OSX System Profiling

Rebecca McKinley
University of California, San Diego
rmckinle@eng.ucsd.edu

Daniel Reznikov
University of California, San Diego
drezniko@eng.ucsd.edu

Aaron Trefler
University of California, San Diego
atrefler@eng.ucsd.edu

ABSTRACT

The characteristic meta-challenge of profiling Computer Systems is the ability to isolate system components, control dependencies and optimizations, and to conduct well-defined, repeatable experimentation. The goal of this project is profile the OSX Operation System.

1 INTRODUCTION

We seek to measure the performance of our PC system components including CPU, RAM, disk and the network. We implement the supporting experimentation code in the language C as it is low-level enough to allow us to control for many system optimization, and at times to execute raw X86 Assembly instructions.

Our team members contribute equally to each part of the project. For each milestone, we would walk-through the design of each experiment together, and implement equal parts individually. We use Github for source control.

2 MACHINE DESCRIPTION

Using the command *sysctl hw* and the *System Information* OSX utility, we are able to identify the characteristic metrics of each system component which impacts our performance profiling. The specification summary is detailed in Table 1.

Table 1: Machine Specification

Component	Specification
CPU Model	Intel Core i7, 2.7 GHz, Dual Core
Cycle Time	(1/2.7GHz) = 0.38ns
L1 Cache	Instruction Cache: 32KB (per core) Data Cache: 32KB (per core)
L2 Cache	256KB (per core) Level 2 Cache
L3 Cache	6MB Total shared Level 3 Cache
RAM Size	16GB (2 Banks of 8GB DDR3)
Instruction Set	x86-64
Memory Bus	Type: DDR3 Speed: 1600MHz Width: 64-bit
I/O Bus	Interconnect: SATA Link Speed: 6 Gigabit AHCI Version 1.30 Supported
Disk	Capacity: 500GB Type: SSD Mode: APPLE SSD SD512E
Network Card	Card Type: AirPort Extreme (0x14E4, 0xEF) Firmware Version: Broadcom BCM43xx 1.0
Operating System	OSX 10.12

3 CPU OPERATIONS

3.1 Read Time Overhead

The x86 Instruction Set Architecture (ISA) supports an operation which allows the processor to increment the a register value at every clock cycle. It is well known ([1], [2]) that, having isolated environment optimizations as described in, a ***rdtsc()*** method can be implemented in x86 assembly as follows:

```
static inline uint64_t
rdtsc(void) {
   uint32_t eax = 0, edx;

   __asm__ __volatile__("cpuid;" "rdtsc;" : "+a" (eax), "=d" (edx) : : "%rcx", "%rbx", "memory");

   __asm__ __volatile__("xorl %%eax, %%eax;" "cpuid;" : : : "%rax", "%rbx", "%rcx", "%rdx", "memory");

   return (((uint64_t)edx << 32) | eax);
}
```

To profile the overhead of reading time, we execute an experiment which makes two consecutive calls to ***rdtsc()***, computes the number of elapsed cycles, and adds this to a running total which iterates this procedure 10M times. Aggregating over 10 experiment runs, we find the operation takes $avg(186) \pm \sigma(5.34)$ cycles.

To profile the overhead of using a loop to measure many iterations of an operation, we orchestrate an experiment which executes an empty loop body 10M times, wrapping the loop in ***rdtsc()*** calls. Aggregating over 10 experiment runs, we find that the overhead associated with a single loop iteration is $avg(5.4) \pm \sigma(0.663325)$ cycles.

4 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

[1] [n. d.]. x86 instruction set reference - rdtsc. ([n. d.]). https://c9x.me/x86/html/file_module_x86_id_278.html
[2] Gabriele Paoloni. 2010. How to Benchmark Code Execution Times on IntelÂő IA-32 and IA-64 Instruction Set Architectures. (Sep 2010). Intel.com