Understanding and Overcoming Parallelism Bottlenecks in ForkJoin Applications

Gustavo Pinto[1] Anthony Canino[2] Fernando Castor[3] Guoqing Xu[4] Yu David Liu[2]
UFPA, Brazil[1] SUNY Binghamton, USA[2] UFPE, Brazil[3] UC Irvine, USA[4]

***Abstract*—FORKJOIN framework is a widely used parallel programming framework upon which both core concurrency libraries and real-world applications are built. Beneath its simple and user-friendly APIs, FORKJOIN is a sophisticated managed parallel runtime unfamiliar to many application programmers: the framework core is a *work-stealing* scheduler, handles *fine-grained* tasks, and sustains the pressure from *automatic* memory management. FORKJOIN poses a unique gap in the compute stack between high-level software engineering and low-level system optimization. Understanding and bridging this gap is crucial for the future of parallelism support in JVM-supported applications.**

This paper describes a comprehensive study on parallelism bottlenecks in FORKJOIN applications, with a unique focus on how they interact with underlying system-level features, such as work stealing and memory management. We identify 6 bottlenecks, and found that refactoring them can significantly improve performance and energy efficiency. Our field study includes an in-depth analysis of AKKA — a real-world actor framework — and 30 additional open-source FORKJOIN projects. We sent our patches to the developers of 15 projects, and 7 out of the 9 projects that replied to our patches have accepted them.

I. INTRODUCTION

Modern Java applications predominately run on parallel architectures, whose performance and energy efficiency critically depend on efficient thread management. FORKJOIN [1] is an influential parallel framework at the core of Java concurrency design. It not only provides thread management to numerous real-world applications, but also serves as the bedrock for higher-level Java concurrent libraries [2]. The impact of FORKJOIN also goes beyond Java applications *per se*, as several new programming languages [3], [4], [5] continue to operate on Java Virtual Machines (JVMs) and rely on FORKJOIN for thread management. FORKJOIN is known for its intuitive programming interface, particularly suitable for programming task-parallel and data-parallel jobs that have a divide-and-conquer nature.

FORKJOIN employs a *work-stealing* runtime [1]. While work stealing provides many benefits in resource utilization and scalability, efficient stealing dictates careful coordination across the layers of applications, runtime systems, and the OS. System-level performance and energy optimizations for C-based work-stealing programs are not new [6], [7], [8], [9], but combining work stealing with a Java-like managed runtime and, more importantly, reorienting it to application programming comes with a distinct set of unique challenges:

- *Thread Management*: work stealing by nature is "decentralized coordination," where threads coordinate on system utilization but thread management decisions are made by individual threads. This is in contrast with existing approaches either lacking coordination (*e.g.*, Java `Thread` objects) or requiring centralized management (*e.g.*, thread pooling).
- *Synchronization Management*: work stealing presents unique features in managing synchronization and thread states. Unfortunately, they conflict with traditional approaches such as locks (*e.g.*, synchronized methods) and explicit thread state management (*e.g.*, sleep) [10], leading to erratic performance surprising to application programmers. This problem is exacerbated by the large legacy code base of Java applications and libraries.
- *Data Management*: the Java runtime primarily allocates objects in the heap, and deallocation is managed by garbage collection. This is in sharp contrast with C-based work-stealing frameworks [11], where data are routinely represented as arrays of primitive data types. As a result, the allocation and distribution of data among worker threads plays a pivotal role in application performance.

Do these challenges introduce bottlenecks in modern parallel applications running on FORKJOIN? How severe are these bottlenecks in terms of performance and energy efficiency? Is there generalizable wisdom that can be shared with FORKJOIN programmers to avoid the bottlenecks?

This Paper We present the first empirical study to bridge the gap between modern software engineering and work-stealing systems in the context of FORKJOIN. It aims at providing a better understanding—as well as raising the awareness—of the subtleties and common performance pitfalls in FORKJOIN programming through a comprehensive study of characteristics and behaviors of real-world FORKJOIN applications. We identify potential bottlenecks against parallelism in these applications, illustrate their impact on system performance and energy, and demonstrate how such bottlenecks can be overcome through refactoring.

Our study follows a unique cross-layer approach: it is *application-driven* and *system-aware*. On the one hand, we are more interested in how real-world applications built on FORKJOIN behave—and how their performance can be improved through application-level programming—rather than an "under-the-hood" system-level optimization. On the other hand, we are aimed at finding the root causes of the bottlenecks on the systems stack, such as how each bottleneck may potentially hamper the desired behavior of the work stealing scheduler, garbage collector, and underlying hardware. This cross-layer approach advances software engineering by provid-

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

Accepted for publication by IEEE. © 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

TABLE I
PLACING FORKJOIN IN CONTEXT.

	work stealing	fine-grained parallelism	dynamic allocation	garbage collection	unstructured synchronization	programmable thread states
Fortran	no	no	no	no	yes	uncommon
Pthread	no	no	uncommon	no	prevalent	prevalent
OpenMP	no	no	uncommon	no	uncommon	uncommon
MPI	yes	yes	uncommon	no	uncommon	uncommon
Cilk	yes	yes	uncommon	no	uncommon	uncommon
Java threads	no	no	prevalent	yes	prevalent	prevalent
X10	yes	yes	prevalent	yes	uncommon	uncommon
Haskell	yes	yes	prevalent	yes	uncommon	uncommon
FORKJOIN	yes	yes	prevalent	yes	prevalent	prevalent

ing guidelines for performance improvement *and* illuminating why programming patterns and performance are intimately linked. The approach also advances system research by filling a void of assessing work stealing through an empirical and application-oriented route, taking advantage of the fact that FORKJOIN is the first work stealing framework with a large application developer base.

We take a two-pronged approach for our empirical study. First, we conduct a *depth-oriented* study on AKKA [12], a sophisticated middleware FORKJOIN-based framework. We identify a bottleneck at the junction of AKKA's messaging engine and FORKJOIN, and demonstrate an average speedup of 3.1× and up to a 13.1×, and an average energy savings of 31.6% up to 80.2% through an in-depth refactoring of AKKA's core messaging engine. Second, we conduct a *breadth-oriented* study through investigating 30 real-world FORKJOIN projects from GitHub, with a total of 791K LOC. We summarize our findings as a taxonomy of 6 bottlenecks and present a cross-layer analysis on the root causes of these bottlenecks. By removing these bottlenecks, the optimized applications can produce an average of 26% of performance improvement and 23% of energy savings. Our optimization patches were confirmed by the majority of application developers we communicated with.

This paper makes the following contributions:

- We present a comprehensive application-driven system-aware empirical study on performance and energy efficiency of FORKJOIN applications.
- We identify 6 bottlenecks latent in FORKJOIN applications, analyze their root causes, and provide programming patterns for mitigating them.
- We develop FJDETECTOR, a bottleneck detection and refactoring tool that can perform interactive source-code-level optimizations of some FORKJOIN applications.

The source code of the tool we have developed, as well as all raw data, can be found online.[1]

II. BACKGROUND

We now provide a brief background on the work stealing algorithm, its implementation in FORKJOIN, and applications built on FORKJOIN.

[1]https://github.com/gustavopinto/fjdetector

Work Stealing Work stealing was popularized by the Cilk language [11], a C-like language designed for parallel programming. In the work-stealing runtime, each CPU core is managed by a *worker*, which is often directly mapped to an OS thread. The computational unit executed by each worker is called a *task*, during whose execution may *fork* additional tasks. These tasks are placed on a *decentralized* per-worker queue. When a worker completes a task, it picks up one more from its queue. When the queue is empty, the worker *steals* a task from the queue of another worker. In this case we call the stealing worker a *thief* while the worker whose item was stolen is called a *victim*. Ultimately, workers are *joined* to compute a result.

Observe that the *logical* parallel processing unit, a task, is different from the *physical* parallel unit, a worker. In practice, the number of workers (physical threads) is statically determined—often the same as the number of CPU cores—whereas the number of tasks (logical processing unit) far exceeds the number of workers. Work stealing, therefore, is an instance of *fine-grained* parallelism.

The* FORKJOIN *Framework FORKJOIN is Java's parallel programming framework with a unique set of features. Table I places FORKJOIN in the context of commonly used frameworks and languages.

At its core, FORKJOIN is a concrete implementation of work stealing. The `ForkJoinPool` class is the entry point of the FORKJOIN framework, where the programmer can specify the number of workers. Tasks are modeled as subclasses of the `ForkJoinTask` class: `RecursiveTask` and `RecursiveAction`. The two differ in that only the former can return the result of a computation. Upon execution, a `ForkJoinTask` instance may in turn fork additional tasks—called child tasks—via the `fork` method. Invoking the `join` method introduces synchronization between the enclosing task and its children. The framework also provides additional utility methods. For example, method `invokeAll` is syntactic sugar for a `fork` immediately followed by a `join`. Method `isDone` inspects whether a task has completed.

FORKJOIN *Applications* As FORKJOIN runs on the JVM, its influence extends beyond applications written in Java. A growing number of object-oriented languages—such as Scala—are translated to Java (bytecode) and operate on the

JVM. Such programming languages also take advantage of FORKJOIN for thread management. As our main interest is on *dynamic* behavior of FORKJOIN—performance and energy efficiency in particular—we view applications written in these languages also as FORKJOIN applications.

One example is AKKA, an actor framework written in Scala but built on Java's FORKJOIN. Scala's programming interface for FORKJOIN is identical to the interface described above, with the exception of language-specific grammatical differences. Conceptually, actors are a message-passing framework where each actor serves as a logical processing unit that communicates with each other. No two actors share memory, leading to benefits such as race condition freedom by design. AKKA has been deployed by companies such as Groupon, eBay, and Amazon.

III. METHODOLOGY

Benchmarks Table II shows the applications within the scope of this study. The first row provides the information for AKKA. We selected this application because: (1) AKKA is among the largest open-source projects built on top of FORKJOIN; (2) AKKA has been extensively deployed in the real world; (3) AKKA is a middleware framework rather than an "end-user" application. Its performance improvement may lead to significant impact on a large number of end-user applications.

For the breadth-oriented study we searched Github for the key word "ForkJoin" and selected a set of 30 open-source projects, covering a wide range of application domains from supervisor management to raytracing. Our selection criteria are: (1) they should not be tutorials; (2) they should be recent, but not currently under rapid changes. For example, we did not select any projects whose first commit and last commit are both within 6 months; (3) they must be able to compile and run. For each project, we investigate its source code looking for possible bottlenecks. If the project has tests, we executed the tests that perform FORKJOIN computations; otherwise, we wrote the tests. We discarded projects where we were unable to find any bottleneck.

Experiments We ran each selected application in a machine with a 2×8-core (32-core when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670 CPU (2.60GHz) and 64GB of DDR3 1600 memory, running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 25.5-b02, mixed mode, JDK version 1.8.0_05-b13). The machine has three cache levels (L1, L2 and L3), whose sizes are 64KB per core (128KB total), 256KB per core (512KB total), and 3MB (smart cache), respectively. All experiments were performed in the OS-exclusive mode without any other loads running simultaneously.

The default settings of both the OS and the JVM were used. In particular, (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size were set to be 1GB and 16GB respectively. Hyper-threading is enabled and the Turbo Boost feature is disabled.

TABLE II
A SAMPLE OF PROJECTS USED IN THIS STUDY. LOCS ENCOMPASS ONLY NON-BLANK AND NON-COMMENTED LINES OF CODE COMPUTED USING THE `CLOC` PROGRAM.

Projects	# LoC	# Commits	# Bottlenecks
`Akka`	326,341	20,759	1
`itemupdown`	4,925	2	2
`jAcer`	4,476	35	2
`educational`	1,323	7	2
`scalatuts`	253	5	2
`knn`	3,099	27	2
`doms-transformers`	3,714	254	2
`ForkAndJoinUtility`	127	12	2
`Solitaire`	1,527	39	2
`mywiki`	1,920	17	2
`MagicSquares`	664	153	2
`ejisto`	12,330	274	2, 3
`exhibitor`	15,314	701	2, 3, 4
`cq4j`	5,815	23	2, 3
`netflixoss`	231,361	1	2, 3
`javaOneBR-2012`	518	4	2, 3
`jadira`	46,095	630	3
`ecco`	5,849	119	3
`conflate`	934	9	3
`bazzar-base`	7,766	15	3, 4
`DocumentIndexing`	1,127	1	4
`CSSTProto`	10,721	17	4
`Fibonacci`	79	2	5
`Mandelbrot`	1,442	30	5
`Solitaire`	1,527	39	5
`Matrices`	2,356	15	5
`LockedBasedGrid`	1,390	1	5
`Basic-Blocks`	4,821	41	5
`warp`	15,287	338	6
`j7cc`	5,110	76	6
`lowlatency`	3,018	18	6

For all applications other than AKKA, we ran each benchmark 10 times; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs to warm up the JIT optimizations [13]. For our AKKA study, we ran each benchmark 27 times, discarding the first 7 runs. Message passing frameworks such as actors are known to have a higher degree of nondeterminism. We observed higher variation in our experiments and choose to represent results with a larger sample of data.

Energy consumption was measured using `jRAPL` [14], a framework that contains a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) [15] support. Our energy consumption data include CPU core, CPU uncore, and DRAM energy consumption.

IV. A STUDY ON AKKA

In this section we conduct a depth-oriented study on potential parallelism bottlenecks latent in FORKJOIN applications, with a focus on AKKA.

An Overview of* AKKA *(Messaging) Core AKKA's internal messaging structure is detailed in Listing 1. Messages are encapsulated by an `Envelope` that bundles an abstract message with its sending actor. Messages are sent between actors by forwarding the message to the `Dispatcher`. Messages sent to an actor are queued up in the actor's `Mailbox`, which is an instance of a `ForkJoinTask`. Once scheduled, a `Mailbox` task will process *all* messages held in its queue, one at a time

```
case class Envelope(message: Any, sender: Actor)
class Mailbox(queue: Queue[Envelope])
   extends ForkJoinTask {
  var receiver: Actor = _
  def setActor(a: Actor) = receiver = a
  def hasMessages: Boolean = { ... }
  def isScheduled: Boolean = { ... }
  def setScheduled() = { ... }
  def setNotScheduled() = { ... }
  def run: Boolean = { processMailbox() }
  def processMailbox(): Unit = {
    val next = queue.next()
    if (next != null) {
      receiver.invoke(next)
      processMailbox()
    }
    setNotScheduled()
  } }
class Dispatcher {
  val pool: ExecutorService
  def dispatch(receiver: Actor, msg: Envelope) = {
    val mbox = receiver.mbox
    mbox.enqueue(msg)
    if (!mbox.isScheduled) {
      mbox.setScheduled()
      pool.execute(mbox)
    }
  } }
class Actor(mbox: Mailbox, dispatcher: Dispatcher) {
  def sendMessage(msg: Envelope): Unit = {
    // ...
    dispatcher.dispatch(this, msg)
  } }
```

Listing 1. The Core AKKA Messaging Logic (Classes `Mailbox`, `Dispatcher`, and `Actor` have additional unrelated methods not shown)

```
class Envelope(message: Any, sender: Actor)
   extends ForkJoinTask {
  var receiver: Actor = _
  def setActor(a: Actor) = receiver = a
  def run(): Unit = {
    receiver.invoke(message)
    receiver.mbox.setNotRunning()
  }
}
class Mailbox(queue: Queue[Envelope])
   extends ForkJoinTask {
  var receiver: Actor = _
  def setActor(a: Actor) = receiver = a
  /* hasMessages, isScheduled, setScheduled,
       setNotScheduled same as Listing 1 */
  def isRunning: Boolean = { ... }
  def setRunning() = { ... }
  def setNotRunning() = { ... }
  def run: Boolean = {
    if (!isRunning) { processMailbox() }
    else { run() }
  }
  /* processMailbox same as Listing 1 */
}
class Dispatcher { /* Same as Listing 1 */ }
class Actor(mbox: Mailbox, dispatcher: Dispatcher) {
  def sendMessage(msg: Envelope): Unit = {
    msg.setActor(this)
    if (!mbox.hasMessages && mbox.notRunning) {
      mbox.setRunning()
      msg.fork()
    } else {
      dispatcher.dispatch(this, msg)
    }
  } }
```

Listing 2. Refactored AKKA Messaging Logic

in the same order that the messages were received. A message is processed when the message handler defined in the actor has been executed. Note that a `Mailbox`, once scheduled, may represent a long-running task. Furthermore, a `Mailbox` handles synchronization via status bits and compare-and-swap, abstractly represented with `isScheduled`, `setScheduled`, and `setNotScheduled`. Overall, an AKKA runtime may consist of a large number of actors, and FORKJOIN provides fine-grained parallelism for message processing of different actors, as illustrated by the `Mailbox` class.

Bottleneck: Centralized Pooling FORKJOIN as a work-stealing runtime in essence features *decentralized* thread management: decisions on task creation, execution, and migration are managed by individual worker threads and there is no centralized control. For backward compatibility purposes, FORKJOIN in addition supports *centralized pooling*: maintaining a centralized task pool where all newly created tasks should be submitted, and from which all FORKJOIN workers steal[2]. Centralized pooling, however, goes against the spirit of work stealing, which may lead to performance penalties.

AKKA handles `Mailbox` tasks through centralized pooling. This can be seen in the `dispatch` method in Listing 1, where the mailbox is submitted through `execute` to the centralized pool. Indeed, the `Mailbox` abstraction is a natural design choice considering AKKA needs to maintain the semantic guarantee that messages are processed one at a time in the well-preserved order. The sacrifice to be made is that a task cannot be forked for every message sent, *de facto* foregoing the decentralized nature of FORKJOIN design. This may lead to performance penalties, especially when an actor does not continuously receive a backlog of messages, *i.e*, incoming messages do not need to be queued.

Centralized pooling is unfriendly to `ForkJoin` for several reasons. First, there is greater synchronization overhead associated with scanning the centralized pool. Second, the processing of individual tasks must go through centralized scheduling, often delayed compared with the decentralized design.

Overcoming the Bottleneck We illustrate a modified version of AKKA that takes advantage of `fork` in Listing 2. Our intuition is that when the `Mailbox` is empty we can immediately `fork` the message handling as a task at message-send time. We transformed the `Envelope` class into a `ForkJoinTask`, which upon `run`, will invoke the handler of the message `receiver`. To determine whether the mailbox is empty we introduce a flag `isRunning` which will be atomically accessed. When the `Mailbox` is not empty the program defaults to AKKA's one-at-a-time message processing. Algorithmically, this refactoring may improve performance of AKKA programs because it removes the handling of the first actor message from the critical path of actor message handling. The synchronization introduced by `isRunning` is per mailbox, decentralized in nature.

In our implementation we further enable a light-weight tracking on how often `Mailbox` is empty when a message

[2]For backward compatibility reasons, the conceptually centralized pool is implemented as the union of all FORKJOIN worker thread queues.

TABLE III
DETAILED PERFORMANCE STATISTICS: AKKA WITH 32 FORKJOIN WORKER THREADS

benchmark	Runtime (ms)					Energy (J)				
	original	σ	custom	σ	speedup	original	σ	custom	σ	savings
`max-throughput`	2057.6	621.41	1978.5	394.13	1.04x	397.93	119.98	388.15	81.89	2.46%
`single-ping`	11716.6	739.74	8694.45	1568.18	1.35x	1921.59	289.79	1737.14	277.48	9.6%
`ping-throughput`	2507.4	107.89	701.3	69.13	3.58x	526.67	24.19	195.67	15.19	62.85%
`single-producer`	4078.7	1331.51	5257.45	2651.6	0.78x	512.38	157.05	580.63	249.51	-13.32%
`multi-producer`	7236.95	993.1	7907.25	1557.71	0.92x	732.65	174.19	846.34	283.68	-15.52%
`middle-man`	3827.4	151.74	1752.9	185.51	2.18x	807.48	31.31	437.72	38.06	45.79%
`mediator`	4505.05	376.8	679.7	70.38	6.63x	912.59	78.22	194.78	14.65	78.66%

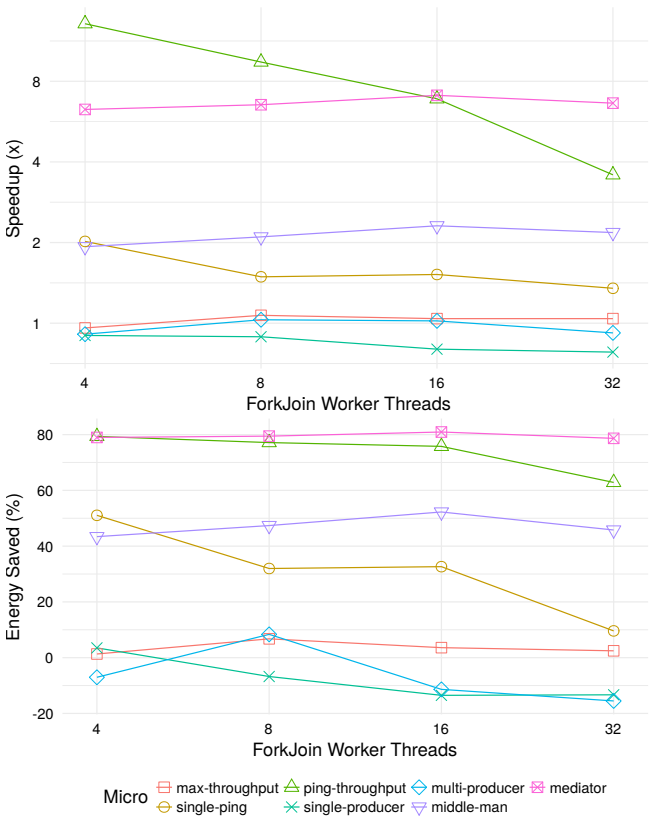


Fig. 1. Speedup (Y-Axis in Logarithmic Scale) and Energy Saving of Refactored AKKA Implementation

is sent to it. The AKKA runtime adaptively switches to the default when the likelihood is small.

TABLE IV
AKKA BENCHMARK CONFIGURATIONS

	actors	messages per actor
`max-throughput`	8+8	1,500,000
`single-ping`	2	300,000,000
`ping-throughput`	20,000	1000
`single-producer`	1	600,000,000
`multi-producer`	1+8	600,000,000
`middle-man`	30,000	2000
`mediator`	30,000	1000

Performance Impact We have modified AKKA 2.5 — compiled and run with Scala 2.11 — to incorporate these changes. We have evaluated these changes within AKKA with 7 micro benchmarks provided by the `actors` benchmarking suite, with minor changes to include our performance measurements [16]. `ping-throughput` creates several pairs of actors that ping messages. `single-ping` is an instance of this general pattern where only one pair of actors with a large number of messages are created. Additionally, we created two variations of `ping-throughput`: `middle-man`, where two pinging actors compete to send messages to a third, "middle man" actor; and `mediator`, where a ping messages must first pass through a third actor. `single-producer` taxes a single actor by sending a large number of messages without waiting. `multi-producer` spawns 8 application threads that all send messages without waiting to a single actor. `max-throughput` spawns 8 application threads that each send messages without waiting to their own actor. The number of actors and messages per actor for each benchmark are detailed in Table IV.

As shown in Figure 1, eliminating the centralized pooling bottleneck results in a remarkable improvement in performance and energy efficiency. We observe an average of 3.1× speedup and 31.6% energy savings in a wide spectrum of settings. Among them, `ping-throughput`, `single-ping`, and `mediator`, reacted to our refactoring with overwhelmingly positive results. These three benchmarks capture the scenario when an actor or some actors are able to process messages without them queueing up, and represent the case where our experiments confirm that `fork` leads to performance benefits. For `ping-throughput`, the observed speedup ranges from 3.6× to 13.1×, and the energy savings range from 62.9% to 79.4%. In contrast, `max-throughput`, `single-producer`, and `multi-producer`, capture the scenario where an actor will receive messages faster than it can process them, and represent the case where our experiments indeed show a mild slowdown. We will discuss these details shortly. The most intriguing case is perhaps `middle-man`, a hybrid case where some running actors are observed to have a backup of messages. Encouragingly, `middle-man` has a stable 1.93× to 2.31× speedup and 43.4% to 52.2% energy savings.

AKKA as a middleware framework may be subjected to diverse workloads. Refactoring at the level of the core service of AKKA cannot — nor should it be expected to — benefit all workloads. In our experiments we find our new implementation of AKKA is effective in the presence of heavy workloads

in terms of the number of actors, the number of messages, and the number of workers. The workload it does not handle well is the case when the throughput rate of an actor's message handler is far below the rate of its message reception. Our current sampling algorithm partially addresses this issue, but a more refined workload characterization is likely needed for an industrial-strength AKKA re-implementation. We highlight the 32-thread configuration in Table III. Observe that in the case that we do not perform well, the slowdown remains within the deviation.

V. A TAXONOMY OF FORKJOIN PARALLELISM BOTTLENECKS

Centralized pooling is an important bottleneck we have discovered for FORKJOIN applications, but not the only one. In this section, we summarize additional bottlenecks we have found in our study. From now on, centralized pooling is also called **Bottleneck 1**.

Bottleneck 2: Copy on Fork For data-intensive applications, a performance-sensitive dimension of design is data distribution, *i.e.*, how data are spread through parallel execution units. In divide-and-conquer frameworks — including FORKJOIN — the general strategy is to represent the data as an indexible structure, *e.g.*, a (potentially multidimensional) array, which in turn can be partitioned into slices and fed to individual parallel execution units.

This simple process may pose challenges to a FORKJOIN programmer. In particular, data in Java are often represented as objects, and arrays are dynamically allocated. The combination effect of aliasing and shared-memory programming implies that data distribution "by reference" at forking time may introduce race conditions.

As a conservative approach, many FORKJOIN programmers choose to copy data at the forking time. Observe the following usage of the `copyOfRange` in Figure 2.

```
import static Arrays.*;
class Task extends RecursiveAction {
 public Task (User[] u) { ... }
 protected void compute() {
  if (u.length < N) { local(u); }
  else {
   int split = u.length / 2;
   User[] u1 = copyOfRange(u, 0, split);
   User[] u2 = copyOfRange(u, split, u.length);
   invokeAll(new Task(u1), new Task(u2));
} } }
```

Fig. 2. Example of copying data over sub-tasks

Beyond the obvious consequences such as memory bloat [17], excessive copying turns out to be uniquely unfriendly to FORKJOIN, for a number of reasons. (1) As a fine-grained parallelism framework, most tasks are completed within milliseconds. Copying upon fork implies the dominating growth of short-lived objects, creating a severe burden for garbage collection. (2) The cascaded division common in FORKJOIN applications means that data are copied at every level of recursion, potentially leading to an O(log n) growth in memory. In contrast, copying for flat data partitioning can only lead to a constant growth in memory. (3) Unlike copying with flat data partitioning where all allocations are done once and for all, a strategy somewhat friendly for the memory allocator due to batching, copying with cascaded data partitioning leads to frequent yet intermittent allocation requests, hampering performance.

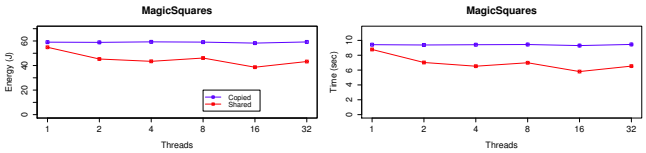


Fig. 4. A comparison on energy and performance with varying numbers of threads before and after copies are removed in `MagicSquares`.

Among the 30 programs we have studied, we found 18 occurrences of this bottleneck, in 15 FORKJOIN programs. Fixing the bottleneck requires simple modification of the source code that shares the input data structure and lets subtasks work on distinct regions of the data structure. Figure 3 shows the energy gains from fixing this bottleneck. Clearly, the energy consumption is reduced in all the refactored programs. The average reduction in energy consumption is 12.63%. The execution time decreases proportionally. Interestingly, 9 out of the 15 analyzed projects cross the 10% barrier of energy savings. However, 5 of the analyzed projects have energy savings of less than 5%. For the projects above 5%, the minimum energy saving was 8.23% (for `itemupdown`), and the maximum was 23.51% (for `MagicSquares`). After inspecting these projects, we have observed that the amount of energy savings is related to the width of forking. That is, the more the program creates redundant copies of the data structure, the more effective our refactoring is.

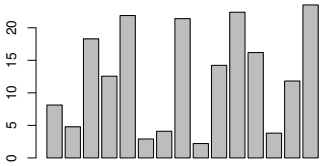


Fig. 3. Energy savings (%) when removing the *Copy on Fork* bottleneck. From left to right, projects are `itemupdown`, `jAcer`, `educational`, `scalatuts`, `knn`, `netflixoss`, `doms-transformers`, `ForkAndJoinUtility`, `exhibitor`, `Solitaire`, `javaOneBR-2012`, `mywiki`, `ejisto`, `cq4j`, and `MagicSquares`

Figure 4 shows the comparison results before and after eliminating copies for `MagiSquares`, a computational benchmark for computing the magic square puzzle[3]. The data-parallel computation is based on the number of permutations available, which represents all possible rows, columns, and diagonals. Each parallel task attempts to construct a matrix whose first row is the permutation and whose first column is another permutation that begins with the same entry and contains no other duplicate entries. The algorithm attempts to find sum permutations to fill in the remaining rows and columns. When sharing the data structure, we saved the program from creating 128 additional data structures (with integer data type), leading to a 23.51% energy saving, when running with 32 threads.

[3]http://mathworld.wolfram.com/MagicSquare.html

Enabled by jRAPL, we also report our results on the hardware component-level for DRAM, CPU, and Uncore respectively. We observed that roughly the same amount of CPU energy was consumed before and after removing the copies (*i.e.*, 8.32 Joules and 6.67 Joules, respectively). However, the difference is more obvious when the energy consumptions of DRAM and Uncore are compared. Due to the excessive object creation, DRAM and Uncore of the original version consume 1.39× and 1.43× more energy than the optimized version.

Since *Copy on Fork* creates large volumes of small, shortly-lived data structure objects, it is interesting to understand how different GC algorithms may impact our results, we conducted experiments over 5 GC options in Hotspot: (a) SerialGC: the stop-the-world serial collector, (b) ParallelGC: the parallel collector, (c) ParallelOldGC: the parallel collector with data compression, (d) ConcMarkSweepGC: concurrent mark sweep collector, and (e) G1GC: the garbage-first collector. Figure 5 shows the results for MagicSquare. For almost all algorithms, the fix can speed up GC by 20%–40%.

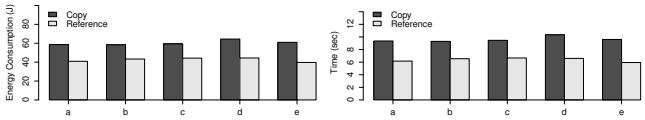


Fig. 5. A Comparison of GC costs (MagicSquares, 32 threads; GC algorithms are: a: SerialGC, b: ParallelGC, c: ParallelOldGC, d: ParallelNewGC, e: G1GC).

Bottleneck 3: Copy on Join The counterpart of *Copy on Fork* is *Copy on Join*: after having joined on its subtasks, a task must usually combine the results of the subtasks into a result for the larger problem. Consider the program in Figure 6, extracted from the cq4j benchmark.

```
protected List<T> compute() {
  int size = dataSource.size();
  if (size < FORK_SIZE) {
    return computeDirectly();
  } else {
    List<T> result = new ArrayList<T>();
    int mid = size / 2;
    RecursiveFilteringTask<T> first = new
         RecursiveFilteringTask<T>(filter, dataSource.
         subList(0, mid));
    first.fork();
    RecursiveFilteringTask<T> second = new
         RecursiveFilteringTask<T>(filter, dataSource.
         subList(mid, size));
    second.fork();
    result.addAll(first.join());
    result.addAll(second.join());
    return result;
  }
}
```

Fig. 6. Example of joining data with sub-tasks.

As one reader might observe, this particular code snippet suffer from the same bottleneck previously explained (creating sublists of the current data structure). However, this benchmark also presents a different bottleneck. At the end of the execution, an expensive operation addAll is invoked to copy merge collections. *Copy on Join* has many negative consequences similarly to *Copy on Fork*, with one additional unique drawback: since joining in a work stealing system is implemented by barriers, *Copy on Join* increases the wait time at barriers, particularly unfriendly for *energy consumption*. Note that this is an established fact [18], [19], [9], [13]: barrier wait at the low level is either implemented as spin locks or context switch, both of which can lead to energy waste without contributing to program progress. We have found 5 occurrences of this bottleneck in the 30 programs studied. A fix of this bottleneck is similar to that of *Copy on Fork*: a shared data structure can be passed into subtasks to carry results. After applying these changes in 5 programs, we have achieved overall 3% – 13% energy savings. The results are shown in Figure 7.

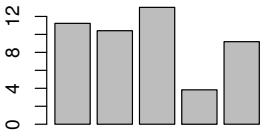


Fig. 7. Energy savings (%) after removing the *Copy on Join* bottleneck. From left to right, projects are: cq4j, ejisto, javaOneBr-2012, exhibitor, conflate.

Bottleneck 4: Scattered Data We next investigate the impact of data locality on performance and energy consumption. An important pattern we found is that the execution of a task follows the sequence of $abababababc$, where a performs memory copies for a subtask, b forks the subtask, and c does the computation of the current task. Figure 8 shows a code snippet of this case, extracted from benchmark CSSTProto.

```
protected R compute() {
  if (len == 1) {
    RecursiveTask<R> task = createTask(from);
    return task.invoke();
  } else {
    ForkJoinTask<R>[] tasks = new ForkJoinTask[len];
    for (int i = 0; i < len; i++) {
       ForkJoinTask<R> task = createTask(from+i);
       task.fork();
       tasks[i] = task;
    }
    R result = tasks[0].join();
    tasks[0] = null;
    for (int i = 1; i < len; i++) {
       R next = tasks[i].join();
       tasks[i] = null;
       result = merge(result, next);
    } return result; }
```

Fig. 8. Example of scattered data.

This pattern has impact on energy consumption and performance for several reasons. First, the copy operation has the potential of polluting caches, increasing the chance of memory round-trips. Second, the number of context switches might also increase, due to the sparse task creations. A possible solution to this problem is to create a list of tasks and, during the for loop, add each new task object to the list. After the execution of the for loop, one might call the invokeAll method, which is responsible for forking and joining all tasks in the list. With this fix, we have observed an energy saving of 9.82% for CSSTProto. Regarding cache behavior, we observed that the original implementation had a 34.24% cache misses, whereas the fix reduced it to 31.98%. We also observed a reduction on context switches, from 24,550 to 23,193. Yet, the number of branch misses is also reduced: from 1.82% of all branches

to 1.14% [4], which we believe is due to the boilerplate code used our initial example; `invokeAll` eliminates the first for loop, then reducing the overall number of branches and, as a consequence, the number of branch misses.

Bottleneck 5: Exacting Intra-Task Synchronization As locks play a central role in Java shared-memory programming and metadata representation, unstructured synchronization (*i.e.*, object locks) is pervasive in Java applications. Synchronization occurs via invoking `synchronized` methods or code blocks, or using popular concurrent library classes such as `CountDownLatch`. Improving performance and energy efficiency for systems where unstructured synchronization is the only mechanism to achieve concurrency safety — such as Pthreads or the Java `Thread` library— is a well understood topic.

Mixing unstructured synchronization in a structured parallel system such as work stealing leads to additional subtle interactions between the application runtime and the OS. When unstructured synchronization happens *in the middle* of the task execution, it effectively stalls stealing from that worker. Unfortunately, the stalled worker cannot forgo the current task and select another task from its deque — even if there are many other task items in it — because tasks on the deque in a work stealing system carry inherent logical dependencies, analogous to stack frames. At best, the worker itself can be context-switched by the OS. Observe however, even though there may be thousands of *tasks* in the work-stealing runtime, the number of *workers* — the JVM representation of OS threads — is few, typically smaller than the number of CPU cores. In other words, OS-level context switch may at best help *other* applications in a time-sharing environment, but will not contribute to improving the performance or energy efficiency of the application *itself*.

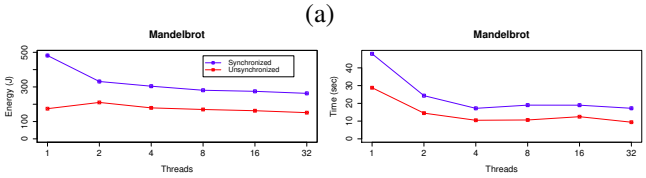
The most principled solution to avoid the bottleneck is to eradicate unstructured synchronization from Java. There is encouraging progress in recent Java development to support asynchronous abstractions, such as futures [20]. However, it may take time before Java practitioners fully embrace these features [10]. In this study, we investigate into legacy programs, attempting to understand how unstructured synchronization is used in the real world. Overall, we found 7 occurrences of this bottleneck. Surprisingly, we found in a significant number of projects, an easier solution exists: many synchronizations are simply to implement *exact computations*, which can be safely *relaxed* [21] without creating any impact on correctness [22].

We illustrate this bottleneck with benchmark `Mandelbrot`. A mandelbrot is a mathematical set of points whose boundary is a distinctive two-dimensional fractal shape. Each parallel task works on a set of points, and the `synchronized` block is then used when a task needs to render the fractal image. This is done by calling the `setRGB` method, available on the `BufferedImage` class, as showed in Figure 9-(a).

[4]We used the `perf` linux tool to calculate cache misses, context switches, and branch misses.

```
if (!isBenchmarking && mb.isLiveRendering) {
  synchronized (mb.lock) {
    mb.renderImage.setRGB(j, i, color.getRGB());
  }
  mb.repaint();
}
```

(a)



(b)

Fig. 9. Example of an hidden over-synchronization (a) and a comparison of energy and performance, with and without synchronization (b), on `Mandelbrot`

Figure 9-(b) shows the results for this benchmark, In this benchmark, a task has a range of values of which it should work on. For our input data (width: 1000, height: 10000), the benchmark creates a total of 2,048 tasks. As we can see, there is a great difference between the synchronized version and the unsynchronized one. On average, the unsynchronized version consumes 42% less energy then its counterpart (38% faster).

After inspecting the implementation, we observed that the method `setRGB` is already `synchronized`, so there is no need to use another synchronization construct to wrap up this single method call. In fact, we could not find any visible difference between the images generated by executions with and without the synchronization. We sent the modified source code as a patch to its developer, who then acknowledged the over-synchronization and accepted our patch.[5].

Bottleneck 6: Sleepy Workers A more extreme case — but along the same line of *Intra-Task Synchronization* — is the use of `Thread.sleep` during task execution. Just as the previous bottleneck, the invocation of this thread management primitive stalls stealing, and explicitly requests OS context switches. From a logical perspective, the intention of the programmer may be to put the *task* to sleep, but unfortunately, the work stealing runtime will place the *worker* to sleep. As described earlier, the worker cannot forgo the sleep-inducing task and pick up other tasks from its deque; neither can the idle CPU core help other workers of the same application. What is worse is that unless the OS has other applications running, an idle core under the widely used on-demand governor would put the core in a low-power state, which later needs a long time to wake up. In a work stealing runtime where competitive performance is of its first priority, user-level sleeping is often more detrimental than beneficial. We found 3 occurrences of this bottleneck.

The `CTask` benchmark presents the worst scenario of this bottleneck. During the sequential execution, this benchmark puts every current task to sleep for a second. Figure 10 shows this impact on both performance and energy consumption.

[5]https://github.com/catree/SimpleMandelbrotDemo/pull/1

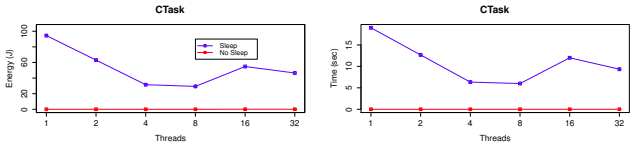


Fig. 10. A comparison on energy and Performance, with and without thread sleeping, for varying numbers of threads in `ctask`.

The sleep construct creates significant penalties in both performance and energy consumption. The execution without sleep can be 13,495.26× more energy efficient than the execution with (and 1,917× reduction in running time). After inspecting the source code, we observed that the developer used sleep to force the program to wait for a result from another computation. However, this sleep is unnecessary, since the computation on which the sleep is waiting is a *synchronous* operation.

VI. Detecting Refactoring Opportunities

Some bottlenecks can be detected and refactored automatically. As a proof of concept, we have built a tool named FJDetector capable of automatically detecting and refactoring copy-related bottlenecks as explained in Bottleneck 2.

A. FJDetector

FJDetector works as a plugin for Eclipse IDE. It performs source code analysis on ForkJoin programs, focusing on programs with divide-and-conquer data parallelism. We check if the ForkJoin computation is operated on a *data structure*, such as `array` or `ArrayList`. Since most of the `ArrayList` methods provide accesses over arrays, our approach handles them in a similar way. ForkJoin computations are usually described in terms of inner-classes, where the data is passed through the inner-class constructor. Hence, for each parameter of the constructor, we inspect (a) if it is a data structure, (b) if it is splitted and copied inside the `compute` method, and (c) if the variables containing the copy results are passed into new instances of the `Task` class.

We identify potential divide-and-conquer programs through pattern matching ForkJoin's `compute` method body. Specifically, we are looking for a branching statement that falls into one of three patterns.(1) sequential computation in the `if` block and parallel computation in the `else` block; (2) parallel computation in the `if` block and sequential computation in the `else` block; and (3) sequential computation in the `if` block plus a `return` at the end of the block, and the parallel computation in the remainder of the method. FJDetector is not able to work with ForkJoin classes structured in a different manner.

Once a bottleneck is confirmed by the developer, FJDetector performs a set of transformations on the ForkJoin code. Our transformations remove copies by computing indices for each subtask and letting them work on distinct regions of the same (shared) data structure.

TABLE V
The benchmarks selected. Columns *Add* and *Del* indicate the number of additions and deletions applied by FJDetector. "Rep?" means "Replied?" and "Acc?" means "Accepted?". The symbols ✓, × and — mean, respectively, "accepted", "not accepted", and 'no response".

Projects	Add	Del	Rep?	Acc?	Savings
`itemupdown`	13	7	—	—	8.23%
`jAcer`	14	8	✓	✓	4.21%
`educational`	13	17	—	—	18.51%
`scalatuts`	12	6	✓	✓	12.41%
`knn`	20	8	✓	✓	21.3%
`netflixoss`	17	13	—	×	2.18%
`doms-transformers`	20	9	✓	—	3.82%
`ForkAndJoinUtility`	13	6	✓	✓	21.17%
`exhibitor`	21	15	✓	—	1.23%
`Solitaire`	14	5	—	—	14.12%
`javaOneBR-2012`	13	4	✓	✓	22.21%
`mywiki`	17	18	—	—	16.12%
`ejisto`	18	9	✓	✓	3.2%
`cq4j`	14	7	—	—	11.23%
`MagicSquares`	12	11	✓	—	23.51%

B. FJDetector *Results*

We have applied FJDetector to 15 of the benchmarks listed in §III. The benchmarks were selected due to the presence of Bottleneck 2 (§V). Table V lists the selected benchmarks. We assess FJDetector in terms of the following evaluation questions:

- **EQ1.** Is our approach *useful*?
- **EQ2.** How intrusive is FJDetector?

Results of EQ1 To answer **EQ1**, we have sent modified versions of the benchmarks to their developers as patches. If these matches are useful, they will eventually be merged into the benchmarks. To assess the intrusiveness of FJDetector, we measured the number of lines of code that FJDetector adds to and removes from the benchmarks in order to refactor them. A large number of modifications makes the code harder to understand and modify for its developers.

With FJDetector, 18 instances of refactorings were performed over 15 projects. We sent these modified versions as patches to the owners of the corresponding repositories via the pull request feature of Github. On Table V, columns "Replied?" and "Accepted?" flag the projects that have replied and accepted our patch. 9 projects have replied showing an intention to accept our patch. One project is no longer actively maintained (`doms-transformers`). For the remaining 8 projects that replied, 7 of them have already accepted and merged our patches.

Benchmark `netflixoss` was the only project that closed our pull request with no response. This particular project seems to be a fork from another existing project (it has 231,361 lines of Java code performed by a single developer in a single commit), and does not seem to be maintained anymore. The owners of the remaining 7 projects did not provide any comments for our patches.

Results of EQ2 To answer EQ2, we measured the number of new statements that were added to and the number existing statements that were deleted from the benchmarks. A large

number of modifications can produce code that is hard to understand and modify. So, a refactoring that results in a small number of modifications is desirable.

Overall, our approach has added 231 statements and removed 143 ones to the 15 benchmarks. Considering that one of them has 4 instances of Bottleneck 2, the mean number of modifications for each transformation was 12.8 additions and 7.9 deletions. Thus, our approach is not very intrusive. Most of the additions are due to the addition of a new constructor, which means that preexisting code, e.g., the `compute` method, is the target of only a few modifications. The refactoring of the parallel code added an average 5.3 new statements. Deletions have different explanations. For instance, most of the deletions on project `exhibitor` are due rewriting the parallel computation (10 out of the 15 deletions). Initially, this project used a more verbose approach, iterating through the data structure, creating and forking each new parallel task, and joining them at the end. We simplified this computation by just using the `invokeAll` method, as shown Figure 11.

```
protected List<ServerStatus> compute() {
  for (List<ServerSpec> subList : Lists.partition(
      specs, size / 2)) {
    Task task = new Task(exhibitor, subList);
    task.fork(); tasks.add(task);
  }
  for (Task task : tasks) {
    statuses.addAll(task.join());
  }}
```

⇓

```
protected List<ServerStatus> compute() {
  // ...
  int split = (from + to)/2;
  invokeAll(
    new Task(exhibitor, specs, from, split),
    new Task(exhibitor, specs, from + split, to)
  );
  // ...
}
```

Fig. 11. FJDETECTOR Refactor Example.

VII. Related Work

Parallel programming is a well-established topic. In the last decade, efforts have been made on introducing novel programming models [23], [24], as well as performance [25], [26], programmer effort, satisfaction and error-proneness [27], [28] and even energy consumption [13], [29] evaluations.

There exists a considerable number of studies about the characteristics of bugs in modern software systems, including concurrency bugs [30], performance bugs [31], and, more recently, bugs in the cloud [32], [33]. Closely related to this work are empirical studies focusing on uses and misuses of concurrent libraries have been conducted [34], [35], [36]. For instance, the `java.util.concurrent` package, in which the FORKJOIN framework resides, is the focus of a large-scale study, covering over 2,000 projects [10]. However, this work does not consider the FORKJOIN framework. Although the work of Dig *et al.* [37] considered the FORKJOIN framework when converting sequential code to parallel code, the authors did not studied anti-patterns related to FORKJOIN usage.

Okur *et al.* [36] observed that misuses can account for 10% of the overall uses of parallel libraries. In the worst case, these misues can make the code run sequentially instead of concurrently. Lin *et al.* [34] found that, even though Java's Concurrent collections provide thread-safe implementations, when composing two or more operations, developers can naïvely misuse these collections and introduce atomicity violations. Other studies propose tools that correct other common mistakes (*e.g.,* [35], [38]). These studies are complementary to ours since none of them focus on the FORKJOIN framework.

A recent study [13] investigated the impact of three threading constructs on application energy consumption, one of which is the FORKJOIN framework. This study found that the energy consumption of a FORKJOIN program is sensitive to the degree of parallelism achievable by the program: it outperforms two other concurrent programming models in applications that are embarrassingly parallel, but underperforms in the presence of large numbers of serial operations. This study did not investigate specific bottenecks faced by FORKJOIN applications.

Another study most closely related to our own was conducted by DeWael *et al.* [39]. In this study, the authors analyzed Java applications that employ FORKJOIN to understand how real-world developers use ForkJoin. However, the authors did not discuss on how the antipatterns identified can be removed. Neither did they analyze the impact of the antipatterns on energy consumption.

VIII. Conclusions

This paper describes a comprehensive study on parallelism bottlenecks in FORKJOIN applications. Based on an in-depth analysis over AKKA, together with 30 open-source FORKJOIN applications on GitHub, we present a taxonomy of 6 bottlenecks, whose removal and mitigation may lead to performance improvement and energy savings. We sent our patches to the developers of 15 projects, and 7 out of the 9 projects that replied to our patches have accepted them.

The bottlenecks we have identified in this paper largely group into three categories: thread management (Bottleneck 1), data management (Bottlenecks 2, 3, and 4), and synchronization management (Bottlenecks 5 and 6). We believe the applicability of identifying and overcoming these bottlenecks may go beyond FORKJOIN. In the future, we plan to generalize these findings, and investigate their applicability on other multi-threaded language runtimes.

Acknowledgements

We would like to thank the reviewers for their valuable comments. This work is partially supported by CNPq (406308/2016-0, 453611/2017-6, 304755/2014-1), PROPESP/UFPA, FACEPE (APQ-0839-1.03/14), FACEPE PRONEX (APQ 0388-1.03/14), NSF CCF-1526205, NSF CNS-1613023, ONR N00014-14-1-0549, and ONR N00014-16-1-2913.

REFERENCES

[1] D. Lea, "A java fork/join framework," in *Java Grande*, 2000, pp. 36–43.

[2] T. J. Tutorials, "Parallelism (The Java Tutorials > Collections > Aggregate Operations)," https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html, accessed: 2015-07-02.

[3] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, Feb. 2009.

[4] Groovy, "The groovy programming language," http://www.groovy-lang.org/, 2015, [Online; accessed 12-Aug-2015].

[5] Clojure, "A dynamic programming language that targets the java virtual machine," http://www.clojure.org/, 2015, [Online; accessed 12-Aug-2015].

[6] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, "Bws: Balanced work stealing for time-sharing multicores," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, 2012, pp. 365–378.

[7] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew, "An adaptive task creation strategy for work-stealing scheduling," in *CGO '10*, 2010, pp. 266–277.

[8] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, "Adaptive work-stealing with parallelism feedback," *ACM Trans. Comput. Syst.*, vol. 26, pp. 7:1–7:32, 2008.

[9] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, 2014, pp. 513–528.

[10] G. Pinto, W. Torres, B. Fernandes, F. Castor, and R. S. Barros, "A large-scale study on the usage of javas concurrent programming constructs," *Journal of Systems and Software*, vol. 106, no. 0, pp. 59 – 81, 2015.

[11] 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*, ser. PLDI '98, 1998, pp. 212–223.

[12] "`Akka`, http://akka.io/."

[13] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14, 2014, pp. 345–360.

[14] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'15, 2015.

[15] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10, 2010, pp. 189–194.

[16] "`actors`, https://github.com/plokhotnyuk/actors."

[17] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, pp. 419–430.

[18] S. Park, W. Jiang, Y. Zhou, and S. Adve, "Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2007.

[19] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04, 2004, pp. 14–.

[20] L. Zhang, C. Krintz, and P. Nagpurkar, "Language and virtual machine support for efficient fine-grained futures in java," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07, 2007, pp. 130–139.

[21] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 169–180.

[22] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, ser. RACES '12, 2012, pp. 51–60.

[23] A. Kulkarni, Y. D. Liu, and S. F. Smith, "Task types for pervasive atomicity," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10, 2010, pp. 671–690.

[24] H. Miller, P. Haller, and M. Odersky, "Spores: A type-based foundation for closures in the age of concurrency and distribution," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, ser. Lecture Notes in Computer Science, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 308–333.

[25] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 33–48.

[26] R. Hassani, A. Malekpour, A. Fazely, and P. Luksch, "High performance concurrent multi-path communication for mpi," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12, 2012, pp. 285–286.

[27] F. Castor, F. Soares-Neto, and A. L. M. Santos, "A preliminary assessment of haskell's software transactional memory constructs," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13, 2013, pp. 1696–1697.

[28] V. Pankratius, F. Schmidt, and G. Garreton, "Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 123–133.

[29] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java's thread-safe collections," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 2016, pp. 20–31.

[30] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, 2008, pp. 329–339.

[31] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 77–88.

[32] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, 2014, pp. 249–265.

[33] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014, pp. 7:1–7:14.

[34] Y. Lin and D. Dig, "Check-then-act misuse of java concurrent collections," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST, 2013.

[35] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 341–352.

[36] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 54:1–54:11.

[37] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 397–407.

[38] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15, 2015, pp. 224–235.

[39] M. De Wael, S. Marr, and T. Van Cutsem, "Fork/join parallelism in the wild: Documenting patterns and anti-patterns in java programs using the fork/join framework," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14, 2014, pp. 39–50.