Writing Efficient Simulators

Gregory M. Oster
DISCUS Research Group
Department of Computer Science
University of Saskatchewan

October 25, 1999

Outline:

1. A Brief History of Computing
2. Writing Efficient Simulators
3. Memory Issues
4. Data Structures
5. Architecture Issues
6. Trace-File Issues
7. Finding the Bottlenecks
8. Conclusions

1. A Brief History of Computing

- Changing architectures
  - 4004, 8080, Z80, 6502, PDP-11, 8086
  - 80286, 68000, R2000, Sparc, HP PA-RISC
  - R3000, 386, 486, 68030, 68040, 21066
  - Pentium, Pentium II, AMD K6, Pentium III
  - UltraSparc, R10000, AMD K7, 21264

- CPU horsepower
  - KIPs to MIPs to BIPs
  - 1 MIP (Vax) to 2 BIP (skorpio)

- Changing RAM sizes
  - "small": 256 bytes, 1K, 4K, 16K, 64K
  - "large": 640K, 1MB, 2MB, 4MB, 8MB
  - "huge": 32MB, 64MB, 128MB, 256MB
  - "HUGE": 512MB, 1GB, 2GB, 4GB, 8GB

- Changing Disk sizes
  - 170K floppies
  - 5MB 'Winchester' disk technology
  - Terrabyte filesystems
  - 73GB SCSI disks

1. A Brief History of Computing (cont.)

- Early years: bytes were valuable!

- Programming Shift
  - less low-level, optimized code
  - more high-level code
    - compilers expected to optimize
  - integrated development environments
    - 'drag and drop' program construction
  - 'creeping featuritis' - more programs, more features, bigger programs
  - Bloated Software!
  - Vicious Circle:
    more RAM $\rightarrow$ bigger programs $\rightarrow$
    need more RAM $\rightarrow$ bigger programs ...

- Problems:
  - Complacency in software development
    - size of software is much less of an issue
  - Everyone needs more RAM, Disk, and CPU horsepower

1. A Brief History of Computing (cont.)

- What can we do?
  - Buy more hardware?
    - short-term solution
  - Write more efficient code?
    - long-term solution

- Simulation specific problems:
  - Focus: trace-driven simulation
- "Reasonable" machines:
  - Pentium II 400MHz CPU, 256MB of RAM, and 10GB of disk
- What's the problem?
  - Trace #1: 10M requests, 15 days, 1.4GB raw.
    - 500MB binary trace
    - simulator needs 100MB RAM
  - Trace #2: 80M requests, 5 days, 11.2GB raw.
    - 4GB binary trace
    - simulator needs ????MB RAM

2. Writing Efficient Simulators

- Analysing the data:
  - Scale of the data:
    - How much data needs to be processed?
      e.g. size of trace files
    - How might that amount change over time?
      e.g. larger traces
  - Nature of the algorithms
    - Are traditional algorithms sufficient?
      e.g. Is a simple O(n) lookup fast enough?
  - THINK about how the data and algorithms fit together
    - e.g. time vs. space trade-offs

3. Memory Issues

- Obtaining memory: variables vs. `new/malloc()`
- Where is memory allocated?
  1. Heap - global variables, `malloc()`ed structures
  2. Stack - local variables
  3. CPU Registers - local variables
- Out of memory blues. . .
  - `limit`
  - `limit datasize unlimited`
  - `limit stacksize unlimited`

cputime	unlimited	unlimited
filesize	unlimited	unlimited
datasize	131072 kbytes	1048576 kbytes
stacksize	2048 kbytes	32768 kbytes
coredumpsize	unlimited	unlimited
memoryuse	58664 kbytes	58668 kbytes
memorylocked	19556 kbytes	19556 kbytes
maxproc	1044	1044
openfiles	64	64

3. Memory Issues (cont.)

- Swapping and Paging
  - Virtual Memory
  - Potentially **very** CPU intensive
    - e.g. only get 2-3% of full CPU

- Who pays for all those `malloc()`'s?
  - malloc is expensive
    - call infrequently if possible
  - allocate only what you need
    - in advance?
  - re-use/re-cycle your own memory
  - free memory if it cannot be recycled
  - avoid memory leaks
    - balance `malloc()`'s and `free()`'s
  - avoid memory fragmentation

3. Memory Issues (cont.)

```
while(!done) {
  if ((space = malloc(...))==0) {...}
  initialize(space);
  ...
  free(space);
}
```

```
if ((space = malloc(...))==0) {...}
while(!done) {
  initialize(space);
  ...
}
free(space);
```

3. Memory Issues (cont.)

- Memory Banks
  - a tradeoff between:
    - "allocate everything at the beginning"
    - "allocate one-at-a-time as needed"
  - each bank is an array of structures
  - use `get_new_structure()` instead of `malloc()`
    - returns a pointer to an un-used structure
  - once an array (bank) is used up, a new bank is allocated
  - maintain a "free-list"
  - O(1) allocating and deallocating
  - Use only a small number of banks
    - tuning required

4. Data Structures

- Balanced binary trees
  - O(log n) insertions, lookup, deletions
  - difficult to maintain if allocated dynamically
    - easy for tree to become unbalanced if "complicated" algorithms are not used
  - very effective if implemented as linear arrays:

    Pros:
    - savings of two pointers per node
    - guarantee of O(log n) for insertions, deletions, and lookups
    - guarantee of balance
    - straightforward algorithms

    Cons:
    - likely need to know the size in advance

4. Data Structures (cont.)

- Singly linked lists
  - O(n) time to search, but can be O(1) for inserts or deletes (stacks/queues)
  - can be very efficient
    - cost of one pointer per data structure

- Doubly linked lists
  - very efficient for finding previous/next elements
  - not always needed - still O(n) searches
  - make sure the reverse link gets used!

- Heaps
  - especially useful for event-based simulators
  - event heaps: store pending events
  - if implemented as an array:
    - insertions are O(log n)
    - deletions are O(log n)
    - queries for next event time are O(1)

4. Data Structures (cont.)

- Hashes/Hashing
  - can be **much** faster than a linear or binary search
  - O(n + e) rather than O(n log n)
    - for n = 10,000,000 , a speedup of 23x

- Use of pointers
  - number of pointers needed depends on the algorithms used
  - ratio of pointers to data:
    - high: too many pointers?
    - low: slower algorithms?

4. Data Structures (cont.)

- Grouping Data
  - Which structure is larger?

```
struct foo1 {
  char a;
  int b;
  char c;
  int d;
  char e;
  char f;
};

struct foo2 {
  int b;
  int d;
  char a;
  char c;
  char e;
  char f;
};
```

4. Data Structures (cont.)

- Grouping Data (cont.)
  - Are all the fields being used?
  - Size of Variables
    - Do you really need an entire `int`?
      - signed vs. unsigned
    - Bit vectors
  - Be efficient:
    - removing 4 bytes from a trace record in a 10,000,000-event trace is 40MB

- Pre-compute Data
  - Are the calculations expensive?
    - Pre-compute values into a table
      e.g. `sin(10)`, `sin(20)`, `sin(30)`

4. Data Structures (cont.)

- Natural word size
  - typically more efficient
    e.g.:

    ```
    signed char x;
    for( x = -40 ; x < 100 ; x++ ) {...}
    ```

    - `int x` may be more efficient
  - use 32-bit integers on 32-bit systems
  - avoid size conversions
  - bus alignment issues?
    - long-word aligned (multiple of 4 or 8 bytes)
  - sign issues
  - variables in registers

5. Architecture Issues

- Endianness
  - Big Endian: 16909060 → `0x01020304`
  - Little Endian: 16909060 → `0x04030201`
- Pointer size
  - 32-bit
  - 64-bit
  - **`sizeof( int ) != sizeof( int * )`**
- Architecture independent code
- Parameter passing
  - pass pointers, not structures
  - may be expensive on some architectures

6. Trace-File Issues

- Prune trace files
  - remove unneeded data
    - save on disk space and IO time

- Binary vs. ASCII input
  - use a binary trace file where possible
  - `fread()` **much** faster than `fscanf()`
  - binary traces usually smaller
  - requires pre-processing

6. Trace-File Issues (cont.)

- Compressed Input Files
  - Smaller disk footprint
  - Less data to read from disk
    - important when disk IO is the bottleneck
  - Easy to do:

```
if (compressed) {
  sprintf(pipe,"gzip -dc %s",filename);
  if ((infile=popen(pipe,"r"))==NULL) {
    cerr << "Unable to open pipe for input.\n";
    exit(1);
  }
} else {
  if ((infile=fopen(filename,"r"))==NULL) {
    cerr << "Unable to open " << filename
         << " for input.\n";
    exit(1);
  }
}
```

6. Trace-File Issues (cont.)

- Multiple simulations on a single trace
  - Requires additional memory
  - Amortize the cost of reading the trace:
    e.g.
    - 30 minutes to read the trace (raw IO)
    - 10 minutes to simulate
    - 1 simulation: $30 + 10 = 40$ minutes
    - 3 simulations: $30 + 3 * 10 = 60$ minutes
    - average time: $60 / 3 = 20$ minutes.

- Endianness of traces
  - byte-swapping code

- Make sure trace files are on local disk

7. Finding the Bottlenecks

- All programs have a bottleneck
  - can only move bottlenecks around
- How to find the bottleneck?
  - Time to read raw trace vs simulation time
    e.g. 25 minutes to run entire simulation,
    22 minutes just to read the trace
  - Profilers: `gprof`
    - compile with `gcc -pg -g -o foo foo.c`
    - run `foo` (creates `gmon.out`)
    - `gprof` displays the profiling results
      - number of times a function was called
      - amount of CPU time spent in each call
      - ...

7. Finding the Bottlenecks (cont.)

- Simple source-code changes
  - Move calculations out of loops
    - pre-compute common parts of formulae

    e.g.:

    ```
    for( i=0; i < big_number; i++ ) {
      sum += a * b + c * d[i];
    }

    sum += a * b * big_number;
    for( i=0; i < big_number; i++ ) {
      sum += c * d[i];
    }
    ```
  - put most likely cases of `if` and `case` statements first
  - "Short circuiting"

    e.g.: `if ( (a==1) && (b<2) && (c>4) )`

7. Finding the Bottlenecks (cont.)

- Simulator output bottlenecks
  - Excessive output?
    - compare the time for a normal run with:
      time mysimulator >& /dev/null
    - much of the "simulation" time may be just output printing time
  - Solutions:
    - display status indicators less frequently
    - only print results at the end of the run
    - write output to a (compressed) file

8. Conclusions

- All programs have bottlenecks
- Software needs to be designed for efficiency
- Analysis of the relationship between algorithms and the data is critical

More Information:

http://www.cs.usask.ca/staff/oster/writing_efficient_software.html

Questions?