CMPT 745
Software Engineering

Measurement & Performance

Nick Sumner
wsumner@sfu.ca
Performance & Measurement

- Real development must manage resources
Performance & Measurement

• Real development must manage resources
  – Time
  – Memory
  – Open connections
  – VM instances
  – Energy consumption
  – ...

Performance & Measurement

- Real development must manage resources
  - Time
  - Memory
  - Open connections
  - VM instances
  - Energy consumption
  - ...

- Resource usage is one form of performance
  - *Performance* – a measure of nonfunctional behavior of a program
Performance & Measurement

- Real development must manage resources
  - Time
  - Memory
  - Open connections
  - VM instances
  - Energy consumption
  - ...

- Resource usage is one form of performance
  - *Performance* – a measure of nonfunctional behavior of a program

- We often need to assess performance or a change in performance
  Data Structure A  vs  Data Structure B
Performance & Measurement

- Real development must manage resources
  - Time
  - Memory
  - Open connections
  - VM instances
  - Energy consumption
  - ...

- Resource usage is one form of performance
  - *Performance* – a measure of nonfunctional behavior of a program

- We often need to assess performance or a change in performance
  
  Data Structure A vs Data Structure B

  How would you approach this in a data structures course?
Performance & Measurement

- Performance assessment is deceptively hard
  [Demo/Exercise]
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
  - Even with the best intentions we can deceive ourselves
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
  - Even with the best intentions we can deceive ourselves

- Good performance evaluation should be rigorous & scientific
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
  - Even with the best intentions we can deceive ourselves

- **Good performance evaluation should be rigorous & scientific**
  - The same process applies in development as in *good* research
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
  - Even with the best intentions we can deceive ourselves

- Good performance evaluation should be rigorous & scientific
  - The same process applies in development as in *good* research
Performance & Measurement

• Performance assessment is deceptively hard
  – Modern systems involve complex actors
  – Theoretical models may be too approximate
  – Even with the best intentions we can deceive ourselves

• Good performance evaluation should be rigorous & scientific
  – The same process applies in development as in good research
    1) Clear claims
    2) Clear evidence
    3) Correct reasoning from evidence to claims
Performance & Measurement

- Performance assessment is deceptively hard
  - Modern systems involve complex actors
  - Theoretical models may be too approximate
  - Even with the best intentions we can deceive ourselves

- Good performance evaluation should be rigorous & scientific
  - The same process applies in development as in *good* research
    1) Clear claims
    2) Clear evidence
    3) Correct reasoning from evidence to claims
  - And yet this is challenging to get right!
Performance & Measurement [Blackburn et al.]

Sin of Exposition (of Evaluation): Irreproducibility

Sin of Exposition (of Claim): Inscrutability

Sins of Reasoning (Derive Claim):
Ignorance,
Inappropriateness,
Inconsistency

Evaluation → Claim
Performance & Measurement [Blackburn et al.]

Scope of Evaluation

Sin of Exposition (of Evaluation): Irreproducibility

Claim

Scope of Claim/Conclusion

Sins of Reasoning (Derive Claim): Ignorance, Inappropriateness, Inconsistency

Sin of Exposition (of Claim): Inscrutability

Consumer
Performance & Measurement [Blackburn et al.]

- Scope of Evaluation
- Validity
- Scope of Claim/Conclusion
Inscrutability
- Lack of clarity on actors or relationships
- Omission, Ambiguity, Distortion
Performance & Measurement [Blackburn et al.]

- **Inscrutability**
  - Lack of clarity on actors or relationships
  - Omission, Ambiguity, Distortion

- **Irreproducibility**
  - Lack of clarity in steps taken or data
  - Causes:
    - Omission of steps
Performance & Measurement [Blackburn et al.]

- **Inscrutability**
  - Lack of clarity on actors or relationships
  - Omission, Ambiguity, Distortion

- **Irreproducibility**
  - Lack of clarity in steps taken or data
  - Causes:
    - Omission of steps
    - Incomplete understanding of factors
Performance & Measurement [Blackburn et al.]

- **Inscrutability**
  - Lack of clarity on actors or relationships
  - Omission, Ambiguity, Distortion

- **Irreproducibility**
  - Lack of clarity in steps taken or data
  - Causes:
    - Omission of steps
    - Incomplete understanding of factors
    - Confidentiality & omission of data

Example ...
static int i = 0, j = 0, k = 0;
int main() {
    int g = 0, inc = 1;
    for (; g<65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
static int i = 0, j = 0, k = 0;
int main() {
    int g = 0, inc = 1;
    for (; g<65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}

Compare gcc -O2 vs -O3

One person may see a deterministic improvement..
static int i = 0, j = 0, k = 0;
int main() {
    int g = 0, inc = 1;
    for (; g<65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}

Compare gcc -O2 vs -O3

One person may see a deterministic improvement..

Another may see a deterministic degradation.
static int i = 0, j = 0, k = 0;

int main() {
    int g = 0, inc = 1;
    for (; g < 65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}

Compare gcc -O2 vs -O3

One person may see a deterministic improvement..

Another may see a deterministic degradation.

Both are right.
static int i = 0, j = 0, k = 0;
int main() {
    int g = 0, inc = 1;
    for (; g<65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}

Compare gcc -O2 vs -O3
Performance & Measurement [Blackburn et al.]

- Ignorance – disregarding data or evidence against a claim
  - Ignoring data points
Performance & Measurement [Blackburn et al.]

- Ignorance – disregarding data or evidence against a claim
  - Ignoring data points

**best of 30**

![Bar chart showing execution time (s) for different methods: CopyMS, GenCopy, GenMS, MarkSweep, SemiSpace. The chart indicates GenMS has the best performance.]

**mean w/ 95% confidence interval**

![Bar chart showing mean execution time (s) with 95% confidence intervals for the same methods as above. The chart shows GenMS has the best performance.]
• Ignorance – disregarding data or evidence against a claim
  – Ignoring data points

- best of 30
- mean w/ 95% confidence interval
Ignorance – disregarding data or evidence against a claim
- Ignoring data points

- **best of 30**

- **mean w/ 95% confidence interval**
Performance & Measurement [Blackburn et al.]

- Ignorance – disregarding data or evidence against a claim
  - Ignoring data points
  - Ignoring distributions
Performance & Measurement [Blackburn et al.]

- Ignorance – disregarding data or evidence against a claim
  - Ignoring data points
  - Ignoring distributions

Gmail latency

![Gmail latency chart](chart.png)
Performance & Measurement [Blackburn et al.]

- Ignorance – disregarding data or evidence against a claim
  - Ignoring data points
  - Ignoring distributions

Gmail latency

If we reason about average latency, why is it misleading?
Ignorance – disregarding data or evidence against a claim
  – Ignoring data points
  – Ignoring distributions

If we reason about average latency, why is it misleading?

What is better?
Inappropriateness – claim is derived from facts not present
Inappropriateness – claim is derived from facts not present
  – Bad metrics (e.g. execution time vs. power)
Performance & Measurement [Blackburn et al.]

- Inappropriateness – claim is derived from facts not present
  - Bad metrics
  - Biased samples
Inappropriateness – claim is derived from facts not present
- Bad metrics
- Biased samples
- ...
Performance & Measurement [Blackburn et al.]

- Inappropriateness – claim is derived from facts not present
  - Bad metrics
  - Biased samples
  - ...

- Inconsistency – comparing apples to oranges
Performance & Measurement [Blackburn et al.]

- Inappropriateness – claim is derived from facts not present
  - Bad metrics
  - Biased samples
  - ...

- Inconsistency – comparing apples to oranges
  - Workload variation (e.g. learner effects, time of day)
Performance & Measurement [Blackburn et al.]

- **Inappropriateness** – claim is derived from facts not present
  - Bad metrics
  - Biased samples
  - ...

- **Inconsistency** – comparing apples to oranges
  - Workload variation (e.g. learner effects, time of day)
  - Incompatible measures (e.g. performance counters across platforms)
Assessing Performance
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement
- Assessing performance is done through benchmarking
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement

- **Assessing performance is done through benchmarking**
  - *Microbenchmarks*
    - Focus on cost of an operation in isolation
    - Can help identify core performance details & explain causes
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement

- **Assessing performance is done through benchmarking**
  - *Microbenchmarks*
    - Focus on cost of an operation in isolation
    - Can help identify core performance details & explain causes
  - *Macrobenchmarks*
    - Real world system performance
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement

- **Assessing performance is done through benchmarking**
  - *Microbenchmarks*
    - Focus on cost of an operation in isolation
    - Can help identify core performance details & explain causes
  - *Macrobenchmarks*
    - Real world system performance

- Workloads (inputs) must be chosen carefully either way.
  - representative, pathological, scenario driven, ...
Benchmarking

- We must reason rigorously about performance during assessment, investigation, & improvement

- **Assessing performance is done through benchmarking**
  - *Microbenchmarks*
    - Focus on cost of an operation in isolation
    - Can help identify core performance details & explain causes
  - *Macrobenchmarks*
    - Real world system performance

- Workloads (inputs) must be chosen carefully either way.
  - representative, pathological, scenario driven, ...

Let’s dig into a common approach to consider issues
Suppose we want to run a microbenchmark

```java
startTime = getCurrentTimeInSeconds();
doWorkloadOfInterest();
endTime = getCurrentTimeInSeconds();
reportResult(endTime - startTime);
```
Suppose we want to run a microbenchmark

```java
startTime = getCurrentTimeInSeconds();
doWorkloadOfInterest();
endTime = getCurrentTimeInSeconds();
reportResult(endTime - startTime);
```

What possible issues do you observe?
Suppose we want to run a microbenchmark:

```java
startTime = getCurrentTimeInSeconds();
doWorkloadOfInterest();
endTime = getCurrentTimeInSeconds();
reportResult(endTime – startTime);
```

- Granularity of measurement
- Warm up effects
- Nondeterminism
- Size of workload
- System interference
- Frequency scaling?
- Interference of other workloads?
- Alignment?
Benchmarking

- **Granularity & Units**
  - Why is granularity a problem?
  - What are alternatives to `getCurrentTimeInSeconds()`?
BENCHMARKING

- **Granularity & Units**
  - Why is granularity a problem?
  - What are alternatives to `getCurrentTimeInSeconds()`?
  - What if I want to predict performance on a different machine?
Benchmarking

- **Granularity & Units**
  - Why is granularity a problem?
  - What are alternatives to `getCurrentTimeInSeconds()`?
  - What if I want to predict performance on a different machine?
    - Using *cycles* instead of wall clock time can be useful, but has its own limitations
Benchmarking

- Warm up time
  - Why is warm up time necessary in general?
BENCHMARKING

- **Warm up time**
  - Why is warm up time necessary *in general*?
  - Why is it especially problematic for systems like Java?
Benchmarking

- Warm up time
  - Why is warm up time necessary in general?
  - Why is it especially problematic for systems like Java?
  - How can we modify our example to facilitate this?
BENCHMARKING

- **Warm up time**
  - Why is warm up time necessary *in general*?
  - Why is it especially problematic for systems like Java?
  - How can we modify our example to facilitate this?

```java
for (...) doWorkloadOfInterest();
startTime = getCurrentTimeInSeconds();
doWorkloadOfInterest();
endTime = getCurrentTimeInSeconds();
reportResult(endTime – startTime);
```
Benchmarks

- **Nondeterministic behavior**
  - Will `getCurrentTimeInSeconds()` always return the same number?

  Why/why not?
Benchmarking

- **Nondeterministic behavior**
  - Will `getCurrentTimeInSeconds()` always return the same number?
  - So what reflects a *meaningful* result?
    - Hint: The Law of Large Numbers!
Benchmarking

• Nondeterministic behavior
  – Will `getCurrentTimeInSeconds()` always return the same number?
  – So what reflects a *meaningful* result?
    • Hint: The Law of Large Numbers!

• By running the same test many times, the arithmetic mean will converge on the expected value
Benchmarking

- Nondeterministic behavior
  - Will `get_current_time_in_seconds()` always return the same number?
  - So what reflects a meaningful result?
    - Hint: The Law of Large Numbers!

- By running the same test many times, the arithmetic mean will converge on the expected value

Is this always what you want?
Benchmarking

- A revised (informal) approach:

```java
for (...) doWorkloadOfInterest();
startTime = getCurrentTimeInNanos();
for (...) doWorkloadOfInterest();
endTime = getCurrentTimeInNanos();
reportResult(endTime - startTime);
```
Benchmarking

- A revised (informal) approach:

```java
for (...) doWorkloadOfInterest();
startTime = getCurrentTimeInNanos();
for (...) doWorkloadOfInterest();
endTime = getCurrentTimeInNanos();
reportResult(endTime - startTime);
```

- This still does not solve everything
  - Frequency scaling?
  - Interference of other workloads?
  - Alignment?
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must *compare*
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must compare
    - Benchmark vs expectation/mental model
    - Different solutions
    - Over time
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must *compare*
    - Benchmark vs expectation/mental model
    - Different solutions
    - Over time
  - Results are often normalized against the baseline
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must *compare*
  - We must remember results are *statistical*
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must *compare*
  - We must remember results are *statistical*
    - Show the distribution (e.g. violin plots)
Benchmarking

- Now we have a benchmark, how do we interpret/report it?
  - We must *compare*
  - We must remember results are *statistical*
    - Show the distribution (e.g. violin plots)
    - Summarize the distribution (e.g. mean and confidence intervals, box & whisker)
Benchmarking

- A benchmark suite comprises multiple benchmarks

![Bar chart showing comparisons between 'Old' and 'New' for benchmarks T1 to T6]
Benchmarking

- A benchmark suite comprises multiple benchmarks
- Now we have multiple results, how should we consider them?
Benchmarking

- A benchmark suite comprises multiple benchmarks

- Now we have multiple results, how should we consider them?
  - 2 major scenarios
    - *Hypothesis testing*
      - Is solution A different than B?
Benchmarking

A benchmark suite comprises multiple benchmarks

Now we have multiple results, how should we consider them?

- 2 major scenarios
  - Hypothesis testing
    - Is solution A different than B?
    - You can use ANOVA
Benchmarking

- A benchmark suite comprises multiple benchmarks

- Now we have multiple results, how should we consider them?
  - 2 major scenarios
    - Hypothesis testing
    - Summary statistics

![Bar chart comparing old and new results for T1 to T6.](image)
Benchmarking

- A benchmark suite comprises multiple benchmarks

- Now we have multiple results, how should we consider them?
  - 2 major scenarios
    - *Hypothesis testing*
    - *Summary statistics*
      - Condensing a suite to a single number
      - Intrinsically lossy, but can still be useful
Benchmarking

- A benchmark suite comprises multiple benchmarks

- Now we have multiple results, how should we consider them?
  - 2 major scenarios
    - Hypothesis testing
    - Summary statistics
      - Condensing a suite to a single number
      - Intrinsically lossy, but can still be useful

---

![Chart showing old and new results for benchmarks T1 to T6]
Summary Statistics

Averages of $r_1, r_2, ..., r_N$

- Many ways to measure expectation or tendency
Summary Statistics

Averages of \( r_1, r_2, \ldots, r_N \)

- Many ways to measure *expectation* or *tendency*
- Arithmetic Mean

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]
Summary Statistics

Averages of $r_1, r_2, ..., r_N$

- Many ways to measure *expectation* or *tendency*
- Arithmetic Mean
  
  $$\frac{1}{N} \sum_{i=1}^{N} r_i$$

- Harmonic Mean
  
  $$\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}$$
Summary Statistics

Averages of $r_1, r_2, \ldots, r_N$

- Many ways to measure expectation or tendency
- Arithmetic Mean
  \[ \frac{1}{N} \sum_{i=1}^{N} r_i \]
- Harmonic Mean
  \[ \frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}} \]
- Geometric Mean
  \[ \sqrt[N]{\prod_{i=1}^{N} r_i} \]
Summary Statistics

Averages of \( r_1, r_2, \ldots, r_N \)

- Many ways to measure *expectation* or *tendency*
- Arithmetic Mean
  \[
  \frac{1}{N} \sum_{i=1}^{N} r_i
  \]
- Harmonic Mean
  \[
  \frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}
  \]
- Geometric Mean
  \[
  \sqrt[\prod_{i=1}^{N} r_i]{}^{N}
  \]

Each type means something different and has valid uses.
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  
  \[
  \frac{1}{N} \sum_{i=1}^{N} r_i
  \]
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  - Used for computing *sample means*

$$\frac{1}{N} \sum_{i=1}^{N} r_i$$
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  - Used for computing sample means
  - e.g. Timing the same workload many times

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  - Used for computing sample means
  - e.g. Timing the same workload many times

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]

Handling Nondeterminism

```plaintext
for (x in 0 to 4)
    times[x] = doWorkloadOfInterest();

E(time) = arithmean(times)
```
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  - Used for computing sample means
  - e.g. Timing the same workload many times

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]

- **Harmonic Mean**
  - Good for reporting *rates*

\[
\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}
\]
Summary Statistics

- **Arithmetic Mean**
  - Good for reporting averages of numbers that mean the same thing
  - Used for computing sample means
  - e.g. Timing the same workload many times

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]

- **Harmonic Mean**
  - Good for reporting *rates*
  - e.g. Required throughput for a set of tasks

\[
\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}
\]
Summary Statistics

Given tasks t1, t2, & t3 serving 40 pages each:
  throughput(t1) = 10 pages/sec
  throughput(t2) = 20 pages/sec
  throughput(t3) = 20 pages/sec

What is the average throughput? What should it mean?

- Good for reporting rates
- e.g. Required throughput for a set of tasks
Summary Statistics

Given tasks t1, t2, & t3 serving 40 pages each:
  throughput(t1) = 10 pages/sec
  throughput(t2) = 20 pages/sec
  throughput(t3) = 20 pages/sec

What is the average throughput? What should it mean?
Arithmetic = 16.7 p/s

- Good for reporting rates
- e.g. Required throughput for a set of tasks
### Summary Statistics

**Arithmetic Mean**
- Good for reporting averages of numbers that mean the same thing
- Used for computing sample means
- e.g. Timing the same workload many times

**Harmonic Mean**
- Good for reporting rates
- e.g. Required throughput for a set of tasks

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]

Given tasks \( t_1, t_2, \) & \( t_3 \) serving 40 pages each:
- \( \text{throughput}(t_1) = 10 \) pages/sec
- \( \text{throughput}(t_2) = 20 \) pages/sec
- \( \text{throughput}(t_3) = 20 \) pages/sec

What is the average throughput? What should it mean?
- Arithmetic = 16.7 p/s
- Harmonic = 15 p/s

- Good for reporting rates
- e.g. Required throughput for a set of tasks

\[
\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}
\]
Summary Statistics

Given tasks t1, t2, & t3 serving 40 pages each:
- throughput(t1) = 10 pages/sec
- throughput(t2) = 20 pages/sec
- throughput(t3) = 20 pages/sec

What is the average throughput? What should it mean?
- Arithmetic = 16.7 p/s
- Harmonic = 15 p/s

\[
\frac{120}{16.7} = 7.2 \quad \frac{120}{15} = 8
\]

- Good for reporting rates
- e.g. Required throughput for a set of tasks

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]

\[
\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}}
\]
Summary Statistics

Given tasks t1, t2, & t3 serving 40 pages each:
- throughput(t1) = 10 pages/sec
- throughput(t2) = 20 pages/sec
- throughput(t3) = 20 pages/sec

What is the average throughput? What should it mean?
Arithmetic = 16.7 p/s
Harmonic = 15 p/s

\[
\frac{120}{16.7} = 7.2 \quad \frac{120}{15} = 8
\]

- Good for reporting rates
- e.g. Required throughput for a set of tasks

Identifies the constant rate required for the same time

\[
\frac{1}{N} \sum_{i=1}^{N} r_i
\]
Summary Statistics

Given tasks t1, t2, & t3 serving 40 pages each:
- throughput(t1) = 10 pages/sec
- throughput(t2) = 20 pages/sec
- throughput(t3) = 20 pages/sec

What is the average throughput? What should it mean?
- Arithmetic = 16.7 p/s
- Harmonic = 15 p/s

\[
\frac{120}{16.7} = 7.2 \quad \frac{120}{15} = 8
\]

- Good for reporting rates
- e.g. Required throughput for a set of tasks

Identifies the constant rate required for the same time

CAVEAT: If the size of each workload changes, a weighted harmonic mean is required!
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[
\sqrt[N]{\prod_{i=1}^{N} r_i}
\]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

Any idea why it may be useful here? (A bit of a thought experiment)
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[ \sqrt[N]{\prod_{i=1}^{N} r_i} \]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[ \sqrt[N]{\prod_{i=1}^{N} r_i} \]

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What happens to the arithmetic mean? halved
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[
\sqrt[N]{\prod_{i=1}^{N} r_i}
\]

What happens to the arithmetic mean?

Old

![Bar chart showing T1 and T2 with halved values]

New 2

![Bar chart showing T1 and T2]

What happens to the arithmetic mean?
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[ N \prod_{i=1}^{N} r_i \]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

Geometric:

\[ \sqrt[N]{\prod_{i=1}^{N} r_i} \]

\[ \sqrt{r_1 \times r_2} \]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[ \sqrt[\text{N}] { \prod_{i=1}^{N} r_i } \]

\[
\begin{align*}
\text{Geometric:} & \quad \sqrt{r_1 \times r_2} \\
\text{Old} & \quad \sqrt{r_1 \times \left(\frac{1}{2} r_2\right)} \\
\text{New 1} & \quad 
\end{align*}
\]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[
\sqrt{r_1 \times r_2}
\]

- Old

\[
\sqrt{\frac{1}{2} r_1 \times r_2}
\]

- New 1

\[
\sqrt{\frac{1}{2} r_1 \times r_2}
\]

- New 2
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks

\[ \sqrt[N]{\prod_{i=1}^{N} r_i} \]

\[
\begin{align*}
\text{Old} & : \sqrt{r_1 \times r_2} \\
\text{New 1} & : \sqrt{r_1 \times \left(\frac{1}{2} r_2\right)} \\
\text{New 2} & : \sqrt{\frac{1}{2} \times r_1 \times r_2} = \sqrt{\left(\frac{1}{2} r_1\right) \times r_2}
\end{align*}
\]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks
  - A 10% difference in any benchmark affects the final value the same way

\[ \sqrt[\text{N}]{\prod_{i=1}^{\text{N}} r_i} \]
Summary Statistics

- **Geometric Mean**
  - Good for reporting results that mean different things
  - e.g. Timing results across *many different* benchmarks
  - A 10% difference in any benchmark affects the final value the same way

Note: It doesn't have an *intuitive* meaning!
It does provides a balanced *score* of performance.

Benchmarking

- In practice applying good benchmarking & statistics is made easier via frameworks
  - Google benchmark (C & C++)
  - Google Caliper (Java)
  - Nonius
  - Celero
  - Easybench
  - Pyperf
  - ...

Investigating Performance
Profiling

- When benchmark results do not make sense, you should investigate why
Profiling

• When benchmark results do not make sense, you should investigate why
  – For resource X, where is X being used, acquired, and or released?
Profiling

- When benchmark results do not make sense, you should investigate why
  - For resource X, where is X being *used*, *acquired*, and or *released*?

- **Sometimes microbenchmarks provide sufficient insight**
Profiling

- When benchmark results do not make sense, you should investigate why
  - For resource X, where is X being used, acquired, and/or released?

- Sometimes microbenchmarks provide sufficient insight

- In other cases you will want to profile
Profiling

• When benchmark results do not make sense, you should investigate why
  – For resource X, where is X being used, acquired, and or released?

• Sometimes microbenchmarks provide sufficient insight

• In other cases you will want to profile
  – Collect additional information about resources in an execution
  – The nature of the tool will depend on the resource and the objective
Profiling

- When benchmark results do not make sense, you should investigate why
  - For resource X, where is X being used, acquired, and or released?
- Sometimes microbenchmarks provide sufficient insight
- In other cases you will want to profile
  - Collect additional information about resources in an execution
  - The nature of the tool will depend on the resource and the objective

You should already be familiar with tools like gprof or jprofile. We’ll examine some more advanced profilers now.
Heap profiling

- Suppose I have a task and it consumes all memory
  - Note: This is not hypothetical. This often happens with grad students!
Suppose I have a task and it consumes all memory

- **Note:** This is not hypothetical. This often happens with grad students!
- If I can identify where & why memory is consumed, I can remediate
  - Maybe better algorithm
  - Maybe competent use of data structures....
Heap profiling

- Suppose I have a task and it consumes all memory
  - **Note:** This is not hypothetical. This often happens with grad students!
  - If I can identify where & why memory is consumed, I can remediate
    - Maybe better algorithm
    - Maybe competent use of data structures....

- Heap profilers track the allocated memory in a program & their provenance
Heap profiling

- Suppose I have a task and it consumes all memory
  - **Note:** This is not hypothetical. This often happens with grad students!
  - If I can identify where & why memory is consumed, I can remediate
    - Maybe better algorithm
    - Maybe competent use of data structures....

- Heap profilers track the allocated memory in a program & their provenance
  - Can identify hotspots, bloat, leaks, short lived allocations, ...
Heap profiling

- Suppose I have a task and it consumes all memory
  - **Note:** This is not hypothetical. This often happens with grad students!
  - If I can identify where & why memory is consumed, I can remediate
    - Maybe better algorithm
    - Maybe competent use of data structures....

- **Heap profilers track the allocated memory in a program & their provenance**
  - Can identify hotspots, bloat, leaks, short lived allocations, ...
  - Usually *sample* based, but sometimes *event* based
  - e.g. Massif, Heaptrack, ...
int main() {
  std::vector<std::unique_ptr<long[]>> data{DATA_SIZE};

  for (auto &element : data) {
    element = std::make_unique<long[]>(BLOCK_SIZE);
    // do something with element
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }

  std::this_thread::sleep_for(std::chrono::seconds(1));
  return 0;
}
Heap profiling

```cpp
int main() {
    std::vector<std::unique_ptr<long[]>> data{DATA_SIZE};

    for (auto &element : data) {
        element = std::make_unique<long[]>(BLOCK_SIZE);
        // do something with element
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 0;
}
```

valgrind --time-unit=ms --tool=massif <program invocation>
heaptrack <program invocation>

massif-visualizer massif.out.<PID>
heaptrack_gui <path to data>
Heap profiling

```cpp
int main() {
    std::vector<std::unique_ptr<long[]>> data{DATA_SIZE};

    for (auto &element : data) {
        element = std::make_unique<long[]>(BLOCK_SIZE);
        // do something with element
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        element.reset();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 0;
}
```

How do we expect this to differ?
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...

- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...

- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually

```
main()
    foo() 70% 20%
    baz() 70% 20%
    quux() 20%
    bar() 20%
```
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...

- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually

- Flame graphs provide a way of structuring and visualizing substantial profiling information
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...
- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually
- Flame graphs provide a way of structuring and visualizing substantial profiling information
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...

- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually

- Flame graphs provide a way of structuring and visualizing substantial profiling information

It is easier to see that optimizing `baz()` could be useful.
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...

- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually

- Flame graphs provide a way of structuring and visualizing substantial profiling information
  - Consumers of CPU on top

```plaintext
main()
  foo()
    baz()
    quux()
  bar()
```
CPU Profiling & Flame Graphs

- When CPU is the resource, investigate where the CPU is spent
  - Classic profilers – gprof, oprofile, jprof, ...
- Classic CPU profilers capture a lot of data and force the user to explore & explain it manually
- Flame graphs provide a way of structuring and visualizing substantial profiling information
  - Consumers of CPU on top
  - ancestry, proportions, components can all be clearly identified

```
main()
  foo()
    baz()
    bar()
  quux()
```
CPU Profiling & Flame Graphs

- Can extract rich information by embedding interesting things in colors

[Gregg, ATC 2017]
CPU Profiling & Flame Graphs

- Flame graphs are not just limited to CPU time!
  - Any countable resource or event can be organized & visualized
CPU Profiling & Flame Graphs

- Flame graphs are not just limited to CPU time!
  - Any countable resource or event can be organized & visualized

- You can also automatically generate them with clang & chrome
  - See project X-Ray in clang
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Missspeculations
  - TLB misses
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Misspeculations
  - TLB misses

  How well does sample based profiling work for these?
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Misspeculations
  - TLB misses

  How well does sample based profiling work for these?

- Instead, we can leverage low level system counters via tools like perf
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Misspeculations
  - TLB misses

  How well does sample based profiling work for these?

- Instead, we can leverage low level system counters via tools like perf
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Misspeculations
  - TLB misses

  How well does sample based profiling work for these?

- Instead, we can leverage low level system counters via tools like `perf`

```
perf stat -e <events> -g <command>
perf record -e <events> -g <command>
perf report
perf list
```
Perf & event profiling

- Sometimes low-level architectural effects determine the performance
  - Cache misses
  - Misspeculations
  - TLB misses

  How well does sample based profiling work for these?

- Instead, we can leverage low level system counters via tools like `perf`

  ```
  perf stat -e <events> -g <command>
  perf record -e <events> -g <command>
  perf report
  perf list
  ```

  events like

  ```
  task-clock, context-switches, cpu-migrations, page-faults, cycles, instructions, branches, branch-misses, cache-misses, cycle_activity.stalls_total
  ```
Profiling for opportunities

- Causal profiling
Profiling for opportunities

- Causal profiling

What should I look at to speed things up?
Profiling for opportunities

- Causal profiling

What should I look at to speed things up?
Profiling for opportunities

- Causal profiling
- Profiling for parallelism
Profiling for opportunities

- Causal profiling
- Profiling for parallelism
Improving Performance
Improving Performance

- We can attack performance at several levels
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
  - Managing the organization of data
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
  - Managing the organization of data
  - Managing the organization of code
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
  - Managing the organization of data
  - Managing the organization of code
  - Better algorithms & algorithmic modeling
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
  - Managing the organization of data
  - Managing the organization of code
  - Better algorithms & algorithmic modeling

- In all cases, we only care about improving performance of hot code
Improving Performance

- We can attack performance at several levels
  - Compilers & tuning the build process
  - Managing the organization of data
  - Managing the organization of code
  - Better algorithms & algorithmic modeling

- In all cases, we only care about improving performance of hot code

- Optimizing cold code can hurt software
Compiling for performance

- Enabling optimizations...
Compiling for performance

- Enabling optimizations...
- LTO (Link Time Optimization / Whole Program Optimization)
Compiling for performance

- Enabling optimizations...
- LTO (Link Time Optimization / Whole Program Optimization)

foo.c ➔ foo.o

bar.c ➔ bar.o
Compiling for performance

- Enabling optimizations...
- LTO (Link Time Optimization / Whole Program Optimization)

```
foo.c
Compile & Optimize
foo.o

bar.c
Compile & Optimize
bar.o

Link
program
```
Compiling for performance

- Enabling optimizations...
- LTO (Link Time Optimization / Whole Program Optimization)

```
foo.c  Compile & Optimize  foo.o

bar.c  Compile & Optimize  bar.o

Merge  program(.o)  Optimize & Link  program
```
Compiling for performance

- Enabling optimizations...
- LTO
- PGO/FDO (Profile Guided Optimization/Feedback Directed Optimization)
  - Incorporate profile information in optimization decisions
Compiling for performance

- Enabling optimizations...
- LTO
- PGO/FDO (Profile Guided Optimization/Feedback Directed Optimization)
  - Incorporate profile information in optimization decisions

funPtr = ?
...
funPtr()
Compiling for performance

- Enabling optimizations...
- LTO
- PGO/FDO (Profile Guided Optimization/Feedback Directed Optimization)
  - Incorporate profile information in optimization decisions
Compiling for performance

- Enabling optimizations...
- LTO
- PGO/FDO (Profile Guided Optimization/Feedback Directed Optimization)
  - Incorporate profile information in optimization decisions

funPtr = ?
... funPtr()

foo(){A}
bar(){B}

funPtr = ?
... if funPtr == bar: B' else: funPtr()
Compiling for performance

- Enabling optimizations...
- LTO
- PGO/FDO (Profile Guided Optimization/Feedback Directed Optimization)
  - Incorporate profile information in optimization decisions
Compiling for performance

- Enabling optimizations...
- LTO
- PGO
- Layout optimization (BOLT and otherwise)
Compiling for performance

- Enabling optimizations...
- LTO
- PGO
- Layout optimization (BOLT and otherwise)
- Polyhedral analysis
Optimizing Your Data

- The basic directions of data optimizations
Optimizing Your Data

- The basic directions of data optimizations
  - Ensure the data you want is available for the tasks you have
Optimizing Your Data

- The basic directions of data optimizations
  - Ensure the data you want is available for the tasks you have
  - Do not spend time processing you do not need
Optimizing Your Data

- The basic directions of data optimizations
  - Ensure the data you want is available for the tasks you have
  - Do not spend time processing you do not need
  - Do not spend extra time managing the data at the system level
Optimizing Your Data

- **The basic directions of data optimizations**
  - Ensure the data you want is available for the tasks you have
  - Do not spend time processing you do not need
  - Do not spend extra time managing the data at the system level

Several aspects of high level design may be in tension with these
Optimizing Your Data

- Basic structure packing
  - Smaller aggregates consume less cache

```c
struct S1 {
    char a;
};
sizeof(S1) == 1
```

```c
struct S2 {
    uint32_t b;
};
sizeof(S2) == 4
```
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache

```c
struct S1 {
    char a;
};
```

```c
struct S2 {
    uint32_t b;
};
```

```c
struct S3 {
    char a;
    uint32_t b;
    char c;
};
```

sizeof(S1) == 1
sizeof(S2) == 4
sizeof(S3) == ?
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache

```c
struct S1 {
    char a;
};
sizeof(S1) == 1

struct S2 {
    uint32_t b;
};
sizeof(S2) == 4

struct S3 {
    char a;
    uint32_t b;
    char c;
};
sizeof(S3) == 12
```

`uint32_t` must be 4 byte aligned. Padding is inserted!
Optimizing Your Data

- Basic structure packing
  - Smaller aggregates consume less cache

```c
struct S1 {
    char a;
};
sizeof(S1) == 1

struct S2 {
    uint32_t b;
};
sizeof(S2) == 4

struct S3 {
    char a;
    uint32_t b;
    char c;
};
sizeof(S3) == 12

struct S4 {
    char a;
    char c;
    uint32_t b;
};
sizeof(S3) == 8
```
Optimizing Your Data

- Basic structure packing
  - Smaller aggregates consume less cache

```c
struct S1 {
    char a;
};
sizeof(S1) == 1

struct S2 {
    uint32_t b;
};
sizeof(S2) == 4

struct S3 {
    char a;
    uint32_t b;
    char c;
};
sizeof(S3) == 12

struct S4 {
    char a;
    char c;
    uint32_t b;
};
sizeof(S3) == 8
```

Careful ordering improves cache utilization
Optimizing Your Data

- Basic structure packing
  - Smaller aggregates consume less cache
  - Carefully *encoding* data or *reusing* storage can do more
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache
  - Carefully *encoding* data or *reusing* storage can do more
    - Operate on compressed data
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache
  - Carefully *encoding* data or *reusing* storage can do more
    - Operate on compressed data
    - Steal low/high order bits of pointers
Optimizing Your Data

- Basic structure packing
  - Smaller aggregates consume less cache
  - Carefully **encoding** data or **reusing** storage can do more
    - Operate on compressed data
    - Steal low/high order bits of pointers

```cpp
template <class PointedTo>
class PointerValuePair<PointedTo,int> {
    uintptr_t compact;
    PointedTo* getP() {
        return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFFF8);
    }
    Value getV() { return compact & 0x00000007; }
};
```
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache
  - Carefully *encoding* data or *reusing* storage can do more
    - Operate on compressed data
    - Steal low/high order bits of pointers

```cpp
template <class PointedTo>
class PointerValuePair<PointedTo,int> {
  uintptr_t compact;
  PointedTo* getP() {
    return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFFF8);
  }
  Value getV() { return compact & 0x00000007; }
};
```
Optimizing Your Data

- **Basic structure packing**
  - Smaller aggregates consume less cache
  - Carefully *encoding* data or *reusing* storage can do more
    - Operate on compressed data
    - Steal low/high order bits of pointers

```cpp
template <class PointedTo>
class PointerValuePair<PointedTo,int> {
  uintptr_t compact;
  PointedTo* getP() {
    return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFF8);
  }
  Value getV() { return compact & 0x00000007; }
};
```
Optimizing Your Data

**Basic structure packing**
- Smaller aggregates consume less cache
- Carefully *encoding* data or *reusing* storage can do more
  - Operate on compressed data
  - Steal low/high order bits of pointers

```cpp
template <class PointedTo>
class PointerValuePair<PointedTo,int> {
  uintptr_t compact;
  PointerTo* getP() {
    return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFFF8);
  }
  Value getV() { return compact & 0x00000007; }
};
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory
Optimizing Your Data

- **Managing indirection**
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
    ...
}
```

We already saw this. Traversing a linked list is expensive!
Optimizing Your Data

- **Managing indirection**
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
    ...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
    ...
}
```

These elements are unlikely to be in cache and unlikely to be prefetched automatically.
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
    ...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...  
for (auto& i : numbers) {
    ...  
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
  ...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ... for (auto& i : numbers) {
  ...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

```cpp
std::list numbers = ...
for (auto& i : numbers) {
  ...
}
```
Optimizing Your Data

- Managing indirection
  - Pointers and indirection can stall the CPU while waiting on memory

How does this relate to design tools that we have seen?
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```c
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```cpp
struct Dog {
  uint32_t friendliness;
  uint32_t age;
  uint32_t ownerID;
  std::string hobby;
  Food treats[10];
};

for (Dog& d : dogs) {
  play(d.friendliness, d.hobby);
}
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};

for (Dog& d : dogs) {
    play(d.friendliness, d.hobby);
}
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};

for (Dog& d : dogs) {
    play(d.friendliness, d.hobby);
}
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};
```

```cpp
for (Dog& d : dogs) {
    play(d.friendliness, d.hobby);
}
```

We can try to push the cold fields out of the cache
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};

struct HotDog {
    double friendliness;
    std::string hobby;
    unique_ptr<Cold> cold;
};

struct Cold {
    uint32_t age;
    uint32_t ownerID;
    Food treats[10];
};
```

Benefits depend on the size of Cold & the access patterns.
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};

struct DogManager {
    std::vector<uint32_t> friendliness;
    std::vector<uint32_t> age;
    std::vector<uint32_t> ownerID;
    std::vector<std::string> hobby;
    std::vector<std::array<Food,10>> treats;
};
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

```cpp
struct Dog {
    uint32_t friendliness;
    uint32_t age;
    uint32_t ownerID;
    std::string hobby;
    Food treats[10];
};

struct DogManager {
    std::vector<uint32_t> friendliness;
    std::vector<uint32_t> age;
    std::vector<uint32_t> ownerID;
    std::vector<std::string> hobby;
    std::vector<std::array<Food, 10>> treats;
};

for (auto i : range(dogs)) {
    play(friendliness[i], hobby[i]);
}
```
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

| Dog1 | Dog2 |
Optimizing Your Data

- **Grouping things that are accessed together**
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)
### Optimizing Your Data

- **Grouping things that are accessed together**
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

<table>
<thead>
<tr>
<th>friendliness</th>
<th>Dog1</th>
<th>Dog2</th>
<th>Dog3</th>
<th>Dog4</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hobby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Optimizing Your Data

- **Grouping things that are accessed together**
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

<table>
<thead>
<tr>
<th></th>
<th>Dog1</th>
<th>Dog2</th>
<th>Dog3</th>
<th>Dog4</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hobby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can pick and choose while still getting good locality
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

You can pick and choose while still getting good locality
Optimizing Your Data

- Grouping things that are accessed together
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structures vs Struct of Arrays)

You can pick and choose while still getting good locality

Easier for compilers to vectorize
### Optimizing Your Data

- **Grouping things that are accessed together**
  - Guiding spatial design by temporal locality can improve cache utilization
  - Cold field outlining
  - AoS vs SoA (Array of Structs vs Struct of Arrays)

<table>
<thead>
<tr>
<th>Dog1</th>
<th>Dog2</th>
</tr>
</thead>
<tbody>
<tr>
<td>friendliness</td>
<td>Dog1</td>
</tr>
<tr>
<td>age</td>
<td></td>
</tr>
<tr>
<td>hobby</td>
<td></td>
</tr>
</tbody>
</table>

- You can pick and choose while still getting good locality
- Easier for compilers to vectorize
- Also a foundation of modern game engine design (ECS)
Optimizing Your Data

- Loop invariance
  - Avoid recomputing the same values inside a loop
Optimizing Your Data

- **Loop invariance**
  - Avoid recomputing the same values inside a loop

```cpp
for (auto i : ...) {
    auto sqrt2 = sqrt(2);
    auto x = f(i, sqrt2);
    ...
}
```

```cpp
auto sqrt2 = sqrt(2);
for (auto i : ...) {
    auto x = f(i, sqrt2);
    ...
}
```
Optimizing Your Data

- **Loop invariance**
  - Avoid recomputing the same values inside a loop
  - Compilers automate this but cannot always succeed (LICM)

```cpp
for (auto i : ...) {
    auto sqrt2 = sqrt(2);
    auto x = f(i, sqrt2);
    ...
}

auto sqrt2 = sqrt(2);
for (auto i : ...) {
    auto x = f(i, sqrt2);
    ...
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw
Inner loop locality

- The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw:

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```

Memory accesses are consecutive!
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t matrix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}

uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
  for (size_t col = 0; col < cols; ++col) {
    foo(matrix[cols*row + col]);
  }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
  for (size_t col = 0; col < cols; ++col) {
    foo(matrix[rows*col + row]);
  }
}
```
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[cols*row + col]);
    }
}
```

```c
uint32_t marix[rows*cols];
for (size_t row = 0; row < rows; ++row) {
    for (size_t col = 0; col < cols; ++col) {
        foo(matrix[rows*col + row]);
    }
}
```

Memory accesses jump around & thrash the cache!
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work
Optimizing Your Data

- Inner loop locality
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work

Problem:
Using the same layout creates bad locality.
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work

Solution: Transpose first. Implement over the transpose instead.
Optimizing Your Data

- **Inner loop locality**
  - The simplest scenarios are like the matrix example we first saw
  - Matrix operations (e.g. multiplication) can require extra work

Note: Better solutions further leverage layout & parallelization.
Optimizing Your Data

- Memory management effects
  - Data structure packing & access patterns affect deeper system behavior
Optimizing Your Data

- Memory management effects
  - Data structure packing & access patterns affect deeper system behavior
    - What about virtual memory, page tables, & the TLB?
Optimizing Your Data

- Memory management effects
  - Data structure packing & access patterns affect deeper system behavior
    - What about virtual memory, page tables, & the TLB?
    - What about allocation strategies & fragmentation?
Optimizing Your Data

- Designing with clear ownership policies in mind
Optimizing Your Data

- Designing with clear ownership policies in mind
  - Resource acquisition should not happen in hot code
Optimizing Your Data

- Designing with clear ownership policies in mind
  - Resource acquisition should not happen in hot code
  - Use APIs that express intent & prevent copying
Optimizing Your Data

- Designing with clear ownership policies in mind
  - Resource acquisition should not happen in hot code
  - Use APIs that express intent & prevent copying

  “std::string is responsible for almost half of all allocations in the Chrome”
Optimizing Your Data

- Designing with clear ownership policies in mind
  - Resource acquisition should not happen in hot code
  - Use APIs that express intent & prevent copying

“std::string is responsible for almost half of all allocations in the Chrome”

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);

    template<class E>
    Span(const std::vector<E>& c);

    E* first;
    size_t count;
};
```
Optimizing Your Code

- Basic ideas for code optimization
Optimizing Your Code

- Basic ideas for code optimization
  - Avoid branching whenever possible
Optimizing Your Code

- Basic ideas for code optimization
  - Avoid branching whenever possible

Misspeculating over a branch is costly
Optimizing Your Code

- Basic ideas for code optimization
  - Avoid branching whenever possible
  - Make code that does the same thing occur close together temporally
Optimizing Your Code

- Basic ideas for code optimization
  - Avoid branching whenever possible
  - Make code that does the same thing occur close together temporally

Leverage the instruction cache if you can
Optimizing Your Code

- Branch prediction & speculation
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```cpp
for (...) {
    if (foo(c)) {
        bar();  // A
    } else {
        baz();  // B
    }
}
```
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```java
for (...) {
    if (foo(c)) {
        bar();
    } else {
        baz();
    }
}
```
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```cpp
for (...) {
    if (foo(c)) {
        bar();
    } else {
        baz();
    }
}
```

Pipeline: A A A
Actual: A A
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```c
for (...) {
  if (foo(c)) {
    bar();
  } else {
    baz();
  }
}
```
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```java
for (...) {
    if (foo(c)) {
        bar(); // A
    } else {
        baz(); // B
    }
}
```

Pipeline: A A A
Actual: A A B
Stall, but relatively infrequently
Optimizing Your Code

- Branch prediction & speculation
  - On if statements

```c
for (...) {
    if (foo(c)) {
        bar();
    } else {
        baz();
    }
}
```
Optimizing Your Code

- **Branch prediction & speculation**
  - On *if* statements

```java
for (...) {
    if (foo(c)) {
        bar();
    } else {
        baz();
    }
}
```

Pipeline:
- 51% A
- 49% B

Actual:
- A
- B

Stall, frequently
Optimizing Your Code

- Branch prediction & speculation
  - On if statements
  - On function pointers!

```c
for (...) {
  foo();
}
bar() {}
baz() {} A 51%
```
Optimizing Your Code

- Branch prediction & speculation
  - On if statements
  - On function pointers!
**Optimizing Your Code**

- **Branch prediction & speculation**
  - On if statements
  - On function pointers!

```c
for (...) {
    foo();
}
bar() {}
baz() {}
```

The same problems arise
- Consistent call targets perform better
Optimizing Your Code

- Designing away checks
  - Repeated checks can be removed by maintaining invariants
Optimizing Your Code

- Designing away checks
  - Repeated checks can be removed by maintaining invariants

```python
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    i ← i + 1
```

[Wikipedia’s Insertion Sort]
Optimizing Your Code

- **Designing away checks**
  - Repeated checks can be removed by maintaining invariants

```pseudocode
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    i ← i + 1
```

[Wikipedia’s Insertion Sort]
Optimizing Your Code

- Designing away checks
  - Repeated checks can be removed by maintaining invariants

```plaintext
i ← 1
while i < length(A)
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  i ← i + 1

[Wikipedia’s Insertion Sort]

Can we turn the semantic check into a bounds check?
Optimizing Your Code

- Designing away checks
  - Repeated checks can be removed by maintaining invariants

```plaintext
i ← 1
while i < length(A)
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  i ← i + 1
```

[Wikipedia's Insertion Sort]

We just guarantee that A starts with the smallest element!
Optimizing Your Code

- **Designing away checks**
  - Repeated checks can be removed by maintaining invariants

```plaintext
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    i ← i + 1
```

```plaintext
[k ← find_smallest(A)]
swap A[0] and A[k]
```

```plaintext
i ← 1
while i < length(A)
    j ← i
        swap A[j] and A[j-1]
        j ← j - 1
    i ← i + 1
```

[Wikipedia’s Insertion Sort]

We just guarantee that A starts with the smallest element!
Optimizing Your Code

- Designing away checks
  - Repeated checks can be removed by maintaining invariants

```plaintext
A[-1] ← MIN_VALUE
i ← 1
while i < length(A)
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  i ← i + 1
```

We just guarantee that A starts with the smallest element!
Optimizing Algorithms

- Improving real world algorithmic performance comes from recognizing the *interplay* between *theory* and *hardware*
Optimizing Algorithms

- Improving real world algorithmic performance comes from recognizing the *interplay* between *theory* and *hardware*.

- **Hybrid algorithms**
  - Constants matter. Use thresholds to select algorithms.
Optimizing Algorithms

- Improving real world algorithmic performance comes from recognizing the *interplay* between *theory* and *hardware*

- **Hybrid algorithms**
  - Constants matter. Use thresholds to select algorithms.
  - Use general $N \log N$ sorting for $N$ above 300 [Alexandrescu 2019]
Optimizing Algorithms

- Improving real world algorithmic performance comes from recognizing the *interplay* between *theory* and *hardware*

- Hybrid algorithms
  - Constants matter. Use thresholds to select algorithms.
  - Use general $N \log N$ sorting for $N$ above 300 [Alexandrescu 2019]

- **Caching & Precomputing**
Optimizing Algorithms

• Improving real world algorithmic performance comes from recognizing the **interplay** between **theory** and **hardware**

• Hybrid algorithms
  – Constants matter. Use thresholds to select algorithms.
  – Use general $N \log N$ sorting for $N$ above 300 [Alexandrescu 2019]

• **Caching & Precomputing**
  – If you will reuse results, save them and avoid recomputing
Optimizing Algorithms

- Improving real world algorithmic performance comes from recognizing the *interplay* between *theory* and *hardware*

- Hybrid algorithms
  - Constants matter. Use thresholds to select algorithms.
  - Use general $N \log N$ sorting for $N$ above 300 [Alexandrescu 2019]

- **Caching & Precomputing**
  - If you will reuse results, save them and avoid recomputing
  - If all possible results are compact, just compute a table up front
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an *abstract machine model* that is too approximate!
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance

A uniform cost model throws necessary information away
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- **Classic asymptotic complexity less useful in practice**
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads

- Alternative approaches
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads

- Alternative approaches
  - I/O complexity, I/O efficiency and cache awareness
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts
- Classic asymptotic complexity less useful in practice
  - It uses a restricted abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shape of workloads

Alternative approaches
- I/O complexity, I/O efficiency and cache awareness
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts
- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation

- Alternative approaches
  - I/O complexity, I/O efficiency and cache awareness

Complexity measured in block transfers

[Diagram showing CPU connected to Memory 1 and Memory 2 with block size B]
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads

- Alternative approaches
  - I/O complexity, I/O efficiency and cache awareness
  - Cache oblivious algorithms & data structures
Optimizing Algorithms

- Better performance modeling & algorithms
  - The core approaches we use have not adapted to changing contexts

- Classic asymptotic complexity less useful in practice
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads

- Alternative approaches
  - I/O complexity, I/O efficiency and cache awareness
  - Cache oblivious algorithms & data structures
    Similar to I/O, but agnostic to block size
Optimizing Algorithms

- **Better performance modeling & algorithms**
  - The core approaches we use have not adapted to changing contexts

- **Classic asymptotic complexity less useful in practice**
  - It uses an abstract machine model that is too approximate!
  - Constants and artifacts of scale can actually dominate the real world performance
  - We want modeling & algorithms that account for artifacts like: memory, I/O, consistency & speculation, shapes of workloads

- **Alternative approaches**
  - I/O complexity, I/O efficiency and cache awareness
  - Cache oblivious algorithms & data structures
  - Parameterized complexity
Optimizing Algorithms

- Classic design mistakes [Lu 2012]
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)

```cpp
for (auto& action : actions) {
  action.do();
}
```

```cpp
Action::do() {
  acquire(mutex);
  ...
  release(mutex);
}
```
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)

```c++
for (auto& action : actions) {
    action.do()
}
```

```c++
Action::do() {
    acquire(mutex)
    ...
    release(mutex)
}
```

**vs**

```c++
acquire(mutex)
for (auto& action : actions) {
    action.do()
}
release(mutex)
```
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)
  - Skippable functions (e.g. transparent draws)
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)
  - Skippable functions (e.g. transparent draws)
  - Poor/unclear synchronization
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)
  - Skippable functions (e.g. transparent draws)
  - Poor/unclear synchronization

```plaintext
foo() {
  bar()
}

bar() {
  baz()
}

baz() {
  quux()
}

quux() {
  random()
}

random() {
  acquire(mutex)
  ...
  release(mutex)
}
```
Optimizing Algorithms

- **Classic design mistakes** [Lu 2012]
  - Uncoordinated functions (e.g. lack of batching)
  - Skippable functions (e.g. transparent draws)
  - Poor/unclear synchronization
  - Poor data structure selection
Summary

- Reasoning rigorously about performance is challenging
Summary

- Reasoning rigorously about performance is challenging
- Good tooling can allow you to investigate performance well
Summary

- Reasoning rigorously about performance is challenging
- Good tooling can allow you to investigate performance well
- **We can improve performance through**
  - compilers
  - managing data
  - managing code
  - better algorithmic thinking