Vivek Kumar[1], Julian Dolby[2], Stephen M Blackburn[3]

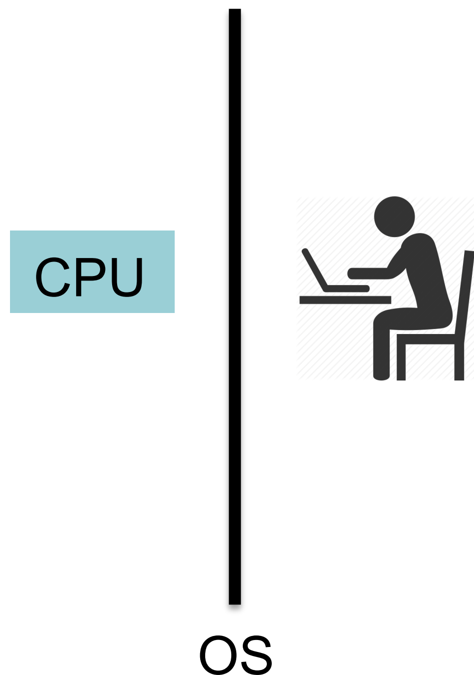
1 Rice University (Work done during affiliation with Australian National University)

2 IBM T.J. Watson Research

3 Australian National University

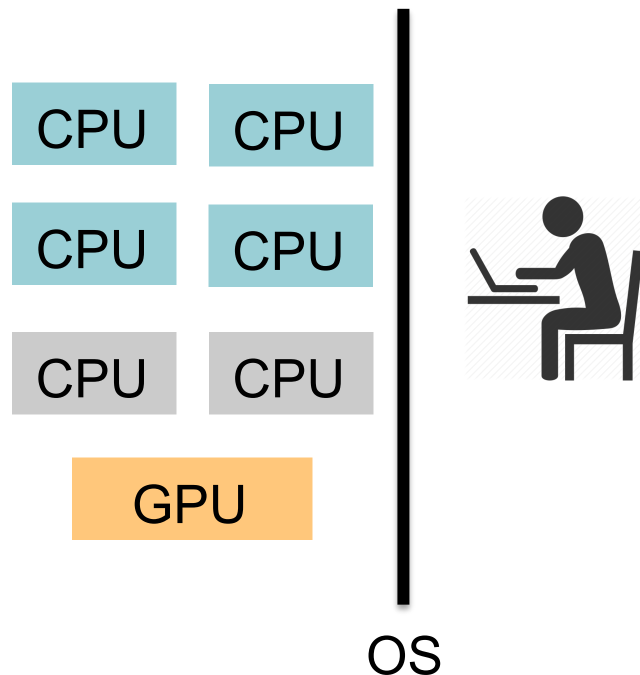
Hardware and Software Yesterday

1990s, early 2000



"What Andy giveth,
Bill taketh away"

Hardware and Software Today



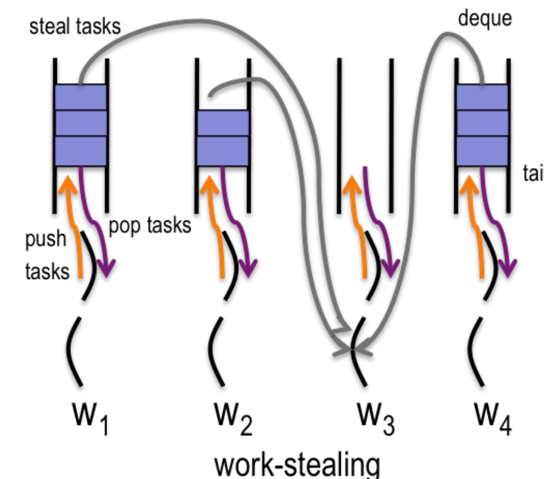
"The Free Lunch is Now Over!"

The 3P Challenge

- Productivity
- Performance
- Portability

Options ?

- Productivity
  - “Doing” lots of things at once (parallelism)
    - Threads, fork/join(Java), async-finish (X10, HJ)
  - “Dealing” with lots of things at once (concurrency)
    - locks, transactions, isolated (HJ)
- Performance
  - Work–stealing scheduling
- Portability
  - Managed runtime to hide hardware complexities



Productivity and Performance Challenge

- Parallelism
  - Expose "right" amount of parallelism
- Concurrency
  - "Control-centric" approach might lead to data races and deadlocks

Contributions

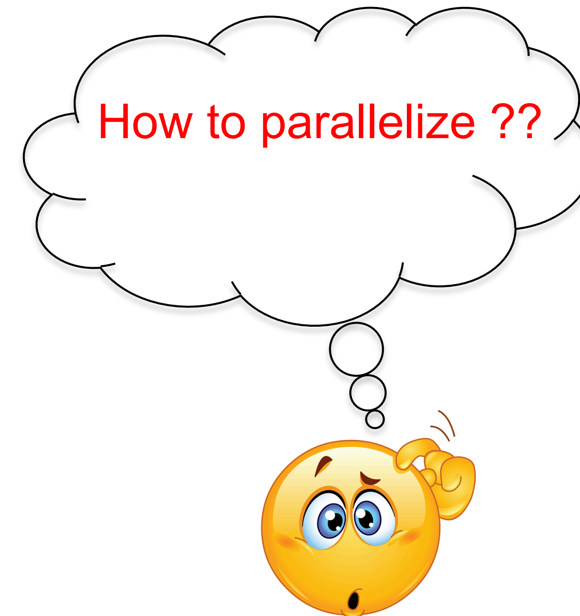
- ✔ Atomic Java with Work-Stealing (AJWS)
  - A new parallel programming model
- ✔ Annotations to expose task parallelism and data-centric concurrency control
  - Compiler transformation of AJWS to vanilla Java that uses a highly efficient work-stealing runtime implemented directly inside a JVM
- ✔ Detailed performance study
  - Using three large open-source Java applications and evaluating the performance on a multicore smartphone
- ✔ Results
  - That shows that AJWS improves both productivity and performance over conventional approaches

Motivating Analysis

Bank Transaction

```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}



}
```

```
class Bank {
    void interest() {
        Account[N] a;
          for (i=0; i<N; i++) {

                a[i].addInterest();

         }}
```



Bank Transaction

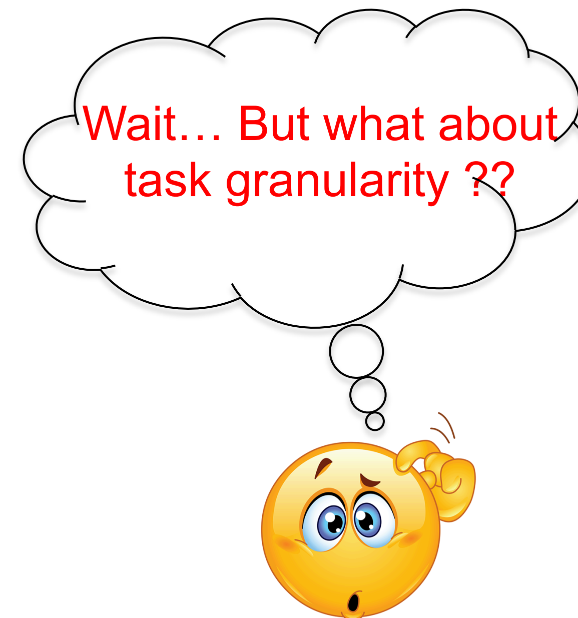
```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}



}
```

```
class Bank {
    void interest() {
        Account[N] a;
  finish for (i=0; i<N; i++) {
    async {
            a[i].addInterest();

        }}}
```

Bank Transaction

```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}



}
```

```
class Bank {
    void interest() {
        Account[N] a;
    finish for (i=0; i<N; i++) {
      async {
            a[i].addInterest();

        }}}
```

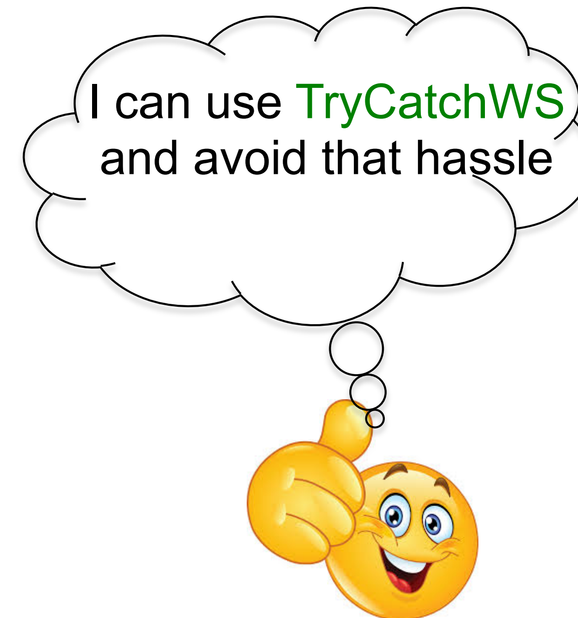


Bank Transaction

```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}



}
```

```
class Bank {
    void interest() {
        Account[N] a;
   finish for (i=0; i<N; i++) {
     async {
              a[i].addInterest();

         }}}
```

- TryCatchWS
  - Kumar et. al., OOPSLA 2012
  - JVM support for work-stealing for **X10**
  - Extremely low overheads

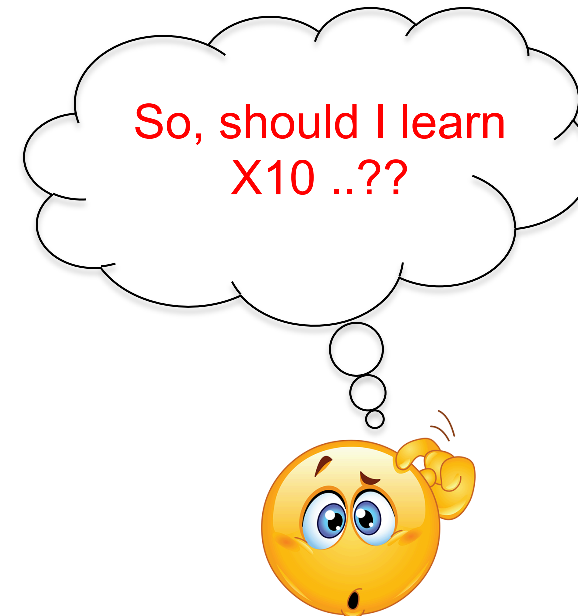


Bank Transaction

```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}


}
```

```
class Bank {
    void interest() {
        Account[N] a;
   finish for (i=0; i<N; i++) {
     async {
              a[i].addInterest();

       }}}
```

- TryCatchWS
  - Kumar et. al., OOPSLA 2012
  - JVM support for work-stealing for **X10**
  - Extremely low overheads



Bank Transaction

```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}
    void debit(amount) {  ... }
    void credit(amount) { ... }
}
```

```
class Transfer {
    Account from, to;
    int amount;

    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
        for (i=0; i<N; i++) {
            t[i].run();
        }
    }
}
```

```
void interest() {
    Account[N] a;
    for (i=0; i<N; i++) {

        a[i].addInterest();

    }
}
```

Bank Transaction

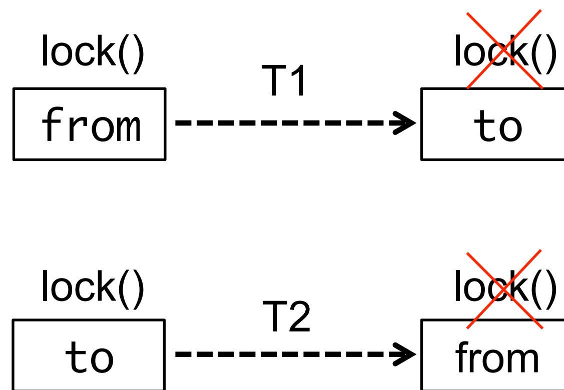
```
class Account {

    int balance;
    public Account(b) { ... }
    void interest() { ...}
    void debit(amount) {  ... }
    void credit(amount) { ... }
}
```

```
class Transfer {
    Account from, to;
    int amount;

    void run() {
     from.lock(); to.lock()
        from.debit(amount);
        to.credit(amount);
     from.unlock();to.unlock()
    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
        for (i=0; i<N; i++) {
            t[i].run();
        }
    }
}
```

```
void interest() {
    Account[N] a;
        for (i=0; i<N; i++) {
            a[i].lock();
            a[i].addInterest();
            a[i].unlock();
        }
}
```

Bank Transaction




```
class Transfer {
    Account from, to;
    int amount;

    void run() {
      from.lock(); to.lock()
        from.debit(amount);
        to.credit(amount);
      from.unlock();to.unlock()
    }
}
```

Bank Transaction

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    void debit(amount) {  ... }
    void credit(amount) { ... }
}
```

```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
        for (i=0; i<N; i++) {
            t[i].run();
        }
    }
}
```

Dolby et. al.'s data-centric annotations

```
void interest() {
    Account[N] a;
    for (i=0; i<N; i++) {

        a[i].addInterest();

    }
}
```

Motivating Analysis

Bank Transaction

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    void debit(amount) {  ... }
    void credit(amount) { ... }
}
```

```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
        for (i=0; i<N; i++) {
            t[i].run();
        }
    }
}
```

But that’s not available to me..

How to parallelize..

```
void interest() {
    Account[N] a;
        for (i=0; i<N; i++) {

            a[i].addInterest();

        }
}
```

Insights

- Integrate `async-finish` task parallelism and data-centric approach for synchronization
- For high performance load balancing, rely on TryCatchWS work-stealing runtime

AJWS Programming Model

AJWS – Atomic Java with Work-Stealing

Bank Transaction in **AJWS**

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    void debit(amount) {  ... }
    void credit(amount) { ... }
}
```

```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
finish  for (i=0; i<N; i++) {
        async t[i].run();
        }
    }
}
```

```
void interest() {
    Account[N] a;
 finish for (i=0; i<N; i++) {
    async {
        a[i].addInterest();

    } }
}
```

AJWS Programming Model

Annotations in AJWS

- Parallelism (Kumar et. al. OOPSLA 2012)
  - `async`, `finish`
- Data-centric Atomicity (Dolby et. al. TOPLAS 2012)
  - `@Atomicset(A)`
    - Denotes a group of memory locations that shares same consistency property and should be updated atomically
  - `@Atomic(A)`
    - Annotation on fields that belong to atomic set A
    - Annotation on a method to declare it to be additional unit for work for atomic set A
  - `@AliasAtomic(A=this.A)`
    - Annotation on an object to specify that the object's atomic set A is unified with current class's atomic set (this.A)

Translating AJWS to Vanilla Java

AJWS Programming Model

Bank Transaction in AJWS

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    ...

}
```

```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
    finish for (i=0; i<N; i++) {
        async t[i].run();
        }
    }
}
```

```
void interest() {
    Account[N] a;
    finish for (i=0; i<N; i++) {
      async {
        a[i].addInterest();

    } }
}
```

Translating Data-centric Annotations

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    ...

}
```

```
class Account {
    OrderedLock _lockA;
    int balance;

    public Account(b) {
      ...
      _lockA = new OrderedLock();
    }

    void interest() {
      synchronized(_lockA) {
        ...
      }
    }
    void interest_internal() {...}
```

Bank Transaction in AJWS

```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() { ...}
    ...
}
```

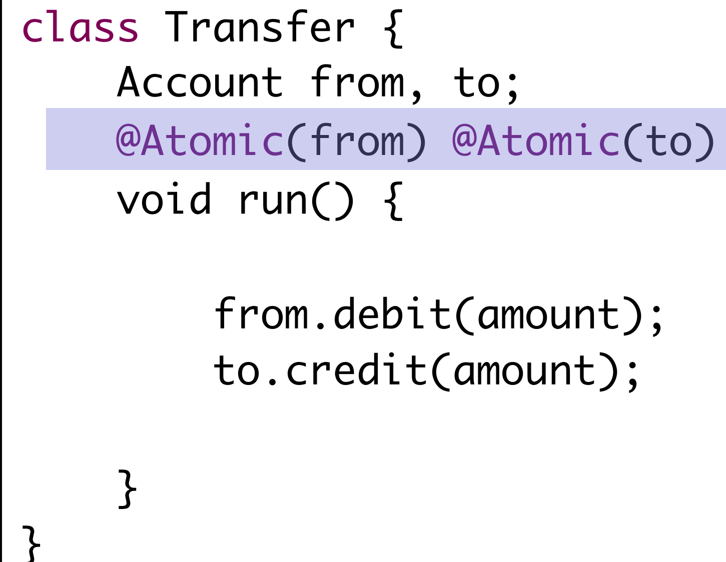
```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
finish for (i=0; i<N; i++) {
      async t[i].run();
        }
    }
}
```

```
void interest() {
    Account[N] a;
 finish for (i=0; i<N; i++) {
   async {
         a[i].addInterest();

      } }
}
```

Translating Data-centric Annotations

```
class Transfer {
    Account from, to;
    void run() {
        OrderedLock[2] lock;
        lock[0] = from.getLock();
        lock[1] = to.getLock();
        sort(lock);
        synchronized(lock[0]) {
        synchronized(lock[1]) {
          from.debit_internal(amount);
          to.credit_internal (amount);
        }
        }
    } // run()
}
```



AJWS Programming Model

Bank Transaction in AJWS

```
class Transfer {
    Account from, to;
    int amount;
    @Atomic(from) @Atomic(to)
    void run() {

        from.debit(amount);
        to.credit(amount);

    }
}
```

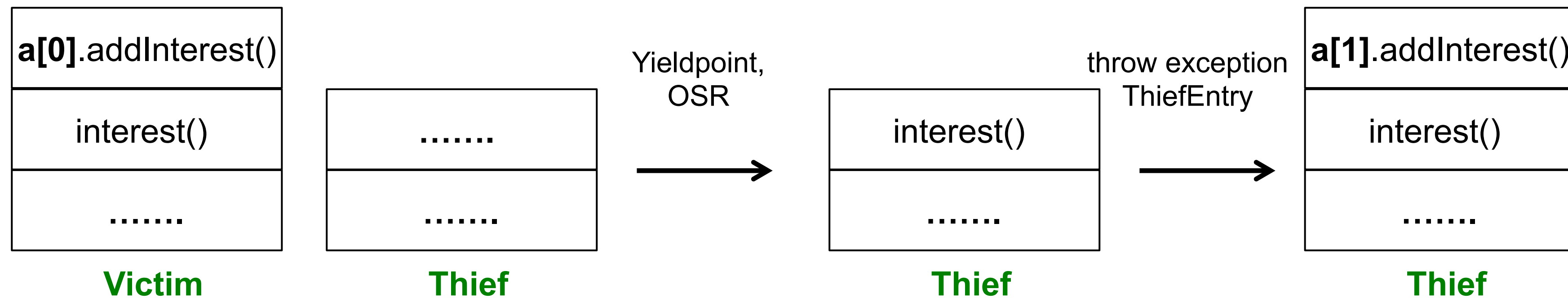
```
class Account {
    @Atomicset(A);
    @Atomic(A) int balance;
    public Account(b) { ... }
    void interest() {  ... }
    ...
}
```

```
class Bank {
    void fundTransfer() {
        Transfer[N] t;
finish for (i=0; i<N; i++) {
    async t[i].run();
        }
    }
}
```

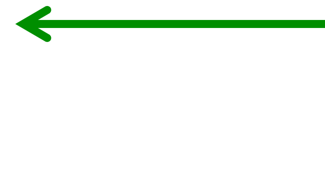
```
void interest() {
    Account[N] a;
 finish for (i=0; i<N; i++) {
    async {
        a[i].addInterest();

      } }
}
```

AJWS Programming Model

Translating Work-Stealing Annotations




```
void interest() {
  try {
    for (i=0; i<N; i++) {
      try {
        RT.continuation();
        a[i].addInterest();
        RT.checkIfStolen();
      }catch(ThiefEntry e) { }
    }
    RT.doFinish();
  catch(Finish f) { }
}
```



```
void interest() {
     Account[N] a;
  finish for (i=0; i<N; i++) {
    async {
         a[i].addInterest();

    } }
}
```

Implementation

Implementation

- Open-sourced prototype implementation of AJWS
  - Uses JastAdd compilation framework (unlike eclipse refactoring in AJ and polyglot in X10's TryCatchWS)
- Currently uses Java `synchronized` blocks for concurrency control
  - Future work would explore other options (e.g. transactions)
- Using parallelism annotation inside atomic section is not allowed

Performance Evaluation

Benchmarks

- jMetal
  - Framework providing set of classes that can be used as template for multi-objective optimization
  - Uses Java `Executer` , `Thread`, `synchronized`
- JTransforms
  - Multithreaded FFT library
  - Uses Java `Future`
- SJXP
  - XML parser build for Android OS
  - Original implementation is sequential

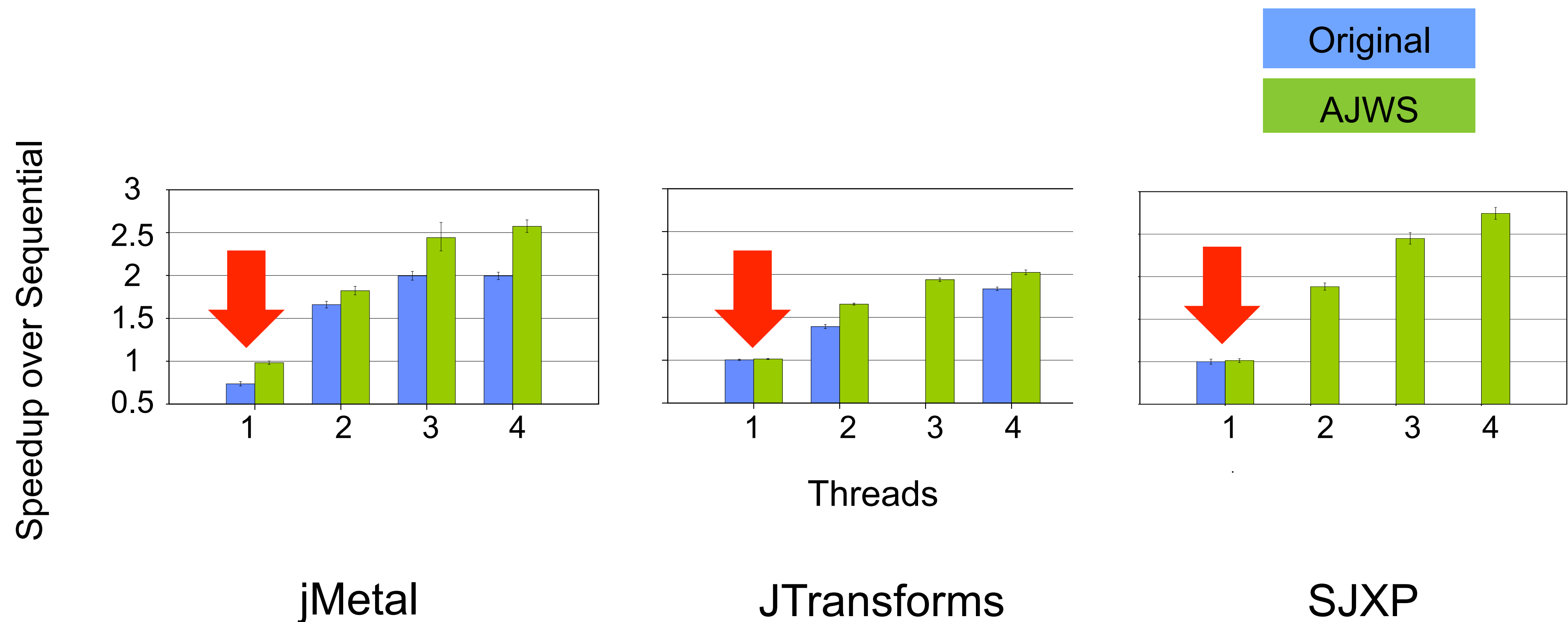
Experimental Infrastructure

- Hardware platform
  - 4 core ASUS ZenFone 2
  - Intel Atom core running at 2.3 GHz
  - LinuxDeploy application to install Ubuntu 15.10
- Software platform
  - JikesRVM Java VM mercurial version 11181
- Measurements
  - 15 invocations of each experiment with 6 iterations per invocation
  - We report the mean of final iteration in each invocation

Evaluating AJWS Productivity

Benchmark	Sequential		`Original`			AJWS			
	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%

Evaluating AJWS Performance



Conclusion

- AJWS – provides annotations in Java to enhance the productivity in parallel programming
- Prototype implementation that integrates task parallelism and data-centric atomicity
- High performance load balancing without incurring overheads
- Evaluation of AJWS using 3 large open-sourced applications
  - Extremely low syntactic overheads
  - Delivers better performance than original versions