

Contents

1	Introduction	1
1.1	Context and motivation	1
1.2	Contributions	2
1.3	Organization of the manuscript	4
<hr/>		
	<i>Part I – Background</i>	7
2	Context: Multi-core Architectures	9
2.1	Shared memory	10
2.2	Concurrent programming	10
3	Transactional Memory	13
3.1	Basics	13
3.2	Software transactional memory	14
3.3	Hardware transactional memory	15
3.3.1	HTM limitations	15
3.3.2	Case study: Intel TSX	16
3.4	Summary	17
4	Automatic Memory Management	19
4.1	Garbage collection in Java	19
4.1.1	Basics	20
4.1.2	ParallelOld GC	22
4.1.3	ConcurrentMarkSweep GC	23
4.1.4	GarbageFirst GC	24
4.2	C++ smart pointers	25
4.2.1	Types of smart pointers	25
4.2.2	Case study: <code>std::shared_ptr</code>	26
4.3	Summary	28

<i>Part II – Exploiting HTM for Concurrent Java GC</i>	29
5 Garbage Collection: Related Work	31
5.1 Real-time garbage collectors	32
5.2 TM-based garbage collectors	33
5.3 Summary	33
6 Performance Study of Java GCs	35
6.1 Experimental setup	37
6.1.1 Infrastructure details	37
6.1.2 DaCapo benchmark suite	37
6.1.3 Apache Cassandra database server	38
6.1.4 Workload generator: YCSB client	39
6.2 Experiments on benchmark applications	39
6.2.1 GC pause time	39
6.2.2 TLAB influence	43
6.2.3 GC ranking	44
6.3 Experiments on client-server applications	45
6.3.1 GC impact on the server side	46
6.3.2 GC impact on the client side	47
6.4 Summary	48
7 Transactional GC Algorithm	51
7.1 Introduction	52
7.1.1 Motivation	52
7.1.2 Design choices	53
7.2 Our approach	53
7.2.1 Brooks forwarding pointer	53
7.2.2 Access barriers	54
7.2.3 Algorithm design	54
7.2.4 Discussion	56
7.3 Summary	57
8 Initial Implementation	59
8.1 Implementation details	60
8.1.1 Java object's header	60
8.1.2 Busy-bit implementation	61
8.1.3 Access barriers in the Java interpreter	62
8.1.4 Access barriers in the JIT compiler	64

8.2 Early results	66
8.3 Summary	68
9 Optimized Implementation	69
9.1 Volatile-only read barriers	69
9.1.1 Preliminary observations	70
9.1.2 Optimization details	71
9.1.3 Optimization validation	71
9.2 Selective transactional barriers	72
9.2.1 Algorithm	72
9.2.2 HotSpot JVM internals	73
9.2.3 Implementation details	75
9.3 Summary	75
10 Evaluation	77
10.1 Experimental setup	77
10.2 Experimental results	78
10.2.1 Transactional barriers overhead in the interpreter	78
10.2.2 Transactional barriers overhead in the JIT	79
10.3 Selective barriers evaluation	81
10.3.1 Transaction duration	81
10.3.2 Benchmark suite	81
10.3.3 Client-server scenario	83
10.4 Discussion: removing GC pauses	85
10.5 Summary	87
<i>Part III – Exploiting HTM for Improved C++ Smart Pointers</i>	89
11 Transactional Smart Pointers	91
11.1 Motivation	91
11.2 Related work	92
11.3 Algorithm	93
11.4 Implementation details	95
11.5 Use cases	97
11.5.1 Baseline: single-threaded micro-benchmark	97
11.5.2 Short-lived pointers micro-benchmark	98
11.6 Summary	99

12 Evaluation on Micro-benchmarks	101
12.1 Experimental setup	101
12.2 Results for single-threaded experiments	102
12.3 Results for short-lived pointers experiments	103
12.4 Summary	105
13 Transactional Pointers for Concurrent Data Structures Traversal	107
13.1 Motivation	108
13.2 Concurrent queue	108
13.2.1 Algorithm overview	109
13.2.2 Implementation with smart pointers	109
13.2.3 Implementation with transactional pointers	110
13.3 Concurrent linked list	111
13.3.1 Algorithm overview	112
13.3.2 Implementation details	112
13.4 Modifications to the initial <code>tx_ptr</code> implementation	113
13.5 Summary	115
14 Evaluation on Concurrent Data Structures	117
14.1 Experimental setup	118
14.2 Read-write workloads	119
14.2.1 Concurrent queue evaluation	119
14.2.2 Concurrent linked list evaluation	120
14.3 Read-only workload	121
14.4 Discussion: abort rate	123
14.5 Summary	124
<hr/>	
<i>Part IV – Conclusions and Future Work</i>	125
15 Conclusions	127
16 Future Work	131
16.1 HTM-based pauseless GC	131
16.2 HTM-based reference-counting	132
Bibliography	135