Today's Lecture

1. Profiling

2. Debugging

3. GC (once again...)

4. Java Security (Drew Dean)

Profiling

Making programs run fast

- first, seek algorithmic improvements
- trade expensive operations for inexpensive operations
- avoid unnecessary recomputations
  (cache intermediate results, for example)
- understand your machine's memory hierarchy
- optimize data locality

"90% of execution time is spend in 10% of the code"

Profilers identify where the time's being spent:

1. `gprof`

2. `pixie`

How would you implement a profiler?

Unix `gprof`

- Execution based profiling
- Function-call granularity
  (some sub-functions support)
- PC-sampling
- Call-graph based
- Requires compilation with profiling-aware compilers

MIPS's `pixie`

- Instruction-level profile
- Instruments executables (`a.out`) with counters
- Does not require source or special compilation
- Highly architecture dependent

```
# instrument the main program
pixie myprog
# run the instrumented program
myprog.pixie
# profile using pixstats
pixstats myprog
# profile using prof
prof myprog -pixie
```

Sample pixie output


```
pixstats fib:
     22663 (1.055) cycles (0.0009065s @ 25.0MHz)
     21479 (1.000) instructions
      1184 (0.055) divide interlock cycles (35 cycles)
         0 (0.000) interlock cycles due to basic block boundary

      3950 (0.184) nops
      6816 (0.317) alu
      4237 (0.197) loads
      2710 (0.126) stores
      6947 (0.323) loads+stores
      6947 (0.323) data bus use
      1433 (0.067) partial word references
      4166 (0.194) sp+gp load/stores
         0 (0.000) flops (0 mflop/s @ 25.0MHz)
      2566 (0.119) conditional branches
       643 (0.030) branch to branch
       870 (0.041) branch delay filled with target-1 instruction
      4297 (0.200) basic blocks
       369 (0.017) calls
        38 (0.002) non-R31 JR
        45 (0.002) addui opportunities
         5 (0.000) skip

0.380 load nops per load
0.390 stores per memory reference
0.206 partial word references per reference
         1 write buffer overflow locations
        19 write buffer overflows

  5.0 instructions per basic block
  7.2 instructions per branch
0.118 backward branches per branch
0.686 branch nops per branch
0.167 SLTs per conditional branch
0.167 BEQ/BNE on SLT result per conditional branch
0.102 BEQ/BNE on LI result per conditional branch
61.4 cycles per call
58.2 instructions per call

40.7% load nops per nop
51.5% branch nops per nop
```



```
 3.7% jump nops per nop
 4.0% other nops per nop

Opcode distribution:
    spec      6882   32.04% *instruction class*
      lw      3118   14.52%   14.5%
      sw      2396   11.16%   25.7%
    bnop      2036    9.48%   35.2%
   addiu      1797    8.37%   43.5%
    addu      1667    7.76%   51.3%
    lnop      1609    7.49%   58.8%
    beqz      1304    6.07%   64.8%
     lbu      1107    5.15%   70.0%
      li      1052    4.90%   74.9%
    andi       968    4.51%   79.4%
    bnez       428    1.99%   81.4%
      jr       405    1.89%   83.3%
       b       401    1.87%   85.1%
     jal       369    1.72%   86.9%
     lui       359    1.67%   88.5%
      sb       313    1.46%   90.0%
     bne       285    1.33%   91.3%
     sll       271    1.26%   92.6%
     beq       253    1.18%   93.8%
    sltu       239    1.11%   94.9%
   bcond       199    0.93% *instruction class*
    bgez       199    0.93%   95.8%
     nop       159    0.74%   96.5%
    jnop       146    0.68%   97.2%
    subu       115    0.54%   97.8%
    blez        97    0.45%   98.2%
    slti        94    0.44%   98.6%
     slt        58    0.27%   98.9%
     and        57    0.27%   99.2%
   sltiu        38    0.18%   99.4%
    mflo        37    0.17%   99.5%
     div        37    0.17%   99.7%
 syscall        25    0.12%   99.8%
      or        19    0.09%   99.9%
      lb         8    0.04%   99.9%
    addi         3    0.01%  100.0%
     ori         2    0.01%  100.0%
     lwl         2    0.01%  100.0%
     lwr         2    0.01%  100.0%
     swl         1    0.00%  100.0%
```



```
     add          1    0.00%  100.0%
     sub          1    0.00%  100.0%
    move          1    0.00%  100.0%

    cycles %cycles  cum%     instrs  c/i     calls     c/call name
     9365  41.3%  41.3%       8181  1.1        19        493 _doprnt
     6604  29.1%  70.5%       6604  1.0       137         48 _flsbuf
     2443  10.8%  81.2%       2443  1.0       100         24 fclose
     1215   5.4%  86.6%       1215  1.0         1       1215 _cleanup
      900   4.0%  90.6%        900  1.0        20         45 _xflsbuf
      570   2.5%  93.1%        570  1.0        19         30 printf
      476   2.1%  95.2%        476  1.0        35         14 nvmatch
      372   1.6%  96.8%        372  1.0         1        372 getenv
      325   1.4%  98.3%        325  1.0         1        325 main
      120   0.5%  98.8%        120  1.0        20          6 _write
       44   0.2%  99.0%         44  1.0         1         44 _findbuf
       36   0.2%  99.1%         36  1.0         1         36 strcpy
       33   0.1%  99.3%         33  1.0         1         33 fflush
       32   0.1%  99.4%         32  1.0         1         32 _wrtchk
       26   0.1%  99.5%         26  1.0         1         26 __start
       26   0.1%  99.7%         26  1.0         1         26 _setchrclass
       18   0.1%  99.7%         18  1.0         1         18 _isatty
       18   0.1%  99.8%         18  1.0         3          6 _close
       13   0.1%  99.9%         13  1.0         1         13 __call_exitfns
       10   0.0%  99.9%         10  1.0         1         10 exit
        9   0.0% 100.0%          9  1.0         1          9 _ioctl
        4   0.0% 100.0%          4  1.0         1          4 _oserror
        2   0.0% 100.0%          2  1.0         1          2 __readenv_sigfpe
        2   0.0% 100.0%          2  1.0         1          2 _exit
```


Debugging

Making programs run (correctly)

- remove degrees of freedom
- find smallest input that triggers bug (binary search)
- view bug as a puzzle; deduce clues; "force" the program to reveal clues
- unexplained bugs that just "go away" are dangerous

`printf` is often the best debugger, but it's good to know something about *source-level* debuggers

Most debuggers allow:

- breakpoints
- memory inspection
  (in various formats)
- single-stepping
- execution traces
- "watched" variables
- interactive evaluation of expressions

How would you implement the above debugger features?


```
/* tiny-gc.c */
/* a very simple mark-sweep garbage collector */

#define M_SZ 1000

#define MARK 1
#define NO_MARK 0

#define JUNK_INT 12345678

#define IS_INT(i) (((int) i) & 0x1)
#define IS_PTR(p) (!IS_INT(p))

/* types */
typedef int tag_t;

union value {
  int i_val;
  struct cell *p_val;
};

typedef union value value_t;

struct cell {
  tag_t tag;
  value_t car, cdr;
};

typedef struct cell cell_t;

/* external global vars */
cell_t *root;

/* internal global vars */
static cell_t M[M_SZ];

void init()
{
  int i;

  for (i = 0; i < M_SZ; i++) {
    M[i].tag = NO_MARK;
```



```
    M[i].car.i_val = i+1;
    M[i].cdr.i_val = JUNK_INT;
  }
}

static void mark(cell_t *p)
{
  if (IS_PTR(p)) {
    if (p->tag == NO_MARK) {
      p->tag = MARK;
      mark(p->car.p_val);
      mark(p->cdr.p_val);
    }
  }
}

static int next_free_cell = 0;

static void sweep()
{
  int i;

  next_free_cell = M_SZ;
  for (i = 0; i < M_SZ; i++) {
    if (M[i].tag == NO_MARK) {
      M[i].car.i_val = next_free_cell;
      M[i].cdr.i_val = JUNK_INT;
      next_free_cell = i;
    } else {
      M[i].tag = NO_MARK;
    }
  }
}

static void gc()
{
  mark(root);
  sweep();
}

static int get_cell()
{
  int this_cell;

  if (next_free_cell == M_SZ) {
```



```
    gc();
    if (next_free_cell == M_SZ) {
      error("out of memory");
    }
  }
  this_cell = next_free_cell;
  next_free_cell = M[this_cell].car.i_val;
  return this_cell;
}

cell_t *alloc(cell_t *car,cell_t *cdr)
{
  int the_cell = get_cell();
  M[the_cell].car.p_val = car;
  M[the_cell].cdr.p_val = cdr;
  return &(M[the_cell]);
}

/* end of tiny-gc.c */
```

Loose ends; course evaluation

Discussion topics?