Achieving Deterministic Execution in Singlethreaded Linux Services

by

Syed Aunn Hasan Raza

S.B., Computer Science and Engineering, M.I.T., May 2010

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author . . . .
Department of Electrical Engineering and Computer Science
June 21, 2011

Certified by . . . .
Dr. Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Achieving Deterministic Execution in Singlethreaded Linux Services

by

Syed Aunn Hasan Raza

Submitted to the Department of Electrical Engineering and Computer Science
on June 21, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Recent hardware virtualization schemes have not provided the same performance benefits to managed code applications as they have to most other applications.

This project focuses on optimizing the Java Virtual Machine (JVM) for high performance; specifically, it targets the performance of Hadoop benchmarks running on Windows Azure clusters.

We hoped to: identify areas of the JVM that cause excessive traps to the hypervisor and instead use instructions that do not need emulation; and, to optimize the garbage collector and memory manager around the unique TLB constraints that multiple VMs running concurrently on a hardware virtualization present.

Using a combination of simple optimizations tailored for virtual execution environments, we were able to improve the performance of the JVM by as much as 30 percent on some benchmarks and by 500 percent on some micr-benchmarks.

Thesis Supervisor: Dr. Saman P. Amarasinghe
Title: Professor

Acknowledgments

I would like to thank Professor Saman Amarasinghe for his huge role in both this project and my wonderful undergraduate experience at MIT. Saman was my professor for 6.005 (Spring 2008), 6.197/6.172 (Fall 2009) and 6.035 (Spring 2009). These three exciting semesters not only convinced me of his unparalleled genius, but also ignited my interest in computer systems. Over the past year, as I have experienced the highs and lows of research, I would have probably given up except for Saman's infinite insight, encouragement and patience.

As an M-Eng student, I have been blessed to work with two truly gifted and inspirational people from the COMMIT group: Marek Olszewski and Qin Zhao. Marek and Qin's expertise and brilliance is probably only eclipsed by their humility and helpfulness. Throughout this challenging year, I have learned more from them than I probably realize, and their knowledge of Operating Systems and Dynamic Instrumentation is invaluable and, frankly, immensely intimidating. I hope to emulate (or even approximate) their excellence some day.

Over the course of this past year, I have also had the opportunity to work with Professor Srini Devadas, Professor Fraans Kaashoek, and Professor Dina Katabi as a Teaching Assistant for 6.005 and 6.033. It has been an extraordinarily rewarding experience, and I have learned tremendously from simply interacting with these peerless individuals. Professor Dina Katabi was especially kind to me for letting me work in the G916 office for the past two months.

I would like to thank Abdul Munir, whom I have known since my first day at MIT; I simply don't deserve the unflinchingly loyal and supportive friend I have found in him. I am also indebted to Osama Badar, Usman Masood, Brian Joseph, Adnan Esmail and Nabeel Ahmed for their unrelenting support and encouragement; this past year would have been especially boring without the never-ending arguments and unproductive 'all-nighters' that accompany our friendship. I also owe a debt of gratitude to my partners-in-crime Prannay Budhraja, Daniel Firestone, Ankit Gordhandas and Maciej Pacula, who have been great friends and collaborators over the past few years.

My father has been the single biggest inspiration and support in my life since childhood, and I owe any and all of my success to him. He epitomizes, for me, the meaning of selflessness and resilience in life. This thesis, my work and few achievements were enabled by – and dedicated to – him, my mother and my two siblings. I am humbled by the countless sacrifices made by my family in order for me to be where I am today.

Finally, I would like to thank Amina for her unwavering love and support throughout my stay at MIT, for knowing me better than even I know myself, and for improving and enriching my life every single day since I have known her. Through her, I have also met two exceptional individuals, Dr. Fatima and Anwer Basha, whom I have already learnt a lot from.

My inspiration over the past few months:

"It is impossible to live without failing at something, unless you live so cautiously that you might as well not have lived at all – in which case, you fail by default."
J.K. Rowling, Harvard Commencement Speech 2008

"Perseverance Commands Success"
Words etched on my school uniform 1994-2006, Aitchison College Lahore motto.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Take a single program, and execute it several times. Does the program always execute the same sequence of instructions? If not, what can cause multiple executions of the same program be different? Do the results change if we run the same program in separate *e.g.* two identical but distinct virtual machines, or two slightly different virtual machines?

One can immediately find a few obvious reasons for multiple executions of the same program to behave differently: input or data can vary; many programs deliberately use randomization schemes; instructions in multithreaded programs can interleave in myriad ways, and so on. Surprisingly, however, even a simple "Hello, world!" application written in C does not execute the same sequence of instructions.

To the best of our knowledge, the sources of nondeterminism in modern applications have not been studied in great detail or documented in a single place. This thesis synthesizes sources of nondeterminism – both obvious and obscure – for a select group of programs, and describes the design and implementation of a tool that makes program execution as deterministic as possible. Despite its applications, such a study is novel and interesting in its own right, for it reveals subtle interactions between applications, underlying operating systems and intermediate software layers.

1.2 Importance of Deterministic Execution

Deterministic execution of programs can be beneficial in many different scenarios, and has motivated the design of Kendo, a system which enables deterministic multithreading in applications[]. Our tool complements Kendo because it focuses on deterministic execution of single-threaded services in Linux, at the granularity of individual instructions and their side-effects in memory.

The motivations for deterministic multhreading listed in [] apply to this thesis as well.

1.2.1 Mainstream Computing and Security

1.2.2 Repeatability

Users expect programs to produce the same outputs, given the same inputs. For safety-critical systems, it may be even better if we can guarantee that given the same inputs, a program would always execute some *precise* set of instructions, in the same order, with the same side-effects. Record/replay systems are not suitable for achieving such strong guarantees of repeatability, given the storage overhead for logs and the typically enormous input space. Our system, like Kendo, requires minimal storage to achieve single-threaded determinism.

1.2.3 Debugging

Determinism is important for debugging, because developers often need to reproduce erroneous behavior in order to diagnose and fix it. Nondeterminism is typically a major issue for debugging multithreaded applications, and is a lesser issue for single-threaded applications. Our work will help developers easily reproduce erroneous behavior given the same input for single-threaded applications. Record/replay systems have high overhead, so it is unlikely that the initial buggy execution of a program was recorded. Deterministic execution also precludes the need for storing many gigabytes of logs needed for record/replay.

1.2.4 Testing

Deterministic execution in general facilitates testing, because outputs and internal state can be checked at certain points with respect to expected values. Our version of determinism allows for a particularly strong kind of test case that may be necessary for safety-critical systems: with deterministic execution, a program must execute the exact same instructions across different executions, for the same inputs. Test cases can check for deviations from the expected instruction sequences.

1.3 The VM *Bootstorm* Problem

1.4 Contributions

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.

1.4.1 Organization

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 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.

1.4.2 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:

$$t = 2s + s$$

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

Similar combinations can be found for almost all of the smaller integers[1]. [?]

1.5 Other optimizations

1.5.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[2]

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. [?, ?] 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

[1] 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.

[2] 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. [?]

executed instructions.

1.5.2 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. [**?**]

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.