

THEORY AND APPLICATIONS OF NON-BLOCKING SLOT SCHEDULERS

POOJA AGGARWAL



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
JUNE 2016

THEORY AND APPLICATIONS OF NON-BLOCKING SLOT SCHEDULERS

by

POOJA AGGARWAL

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of

Doctor of Philosophy

to the



Indian Institute of Technology Delhi

JUNE 2016

Certificate

This is to certify that the thesis titled **THEORY AND APPLICATIONS OF NON-BLOCKING SLOT SCHEDULERS** being submitted by **Ms. Pooja Aggarwal** for the award of **Doctor of Philosophy** in **Computer Science and Engineering** is a record of bona fide work carried out by her under my guidance and supervision at the **Department of Computer Science and Engineering, Indian Institute of Technology Delhi**. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Smruti Ranjan Sarangi

Assistant Professor

Department of Computer Science and Engineering

Indian Institute of Technology Delhi

New Delhi- 110016

Acknowledgements

I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost, I would like to thank my advisor Prof. Smruti Ranjan Sarangi for giving me an opportunity to work with him and for guiding me through out my research work as well as courses. The blessings, help and guidance given by him shall carry me a long way in the journey of life, on which I am about to embark. He has supported, inspired, and encouraged me in many different ways through different stages. I feel so fortunate to have been one of his students.

My sincere thanks to Prof. Kolin Paul, Prof. Sanjeeva Parsad and Prof. S. Rao for serving on my Student Research Committee and for the suggestions which incited me to widen my research from various perspectives. I would like to thank NetApp India for their support and collaboration.

I also take this opportunity to thank all my fellow mates of Srishti group, for their effort to make the department such an excellent environment to work. I would like to specially thank Seep, Abhishek, Prathmesh, Sandeep, Rajshekhar and Kapil for all the help, discussions and great company.

I would like to thank the staff of the Department of Computer Science for their administrative and technical help. I would thank Vandana Ahluwalia for her timely assistance in lab related issues.

Deepak Ravi, thanks for all the brainstorming sessions and constructive criticisms at different stages of my research which were thought-provoking and they helped me focus my ideas. My roommate, Anuradha Sharma, thank you very much for making the atmosphere of our room as friendly as possible and for motivating me through out my duration at IIT Delhi.

Most importantly, my thanks go to my family. I wish to thank my parents and my brother and

sister for their endless and unconditional love, for their sustained support and encouragement. I am very grateful to Prateek, my husband, for his love, support and care. I would for certain not be here without your support.

Pooja Aggarwal

Abstract

In this thesis, we consider the design and applications of parallel non-blocking slot scheduling algorithms. Slot schedulers divide time into discrete quanta called *slots*, and schedule resources at the granularity of slots. They are typically used in high throughput I/O systems, data centers, video servers, and network drivers. In this thesis, we first present an efficient implementation of a concurrent non-blocking slot scheduler, and then we apply our slot scheduler to solve scheduling and resource allocation problems. Typically, a resource in a concurrent system may be accessed by only one thread at a time. To share a single resource, threads co-ordinate using mutual exclusion algorithms. When threads require simultaneous access to multiple resources, the task of coordinating them properly is the resource allocation problem. However, these mutual exclusion algorithms (using locks) suffer from inherent problems such as deadlocks, starvation, priority inversion, which makes the lock-based implementation untenable.

We propose a family of parallel lock-free and wait-free linearizable algorithms to map the resource demands to a set of time slots. This mapping of resources to currently available time slots introduces many problems. In specific, we look at problems for reserving, as well as freeing a set of contiguous slots.

We show that in a system with 64 threads, it is possible to get speedups of 10X by using lock-free algorithms as compared to a baseline implementation that uses locks. We additionally propose wait-free algorithms, whose mean performance is roughly the same as the version with locks. However, they suffer from significantly lower jitter and ensure a high degree of fairness among threads. We next propose a class of algorithms using software transactional memory (STM) that are in the middle of the spectrum. They equitably balance fairness and throughput, and are much simpler to design and verify.

Subsequently, we propose applications of our slot scheduler. First, we applied our slot scheduler in developing a parallel architectural simulator to schedule concurrent accesses of shared memory structures. Prior approaches for this problem either used locks, or used methods that introduced a lot of simulator error. Our method provides impressive speedups and minimizes

simulation error.

With the advent of solid state drives (SSDs), storage vendors are increasingly preferring them because their I/O throughput can scale up to a million IOPS (I/O operations per second). These disruptive changes in the world of storage systems and software has made the scheduler a bottleneck. Supporting these devices is well beyond the means of conventional scheduling techniques, and thus a more scalable solution is required. To genuinely satisfy our requirements of having a high throughput scheduler, we extend our non-blocking slot scheduling algorithms for I/O intensive workloads and make them aware of dependencies, deadlines, request lengths, priorities, and evaluate their feasibility for SSDs with different RAID configurations.

We next propose a data structure inspired by our techniques to help us implement lock-free and wait-free algorithms. To implement non-blocking operations in a linearizable and non-blocking manner, requires us to associate additional information with memory words in a data structure. Current approaches either use some bits in a memory word to pack additional information (*packing*), or use the bits to store a pointer to an object that contains additional information (*redirection*), and the original data item. The former approach restricts the number of bits for each additional field and this reduces the range of the field, and the latter approach is wasteful in terms of space. We propose a novel method called a memory word expander. It caches information for a set of memory locations that need to be augmented with additional information. It supports traditional get, set, and CAS operations, and tries to maintain state for a minimum number of entries.

Contents

Certificate	i
Acknowledgements	iii
Abstract	v
1 Introduction	1
1.1 Contribution	5
2 Background	7
2.1 Atomic Primitives	8
2.2 Synchronization and Progress Guarantees	9
2.3 Linearizability	10
3 Lock-free and Wait-free Slot Scheduling Algorithms	13
3.1 Introduction	13
3.2 Related Work	14
3.2.1 Slot Scheduling	14
3.2.2 Non-blocking Algorithms	15
3.3 Overview of Slot Scheduling	16

3.3.1	Definition of the Problem	16
3.3.2	Basic Approach	16
3.4	Slot Scheduling Algorithm	20
3.4.1	Data Structures	20
3.4.2	Entry Points	22
3.4.3	<i>schedule</i> Operation: In a nutshell	24
3.4.4	The $N \times M$ schedule Problem	26
3.5	The $1 \times M$ schedule Problem	34
3.5.1	Reserve Slots	36
3.6	Auxiliary Functions	38
3.6.1	Functions to Book Slots	39
3.6.2	Functions to Co-ordinate Among Helpers	41
3.6.3	Functions to Reset the State	42
3.6.4	The <i>otherCancel</i> Function	44
3.7	The <i>free</i> Operation	45
3.8	Slot Scheduling – Locked version	48
3.8.1	Coarse Grain Locking	48
3.8.2	Slot Scheduling with Fair Locking	50
3.8.3	Slot Scheduling with Fine Grain Locking	50
3.9	ABA Issues, Sizing of Fields, Recycling	51
3.10	Proofs	52
3.10.1	Legal Sequential Specification	52
3.11	Verification of Linearizability	60

3.11.1	Basic Correctness Conditions	60
3.11.2	Verification Process	63
3.11.3	Verification Results	65
3.12	Evaluation	66
3.12.1	Setup	66
3.12.2	Performance of the $N \times M$ and $1 \times M$ Algorithms	67
3.12.3	Throughput	69
3.12.4	Sensitivity	70
3.12.5	Results: Characterization of Locking	72
3.12.6	Results for Slot Scheduling	73
3.13	Relaxed Slot Scheduling	74
3.13.1	Results	75
3.14	Conclusion	77
4	Software Transactional Memory Friendly Slot Schedulers	79
4.1	Introduction	79
4.2	Background	80
4.2.1	Transactional Memory	80
4.3	Parallel Slot Scheduling using STMs	81
4.3.1	Data Structures	81
4.3.2	The <i>SimpleScan</i> Algorithm	82
4.3.3	The <i>SoftVisible</i> Algorithm	83
4.3.4	The <i>SoftVisibleMerge</i> Algorithm	86
4.4	Proofs	87

4.5	Evaluation	90
4.5.1	Experimental Setup	90
4.5.2	Performance of the Algorithm	90
4.6	Conclusion	92
5	ParTejas: A Parallel Simulator for Multicore Processors	93
5.1	Introduction	93
5.2	Related work	94
5.3	System Architecture	96
5.3.1	Parallel Ports	97
5.4	Evaluation	99
5.4.1	Experimental Setup	99
5.4.2	Performance Results	100
5.5	Conclusion	103
6	RADIR: Non Blocking Bandwidth Allocation Models for Solid State Drives	105
6.1	Introduction	105
6.2	Related Work	109
6.3	Bandwidth Allocation Model for I/O Scheduling	109
6.3.1	Request Parameters	109
6.3.2	Redundant Disk Arrays	111
6.3.3	Dummy Slots	112
6.3.4	Load Balancing	112
6.4	Design And Implementation	113

6.4.1	Data Structures	113
6.4.2	Reserve Operation	115
6.4.3	Request Priority Rule	119
6.4.4	Disk Redundancy	122
6.5	Proof	122
6.6	Performance Evaluation	123
6.6.1	Storage Device Model	124
6.6.2	Performance of RADIR	124
6.6.3	Impact of Heuristics	126
6.7	Conclusion	129
7	Expander: Lock-free Cache for a Concurrent Data Structure	131
7.1	Introduction	131
7.2	Related Work	132
7.3	Background	133
7.4	Overview	134
7.4.1	Data Structures	135
7.5	Algorithm	138
7.5.1	lookUp()	138
7.5.2	lookUpAlloc()	139
7.5.3	kGet()	141
7.5.4	kSet()	142
7.5.5	kCAS()	143
7.5.6	kRM()	144

7.6	Correctness	147
7.7	Usage with Wait-Free Algorithms	153
7.8	Evaluation	154
7.8.1	Setup	154
7.8.2	Benchmark details	155
7.8.3	Performance	158
7.8.4	Fairness	160
7.8.5	Sensitivity	161
7.9	Implementation Details for Wait-free Queues	163
7.10	Conclusion	168
8	Conclusion and Future Work	169
	Bibliography	171
	List of Publications	181
	Biography	183

List of Figures

1.1	The Ousterhout matrix for scheduling	2
3.1	Finite state machine of a <i>schedule</i> request	18
3.2	Finite state machine of a <i>free</i> request	19
3.3	Finite state machine of a slot	20
3.4	The SLOT matrix	21
3.5	The REQUEST array	21
3.6	The PATH and SHADOWPATH arrays	22
3.7	The <i>processSchedule</i> function	26
3.8	The <i>getSlotStatus</i> function	30
3.9	Various steps involved in reserving a slot (temporarily)	32
3.10	An example containing three requests	53
3.11	t_{req} for the $N \times M$ algorithm	67
3.12	Fairness (<i>frn</i>) for the $N \times M$ algorithm	67
3.13	Fairness (<i>frn</i>) for the $N \times M$ algorithm across threads	68
3.14	t_{req} for the $1 \times M$ algorithm	69
3.15	Fairness (<i>frn</i>) for the $1 \times M$ algorithm	69
3.16	Request throughput for the $N \times M$ algorithm	70

3.17 t_{req} across different capacities (N)	70
3.18 t_{req} for WF across different values of REQUESTTHRESHOLD	71
3.19 Fairness (frn) for WF across different values of REQUESTTHRESHOLD	71
3.20 t_{req} of WF for the $N \times M$ algorithm with varied CAS latencies	71
3.21 Baseline jitter	73
3.22 t_{req}	73
3.23 t_{req} for the $1 \times M$ algorithm	74
3.24 t_{req} for the $N \times M$ algorithm	74
3.25 t_{req} for relaxed version of WF	76
3.26 t_{req} for relaxed version of LF	76
3.27 $delay$ per request for WF	76
3.28 $delay$ per request for LF	76
4.1 Example of an atomic method	81
4.2 High level architecture of DEUCE	81
4.3 The REQUEST class	82
4.4 request.state and slot.state	82
4.5 Flowchart of the <i>SoftVisible</i> and <i>SoftVisibleMerge</i> algorithms	87
4.6 Time	91
4.7 $fairness$	91
4.8 Ratio of aborts to commits	91
4.9 Fairness (STM based algorithms only)	91
4.10 Time (with jitter)	91
4.11 Fairness (with jitter)	91

5.1	Overview of parTejas	96
5.2	Operation of a parallel port	98
5.3	Performance results (speedups with respect to 1 core)	101
5.4	Relative error of PS_{nosync} vs <i>parallelport</i>	101
6.1	I/O throughput of solid state drives	106
6.2	The Resource allocation model	108
6.3	Request dependence graph	111
6.4	Various request types	112
6.5	Slot state	113
6.6	The Request class	114
6.7	Request to drive slot mapping	120
6.8	Time/request	125
6.9	Fairness	125
6.10	Request throughput	126
6.11	Request scheduling delay	126
6.12	<i>time</i> with varying request size	127
6.13	<i>delay</i> with varying request size	127
6.14	Impact of I/O intensity on <i>delay</i>	128
6.15	<i>fairness</i> with varying dependency length	128
6.16	<i>time</i> with redundant disks	129
7.1	High level design of the EXPANDER	135
7.2	Types and Structures	136

7.3	FSM of a node in the <code>EXPANDER</code>	137
7.4	Slowdown in the time per operation (t_{req}) with the <code>EXPANDER</code>	159
7.5	Reduction in memory usage for benchmarks (using redirection)	159
7.6	Fairness	160
7.7	Space overhead for different schemes – <i>WFRadirExpander</i>	161
7.8	t_{req} for <i>WFRadir-Expander</i>	161
7.9	t_{req} for <i>WFRadirExpander</i>	162
7.10	Impact of deletions on t_{req} for <i>WFMCaseExpander</i>	162
7.11	Additional memory usage in <i>MCASExpFull</i>	162
7.12	Impact of hashtable size on t_{req} for <i>WFMCaseExpander</i>	162

List of Tables

3.1	Number of requests helped by a single request	68
6.1	SSD characteristics (Seagate 600 Pro SSD)	124
7.1	Packing and redirection in concurrent data structures	133
7.2	List of benchmarks (redirection)	155
7.3	List of benchmarks (packing)	155