

이펙티브 퍼포먼스 측정

김규래

서강대학교 전자공학과

September 29 2019



세션에서 다룰 내용

1 왜 성능 측정을 해야 하는가?

2 예제 소개

3 표본 갯수의 중요성

4 신뢰구간

5 가설 검정

6 Warm Starting

7 시스템 노이즈 최소화




세션에서 다루지 않을 내용

1 코드 프로파일링



최적화 이전에는 반드시 측정 또 측정!

cppcon 

CHANDLER CARRUTH
@chandlerc1024

Tuning C++
Benchmarks, and CPUs,
and Compilers! Oh My!

Measure first, tune what matters

“성능을 실제로 벤치마킹 해보기 전까지는
절대 성능을 논하지 말아야 한다.”
Chandler Carruth, CppCon 2015



Case 1: Andrei Alexandrescu



Andrei Alexandrescu

Speed Is Found In The Minds Of People

Video Sponsorship Provided By: **ansatz**

If You Thought It Ain't Weird Enough

- See "Binary Search Eliminates Branch Mispredictions"
- Concludes repeated binary search faster
- Specialized case (search only, powers of 2)
- Branchless binary search exists
- Coded it, tried it
- Slower... even than branchy binary search!

18 / 72

“이진 탐색이 당연히 빠르다고 교과서에서는 알려주지만,
introsort 에서는 더 느리다!”
Andrei Alexandrescu, CppCon 2019



Case 2: Emery Berger




Sept 13-14, 2019
thestrangeloop.com

Might be faster *today*...

A'

Run in a new directory
Changes layout

```
% pwd  
/users/vader  
% cd /users/luke
```



“실험을 돌린 폴더의 이름을
바꾸자 성능이 완전히 달라졌다.”
Emery Berger, Strange Loop 2019



왜 측정을 하기가 어려운가?

Abstraction layer 의 모든 레벨에서 노이즈가 발생한다

Computer System Abstraction Layers

- 1 Application
- 2 Compiler, Driver, Middleware
- 3 OS
- 4 Instruction Set Architecture
- 5 Micro-architecture
- 6 그 이하 전자과 stuff



컴퓨터 시스템의 다양한 노이즈 소스

Micro Architecture

-
-
-
-

Operating System

-
-
-
-



컴퓨터 시스템의 다양한 노이즈 소스

Micro Architecture

- Cache
- Branch-predictor
- Processor Frequency Scaling
- Out-of-order (OoO) pipeline

Operating System

-
-
-
-



컴퓨터 시스템의 다양한 노이즈 소스

Micro Architecture

- Cache
- Branch-predictor
- Processor Frequency Scaling
- Out-of-order (OoO) pipeline

Operating System

- Context switching
- Page Translation
- Blocking I/O
- Interrupts



Micro Architecture

Memory load 한 건 만으로도 발생할 수 있는 시간 편차¹

Memory Fetch Cost

- L1 cache 0.95ns
- L2 cache 3.00ns
- L3 cache 9.2ns
- RAM 66ns

¹AMD Ryzen 기준



컴퓨터 시스템의 다양한 노이즈 소스

Compiler

- Inlining
- Cache Prefetching
- Dead-code Removal



Matrix Multiplication

- 256×256 행렬곱
- 루프 vs 루프 언롤링

$$A \in \mathbb{R}^{256} \quad B \in \mathbb{R}^{256}$$

$$C = A \cdot B$$



Matrix Multiplication

- 256×256 행렬곱
- 루프 vs 루프 언롤링

$$A \in \mathbb{R}^{256} \quad B \in \mathbb{R}^{256}$$

$$C = A \cdot B$$

루프 언롤링을 하면 빨라진다니까 이왕 하는거 **화끈하게** 하면 되겠지?



Matrix Multiplication (Naive)

```

inline dynamic_matrix<double>
gemm(dynamic_matrix<double> const& A,
      dynamic_matrix<double> const& B)
{
    size_t M = A.rows();
    size_t K = A.columns();
    size_t N = B.columns();
    auto C = dynamic_matrix<double>(M, N);
    for(size_t i = 0; i < M; ++i)
    {
        for(size_t j = 0; j < N; ++j)
        {
            double sum = 0;
            for(size_t k = 0; k < K; ++k)
            {
                sum += A(i, k) * B(k, j);
            }
            C(i, j) = sum;
        }
    }
    return C;
}

```



Matrix Multiplication (Unrolled)

```

inline dynamic_matrix<double>
gemm(dynamic_matrix<double> const& A,
      dynamic_matrix<double> const& B)
{
    size_t M = A.rows();
    size_t K = A.columns();
    size_t N = B.columns();
    auto C = dynamic_matrix<double>(M, N);
    for(size_t i = 0; i < M; ++i)
    {
        for(size_t j = 0; j < N; ++j)
        {
            double sum = 0;
            for(size_t k = 0; k < K; ++k)
            {
                sum += A(i, k) * B(k, j);
            }
            C(i, j) = sum;
        }
    }
    return C;
}

```



Matrix Multiplication (Naive)

64배 Loop unrolling 을 진행

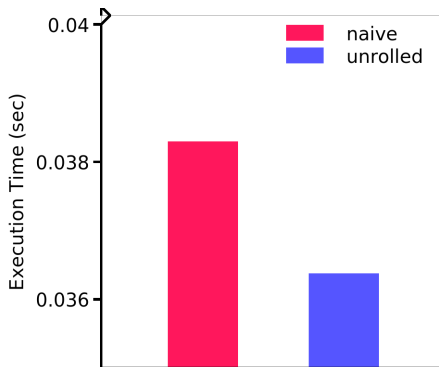
Loop unrolling 은 가장 기본적인 최적화 중에 하나

```
double sum = 0;
for(size_t k = 0; k < K; ++k)
{
    if(K - k > 64)
    {
        sum += A(i, k)      * B(k, j);
        sum += A(i, k+1)    * B(k+1, j);
        sum += A(i, k+2)    * B(k+2, j);
        sum += A(i, k+3)    * B(k+3, j);
        sum += A(i, k+4)    * B(k+4, j);
        sum += A(i, k+5)    * B(k+5, j);

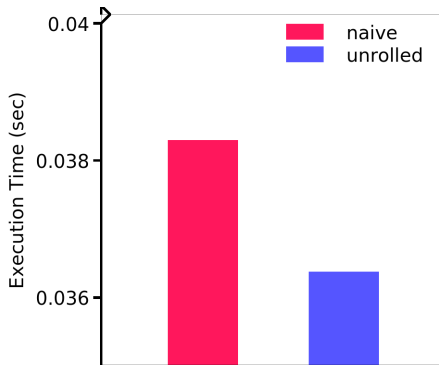
        /*      */
        k += 63;
    }
    else
        sum += A(i, k) * B(k, j);
}
```



벤치마크 결과



벤치마크 결과



여기서 결과를 받아들이면 **확증편향 (confirmation bias)**!



방금 본 그래프에서 가져야 하는 의문

- 어떤 환경에서 측정하였는가?
 - 입력한 데이터의 타입, 양은 얼마인가?
 - 데이터의 분포는 얼마인가?
- 어떻게 측정하였는가?
 - 몇번이나 측정을 했는가?
 - 측정한 샘플들을 어떻게 종합하였는가?
 - 신뢰구간은 얼마인가?
- 제시한 결과는 다른 조건에서도 성립하는가?
 - 데이터의 크기나 양이 바뀌어도 비슷한가?
 - 다른 환경에서 실행해도 비슷한가?



측정 환경

벤치마크를 읽을 때는 항상 측정 환경을 먼저 확인해야 합니다

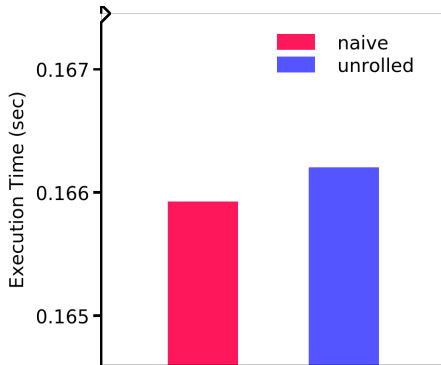
Environment

- intel i7-7700HQ
- Linux 4.14
- gcc 9.1 -00

측정 횟수: 1회



128 회 반복



정말 unrolling 을 한게 더 느릴까?



신뢰구간이란?

(지루한 ...) 정의

표본 데이터로부터 구성되는 통계량으로부터 모지모수를 추정하는 경우, 모지모수가 어떤 확률로 지정되는 구간에 들어갈 확률

코드의 실제 실행시간 μ 가 임의로 지정한 확률만큼 존재하는 범위!

Z-Score

$$\{T_1, T_2, T_3 \dots\}, \hat{T} = \mathbb{E}[T], T_i \sim \mathcal{N}(\mu, \sigma)$$

$$\hat{T} - 1.96 \frac{\sigma}{\sqrt{n}} \leq \mu \leq \hat{T} + 1.96 \frac{\sigma}{\sqrt{n}}$$



프로그램 실행 시간은 정규분포를 따르는가?

중심 극한 정리 (Central Limit Theorem)

충분히 많은 독립 확률변수들의 합은 표준 정규분포를 따른다

■ “독립 확률변수”

확률변수들은 컴퓨터 시스템의 노이즈 ϵ

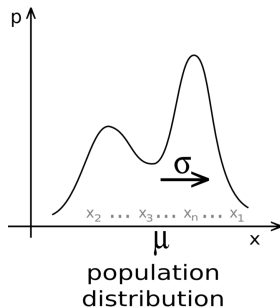
Temporal, spatial locality 로 인해서 완벽한 독립은 아니나 꽤 독립

■ “충분히 많은”

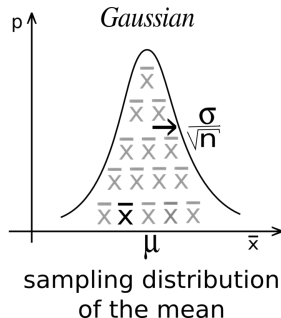
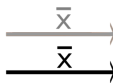
측정을 하는 구간이 넓을수록 더 많은 노이즈가 영향을 미치면서 정규분포를 따르게 됨



중심극한 정리



samples
of size n



실제로 컴퓨터 프로그램들은 쉽게 정규분포를 따르게 되는 것으로 알려져 있음²

²Adve, Vikram Sadanand, and Mary K. Vernon. "The influence of random delays on parallel execution times." 1993 ACM SIGMETRICS



프로그램 실행 시간은 정규분포를 따르는가?

```

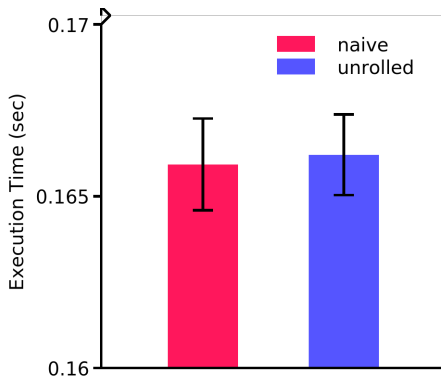
if( /* ... */ )
{
    /* probability 50% */
}
else
{
    /* probability 50% */
}

```

- 프로그램의 형태에 따라서는 정규분포와 매우 다른 분포가 나오기도 함
위의 예제의 경우 bimodal distribution 이 나올 확률이 높음
- 근본적으로 정규분포를 따르지 않는 실행시간도 있음
병렬 루프의 실행 시간은 $\max(T_1, T_2, \dots)$ 으로, 정규분포가 아님
- 프로그램이 최적화될수록 정규분포로부터 멀어짐



z-score 신뢰구간



Hypothesis Testing

가설 검정 (Hypothesis Testing)

가설검정은 연구자의 주장에 경험적 합리성이 있는가를 검토하는 통계적 방법이다.

- 가설 1 (H_1)
Unrolling 을 수행했을 때 execution time 의 평균이 더 낮다
- 가설 0 (H_0)
그렇지 않다 (Null hypothesis)

이 경우 execution time 은 정규분포를 따를 가능성이 높기 때문에 T-Test 를 사용할 수 있습니다.



검정 결과

- 가설 0 (H_0)
그렇지 않다 (Null hypothesis)
- 가설 1 (H_1)
Unrolling 을 수행했을 때 execution time 의 평균이 더 낮다

One sample T-Test

Test summary:

outcome with 95% confidence: fails to reject H_0

two-sided p-value: 0.5540

Details:

number of observations: 256

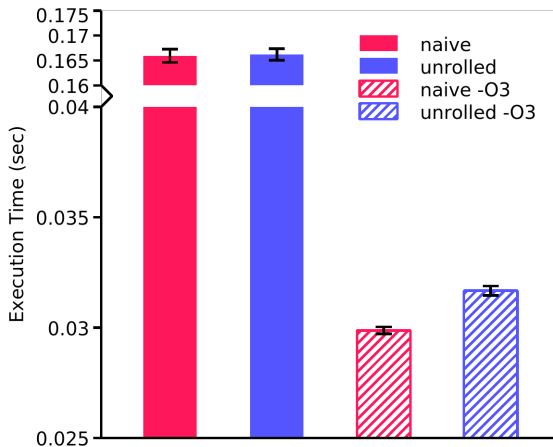
t-statistic: -0.5925551366027152

degrees of freedom: 255

empirical standard error: 0.0008589782340963979



compiler optimization



검정 결과

- 가설 0 (H_0)
그렇지 않다 (Null hypothesis)
- 가설 1 (H_1)
Unrolling 을 수행했을 때 execution time 의 평균이 더 낮다

One sample T-Test

Test summary:

outcome with 95% confidence: reject H_0

two-sided p-value: $<1e-27$

Details:

number of observations: 256

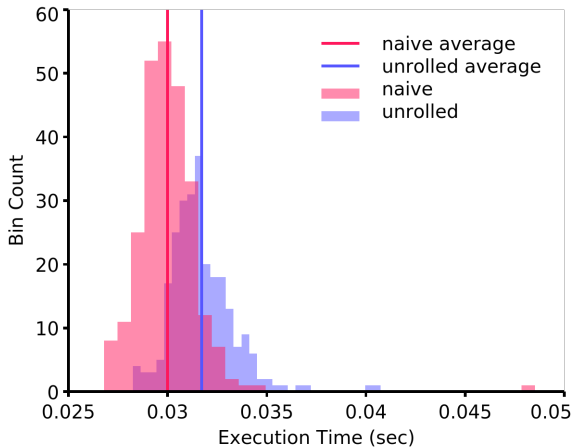
t-statistic: -12.382404994083672

degrees of freedom: 255

empirical standard error: 0.00013816665575860985



데이터의 히스토그램



Locality 로 인한 오차

Temporal locality, spatial locality 관련된 기능들,

- Cache
- Translation Look-aside Buffer (TLB)
- Branch predictor
- etc ...

로 인해서 생기는 오차를 줄이기 위해서는,

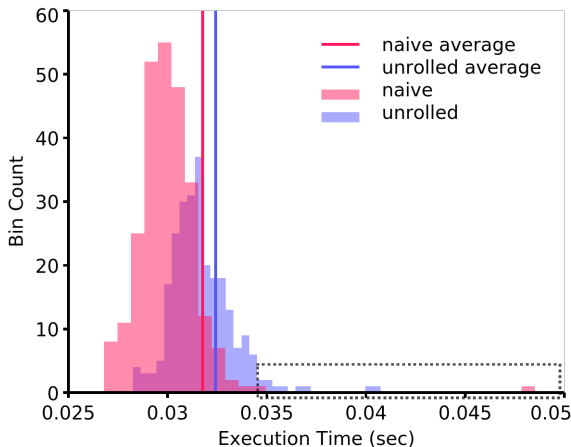
- 측정 샘플들을 최대한으로 모으거나
- 최초 몇 회의 측정들을 버리거나

해야 합니다.

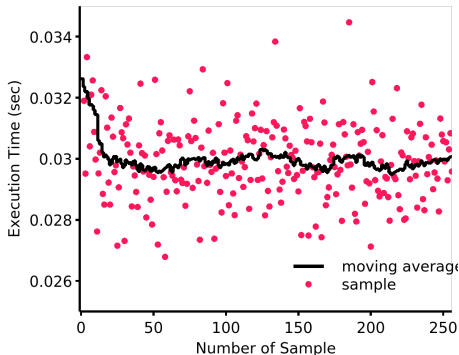


샘플의 수가 충분하지 않은 경우

Sample size = 16



Effect of Locality



- 데이터의 엄청난 노이즈 레벨
- 최초 샘플들의 positive bias

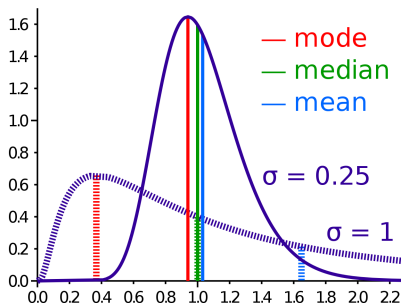
등을 확인할 수 있습니다.



중간값 (Median) 트릭

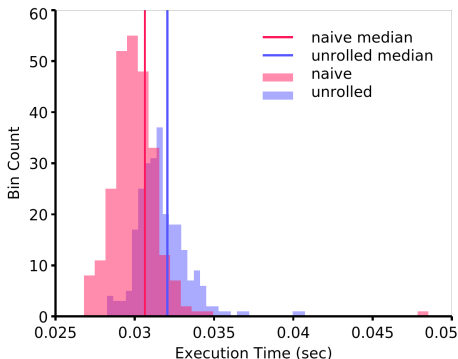
평균값 대신에 중간값을 사용하는 방법

- 데이터 노이즈에 덜 취약
- 극단적인 값 (cold start) 들에 덜 취약



샘플의 수가 충분하지 않은 경우

Sample size = 16, median



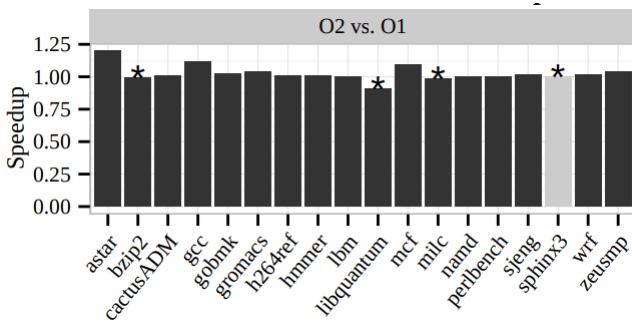
256 샘플들의 평균 만큼은 아니지만
더 평균에 가까운 것을 확인할 수 있습니다.



Warm starting 에 관한 여담

- 엄밀하게는 다양한 시스템 상태를 모두 실험해봐야 합니다

Stabilizer³ 와 같은 툴 사용



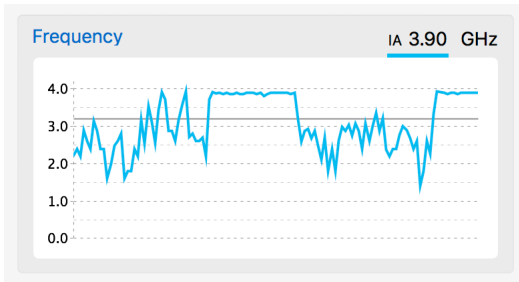
³Curtsinger, Charlie, and Emery D. Berger. "STABILIZER: statistically sound performance evaluation." ACM SIGPLAN Notices. 2013.



OS 노이즈 최소화

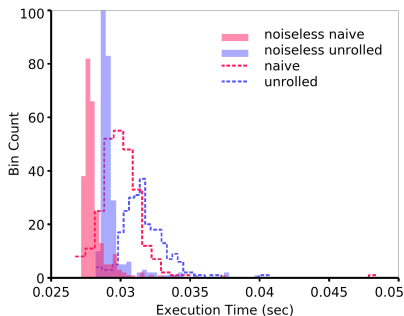
측정 OS 의 노이즈를 최소화해야됩니다.

- 측정과 무관한 프로그램들은 종료
- 서버의 경우 접속자 통제
- CPU 의 frequency scaling 해제



OS 노이즈 최소화

- Chromium 을 끄고 리눅스의 cpupower 를 이용해서 CPU Clock 을 3.0 GHz 로 고정한 결과



Kolmogorov-Smirnov Test

- naive: p-value < $1e-49$
- unrolled: p-value < $1e-66$



Out circuit 측정

회로의 부품을 측정할 때는 회로에서 제거를 하고 회로 밖에서 측정

- 회로 내에 있을 때는 다른 부품들과의 연결로 인해서 측정값이 부정확
- 인라이닝이 되는 경우에는 기존 코드와 매우 복잡하게 상호작용
- locality 로 인한 노이즈와도 연관이 있음

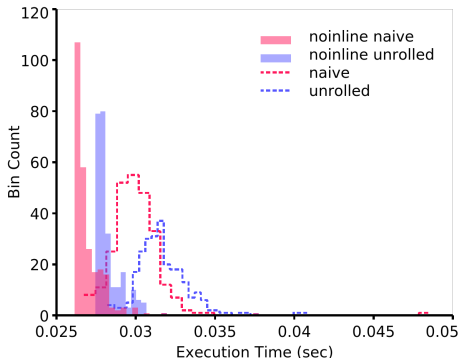
마이크로벤치마킹의 결과가 현장 성능과 매우 다를 수 있습니다.

- quick-bench.com, [googlebench](https://googlebench.com) 에서의 결과를 의심할것
- 실제 배포현장과 최대한 비슷한 측정 환경을 만드는데 중요



Noinline

■ gemm 을 noinline 으로 설정



Kolmogorov-Smirnov Test

- naive: p-value < $1e-58$
- unrolled: p-value < $1e-55$



결론

Takeways

- 최적화 등을 하기 전에는 반드시 측정 또 측정
- OS 단에서 최대한 노이즈를 통제
- 측정 횟수를 최대한 많이
- 측정하는 영역이 정규분포를 따르는지 확인
- 측정값을 비교할 때는 신뢰구간을 반드시 그릴것
- 유의미함을 확실하게 하기 위해서는 가설검정
- 표본이 적거나 noise 가 심한 경우 median 트릭
- 실제 배포현장과 최대한 비슷한 실험 환경
- 정말 엄밀한 성능이 필요한 경우 randomizer 사용



Noinline

결론

감사합니다.

