freescale™

Benchmarking

Diana Crăciun

Software Engineer, Virtualization Team

June 2013

What is a benchmark?

- A benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative **performance** of an object, normally by running a number of standard tests and trials against it. (Wikipedia)

Why do we need benchmarks?

- Compare the performance of different systems
- Asses the effect of new features for the same product
- Asses the performance of subsystems

Benchmarks vs profiling

- Profiling
  - Method of diagnosing performance bottlenecks
- Benchmarking
  - Used to determine performance
  - Determines how well an application performs given a specific configuration

Benchmark types (1)

- Component level benchmarks (microbenchmarks)
  - Test a specific component of the system
    - Memory subsystem
    - Disk subsystem
    - Network
- System level benchmarks
  - Evaluate the overall performance
  - Take into account the interaction between different subsystems

Benchmark types (2)

- Synthetic benchmarks
  - Component level benchmarks
- Application benchmarks
  - Real applications
  - (in general) system benchmarks
  - Take into account the interaction between different subsystems

Developing/choosing a benchmark

- Decide between system level benchmark and component benchmark
  - subsystems of the system
  - hardware & software configurations
  - technology used to develop the system
  - the architecture of the system
- Workload
  - What is the system executing when running the benchmark?

Benchmarks in embedded (example)

- OS benchmarks
  - lmbench
- IO benchmarks
  - IOzone
- Network benchmarks
  - Netperf
  - Iperf
- CPU benchmarks
  - Dhrystone
  - CoreMark
- Others

Dhrystone

- Synthetic benchmark
- Developed in 1984
- Integer performance
- Used for comparison between different CPUs
- Very dependent on the compiler and standard libraries
  - Compiler optimization affects the results
  - Standard library (e.g. malloc, memcpy, strcmp) affects the results

EEMBC CoreMark

- Lists, strings and arrays (matrix)
- Basic data structures and algorithms common in many applications
- Can test also cache & memory hierarchy
  - The list size should be carefully chosen
- List processing
  - Reversing, sorting, searching according to different parameters
  - It does not use malloc
    - Malloc is not widely used in systems with memory constraints
  - Non-serial access patterns

EEMBC CoreMark (cont'd)

- Matrix manipulation
  - Multiplication with a constant, vector, another matrix
- State machine processing
  - Control statements (switch and if)

LMbench

- OS benchmark
- Bandwidth benchmarks
  - File read
  - Memory copy
  - Memory read
  - Memory write
  - Pipe
  - TCP
- Latency benchmarks
  - Context switching
  - Networking: connection establishment, TCP, UDP
  - File system creates and deletes
  - process creation
  - Signal handling
  - System call overhead
  - Memory read latency
- Miscellaneous
  - Processor clock rate calculation

Lat_mem_rd

- Memory read latency
- The entire memory hierarchy is measured:
  - Cache, main memory, TLB misses
- lat_mem_rd <arraysize> <stridesize1> <stridesize2> …
  - Arraysize is in MB and should be larger than processor cache
- Accesses an array in a loop using a stridesize step
- Can be used to determine cache sizes

Context switch

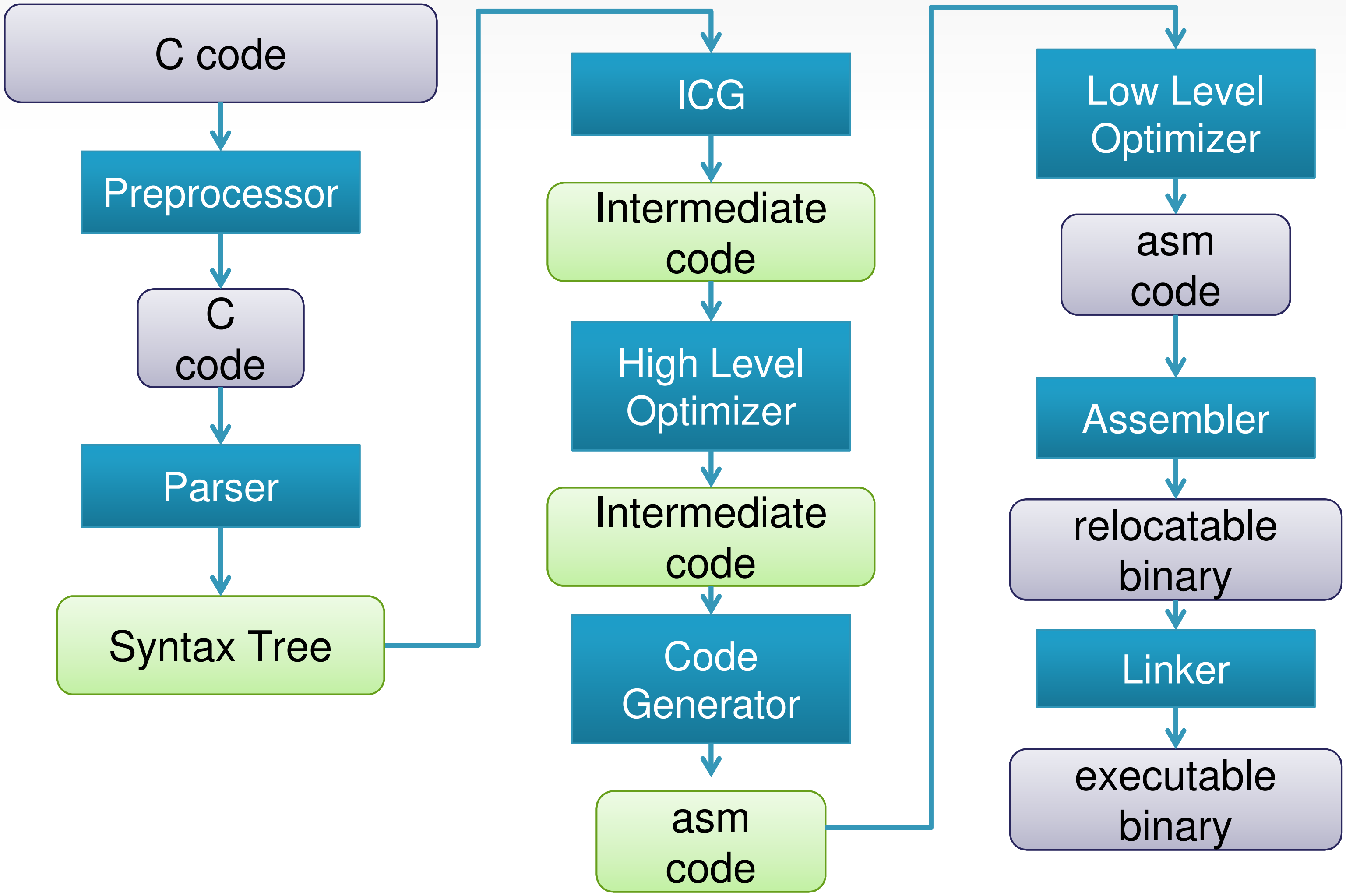
- Lat_ctx
- Test case creates a number of processes
- A ring of pipes is created between the processes
- When receives the token each process is reading some data (with sizes first = 4k, next=next*2, last = 64K)
- For each data size, the test is repeated with different number of processes.
- The parent processes generates the first token. The test for the current number of processes ends when all children finish.
- When the parent finishes kills all the children and closes the pipes

What to consider when benchmarking?

- Is the benchmark representative for your type of application?
- Is the benchmark proper configured?
  - Compare “apples to apples”
  - Things to consider: processor caches, compilers, libraries, system load
- Are the results consistent for different runs?

freescale™

From C to machine code



freescale™

Optimizing compiler: example #1

```c
static const int arr[] = {1, 7, 3};

int bar(unsigned int p1) {
        if (p1 < 0) { // always false ;-)
                return arr[0];
        }
        return arr[1];
}
```

O0

```
_bar:
    pushq       %rbp
    movq %rsp, %rbp
    movl %edi, -4(%rbp)
    movl _arr+4(%rip),%eax
    movl %eax, -12(%rbp)
    movl -12(%rbp), %eax
    movl %eax, -8(%rbp)
    movl -8(%rbp), %eax
    popq %rbp
    ret
```

O3

```
_bar:
    pushq       %rbp
    movq %rsp, %rbp
    movl $7, %eax
    popq %rbp
    ret
```

Optimizing compiler: example #1

```c
int foo(int x) {
        int i = 0;
        int j = x;
        for (i = 0; i < x; i++) {
                j = j * 2;
        }
        return j;
}

int bar(int y) {
        return foo(8);
}
```

O0

```
_bar:
        ;...
        callq   _foo
        ;...
        ret
```

O3

```
_bar:
    pushq       %rbp
    movq %rsp, %rbp
    movl $2048, %eax
    popq %rbp
    ret
```

freescale™