CSE211: Compiler Design
Nov. 29, 2021

- **Topic**: Optimization Policies
- **Discussion questions**:
  - How can you determine good optimizations for a program?

Announcements

- Friday is a big day:
  - Homework 4 is due
  - Paper reviews are due
  - Final project presentations
  - 2 hour class (1 hour extended after)

- Class on Wednesday canceled
  - use the time to study for the final, work on homeworks, or work on final project

Announcements

- Sign-up for time slots
  - Priority given to those who cannot attend off-hours
  - For those who cannot attend off-hours, please read the blog posts for the projects you miss
- 120 minutes for 11 presentations:
  - 9 minutes per presentation (HARD, I will be using the unforgiving iphone timer)
  - try for 7 minute presentation and 2 minutes for questions.
  - Use your own computer, or if you send me your presentation, you can use mine.

Announcements

- For blog post:
  - please submit as a PR to the class git repo:
  - https://github.com/SorensenUCSC/CSE211-fa2021/
  - follow the example project
  - create a directory with your name, include an .md file and all images
  - link to it in projects.md
- Write the blog post like how you'd like to read one! Lots of background, lots of images and code snippets.
  - Use only original images please!
  - Should roughly be the same amount of content as the final report would be.

Announcements

- For reports (project and paper):
    - if you are having trouble filling in the space:
    - give more background. Imagine you are giving a CSE211 lecture!
    - give more examples and walk through them
    - show code snippets
    - discuss related works

- *At some point in your career you will transition to wanting more space rather than trying to fill up space!*

Announcements

- Office hours:
  - Since thanksgiving office hours got canceled, I will hold a make-up hour tomorrow from 2 - 3 pm
  - There will also be normal Thursday office hours
- After Friday:
  - I will start grading HW3, HW4 and paper reviews
  - Please discuss grades with me ASAP if there are issues

Announcements

- SETs:
  - Please fill them out!
  - They are important for non-core classes like this one
- Individual feedback is also appreciated: feel free to send an email with any thoughts you have:
  - what you enjoyed ☺
  - what you wish we would have discussed
  - what you wish we would have spent more time on
- I will also release an anonymous survey on canvas asking some of these questions. It should not replace the SETs though!

CSE211: Compiler Design
Nov. 29, 2021

- **Topic**: Optimization Policies
- **Discussion questions**:
  - How can you determine good optimizations for a program?

CSE211: Compiler Design

Nov. 29, 2021

- **Topic**: Optimization Policies

- **Discussion questions**:
  - How can you determine good optimizations for a program?

CSE211: Compiler Design

Nov. 29, 2021

- **Topic**: Optimization Policies

- **Discussion questions**:
  - How can you determine good optimizations for a program?

- **auto-tuning**: Halide approach

CSE211: Compiler Design

Nov. 29, 2021

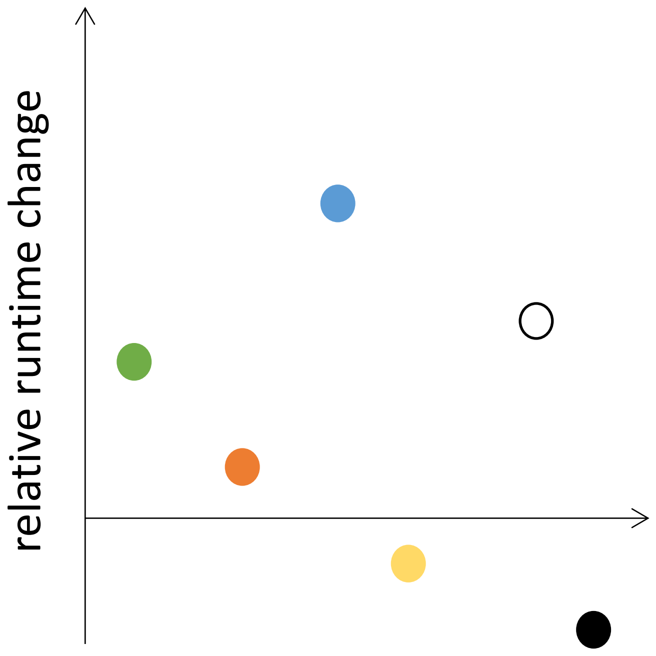
- **Topic**: Optimization Policies
- **Discussion questions**:
  - How can you determine good optimizations for a program?

- **auto-tuning**: Halide approach
- **exhaustive enumeration**: irgl approach

Rank-based

Pros and cons for this approach?

are points more likely to be above or below the line?



Opt. On

Optimizations
LB – Local
LB - Subgroup
LB - Workgroup
OC - Sync
RMW-Cls

Optimization Space

Applications
BFS
SSSP
PR
CC
MIS
MST
TRI

Inputs
Uniform
RMAT
NY-Road

GPUs
Nvidia-Quadro
Nvidia-1080
AMD-R9
Intel-Iris
Intel-HD5500
ARM-Mali T628

Domain

CSE211: Compiler Design

Nov. 29, 2021

- **Topic**: Optimization Policies

- **Discussion questions**:
  - How can you determine good optimizations for a program?

- **auto-tuning**: Halide approach
- **exhaustive enumeration**: irgl approach
- What else?

Big question

- When should optimizations be enabled or disabled?
  - if optimization adds a large compile time
  - if optimization makes debugging harder
  - if optimization makes smaller binaries
  - if optimization is not well tested
  - if optimization is likely to provide a performance increase

What do modern compilers do?

- gcc?
  - -O0, -O1, -O2

- See differences at:

  https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- different optimizations for different use cases
  - -Os, -Og, -Ofast

Making programs go faster

- All of the optimizations we’ve examined have had performance trade-offs

- Local value numbering?

```
x = a + b;
...
y = a + b;
```

```
x = a + b;
...
y = x;
```

what can go wrong?

x might have gone to memory if there isn’t enough registers. A memory access is more expensive than some arithmetic operations

Same issue for Pipelining and Super Scalar re-orderings!

Making programs go faster

- All of the optimizations we’ve examined have had performance trade-offs

- Loop unrolling?
  - Pros/cons?

- Making DOALL loops parallel?
  - Pros/cons?

Compilers are evaluated on benchmark suites

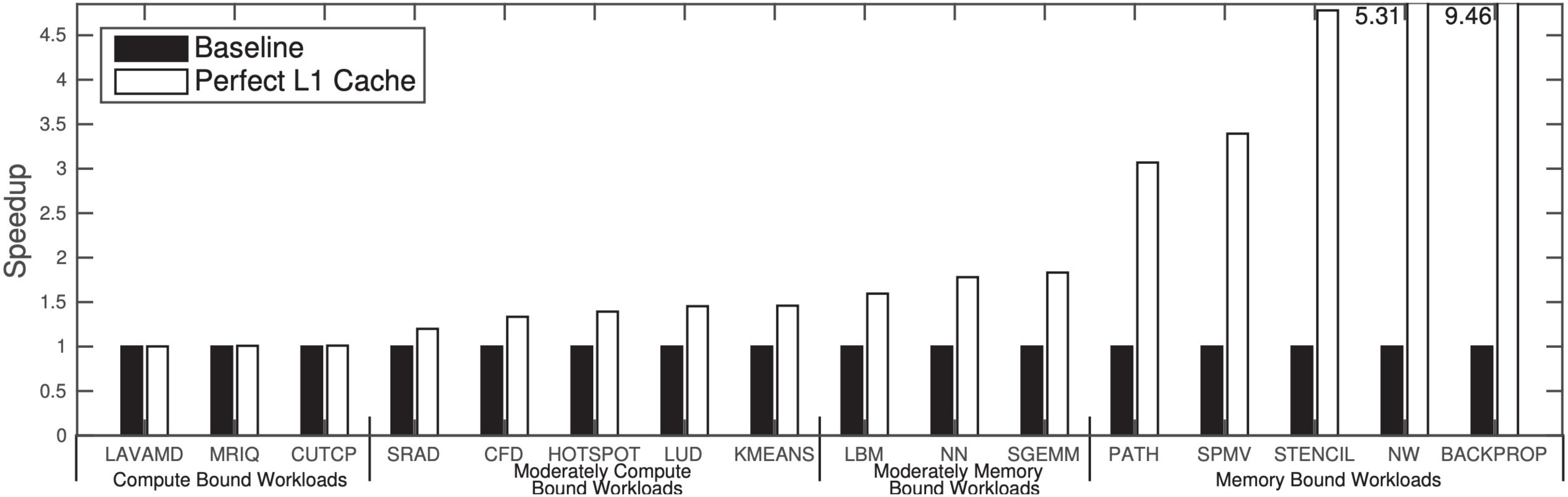
- Scientific computing
  - Rodinia, Parboil, Linpack
- Managed Languages:
  - Decapo (Java)
- Heterogeneous systems
  - SHOC
- GPU
  - Magma
- Graphs
  - GAPs

combination?
https://www.phoronix.com/scan.php?page=article&item=gcc-clang-2019&num=1

For general compilers, performance differences are tiny: e.g. 2%

Benchmarks can have a variety of characteristics

parboil and rodinia



From: DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. Ham et al., MICRO 2015

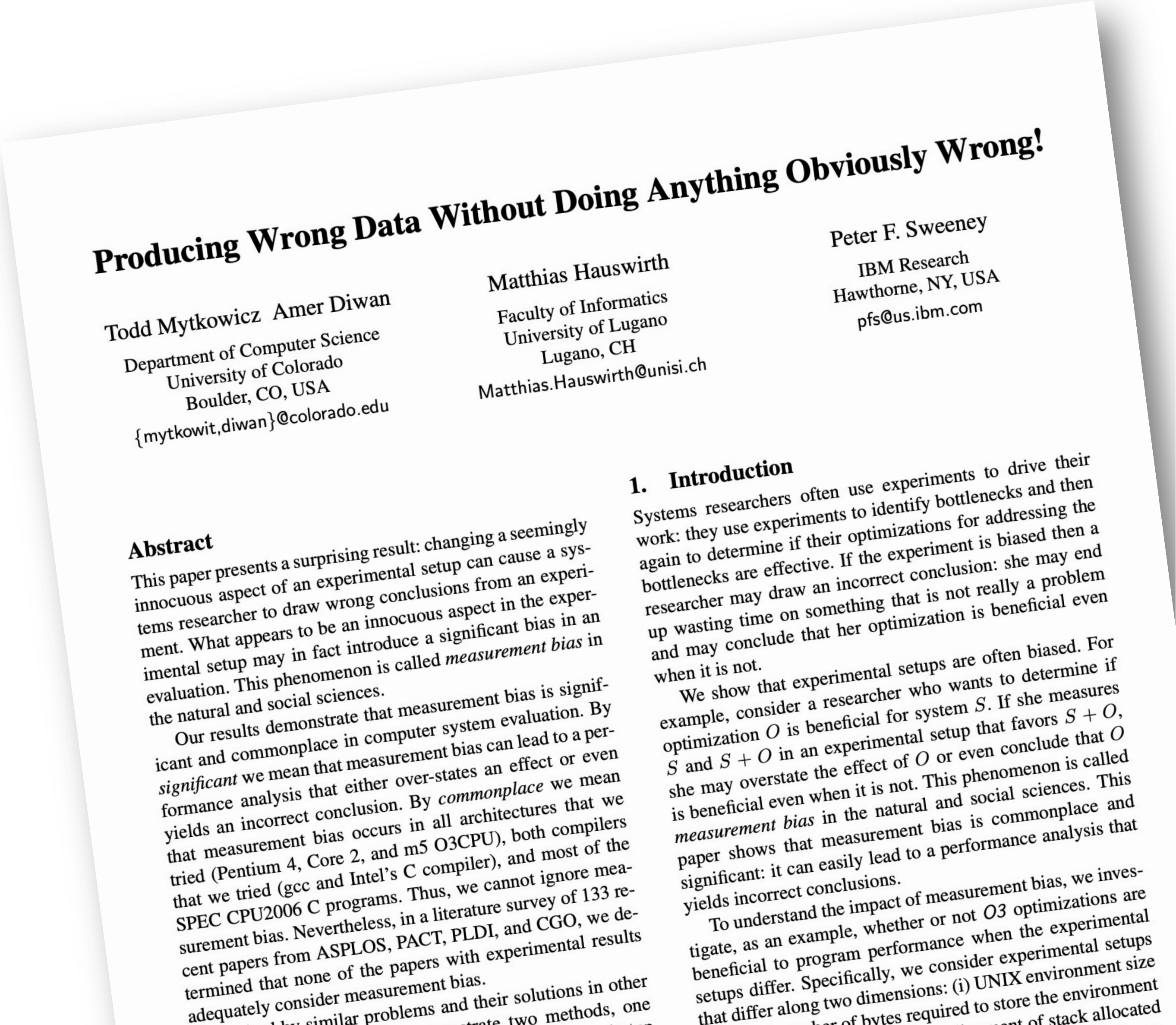
Running benchmarks

- Just run it?

- Need to be careful...

Measurement bias

Environment factors can influence performance measurements. Sometimes significantly!



Producing Wrong Data Without Doing Anything Obviously Wrong!

Todd Mytkowicz Amer Diwan
Department of Computer Science
University of Colorado
Boulder, CO, USA
{mytkowit,diwan}@colorado.edu

Matthias Hauswirth
Faculty of Informatics
University of Lugano
Lugano, CH
Matthias.Hauswirth@unisi.ch

Peter F. Sweeney
IBM Research
Hawthorne, NY, USA
pfs@us.ibm.com

Abstract

This paper presents a surprising result: changing a seemingly innocuous aspect of an experimental setup can cause a systems researcher to draw wrong conclusions from an experiment. What appears to be an innocuous aspect in the experimental setup may in fact introduce a significant bias in an evaluation. This phenomenon is called *measurement bias* in the natural and social sciences.

Our results demonstrate that measurement bias is significant and commonplace in computer system evaluation. By *significant* we mean that measurement bias can lead to a performance analysis that either over-states an effect or even yields an incorrect conclusion. By *commonplace* we mean that measurement bias occurs in all architectures that we tried (Pentium 4, Core 2, and m5 O3CPU), both compilers that we tried (gcc and Intel's C compiler), and most of the SPEC CPU2006 C programs. Thus, we cannot ignore measurement bias. Nevertheless, in a literature survey of 133 recent papers from ASPLOS, PACT, PLDI, and CGO, we determined that none of the papers with experimental results adequately consider measurement bias.

1. Introduction

Systems researchers often use experiments to drive their work: they use experiments to identify bottlenecks and then again to determine if their optimizations for addressing the bottlenecks are effective. If the experiment is biased then a researcher may draw an incorrect conclusion: she may end up wasting time on something that is not really a problem and may conclude that her optimization is beneficial even when it is not.

We show that experimental setups are often biased. For example, consider a researcher who wants to determine if optimization O is beneficial for system S. If she measures S and $S+O$ in an experimental setup that favors $S+O$, she may overstate the effect of O or even conclude that O is beneficial even when it is not. This phenomenon is called *measurement bias* in the natural and social sciences. This paper shows that measurement bias is commonplace and significant: it can easily lead to a performance analysis that yields incorrect conclusions.

To understand the impact of measurement bias, we investigate, as an example, whether or not *O3* optimizations are beneficial to program performance when the experimental setups differ. Specifically, we consider experimental setups that differ along two dimensions: (i) UNIX environment size

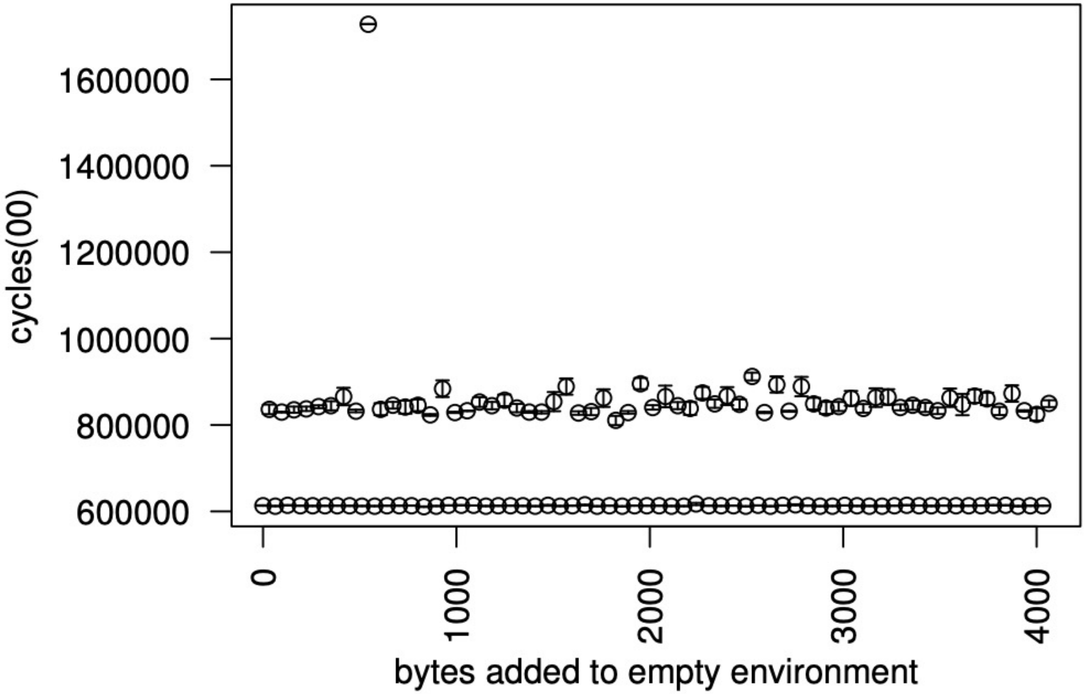
Measurement bias

Size of environment variables on Linux?

From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias

Size of environment variables on Linux?



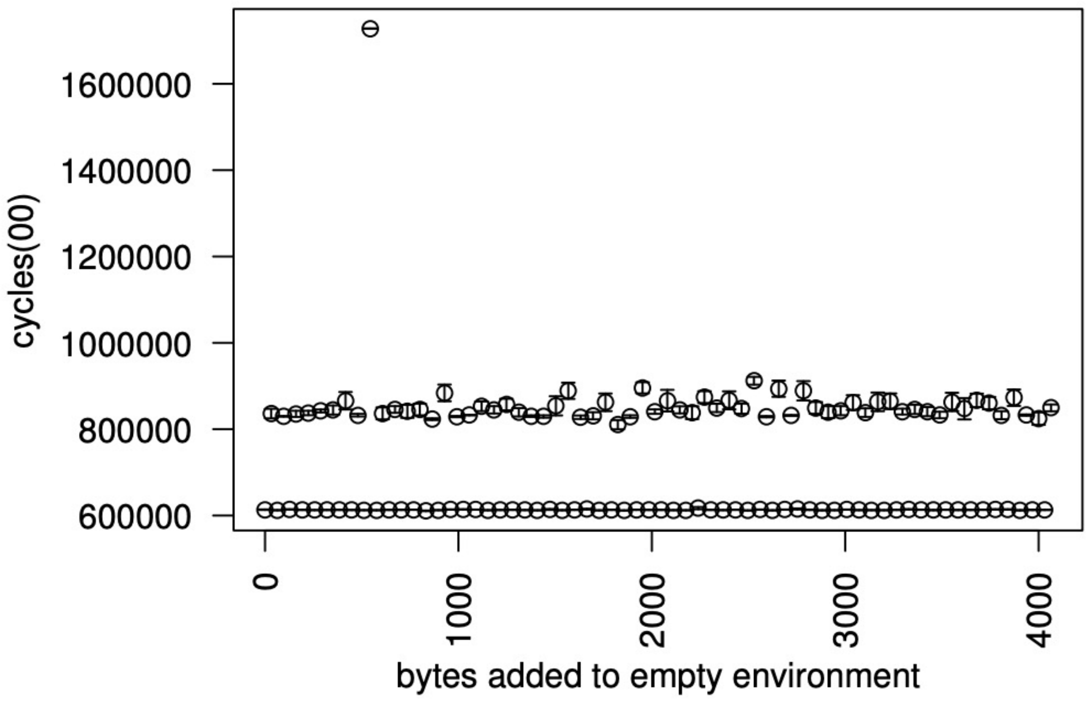
From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias

Size of environment variables on Linux?

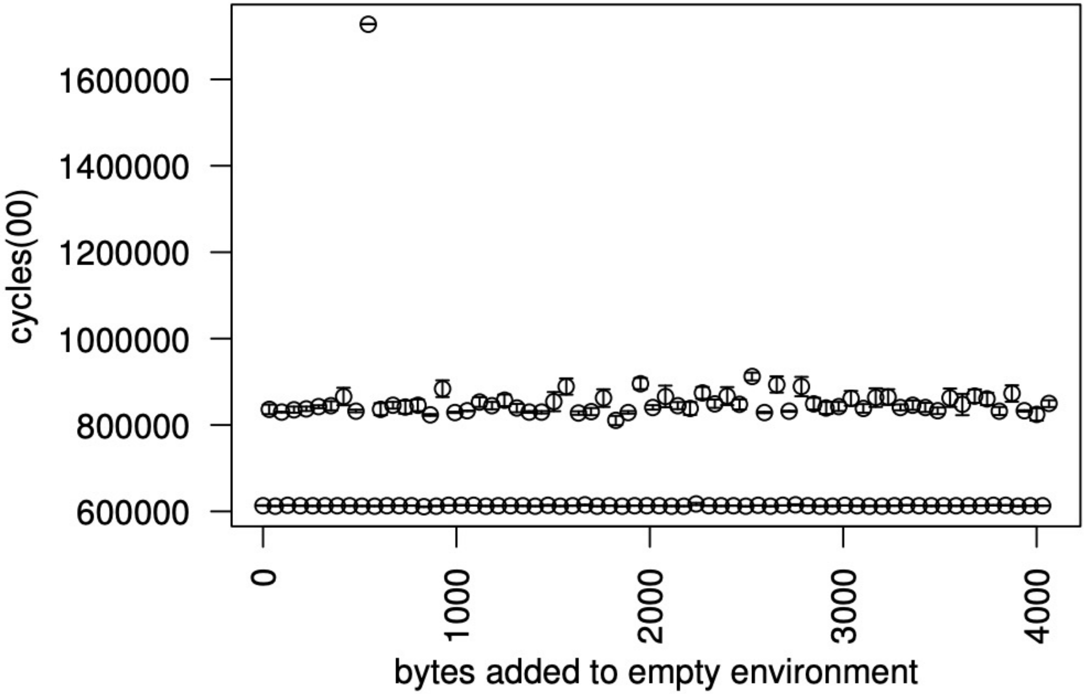
frequently performance difference is 33%

Max is 300%



From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias



Size of environment variables on Linux?

frequently performance difference is 33%

Max is 300%

This phenomenon occurs because the UNIX environment is loaded into memory before the call stack. Thus, changing the UNIX environment size changes the location of the call stack which in turn affects the alignment of local variables in various hardware structures.

From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias

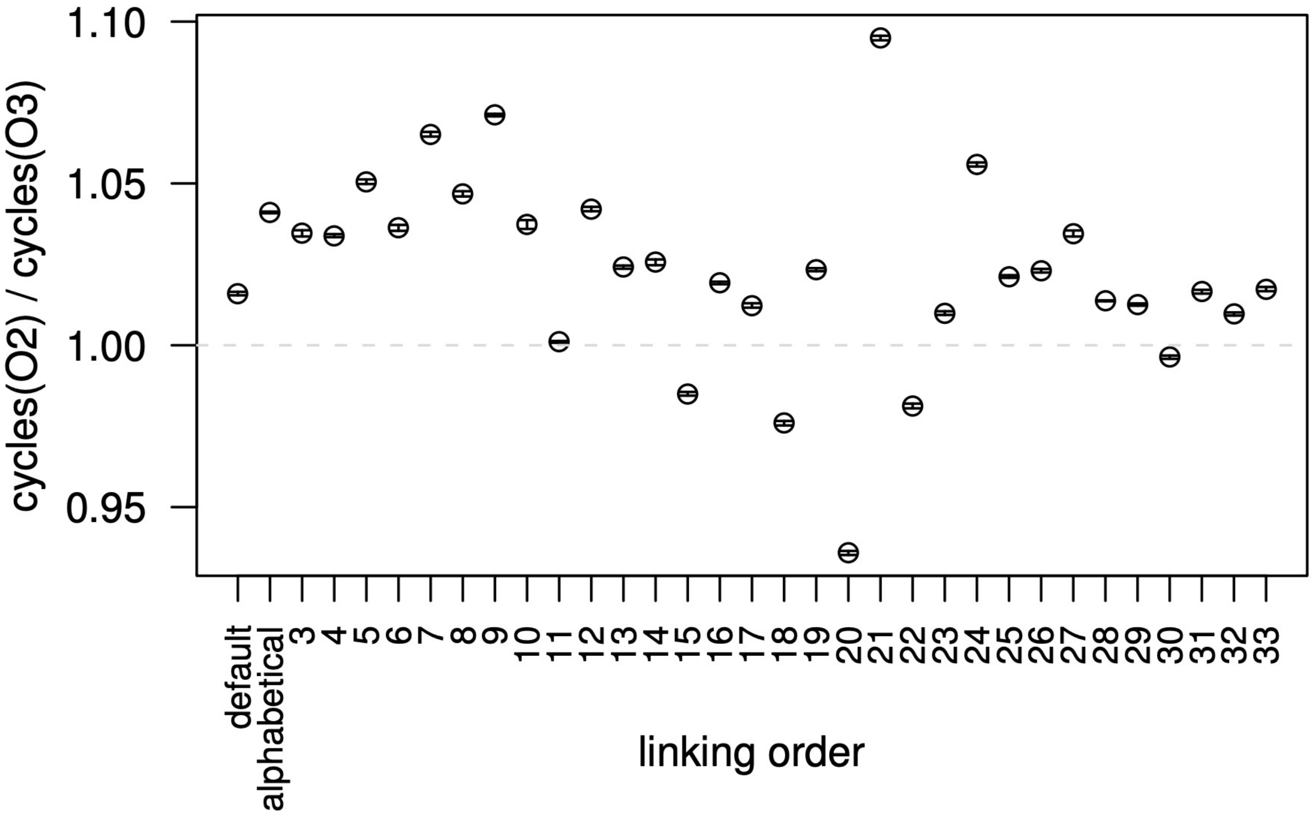
The order in which libraries are linked?

From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias

The order in which libraries are linked?

In some cases O3 is slower than O2!



(a) Perlbench

From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Measurement bias

Processes running on other cores can influence timing:

Intel chips: max of **1.15x** difference
Mobile chips: max of **10x** difference

From "Slow and Steady: Measuring and Tuning Multicore Interference" Iorga et al. RTAS 2019

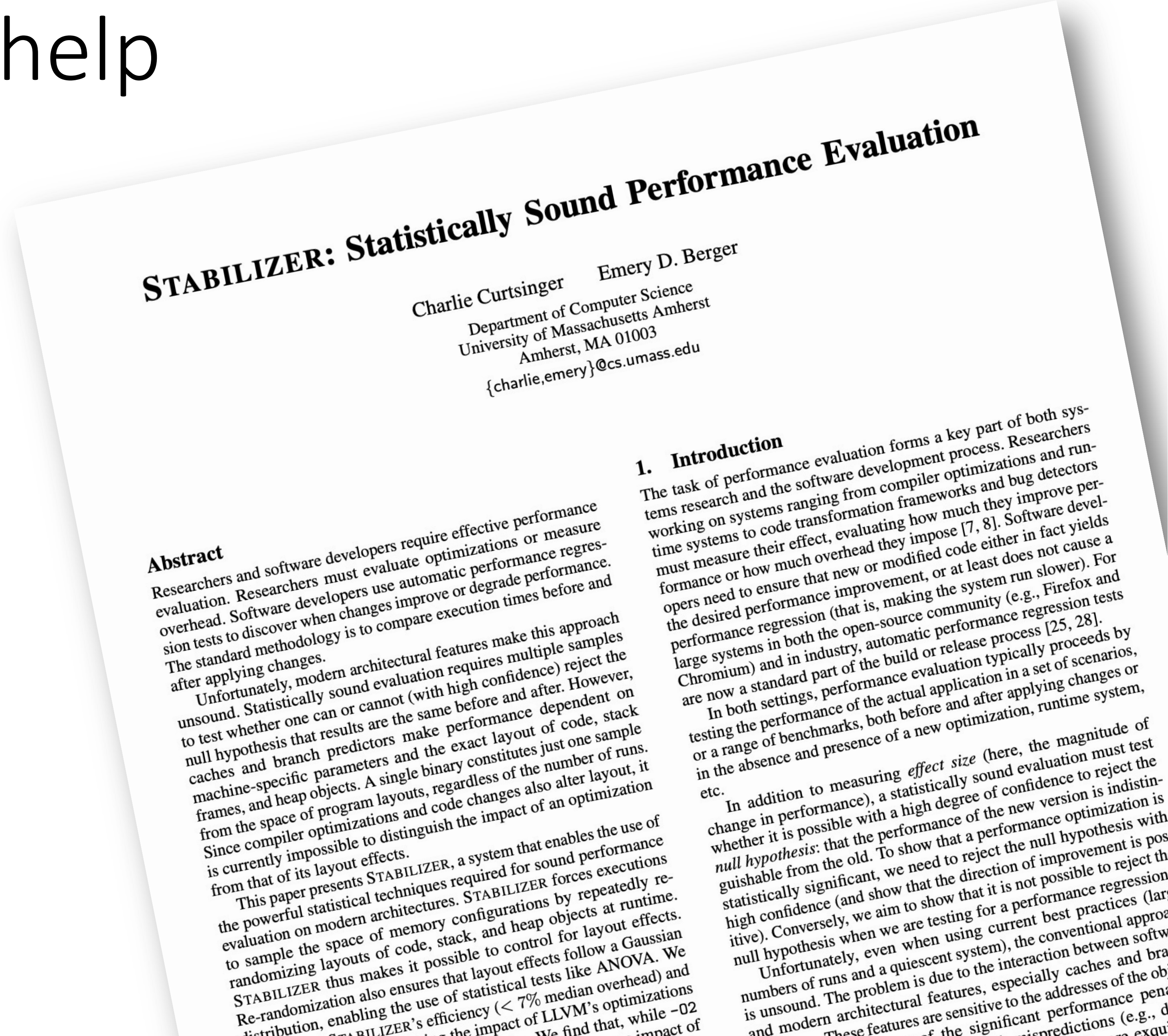
How to combat measurement bias?

- Run lots of times
  - The homeworks in this class have not emphasized this enough!
- Run a large enough benchmark suite
- Run in many different configurations (environment sizes, etc.)
- Results in the paper show that the difference between O2 and O3 is an average of 1.007x

From “Producing Wrong Data Without Doing Anything Obviously Wrong!” Mytkowicz ASPLOS 2009

Stabilizer: a tool to help

A compiler tool to...
evaluate compiler optimizations!



STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger Emery D. Berger
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Abstract

Researchers and software developers require effective performance evaluation. Researchers must evaluate optimizations or measure overhead. Software developers use automatic performance regression tests to discover when changes improve or degrade performance. The standard methodology is to compare execution times before and after applying changes.

Unfortunately, modern architectural features make this approach unsound. Statistically sound evaluation requires multiple samples to test whether one can or cannot (with high confidence) reject the null hypothesis that results are the same before and after. However, caches and branch predictors make performance dependent on machine-specific parameters and the exact layout of code, stack frames, and heap objects. A single binary constitutes just one sample from the space of program layouts, regardless of the number of runs. Since compiler optimizations and code changes also alter layout, it is currently impossible to distinguish the impact of an optimization from that of its layout effects.

This paper presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by repeatedly re-randomizing layouts of code, stack, and heap objects at runtime. STABILIZER thus makes it possible to control for layout effects. Re-randomization also ensures that layout effects follow a Gaussian distribution, enabling the use of statistical tests like ANOVA. We ... STABILIZER's efficiency (< 7% median overhead) and ... the impact of LLVM's optimizations ... We find that, while -O2 ... impact of

1. Introduction

The task of performance evaluation forms a key part of both systems research and the software development process. Researchers working on systems ranging from compiler optimizations and runtime systems to code transformation frameworks and bug detectors must measure their effect, evaluating how much they improve performance or how much overhead they impose [7, 8]. Software developers need to ensure that new or modified code either in fact yields the desired performance improvement, or at least does not cause a performance regression (that is, making the system run slower). For large systems in both the open-source community (e.g., Firefox and Chromium) and in industry, automatic performance regression tests are now a standard part of the build or release process [25, 28].

In both settings, performance evaluation typically proceeds by testing the performance of the actual application in a set of scenarios, or a range of benchmarks, both before and after applying changes or in the absence and presence of a new optimization, runtime system, etc.

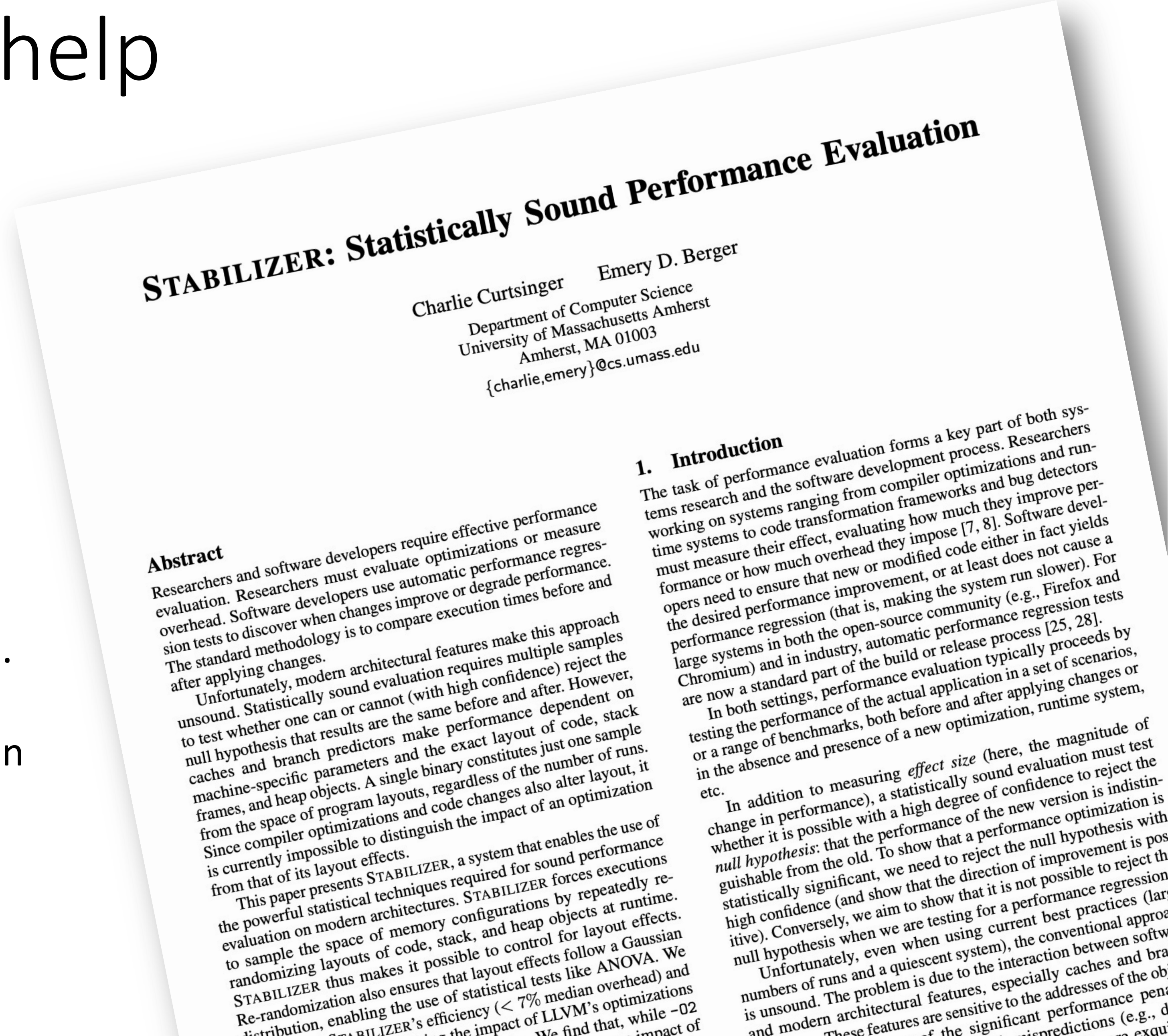
In addition to measuring *effect size* (here, the magnitude of change in performance), a statistically sound evaluation must test whether it is possible with a high degree of confidence to reject the *null hypothesis*: that the performance of the new version is indistinguishable from the old. To show that a performance optimization is statistically significant, we need to reject the null hypothesis with high confidence (and show that the direction of improvement is pos- itive). Conversely, we aim to show that it is not possible to reject the null hypothesis when we are testing for a performance regression. Unfortunately, even when using current best practices (lar- numbers of runs and a quiescent system), the conventional approa- is unsound. The problem is due to the interaction between softw- and modern architectural features, especially caches and bra- These features are sensitive to the addresses of the ob- ...

Stabilizer: a tool to help

A compiler tool to...
evaluate compiler optimizations!

Given a program p, Stabilizer creates $S(p)$

- Memory allocation is randomized in the heap.
- Function calls are trapped and their location in program memory is randomized.
- Function stack locations are randomly offset.



STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger Emery D. Berger
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Abstract

Researchers and software developers require effective performance evaluation. Researchers must evaluate optimizations or measure overhead. Software developers use automatic performance regression tests to discover when changes improve or degrade performance. The standard methodology is to compare execution times before and after applying changes.

Unfortunately, modern architectural features make this approach unsound. Statistically sound evaluation requires multiple samples to test whether one can or cannot (with high confidence) reject the null hypothesis that results are the same before and after. However, caches and branch predictors make performance dependent on machine-specific parameters and the exact layout of code, stack frames, and heap objects. A single binary constitutes just one sample from the space of program layouts, regardless of the number of runs. Since compiler optimizations and code changes also alter layout, it is currently impossible to distinguish the impact of an optimization from that of its layout effects.

This paper presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by repeatedly re-randomizing layouts of code, stack, and heap objects at runtime. STABILIZER thus makes it possible to control for layout effects. Re-randomization also ensures that layout effects follow a Gaussian distribution, enabling the use of statistical tests like ANOVA. We ... STABILIZER's efficiency (< 7% median overhead) and ... the impact of LLVM's optimizations ... We find that, while -O2 ... impact of

1. Introduction

The task of performance evaluation forms a key part of both systems research and the software development process. Researchers working on systems ranging from compiler optimizations and runtime systems to code transformation frameworks and bug detectors must measure their effect, evaluating how much they improve performance or how much overhead they impose [7, 8]. Software developers need to ensure that new or modified code either in fact yields the desired performance improvement, or at least does not cause a performance regression (that is, making the system run slower). For large systems in both the open-source community (e.g., Firefox and Chromium) and in industry, automatic performance regression tests are now a standard part of the build or release process [25, 28].

In both settings, performance evaluation typically proceeds by testing the performance of the actual application in a set of scenarios, or a range of benchmarks, both before and after applying changes or in the absence and presence of a new optimization, runtime system, etc.

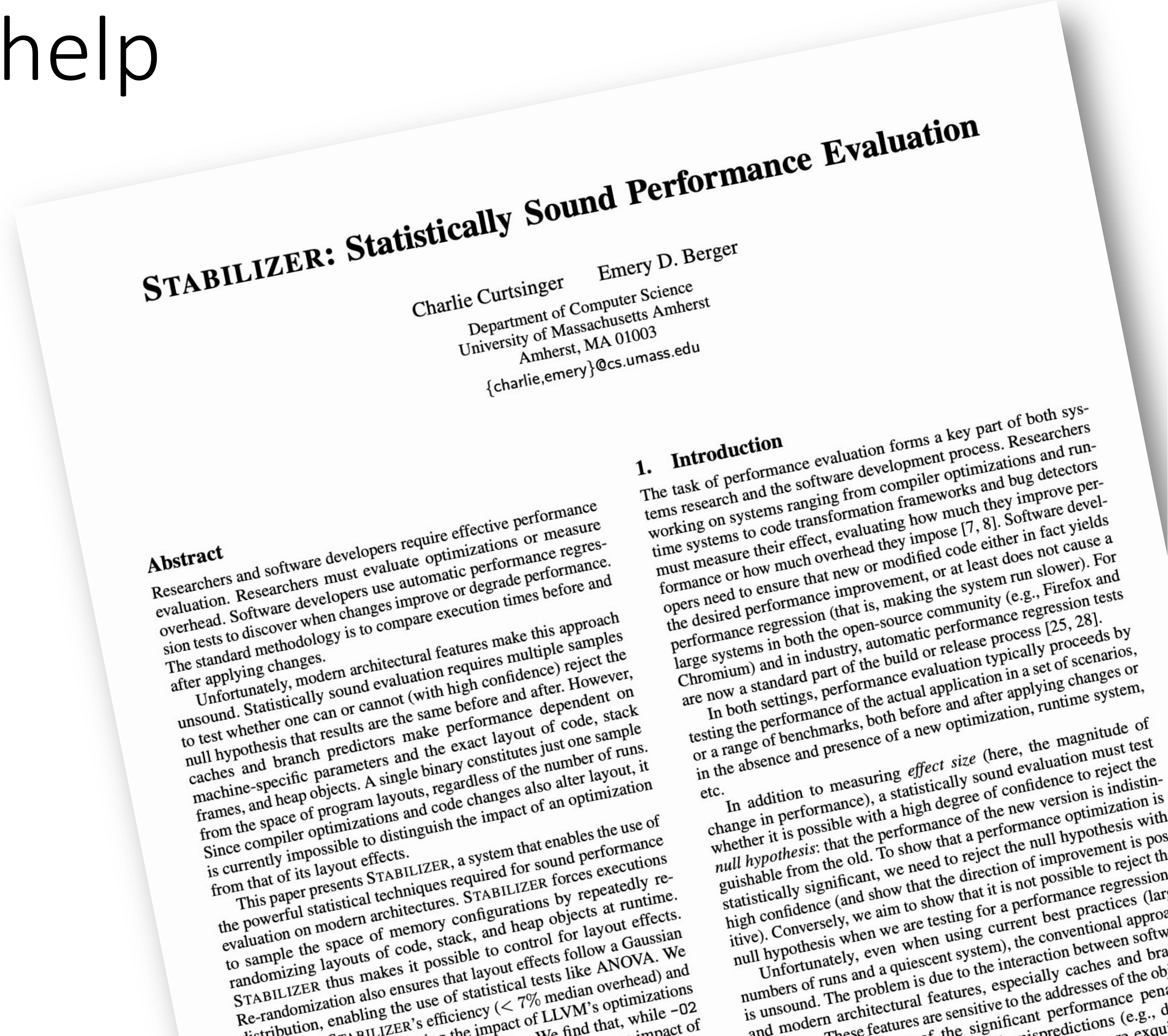
In addition to measuring *effect size* (here, the magnitude of change in performance), a statistically sound evaluation must test whether it is possible with a high degree of confidence to reject the *null hypothesis*: that the performance of the new version is indistinguishable from the old. To show that a performance optimization is statistically significant, we need to reject the null hypothesis with high confidence (and show that the direction of improvement is pos...

Stabilizer: a tool to help

A compiler tool to...
evaluate compiler optimizations!

Running $S(p)$ for many iterations provides a uniform distribution of runtimes.

They show that there is no statistical difference between O2 and O3 in LLVM (2013)



STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger   Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Abstract

Researchers and software developers require effective performance evaluation. Researchers must evaluate optimizations or measure overhead. Software developers use automatic performance regression tests to discover when changes improve or degrade performance. The standard methodology is to compare execution times before and after applying changes.

Unfortunately, modern architectural features make this approach unsound. Statistically sound evaluation requires multiple samples to test whether one can or cannot (with high confidence) reject the null hypothesis that results are the same before and after. However, caches and branch predictors make performance dependent on machine-specific parameters and the exact layout of code, stack frames, and heap objects. A single binary constitutes just one sample from the space of program layouts, regardless of the number of runs. Since compiler optimizations and code changes also alter layout, it is currently impossible to distinguish the impact of an optimization from that of its layout effects.

This paper presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by repeatedly re-randomizing layouts of code, stack, and heap objects at runtime. STABILIZER thus makes it possible to control for layout effects. Re-randomization also ensures that layout effects follow a Gaussian distribution, enabling the use of statistical tests like ANOVA. We

1. Introduction

The task of performance evaluation forms a key part of both systems research and the software development process. Researchers working on systems ranging from compiler optimizations and runtime systems to code transformation frameworks and bug detectors must measure their effect, evaluating how much they improve performance or how much overhead they impose [7, 8]. Software developers need to ensure that new or modified code either in fact yields the desired performance improvement, or at least does not cause a performance regression (that is, making the system run slower). For large systems in both the open-source community (e.g., Firefox and Chromium) and in industry, automatic performance regression tests are now a standard part of the build or release process [25, 28].

In both settings, performance evaluation typically proceeds by testing the performance of the actual application in a set of scenarios, or a range of benchmarks, both before and after applying changes or in the absence and presence of a new optimization, runtime system, etc.

In addition to measuring *effect size* (here, the magnitude of change in performance), a statistically sound evaluation must test whether it is possible with a high degree of confidence to reject the *null hypothesis*: that the performance of the new version is indistinguishable from the old. To show that a performance optimization is statistically significant, we need to reject the null hypothesis with high confidence (and show that the direction of improvement is pos-

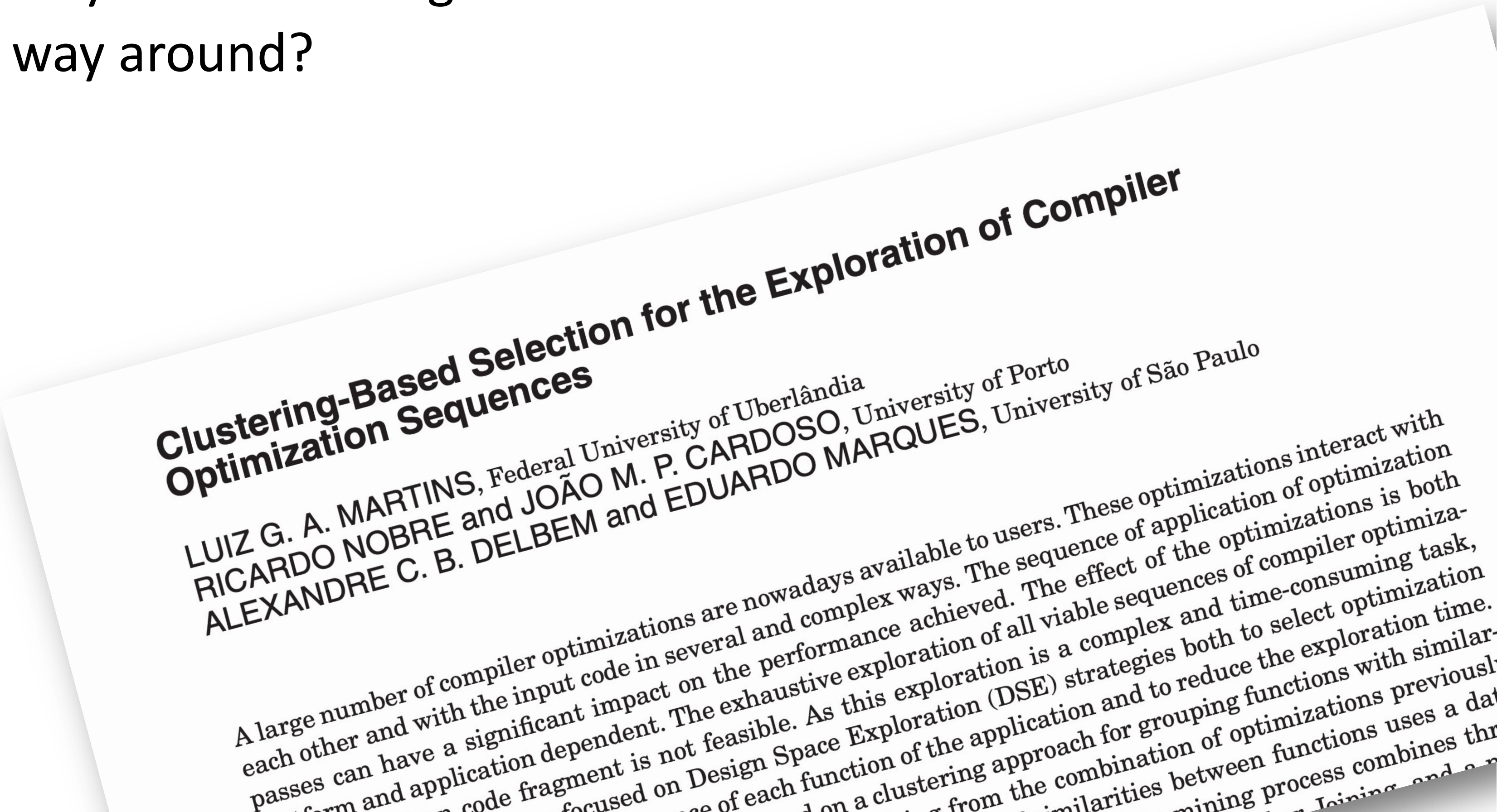
Order in which optimizations are applied?

- Example:
  - Loop unrolling followed by ILP scheduling
  - What about the other way around?

Order in which optimizations are applied?

- Example:
  - Loop unrolling followed by ILP scheduling
  - What about the other way around?

they can achieve 7% performance improvement over O2 by specializing optimization order


Clustering-Based Selection for the Exploration of Compiler Optimization Sequences

LUIZ G. A. MARTINS, Federal University of Uberlândia
RICARDO NOBRE and JOÃO M. P. CARDOSO, University of Porto
ALEXANDRE C. B. DELBEM and EDUARDO MARQUES, University of São Paulo

A large number of compiler optimizations are nowadays available to users. These optimizations interact with each other and with the input code in several and complex ways. The sequence of application of optimization passes can have a significant impact on the performance achieved. The effect of the optimizations is both platform and application dependent. The exhaustive exploration of all viable sequences of compiler optimizations for a given code fragment is not feasible. As this exploration is a complex and time-consuming task, ... focused on Design Space Exploration (DSE) strategies both to select optimization ... of each function of the application and to reduce the exploration time. ... on a clustering approach for grouping functions with similar- ... from the combination of optimizations previousl... similarities between functions uses a da... mining process combines thr...

Compiler optimization domains

- General case:
  - Compile many diverse pieces of code, run on many different inputs and architectures
  - examples: gcc at -O3

Compiler optimization domains

- General case:
  - Compile many diverse pieces of code, run on many different inputs and architectures
  - examples: gcc at -O3

- Fully Specialized:
  - Compile one piece of code for one architecture and one input
  - examples?
  - optimizations?

Compiler optimization domains

- General case:
  - Compile many diverse pieces of code, run on many different inputs and architectures
  - examples: gcc at -O3
- Fully Specialized:
  - Compile one piece of code for one architecture and one input
  - examples?
  - optimizations?
- Semi-specialized?

Semi-specialization Examples

One binary, many architectures

- x86 binary runs on machines with different number of cores, pipeline depths, super scalar widths etc.

Many programs, one architecture

- Modern compilers are often tuned (or query device info) when they are installed

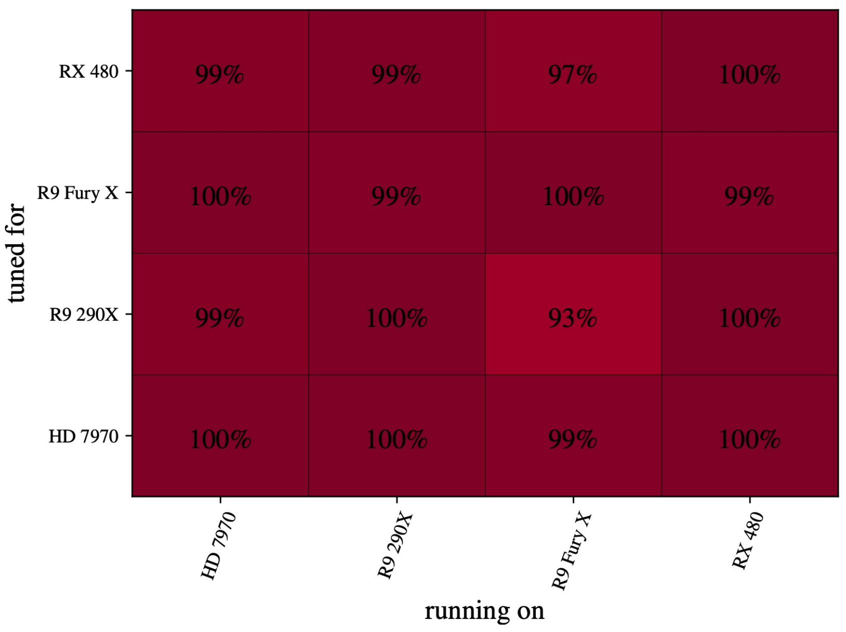
Are fully specialized applications portable?

tuned for: \ ran on:	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

from: “Analyzing and improving performance portability of OpenCL applications via auto-tuning” Price et al. IWOCL 2017

Tuning for same vendor?

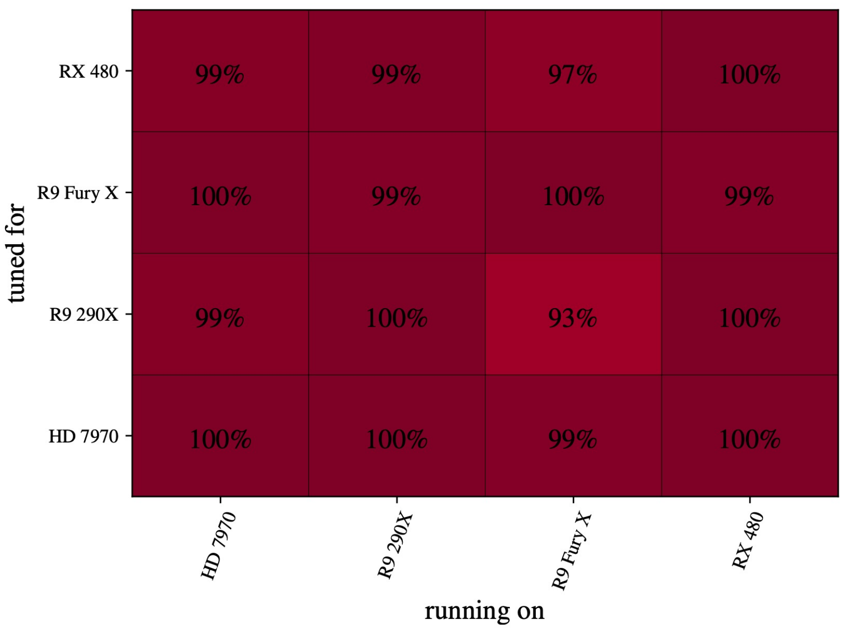
AMD



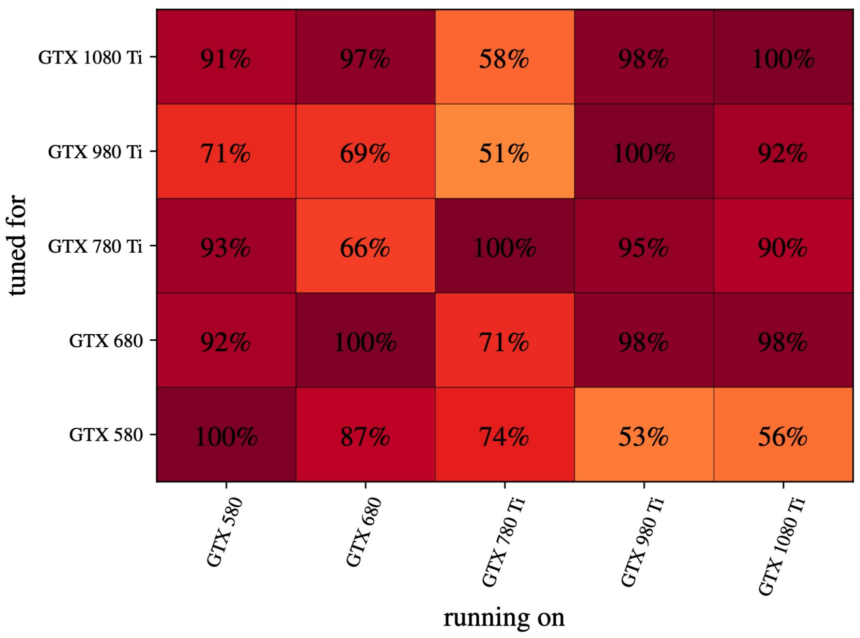
from: “Analyzing and improving performance portability of OpenCL applications via auto-tuning” Price et al. IWOCL 2017

Tuning for same vendor?

AMD



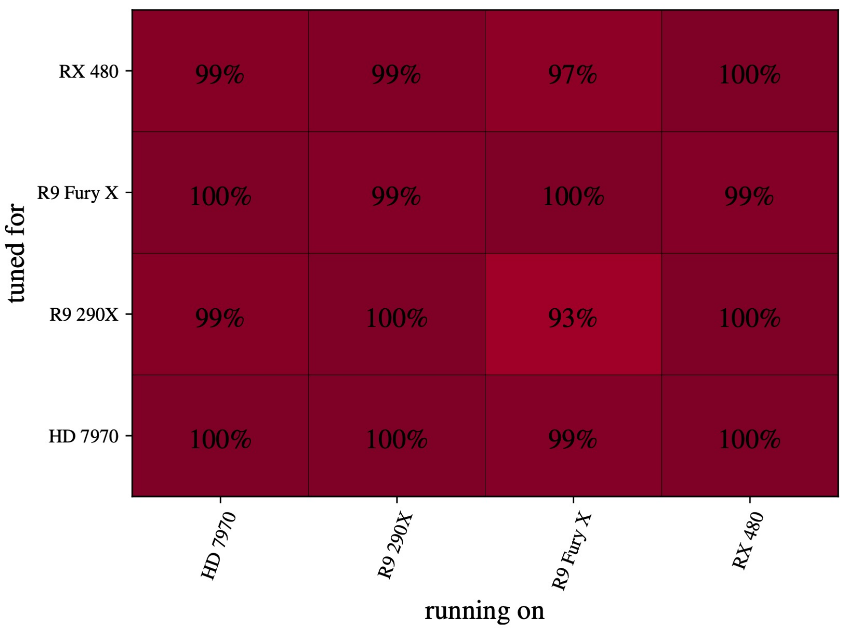
Nvidia



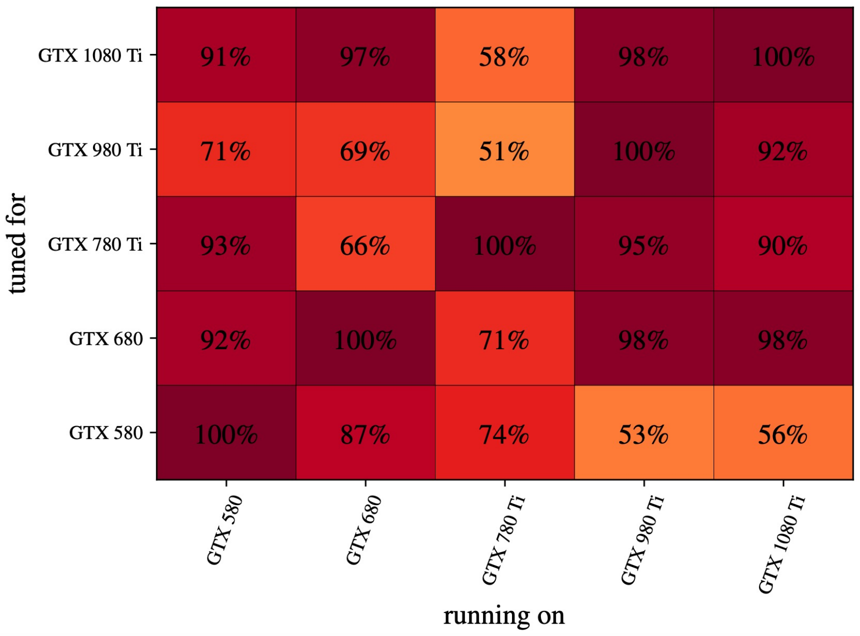
from: “Analyzing and improving performance portability of OpenCL applications via auto-tuning” Price et al. IWOCL 2017

Tuning for same vendor?

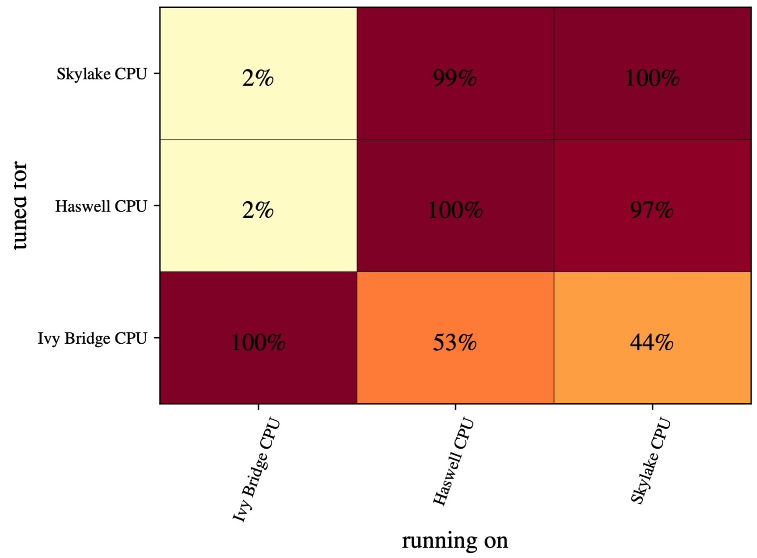
AMD



Nvidia



Intel



from: “Analyzing and improving performance portability of OpenCL applications via auto-tuning” Price et al. IWOCL 2017

Multi-objective tuning

- Example, being portable across architectures:

- $E(p, i, a, o)$ is the execution time of running program p on input i on architecture a with optimization settings o

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

Multi-objective tuning

- Example, being portable across architectures:

- $E(p, i, a, o)$ is the execution time of running program p on input i on architecture a with optimization settings o

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

Multi-objective tuning

- Example, being portable across architectures:

- $E(p, i, a, o)$ is the execution time of running program p on input i on architecture a with optimization settings o

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

$$\frac{E(p, i, a0, o)}{E(p, i, a0, o + c)}$$

$$\frac{E(p, i, a1, o)}{E(p, i, a1, o + c)}$$

$$\frac{E(p, i, a2, o)}{E(p, i, a1, o + c)}$$

Multi-objective tuning

- Example, being portable across architectures:

- $E(p, i, a, o)$ is the execution time of running program p on input i on architecture a with optimization settings o

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

$speedup_0$

$speedup_1$

$speedup_n$

Multi-objective tuning

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

- Define a fitness function F to collapse multiple speedups into a single value:
  - $F(speedup_0, speedup_1, speedup_2)$

$speedup_0$

$speedup_1$

$speedup_n$

Multi-objective tuning

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?

- Define a fitness function F to collapse multiple speedups into a single value:
  - $F(speedup_0, speedup_1, speedup_2)$

- Options?

Multi-objective tuning

- How to evaluate a binary optimization c?
  - i.e. should c be enabled?
- Define a fitness function F to collapse multiple speedups into a single value:
  - $F(speedup_0, speedup_1, speedup_2)$
- Options?
  - average (geomean)
  - max, min?

Multi-objective tuning

- How to evaluate a binary compiler optimization c

- Baseline: runtimes at $E(p, i, a_n, o)$
  - p is a program
  - i is an input
  - a_n is an architecture (we can have many of these)
  - o is an optimization setting. The baseline has c disabled

- Call a baseline runtime for architecture n : B_n

Multi-objective tuning

- How to evaluate a binary compiler optimization c

- Baseline: runtimes at $E(p, i, a_n, o)$
  - p is a program
  - i is an input
  - a_n is an architecture (we can have many of these)
  - o is an optimization setting. The baseline has c disabled

- Call a baseline runtime for architecture n : B_n

- optimization times: evaluate runtimes at $E(p, i, a_n, o + c)$
  - Same programs and baselines, except with c enabled
  - Call these runtimes : C_n

Multi-objective tuning

A speedup for architecture n is $\frac{B_n}{C_n}$, call it S_n

Check:

$$F(S_0, S_1, S_2, \ \dots \ S_n) > 1.0$$

For example: if F is the **average,** then this will measure if the average effect of the optimization caused a speedup or slowdown.

If F is **min,** then this will determine if the worst-off architecture still saw a speedup.

Multi-objective tuning

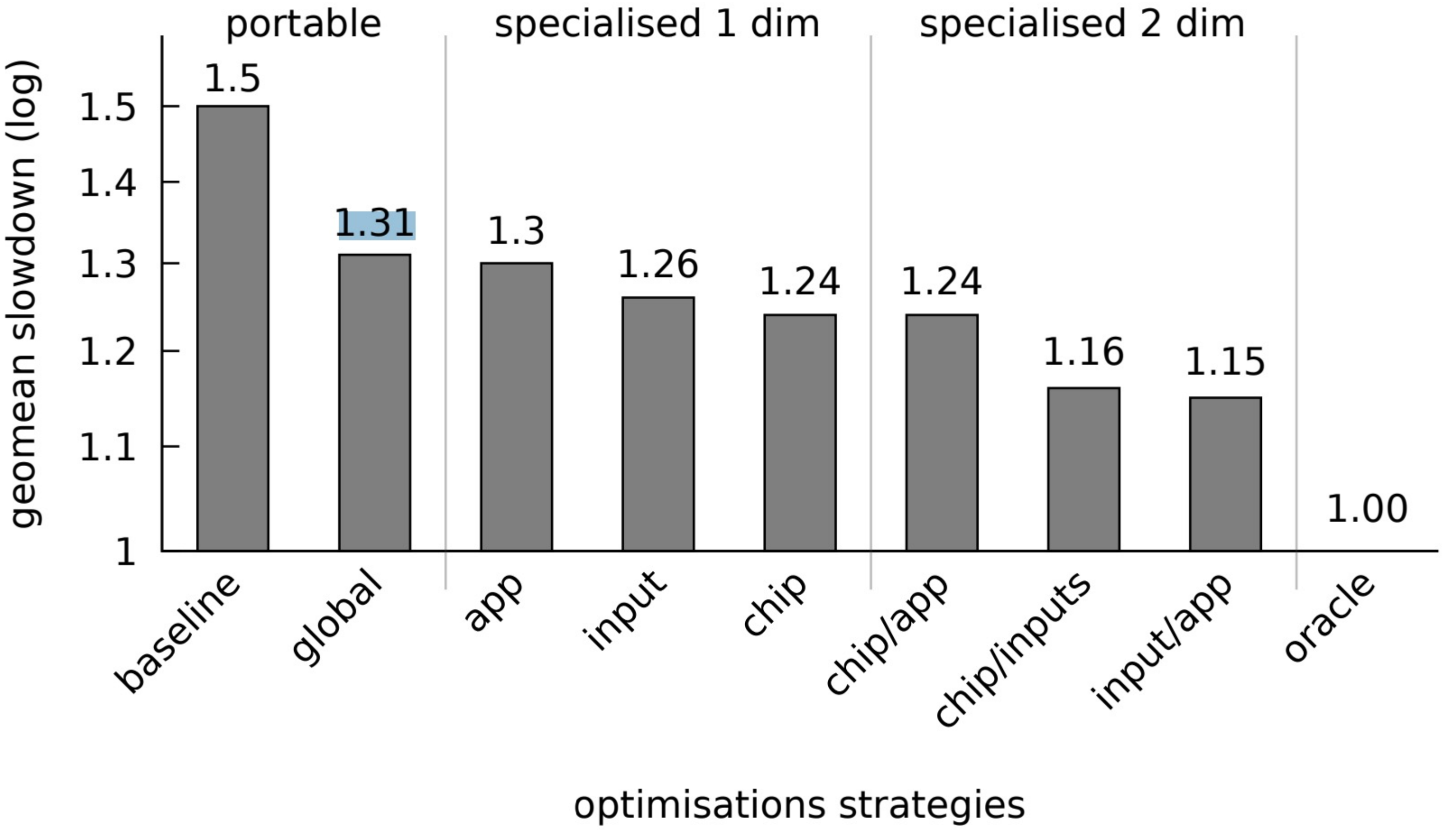
- Options?
  - average (geomean)
  - max, min?

- For 3 applications, architecture portability got within:
- 85%, 70% and 70% of maximum performance

	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

from: “Analyzing and improving performance portability of OpenCL applications via auto-tuning” Price et al. IWOCL 2017

Performance Penalties for Portability



From: "One Size Doesn't Fit All: Quantifying Performance Portability of Graph Applications on GPUs" IISWC 2019.

Wrapping up

- No class on Wednesday
- Friday is an extended class, keep an eye out for sign-up sheets for presenters
- Office hours on Tuesday (2-3 pm) and Thursday (2-3 pm)