

Bias in Context

by

Sophie Beiying Chou

Submitted to the MIT Media Lab,
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of

MS in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author
MIT Media Lab
May 5, 2016

Certified by
Deb Roy
Associate Professor
Thesis Supervisor

Accepted by
WHO IS THE CHAIR(WO)MAN?
Chairman, Department Committee on Graduate Theses

Bias in Context

by

Sophie Beiying Chou

Submitted to the MIT Media Lab
on May 5, 2016, in partial fulfillment of the
requirements for the degree of
MS in Media Arts and Sciences

Abstract

Lorem ipsum dolor sit amet, no nibh deleniti pri, docendi omnesque no cum, sed rationibus consetetur ne. Nam mentitum maluisset te, est eleifend intellegebat ex. Stet volutpat deseruisse pro an, at causae alienum assueverit vel. Vis timeam atomorum cu, solet epicurei temporibus ut ius. Pertinax consetetur sea te. Ne quas harum denique ius. Et sit vocibus sententiae definiebas, ei usu minim abhorreant. Nam cu errem equidem, omnesque offendit ea duo. Duo an dicant definitiones. Tation graece melius cum ut, ea dicta vulputate reprehendunt vix, eu quis fuisset expetendis mea. Has blandit praesent reprehendunt ei. Animal iuvaret has ea, vis quodsi sanctus an. Duo albucius hendrerit definitionem at, vide malorum vel an. No sit debet blandit, mentitum temporibus cu sea. Id vitae aperiam vis, virtute copiosae accusata no ius. Invenire dignissim at cum, an adhuc vivendo principes has. Ut mei mutat voluptua suavitate, aliquid equidem has et. Cum eu erant putant, ne facete timeam euismod sed, usu ei erroribus hendrerit. Est id vero dictas legendos. Et ullum iriure mel, ei eum graeci interpretaris, pro atqui oblique id. Enim mundi liberavisse mel ei, pri et quodsi eleifend. Habeo molestie quo et, mundi primis accumsan eu vim, pro ei impetus prodesset efficiantur.

Thesis Supervisor: Deb Roy
Title: Associate Professor

The following people served as readers for this thesis:

Sepandar Kamvar.....
Associate Professor of Media Arts and Sciences
MIT Media Lab

Iyad Rahwan
Associate Professor of Media Arts and Sciences
MIT Media Lab

Acknowledgments

Thank you famiry– sister Thank you Linda and Keira Thank you Deb Thank you Ethan Thank you readers Thank you LSM Thank you Heather Thank you figure makers Thank you Ramesh

Contents

1	Introduction	11
1.1	Motivations for micro-optimization	12
1.2	Description of micro-optimization	13
1.2.1	Post Multiply Normalization	13
1.2.2	Block Exponent	14
1.3	Integer optimizations	15
1.3.1	Conversion to fixed point	15
1.3.2	Small Constant Multiplications	15
1.4	Other optimizations	17
1.4.1	Low-level parallelism	17
1.4.2	Pipeline optimizations	17
A	Tables	19
B	Figures	21

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

B-1	Armadillo slaying lawyer.	21
B-2	Armadillo eradicating national debt.	22

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

A.1 Armadillos	19
--------------------------	----

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Most Americans say that they want to read news that's unbiased. A survey from Pew Research in 2012 showed that more than two-thirds (68%) of readers want to read political articles with a neutral stance, compared to just a little less than a quarter (23%) of those who want to read those stories that share their point of view.¹ But what exactly does that mean?

To begin with, whether or not we perceive news as biased is biased in itself. Conservative readers tend to view media as more biased than both Democrats and Independents (49% to 32% and 35%, respectively)[?].

The Hostile Media Effect, first studied by Vallone, Ross, and Lepper in 1985, gives one possible explanation for discrepancies: it describes a phenomenon where people with strong stances on an issue tend to perceive media covered as biased against their opinions, even on the same article.²

Clearly, finding bias in news depends on who the reader is as much as what they are reading.

In my thesis, I will examine perceptions of media bias in its social context. In particular, how do social structures surrounding the reader and the writer of the story, along with entities in the story itself, affect the likelihood perceived bias? Although surveys have been performed with respect to bias perception and ideological and demographic data, little has been studied in respect to their social context.

To do so, I will perform a study for a broad range of readers to read and annotate

political news stories (collected daily and sorted using a machine learning classifier) as supportive of a certain candidate. For each reader, I will collect their social media handles, demographic information, and self-reported political stances.

I will then analyze the networks of both the readers and writers of the articles, along with linguistic features of the stories themselves and a constructed graph of related entities mentioned in the news, to find features that are predictive of whether or not a story is perceived as biased.

In this way, I will be able to capture clues that cause a story to be perceived as bias beyond the writing itself. Hypotheses to be tested: Is homophily between the reader’s social network and the writer’s predictive of perceived bias? How do linguistic metrics (ie: computationally analyzed tone, sentiment, stance) of a story compare with social context in being perceived as biased?

1.1 Motivations for micro-optimization

The idea of micro-optimization is motivated by the recent trends in computer architecture towards low-level parallelism and small, pipelineable instruction sets [?, ?]. By getting rid of more complex instructions and concentrating on optimizing frequently used instructions, substantial increases in performance were realized.

Another important motivation was the trend towards placing more of the burden of performance on the compiler. Many of the new architectures depend on an intelligent, optimizing compiler in order to realize anywhere near their peak performance [?, ?, ?]. In these cases, the compiler not only is responsible for faithfully generating native code to match the source language, but also must be aware of instruction latencies, delayed branches, pipeline stages, and a multitude of other factors in order to generate fast code [?].

Taking these ideas one step further, it seems that the floating point operations that are normally single, large instructions can be further broken down into smaller, simpler, faster instructions, with more control in the compiler and less in the hardware. This is the idea behind a micro-optimizing FPU; break the floating point instruc-

tions down into their basic components and use a small, fast implementation, with a large part of the burden of hardware allocation and optimization shifted towards compile-time.

Along with the hardware speedups possible by using a μ FPU, there are also optimizations that the compiler can perform on the code that is generated. In a normal sequence of floating point operations, there are many hidden redundancies that can be eliminated by allowing the compiler to control the floating point operations down to their lowest level. These optimizations are described in detail in section 1.2.

1.2 Description of micro-optimization

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra “guard bits” on the standard floating point formats, to allow several unnormalized operations to be performed in a row without the loss information¹. A discussion of the mathematics behind unnormalized arithmetic is in appendix ??.

The optimizations that the compiler can perform fall into several categories:

1.2.1 Post Multiply Normalization

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit. If there are guard bits on the mantissas to prevent bits from “falling off” the end

¹A description of the floating point format used is shown in figures ?? and ??.

during multiplications, the normalization can be postponed until after a sequence of several multiplies².

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory³.

1.2.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1, e_1) and (m_2, e_2) .

1. Compare e_1 and e_2 .
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2 .
4. Find the first one in the resulting mantissa.
5. Shift the resulting mantissa so that normalized
6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been shifted, the additions can take place one after another⁴. An example of the Block Exponent optimization on the expression $X = A + B + C$ is given in figure ??.

²Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [?] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate NORMALIZE instruction.

³Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

⁴This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μ FPU, there are 3 guard bits, making up to 8 consecutive additions possible.

1.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μ FPU. In concert with the floating point optimizations, these can provide a significant speedup.

1.3.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section ??, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

1.3.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some

combination of

$$a_i = a_j + a_k$$

$$a_i = 2a_j + a_k$$

$$a_i = 4a_j + a_k$$

$$a_i = 8a_j + a_k$$

$$a_i = a_j - a_k$$

$$a_i = a_j \ll m\text{shift}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r , you could use:

$$r = 4s + s$$

$$r = r + r$$

Or by 59:

$$t = 2s + s$$

$$r = 2t + s$$

$$r = 8r + t$$

Similar combinations can be found for almost all of the smaller integers⁵. [?]

⁵This optimization is only an “optimization”, of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism⁶

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μ FPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is “trace scheduling”. This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [?, ?] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential “trace” of program execution. In this way, instructions that *might* be executed depending on a conditional branch further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn’t taken, correction code is inserted after the branches to undo the effects of any prematurely executed instructions.

1.4.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

⁶This can be seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [?]

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as described by Gibbons. [?]

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography