PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation

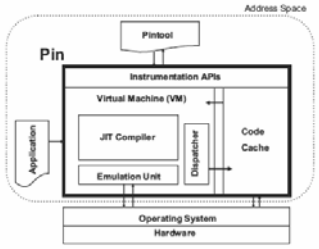
Luk et al PLDI 2005

Presented by Godmar Back

PIN

- Dynamic instrumentation framework
- Goals:
  - Easy-to-use
  - Portable
  - Transparent
  - Efficient
  - Robust

PIN Architecture



Sample

- Note: no architecture -dependent code

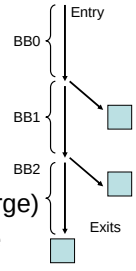
```
FILE * trace;

// 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;
}
```


How PIN works

- Reads binary:
  - Parse ELF binary same way OS would – finds entry point
- Parses machine instructions of binary (1)
- Parses machine instructions of (compiled) instrumentation code (2)
- Inserts (2) in (1) as directed by tool
- Translates mix to (same-architecture) machine instructions
  - No IR is used
  - Translated instructions stored in a cache
- Executes translated instructions only

Traces

- PIN offers instrumentation at different levels.
- Key concept is a trace.
- Trace: straight-line sequence of instructions that end with unconditional transfer (or if too large)
- PIN translates one trace at a time



Instrumentation Levels

- By default instrumentation done by trace
- Provides "by instrumentation" – implemented as convenience only

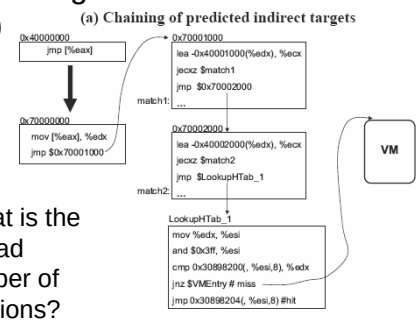
```
for (b : basicblocks) {
        for (i : instructions(b)) { …. }}
```

- Routine:
  - Add instrumentation once a routine is entered
- Image:
  - Perform some instrumentation when an image is loaded (executable, .so file, etc.)

Techniques (1): Trace Linking

- When a trace ends
  - Examples:
    - virtual method dispatch: jmp *eax
    - Function call return
  - First time, return to VM, examine where it ended and where it goes, translate subsequent trace.
  - Second time, would like to jump directly to successor trace if at all possible
- Easy if ends with direct jump
- Need prediction if it ends with indirect jump

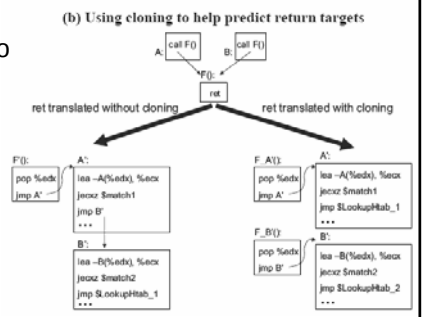
Trace Linking (cont'd)



- Q.: what is the overhead in number of instructions?

Cloning & Trace Linking

- Q. Why do we still need a mis-prediction check here?



Register Reallocation

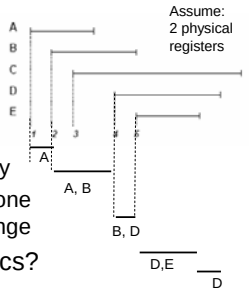
- Virtual vs. physical registers
  - "Virtual registers" are machine registers (%EAX, %EBX, etc.) as seen by the application program's compiler
  - "Physical registers" are the ones holding the actual values during the execution of translated code
- Must map virtual to physical
  - Must guarantee that life virtual registers are not destroyed; spill to memory if needed.
- Register allocation problem!

Register Allocation

- Traditional approach:
  - Build CFG. Do Liveness analysis. Compute interference graph. Color it. Assign registers
  - Won't work here because entire CFG is not known – it's incrementally built.
- Alternative:
  - Linear-scan register assignment

Linear Scan Allocation [Poletto'99]

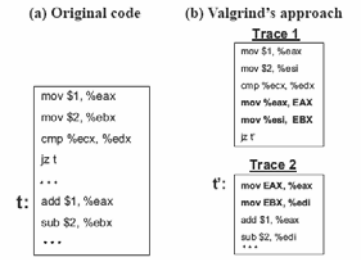
- Idea:
  - Determine live ranges
  - Range defined as instruction index
  - Assign registers greedily
  - When spilling, spill the one with the farthest end range
- Q.: What is the heuristics?



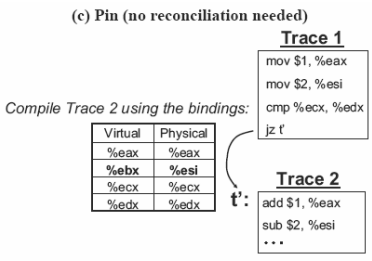
Register Reallocation (cont'd)

- When linking traces, would like to avoid rearranging registers: thus, on code cache miss, jit target trace with v-to-p mapping that origin trace ended with.
- Second time around (if target trace is reached from different origin):
  - need compensation code
- By comparison: valgrind always spills all virtual registers to memory

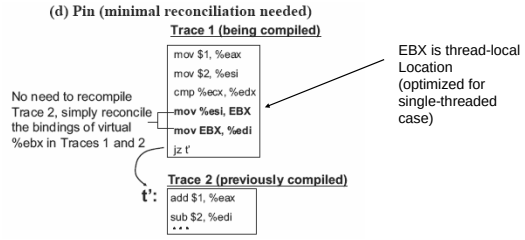
Register Reconciliation (1)



Register Reconciliation (2)



Register Reconciliation (3)



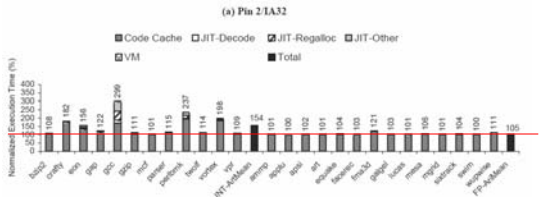
Other Optimizations

- Inlining vs. Bridging
- Big question: when to inline (will examine on Thursday)
- Inlining optimization:
  - Can avoid saving caller-saved registers blindly (including eflags) – why?

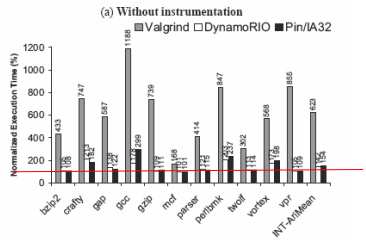
Performance Analysis

- What counts is overhead/slowdown.
  - How much is acceptable? 120%? 200%? 2000%?
- Must know NULL-tool overhead/baseline slowdown
- Obviously, tool overhead depends on tool
  - How much work is done in tool code (in common path?)
  - How efficient is bridging code/how often could inlining be applied?

NULL-tool overhead



NULL-tool: PIN vs Competition



Count-BB Tool

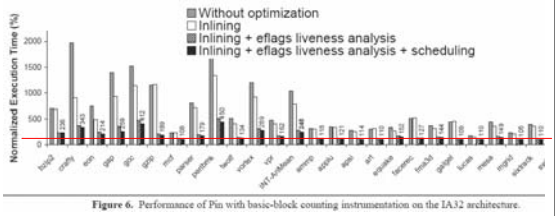
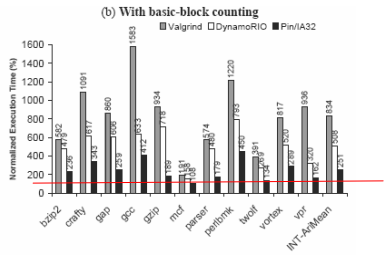


Figure 6. Performance of Pin with basic-block counting instrumentation on the IA32 architecture.

BB-tool: PIN vs Competition



Applications

- Only few at the time PIN was published; many more now, see http://rogue.colorado.edu/pin
- Mainly used in architecture community so far
  - Cache simulation, program phase analysis, etc.
- Top

PIN Goals Revisited

- Easy-to-use
  - Yes, but little support for accessing internals (e.g. liveness ranges etc.); little support for accessing symbolic information
- Portable
  - Yes: four architectures, 3 OS
- Transparent
  - Almost completely (minus address space effects)
- Efficient
  - According to their benchmarks for simple codes
- Robust
  - In my experience, pretty robust

Discussion/Questions