# Integrating Asynchronous Task Parallelism and Data-centric Atomicity *

Vivek Kumar †
Rice University

Julian Dolby
IBM T.J. Watson Research

Stephen M. Blackburn
Australian National University

## ABSTRACT

Processor design has turned toward parallelism and heterogeneous cores to achieve performance and energy efficiency. Developers find high-level languages attractive as they use abstraction to offer *productivity* and *portability* over these hardware complexities. Over the past few decades, researchers have developed increasingly advanced mechanisms to deliver *performance* despite the overheads naturally imposed by this abstraction. Recent work has demonstrated that such mechanisms can be exploited to attack overheads that arise in emerging high-level languages, which provide strong abstractions over parallelism. However, current implementation of existing popular high-level languages, such as Java, offer little by way of abstractions that allow the developer to achieve performance in the face of extensive hardware parallelism.

In this paper, we present a small set of extensions to the Java programming language that aims to achieve both high performance and high productivity with minimal programmer effort. We incorporate ideas from languages like X10 and AJ to develop five annotations in Java for achieving *asynchronous task parallelism* and *data-centric concurrency control*. These annotations allow the use of a highly efficient implementation of a work-stealing scheduler for task parallelism. We evaluate our proposal by refactoring classes from a number of existing multithreaded open source projects to use our new annotations. Our results suggest that these annotations significantly reduce the programming effort while achieving performance improvements up to 30% compared to conventional approaches.

* This work was supported by IBM and ARC LP0989872. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

† Work done while the author was affiliated with Australian National University.



*PPPJ'16, August 29-30, 2016, Lugano,Switzerland*

 ISBN 978-1-4503-4135-6/16/08…$15.00

DOI: http://dx.doi.org/...

## CCS Concepts

•**Software and its engineering → Concurrent programming structures; Source code generation; Runtime environments;**

## Keywords

Task Parallelism, Work-Stealing, Data-centric Atomicity, Jikes RVM

## 1. INTRODUCTION

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, which together have put a stop to hardware delivering ever faster sequential performance. 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 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. X10 [13] is one such recently developed parallel programming language that provides `async-finish` based annotations to the users to specify the tasks that can run asynchronously. Other closely related programming models are ICC++ [15], Cilk [20], and Habanero-Java [11, 24]. All these implementations employ a work-stealing scheduler [9] within the underlying language runtime that schedules fine-grained packets of work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. However, a drawback of this approach is the runtime overheads due to over decomposition of tasks. As a common work-around, programmer chooses a granularity size, which can stop the creation of new tasks. Again, choosing a right granularity size for each `async-finish` block in a large codebase can be a daunting task.

Further, to ensure the correctness of parallel program, it is important that threads of execution do not interfere with each other while sharing data in the memory (data-race). Traditional approaches to avoid data-race follow *control-centric* approaches where the instruction sequences are pro-

tected with synchronization operations. This is a burden on the programmer, as he has to ensure that all shared data are consistently protected. In a large code-base, this control-centric approach can significantly decrease the productivity and can lead to data-races that are frustratingly difficult to track down and eliminate. To remove this burden from the programmer, Atomic Sets for Java (AJ) introduced a *data-centric* approach for synchronization [17]. In this model, the user simply groups fields of objects into atomic sets to specify that these objects must always be updated atomically. AJ then uses the Eclipse Refactoring [32] tool to automatically generate synchronization statements at all relevant places. The basic idea behind atomic sets is simple: the reason we need synchronization at all is that some sets of fields have some consistency property that must be maintained. The atomic sets model abstracts away from what these properties are, and has them declared explicitly. While AJ has been extensively studied in several prior works [36, 17, 37, 31, 22, 26], it is still not open-sourced. Moreover, all prior studies on AJ have been done only by using explicit Java threading for expressing parallelism.

Our principal contribution is an *open-sourced* new system, *AJWS*[1]—Atomic Java with Work-Stealing, which by drawing together the benefits of work-stealing and AJ, allows the application programmer to conveniently and succinctly expose the parallelism inherent in their program in a mainstream object-oriented language. We have implemented AJWS by using JastAdd meta-compilation system [18]. AJWS uses a highly scalable and low-overhead implementation of a work-stealing runtime [28, 27], which relieves the programmer in choosing the granularity of each parallel section.

In summary, the contributions of this paper are as following:

- We introduce a new parallel programming model (AJWS) that provide a set of five annotations to bring together the benefits of work-stealing and data-centric approach for synchronization.
- We provide compiler transformation of AJWS annotations to vanilla Java that uses a recently developed highly scalable implementation of a work-stealing runtime.
- We demonstrate the benefits of AJWS by modifying three large open-sourced Java applications and by evaluating the performance on a quad-core smartphone, to evaluate AJWS on mobile devices that are increasingly multicore.

The rest of the paper is structured as follows. We start by giving a motivating analysis in Section 2 to show the benefits of our new AJWS parallel programming model. In Section 3 we provide a background on AJWS's work-stealing runtime. Section 4 explains the design and implementation of AJWS. Section 5 discusses our evaluation methodology for AJWS. Section 6 discusses the improvements to productivity and performance by using AJWS. Section 7 discusses the related work and finally Section 8 concludes the paper with some directions to future work.

[1]https://github.com/vivkumar/ajws

# 2. MOTIVATION

Pundits predict that for the next twenty years processor designs will use large-scale parallelism with heterogeneous cores to control power and energy [10]. 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. Java is a high-level language that has gained huge popularity. As evident from the TIOBE [6] index for June 2016, Java is on number one spot in the programming language community.

However, mainstream languages such as Java do not provide support for parallel programming that is both easy to use and efficient. Figure 1 shows a bank account transaction code written both in plain Java and AJWS. The `class Bank` carries out money transfer between a pair of sender and receiver accounts. The `class Transfer` contains the detail of sender and receiver accounts. Multiple transfers might be taking place to/from the same account. The other operation performed by `class Bank` is adding interest to each account.

## 2.1 Bank transaction in plain Java

Figure 1(a) shows a plain Java pseudocode for this bank program. The methods `processTransfer` (Line 35) and `addInterest` (Line 41) contain codes that can exploit parallelism. Different techniques that the user can choose for parallelizing are (a) partitioning the `for` loops (Line 36 and 42) among threads, (b) using Java's concurrent utilities for parallelizing these `for` loops, and (c) using `async-finish` constructs from Habanero-Java. The first two approaches incur significant syntactic overheads with respect to the sequential version of the same code. Moreover, in all the above approaches the programmer must control the granularity of the parallel executions to avoid runtime overheads. To ensure consistency, `class Account` requires a `lock` object to access any member methods (e.g., method `addInterest` at Line 15). Further, to ensure proper accounting in case of inter-account transfer (Line 22), both the participating accounts should be locked before the transfer actually happens (Line 24). However, Line 24 can create a potential deadlock. Suppose there is a thread `T1` doing transfer from account `A` to `B` and another thread `T2` is doing transfer from account `B` to `A`. At Line 24, `T1` locks `A` and `T2` locks `B`. Next, `T1` try locking `B` and `T2` try locking `A`, thereby creating a deadlock.

## 2.2 Bank transaction using AJWS

To minimize programming complexities, our AJWS system provides the user a set of five annotations to ensure race free concurrency (syntax and semantics are explained in Section 4.1). Figure 1(b) shows the AJWS version of the code in Figure 1(a). AJWS's `async-finish` annotations are used inside methods `processTransfer` (Line 28) and `addInterest` (Line 35) for using task-parallelism. An `async` construct launches an asynchronous task that can execute in parallel to other `async` tasks. To synchronize on `async`, a `finish` construct is used. Control does not return from inside a `finish`, until all transitively launched `async` within the scope of the `finish` have terminated. The user need not worry about formulating correct granularity, as our AJWS system perform load-balancing of these `async` tasks by using the TryCatchWS work-stealing runtime from Kumar et. al., which has extremely low runtime overheads (Section 3). To ensure consistency, AJWS reuses the data-centric concur-

```
class Account {
  double balance; long id;
  Lock l;
  Account(i) {
    id = i;
    l = new ReentrantLock();
  }
  int getId() {return id;}
  void lock() {l.lock();}
  void unlock() {l.unlock();}
  // user must take lock
  // before calling any method
  void credit(amount) { ... }
  void debit(amount) { ... }
  void addInterest() {
    lock(); ...; unlock();
  }
  ...
}
class Transfer {
  Account a1, a2;
  void run() {
    // potential deadlock
    a1.lock(); a2.lock();
    a1.debit(amount);
    a2.credit(amount);
    a1.unlock(); a2.unlock();
  }
  ...
}
class Bank {
  Transfer[] transfers;
  Account[] accounts;
  // how parallelize? granularity?
  void processTransfer() {
    for(int i=0; i<total; i++) {
      transfers[i].run();
    }
  }
  // how parallelize? granularity?
  void addInterest() {
    for(int i=0; i<total; i++) {
      accounts[i].addInterest();
    }
  }
  ...
}
```

(a) A sample scenario that can lead to deadlock. Here, user ensures atomicity by calling methods of `class Account` within `lock()` and `unlock()`

```
class Account {
  @AtomicSet(A);
  @Atomic(A) double balance;
  long id;
  Account(i) {
    id = i;
  }
  int getId() {return id;}
  // It is now AJWS's resposibility
  // for proper synchronizations
  void credit(amount) { ... }
  void debit(amount) { ... }
  void addInterest() { ... }
  ...
}
class Transfer {
  Account a1, a2;
  @Atomic(a1.A) @Atomic(a2.A)
  void run() {
    a1.debit(amount);
    a2.credit(amount);
  }
  ...
}
class Bank {
  Transfer[] transfers;
  Account[] accounts;
  void processTransfer() {
    finish {
      for(int i=0; i<total; i++) {
        async {transfers[i].run();}
      }
    }
  }
  void addInterest() {
    finish {
      for(int i=0; i<total; i++) {
        async {accounts[i].addInterest();}
      }
    }
  }
  ...
}
```

(b) Programming using our new AJWS system, where it is AJWS's responsibility to ensure proper synchronizations

**Figure 1: Pseudocode in Java for a concurrent bank account transactions. (complete code available at [1])**

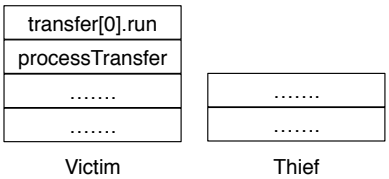
rency control mechanism introduced by Atomic Sets programming model for Java (AJ) [36, 17, 37]. In this model, the programmer specifies that sets of object fields share some consistency property, without specifying what the property may be. This differs from traditional object oriented models in two key ways: the first is that the programmer specifies all consistency, 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. In Figure 1(b), the `class Account` declares a single Atomic Set `A` using the `@AtomicSet` annotation (Line 2) and specifies that the field `balance` belong to Atomic Set `A` using the `@Atomic` annotation (Line 3). The field `id` is fixed for each `Account`, hence its not annotated as `@Atomic` access (Line 4). For the case of inter-account transfer (Line 19), programmer simply uses `@Atomic` annotations on the method `run` to declare that this method is an additional unit of work for the Atomic Sets `a1.A` and `a2.A` (Line 18). Our AJWS annotations ensure that the calls to method `run` never deadlocks. It also ensures that the calls to method `addInterest` (Line 13) are always synchronized. We explain our implementation in Section 4.

# 3. TryCatchWS WORK-STEALING RUNTIME FOR AJWS

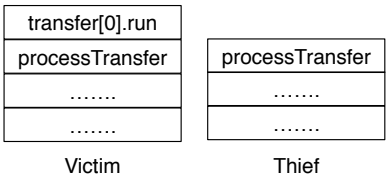
In our new AJWS system, we translate the `async-finish` annotations to use Kumar et. al.'s TryCatchWS work-stealing framework [28, 27] that was originally designed for the Java backend of X10 language [33]. To perform load-balancing of these asynchronous tasks, a work-stealing runtime is employed. Work-stealing [9] is a popular load balancing technique for dynamic task-parallelism. It maintains a pool of *workers*, each of which maintains a double-ended queue (*deque*) of tasks and continuations. When a local deque becomes empty, the worker becomes a *thief* and seeks a *victim* from which to *steal* work.

Although the specific details vary among the various implementations of work-stealing schedulers, they all incur some form of sequential overhead—the increase in single-core execution time due to transformations that expose parallelism [28]. As observed by Kumar et. al. [28], this overhead could be as high as 2× to 12× over the sequential implementation. Programmers can reduce these overheads to some extent by stopping the creation of new `async` tasks beyond certain granularity. However, choosing a right granularity for multiple occurrences of `async-finish` blocks in a large codebase is extremely difficult. To reduce these sequential overheads, Kumar et. al. implemented the TryCatchWS work-stealing runtime for the Java backend of X10 language. This work-stealing runtime is implemented within the Jikes RVM [8] Java runtime. They reuse several existing mechanisms in managed runtimes to reduce the sequential overheads to just 15%. They demonstrated that by using this same mechanism, TryCatchWS also achieves very low dynamic overhead—an overhead that increases with core count and is most evident when parallelism is greatest [27]. Managed runtime features used by Kumar et. al. are yieldpoint mechanism [30], on-stack-replacement [19], dynamic code-patching [34], exception delivery mechanism [21], and return barriers [23].

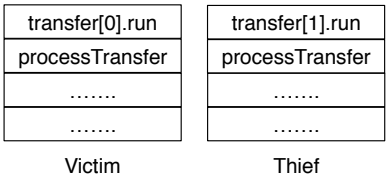
The TryCatchWS system re-writes X10's `async-finish` into regular Java and exploiting the semantics Java offers for exception handling, which is very efficiently implemented in most modern JVMs. The result is that the runtime can walk a victim's execution stack and identify all `async` and `finish` contexts. TryCatchWS uses the Java thread stack of both the victim and thief as their implicit deque. TryCatchWS follows the work-first principal for task scheduling, which is similar to Cilk [20]. Under this policy, a victim executes an `async` task and leaves the continuation to be stolen and executed by the thief. Figure 2 shows the state of the execution stack for both victim and thief when the thief is trying to steal the continuation of the `async` at Line 31 in Figure 1(b). In the Figure 2(a), victim follows the work-first policy and starts executing the method `run` for the first iteration (`i=0`). A thief queries the runtime and find that this victim has steal-able task. It then requests the runtime to stop the victim (using yieldpoint mechanism), so that it may safely walk the victim's execution stack. It then copies the stack frame `processTransfer` from victim's execution stack on to its own stack (Figure 2(b)). It copies frame `processTransfer` because it contains the steal-able task (continuation to an `async`). The thief then runs a modified version of the runtime's exception delivery code to start this stolen task (Figure 2(c)). In this case it is the call to the method `run` for the next iteration (`i=1`). For a detailed



(a) Initial stack states when the victim is executing the method `run` for the first iteration (`i=0`)



(b) Victim is stopped using yieldpoint mechanism and the thief copies stack frame `processTransfer`



(c) Victim is released from yieldpoint and the thief throws a special exception to start executing method `run` for the next iteration (`i=1`)

**Figure 2: Execution stack states of the victim and thief. The victim is executing the `async` at Line 31 in Figure 1(b) and a thief is stealing the continuation of this `async`. Stack growth is from the bottom to the top.**

overview of TryCatchWS we refer readers to [28, 27].

# 4. IMPLEMENTATION

Section 2.1 identified shortcomings in current implementations of 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. 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 work-stealing and data-centric concurrency control respectively, and then bring those together into a single, simple model within Java that combines data-centric concurrency control with a high performance work-stealing implementation. We call this new parallel programming model AJWS — Atomic Java with Work-Stealing. Section 2.2 describes a simple usecase of AJWS.

We now discuss the design and implementation of AJWS.

## 4.1 Annotations in AJWS

AJWS provides three annotations for data-centric concurrency control and two annotations for expressing parallelism with work-stealing. These annotations are: a) `@AtomicSet`, b) `@Atomic`, c) `@AliasAtomic`, d) `finish`, and e) `async`. Syntax and usage of these annotations (except `@AliasAtomic`) is shown in Figure 1(b) by using a bank transaction program.

The `@AtomicSet(A)` is used to declare a new Atomic Set in a `class` or an `interface`. Currently we support only one Atomic Set declaration inside a `class` (Line 2) or `interface`. Extending AJWS to support more than one Atomic Set per class is only a software engineering effort and not a limitation of our approach. We would extend AJWS in the future for supporting multiple Atomic Set declarations inside a `class`. However, due to this current restriction, if the `class` inherits an Atomic Set, it is not allowed to declare a new Atomic Set.

The `@Atomic(A)` annotation is allowed on instance fields and classes. A field can belong to at most one Atomic Set (Line 3). Annotated fields can only be accessed from the `this` reference. In AJWS, the `@Atomic` annotation subsumes the role of AJ's `unifor(A)` annotation. Hence, `@Atomic` annotation can also be used to annotate a method (or its arguments). This declares the method to be an additional unit of work for the specified Atomic Set in the argument object (Line 18).

The `@AliasAtomic(A=this.B)` annotation could be applied on variable declarations and constructor calls. This unifies the Atomic Set `A` in the annotated variable or constructed object with the current object's Atomic Set `B`. In Figure 3 we show a sample use case of `@AliasAtomic` annotation. With the annotation `@AliasAtomic(A=this.A)` on the field `parent` and the constructor parameter `p`, AJWS ensures that both the `parent` and `LinkedAccount` instance are associated with the same lock (`LinkedAccount` is inheriting Atomic Set `A` from `Account class`).

The `async-finish` computation (Section 3) in AJWS is a terminally strict computation. Hence, the parent `async` is allowed to terminate before its child `async`. The current implementation of AJWS limits `async` call only on method calls and `for` loops. AJWS does not permit the use of `async-finish` blocks within an atomic section (both directly or indirectly via a method call).

```
1  class LinkedAccount extends Account{
2    @AliasAtomic(A=this.A) Account parent;
3    LinkedAccount(@AliasAtomic(A=this.A) Account p){
4      parent = p;
5    }
6    void debit(amount) {
7      parent.debit(amount);
8    }
9    ...
10 }
```

**Figure 3: Using AJWS's `@AliasAtomic` annotation inside a `LinkedAccount class` that inherits from the `Account class` shown in Figure 1(b)**

## 4.2 Translating AJWS to Java

AJ [17] annotations for data-centric atomicity were expressed as Java comments in the program, which were translated to Java using Eclipse refactoring. Expressing AJ annotations as comments is not very expressive and also Eclipse refactoring is a heavy process. To avoid these shortcomings, we use JastAdd [18], an extensible Java compiler to perform the translation of our 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. The semantics of AJWS's annotations for data-centric atomicity are very close to those of AJ, so we follow a similar pattern in performing our rewrites. We perform the Java rewrite via the JastAdd AST classes for `VariableDeclaration`, `ClassDeclaration`, `TypeDeclaration`, `MethodAccess`, `ConstructorDeclaration`, `Block`, and `MethodDeclaration`.

### *4.2.1 Translating Annotations for Data-centric Atomicity*

Figure 4 shows the translation of AJWS code in Figure 1(b) to vanilla Java. The `Account class` in Figure 1(b) declares an `@AtomicSet`, hence in Figure 4 it declares a lock field `_lockA` of type `OrderedLock` (Line 4). Constructor of `Account class` is transformed to take lock object as an extra parameter (Line 6). Classes that inherit `Account class` (e.g., `LinkedAccount` in Figure 3), the lock object is passed to `Account class` constructor. If a `class` declares or inherits an Atomic Set (or if a method has some `@Atomic` annotation, e.g., Line 18 in Figure 1(b)), two versions of each of its method are generated. The first version has the default name, whereas the second version has the suffix `_internal` that is called only when AJWS is sure that all needed locks are already acquired. If the method performs atomic accesses of any fields, the default version of method uses proper `synchronized` blocks (e.g., `credit` method at Line 18 in Figure 4). In case of non-atomic access, `synchronized` is not required (e.g., `getId` method at Line 15 in Figure 4). For detailed overview of translating data-centric annotations to Java, we refer readers to [17].

```
1  import ajws.Atomic;
2  import ajws.OrderedLock;
3  class Account implements Atomic {
4   final OrderedLock _lockA;
5   double balance; long id;
6   Account(i,l) {
7    id=i; _lockA=l;
8   }
9   Account(i) {
10   this(i, new OrderedLock());
11  }
12  OrderedLock getLockObj() {
13   return _lockA;
14  }
15  int getId() {return id;}
16  int getId_internal() {return id;}
17  void credit(amount) {
18   synchronized(_lockA) { ... }
19  }
20  void credit_internal(amount){...}
21  void debit(amount) {
22   synchronized(_lockA) {...}
23  }
24  void debit_internal(amount){...}
25  void addInterest(amount) {
26   synchronized(_lockA) {...}
27  }
28  void addInterest_internal(){...}
29 }
```

```
30 class Transfer {
31  Account a1, a2;
32  void run() {
33   OrderedLock l1=null,l2=null;
34   OrderedLock l3=a2.getLockObj();
35   OrderedLock l4=a1.getLockObj();
36   if (l3.index() > l4.index()){
37    l1 = l3; l2 = l4;
38   } else {
39    l1 = l4; l2 = l3;
40   }
41   synchronized(l1) {
42    synchronized(l2) {
43     a1.debit(amount);
44     a2.credit(amount);
45    }
46   }
47  }
48  void run_internal() {
49   a1.debit(amount);
50   a2.credit(amount);
51  }
52  ...
53 }
```

```
54 class Bank {
55  Transfer[] transfers;
56  void processTransfer() {
57   try {
58    for(int i=0;i<total;i++){
59     try {
60      RT.continuationAvail();
61      transfers[i].run();
62      RT.checkIfStolen();
63     }
64     catch(ExceptionEntryThief c)
65     { }
66    }
67    RT.doFinish();
68   }catch(ExceptionFinish f) { }
69  }
70   ...
71 }
```

**Figure 4: Translation of AJWS code in Figure 1(b) to vanilla Java**

In case a critical section involves locking on multiple lock objects (e.g., `run` method at Line 32 in Figure 4), AJWS ensures that all locks must be acquired without introducing deadlock. This is achieved by enforcing an ordering among lock objects (similar to AJ). Each lock object is given a unique id in `OrderedLock`, and locks are always acquired in the order of increasing id (Lines 34 to 42). AJ does not detail about its implementation of the `OrderedLock`, hence in AJWS we used Java `ReentrantLock` (`static` field) to ensure consistency while assigning unique id to each lock objects.

#### 4.2.2 *Translating Work-Stealing Annotations*

AJWS performs the translation of `async-finish` annotation to vanilla Java in much the same way as Try-CatchWS [28]. In Figure 4, we show this translation for the `async-finish` blocks in the method `processTransfer` (Line 56). The runtime (`RT`) method `continuationAvail` marks the availability of a continuation (Line 60). Thief who steals the continuation, throws `ExceptionEntryThief` (a special exception) to start the stolen continuation (Line 64). The runtime call `doFinish` performs synchronization of all the spawned `async` (Line 67). The special exception `ExceptionFinish` is used only to store the partial results from those `async` that can return values (Line 68).

## 5. METHODOLOGY

Before presenting the evaluation of AJWS, we first describe our experimental methodology.

### 5.1 Benchmarks

Because our goal is to show productivity with performance, we have targeted open-sourced applications with large codebases that addresses real world problems. Our three benchmarks are (available online [1]):

**jMetal** It is a Java-based framework for multi-objective optimization with metaheuristic techniques [2]. It provides several sets of classes, which can be used as the building blocks of multi-objective techniques. The Default implementation of jMetal offers limited parallelism. We have parallelized the entire benchmark, but for performance comparison (total 6 parallel sections) we only use two of their algorithms, for which the Default implementation of jMetal also offers parallelism. We are using the release 4.4 of jMetal. Default jMetal uses `java.util.concurrent.Executer` and Java threads for parallelism.

**JTransforms** This is a multithreaded FFT library written in Java [3] and there are many open source projects (including applications for mobile devices) that use JTransforms internally. This library offers benchmarks, which we have used for the performance comparison of total 186 parallel sections. We are using the release 2.4 of JTransforms. Default implementation of JTransforms uses `java.util.concurrent.Future` for parallelism.

**SJXP** Simple-Java-XML-Parser is a XML parser build for Android OS and other Java platforms [4]. The Default implementation of SJXP is sequential. It also provides a benchmark for testing SJXP's performance in parsing XML files (one after another). For performance comparison we use this same benchmark for parsing a total of 56 XML files of different sizes. We chose SJXP to represent XML workloads on mobile. We are using the release 2.2.

In our evaluations, we are using the sequential version of each of the above benchmarks as the baseline case. To get the plain sequential Java version, we have removed all concurrency control (`synchronized` keywords) from the default implementation of benchmarks. We have also removed all code that expresses parallelism and converted each benchmark to plain sequential Java.

## 5.2 Hardware Platform

Performance evaluation of TryCatchWS (as managed X10 backend) has already been performed in the past on a wide variety of multicore processors. These study also included comparison with several existing work-stealing implementations (e.g., Cilk, Java fork/join, managed X10 and Habanero-Java) [28, 27]. To further demonstrate the effectiveness of TryCatchWS (now with our new AJWS backend) on modern multicore mobile devices, we performed all experiments on a quad-core ASUS ZenFone 2 (ZE551ML) smartphone having Android OS (version 5). The processor was Intel Atom Z3580 running at 2.3 GHz. We used Linux Deploy mobile application to install Ubuntu (64 bit) version 15.10 on this smartphone.

## 5.3 Software Platform

**Jikes RVM** Version hg_11181. We used the a production build. This is used as the Java runtime. A fixed heap size of 1 GB and single garbage collector thread is used across all experiments. Other than this, we preserve the default settings of Jikes RVM.

**TryCatchWS** This is the Java work-stealing implementation from Kumar et al. [28]. We have ported their implementation in Jikes RVM Java runtime version hg_11181. Also available online [5].

**JastAdd** This is an implementation of an extensible compiler for Java [18]. 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 uses JastAdd internally (eg. Soot [35]).

## 5.4 Measurements

For each benchmark, we ran fifteen invocations, with six iterations per invocation where each iteration performed the kernel of the benchmark. We report the final iterations, along with a 95% confidence interval based on a Student t-test. We report the total execution time in all experiments (including time for garbage collection).

# 6. RESULTS

We now evaluate both the programmatic and performance impact of AJWS. We begin the evaluation by measuring the reduction in programming effort, both with respect to parallelism and atomicity. We then evaluate the parallel performance of AJWS.

## 6.1 Evaluation of AJWS's productivity

We start our evaluation by evaluating the syntactic overhead associated with AJWS compared to existing alternatives for Java. Figure 5 shows the syntactic overhead of parallelism and atomicity constructs in all three benchmarks, both for the default case and AJWS implementations. We removed all parallelism related code in Default versions to get the corresponding sequential version (except SJXP that was already sequential). Syntactic overhead in both Default and AJWS version is the difference in LOC (Lines-Of-Code) from the sequential version. Introducing parallelism in traditional ways can bloat the code. From Figure 5, we can see this is true for Default version of jMetal and JTransforms that has a syntactic overhead of 1.2% and 14.3% respectively. Default jMetal and JTransforms has a total of 6 and 372 parallel sections respectively. Due to simplicity of AJWS's `async-finish` block, where the programmer need not induce extra code for controlling granularity, we are able to introduce 7× more parallelism in jMetal with just 0.1% syntactic overhead. Even with extremely large number of parallel sections in JTransforms, AJWS version has just 3.7% syntactic overhead. To parallelize SJXP, we implemented a new `class` (80 lines) for parallel parsing of XML files, which resulted in slightly higher syntactic overhead (6.5%).

Default jMetal has a total of 10 `synchronized` blocks for just 6 parallel sections. AJWS version of jMetal is able to achieve this same effect with just 3 `@Atomic` annotations for 7× more parallelism. Recall, that for declaring any field as `@Atomic`, user first need to declare the corresponding `@AtomicSet`. We were able to parallelize SJXP using just 2 `@Atomic` annotations.

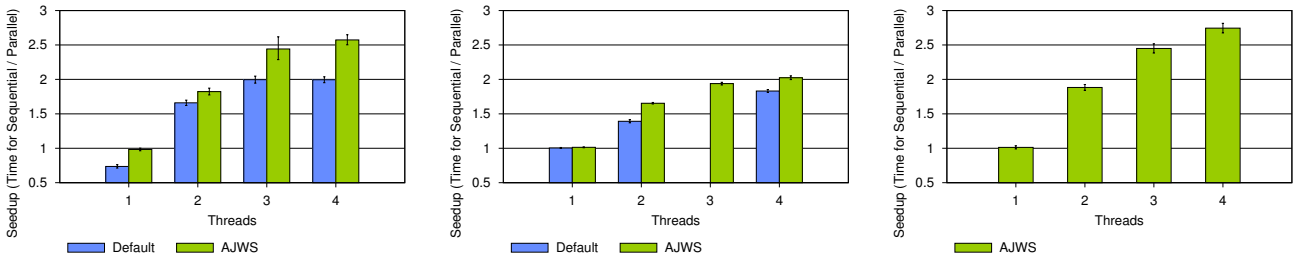
## 6.2 Evaluation of AJWS's performance

We now turn to performance evaluation of AJWS. Figure 6(a), Figure 6(b) and Figure 6(c) shows the speedup (over sequential version) for jMetal, JTransforms, and SJXP respectively. In this evaluation we are using the same amount of parallelism in both Default and AJWS versions. Recall that by using TryCatchWS runtime, AJWS relieves the programmer from worrying about setting correct granularity size for `async-finish` computation. Due to this, none of the AJWS version of our benchmarks is controlling the granularity of `async-finish` blocks. To evaluate the effect of this, we consider the single thread speedup of AJWS. We noticed that single thread execution of AJWS performed very similar to the sequential execution (for all benchmarks), thereby supporting our claim that AJWS removes the need for granularity control. Default version of JTransforms strictly controls the granularity in all 372 parallel section, hence its single thread execution performs similar to its sequential version. This is not true for Default jMetal as it does not controls the granularity in its 6 parallel sections (resulting in 25% slowdown with single thread). The existing mechanism for granularity control in Default JTransforms restricts its execution with only an even number of threads. However, as the AJWS version of JTransforms does not uses granularity control, it can be executed with any number of threads. Overall, AJWS performed better than Default version with any number of threads (for each benchmark). With four threads, compared to the Default, AJWS is 30% and 10% faster for jMetal and JTransforms, respectively.

These results demonstrate that AJWS 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. We also show that a high performance implementation of work-stealing can significantly boost *performance* relative to existing parallel Java implementations. Java is also the most widely used language for mobile application development. Modern mobile devices are becoming increasingly multicore and pose a significant challenge for writing high performance Java applications using traditional approaches. Our above performance evaluation of AJWS on a low-end multicore smartphone, suggests that it is a very simple platform for performing high performance parallel programming on mobile devices.

| | Sequential | | Default | | | AJWS | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | Files | LOC | synchs | ‖ism | Effort | `@AtomicSet` | `@Atomic` | ‖ism | Effort |
| **jMetal** | 329 | 28,216 | 10 | 6 | 1.2% | 2 | 3 | 47 | 0.1% |
| **JTransforms** | 45 | 42,756 | 0 | 372 | 14.3% | 0 | 0 | 372 | 3.7% |
| **SJXP** | 17 | 1,250 | – | – | – | 1 | 2 | 1 | 6.5% |

**Figure 5: This table compares the productivity of Default and AJWS versions of each benchmark. LOC is generated using David A. Wheeler's 'SLOCCount' [38]. LOC overhead ("Effort") is the difference between sequential and parallel version. "synchs" is the total number of `synchronized` blocks used in Default version. ‖ism denotes total number of parallel blocks. In case of AJWS, each ‖ism denotes a pair of `async-finish` block. In AJWS version of jMetal, we are able to introduce more parallelism. JTransforms do not have any atomic block. The Default version of SJXP is sequential, hence it does not have any entry in the table. In AJWS version, we only show the annotations that we have used in the benchmark.**




(a) jMetal (b) JTransforms (c) SJXP

**Figure 6: Speedup of AJWS over the Sequential Java version on a quad-core smartphone. Here, we have used equal amount of parallelism across both versions of benchmarks. Speedup with 3 threads for Default JTransforms is missing as it only support even number of threads**

## 7. 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 [7] was instantiated in many languages, such as the ABCL [40] family, and there have been numerous efforts to blend object-oriented features with parallelism (see, for instance, this survey that covers just C++[39]). This work has received new impetus due to increasing hardware parallelism, giving rise to recent high-level languages like X10 [13] and Chapel [12]. However, while some of these languages attempted to interoperate with existing languages, they have not become mainstream. In contrast, our focus 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.

In writing parallel programs, one of the most serious challenges is coordinating access to the shared data among threads. For concurrency control, most widely used techniques in high-level languages like Java and C# are mutual-exclusion locks [16]. These are control-centric approaches that do not guarantee deadlock freedom. Transactional Memory (TM) is a famous control-centric approach that offers a simpler alternative to mutual exclusion by shifting the burden of correct synchronization from a programmer to the TM system [29]. TM guarantee lock-free and deadlock avoidance by employing a rollback in case of transactional conflicts. However, TM has its own side effects as these rollbacks can degrade the performance. Lock inference [14] is another control-centric approach that combines compiler transformations and runtime techniques in building deadlock free critical sections. Our AJWS system differs from all these approaches as it builds upon AJ's data-centric approach for concurrency control that guarantees deadlock freedom [31].

Most closely related to our own work is Habanero-Java [24, 11]. It has a pure based library implementation that can run on top of any JVM. It provides Java 8 lambda based APIs to expose variety of asynchronous tasks, and uses a scalable work-stealing runtime for load-balancing of these tasks. Similar to X10, Habanero-Java is also prone to sequential overheads. Users can avoid these overheads by controlling the task granularity. This is not mandatory in the case of AJWS as it uses the TryCatchWS work-stealing runtime that has extremely low sequential overheads [28]. More recently, Habanero-Java utilized the benefits of rollbacks in transactions and shared-exclusive locks, to provide an object-based isolation technique that could resolve deadlocks at runtime [25]. This is also a control-centric approach approach that differs from the data-centric approach in AJWS.

## 8. CONCLUSION AND FUTURE WORK

To exploit the parallelism from increasing number of cores on processors, it is important to find better abstractions for expressing and writing parallel computations. In this paper we focused on this goal and presented a set of five annotations for the popular Java language, for writing parallel programs with high productivity and high performance. We implemented these annotations in a new AJWS system that builds upon two recent developments: a) a data-centric ap-

proach to concurrency control; and b) a very efficient work-stealing implementation designed specifically for managed runtimes. We evaluated the syntactic overhead of AJWS by modifying three large open-sourced applications and evaluated its performance on a multicore mobile device. Our approach significantly lowers the syntactic overhead of exposing parallelism, and delivers performance improvements up to 30% compared to conventional approaches.

There are several directions for the future work. We plan to provide annotations in AJWS to expose data-parallelism that could be executed over a graphical processing unit (GPU). Today almost every mobile device contain GPUs. Providing a special GPU centric annotation in AJWS would greatly help in mobile application development. Using AJWS, we plan to study how work-stealing affects the energy consumption in power-critical mobile devices.

# 9. REFERENCES


[1] AJWS. *https://github.com/vivkumar/ajws*.

[2] jMetal. *http://jmetal.sourceforge.net/*.

[3] JTransforms. *https://sites.google.com/site/piotrwendykier/software/jtransforms*.

[4] Simple-Java-XML-Parser. *https://github.com/thebuzzmedia/simple-java-xml-parser*.

[5] TryCatchWS. *https://github.com/vivkumar/TryCatchWS*.

[6] Tiobe index for ranking the popularity of programming languages. *http://www.tiobe.com/tiobe_index*, June 2016.

[7] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[8] B. Alpern, , S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005.

[9] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[10] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[11] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. 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, pages 51–61, New York, NY, USA, 2011. ACM.

[12] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 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, pages 519–538, New York, NY, USA, 2005. ACM.

[14] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM.

[15] A. A. Chien. ICC++-A C++ dialect for high performance parallel computing. volume 4, pages 19–23, New York, NY, USA, Apr. 1996. ACM.

[16] B. Demsky and P. Lam. Views: Synthesizing fine-grained concurrency control. *ACM Trans. Softw. Eng. Methodol.*, 22(1):4:1–4:33, Mar. 2013.

[17] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4:1–4:48, May 2012.

[18] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[19] S. J. Fink and F. Qian. 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, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[21] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, Dec. 1975.

[22] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 231–240, New York, NY, USA, 2008. ACM.

[23] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[24] S. Imam and V. Sarkar. Habanero-java library: A java 8 framework for multicore programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 75–86, New York, NY, USA, 2014. ACM.

[25] S. Imam, J. Zhao, and V. Sarkar. Euro-par 2015: Parallel processing: 21st international conference on parallel and distributed computing, vienna, austria, august 24-28, 2015, proceedings. chapter A Composable Deadlock-Free Approach to Object-Based Isolation, pages 426–437. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[26] N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding

concurrency-related bugs using random isolation. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 198–213, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] V. Kumar, S. M. Blackburn, and D. Grove. Friendly barriers: Efficient work-stealing with return barriers. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 165–176, New York, NY, USA, 2014. ACM.

[28] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM.

[29] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.

[30] Y. Lin, K. Wang, S. M. Blackburn, A. L. Hosking, and M. Norrish. Stop and go: Understanding yieldpoint behavior. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 70–80, New York, NY, USA, 2015. ACM.

[31] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 322–331, Piscataway, NJ, USA, 2013. IEEE Press.

[32] M. Petito. Eclipse refactoring. *http://people. clarkson. edu/~ dhou/courses/EE564-s07/Eclipse-Refactoring. pdf*, 5:2010, 2007.

[33] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification, 2011.

[34] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. 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, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[36] M. Vaziri, F. Tip, and J. Dolby. 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, pages 334–345, New York, NY, USA, 2006. ACM.

[37] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 304–328, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] D. A. Wheeler. SLOCCount. *http://www.dwheeler.com/sloccount/*, 2001.

[39] G. V. Wilson. *Parallel Programming Using C++*. MIT Press, 1996.

[40] A. Yonezawa, editor. *ABCL: An Object-oriented Concurrent System*. MIT Press, 1990.