High-Performance Computing (Custom)

Abstract:

The goal of this course is to explore Modern Computer Hardware in explicit detail. Modern computer hardware is a very broad subject. For a more **incisive** coverage, the course focuses on "server-class" X86 architecture-based systems. The aspects of the exploration are:

1. The underlying concept of the hardware. (for e.g. CPU)
2. How the software interfaces with it. (for e.g. How does the OS interface with the CPU)
3. Tools used to measure the effect.

If the hardware is understood in the right manner, then it can be included in the design phase of application development. This impacts the performance (latency and throughput) greatly.

Brief Table of contents

1. Machine-Level Representation of Programs

(This section shows how the program representation and its use of architectural registers of the system. Tools like ftrace can be used to verify consolidate the understanding.)

2. Introduction to Micro-Architectures

(This section establishes the core concepts of the course. Focuses on the internals of a microprocessor core, especially the flow of instructions and code. This will be the most involving and eventually the most satisfying section. Post this section, one will be able to visualize program execution on a CPU quite clearly.)

3. Optimizing Code for Performance

(Microoptimizing Code for Performance, based on Section-2. This is specifically to verify the theories and assumptions in Section-2. For e.g. utilization of one's knowledge of CPU pipeline and superscalar architecture to gain performance.)

4. Storage

(Solid idea of RAM, Cache, and Disk. More importantly, exploit the cache with cache-aware data structures and hide the CPU-memory GAP. Similarly, the OS page cache to DISK read/write gap.)

5. Prefetchers

(Prefetchers can optimize the memory reads if the read patterns can be understood.)

6. Virtual Memory

(Virtual memory has a lot of uses but here we explore the optimizations done for disk reads and writes using OS page cache.)

7. MicroProcessor Components

(Linux interface to CPU and how can we use it to improve performance(Latency and throughput). Understanding the CPU driver in Linux. Understanding the process of OS interfacing with the CPU.)

8. Microprocessor Performance and energy

(This is the practical part of section-7. Power and performance(Latency and throughput) are interrelated. This section explores the trade-off with examples.)

9. Multicore architectures

(Multiprocessor systems and the effects of cache coherency. Multithreaded programming requires the idea of shared data in the presence of multicore systems.)

10. Distributed Computing

(In this section we change the approach a bit. We take an open-source, high performance(Ultra-Low-Latency) messaging framework and dissect it. We see an industry-standard implementation of the majority of the concepts below, including queuing theory, event-based architecture, etc. On one hand, we explore the adaptation of the framework on Solarflare like hardware, and on the other hand, we explore making the framework reactive.)

11. Tools

Table of contents

1. Machine-Level Representation of Programs

(This section shows how the program representation and its use of architectural registers of the system. Tools like ftrace can be used to verify consolidate the understanding.)

2. Introduction to Micro-Architectures

(This section establishes the core concepts of the course. Focuses on the internals of a microprocessor core, especially the flow of instructions and code. This will be the most involving and eventually the most satisfying section. Post this section, one will be able to visualize program execution on a CPU quite clearly.)

- Hardware Stages
- PipeLines
    - Real World Pipelines
    - Computational Example
    - 3-Way Pipeline
    - Operating a Pipeline
    - Non Uniform Delays
    - Register Overhead
    - Data Dependencies
    - Data Hazards
    - Data Dependencies in Processors
    - Pipeline Demonstration
    - Nops
    - Stalling for Data Dependency
    - Stall Conditions
    - Detecting Stall Conditions
    - What Happens When Stalling?
    - Data Forwarding
    - Data Forwarding Example
    - Forwarding Priority
    - Limitation of Priority
    - Avoiding Load/Use Hazard
    - Detecting Load/Use Hazard
    - Control of Load/Use Hazard
    - Modern CPU Design
    - Instruction Control
    - Execution units
- Superscalar Units
    - Superscalar Execution
    - In-Order Superscalar Processor Example
    - Superscalar Performance with Dependencies
    - Superscalar Execution Tradeoffs
- Branch Prediction
    - The Branch Problem
    - Importance of The Branch Problem
    - Branch Prediction
    - Branch Prediction: Guess the Next Instruction to Fetch
    - Misprediction Penalty
    - Simplest: Always Guess NextPC = PC + 4
    - Pipeline Flush on a Misprediction
    - Performance Analysis
    - Reducing Branch Misprediction Penalty
    - Two-Level Prediction

Global Branch Correlation
Hybrid Branch Predictors
Case Study
Sandy Bridge
Haswell

Tools: Perf for event based hardware profiling and an even more fingrained tool (**overseer**) that can be integrated into the application.

3. Optimizing Code for Performance

(Microoptimizing Code for Performance, based on Section-2. This is specifically to verify the theories and assumptions in Section-2. For e.g. utilization of one's knowledge of CPU pipeline and superscalar architecture to gain performance.)

Optimization Realities
Optimizing Compilers
Limitations of Optimizing Compilers
Optimization Blockers
Memory Aliasing
Procedure Calls
Cycles Per Element (CPE)
Optimization examples
Removing loop inefficiency
Procedure Calls
Lower Case Conversion Performance
Convert Loop To Goto Form
Understanding Modern CPU:Haswell
Loop Unrolling
Going Superscalar
Re-association
SSE and Friends
Limiting Factors
Branch Prediction
RegisterSpilling

4. Storage

(Solid idea of RAM, Cache, and Disk. More importantly, exploit the cache with cache-aware data structures and hide the CPU-memory GAP. Similarly, the OS page cache to DISK read/write gap.)

(This section is to understand "CPU Memory Gap". This can be plugged with cache.)
Memory hierarchies
RAM
SRAM vs DRAM
Non Volatile Memory
Traditional Bus Structure Connecting CPU and Memory
Memory Read Transaction

- Memory Write Transaction
- CPU Memory Gap
  - Locality to the Rescue!
  - Qualitative Estimates of Locality
  - Memory Hierarchies
  - Example Memory Hierarchy
- Conventional DRAM Organization
- Reading DRAM Supercell
- Memory Modules
- Enhanced DRAMs
- Storage Trends
- Clock Rates

Caches
- General Cache Concepts
- Types of Cache Misses
- Examples of Caching in the Mem. Hierarchy
- General Cache Organization
- Cache Reads
- Direct Mapped Cache
- Direct-Mapped Cache Simulation
- E-way Set Associative Cache
- What about writes?
- Intel Core i7 Cache Hierarchy
- Cache Performance Metrics
- Writing Cache Friendly Code
- The Memory Mountain
- Memory Mountain Test
- Matrix Multiplication Example
- Miss Rate Analysis for Matrix Multiply
- Core i7 Matrix Multiply Performance
- Layout of C Arrays in Memory (review)
- Cache Miss Analysis
- Blocked Matrix Multiplication

Disk
- What's Inside A Disk Drive?
- Disk Geometry
- Disk Geometry (Muliple-Platter View)
- Disk Capacity
- Recording zones

Tools:

Perf hardware profiling to measure various cache/memory/disk metrics. This is done to check if a particular design is taking effect.

EBPF based customized tools can be built to show the impact of the above.

5. Prefetchers

(Prefetchers can optimize the memory reads if the read patterns can be understood.)

Tolerating Memory Latency

- Caching
- Prefetching
- Multithreading
- Out-Of-Order Execution

Prefetching and Correctness

How a HW Prefetcher Fits in the Memory System

Prefetching: The Four Questions

Software Prefetching

X86 PREFETCH Instruction

Next-Line Prefetchers

Stride Prefetchers

Instruction Based Stride Prefetching

Prefetcher Performance

Tool: Same as section-4 but exact metrics can differ.

6. Virtual Memory

(Virtual memory has a lot of uses but here we explore the optimizations done for disk reads and writes using OS page cache.)

A System Using Physical Addressing

Address Spaces

Why Virtual Memory (VM)?

VM as a Tool for Caching

DRAM Cache Organization

Enabling Data Structure: Page Table

Page Hit

Page Fault

Handling Page Fault

Allocating Pages

Locality to the Rescue Again!

VM as a Tool for Memory Management

Simplifying Linking and Loading

VM as a Tool for Memory Protection

VM Address Translation

Address Translation: Page Hit

Address Translation: Page Fault

Integrating VM and Cache

Speeding up Translation with a TLB

Accessing the TLB

TLB Hit
TLB Miss
Programmers View of Virtual Memory
System's View of Virtual Memory
Tool: Same as section-4 but exact metrics can differ.

7. MicroProcessor Components

(Linux interface to CPU and how can we use it to improve performance(Latency and throughput). Understanding the CPU driver in Linux. Understanding the process of OS interfacing with the CPU.)

Core
- Sandy Bridge Pipeline:Frontend(Instruction load, decode, cache)
- Sandy Bridge Pipeline:Execution
- Sandy Bridge Pipeline:Backend (Data load and Store)
- Haswell Pipeline

Uncore
- L3 Cache
- Integrated Graphics
- Integrated memory controller
- QuickPath Interconnect

Linux Inteface to CPUIDLE
- CPUIDLE subsystem
- CPUIDLE subsystem:Driver load
- CPUIDLE subsystem:Call the Driver
- CPUIDLE subsystem:Governor
- CPUIDLE subsystem:Gathering and undertanding latency data

8. Microprocessor Performance and energy

(This is the practical part of section-7. Power and performance(Latency and throughput) are interrelated. This section explores the trade-off with examples.)

Power
- Power:Turn things off
- Power:c-states
- Power:Tuned
- Power:Tuned:c-states requests
- Power:Tuned:Hardware State Residency
- Power:Tuned:Influx:Grafana:c-states
- Power:Tuned:Measuring Latency
- Power:Tuned:Hardware Latency
- Power:Tuned:Wakeup Latency
- Power:Tuned:Influx:Grafana:latency
- Power:PMQOS
- Power:Turn things down

Power:Turn things down:P-states:Hardware Latency

Core and Uncore:Uncore

Core and Uncore:Uncore:montioring and Tuning

Core and Uncore:Uncore:montioring and Tuning:Hardware Latency

Core and Uncore:Uncore:montioring and Tuning:Wakeup Latency

Core and Uncore:Uncore:montioring and Tuning:Application Latency

Tools: The primary tool here is **ftrace** but it majorly used to gather data as it is the **lowest latency** flow-based tracer. This data is then post-processed to visualize it.

9. Multicore architectures

(Multiprocessor systems and the effects of cache coherency. Multithreaded programming requires the idea of shared data in the presence of multicore systems.)

Introduction

Multiprocessing

Cache Coherence

Flynn's Taxonomy of Computers

Why Parallel Computers?

Types of Parallelism and How to Exploit Them

Task-Level Parallelism: Creating Tasks

Multiprocessing Fundamentals

Multiprocessor Types

Main Issues in Tightly-Coupled MP

Hardware-based Multithreading

Parallel Speedup Example

Speedup with N Processors

Revisiting the Single-Processor Algorithm

Superlinear Speedup

Utilization, Redundancy, Efficiency

Utilization of a Multiprocessor

Caveats of Parallelism

Amdahl's Law

Sequential Bottleneck

Why the Sequential Bottleneck?

Bottlenecks in Parallel Portion

Cache Coherence

Introduction

Multi-Core Cache Coherence

Memory Ordering in Multiprocessors

Ordering of Operations

Memory Ordering in a Single Processor

Memory Ordering in a Dataflow Processor

Memory Ordering in a MIMD Processor

Why Does This Even Matter?
Protecting Shared Data
Supporting Mutual Exclusion
Sequential Consistency
Programmer's Abstraction
Issues with Sequential Consistency?
Weaker Memory Consistency
Tradeoffs: Weaker Consistency
Cache Coherence
Shared Memory Model
The Cache Coherence Problem
Hardware Cache Coherence
Two Cache Coherence Methods
Snoopy Cache Coherence
MESI

Tools: Measure the effects of cache coherency with tools like **perf, Solaris Analyzer**, and **overseer** e.t.c.

10. Tools

Tools:**EBPF**

(One of the best tool for customizing tracing of kernel code. This gives Linux tracing superpowers beyond even Solaris.)

Tools:**Ftrace**

(The best (least overhead) software flow-based tracer for Linux. It is a part of Linux Kernel and its official tracer.)

Tools:**Perf**

(One of the best event-based profilers in the business.)

Tools:**Solaris Analyzer**
Tools:**Overseer**
Tools:**pcm-master**
Tools:**Core-freq**
Tools:**Powertop**
Tools:**Turbostat**

11. Distributed Computing

(In this section we change the approach a bit. We take an open-source, high performance messaging framework and dissect it. We see an industry-standard implementation of the majority of the concepts below, including queuing theory, event-based architecture, etc. On one hand, we explore the adaptation of the framework on Solarflare like hardware, and on the other hand, we explore making the framework reactive.)

Introduction
Hardware Networking stack
Software Networking stack

- Modes of parallelism
- Distributed Memory Models
- Understanding the cost of Communication vs Computation
- OpenMP, MPI overview

Scheduling considerations for distributed computing

- Intro to Queueing Theory
- Latency vs Throughput
- Scheduling to meet SLAs
- Fault-Tolerant execution

Approaches to distributed computing

- Message Passing
- Event-driven approach to grid computing