COSC 340: Software Engineering

Debugging with Dynamic Binary Analysis & Instrumentation

Michael Jantz

Program Analysis

- Programmers often use *analysis tools* to improve program quality
- Static Analysis
  - Analyze program source code or machine code without running it
  - Often performed by compilers (type checking, optimizations)
- Dynamic Analysis
  - Analyze a client program as it runs
  - Profilers, error checkers, execution visualizers
  - DA instruments the client program with analysis code
  - May change depending on program input

Source Analysis vs. Binary Analysis

- Source Analysis
  - Analyze program at level of source code
  - Generally performed in terms of programming language constructs (functions, statements, expressions, variables)
  - Example: control-flow graph
- Binary Analysis
  - Analyze program at the level of machine code: object code (pre-linking) or executable code (post-linking)
  - Includes executable intermediate representations (bytecodes)
  - Generally performed in terms of machine entities (procedures, instructions, registers, memory locations)

Analysis Properties

	Static	Dynamic
Source	Considers all execution paths Platform independent Access to high-level information (functions, expressions, variables)	Considers run-time information, but only for one execution path Platform independent Access to high-level information (functions, expressions, variables)
Binary	Considers all execution paths Does not require source Access to low-level information (registers, memory locations)	Considers run-time information, but only for one execution path Does not require source Access to low-level information (registers, memory locations)

Analysis Properties

	Static	Dynamic
Source	Considers all execution paths Platform independent Access to high-level information (functions, expressions, variables)	Considers run-time information, but only for one execution path Platform independent Access to high-level information (functions, expressions, variables)
Binary	Considers all execution paths Does not require source code Access to low-level information (registers, memory locations)	**Considers run-time information, but only for one execution path** **Does not require source code** **Access to low-level information (registers, memory locations)**

Instrumentation for Dynamic Binary Analysis

- Static Binary Instrumentation:
  - Before program is run, rewrite the binary
- Dynamic Binary Instrumentation
  - Code is injected into the client process at runtime (either by a program that is grafted onto the client, or by an external process)
- Focus on DBI:
  - Advantages:
    - Does not require preparation of the client program
    - Easy to naturally cover all the reached code
  - Disadvantages
    - Cost of instrumentation incurred at run-time
    - May be difficult to implement

Valgrind

- DBI framework for building *heavyweight* analysis
  - Every instruction is instrumented
  - Tools track a lot of metadata (e.g. every register is memory value is shadowed by a metavalue)
- Valgrind Tools
  - MemCheck
  - Cachegrind
  - Massif

MemCheck

- Memory error detector for C and C++ programs
- Can help detect the following problems:
  - Accessing memory you shouldn't, e.g. overrunning heap blocks, overrunning the top of the stack, and accessing memory after it has been freed.
  - Using undefined values
  - Double-freeing heap blocks
  - Overlapping *src* and *dst* in *memcpy* and related functions
  - Passing negative values as the size to malloc
  - Memory leaks

MemCheck Example

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

void f(void)
{
        int* x = malloc(10 * sizeof(int));
        x[10] = 0;                  // problem 1: heap block overrun
}                                   // problem 2: memory leak -- x not freed

int main(void)
{
        int y;
        printf("y = %d\n", y);              // problem 3: y used before initialized

        f();
        return 0;
}
```

MemCheck Example

- Running MemCheck

```
valgrind --leak-check=full ./mc
==27967== Memcheck, a memory error detector
==27967== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==27967== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==27967== Command: ./mc
```

- Error seen when MemCheck detects an uninitialized value

```
==27967== Conditional jump or move depends on uninitialised value(s)
==27967==    at 0x4E7CE12: vfprintf (in /usr/lib64/libc-2.17.so)
==27967==    by 0x4E86C98: printf (in /usr/lib64/libc-2.17.so)
==27967==    by 0x4005C1: main (mc.c:13)
```

MemCheck Example

- Error seen when it detects a write to an invalid address

```
==27967== Invalid write of size 4
==27967==    at 0x40059E: f (mc.c:7)
==27967==    by 0x4005C6: main (mc.c:15)
```

- Error seen when it detects a memory leak

```
==27967== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==27967==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==27967==    by 0x400591: f (mc.c:6)
==27967==    by 0x4005C6: main (mc.c:15)
==27967==
```

Cachegrind

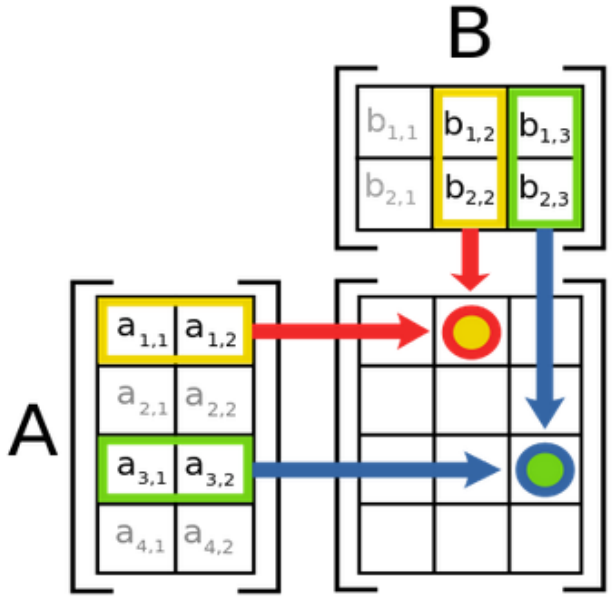
- Simulates how your program interacts with the cache hierarchy and branch predictor
  - Includes simulation for first-level instruction and data caches (I1 and D1) and the last level (LL) cache
  - Does not simulate caches between first and last level

Cachegrind

- Collects the following:
  - I cache reads (Ir which is the same as the instructions executed), I1 cache read misses (I1mr) and LL cache reads and misses (ILmr)
  - D cache reads (Dr, which equals the number of memory reads), D1 cache read misses (D1mr) and LL cache data read misses (DLmr)
  - D cache writes (Dw, which equals the number of memory writes), D1 cache write misses (D1mw) and LL cache data write misses (DLmw)
  - Conditional branches executed (Sc) and conditional branches mispredicted (Scm)
  - Indirect branches executed (Bi) and indirect branches mispredicted (Bim)

Cachegrind Example: Matrix Multiply

```
class MM {
        public:
                        vector < vector <double> > M1;
                        vector < vector <double> > M2;
                        vector < vector <double> > P;
                        int Print;
                        void Multiply();
                        void PrintAll();
};

void MM::Multiply()
{
        int i, j, k;

        for (i = 0; i <  P.size(); i++) {
                        for (j = 0; j <  P[0].size(); j++) {
                                        for (k = 0; k <  M2.size(); k++) P[i][j] += (M1[i][k] * M2[k][j]);
                        }
        }
}
```



Cachegrind Example: Matrix Multiply

- Solution: store the second matrix with rows as columns / columns as rows (i.e. store its transpose)

```
void MM::Multiply()
{
        int i, j, k;

        for (i = 0; i <  P.size(); i++) {
                for (j = 0; j <  P[0].size(); j++) {
                        for (k = 0; k <  M1[0].size(); k++) P[i][j] += (M1[i][k] * M2[j][k]);
                                //                                This is the change: ^^^^
                }
        }
}
```

Cachegrind Example: Matrix Multiply

```
[~/teaching/matrix]
mrjantz@com1643$ valgrind --tool=cachegrind ./mm-plain 600 600 600 1 n
==28382== Cachegrind, a cache and branch-prediction profiler
==28382== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==28382== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==28382== Command: ./mm-plain 600 600 600 1 n
==28382==
--28382-- warning: L3 cache found, using its data for the LL simulation.
Time: 11.8845
==28382==
==28382== I   refs:      2,441,613,670
==28382== I1  misses:            1,674
==28382== LLi misses:            1,662
==28382== I1  miss rate:          0.00%
==28382== LLi miss rate:          0.00%
==28382==
==28382== D   refs:        889,692,360  (665,023,032 rd   + 224,669,328 wr)
==28382== D1  misses:      325,995,459  (325,767,145 rd   +     228,314 wr)
==28382== LLd misses:          317,352  (     89,647 rd   +     227,705 wr)
==28382== D1  miss rate:          36.6% (       48.9%     +         0.1%  )
==28382== LLd miss rate:           0.0% (        0.0%     +         0.1%  )
==28382==
==28382== LL refs:         325,997,133  (325,768,819 rd   +     228,314 wr)
==28382== LL misses:           319,014  (     91,309 rd   +     227,705 wr)
==28382== LL miss rate:            0.0% (        0.0%     +         0.1%  )
```

Cachegrind Example: Matrix Multiply

```
[~/teaching/matrix]
mrjantz@com1643$ valgrind --tool=cachegrind ./mm-transpose 600 600 600 1 n
==28384== Cachegrind, a cache and branch-prediction profiler
==28384== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==28384== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==28384== Command: ./mm-transpose 600 600 600 1 n
==28384==
--28384-- warning: L3 cache found, using its data for the LL simulation.
Time: 7.0942
==28384==
==28384== I   refs:      2,010,691,576
==28384== I1  misses:            1,652
==28384== LLi misses:            1,638
==28384== I1  miss rate:          0.00%
==28384== LLi miss rate:          0.00%
==28384==
==28384== D   refs:        674,051,405  (449,382,209 rd   + 224,669,196 wr)
==28384== D1  misses:       28,008,372  ( 27,465,213 rd   +     543,159 wr)
==28384== LLd misses:          314,211  (     89,619 rd   +     224,592 wr)
==28384== D1  miss rate:           4.1% (        6.1%     +         0.2%  )
==28384== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==28384==
==28384== LL refs:          28,010,024  ( 27,466,865 rd   +     543,159 wr)
==28384== LL misses:           315,849  (     91,257 rd   +     224,592 wr)
==28384== LL miss rate:            0.0% (        0.0%     +         0.0%  )
```

Massif: Heap Profiler

- Measures how much heap space your program uses
  - Useful space and extra bytes for alignment
  - Can also measure stack space (but does not do so by default)
- Collects 'snapshots' of memory usage at certain point in time
- Also tells you how much heap data is allocated at each allocation site

Massif Example

```c
#include <stdlib.h>

void g(void)
{
   malloc(4000);
}

void f(void)
{
   malloc(2000);
   g();
}

int main(void)
{
   int i;
   int* a[10];

   for (i = 0; i < 10; i++) {
      a[i] = malloc(1000);
   }

   f();

   g();

   for (i = 0; i < 10; i++) {
      free(a[i]);
   }

   return 0;
}
```

Massif Example

- Run massif:
  - > valgrind --tool=massif ./prog
- View information collected by massif
  - > ms_print massif.out.PID
- By default, snapshots measured in instructions executed
  - For short-running programs, use --time-unit=B to measure snapshots in bytes allocated

Massif Output Graph

```
[~/teaching/massif]
mrjantz@com1643$ ms_print massif.out.26288
--------------------------------------------------------------------------------
Command:            ./prog
Massif arguments:   (none)
ms_print arguments: massif.out.26288
--------------------------------------------------------------------------------


    KB
19.71^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
   0 +----------------------------------------------------------------------->ki
     0                                                                   106.9

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

Massif Output Graph

```
[~/teaching/massif]
mrjantz@com1643$ ms_print massif.out.27329
--------------------------------------------------------------------------------
Command:            ./prog
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.27329
--------------------------------------------------------------------------------


    KB
19.71^                                                  ###
     |                                                  #
     |                                                  #  ::
     |                                                  #  : :::
     |                                         :::::::::#  : :  ::
     |                                         :        #  : :  : ::
     |                                         :        #  : :  : : :::
     |                                         :        #  : :  : : :  ::
     |                                 ::::::::::       #  : :  : : :  : :::
     |                                 :        :       #  : :  : : :  : :  ::
     |                           :::::          :       #  : :  : : :  : :  : ::
     |                       @@@:   :          :       #  : :  : : :  : :  : : @
     |                     ::@  :   :          :       #  : :  : : :  : :  : : @
     |                 :::: @  :   :          :       #  : :  : : :  : :  : : @
     |               :::  : @  :   :          :       #  : :  : : :  : :  : : @
     |            ::: :  : @  :   :          :       #  : :  : : :  : :  : : @
     |         :::: : :  : @  :   :          :       #  : :  : : :  : :  : : @
     |       :::  : : :  : @  :   :          :       #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :          :       #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :          :       #  : :  : : :  : :  : : @
   0 +-------------------------------------------------------------------------->KB
     0                                                                     29.63

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

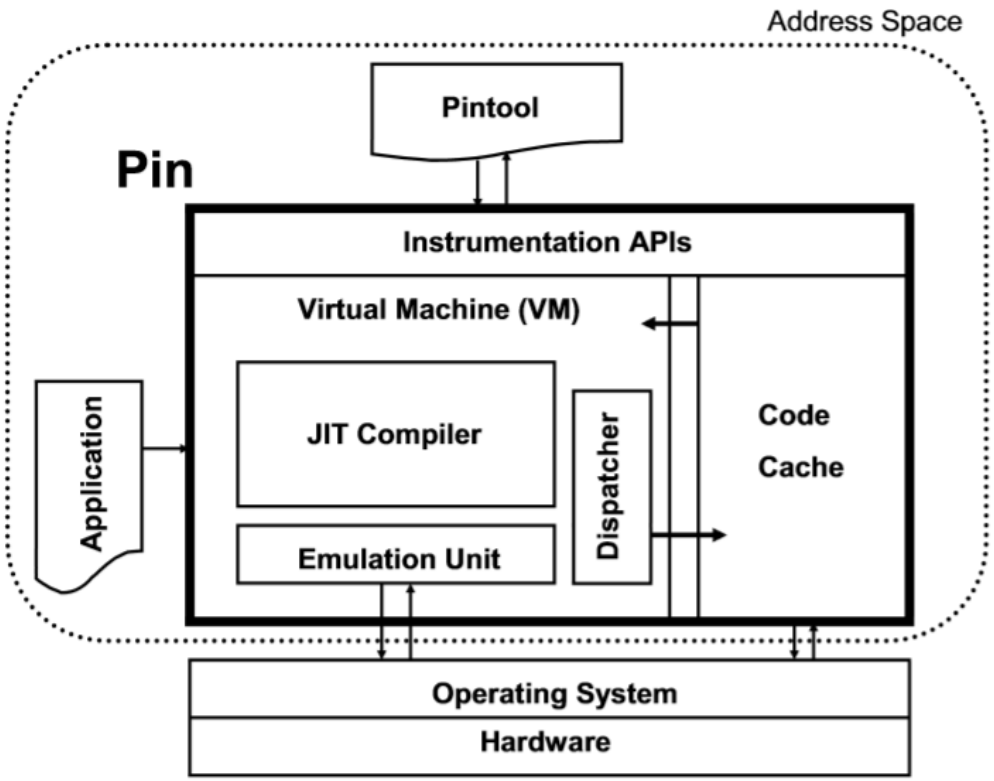
Snapshot Details

```
Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,016            1,016            1,000            16            0
  2          2,032            2,032            2,000            32            0
  3          3,048            3,048            3,000            48            0
  4          4,064            4,064            4,000            64            0
  5          5,080            5,080            5,000            80            0
  6          6,096            6,096            6,000            96            0
  7          7,112            7,112            7,000           112            0
  8          8,128            8,128            8,000           128            0
  9          9,144            9,144            9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005BE: main (prog.c:20)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,160           10,160           10,000           160            0
 11         12,168           12,168           12,000           168            0
 12         16,176           16,176           16,000           176            0
 13         20,184           20,184           20,000           184            0
 14         20,184           20,184           20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005BE: main (prog.c:20)
|
->39.64% (8,000B) 0x40058C: g (prog.c:5)
| ->19.82% (4,000B) 0x4005A1: f (prog.c:11)
| | ->19.82% (4,000B) 0x4005DA: main (prog.c:23)
| |
| ->19.82% (4,000B) 0x4005DF: main (prog.c:25)
|
->09.91% (2,000B) 0x40059C: f (prog.c:10)
  ->09.91% (2,000B) 0x4005DA: main (prog.c:23)
```

Intel Pin

- Another tool for dynamic binary instrumentation
- Allows users to write their own DBI tools
  - Insert 'arbitrary' code in 'arbitrary' places in the executable
  - Uses *dynamic compilation* to instrument executables as they are running
  - Provides a platform-independent API for doing useful things with DBI
- Some example tools written in Pin
  - Memory tracers
  - Call site profilers
  - Cache simulators

Pin's Software Architecture



Pin Tool for Tracing Memory Writes

```
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;
}
```

Output of pinatrace (Memory Tracing Tool)

```
0x7f39f273c433: W 0x7ffd0d97a208
0x7f39f273f940: W 0x7ffd0d97a200
0x7f39f273f944: W 0x7ffd0d97a1f8
0x7f39f273f946: W 0x7ffd0d97a1f0
0x7f39f273f948: W 0x7ffd0d97a1e8
0x7f39f273f94a: W 0x7ffd0d97a1e0
0x7f39f273f94f: W 0x7ffd0d97a1d8
0x7f39f273f95f: R 0x7f39f295cdf0
0x7f39f273f966: W 0x7f39f295cc18
0x7f39f273f977: R 0x7f39f295cfa0
0x7f39f273f981: W 0x7f39f295d9a8
0x7f39f273f988: W 0x7f39f295d998
0x7f39f273f9df: W 0x7f39f295da48
0x7f39f273f9e7: R 0x7f39f295ce00
0x7f39f273f9df: W 0x7f39f295daa0
0x7f39f273f9e7: R 0x7f39f295ce10
0x7f39f273f9df: W 0x7f39f295dab0
0x7f39f273f9e7: R 0x7f39f295ce20
0x7f39f273f9df: W 0x7f39f295d9f8
0x7f39f273f9e7: R 0x7f39f295ce30
0x7f39f273fc33: W 0x7f39f295dc30
0x7f39f273f9e7: R 0x7f39f295ce40
0x7f39f273f9df: W 0x7f39f295da00
0x7f39f273f9e7: R 0x7f39f295ce50
0x7f39f273f9df: W 0x7f39f295da08
0x7f39f273f9e7: R 0x7f39f295ce60
```

Output of proccount.so (Call Site Profiler)

Procedure	Image	Address	Calls	Instructions
BZ2_hbCreateDecodeTables	bzip2	40b590	66	972564
BZ2_hbAssignCodes	bzip2	40b540	66	840460
BZ2_hbMakeCodeLengths	bzip2	40b090	264	26536685
BZ2_decompress	bzip2	4088f0	738	1348674514
BZ2_compressBlock	bzip2	4066f0	11	2340829139
bsPutUInt32	bzip2	406670	14	350
bsW	bzip2	4065e0	5460796	139696616
BZ2_bzReadGetUnused	bzip2	406180	3	51
BZ2_bzRead	bzip2	405f30	1260	113190
BZ2_bzReadClose	bzip2	405ea0	3	54
BZ2_bzReadOpen	bzip2	405b40	3	189
BZ2_bzWriteClose64	bzip2	4058f0	3	4522
BZ2_bzWrite	bzip2	405780	1260	74374
BZ2_bzWriteOpen	bzip2	405620	3	180
BZ2_bzDecompressEnd	bzip2	405590	3	96
BZ2_bzDecompress	bzip2	4047c0	1984	157496191
BZ2_bzDecompressInit	bzip2	404670	3	150
BZ2_bzCompressEnd	bzip2	4045f0	3	108
BZ2_bzCompress	bzip2	404470	1985	46149
BZ2_bzCompressInit	bzip2	404220	3	3375
handle_compress.isra.2	bzip2	403db0	1985	323506368
default_bzalloc	bzip2	403da0	18	54
default_bzfree	bzip2	403d80	18	72
add_pair_to_block	bzip2	403c40	33894	1506675
uncompressStream	bzip2	403920	3	25386
compressStream	bzip2	4035c0	3	24084
myfeof	bzip2	403430	1266	16434
BZ2_blockSort	bzip2	4031c0	11	29973150
mainSort	bzip2	402300	11	176057843
mainGtU	bzip2	4020b0	312452	4120033586
fallbackSort	bzip2	4017b0	9	9403779636
spec_fwrite	bzip2	401600	1993	65769
spec_reset	bzip2	401510	6	84
spec_ungetc	bzip2	401420	3751	108779

Cache Simulation with Pin

- Multiple cache simulation tools ship with Pin
  - dcache: simulates a simple L1 data cache
  - allcache: simulates instruction, data caches and TLB's
- Pin cache tools are configurable
  - Tune total size, line size, associativity
  - Allows HW/SW design space exploration
- Can combine with other tools for advanced analysis
  - For example, combine with memory tracing, allocation site profiling to understand cache locality of particular application data

Tuning Cache Size with Pin

- Evaluate D1 cache miss rate with matrix multiply
  - Multiply two 800x800 matrices of doubles
  - Use Pin to vary cache size with 32KB, 64KB, and 128KB of D1 cache

D1 cache size	mm-plain D1 miss %	mm-transpose D1 miss %
32 KB	47.55	9.9
64 KB	10.74	9.03
128 KB	7.51	8.62