Achieving High Performance and High Productivity in Next Generation Parallel Programming Languages

Vivek Kumar

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The Australian National University

May 2015

Except where otherwise indicated, this thesis is my own original work.

Vivek Kumar
5 May 2015

न हि ज्ञानेन सदृशं पवित्रमिह विद्यते ।

— Bhagavad Gita, Chapter 4, Verse 38

There is nothing as pure as the knowledge.

to Neha

Acknowledgments

आचार्‌यात्‌ पादमादते पादं शष्‌यः स्‌वमेधया।
पादं सब्‌रह्‌मचारभ्‌यः पादं कालक्‌रमेण च॥

— *The Vedas*

A student learns a quarter from teacher, a quarter from own intelligence, a quarter from fellow students, and the rest in course of time.

A decade ago, when I completed my Bachelor degree in Mechanical Engineering, the dream of getting a doctoral degree in Computer Science seemed quite vague. However, with the blessings of the almighty, love from my near and dear ones, and support from the kind people around me, it has become possible to write this doctoral thesis.

First and foremost I offer my sincerest gratitude to my chair supervisor, Prof. Steve Blackburn. This thesis would not have been possible without his immense support, knowledge, guidance and patience. Thanks Steve for helping me happily sail through this academic journey.

I would also like to thank Dr. Dave Grove and Dr. Daniel Frampton for helping me throughout my doctoral study. Their informative discussions, feedback and cooperative nature has always motivated and guided me.

I would also like to thank the Australian Research Council and Australian National University for helping me financially throughout the duration of my studies and stay in Australia.

Special thanks to Dr. Alistair Rendell for helping me in grabbing the golden opportunity of pursuing a doctorate degree at ANU. He was always very kind in replying to my naive emails while I was in India and looking for such opportunities. And he is still very helpful!

Thanks to Prof. Antony Hosking for allowing me to use his machine to run my experiments. I am also thankful to Dr. Julian Dolby for helping me in formulating one of the contributions in this thesis.

I would also like to express my gratitude to all my fellow students for rendering their friendliness and support, which made my stay in Australia very pleasant. Thank you Xi Yang and Rifat Shahriyar for the helpful discussions. Thank you Josh

Milthorpe and James Bornholt for helping me with all my queries, including the ones on graph plotting and LaTeX.

My thesis would be incomplete without acknowledging the love and support of my family members. Especially, my father, Jagdish Prasad; my mother, Vedwati Gupta; my wife, Neha Shaw, and my brother, Abhisek Kumar. Last but not the least, thanks Neha for giving me the most beautiful gift of my life—my gorgeous daughter Saanvi. Saanvi, your arrival has made my PhD journey blissful !

Abstract

Processor design has turned toward parallelism and heterogeneity cores to achieve performance and energy efficiency. Developers find high-level languages attractive because they use abstraction to offer *productivity* and *portability* over hardware complexities. To achieve *performance*, some modern implementations of high-level languages use work-stealing scheduling for load balancing of dynamically created tasks. Work-stealing is a promising approach for effectively exploiting software parallelism on parallel hardware. A programmer who uses work-stealing explicitly identifies potential parallelism and the runtime then schedules work, keeping otherwise idle hardware busy while relieving overloaded hardware of its burden.

However, work-stealing comes with substantial overheads. These overheads arise as a necessary side effect of the implementation and hamper parallel performance. In addition to runtime-imposed overheads, there is a substantial cognitive load associated with ensuring that parallel code is data-race free. This dissertation explores the overheads associated with achieving high performance parallelism in modern high-level languages.

My thesis is that, by exploiting existing underlying mechanisms of managed runtimes; and by extending existing language design, high-level languages will be able to deliver productivity and parallel performance at the levels necessary for widespread uptake.

The key contributions of my thesis are: 1) a detailed analysis of the key sources of overhead associated with a work-stealing runtime, namely *sequential* and *dynamic* overheads; 2) novel techniques to reduce these overheads that use rich features of managed runtimes such as the yieldpoint mechanism, on-stack replacement, dynamic code-patching, exception handling support, and return barriers; 3) comprehensive analysis of the resulting benefits, which demonstrate that work-stealing overheads can be significantly reduced, leading to substantial performance improvements; and 4) a small set of language extensions that achieve both high performance and high productivity with minimal programmer effort.

A managed runtime forms the backbone of any modern implementation of a high-level language. Managed runtimes enjoy the benefits of a long history of research and their implementations are highly optimized. My thesis demonstrates that converging these highly optimized features together with the expressiveness of high-level languages, gives further hope for achieving high performance and high productivity on modern parallel hardware.

Contents

Acknowledgments **ix**

Abstract **xi**

1 Introduction **1**
1.1 Problem Statement 1
1.2 Scope and Contributions 2
1.3 Thesis Outline 3

2 Background **5**
2.1 Parallel Programming 5
2.2 Parallel Programming Models 5
2.2.1 The Address Space Model 6
2.2.1.1 X10 6
2.2.1.2 Habanero-Java 7
2.2.2 Task Parallelism 7
2.2.2.1 Work-Sharing 8
2.2.2.2 Work-Stealing 8
2.2.3 The Concurrent Object-oriented Model 9
2.2.3.1 Atomic Sets for Java (AJ) 10
2.3 An Implementation-Oriented Overview of Work-Stealing 11
2.3.1 Initiation 11
2.3.2 State Management 11
2.3.3 Termination 13
2.3.4 Work-Stealing in X10 13
2.3.5 Work-Stealing in Habanero-Java 16
2.3.6 Work-Stealing in Java ForkJoin 17
2.4 Managed Runtime Environments 18
2.4.1 The Quest for Productivity 18
2.4.2 Managed Runtime Features 18
2.4.2.1 The Yieldpoint Mechanism 18
2.4.2.2 Dynamic Compilation 19
2.4.2.3 On-Stack Replacement 19
2.4.2.4 Dynamic Code Patching 19
2.4.2.5 Exception Delivery Mechanism 20
2.4.2.6 Return Barriers 20
2.4.3 Jikes RVM 21

2.4.3.1 A Metacircular JVM 21
2.4.3.2 Support for Low-level Programming 21
2.4.3.3 Highly Optimized Compilers 22
2.4.3.4 Light-weight Locking Mechanism 22
2.5 Summary 22

3 Experimental Methodology 23
3.1 Benchmarks 23
3.1.1 Serial Elision 25
3.2 Software Platform 25
3.3 Hardware Platform 26
3.4 Measurements 26

4 Sequential Overheads of Work-Stealing 27
4.1 Introduction 27
4.2 Motivating Analysis 28
4.2.1 Sequential Overheads 28
4.2.2 Steal Ratio 31
4.3 Approach 33
4.3.1 Scalability Concerns 33
4.3.2 Techniques 33
4.4 Implementation 34
4.4.1 Runtime Supported X10OffStackWS 34
4.4.1.1 Initiation 34
4.4.1.2 State Management 35
4.4.1.3 Termination 36
4.4.2 X10TryCatchWS Java implementation 36
4.4.2.1 Leveraging Exception Handling Support 36
4.4.2.2 Initiation 37
4.4.2.3 State Management 39
4.4.2.4 Termination 40
4.4.2.5 Optimizing Runtime Support Calls 40
4.5 Results 40
4.5.1 Sequential Overhead 41
4.5.2 Work Stealing Performance 42
4.5.3 Memory Management Overheads 42
4.5.4 Steal Ratios 53
4.6 Related work 56
4.6.1 Languages versus Libraries 56
4.6.2 Work-stealing Deques 56
4.6.3 Harnessing Rich Features of a Managed Runtime 58
4.7 Summary 58

5 Dynamic Overheads of Work-Stealing 61
5.1 Introduction 61
5.2 Motivating Analysis 62
5.2.1 Steal Rate 62
5.2.2 Steal Overhead 64
5.3 Design and Implementation 65
5.3.1 Return Barrier Implementation 65
5.3.2 Overview of Conventional Steal Process 65
5.3.3 Installing the First Return Barrier 67
5.3.4 Synchronization Between Thief and Victim During Steal Process 68
5.3.5 Victim Moves the Return Barrier 68
5.3.6 Stealing From a Victim with Return Barrier Pre-installed 69
5.4 Results 69
5.4.1 Dynamic Overhead 69
5.4.2 Overhead of Executing Return Barrier 76
5.4.3 Free Steals From Return Barrier 77
5.4.4 Overall Work-Stealing Performance 79
5.5 Related Work 79
5.5.1 Stealing overheads 79
5.5.2 Return barriers 80
5.6 Summary 80

6 Performance and Productivity via Data-Centric Atomicity and Work-Stealing 81
6.1 Introduction 81
6.2 Motivation and Motivating Analysis 82
6.3 Design and Implementation 84
6.3.1 Annotations in AJWS 85
6.3.2 Translating AJWS to Java 88
6.3.2.1 Translating Concurrency Control Annotations to Java . 88
6.3.2.2 Translating Work-Stealing Annotations to Java 89
6.3.3 Build Integration 89
6.4 Results 90
6.4.1 Programmer Effort 90
6.4.1.1 Expressing Parallelism 90
6.4.1.2 Concurrency Control 91
6.4.2 Performance Evaluation 92
6.4.2.1 Sequential Overhead 92
6.4.2.2 Parallel Performance 94
6.5 Related Work 97
6.6 Summary 98

7 Conclusion 99
7.1 Future Work 100
7.1.1 Adaptive work-stealing 100
7.1.2 Utilizing slow path of thief for VM services 100
7.1.3 Steal-*N* work-stealing 101
7.1.4 Task Prioritization 101

A Jikes RVM Modifications for JavaTryCatchWS 103
A.1 Basic Infrastructure 103
A.2 Leveraging Exception Handling Support (Section 4.4.2.1) 103
A.3 Initiation (Section 4.4.2.2) 104
A.4 State Management (Section 4.4.2.3) 104
A.5 Termination (Section 4.4.2.4) 105
A.6 Return Barrier Implementation (Section 5.3.1) 105
A.7 Installing the First Return Barrier (Section 5.3.3) 105
A.8 Synchronization Between Thief and Victim During Steal Process (Section 5.3.4) 105
A.9 Victim Moves the Return Barrier (Section 5.3.5) 106
A.10 Stealing From a Victim with Return Barrier Pre-installed (Section 5.3.6) 106
A.11 Disabling work-stealing within a synchronized code block (Section 6.3.2.1)106

Bibliography 107

List of Figures

2.1 C++ and ICC++ variants of the same code. The **conc** annotation identifies potential concurrency. 9
2.2 Data-centric concurrency control in AJ language from Dolby et al. [2012]. 10
2.3 Using three different work-stealing implementations to write the Fib micro-benchmark. 12
2.4 Execution graph for `fib(4)`. 13
2.5 State during evaluation of `fib(4)`. Execution is at the first `fib(0)` task. Dashed arrows indicate work to be done. Dotted boxes indicate completed work. 14
2.6 State for victim and thief after stealing the continuation of `fib(4)`. . . . 15

4.1 Sequential overheads in work-stealing runtimes. Y-axis value 1.0 represents the execution of serial elision and anything above that is an overhead. Benchmarks not having any sequential overheads do not appear in the figure. 29
4.2 Work-stealing runtime consists of 3 different phase: initiation, state management and code-restructuring (termination). Initiation overhead in X10DefaultWS is the difference between X10DefaultWS and X10DefaultWS (No Deque), whereas in ForkJoin its the difference between ForkJoin and ForkJoin (No Deque). State management overhead in X10DefaultWS is the difference between X10DefaultWS (No Deque) and X10DefaultWS (No Deque, No Context). In ForkJoin, this is the value of ForkJoin (No Deque). Code-restructuring overhead is encountered only in X10 and this is represented by the value of X10DefaultWS (No Deque, No Context). 30
4.3 Steals to task ratio for each benchmark on different work-stealing runtimes. A low ratio is always preferred to ensure that sufficient tasks are created to keep all processors busy. 32
4.4 Pseudocode for the transformation of `fib(n)` from X10 (Figure 2.3(a), page 12) into X10TryCatchWS. We have modified the default X10 compiler to automatically generate the code to support X10TryCatchWS. . . 38
4.5 Sequential overheads in work-stealing runtimes. Y-axis value 1.0 represents the execution of serial elision and anything above that is an overhead. Benchmarks not having any sequential overheads do not appear in the figure. 41

4.6 (Cont.) Speedup relative to serial elision version of each benchmark on Jikes RVM. Y-axis value 1.0 denotes the execution of serial elision. Y-axis values greater than 1.0 represents faster execution than the corresponding serial elision version. 43
4.6 (Cont.) 44
4.6 (Cont.) 45
4.6 (Cont.) 46
4.6 (Cont.) 47
4.7 (Cont.) Speedup obtained by dividing the execution time of single thread JavaTryCatchWS work-stealing framework with the execution time of different runtimes for threads 1 to 16. Y-axis values greater than 1.0 shows better speedup over single threaded JavaTryCatchWS. JavaTryCatchWS runs over Jikes RVM; X10DefaultWS, Habanero-Java and ForkJoin is executed over OpenJDK; whereas NativeX10WS is the X10 running over it's C++ backend. 48
4.7 (Cont.) 49
4.7 (Cont.) 50
4.7 (Cont.) 51
4.7 (Cont.) 52
4.8 Fraction of time spent performing garbage collection work in different benchmarks. Memory management overhead in JavaTryCatchWS is very similar to that required in serial elision version. X10TryCatchWS and X10OffStackWS operates on X10 arrays rather than Java arrays. This results in relatively extra memory management overhead in these two runtimes as compared to JavaTryCatchWS. 53
4.9 Steals to task ratio for X10OffStackWS and X10TryCatchWS. A low ratio is always preferred to ensure that sufficient tasks are created to keep all processors busy. 54
4.10 Seal failure rates for X10OffStackWS and X10TryCatchWS. This is the ratio of total failed steal attempts and total successful steals. Steal attempts may fail due to either another thief or the victim winning the race to start that continuation. 55

5.1 Steal statistics for DefaultWS show that steal ratio and steal rate can significantly differ across benchmarks. Jacobi and Barnes-Hut exhibit a low steal ratio at 16 threads, but shows a high steal rate with same number of threads. 63
5.2 The dynamic overhead of DefaultWS is strongly correlated with the steal rate (Figure 5.1(b)). Jacobi has the highest steal rate and it has the highest dynamic overhead. 64
5.3 The victim's stack, installation, and movement of the return barrier. . . 66
5.4 (Cont.) Dynamic overhead in our old and new system. 70
5.4 (Cont.) 71
5.4 (Cont.) 72

5.4 (Cont.) 73
5.4 (Cont.) 74
5.5 Total Steals 75
5.6 Overhead of executing return barrier in victims. Barnes-Hut has the highest overhead because of the high number of steals during its execution. 76
5.7 Free steals due to the presence of return barrier on stack. Lower percentage of free steals tends to reflect a shallow stack and hence, lesser benefit from return barrier. 77
5.8 ReturnBarrierWS performance relative to JavaTryCatchWS, where time with *n* worker threads in ReturnBarrierWS is normalized to the time for same *n* worker threads in DefaultWS. Anything above 1.0 is a benefit. 78

6.1 Sample code written in AJWS, showing a simple example of **for** loop parallelism. 86
6.2 AJWS compiles AJWS code to vanilla Java, much like JavaTryCatchWS. The code above shows the vanilla Java translation of the AJWS example from Figure 6.1. 87
6.3 The sequential overheads associated with parallelism. Error bars indicate 95% confidence intervals. For lusearch-fix, the cost of introducing parallelism is around 7-8% for both the default Java implementation and AJWS. For jMetal, the performance overhead of introducing parallelism is around 50% for the default implementation and 20% for AJWS. We did not observe any performance overhead associated with parallelism in JTransforms. 92
6.4 Pseudocode showing the style of parallelism in default implementations of jMetal and JTransforms. 93
6.5 (Cont.) Speedup over serial elision version for both the Default and AJWS implementations. Only lusearch-fix is also evaluated over JavaTryCatchWS to determine whether our concurrency control annotations were hampering performance. 95
6.5 (Cont.) 96
6.6 A lower steals to task ratio is highly desirable to ensure a proper load-balancing. A higher steal ratio in JTransforms suggests the presence of very limited amount of parallelism. 96

List of Tables

6.1 Expressing parallelism in conventional Java has a major impact on how programs are written. For each of our three benchmarks we show, from left to right: a) the total number of class files; b) the total number of **synchronized** blocks and methods; c) the total number of files containing the **synchronized** keyword; d) the number of code fragments that express parallelism; e) the number of files containing explicitly parallel code; and f) the lines-of-code (LOC) overhead due to parallelism constructs within this code. 83

6.2 The syntactic overhead of using AJWS's work-stealing annotations to express parallelism. For both default and work-stealing implementations, we show the number of affected files, the number of affected code blocks and the overhead in lines of code. For all three benchmarks, parallelism can be expressed substantially more succinctly using work-stealing. 90

6.3 The syntactic overhead of using AJWS's data-centric annotations on our three benchmarks. JTransforms is lock-free, so there is no overhead for either approach. For jMetal there is a very small increase in lines of code, and for lusearch-fix, the overhead in lines of code is about half that of the default which uses the **synchronized** keyword. 91

6.4 Use of parallel regions that are exercised during execution of each of the benchmarks. For lusearch-fix and jMetal, this is substantially lower than the total number of parallel regions in the benchmark code bases (compare to Table 6.2), while for JTransforms, the number is slightly reduced. 93

Chapter 1

Introduction

This thesis addresses the problem of productively achieving high performance parallelism in managed languages.

1.1 Problem Statement

Computing hardware is becoming more and more complex. Today and in the foreseeable future, performance will be delivered principally in terms of increased hardware parallelism. This fact is an apparently unavoidable consequence of wire delay and the breakdown of Dennard scaling [Bohr, 2007], which together have put a stop to hardware delivering ever faster sequential performance. Single systems now scale to over one hundred cores.

The software community is facing orthogonal challenges of a similar magnitude with major changes in the way software is deployed, sold, and interacts with hardware. Software demands for correctness, complexity management, programmer productivity, time-to-market, reliability, security, and portability have pushed developers away from low-level programming languages towards high-level ones. These languages offer productivity and portability by building upon a managed runtime.

It is essential that these managed languages be able to efficiently exploit high degrees of hardware parallelism offered by modern hardware. Unfortunately, software parallelism is often difficult to identify and expose, which means it is often hard to realize the performance potential of modern processors. Common programming models using threads impose significant complexity to organize code into multiple threads of control and to balance work amongst threads to ensure good utilization of multiple cores. This shortcoming has helped work-stealing scheduling [Frigo et al., 1998a; Reinders, 2010] gain popularity. Work-stealing is a framework for allowing programmers to explicitly expose *potential* parallelism. A work-stealing scheduler within the underlying language runtime schedules work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. Work-stealing has been adopted by several high-level languages, such as X10 [Charles et al., 2005], Habanero-Java [Cavé et al., 2011] and Chapel [Chamberlain et al., 2007]. It is also offered as library in some managed languages, such as the Java ForkJoin framework [Lea, 2000] in Java 7 and the Task Parallel Library [Leijen et al., 2009] in the

Microsoft .NET framework [Platt, 2002].

Although the details vary among the various implementations of work-stealing schedulers, they all incur some form of overhead as a necessary side effect of enabling dynamic task parallelism. If these overheads are significant, then programmers are forced to carefully tune their applications to expose the 'right' amount of potential parallelism for maximum performance on the targeted hardware. Failure to expose enough parallelism results in under-utilization of the cores; exposing too much parallelism results in increased overheads and lower overall performance. Over-tuning of task size can lead to brittle applications that fail to perform well across a range of hardware systems and may fail to properly exploit future hardware. Therefore, techniques that significantly reduce the overheads of work-stealing schedulers simplify the programmer's task by largely eliminating the need to tune task size.

Achieving load balancing using work-stealing is just one dimension to productively achieving parallelism in managed languages. There is also a substantial cognitive load associated with ensuring parallel code is data-race free. This becomes a monolithic exercise in a large code-base. The programmer must ensure the correctness in the face of concurrency. Traditionally, managed languages provide object-level consistency by default for all objects. Recent work on Atomic Sets [Vaziri et al., 2006] takes a data-centric approach for consistency requirements. In this model, the programmer simply specifies that sets of object fields share some consistency property, without specifying what the property may be. The compiler ensures consistency by wrapping all race-prone code blocks with synchronized access. However, Vaziri et al.'s data-centric approach to achieving concurrency correctness is limited to the conventional thread-based parallelism.

The software community is thus facing the significant challenge of harnessing the true performance potential of modern hardware with managed languages.

1.2 Scope and Contributions

The aim of my research is to mitigate the two major challenges for productively exploiting high performance parallelism in managed languages—lowering the overheads associated with work-stealing scheduling and naively achieving correct parallelism by extending the design of high-level languages.

To do this, I chose X10 and Java as the representatives of modern high-level language and Jikes RVM [Alpern et al., 2000] Java virtual machine as the managed runtime implementation. To evaluate work-stealing, I chose a language-based approach which can execute over Jikes RVM. A language based approach for using work-stealing aims to increase programmer productivity [Charles et al., 2005]. Irrespective of these choices, the methodology and insights developed here should be applicable beyond this specific context.

Sequential overhead of work-stealing A work-stealing implementation can be viewed in terms of three phases, namely *initiation*, *state management* and *termination*. This thesis identifies that each of these phases contribute significantly to the sequential overheads—the increase in single-core execution time due to transformations that expose parallelism. We introduce two efficient designs which significantly reduce these overheads. The key to our approach is to exploit mechanisms already available within managed runtimes, namely: a) the yieldpoint mechanism; b) on-stack-replacement; c) dynamic code-patching; and d) support for efficient exception delivery. By combining these approaches, we lower the sequential overheads from 195% to 40%.

Dynamic overhead of work-stealing Once parallelism is introduced, additional dynamic overheads emerge in a work-stealing runtime. This overhead is associated with stealing amongst the threads and is most evident when parallelism is greatest. As core counts rise, dynamic overheads are an increasingly important factor in the performance of work-stealing runtimes. This thesis evaluates the dynamic overhead associated with work-stealing and proposes a new design to lower the cost of stealing. To do this, we apply the same strategy that we used successfully to reduce sequential overheads—reuse existing mechanisms in managed runtimes. Our design exploits a return barrier mechanism to reduce the dynamic overhead of work-stealing by almost 50%.

Performance and Productivity via Data-Centric Atomicity and Work-Stealing High-level languages should provide an easy to use interface for achieving parallelism–both in terms of correctness and performance. A parallel program is correct when program execution is free from data races. This thesis provides a small set of extensions to Java for achieving race-free concurrency. The approach is to identify the principal benefits of work-stealing and data-centric concurrency control [Vaziri et al., 2006; Dolby et al., 2012] respectively, and then bring those together into a single, simple framework within Java that combines data-centric concurrency control with a high performance work-stealing implementation. Using our Java extensions, we are able to substantially improve the performance of a number of realistic Java workloads.

In summary, this thesis addresses the goal of hiding the complexities associated with effectively taming the large-scale parallelism in modern hardware via managed languages. We show the potential of exploiting the rich features of managed runtimes in improving the efficiency of work-stealing implementations.

1.3 Thesis Outline

The body of this thesis is structured around the three key contributions outlined above.

Chapter 2 provides an overview of parallel programming models and managed runtime environments. It provides background on different work-stealing implementations used in this thesis. It also presents key features of managed runtimes which are extensively used in subsequent chapters. Chapter 3 discusses our experimental methodology.

Chapters 4, 5 and 6 comprise the main body of the thesis, covering the three key contributions. Chapter 4 and Chapter 5 identify sequential and dynamic overheads respectively in work-stealing implementations. They use features of managed runtime environments to develop novel techniques for addressing these overheads. Then Chapter 6 designs and implements easy to use annotations for achieving data-race free parallelism in Java.

Finally Chapter 7 concludes the thesis, describing how my contributions have identified, quantified, and addressed the challenges of achieving high performance parallelism in modern managed languages. It further identifies key future directions for research.

Chapter 2

Background

This thesis takes a well-known idea, work-stealing to exploit parallel hardware, and asks why it doesn't work as well as it should with managed languages. This chapter provides relevant background information on parallel programming models and key features of managed runtimes.

The chapter starts with a brief discussion on parallel programming in Section 2.1 and parallel programming models in Section 2.2. Section 2.3 provides an implementation-oriented overview of work-stealing scheduling. Section 2.4 discusses managed runtime environments. The chapter concludes by introducing Jikes RVM and features relevant to this thesis.

2.1 Parallel Programming

Parallel programming is the technique of using two or more computing elements to solve a single problem. Parallel programming is used heavily in various areas including weather forecasting, astronomy, oceanographics, and data mining. Parallel programming dates back to the late 1950's with the advent of shared memory multiprocessors. The scale of parallel systems grew until massively parallel processors came to existence. Starting in the late 1980's, cluster computing came into picture and became the dominant architecture for parallel programming. However, with the advent of multicore processors from 2001 [IBM, 2002], parallel programming is now no longer a niche area and is becoming mainstream. Today, dual and quad core processors are at the heart of most modern computing devices. The need to achieve more and more performance without increasing the power consumption and heat dissipation, promises that today's multicore processors will be replaced with many-core processors [Jeffers, 2013; Intel Corporation, 2013].

2.2 Parallel Programming Models

Parallel programming models are concerned with different ways of expressing a parallel program. The ultimate goal is to parallelize a sequential code in such way that is simple and achieves the maximum performance when executed. To serve this goal of productivity and performance, there has been a plethora of research. Accordingly, there are various ways to classify different parallel programming models existing today. For this thesis, we classify it in three broad categories: *address space model*, *task parallelism model* and *concurrent object-oriented model*.

2.2.1 The Address Space Model

The address space model defines how data are referred to by the parallel program. The two dominant models are the *shared memory* and *distributed memory* models. When using shared memory, a computation is generally divided into threads, which uniformly share the memory. The sharing of a single logical memory may be implemented in hardware [Owicki and Agarwal, 1989], or in software [Keleher et al., 1994; Carter et al., 1991]. Contrarily, with distributed memory, the computation is generally divided among processes and each of the processes execute on a separate processor having its own private address space. To communicate, these processes generally send messages over a high speed interconnect. OpenMP [Dagum and Menon, 1998] is a popular standard for shared memory parallel programs, whereas Message Passing Interface (MPI) [Snir et al., 1995] is the de-facto standard for programming large distributed memory system. The cost of maintaining coherence (whether in hardware or software) means that shared memory does not scale as well as distributed memory.

The *Partitioned Global Address Space* (PGAS) programming model [PGAS, 2011] has recently gained popularity. It strikes a balance between shared and distributed memory models. It provides ease of programming due to its global address memory model and performance due to locality awareness. Here, the threads are mapped to processes and threads, as supported by the language runtime. Languages in this category include, Co-Array Fortran [Numrich and Reid, 1998], Titanium [Yelick et al., 1998] and UPC [El-Ghazawi and Smith, 2006].

Despite the benefits, PGAS model suffers two drawbacks. First, it implicitly assumes that all processes run on similar hardware and second, it does not support dynamically spawning multiple activities [Saraswat et al., 2010]. The *Asynchronous Partitioned Global Address Space* (APGAS) programming model [Saraswat et al., 2010] addresses these drawbacks and can be thought of as being derived from both the MPI and OpenMP models by extending the PGAS model with **`place`** and **`async`** (Section 2.2.1.1). Languages that follow this model include, X10 [Charles et al., 2005], Chapel [Chamberlain et al., 2007], Fortress [Allen et al., 2005] and Habanero-Java [Cavé et al., 2011].

We will now briefly discuss X10 and Habanero-Java, which are used in this thesis.

2.2.1.1 X10

X10 is a strongly-typed, imperative, class-based, object-oriented programming language designed for high performance computing. The sequential core of X10 is very similar to the Java and C++ programming languages [Saraswat et al., 2013]. X10 comes in two flavors: a) Managed X10, built on a Java backend; and b) Native X10, built on a C++ backend.

X10 includes specific features to support parallel and distributed programming. In X10 locality is explicitly represented in the form of *places*. A **place** can be thought of as a virtual shared memory multiprocessor, where a computational unit has a finite number of threads sharing a bounded amount of shared memory uniformly. Asynchronous *activities* address the requirements of both thread-based parallelism and asynchronous data transfers in X10. Every X10 activity runs inside a **place**. While an activity executes at the same **place** throughout its lifetime, it may dynamically spawn activities in remote places. A new activity, S, is created by the statement **async** S. To synchronize activities, X10 provides the statement **finish** S. Control will not return from within a finish until all activities spawned within the scope of the finish have terminated. For achieving concurrency, X10 provides language constructs such as **when** and **atomic**. This thesis uses only two of the X10 language constructs for achieving work-stealing task parallelism in Managed X10 (**async** and **finish**). Sample code written in X10 using **finish** and **async** is shown in Figure 2.3(a) (explained in detail in Section 2.3.4). X10 restricts the use of a local mutable variables inside **async** statements. A mutable variable (**var**) can only be assigned to or read from within the **async** it was declared in. To mitigate this restriction, X10 permits the asynchronous initialization of final variables (**val**). A final variable may be initialized in a child **async** of the declaring **async**. A definite assignment analysis guarantees statically that only one such initialization will be executed on any code path, so there will never be two conflicting writes to the same variable.

2.2.1.2 Habanero-Java

The Habanero-Java language was developed at Rice University as an extension to the original Java-based definition of the X10 language. Habanero-Java can also be thought of an extension of Java for the APGAS programming model. Parallel programming constructs in Habanero-Java are similar to that in X10 (**place**, **async** and **finish**). To achieve concurrency, Habanero-Java allows Java concurrent locks [Lea, 2004] and the **isolated** construct (similar to **atomic** in X10). As with X10, for Habanero-Java, this thesis only uses **finish** and **async** constructs. Figure 2.3(b) shows sample code written in Habanero-Java, explained in detail in Section 2.3.5..

2.2.2 Task Parallelism

A programmer using task parallelism decomposes a sequential computation into several small sub-computations, called as *tasks*. These tasks are created such that they can be executed in parallel. Scheduling parallel tasks is not easy, so the runtime takes the burden of scheduling the tasks. The runtime generally creates a fixed size thread pool, which can execute each of the tasks from start to finish. Once the execution of tasks is complete, the results are joined together to complete the actual computation.

Two commonly used task parallelism models are *work-sharing* and *work-stealing*, which are explained below.

2.2.2.1 Work-Sharing

Work-sharing task parallelism relies on a centralized shared task queue. Surplus tasks are continuously added to this queue by the runtime. Whenever a worker thread is idle, it will poll this queue and retrieve a task to execute locally. Due to this multi-threaded environment, any addition or removal of tasks to or from the queue always needs to be synchronized. However, as the thread pool size increases, this synchronization can become a scalability bottleneck. Work-stealing addresses this limitation and improves the scalability. OpenMP supports a work-sharing runtime, where the programmer declares the parallel regions in his code using compiler directives (*pragmas*).

2.2.2.2 Work-Stealing

Work-stealing is a very efficient strategy for distributing work in a parallel system. The ideas behind work-stealing have a long history which includes lazy task creation [Mohr et al., 1990] and the MIT Cilk project [Frigo et al., 1998b], which offered both a theoretical and practical framework. In this model, the runtime maintains a pool of *worker threads*, each of which maintains a local set of *tasks*. When local work runs out, the worker becomes a *thief* and seeks out a *victim* thread from which to *steal* work.

The elements of a work-stealing runtime are often characterized in terms of the following aspects of the execution of a task-parallel program:

Fork A fork describes the creation of new, potentially parallel, work by a worker thread. The runtime makes new work items available to other worker threads.

Steal A steal occurs when a thief takes work from a victim. The runtime provides the thief with the execution context of the stolen work, including the execution entry point and sufficient program state for execution to proceed. The runtime updates the victim to ensure work is never executed twice.

Join A join is a point in execution where a worker waits for completion of a task. The runtime implements the synchronization semantics and ensures that the state of the program reflects the contribution of all the workers.

Work-stealing implementations include: Cilk [Frigo et al., 1998a], Java ForkJoin [Lea, 2000], Habanero-Java, Microsoft's Task Parallel Library [Leijen et al., 2009], Intel Threading Building Blocks [Reinders, 2007], X10 and Chapel.

```
{
  int x = a.foo();
  b.bar();
  c.baz(x);
}
```

(a) C++

```
conc {
  int x = a.foo();
  b.bar();
  c.baz(x);
}
```

(b) ICC++

Figure 2.1: C++ and ICC++ variants of the same code. The **`conc`** annotation identifies potential concurrency.

2.2.3 The Concurrent Object-oriented Model

The concurrent object-oriented model goes back at least to Actors [Agha, 1986]. In this model, synchronization is managed at the level of each object, with each object handling messages sent to it in some sequential order. The original model made state changes atomic with a single `become` operation; subsequent languages such as ABCL [Yonezawa, 1990] and ICC++ [Chien, 1996] present a more conventional imperative semantics in which synchronization is still enforced at the object level, but with some form of lock.

The actor model naturally expresses concurrency in terms of concurrent message sends to objects, and languages like ICC++ carry on this style. A key semantic of languages like ICC++ is that concurrency is *elective* in the sense that concurrency annotations indicate potential concurrency but the runtime system is left with flexibility to determine when to actually create parallel tasks. The other key design goal was to make expressing concurrency as simple as possible. With ICC++, it takes the form of the **`conc`** annotation as shown in Figure 2.1, where a standard C++ block (Figure 2.1(a)) is shown in contrast to a **`conc`** block (Figure 2.1(b)).

In Figure 2.1, the semantics of the **`conc`** block constrains `x = a.foo()` to execute before `c.baz(x)` due to the data dependence but otherwise allows all work in the block to execute concurrently. These semantics naturally generalize to support concurrent loops with the same **`conc`** annotation applied to the loop. Note the syntactic minimality of the **`conc`** annotation with respect to the sequential code in Figure 2.1(a).

In this model, there is an implicit synchronization at the end of the block, essentially a join with the tasks inside the **`conc`** block.

```
class Counter {
  atomicset a;
  atomic(a) int value;
  int get() { return value; }
  void dec() { value--; }
  void inc() { value++; }
}
```

(a) Example of **atomicset** and **atomic** annotations in AJ.

```
class PairCounter {
  atomicset b;
  atomic(b) int diff;
  Counter|a=this.b| low = new Counter|a=this.b|();
  Counter|a=this.b| high = new Counter|a=this.b|();
  void incHigh() {
    high.inc();
    diff = high.get()-low.get();
  }
  ...
}
```

(b) Example of **atomicset** aliasing in AJ.

Figure 2.2: Data-centric concurrency control in AJ language from Dolby et al. [2012].

2.2.3.1 Atomic Sets for Java (AJ)

Traditionally, concurrent object-oriented languages provide object-level consistency by default for all objects. More recently, the Atomic Sets model [Vaziri et al., 2006; Dolby et al., 2012] extended Java with explicit data-centric synchronization in which the programmer specifies that sets of object fields share some consistency property, without specifying what the property may be. This differs from traditional object-oriented models in two key ways: the first is that all consistency is specified by the programmer, and unannotated fields have no synchronization. The second is that the idiom naturally encompasses specifying both subsets of an object's fields and sets of fields that span multiple objects.

Figure 2.2 illustrates the Atomic Sets constructs as described in [Dolby et al., 2012]. Figure 2.2(a) is a simple example in which class `Counter` declares a single Atomic Set, `a`, with the **`atomicset`**`(a)` declaration and that the field `value` is in `a` with the **`atomic`**`(a)` declaration. Figure 2.2(b) is a more complex example in which `PairCounter` declares an Atomic Set, `b`, containing `diff`. It also includes two `Counter` objects, `low`, and `high`; the `|a=`**`this`**`.b|` declarations indicate that the Atomic Set `b` is aliased to Atomic Set `a`, i.e. both the objects have the same Atomic Set. This is how Atomic Sets can be extended to encompass the state of multiple objects.

2.3 An Implementation-Oriented Overview of Work-Stealing

The focus of this thesis is on the implementation of work-stealing scheduling. This section discusses a work-stealing scheduler from an implementation point of view. A work-stealing implementation can be thought of as following basic phases, each of which require special support from the runtime or library:

1. Initiation. (Allow tasks to be created and stolen atomically).
2. State management. (Provide sufficient context for the thief to be able to execute stolen execution, and the ability to return results).
3. Termination. (Join tasks and ensure correct termination).

We now explain each of these and what they require of the runtime, first generally, then concretely in terms of X10, Habanero-Java and Java ForkJoin implementations.

To help illustrate work-stealing, we use a running example of the recursive calculation of Fibonacci (Fib) numbers. Figure 2.3 shows X10, Habanero-Java and ForkJoin code for computing Fib numbers. Figure 2.4 shows the graph of the recursive calls made when executing `fib(4)` in Figure 2.3. Calls to the non-recursive base case (`n < 2`) are shown as rectangles.

2.3.1 Initiation

Initiation is concerned with ensuring that: 1) tasks are available to be stolen whenever appropriate, and 2) each task is only executed once. A *task* in this thesis is similar to *frame* in the Cilk [Frigo et al., 1998b] implementation. An idle thread may make itself useful by stealing work, so becoming a thief. This begins with the thief identifying a victim from which to steal. For example, the thief may randomly select a potential victim and if they appear to have work available, attempt a steal.

Tasks are typically managed by each worker using a double ended queue (deque), one deque per worker, as illustrated in Figure 2.5. Each worker pushes tasks onto the tail of its deque using an unsynchronized store operation. Both the worker and any potential thieves then use atomic compare-and-swap (CAS) instructions to remove tasks from the worker's deque, with the worker acquiring from the tail (newest), and thieves attempting to acquire from the head (oldest). Tasks are thus made available and only stolen once, with the deque discipline minimizing contention and increasing the probability that long-running tasks are stolen.

2.3.2 State Management

When a task is stolen, the thief must: 1) acquire all state required to execute that task, and 2) provide an entrypoint to begin execution of the task, and 3) be able to return or combine return state with other tasks. Work-stealing implementations typically meet requirements 1) and 3) through the use of *state objects* that capture the required information about the task, and provide a location for data to be stored and

```
def fib(n:Int):Int {
  if (n < 2) return 1;

  val a:Int;
  val b:Int;

  finish {
    async a = fib(n-1);
    b = fib(n-2);
  }

  return a + b;
}
```

(a) Fib in X10.

```
static class IntegerBox {
  public int v;
}

static void fib (int n, IntegerBox res) {
  if (n < 2) return 1;

  final IntegerBox x = new IntegerBox();
  final IntegerBox y = new IntegerBox();

  finish {
    async fib (n-1,x);
    fib (n-2,y);
  }

  res.v = x.v + y.v;
}
```

(b) Fib in Habanero-Java.

```
Integer compute() {
  if (n < 2) return 1;

  Fib f1 = new Fib(n - 1);
  Fib f2 = new Fib(n - 2);

  f1.fork();
  int a = f2.compute();
  int b = f1.join();

  return a + b;
}
```

(c) Fib in Java ForkJoin.

Figure 2.3: Using three different work-stealing implementations to write the Fib microbenchmark.

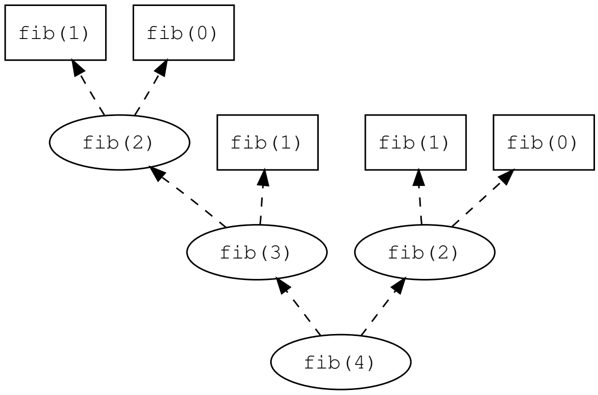


Figure 2.4: Execution graph for `fib(4)`.

shared across multiple tasks. Requirement 2) is handled differently depending on the execution model, and is discussed in more detail below for specific systems.

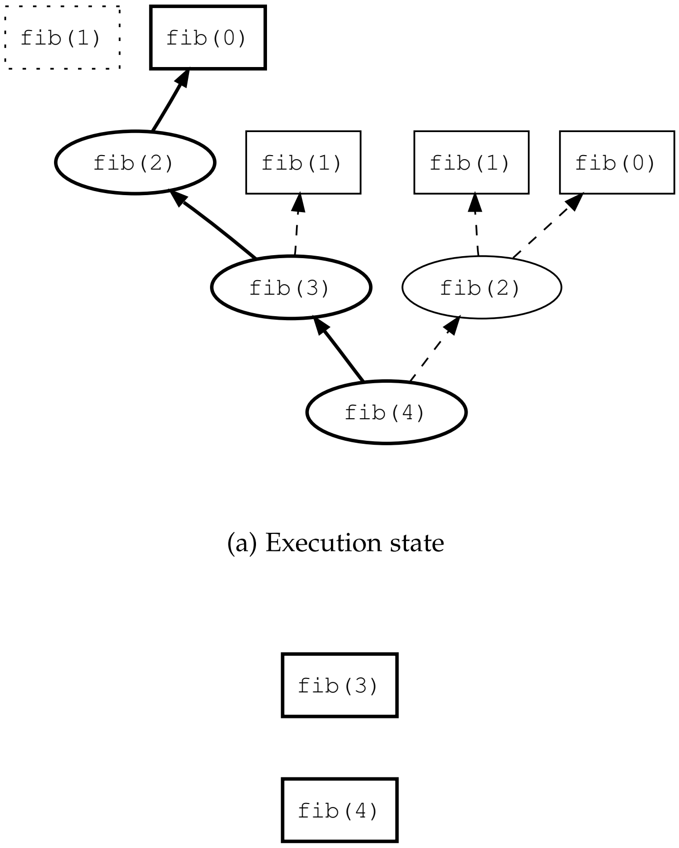
2.3.3 Termination

In general, the continued execution of a worker is dependent on completion of some set of tasks, each of which may be executed locally, or by a thief. Such dependencies are made explicit by the programmer and must be respected by the implementation of the work-stealing runtime. The work-stealing runtime must: 1) handle the general case where execution waits, dependent on completion of stolen tasks executing in parallel. However, it is also critical for the scheduler to: 2) efficiently handle the common case where no tasks in a particular context are stolen, and therefore are all executed in sequence by a single worker. Furthermore, work-stealing schedulers also aim to: 3) maintain a particular level of parallelism. To ensure that this occurs, when a worker is waiting on completion of a stolen task, instead of suspending the worker, the scheduler may attempt to have that worker find and execute another task.

2.3.4 Work-Stealing in X10

X10's work-stealing scheduler is implemented by a combination of an X10-source-to-X10-source program transformation and a runtime library. The program transformation synthesizes code artifacts (continuation methods and frame classes) required by the runtime scheduler. X10 meets the key work-stealing requirements as follows:

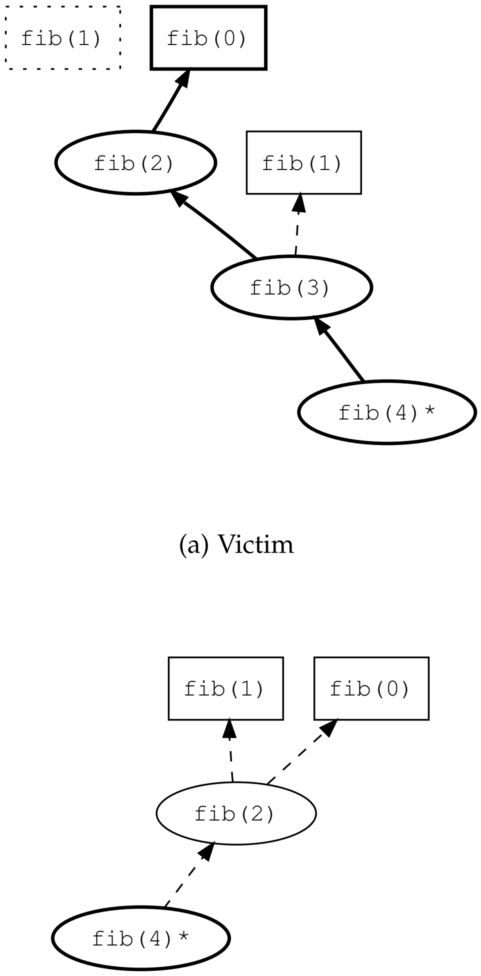
Initiation. X10's work-stealing workers use deques as described above. Like Cilk, X10 adopts a *work-first* scheduling policy: when a worker encounters an **`async`** statement, it pushes the *continuation* of the current task to its deque and proceeds with



(a) Execution state

(b) Deque

Figure 2.5: State during evaluation of `fib(4)`. Execution is at the first `fib(0)` task. Dashed arrows indicate work to be done. Dotted boxes indicate completed work.



(a) Victim

(b) Thief

Figure 2.6: State for victim and thief after stealing the continuation of `fib(4)`.

the execution of the **async** body. For instance, a worker running `fib(4)` first executes `fib(3)` (Figure 2.3(a) line 8), making the `fib(2)` work item (line 9) available for others to steal. When done with **async** `fib(3)`, the worker attempts to pop the tail deque item and if non-null, will execute the continuation (`fib(2)`, line 9).

State management. Each thread's stack is private, so in order to permit multiple workers to concurrently access and update the program state, the X10 compiler encapsulates sharable state into *frame objects*. Consequently, methods are rewritten to operate on fields of frame objects instead of local variables. Frame objects are linked together into trees that shadow the tree structure of the task graph. In other words, Figure 2.4 represents the tree of frame objects assembled during the execution of `fib(4)`. When X10 is compiled to Java, frame objects are created on the heap to ensure that they are accessible to both the worker and potential thieves. In the C++ implementation however, an optimization is performed that sees frame objects stack-allocated by default, and only lazily migrated to the heap when a steal occurs [Tardieu et al., 2012]. The managed X10 compiler analyzes the source code and indexes all of the points immediately after **async** statements ('reentry' points). It then generates a second copy of the source code in which methods take a *pc* (program counter) as an extra argument. The control flow of the generated methods is altered so as to permit starting execution at the specified pc.

Termination. If a worker proceeds from the beginning of a finish block to its end without detecting a steal, then that worker has itself completed every task in the finish context and may return. Termination is more complex when a steal occurs. When a thief steals a work item within the scope of a **finish**, the scheduler begins maintaining an atomic count of the active tasks within that **finish** body. When a worker completes a task, or execution reaches the end of the **finish** body, the count is atomically reduced and checked. If the count is non-zero, the worker gives up and searches for other work to process. When the count is zero, then the finish is complete and the worker starts executing the continuation of the finish statement.

2.3.5 Work-Stealing in Habanero-Java

Habanero-Java (Section 2.2.1.2) evolved from early versions of X10 and the **finish**-**async** programming in Habanero-Java is very similar to X10. A broader description of **finish**-**async** work-stealing in Habanero-Java appears in [Guo, 2010]. Unlike X10, Habanero-Java offers three different work-stealing policies: a) a work-first policy; b) a help-first policy [Guo et al., 2009]; and c) an adaptive policy [Guo et al., 2010].

The implementation of Fib for Habanero-Java is shown Figure 2.3(b).

Initiation. The work-first version in Habanero-Java works very much like X10 (Section 2.3.4). In all the three work-stealing policies, each worker maintains a local deque similar to the X10 work-stealing implementation. In the help-first version, the **async** body is pushed to the deque and the *continuation* is executed first. If this policy is used, the worker running `fib(4)` will push `fib(3)` (Figure 2.3(b), line 12) on the deque and first executes `fib(2)` (line 13). In the adaptive policy, the runtime dynamically switches between work-first and help-first policies.

State Management. In order to permit multiple workers to concurrently access and update the program state, Habanero-Java maintains heap allocated *frame objects* similar to X10. For the `fib` function (Figure 2.3(b)), slow and fast clone versions of the method generated by the work-stealing compiler [Raman, 2009]. To mimic **`val`** in X10 (Figure 2.3(a), lines 4 and 5), Habanero-Java requires heap allocation of objects to capture results (Figure 2.3(b), lines 8 and 9).

Termination. The **`finish`** guarded termination in Habanero-Java is very similar to X10.

2.3.6 Work-Stealing in Java ForkJoin

The Java ForkJoin framework [Lea, 2000] is an implementation of the interface `java.util.concurrent.ExecutorService` that takes advantage of work-stealing scheduling in Java 7 [Oracle Corporation, 2013]. It was originally implemented by Doug Lea [Lea, 1999]. The general design of the ForkJoin framework is a variant of the work-stealing framework devised by Cilk and is explained in detail in [Lea, 2000]. Here we briefly discuss its key components. The ForkJoin implementation of Fib is shown in Figure 2.3(c).

Initiation. To allow `fib(n-1)` to be stolen, the user explicitly heap-allocates a `Fib` object (Figure 2.3(c), line 4) and calls **`fork`**`()` on this object (line 7). Like X10, every worker thread maintains a deque. **`fork`** pushes a task to the deque, making it available to be stolen.

State Management. In ForkJoin, tasks are represented as task objects. These objects include: methods for scheduling and synchronizing with the task, any state associated with the task, and an explicit entrypoint for executing the task.

Termination. When a worker thread encounters a **`join`** operation, it processes other tasks, if available, until the subject of the **`join`** has been completed (either by the worker or by a thief). When a worker thread has no work and fails to steal any from others, it backs off (via `yield`, `sleep`, and/or priority adjustment) and tries again later unless all workers are known to be similarly idle, in which case they all block until another task is invoked from the top-level.

2.4 Managed Runtime Environments

This section provides the background on the managed runtime environments and features that are pertinent to this thesis.

2.4.1 The Quest for Productivity

Processor vendors can no longer deliver performance by increasing single core frequency. Instead, they have started manufacturing chips with with multiple processor cores and these are now mainstream. Further, to reduce power, processor architects are now turning to customization and heterogeneity [Borkar and Chien, 2011; Cao et al., 2012; Balakrishnan et al., 2005; ARM Corporation, 2011; Morad et al., 2006; Venkatesh et al., 2010; Suleman et al., 2009; Li et al., 2010; Koufaty et al., 2010]. Taming this new generation of processors is a very daunting task and is a first order concern for software deployment. Managed languages aim to enhance the programmer's productivity even in the face of complex hardware. They provide abstraction over the hardware by using managed runtime environments, which deliver portability, reliability, security and faster time to market.

2.4.2 Managed Runtime Features

The key features of modern managed runtimes were nailed down by Smalltalk-80 [Deutsch and Schiffman, 1984] and Self-91 [Chambers and Ungar, 1991] systems. Managed runtimes came into their own with the advent of the Java programming language [Gosling et al., 1996] and has been a very active research area since. Recent examples of widely used virtual machine implementations include Java Virtual Machine [Oracle, 2013] for Java; Common Language Runtime for Microsoft's .NET framework [Box and Pattison, 2002]; Parrot Virtual Machine [Wall and Schwartz, 1991] for dynamic programming languages like Perl; Dalvik virtual machine [Bornstein, 2008] as a part of Android operating system [Rogers et al., 2009]; PyPy [Rigo and Pedroni, 2006] for Python [Martelli, 2003]; YARV for Ruby [Sasada, 2005]; etc.

Managed runtimes are widely used due to their rich features. Some of the key features, which are exploited in this thesis are as discussed below.

2.4.2.1 The Yieldpoint Mechanism

A yieldpoint is a mechanism for supporting quasi-preemptive thread scheduling [Arnold et al., 2000]. Yieldpoints are also the program locations where it is *safe* to run an exact garbage collector. Apart from garbage collection, the VM uses yieldpoints to implement services such as adaptive optimization. Yieldpoints are generated by the compiler as program points where a running thread checks a dedicated bit in a machine control register to determine whether it should yield. The compiler generates precise stack maps at each yieldpoint. If the bit is set, the yieldpoint is taken and some action is performed. If the bit is not set, no action is taken and the next instruction after the yieldpoint is executed. In Jikes RVM virtual machine (Section 2.4.3), yieldpoints are inserted in method prologues and on loop back edges [Arnold et al., 2000].

2.4.2.2 Dynamic Compilation

Dynamic compilation, also known as Just-In-time compilation (JIT) allows dynamically loaded code to be compiled. The alternatives are to interpret the code, or to prohibit dynamic loading. Dynamic compilation is also used to implement adaptive, or feedback directed optimization [Arnold et al., 2002]. Software systems have been using JIT compilation techniques since the 1960s [McCarthy, 1960]. Aycock [2003] describes the history of JIT compilers in detail. Modern virtual machines have very sophisticated JIT compilers. Generally code is first interpreted, or compiled with a baseline non-optimizing compiler. When the virtual machine notices that code is executed frequently (i.e. hot), the JIT compiler is invoked to recompile the code with higher optimization levels.

2.4.2.3 On-Stack Replacement

There are many adaptive strategies employed to selectively compile and recompile hot methods [Arnold et al., 2000, 2002; Paleczny et al., 2001; Suganuma et al., 2001]. Transition to the newly compiled version of a method can be done by modifying dispatch structures, so that future method invocations branch to the new compiled version. However, the transition for a method that is currently executing on some thread's stack presents a formidable engineering challenge. One such example is inlining of methods inside a long running loop. On-stack replacement (OSR) is a mechanism employed by virtual machines for replacing the currently executing method and re-initiating the execution in a new version. This mechanism was first deviced by Self programming language [Hölzle and Ungar, 1994; Chambers and Ungar, 1991]. OSR is also exploited to provide enhancements such as debugging optimized code via de-optimization [Hölzle et al., 1992], deferred compilation to improve compiler speed and/or code quality [Chambers, 1992] and optimization and code generation based on speculative program invariants [Paleczny et al., 2001]. In a nutshell, OSR mechanism works as follows [Fink and Qian, 2003]: 1) extract compiler-independent state (program variables, etc.) from a suspended thread; 2) generate new code for the suspended method; and 3) transfer execution in the suspended thread to the new compiled code. OSR is used in several modern virtual machines such as Jikes RVM, HotSpot JVM [Paleczny et al., 2001] and V8 JavaScript engine [Google, 2013].

2.4.2.4 Dynamic Code Patching

Dynamic code patching refers to the modification of one instruction to transform it into another in the presence of multiple execution threads [Sundaresan et al., 2006]. The oldest implementation of Dynamic code patching dates at least as far back as the SELF language which used it for supporting incremental recompilation [Chambers et al., 1989]. In modern JVMs it is extensively used to perform JIT compilation, such as: class resolution; interface method dispatch; method recompilation support; speculative optimization support; and method dispatch in case of static and devirtualized calls [Sundaresan et al., 2006; Ishizaki et al., 2000; Cierniak et al., 2000]. Dynamic code patching is also used to implement return barriers (Section 2.4.2.6).

To understand dynamic code patching, we will refer to the case of dynamic class resolution. According to JVM specifications [Lindholm and Yellin, 1999], class resolution should be performed lazily. When a method is compiled for the first time, it might contain references to unresolved classes on paths that have not executed prior to this compilation. During the execution, once the unresolved references are resolved, code patching is used to replace the old instructions to point to the resolved references.

2.4.2.5 Exception Delivery Mechanism

An exception is an event which can occur during the execution of the program and disrupts the program's control flow. Handlers are subroutines where the exceptions are resolved and they may allow resuming the execution from the original location of the exception. Exception handling is an old technique [Goodenough, 1975]. Many modern programming languages like C++ [Schilling, 1998], Java [Gupta et al., 2000] and C# [Petzold, 2010] support exception handling mechanisms. There are two commonly used techniques to implement exception handling: stack unwinding and stack cutting [Ramsey and Peyton Jones, 2000]. When an exception is thrown, stack unwinding will unwind the stack frames to search for the corresponding exception handler. On the other hand, stack cutting will search a list of registered exception handlers. Languages like Java and C++ use **`try`** and **`catch`** blocks for exception handling. When an exception is thrown, the key steps of Handling the exception are as follows: 1) mapping program context (i.e. program counter), through which an exception is thrown to the corresponding **`try`** block; 2) filtering the exception as there can be more than one handler (**`catch`** block) for the **`try`** block; and 3) searching for the **`try`** blocks of caller methods if the correct **`catch`** block is not found.

2.4.2.6 Return Barriers

The return barrier mechanism was first used by Hölzle et al. [1992] in the context of debugging optimized code, to allow lazy dynamic deoptimization of the stack. A return barrier, like a write barrier, allows the runtime to intercept a common event (using dynamic code patching), and (conditionally) interpose special semantics. In the case of a write barrier, a runtime typically interposes itself on pointer field updates, conditionally remembering updates of pointers in certain conditions. On the other hand, a return barrier interposes special semantics upon the return from a method (which corresponds to the popping of a stack frame). One use of return barrier [Yuasa et al., 2002] is to consistently divide the scanning of the stack (at the begining of a "single" GC cycle) into multiple steps each of which scans a fixed, small number of stack frames in order to reduce the GC pause time.

2.4.3 Jikes RVM

Jikes RVM is an open source Java virtual machine (JVM), originally known as Jalapeño, and developed at IBM research [Alpern et al., 2000]. It is a high performance JVM, which also provides a flexible open testbed to prototype virtual machine technologies and experiment with a large variety of design alternatives. Some of the key features of Jikes RVM, which influenced our decision to use it in this thesis is are described below.

2.4.3.1 A Metacircular JVM

Most JVMs implement their runtime services in 'native code' i.e. a non-Java language such as C, C++ or assembler. One of the unique characteristics of Jikes RVM is that it is a metacircular JVM — a Java VM written in Java. Other metacircular VM implementations include: Maxine [Wimmer et al., 2013], Squawk [Simon and Cifuentes, 2005], PyPy [Rigo and Pedroni, 2006] and Klein [Ungar et al., 2005]. There are several benefits of writing a JVM in Java. A number of complex processes performed by a JVM can be better expressed in Java, which offers features such as type safety, garbage collection and exception handling. Metacircularity also removes the impedance mismatch between application and runtime code, which can provide a significant performance advantages [Alpern et al., 2000]. When the language impedance mismatch is removed, the frequently executed runtime services (such as object allocations) can be inlined and optimized into user code, producing a better optimized code.

2.4.3.2 Support for Low-level Programming

Implementing a VM in a high level language like Java has several benefits but it can create difficulties when there is a need to do low-level programming. Examples of low-level programming include access to memory layout, machine register usage, or the ability to access hardware-specific features such as special machine instructions. Fortunately, Jikes RVM has a rich framework, `org.vmmagic`, that allows high-level low-level programming in Java [Frampton et al., 2009]. It is able to do this by providing type-system extensions and semantic extensions. Type-system extension is achieved by using two mechanisms: a) the concept of raw storage (`@RawStorage` annotation) which allows user to associate a type with a raw chunk of backing data of specified size (byte or architectural width word); and b) unboxed types (`@Unboxed` annotation), which are treated like Java primitive types (rather than an object) and hence allocated only on the stack. Semantic extension is also provided in two ways: a) through intrinsic functions, allowing special semantics in Java, for example `@Intrinsic(LOAD_BYTE)` for loading memory from an address; and b) through semantic regimes, which allow certain static scopes to operate under a regime of altered semantics such as unchecked memory operations, turning on and off bounds check etc.

2.4.3.3 Highly Optimized Compilers

Jikes RVM follows a compile-only approach. In the first pass, all the methods are compiled by a fast but non-optimizing compiler, called the baseline compiler. The baseline compiler produces code slightly better than the interpreted code. Next is the JIT compiler (Section 2.4.2.2), which employees an Adaptive Optimization System (AOS) to decide *when* to recompile a method and to *what* optimization level [Arnold et al., 2000]. The AOS drives recompilation decisions based on the estimated cost and benefit of compiling each method. The AOS in Jikes RVM uses the OSR mechanism for transferring execution from one version of compiled code to another. The OSR is high performance and has a relatively simple implementation in Jikes RVM [Fink and Qian, 2003]. The implementation of OSR helped the evolution of novel ideas discussed in further chapters of this thesis.

2.4.3.4 Light-weight Locking Mechanism

High-level languages like Java allow objects to serve as a lock at any time (using the `synchronized` statement). This can prove disadvantageous to the runtime, since extra space must be allocated by the runtime to hold the lock for each object. The other issue is the overhead of lock and unlock operations which require expensive compare-and-swap operations, especially if they are very frequent. Several techniques have been devised in this area to improve locking performance [Bacon et al., 1998; Onodera and Kawachiya, 1999; Kawachiya et al., 2002]. Thin locks is one such technique, where the space overhead of the lock is reduced by using bits in the object header as a lock. Biased locking is another technique, which avoids the need for CAS operations by 'biasing' the lock to an 'owner' thread. Jikes RVM implements thin locks and a refinement of bias locks called as Fine-Grain Adaptive Bias Locks (FABLE) [Pizlo et al., 2011]. This adaptive technique can automatically decide at runtime whether to invoke bias or thin mode locking.

Locks play an integral role in the implementation of any scheduling techniques. The light-weight locking implementation in Jikes RVM supports a low overhead implementation of novel ideas developed in this thesis.

2.5 Summary

This chapter introduces key background material. We provide an introduction to work-stealing task parallelism, together with a detailed implementation oriented overview of X10, Habanero-Java and Java ForkJoin. We further discussed the concurrent object-oriented programming model, which provides necessary background for Chapter 6. This chapter also discusses managed runtime environments and Jikes RVM. This discussion covers managed runtime features that are extensively used in the following chapters.

Before we move to the heart of this thesis and its primary contributions, we first give an overview of our experimental methodology, in the next chapter.

Chapter 3

Experimental Methodology

This chapter describes the benchmarks, Java Virtual Machines, compilers, work-stealing implementations, operating system, hardware, and performance measurement methodologies used in this thesis. To maintain coherency in experimental methodology across different chapters, a common set of benchmarks, workloads, measurements and machine is especially chosen for this thesis. For this reason, this methodology differs slightly from that used in the published work [Kumar et al., 2012, 2014b,a].

3.1 Benchmarks

The following methodological choices prescribe the choice of benchmarks. (1) *Fine-grained task structures:* Coarser granularity of parallel tasks is not an effective approach for achieving load balancing in the face of large scale hardware parallelism. Due to this, the benchmarks used in this thesis are intentionally selected with fairly fine-grained task structures. This also helps analyzing and reducing sequential overheads of work-stealing. (2) *Higher steal frequency:* Dynamic overheads of work-stealing are more evident in benchmarks having higher steal frequency. This also governs the choice of some of the benchmarks used in this thesis. (3) *Large code base:* To evaluate productivity and performance of parallelism in real world problems, three of the benchmarks used in this thesis comprise of large codebases. All of the benchmarks are available at http://cs.anu.edu.au/ ~ vivek/phd-thesis/.

Fib A simple recursive computation of 40th Fibonacci number. This benchmark is a commonly used micro-benchmark for task scheduling overhead, as the problem is embarrassingly parallel and the amount of computation done within each task is trivial. The version used here is adopted from the X10 distribution [IBM Research, 2012].

Integrate Recursively calculate the area under a curve for the polynomial function $x^3 + x$ in the range 0 <= x <= 1000. This benchmark is similar in spirit to Fibonacci, but each task contains an order of magnitude more work. Adopted from the X10 distribution [IBM Research, 2012].

Jacobi Iterative mesh relaxation with barriers: 10 steps of nearest neighbor averaging on 1024 × 1024 matrices of doubles (based on an algorithm taken from the Java ForkJoin framework [Lea, 2000]).

NQueens The classic N-queens problem where 12 queens are placed on a 12 × 12 board such that no piece could move to take another in a single move (based on an algorithm from the Barcelona OpenMP Tasks Suite [Duran et al., 2009]).

Matmul Matrix multiplication of 1024 × 1024 matrices of doubles (based on an algorithm taken from the Java ForkJoin framework [Lea, 2000]).

FFT This is a Cooley-Tukey Fast Fourier Transform algorithm (adopted from Cilk [Frigo et al., 1998a]). Input size is 1024 × 1024.

CilkSort A divide and conquer variant of mergesort (adopted from Cilk [Frigo et al., 1998a]) for sorting 10 million integers. It begins by dividing an array of elements in half and sorting each half. It then merges the two sorted halves back together but in a divide-and-conquer approach.

Barnes-Hut A n-body algorithm to calculate gravitational forces acting on a galactic cluster of 100000 bodies (adopted from the lonestar benchmark suite [Kulkarni et al., 2009]).

UTS The unbalanced tree search benchmark designed by Olivier et al. [2007]. Tree type used is T2.

LUD Decomposition of 1024 × 1024 matrices of doubles (based on algorithm from Cilk [Frigo et al., 1998a]).

The following set of benchmarks are used in Chapter 6 for evaluating parallelism in real world problems. These benchmarks are adopted from SourceForge projects.

lusearch-fix This is from the Java DaCapo benchmark suite [Blackburn et al., 2006]. It uses Apache lucene [Apache Lucene, 2008] to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. The version of lusearch used here fixes a pathological allocation bug [Yang et al., 2011]. The study of lusearch-fix also includes the core packages of lucene version 2.4.1, which lusearch-fix depends upon. Lucene is an open source Java project, which provides Java-based indexing and search technology, as well as spell checking, hit highlighting and advanced analysis/tokenization capabilities. Lucene is a very widely used search library. The default implementation of lusearch-fix uses explicit Java threading for parallelism.

jMetal This stands for Metaheuristic Algorithms in Java, and it is an object-oriented Java-based framework aimed at multi-objective optimization with metaheuristic techniques [Nebro and Durillo, 2013; Durillo and Nebro, 2011; Durillo et al., 2010]. It provides several sets of classes which can be used as the building blocks of multi-objective techniques. The jMetal website shows more than

8000 downloads of this project to date. The default implementation is not entirely multi-threaded and offers parallelism for couple of their algorithms. The default implementation of jMetal uses `java.util.concurrent.Executer` and Java threads for parallelism.

JTransforms This is a multithreaded FFT library written in Java [Wendykier, 2011] and provides several types of FFT transformations. There are a few open source projects which use JTransforms internally. Currently the download counter for JTransforms is more than 9000. This library offers 8 benchmarks, which is used for the performance comparison. For parallelism the default implementation of JTransforms uses `java.util.concurrent.Future`.

3.1.1 Serial Elision

For each of the benchmarks described above, we also created a serial elision [Frigo, 2007] version. This is a version of the code with all the work-stealing keywords removed. A serial elision in our case is the vanilla Java code, which does not involve any code that expresses parallelism or synchronizations.

3.2 Software Platform

X10 Release 2.2.2.2, svn revision 23688. The latest revision is not used because the work-stealing implementation for the managed X10 (Java backend) is broken. Native X10 (C++ backend) runtime is build with flags -DOPTIMIZE=true -DNO_CHECKS=true -DDISABLE_GC=true. To build the managed X10 runtime, the flag -DDISABLE_GC is not used. Benchmarks are compiled with options -WORK_STEALING -O -NO_CHECKS. With native X10, benchmarks are linked with the thread-caching malloc memory allocator of the Google performance tools [Ghemawat and Menage, 2009]. This evaluation strategy is similar to that used by Tardieu et al. [2012].

ForkJoin Java ForkJoin work-stealing framework [Lea, 2000]. Version 1.7.0.

Habanero-Java Version 1.3.1. Benchmarks used in this thesis did not compile with the adaptive policy of Habanero-Java [Guo et al., 2010]. Hence, they were built with both work-first and help-first policies [Guo et al., 2009] but the execution time is reported for the policy, which performs best for a particular benchmark.

Jikes RVM Version 3.1.3 with the production build. The command line arguments used are: -Xms1024M -X:gc:variableSizeHeap=false -X:gc:threads=1.

OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode). The command line argument used is: -Xmx1024M.

JastAdd This is an implementation of an extensible compiler for Java [Ekman and Hedin, 2007]. The compiler consists of basic modules for Java 1.4, and extension modules for Java 1.5 and Java 7. There are several open source projects which use JastAdd internally (for example Soot [Vallée-Rai et al., 1999; Sable McGill, 2012]).

Operating System Ubuntu 12.04.3 LTS.

3.3 Hardware Platform

To maintain coherency across experiments, all the experiments were performed on a dual-socket machine with two Intel Xeon E5-2450 Sandy Bridge processors. Each processor has eight cores running at 2.10 GHz sharing a 20 MB L3 cache. The machine was configured with 47 GB of memory.

To ensure the robustness of the experimental results, we have also verified our findings on several other machines.

3.4 Measurements

For each benchmark, twenty invocations are executed, with fifteen iterations per invocation where each iteration performed the kernel of the benchmark. The final iterations are reported, along with a 95% confidence interval based on a Student t-test. For performance comparison on Jikes RVM, only the mutator time is reported. This is to eliminate garbage collection scalability as a factor in the experiments. For validating the results with OpenJDK, absolute execution time is used.

Chapter 4

Sequential Overheads of Work-Stealing

Recall from Section 2.2.2.2 (page 8) that work-stealing is a very efficiently strategy for distributing work in a parallel system. However, work-stealing scheduling incurs some form of sequential overheads as a necessary side effect of enabling dynamic task parallelism. This overhead is seen as the increase in single-core execution time due to transformations to expose parallelism. This chapter quantifies and analyzes the sources of the sequential overheads for a work-stealing scheduler. To reduce these overheads, two substantially more efficient designs are proposed and evaluated.

This chapter is structured as follows. Section 4.2 describes the motivation for this work. Section 4.3 discusses the approach taken towards reducing the sequential overheads of work-stealing. Section 4.4 describes the implementation of two new efficient work-stealing designs developed in this chapter. Section 4.6 provides an overview of various work-stealing implementations, which relates to the contributions in this chapter. Finally, Section 4.5 evaluates our novel work-stealing implementations.

This chapter describes work published as "Work-stealing Without the Baggage" [Kumar, Frampton, Blackburn, Grove, and Tardieu, 2012].

4.1 Introduction

This chapter identifies the key sources of sequential overheads in work-stealing schedulers and presents two significant refinements to their implementation. The two designs proposed in this chapter exploits the runtime mechanisms already available within managed runtimes, namely: a) the yieldpoint mechanism (Section 2.4.2.1, page 18), b) on-stack-replacement (Section 2.4.2.3, page 19), c) dynamic code-patching (Section 2.4.2.4, page 19), and d) support for exception delivery (Section 2.4.2.5, page 20). We use Jikes RVM (Section 2.4.3, page 21) infrastructure to implement our ideas and have empirically assessed them using both a language-based work stealing scheduler, X10 (Section 2.2.1.1, page 6), Habanero-Java (Section 2.2.1.2, page 7), and a library-based framework, Java ForkJoin (Section 2.3.6, page 17). The results are also validated with OpenJDK. Compared to serial elision (Section 3.1.1, page 25), the fastest design proposed in this chapter shows a sequential overhead of just 10%.

By contrast, X10 has 195% overhead, Habanero-Java has 448% overhead and Java ForkJoin has 124% overhead.

The principal contributions in this chapter are as follows: a) a detailed study of the sources of overhead in an existing work-stealing implementation; b) two new designs for work-stealing that leverage mechanisms that exist within modern JVMs; c) an evaluation using a set of benchmarks ported to run in X10, Habanero-Java, Java ForkJoin framework, and the serial elision; and d) performance results that shows that the sequential overhead from a work-stealing implementation can be almost removed. These contributions give further impetus to work-stealing as a model for effectively utilizing increasingly available hardware parallelism.

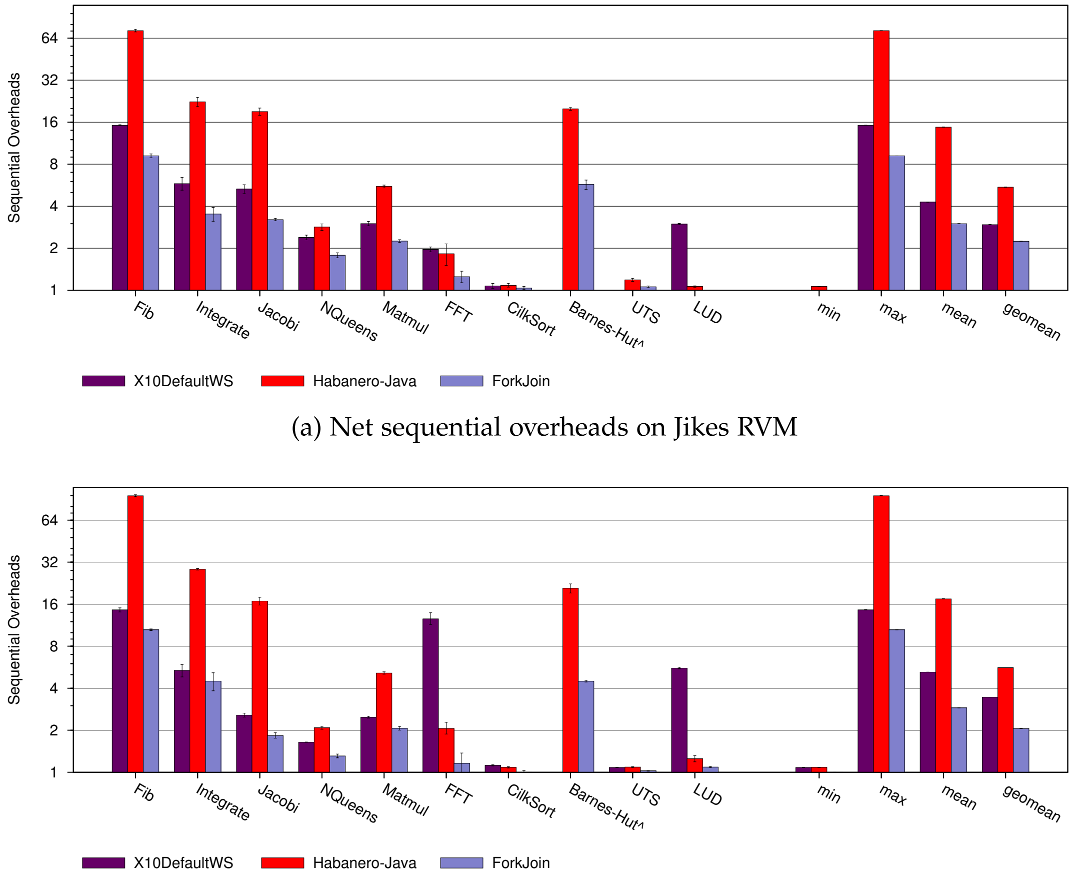
4.2 Motivating Analysis

Work-stealing is a very promising mechanism for exploiting software parallelism, but it can bring with it formidable overheads to the simple serial elision case (Section 3.1.1, page 25). These overheads make the task of the programmer challenging because they must use work-stealing judiciously so as to extract maximum parallelism without incurring crippling sequential overheads. To shed light on this and further motivate the new designs, we now: 1) identify and measure the *sequential overheads* imposed by existing work-stealing runtimes, and 2) measure the dynamic *steal ratio* across a range of well-known parallel benchmarks (Section 3.1, page 23), showing that unstolen tasks are by far the common case. The ten benchmarks used in this analysis are ported to X10, Habanero-Java, Java ForkJoin, and serial elision.

4.2.1 Sequential Overheads

In order to measure sequential overheads, we take three popular work-stealing runtimes (X10DefaultWS—default work-stealing implementation in managed X10, Habanero-Java, and Java ForkJoin) and execute each of them on Jikes RVM and OpenJDK with a *single worker*. No stealing can occur in this case, so the runtime support for stealing is entirely redundant to this set of experiments. This artificial situation allows us to selectively *leave out* aspects of the runtime support, providing an opportunity to analyze the overheads due to work-stealing in more detail. As a baseline we use the serial elision version of each benchmark and execute them both on Jikes RVM and OpenJDK. Figures 4.1(a) and 4.1(b) shows the result of this analysis. The results show that the work-stealing runtimes incurs serious sequential overheads. This overhead is nearly similar across both Jikes RVM and OpenJDK. The geomean on Jikes RVM is: 195% for X10DefaultWS, 448% for Habanero-Java and 124% for Java ForkJoin. On OpenJDK, it is: 245% for X10DefaultWS, 462% for Habanero-Java, and 106% for Java ForkJoin.

We have identified three major sources, which significantly contribute to this high sequential overhead. Two are closely related to the overheads identified in the Section 2.3 (page 11), namely initiation and state management. The final overheads relate to code restructuring and other changes necessary to support work-stealing.



(a) Net sequential overheads on Jikes RVM

(b) Net sequential overheads on OpenJDK

∧ We were unable to run Barnes-Hut on X10DefaultWS.

Figure 4.1: Sequential overheads in work-stealing runtimes. Y-axis value 1.0 represents the execution of serial elision and anything above that is an overhead. Benchmarks not having any sequential overheads do not appear in the figure.

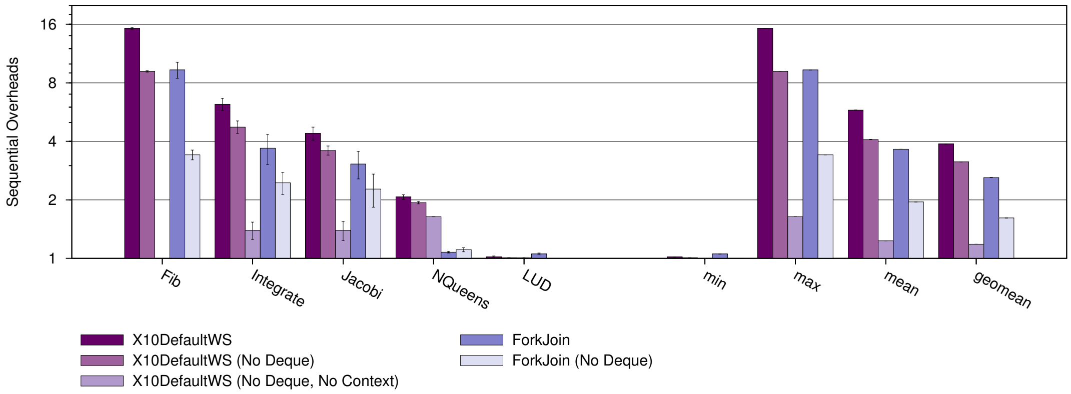


Figure 4.2: Work-stealing runtime consists of 3 different phase: initiation, state management and code-restructuring (termination). Initiation overhead in X10DefaultWS is the difference between X10DefaultWS and X10DefaultWS (No Deque), whereas in ForkJoin its the difference between ForkJoin and ForkJoin (No Deque). State management overhead in X10DefaultWS is the difference between X10DefaultWS (No Deque) and X10DefaultWS (No Deque, No Context). In ForkJoin, this is the value of ForkJoin (No Deque). Code-restructuring overhead is encountered only in X10 and this is represented by the value of X10DefaultWS (No Deque, No Context).

Initiation. The deque is an obvious source of overhead for the victim, which must use synchronized operations to perform work (even when nothing is stolen). This overhead may be a significant problem for programs with fine-grained concurrency. To measure this overhead, we took the X10DefaultWS and Java ForkJoin work-stealing runtimes and measured single worker performance with all deque operations removed. (Recall from Section 2.3 (page 11) that the deque manages pending work, so the strictly sequential case of a single worker, it is entirely redundant.) We did not include Habanero-Java in this analysis because the Habanero-Java compiler and the runtime are not open sourced. Also, to perform this analysis, we have to hand modify the generated code from X10DefaultWS compiler. This is a daunting task for complex benchmarks because of the heavy code transformations. For this reason, we use only 5 benchmarks in the following analysis. We have performed this experiment only on Jikes RVM. The results are shown as X10DefaultWS (No Deque) and ForkJoin (No Deque) in Figure 4.2. This figure only shows the values, which are greater than 1.0. These results show that the deque accounts for just over 20% and 40% of all sequential overheads for X10DefaultWS and ForkJoin respectively.

State Management. As discussed in Section 2.2.2.2 (page 8), work-stealing runtimes typically allocate state objects to allow sharing and movement of execution state between tasks. In pure Java, these objects must be heap allocated, leading to significant overheads. In addition to the direct cost of allocation and garbage collection, these objects may also be chained together, and may limit compiler optimizations. Figure 4.2 shows the overhead of allocating these state objects in the X10DefaultWS Java work-stealing runtime by removing their allocation in a system that already had the deques removed. In this case all values are passed on the stack, and no copying was required because only a single worker exists. This is shown as X10DefaultWS (No Deque, No Context) in Figure 4.2. We did not need to perform a similar experiment for ForkJoin, as it would reduce to the serial elision case (and thus would show zero overhead). We can see that the allocation of these state objects is a very significant cost, averaging just under half of the total overhead.

Code Restructuring. In order to support the stealing of tasks, the runtime must generate entrypoints with which the thief can resume execution. This is typically performed by splitting up methods for the different **`finish`** and **`async`**. This code restructuring accounts for part of the performance gap between X10DefaultWS (No Deque, No Context) and serial elision. In effect, this overhead includes all overheads due to X10-to-Java compilation, of which only a subset would be necessary to support work-stealing.

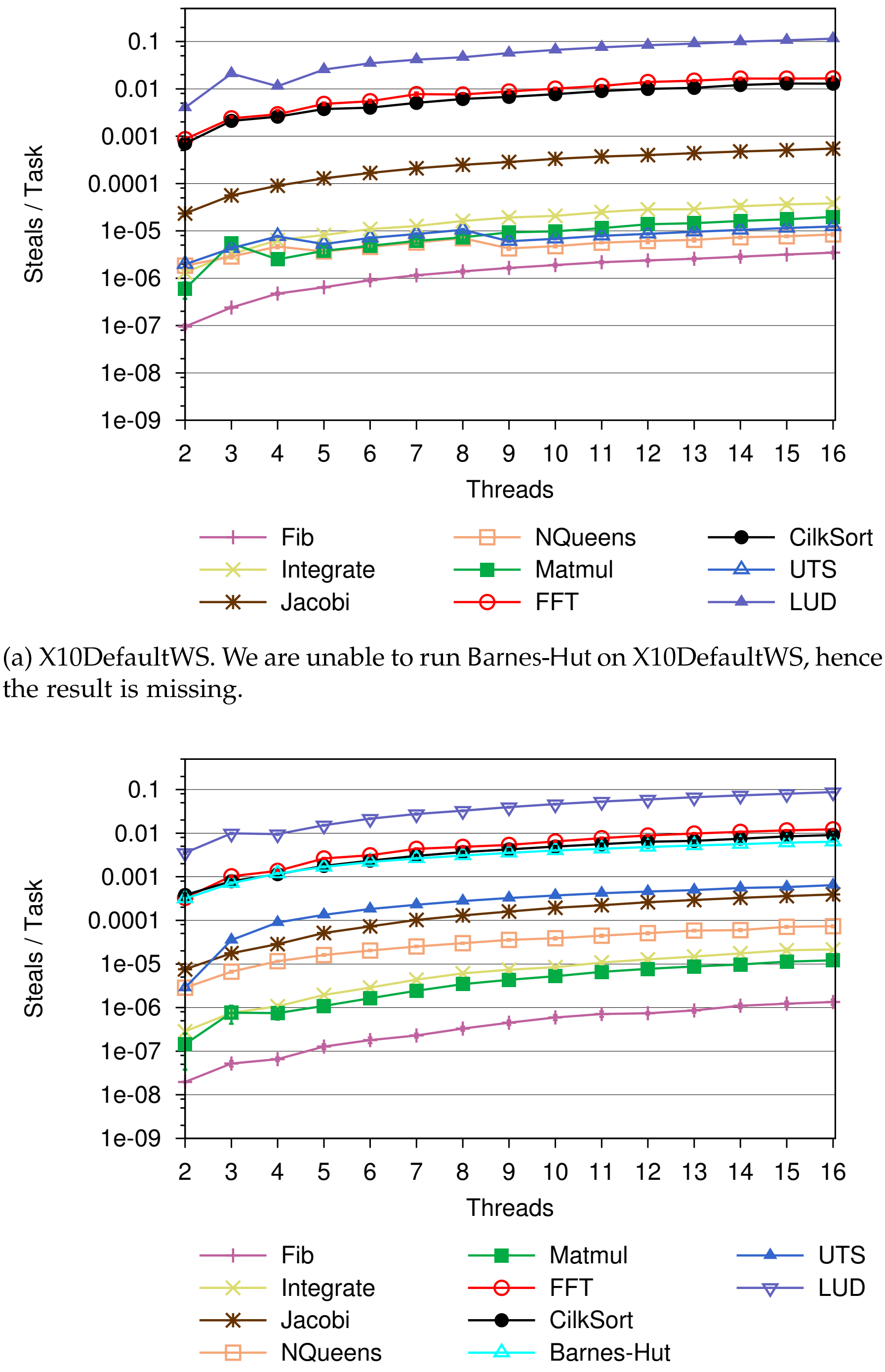
Habanero-Java was not included in this analysis, but similar to X10DefaultWS and ForkJoin, Habanero-Java too relies on: a) separate task deque per worker for initiation; b) heap allocation of program state for state management; and c) **`finish`**-**`async`** code restructuring similar to X10DefaultWS. This approach adds to the sequential overheads and is visible in Figure 4.1(a).

4.2.2 Steal Ratio

Work-stealing algorithms aim to ensure that sufficient tasks are created to keep all processors busy. In practice, however, much of this *potential* parallelism is not realized, due to other activities or a reduced availability of parallelism. Consequently it may be the case that most tasks are consumed locally. Clearly the number of stolen tasks is bounded by the total number of tasks, but the fraction of tasks actually stolen (the *steal ratio*) is an important component in determining if, and how, cost should be traded off between *all* tasks and *stolen* tasks.

We performed a study to understand the steal ratios across the range of our benchmarks. We instrumented X10DefaultWS and Java ForkJoin work-stealing runtimes to measure both the total number of tasks produced (executed **`async`** blocks in X10, and **`fork`**() calls in ForkJoin) as well as total number of tasks stolen. We show the measured ratio in Figure 4.3.

It is clear from the figure that stealing is generally uncommon and in many cases extremely rare. A single steal may move substantial work (consider divide-and-conquer algorithms). Because of this, relatively few steals may address a load imbalance. Although LUD has steal ratio that approaches one in ten, for all other benchmarks the ratio is between one in a hundred and one in a million. This result shows that steals are generally uncommon and suggests that eagerly preparing for a steal is likely to be an inefficient strategy.



(a) X10DefaultWS. We are unable to run Barnes-Hut on X10DefaultWS, hence the result is missing.

(b) ForkJoin

Figure 4.3: Steals to task ratio for each benchmark on different work-stealing runtimes. A low ratio is always preferred to ensure that sufficient tasks are created to keep all processors busy.

4.3 Approach

As discussed in Section 2.2.2.2 (page 8), for work-stealing to function correctly we must be able to 1) identify a task to be stolen, 2) provide sufficient context to allow a thief to execute a stolen task, and 3) reintegrate state due to computation performed by a thief back into the victim's execution context.

Each of these operations is only required for tasks that are *actually* stolen, and as we saw in Section 4.2.2, steal ratios for many programs are close to zero. The performance of *unstolen* tasks is therefore critical to overall performance. In this chapter, we try to push the limits as to what aspects of the above functions can be deferred until a steal occurs, moving them off the critical path of unstolen tasks. Our particular approach is to leverage advanced facilities that exist within the implementation of a modern managed runtime.

4.3.1 Scalability Concerns

A simple measure of the success of a parallelization construct is *scalability*. Of course one way to 'improve' scalability is to enhance the parallel case at the expense of the base sequential case. In practice, this is what happens with existing work-stealing frameworks, which involve substantial overheads in the sequential case. By corollary, our approach of moving overhead off the common sequential case (to be absorbed at steal time by the thief) will reduce the apparent scalability. In our evaluation we express scalability for all systems as speedup relative to the serial elision base case, sidestepping this pitfall by focusing instead on overall performance. Our argument is that scalability is a means to improved overall performance, not an end in and of itself. The question then becomes, is it possible to build a system that aggressively defers steal-related work, and what is the actual cost tradeoff of doing so.

4.3.2 Techniques

Our approach is based on several basic techniques, each described in more detail in the context of the implementations discussed in the following section.

1. We use the victim's execution stack as an *implicit* deque.

2. We modify the runtime to extract execution state directly from the victim's stack and registers.

3. We dynamically switch the victim to *slow* versions of code to reduce coordination overhead.

We are unaware of any previous work that uses either of the first two approaches, and due to the support of a managed runtime, we are able to perform the third more aggressively than prior work, and with reduced overhead in the common case.

4.4 Implementation

We have implemented and evaluated two work-stealing runtimes, X10OffStackWS: a modification of the default X10 work-stealing runtime for JVMs, and X10TryCatchWS: a simple runtime implementation on plain Java. Both implementations support the X10 **`finish`**-**`async`** model of execution. Our X10TryCatchWS runtime is targeted by the X10 compiler.

This section describes each of our work stealing runtime implementations in terms of the work-stealing requirements we enumerated in Section 2.3 (page 11): *initiation*, *state management*, and *termination*.

4.4.1 Runtime Supported X10OffStackWS

4.4.1.1 Initiation

One of the key insights behind the X10OffStackWS design is that we can avoid maintaining an explicit deque by using existing runtime mechanisms to extract the information from the worker's call stack. This approach eliminates the significant overhead of managing an explicit deque, but requires alternative mechanisms to synchronize the victim and thief, and to manage the set of stealable tasks.

Stack as an implicit deque. In our system the deque is *implicitly* stored within the worker's call stack. The X10 compiler transforms each X10 **`async`** body into a separate Java method (as it does normally). We attach **`@IsAsync`** annotations to these methods, and **`@HasContinuation`** annotations to all methods that call **`async`** (and thus have continuations). A stealable continuation is identified by a caller–callee pair with a caller marked **`@HasContinuation`** and a callee marked **`@IsAsync`**. The *tail* of the deque corresponds to the top of the worker's stack. Each worker maintains a `stealFrame` pointer, which points to the head of the deque and is managed as described below. The body of the deque is the set of all stealable frames between the top of the stack and the frame marked by `stealFrame`. Any worker thread with a non-null `stealFrame` field is a potential victim.

Victim–Thief handshake. Once a thief finds a potential victim, it uses the runtime's existing yieldpoint mechanism to force the victim to yield—the victim is stopped while the steal is performed. The yieldpoint mechanism is used extensively within the runtime to support key features, including exact garbage collection, biased locking, and adaptive optimization. Reusing this mechanism allows us to add the hooks to stop the victim without any additional overhead. Note that between the point at which a thief attempts a steal, and the point the victim–thief handshake begins, it is possible that a task may no longer be available to steal. We measured the frequency of such failed steal attempts in our evaluation at around 2-10%.

Maintaining `stealFrame`. Recall that `stealFrame` is the pointer to the head of the implicit deque. Workers and thieves maintain `stealFrame` cooperatively. When a worker starts executing a task, `stealFrame` is **`null`**, signifying that there is nothing available to steal. When a worker wants to add a task to its (implicit) deque, it first checks `stealFrame`. If `stealFrame` is **`null`**, then the implicit deque is empty so `stealFrame` is updated to point to the new task, which is now the head of the (implicit) deque. If `stealFrame` is already set, then the new task is not the head so `stealFrame` is left unmodified. `stealFrame` must also be updated as tasks are removed. A worker must clear `stealFrame` when it consumes the head. A thief updates `stealFrame` during a steal to either point to the next stealable continuation, or **`null`** if no other stealable continuation exists.

Ensuring atomicity. A worker detects that a continuation it expected to return to has been stolen by checking if `stealFrame` has been set to **`null`**. In this case there is no work left locally, and the worker becomes a thief and searches for other work to execute.

4.4.1.2 State Management

When a task is stolen, the thief must take with it sufficient state to run the task within the thief's own context. This includes all local variables used by the stolen task. In our running example, it is just the parameter `n`, which is used on line 9 of Figure 2.3(a). In the X10OffStackWS system, we perform lazy state extraction, extracting the state from the victim thread only when a steal occurs.

Extracting state off the stack. We extract state from the victim stack into the heap so that the thief may access the state while the victim continues to execute. Because the victim is stopped during a steal, we are trivially able to duplicate its complete execution state down to the steal point, including the stack and registers. At this point the victim may resume execution. The thief then extracts the state out of the duplicate stack for each stolen continuation. It does this by unwinding the duplicate stack whilst a `copyingStates` flag is set, which causes reflectively generated code to be executed for each frame. The reflective code captures local state for the frame and interns it in a linked list of heap objects, one object per frame. At this point all necessary victim state has been captured and the thief may commence execution.

The principal difference between our approach and the default X10 mechanism is that we perform state extraction *lazily*, only when an effective steal occurs. Compared to the X10 Java backend, our lazy approach avoids a large amount of heap allocation. The X10 C++ backend also has an approach which delays the allocation, but not the use of state objects, by initially allocating them on the stack and only lazily moving them to the heap when a steal occurs [Tardieu et al., 2012]. In the common case our approach avoids state extraction and allocation altogether.

Executing stolen tasks. Once state has been extracted, the thief executes against its heap-interned duplicate stack. The thief executes specially generated 'slow' versions of the continuations, which access the heap rather than the call stack for local variables. This is essentially identical to the X10DefaultWS implementation.

4.4.1.3 Termination

Control must only return from a **`finish`** context when all tasks within the context have terminated. To support this, each thread has a singly linked list with a node for each **`finish`** context that the thread is executing. Each dynamic **`finish`** context has a unique node shared by all threads running in that context. These nodes form a tree structure, with a root node for the **`finish`** context that represents the entire program. In X10OffStackWS, four important pieces of information are saved at each finish node:

- A linked list of stolen states.
- Frame pointers that identify where it was stolen from the victim, and where it is now running in the thief.
- A synchronized count of the number of active workers (initially 2 for the thief and victim).
- An object for storing partial results.

To ensure termination, when each thread leaves the **`finish`** context they decrement the synchronized count. The thread that drops the count to zero is responsible for executing the continuation of the **`finish`** context. The nodes are also used as the point for communicating any data that is required to be made available after the **`finish`**.

4.4.2 X10TryCatchWS Java implementation

Our X10TryCatchWS implementation is more radical. Our principal goal was to understand just how far we could reduce sequential overheads. To do this we started with plain Java and built a basic work-stealing framework upon it. We have modified the X10 compiler to compile to X10TryCatchWS. Thus X10TryCatchWS represents a new backend for work-stealing. Because we express X10TryCatchWS directly in plain Java code, it is straightforward to make direct comparisons with Java ForkJoin and serial elision.

4.4.2.1 Leveraging Exception Handling Support

In Java, the programmer may wrap sections of code in **`try`** blocks, and provide **`catch`** blocks to handle particular types of runtime exceptions. When an exception is thrown within the context of a **`try`** block for which there is a **`catch`** block that matches the exception's type, control is transferred to the start of the **`catch`** block. Exceptions propagate up the call stack until a matching **`catch`** block is found, or if no matching block is found the thread is terminated. To support exception handling, the runtime must maintain a table with entries that map the instruction address range of a **`try`** block to the instruction address for the **`catch`** block, annotated by the type of exception that can be handled.

Exception handling is designed to allow for the graceful handling of errors. Because exceptions are important and potentially expensive, JVM implementers have invested heavily in optimizing the mechanisms. Our insight is to leverage these optimized mechanisms to efficiently implement the peculiar control flow requirements of work-stealing. We can avoid much of the expense generally associated with exception handling as we never generate a user-level stack trace; we do not require this trace for work-stealing (we only need the VM-level stack walk).

Our X10TryCatchWS system annotates **`async`** and **`finish`** blocks by wrapping them with **`try`**/**`catch`** blocks with special work-stealing exceptions as shown in Figure 4.4. We can then leverage the exception handling support within the runtime and runtime compilers to generate exception table entries to support work stealing. These allow the X10TryCatchWS runtime to walk the stack and identify all **`async`** and **`finish`** contexts within which a thread is executing. The role of each exception type, and how the information is used in the runtime are described in more detail over the following sections.

4.4.2.2 Initiation

As in the X10OffStackWS implementation, X10TryCatchWS avoids maintaining an explicit deque. The key difference between the implementations is that in X10TryCatchWS we use marker **`try`**/**`catch`** blocks, not method annotations, to communicate the current deque state to the work-stealing runtime. Instead of identifying a continuation by a pair of methods using a frame pointer, we use a combination of the frame pointer and the offset of a specific **`catch`** block. We use the same thief-initiated handshake for synchronization between the victim and the thief.

Stack as an implicit deque. X10TryCatchWS maintains a *steal flag* for each worker thread that indicates that its deque may have work available to steal. The steal flag is set as the first action within an **`async`** (see line 6 in Figure 4.4). The steal flag is cleared when the worker or a thief determines that there is no more work to steal. As with X10OffStackWS, the tail of the task deque corresponds to the top of the call stack. The list of continuations (from newest to oldest) is established by walking the set of **`catch`** blocks that wrap the current execution state. We walk this list by running a modified version of the runtime's exception delivery code, searching for **`catch`** blocks that handle `WS.Continuation` exceptions. As we find entries, we simulate advancing into the **`catch`** block and repeat the search for exception handlers again, finding successively older continuations. Each worker has a `stealToken` that acts as a *head* for the deque. The `stealToken` indicates the point at which the continuation

```
int fib(n) {
  if (n < 2) return 1;
  int a,b;
  try {
    try {
      WS.setFlag();
      a = fib(n-1);
      WS.join();
    } catch (WS.JoinFirst j) {
      j.addFinishData(0, a);
      WS.completeStolen();
    } catch (WS.Continuation c) {
      // entry point for thief
    }
    b = fib(n-2);
    WS.finish();
  } catch (WS.FinishFirst f) {
    f.addFinishData(1, b);
    WS.completeFinish();
  } catch (WS.Finish f) {
    for(WS.FinishData fd: f.data) {
      if (f.key == 0) a = fd.value;
      if (f.key == 1) b = fd.value;
    }
  }
  return a + b;
}
```

Figure 4.4: Pseudocode for the transformation of `fib(n)` from X10 (Figure 2.3(a), page 12) into X10TryCatchWS. We have modified the default X10 compiler to automatically generate the code to support X10TryCatchWS.

of that worker has been stolen. Any continuations discovered after that point do not belong to that worker, and must therefore not be stolen from it.

Ensuring atomicity. Atomicity is guaranteed through the use of the `stealToken`, which acts as a *roadblock* for the worker and thieves to prevent either running or stealing continuations past that point. We saw above how the `stealToken` is used during the steal operation. To ensure that the victim does not run the continuation again, each **`async`** ends with a call to `WS.`**`join`**`()`. This call is responsible for checking whether the continuation has been stolen. This requires checking whether the frame pointer and **`catch`** block offset for the `stealToken` match the *innermost* continuation for that **`async`**, which is discovered using modified exception delivery code. When a steal is performed, the thief must also *steal* the `stealToken` from the victim thread, and place a new `stealToken` on the victim to prevent it from executing the stolen continuation.

4.4.2.3 State Management

In X10TryCatchWS, state is acquired by the thief through duplicating the entire execution state of the victim thread, including the stack and register state. Unlike in X10OffStackWS, the state is not extracted to the heap, but is used directly. No additional resume method is required for the entrypoint; execution can be transferred to the appropriate continuation point by delivering a `WS.Continuation` exception. The exception delivery code must be slightly different, because the exception must be delivered to the correct handler (it is not always the innermost exception handler for `WS.Continuation` that is correct).

Merging local variable state. While providing the correct state to *start* the continuation is made easy, X10TryCatchWS does not have a *resume* mode to fall back on where local variable state is all stored on the heap. This complicates merging the results of each of the tasks because the system must merge the local variables held by multiple copies of the same frame. In the Fibonacci example, the value of `a` is set within the **`async`**, while the value of `b` is set in the continuation. After the finish, both `a` and `b` must be set to ensure that the correct value is returned. At the end of each **`async`** or **`finish`** block there is a call to a runtime support method (`WS.`**`join`**`()` or `WS.`**`finish`**`()`). When these methods are called, two conditions are checked: 1) a steal has occurred within the appropriate finish block, and 2) the programmer has defined a **`catch`** block (`WS.Join` or `WS.FinishFirst` respectively) to save results. If both conditions are met, control is returned to the **`catch`** block by throwing an exception, allowing the code to provide local variable values with calls to `addFinishData(key, value)`. Each `key` represents a local variable: in our example `key` 0 maps to `a` and 1 maps to `b`. The last task to finish executing within the finish can then access all of these provided values, ensuring that all results are set correctly.

4.4.2.4 Termination

Termination is handled in a similar way to X10OffStackWS. A node is lazily created for each **finish** context in which a task is stolen. This node maintains an atomic count of the number of active tasks in the **finish** context, and provides a location for local variable state to be passed between threads, as described in the previous section. When a thread decrements the atomic count to zero, it becomes responsible for running the continuation of the **finish** context. The X10TryCatchWS runtime will deliver a WS.Finish exception at the appropriate point, allowing the thread to extract local variable state and continue out from the **finish**. This may also update the thread's stealToken, if the last thread to finish execution was not the thread running the end of the **finish** body. When this occurs, that thread runs the body of any WS.FinishFirst handler to communicate local variables, *deposits* its stealToken in the finish node, and searches for other work to complete.

4.4.2.5 Optimizing Runtime Support Calls

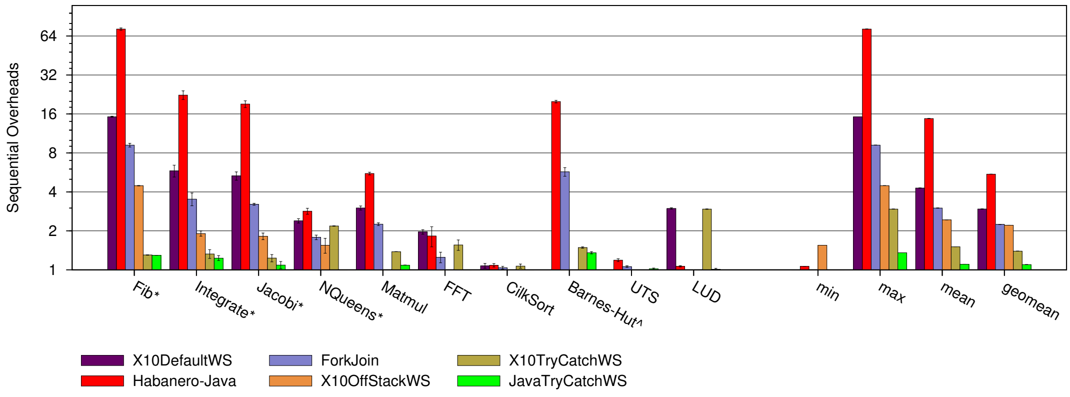
Within the X10TryCatchWS runtime, there are many calls to various WS methods. In the common case where no steal has occurred, only the call to WS.setFlag() needs to be executed. The call to WS.**join**() is only required if the continuation for the enclosing **async** has been stolen. Similarly, the call to WS.**finish**() is only required if at least one steal has occurred within that **finish** context. To avoid sequential overhead in the common case, we generate special *fast* versions of methods with these calls, where compiled code for the calls to these methods are overwritten by NOP instructions. This makes it simple for us to transfer execution between these methods as required, without requiring any additional exception handling tables for the fast version of the code. Both fast and slow versions of the code always make calls to fast versions. We also force calls to WS.setFlag() to be inlined by the optimizing compiler, which on our Intel target reduces to a simple store instruction.

Excluding indirect changes in compilation due to the presence of the **try**/**catch** blocks, the only sequential overhead in X10TryCatchWS is the execution of WS.setFlag(), and some additional NOP instructions.

4.5 Results

We start by measuring the sequential overhead of each of the systems before evaluating overall performance, including speedup. We then examine the effect of the different approaches on memory management overheads. We finish by measuring steal ratios and failed steal attempts for each benchmark using the modified systems.

For four of the ten benchmarks we manually generated the code to target the X10OffStackWS runtime (we did not implement automatic codegen support for X10OffStackWS). The remaining six benchmarks are significantly more complicated and we did not perform the manual code translation required for X10OffStackWS.



∗ We have run only these benchmarks on X10OffStackWS.
∧ We were unable to run Barnes-Hut on X10DefaultWS.

Figure 4.5: Sequential overheads in work-stealing runtimes. Y-axis value 1.0 represents the execution of serial elision and anything above that is an overhead. Benchmarks not having any sequential overheads do not appear in the figure.

Also, we don't have results of Barnes-Hut over X10DefaultWS because its execution failed.

Almost all of the benchmarks make extensive use of arrays. While the Habanero-Java, ForkJoin and serial elision versions of the benchmarks use Java arrays directly, the X10 compiler is not currently able to optimize X10 array operations directly into Java array operations, but does so through a wrapper with get/set routines. To understand the significance of this overhead, we also measure a system that we call JavaTryCatchWS, which uses try-catch work-stealing but operates directly on Java arrays without X10. As mentioned in Section 3.4 (page 26), for performance comparison of different runtimes on Jikes RVM, we report the mutator time only.

4.5.1 Sequential Overhead

Our primary focus in this chapter is the reduction of sequential overheads as a means of improving overall throughput. Using the same methodology as in Section 4.2.1, we restrict the work-stealing runtimes so that they only use a single worker thread and then compare their performance to the serial elision version of the program.

Figure 4.5 shows the sequential overhead of the X10DefaultWS, Habanero-Java, ForkJoin, our two optimized implementations (X10OffStackWS and X10TryCatchWS), and the JavaTryCatchWS system that uses regular Java arrays.

For Fib the sequential overheads for X10DefaultWS, Habanero-Java and ForkJoin are as high as 15×, 72× and 9× respectively. On average X10OffStackWS eliminates more than half of the sequential overheads of X10DefaultWS and performs slightly better than ForkJoin but significantly better than Habanero-Java. The X10TryCatchWS implementation has the sequential overhead of 40% (geomean). However, this also includes the overhead of X10 array accesses. The JavaTryCatchWS implementation, has consistently low sequential overheads across all benchmarks (geomean as just 10%).

4.5.2 Work Stealing Performance

Figure 4.6 shows the speedup relative to serial elision for each of the benchmarks and runtimes on our 16 core machine. Note that we only measure four of ten benchmarks for X10OffStackWS (because no automatic codegen support for X10OffStackWS) and do not measure Barnes-Hut for X10DefaultWS. These results clearly illustrate that the sequential overheads of work-stealing are the dominant factor in overall program performance. The results for the JavaTryCatchWS runtime are extremely promising. Even in extreme examples of fine-grained concurrency like Fib and Integrate it is able to outperform the serial elision of the program at 2 cores and deliver a significant speedup at 16 cores (10× and 8.5× respectively). Despite exhibiting excellent scalability, neither X10DefaultWS, Habanero-Java or ForkJoin are able to overcome their larger sequential overheads and show significant performance improvements over the serial elision code for Fib or Integrate even when using all 16 cores. The differences between the runtimes are less dramatic on the other eight benchmarks, but the overall trend holds [1]. All three runtimes (X10DefaultWS, Habanero-Java and ForkJoin) show reasonable levels of scalability, but the lower sequential overheads of JavaTryCatchWS and X10OffStackWS result in better overall performance.

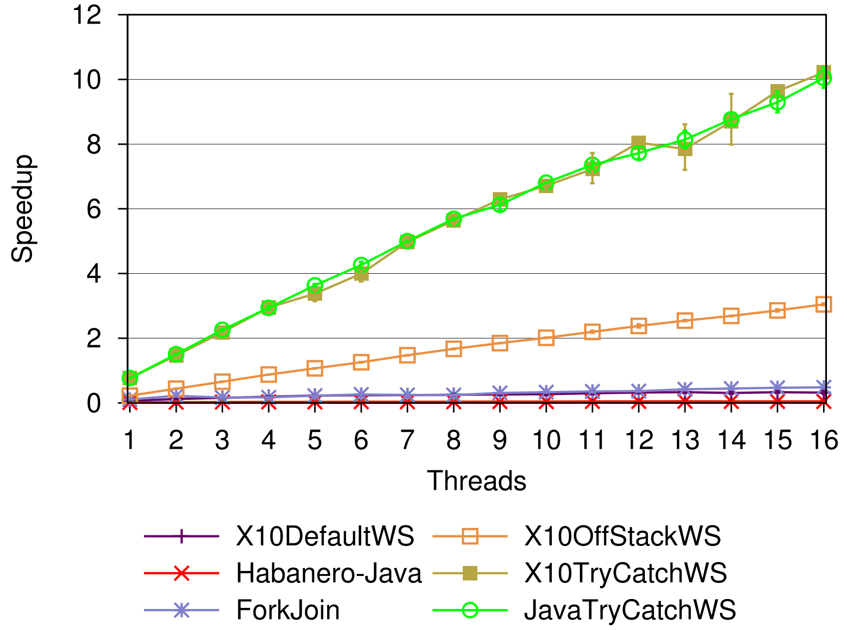
To increase the confidence in our results, we also compared overall performance with NativeX10WS (X10 with C++ backend), Habanero-Java, X10DefaultWS and ForkJoin running on OpenJDK. Figure 4.7 shows the result of this experiment. Here we show the speedup over the absolute execution time (not just mutator time) of single threaded JavaTryCatchWS. In most cases, the running time for JavaTryCatchWS is very competitive, particularly as the number of threads is increased [2].

4.5.3 Memory Management Overheads

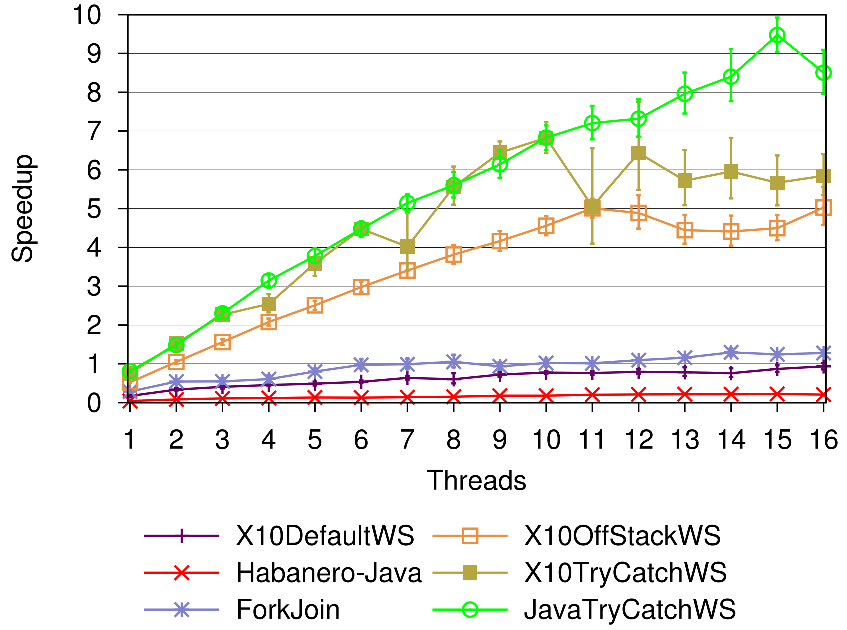
A significant source of performance improvement is due to the fact that X10OffStackWS dramatically reduces the number of heap-allocated frame objects, and JavaTryCatchWS

[1] In Figure 4.6(i), X10TryCatchWS performs better than JavaTryCatchWS. This is true even for single worker thread, where X10TryCatchWS is around 62% faster than JavaTryCatchWS. We have found that on Jikes RVM the serial elision version of UTS implemented in X10, is around 60% faster (X10 on Java backend) than the serial elision implemented directly in Java. However, we did not notice this on OpenJDK where X10 performed similar to Java. This suggests some pathology in the underlying VM, which is independent of our work stealing implementations. In Figure 4.6(j), Habanero-Java outperforms JavaTryCatchWS, whereas ForkJoin performs nearly identical to JavaTryCatchWS. Figure 4.3(a) shows that LUD has a high steal ratio (10% at 16 cores). From Figure 4.5 we can also see that LUD has almost zero sequential overhead in Habanero-Java and ForkJoin. This situation, where the sequential overhead is already zero and major percentage of tasks are stolen, would not benefit significantly from our approach. But we can see that JavaTryCatchWS is still competitive.

[2] A notable exception is LUD where the JavaTryCatchWS implementation is significantly slower than Habanero-Java and ForkJoin. The Jikes RVM results in Figure 4.6(j) show that this slowdown affects all Jikes RVM configurations, not just JavaTryCatchWS, suggesting pathology in the underlying VM, which is independent of our work stealing implementations.

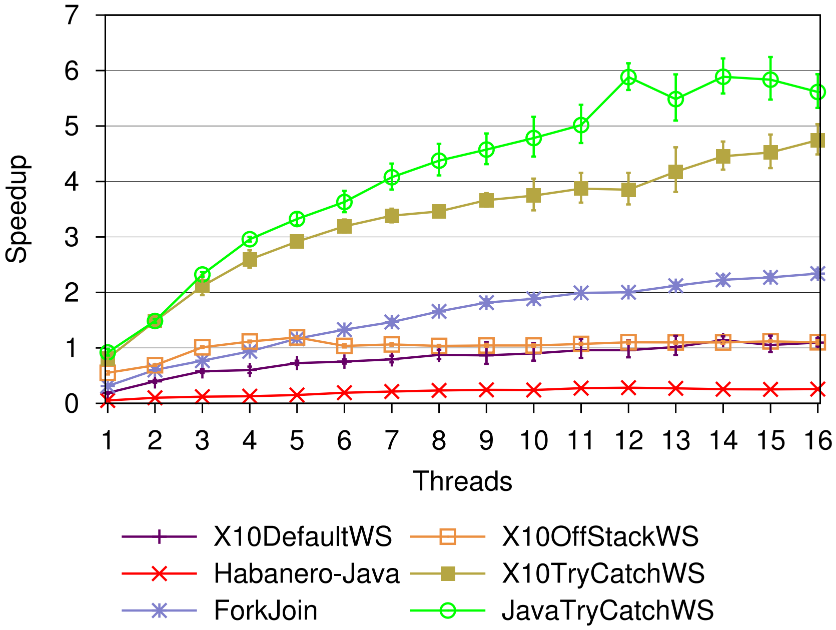


(a) Fib

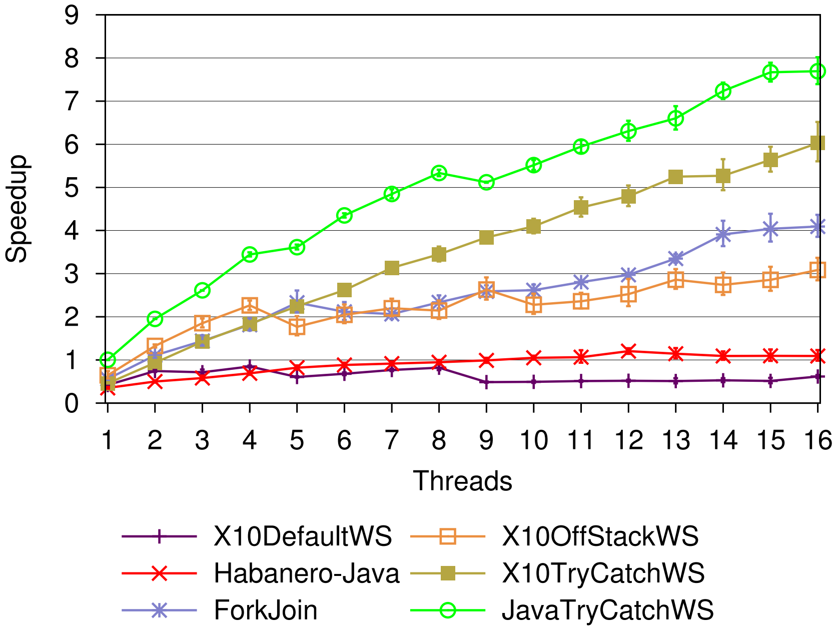


(b) Integrate

Figure 4.6: (Cont.) Speedup relative to serial elision version of each benchmark on Jikes RVM. Y-axis value 1.0 denotes the execution of serial elision. Y-axis values greater than 1.0 represents faster execution than the corresponding serial elision version.

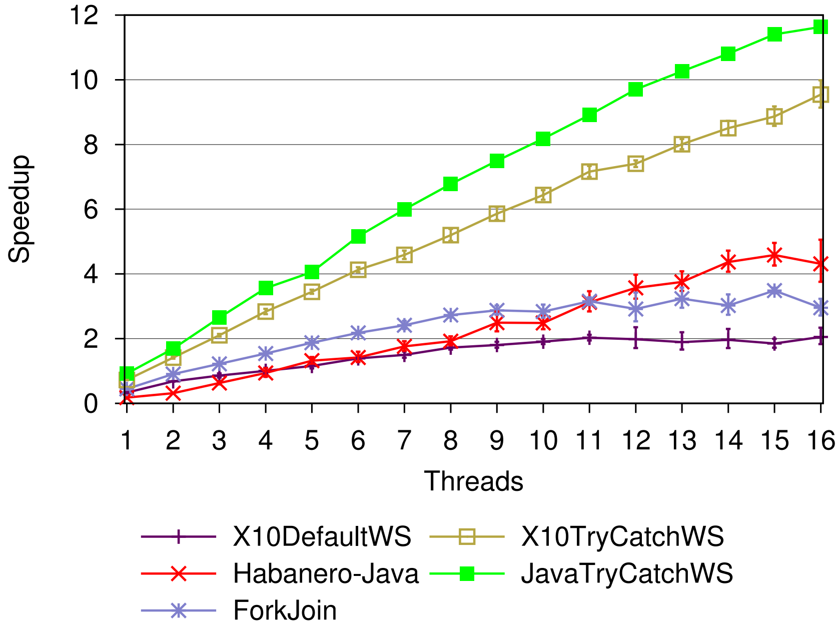


(c) Jacobi

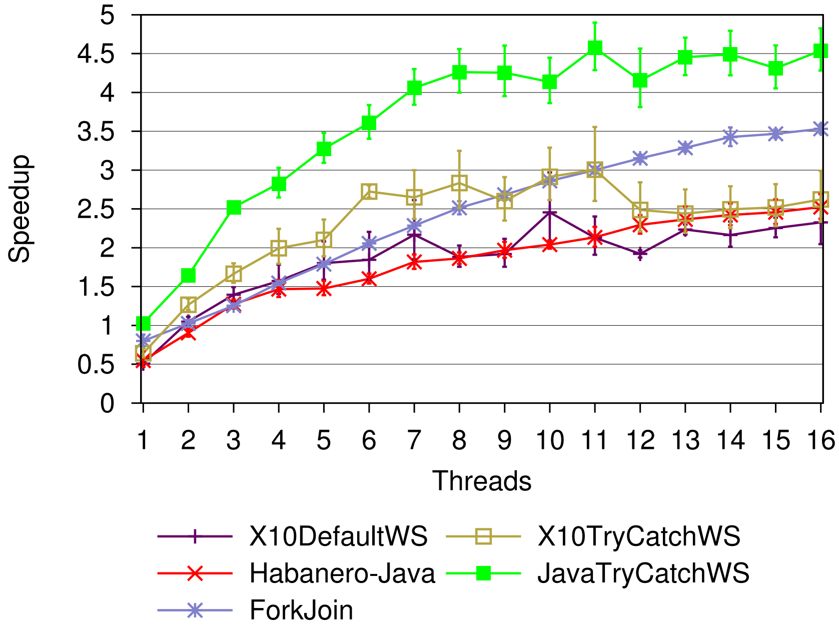


(d) NQueens

Figure 4.6: (Cont.)

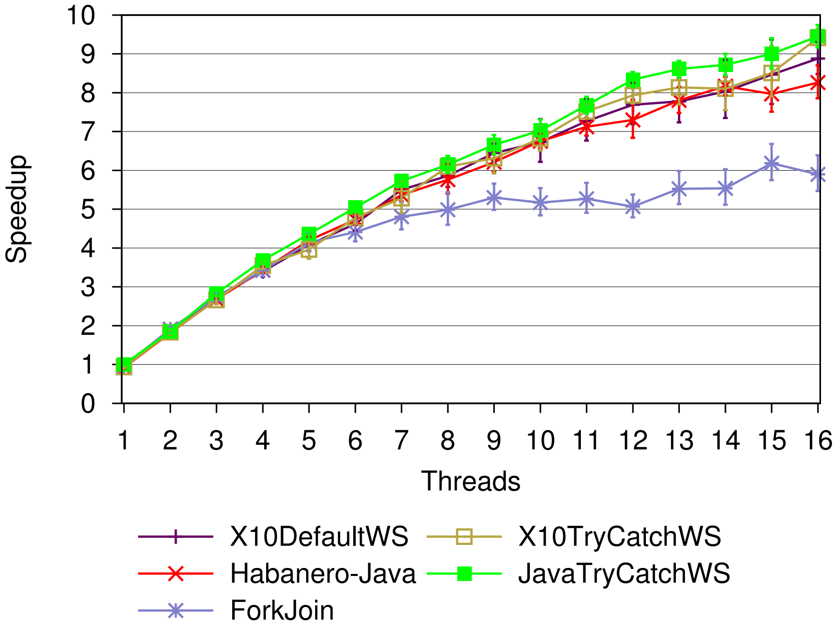


(e) Matmul

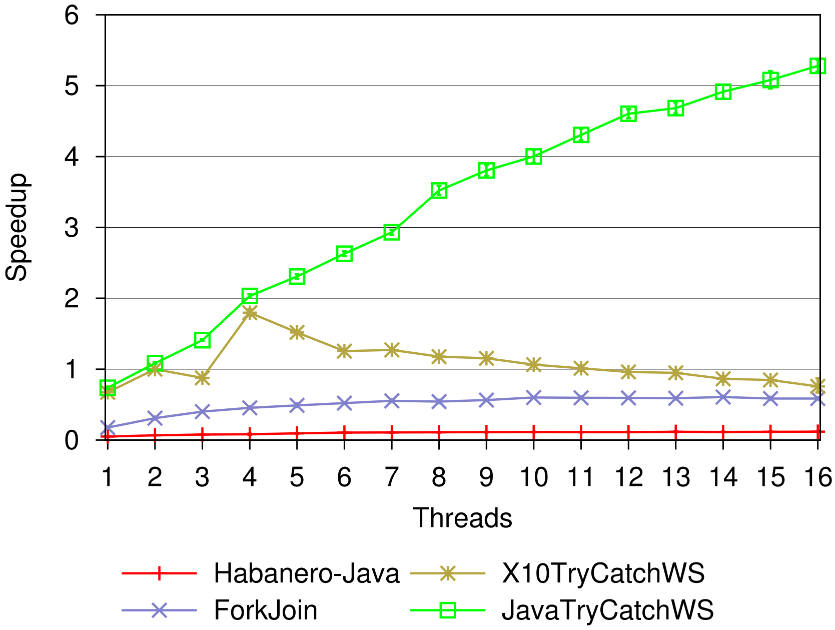


(f) FFT

Figure 4.6: (Cont.)

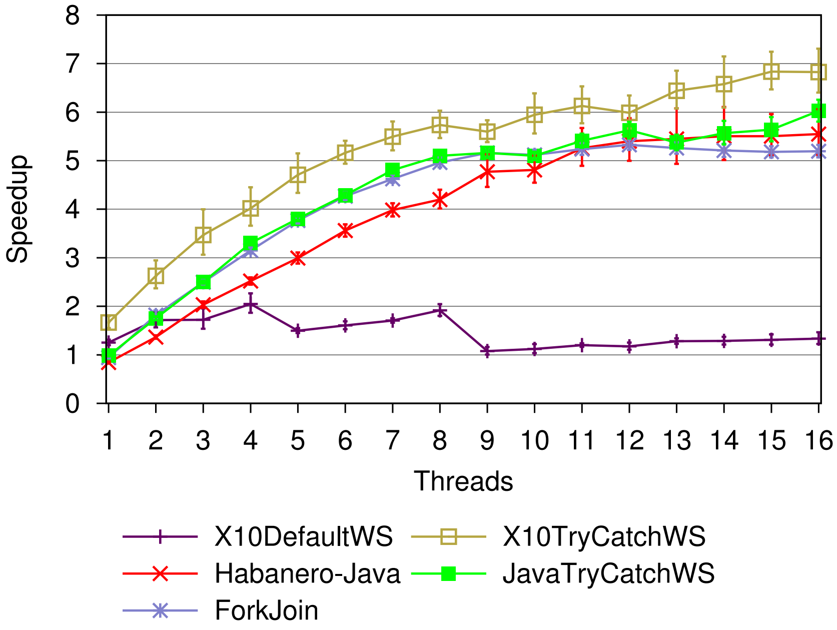


(g) CilkSort

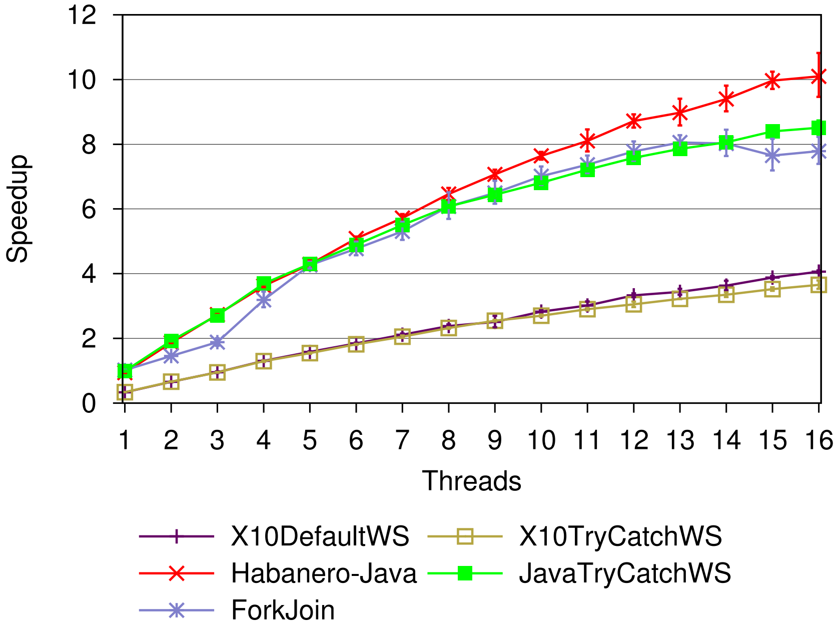


(h) Barnes-Hut

Figure 4.6: (Cont.)

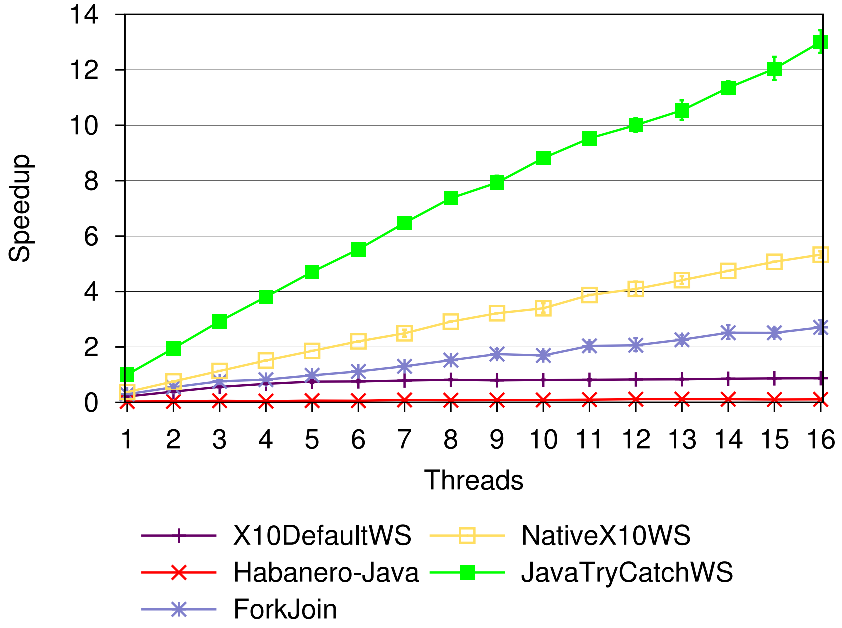


(i) UTS

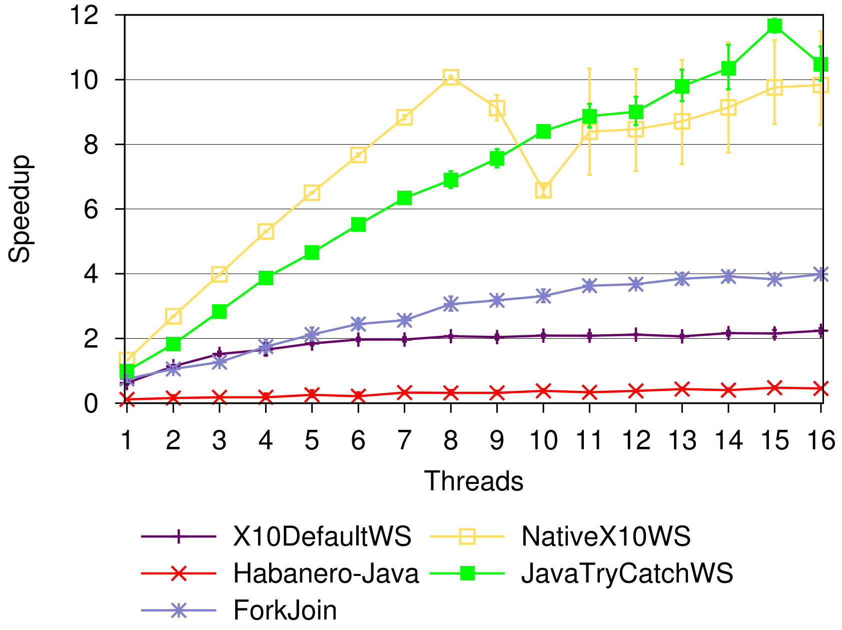


(j) LUD

Figure 4.6: (Cont.)

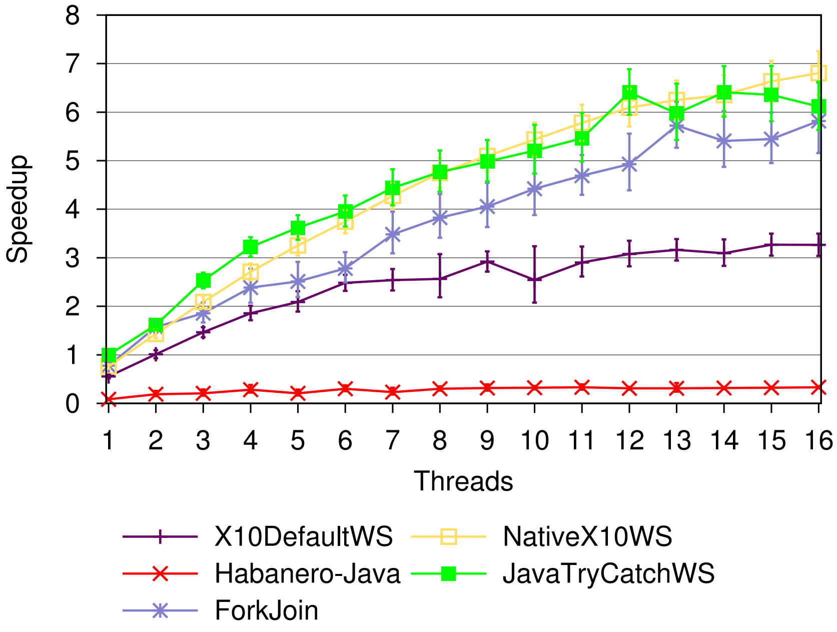


(a) Fib

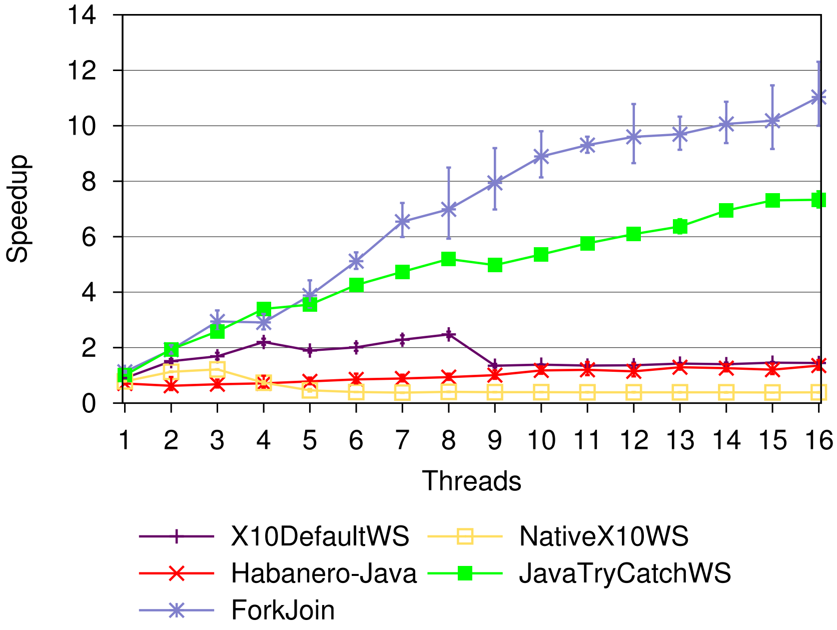


(b) Integrate

Figure 4.7: (Cont.) Speedup obtained by dividing the execution time of single thread JavaTryCatchWS work-stealing framework with the execution time of different runtimes for threads 1 to 16. Y-axis values greater than 1.0 shows better speedup over single threaded JavaTryCatchWS. JavaTryCatchWS runs over Jikes RVM; X10DefaultWS, Habanero-Java and ForkJoin is executed over OpenJDK; whereas NativeX10WS is the X10 running over it's C++ backend.

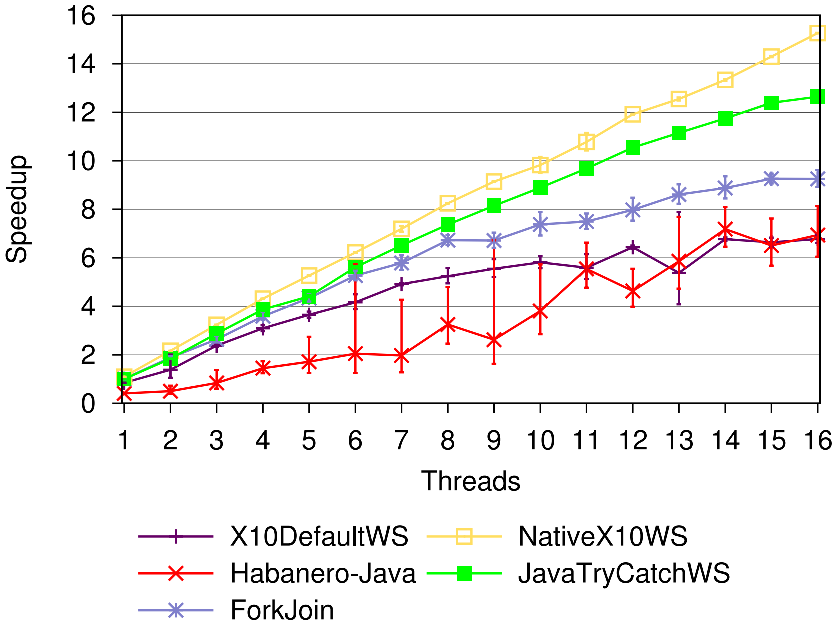


(c) Jacobi

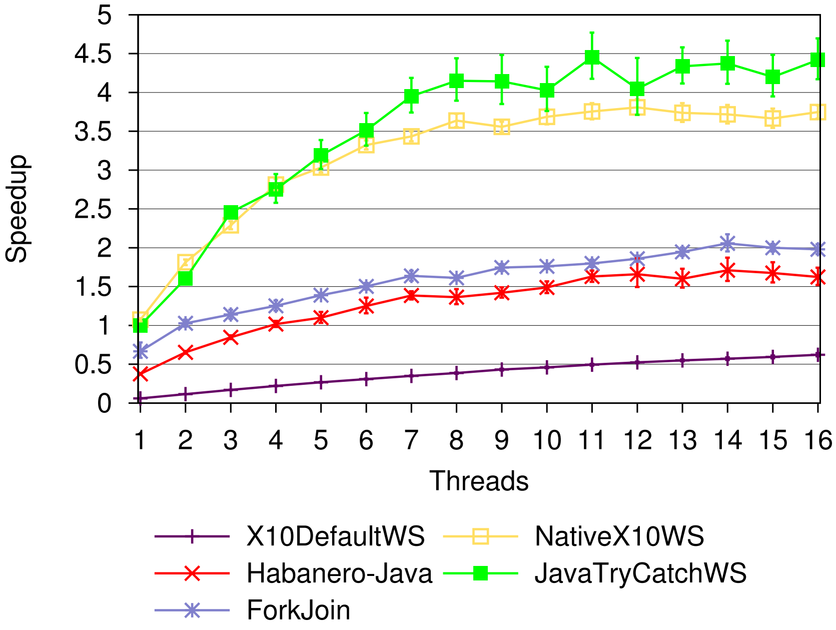


(d) NQueens

Figure 4.7: (Cont.)

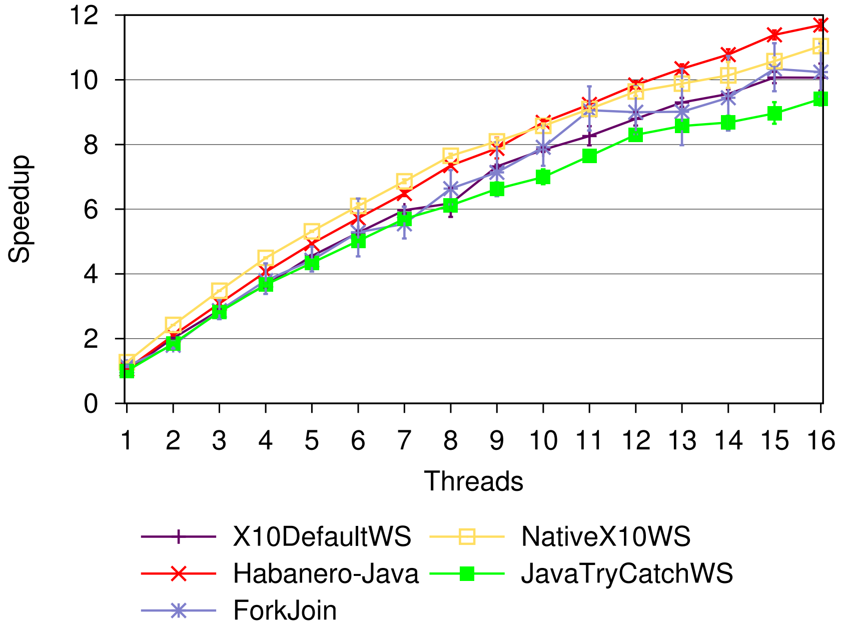


(e) Matmul

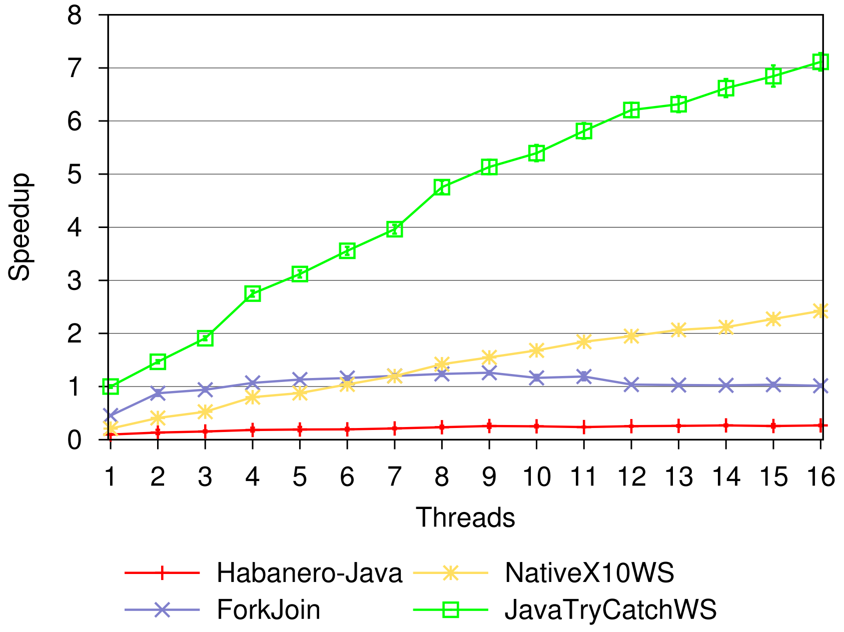


(f) FFT

Figure 4.7: (Cont.)

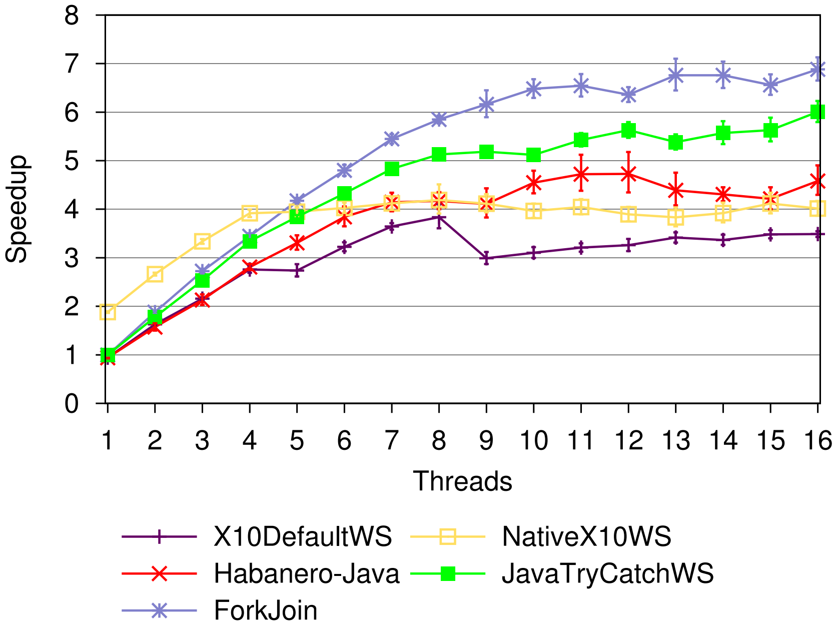


(g) CilkSort

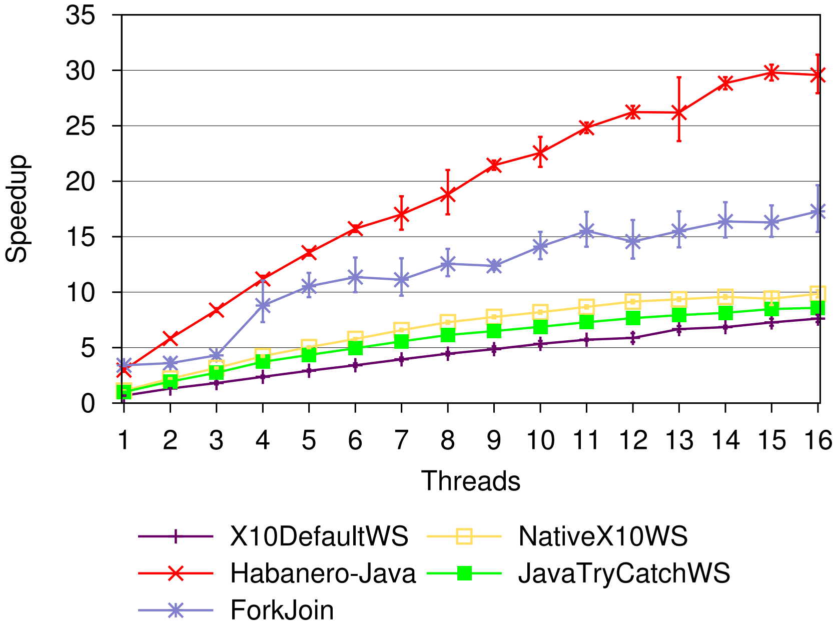


(h) Barnes-Hut

Figure 4.7: (Cont.)

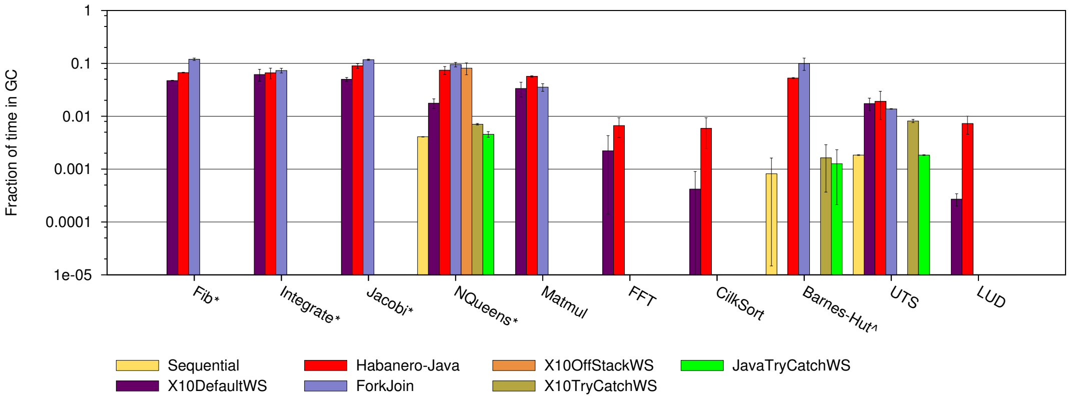


(i) UTS



(j) LUD

Figure 4.7: (Cont.)



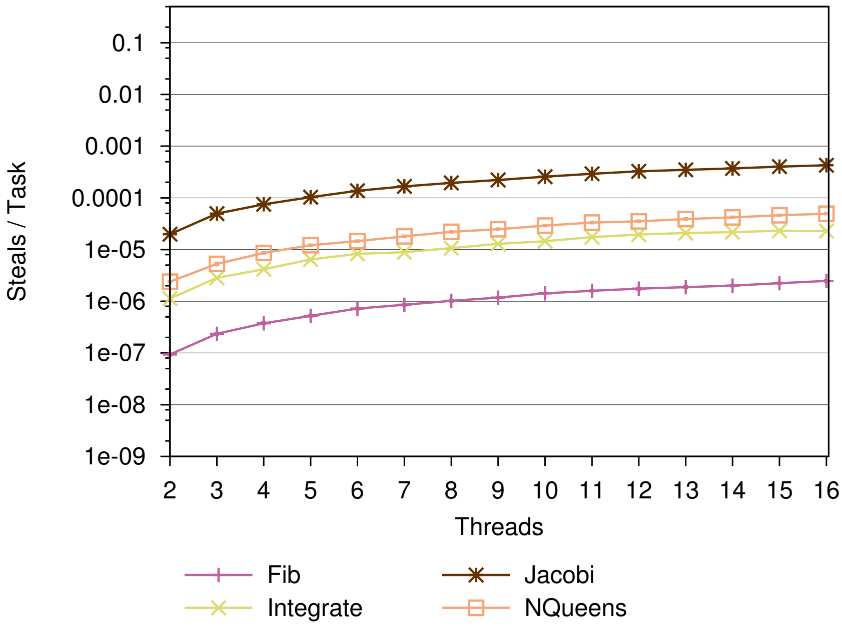
∗ We have run only these benchmarks on X10OffStackWS.
∧ We were unable to run Barnes-Hut on X10DefaultWS.

Figure 4.8: Fraction of time spent performing garbage collection work in different benchmarks. Memory management overhead in JavaTryCatchWS is very similar to that required in serial elision version. X10TryCatchWS and X10OffStackWS operates on X10 arrays rather than Java arrays. This results in relatively extra memory management overhead in these two runtimes as compared to JavaTryCatchWS.

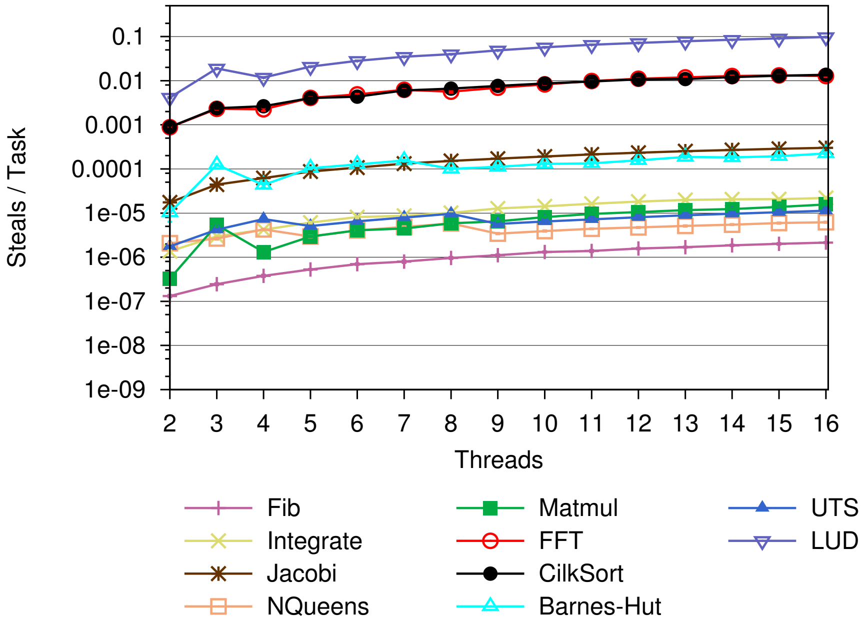
removes them altogether. Figure 4.8 shows the fraction of time spent performing garbage collection for each of the systems measured. As expected, X10OffStackWS and JavaTryCatchWS have significantly lower memory management overheads than the other work-stealing runtimes. There is still measurable time spent in garbage collection for the NQueens, UTS and Barnes-Hut benchmarks, but this is the case even for the serial elision versions of these benchmarks. Across all programs the garbage collection fraction is less than 12%. Note that the garbage collection fraction does not include the potentially significant cost of object allocation during application execution. To ensure our performance improvements were not due to poor collector performance in Jikes RVM, we also measured the Java based systems on OpenJDK, and saw that the collection time fraction was similar, and we know from previous work [Yang et al., 2011] that the allocation performance of Jikes RVM is highly competitive.

4.5.4 Steal Ratios

To ensure that our modifications did not dramatically affect behavior, we also measured the steal ratios for our optimized systems. The results in Figure 4.9(a) for X10OffStackWS do not differ significantly to those for the original system in Figure 4.3(a). The steal ratio for X10TryCatchWS is between the steal ratios for the other two systems and is shown in Figure 4.9(b). We also measured the frequency at which steal attempts failed (due to either another thief or the victim winning the race to start that continuation). Figure 4.10(b) and Figure 4.10(a) show the failed steal attempts for JavaTryCatchWS and X10OffStackWS respectively. In general, the fraction of steal

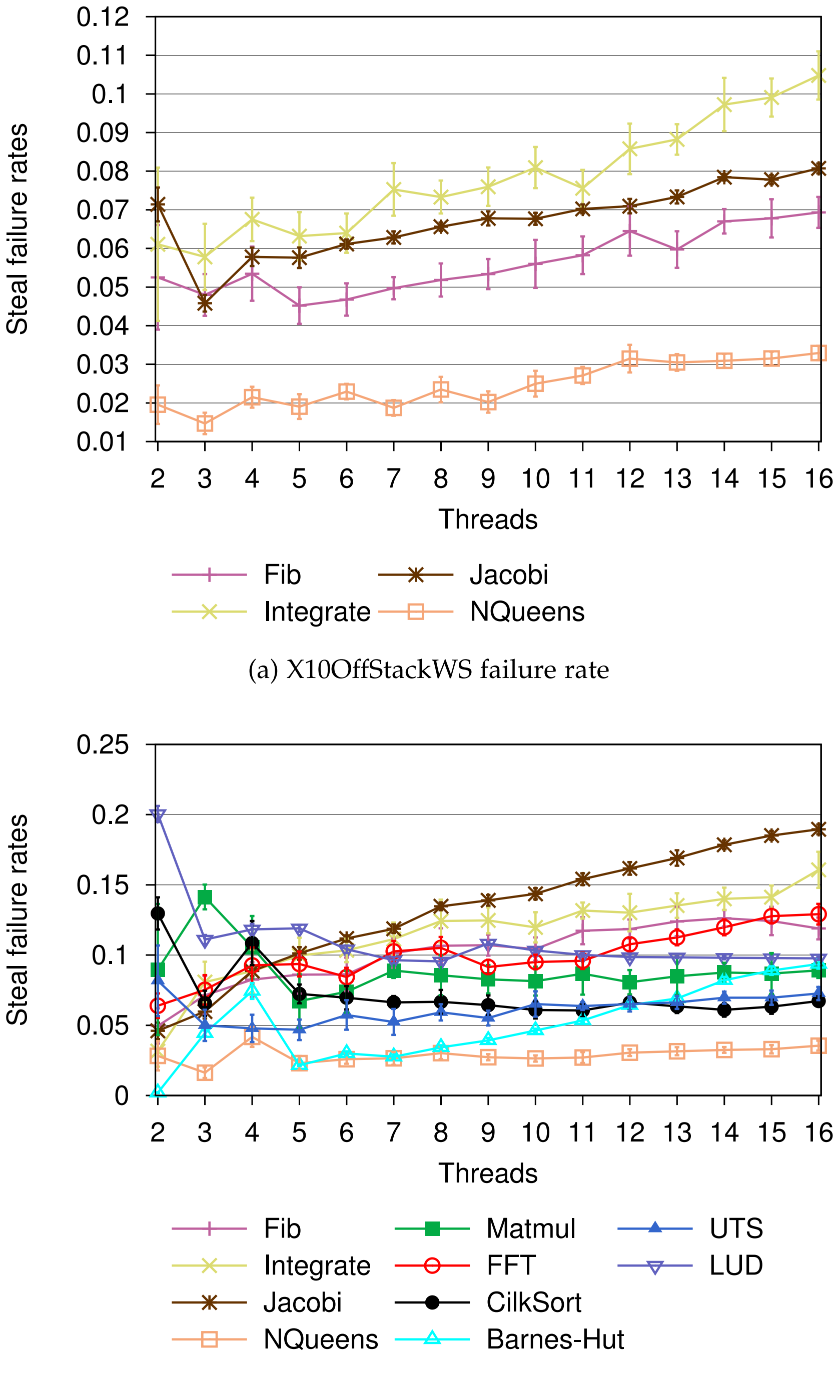


(a) X10OffStackWS steal ratio. This is very similar to X10DefaultWS (Figure 4.3(a)).



(b) X10TryCatchWS steal ratio. This is also nearly similar to X10DefaultWS (Figure 4.3(a)).

Figure 4.9: Steals to task ratio for X10OffStackWS and X10TryCatchWS. A low ratio is always preferred to ensure that sufficient tasks are created to keep all processors busy.



(a) X10OffStackWS failure rate

(b) X10TryCatchWS failure rate

Figure 4.10: Seal failure rates for X10OffStackWS and X10TryCatchWS. This is the ratio of total failed steal attempts and total successful steals. Steal attempts may fail due to either another thief or the victim winning the race to start that continuation.

failures is less than 13%, only rising above this figure for some of the benchmarks.

4.6 Related work

4.6.1 Languages versus Libraries

Work-stealing has been made available to the programmer as libraries or as part of languages. Java ForkJoin framework, Intel's Threading Building Blocks [Reinders, 2007], PFunc [Kambadur et al., 2009], and Microsoft's Task Parallel Library [Leijen et al., 2009] are all examples of libraries that implement work-stealing. Users write explicit calls to the library to parallelize their computation, as in Figure 2.3(c). X10, Habanero-Java and the Cilk-5 runtime [Frigo et al., 1998a] on the other hand are all examples of direct language support for work-stealing (as in Figures 2.3(a) and 2.3(b)). In principle, a language supported work-stealing implementation has more opportunities for optimization because it can bind to and leverage internal runtime mechanisms that are not visible to a library implementation. Conversely, library implementations have the pragmatic advantage of being applicable to pre-existing languages.

4.6.2 Work-stealing Deques

Cilk introduced the concept of the *THE* protocol [Frigo et al., 1998a] to manage the deque. Actions by the worker on the tail of the deque contribute to sequential overhead, while actions by the thieves on the head of the deque contribute only to non-critical-path overhead. Almost all modern work-stealing schedulers follow this approach. We do not take this approach. Instead we observe that steals are infrequent and force the victim to yield when a steal occurs. Although this implies that the victim does some steal-related work, it only does so when a steal occurs. As long as the steal ratios are relatively modest the gain in overall system performance from our approach results in better scalability than the traditional approach.

Some prior studies also reuse the execution stack as an implicit deque. Strumpen [1998] is the oldest in this category. They use a C language based implementation and employ both implicit and regular deques. Thieves only have access to the regular deques. Victims delay the pushing of heap allocated task on regular deque until an unsuccessful steal attempt happens. When a victim detects a failed steal, it unrolls its execution stack (extracting variables from stack and registers), creating several heap allocated tasks and then pushing them to its regular deque. The whole of C stack of the victim is restored from the just filled regular deque to resume the computation of victim. This study only uses Fib benchmark for its evaluation. Across variety of processor architectures, the sequential overhead in Fib ranges between 28% and 113% for implicit deques. Similar to Strumpen [1998], [Taura et al., 1999; Umatani et al., 2003; Hiraishi et al., 2009] also uses execution stack as an implicit deque. StackThreads/MP [Taura et al., 1999] employs an assembly language postprocessor to achieve portability across unmodified GNU C compiler. Thieves can walk victim's execution stack using frame pointers. The implementation is evaluated using several benchmarks ported from Cilk distribution. The sequential overheads in each benchmark are very negligible. However, StackThreads/MP does not seem to outperform the parallel performance of regular deques. Its parallel performance across all the benchmarks is very comparable with Cilk, which uses heap allocated tasks and regular deques. The other drawbacks in StackThreads/MP are the lack of support for dynamically-scoped synchronization (e.g., **finish** construct in X10) and inability to return values from threads. Umatani et al. [2003] aims to address the drawbacks in StackThreads/MP. It provides a Java based fork/join style programming model. The implementation consists of a Java-to-C translator and a runtime system in C. Similar to Strumpen [1998], it employs both implicit and regular deques. Thieves send steal request to victims and wait for response instead of directly operating on the regular deque. Unlike Strumpen [1998], the whole of C stack is not rebuilt from the regular deque in [Umatani et al., 2003]. The victim simply employes the regular deque as a list of continuation from which the frames can be popped. This implementation also allows dynamically scoped synchronization as well as Java style reentrant `synchronization` blocks. However, this implementation is evaluated only using Fib and Matmul benchmarks, where the sequential overhead is found to be 136% and 25% respectively. Parallel performance of Fib becomes better than Cilk at higher worker count. However, parallel performance of Matmul remains comparable with Cilk. Tascell [Hiraishi et al., 2009] is the latest implementation that does not use regular deques. It is an extended C language where programmers can write parallel program using Tascell's parallel constructs. It uses inner functions to implement backtracing and avoid stack walk of victim's execution stack. A nested function is the function defined inside another function, in places where variable definitions are allowed. Its evaluation creates a lexical closure accompanying the creation-time environment, and indirect calls to it provide stack access. When a victim receives a steal request, it spawns a task by temporarily backtracing and restoring its oldest task-spawnable state. As the actual tasks are created only when steals are requested, Tascell avoids the cost of deque management. Tascell uses several benchmarks for evaluation. However, the sequential overheads can still be as high as 148% (Fib). For NQueens the sequential overhead was 53%. The parallel performance of Tascell is always better than Cilk for all the benchmarks.

Our approach shares some similarity with previously discussed approaches. The fundamental insight is the same: work-stealing overheads can be significantly reduced by deferring operations that most other work-stealing systems perform eagerly. The biggest difference in our approach is: we abstain from reinventing the wheel and rely on highly optimized features already prevalent in modern JVMs. Apart from providing portability this also helps us achieve significantly better performance over the traditional approaches. We use a wide variety of benchmarks to demonstrate the superiority of our implementation. Fib is considered to demonstrate the worst-case performance of work-stealing implementations, as the tasks in Fib do not do any real computation other than wickedly launching several sub-tasks. Even with Fib, the sequential overhead on JavaTryCatchWS is just 29%. For Matmul and

NQueens (being used by prior work), the sequential overhead on JavaTryCatchWS is 8% and 10%, respectively.

4.6.3 Harnessing Rich Features of a Managed Runtime

Managed runtimes provide many sophisticated features (Section 2.4.2, page 18), which are not usually available in a low-level language implementation. A key runtime feature we use in our work is on-stack replacement (Section 2.4.2.3, page 19), which is already employed in Jikes RVM for speculative optimizations and adaptive multi-level compilation. To support on-stack replacement, Jikes RVM's compilers generate machine code mapping information for selected program points that enable extraction of the Java-level program state from the machine registers and thread stack and the transfer of this state to newly created stack frames. X10TryCatchWS and X10OffStackWS runtimes described in this chapter, uses these existing mechanisms inside Jikes RVM to walk a victim's Java thread stack and extract all the program state. The thief uses this to establish the necessary context for it to be able to execute stolen work.

The C++ implementation of X10 performs speculative stack allocation [Tardieu et al., 2012]. The victim starts by allocating the frames on a stack. The thief is responsible for copying the stolen frames from the victim's stack to the heap. This is not possible in the Java X10 implementation since Java does not support stack allocation. However we are able to leverage the runtime's stack walking mechanism to achieve an even simpler result—the thread state is not preprocessed. There are no frame objects on either the stack or the heap. Instead, by using the virtual machine's internal thread stack walking capability, the state is directly extracted from the stack when a steal occur. The approach used in this chapter radically lowers the memory management load of work-stealing. We are not aware of such functionality in any work-stealing scheduler.

4.7 Summary

This chapter first showed the extent of sequential overheads associated with work-stealing in well-known implementations such as, X10, Habanero-Java and ForkJoin. We demonstrate that stealing is generally uncommon and hence, take the approach of moving the overheads from the common case to the rare case. We propose two new designs, which leverage advanced facilities that exist within the implementation of a modern managed runtime. The fastest design was able to reduce the sequential overhead to just 10%. By contrast, X10 has the sequential overhead of 195%, Habanero-Java has 448% and ForkJoin has 124% overhead. We performed extensive performance analysis of the new designs and demonstrated that the approach followed is extremely effective at reducing sequential overheads and also without affecting the scalability. While the size of the benefit depends on the nature of the benchmark, in the cases where there was no significant benefit, importantly it also does not negatively affect performance.

In the next chapter, we focus on the dynamic overheads of work-stealing. These are the overheads, which increases with parallelism.

Chapter 5

Dynamic Overheads of Work-Stealing

The previous chapter attacks the problem of sequential overheads in work-stealing; those that manifest independent of the level of actual parallelism. This chapter explores the problem of dynamic overheads in work-stealing; those that manifest as steal rates grow, and are thus most evident when parallelism is greatest. To address the dynamic overhead, this chapter follows the trail of previous chapter and reuses rich features of modern managed runtimes to further refine the high performance JavaTryCatchWS work-stealing framework.

The chapter is structured as follows: Section 5.2 discusses the motivation for this work. Section 5.3 explains the design of the new system. Section 5.4 evaluates the performance of this new design. Finally, Section 5.5 provides an overview of related work.

This chapter describes work in a paper currently under review "Friendly Barriers: Efficient Work-Stealing With Return Barriers" [Kumar, Blackburn, and Grove, 2014b].

5.1 Introduction

This chapter identifies the dynamic overheads in work-stealing implementations. Our analysis correlates this overhead with increasing parallelism, which is an important factor in the modern hardware.

Our work-stealing runtime, JavaTryCatchWS, which was introduced in the previous chapter (Section 4.4.2, page 36) almost completely removes the sequential overheads and achieves both good scalability and good absolute performance. In this chapter, we use this high performance work-stealing runtime as the baseline system to assess our ideas.

Recall from Section 4.4.2.2 (page 37) that in JavaTryCatchWS, when a thief attempts to steal a task, it first requests the runtime to stop the victim so that it may safely walk the victim's execution stack. If the thief finds a steal-able task, it duplicates the victim's stack before allowing the victim to resume. The thief then runs a modified version of the runtime's exception delivery code to start the stolen task.

We identify the interruption of the victim as a contributor to dynamic overheads, at every steal attempt. In this chapter, we first evaluate this source of overhead and then develop a new design on top of JavaTryCatchWS. This design uses a *return barrier* (Section 2.4.2.6, page 20), which reduces the time spent scanning the victim's execution stack. By using this design, we are able to reduce the dynamic overhead by around 50% and achieve overall performance improvements of as much as 20%.

The principal contributions of this chapter are as follows: a) a detailed study of the *dynamic* costs of work-stealing — costs associated with stealing work from victims; b) an approach for reducing this overhead; and c) an evaluation of this new design against the baseline JavaTryCatchWS runtime from previous chapter.

We now conduct a quantitative analysis to characterize the dynamic overheads of work-stealing.

5.2 Motivating Analysis

The principal sequential costs in work-stealing relate to organizing normal computation in such a way as to facilitate movement of a task to another thread if a steal should happen to occur. On the other hand, the principal cost in the dynamic case lies in synchronizing victim and thief threads at the time of a steal to ensure that the thief is able to take the victim's work without tripping upon each other.

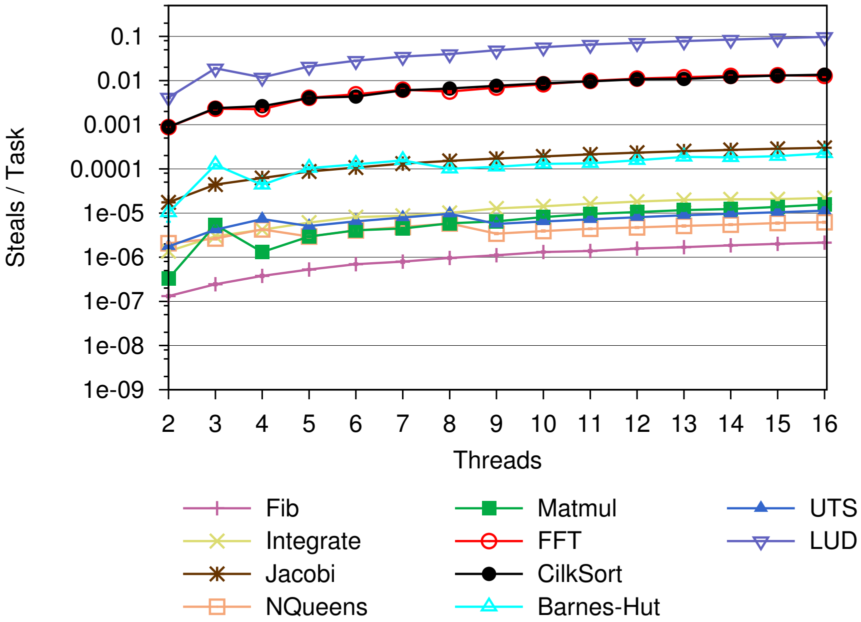
JavaTryCatchWS leveraged Jikes RVM's yieldpoint mechanism to yield the victim while each steal took place. The yieldpoint mechanism (Section 2.4.2.1, page 18) is designed precisely for preemption of threads and has been highly optimized. When a thief initiates a steal, it sets a yield bit in the victim's runtime state. The next time the victim executes a yieldpoint, it will see the yield bit and yield to the thief. The JVM's JIT compiler injects yieldpoints on method prologues and loop back edges, tightly bounding the time it takes the victim to yield. Notwithstanding the efficiency of the yieldpoint mechanism, this approach nonetheless requires the victim to yield for the duration of the steal, whether or not the steal is successful.

To shed light on the dynamic costs due to stealing and further motivate our design, we now measure 1) the steal rate (steals/msec), and 2) the overhead imposed by the steal mechanism upon the victims. Since Chapter 4's JavaTryCatchWS is our baseline system here, we call it DefaultWS for the remainder of this chapter.

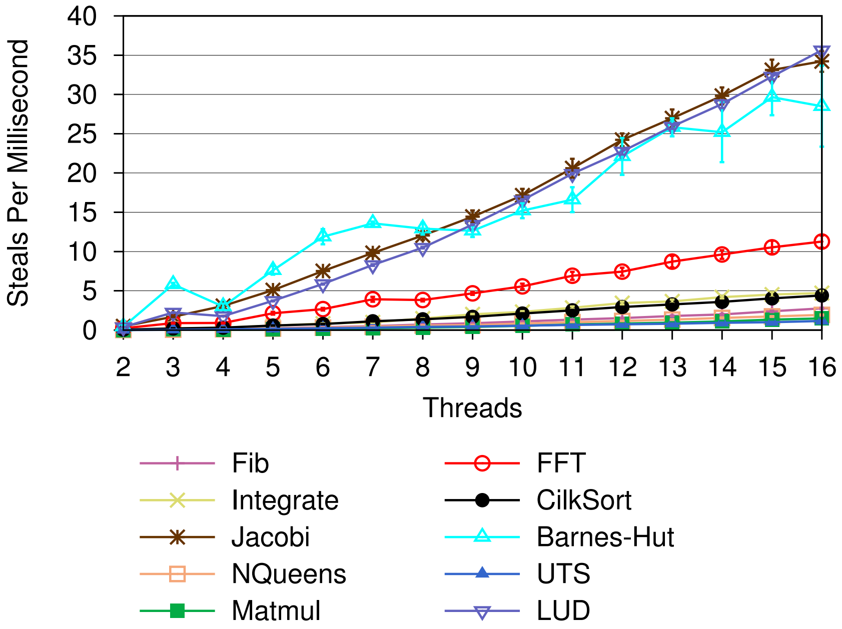
5.2.1 Steal Rate

The steal ratio is a common metric (Figure 5.1(a)) but is only one dimension of steal overheads. We also measure the steal *rate* (steals per millisecond), which is shown in Figure 5.1(b). This steal rate is calculated by dividing the total number of steals by the benchmark execution time. This indicates how frequently we are forcing the victim to execute the yieldpoint.

From Figure 5.1(a), we notice that the steal *ratio* for Jacobi and Barnes-Hut at 16 threads is as low as 0.0004 and 0.0002 respectively. However, their steal rate (Figure 5.1(b)) are as high as 34 and 29 steals per millisecond respectively, with the



(a) Steal ratio



(b) Steal rate

Figure 5.1: Steal statistics for DefaultWS show that steal ratio and steal rate can significantly differ across benchmarks. Jacobi and Barnes-Hut exhibit a low steal ratio at 16 threads, but shows a high steal rate with same number of threads.

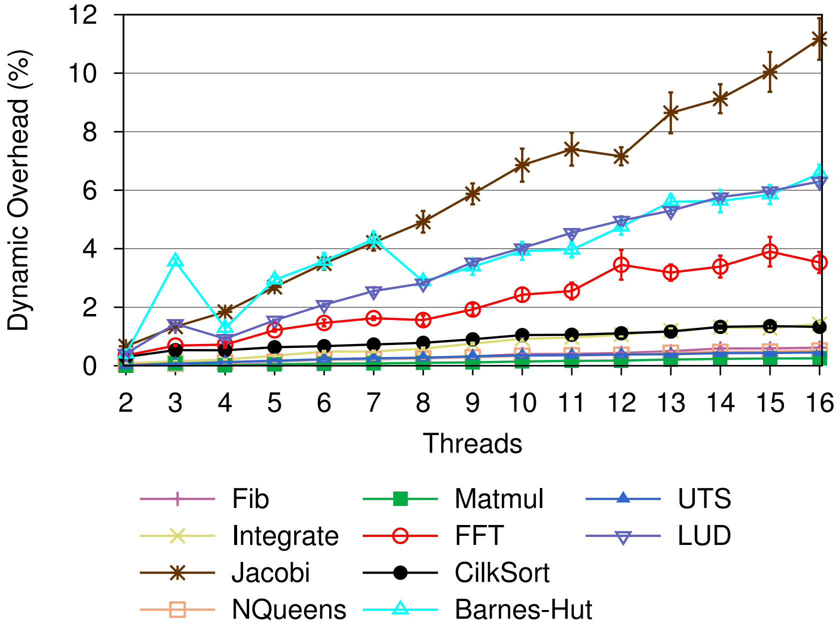


Figure 5.2: The dynamic overhead of DefaultWS is strongly correlated with the steal rate (Figure 5.1(b)). Jacobi has the highest steal rate and it has the highest dynamic overhead.

same number of threads. This result shows that even a benchmark with a very low steal ratio can have a very high steal rate. In our next study we will explore how high steal rates can affect the overall performance of the benchmarks.

5.2.2 Steal Overhead

In this study, we measure the cost of steals as imposed upon the victim by the thief. We measure this by calculating the percentage of CPU cycles lost by the victim while waiting for the thief to release it from the yieldpoint. For measuring the CPU cycles utilized by the work-stealing threads, we use hardware performance counters. We use the time stamp counter (TSC) [Intel Corporation, 1997] for measuring the cycles lost by the victim waiting to be released from yieldpoint. These cycles are summed for all the steals over the benchmark execution. The result in Figure 5.2 is calculated by dividing these cycles by total program execution cycles obtained from hardware performance counters as mentioned above.

By comparing this overhead, seen in Figure 5.2, with the steal rate in Figure 5.1(b), we can see that higher steal rates correlate with higher overheads. The six benchmark, which have noticeable dynamic overheads are: Jacobi, Barnes-Hut, LUD, FFT, CilkSort and Integrate. The steal overhead is as much as 11.2% (Jacobi with 16 threads). The remaining four benchmarks (Fib, NQueens, Matmul and UTS) have very small dynamic overheads (less than 0.6%). This study clearly shows that forcing the victim to wait inside a yieldpoint at every steal is not an efficient strategy, especially for benchmarks with high steal rates.

5.3 Design and Implementation

The previous sections identified the problem of dynamic overhead in a work-stealing runtime, highlighting the inefficiency of forcing the victim to wait inside a yieldpoint each time an attempt is made to steal work from it. We approach the problem by using a return barrier (Section 2.4.2.6, page 20), to 'protect' the victim from any thief, which may be performing a steal lower down on the victim's stack. The insight is that the cost of the barrier is only incurred each time the victim unwinds past the barrier. So long as the victim remains above the protected frame, it sees no cost at all, and yet is fully protected from any thief stealing work lower down on the stack.

We now discuss the design and implementation of our return barrier, and the modifications made to DefaultWS.

5.3.1 Return Barrier Implementation

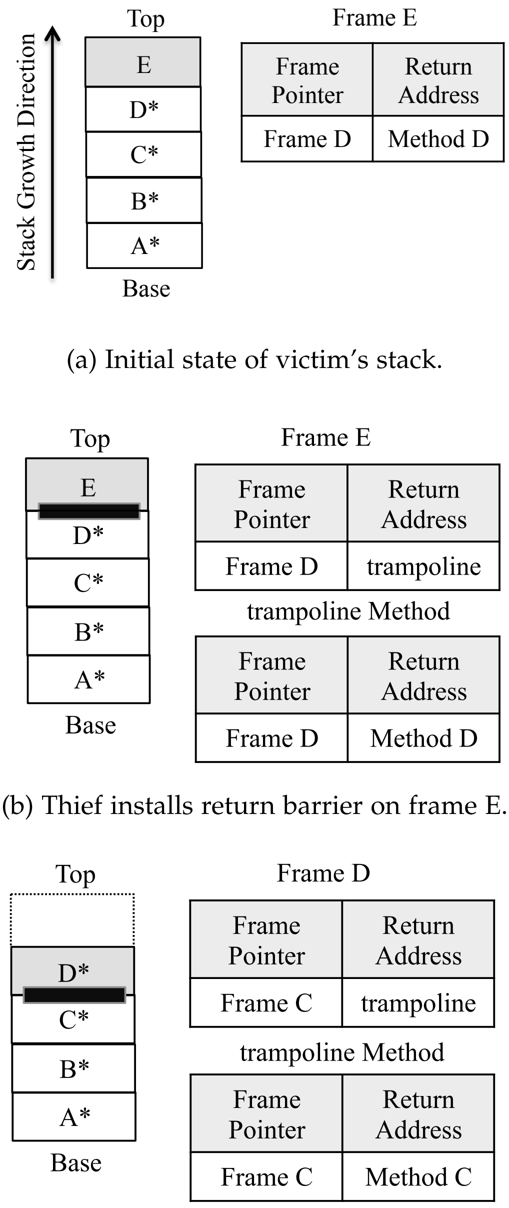
We use a return barrier to 'protect' the victim from stumbling upon an active thief. We do this by installing a return barrier above the stealable frames, allowing the victim to ignore all steal activity that occurs below the frame in which the barrier is installed. Only when the frame above the return barrier is unwound does the victim need to consider the possibility of an active thief.

A naive implementation of a return barrier would require some (modest) code to be executed upon every return, just as a write barrier is typically executed upon every pointer update. Instead we use an approach similar to that of Yuasa [Yuasa et al., 2002], where they used a return barrier to pause for garbage collection. We hijack the return address for a given frame, redirecting it to point to a special return barrier trampoline method, remembering the original return address in a separate data structure. When the affected frame is unwound, the return takes execution to our trampoline method rather than the caller of the returning frame. The trampoline method executes the return barrier semantics (which may include re-installing the return barrier at a lower frame), before returning to the correct calling frame (whose address was remembered in a side data structure). This barrier has absolutely no overhead in the common case, and only incurs a modest cost when the frame targeted by the return barrier is unwound.

We can use the return barrier trampoline to protect the victim from active thieves — ensuring that the victim never unwinds to a frame a thief is actively stealing. We now discuss the general process of stealing work before detailing how we use the return barrier to perform efficient work stealing.

5.3.2 Overview of Conventional Steal Process

Before describing our return barrier-based implementation, we outline the steps used to perform a steal in the DefaultWS implementation. In this process the thief steals



Frame E

Frame Pointer	Return Address
Frame D	Method D

(a) Initial state of victim's stack.

Frame E

Frame Pointer	Return Address
Frame D	trampoline

trampoline Method

Frame Pointer	Return Address
Frame D	Method D

(b) Thief installs return barrier on frame E.

Frame D

Frame Pointer	Return Address
Frame C	trampoline

trampoline Method

Frame Pointer	Return Address
Frame C	Method C

(c) Victim moves the return barrier on frame D.

Figure 5.3: The victim's stack, installation, and movement of the return barrier.

the *oldest unstolen* continuation from the victim.

1. The *thief* **initiates** a steal.
2. The *victim* **yields** execution at the next yieldpoint.
3. The *thief* performs a **walk** of the victim's stack to find the oldest *unstolen* continuation frame.
4. The *thief* adjusts the return addresses of the callee of the stolen continuation to ensure the unstolen callee is correctly **joined** with the stolen continuation upon return.
5. The *thief* **copies** the frame of the stolen continuation and those of each of its callers onto a secondary stack in the following steps:
   - The *thief* **links** the copied frame on the secondary stack.
   - The *thief* **scans** callee frames to capture any references pertinent to the stolen frames. This is necessary due to a callee-save calling conventions used by many compilers.
6. The *victim* **resumes** execution.
7. The *thief* **throws** a special exception, which has the effect of resuming its execution on the secondary stack (which is now its primary stack).

Notice that the victim must yield to the thief throughout steps 2 to 6. We now discuss how the return barrier can be used to avoid such yields when possible.

5.3.3 Installing the First Return Barrier

Figure 5.3(a) depicts a typical snapshot of a victim's execution stack. The stack frames with stealable continuations are marked with a '*' in this figure. The newly executed methods occupy the stack frame slots on the top of the execution stack. Each stack frame is recognized with the help of a frame pointer. The value stored inside this pointer is the frame pointer of the last executed method. The other information of interest to us is the return address, which holds the address where the control should be transferred after unwinding to the caller frame.

Once the thief has decided to rob this victim, it first checks whether a return barrier is already installed on the victim's execution stack. If it discovers that there is no return barrier installed, the thief then stops the victim by forcing it to execute the yieldpoint mechanism (steal step 2). Once the victim has stopped, the thief starts walking the stack frames to identify the oldest unstolen continuation (steal step 3). In our example, the oldest unstolen continuation frame *A*. However, before the thief reaches frame *A*, it notices that the first (newest) available continuation is *D*. It installs a return barrier to intercept the return from method *E* to *D*. The return address and return frame pointer in *E* is hijacked by the return barrier trampoline.

The return address stored in *E* is changed to that of the return barrier trampoline method. Figure 5.3(b) depicts the victim's modified execution stack.

The victim holds two boolean fields *stealInProgress* and *safeToUnwindBarrier*, which are now marked as *true* and *false* respectively by the thief. The flag *stealInProgress* is marked as *false* at the end of steal step 5, whereas *safeToUnwindBarrier* is marked *true* at the end of steal step 3. After installing the return barrier, the thief creates a clone of the entire stack of the victim and then allows the victim to continue. The victim continues the rest of its computation (frame *E*), oblivious to the activity of the thief, while the thief proceeds further with the stack walk in steal step 3. However, the thief now switches to the cloned stack of the victim.

5.3.4 Synchronization Between Thief and Victim During Steal Process

When the victim finishes executing method *E*, it returns via the trampoline method of the return barrier. It checks whether *safeToUnwindBarrier* is *true*. In this example, we assume it's still *false*.

Apart from the above two boolean flags, the victim also has a fixed size address array (we used size 20) to store frame pointers of its unstolen continuations. During the stack walk up to frame *A*, the thief updates the victim's frame pointer address array with the frame pointers of *C* and *B* (unstolen continuations). However, in reality there could be some unstealable frames in between stealable frames *D*–*A*. To make the description simpler, we have chosen this layout. In cases where there are more continuations than the victim's address array size, the thief starts inserting the surplus addresses from the middle index. After completing steal step 3, the thief marks the flag *safeToUnwindBarrier* as *true*. The victim is now ready to unwind to frame *D*.

There are situations when the frame *D* is the only unstolen continuation remaining on victim's stack. In this case the flag *safeToUnwindBarrier* will be marked as *true* only at the end of steal step 5. In this case, the victim cannot continue in parallel to the steal procedure so must wait on a condition variable until *stealInProgress* is *false*.

5.3.5 Victim Moves the Return Barrier

In our running example, the victim is now inside the return barrier trampoline method. The thief has finished steal step 3 and marked *safeToUnwindBarrier* as *true*. Frame *C* is the first frame pointer inside victim's frame pointer address array. The victim changes the position of the return barrier and re-installs on frame *C*. After this the victim safely unwinds to frame *D* and starts execution of the method *D*. Figure 5.3(c) shows this newly modified stack frame of the victim. It keeps on changing the return barrier position until the last available frame pointer in its address array. Once the steal is complete, the thief sets the victim's field *stealInProgress* to *false* and signals the victim. The victim is now ready to branch to join its part of the computation and become a thief itself.

Hence, the return barrier allows the victim to continue its computation in parallel to the thief's steal steps 3–5.

5.3.6 Stealing From a Victim with Return Barrier Pre-installed

Once installed, the return barrier removes the need for the yieldpoint mechanism in the steal step 2. Any thief that attempts to steal from a stack with the return barrier installed simply marks the victim's field *stealInProgress* as *true* and continues the rest of the steal steps 3– 5 concurrently with the victim's computation. The thief uses the cloned stack of the victim (from the previous thief) to complete the rest of its steal phases. We call this type of steal a *free steal*. There is no overhead imposed on the victim (unless the victim waits inside trampoline).

5.4 Results

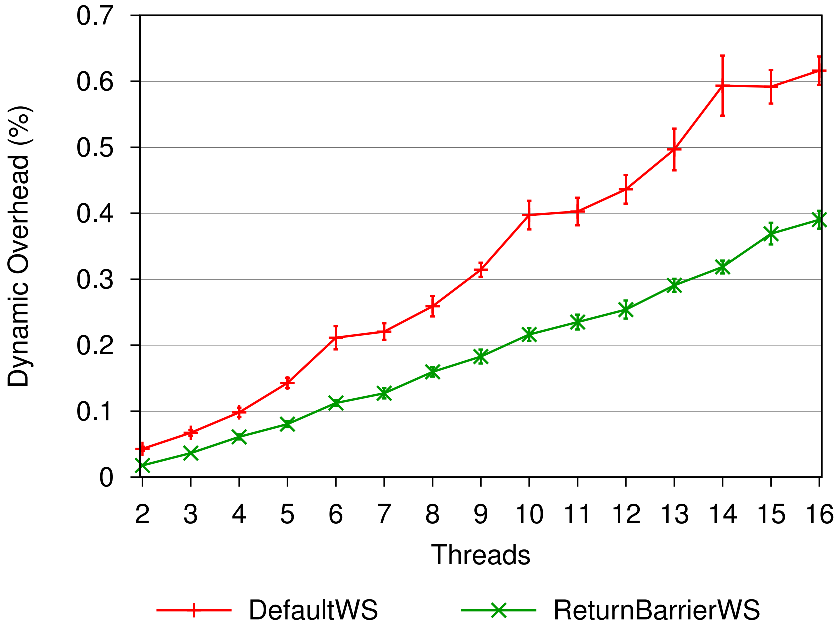
We begin our evaluation of return barriers by measuring the reduction in dynamic overhead before evaluating the overall performance gain. We call our new system ReturnBarrierWS in all further discussions below.

5.4.1 Dynamic Overhead

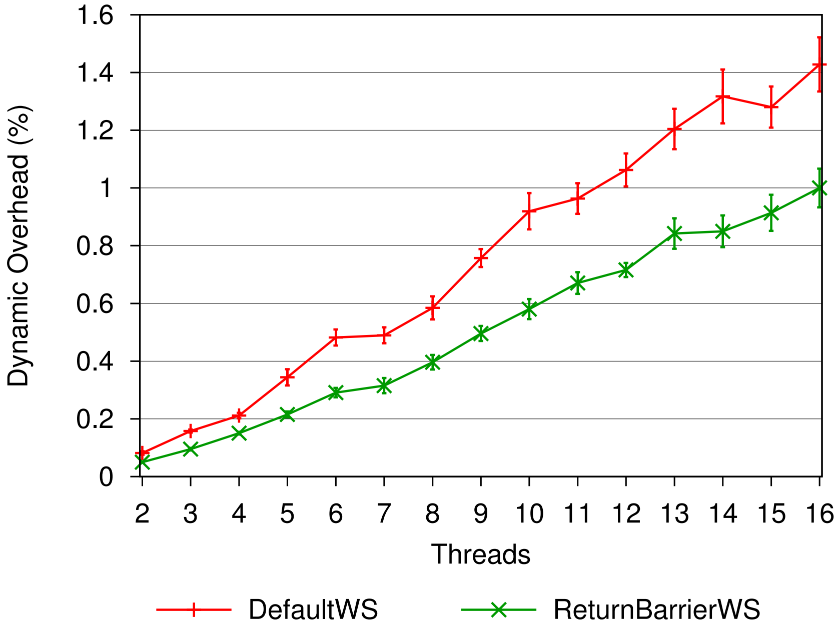
We measure the dynamic overhead of work stealing in both the DefaultWS implementation and ReturnBarrierWS implementation, which uses the return-barrier. Our methodology remains the same as that used in Section 5.2.2; we use the TSC to accurately measure the cycles spent waiting for steals and express that as a percentage of total execution time.

Figure 5.4 shows the dynamic overhead in both systems as a function of the number of worker threads. The results show that return barrier is very effective in reducing the dynamic overhead across all benchmarks, especially as the parallelism increases. For the benchmarks which have high steal rates (Figure 5.1(b)), the reduction in dynamic overhead with 16 threads is as follows: Jacobi by 29% (i.e. from 11.2% to 8%), in LUD by 24%, in Barnes-Hut by 30%, in FFT by 40%, in CilkSort by 46% and in Integrate by 30%. Similar trends hold even in the other four benchmarks, which have low steal rates.

Now we explore how the use of the return barrier affects total steal. Figure 5.5 compares the total number of steals in ReturnBarrierWS relative to those in DefaultWS. Values above 1.0 represent a higher number of steals in ReturnBarrierWS than DefaultWS, and vice versa. We observe from this figure that with the exception of CilkSort (at all thread counts) and Barnes-Hut (low thread counts), all other benchmarks exhibit similar number of steals in both systems. The higher number of steals at low thread counts in Barnes-Hut show up in Figure 5.4(h) as nullifying the return barrier advantage. On the other hand, the 15% reduction in steals in CilkSort is consistent with the good result seen in Figure 5.4(g). From these results, it can be seen that return barrier does not drastically change the total number of steals from

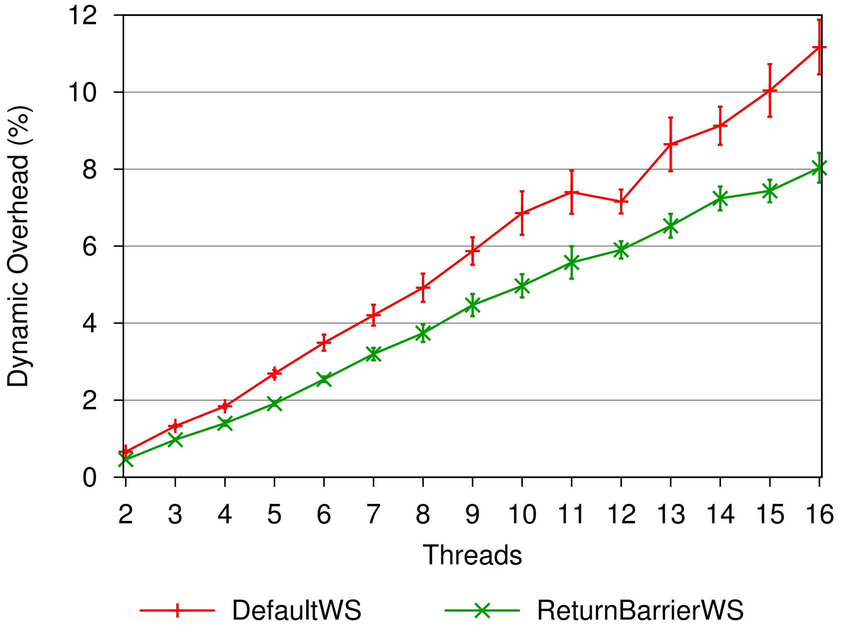


(a) Fib

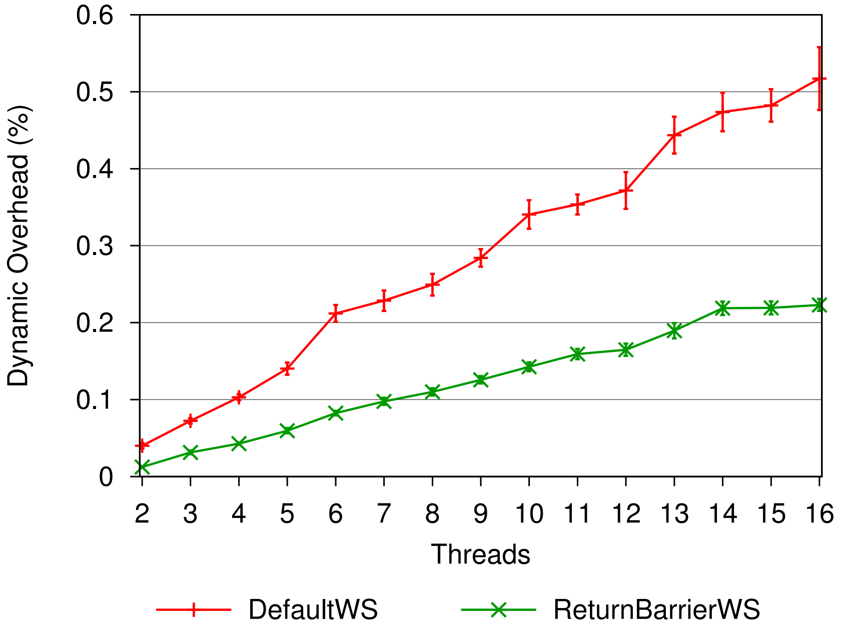


(b) Integrate

Figure 5.4: (Cont.) Dynamic overhead in our old and new system.

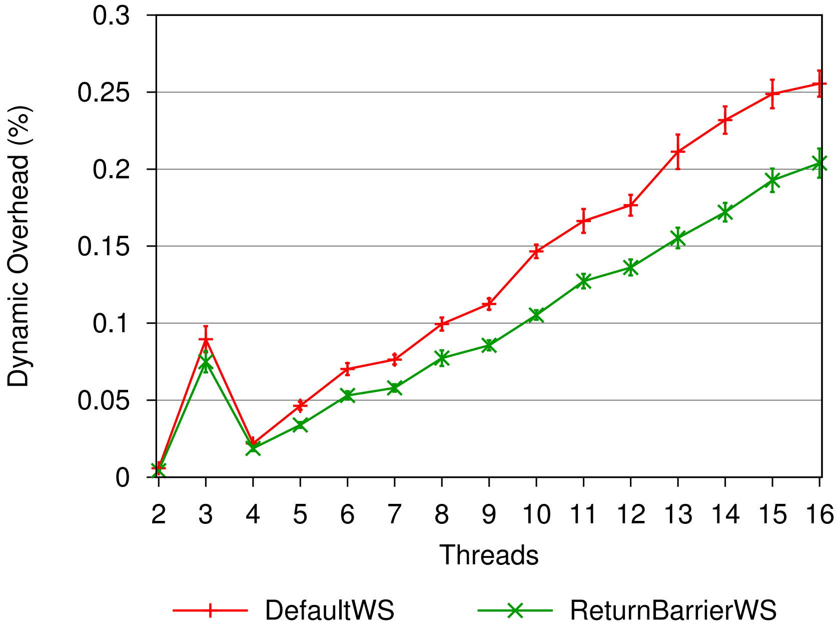


(c) Jacobi

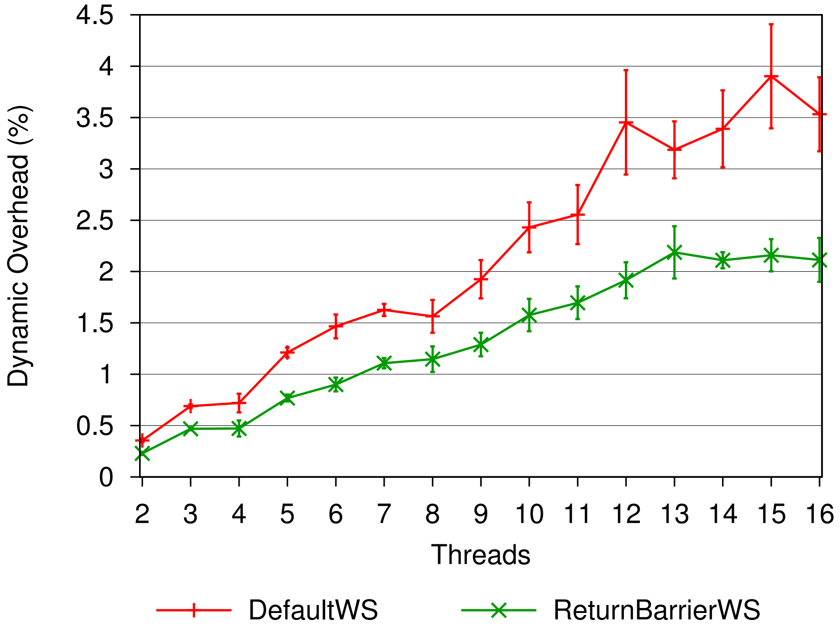


(d) NQueens

Figure 5.4: (Cont.)

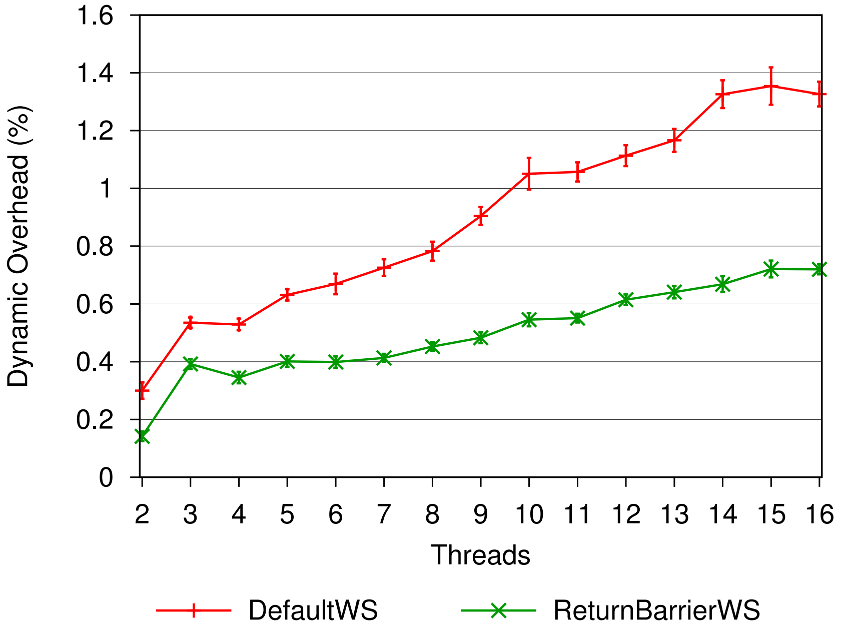


(e) Matmul

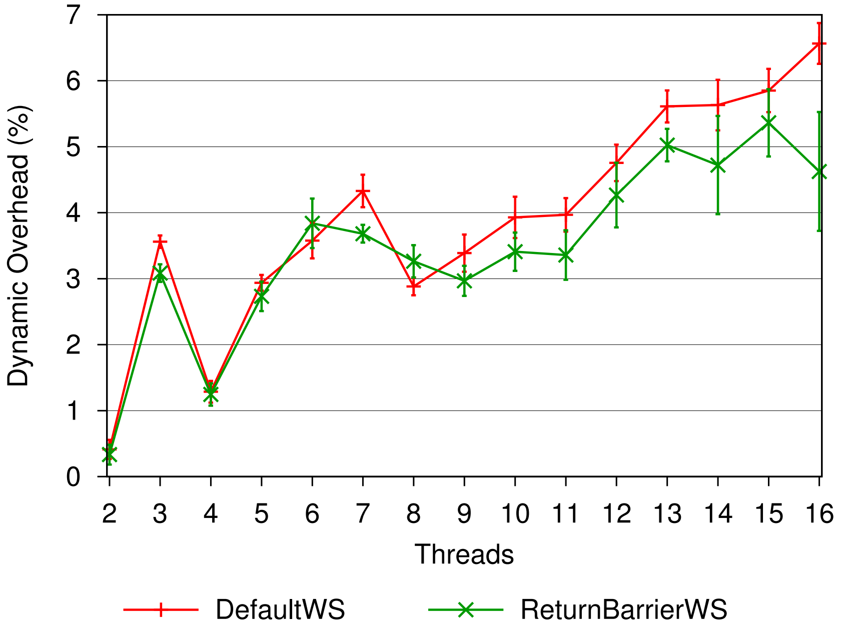


(f) FFT

Figure 5.4: (Cont.)

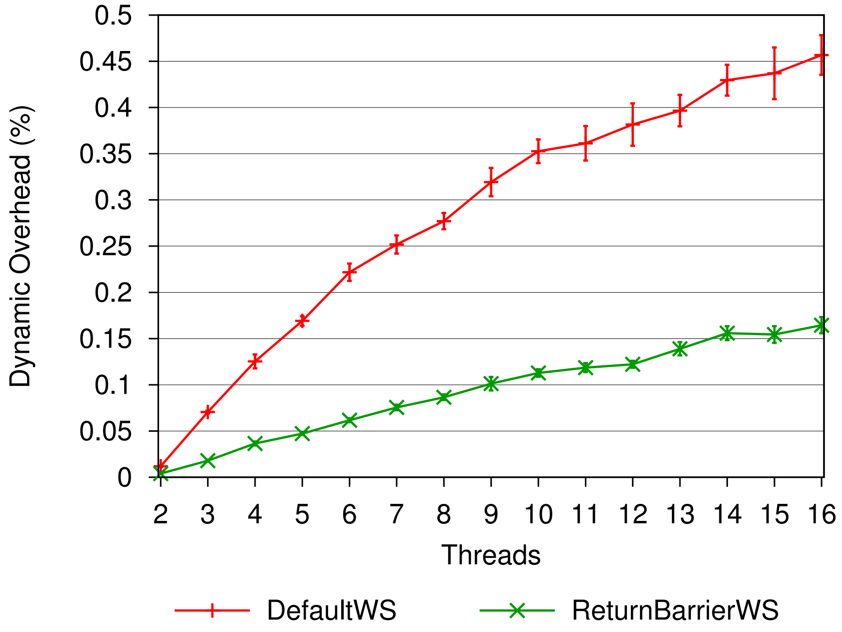


(g) CilkSort

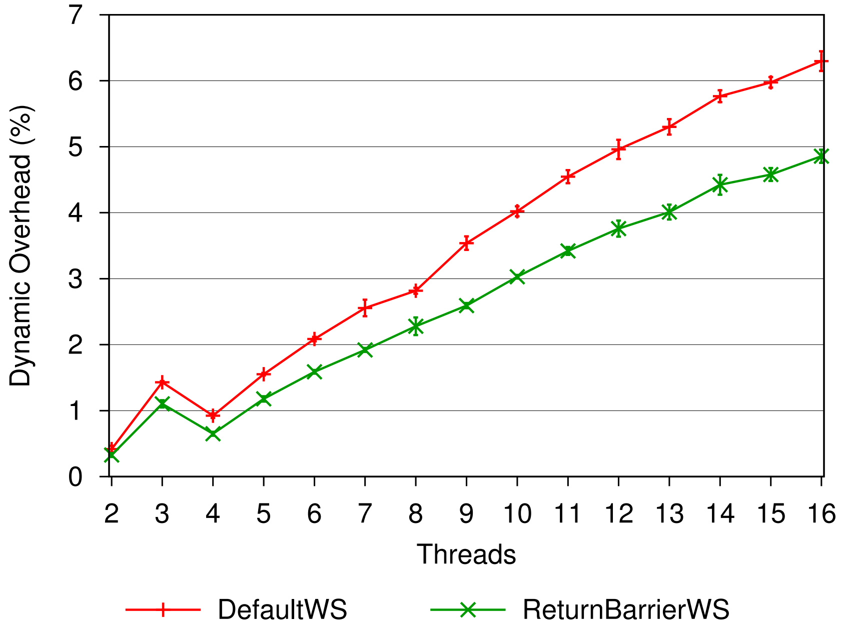


(h) Barnes-Hut

Figure 5.4: (Cont.)

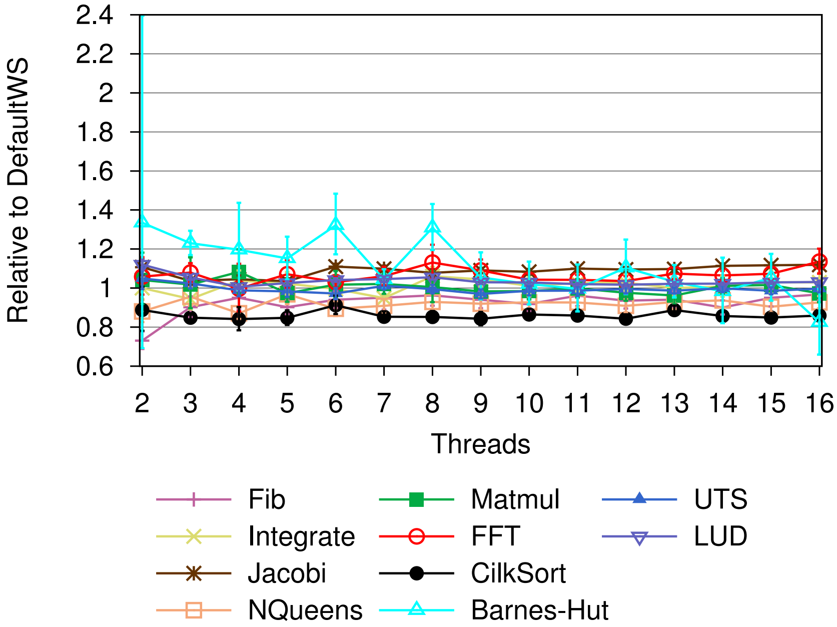


(i) UTS

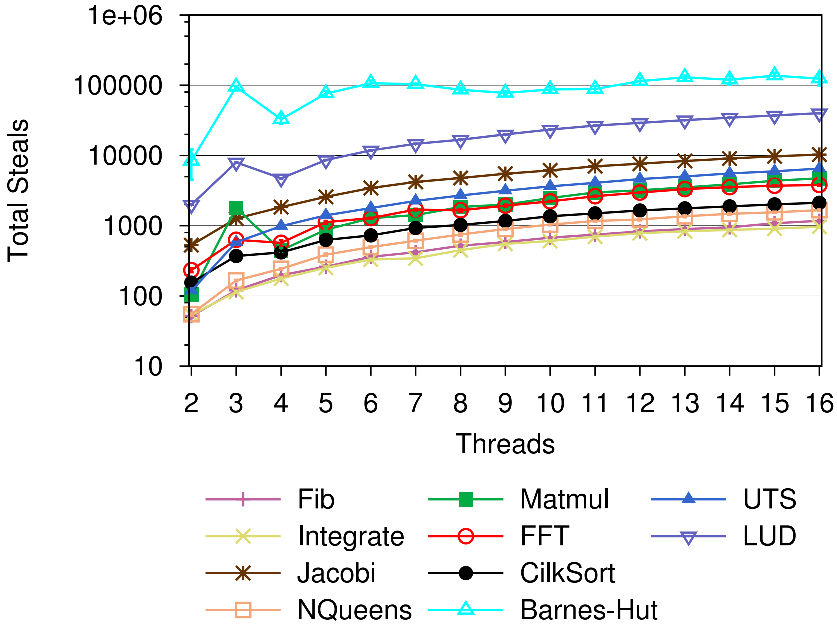


(j) LUD

Figure 5.4: (Cont.)



(a) Total Steals in ReturnBarrierWS relative to DefaultWS. With the exception of Barnes-Hut and CilkSort, all other benchmarks exhibit nearly similar number of steals in both systems.



(b) Total Steals in ReturnBarrierWS. Higher the steals, higher is the frequency of trampoline visits by victims.

Figure 5.5: Total Steals

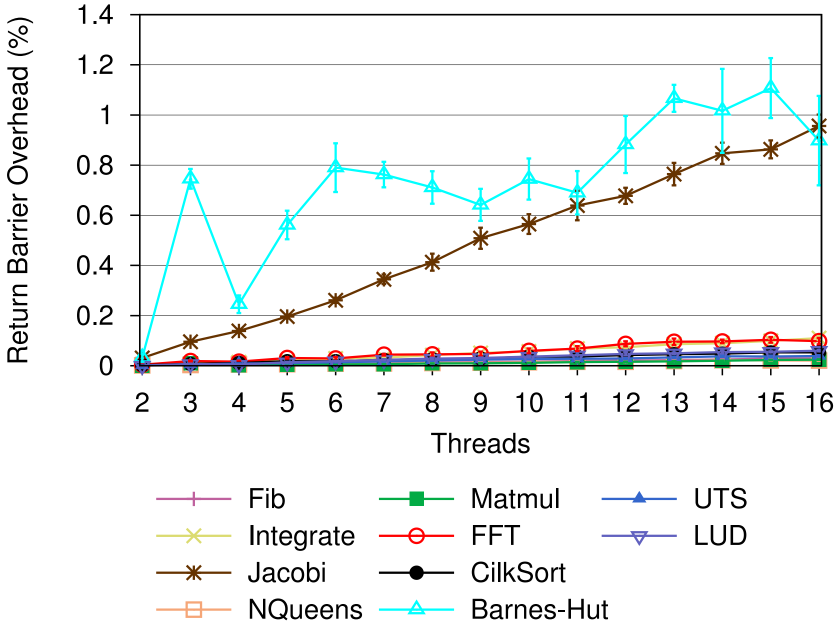


Figure 5.6: Overhead of executing return barrier in victims. Barnes-Hut has the highest overhead because of the high number of steals during its execution.

our baseline DefaultWS and also consistently reduces the dynamic overhead in all benchmarks.

5.4.2 Overhead of Executing Return Barrier

We now examine the cost to each thread of using the return barrier. Recall that this cost is only encountered when the stack unwinds to the point where a trampoline is installed. The trampoline is executed and it will: a) either re-install itself on the next unstolen continuation frame further down before returning to the hijacked frame; or b) wait on a condition lock if there are no more unstolen continuations left and the steal is still in progress. We measure this overhead by using a high-resolution timer and timing the time spent performing this operation. Figure 5.6 shows this overhead as a percentage of total program execution. With sixteen worker threads, the maximum overhead is around 0.95% in Jacobi and the minimum is 0.02% in NQueens. However, Barnes-Hut shows an overhead of 0.7% even with just 3 threads.

Figure 5.5(b), shows that Barnes-Hut has the highest number of steals. Even with just three threads, there are around 95000 steals, whereas the next closest, LUD, has just 8000 steals. More steals means more frequent trampoline visits by victims. This combined with a shallow stack (Section 5.4.3) leads to Barnes-Hut showing the highest overhead for the return barrier (max 1.1%).

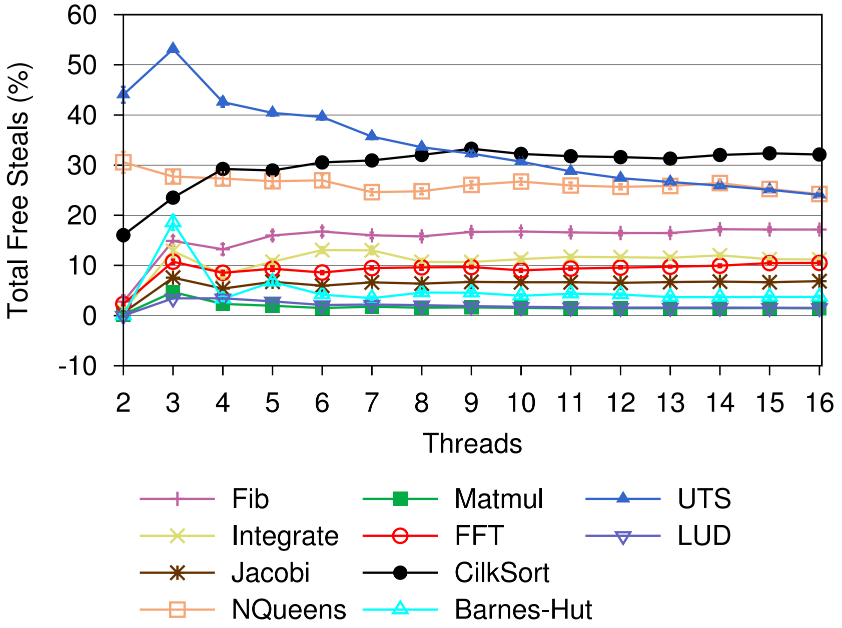


Figure 5.7: Free steals due to the presence of return barrier on stack. Lower percentage of free steals tends to reflect a shallow stack and hence, lesser benefit from return barrier.

5.4.3 Free Steals From Return Barrier

Recall from Section 5.3.6 that return barriers allow thieves to perform some steals for free. Figure 5.7 shows the percentage of steals that are free. UTS, CilkSort and NQueens have the maximum number of free steals. As the thread count increases, the percentage of free steels for these benchmarks converge close to 30%. Barnes-Hut, LUD and Matmul have the lowest percentage (3.5%, 1.5% and 1.4% respectively).

A higher free steal count reflects the return barrier staying longer on the victim's stack. This tends to reflect the depth of the victim's stack. A shallow stack will mean that the victim will tend to more often unwound past the return barrier, meaning that the thief tends to more often require the victim to execute the yieldpoint mechanism. This reduces opportunities for the return barrier to reduce the dynamic overhead. Figure 5.4 supports this conjecture. Barnes-Hut, LUD and Matmul benefit the least, whereas UTS, CilkSort and NQueens benefit the most.

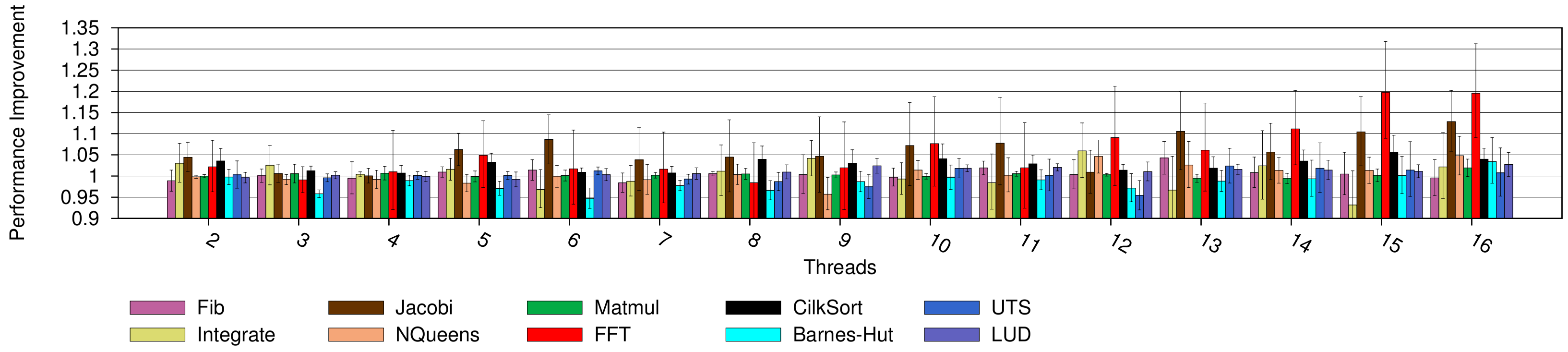


Figure 5.8: ReturnBarrierWS performance relative to JavaTryCatchWS, where time with n worker threads in ReturnBarrierWS is normalized to the time for same n worker threads in DefaultWS. Anything above 1.0 is a benefit.

5.4.4 Overall Work-Stealing Performance

The goal of this work was to reduce dynamic overheads, which are naturally most evident when the level of parallelism is high. Now we explore how the use of the return barrier affects overall performance when the number of threads grows to 16.

Figure 5.8 shows performance relative to DefaultWS for each of the benchmarks on our 16 core machine. The time with n worker threads in ReturnBarrierWS is normalized to the time for same n worker threads in DefaultWS. Anything above 1.0 is a benefit. We can expect benefits at high steal rates, which also happens as parallelism increases. Jacobi reaches the 10% mark with 13 threads. With 16 threads, Jacobi is 13% faster and FFT is 20% faster. CilkSort gets a maximum benefit of 5% (15 threads), whereas Integrate gets a maximum benefit of 6% (12 threads). For most of the other benchmarks the benefit is below 2% and is not significant.

The absence of a performance improvement in LUD and Barnes-Hut is despite the fact that their maximum dynamic overheads (close to 6%) are even higher than that of FFT (3.5%). The reason for this is little improvement in their dynamic overheads due to the presence of a shallow stack (Section 5.4.3). On the other hand, the remaining four benchmarks (Fib, NQueens, Matmul and UTS) already have very low dynamic overhead in DefaultWS (Section 5.2.2). This results in no significant benefit in performance even by reducing their dynamic overhead by a significant factor.

5.5 Related Work

5.5.1 Stealing overheads

In our work-stealing implementation, we steal only one task at a time as in X10, Habanero-Java, ForkJoin and Cilk etc. Though stealing one task at a time has been shown to be sufficient to optimize computation along the 'critical path' to within a constant factor [Arora et al., 1998; Blumofe and Leiserson, 1999], several authors have argued that the scheme can be improved by allowing multiple tasks to be stolen at a time [Berenbrink et al., 2003; Dinan et al., 2009]. Dinan et al. [2009] demonstrate that a 'steal-half' policy gave the best performance in their distributed setting. Stealing multiple tasks in a distributed setting has proven to be better in several other studies [Mitzenmacher, 1998; Lüling and Monien, 1993; Rudolph et al., 1991]. Cong et al. [2008] explore the idea of adaptive task batching for irregular graph algorithms. The thieves steal a batch of tasks at a time, where the batch size is determined adaptively. Several other authors have also argued that stealing multiple tasks works best in irregular algorithms [Sanchez et al., 2010; Acar et al., 2013; Hoffmann and Rauber, 2011; Hendler and Shavit, 2002].

These studies show that stealing multiple tasks is better in two cases: a) when performing work-stealing over a distributed setting, where the cost of stealing from remote node is substantial and hence stealing multiple tasks amortizes the communication overhead; and b) in irregular problems, such as depth-first-search algorithm, that do not fit into the divide-and-conquer model. However, not all the workloads are irregular in design nor do all follow the divide-and-conquer style algorithm, where the steal one approach always works better in a non-distributed setting. Though we have targeted divide-and-conquer style algorithms, our insight of using a return barrier to reduce the cost of stealing will perform well in irregular algorithms as well.

As described in Section 4.6 (page 56), JavaTryCatchWS (our baseline system) shares some similarity with Umatani et al. [2003]. Our return barrier implementation, ReturnBarrierWS also shares a similarity with that implementation. In Umatani et al.'s system, when the victim's stack is empty, it tries to get an unstolen heap frame from its deque. This transition between stack to deque imitates a return barrier. For the first steal, victim required synchronization and then again while fetching a heap allocated frame. However, we use explicit return barriers as we treat the victim's execution stack as its implicit deque.

5.5.2 Return barriers

The return barrier mechanism was first used in [Hölzle et al., 1992] in the context of debugging optimized code, to allow lazy dynamic deoptimization of the stack. It has also been used in various garbage collector algorithms [Yuasa et al., 2002; Saiki et al., 2005; Kliot et al., 2009]. Return barriers give Yuasa [1990] better realtime properties. In this work, we exploit the return barrier mechanism to optimize the steal process. To our knowledge, return barriers have not been applied to work-stealing until now.

5.6 Summary

This chapter further refines the high performance work-stealing framework JavaTryCatchWS, which we proposed in the previous chapter for reducing the sequential overheads of work-stealing. In this chapter, we identify the dynamic overheads of work-stealing; those that increase with parallelism. Our evaluations suggest that steal rate plays an important factor in determining the dynamic overheads. The higher the steal rate, the higher is the dynamic overhead. We show that the highly efficient yieldpoint mechanism to achieve synchronization between thief and victim in JavaTryCatchWS leads to overheads at higher steal frequencies. This is because of frequently stopping the victim to perform steal. We address this issue by exploiting the idea of a low overhead return barrier. This new design reduces the dynamic overhead by around 50% and achieves overall performance improvements of as much as 20%. Our design not only reduces the dynamic overhead across all benchmarks, but also never negatively affects performance.

In the next chapter we design a high productivity parallel programming environment for naively achieving data-race free parallelism. To achieve scalability we use the JavaTryCatchWS work-stealing framework and for ensuring data safety we use a data-centric concurrency control mechanism.

Chapter 6

Performance and Productivity via Data-Centric Atomicity and Work-Stealing

The previous two chapters identified the sequential and dynamic overheads in work-stealing runtime and introduced a high performance work-stealing framework – JavaTryCatchWS, which has very low overheads. This chapter presents an easy to use, small set of annotations to the Java programming language for achieving data-race free parallelism in real world problems, which otherwise is a monolithic exercise in large code-bases. The annotations developed in this chapter significantly lower the syntactic overhead of exposing parallelism and achieving data safety from concurrency.

This chapter is structured as follows: Section 6.2 discusses the motivation and performs motivational analysis. Section 6.3 explains the design and implementation of the new system. Section 6.4 discusses the improvements to productivity and performance from this new system and finally Section 6.5 provides an overview of the related work.

6.1 Introduction

Common programming models using threads impose significant complexity to organize code into multiple threads of control and to manage the balance of work amongst threads to ensure good utilization of multiple cores. Much research has been focused on developing programming models in which programmers simply annotate portions of work that may be done concurrently and allow the system to determine how the work can be executed efficiently and correctly. Concurrent object-oriented models such as ICC++ (Section 2.2.3, page 9) have combined fine-grained elective concurrency and object-oriented concurrency control in an effort to address these issues.

Models such as those offered by ICC++ provide flexibility for programmers but impose two major implementation challenges: a) object-oriented concurrency control can have high overheads if the programmer uses naively with extensive object-level locking; and b) potential concurrency annotations encourage a fine-grained division of work that would impose a great overhead if implemented directly as OS-level threads. To address these challenges we take a two fold approach: a) we use the data-centric programmer-declared consistency requirements from AJ (Section 2.2.3.1, page 10) to achieve concurrency control; and b) we use JavaTryCatchWS work-stealing framework to efficiently handle fine-grained parallelism.

The principal contribution in this chapter is a new system, which by drawing together these two techniques, allows the application programmer to conveniently and succinctly expose the parallelism inherent in their program in a main-stream object-oriented language. We develop a set of five annotations in the Java programming language to express parallelism and data-centric concurrency control. We further design a new system, which can translate this expression of parallelism into a form that can be efficiently executed by the JavaTryCatchWS work-stealing framework. To evaluate this new system, we modify three large existing parallel Java workloads. The results of this evaluation show that the proposed annotations allow software parallelism to be expressed concisely and that once expressed as fine grained parallel work units, the runtime can effectively exploit the hardware parallelism offered by modern multicore processors. Compared to Java's conventional object-oriented concurrency primitives, our annotations-based approach significantly lowers the syntactic overhead, and delivers significant performance and scalability improvements, of 40% – 53%.

6.2 Motivation and Motivating Analysis

Java has gained huge popularity over the last decade and has become one of the most widely used languages [Meyerovich and Rabkin, 2013]. However, mainstream languages such as Java do not provide support for parallel programming that is both easy to use and efficient. Expressing parallelism with either threads or with extensions such as Java's concurrent utilities has a significant syntactic overhead with respect to the serial elision (Section 3.1.1, page 25) version of the same code. In addition to the syntactic imposition, there is a substantial cognitive load associated with ensuring correctness of parallel code. This becomes a monolithic exercise in a large code-base, which may have dependencies spanning multiple classes. Vaziri et al. introduced the idea of data-centric concurrency control in Java, which significantly reduces the programming effort in managing concurrency issues [Vaziri et al., 2006]. Later, Dolby et al. [2012] extended this idea and evaluated against a large number of real world benchmarks. Their implementation was implemented via IDE-integrated refactoring of Java classes with data-centric annotations provided within Java comments. However, their implementation was limited to applications that use explicit Java threading.

With respect to parallel efficiency, work-stealing has gained lot of popularity because it relieves the programmer from using explicit threading and worrying about

	All Code			Parallel Portion		
		Synchronized		Parallel		LOC
Benchmark	**Files**	**Blocks**	**Files**	**Regions**	**Files**	**Overhead**
lusearch-fix	332	351	41	4	2	25%
jMetal	335	11	5	6	7	15%
JTransforms	44	0	0	372	24	16%

Table 6.1: Expressing parallelism in conventional Java has a major impact on how programs are written. For each of our three benchmarks we show, from left to right: a) the total number of class files; b) the total number of **synchronized** blocks and methods; c) the total number of files containing the **synchronized** keyword; d) the number of code fragments that express parallelism; e) the number of files containing explicitly parallel code; and f) the lines-of-code (LOC) overhead due to parallelism constructs within this code.

load-balancing. Java concurrent utilities provide an Executer interface, which uses Java ForkJoin work-stealing internally. However, to use these utilities the programmer must significantly modify the serial elision version of the code. Alternatively, the new breed of high-level languages like X10 (Section 2.2.1.1, page 6), aim to improve programmer productivity by providing built-in language constructs for expressing parallelism (e.g. the **finish-async** idiom). These constructs are translated from sequential code to parallel code via compiler transformations. Languages like X10 are currently not yet in the mainstream. Moreover, as observed in Chapter 4, current implementations of work-stealing for managed languages incur serious overheads.

Our approach, which we describe in Section 6.3, is to improve both productivity and performance by combining the data-centric programming model offered by Dolby et al. [2012] with the fine-grained work stealing made efficient by JavaTryCatchWS.

We start our motivating analysis with Table 6.1, which provides a basic evaluation of the syntactic load of different approaches to race-free parallelism. Because our goal is to show productivity with performance, we have targeted three benchmarks with large codebases that addresses real world problems (lusearch-fix, jMetal and JTransforms; Section 3.1 (page 23)). We view syntactic overhead as an indicator of ease of programming and thus productivity. The three leftmost columns provide statistics with respect to the entire code base for each benchmark, showing the total number of files and the prevalence of the **synchronized** keyword.

From Table 6.1, lusearch-fix has the largest number of **synchronized** blocks/methods. jMetal provides parallel implementation for only a subset of its code, hence the total number of **synchronized** is only 11. On the other hand, JTransforms provides a lock free parallel implementation, which is rare among real world benchmarks. Among the files included in lusearch-fix, two are the core of the benchmark which explicitly consider parallel execution of queries, while the remaining 330 are the Lucene library (Section 3.1 (page 23)), which account for 349 out of the 351 **synchronized** calls. When writing large libraries like this, programmers must keep track of all fields that are prone to data races. Missing any of them may be disastrous. Similarly, overuse of **synchronized** keywords is also undesirable as it will hamper parallel performance.

The three rightmost columns of Table 6.1 are with respect to the portion of the benchmark that is explicitly concerned with parallelism. In the case of lusearch-fix and jMetal, only a small portion of the code base is explicitly concerned with parallel execution. For example, in lusearch-fix there are just two classes which are concerned with the parallel distribution of work for a query. Of course the remaining 330 classes were written with concurrency in mind, so we see that there are many uses of the **synchronized** keyword in that non-parallel portion of the codebase. On the other hand, JTransforms has a large number of classes concerned with parallel work distribution. The LOC overhead is calculated by serializing the parallel portion of the benchmark. We do this by removing all parallel constructs. We then measure the LOC overhead by comparing parallel and serial versions of the code. For example, in lusearch-fix, the serial version has 448 LOC while the parallel version 601 LOC, which is a 25% overhead.

Table 6.1 shows that the maximum number of parallel code blocks occur inside JTransforms, followed by jMetal and lusearch-fix. JTransforms completely relies on `java.util.concurrent.Future` interfaces, whereas jMetal uses both Java threads and `java.util.concurrent.Executer` interface. lusearch-fix only uses explicit threading. For using either work-stealing via `java.util.concurrent.*` or explicit threading the programmer has to modify their serial elision code. The total percentage of extra code added to the serial elision version is an indicator of the parallel programming effort. From Table 6.1, we can observe that it ranges between 15% to 25% among our three benchmarks. The effort expended on parallelization varies among benchmarks, but for benchmarks like JTransforms, where there are 372 parallel blocks, introducing parallelism requires very significant code changes and is a daunting task for the programmer.

This study shows that to achieve parallelism in a large code base, serious programming effort is required. Ensuring atomicity further adds to programmer's pain.

6.3 Design and Implementation

The previous section identified shortcomings in current implementations of modern high-level language such as Java with respect to both concurrency control and the expression of parallelism. Data-centric concurrency control annotations as used in the AJ language provide an elegant solution for concurrency control and have been demonstrated to be very effective [Dolby et al., 2012]. However, AJ is limited to programs that use explicit Java threading to express parallelism, which leaves the programmer with the significant burden of efficiently balancing work on modern multicore hardware. On the other hand work-stealing offers a means of expressing parallelism that carries a lower syntactic load which may improve programmer productivity, as well as offering high performance and natural load-balancing.

Our contribution is to identify the principal benefits of JavaTryCatchWS work-stealing framework and data-centric concurrency control respectively, and then bring those together into a single, simple framework within Java that combines data-centric concurrency control with the high performance work-stealing implementation. We call this language AJWS — Atomic Java with Work-Stealing.

We now discuss the design and implementation of AJWS.

6.3.1 Annotations in AJWS

AJWS provides three annotations for data-centric concurrency control and two annotations for expressing parallelism with work-stealing. The five annotations are as following:

`@AtomicSet(a)` This annotation declares a new atomic set, `a`, just as '**`atomicset`** `a`' does in AJ.

`@Atomic(a) i` This annotation identifies `i` as a member of atomic set `a`. This annotation may be applied to: a) variable declarations, b) parameter declarations, and c) return types. This annotation subsumes the roles of AJ's '**`atomic`**`(a)`' and '`unifor(a)`' annotations.

`@AliasAtomicSet(a=this.b)` This annotation aliases set `b` to `a`, just as the '`|a=`**`this`**`.b|`' annotation does in AJ.

`@Steal{S1; S2; ...}` This annotation declares that statements `S1`, `S2`, ... may be executed in parallel. The current implementation of AJWS limits statements to method calls and **`for`** loops. Methods are each executed as stealable continuations and **`for`** loops are decomposed into parallel **`for`** loops, with each iteration executed as a continuation. This provides similar semantics to **`async`** in X10 (Figure 2.3(a), page 12).

`@SyncSteal{...}` This annotation provides synchronization for parallel execution; all continuations declared (via `@Steal`) within the block must complete before any thread leaves the `@SyncSteal`. This is very similar to **`finish`** in X10 (Figure 2.3(a), page 12).

Before we dive into the implementation details, we first use Figure 6.1 to show a sample of code written in AJWS. The example demonstrates the common case of **`for`** loop parallelism. In this example, the programmer has used the `@Steal` annotation in line 17 to identify that the iterations of the loop in lines 19 to 22 may be executed in parallel. Every **`for`** loop iteration will perform some computation by calling method `something()` (line 19). This method returns an integer value containing the result of the computation. If the return value is not zero, the `status` counter within class `Sample` will be incremented by calling the method `inc()` (line 21). However, this increment should be performed atomically. To achieve this, the declaration of field `status` is annotated as a member of the atomic set `a` using the `@Atomic` annotation (line 3). The atomic set `a` is declared in line 2 via the `@AtomicSet(a)` annotation.

```
public class Sample {
  @Atomicset(a);
  @Atomic(a) private int status;
  public int get() {
    return status;
  }
  private void inc() {
    status++;
  }
  private int something() {
    int res=0;
    // do something
    return res;
  }
  public void compute(int size) {
    @SyncSteal {
      @Steal {
        for(int i=0; i<size; i++) {
          int res = something();
          if(res != 0) {
            inc();
          }
        }
      }
    }
  }
  public static void main(String[] args) {
    Sample s = new Sample();
    s.compute(100);
    System.out.println("Status="+s.get());
  }
}
```

Figure 6.1: Sample code written in AJWS, showing a simple example of `for` loop parallelism.

```java
public class Sample implements atomicsets.Atomic {
 private int status;
 protected final OrderedLock locka;
 public final OrderedLock getLockFora() {
  return locka;
 }
 public final OrderedLock getLock() {
  return this.getLockFora();
 }
 public Sample() {
  this(new OrderedLock());
 }
 public Sample(OrderedLock a) {
  super();
  locka = a;
 }
 public int get(){
  synchronized(locka) {
    return status;
  }
 }
 public int get_internal(){
  return status;
 }
 private void inc(){
  synchronized(locka) {
    status++;
  }
 }
 private void inc_internal(){
  status++;
 }
 private int something(){
  int res;
  // do something
  return res;
 }
 private int something_internal(){
  int res = 0;
  // do something
  return res;
 }
public void compute(int size){
 try {
  dcFor(0, size, 1, size);
  Runtime.finish();
 } catch(ExceptionFinish ff) {}
}
private void dcFor(int lower,int upper,
          int slice, int size) {
 final int threads = Runtime.threads;
 if (slice >> 2 < threads) {
  int var0 = lower + upper >> 1;
  int var1 = slice << 1;
  try {
   Runtime.continuationAvailable();
   dcFor(lower, var0, var1, size);
   Runtime.checkIfContinuationStolen();
  }
  catch (ExceptionEntryThief t){}
  dcFor(var0, upper, var1, size);
 } else {
 for (int i=lower; i<upper; i++) {
  int res = something();
   if(res != 0) {
     inc();
   }
  }
 }
}
public static void main(String[] args){
 Sample s = new Sample();
 s.compute(100);
 System.out.println("Status="+s.get());
}
/* end of class Sample */
```

Figure 6.2: AJWS compiles AJWS code to vanilla Java, much like JavaTryCatchWS. The code above shows the vanilla Java translation of the AJWS example from Figure 6.1.

AJWS translates the annotated Java code to vanilla Java in much the same way that JavaTryCatchWS does, producing a very efficient work-stealing implementation. The translated Java code from this source is shown in Figure 6.2. We have intentionally selected the **`for`** loop parallelism in this example because it is perhaps the most important expression of parallelism. We found that all our benchmarks have the **`for`** loop parallelism only. We will use these two figures to help understand our implementation, which we now explain below.

6.3.2 Translating AJWS to Java

Dolby et al. [2012] performed translation from AJ to Java using Eclipse refactoring [Petito, 2007]. The data-centric annotations were expressed as Java comments in the program, which were translated to Java by launching the refactoring via the Eclipse IDE. However, this implementation has some shortcomings, notably: a) expressing AJ annotations as Java comments is not as expressive as standard annotations provided by Java; and b) AJ is dependent on Eclipse and a heavy refactoring process. Due to these shortcomings we take an alternative approach for translating AJWS to vanilla Java.

We use JastAdd [Ekman and Hedin, 2007], an extensible Java compiler to perform the translation of AJWS to vanilla Java. The benefits of using JastAdd are: a) AJWS annotations can be expressed as standard Java annotations; b) JastAdd provides an easy to use interface for extending the Java programming language, making the implementation of AJWS fairly straightforward; and c) AJWS can be straightforwardly integrated into build processes, allowing AJWS to be used in large codebases easily (Section 6.3.3). We will now discuss how we have used JastAdd to support AJWS.

6.3.2.1 Translating Concurrency Control Annotations to Java

The semantics of AJWS concurrency control are very close to those of AJ (Section 2.2.3.1, page 10), so we follow a similar pattern in performing our rewrites, although we use a quite different framework. We perform the Java rewrite [Ekman, 2004] via various AST classes in JastAdd, such as, `VariableDeclaration`, `MethodAccess`, `Block`, `ClassDeclaration`, `TypeDeclaration`, `MethodDeclaration` and `ConstructorDeclaration`. During the parsing phase, we maintain a list of atomic sets declared in each class. As in AJ, AJWS is currently limited to one atomic set per class and all of its subclasses. The `OrderedLock` object (Figure 6.2, lines 3–9) and constructor declarations (Figure 6.2, lines 10–16) are generated just as in AJ.

To generate a method declaration an analysis is first performed to find whether there are any atomic fields in the method's body. If there is an atomic access, then two versions of the method are generated (Figure 6.2, lines 17–24), a synchronized version and a lock-free version, which has the `_internal` suffix, following Dolby et al.'s naming. The unsynchronized `_internal` version of the method always calls an `_internal` version of any method.

When generating the default (synchronized) version of the method, unlike AJ, our implementation must allow the method to perform work-stealing. The default version of this method will try wrapping statements inside **synchronized** blocks. This method might call some other method which has work-stealing annotations. In that case only a non-_internal version of that method is called and also without any **synchronized** block (Figure 6.2, line 66). We assume that when the programmer uses a work-stealing annotation they understand that the entire parallel code block can never be performed atomically. In the case of non-work-stealing methods, the default version of our method will start the scope of **synchronized** block the first time it encounters a statement having any atomic access (Figure 6.2, line 26). It then performs an analysis to check how many more statements are dependent on this same type of atomic access. The scope of the **synchronized** block is closed as soon as the last statement (having atomic access) is reached from the previous analysis (Figure 6.2, line 28).

We suspend the work-stealing on a victim for the duration it is executing the critical-section (code within **synchronized** block). The reason for this approach is: (a) it does not makes sense to create a continuation within a **synchronized** block; and (b) we assume that the **synchronized** code block is not compute intensive and takes only few cycles to complete. Hence, in AJWS the victim turn-off work-stealing as soon as it enters a **synchronized** block. The work-stealing is turn-on once the victim is outside the **synchronized** block. The only drawback in this approach is starvation of thieves when this victim has continuations in his caller frames (not belonging to current **synchronized** scope). In our benchmarks, the **synchronized** blocks are very small (mostly counter update codes) and hence completes quickly.

6.3.2.2 Translating Work-Stealing Annotations to Java

The work-stealing code block in our example involves a **for** loop (Figure 6.1, lines 18). From our real world benchmarks we found that they all use only **for** loop parallelism. For this, they either generate explicit threads or use Java concurrent utilities. In either case they have to divide the load of the **for** loop equally among the total threads. This is inefficient, both in terms of productivity and load-balancing. We borrow ideas from the implementation of work-stealing in X10 from Tardieu et al. [2012] and use a divide and conquer style `dcFor` method corresponding to this **for** loop (Figure 6.2, line 45). This `dcFor` method finally uses JavaTryCatchWS (Figure 6.2, lines 44–47 and lines 55–60). This `dcFor` is valid only for basic **for** loops. If the increment of variable `i++` is performed in some other way, then this `dcFor` would fail. Hence, for those cases JavaTryCatchWS is used without converting the **for** loop to `dcFor` method. From our experience with our benchmarks, we found that all of them use this basic looping only, so are amenable to this transformation.

6.3.3 Build Integration

Using JastAdd gives us the benefit of AJWS being integrated into standard build tools such as Ant. Before `javac` is invoked, we use JastAdd to translate AJWS to Java. The

	Default			**Work-Stealing Annotations**		
Benchmark	Files	Regions	o/h	Files	Regions	o/h
lusearch-fix	2	4	25%	5	9	3%
jMetal	7	6	15%	28	34	2%
JTransforms	24	372	16%	24	276	3%

Table 6.2: The syntactic overhead of using AJWS's work-stealing annotations to express parallelism. For both default and work-stealing implementations, we show the number of affected files, the number of affected code blocks and the overhead in lines of code. For all three benchmarks, parallelism can be expressed substantially more succinctly using work-stealing.

JastAdd translation can be trivially parallelized. Since AJWS may have been used in several files in a large project, we have parallelized the AJWS translation. We use a script, which calculates the set of files in the project that use our annotations. It performs parallel translation, as the translation of files are not dependent on each other. Finally the translated files can be compiled using `javac`. This makes the integration of AJWS into large projects straightforward.

6.4 Results

We now evaluate both the programmatic and performance impact of AJWS. We begin the evaluation by measuring the reduction in programming effort, first with respect to parallelism, then with respect to atomicity. We evaluate the performance impact by first measuring the *sequential* overhead (i.e. the increase in single-core execution time due to transformations to expose parallelism), and then by measuring the *parallel* performance of AJWS.

6.4.1 Programmer Effort

We start our evaluation by measuring the syntactic overhead associated with AJWS compared to existing alternatives for Java. In each case we use the same methodology as we used in our motivating analysis (Section 6.2), measuring the number of lines of code required to correctly express parallelism and concurrency control respectively, using serial elision as a baseline.

6.4.1.1 Expressing Parallelism

To explore how *parallelism* is expressed, we evaluate the syntactic overhead of using our work-stealing annotations to express parallelism and compare to the default implementation of parallelism, and serial elision versions of each benchmark. Table 6.2 shows that in terms of lines of code, the programmatic impact of using work stealing to express parallelism is about one fifth that of the default implementations.

	Default		Data-Centric Concurrency Annotations				
Benchmark	Syncs	Files	`@AtomicSet`	`@Atomic`	`@AliasAtomic`	Total	Files
lusearch-fix	351	41	35	17	95	147	43
jMetal	11	5	3	1	8	12	6
JTransforms	–	–	–	–	–	–	–

Table 6.3: The syntactic overhead of using AJWS's data-centric annotations on our three benchmarks. JTransforms is lock-free, so there is no overhead for either approach. For jMetal there is a very small increase in lines of code, and for lusearch-fix, the overhead in lines of code is about half that of the default which uses the **`synchronized`** keyword.

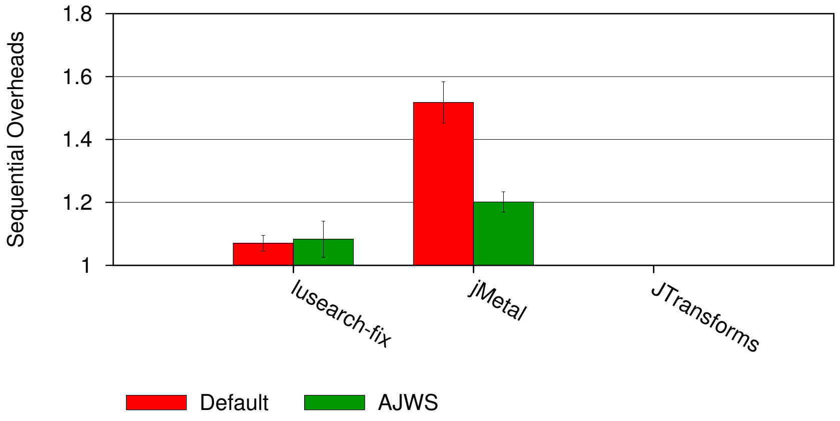
In both lusearch-fix and jMetal, we have introduced parallelism in more places than the default implementation of the benchmarks. We modified lusearch-fix to use parallelism at five extra places inside Lucene while jMetal has been parallelized at 28 more places than its default implementations. JTransforms is already heavily parallelized in its default implementation. We removed some parallelism as we found some computations were too small to be parallelized. Irrespective of this, we used our work-stealing annotations at 276 places in JTransforms. Despite this heavy usage, the average overhead in lines of code for using the work-stealing annotations in all our benchmarks is just 3%.

6.4.1.2 Concurrency Control

We now turn to *concurrency control* and look at the impact of moving from Java's use of locks and critical sections via the **`synchronized`** keyword to our annotations, which implement data-centric concurrency control. Table 6.3 shows the metrics for code changes related to the data centric annotations. Because JTransforms implements a lock-free algorithm, it does not use any explicit concurrency control mechanisms.

In lusearch-fix we needed only half as many data-centric annotations as the number of **`synchronized`** statements used in the default implementation. Usage of each of the three data-centric annotations is explained in Section 2.2.3.1 (page 10). We found one file in Lucene had 87 occurrences of **`synchronized`** keyword; almost every method was **`synchronized`**. However, with our implementation this file required just one `@AtomicSet` annotation for entire class. Our compiler can understand this hint and generate a **`synchronized`** version of any method that accesses any of the class fields. Not all classes are build like this, so in other cases we use `@Atomic` annotations as well. Lucene is a big project and has several interdependencies among files. We use `@AliasAtomic` annotations at 95 places to ensure we don't try acquiring locks from other classes unnecessarily.

Recall that we introduced parallelism in more places than the default implementation. This does not affect lusearch-fix, but does slightly affect jMetal. To ensure concurrent access to shared data, we had to use annotations at more places. Due to this jMetal has one extra concurrency annotation.



∗ JTransforms does not have any sequential overhead.

Figure 6.3: The sequential overheads associated with parallelism. Error bars indicate 95% confidence intervals. For lusearch-fix, the cost of introducing parallelism is around 7-8% for both the default Java implementation and AJWS. For jMetal, the performance overhead of introducing parallelism is around 50% for the default implementation and 20% for AJWS. We did not observe any performance overhead associated with parallelism in JTransforms.

6.4.2 Performance Evaluation

We now turn to performance evaluation. We first consider the *sequential overhead*, which measures the performance impact of adding parallelism to a program, evaluated by executing the parallel program with just a single thread and comparing it to the serial elision version of the program. We then consider *parallel performance*, where we evaluate the scalability of parallel programs, comparing AJWS with default implementations of parallelism. As in previous chapters, we wanted to eliminate memory management as a primary consideration to ensure our evaluation focused on the parallelism of the application rather than the scalability of the GC. Hence, we report mutator time only in all the performance related measurements.

6.4.2.1 Sequential Overhead

To evaluate the sequential overheads in AJWS, we use the same methodology as in Section 4.2.1 (page 28). Figure 6.3 shows the sequential overhead for each of our benchmarks. Y-axis values greater than 1.0 show this overhead. Our AJWS enjoys a low sequential overhead, like JavaTryCatchWS (Section 4.5.1, page 41), which it builds upon. JTransforms did not have any sequential overhead in either of the systems, hence it does not show up in the figure.

```java
Collection<Callable<Task>> task_list = new ArrayList<Callable<Task>>();
// create fixed thread pool
ExecutorService executor = Executors.newFixedThreadPool(total_threads) ;
for (int i = 0; i < total_tasks; i++) {
  task_i = new Task();  // allocate task
  task_list.add(task_i);  // grow task list
}
executor.invokeAll(task_list); // submit tasks to the thread pool
```

(a) Parallelism in jMetal. Here tasks are eagerly created. It is the job of the Java ForkJoin work-stealing runtime to ensure proper task scheduling. Task initiation and state management (Section 4.2.1, page 28) adds to the sequential overhead.

```java
Future<?>[] futures = new Future[total_threads];
for (int i = 0; i < total_threads; i++) {
  futures[i] = ConcurrencyUtils.submit(new Runnable() {
    public void run() {
      // divide total work among each thread
    }
  });
}
```

(b) Parallelism in JTransforms. Here the parallelism is more like a threading approach. Total number of asynchronous tasks created depends on the available thread pool size. This approach will have almost zero sequential overheads.

Figure 6.4: Pseudocode showing the style of parallelism in default implementations of jMetal and JTransforms.

Benchmark	Default	AJWS
lusearch-fix	2	2
jMetal	5	4
JTransforms	372	276

Table 6.4: Use of parallel regions that are exercised during execution of each of the benchmarks. For lusearch-fix and jMetal, this is substantially lower than the total number of parallel regions in the benchmark code bases (compare to Table 6.2), while for JTransforms, the number is slightly reduced.

JTransforms has an extremely large number of parallel regions but still it does not encounter any sequential overhead. To understand this we look to Figure 6.4. This figure shows the style of parallelism in default implementations of jMetal and JTransforms. jMetal (Figure 6.4(a)) takes the approach of dividing the entire computation into small tasks. These tasks are submitted to the Java ForkJoin thread pool via the `ExecutorService` interface in Java 7. Recall from Section 4.2.1 (page 28), that this approach encounters sequential overhead due to task initiation and state management. However, JTransforms (Figure 6.4(b)) does not eagerly divide the computation into small tasks. It divides the computation equally among the available threads. In Figure 6.3, the default implementation of JTransforms has been executed with only one thread, hence it performs similar to the serial elision. Despite the zero sequential overhead, this approach to harnessing parallelism in default JTransforms suffers in scalability (discussed in next section).

6.4.2.2 Parallel Performance

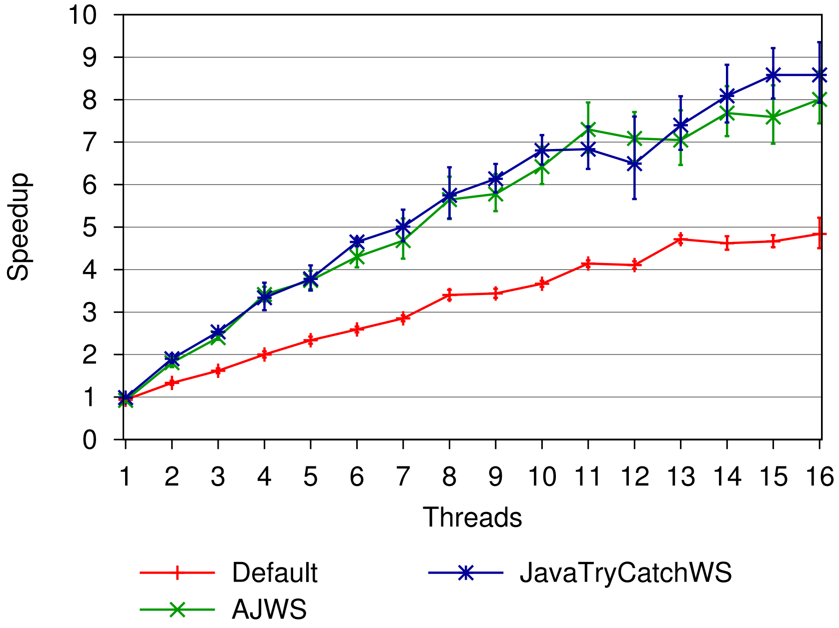
We now examine the parallel performance of AJWS. In Table 6.2, we identified the syntactic overhead of parallelism constructs on the entire code base of each of the three benchmarks we evaluate. In Table 6.4 we list the number of parallel regions within the portion of each codebase that is actually exercised when executing the benchmark, for both the default implementation and AJWS.

In Figure 6.5 we show the speedup of each benchmark in both the systems against their serial elision version, as the number of available cores increases from one to sixteen. In each case AJWS delivers substantially better performance than default Java implementations, and better scalability. It is important to remember that in the case of lusearch-fix and jMetal, the default implementation is a mature, well tuned parallel Java implementation (default implementation of JTransforms being the exception, which is discussed at the end of this section).

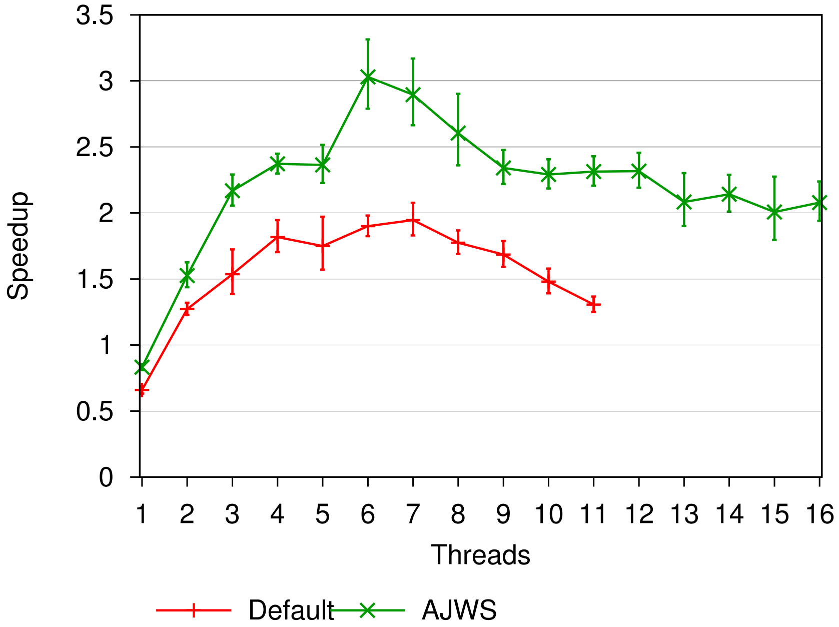
The AJWS implementation of lusearch-fix achieves a maximum speedup of 8.5× as compared to a maximum of 5.2× in the default implementation. To determine whether our concurrency control annotations were hampering performance, we also measure the speedup of the JavaTryCatchWS version of lusearch-fix. The performance and speedup of this implementation is similar to AJWS, demonstrating that concurrency annotations do not hamper the performance.

We were unable to run the default implementation of jMetal beyond 11 threads. This benchmark does not have a high degree of parallelism and hence we do not see any speedup benefit beyond 7 threads. jMetal achieves a maximum speedup of 4.5× in AJWS as compared to 3× in the default implementation. JTransforms shows very interesting results. The default implementation does not scale at all. With AJWS, JTransforms does exhibit modest speedup. It is able to achieve a maximum speedup of 2.4× against 1.1× in the default implementation.

To achieve parallelism using explicit threading, the programmer is required to divide the entire workload among the available threads at the program launch (e.g. lusearch-fix). However, when some threads finish their work sooner than others, they will be starved. This does not happen in a work-stealing implementation as they aim to ensure that sufficient tasks are created to keep all threads busy. However, in practice only a few of these tasks are actually stolen. A low ratio of total tasks stolen to total tasks created (the *steal ratio*) is highly desirable to ensure proper load-balancing. Figure 6.6 shows the steal ratio for our benchmarks on AJWS. It is very clear that AJWS also ensures a low steal ratio for almost all the benchmarks and ranging between 0.003 (lusearch-fix with 2 threads) to 0.341 (JTransforms with 16 threads). The steal ratio in JTransforms is on the higher side, which suggests — very limited



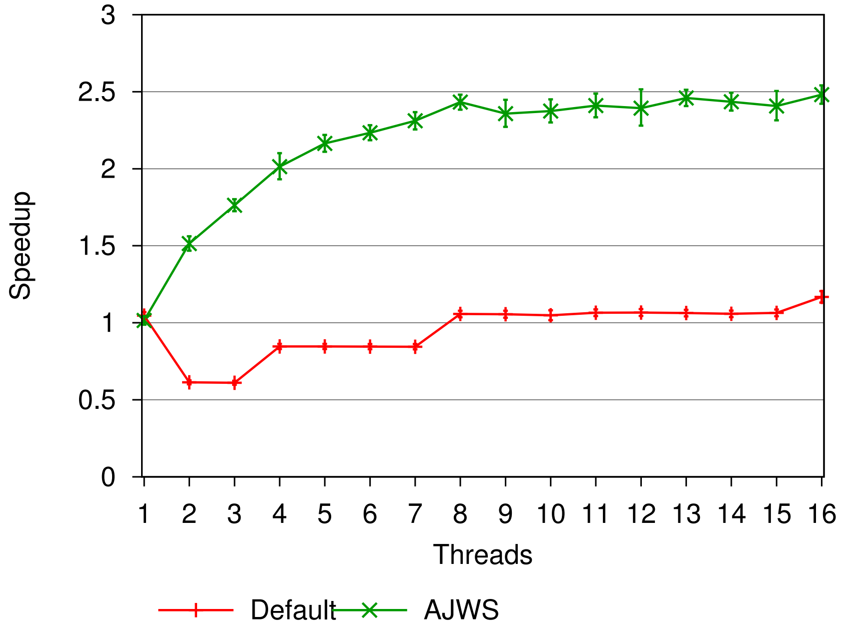
(a) lusearch-fix



(b) jMetal ∗

∗ We were unable to run Default jMetal beyond 11 threads.

Figure 6.5: (Cont.) Speedup over serial elision version for both the Default and AJWS implementations. Only lusearch-fix is also evaluated over JavaTryCatchWS to determine whether our concurrency control annotations were hampering performance.



(c) JTransforms

Figure 6.5: (Cont.)

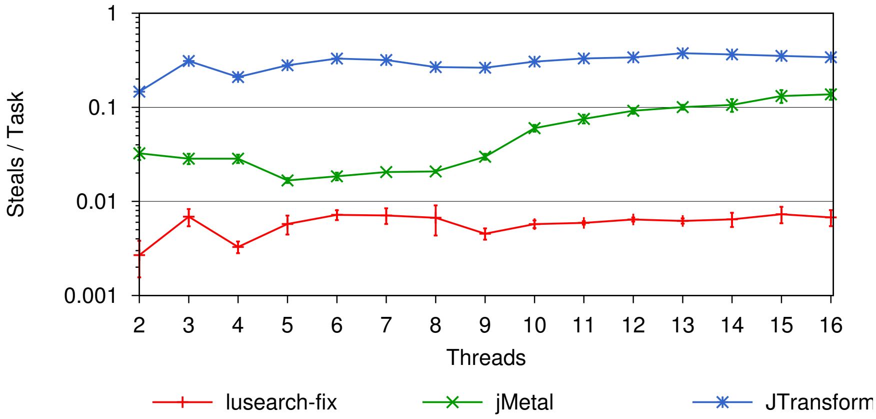


Figure 6.6: A lower steals to task ratio is highly desirable to ensure a proper load-balancing. A higher steal ratio in JTransforms suggests the presence of very limited amount of parallelism.

parallelism in its parallel regions. Due to this and a threading-*like* approach for parallelism (Figure 6.4(b)) there is no speedup in default implementation of JTransforms (Figure 6.5(b)).

6.5 Related Work

Attempting to create systems that provide a high-level interface to parallelism has a long history that predates current hardware trends. The actor model [Agha, 1986] was instantiated in many languages, such as the ABCL [Yonezawa, 1990] family, and there have been numerous efforts to blend object-oriented features with parallelism (see, for instance, this survey that covers just C++ [Wilson, 1996]). This work has received new impetus due to increasing hardware parallelism, giving rise to recent high-level languages like X10 (Section 2.2.1.1, page 6) and Chapel [Chamberlain et al., 2007]. However, while some of these languages attempted to interoperate with existing languages, they have not become mainstream. In contrast, the focus in this chapter has been on defining a small set of extensions for parallelism and synchronization that exist in a mainstream language, leveraging both existing code and also existing runtime mechanisms with minimal change.

Habanero-Java (Section 2.2.1.2, page 7), which was evolved from early versions of X10, shares some commonality with AJWS. Both Habanero-Java and AJWS use work-stealing for task parallelism. However, as we saw in Chapter 4, Habanero-Java work-stealing has serious sequential overheads and hence does not perform as good as JavaTryCatchWS work-stealing. Habanero-Java also differs significantly in its handling of concurrency control. Habanero-Java programs can either use `java.util.concurrent.locks` or the **`isolated`** construct to ensure atomicity. This is similar to **`atomic`** and **`when`** constructs in the X10 language [Saraswat et al., 2011]. However, just like **`synchronized`**, these need to be explicitly inserted in all necessary portions of code by the programmer, by strong contrast to the data-centric synchronization of AJ and AJWS.

Recently, Habanero-Java introduced a unified concurrent programming model, which combines the Actor model [Agha, 1986] with the **`finish-async`** task parallelism [Imam and Sarkar, 2012]. Their proposed model aims to combine the no-shared mutable state and the event-driven philosophy of the Actor model with the divide-and-conquer approach of the **`finish-async`** model. The key idea in an Actor model is to encapsulate mutable states and use asynchronous messaging to coordinate activities among Actors (the central entity, which determines how computation proceeds). However, an Actor model suffers overheads for: a) message delivery; and b) contention on the mailbox (a shared resource). This was observed by Imam et. al. [Imam and Sarkar, 2012], although their implementation is much faster than other Java based Actor frameworks such as Akka [Akka Team, 2013], Jetlang [Rettig, 2013] and Kilim [Srinivasan and Mycroft, 2008]. Habanero-Java also provides a dynamic data-race detector tool [Raman et al., 2012] which can be used to determine correctness.

We rely on compile time decisions to discover data-races and generate atomic blocks. This ensures no runtime overhead to discover and rectify data-races. Combining this approach with a high performance implementation of work-stealing further promises both performance and productivity.

6.6 Summary

Despite their huge popularity, mainstream languages like Java lag in terms of productivity and performance in the face of large scale parallelism. This chapter discusses this issue in details by evaluating three well matured open sourced benchmarks, which are also having a large codebase. We propose a small set of five annotations in Java programming language. Our new system, AJWS described in this chapter, translates these annotations from a regular sequential Java program into a parallel program that uses data-centric concurrency and JavaTryCatchWS work-stealing framework. We evaluate this system against the default implementation of each benchmark. The results demonstrate that these five annotations are extremely effective in reducing the syntactic overhead of parallel and concurrent constructs, enhancing programmer *productivity*, an important consideration given current hardware trends. The results also show that JavaTryCatchWS work-stealing can significantly boost *performance* relative to existing parallel Java implementations in the face of increasing core counts.

Chapter 7

Conclusion

Development costs and increasing hardware complexities have pushed the software community away from low-level programming languages towards high-level languages. These languages generally use a managed runtime to achieve portability and productivity over different hardware. However, while these managed languages abstract over hardware complexity, their performance is significantly low and will not improve greatly unless they are able to exploit increasing hardware parallelism.

This thesis explores the challenge of achieving high performance parallelism in managed languages, and harnessing rich features of modern managed runtimes, together with alterations to language design, to solve this challenge.

We present a quantitative analysis of the performance of the work-stealing scheduling technique, a popular approach for exploiting software parallelism on parallel hardware. We show that the substantial sequential and dynamic overheads of work-stealing can be greatly mitigated by exploiting and repurposing runtime mechanisms already available within managed runtimes.

We identify three key components that contribute to the sequential overheads of work-stealing, namely, a) initiation, b) state management, and c) termination of parallel tasks. To address these sources of overheads, we implemented two efficient work-stealing designs using key features from Java virtual machine such as: the yieldpoint mechanism, dynamic code-patching, on-stack replacement, and exception handling support. Using these techniques, the fastest design developed in this thesis has a sequential overhead of just 10%, as compared to 195% in X10, 448% in Habanero-Java and 124% in Java ForkJoin.

We identify that despite the efficiency of the yieldpoint mechanism, it could lead to dynamic overheads in work-stealing. This is because of frequent synchronizations between victim and thief. To lower this overhead, we exploited the idea of a low overhead return barrier to approximately halve the dynamic overhead and achieve overall performance improvements of as much as 20%.

The above work was done in X10, which is not yet a mainstream language. To ease the use of this highly efficient work-stealing framework in Java programming language, we developed a set of five Java annotations. These annotations facilitate translation of a regular sequential Java program into a parallel program that uses data-centric concurrency and our low overhead work-stealing framework. Our evaluation of this system against real world problems demonstrates that these five annotations are extremely effective in reducing the syntactic overhead of parallel and concurrent constructs, enhancing programmer productivity, an important consideration given the current hardware trend. Further, the performance results of this new system shows significant benefits relative to conventional approaches.

In combination, these contributions demonstrate that the wealth of research invested in the evolution of modern managed runtimes for achieving productivity and performance is a fruitful and exciting source of engineering opportunities. The insights developed in this thesis provides further hope to those pursuing the ambitious goal of 'abstracting without guilt' in the face of challenging hardware changes.

7.1 Future Work

The following sections focus on future directions that may significantly further improve the performance of our high performance work-stealing framework JavaTryCatchWS (Chapters 4 and 5).

7.1.1 Adaptive work-stealing

We have observed that in several benchmarks, the frequency of steals shoots up when the execution is nearing completion. In a divide-n-conquer approach the initial tasks are the ones with the maximum computations. The computation size gradually reduces as the program execution advances. Because of this, the worker threads very quickly finish the computation from the stolen task, and frequently become a thief. This implies that the maximum dynamic overhead is encountered during the last stages of program execution.

A promising approach to address this situation is using an adaptive work-stealing strategy. This adaptive approach will use a conventional style of work-stealing when the dynamic overhead shoots up and the JavaTryCatchWS style for the remaining execution. However, to achieve this the following changes would be required: a) *Changes to code transformation:* The AJWS compiler would need to be modified so that it can generate two versions of the code. One version would support the conventional work-stealing and the other would support JavaTryCatchWS work-stealing; and b) *Changes to runtime:* The runtime should be intelligent enough to decide when to trigger the switch from one style of work-stealing to another. One possibility is to constantly measure the CPU cycles required to finish a stolen task and the cycles required for stealing. When the cycles required to steal consistently exceeds the cycles required to execute the task, the switch can be triggered. This way, we would be able to harness the benefits of the conventional work-stealing approach and the JavaTryCatchWS framework.

7.1.2 Utilizing slow path of thief for VM services

Current implementations of garbage collection and JIT compilation are not tuned to work-stealing runtimes. They may stop the worker threads in the midst of some computation. As we know, in work-stealing when a thief runs out of work, it will start searching for next victim. This phase is not on the fast path of program execution. One possibility to keep would-be thief busy would be to task them with other works, such as thread local garbage collection [Marlow and Peyton Jones, 2011; Jones and King, 2005] or JIT compilation. This would avoid stopping the victim thread and speed the program execution.

7.1.3 Steal-*N* work-stealing

Several papers have argued that stealing multiple tasks works best for irregular algorithms [Dinan et al., 2009; Cong et al., 2008; jai Min et al., 2011]. Stealing multiple tasks from victim's stack is very easy in JavaTryCatchWS. This is because we simply do a plain execution stack copy to steal task from victim. The overhead to steal-*N* tasks is similar to that of stealing one task. Hence, a similar evaluation for irregular benchmarks with JavaTryCatchWS would be very interesting.

7.1.4 Task Prioritization

Benchmarks (such as Barnes-Hut) could be composed of several types of tasks. Some tasks (or methods) may be computationally more expensive than others. JavaTryCatchWS currently performs randomized work-stealing, just like X10, Habanero-Java and Java ForkJoin. It randomly selects a victim and steals the task. However, this randomized approach could be replaced with a task priority approach. The JVM runtime knows which methods (i.e. tasks) are frequently executed (i.e. hot). Methods, which are frequently executed may imply a smaller computation. One such example is a parallel `for` loop, where the iterations may call another set of parallel tasks. In this case, it might be best to steal the `for` loop iterations than the forked parallel tasks from the iterations. The thief can query the JVM to find the hotness of methods and can prioritize their steals accordingly.

Appendix A

Jikes RVM Modifications for JavaTryCatchWS

We have modified Jikes RVM to implement JavaTryCatchWS. This appendix contains a short description of all the relevant changes. We describe these modifications by relating it to the implementation details in this thesis.

A.1 Basic Infrastructure

a) `org.jikesrvm.CommandLineArgs` — We have added new command line arguments in this class to support the execution of JavaTryCatchWS (e.g., `-Xws:procs` to specify the total number of workers).
b) `org.jikesrvm.scheduler.MainThread.run()` — We modify this method to support the launch of work-stealing workers prior to the execution of the user `main` method.
c) `org.jikesrvm.runtime.Entrypoints` — This class contains the declaration of fields and methods of the virtual machine, which is required by the compiler-generated machine code or the C runtime code. We declare the work-stealing methods/fields (e.g., `WS.`**`join`** and `WS.`**`finish`**) in this class that falls into this category.

A.2 Leveraging Exception Handling Support (Section 4.4.2.1)

a) `org.jikesrvm.classloader.RVMType` — This class is the base of the Java type system. We declare work-stealing related special exceptions in this class (e.g., `WS.Continuation` and `WS.Finish`).
b) `org.jikesrvm.scheduler.WS` — We added this new class to implement **`public`** methods in JavaTryCatchWS (methods accessible from the user code). This class also contains the class declaration of the above exceptions.
b) `org.jikesrvm.classloader.TypeReference` — This class represents the reference in a class file to some type (class, primitive or array). We declare the references to the above exception classes in this class.

A.3 Initiation (Section 4.4.2.2)

a) `org.jikesrvm.scheduler.WS.searchForWork()` — We implemented this method for the thief to find a potential victim.
b) `org.jikesrvm.scheduler.WS.setFlag()` — This method is used by the victim for setting the steal flag.
c) `org.jikesrvm.scheduler.RVMThread.beginPairHandshake()` — Several Jikes RVM services (e.g., J.I.T. and garbage collection) use this method. In JavaTryCatchWS, a thief use this method to initiate the handshake with the victim. This method blocks until the victim has yielded.
d) `org.jikesrvm.scheduler.RVMThread.endPairHandshake()` — This method is used to end the thief and victim handshake.

A.4 State Management (Section 4.4.2.3)

a) `org.jikesrvm.scheduler.RVMThread.wsFindSteal()` — We implemented this method to enable the thief to stack walk victim's stack, and find the oldest continuation available for stealing; and the enclosing finish for this continuation.
b) `org.jikesrvm.compilers.common.CompiledMethod` — This abstract class represents a method, which has been compiled into machine code by one of the compilers in Jikes RVM. It stores the offset of instructions in the compiled machine code for the said method. This offset is changed by the thief every time it steals a continuation (pointing to catch block for the exception `WS.JoinFirst`). We added another field in this class, which represents this new offset. Aforementioned is always `NULL` by default.
c) `org.jikesrvm.scheduler.WS.installJoinInstructions()` — We implemented this method for the thief to install the above-mentioned offset.
d) `org.jikesrvm.scheduler.StackFrameCopier.copyStack()` — We implemented this method to enable the thief to copy stack frames from the victimâĂŹs stack to its stack.
e) `org.jikesrvm.mm.mminterface.GCMapIteratorGroup.copyRegisterValues()` — We implemented this method to copy the general purpose registers from the victim to the thief.
f) `org.jikesrvm.scheduler.StackFrameCopier.processFrameAndUpdateThread()` — We implemented this method to scan the copied stack frame and fix the object pointers.
g) `org.jikesrvm.scheduler.RVMThread.wsCompleteJoinInternal()` — We implemented this method to enable a victim to save its results in case of stolen continuation.
h) `org.jikesrvm.scheduler.WS.`**`finish`**`()` — We implemented this for the **`finish`** functionality and merging of computation results from the thief and the victim.

A.5 Termination (Section 4.4.2.4)

a) `org.jikesrvm.scheduler.WS.incFinish()/decFinish()` — We implemented these methods to atomically increment/decrement the finish counter.

A.6 Return Barrier Implementation (Section 5.3.1)

a) `org.jikesrvm.ia32.OutOfLineMachineCode .generateStackTrampolineBridgeInstructions()` — This method performs the stack trampoline bridge for implementing a return barrier.
b) `org.jikesrvm.scheduler.RVMThread.returnBarrier()` — This method is executed when the callee frame returns to the caller frame, which has been hijacked by the return barrier mechanism.
c) `org.jikesrvm.scheduler.RVMThread.wsStealInternal_retbarrier()` — This method is similar to `org.jikesrvm.scheduler.RVMThread.wsStealInternal` but we implemented this to support the case of return barriers.
d) `org.jikesrvm.scheduler.RVMThread.wsFindSteal_retBarrier()` — This method is similar to `org.jikesrvm.scheduler.RVMThread.wsFindSteal` but we implemented this to support the case of return barriers.
e) `org.jikesrvm.runtime.RuntimeEntrypoints.deliverException()` — This method is used by the Jikes RVM to deliver an exception to the current Java thread. We modified this to a) disable return barrier while delivering runtime exception; b) branch into the correct method if a join instruction is installed for a frame stolen by a thief.

A.7 Installing the First Return Barrier (Section 5.3.3)

a) `org.jikesrvm.scheduler.RVMThread.wsCloneVictimStack()` — We implemented this to allow the thief to clone the victim's stack, before installing the return barrier on this victim for the first time.
b) `org.jikesrvm.scheduler.RVMThread.wsInstallStackTrampolineBridge()` — This method installs the stack trampoline bridge at a given frame. It will hijack that frame, saving the hijacked return address and callee frame pointer in thread-local state to allow the execution of the hijacked frame later.

A.8 Synchronization Between Thief and Victim During Steal Process (Section 5.3.4)

a) `org.jikesrvm.scheduler.RVMThread.returnBarrier_internal()` — This is the kernel, which the victim executes after branching into the method `returnBarrier()`.
b) `org.jikesrvm.scheduler.RVMThread.wsUnlockFromReturnBarrier()` — We implemented this method to synchronize between a thief and the victim, whenever the

victim is waiting inside the return barrier. Thief uses this method to signal the waiting victim.

A.9 Victim Moves the Return Barrier (Section 5.3.5)

a) `org.jikesrvm.scheduler.RVMThread.deInstallStackTrampoline()` — This method is used to remove the return barrier from the victim's stack.
b) `org.jikesrvm.scheduler.RVMThread.resetTramoplineInfo()` — This method reinitializes the data-structures when a return barrier is removed from the victim's stack.
c) `org.jikesrvm.scheduler.RVMThread.getHijackedReturnAddress()` — This method returns the real (hijacked) return address of a frame that has been hijacked by the stack trampoline.

A.10 Stealing From a Victim with Return Barrier Pre-installed (Section 5.3.6)

a) `org.jikesrvm.mm.mmtk.ScanThread.scanThread()` — This is a very important method in Jikes RVM. It supports the scanning of thread stacks for object references during garbage collections. We have modified this method to support return barriers. While performing work-stealing with return barrier, thieves create a clone of victim's thread stack. This cloned stack contains references to live objects as long as the victim has return barrier installed on its main stack. Hence, while performing garbage collections, we need to scan both cloned stack and main stack so that we do not loose references to live objects. If only main stack were scanned, some of the live objects may appear dead and will be collected leading to inconsistencies. We modified this method to compute the roots by scanning both the cloned and main stack, as long as the return barrier is installed on the victim's main stack.

A.11 Disabling work-stealing within a synchronized code block (Section 6.3.2.1)

a) `org.jikesrvm.scheduler.WS.pauseStealOnThread()` — We implemented this method to disable work-stealing on the current thread. This method is executed by the victim whenever it executes synchronized statements.
b) `org.jikesrvm.scheduler.WS.resumeStealOnThread()` — We implemented this to enable work-stealing on the current thread. It is executed by the victim as soon as it finishes the execution of synchronized statements.

Bibliography

Acar, U. A.; Chargueraud, A.; and Rainey, M., 2013. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13 (Shenzhen, China, 2013), 219–228. ACM, New York, NY, USA. doi:10.1145/2442516.2442538. (cited on page 79)

Agha, G., 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA. ISBN 0-262-01092-5. (cited on pages 9 and 97)

Akka Team, 2013. Typesafe Inc. Akka. http://akka.io/. (cited on page 97)

Allen, E.; Chase, D.; Hallett, J.; Luchangco, V.; Maessen, J.-W.; Ryu, S.; Steele Jr, G. L.; Tobin-Hochstadt, S.; Dias, J.; Eastlund, C.; et al., 2005. The Fortress language specification. *Sun Microsystems*, 139 (2005), 140. (cited on page 6)

Alpern, B.; Attanasio, C. R.; Barton, J. J.; Burke, M. G.; Cheng, P.; Choi, J.-D.; Cocchi, A.; Fink, S. J.; Grove, D.; Hind, M.; Hummel, S. F.; Lieber, D.; Litvinov, V.; Mergen, M. F.; Ngo, T.; Russell, J. R.; Sarkar, V.; Serrano, M. J.; Shepherd, J. C.; Smith, S. E.; Sreedhar, V. C.; Srinivasan, H.; and Whaley, J., 2000. The Jalapeño virtual machine. *IBM Syst. J.*, 39, 1 (Jan. 2000), 211–238. doi:10.1147/sj.391.0211. (cited on pages 2 and 21)

Apache Lucene, 2008. Apache Lucene. http://lucene.apache.org/. (cited on page 24)

ARM Corporation, 2011. big.LITTLE processing. *ARM White Paper*, (2011). http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php. (cited on page 18)

Arnold, M.; Fink, S.; Grove, D.; Hind, M.; and Sweeney, P. F., 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '00 (Minneapolis, Minnesota, USA, 2000), 47–65. ACM, New York, NY, USA. doi:10.1145/353171.353175. (cited on pages 18, 19, and 22)

Arnold, M.; Hind, M.; and Ryder, B. G., 2002. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02 (Seattle, Washington, USA, 2002), 111–129. ACM, New York, NY, USA. doi:10.1145/582419.582432. (cited on page 19)

Arora, N. S.; Blumofe, R. D.; and Plaxton, C. G., 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98 (Puerto Vallarta, Mexico, 1998), 119–129. ACM, New York, NY, USA. doi:10.1145/277651.277678. (cited on page 79)

Aycock, J., 2003. A brief history of Just-In-Time. *ACM Comput. Surv.*, 35, 2 (Jun. 2003), 97–113. doi:10.1145/857076.857077. (cited on page 19)

Bacon, D. F.; Konuru, R.; Murthy, C.; and Serrano, M., 1998. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98 (Montreal, Quebec, Canada, 1998), 258–268. ACM, New York, NY, USA. doi:10.1145/277650.277734. (cited on page 22)

Balakrishnan, S.; Rajwar, R.; Upton, M.; and Lai, K., 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, 506–517. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ISCA.2005.51. (cited on page 18)

Berenbrink, P.; Friedetzky, T.; and Goldberg, L. A., 2003. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32, 5 (May 2003), 1260–1279. doi:10.1137/S0097539701399551. (cited on page 79)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA, Oct. 2006), 169–190. ACM Press, New York, NY, USA. doi:http://doi.acm.org/10.1145/1167473.1167488. (cited on page 24)

Blumofe, R. D. and Leiserson, C. E., 1999. Scheduling multithreaded computations by work stealing. *J. ACM*, 46, 5 (Sep. 1999), 720–748. doi:10.1145/324133.324234. (cited on page 79)

Bohr, M., 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Newsletter, IEEE*, 12, 1 (2007), 11–13. http://dx.doi.org/10.1109/N-SSC.2007.4785534. (cited on page 1)

Borkar, S. and Chien, A. A., 2011. The future of microprocessors. *Commun. ACM*, 54, 5 (May 2011), 67–77. doi:10.1145/1941487.1941507. (cited on page 18)

Bornstein, D., 2008. Dalvik VM internals. https://sites.google.com/site/io/dalvik-vm-internals. Accessed June 2013. (cited on page 18)

Box, D. and Pattison, T., 2002. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN

0201734117. (cited on page 18)

Cao, T.; Blackburn, S. M.; Gao, T.; and McKinley, K. S., 2012. The yin and yang of power and performance for asymmetric hardware and managed software. *SIGARCH Comput. Archit. News*, 40, 3 (Jun. 2012), 225–236. doi:10.1145/2366231.2337185. (cited on page 18)

Carter, J. B.; Bennett, J. K.; and Zwaenepoel, W., 1991. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91 (Pacific Grove, California, USA, 1991), 152–164. ACM, New York, NY, USA. doi:10.1145/121132.121159. (cited on page 6)

Cavé, V.; Zhao, J.; Shirako, J.; and Sarkar, V., 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11 (Kongens Lyngby, Denmark, 2011), 51–61. ACM, New York, NY, USA. doi:10.1145/2093157.2093165. (cited on pages 1 and 6)

Chamberlain, B.; Callahan, D.; and Zima, H., 2007. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21, 3 (Aug. 2007), 291–312. doi:10.1177/1094342007078442. (cited on pages 1, 6, and 97)

Chambers, C. and Ungar, D., 1991. Making pure object-oriented languages practical. In *Conference proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91 (Phoenix, Arizona, USA, 1991), 1–15. ACM, New York, NY, USA. doi:10.1145/117954.117955. (cited on pages 18 and 19)

Chambers, C.; Ungar, D.; and Lee, E., 1989. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89 (New Orleans, Louisiana, USA, 1989), 49–70. ACM, New York, NY, USA. doi:10.1145/74877.74884. (cited on page 19)

Chambers, C. D., 1992. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-oriented Programming Languages*. Ph.D. thesis, Stanford University, Stanford, CA, USA. (cited on page 19)

Charles, P.; Grothoff, C.; Saraswat, V.; Donawa, C.; Kielstra, A.; Ebcioglu, K.; von Praun, C.; and Sarkar, V., 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '05 (San Diego, CA, USA, 2005), 519–538. ACM, New York, NY, USA. doi:10.1145/1094811.1094852. (cited on pages 1, 2, and 6)

Chien, A. A., 1996. ICC++ – a C++ dialect for high performance parallel computing. *SIGAPP Appl. Comput. Rev.*, 4, 1 (Apr. 1996), 19–23. doi:10.1145/240732.240740. (cited on page 9)

Cierniak, M.; Lueh, G.-Y.; and Stichnoth, J. M., 2000. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00 (Vancouver, British Columbia, Canada, 2000), 13–26. ACM, New York, NY, USA. doi:10.1145/349299.349306. (cited on page 20)

Cong, G.; Kodali, S.; Krishnamoorthy, S.; Lea, D.; Saraswat, V.; and Wen, T., 2008. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, 536–545. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ICPP.2008.88. (cited on pages 79 and 101)

Dagum, L. and Menon, R., 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5, 1 (Jan. 1998), 46–55. doi:10.1109/99.660313. (cited on page 6)

Deutsch, L. P. and Schiffman, A. M., 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84 (Salt Lake City, Utah, USA, 1984), 297–302. ACM, New York, NY, USA. doi:10.1145/800017.800542. (cited on page 18)

Dinan, J.; Larkins, D. B.; Sadayappan, P.; Krishnamoorthy, S.; and Nieplocha, J., 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09 (Portland, Oregon, 2009), 53:1–53:11. ACM, New York, NY, USA. doi:10.1145/1654059.1654113. (cited on pages 79 and 101)

Dolby, J.; Hammer, C.; Marino, D.; Tip, F.; Vaziri, M.; and Vitek, J., 2012. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34, 1 (May 2012), 4:1–4:48. doi:10.1145/2160910.2160913. (cited on pages xvii, 3, 10, 82, 83, 84, and 88)

Duran, A.; Teruel, X.; Ferrer, R.; Martorell, X.; and Ayguade, E., 2009. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, 124–131. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ICPP.2009.64. (cited on page 24)

Durillo, J.; Nebro, A.; and Alba, E., 2010. The jMetal framework for multi-objective optimization: Design and architecture. In *CEC 2010*, 4138–4325. Barcelona, Spain. (cited on page 24)

Durillo, J. J. and Nebro, A. J., 2011. jmetal: A java framework for multi-objective optimization. *Adv. Eng. Softw.*, 42, 10 (Oct. 2011), 760–771. doi:10.1016/j.advengsoft.2011.05.014. (cited on page 24)

Ekman, T., 2004. *Rewritable reference attributed grammars, design, implementation, and applications*. Ph.D. thesis, Lund University. (cited on page 88)

Ekman, T. and Hedin, G., 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07 (Montreal, Quebec, Canada, 2007), 1–18. ACM, New York, NY, USA. doi:10.1145/1297027.1297029. (cited on pages 25 and 88)

El-Ghazawi, T. and Smith, L., 2006. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06 (Tampa, Florida, 2006). ACM, New York, NY, USA. doi:10.1145/1188455.1188483. (cited on page 6)

Fink, S. J. and Qian, F., 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03 (San Francisco, California, 2003), 241–252. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=776261.776288. (cited on pages 19 and 22)

Frampton, D.; Blackburn, S. M.; Cheng, P.; Garner, R. J.; Grove, D.; Moss, J. E. B.; and Salishev, S. I., 2009. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09 (Washington, DC, USA, 2009), 81–90. ACM, New York, NY, USA. doi:10.1145/1508293.1508305. (cited on page 21)

Frigo, M., 2007. Multithreaded programming in Cilk. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07 (London, Ontario, Canada, 2007), 13–14. ACM, New York, NY, USA. doi:10.1145/1278177.1278181. (cited on page 25)

Frigo, M.; Leiserson, C. E.; and Randall, K. H., 1998a. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, PLDI '98 (Montreal, Quebec, Canada, 1998), 212–223. ACM, New York, NY, USA. doi:10.1145/277650.277725. (cited on pages 1, 8, 24, and 56)

Frigo, M.; Prokop, H.; Frigo, M.; Leiserson, C.; Prokop, H.; Ramachandran, S.; Dailey, D.; Leiserson, C.; Lyubashevskiy, I.; Kushman, N.; et al., 1998b. The Cilk project. *Algorithms*, (1998). (cited on pages 8 and 11)

Ghemawat, S. and Menage, P., 2009. TCMalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. (cited on page 25)

Goodenough, J. B., 1975. Exception handling: issues and a proposed notation. *Commun. ACM*, 18, 12 (Dec. 1975), 683–696. doi:10.1145/361227.361230. (cited on page 20)

Google, 2013. V8 JavaScript engine. http://code.google.com/p/v8/. Accessed June 2013. (cited on page 19)

Gosling, J.; Joy, B.; and Steele, G. L., 1996. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn.

ISBN 0201634511. (cited on page 18)

Guo, Y., 2010. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. Ph.D. thesis, Rice University. (cited on page 16)

Guo, Y.; Barik, R.; Raman, R.; and Sarkar, V., 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, 1–12. IEEE Computer Society, Washington, DC, USA. doi:10.1109/IPDPS.2009.5161079. (cited on pages 16 and 25)

Guo, Y.; Zhao, J.; Cave, V.; and Sarkar, V., 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10 (Bangalore, India, 2010), 341–342. ACM, New York, NY, USA. doi:10.1145/1693453.1693504. (cited on pages 16 and 25)

Gupta, M.; Choi, J.-D.; and Hind, M., 2000. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, 422–446. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=646157.679855. (cited on page 20)

Hendler, D. and Shavit, N., 2002. Non-blocking steal-half work queues. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02 (Monterey, California, 2002), 280–289. ACM, New York, NY, USA. doi:10.1145/571825.571876. (cited on page 79)

Hiraishi, T.; Yasugi, M.; Umatani, S.; and Yuasa, T., 2009. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09 (Raleigh, NC, USA, 2009), 55–64. ACM, New York, NY, USA. doi:10.1145/1504176.1504187. (cited on pages 56 and 57)

Hoffmann, R. and Rauber, T., 2011. Adaptive task pools: efficiently balancing large number of tasks on shared-address spaces. *International Journal of Parallel Programming*, 39, 5 (2011), 553–581. (cited on page 79)

Hölzle, U.; Chambers, C.; and Ungar, D., 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92 (San Francisco, California, USA, 1992), 32–43. ACM, New York, NY, USA. doi:10.1145/143095.143114. (cited on pages 19, 20, and 80)

Hölzle, U. and Ungar, D., 1994. A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the 9th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '94 (Portland, Oregon, USA, 1994), 229–243. ACM, New York, NY, USA. doi:10.1145/191080.191116. (cited on page 19)

IBM, 2002. Power 4 the first multi-core, 1GHz processor. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/. (cited on page 5)

IBM Research, 2012. X10: Performance and productivity at scale. http://x10-lang.org/software/download-x10/release-list.html?id=210. (cited on page 23)

Imam, S. M. and Sarkar, V., 2012. Integrating task parallelism with actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12 (Tucson, Arizona, USA, 2012), 753–772. ACM, New York, NY, USA. doi:10.1145/2384616.2384671. (cited on page 97)

Intel Corporation, 1997. Using the RDTSC instruction for performance monitoring. http://www.intel.com.au/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. (cited on page 64)

Intel Corporation, 2013. Intel® many integrated core architecture (intel® MIC architecture) – advanced. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. (cited on page 5)

Ishizaki, K.; Kawahito, M.; Yasue, T.; Komatsu, H.; and Nakatani, T., 2000. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00 (Minneapolis, Minnesota, USA, 2000), 294–310. ACM, New York, NY, USA. doi:10.1145/353171.353191. (cited on page 20)

jai Min, S.; Iancu, C.; and Yelick, K., 2011. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*. (cited on page 101)

Jeffers, J., 2013. Intel® Xeon Phi® Coprocessors. In *Modern Accelerator Technologies for Geographic Information Science*, 25–39. Springer. (cited on page 5)

Jones, R. and King, A., 2005. A fast analysis for thread-local garbage collection with dynamic class loading. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, 129–138. doi:10.1109/SCAM.2005.1. (cited on page 101)

Kambadur, P.; Gupta, A.; Ghoting, A.; Avron, H.; and Lumsdaine, A., 2009. PFunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09 (Portland, Oregon, 2009), 43:1–43:11. ACM, New York, NY, USA. doi:10.1145/1654059.1654103. (cited on page 56)

Kawachiya, K.; Koseki, A.; and Onodera, T., 2002. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02 (Seattle, Washington, USA, 2002), 130–141. ACM, New York, NY, USA. doi:10.1145/582419.582433. (cited on page 22)

KELEHER, P.; COX, A. L.; DWARKADAS, S.; AND ZWAENEPOEL, W., 1994. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94 (San Francisco, California, 1994), 10–10. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1267074.1267084. (cited on page 6)

KLIOT, G.; PETRANK, E.; AND STEENSGAARD, B., 2009. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09 (Washington, DC, USA, 2009), 11–20. ACM, New York, NY, USA. doi:10.1145/1508293.1508296. (cited on page 80)

KOUFATY, D.; REDDY, D.; AND HAHN, S., 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10 (Paris, France, 2010), 125–138. ACM, New York, NY, USA. doi:10.1145/1755913.1755928. (cited on page 18)

KULKARNI, M.; BURTSCHER, M.; CASCAVAL, C.; AND PINGALI, K., 2009. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 65–76. doi:10.1109/ISPASS.2009.4919639. (cited on page 24)

KUMAR, V.; BLACKBURN, S.; AND DOLBY, J., 2014a. AJWS: Performance and productivity for Java via data-centric atomicity and work-stealing. In *Under review*. (cited on page 23)

KUMAR, V.; BLACKBURN, S.; AND GROVE, D., 2014b. Friendly barriers: Efficient work-stealing with return barriers. In *Under review*. (cited on pages 23 and 61)

KUMAR, V.; FRAMPTON, D.; BLACKBURN, S. M.; GROVE, D.; AND TARDIEU, O., 2012. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12 (Tucson, Arizona, USA, 2012), 297–314. ACM, New York, NY, USA. doi:10.1145/2384616.2384639. (cited on pages 23 and 27)

LEA, D., 1999. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. ISBN 0201310090. (cited on page 17)

LEA, D., 2000. A Java Fork/Join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00 (San Francisco, California, USA, 2000), 36–43. ACM, New York, NY, USA. doi:10.1145/337449.337465. (cited on pages 1, 8, 17, 24, and 25)

LEA, D., 2004. JSR 166 Introduction. http://gee.cs.oswego.edu/dl/jsr166/dist/docs/. (cited on page 7)

LEIJEN, D.; SCHULTE, W.; AND BURCKHARDT, S., 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object*

Oriented Programming Systems Languages and Applications, OOPSLA '09 (Orlando, Florida, USA, 2009), 227–242. ACM, New York, NY, USA. doi:10.1145/1640089.1640106. (cited on pages 1, 8, and 56)

Li, T.; Brett, P.; Knauerhase, R.; Koufaty, D.; Reddy, D.; and Hahn, S., 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 1–12. doi:10.1109/HPCA.2010.5416660. (cited on page 18)

Lindholm, T. and Yellin, F., 1999. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. ISBN 0201432943. (cited on page 20)

Lüling, R. and Monien, B., 1993. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93 (Velen, Germany, 1993), 164–172. ACM, New York, NY, USA. doi:10.1145/165231.165252. (cited on page 79)

Marlow, S. and Peyton Jones, S., 2011. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory Management*, ISMM '11 (San Jose, California, USA, 2011), 21–32. ACM, New York, NY, USA. doi:10.1145/1993478.1993482. (cited on page 101)

Martelli, A., 2003. *Python in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 0596001886. (cited on page 18)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3, 4 (Apr. 1960), 184–195. doi:10.1145/367177.367199. (cited on page 19)

Meyerovich, L. A. and Rabkin, A. S., 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13 (Indianapolis, Indiana, USA, 2013), 1–18. ACM, New York, NY, USA. doi:10.1145/2509136.2509515. (cited on page 82)

Mitzenmacher, M., 1998. Analyses of load stealing models based on differential equations. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98 (Puerto Vallarta, Mexico, 1998), 212–221. ACM, New York, NY, USA. doi:10.1145/277651.277687. (cited on page 79)

Mohr, E.; Kranz, D. A.; and Halstead, R. H., Jr., 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90 (Nice, France, 1990), 185–197. ACM, New York, NY, USA. doi:10.1145/91556.91631. http://doi.acm.org/10.1145/91556.91631. (cited on page 8)

Morad, T. Y.; Weiser, U. C.; Kolodny, A.; Valero, M.; and Ayguade, E., 2006. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Comput. Archit. Lett.*, 5, 1 (Jan. 2006), 4–17. doi:10.1109/L-CA.2006.6. (cited on page 18)

Nebro, A. J. and Durillo, J. J., 2013. jMetal. http://jmetal.sourceforge.net/. (cited on page 24)

Numrich, R. W. and Reid, J., 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17, 2 (Aug. 1998), 1–31. doi:10.1145/289918.289920. (cited on page 6)

Olivier, S.; Huan, J.; Liu, J.; Prins, J.; Dinan, J.; Sadayappan, P.; and Tseng, C.-W., 2007. UTS: an unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06 (New Orleans, LA, USA, 2007), 235–250. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1757112.1757137. (cited on page 24)

Onodera, T. and Kawachiya, K., 1999. A study of locking objects with bimodal fields. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99 (Denver, Colorado, USA, 1999), 223–237. ACM, New York, NY, USA. doi:10.1145/320384.320405. (cited on page 22)

Oracle, 2013. The Java® virtual machine specification. http://docs.oracle.com/javase/specs/jvms/se7/html/. Accessed June 2013. (cited on page 18)

Oracle Corporation, 2013. Java™ Platform, Standard Edition 7 API Specification. http://docs.oracle.com/javase/7/docs/api/. Accessed January 2014. (cited on page 17)

Owicki, S. and Agarwal, A., 1989. Evaluating the performance of software cache coherence. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III (Boston, Massachusetts, USA, 1989), 230–242. ACM, New York, NY, USA. doi:10.1145/70082.68204. (cited on page 6)

Paleczny, M.; Vick, C.; and Click, C., 2001. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01 (Monterey, California, 2001), 1–1. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1267847.1267848. (cited on page 19)

Petito, M., 2007. Eclipse refactoring. *http://people. clarkson. edu/˜ dhou/courses/EE564-s07/Eclipse-Refactoring. pdf*, 5 (2007), 2010. (cited on page 88)

Petzold, C., 2010. *Programming Microsoft® Windows® with C#*. Microsoft Press. (cited on page 20)

PGAS, 2011. Partitioned Global Address Space. http://www.pgas.org/. (cited on page 6)

Pizlo, F.; Frampton, D.; and Hosking, A. L., 2011. Fine-grained adaptive biased locking. In *Proceedings of the 9th International Conference on Principles and Practice* *of Programming in Java*, PPPJ '11 (Kongens Lyngby, Denmark, 2011), 171–181. ACM, New York, NY, USA. doi:10.1145/2093157.2093184. (cited on page 22)

Platt, D. S., 2002. *Introducing Microsoft .NET, Second Edition*. Microsoft Press, Redmond, WA, USA, 2nd edn. ISBN 0735615713. (cited on page 2)

Raman, R., 2009. *Compiler support for work-stealing parallel runtime systems*. Ph.D. thesis, Rice University. (cited on page 17)

Raman, R.; Zhao, J.; Sarkar, V.; Vechev, M.; and Yahav, E., 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12 (Beijing, China, 2012), 531–542. ACM, New York, NY, USA. doi:10.1145/2254064.2254127. (cited on page 97)

Ramsey, N. and Peyton Jones, S., 2000. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00 (Vancouver, British Columbia, Canada, 2000), 285–298. ACM, New York, NY, USA. doi:10.1145/349299.349337. (cited on page 20)

Reinders, J., 2007. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. ISBN 9780596514808. (cited on pages 8 and 56)

Reinders, J., 2010. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc. (cited on page 1)

Rettig, M., 2013. Jetlang: Message based concurrency for Java. http://code.google.com/p/jetlang. (cited on page 97)

Rigo, A. and Pedroni, S., 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06 (Portland, Oregon, USA, 2006), 944–953. ACM, New York, NY, USA. doi:10.1145/1176617.1176753. (cited on pages 18 and 21)

Rogers, R.; Lombardo, J.; Mednieks, Z.; and Meike, B., 2009. *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., 1st edn. ISBN 0596521472, 9780596521479. (cited on page 18)

Rudolph, L.; Slivkin-Allalouf, M.; and Upfal, E., 1991. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91 (Hilton Head, South Carolina, USA, 1991), 237–245. ACM, New York, NY, USA. doi:10.1145/113379.113401. (cited on page 79)

Sable McGill, 2012. Soot. http://www.sable.mcgill.ca/soot/. (cited on page 26)

Saiki, H.; Konaka, Y.; Komiya, T.; Yasugi, M.; and Yuasa, T., 2005. Real-time GC in JeRTy^{TM} VM using the return-barrier method. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, 140–148. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ISORC.2005.45. (cited on page 80)

Sanchez, D.; Yoo, R. M.; and Kozyrakis, C., 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV (Pittsburgh, Pennsylvania, USA, 2010), 311–322. ACM, New York, NY, USA. doi:10.1145/1736020.1736055. (cited on page 79)

Saraswat, V.; Almasi, G.; Bikshandi, G.; Cascaval, C.; Cunningham, D.; Grove, D.; Kodali, S.; Peshansky, I.; and Tardieu, O., 2010. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*. (cited on page 6)

Saraswat, V.; Bloom, B.; Peshansky, I.; Tardieu, O.; and Grove, D., 2011. X10 language specification. (cited on page 97)

Saraswat, V.; Bloom, B.; Peshansky, I.; Tardieu, O.; and Grove, D., 2013. X10 language specification version 2.4. Technical report, IBM. http://x10.sourceforge.net/documentation/languagespec/x10-240.pdf. (cited on page 6)

Sasada, K., 2005. YARV: yet another RubyVM: innovating the ruby interpreter. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05 (San Diego, CA, USA, 2005), 158–159. ACM, New York, NY, USA. doi:10.1145/1094855.1094912. (cited on page 18)

Schilling, J. L., 1998. Optimizing away C++ exception handling. *SIGPLAN Not.*, 33, 8 (Aug. 1998), 40–47. doi:10.1145/286385.286390. (cited on page 20)

Simon, D. and Cifuentes, C., 2005. The squawk virtual machine: Java^{TM} on the bare metal. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05 (San Diego, CA, USA, 2005), 150–151. ACM, New York, NY, USA. doi:10.1145/1094855.1094908. (cited on page 21)

Snir, M.; Otto, S. W.; Walker, D. W.; Dongarra, J.; and Huss-Lederman, S., 1995. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA. ISBN 0262691841. (cited on page 6)

Srinivasan, S. and Mycroft, A., 2008. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08 (Paphos, Cypress, 2008), 104–128. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-70592-5_6. (cited on page 97)

Strumpen, V., 1998. Indolent closure creation. Technical report, Cambridge, MA, USA. (cited on pages 56 and 57)

Suganuma, T.; Yasue, T.; Kawahito, M.; Komatsu, H.; and Nakatani, T., 2001. A dynamic optimization framework for a Java Just-In-Time compiler. In *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01 (Tampa Bay, FL, USA, 2001), 180–195. ACM, New York, NY, USA. doi:10.1145/504282.504296. (cited on page 19)

Suleman, M. A.; Mutlu, O.; Qureshi, M. K.; and Patt, Y. N., 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV (Washington, DC, USA, 2009), 253–264. ACM, New York, NY, USA. doi:10.1145/1508244.1508274. (cited on page 18)

Sundaresan, V.; Maier, D.; Ramarao, P.; and Stoodley, M., 2006. Experiences with multi-threading and dynamic class loading in a Java Just-In-Time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, 87–97. IEEE Computer Society, Washington, DC, USA. doi:10.1109/CGO.2006.16. (cited on pages 19 and 20)

Tardieu, O.; Wang, H.; and Lin, H., 2012. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12 (New Orleans, Louisiana, USA, 2012), 267–276. ACM, New York, NY, USA. doi:10.1145/2145816.2145850. (cited on pages 16, 25, 35, 58, and 89)

Taura, K.; Tabata, K.; and Yonezawa, A., 1999. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99 (Atlanta, Georgia, USA, 1999), 60–71. ACM, New York, NY, USA. doi:10.1145/301104.301110. (cited on page 56)

Umatani, S.; Yasugi, M.; Komiya, T.; and Yuasa, T., 2003. Pursuing laziness for efficient implementation of modern multithreaded languages. In *High Performance Computing* (Eds. A. Veidenbaum; K. Joe; H. Amano; and H. Aiso), vol. 2858 of *Lecture Notes in Computer Science*, 174–188. Springer Berlin Heidelberg. ISBN 978-3-540-20359-9. doi:10.1007/978-3-540-39707-6_13. (cited on pages 56, 57, and 80)

Ungar, D.; Spitz, A.; and Ausch, A., 2005. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05 (San Diego, CA, USA, 2005), 11–20. ACM, New York, NY, USA. doi:10.1145/1094855.1094865. (cited on page 21)

Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P.; and Sundaresan, V., 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99 (Mississauga, Ontario, Canada, 1999), 13–. IBM Press. http://dl.acm.org/citation.cfm?id=781995.782008. (cited on page 26)

Vaziri, M.; Tip, F.; and Dolby, J., 2006. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06 (Charleston, South Carolina, USA, 2006), 334–345. ACM, New York, NY, USA. doi:10.1145/1111037.1111067. (cited on pages 2, 3, 10, and 82)

Venkatesh, G.; Sampson, J.; Goulding, N.; Garcia, S.; Bryksin, V.; Lugo-Martinez, J.; Swanson, S.; and Taylor, M. B., 2010. Conservation cores: reducing the energy of mature computations. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV (Pittsburgh, Pennsylvania, USA, 2010), 205–218. ACM, New York, NY, USA. doi:10.1145/1736020.1736044. (cited on page 18)

Wall, L. and Schwartz, R. L., 1991. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 0-937175-64-1. (cited on page 18)

Wendykier, P., 2011. JTransforms. http://sourceforge.net/projects/jtransforms/. (cited on page 25)

Wilson, G. V., 1996. *Parallel Programming Using C++*. MIT Press, Cambridge, MA, USA. ISBN 0262731185. (cited on page 97)

Wimmer, C.; Haupt, M.; Van De Vanter, M. L.; Jordan, M.; Daynès, L.; and Simon, D., 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9, 4 (Jan. 2013), 30:1–30:24. doi:10.1145/2400682.2400689. (cited on page 21)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J. B.; and McKinley, K. S., 2011. Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11 (Portland, Oregon, USA, 2011), 307–324. ACM, New York, NY, USA. doi:10.1145/2048066.2048092. (cited on pages 24 and 53)

Yelick, K.; Semenzato, L.; Pike, G.; Miyamoto, C.; Liblit, B.; Krishnamurthy, A.; Hilfinger, P.; Graham, S.; Gay, D.; Colella, P.; and Aiken, A., 1998. Titanium: A high-performance Java dialect. In *In ACM*, 10–11. (cited on page 6)

Yonezawa, A. (Ed.), 1990. *ABCL: An Object-oriented Concurrent System*. MIT Press, Cambridge, MA, USA. ISBN 0-262-24029-7. (cited on pages 9 and 97)

Yuasa, T., 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11, 3 (Mar. 1990), 181–198. doi:10.1016/0164-1212(90)90084-Y. (cited on page 80)

Yuasa, T.; Nakagawa, Y.; Komiya, T.; and Yasugi, M., 2002. Return barrier. In *Proceedings of the International Lisp Conference*. (cited on pages 20, 65, and 80)