Profiler

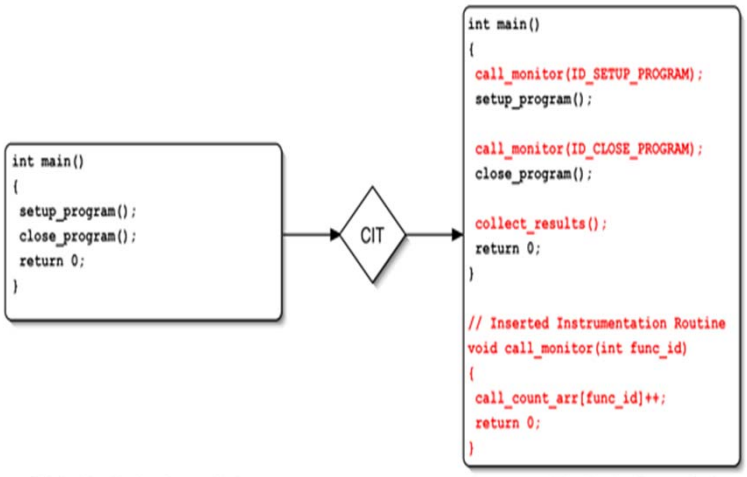
- In software engineering, **profiling** ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. (Wikipedia)
- Static Analysis Tools
- Dynamic Analysis Tools
- Hybrid Analysis Tools

(Source - **An Overview of Software Performance Analysis Tools and Techniques: From GProf to Dtrace**)

Static Analysis Tools

- Two methods
  - Source Code Instrumentation – data collection or sampling routines are added
    - Performance slowdown due to instrumented code
    - Can potentially modify the behaviour of program
  - Sampling
- Compile Time instrumentation Tools
  - Source code instrumented by compiler
  - Counters and Monitor Calls inserted
  - Generate Function Call Graphs
  - gprof
- Sampling Tools
  - At regular intervals record state
  - Accuracy and runtime depend on Sampling rate
  - User level monitoring code (*qprof)*
  - Kernel level monitoring code (*oprofile*)

Compile Time Instrumentation Tool



Original Application Source Code

Instrumented Application Source Code
(Inserted Segments Highlighted in Red)

Sampling Tool

```
// Main Routine of Sampling Tool
int main(char *app_to_monitor)
{
  status = 0;

  // Setup Timer to Collect Data
  setup_timer(TIMER_RES, "interrupt_routine");

  // Fork the App Being Analyzed
  status = fork(app_to_monitor);

  // Wait Until the App Completes
  while( wait(status) != 0);

  // Collect Results of Analysis
  collect_results();
  return 0;
}

// Routine Used to Handle Timer Executions
void interrupt_routine(int previous_pc)
{
  // Update Count for The Calling PC
  // Can Relate to Specific Function in Post Analysis
  update_count(previous_pc);

  // Reset Interrupt Routine for Next Time Slice
  reset_counter("interrupt_routine");

  return;
}
```

Source Code of a Typical Sampling Tool (ST)

Static Analysis Tools

- Hardware Counter Tools
  - Use h/w event counters provided in processors (e.g. cache misses, number of floating point instructions etc.)
  - Perfsuite (linux based)
- Compound Tools
  - ST plus HCT
  - vTune (Intel), Code XL(AMD)

gprof

- Uses Compile Time Instrumentation and Sampling
- Compile source code with –pg
- Run the executable, will generate *gmon.out*
- gprof gmon.out > profiler.output
- Flat profile – time spent in each function

Flat profile:

Each sample counts as 0.01 seconds**.**

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write

Dynamic Analysis Tools

- Binary level modifications at runtime
- Two types
  - Binary Instrumentation
    - Customized analysis routines inserted at arbitrary points at runtime
    - APIs provided by the tool can be used to decide what to monitor
    - Pin
  - Probing
    - Predefined Shared library or kernel routines (Probes) to collect data
    - DTrace

PIN

- pin –t pintool – application
- pin –t pintool –pid 1234 (attach to a process)
  - pin -> instrumentation engine
  - pintool -> instrumentation tool
- Instrumentation routines – where instrumentation is inserted (e.g. before instr)
- Analysis routines – what to do (e.g. incr counter)

Pintool example

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

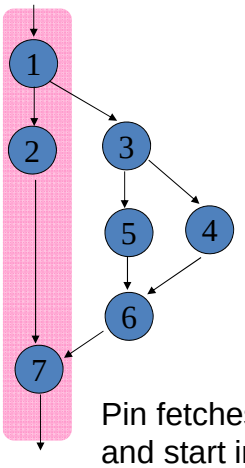
Instr pointer (PC)

Effective addr of a mem write

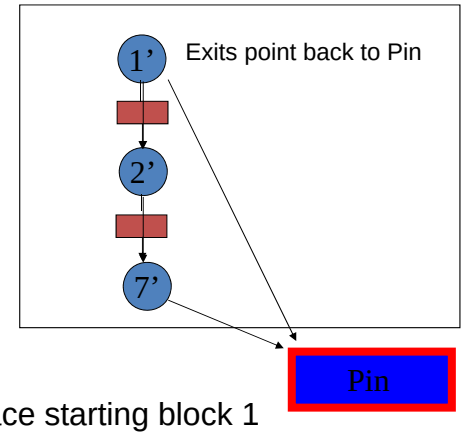
Figure 1. A Pintool for tracing memory writes.

Dynamic Instrumentation

Original code



Code cache

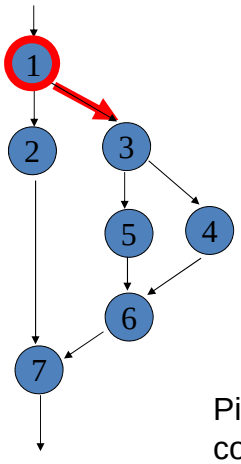


Pin fetches trace starting block 1 and start instrumentation

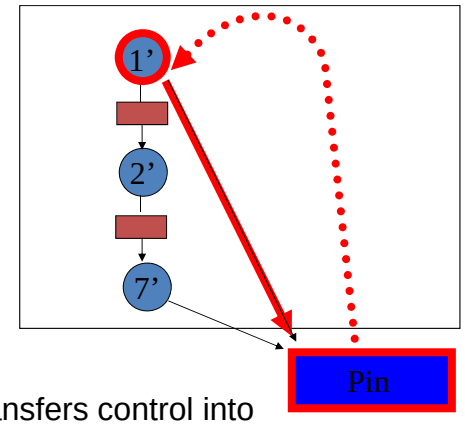
PLDI'05 10

Dynamic Instrumentation

Original code



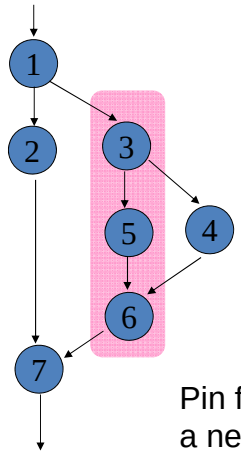
Code cache



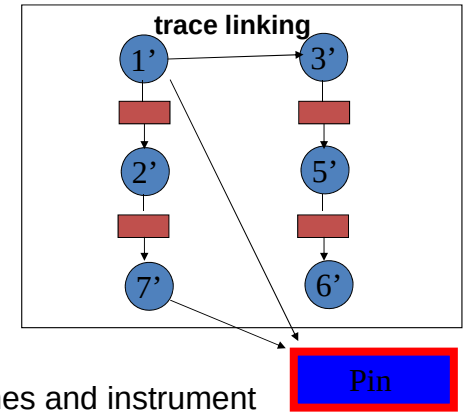
Pin transfers control into code cache (block 1)

Dynamic Instrumentation

Original code

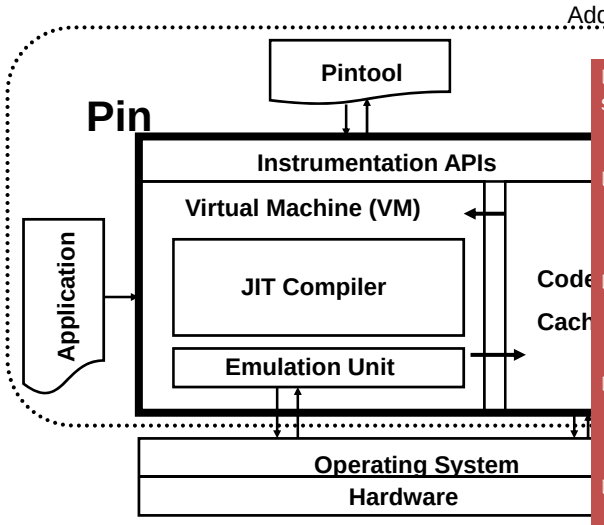


Code cache



Pin fetches and instrument a new trace

Pin's Software Architecture



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

=> Coordinated by VM

CodeXL(AMD)

- **Source – CodeXL User Manual**
- Available on Windows and Linux
- Statistical Sampling Based
- Three profile modes
  - Time Based Profile (TBP) – to identify 'hotspots'
  - Event Based Profile (EBP) – uses HW Performance Monitor Counters (PMC). Example: processor clock cycles, retired instructions, data cache accesses, and data cache misses
  - Instruction Based Sampling (IBS)

CodeXL Usage

- Two modes:
  - Execute a CPU Profile Session
  - Attach to a process
- Select a CPU profile type
  - Time based profile
  - Event based profile
  - Instruction based sampling

Event Based Profiling
(Assess Performance)

- Retired Instructions
- CPU clock cycles not halted
- Retired branch instructions
- Retired mispredicted branch instructions
- Data cache accesses
- Data cache misses
- L1 DTLB and L2 DTLB misses
- Misaligned accesses

Event Based Profiling
(Investigate Branching)

- Retired Instructions
- Retired branch instructions
- Retired mispredicted branch instructions
- Retired taken branch instructions
- Retired near returns
- Retired mispredicted near returns
- Retired mispredicted indirect branches

Event Based Profiling
(Investigate Data Access)

- Retired Instructions
- Data cache accesses
- Data cache misses
- Data cache refills from L2 or Northbridge
- L1 DTLB miss and L2 DTLB hit
- L1 DTLB and L2 DTLB misses
- Misaligned accesses

Event Based Profiling
(Investigate Instruction Access)

- Retired Instructions
- Instruction Cache fetches
- Instruction Cache misses
- L1 ITLB miss and L2 ITLB hits
- L1 ITLB miss and L2 ITLB miss

Event Based Profiling
(Investigate L2 Cache Access)

- Retired Instructions
- Requests to L2 cache
- L2 cache misses
- L2 fill/writeback

Event Based Profiling (**Cache Line Utilization**)

- **(CLU)** measures how much of a cache line is used (read or written) before it is evicted from the cache.

Event	Description
Cache Line Utilization Percentage	The cache line utilization percentage for all cache lines on all cores accessed by this instruction / function / module.
Line Boundary Crossings	The number of accesses to the cache line that spanned two cache lines. This happens when an unaligned access is made that causes two cache lines to be touched.
Bytes/L1 Eviction	The number of bytes accessed between cache line evictions.
Accesses/L1 Eviction	The number of accesses (loads plus stores) to a cache line between evictions.
L1 Evictions	The number of times a cache line was evicted where this instruction depended on the data in the cache line.
Accesses	The total number of loads and stores samples for this instruction / function / module.
Bytes Accessed	The total number of bytes accessed by this instruction / function / module.

Instruction Based Sampling

- Two main categories of pipeline stages:
  - Instruction Fetch
  - Instruction Execution
- IBS Fetch Sampling:
  - Select a fetch at the end of a sampling period (certain number of fetched instructions)
  - IBS fetch sample is taken when the selected fetch completes/aborts
    - The fetch address
    - Whether the fetch completed or aborted
    - Whether the fetch missed in the instruction cache (IC)
    - Whether the fetch missed in the level 1 or level 2 ITLB
    - The page size of the address translation
    - The fetch latency, i.e., cycles from when the fetch was initiated to when the fetch either completed or aborted

Instruction Based Sampling

- IBS op Sampling : Each AMD64 instruction translated into or more macro ops.
- Counts processor cycles/dispatchd ops and periodically selects an op to be tagged and monitored and IBS sample is generated only if the op retires
  - The AMD64 instruction address for the op
  - The tag-to-retire time (cycles from when the op was tagged to when the op retired)
  - The completion-to-retire time (cycles from when the op completed to when the op retired)
  - Whether the op implements AMD64 branch semantics (a "branch op")
    - If the branch op was mispredicted
    - If the branch was taken
    - If the branch was a return
    - If the return was mispredicted
  - Whether the op performed a load and/or store operation
    - If the operation missed in the data cache
    - If the operation missed in the level 1 or level 2 DTLB
    - The page size of the level 1 or level 2 address translation
    - If the operation caused a misaligned access
    - The DC miss latency (in cycles) if the load operation missed in the data cache
    - The virtual and physical address of the requested memory location
    - If the access was made to local or remote memory