux tools. If you are writing a tool for Windows, please see source/tools/ManualExamples/buffer\_windows.cpp

```

/*
 * Sample buffering tool
 *
 * This tool collects an address trace of instructions that access memory
 * by filling a buffer. When the buffer overflows, the callback writes all
 * of the collected records to a file.
 *
 */

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stddef.h>

#include "pin.H"
#include "portability.H"
using namespace std;

/*
 * Name of the output file
 */
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "buffer.out", "output file");

/*
 * The ID of the buffer
 */
BUFFER_ID bufId;

/*
 * Thread specific data
 */
TLS_KEY mlog_key;

/*
 * Number of OS pages for the buffer
 */

```

```

*/
#define NUM_BUF_PAGES 1024

/*
 * Record of memory references. Rather than having two separate
 * buffers for reads and writes, we just use one struct that includes a
 * flag for type.
 */
struct MEMREF
{
    ADDRINT    pc;
    ADDRINT    ea;
    UINT32     size;
    BOOL       read;
};

/*
 * MLOG - thread specific data that is not handled by the buffering API.
 */
class MLOG
{
public:
    MLOG(THREADID tid);
    ~MLOG();

    VOID DumpBufferToFile( struct MEMREF * reference, UINT64 numElements, THREADID tid );

private:
    ofstream _ofile;
};

MLOG::MLOG(THREADID tid)
{
    string filename = KnobOutputFile.Value() + "." + decstr(getpid_portable()) + "." + decstr(tid);
    _ofile.open(filename.c_str());

    if ( ! _ofile )
    {
        cerr << "Error: could not open output file." << endl;
        exit(1);
    }

    _ofile << hex;
}

MLOG::~~MLOG()
{
    _ofile.close();
}

VOID MLOG::DumpBufferToFile( struct MEMREF * reference, UINT64 numElements, THREADID tid )
{
    for(UINT64 i=0; i<numElements; i++, reference++)
    {
        if (reference->ea != 0)
            _ofile << reference->pc << " " << reference->ea << endl;
    }
}

/*****
 *
 * Instrumentation routines
 *
 *****/

/*
 * Insert code to write data to a thread-specific buffer for instructions
 * that access memory.
 */
VOID Trace(TRACE trace, VOID *v)
{
    for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl=BBL_Next(bbl))
    {
        for(INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins=INS_Next(ins))
        {
            if (!INS_IsStandardMemop(ins) && !INS_HasMemoryVector(ins))
            {
                // We don't know how to treat these instructions
                continue;
            }

            UINT32 memoryOperands = INS_MemoryOperandCount(ins);

            for (UINT32 memOp = 0; memOp < memoryOperands; memOp++)
            {
                UINT32 refSize = INS_MemoryOperandSize(ins, memOp);

                // Note that if the operand is both read and written we log it once
                // for each.
                if (INS_MemoryOperandIsRead(ins, memOp))
                {
                    INS_InsertFillBuffer(ins, IPOINT_BEFORE, bufId,
                                           IARG_INST_PTR, offsetof(struct MEMREF, pc),
                                           IARG_MEMORYOP_EA, memOp, offsetof(struct MEMREF, ea),
                                           IARG_UINT32, refSize, offsetof(struct MEMREF, size),
                                           IARG_BOOL, TRUE, offsetof(struct MEMREF, read),

```

```

        IARG_END);
    }

    if (INS_MemoryOperandIsWritten(ins, memOp))
    {
        INS_InsertFillBuffer(ins, IPOINT_BEFORE, bufId,
                               IARG_INST_PTR, offsetof(struct MEMREF, pc),
                               IARG_MEMORYOP_EA, memOp, offsetof(struct MEMREF, ea),
                               IARG_UINT32, refSize, offsetof(struct MEMREF, size),
                               IARG_BOOL, FALSE, offsetof(struct MEMREF, read),
                               IARG_END);
    }
}
}
}

/*****
 *
 * Callback Routines
 *
 *****/

VOID * BufferFull(BUFFER_ID id, THREADID tid, const CONTEXT *ctxt, VOID *buf,
                  UINT64 numElements, VOID *v)
{
    struct MEMREF * reference=(struct MEMREF*)buf;

    MLOG * mlog = static_cast<MLOG*>( PIN_GetThreadData( mlog_key, tid ) );

    mlog->DumpBufferToFile( reference, numElements, tid );

    return buf;
}

/*
 * Note that opening a file in a callback is only supported on Linux systems.
 * See buffer-win.cpp for how to work around this issue on Windows.
 */
VOID ThreadStart(THREADID tid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    // There is a new MLOG for every thread. Opens the output file.
    MLOG * mlog = new MLOG(tid);

    // A thread will need to look up its MLOG, so save pointer in TLS
    PIN_SetThreadData(mlog_key, mlog, tid);
}

VOID ThreadFini(THREADID tid, const CONTEXT *ctxt, INT32 code, VOID *v)
{
    MLOG * mlog = static_cast<MLOG*>(PIN_GetThreadData(mlog_key, tid));

    delete mlog;

    PIN_SetThreadData(mlog_key, 0, tid);
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool demonstrates the basic use of the buffering API." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */
int main(int argc, char *argv[])
{
    // Initialize PIN library. Print help message if -h(elp) is specified
    // in the command line or the command line is invalid
    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }

    // Initialize the memory reference buffer;
    // set up the callback to process the buffer.
    //
    bufId = PIN_DefineTraceBuffer(sizeof(struct MEMREF), NUM_BUF_PAGES,
                                  BufferFull, 0);

    if(bufId == BUFFER_ID_INVALID)
    {
        cerr << "Error: could not allocate initial buffer" << endl;
        return 1;
    }

    // Initialize thread-specific data not handled by buffering api.
    mlog_key = PIN_CreateThreadDataKey(0);

    // add an instrumentation function
    TRACE_AddInstrumentFunction(Trace, 0);
}

```

```
// add callbacks
PIN_AddThreadStartFunction(ThreadStart, 0);
PIN_AddThreadFiniFunction(ThreadFini, 0);

// Start the program, never returns
PIN_StartProgram();

return 0;
}
```

### Finding the Static Properties of an Image

It is also possible to use pin to examine binaries without instrumenting them. This is useful when you need to know static properties of an image. The sample tool below counts the number of instructions in an image, but does not insert any instrumentation.

The example can be found in `source/tools/ManualExamples/staticcount.cpp`

```
//
// This tool prints a trace of image load and unload events
//

#include <stdio.h>
#include <iostream>
#include "pin.H"

// Pin calls this function every time a new img is loaded
// It can instrument the image, but this example merely
// counts the number of static instructions in the image

VOID ImageLoad(IMG img, VOID *v)
{
    UINT32 count = 0;

    for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
    {
        for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn))
        {
            // Prepare for processing of RTN, an RTN is not broken up into BBLs,
            // it is merely a sequence of INSS
            RTN_Open(rtn);

            for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins))
            {
                count++;
            }

            // to preserve space, release data associated with RTN after we have processed it
            RTN_Close(rtn);
        }
    }
    fprintf(stderr, "Image %s has %d instructions\n", IMG_Name(img).c_str(), count);
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool prints a log of image load and unload events" << endl;
    cerr << " along with static instruction counts for each image." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    // prepare for image instrumentation mode
    PIN_InitSymbols();

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

### Detaching Pin from the Application

Pin can relinquish control of application any time when invoked via **PIN\_Detach**. Control is returned to the original uninstrumented code and the application runs at native speed. Thereafter no instrumented code is ever executed.

The example can be found in `source/tools/ManualExamples/detach.cpp`

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

// This tool shows how to detach Pin from an
// application that is under Pin's control.
```



```

UINT64 icount = 0;

#define N 10000
VOID docount()
{
    icount++;

    // Release control of application if 10000
    // instructions have been executed
    if ((icount % N) == 0)
    {
        PIN_Detach();
    }
}

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

VOID ByeWorld(VOID *v)
{
    std::cerr << endl << "Detached at icount = " << N << endl;
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool demonstrates how to detach Pin from an " << endl;
    cerr << "application that is under Pin's control" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main */
/* ===== */

int main(int argc, char * argv[])
{
    if (PIN_Init(argc, argv)) return Usage();

    // Callback function to invoke for every
    // execution of an instruction
    INS_AddInstrumentFunction(Instruction, 0);

    // Callback functions to invoke before
    // Pin releases control of the application
    PIN_AddDetachFunction(ByeWorld, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

```

## Replacing a Routine in Probe Mode

Probe mode is a method of using Pin to insert probes at the start of specified routines. A probe is a jump instruction that is placed at the start of the specified routine. The probe redirects the flow of control to the replacement function. Before the probe is inserted, the first few instructions of the specified routine are relocated. It is not uncommon for the replacement function to call the replaced routine. Pin provides the relocated address to facilitate this. See the example below.

In probe mode, the application and the replacement routine are run natively. This improves performance, but it puts more responsibility on the tool writer. Probes can only be placed on RTN boundaries.

Many of the PIN APIs that are available in JIT mode are not applicable in Probe mode. In particular, the Pin thread APIs are not supported in Probe mode, because Pin has no information about the threads when the application is run natively. For more information, check the RTN API documentation.

The tool writer must guarantee that there is no jump target where the probe is placed. A probe may be up to 14 bytes long.

Also, it is the tool writer's responsibility to ensure that no thread is currently executing the code where a probe is inserted. Tool writers are encouraged to insert probes when an image is loaded to avoid this problem. Pin will automatically remove the probes when an image is unloaded.

When using probes, Pin must be started with the [PIN\\_StartProgramProbed\(\)](#) API.

Additional libraries must be installed to use Probe mode. See [Libraries for Linux](#) for further information.

The example can be found in `source/tools/ManualExamples/replacesigprobed.cpp`. To build this test, execute:

```
$ make replacesigprobed.test
```

```

// Replace an original function with a custom function defined in the tool using
// probes. The replacement function has a different signature from that of the
// original replaced function.

#include "pin.H"
#include <iostream>
using namespace std;

typedef VOID * ( *FP_MALLOC) ( size_t );

// This is the replacement routine.
//
VOID * NewMalloc( FP_MALLOC orgFuncptr, UINT32 arg0, ADDRINT returnIp )
{
    // Normally one would do something more interesting with this data.
    //
    cout << "NewMalloc ("

```

```

    << hex << ADDRINT ( orgFuncptr ) << ", "
    << dec << arg0 << ", "
    << hex << returnIp << ")"
    << endl << flush;

// Call the relocated entry point of the original (replaced) routine.
//
VOID * v = orgFuncptr( arg0 );

return v;
}

// Pin calls this function every time a new img is loaded.
// It is best to do probe replacement when the image is loaded,
// because only one thread knows about the image at this time.
//
VOID ImageLoad( IMG img, VOID *v )
{
    // See if malloc() is present in the image. If so, replace it.
    //
    RTN rtn = RTN_FindByName( img, "malloc" );

    if (RTN_Valid(rtn))
    {
        cout << "Replacing malloc in " << IMG_Name(img) << endl;

        // Define a function prototype that describes the application routine
        // that will be replaced.
        //
        PROTO proto_malloc = PROTO_Allocate( PIN_PARG(void *), CALLINGSTD_DEFAULT,
                                             "malloc", PIN_PARG(int), PIN_PARG_END() );

        // Replace the application routine with the replacement function.
        // Additional arguments have been added to the replacement routine.
        //
        RTN_ReplaceSignatureProbed(rtn, AFUNPTR(NewMalloc),
                                   IARG_PROTOTYPE, proto_malloc,
                                   IARG_ORIG_FUNCPTR,
                                   IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                   IARG_RETURN_IP,
                                   IARG_END);

        // Free the function prototype.
        //
        PROTO_Free( proto_malloc );
    }
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool demonstrates how to replace an original" << endl;
    cerr << " function with a custom function defined in the tool " << endl;
    cerr << " using probes. The replacement function has a different " << endl;
    cerr << " signature from that of the original replaced function." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main: Initialize and start Pin in Probe mode. */
/* ===== */

int main( INT32 argc, CHAR *argv[] )
{
    // Initialize symbol processing
    //
    PIN_InitSymbols();

    // Initialize pin
    //
    if (PIN_Init(argc, argv)) return Usage();

    // Register ImageLoad to be called when an image is loaded
    //
    IMG_AddInstrumentFunction( ImageLoad, 0 );

    // Start the program in probe mode, never returns
    //
    PIN_StartProgramProbed();

    return 0;
}

```

## Instrumenting Child Processes

The **PIN\_AddFollowChildProcessFunction()** allows you to define the function you will like to execute before an execv'd process starts. Use the `-follow_execv` option on the command line to instrument the child processes, like this:

```
$ ../../pin -follow_execv -t obj-intel64/follow_child_tool.so -- obj-intel64/follow_child_app1 obj-intel64/follow_child_app2
```

The example can be found in `source/tools/ManualExamples/follow_child_tool.cpp`. To build this test, execute:

```
$ make follow_child_tool.test
```

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

```
#include <stdio.h>
#include <unistd.h>

/* ===== */
/* Command line Switches */
/* ===== */

BOOL FollowChild(CHILD_PROCESS childProcess, VOID * userData)
{
    fprintf(stdout, "before child:%u\n", getpid());
    return TRUE;
}

/* ===== */

int main(INT32 argc, CHAR **argv)
{
    PIN_Init(argc, argv);

    PIN_AddFollowChildProcessFunction(FollowChild, 0);

    PIN_StartProgram();

    return 0;
}
```

### Instrumenting Before and After Forks

Pin allows Pintools to register for notification callbacks around forks. The **PIN\_AddForkFunction()** and **PIN\_AddForkFunctionProbed()** callbacks allow you to define the function you want to execute at one of these FPOINTS:

FPOINT_BEFORE	Call-back in parent, just before fork.
FPOINT_AFTER_IN_PARENT	Call-back in parent, immediately after fork.
FPOINT_AFTER_IN_CHILD	Call-back in child, immediately after fork.

Note that **PIN\_AddForkFunction()** is used for JIT mode and **PIN\_AddForkFunctionProbed()** is used for Probed mode. If the fork() fails, the FPOINT\_AFTER\_IN\_PARENT callback, if it is defined, will execute anyway.

The example can be found in source/tools/ManualExamples/fork\_jit\_tool.cpp. To build this test, execute:

```
$ make fork_jit_tool.test
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

#include "pin.H"

#include <iostream>
#include <fstream>

using namespace std;

INT32 Usage()
{
    cerr <<
        "This pin tool registers callbacks around fork().\n"
        "\n";
    cerr << KNOB_BASE::StringKnobSummary();
    cerr << endl;
    return -1;
}

pid_t parent_pid;
PIN_LOCK lock;

VOID BeforeFork(THREADID threadid, const CONTEXT* ctxt, VOID * arg)
{
    PIN_GetLock(&lock, threadid+1);
    cerr << "TOOL: Before fork." << endl;
    PIN_ReleaseLock(&lock);
    parent_pid = PIN_GetPid();
}

VOID AfterForkInParent(THREADID threadid, const CONTEXT* ctxt, VOID * arg)
{
    PIN_GetLock(&lock, threadid+1);
    cerr << "TOOL: After fork in parent." << endl;
    PIN_ReleaseLock(&lock);

    if (PIN_GetPid() != parent_pid)
    {
        cerr << "PIN_GetPid() fails in parent process" << endl;
        exit(-1);
    }
}

VOID AfterForkInChild(THREADID threadid, const CONTEXT* ctxt, VOID * arg)
{
    PIN_GetLock(&lock, threadid+1);
    cerr << "TOOL: After fork in child." << endl;
    PIN_ReleaseLock(&lock);

    if ((PIN_GetPid() == parent_pid) || (getppid() != parent_pid))
    {
        cerr << "PIN_GetPid() fails in child process" << endl;
        exit(-1);
    }
}
```

```

}

int main(INT32 argc, CHAR **argv)
{
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }

    // Initialize the pin lock
    PIN_InitLock(&lock);

    // Register a notification handler that is called when the application
    // forks a new process.
    PIN_AddForkFunction(FPOINT_BEFORE, BeforeFork, 0);
    PIN_AddForkFunction(FPOINT_AFTER_IN_PARENT, AfterForkInParent, 0);
    PIN_AddForkFunction(FPOINT_AFTER_IN_CHILD, AfterForkInChild, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}

```

## Managed platforms support

Pin allows Pintools to identify dynamically created code using **RTN\_IsDynamic()** API (only code of functions which are reported by [Jit Profiling API](#)). The following example demonstrates use of **RTN\_IsDynamic()** API. This example instruments a program to count the total number of instructions discovered and executed. The instructions are divided to three categories: native instructions, dynamic instructions and instructions without any known routine.

Here is how to run it and display its output with a 32 bit OpenCL sample on Windows:

```

$ set CL_CONFIG USE_VTUNE=True
$ set INTEL_JIT_PROFILER32=ia32\bin\pinjitprofiling.dll
$ ia32\bin\pin.exe -t source\tools\JitProfilingApiTests\obj-ia32\DynamicInsCount.dll -support_jit_api -o DynamicInsCount.out -- ..\OpenCL\Win32\Debug\BitonicSort.
No command line arguments specified, using default values.
Initializing OpenCL runtime...
Trying to run on a CPU
OpenCL data alignment is 128 bytes.
Reading file 'BitonicSort.cl' (size 3435 bytes)
Sort order is ascending
Input size is 1048576 items
Executing OpenCL kernel...
Executing reference...
Performing verification...
Verification succeeded.
NDRange perf. counter time 12994.272962 ms.
Releasing resources...
$ type JitInsCount.out

=====
Number of executed native instructions: 7631596649
Number of executed jitted instructions: 438983207
Number of executed instructions without any known routine: 12246
=====
Number of discovered native instructions: 870531
Number of discovered jitted instructions: 223
Number of discovered instructions without any known routine: 36
=====

$

```

The example can be found in source.cpp

```

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

// =====
// Global variables
// =====

UINT64 insNativeDiscoveredCount = 0; //number of discovered native instructions
UINT64 insDynamicDiscoveredCount = 0; //number of discovered dynamic instructions
UINT64 insNoRtnDiscoveredCount = 0; //number of discovered instructions without any known routine

UINT64 insNativeExecutedCount = 0; //number of executed native instructions
UINT64 insDynamicExecutedCount = 0; //number of executed dynamic instructions
UINT64 insNoRtnExecutedCount = 0; //number of executed instructions without any known routine

std::ostream * out = &cerr;

// =====
// Command line switches
// =====

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "", "specify file name for output");

// =====
// Utilities
// =====

// Print out help message.
INT32 Usage()
{
    cerr << "This tool prints out the number of native and dynamic instructions" << endl;
    cerr << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

// =====
// Analysis routines

```

```

// =====

// This function is called before every native instruction is executed
VOID InsNativeCount()
{
    ++insNativeExecutedCount;
}

// This function is called before every dynamic instruction is executed
VOID InsDynamicCount()
{
    ++insDynamicExecutedCount;
}

// This function is called before every instruction without any known routine is executed
VOID InsNoRtnCount()
{
    ++insNoRtnExecutedCount;
}

// =====
// Instrumentation callbacks
// =====

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    RTN rtn = INS_Rtn(ins);
    if (!RTN_Valid(rtn))
    {
        ++insNoRtnDiscoveredCount;
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)InsNoRtnCount, IARG_END);
    }
    else if (RTN_IsDynamic(rtn))
    {
        ++insDynamicDiscoveredCount;
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)InsDynamicCount, IARG_END);
    }
    else
    {
        ++insNativeDiscoveredCount;
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)InsNativeCount, IARG_END);
    }
}

// Print out analysis results.
// This function is called when the application exits.
// @param[in] code exit code of the application
// @param[in] v value specified by the tool in the
// PIN_AddFiniFunction function call
VOID Fini(INT32 code, VOID *v)
{
    *out << "===== " << endl;
    *out << "Number of executed native instructions: " << insNativeExecutedCount << endl;
    *out << "Number of executed dynamic instructions: " << insDynamicExecutedCount << endl;
    *out << "Number of executed instructions without any known routine: " << insNoRtnExecutedCount << endl;
    *out << "===== " << endl;
    *out << "Number of discovered native instructions: " << insNativeDiscoveredCount << endl;
    *out << "Number of discovered dynamic instructions: " << insDynamicDiscoveredCount << endl;
    *out << "Number of discovered instructions without any known routine: " << insNoRtnDiscoveredCount << endl;
    *out << "===== " << endl;

    string fileName = KnobOutputFile.Value();
    if (!fileName.empty())
    {
        delete out;
    }
}

// The main procedure of the tool.
// This function is called when the application image is loaded but not yet started.
// @param[in] argc total number of elements in the argv array
// @param[in] argv array of command line arguments,
// including pin -t <toolname> -- ...
int main(int argc, char *argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize PIN library. Print help message if -h(elp) is specified
    // in the command line or the command line is invalid
    if(PIN_Init(argc,argv))
    {
        return Usage();
    }

    string fileName = KnobOutputFile.Value();

    if (!fileName.empty())
    {
        out = new std::ofstream(fileName.c_str());
    }

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, NULL);

    // Register function to be called when the application exits
    PIN_AddFiniFunction(Fini, NULL);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

Pin allows Pintools to instrument just compiled functions using [RTN\\_AddInstrumentFunction](#) API. Following example instruments a program to log Jitting and running of dynamic functions which are reported by [Jit Profiling API](#).

Here is how to run it with a 64 bit OpenCL sample on Linux:

```
$ setenv CL_CONFIG_USE_VTUNE True
$ setenv INTEL_JIT_PROFILER64 intel64/lib/libpinjitprofiling.so
$ ./pin -t source/tools/JitProfilingApiTests/obj-intel64/DynamicFuncInstrument.so -support_jit_api -o DynamicFuncInstrument.out -- ..\OpenCL\Win32\Debug\BitonicS
No command line arguments specified, using default values.
Initializing OpenCL runtime...
Trying to run on a CPU
OpenCL data alignment is 128 bytes.
Reading file 'BitonicSort.cl' (size 3435 bytes)
Sort order is ascending
Input size is 1048576 items
Executing OpenCL kernel...
Executing reference...
Performing verification...
Verification succeeded.
NDRange perf. counter time 12994.272962 ms.
Releasing resources...
$
```

The example can be found in source.cpp

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

// =====
// Global variables
// =====

std::ostream * out = &cerr;

// =====
// Command line switches
// =====

Knob<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "", "specify file name for output");

// =====
// Utilities
// =====

// Print out help message.
INT32 Usage()
{
    cerr << "This tool prints out the stack filtered by the dynamically created functions only" << endl;
    cerr << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

// =====
// Analysis routines
// =====

VOID RtnCallPrint(CHAR * rtnName)
{
    *out << "Before run " << rtnName << endl;
}

// =====
// Instrumentation callbacks
// =====

// Pin calls this function every time a new rtn is executed
VOID Routine(RTN rtn, VOID *v)
{
    if (!RTN_IsDynamic(rtn))
    {
        return;
    }

    *out << "Just discovered " << RTN_Name(rtn) << endl;

    RTN_Open(rtn);

    // Insert a call at the entry point of a routine to increment the call count
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)RtnCallPrint, IARG_ADDRINT, RTN_Name(rtn).c_str(), IARG_END);

    RTN_Close(rtn);
}

// Print out analysis results.
// This function is called when the application exits.
// @param[in] code exit code of the application
// @param[in] v value specified by the tool in the
// PIN_AddFiniFunction function call
VOID Fini(INT32 code, VOID *v)
{
    const string fileName = KnobOutputFile.Value();
    if (!fileName.empty())
    {
        delete out;
    }
}

// The main procedure of the tool.
// This function is called when the application image is loaded but not yet started.
// @param[in] argc total number of elements in the argv array
// @param[in] argv array of command line arguments,
// including pin -t <toolname> -- ...
```

```
int main(int argc, char *argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize PIN library. Print help message if -h(elp) is specified
    // in the command line or the command line is invalid
    if(PIN_Init(argc, argv))
    {
        return Usage();
    }

    const string fileName = KnobOutputFile.Value();

    if (!fileName.empty())
    {
        out = new std::ofstream(fileName.c_str());
    }

    // Register Routine to be called to instrument rtn
    RTN_AddInstrumentFunction(Routine, 0);

    // Register function to be called when the application exits
    PIN_AddFiniFunction(Fini, NULL);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

## Callbacks

The examples in the previous section have introduced a number of ways to register callback functions via the Pin API, such as:

- [INS\\_AddInstrumentFunction](#) (INSCALLBACK fun, VOID \*val)
- [TRACE\\_AddInstrumentFunction](#) (TRACECALLBACK fun, VOID \*val)
- [RTN\\_AddInstrumentFunction](#) (RTNCALLBACK fun, VOID \*val)
- [IMG\\_AddInstrumentFunction](#) (IMGCALLBACK fun, VOID \*val)
- [PIN\\_AddFiniFunction](#) (FINICALLBACK fun, VOID \*val)
- [PIN\\_AddDetachFunction](#) (DETACHCALLBACK fun, VOID \*val)

The extra parameter `val` (shared by all the registration functions) will be passed to `fun` as its second argument whenever it is "called back". This is a standard mechanism used in GUI programming with callbacks.

If this feature is not needed, it is safe to pass 0 for `val` when registering a callback. The expected use of `val` is to pass a pointer to an instance of a class. Since `val` is a generic pointer, `fun` must cast it back to an object before dereferencing the pointer.

Note that all callback registration functions return a [PIN\\_CALLBACK](#) object which can later be used to manipulate the properties of the registered callback (for example change the order in which PIN executes callback functions of the same type). This can be done by calling API functions that manipulates the [PIN\\_CALLBACK](#) object (see [PIN callbacks](#))

## Modifying Application Instructions

Although Pin is most commonly used for instrumenting applications, it is also possible to change the application's instructions. The simplest way to do this is to insert an analysis routine to emulate an instruction, and then use [INS\\_Delete\(\)](#) to remove the original instruction. It is also possible to insert direct or indirect branches (using [INS\\_InsertDirectJump](#) and [INS\\_InsertIndirectJump](#)), which makes it easier to emulate instructions that change the control flow.

The memory addresses accessed by an instruction can be modified to refer to a value calculated by an analysis routine using [INS\\_RewriteMemoryOperand](#).

Note that in all of the cases where an instruction is modified, the modification is only made after all of the instrumentation routines have been executed. Therefore all of the instrumentation routines see the original, un-modified instruction.

## The Pin Advanced Debugging Extensions

Pin's advanced debugging extensions allow you to debug an application, even while it runs under Pin in JIT mode. Moreover, your Pin tool can add support for new debugger commands, without making any changes to GDB or Visual Studio. This allows you to interactively control your Pin tool from within a live debugger session. Finally, Pin tools can add powerful new debugger features that are enabled via instrumentation. For example, a Pin tool can use instrumentation to look for an interesting condition (like a memory buffer overwrite) and then stop at a live debugger session when that condition occurs.

This section illustrates these three concepts:

- Enabling all the traditional debugger features even while running an application under Pin in JIT mode.
- Recognizing new debugger commands in your Pin tool to allow interactive control of the tool from a live debugger session.
- Adding support for new debugger features by writing a Pin tool.

These features are available on both Linux (using GDB) and on Windows (using Visual Studio). The Pin APIs are the same in both cases, but their usage from within the debugger is different because each debugger has a different UI. The following tutorial is divided into two sections: one that is Linux centric and another that is Windows centric. They both describe the same example, so you can continue by reading either section.

- [Tutorial for Linux](#)
- [Tutorial for Windows](#)
- [Tutorial for OS X\\*](#)

Finally, note that these advanced debugging extensions are not at all related to debugging your Pintool. If you have a bug in your tool and need to debug it, see the section [Tips for Debugging a Pintool](#) instead.



## Advanced Debugging Extensions on Linux

Pin's debugging extensions on Linux work with nearly all modern versions of GDB, so you can probably use whatever version of GDB is already installed on your system. Pin uses GDB's remote debugger features, so it should work with any version of GDB that supports that feature.

Throughout this section, we demonstrate the debugging extensions in Pin with the example tool "stack-debugger.cpp", which is available in the directory "source/tools/ManualExamples". You may want to compile that tool and follow along:

```
$ cd source/tools/ManualExamples
$ make DEBUG=1 stack-debugger.test
```

The tool and its associated test application, "fibonacci", are built in a directory named "obj-ia32", "obj-intel64", etc., depending on your machine type.

To enable the debugging extensions, run Pin with the **-appdebug** command line switch. This causes Pin to start the application and stop immediately before the first instruction. Pin then prints a message telling you to start GDB.

```
$ ../../../../pin -appdebug -t obj-intel64/stack-debugger.so -- obj-intel64/fibonacci 1000
Application stopped until continued from debugger.
Start GDB, then issue this command at the (gdb) prompt:
target remote :33030
```

In another window, start GDB and enter the command that Pin printed:

```
$ gdb fibonacci
(gdb) target remote :33030
```

At this point, the debugger is attached to the application that is running under Pin. You can set breakpoints, continue execution, print out variables, disassemble code, etc.

```
(gdb) break main
Breakpoint 1 at 0x401194: file fibonacci.cpp, line 12.
(gdb) cont
Continuing.

Breakpoint 1, main (argc=2, argv=0x7fbffff3c8) at fibonacci.cpp:12
12      if (argc > 2)
(gdb) print argc
$1 = 2
(gdb) x/4i $pc
0x401194 <main+27>:    cmpl    $0x2, 0xfffffffffffffe5c(%rbp)
0x40119b <main+34>:    je      0x4011c8 <main+79>
0x40119d <main+36>:    mov     $0x402080,%esi
0x4011a2 <main+41>:    mov     $0x603300,%edi
```

Of course, any information you observe in the debugger shows the application's "pure" state. The details of Pin and the tool's instrumentation are hidden. For example, the disassembly you see above shows only the application's instructions, not any of the instructions inserted by the tool. However, when you use commands like "cont" or "step" to advance execution of the application, your tool's instrumentation runs as it normally would under Pin.

### Note:

After connecting GDB with the "target remote" command, you should NOT use the "run" command. The application is already running and stopped at the first instruction. Instead, use the "cont" command to continue execution.

## Adding New Debugger Commands

The previous section illustrated how you can enable the normal debugger features while running an application under Pin. Now, let's see how your Pintool can add new custom debugger commands, even without changing GDB. Custom debugger commands are useful because they allow you to control your Pintool interactively from within a live debugger session. For example, you can ask your Pintool to print out information that it has collected, or you can interactively enable instrumentation only for certain phases of the application.

To illustrate, see the call to **PIN\_AddDebugInterpreter()** in the stack-debugger tool. That API sets up the following call-back function:

```
static BOOL DebugInterpreter(THREADID tid, CONTEXT *ctxt, const string &cmd, string *result, VOID *)
{
    TINFO_MAP::iterator it = ThreadInfos.find(tid);
    if (it == ThreadInfos.end())
        return FALSE;
    TINFO *tinfo = it->second;

    std::string line = TrimWhitespace(cmd);
    *result = "";

    // [...]

    if (line == "stats")
    {
        ADDRINT sp = PIN_GetContextReg(ctxt, REG_STACK_PTR);
        tinfo->os.str("");
        if (sp <= tinfo->_stackBase)
            tinfo->os << "Current stack usage: " << std::dec << (tinfo->_stackBase - sp) << " bytes.\n";
        else
            tinfo->os << "Current stack usage: -" << std::dec << (sp - tinfo->_stackBase) << " bytes.\n";
        tinfo->os << "Maximum stack usage: " << tinfo->_max << " bytes.\n";
        *result = tinfo->os.str();
        return TRUE;
    }
    else if (line == "stacktrace on")
    {
        if (!EnableInstrumentation)
        {
            PIN_RemoveInstrumentation();
            EnableInstrumentation = true;
            *result = "Stack tracing enabled.\n";
        }
        return TRUE;
    }

    // [...]

    return FALSE; // Unknown command
}
```

```
}

```

The **PIN\_AddDebugInterpreter()** API allows a Pintool to establish a handler for extended GDB commands. For example, the code snippet above implements the new commands "stats" and "stacktrace on". You can execute these commands in GDB by using the "monitor" command:

```
(gdb) monitor stats
Current stack usage: 688 bytes.
Maximum stack usage: 0 bytes.
```

A Pintool can do various things when the user types an extended debugger command. For example, the "stats" command prints out some information that the tool has collected. Any text that the tool writes to the "result" parameter is printed to the GDB console. Note that the CONTEXT parameter has the register state for the debugger's "focus" thread, so the tool can easily display information about this focus thread.

You can also use an extended debugger command to interactively enable or disable instrumentation in your Pintool, as demonstrated by the "stacktrace on" command. For example, if you wanted to quickly run your Pintool over the application's initial start-up phase, you could run with your Pintool's instrumentation disabled until a breakpoint is triggered. Then, you could use an extended command to enable instrumentation only during the interesting part of the application. In the stack-debugger example above, the call to **PIN\_RemoveInstrumentation()** causes Pin to discard any previous instrumentation, so the tool re-instruments the code when the debugger continues execution of the application. As we will see later, the tool's global variable "EnableInstrumentation" adjusts the instrumentation that it inserts.

## Semantic Breakpoints

The last major feature of the advanced debugging extensions is the ability to stop execution at a breakpoint by calling an API from your tool's analysis code. This may sound simple, but it is very powerful. Your Pintool can use instrumentation to look for a complex condition and then stop at a breakpoint when that condition occurs.

The "stack-debugger" tool illustrates this by using instrumentation to observe all the instructions that allocate stack space, and then it stops at a breakpoint whenever the application's stack usage reaches some threshold. In effect, this adds a new feature to the debugger that could not be practically implemented using traditional debugger technology because a traditional debugger can not reasonably find all the instructions that allocate stack space. A Pintool, however, can do this quite easily via instrumentation.

The example code below from the "stack-debugger" tool uses Pin instrumentation to identify all the instructions that allocate stack space.

```
static VOID Instruction(INS ins, VOID *)
{
    if (!EnableInstrumentation)
        return;

    if (INS_RegWContain(ins, REG_STACK_PTR))
    {
        IPOINT where = IPOINT_AFTER;
        if (!INS_HasFallThrough(ins))
            where = IPOINT_TAKEN_BRANCH;

        INS_InsertIfCall(ins, where, (AFUNPTR)OnStackChangeIf, IARG_REG_VALUE, REG_STACK_PTR,
            IARG_REG_VALUE, RegTinfo, IARG_END);
        INS_InsertThenCall(ins, where, (AFUNPTR)DoBreakpoint, IARG_CONST_CONTEXT, IARG_THREAD_ID, IARG_END);
    }
}
```

The call to **INS\_RegWContain()** tests whether an instruction modifies the stack pointer. If it does, we insert an analysis call immediately after the instruction, which checks to see if the application's stack usage exceeds a threshold.

Also notice that all the instrumentation is gated by the global flag "EnableInstrumentation", which we saw earlier in the "stacktrace on" command. Thus, the user can disable instrumentation (with "stacktrace off") in order to execute quickly through uninteresting parts of the application, and then re-enable it (with "stacktrace on") for the interesting parts.

The analysis routine OnStackChangeIf() returns TRUE if the application's stack usage has exceeded the threshold. When this happens, the tool calls the DoBreakpoint() analysis routine, which will stop at the debugger breakpoint. Notice that we use if / then instrumentation here because the call to DoBreakpoint() requires a "CONTEXT \*" parameter, which can be slow.

```
static ADDRINT OnStackChangeIf(ADDRINT sp, ADDRINT addrInfo)
{
    TINFO *tinfo = reinterpret_cast<TINFO *>(addrInfo);

    // The stack pointer may go above the base slightly. (For example, the application's dynamic
    // loader does this briefly during start-up.)
    //
    if (sp > tinfo->_stackBase)
        return 0;

    // Keep track of the maximum stack usage.
    //
    size_t size = tinfo->_stackBase - sp;
    if (size > tinfo->_max)
        tinfo->_max = size;

    // See if we need to trigger a breakpoint.
    //
    if (BreakOnNewMax && size > tinfo->_maxReported)
        return 1;
    if (BreakOnSize && size >= BreakOnSize)
        return 1;
    return 0;
}

static VOID DoBreakpoint(const CONTEXT *ctxt, THREADID tid)
{
    TINFO *tinfo = reinterpret_cast<TINFO *>(PIN_GetContextReg(ctxt, RegTinfo));

    // Keep track of the maximum reported stack usage for "stackbreak newmax".
    //
    size_t size = tinfo->_stackBase - PIN_GetContextReg(ctxt, REG_STACK_PTR);
    if (size > tinfo->_maxReported)
        tinfo->_maxReported = size;

    ConnectDebugger(); // Ask the user to connect a debugger, if it is not already connected.

    // Construct a string that the debugger will print when it stops. If a debugger is
```

```
// not connected, no breakpoint is triggered and execution resumes immediately.
//
tinfo->_os.str("");
tinfo->_os << "Thread " << std::dec << tid << " uses " << size << " bytes of stack.";
PIN_ApplicationBreakpoint(ctxt, tid, FALSE, tinfo->_os.str());
}
```

The analysis routine `OnStackChangelf()` keeps track of some metrics on stack usage and tests whether the threshold has been reached. If the threshold is crossed, it returns non-zero, and Pin executes the `DoBreakpoint()` analysis routine.

The interesting part of `DoBreakpoint()` is at the very end, where it calls `PIN_ApplicationBreakpoint()`. This API causes Pin to stop the execution of all threads and triggers a breakpoint in the debugger. There is also a string parameter to `PIN_ApplicationBreakpoint()`, which GDB prints at the console when the breakpoint triggers. A Pintool can use this string to tell the user why a breakpoint triggered. In our example tool, this string says something like "Thread 10 uses 4000 bytes of stack".

We can see the breakpoint feature in action in our example tool by using the "stackbreak 4000" command like this:

```
(gdb) monitor stackbreak 4000
Will break when thread uses more than 4000 bytes of stack.
(gdb) c
Continuing.
Thread 0 uses 4000 bytes of stack.
Program received signal SIGTRAP, Trace/breakpoint trap.
0x000000000400e27 in Fibonacci (num=0) at fibonacci.cpp:34
(gdb)
```

When you are done, you can either continue the application and let it terminate, or you can quit from the debugger:

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
```

## Connecting the Debugger Later

In the previous example, we used the Pin switch `-appdebug` to stop the application and debug it from the first instruction. You can also enable Pin's debugging extensions without stopping at the first instruction. The following example shows how you can use the stack-debugger tool to start the application and attach with the debugger only after it triggers a stack limit breakpoint.

```
$ ../../../../pin -appdebug_enable -appdebug_silent -t obj-intel64/stack-debugger.so -stackbreak 4000 -- obj-intel64/fibonacci 1000
```

The `-appdebug_enable` switch tells Pin to enable application debugging without stopping at the first instruction. The `-appdebug_silent` switch disables the message that tells how to connect with GDB. As we will see later, the Pintool can print a custom message instead. Finally, the "-stackbreak 4000" switch tells the stack-debugger tool to trigger a breakpoint when the stack grows to 4000 bytes. When the tool does trigger a breakpoint, it prints a message like this:

```
Triggered stack-limit breakpoint.
Start GDB and enter this command:
target remote :45462
```

You can now connect with GDB as you did before, except now GDB stops the application at the point where the stack-debugger tool triggered the stack-limit breakpoint.

```
gdb fibonacci
(gdb) target remote :45462
0x000000000400e27 in Fibonacci (num=0) at fibonacci.cpp:37
(gdb)
```

Let's look at the code in the tool that connects to the debugger now.

```
static void ConnectDebugger()
{
    if (PIN_GetDebugStatus() != DEBUG_STATUS_UNCONNECTED)
        return;

    DEBUG_CONNECTION_INFO info;
    if (!PIN_GetDebugConnectionInfo(&info) || info._type != DEBUG_CONNECTION_TYPE_TCP_SERVER)
        return;

    *Output << "Triggered stack-limit breakpoint.\n";
    *Output << "Start GDB and enter this command:\n";
    *Output << "target remote : " << std::dec << info._tcpServer._tcpPort << "\n";
    *Output << std::flush;

    if (PIN_WaitForDebuggerToConnect(1000*KnobTimeout.Value()))
        return;

    *Output << "No debugger attached after " << KnobTimeout.Value() << " seconds.\n";
    *Output << "Resuming application without stopping.\n";
    *Output << std::flush;
}
```

The `ConnectDebugger()` function is called each time the tool wants to stop at a breakpoint. It first calls `PIN_GetDebugStatus()` to see if Pin is already connected to a debugger. If not, it uses `PIN_GetDebugConnectionInfo()` to get the TCP port number that is needed to connect GDB to Pin. This is, for example, the "45462" number that the user types in the "target remote" command. After asking the user to start GDB, the tool then calls `PIN_WaitForDebuggerToConnect()` to wait for GDB to connect. If the user doesn't start GDB after a timeout period, the tool prints a message and then continues executing the application.

As before, you can either continue the application and let it terminate, or you can quit from the debugger:

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
```

## Advanced Debugging Extensions on Windows

On Windows, the advanced debugging extensions work with Microsoft Visual Studio 2012. There is no support for earlier versions of Visual Studio, so make sure you have that version installed. Also, the Express edition of Visual Studio doesn't support IDE extensions, so it will not work with the Pin debugger extensions. Therefore, you must install the Professional edition (or greater). If you are a student, you may be able to get the Professional edition for free. Check the Microsoft web site or with your school's IT department for details.

After you have installed Visual Studio, you must also install the Pin extension for Visual Studio. Look for an installer named "pinadx-vsextension-X.Y.Z.msi" in the

root of the Pin kit.

If you are developing a Pintool, you must also use Visual Studio to build the tool. However, there is not yet support for building Pintools with Visual Studio 2012. As a result, you must also install either Visual Studio 2010 or 2008 in order to build your tool. You can use any edition to build a Pintool, even the free Express edition.

The remainder of this section assumes that you are able to build the "stack-debugger" tool, so if you want to follow along, you must have the following software installed:

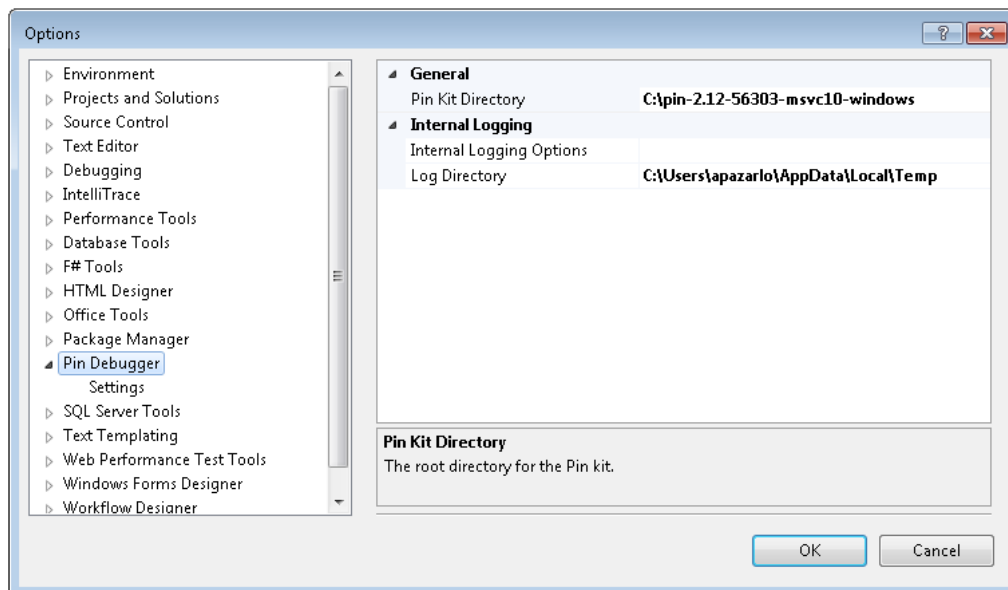
- Visual Studio 2012, Professional edition (or greater).
- The Pin debugger extension for Visual Studio 2012 (pinadx-vsextension-X.Y.Z.msi).
- Either Visual Studio 2010 or 2008, any edition.

In order to start this tutorial, you will probably want to build the example tool "stack-debugger.cpp", which is available in the directory "source\tools\ManualExamples". To do this, open a Visual Studio command shell and type the following commands. (Use "TARGET=intel64" instead, if you want to build a 64-bit version of the tool.)

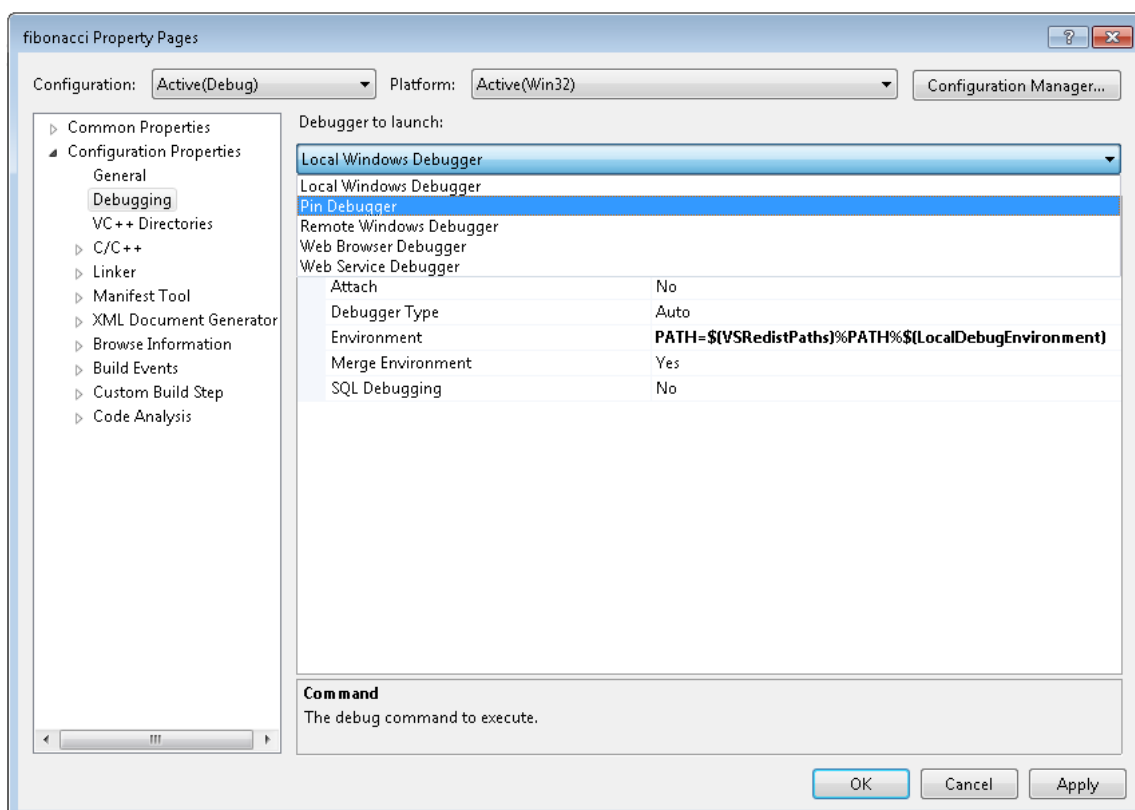
```
C:\> cd source\tools\ManualExamples
C:\> make TARGET=ia32 obj-ia32/stack-debugger.dll
```

After you have done this, start Visual Studio 2012 and open the sample solution file at "source\tools\ManualExamples\stack-debugger-tutorial.sln". Then build the sample application "fibonacci" by pressing F7. Make sure you can run the application natively by pressing CTRL-F5.

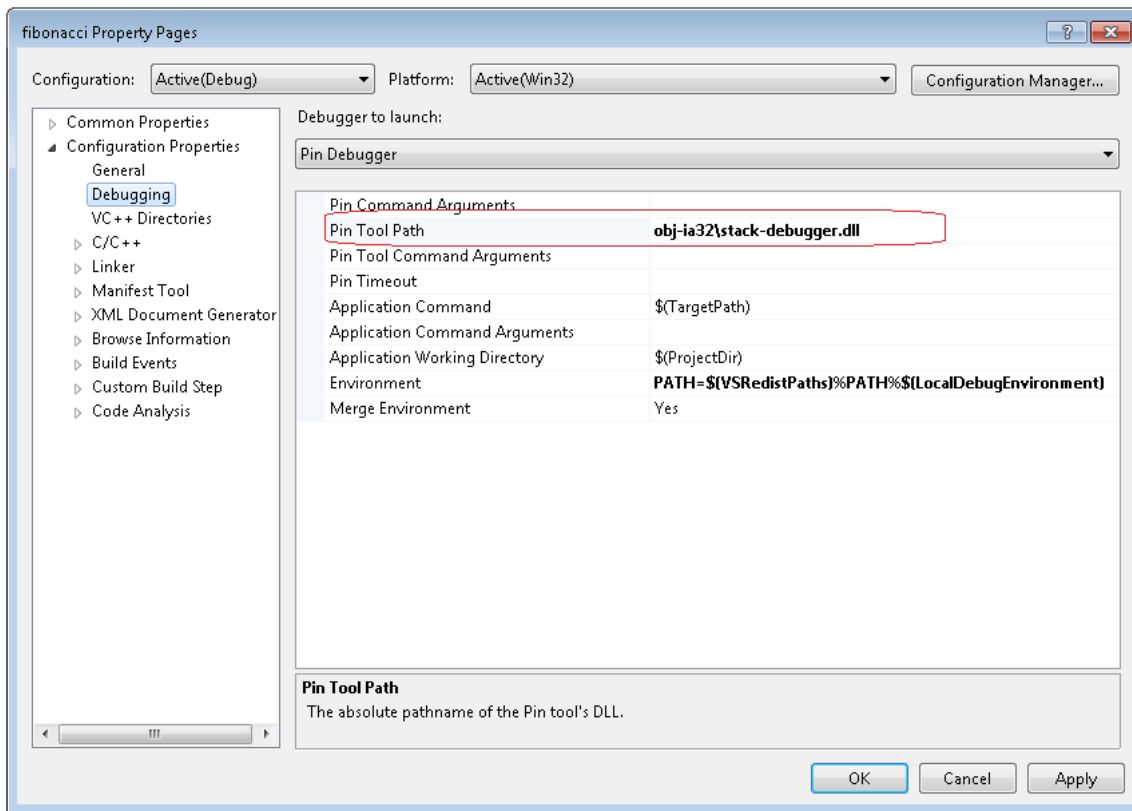
Now let's try running the "fibonacci" application under Pin with the "stack-debugger" tool. To do this, you must first set the "Pin Kit Directory" from TOOLS->Options->Pin Debugger.



Then you have to adjust the "fibonacci" project properties in Visual Studio 2012: right-click on the "fibonacci" project in the Solution Explorer, choose Properties, and then click on Debugging. Change the drop-down titled "Debugger to launch" to "Pin Debugger" as shown in the figure below.



Then, set the "Pin Tool Path" property by browsing to the "stack-debugger.dll". Press OK when you are done.



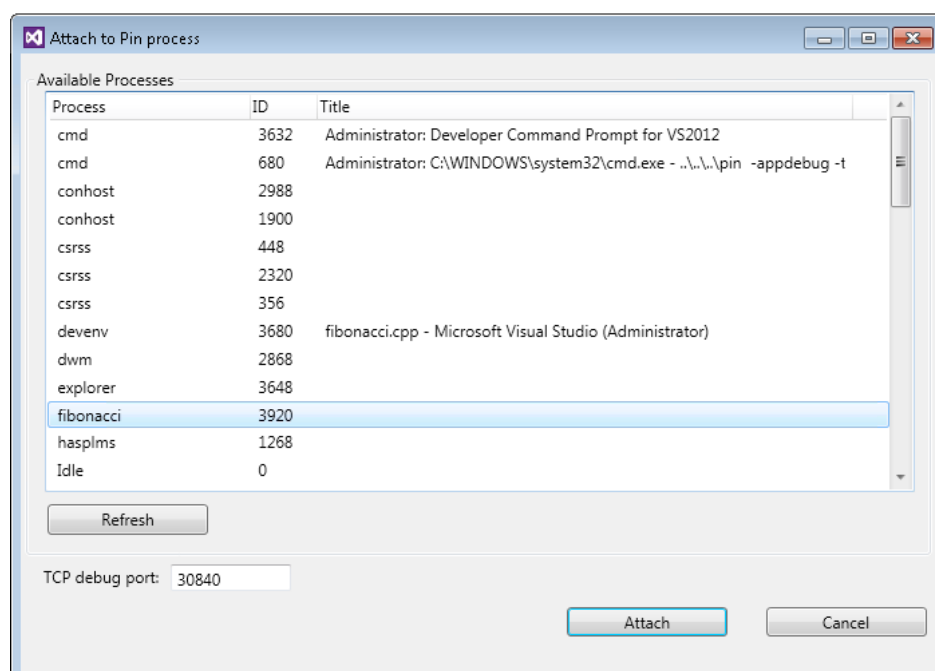
Visual Studio is now configured to run the "fibonacci" application under your Pin tool. However, before you continue, set a breakpoint in "main()" so that execution stops in the debugger. Then press F5 to start debugging.

You should now see a normal-looking debugger session, although your application is really running under control of Pin. All of the debugger features still work as you would expect. You can set breakpoints, continue execution, display the values of variables, and even view the disassembled code. All of the information that you observe in the debugger shows the application's "pure" state. The details of Pin and the tool's instrumentation are hidden. For example, the disassembly view shows only the application's instructions, not any of the instructions inserted by the tool. However, when you continue execution (e.g. with F5 or F10), the application executes along with your tool's instrumentation code.

Now, let's see an alternative way to debug the "fibonacci" application under Pin with the "stack-debugger" tool in Visual Studio 2012. After you have built the "stack-debugger" tool, open a command shell and start the application with the debugging extensions enabled. This will cause Pin to stop immediately before the first instruction.

```
C:\> cd source\tools\ManualExamples
C:\> ..\..\..\pin -appdebug -t obj-ia32\stack-debugger.dll -- debug\fibonacci.exe 1000
Application stopped until continued from debugger.
Pin ready to accept debugger connection on port 30840
```

Open the source.cpp in Visual Studio 2012 and set a breakpoint to stop the execution in the debugger. To attach with Visual Studio to the process that is running under Pin, select "Attach to Pin Process" on the DEBUG menu. Select from the Available Processes table the "fibonacci" process, enter the port number that Pin printed and click Attach.



### Adding New Debugger Commands

The previous section illustrated how you can enable the normal debugger features while running an application under Pin. Now, let's see how your Pintool can add new custom debugger commands, even without changing Visual Studio. Custom debugger commands are useful because they allow you to control your Pintool

interactively from within a live debugger session. For example, you can ask your Pintool to print out information that it has collected, or you can interactively enable instrumentation only for certain phases of the application.

To illustrate, see the call to **PIN\_AddDebugInterpreter()** in the stack-debugger tool. That API sets up the following call-back function:

```
static BOOL DebugInterpreter(THREADID tid, CONTEXT *ctxt, const string &cmd, string *result, VOID *)
{
    TINFO_MAP::iterator it = ThreadInfos.find(tid);
    if (it == ThreadInfos.end())
        return FALSE;
    TINFO *tinfo = it->second;

    std::string line = TrimWhitespace(cmd);
    *result = "";

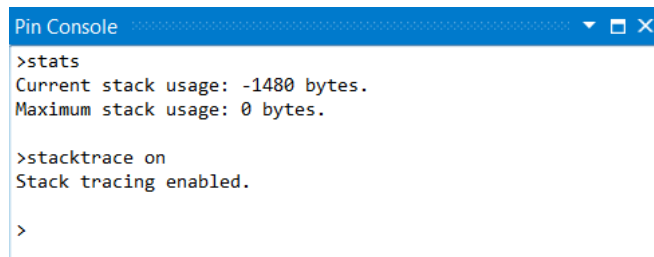
    // [...]

    if (line == "stats")
    {
        ADDRINT sp = PIN_GetContextReg(ctxt, REG_STACK_PTR);
        tinfo->_os.str("");
        if (sp <= tinfo->_stackBase)
            tinfo->_os << "Current stack usage: " << std::dec << (tinfo->_stackBase - sp) << " bytes.\n";
        else
            tinfo->_os << "Current stack usage: -" << std::dec << (sp - tinfo->_stackBase) << " bytes.\n";
        tinfo->_os << "Maximum stack usage: " << tinfo->_max << " bytes.\n";
        *result = tinfo->_os.str();
        return TRUE;
    }
    else if (line == "stacktrace on")
    {
        if (!EnableInstrumentation)
        {
            PIN_RemoveInstrumentation();
            EnableInstrumentation = true;
            *result = "Stack tracing enabled.\n";
        }
        return TRUE;
    }

    // [...]

    return FALSE; // Unknown command
}
```

The **PIN\_AddDebugInterpreter()** API allows a Pintool to establish a handler for extended debugger commands. For example, the code snippet above implements the new commands "stats" and "stacktrace on". You can execute these commands in Visual Studio by opening "DEBUG->Windows->Pin Console" in the IDE.



A Pintool can do various things when the user types an extended debugger command. For example, the "stats" command prints out some information that the tool has collected. Any text that the tool writes to the "result" parameter is printed to the Visual Studio Pin Console window. Note that the CONTEXT parameter has the register state for the debugger's "focus" thread, so the tool can easily display information about this focus thread.

You can also use an extended debugger command to interactively enable or disable instrumentation in your Pintool, as demonstrated by the "stacktrace on" command. For example, if you wanted to quickly run your Pintool over the application's initial start-up phase, you could run with your Pintool's instrumentation disabled until a breakpoint is triggered. Then, you could use an extended command to enable instrumentation only during the interesting part of the application. In the stack-debugger example above, the call to **PIN\_RemoveInstrumentation()** causes Pin to discard any previous instrumentation, so the tool re-instruments the code when the debugger continues execution of the application. As we will see later, the tool's global variable "EnableInstrumentation" adjusts the instrumentation that it inserts.

## Semantic Breakpoints

The last major feature of the advanced debugging extensions is the ability to stop execution at a breakpoint by calling an API from your tool's analysis code. This may sound simple, but it is very powerful. Your Pintool can use instrumentation to look for a complex condition and then stop at a breakpoint when that condition occurs.

The "stack-debugger" tool illustrates this by using instrumentation to observe all the instructions that allocate stack space, and then it stops at a breakpoint whenever the application's stack usage reaches some threshold. In effect, this adds a new feature to the debugger that could not be practically implemented using traditional debugger technology because a traditional debugger can not reasonably find all the instructions that allocate stack space. A Pintool, however, can do this quite easily via instrumentation.

The example code below from the "stack-debugger" tool uses Pin instrumentation to identify all the instructions that allocate stack space.

```
static VOID Instruction(INS ins, VOID *)
{
    if (!EnableInstrumentation)
        return;

    if (INS_RegWContain(ins, REG_STACK_PTR))
    {
        IPOINTE where = IPOINTE_AFTER;
        if (!INS_HasFallThrough(ins))
            where = IPOINTE_TAKEN_BRANCH;

        INS_InsertIfCall(ins, where, (AFUNPTR)OnStackChangeIf, IARG_REG_VALUE, REG_STACK_PTR,
            IARG_REG_VALUE, RegTinfo, IARG_END);
        INS_InsertThenCall(ins, where, (AFUNPTR)DoBreakpoint, IARG_CONST_CONTEXT, IARG_THREAD_ID, IARG_END);
    }
}
```

```
}
}
```

The call to [INS\\_RegWContain\(\)](#) tests whether an instruction modifies the stack pointer. If it does, we insert an analysis call immediately after the instruction, which checks to see if the application's stack usage exceeds a threshold.

Also notice that all the instrumentation is gated by the global flag "EnableInstrumentation", which we saw earlier in the "stacktrace on" command. Thus, the user can disable instrumentation (with "stacktrace off") in order to execute quickly through uninteresting parts of the application, and then re-enable it (with "stacktrace on") for the interesting parts.

The analysis routine OnStackChangeIf() returns TRUE if the application's stack usage has exceeded the threshold. When this happens, the tool calls the DoBreakpoint() analysis routine, which will stop at the debugger breakpoint. Notice that we use if / then instrumentation here because the call to DoBreakpoint() requires a "CONTEXT \*" parameter, which can be slow.

```
static ADDRINT OnStackChangeIf(ADDRINT sp, ADDRINT addrInfo)
{
    TINFO *tinfo = reinterpret_cast<TINFO *>(addrInfo);

    // The stack pointer may go above the base slightly. (For example, the application's dynamic
    // loader does this briefly during start-up.)
    //
    if (sp > tinfo->_stackBase)
        return 0;

    // Keep track of the maximum stack usage.
    //
    size_t size = tinfo->_stackBase - sp;
    if (size > tinfo->_max)
        tinfo->_max = size;

    // See if we need to trigger a breakpoint.
    //
    if (BreakOnNewMax && size > tinfo->_maxReported)
        return 1;
    if (BreakOnSize && size >= BreakOnSize)
        return 1;
    return 0;
}

static VOID DoBreakpoint(const CONTEXT *ctxt, THREADID tid)
{
    TINFO *tinfo = reinterpret_cast<TINFO *>(PIN_GetContextReg(ctxt, RegTinfo));

    // Keep track of the maximum reported stack usage for "stackbreak newmax".
    //
    size_t size = tinfo->_stackBase - PIN_GetContextReg(ctxt, REG_STACK_PTR);
    if (size > tinfo->_maxReported)
        tinfo->_maxReported = size;

    ConnectDebugger(); // Ask the user to connect a debugger, if it is not already connected.

    // Construct a string that the debugger will print when it stops. If a debugger is
    // not connected, no breakpoint is triggered and execution resumes immediately.
    //
    tinfo->_os.str("");
    tinfo->_os << "Thread " << std::dec << tid << " uses " << size << " bytes of stack.";
    PIN_ApplicationBreakpoint(ctxt, tid, FALSE, tinfo->_os.str());
}
```

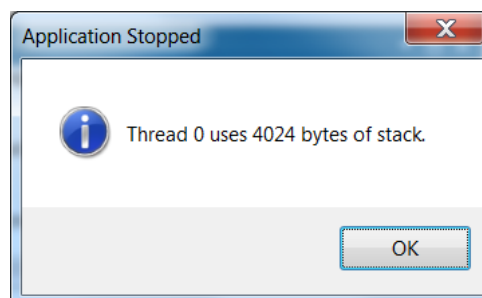
The analysis routine OnStackChangeIf() keeps track of some metrics on stack usage and tests whether the threshold has been reached. If the threshold is crossed, it returns non-zero, and Pin executes the DoBreakpoint() analysis routine.

The interesting part of DoBreakpoint() is at the very end, where it calls [PIN\\_ApplicationBreakpoint\(\)](#). This API causes Pin to stop the execution of all threads and triggers a breakpoint in the debugger. There is also a string parameter to [PIN\\_ApplicationBreakpoint\(\)](#), which is displayed in Visual Studio when the breakpoint triggers. A Pintool can use this string to tell the user why a breakpoint triggered. In our example tool, this string says something like "Thread 10 uses 4000 bytes of stack".

We can see the breakpoint feature in action in our example tool by typing this command in the Pin Console window:

```
>stackbreak 4000
Will break when thread uses more than 4000 bytes of stack.
```

Then press F5 to continue execution. The application should stop in the debugger again with a message like this:



When you are done, you can either continue the application with F5 or terminate it with SHIFT-F5.

### Advanced Debugging Extensions on OS X\*

Pin's debugging extensions on OS X are very similar to Linux, with the one exception that the debugger used is LLDB. To work with LLDB, the -appdebug\_lldb\_options flag must be specified at the pin command line.

```
$ ../.././pin -appdebug -appdebug_lldb_options -t obj-intel64/stack-debugger.dylib -- obj-intel64/fibonacci 1000
Application stopped until continued from debugger.
Start GDB, then issue this command at the (gdb) prompt:
target remote :33030
```



**Warning:**

For the debugger extensions to work properly, they require the target application symbols. Make sure to call [PIN\\_InitSymbols](#) in your pintool.

## Applying a Pintool to an Application

An application and a tool are invoked as follows:

```
pin [pin-option]... -t [toolname] [tool-options]... -- [application] [application-option]..
```

These are a few of the Pin options are currently available. See [Command Line Switches](#) for the complete list.

- **-t *toolname*:** Specifies the Pintool to use. If you are running a 32-bit application in an IA-32 architecture, or a 64-bit application on an Intel(R) 64 architecture, only **-t <toolname>** is needed. If you are running an application on an Intel(R) 64 architecture, where all of the components in the chain are either 32-bit or 64-bit, but not both, only **-t <toolname>** is needed. If you are running an application on an Intel(R) 64 architecture, where components in the chain are both 32-bit and 64-bit, use **-t64 <64-bit toolname>** to specify the 64-bit tool binary followed by **-t <32-bit toolname>** to specify the 32-bit tool binary and the tool options. For more information, see [Instrumenting Applications on Intel\(R\) 64 Architectures](#).
- **-t64 *toolname*:** Specify 64-bit tool binary for Intel(R) 64 architecture. If you are running an application on an Intel(R) 64 architecture, where components in the chain are both 32-bit and 64-bit, use **-t64** together with **-t** as described above. See [Instrumenting Applications on Intel\(R\) 64 Architectures](#).  
**Important:** Using **-t64** without **-t** is not recommended, since in this case when given a 32-bit application, Pin will run the application without applying any tool.
- **-p32 *toolname*:** Specify Pin binary for IA-32 architecture. See [Instrumenting Applications on Intel\(R\) 64 Architectures](#).
- **-p64 *toolname*:** Specify Pin binary for Intel(R) 64 architecture. See [Instrumenting Applications on Intel\(R\) 64 Architectures](#).
- **-pause\_tool n:** is a useful Pin-option which prints out the process id and pauses Pin for n seconds to permit attaching with gdb. See [Tips for Debugging a Pintool](#).
- **-follow\_execv:** Execute with Pin all processes spawned by `execv` class system calls.
- **-injection *mode*:** Where *mode* is one of dynamic, self, child, parent. *UNIX-only* See [Injection](#).

The tool-options follow immediately after the tool specification and depend on the tool used.

Everything following the **--** is the command line for the application.

For example, to apply the `itrace` example ([Instruction Address Trace \(Instruction Instrumentation\)](#)) to a run of the "ls" program:

```
../../../../pin -t obj-intel64/itrace.so -- /bin/ls
```

To get a listing of the available command line options for Pin:

```
pin -help
```

To get a listing of the available command line options for the `itrace` example:

```
../../../../pin -t obj-intel64/itrace.so -help -- /bin/ls
```

Note that in the last case `/bin/ls` is necessary on the command line but will not be executed.

### Instrumenting Applications on Intel(R) 64 Architectures

The Pin kit for IA-32 and Intel(R) 64 architectures is a combined kit. Both a 32-bit version and a 64-bit version of Pin are present in the kit. This allows Pin to instrument complex applications on Intel(R) 64 architectures which may have 32-bit and 64-bit components.

An application and a tool are invoked in "mixed-mode" as follows:

```
pin [pin-option]... -t64 <64-bit toolname> -t <32-bit toolname> [tool-options]...
-- <application> [application-option]..
```

Please note:

- The **-t64** option must precede the **-t** option.
- When using **-t64** together with **-t**, **-t** specifies the 32-bit tool. Using **-t64** without **-t** is not recommended, since in this case when given a 32-bit application, Pin will run the application without applying any tool.
- The **[tool-options]** apply to both the 64-bit and the 32-bit tools and **must** be specified **after -t <32-bit toolname>**. It is not possible to specify different set of options for the 64-bit and the 32-bit tools.

See `source/tools/Crossla32Intel64/makefile` for a few examples.

The file "pin" is a c-based launcher executable that expects the Pin binary "pinbin" to be in the architecture-specific "bin" subdirectory (i.e. intel64/bin). The "pin" launcher distinguishes the 32-bit version of the Pin binary from the 64-bit version of the Pin binary by using the **-p32/-p64** switches, respectively. Today, the 32-bit version of the Pin binary is invoked, and the path of the 64-bit version of Pin is passed as an argument using the **-p64** switch. However, one could change this to invoke the 64-bit version of the Pin binary, and pass the 32-bit version of the Pin binary as an argument using the **-p32** switch.

### Injection

The **-injection** switch is UNIX-only and controls the way pin is injected into the application process. The default, dynamic, is recommended for all users. It uses parent injection unless it is unsupported (Linux 2.4 kernels). Child injection creates the application process as a child of the pin process so you will see both a pin process and the application process running. In parent injection, the pin process exits after injecting the application and is less likely to cause a problem. Using parent injection on an unsupported platform may lead to nondeterministic errors.

**IMPORTANT:** The description about invoking assumes that the application is a program binary (and not a shell script). If your application is invoked indirectly (from a shell script or using 'exec') then you need to change the actual invocation of the program binary by prefixing it with pin/pintool options. Here's one way of doing that:

```
# Track down the actual application binary, say it is 'application_binary'.
% mv application_binary application_binary.real

# Write a shell script named 'application_binary' with the following contents.
# (change 'itrace' to your desired tool)

#!/bin/sh
../../../../pin -t obj-intel64/itrace.so -- application_binary.real $*
```

After you do this, whenever 'application\_binary' is invoked indirectly (from some shell script or using 'exec'), the real binary will get invoked with the right pin/pintool options.

### Restrictions

There is a known problem of using pin on systems protected by the "McAfee Host Intrusion Prevention" antivirus software. We did not test coexistence of pin with other antivirus products that perform run-time execution monitoring.

There is a known problem of using Pin on Linux systems that prevent the use of ptrace attach via the `sysctl /proc/sys/kernel/yama/ptrace_scope`. In this case Pin is not able to use its default (parent) injection mode. To resolve this, either execute the following (as root):

```
$ echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

Or use the "-injection child" option. For more information regarding child injection, See [Injection](#).

## Tips for Debugging a Pintool

### Using gdb on Linux

When running an application under the control of Pin and a Pintool there are two different programs residing in the address space. The application, and the Pin instrumentation engine together with your Pintool. The pintool is normally a shared object loaded by Pin. This section describes how to use gdb to find bugs in a Pintool. You cannot run Pin directly from gdb since Pin uses the debugging API to start the application. Instead, you must invoke Pin from the command line with the `-pause_tool` switch, and use gdb to attach to the Pin process from another window. The `-pause_tool n` switch makes Pin print out the process identifier (pid) and pause for n seconds.

Pin searches for the tool in an internal search algorithm. Therefore in many cases gdb is unable to load the debug info for the tool. There are several options to help gdb find the debug info.

Option 1 is to use full path to the tool when running pin.

Option 2 is to tell gdb to load the debugging information of the tool. Pin prompts with the exact gdb command to be used in this case.

To check that gdb loaded the debugging info to the tool use the command "info sharedlibrary" and you should see that gdb has read the symbols for your tool (as in the example below).

```
(gdb) info sharedlibrary
From      To          Syms Read  Shared Object Library
0x001b3ea0 0x001b4d80 Yes        /lib/libdl.so.2
0x003b3820 0x00431d74 Yes        /usr/intel/pkgs/gcc/4.2.0/lib/libstdc++.so.6
0x0084f4f0 0x00866f8c Yes        /lib/i686/libm.so.6
0x00df8760 0x00dffcc4 Yes        /usr/intel/pkgs/gcc/4.2.0/lib/libgcc_s.so.1
0x00e5fa00 0x00f60398 Yes        /lib/i686/libc.so.6
0x40001c50 0x4001367f Yes        /lib/ld-linux.so.2
0x008977f0 0x00af7784 Yes        ./dcache.so
```

For example, if your tool is called `opcodemix` and the application is `/bin/ls`, you can use gdb as described below. The following example is for the Intel(R) 64 Linux platform. Substitute "ia32" for the IA-32 architecture.

Change directory to the directory where your tool resides, and start gdb with pin, but do not use the run command.

```
$ /usr/bin/gdb ../../intel64/bin/pinbin
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host libthread_db library "/lib64/tls/libthread_db.so.1"
(gdb)
```

In another window, start your application with the `-pause_tool` switch.

```
$ ../../pin -pause_tool 10 -t obj-intel64/opcodemix.so -- /bin/ls
Pausing for 10 seconds to attach to process with pid 28769
To load the tool's debug info to gdb use:
  add-symbol-file .../source/tools/SimpleExamples/obj-intel64/opcodemix.so 0x2a959e9830
```

Then go back to gdb and attach to the process.

```
(gdb) attach 28769
Attaching to program: .../intel64/bin/pinbin, process 28769
0x000000314b38f7a2 in ?? ()
(gdb)
```

Now, you should tell gdb to load the Pintool debugging information, by copying the debugging message we got when invoking pin with the `-pause_tool` switch..

```
(gdb) add-symbol-file .../source/tools/SimpleExamples/obj-intel64/opcodemix.so 0x2a959e9830
add symbol table from file ".../source/tools/SimpleExamples/obj-intel64/opcodemix.so" at
      .text_addr = 0x2a959e9830
      (y or n) y
      Reading symbols from .../source/tools/SimpleExamples/obj-intel64/opcodemix.so...done.
(gdb)
```

Now, instead of using the `gdb run` command, you use the `cont` command to continue execution. You can also set breakpoints as normal.

```
(gdb) b opcodemix.cpp:447
Breakpoint 1 at 0x2a959ecf60: file opcodemix.cpp, line 447.
(gdb) cont
Continuing.

Breakpoint 1, main (argc=7, argv=0x3ff00f12f8) at opcodemix.cpp:447
447   int main(int argc, CHAR *argv[])
(gdb)
```

If the program does not exit, then you should detach so gdb will release control.

```
(gdb) detach
Detaching from program: .../intel64/bin/pinbin, process 28769
(gdb)
```

If you recompile your program and then use the run command, gdb will notice that the binary has been changed and reread the debug information from the file. This does not always happen automatically when using attach. In this case you must use the "add-symbol-file" command again to make gdb reread the debug information.

### Using the Visual Studio Debugger on Windows

When running an application under the control of Pin and a Pintool there are two different programs residing in the address space. The application, and the Pin instrumentation engine together with your Pintool. The pintool is a dynamically loaded library (.dll) loaded by Pin. This section describes how to use the Visual Studio Debugger to find bugs in a Pintool. You cannot run Pin directly from the debugger since Pin uses the debugging API to start the application. Instead, you must invoke Pin from the command line with the -pause\_tool switch, and use Visual Studio to attach to the Pin process from another window. The -pause\_tool n switch makes Pin print out the process identifier (pid) and pause for n seconds. You have n seconds (20 in our example) to attach the application with the debugger. Note, application resumes once the timeout expires. Attaching debugger later will not have the desired effect.

```
% pin <pin options> -pause_tool 20 -t <tool name> <tool options> -- <app name> <app options>
Pausing for 20 seconds to attach to process with pid 28769
```

In the Visual Studio window, attach to the application process using the "Debug"->"Attach to Process" menu selection and wait until a breakpoint occurs. Then you can set breakpoints in your tool in the usual way.

Note, it is necessary to build your pin tool with debug symbols if you want symbolic information.

### Logging Messages from a Pintool

Pin provides a mechanism to write messages from a Pintool to a logfile. To use this capability, call the LOG() API with your message. The default filename is pintool.log, and it is created in the currently working directory. Use the -logfile switch after the tool name to change the path and file name of the log file.

```
LOG( "Replacing function in " + IMG_Name(img) + "\n" );
LOG( "Address = " + hexstr( RTN_Address(rtn)) + "\n" );
LOG( "Image ID = " + decstr( IMG_Id(img) ) + "\n" );
```

### Performance Considerations When Writing a Pintool

The way a Pintool is written can have great impact on the performance of the tool, i.e. how much it slows down the applications it is instrumenting. This section demonstrates some techniques that can be used to improve tool performance. Let's start with an example. The following piece of code is derived from the source/tools/SimpleExamples/edgcnt.cpp:

The instrumentation component of the tool is show below

```
VOID Instruction(INS ins, void *v)
{
    ...

    if ( [ins is a branch or a call instruction] )
    {

        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                        IARG_INST_PTR,
                        IARG_BRANCH_TARGET_ADDR,
                        IARG_BRANCH_TAKEN,
                        IARG_END);
    }

    ...
}
```

The analysis component looks like this:

```
VOID docount2( ADDRINT src, ADDRINT dst, INT32 taken )
{
    if(!taken) return;
    COUNTER *pedg = Lookup( src, dst );
    pedg->_count++;
}
```

The purpose of the tool is to count how often each controlflow changing edge in the control flowgraph is traversed. The tool considers both calls and branches but for brevity we will not mention branches in our description. The tool works as follows: The instrumentation component instruments each branch with a call to docount2. As parameters we pass in the origin and the target of the branch and whether the branch was taken or not. Branch origin and target represent of the source and destination of the controlflow edges. If a branch is not taken the controlflow does not change and hence the analysis routine returns right away. If the branch is taken we use the src and dst parameters to look up the counter associated with this edge (Lookup will create a new one if this edge has not been seen before) and increment the counter. Note, that the tool could have been simplified somewhat by using IPOINT\_TAKEN\_BRANCH option with INS\_InsertCall().

### Shifting Computation for Analysis to Instrumentation Code

About every 5th instruction executed in a typical application is a branch. Lookup will called whenever these instruction are executed, causing significant application slowdown. To improve the situation we note that the instrumentation code is typically called only once for every instruction, while the analysis code is called everytime the instruction is executed. If we can somehow shift computation from the analysis code to the instrumentation code we will improve the overall performance. Our example tools offer multiple such opportunities which will explore in turn. The first observation is that for most branches we can find out inside of Instruction() what the branch target will be. For those branches we can call Lookup inside of Instruction() rather than in docount2(), for indirect branches which are relatively rare we still have to use our original approach. All this is reflected in the folling code. We add a second "lighter" analysis function, docount. While the original docount2() remains unchanged:

```
VOID docount( COUNTER *pedg, INT32 taken )
{
    if( !taken ) return;
    pedg->_count++;
}
```

And the instrumentation will be somewhat more complex:

```

VOID Instruction(INS ins, void *v)
{
    ...

    if (INS_IsDirectBranchOrCall(ins))
    {
        COUNTER *pedg = Lookup( INS_Address(ins),  INS_DirectBranchOrCallTargetAddress(ins) );

        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount,
                        IARG_ADDRINT, pedg,
                        IARG_BRANCH_TAKEN,
                        IARG_END);
    }
    else
    {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                        IARG_INST_PTR,
                        IARG_BRANCH_TARGET_ADDR,
                        IARG_BRANCH_TAKEN,
                        IARG_END);
    }

    ...
}

```

### Eliminating Control Flow

The code for docount() is very compact which provides performance advantages; it may also allow it to be inlined by Pin, thereby avoiding the overhead of a call. The heuristics for when a analysis routine is inlined by Pin are subject to change. But small routines without any control flow (single basic block) are almost guaranteed to be inlined. Unfortunately, docount() does have (albeit limited) control flow. Observing that the parameter, 'taken', will be zero or one we can eliminate the remaining control flow as follows:

```

VOID docount( COUNTER *pedg, INT32 taken )
{
    pedg->_count += taken;
}

```

Now docount() can be inlined.

### Compiler Considerations for Inlining

The way that the tool is built affects inlining as well. If an analysis routine has a function call to another function, it would not be a candidate for inlining by Pin unless the function call was inlined by the compiler. If the function call is inlined by the compiler, the analysis routine would be a candidate for inlining by Pin. Therefore, it is advisable to write any subroutines called by the analysis routine in a way that allows the compiler to inline the subroutines.

On Linux IA-32 architectures, Pin tools are built non-PIC (Position Independent Code), which allows the compiler to inline both local and global functions. Tools for Linux Intel(R) 64 architectures are built PIC, but the compiler will not inline any globally visible function due to function pre-emption. Therefore, it is advisable to declare the subroutines called by the analysis function as 'static' on Linux Intel(R) 64 architectures.

### Letting Pin Decide Where to Instrument

At times we do not care about the exact point where calls to analysis code are being inserted as long as it is within a given basic block. In this case we can let Pin make the decision where to insert. This has the advantage that Pin can select an insertion point that requires minimal register saving and restoring. The following code from ManualExamples/inscount2.cpp shows how this is done for the instruction count example using IPOINT\_ANYWHERE with **BBL\_InsertCall()**.

```

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

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every block
// Use the fast linkage for calls
VOID PIN_FAST_ANALYSIS_CALL docount(ADDRINT c) { icount += c; }

// Pin calls this function every time a new basic block is encountered
// It inserts a call to docount
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount for every bbl, passing the number of instructions.
        // IPOINT_ANYWHERE allows Pin to schedule the call anywhere in the bbl to obtain best performance.
        // Use a fast linkage for the call.
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, AFUNPTR(docount), IARG_FAST_ANALYSIS_CALL, IARG_UINT32, BBL_NumIns(bbl), IARG_END);
    }
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* ===== */
/* Print Help Message */
/* ===== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
}

```

```

    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===== */
/* Main                                           */
/* ===== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    TRACE_AddInstrumentFunction(Trace, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

### Using Fast Call Linkages

For very small analysis functions, the overhead to call the function can be comparable to the work done in the function. Some compilers offer optimized call linkages that eliminate some of the overhead. For example, gcc for the IA-32 architecture has a `regparm` attribute for passing arguments in registers. Pin supports a limited number of alternate linkages. To use it, you must annotate the declaration of the analysis function with **PIN\_FAST\_ANALYSIS\_CALL**. The `InsertCall` function must pass **IARG\_FAST\_ANALYSIS\_CALL**. If you change one without changing the other, the arguments will not be passed correctly. See the `inscount2.cpp` example in the previous section for a sample use. For large analysis functions, the benefit may not be significant, but it is unlikely that **PIN\_FAST\_ANALYSIS\_CALL** would ever cause a slowdown.

Another call linkage optimization is to eliminate the frame pointer. We recommend using `-fomit-frame-pointer` to compile tools with gcc. See the gcc documentation for an explanation of what it does. The standard Pintool makefiles include `-fomit-frame-pointer`. Like **PIN\_FAST\_ANALYSIS\_CALL**, the benefit is largest for small analysis functions. Debuggers rely on frame pointers to display stack traces, so eliminate this option when trying to debug a PinTool. If you are using a standard PinTool makefile, you can do this by overriding the definition of `OPT` on the command line with

```
make OPT=-O0
```

### Rewriting Conditional Analysis Code to Help Pin Inline

Pin improves instrumentation performance by automatically inlining analysis routines that have no control-flow changes. Of course, many analysis routines do have control-flow changes. One particularly common case is that an analysis routine has a single "if-then" test, where a small amount of analysis code plus the test is always executed but the "then" part is executed only once a while. To inline this common case, Pin provides a set of conditional instrumentation APIs for the tool writer to rewrite their analysis routines into a form that does not have control-flow changes. The following example from `source/tools/ManualExamples/isampling.cpp` illustrates how such rewriting can be done:

```

/*
 * This file contains a Pintool for sampling the IPs of instruction executed.
 * It serves as an example of a more efficient way to write analysis routines
 * that include conditional tests.
 * Currently, it works on IA-32 and Intel(R) 64 architectures.
 */

#include <stdio.h>
#include <stdlib.h>
#include "pin.H"

FILE * trace;

const INT32 N = 100000;
const INT32 M = 50000;

INT32 icount = N;

/*
 * IP-sampling could be done in a single analysis routine like:
 *
 * VOID IpSample(VOID *ip)
 * {
 *     --icount;
 *     if (icount == 0)
 *     {
 *         fprintf(trace, "%p\n", ip);
 *         icount = N + rand() % M;
 *     }
 * }
 *
 * However, we break IpSample() into two analysis routines,
 * CountDown() and PrintIp(), to facilitate Pin inlining CountDown()
 * (which is the much more frequently executed one than PrintIp()).
 */

ADDRINT CountDown()
{
    --icount;
    return (icount==0);
}

// The IP of the current instruction will be printed and
// the icount will be reset to a random number between N and N+M.
VOID PrintIp(VOID *ip)
{
    fprintf(trace, "%p\n", ip);

    // Prepare for next period
    icount = N + rand() % M; // random number from N to N+M
}

```

```

}

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // CountDown() is called for every instruction executed
    INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)CountDown, IARG_END);

    // PrintIp() is called only when the last CountDown() returns a non-zero value.
    INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR)PrintIp, IARG_INST_PTR, IARG_END);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace, "#eof\n");
    fclose(trace);
}

/* ===== */
/* Print Help Message                                */
/* ===== */

INT32 Usage()
{
    PIN_ERROR( "This Pintool samples the IPs of instruction executed\n"
               + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===== */
/* Main                                              */
/* ===== */

int main(int argc, char * argv[])
{
    trace = fopen("isampling.out", "w");

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}

```

In the above example, the original analysis routine `IpSample()` has a conditional control-flow change. It is rewritten into two analysis routines: `CountDown()` and `PrintIp()`. `CountDown()` is the simpler one of the two, which doesn't have control-flow change. It also performs the original conditional test and returns the test result. We use the conditional instrumentation APIs `INS_InsertIfCall()` and `INS_InsertThenCall()` to tell Pin that the analysis routine specified by an `INS_InsertThenCall()` (i.e. `PrintIp()` in this example) is executed only if the result of the analysis routine specified by the previous `INS_InsertIfCall()` (i.e. `CountDown()` in this example) is non-zero. Now `CountDown()`, the common case, can be inlined by Pin, and only once a while does Pin need to execute `PrintIp()`, the non-inlined case.

### Optimizing Instrumentation of REP Prefixed Instructions

The IA-32 and Intel(R) 64 architectures include REP prefixed string instructions. These use a REP prefix on a string operation to repeat the execution of the inner operation. For some instructions the repeat count is determined solely by the value in the count register. For others (SCAS, CMPS), the count register provides an upper limit on the number of iterations, while the REP opcode provides a condition to be tested which can exit the REP loop before the full number of iterations has been executed.

Pin treats REP prefixed instructions as an implicit loop around the inner instruction, so `IPOINT_BEFORE` and `IPOINT_AFTER` instrumentation is executed for that instruction once for each iteration of the (implicit) loop. Since each execution of the inner instruction is instrumented, `IARG_MEMORY{READ,READ2,WRITE}_SIZE` can be determined statically from the instruction (1,2,4,8 bytes), and `IARG_MEMORY{OP.READ,READ2,WRITE}_EA` can also be determined (even if `DF==1`, so the inner instructions are decrementing their arguments and moving backwards through store).

REP prefixed instructions are treated as predicated, where the predicate is that the count register is non-zero. Therefore canonical instrumentation for memory accesses such as

```

if (INS_MemoryOperandIsRead(ins, memOp))
{
    INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)logMemory,
                           IARG_MEMORYOP_EA, memOp,
                           IARG_UINT32, INS_MemoryOperandSize(ins, memOp),
                           IARG_END);
}

```

will see all of the memory accesses made by the REP prefixed operations.

To allow tools to count entries into a REP prefixed instruction, and to optimize, Pin provides `IARG_FIRST_REP_ITERATION`, which can be passed as an argument to an analysis routine. It is TRUE if this is the first iteration of a REP prefixed instruction, FALSE otherwise.

Thus to perform an action only on the first iteration of a REP prefixed instruction, one can use code like this (assuming that "takeAction" wants to be called on the first iteration of all REP prefixed instructions, even ones with a zero repeat count):

To obtain the repeat count, you can use

```
IARG_REGISTER_VALUE, INS_RepCountRegister(ins),
```

which will pass the value in the appropriate count register (one of `REG_CX`, `REG_ECX`, `REG_RCX` depending on the instruction).

As an example, here is code which counts the number of times REP prefixed instructions are executed, optimizing cases in which the REP prefixed instruction only depends on the count register.

```

class stats
{
    UINT64 count;                // Times we start the REP prefixed op
    UINT64 repeatedCount;        // Times we execute the inner instruction
    UINT64 zeroLength;           // Times we start but don't execute the inner instruction because count is zero
public:
    stats() : count(0), repeatedCount(0), zeroLength(0) {}
    VOID output() const;
    VOID add(UINT32 firstRep, UINT32 repCount)
    {
        count += firstRep;
        repeatedCount += repCount;
        if (repCount == 0)
            zeroLength += 1;
    }
    BOOL empty() const { return count == 0; }
    stats& operator+= (const stats &other)
    {
        count += other.count;
        repeatedCount += other.repeatedCount;
        zeroLength += other.zeroLength;
        return *this;
    }
};

// Trivial analysis routine to pass its argument back in an IfCall so that we can use it
// to control the next piece of instrumentation.
static ADDRINT returnArg (BOOL arg)
{
    return arg;
}

// Analysis functions for execution counts.
// Analysis routine, FirstRep and Executing tell us the properties of the execution.
static VOID addCount (UINT32 opIdx, UINT32 firstRep, UINT32 repCount)
{
    stats * s = &statistics[opIdx];

    s->add(firstRep, repCount);
}

// Instrumentation routines.
// Insert code for counting how many times the instruction is executed
static VOID insertRepExecutionCountInstrumentation (INS ins, UINT32 opIdx)
{
    if (takesConditionalRep(opIdx))
    {
        // We have no smart way to lessen the number of
        // instrumentation calls because we can't determine when
        // the conditional instruction will finish. So we just
        // let the instruction execute and have our
        // instrumentation be called on each iteration. This is
        // the simplest way of handling REP prefixed instructions, where
        // each iteration appears as a separate instruction, and
        // is independently instrumented.
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)addCount,
                      IARG_UINT32, opIdx,
                      IARG_FIRST_REP_ITERATION,
                      IARG_EXECUTING,
                      IARG_END);
    }
    else
    {
        // The number of iterations is determined solely by the count register value,
        // therefore we can log all we need at the start of each REP "loop", and skip the
        // instrumentation on all the other iterations of the REP prefixed operation. Simply use
        // IF/THEN instrumentation which tests IARG_FIRST_REP_ITERATION.
        //
        INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)returnArg, IARG_FIRST_REP_ITERATION, IARG_END);
        INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR)addCount,
                          IARG_UINT32, opIdx,
                          IARG_UINT32, 1,
                          IARG_REG_VALUE, INS_RepCountRegister(ins),
                          IARG_END);
    }
}

```

To perform this optimization when collecting memory access addresses, you will also need to worry about the state of EFLAGS.DF, since the string operations work from high address to low address when EFLAGS.DF==1.

Here is an example which shows how to handle that.

```

// Compute the base address of the whole access given the initial address,
// repeat count and element size. It has to adjust for DF if it is asserted.
static ADDRINT computeEA (ADDRINT firstEA, UINT32 eflags, UINT32 count, UINT32 elementSize)
{
    enum {
        DF_MASK = 0x0400
    };

    if (eflags & DF_MASK)
    {
        ADDRINT size = elementSize*count;

        // The string ops post-decrement, so the lowest address is one elementSize above
        // where you might think it should be.
        return firstEA - size + elementSize;
    }
    else
        return firstEA;
}

static VOID logMemoryAddress (UINT32 op,          // Index of instruction

```



```

        BOOL first,           // First iteration?
        ADDRINT baseEA,      // Effective address being accessed on this iteration
        ADDRINT count,       // Iteration count
        UINT32 size,         // Size in bytes of the per-iteration access
        UINT32 eflags,       // Eflags
        ADDRINT tag)         // Name for the type of access
{
    const char * tagString = reinterpret_cast<const char *>(tag);
    UINT32 width = 20;

    if (!first)
    {
        out << " ";           // Indent REP iterations
        width -= 2;
    }
    out << opcodes[op].name << ' ' << tagString << ' ';
    out << std::hex << std::setw(width) << computeEA(baseEA, eflags, count, size) << ':';
    out << std::dec << std::setw(20) << size*count << endl;
}

// Insert instrumentation to log memory addresses accessed.
static VOID insertRepMemoryTraceInstrumentation(INS ins, UINT32 opIdx)
{
    const opInfo * op = &opcodes[opIdx];

    if (takesConditionalRep(opIdx))
    {
        if (INS_IsMemoryRead(ins))
        {
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) logMemoryAddress,
                          IARG_UINT32, opIdx,
                          IARG_FIRST_REP_ITERATION,
                          IARG_MEMORYREAD_EA,
                          IARG_EXECUTING,
                          IARG_UINT32, op->size,
                          IARG_UINT32, 0, // Fake Eflags, since we're called at each iteration it doesn't matter
                          IARG_ADDRINT, (ADDRINT)"Read ",
                          IARG_END);
        }
        // And similar code for MEMORYREAD2, MEMORYWRITE
    }
    else
    {
        if (INS_IsMemoryRead(ins))
        {
            INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR) returnArg, IARG_FIRST_REP_ITERATION, IARG_END);
            INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR) logMemoryAddress,
                              IARG_UINT32, opIdx,
                              IARG_BOOL, TRUE, // First must be TRUE else we wouldn't be called
                              IARG_MEMORYREAD_EA,
                              IARG_REG_VALUE, INS_RepCountRegister(ins),
                              IARG_UINT32, op->size,
                              IARG_REG_VALUE, REG_EFLAGS,
                              IARG_ADDRINT, (ADDRINT)"Read ",
                              IARG_END);
        }
        // And similar code for MEMORYREAD2, MEMORYWRITE
    }
}
}

```

Since there are real codes where a significant proportion of all instructions are REP prefixed, using **IARG\_FIRST\_REP\_ITERATION** to collect information at the beginning of the REP "loop" while skipping it for the later iterations can be a significant optimization.

A tool which demonstrates all of these techniques can be found in source/tools/ManualExamples/countreps.cpp, from which these (slightly edited) code snippets were taken.

## Memory management

Pin allows the Pin tool to dynamically allocate memory (e.g. using malloc()) without interfering with the execution of the application that is run under Pin. In order to achieve this, Pin implements its own memory allocator which is separate from the application's memory allocator, and allocates memory in different memory regions.

### Pin's dynamic memory allocation regions

By default, the memory address region used by Pin to dynamically allocate memory for both Pin usage and Pin tool usage is unrestricted. However, if Pin memory allocation should be restricted to specific memory regions, the `-pin_memory-range` knob can be used in Pin's command line to make Pin allocate memory only inside the specified regions. Note that restricting Pin memory allocation to specific regions doesn't mean that it will allocate/reserve the entire memory available those regions!

#### the maximum memory that Pin can allocate

Pin can be forced to limit the amount of memory it can allocate (in bytes) by using the `-pin_memory_size` knob in Pin's command line. When a Pin tool cannot allocate more memory due to `-pin_memory_size` limitation, its out of memory callback is called (see **PIN\_AddOutOfMemoryFunction()**). By default, the number of bytes that Pin can allocate is unlimited. We recommend that if a memory limitation is specified, it will be at least 30MB.

### mode

In JIT mode, Pin needs to manage memory for the code cache in addition to the dynamically allocated memory. This means that the memory regions specified by `-pin_memory-range` restricts both the dynamically allocated memory and the code cache blocks allocated by Pin.

In order to limit the code cache memory allocation, one can specify the `-cc_memory_size` knob in Pin's command line. Note that the specified limit must be a multiple of the code cache block size (specified with `-cache_block_size`).

### Pin

Another component that requires memory while running Pin on an application is the images of Pin, tool, and their shared libraries (aka dynamic link libraries). The shared libraries are usually built as position independent code and hence can be loaded anywhere in the memory space of the Pined process. On the other hand, the Pin executable is currently built as position independent code only on Linux. Note that Pin executable is injected to the Pined process memory space on the



operating systems: OS X\*, Android, and Linux. This means that on OS X\* and Android Pin executable must be injected to a fixed address in the memory space and cannot be relocated.

In order to restrict the memory that Pin image loader will use (while considering the limitation stated above), one can use the `-restrict_memory` knob in Pin's command line to specify memory region where Pin loader should not use. Note that the logic of the `-restrict_memory` knob is reversed from all the other memory range knobs for Pin - as it specifies which memory regions the Pin loader should \*NOT\* use.

## Pintool Information and Restrictions

### General

There are several things that a Pintool writer must be aware of.

- pthread functions cannot be called from an analysis or replacement routine ([Instrumenting Multi-threaded Applications](#))
- `IARG_REG_VALUE` cannot be used to pass floating point register values to an analysis routine.
- Also, see the OS-specific restrictions below. [Windows OS](#) or [Linux OS](#)

Often, a Pintool writer wants to run the SPEC benchmarks to see the results of their research. There are many ways one can update the scripts to invoke Pin on the SPEC tests; this is one. In your `$SPEC/config` file, add the following two lines:

```
submit=$PIN_HOME/intel64/bin/pin -t /my/pin/tool -- $command
use_submit_for_speed=yes
```

Now the SPEC harness will automatically run Pin with whatever benchmarks it runs. Note that you need the full path name for Pin and Pintool binaries. Replace "intel64" with "ia32" if you are using a 32-bit system.

### Linux OS

Pin identifies system calls at the actual system call trap instruction, not the libc function call wrapper. Tools need to be aware of oddities like this when interpreting system call arguments, etc.

### Windows OS

On Windows, Pin has been compiled with `/D_SECURE_SCL=0`. Pintools must also be compiled with `/D_SECURE_SCL=0`, otherwise the STL containers shared between Pin and the Pintool can have different memory layouts.

Pin on Windows guarantees safe usage of C/C++ run-time services in Pin tools, including indirect calls to Windows API through C run-time library. Any other use of Windows API in Pin tool is not guaranteed to be safe:

- reentrant use of shared system resources may cause crashes and lost of transparency
- a tool that directly or indirectly locks shared resource by calling to a system API under Pin lock may cause deadlock
- using alertable system calls or installing handlers of asynchronous system events in a tool may violate the logic of the application, cause recursive invocation of instrumentation/analysis callbacks in the tool.

Pin on Windows does not separate DLLs loaded by the tool from the application DLLs - it uses the same system loader. In order to avoid isolation problems, Pin tool should not load any DLL that can be shared with the application. For the same reason, Pin tool should avoid static links to any common DLL, except for those listed in `PIN_COMMON_LIBS` (see `source.flags` file).

In probe mode, the application runs natively, and the probe is placed in the original code. If a tool replaces a function shared by the tool and the application, an undesirable behavior may occur. For example, if a tool replaces `EnterCriticalSection()` with an analysis routine that calls `printf()`, this could result in an infinite loop, because `printf()` can also call `EnterCriticalSection()`. The application would call `EnterCriticalSection()`, and the control flow would go to the replacement routine, and it would call `EnterCriticalSection()` (via `printf`) which would call the replacement routine, and so on.

### Conflicts between Pin and Windows

Pin uses some base types that conflict with Windows types. If you use "windows.h", you may see compilation errors. To avoid this problem, we recommend wrapping the windows.h file as follows. Items that reside in the windows.h file must be referenced using the `WINDOWS::` prefix.

```
namespace WINDOWS
{
#include <windows.h>
}
```

## Building Tools on windows

### Building Tools in Visual Studio

- In order to use pin kit VC9, you must have Visual studio 2008 installed on your computer.
- In order to use pin kit VC10, you must have Visual studio 2010 installed on your computer.
- In order to use pin kit VC11, you must have Visual studio 2012 installed on your computer.

According to the Visual Studio version an example of visual studio project that builds pin tool in the Visual Studio IDE will be placed in the directory. Enter this directory and open `MyPinTool.vcproj` project or `MyPinTool.sln` solution. To build the tool, select "Build Solution".

To run an application, instrumented by MyPinTool, select Tool->External Tools. In the "Menu contents" window choose "run pin". Add to the "Arguments" box the path of the required application that you want to run with pin. For example: `-t MyPinTool.dll -count 1 -- "C:\Users\...\my_app.exe"` and select "OK". A Popup window may appear on the screen with the following message: "The command is not a valid executable. Would you like to change the command?" select "No". To start running your application select Tool->pin run.

You can select another application and change tool's switches in the "MyPinTool Properties->Debugging" page.

You can use MyPinTool as a template for your own project. Please, look carefully at the compilation and linking switches in the MyPinTool property pages. Mandatory switches can be found in the `win.vars` file in the kit's `source/tools/Config` directory. Also note the library order, as this is important, too. See [Pin's makefile Infrastructure](#) for further details.

### Constructing PinTools from multiple DLLs on Windows

A Pin tool can be composed from multiple DLLs:

- "main DLL", which is specified in the Pin command line after "-t" switch
- a number of "secondary DLLs", linked to the "main DLL" statically.

When considering this configuration, take into account that multi-DLL Pin tool may increase memory fragmentation and cause layout conflicts with application images. If there is no compelling reasons for using multiple DLLs, build your tool as a single DLL to reduce the risk of memory conflicts.

Limitations and instructions:

- Don't use any Pin API in "secondary DLLs". Only "main DLL" can use Pin API!
- In order to run Pin tool put "main DLL" and its "secondary DLLs" in the same directory.
- **IMPORTANT:** Build each DLL with the recommended pin tool building flags (see [Building Tools in Visual Studio](#)).
- Remove /EXPORT:main link flag for "secondary DLLs".
- Specify different base address for each DLL (/BASE link flag). When choosing base addresses, try to minimize memory fragmentation and layout conflicts.

### Supported executables

Pin can instrument Windows\* subsystem executables.

It can't instrument other executables (such as MS-DOS, Win16 or a POSIX subsystem executables).

### Libraries for Windows

Pin on Windows uses dbghelp.dll by Microsoft\* to provide symbolic information. dbghelp.dll version 6.11.1.404 is distributed with the kit. Please use the provided version, as other versions may not work properly with Pin.

### Libraries for Linux

#### Introduction

The Linux version of Pin requires the following shared objects from glibc:

```
libc
libm
libdl
libpthread
ld-linux*
```

and compiler c++ runtime libraries:

```
libstdc++
libgcc_s
```

and the following additional libraries:

```
libelf
libdwwarf
```

Pin supplies all the above libraries for running in JIT mode. For running in probe mode, see [How To Install Libraries If You Use Probe-mode](#) below.

#### The "pin" Executable (Launcher)

The kit's root directory contains a "pin" executable. This is a 32-bit launcher, used for launching Pin in 32 and 64 bit modes. The launcher sets up the environment to find the libraries supplied with the kit. The kit's runtime directories will be searched first, followed by directories that are on the LD\_LIBRARY\_PATH. The launcher will then invoke the actual Pin executable - "pinbin".

On 64-bit systems, the 32-bit glibc package must be installed in order to run the "pin" launcher. An alternative is to invoke the "pin.sh" script instead of the "pin" executable.

If you need to change the directory structure or copy pin to a different directory, then you should note the following. The "pin" launcher expects the binary "pinbin" to be in the architecture-specific "bin" subdirectory (e.g. ia32/bin). The launcher expects the libraries to be found in the architecture-specific "runtime" and subdirectory (i.e. ia32/runtime). If you need a different directory structure, you need to build your own launcher or find a different way to set up the environment to allow the pinbin executable to find the necessary runtime libraries. The pinbin binary itself makes no assumptions about the directory structure. The launcher's sources may be found in <kit root>="/>source/launcher.

#### Linux Compiler Requirements

Pin tools on Linux are shared objects loaded by pin before the application starts. Pin and the tool share the runtime libraries therefore the tool must be compiled with a compiler compatible with Pin. Pin is compiled by GCC 4.4.7 and is tested regularly with tools compiled with various GCC versions ranging from version 3.4.6 to version 4.7.2.

JIT-mode tools may be compiled with any version of GCC. Probe-mode tools may be compiled with versions up to 4.4.X inclusive.

#### How To Install Libraries If You Use Probe-mode

If you are using probe mode, you cannot use the standard glibc on your system. You need to install pin-probe-runtime.tar.gz. This file can be downloaded from <http://www.pintool.org>. Unpack the probe runtime tarball in the kit's root directory. The necessary files will be automatically extracted in the proper directory structure.

```
tproj> tar xzf pin-probe-runtime.tar.gz
```

The pin launcher (described in the section above) expects to find glibc in the architecture-specific "runtime" subdirectory (e.g. ia32/runtime), so be careful about rearranging files and directories.

#### How to build tools for Probe-mode on Linux

Backward compatibility in libraries in Linux is achieved by using version-symbols. New versions of glibc and the standard C++ runtime libraries have backward compatibility. The pin-probe-runtime includes glibc 2.3.4 and a version of C++ runtime libraries that can be used with this version of glibc.

The simplest way to be compatible with this version of glibc is to install red hat el4 or one of its clones (e.g. centos) and use the system compiler. It is also possible to build a tool on newer Linux OS releases, as long as the tool does not reference symbols that are not in glibc 2.3.4. If there is a problem, you will get an error message about not being able to find symbols when starting pin.

#### Packaging Pintools For a Binary Distribution

If you are distributing pin tools in binary form, then we suggest that you preserve the layout of the kit's runtime directories.

The "ia32" subdirectory is used for the IA-32 architecture. Similarly, "intel64" is used for the Intel(R) 64 architecture.

## Installing Pin

Each kit contains Pin and libraries for a specific architecture. Make sure the kit you download is for the right architecture. The Pin libraries use C++, and the compiler you use to build the tool must be compatible with the Pin library. This restriction only applies to building tools; you can instrument applications built by any compiler.

See the README file in the kit for specific information about compiler version and other limitations. If your compiler is not compatible with the kit, send mail to [pinheads@yahogroups.com](mailto:pinheads@yahogroups.com).

To install a kit, unpack a kit and change to the directory.

Linux / OS X\* / Android:

```
$ tar xzf pin-2.12-56759-gcc.4.4.7-linux.tar.gz
$ cd pin-2.12-56759-gcc.4.4.7-linux
```

Use the OS X\* / Android kit names respectively.

Windows: Unzip the installation files, extracting all files in the kit.

```
$ cd pin-2.12-56759-msvc10-windows.zip
```

Use the kit for the Visual Studio version you're using.

## Usage Instructions for Intel(R) Xeon Phi(TM)

### Table of Contents

- [Installing Pin on the Device](#)
- [Building and Installing an Entire Test Directory](#)
- [Building and Installing a Single Tool](#)
- [Running All the Tests in a Directory](#)
- [Running a Single Test on the Device - Launched from the Host](#)
- [Running a Single Test on the Device - Launched from the Device](#)

### Installing Pin on the Device

- Copy the mic-install.tar.gz tarball (located at the kit's root) to your preferred location on the device and unpack it. This will create a directory structure similar to that found on the host.
- Make sure the following libraries are present on the device (copy manually if necessary):  
libimf.so libintlc.so libintlc.so.5 libirng.so libsvml.so

### Building and Installing an Entire Test Directory

Perform the following on the host:

```
cd source/tools/ManualExamples
make CC=icc CXX=icpc TARGET=mic REMOTE_DEVICE=mic0 REMOTE_ROOT=pin install
```

Notes:

- The above will build all the tools and test applications in the ManualExamples library and pack them in a tarball. Then, the tarball will be copied to the device and will be unpacked.
- The device is identified by the REMOTE\_DEVICE flag.
- The install location on the device is specified by the REMOTE\_ROOT flag. REMOTE\_ROOT should be the same location where the mic-install.tar.gz tarball was unpacked.
- The makefile will automatically add the "-mmic" directive to the compilation flags when TARGET=mic is specified.

### Building and Installing a Single Tool

Compile the tool on the host as follows:

```
make CC=icc CXX=icpc TARGET=mic obj-mic/MyPinTool.so
```

Copy the tool to your preferred location on the device.

Notes:

- The makefile will automatically add the "-mmic" directive to the compilation flags when TARGET=mic is specified.

### Running All the Tests in a Directory

Perform the following on the host:

```
cd source/tools/ManualExamples
make TARGET=mic REMOTE_DEVICE=mic0 REMOTE_ROOT=pin test
```

Notes:

- The above will run all tests in the ManualExamples library.
- The directory should be installed prior to running the tests (follow the directions in [Building and Installing an Entire Test Directory](#)).
- See [Building and Installing an Entire Test Directory](#) for details of the flags used in the make command.

### Running a Single Test on the Device - Launched from the Host

Copy the tool, application and any other dependencies of the test (as they appear in the makefile) to the device. Make sure to keep the same directory structure as on the host (this is expected by the makefile). Run the following:

```
cd source/tools/ManualExamples
make REMOTE_DEVICE=mic0 REMOTE_ROOT=pin inscount0.wrap
```

Notes:

- The ".wrap" suffix should be used instead of the ".test" suffix. This will result in running the ".test" make target on the device. The ".wrap" make target is automatically generated by Pin's make configuration. Its purpose is to set the environment for launching the test on the device.
- See [Building and Installing an Entire Test Directory](#) for details of the flags used in the make command.

### Running a Single Test on the Device - Launched from the Device

Copy Pin, tool and application to the device. Then simply log in to the device and launch Pin regularly.

## Building Your Own Tool

### Table of Contents

- [Building a Tool From Within the Kit Directory Tree](#)
- [Building a Tool Out of the Kit Directory Tree](#)

To write your own tool, copy one of the example directories and edit the makefile.rules file to add your tool. The sample tool MyPinTool is recommended. This tool allows you to build either inside or outside the kit directory tree. See [Adding Tests, Tools and Applications to the makefile](#) and [Defining Build Rules for Tools and Applications](#) for further details on makefile modification.

### Building a Tool From Within the Kit Directory Tree

You may either modify MyPinTool or copy it as directed above. If you're using MyPinTool, and the default build rule suffices, you may not have to change makefile.rules. If you are adding a new tool, or you require special build flags for your tool, you will need to modify the makefile.rules file to add your tool and/or specify a customized build rule.

Building YourTool.so (from YourTool.cpp):

```
make obj-intel64/YourTool.so
```

For the IA-32 architecture, use "obj-ia32" instead of "obj-intel64". See for commonly used make flags to add to your build.

### Building a Tool Out of the Kit Directory Tree

Copy the MyPinTool directory to a place of your choosing. This directory will serve as a basis for your tool. Modify the makefile.rules file to add your tool and/or specify a customized build rule.

Building YourTool.so (from YourTool.cpp):

```
make PIN_ROOT=<path to Pin kit> obj-intel64/YourTool.so
```

For the IA-32 architecture, use "obj-ia32" instead of "obj-intel64". See for commonly used make flags to add to your build.

For changing the directory where the tool will be created, override the OBJDIR variable from the command line:

```
make PIN_ROOT=<path to Pin kit> OBJDIR=<path to output dir> <path to output dir>/YourTool.so
```

## Pin's makefile Infrastructure

### Table of Contents

- [Using Pin's makefile Infrastructure](#)
- [The Config Directory](#)
- [The Test Directories](#)
- [Adding Tests, Tools and Applications to the makefile](#)
- [Defining Build Rules for Tools and Applications](#)
- [Defining Test Recipes in makefile.rules](#)
- [Useful make Variables and Flags](#)

### Using Pin's makefile Infrastructure

Pintools are built using make on all target platforms. This section describes the basic flags available in Pin's makefile infrastructure. This is not a makefile tutorial. For general information about makefiles, refer to the makefile manual available at <http://www.gnu.org/software/make/manual/make.html>.

### The Config Directory

The source/tools/Config directory holds the common make configuration files which should not be changed and template files which may serve as a basis for your own makefiles. This sections gives a short overview of the most notable files in the directory. The experienced user is welcome to read through the complete set of configuration files for better understanding the tools' build process.

**makefile.config:** This is the first file to be included in the make include chain. It holds documentation of all the relevant flags and variables available to users, both within the makefile and from the command shell. Also, this file includes the OS-specific configuration files.

**makefile.unix.config:** This file holds the Unix definitions of the makefile variables. See **makefile.win.config** for the Windows definitions.

**unix.vars:** This file holds the Unix definitions of some architectural variables and utilities used by the makefiles. See **win.vars** for the Windows definitions.

**makefile.default.rules:** This file holds the default make targets, test recipes and build rules.

### The Test Directories

Each test directory in source/tools/ contains two files in the makefile chain.

**makefile:** This is the makefile which will be invoked when running make. This file should not be changed. It holds the include directives for all the relevant configuration files of the makefile chain in the correct order. Changing this order may result in unexpected behavior. This is a generic file, it is identical in all test directories.

**makefile.rules:** This is the directory-specific makefile. It holds the logic of the current directory. All tools, applications and tests that should be built and run in a directory are defined in this file. See [Adding Tests, Tools and Applications to the makefile](#) for adding tests, tools and applications to makefile.rules.

### Adding Tests, Tools and Applications to the makefile

This section describes how to define your applications, tools and tests in the makefile. The sections below describe how to build the binaries and how to run the

tests.

The variables detailed below, hold the tests, applications and tools definitions. They are defined in the "Test targets" section of makefile.rules. See this section for additional variables and more detailed documentation for each variable.

**TOOL\_ROOTS:** Define the name of your tool here, without the file extension. The correct extension, according to the OS, will be added automatically by make. For example, for adding YourTool.so:

```
TOOL_ROOTS := YourTool
```

**APP\_ROOTS:** Define your application here, without the file extension. The correct extension according to the OS, will be added automatically by make. For example, for adding YourApp.exe:

```
APP_ROOTS := YourApp
```

**TEST\_ROOTS:** Define your tests here without the .test suffix. This suffix will be added automatically by make. For example, for adding YourTest.test:

```
TEST_ROOTS := YourTest
```

## Defining Build Rules for Tools and Applications

Default build rules for tools and applications are defined in source/tools/Config/makefile.default.rules. The default tool requires a single c/cpp source file and will generate a tool of the same name. For example, for YourTool.cpp make will generate YourTool.so with the default build rule. However, if your tool requires more than one source file, or you need a customized build rule, add your rule at the bottom of makefile.rules in the "Build rules" section". There is no need to add the \$(OBJDIR) dependency to the build rule, it will be added automatically. This dependency creates the build output directory obj-intel64 (or obj-ia32 for the IA-32 architecture). See source/tools/Config/makefile.config for all available compilation and link flags.

Here are a few useful examples:

Building an unoptimized tool from a single source:

```
# Build the intermediate object file.
$(OBJDIR) YourTool$(OBJ_SUFFIX): YourTool.cpp
    $(CXX) $(TOOL_CXXFLAGS_NOOPT) $(COMP_OBJ) $@ $<

# Build the tool as a dll (shared object).
$(OBJDIR) YourTool$(PINTOOL_SUFFIX): $(OBJDIR) YourTool$(OBJ_SUFFIX)
    $(LINKER) $(TOOL_LDFLAGS_NOOPT) $(LINK_EXE) $@ $< $(TOOL_LPATHS) $(TOOL_LIBS)
```

Building an optimized tool from several source files:

```
# Build the intermediate object file.
$(OBJDIR) Source1$(OBJ_SUFFIX): Source1.cpp
    $(CXX) $(TOOL_CXXFLAGS) $(COMP_OBJ) $@ $<

# Build the intermediate object file.
$(OBJDIR) Source2$(OBJ_SUFFIX): Source2.c Source2.h
    $(CC) $(TOOL_CXXFLAGS) $(COMP_OBJ) $@ $<

# Build the tool as a dll (shared object).
$(OBJDIR) YourTool$(PINTOOL_SUFFIX): $(OBJDIR) Source1$(OBJ_SUFFIX) $(OBJDIR) Source2$(OBJ_SUFFIX) Source2.h
    $(LINKER) $(TOOL_LDFLAGS_NOOPT) $(LINK_EXE) $@ $(^:%.h=) $(TOOL_LPATHS) $(TOOL_LIBS)
```

## Defining Test Recipes in makefile.rules

A default test recipe is defined in source/tools/Config/makefile.default.rules. For most users, this recipe is insufficient. You may specify your own test recipes in makefile.rules in the "Test recipes" section. There is no need to add the \$(OBJDIR) dependency to the build rule, it will be added automatically. This dependency creates the build output directory obj-intel64 (or obj-ia32 for the IA-32 architecture).

Example:

```
YourTest.test: $(OBJDIR) YourTool$(PINTOOL_SUFFIX) $(OBJDIR) YourApp$(EXE_SUFFIX)
    $(PIN) -t $< -- $(OBJDIR) YourApp$(EXE_SUFFIX)
```

## Useful make Variables and Flags

For a complete list of all the available variables and flags, see source/tools/Config/makefile.config . Here is a short list of the most useful flags:

**PIN\_ROOT:** Specify the location for the Pin kit when building a tool outside of the kit.

**CC:** Override the default c compiler for tools.

**CXX:** Override the default c++ compiler for tools

**APP\_CC:** Override the default c compiler for applications. If not defined, APP\_CC will be the same as CC.

**APP\_CXX:** Override the default c++ compiler for applications. If not defined, APP\_CXX will be the same as CXX.

**TARGET:** Override the default target architecture e.g. for cross-compilation.

**ICC:** Specify ICC=1 when building tools with the Intel Compiler.

**DEBUG:** When DEBUG=1 is specified, debug information will be generated when building tools and applications. Also, no compilation and/or link optimizations will be performed.

**PROBE:** Specifying PROBE=1 enables the probe-mode tests.

## Questions? Bugs?

Send bugs and questions to [pinheads@yahogroups.com](mailto:pinheads@yahogroups.com). Complete bug reports that are easy to reproduce are fixed faster, so try to provide as much information as possible. Include: kit number, your OS version, compiler version. Try to reproduce the problem in a simple example that you can send us.

## Disclaimer and Legal Information

The information in this manual is subject to change without notice and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. The information in this document is provided in connection with Intel products and should not be construed

as a commitment by Intel Corporation.

EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Copyright (c) 2004-2015, Intel Corporation. All rights reserved.

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.


Java is a registered trademark of Oracle and/or its affiliates.

Other names and brands may be claimed as the property of others.

Copyright Intel Corporation. All rights reserved. Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA.

=====

---

Generated on Wed Jan 21 02:16:16 2015 for Pin by  1.4.6