Urdu Text to Speech Synthesizer

By

Muhammad Hassan Siddiqui

MSCSF15M005

Supervised by

Dr. Muhammad Kamran Malik

Assistant Professor, PUCIT

(June, 2018)

Punjab University College of Information Technology,

University of the Punjab, Lahore, Pakistan.

Urdu Text to Speech Synthesizer

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE

DEGREE OF

MASTER OF PHILOSOPHY

IN

COMPUTER SCIENCE

By

Muhammad Hassan Siddiqui

MSCSF15M005

Supervised by

Dr. Muhammad Kamran Malik

Assistant Professor, PUCIT

(June, 2018)

Punjab University College of Information Technology,

University of the Punjab, Lahore, Pakistan.

Evaluation of M. Phil. Thesis

We have evaluated the M. Phil. thesis titled

Urdu Text to Speech Synthesizer

Submitted by Mr. **Muhammad Hassan Siddiqui**, **MSCSF15M005**, session 2015-2018 in partial fulfillment of the M. Phil. degree in Computer Science. We have also assessed the candidate through viva-voice.

We are satisfied with the thesis and performance of the candidate in the examination and are of the opinion that she fulfills the requirements as set in the rules and regulations for the M.Phil. degree in Computer Science at the University of the Punjab.

Thesis Supervisor: Dr. Muhammad Kamran Malik
Assistant Professor
Punjab University College of Information Technology
University of the Punjab, Lahore

External Examiner: Dr. NAME
Assistant Professor
Department of Computer Science
COMSATS Institute of Information Technology, Lahore

Principal of the College: Dr. Syed Mansoor Sarwar Principal,
Punjab University College of Information Technology
University of the Punjab, Lahore

UNIVERSITY OF THE PUNJAB

Author: **Muhammad Hassan Siddiqui**

Title: **Urdu Text to Speech Synthesizer**

Department: **Punjab University College of Information Technology**

Degree: **M. Phil. (Computer Science)**

Permission is herewith granted to University of the Punjab to circulate and to have copied for non-commercial purposes, at its discretion, the above title, upon the request of individuals or institutions.

Signature of the Author

Dedicated to

Abstract

Computational modeling is branch of computer science which deals with multiple disciplines. It assists other domains in understanding complex systems and phenomena by providing theory, tools and technology to model and simulate related systems and phenomena. In complex systems, behavior of an individual can have butterfly effect and can become root cause of an emergent phenomenon. Interaction of drivers with each other and surrounding environment forms the dynamics of traffic flow. Hence global effects of a traffic flow depend upon behavior of a single driver. In this research.

Keywords: computational modeling, simulation, urban engineering, traffic management, social moral norms, law abidance, butterfly effect.

Acknowledgements

Computational modeling is branch of computer science which deals with multiple disciplines. It assists other domains in understanding complex systems and phenomena by providing theory, tools and technology to model and simulate related systems and phenomena. In complex systems, behavior of an individual can have butterfly effect and can become root cause of an emergent phenomenon. Interaction of drivers with each other and surrounding environment forms the dynamics of traffic flow. Hence global effects of a traffic flow depend upon behavior of a single driver. In this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

(Questions to be explored) What is minimal invasive surgery? What are the complications faced by doctors and patients? Solution to complications?

In minimal invasive surgery (MIS), surgical instruments are inserted in human body through a natural orifices/small cut(s) with the help of endo vision system displayed on screen. The surgeons perform surgery by keeping eye on the screen.

For the young trainee doctors, they need some experience for any surgical procedure before performing it. To overcome this problem, a virtual reality based surgical simulator provides a platform to train doctors for ordinary as well as complex surgeries. These simulators are being manufactured in modern world by multinational companies which are very costly. So, for this purpose, we are taking initiative at our department to help by saving foreign exchange and to promote research in Pakistan.

Micro-optimization is a technique to reduce the overall operation count of floating point operations. In a standard floating point unit, floating point operations are fairly high level, such as "multiply" and "add"; in a micro floating point unit (μFPU), these have been broken down into their constituent low-level floating point operations on the mantissas and exponents of the floating point numbers.

Chapter two describes the architecture of the μFPU unit, and the motivations for the design decisions made.

Chapter three describes the design of the compiler, as well as how the optimizations discussed in section 1.2 were implemented.

Chapter four describes the purpose of test code that was compiled, and which statistics

were gathered by running it through the simulator. The purpose is to measure what effect the micro-optimizations had, compared to unoptimized code. Possible future expansions to the project are also discussed.

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 [**patterson:risc**, **rad83**]. 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 [**ellis:bulldog**, **pet87**, **coutant:precision-compilers**]. 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 [**gib86**].

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 instructions 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 μFPU, 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 information[1]. 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 during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

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 memory[3].

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

[2] Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**thornton:cdc6600**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

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

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 another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

1.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. 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 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

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

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

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \\ a_i &= 4a_j + a_k \\ a_i &= 8a_j + a_k \\ a_i &= a_j - a_k \\ a_i &= a_j \ll m\text{shift} \end{aligned}$$

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

$$\begin{aligned} r &= 4s + s \\ r &= r + r \end{aligned}$$

Or by 59:

$$\begin{aligned} t &= 2s + s \\ r &= 2t + s \\ r &= 8r + t \end{aligned}$$

Similar combinations can be found for almost all of the smaller integers[5]. [**magenheimer:precision**]

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, 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. [**ellis:bulldog**, **colwell:vliw**] 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.

[5]This 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.

[6]This can been 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. [**byte:i860**]

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.

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 seendescribed by Gibbons. [**gib86**]

Chapter 2

Related Work

Let's cite! The Einstein's journal paper [**westwood1998validation**] and the Dirac's book [**dirac**] are physics related items. [**ryan2001narrative**] virtual

Appendix A

Figures

Figure A-1: Armadillo slaying lawyer.

Figure A-2: Armadillo eradicating national debt.

Appendix B

Tables

Table B.1: Armadillos

Armadillos	are
our	friends