# Featherlight Speculative Task Parallelism

Vivek Kumar(✉)

IIIT-Delhi, New Delhi, India
vivekk@iiitd.ac.in

**Abstract.** Speculative task parallelism is a widely used technique for solving search based irregular computations such as graph algorithms. Here, tasks are created speculatively to traverse different search spaces in parallel. Only a few of these tasks succeed in finding the solution, after which the remaining tasks are canceled. For ensuring timely cancellation of tasks, existing frameworks either require programmer introduced cancellation checks inside every method in the call chain, thereby hurting the productivity, or provide limited parallel performance.

In this paper we propose *Featherlight*, a new programming model for speculative task parallelism that satisfies the serial elision property and doesn't require any task cancellation checks. We show that Featherlight improves productivity through a classroom-based study. Further, to support Featherlight, we present the design and implementation of a task cancellation technique that exploits runtime mechanisms already available within managed runtimes and achieves a geometric mean speedup of 1.6× over the popular Java ForkJoin framework on a 20 core machine.

**Keywords:** Speculative parallelism · Async-finish programming · Task cancellation · Managed runtimes · Work-stealing

## 1 Introduction

With the advent of multicore processors, variants of tasks based parallel programming models [1,3,9,10,12,18,20,22] have gained a lot of popularity. They are extremely well suited for parallelizing irregular computations such as graph search algorithms. Some well-known examples are route planning in navigation and guidance systems, searching for entities on social networking sites like people and places, and finding the winning move in a game tree. Programmers using these parallel programming frameworks expose a parallel task and rely on an underlying work-stealing [10] runtime for dynamic load balancing. These frameworks often satisfy the serial elision for basic tasking support, the property that is eliding all parallel constructs results in a valid sequential program [10]. While serial elision improves programmer's productivity, using an underlying work-stealing runtime improves the parallel performance over multicore processors. However, this is not the case when using these frameworks for applications requiring speculative task parallelism, where only a few tasks could provide desirable results, as all remaining tasks should terminate after the goal is found.


R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 391–404, 2019.
https://doi.org/10.1007/978-3-030-29400-7_28

Productivity becomes a first-order concern as several frameworks such as Intel TBB [22], C# [1], X10 [9] and TryCatchWS [18], require programmer inserted task cancellation checks inside every method in the call chain for timely cancellation of speculative tasks. Cilk [10] provides special support for task cancellation but the programmer has to implement an *inlet* that is essentially a C function internal to a Cilk procedure. The Java *fork/join* framework [20] does not support serial elision but provides a *shutdownNow* API for global task cancellation. OpenMP 4.0 [3] tasking pragmas and Eureka programming in HJlib [13] provide task cancellation checks, but the programmer must ensure optimal granularity for calling these checks to avoid performance degradation.

In this paper, we introduce a new programming model, *Featherlight*, for speculative task parallelism that doesn't require any form of task cancellation checks, and improves the productivity by satisfying the property of serial elision. For achieving high performance, Featherlight exploits runtime mechanisms already available within managed runtimes, namely: (a) yieldpoint mechanism [21], (b) ability to walk the execution stack of a running thread, and (c) support for exception delivery. We use six well-know search based micro-benchmarks and one real-world application from Dacapo benchmark suite [8] to compare the productivity and performance of Featherlight with the ForkJoin framework, and also with an approach that uses hand-coded implementation of the task cancellation policy. We use lines of code and time to code based empirical analysis to demonstrate high productivity in Featherlight. We show that Featherlight is highly competitive. It achieves performance comparable to the hand-coded implementation and significantly better than the ForkJoin implementation.

In summary, this paper makes the following contributions:

- *Featherlight*, a new task-based parallel programming model for speculative parallelism that satisfies serial elision property.
- A lightweight runtime implementation that supports Featherlight by exploiting existing managed runtime techniques.
- Productivity analysis of Featherlight by using it in a classroom-based study.
- Performance evaluation of Featherlight as compared to Java ForkJoin framework and a hand-coded implementation of task cancellation policy by using seven popular search based problems on a 20 core machine.

## 2 Background

### 2.1 Async-Finish Programming Model

Cilk language [10] popularized task parallelism by using *spawn* and *sync* keywords for creating and joining a parallel task. For scheduling these tasks, an underlying work-stealing runtime is employed that maintains a pool of *worker* threads, each of which maintains a double-ended queue (*deque*) of

```
1 void baz() {
2   finish {
3     async S1();
4     S2();
5   }
6   S3();
7 }
```

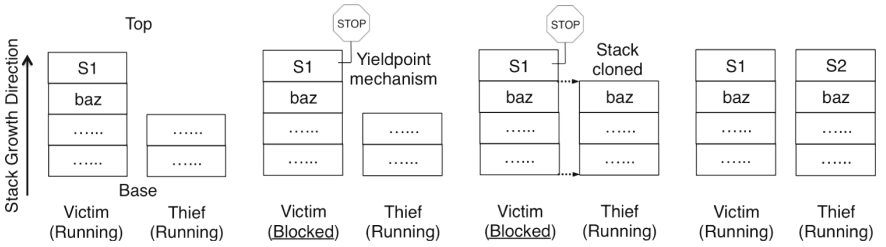
**Fig. 1.** An **async**–**finish** program

tasks. When the local deque becomes empty, the worker becomes a *thief* and seeks a *victim* thread from which to *steal* work. Likewise, Java supports a ForkJoin framework [20], and X10 language [9] introduced **async**–**finish** constructs for task parallelism. This **async**–**finish** construct satisfy serial elision and have been adopted by other frameworks such as Habanero Java library (HJlib) [12], Habanero C library (HClib) [19], and TryCatchWS [18]. Featherlight is built on top of TryCatchWS and supports **async**–**finish**. Figure 1 shows a sample TryCatchWS program that uses **async**–**finish** constructs. The **async** clause at Line 3 creates a task *S1*, which can run in parallel with the continuation *S2*. An **async** can be used to enable any statement to execute as a parallel task, including statement blocks, *for* loop iterations, and function calls. A **finish** is a generalized join operation. Lines 2–5 encloses a **finish** scope and ensures that both the parallel tasks *S1* and *S2* has completed before starting the computation *S3* at Line 6. Both **async** and **finish** can be arbitrarily nested.

### 2.2 Managed Runtime Services

Managed runtimes gained popularity with the advent of the Java language and have been a very active research area since then. Some of the key features of managed runtimes exploited in Featherlight are (a) yieldpoint mechanism, (b) ability to walk the execution stack of a running thread and (c) support for exception delivery. Yieldpoints are the program locations where it is *safe* to run a garbage collector and implement services such as adaptive optimization. The compiler generates yieldpoints as program points where a running thread checks a dedicated bit in a machine control register to determine whether it should yield. The compiler generates precise stack maps at each yieldpoint. If the bit is set, the yieldpoint is taken and some action is performed. If the bit is not set, no action is taken and the next instruction after the yieldpoint is executed. In JikesRVM virtual machine [7], yieldpoints are inserted in method prologues and on loop back edges.

Exception handling is also an important feature that is well supported by many modern programming languages like C++, Java, and C#. An exception is an event that can occur during the execution of the program and disrupts the program's control flow. Handlers are subroutines where the exceptions are resolved, and they may allow resuming the execution from the original location of the exception. Java uses *try* and *catch* blocks for exception handling. Exception delivery mechanism and other runtime services such as garbage collection are supported in a managed runtime with the ability to walk the execution stack of any thread. For walking the stack of a running thread (victim), the victim is first stopped by using yieldpoint mechanism. The victim saves all its registers and live program states before stopping. The thread requesting the stack walk can then easily go over each frame on victim's execution stack and can manipulate them as well. The garbage collector uses a stack walk to identify live and dead objects in the application thread.




(a) The initial state of execution stacks of the victim and the thief. Victim is executing **async** task `S1` while thief is about to steal the continuation of this **async** (i.e., `S2`) from this victim

(b) The thief sets a thread-local conditional variable in the victim as **true**. Victim discovers this while executing method prologues or loop back edges and stops

(c) Thief clones the victim's execution stack up to the stack frame `baz` as this method contains the oldest unstolen continuation (`S2`)

(d) Thief replaces its original execution stack with this cloned execution stack. It then releases the victim from yieldpoint and throws an exception to start the execution of `S2`

**Fig. 2.** TryCatchWS work-stealing implementation for executing the **async**–**finish** program shown in Fig. 1

### 2.3 TryCatchWS Work-Stealing Runtime

We have implemented the runtime support for Featherlight by modifying Java TryCatchWS work-stealing runtime developed by Kumar et al. [18]. TryCatchWS is implemented directly inside JikesRVM [7] Java Virtual Machine, and as demonstrated by Kumar et al. [16,18], it helps in achieving both good scalability and good absolute performance. Figure 2 illustrates the TryCatchWS work-stealing implementation for executing the **async**–**finish** program shown in Fig. 1 by using two workers. The user code written by using **async**–**finish** is first translated to plain Java code by using AJWS compiler [17]. TryCatchWS follows the work-first principle for task scheduling. The generated code exploits the semantics Java offers for exception handling, which is very efficiently implemented in most modern JVMs. The result is that the runtime does not need to maintain *explicit* deques. It uses the Java thread (execution) stack of both the victim and thief as their *implicit* deque. The thief can directly walk a victim's execution stack and identify all **async** and **finish** contexts, resulting in a significant reduction in overheads. This allows the programmer to expose fine granular tasks without worrying about the task creation overheads. For a detailed overview of TryCatchWS we refer readers to [18].

## 3 Featherlight Programming Model

Featherlight extends **async**–**finish** programming supported by TryCatchWS with two new constructs:

– **finish_abort**: This is the regular **finish** with an added responsibility to ensure graceful cancellation of speculative **async** tasks without having any cancellation condition checking code. Both **async** and **finish** can be

```
class UTS {
 boolean found=false;
 void search(){
  finish_abort recurse(root);
 }
 void recurse(Node n){
  if(n.equals(goal){
   found=true;
   abort;
   return;
  }
  for(int i=0;i<n.nChild;i++){
   async recurse(n.child[i]);
  }
 }
}
```

(a) Featherlight

```
class UTS {
 /*atomic cancelation token*/
 AtomicBoolean found=new AtomicBoolean();
 void search(){
  finish recurse(root);
 }
 void recurse(Node n){
  if(found.get()) return; /*check*/
  if(n.equals(goal){
   found.set(true);
   return;
  }
  for(int i=0;i<n.nChild;i++){
   if(found.get()) return; /*check*/
   async recurse(n.child[i]);
  }
 }
}
```

(b) ManualAbort obtained by hand-coding task cancellation checks in TryCatchWS

```
class UTS { boolean found=false;
 ForkJoinPool pool=new ForkJoinPool(2);
 void search(){
  try {
   pool.invoke(new RecursiveAction(){
    public void compute() {
     new Recurse(root).fork();
     helpQuiesce();
    }
   });
  } catch(CancellationException e){}
 }
 class Recurse extends RecursiveAction{
  Node n;
  public Recurse(Node _n) {n=_n;}
  public void compute() {
   if(n.equals(goal){
    found=true;
    pool.shutdownNow();
   }
   for(int i=0;i<n.nChild;i++){
    new Recurse(n.child[i]).fork();
   }
  }/*compute*/ }/*Recurse*/
}
```

(c) Java ForkJoin framework

**Fig. 3.** Searching a unique node in the UTS tree by using three different implementations of speculative task parallelism. Only Featherlight supports serial elision as erasing the keywords *finish_ abort*, *async*, and *abort* will fetch the original sequential UTS.

nested inside a **finish_abort**. A **finish_abort** cannot be nested but can be called in parallel by placing it inside an **async** within a **finish** scope.

– **abort**: This construct cancels all speculative **async** tasks once the goal is found. Cilk also supports a variant of **abort** but the semantics are very different (Sect. 6). An **abort** can be called inside a Featherlight program only if there is an encapsulating **finish_abort** in the method call stack. If a worker encounters an **abort** statement, it will cancel only those **async** that are (or yet to be) spawned inside a **finish_abort** scope in this worker's method call stack. For example, in the statement "**finish**{**async**{**finish_abort**{S1;}} **async**{**finish_abort**{S2;}}}", calling an **abort** inside *S1* will only cancel task *S1* and its children (both pending and running) but will not affect the execution of *S2*.

```
1  class UTS {
2  boolean found=false;
3  void search() {
4   try {
5    Runtime.allocateNewFinishAbort();
6    try {
7     recurse(root);
8     Runtime.finish();
9    } catch(ExceptionFinish f) { }
10  } catch(ExceptionSuccessAbort sa) {
11   Runtime.updateFinishScope();
12   Object currentFA = Runtime.getFinishAbort();
13  currentFA.waitForExpectedExceptionFailAborts();
14   Runtime.allowWorkStealingAtAllWorkers();
15  } catch(ExceptionFailAbort fa) {
16   Object currentFA = Runtime.getFinishAbort();
17   currentFA.notifyExceptionFailAbort();
18   Runtime.restartAsThief();
19  }
20 }
```

```
22 void recurse(Node n) {
23  if(n.equals(goal)) found=true;
24   Object currentFA = Runtime.getFinishAbort();
25   if(currentFA.abortAlreadyInitiated()) return;
26   Runtime.pauseWorkStealingAtAllWorkers();
27   for(int w=0; w<Runtime.numWorkers(); w++) {
28    if(w == Runtime.getMyWorkerID()) continue;
29    Runtime.forceWorkerToEnterYieldpoint(w);
30    if(currentFA.equals(Runtime.getFinishAbort(w))){
31     Runtime.worker[w].throwExceptionFailAbort=true;
32    currentFA.incrementExpectedExceptionFailAborts();
33    }/*if*/ Runtime.releaseWorkerFromYieldpoint(w);
34   }/*for*/throw new ExceptionSuccessAbort();}/*If*/
35  for(int i=0;i<n.nChild;i++){
36   try { Runtime.continuationAvailable();
37    recurse(n.child[i]);
38    Runtime.checkContinuationAvailability();
39   } catch(ExceptionEntryThief e) {/*thief entry*/
40   }/*catch*/ }/*for*/ }/*recurse*/ } /*UTS*/
```

**Fig. 4.** Source-to-source translation of the code shown in Fig. 3(a) to vanilla Java. Underlined code is the default code generated by the compiler to support **async**–**finish**.

To further motivate Featherlight, in Fig. 3, we show three different implementations of UTS program (Sect. 5) as a motivating example. These implementations use speculative task parallelism for searching a unique node in the tree. Figure 3(a) shows Featherlight, Fig. 3(b) shows ManualAbort obtained by hand-coding task cancellation checks in TryCatchWS, and Fig. 3(c) shows Java ForkJoin implementation. Out of all these three implementations, only Featherlight supports serial elision. Calling an **abort** at Line 9 in Featherlight will cancel all **async** (as all **async** are inside single **finish_abort** scope), and resume the execution right after **finish_abort** (i.e., Line 5). ManualAbort requires an atomic cancellation token (Line 3), checking this cancellation token before executing the task (Line 8), and also checking before creating any new task (Line 14). These checks are prone to data races if not used properly. It could also delay task cancellation if not used inside every method in the call chain. Moreover, having multiple search criteria in the program can make it difficult to identify the program points where the code for cancellation checks should be added. Although Java ForkJoin does not require manual cancellation checks, it does not supports serial elision and requires extensive changes to the sequential program.

## 4 Design and Implementation

In this section, we briefly describe our implementation of Featherlight that builds on Java TryCatchWS work-stealing runtime (Sect. 2.3). Our implementation and the benchmarks are released open source online on GitHub [15].

For implementing Featherlight we exploit runtime mechanisms already available within managed runtimes, namely: (a) yieldpoint mechanism, (b) support for exception delivery, and (c) ability to stack walk the execution stack of a running thread. When a worker encounters an **abort** call in a Featherlight program, it will *pause* the execution of other workers by using yieldpoint mechanism. It will then identify the subset of workers that are executing **async** spawned from the same **finish_abort** scope as this worker. These shortlisted workers would

then relinquish all pending and currently running **async**, and throw special exceptions to start the computation from another program point. The insight is that the cost of canceling the speculatively spawned **async** should not be incurred in the common case and should occur only once when the goal has been found. Our contribution is to design and implement Featherlight, a novel-programming model for speculative task parallelism that implements our above insight.

### 4.1 Source Code Translation of Featherlight to Java

Kumar et al. implemented the AJWS compiler [17] that could translate an **async**–**finish** program into a plain Java program capable of running using TryCatchWS runtime. We have extended their AJWS compiler to support Featherlight by translating the two new constructs **finish_abort** and **abort** into plain Java code. Figure 4 shows this generated Java code for Featherlight's implementation of UTS, shown in Fig. 3(a). All underlined code is the default code generated by the AJWS compiler to support TryCatchWS. For details on this default code, we refer readers to [18].

### 4.2 Canceling Speculative **async** Once the Goal Is Found

In Featherlight's implementation, the worker (victim) who started the computation (Line 3) will first create an object for this new **finish_abort** scope at Line 5 and then continue its execution. Any thief who attempts to steal from this victim will use default TryCatchWS, where it will stop this victim in yield-point and perform a stack walk of victim's execution stack to find out the oldest unstolen continuation. To support Featherlight, we extended this victim's stack walk such that now the thief also searches the reachability to any *catch* block for handling *ExceptionFailAbort* (Line 15). The thief will then verify if this *catch* block is still reachable from the program point where this thief has to resume the stolen computation (Line 39). If it is still reachable, this thief will copy the object corresponding to **finish_abort** scope stored at the victim (created at Line 5). Note that the thief will overwrite its copy of this scope object if it enters another **finish_abort**.

Assume a worker *W1* has found the goal (Line 23). *W1* has to decide which other workers should cancel their **async** tasks. Essentially, these should be the workers having the same **finish_abort** scope object as with *W1*. This will ensure that if there were **async** created from another **finish_abort** scope, then they would not be canceled. *W1* will first ensure that no other worker has initiated **abort** inside this same **finish_abort** scope (Lines 24–25). `W1` will then temporarily pause the work-stealing on all workers, including itself (Line 26) to avoid deadlock. As thief also relies on the yieldpoint mechanism to steal from a victim, there could be a deadlock when the thief is attempting to steal from a victim that is, in turn, trying to yield that same thief from **abort** call. After pausing work-stealing globally, `W1` will force all other workers to execute yieldpoint one by one (Line 29). Note that at this point, if any worker

is executing a critical section by taking a mutual exclusion lock, `W1` will wait inside the call at Line 29 until this worker has released the mutual exclusion lock. After a worker has paused its execution at yieldpoint, `W1` will compare its **`finish_abort`** scope object with that of this worker (Line 30). If it matches, it will set a flag at this worker for throwing *ExceptionFailAbort* (Line 31) and increment the join counter at this **`finish_abort`** scope object (Line 32). It will then release this worker from yieldpoint (Line 33). Finally, `W1` will throw `ExceptionSuccessAbort` (Line 34) to resume its execution from Line 11.

### 4.3 `ExceptionFailureAbort` and `ExceptionSuccessAbort`

Recall that `W1` paused the execution of all other workers inside yieldpoint (Line 29) and released them after setting the flag `throwExceptionFailureAbort` (Line 31) in some of them. After resuming the execution inside yieldpoint, every worker will first check and reset their flag `throwExceptionFailureAbort`. Those who found `throwExceptionFailureAbort` set to **`true`** will throw `ExceptionFailureAbort` from yieldpoint to resume their execution from Line 16. They will first decrement the join counter at this **`finish_abort`** scope object (Line 17), and if the counter value reaches zero, this information is broadcasted to `W1` who is currently waiting at Line 13.

After setting the flag in relavant workers (Line 31), worker `W1` will throw `ExceptionSuccessAbort` at Line 34 and resume its execution from Line 11. It will first update its immediate **`finish`** scope at Line 11 (if any) and then wait until all the other relevant workers have resumed their execution inside **`catch`** block for `ExceptionFailureAbort`. Finally, `W1` will allow all workers to resume their work-stealing (Line 14). After this, it will continue the execution of user code from Line 19 onward.

## 5 Experimental Evaluation

We have used six well-known search based micro-benchmarks and one application from Java DaCapo benchmark suite [8] for our experimental evaluation:

**UTS**. Variant of Unbalanced Tree Search [24] where it searches for a specific goal node in the tree. We used T1 configuration (geometric tree) with a maximum height of 10. Applications that fit in this category include many search and optimization problems that must enumerate an ample state space of the unknown or unpredictable structure.

**LinearSearch**. It searches for 10 items in a 2D array of size $1000 \times 20000$.

**NQueens**. Goal is to find 20% of total possible solutions for placing 14 queens on a $14 \times 14$ board such that no two queens can attack each other [18]. This benchmark uses a backtracking algorithm that is also used for solving constraint satisfaction and combinatorial optimization based problems.

**SLP**. It adds edge weights in UTS and then finds shortest and longest path from the root to goal nodes within a given range of 1058–7563 [13]. We used T3 configuration (binomial tree) with a maximum height of 10 and a total of 254 goal nodes. This algorithm is also used for a large variety of optimization problems in network and transportation analysis.

**Sudoku**. This solves a Sudoku puzzle by exploring a game tree [13]. Board size was $16 \times 16$ and a total of 148 unsolved entries.

**TSP**. Traveling salesman problem [13] for 20 cities that searches for a path $\leq 156$. Similar to SLP, TSP is used in complex optimization problems such as planning and scheduling.

**lusearch**. Variant of lusearch-fix from Java DaCapo benchmark suite [8]. It uses Apache Lucene [23] for text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. In this variant, the search terminates when 80% of total search queries are completed.

To ensure serial elision we did not control task granularity in any of the above benchmarks. We implemented four different versions for each benchmark: (a) *Featherlight* that uses **`async`**, **`finish_abort`**, and **`abort`** constructs; (b) *ManualAbort* that replaces **`finish_abort`** with **`finish`** in Featherlight, removes **`abort`**, and uses token-based cancellation checks to cancel speculatively spawned **`async`**; (c) *ForkJoin* based on Java **`ForkJoinPool`** that uses **`shutdownNow`** library call for task cancellation; and (d) *Sequential* Java that is the serial elision of Featherlight.

The benchmarks were run on a dual socket 10 core Intel Xeon E5-2650 v3 processor running at 2.3 GHz and with a total of 96 GB of RAM. The operating system was Ubuntu 16.04.3 LTS. We have ported Kumar et al.'s TryCatchWS runtime on JikesRVM GitHub version 087d300. This version of TryCatchWS was used for the evaluation of ManualAbort. Sequential Java version of each benchmark was run directly on the above version of unmodified JikesRVM. Featherlight was implemented and evaluated on the above mentioned TryCatchWS version. For all evaluations, we used the production build of JikesRVM. Fixed heap size of 3 GB and single garbage collector thread was used across all experiments. We bound work-stealing worker threads to individual CPU cores and did not use hyper-threading. Other than this, we preserved the default settings of JikesRVM. AJWS compiler version bd5535f on GitHub was extended to support code generation for Featherlight. For each benchmark, we ran 30 invocations, with 15 iterations per invocation where each iteration performed the kernel of the benchmark. In each invocation, we report the mean of the final five 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).

### 5.1 Productivity Analysis

Program size or Lines of Code (LoC) is a widely used metric for measuring software productivity [13,17]. Table 1 shows this number for each benchmark and its corresponding four variants. A support code is the common code across

**Table 1.** Productivity metrics in terms of LoC in actual implementations and time spent by students in classroom [6] for implementing speculative task parallelism.

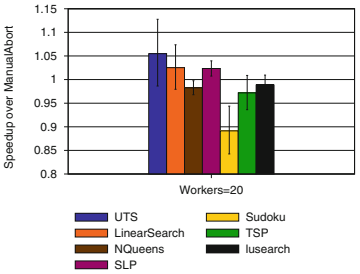
| | Lines of Code (LoC) generated using David A. Wheeler's 'SLOCCount' [26] | | | | | Time (minutes) spent by students | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Featherlight | | | ForkJoin | | |
| Benchmark | Common code | Sequential | Featherlight | ManualAbort | ForkJoin | Subjects | Mean | St.Dev. | Subjects | Mean | St.Dev. |
| **UTS** | 545 | 39 | 39 | 45 | 58 | 9 | 8.6 | 6.1 | 9 | 52.2 | 13.5 |
| **LinearSearch** | 88 | 44 | 44 | 46 | 75 | - | - | - | - | - | - |
| **NQueens** | 75 | 48 | 48 | 53 | 68 | 7 | 13.6 | 10.2 | 8 | 61 | 17.2 |
| **SLP** | 558 | 54 | 54 | 60 | 76 | 6 | 11.7 | 4.1 | 8 | 43.4 | 16.6 |
| **Sudoku** | 469 | 48 | 48 | 54 | 66 | 6 | 6 | 2.6 | 8 | 58.8 | 11.6 |
| **TSP** | 158 | 55 | 55 | 61 | 84 | 7 | 10.4 | 4 | 8 | 53.1 | 24.8 |
| **lusearch** | >126K | 222 | 222 | 242 | 239 | - | - | - | - | - | - |

all four variants of a benchmark. It is highest for lusearch as it uses Apache Lucene for text search. Featherlight supports serial elision and has same LoC as in Sequential. ManualAbort has more LoC than Sequential as it has hand-coded task cancellation checks. ForkJoin requires significant modifications to the Sequential and hence has the maximum LoC.

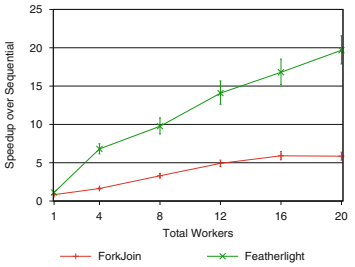
We also conducted an empirical study [6] to quantify programming effort based on the time required for programming Featherlight and ForkJoin implementations. This classroom study was of 90 min and involved 41 senior undergraduate and postgraduate students from an introductory parallel programming course at IIIT Delhi (CSE502, Spring 2019 semester). These students were first taught about speculative task parallelism and were provided with working copies of both Featherlight and ForkJoin implementations of LinearSearch benchmark as a reference. For this study, each student was provided with two different Sequential benchmarks (except lusearch due to its cumbersome setup). They were then asked to implement ForkJoin version of one of these two Sequential benchmarks and Featherlight version of the other one. We recorded the time taken by students for both these implementations and report the average time along with the standard deviation in Table 1. Average time required for Featherlight implementation ranged between 6–13.6 min verses 43.4–61 min for ForkJoin. Support for serial elision and lesser time to code demonstrate that Featherlight is extremely effective in enhancing programmer's productivity, an important consideration given the current hardware trend and the plethora of real-world search based problems existing today.

### 5.2 Performance Analysis

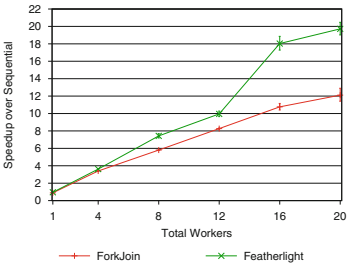
Figure 5(a) shows the speedup of Featherlight relative to ManualAbort for all benchmarks by using 20 workers. Except for Sudoku, both Featherlight and ManualAbort perform within 5% of each other. This shows that Featherlight implementation does not add significant runtime overheads. Figures 5(b)–(g) shows the speedup of both Featherlight and ForkJoin relative to Sequential implementation for each of the benchmarks. We can observe that Featherlight can achieve significant speedup over the Sequential counterpart. Featherlight was also able to



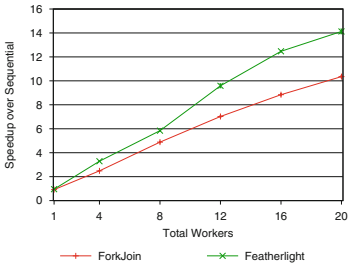

(a) Speedup of Featherlight over ManualAbort by using 20 workers



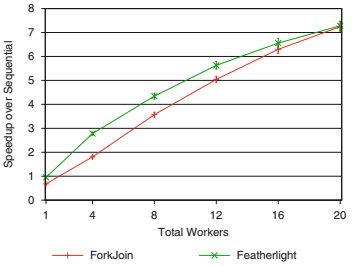

(b) UTS speedup over Sequential



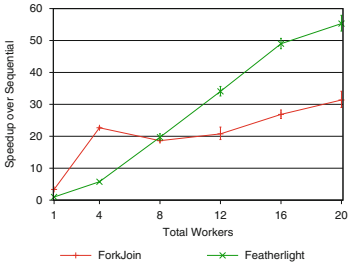

(c) LinearSearch speedup over Sequential



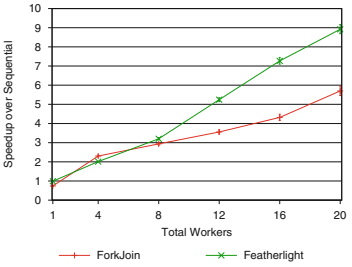

(d) NQueens speedup over Sequential



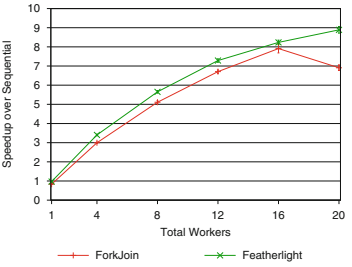

(e) SLP speedup over Sequential




(f) Sudoku speedup over Sequential




(g) TSP speedup over Sequential




(h) lusearch speedup over Sequential

**Fig. 5.** Performance analysis of Featherlight and ForkJoin on a 20 core machine. Some benchmarks achieve super-linear speedup, which is possible in speculative parallel programming.

outperform ForkJoin across all benchmarks by increasing the parallelism (except for SLP at 20 workers). Average speedup of Featherlight over ForkJoin across all benchmarks and by using 20 workers was 1.7× with a geometric mean of 1.6×. Few benchmarks (UTS, LinearSearch, and Sudoku) achieved super-linear speedup. It is possible as speculative decomposition may change the amount of work done in parallel, thereby resulting in either sub-linear or super-linear speedups [11].

For Sudoku benchmark, ForkJoin performed better than Featherlight at lower worker counts. ForkJoin follows help-first work-stealing policy [27], i.e., **`fork`** call merely pushes the task on the deque, and the continuation is executed before the task. Featherlight follows work-first work-stealing policy, i.e., **`async`** is executed before the continuation. We found that due to this difference, ForkJoin created a lesser number of tasks in Sudoku than Featherlight, thereby performing better at lower worker count. However, Featherlight outperformed ForkJoin by increasing the parallelism.

We also noticed that unlike all other benchmarks, calling **`shutdownNow`** in ForkJoin implementation of lusearch only partially canceled the tasks. As per Javadoc [5], **`shutdownNow`** is typically implemented via *Thread.interrupt()*, so any task that fails to respond to interrupts may never terminate. We found it to be true in this case as there are several *catch* blocks for *InterruptedException* inside Apache Lucene codebase over which lusearch is implemented. This is a limitation that Featherlight does not suffer as long as appropriate *Exception* subclasses are used instead of *catch(Exception e){}* blocks.

## 6 Related Work

Kolesnichenko et al. provided a detailed classification of task cancellation patterns [14]. Java ForkJoin [20], Scala [4] and Python [2] simply terminate all the threads once a cancellation is invoked by the user. Cilk allows speculative work to be canceled through the use of Cilk's **`abort`** statement inside function-scoped inlets [10]. The **`abort`** statement, when executed inside an inlet, causes cancellation of only the extant children but not the future children. For preventing future children from being spawned, users should set a flag in the inlet indicating that an abort has taken place, and then test that flag in the procedure before spawning a child. This approach differs in Featherlight as an **`abort`** will cancel both extant and future tasks inside the scope of **`finish_abort`** without the need of any cancellation flag. OpenMP supports a task cancellation pragma that allows grouping of tasks that could be canceled [3]. However, cancellation could only be trigger by user-provided cancellation checks on task cancellation pragma. Although OpenMP supports serial elision, user provided cancellation checks hampers the productivity. Unlike Featherlight, both Cilk and OpenMP have another limitation that cancellation is not possible when the code is executing in a nested function call (long running tasks). TBB users must use either cancellation tokens or interrupt checking for task cancellation [22,25]. C# supports cooperative cancellation by using cancellation token checking and throwing an exception once the result is found [1]. Eureka programming in HJlib [13]

allows the user to identify the program points where the task could be canceled by using the runtime provided cancellation check call. A drawback is that the programmer has to determine the frequency (granularity) of the check calls to avoid overheads.

Featherlight radically differs from all existing approaches in following ways: (a) it improves the productivity by satisfying serial elision, (b) it doesn't require any cancellation checks inside **`async`** tasks, and (c) it uses managed runtime techniques to gracefully and safely cancel all speculatively spawned **`async`** when **`abort`** is called.

## 7 Conclusion

Several modern real-world applications are comprised of search based problems that perform best by using speculative parallelism. This parallel programming technique often requires programmer inserted cancellation checks inside speculatively spawned parallel tasks to terminate them once the search result has been found. In this paper, we designed and implemented a new programming model for speculative task parallelism that improves the programmer productivity by removing the need for any cancellation checking code and by satisfying serial elision. It uses existing mechanisms in modern managed runtimes to cancel all ongoing and pending computations once the search result is found. Our empirical results demonstrate that we can achieve better productivity and performance compared to traditional approaches for speculative task parallelism.

**Acknowledgments.** The author is grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper, and to Imam et al. for open-sourcing HJlib micro-benchmarks for speculative task parallelism [13].

## References

1. Destroying threads in C#. https://docs.microsoft.com/en-us/dotnet/standard/threading/destroying-threads. Accessed Feb 2019
2. Documentation on The Python standard library. https://docs.python.org/3/library/concurrent.futures.html. Accessed Feb 2019
3. OpenMP API, version 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Accessed Feb 2019
4. Scala scheduler. https://doc.akka.io/docs/akka/snapshot/scheduler.html?language=scala. Accessed Feb 2019
5. Oracle docs, February 2019. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html
6. Productivity analysis, February 2019. https://www.usebackpack.com/iiitd/m2018/cse000
7. Alpern, B., et al.: The Jalapeño virtual machine. IBM Syst. J. **39**(1), 211–238 (2000). https://doi.org/10.1147/sj.391.0211
8. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA, pp. 169–190 (2006). https://doi.org/10.1145/1167473.1167488

9. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, pp. 519–538 (2005). https://doi.org/10.1145/1094811.1094852
10. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI, pp. 212–223 (1998). https://doi.org/10.1145/277650.277725
11. Grama, A., Kumar, V., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Pearson Education, Upper Saddle River (2003)
12. Imam, S., Sarkar, V.: Habanero-Java library: a Java 8 framework for multicore programming. In: PPPJ, pp. 75–86 (2014). https://doi.org/10.1145/2647508.2647514
13. Imam, S., Sarkar, V.: The Eureka programming model for speculative task parallelism. In: ECOOP, vol. 37, pp. 421–444 (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.421
14. Kolesnichenko, A., Nanz, S., Meyer, B.: How to cancel a task. In: Lourenço, J.M., Farchi, E. (eds.) MUSEPAT 2013. LNCS, vol. 8063, pp. 61–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39955-8_6
15. Kumar, V.: Featherlight implementation (2019). https://github.com/hipec/featherlight/archive/d60047a.tar.gz
16. Kumar, V., Blackburn, S.M., Grove, D.: Friendly barriers: efficient work-stealing with return barriers. In: VEE, pp. 165–176 (2014). https://doi.org/10.1145/2576195.2576207
17. Kumar, V., Dolby, J., Blackburn, S.M.: Integrating asynchronous task parallelism and data-centric atomicity. In: PPPJ, pp. 7:1–7:10 (2016). https://doi.org/10.1145/2972206.2972214
18. Kumar, V., Frampton, D., Blackburn, S.M., Grove, D., Tardieu, O.: Work-stealing without the baggage. In: OOPSLA, pp. 297–314 (2012). https://doi.org/10.1145/2398857.2384639
19. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: a compiler-free PGAS library. In: PGAS, pp. 5:1–5:10 (2014). https://doi.org/10.1145/2676870.2676879
20. Lea, D.: A Java fork/join framework. In: JAVA, pp. 36–43 (2000). https://doi.org/10.1145/337449.337465
21. Lin, Y., Wang, K., Blackburn, S.M., Hosking, A.L., Norrish, M.: Stop and go: understanding yieldpoint behavior. In: ISMM, pp. 70–80 (2015). https://doi.org/10.1145/2754169.2754187
22. Marochko, A.: Exception handling and cancellation in TBB–Part II (2008)
23. McCandless, M., Hatcher, E., Gospodnetic, O.: Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Manning Publications Co., Greenwich (2010)
24. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: LCPC, pp. 235–250 (2007). http://dl.acm.org/citation.cfm?id=1757112.1757137
25. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2005)
26. Wheeler, D.A.: SLOCCount (2001). http://www.dwheeler.com/sloccount/
27. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for Async-finish task parallelism. In: IPDPS, pp. 1–12 (2009). https://doi.org/10.1109/IPDPS.2009.5161079