SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance

Steven Wallace and Kim Hazelwood

Dynamic Binary Instrumentation
Inserts user-defined instructions into executing binaries
- Easily
- Efficiently
- Transparently

Why?
- Detect inefficiencies
- Detect bugs
- Security checks
- Add features

Examples
- Valgrind, DynamoRIO, Strata, HDTrans, Pin

Intel Pin
- A dynamic binary instrumentation system
- Easy-to-use instrumentation interface
- Supports multiple platforms
  - Four ISAs – IA32, Intel64, IPF, ARM
  - Four OSes – Linux, Windows, FreeBSD, MacOS
- Robust and stable (Pin can run itself!)
  - 12+ active developers
  - Nightly testing of 25000 binaries on 15 platforms
  - Large user base in academia and industry
  - Active mailing list (Pinheads)
- 11,500 downloads

Our Goal: Improve Performance
The latest Pin overhead numbers ...

Adding Instrumentation

Sources of Overhead
Internal
- Compiling code & exit stubs (region detection, region formation, code generation)
- Managing code (eviction, linking)
- Managing directories and performing lookups
- Maintaining consistency (SMC, DLLs)

External
- User-inserted instrumentation
“Normal Pin” Execution Flow

Instrumentation is interleaved with application

Uninstrumented Application

“Pinned” Application

Instrumented Application

Pin Overhead

Instrumentation Overhead

“SuperPin” Execution Flow

SuperPin creates instrumented slices

Uninstrumented Application

SuperPinned Application

Instrumented Slices

Issues and Design Decisions

Creating slices
• How/when to start a slice
• How/when to end a slice

System calls

Merging results

Execution Timeline

Starting Slices

How?
• Master application process – ptrace
• Controlling process
• Child slices – fork
  – Reserve memory for transparency
  – Each slice has its own code cache (for now)

When?
• Timeouts
  – Uses a special timer process
  – Tunable parameter
• System calls

Handling System Calls

Problem: Don’t want to duplicate system calls in the main application/slices

Solutions:
• brk or anonymous mmap – duplication OK
• frequent calls – record and playback
• default – trigger new timeslice
Ending Slices

Each slice is responsible for detecting its own end-of-slice condition

S4+
record sig4, sleep
resume
detect sig5

Challenges:
• Need to efficiently capture a point in time (signature)
• Need to efficiently detect when we’ve reached that point

Signature Detection

End-of-slice conditions:
1. System calls – easy to detect
2. Timeouts at arbitrary points – harder to detect

Signature match:
• Instruction pointer
• Architectural state
• Top of stack

Implementing Signature Detection

Uses Pin’s lightweight conditional analysis
• INS_InsertIfCall – lightweight inlined check
• INS_InsertThenCall – heavyweight (conditional) analysis routine

Instrument the end-of-slice instruction pointer
1. Lightweight check – two registers
2. Heavyweight check – full architectural state
3. Heavyweight check – top 100 words on the stack

• Lightweight triggers heavyweight: ~2%
• Stack check fails: ~0%

Performance Results

Icount1 – Instruments every instruction with count++

% pin -t icount1 --./hello
Hello CGO
Count: 496043

Icount2 – Instruments every basic block with count += bblength

% pin -t icount2 -- ./hello
Hello CGO
Count: 496043

Performance – icount1

Performance – icount2
Performance Scalability
Running on an 8-processor HT-enabled machine (16 virtual processors)

Overhead Categorization
Where is SuperPin spending its time?
- Executing the application
- Fork overheads
- Sleeping (waiting to start a slice)
- Pipeline delays

Overhead Categorization

The SuperPin API
You may write SuperPin-aware Pintools:
- SP_Init(fun)
- SP_AddSliceBeginFunction(fun,val)
- SP_AddSliceEndFunction(fun,val)
- SP_EndSlice()
- SP_CreateSharedArea(local,size,merge)

You may also control (via switches):
- Spmsec {value}: milliseconds per timeslice
- Spmp {value}: maximum slice count
- Spsysrecs {value}: maximum syscalls per slice

Pin Instrumentation API – icount2

SuperPin Version of Icount2

```c
VOID DoCount(INT32 c) { icount += c; }
VOID ToolReset(INT32 c) { icount = 0; }
VOID Merge(INT32 sliceNum) { *sharedData += icount; }

int main(INT32 argc, CHAR **argv) {
    PIN_Init(argc, argv);
    SP_Init(ToolReset);
    sharedData = (UINT64*) SP_CreateSharedArea(&icount, sizeof(icount), 0);
    SP_AddSliceBeginFunction(Merge);
    TRACE_AddInstrumentFunction(Trace);
    PIN_StartProgram();
    return 0;
}
```
SuperPin Limitations
Not all instrumentation tasks are a good fit
Great fit – independent tasks
• Instruction profiling (counts, distributions)
• Trace generation
Requires massaging – dependent tasks
• Branch prediction
• Data cache simulation
  – Assume a starting state, resolve later
Stick with regular Pin – path modification
• Adaptive execution

Future (Rainy Day) Extensions
• Adaptive parallelism detection
  – Hardware feedback: adapts to available processors
  – OS feedback: adapts to present load
• Adaptive slice timeouts
• Slice-shared code caches
• Multithreaded application support

SuperPin Summary
Allows users to leverage available parallelism for certain instrumentation tasks
• Hides the gory details
• Enables significant speedup (for the right tasks ... on the right machines)
• Exposed as Pin API extensions

Download it today!
http://rogue.colorado.edu/pin