Work-Stealing Without The Baggage [*]

Vivek Kumar[†], Daniel Frampton[†§], Stephen M. Blackburn[†], David Grove[‡], Olivier Tardieu[‡]

[†]Australian National University [§]Microsoft [‡]IBM T.J. Watson Research

Abstract

Work-stealing is a promising approach for effectively exploiting software parallelism on parallel hardware. A programmer who uses work-stealing explicitly identifies potential parallelism and the runtime then schedules work, keeping otherwise idle hardware busy while relieving overloaded hardware of its burden. Prior work has demonstrated that work-stealing is very effective in practice. However, work-stealing comes with a substantial overhead: as much as 2× to 12× slowdown over orthodox sequential code.

In this paper we identify the key sources of overhead in work-stealing schedulers and present two significant refinements to their implementation. We evaluate our work-stealing designs using a range of benchmarks, four different work-stealing implementations, including the popular fork-join framework, and a range of architectures. On these benchmarks, compared to orthodox sequential Java, our fastest design has an overhead of just 15%. By contrast, fork-join has a 2.3× overhead and the previous implementation of the system we use has an overhead of 4.1×. These results and our insight into the sources of overhead for work-stealing implementations give further hope to an already promising technique for exploiting increasingly available hardware parallelism.

Categories and Subject Descriptors D1.3 [*Software*]: Concurrent Programming – Parallel programming; D3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures; D3.4 [*Programming Languages*]: Processors – Code generation; Compilers; Optimization; Run-time environments.

General Terms Design, Languages, Performance.

Keywords Scheduling, Task Parallelism, Work-Stealing, X10, Managed Languages.

1. Introduction

Today and in the foreseeable future, performance will be delivered principally in terms of increased hardware parallelism. This fact is an apparently unavoidable consequence of wire delay and the breakdown of Dennard scaling, which together have put a stop to hardware delivering ever faster sequential performance. Unfortunately, software parallelism is often difficult to identify and expose, which means it is often hard to realize the performance potential of modern processors. Work-stealing [3, 9, 12, 18] is a framework for allowing programmers to explicitly expose *potential* parallelism. A work-stealing scheduler within the underlying language runtime schedules work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. Work-stealing schedulers are used in programming languages, such as Cilk [9] and X10 [3], and in application frameworks, such as the Java fork/join framework [12] and Intel Threading Building Blocks [18].

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

In this paper, we analyze the source of the sequential overheads in the X10 work-stealing implementation. We then identify two substantially more efficient designs. The key to our approach is exploiting runtime mechanisms already available within managed runtimes, namely: a) dynamic compilation, b) speculative optimization via code-patching and on-stack-replacement, and c) support for exception delivery. We implement our ideas in the Jikes RVM

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

infrastructure and empirically assess them using both a language-based work stealing scheduler, X10, and a library-based framework, Java fork/join. We evaluate our implementation on a range of x86 processors and use the OpenJDK JVM to validate some of our important results. Our new designs reduce the sequential overhead of X10's work-stealing implementation to just 15%, down from a factor of 4.1×. Our implementation also performs substantially better than the alternative fork-join framework on our benchmark set.

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

2. Background

This section provides a brief overview of work-stealing before discussing the fundamental operations required to implement work-stealing. The section also describes two basic approaches for expressing work-stealing: the **`finish-async`** model (as used by X10), and **`fork-join`**.

Abstractly, work-stealing is a simple concept. *Worker threads* maintain a local set of *tasks* and when local work runs out, they become a *thief* and seek out a *victim* thread from which to *steal* work.

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

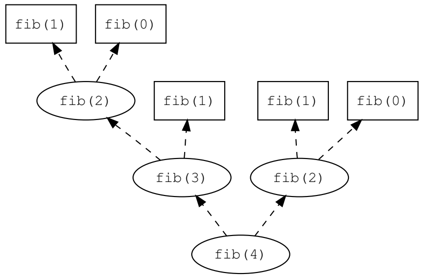
Fork A fork describes the creation of new, potentially parallel, work by a worker thread. The runtime makes new work items available to other worker threads.

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

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

2.1 An Implementation-Oriented Overview

The focus of this is paper is on identifying and reducing the sequential overheads of work-stealing, so we now turn to those issues that are pertinent to implementing work-stealing. It may help to think of the implementation of work-stealing in terms of the following basic phases, each of which require special support from the runtime or library:



(a) Execution graph for `fib(4)`

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

  val a:Int;
  val b:Int;

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

  return a + b;
}
```

(b) Coded in X10.

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

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

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

  return a + b;
}
```

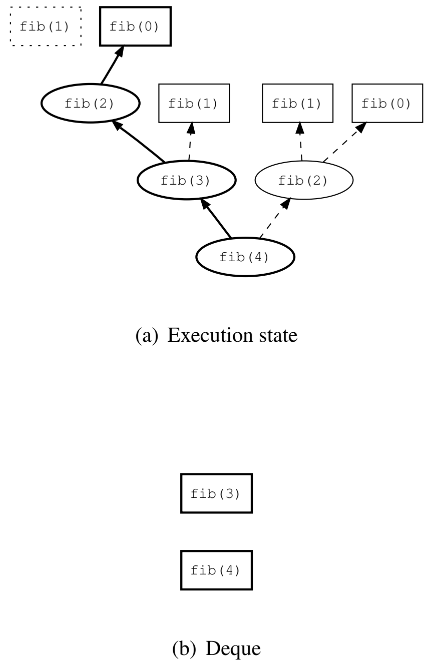
(c) Coded in Fork-Join.

Figure 1. Using work-stealing to implement Fibonacci.

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

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

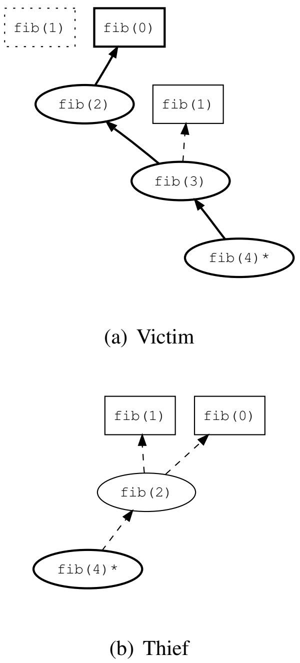
To help illustrate work-stealing, we use a running example of the recursive calculation of Fibonacci numbers. Figure 1 shows X10 and Fork-Join code for computing Fibonacci numbers, along with a graph of the recursive calls made when executing `fib(4)`. Calls to the non-recursive base case (`n < 2`) are shown as rectangles.



(a) Execution state

(b) Deque

Figure 2. State during evaluation of `fib(4)`. Execution is at the first `fib(0)` task. Dashed arrows indicate work to be done. Dotted boxes indicate completed work.



(a) Victim

(b) Thief

Figure 3. State for victim and thief after stealing the continuation of `fib(4)`.

2.2 Initiation

Initiation is concerned with ensuring that: 1) tasks are available to be stolen whenever appropriate, and 2) each task is only executed once. An idle thread may make itself useful by stealing work, so becoming a thief. This begins with the thief identifying a victim from which to steal. For example, the thief may randomly select a potential victim and if they appear to have work available, attempt a steal.

Tasks are typically managed by each worker using a double ended queue (deque), one deque per worker, as illustrated in Figure 2. Each worker pushes tasks onto the head of its deque using an unsynchronized store operation. Both the worker and any potential thieves then use atomic compare-and-swap (CAS) instructions to remove tasks from the worker's deque, with the worker acquiring from the head (newest), and thieves attempting to acquire from the tail (oldest). Tasks are thus made available and only stolen once, with the deque discipline minimizing contention and increasing the probability that long-running tasks are stolen.

2.3 State Management

When a task is stolen, the thief must: 1) acquire all state required to execute that task, and 2) provide an entrypoint to begin execution of the task, and 3) be able to return or combine return state with other tasks. Work-stealing implementations typically meet requirements 1) and 3) through the use of *state objects* that capture the required information about the task, and provide a location for data to be stored and shared across multiple tasks. Requirement 2) is handled differently depending on the execution model, and is discussed in more detail below for specific systems.

2.4 Termination

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

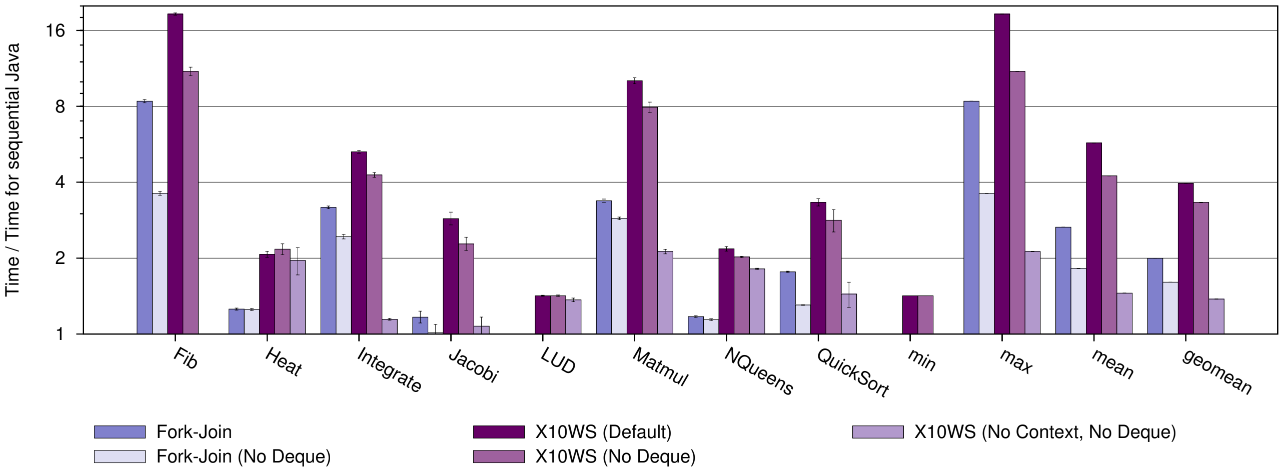


Figure 4. Sequential overheads for work-stealing runtimes on Jikes RVM.

2.5 Work-Stealing in X10

X10 is a strongly-typed, imperative, class-based, object-oriented programming language. X10 includes specific features to support parallel and distributed programming. A computation in X10 consists of one or more asynchronous activities (light-weight tasks). A new activity is created by the statement **`async`** `S`. To synchronize activities, X10 provides the statement **`finish`** `S`. Control will not return from within a finish until all activities spawned within the scope of the finish have terminated.

X10 restricts the use of a local mutable variable inside **`async`** statements. A mutable variable (`var`) can only be assigned to or read from within the **`async`** it was declared in. To mitigate this restriction, X10 permits the asynchronous initialization of final variables (**`val`**). A final variable may be initialized in a child **`async`** of the declaring **`async`**. A definite assignment analysis guarantees statically that only one such initialization will be executed on any code path, so there will never be two conflicting writes to the same variable.

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

Initiation. X10's work-stealing workers use deques as described above. Like Cilk, X10 adopts a *work-first* scheduling policy: when a worker encounters an **`async`** statement, it pushes the *continuation* of the current task to its deque and proceeds with the execution of the **`async`** body. For instance, a worker running `fib(4)` first executes `fib(3)` (Figure 1(b) line 8), making the `fib(2)` work item (line 9) available for others to steal. When done with **`async`** `fib(3)`, the worker attempts to pop the head deque item and if non-null, will execute the continuation (`fib(2)`, line 9).

State management. Each thread's stack is private, so in order to permit multiple workers to concurrently access and update the program state, the X10 compiler encapsulates sharable state into *frame objects*. Consequently, methods are rewritten to operate on fields of frame objects instead of local variables. Frame objects are linked together into trees that shadow the tree structure of the task graph. In other words, Figure 1(a) represents the tree of frame objects assembled during the execution of `fib(4)`. When X10 is compiled to Java, frame objects are created on the heap to ensure that they are accessible to both the worker and potential thieves. In the C++ implementation however, an optimization is performed that sees frame objects stack-allocated by default, and only lazily migrated to the heap when a steal occurs [19]. The X10 compiler analyzes the source code and indexes all of the points immediately after **`async`** statements ('reentry' points). It then generates a second copy of the source code in which methods take a *pc* (program counter) as an extra argument. The control flow of the generated methods is altered so as to permit starting execution at the specified pc.

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

2.6 Work-Stealing in Java Fork-Join

The general design of Fork-Join framework is a variant of the work-stealing framework devised by Cilk and is explained in detail in [12]. Here we briefly discuss its key components. The Fibonacci code for Fork-Join is shown Figure 1(c).

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

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

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

3. Motivating Analysis

As we noted earlier, although work-stealing is a very promising mechanism for exploiting software parallelism, it can bring with it formidable overheads to the simple sequential case. These overheads make the task of the programmer challenging because they must use work-stealing judiciously so as to extract maximum parallelism without incurring crippling sequential overheads. To shed light on this and further motivate our designs, we now: 1) identify and measure the *sequential overheads* imposed by existing work-stealing runtimes, and 2) measure the dynamic *steal ratio* across a range of parallel benchmarks, showing that unstolen tasks are by far the common case. We use eight well-known benchmarks expressed in X10, Fork-Join and sequential Java. We ported the code where necessary and have made the code publicly available (section 6). We discuss the details of our methodology in Section 6.

3.1 Sequential Overheads

In order to measure sequential overheads, we take work-stealing runtimes and execute each of them with a *single worker*. No stealing can occur in this case, so the runtime support for stealing is entirely redundant to this set of experiments. This artificial situation allows us to selectively *leave out* aspects of the runtime support, providing an opportunity to analyze the overheads due to work-stealing in more detail. As a baseline we use a straightforward (sequential) Java implementation of each benchmark.

We have identified three major sources of sequential overhead in existing work-stealing runtimes. Two are closely related to the overheads identified in the previous sections, namely initiation and state management. The final overheads relate to code restructuring and other changes necessary to support work-stealing.

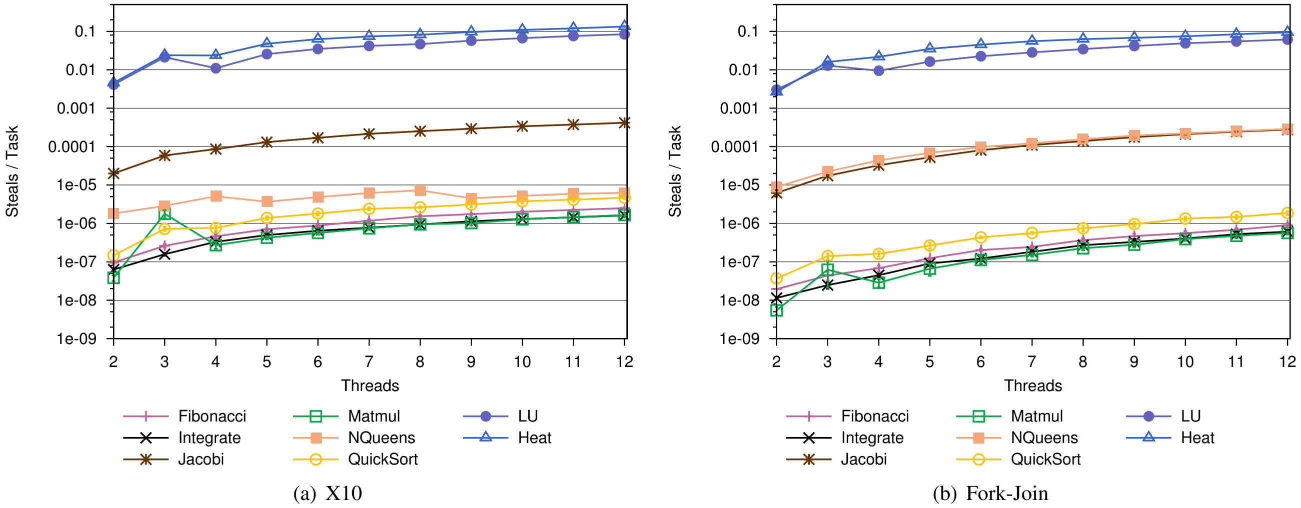
Initiation. The deque is an obvious source of overhead for the victim, which must use synchronized operations to perform work (even when nothing is stolen). This overhead may be a significant problem for programs with fine-grained concurrency. To measure this overhead, we took the X10 and Fork-Join work-stealing runtimes and measured single worker performance with all deque operations removed. (Recall that the deque manages pending work, so the strictly sequential case of a single worker, it is entirely redundant.) These results are shown as X10WS (No Deque) and Fork-Join (No Deque) in Figure 4. These results show that the deque accounts for just over 30% and 50% of all sequential overheads for X10WS (Default) and Fork-Join respectively.

State Management. As discussed in Section 2, work-stealing runtimes typically allocate state objects to allow sharing and movement of execution state between tasks. In pure Java, these objects must be heap allocated, leading to significant overheads. In addition to the direct cost of allocation and garbage collection, these objects may also be chained together, and may limit compiler optimizations. Figure 4 shows the overhead of allocating these state objects in the X10 Java work-stealing runtime by removing their allocation in a system that already had the deques removed. In this case all values are passed on the stack, and no copying was required because only a single worker exists. This is shown as X10WS (No Context, No Deque) in Figure 4. We did not need to perform a similar experiment for Fork-Join as it would reduce to the sequential Java case (and thus would show zero overhead). We can see that the allocation of these state objects is a very significant cost, averaging just under half of the total overhead.

Code Restructuring. In order to support the stealing of tasks, the runtime must generate entrypoints with which the thief can resume execution. This is typically performed by splitting up methods for the different **`finish`** and **`async`**s. This code restructuring accounts for part of the performance gap between X10WS (No Context, No Deque) and sequential Java. In effect, this overhead includes all overheads due to X10-to-Java compilation, of which only a subset would be necessary to support work-stealing.

3.2 Steal Ratio

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



(a) X10

(b) Fork-Join

Figure 5. Steals to task ratio

if, and how, cost should be traded off between *all* tasks and *stolen* tasks.

We performed a study to understand the steal ratios across a range of benchmarks. We instrumented each of the work-stealing runtimes to measure both the total number of tasks produced (executed **async** blocks in X10, and **fork**() calls in Fork-Join) as well as total number of tasks stolen. We show the measured ratio in Figure 5.

It is clear from the figure that stealing is generally uncommon and in many cases extremely rare. A single steal may move substantial work (consider divide-and-conquer algorithms). Because of this, relatively few steals may address a load imbalance. Although both LU and Heat workloads have steal ratios that approach one in ten, for many of the benchmarks the ratio is around one in a million. This result shows that steals are generally uncommon and suggest that eagerly preparing for a steal is likely to be an inefficient strategy.

4. Approach

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

Each of these operations is only required for tasks that are *actually* stolen, and as we saw in 3.2, steal ratios for many programs are close to zero. The performance of *unstolen* tasks is therefore critical to overall performance. In this work, we try to push the limits as to what aspects of the above functions can be deferred until a steal occurs, moving them off the critical path of unstolen tasks. Our particular approach is to leverage advanced facilities that exist within the implementation of a modern managed runtime.

4.1 Scalability Concerns

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

4.2 Techniques

Our approach is based on several basic techniques, each described in more detail in the context of the implementations discussed in Section 5.

1. We use the victim's execution stack as an *implicit* deque.
2. We modify the runtime to extract execution state directly from the victim's stack and registers.
3. We dynamically switch the victim to *slow* versions of code to reduce coordination overhead.

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

5. Implementation

We have implemented and evaluated two work-stealing runtimes, X10WS (OffStack): a modification of the default X10 work-stealing runtime for JVMs, and X10 (Try-Catch): a simple runtime implementation on plain Java. Both implementations support the X10 **`finish-async`** model of execution. Our plain Java X10 (Try-Catch) runtime is targeted by the X10 compiler.

This section describes each of our work stealing runtime implementations in terms of the work-stealing requirements we enumerated in Section 2: *initiation*, *state management*, and *termination*.

5.1 Runtime Supported X10WS (OffStack)

5.1.1 Initiation

One of the key insights behind the X10WS (OffStack) design is that we can avoid maintaining an explicit deque by using existing runtime mechanisms to extract the information from the worker's call stack. This approach eliminates the significant overhead of managing an explicit deque, but requires alternative mechanisms to synchronize the victim and thief, and to manage the set of stealable tasks.

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

Victim–Thief handshake. Once a thief finds a potential victim, it uses the runtime's yieldpoint mechanism to force the victim to yield—the victim is stopped while the steal is performed. The yieldpoint mechanism is used extensively within the runtime to support key features, including exact garbage collection, biased locking, and adaptive optimization. Reusing this mechanism allows us to add the hooks to stop the victim without any additional overhead. Note that between the point at which a thief attempts a steal, and the point the victim–thief handshake begins, it is possible that a task may no longer be available to steal. We measured the frequency of such failed steal attempts in our evaluation at around 5-10%.

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

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

5.1.2 State Management

When a task is stolen, the thief must take with it sufficient state to run the task within the thief's own context. This includes all local variables used by the stolen task. In our running example, it is just the parameter `n`, which is used on line 9 of Figure 1(b). In the X10WS (OffStack) system, we perform lazy state extraction, extracting the state from the victim thread only when a steal occurs.

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

The principal difference between our approach and the default X10 mechanism is that we perform state extraction *lazily*, only when an effective steal occurs. Compared to the X10 Java backend, our lazy approach avoids a large amount of heap allocation. The X10 C++ backend also has an approach which delays the allocation, but not the use of state objects, by initially allocating them on the stack and only lazily moving them to the heap when a steal occurs [19]. In the common case our approach avoids state extraction and allocation altogether.

Executing stolen tasks. Once state has been extracted, the thief executes against its heap-interned duplicate stack. The thief executes specially generated 'slow' versions of the continuations, which access the heap rather than the call stack for local variables. This is essentially identical to the default X10 implementation.

5.1.3 Termination

Control must only return from a **finish** context when all tasks within the context have terminated. To support this, each thread has a singly linked list with a node for each **finish** context that the thread is executing. Each dynamic **finish** context has a unique node shared by all threads running in that context. These nodes form a tree structure, with a root node for the **finish** context that represents the entire program. In X10WS (OffStack), four important pieces of information are saved at each finish node:

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

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

5.2 X10 (Try-Catch) Java implementation

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

5.2.1 Leveraging Exception Handling Support

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

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

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

Figure 6. Pseudocode for the transformation of `fib(n)` into X10 (Try-Catch).

Our X10 (Try-Catch) system annotates **async** and **finish** blocks by wrapping them with **try/catch** blocks with special work-stealing exceptions as shown in Figure 6. We can then leverage the exception handling support within the runtime and runtime compilers to generate exception table entries to support work stealing. These allow the X10 (Try-Catch) runtime to walk the stack and identify all **async** and **finish** contexts within which a thread is executing. The role of each exception type, and how the information is used in the runtime are described in more detail over the following sections.

5.2.2 Initiation

As in the X10WS (OffStack) implementation, X10 (Try-Catch) avoids maintaining an explicit deque. The key difference between the implementations is that in X10 (Try-Catch) we use marker **try/catch** blocks, not method annotations, to communicate the current deque state to the work-stealing runtime. Instead of identifying a continuation by a pair of methods using a frame pointer, we use a combination of the frame pointer and the offset of a specific **catch** block. We use the same thief-initiated handshake for synchronization between the victim and the thief.

Stack as an implicit deque. X10 (Try-Catch) maintains a *steal flag* for each worker thread that indicates that its deque may have work available to steal. The steal flag is set as the first action within an **async** (see line 6 in Figure 6). The steal flag is cleared when the worker or a thief determines that there is no more work to steal. As with X10WS (OffStack), the head of the task deque corresponds to the top of the call stack. The list of continuations (from newest to oldest) is established by walking the set of **catch** blocks that wrap the current execution state. We walk this list by running a modified version of the runtime's exception delivery code, searching for **catch** blocks that handle `WS.Continuation` exceptions. As we find entries, we simulate advancing into the **catch** block and repeat the search for exception handlers again, finding successively older continuations. Each worker has a `stealToken` that acts as a *tail* for the deque. The `stealToken` indicates the point at which the continuation of that worker has been stolen. Any continuations discovered after that point do not belong to that worker, and must therefore not be stolen from it.

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

5.2.3 State Management

In X10 (Try-Catch), state is acquired by the thief through duplicating the entire execution state of the victim thread, including the stack and register state. Unlike in X10WS (OffStack), the state is not extracted to the heap, but is used directly. No additional resume method is required for the entrypoint; execution can be transferred to the appropriate continuation point by delivering a `WS.Continuation` exception. The exception delivery code must be slightly different, because the exception must be delivered to the correct handler (it is not always the innermost exception handler for `WS.Continuation` that is correct).

Merging local variable state. While providing the correct state to *start* the continuation is made easy, X10 (Try-Catch) does not have a *resume* mode to fall back on where local variable state is all stored on the heap. This complicates merging the results of each of the tasks because the system must merge the local variables held by multiple copies of the same frame. In the Fibonacci example, the value of `a` is set within the **async**, while the value of `b` is set in the continuation. After the finish, both `a` and `b` must be set to ensure that the correct value is returned. At the end of each **async** or **finish** block there is a call to a runtime support method (`WS.join()` or `WS.finish()`). When these methods are called, two conditions are checked: 1) a steal has occurred within the appropriate finish block, and 2) the programmer has defined a **catch** block (`WS.Join` or `WS.FinishFirst` respectively) to save results. If both conditions are met, control is returned to the **catch** block by throwing an exception, allowing the code to provide local variable values with calls to `addFinishData(key, value)`. Each `key` represents a local variable: in our example `key` 0 maps to `a` and 1 maps to `b`. The last task to finish executing within the finish can then access all of these provided values, ensuring that all results are set correctly.

5.2.4 Termination

Termination is handled in a similar way to X10WS (OffStack). A node is lazily created for each **finish** context in which a task is stolen. This node maintains an atomic count of the number of active tasks in the **finish** context, and provides a location for local variable state to be passed between threads, as described in the previous section. When a thread decrements the atomic count to zero, it becomes responsible for running the continuation of the **finish** context. The X10 (Try-Catch) runtime will deliver a `WS.Finish` exception at the appropriate point, allowing the thread to extract local variable state and continue out from the **finish**. This may also update the thread's `stealToken`, if the last thread to finish execution was not the thread running the end of the **finish** body. When this occurs, that thread runs the body of any `WS.FinishFirst` handler to communicate local variables, *deposits* its `stealToken` in the finish node, and searches for other work to complete.

5.2.5 Optimizing Runtime Support Calls

Within the X10 (Try-Catch) runtime, there are many calls to various `WS` methods. In the common case where no steal has occurred, only the call to `WS.setFlag()` needs to be executed. The call to `WS.join()` is only required if the continuation for the enclosing **async** has been stolen. Similarly, the call to `WS.finish()` is only required if at least one steal has occurred within that **finish** context. To avoid sequential overhead in the common case, we generate special *fast* versions of methods with these calls, where compiled code for the calls to these methods are overwritten by `NOP` instructions. This makes it simple for us to transfer execution between these methods as required, without requiring any additional exception handling tables for the fast version of the code. Both fast and slow versions of the code always make calls to fast versions. We also force calls to `WS.setFlag()` to be inlined by the optimizing compiler, which on our primary Intel platform reduces to a simple store instruction.

Excluding indirect changes in compilation due to the presence of the **try/catch** blocks, the only sequential overhead in X10 (Try-Catch) is the execution of `WS.setFlag()`, and some additional `NOP` instructions.

6. Methodology

6.1 Benchmarks

Because the primary goal of our work is to reduce the sequential overheads of work-stealing, we have intentionally selected benchmarks with fairly fine-grained task structures. As the parallel tasks become coarser, the overheads of work-stealing become less significant for overall performance and the performance of all the approaches to work-stealing tend to converge. We have used a collection of eight benchmarks, which are briefly described below (they are available at `http://cs.anu.edu.au/~vivek/ws-oopsla-2012/`). For each case we ported the benchmark to native Java (for the sequential case), Java Fork-Join, X10 (using both the try-catch and default targets), and our plain Java try-catch system. For six of the eight benchmarks we manually generated the code to target the X10WS (OffStack) runtime (we did not implement automatic codegen support for X10WS (OffStack)). Having implementations of a common set of benchmarks allowed us to perform apples-to-apples comparisons of the different work-stealing systems. Unless specified below, each benchmark is run without any granularity parameter. The six benchmarks we have for all systems are:

Fibonacci A simple recursive computation of Fibonacci numbers. This benchmark is a commonly used micro-benchmark for task scheduling overhead, as the problem is embarrassingly parallel and the amount of computation done within each task is trivial. For our experiments we computed the 40th Fibonacci number.

Integrate Recursively calculate area under a curve for the polynomial function $x^3 + x$ in the range $0 <= x <= 10000$. This benchmark is similar in spirit to Fibonacci, but each task contains an order of magnitude more work.

Jacobi Iterative mesh relaxation with barriers: 100 steps of nearest neighbor averaging on 1024×1024 matrices of doubles (based on an algorithm taken from Fork-Join [12]).

Matmul Matrix multiplication of 1024×1024 matrices of doubles (based on an algorithm from Habanero Java [2]).

NQueens The classic N-queens problem where 12 queens are placed on a 12×12 board such that no piece could move to take another in a single move (based on an algorithm from Barcelona OpenMP Tasks Suite [6]).

Quicksort A recursive algorithm to quicksort a 100 million element array. It is very sensitive to memory bandwidth due to the consequences of having to move data between processors, and thus to aggregate memory bandwidth of the system as a whole.

The final two benchmarks are significantly more complicated and we did not perform the manual code translation required for X10WS (OffStack):

LU Decomposition Decomposition of 1024×1024 matrices of doubles (based on algorithm from Cilk-5.4.6 [16]). Block size of 16 is used to control the granularity.

Heat Diffusion Heat diffusion simulation across a mesh of size 4096×1024 (based on algorithm from Cilk-5.4.6). Leaf column size of 10 is the granularity parameter. Timestep used is 200.

6.2 Hardware Platform

All experiments were run on a machine with two Intel Xeon E7530 Nehalem processors. Each processor has six cores running at 1.87 GHz sharing a 12 MB L3 cache. The machine is configured with 16 GB of memory.

6.3 Software Platform

We modified both X10 and Jikes RVM, starting with the base versions described below.

Jikes RVM Version 3.1.2. We used the production build.

X10 (Try-Catch) and X10WS (Default) Based on X10 2.2.2.1, svn revision 23688.

Fork-Join Version 1.7.0.

X10WS (OffStack) Based on X10 2.1.2, svn revision 20276.

Cilk++ Intel's Cilk++ SDK preview (build 8503) [1]. We compile our benchmarks with optimization level *-O2* and used the *Miser* memory manager to avoid the lock contention and false sharing associated with C/C++ runtime memory management functions [15].

OpenJDK 64-Bit Server VM (build 20.0-b11, mixed mode).

6.4 Measurements

For each benchmark–configuration combination, we ran six invocations, with three iterations per invocation, where each iteration performed the kernel of the benchmark five times. We report the mean of the final iteration, along with a 95% confidence interval based on a Student t-test. For each invocation of the benchmark, the total number of garbage collector threads is kept the same as application threads. A heap size of 921 MB is used across all systems. Other than this, all VMs used in our experiments preserve their default settings.

Many of the benchmarks make extensive use of arrays. While the Fork-Join and sequential versions of the benchmarks use Java arrays directly, the X10 compiler is not currently able to optimize X10 array operations directly into Java array operations, but does so through a wrapper with get/set routines. To understand the significance of this overhead, we also measure a system that we call JavaWS (Try-Catch), which uses try-catch work-stealing but operates directly on Java arrays without X10.

7. Results

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

7.1 Sequential Overhead

Our primary focus for this paper is the reduction of sequential overheads as a means of improving overall throughput. Using the same methodology as in Section 3.1, we restrict the work-stealing runtimes so that they only use a single worker thread and then compare their performance to the purely sequential version of the program.

Figure 7 shows the sequential overhead of the original X10WS (Default), Fork-Join, our two optimized implementations, and the JavaWS (Try-Catch) system that uses regular Java arrays.

Sequential overheads for X10WS (Default) and Fork-Join are as high as 18× and 8.4× respectively (both for the Fibonacci benchmark). On average X10WS (OffStack) eliminates more than half of the sequential overheads of X10WS (Default) and performs slightly better than Fork-Join. The X10 (Try-Catch) implementation has consistently low sequential overheads across all benchmarks, including for Heat Diffusion and LU Decomposition, where the sequential overhead is already quite low on all the systems.

7.2 Work Stealing Performance

Figure 8 shows the speedup relative to sequential Java for each of the benchmarks and runtimes on our 12 core machine. Note that we did not measure LU Decomposition and Heat Diffusion for X10WS (OffStack). This is because we do not have automatic codegen support for X10WS (OffStack). These results clearly illustrate that the sequential overheads of work-stealing are the dominant factor in overall program performance. The results for the JavaWS (Try-Catch) runtime are extremely promising. Even in extreme examples of fine-grained concurrency like Fibonacci and Integrate it is able to outperform the sequential version of the program at 2 cores and deliver a significant speedup at 12 cores (7× and 8.5× respectively). Despite exhibiting excellent scalability, neither X10WS (Default) or Fork-Join are able to overcome their larger sequential overheads and show significant performance improvements over the sequential code for Fibonacci or Integrate even when using all 12 cores. The differences between the runtimes are less dramatic on the other five benchmarks, but the overall trend holds. All four runtimes show reasonable levels of scalability, but the lower sequential overheads of JavaWS (Try-Catch) and X10WS (OffStack) result in better overall performance.

In Figure 8 LU Decomposition and Heat Diffusion show unusual behavior. In LU Decomposition Fork-Join outperforms X10 (Try-Catch), and in Heat Diffusion all of the systems perform nearly identically. Figure 5(a) shows that LU Decomposition and Heat Diffusion are two benchmarks with high steal ratios (8% and 13% respectively at 12 cores). From Figure 2.4 we can also see that the LU Decomposition and Heat Diffusion benchmarks have almost zero sequential overhead, indicating that the total number of stealable tasks is low. This is not a situation where our approach will deliver significant gains, but we can see that JavaWS (Try-Catch) still performs nearly the same as all other systems.

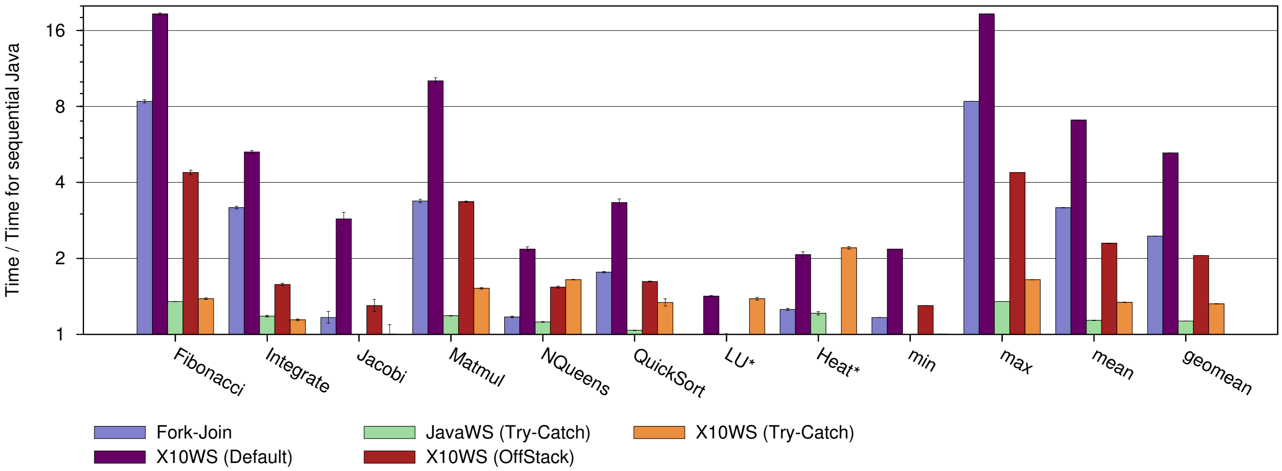
To increase the confidence in our results, we also compared overall performance with Cilk++, a C++ implementation of work stealing, and Fork-Join running on OpenJDK. Figure 9 shows the result of this experiment. In most cases, the running time for JavaWS (Try-Catch) is very competitive, particularly as the number of threads is increased. A notable exception is LU Decomposition where the JavaWS (Try-Catch) implementation is significantly slower. The Jikes RVM results in Figure 8(g) show that this slowdown affects all Jikes RVM configurations, not just JavaWS (Try-Catch), suggesting pathology in the underlying VM which is independent of our work stealing implementations. We are looking into this.

7.3 Memory Management Overheads

A significant source of performance improvement is due to the fact that X10WS (OffStack) dramatically reduces the number of heap-allocated frame objects, and X10 (Try-Catch) removes them altogether. Figure 10 shows the fraction of time spent performing garbage collection for each of the systems measured. As expected, X10WS (OffStack) and X10 (Try-Catch) have significantly lower memory management overheads than the other work-stealing runtimes. There is still measurable time spent in garbage collection for the NQueens and LU Decomposition benchmarks, but this is the case even for the sequential versions of these benchmarks. Across all programs the garbage collection fraction is less than 10%. Note that the garbage collection fraction does not include the potentially significant cost of object allocation during application execution. To ensure our performance improvements were not due to poor collector performance in Jikes RVM, we also measured the Java based systems on OpenJDK, and saw that the collection time fraction was similar, and we know from previous work [21] that the allocation performance of Jikes RVM is highly competitive.

7.4 Steal Ratios

To ensure that our modifications did not dramatically affect behavior, we also measured the steal ratios for our optimized systems. The results in Figure 11(c) for X10WS (OffStack) do not differ significantly to those for the original system in Figure 5(a). The steal ratio for X10 (Try-Catch) is between the steal ratios for the other two systems and is shown in



∗ We have no implementation of LU Decomposition or Heat Diffusion for X10WS (OffStack).

Figure 7. Overhead when running with a single thread (relative to sequential Java).

Figure 11(a). We also measured the frequency at which steal attempts failed (due to either another thief or the victim winning the race to start that continuation). Figure 11(b) and Figure 11(d) show the failed steal attempts for X10 (Try-Catch) and X10WS (OffStack) respectively. In general, the fraction of steal failures is less than 10%, only rising above this figure for a subset of benchmarks (Matmul, LU Decomposition, and Heat Diffusion) when running with a small number of threads.

We noticed that the steal ratio for Matmul with three threads is a persistent outlier for X10 (Try-Catch), X10WS (OffStack), and also Fork-Join and X10WS (Default) (Figures 5(a) and 5(b)). We are investigating this anomaly.

7.5 Summary

These results demonstrate that our approach is extremely effective at reducing sequential overheads, and does not do this at the cost of scalability. While the size of the benefit depends on the nature of the benchmark, in the cases where our approach does not provide a significant benefit, importantly it also does not negatively affect performance.

8. Related Work

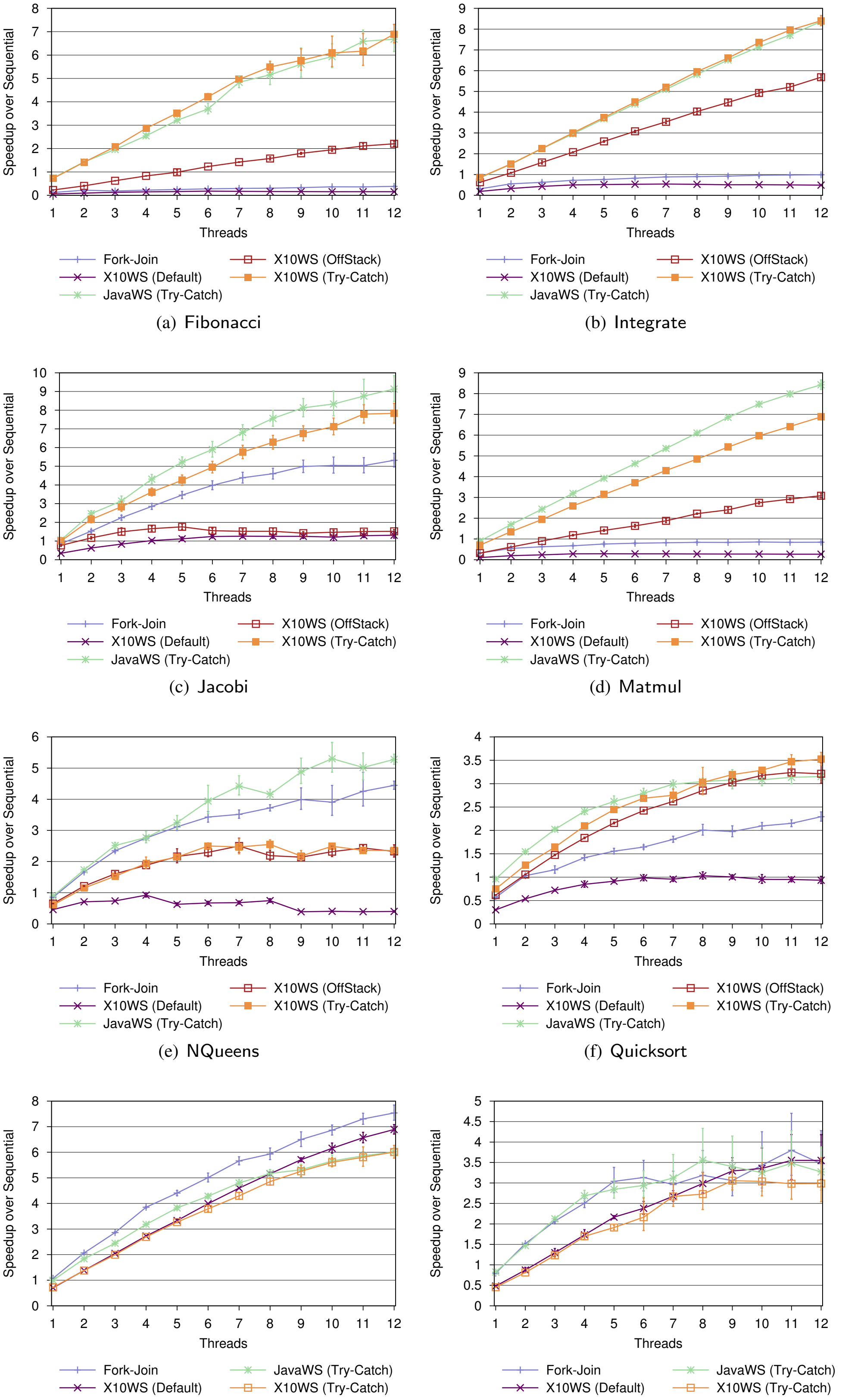
The ideas behind work-stealing have a long history which includes lazy task creation [17] and the MIT Cilk project [9], which offered both a theoretical and practical framework.

Languages versus Libraries Work-stealing has been made available to the programmer as libraries or as part of languages. Java's fork/join framework [12], Intel's Threading Building Blocks [18], PFunc [11], and Microsoft's Task Parallel Library [13] are all examples of libraries that implement work-stealing. Users write explicit calls to the library to parallelize their computation, as in Figure 1(c). X10, the Cilk-5 runtime [8] and the Habanero runtime [2] on the other hand are all examples of direct language support for work-stealing. In principle, a language supported work-stealing implementation has more opportunities for optimization because it can bind to and leverage internal runtime mechanisms that are not visible to a library implementation. Conversely, library implementations have the pragmatic advantage of being applicable to pre-existing languages.

Work-stealing Deques Several prior studies [4, 5, 14, 20] use cut-off strategies to control the recursion depth of function calls during the task generation. This is intended to reduce the overhead of task creation and deque operations. One of our contributions is to be able to use the worker's Java thread stack as an *implicit* deque and thus eliminate the need to employ cut-off strategies to control sequential overhead.

Cilk introduced the concept of *THE* protocol [8] to manage the deque. Actions by the worker on the head of the deque contribute to sequential overhead, while actions by the thieves on the tail of the deque contribute only to non-critical-path overhead. Almost all modern work-stealing schedulers follow this approach. We do not take this approach. Instead we observe that steals are infrequent and force the victim to yield when a steal occurs. Although this implies that the victim does some steal-related work, it only does so when a steal occurs. As long as the steal ratios are relatively modest the gain in overall system performance from our approach results in better scalability than the traditional approach.

Harnessing Rich Features of a Managed Runtime Managed runtimes provide many sophisticated features, which are not usually available in a low-level language implementation. A key runtime feature we use in our work is on-stack replacement [10], which is already employed in Jikes RVM for speculative optimizations and adaptive multi-level com-



(a) Fibonacci

(b) Integrate

(c) Jacobi

(d) Matmul

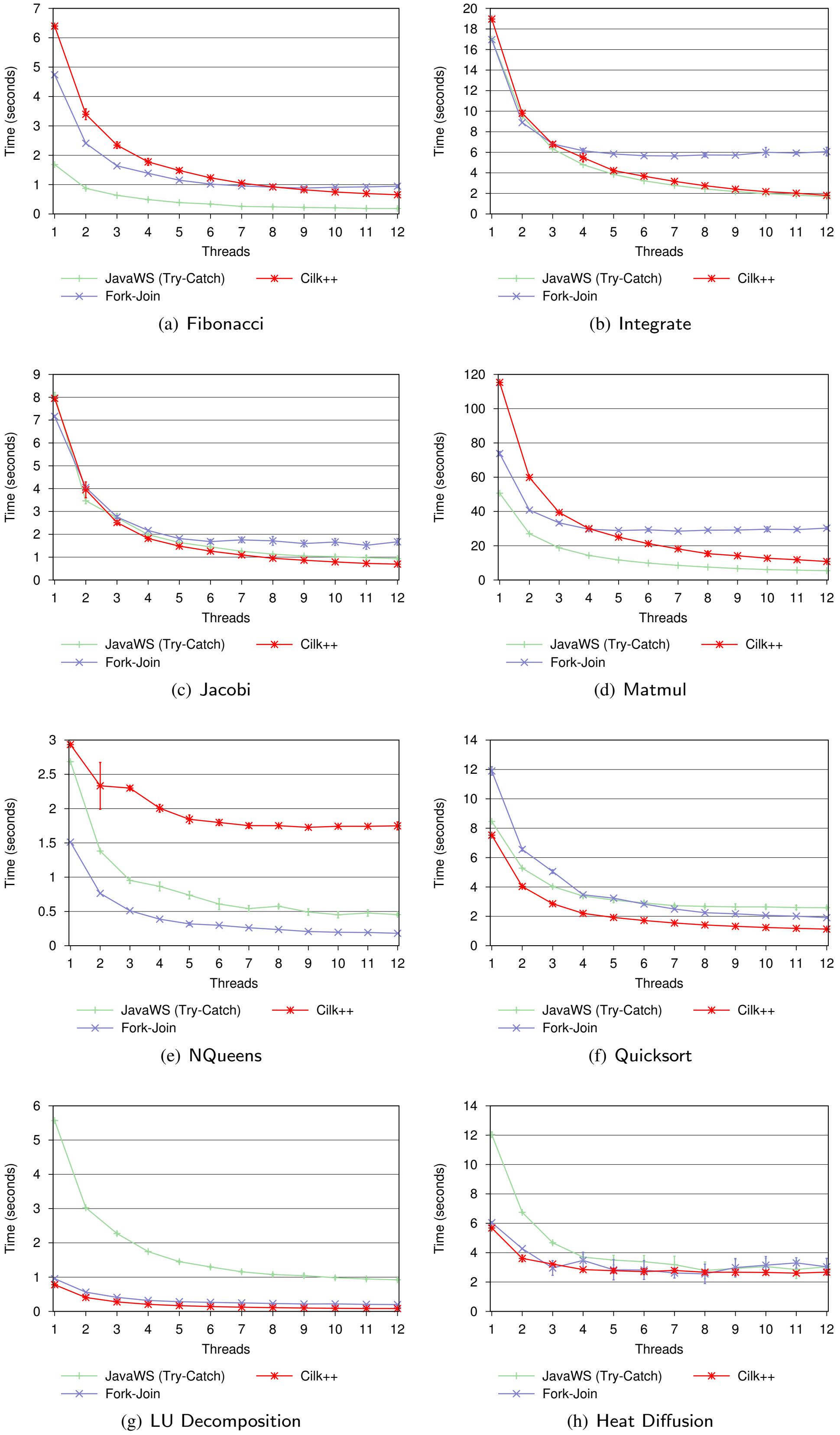
(e) NQueens

(f) Quicksort

(g) LU Decomposition

(h) Heat Diffusion

Figure 8. Speedup relative to sequential Java.



(a) Fibonacci

(b) Integrate

(c) Jacobi

(d) Matmul

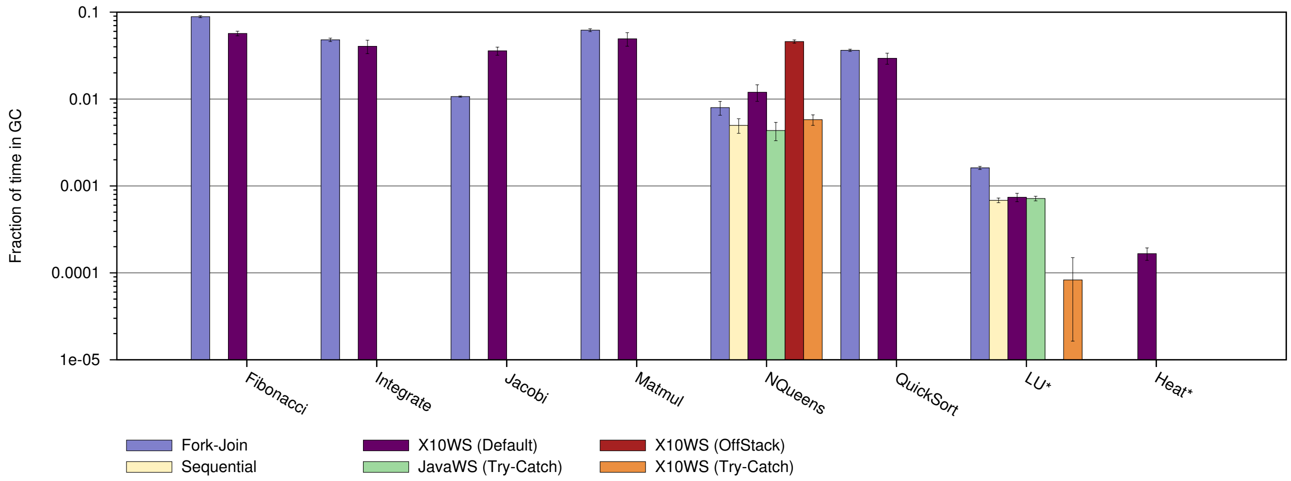
(e) NQueens

(f) Quicksort

(g) LU Decomposition

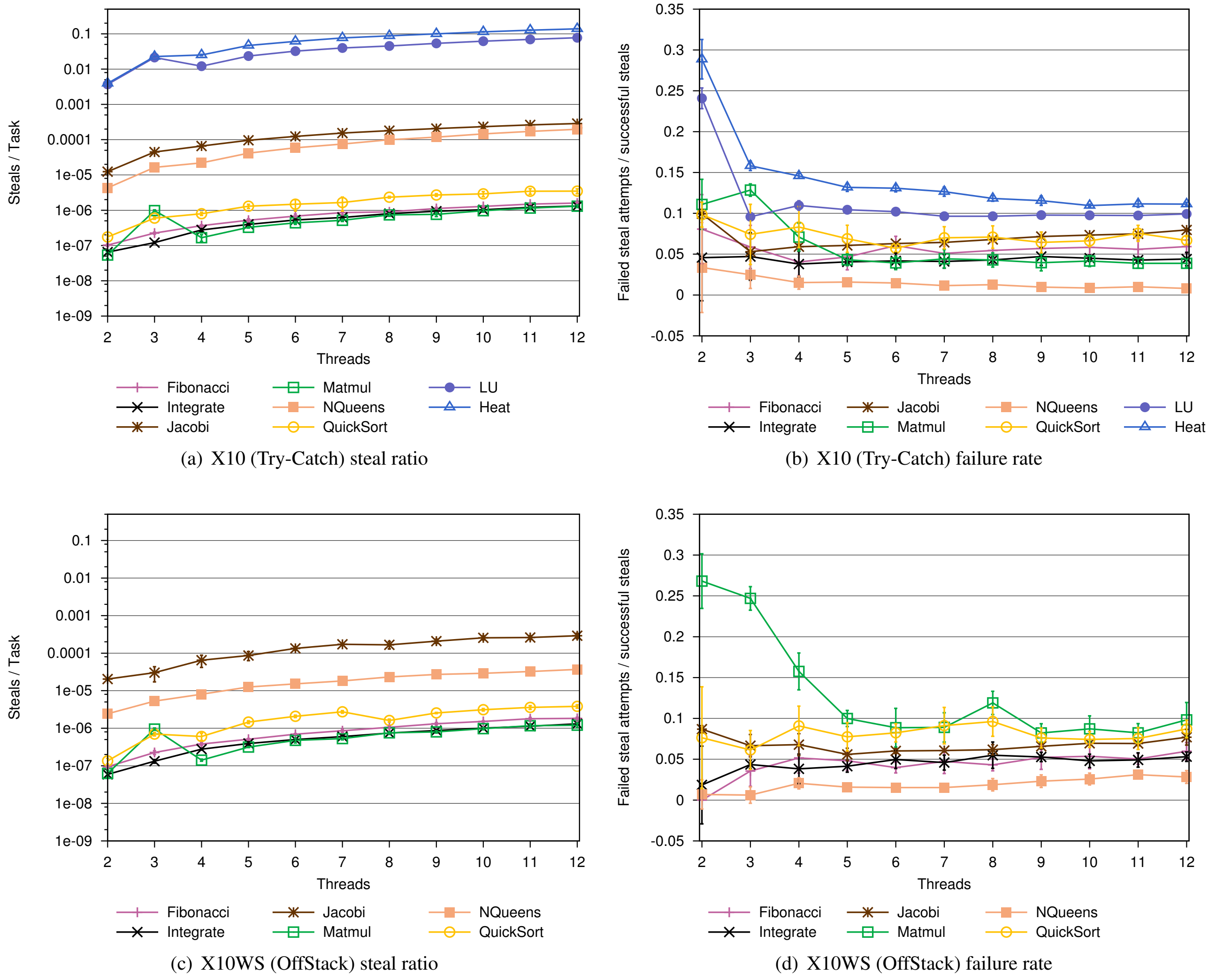
(h) Heat Diffusion

Figure 9. Running time



∗ We have no implementation of LU Decomposition or Heat Diffusion for X10WS (OffStack).

Figure 10. Fraction of time spent performing garbage collection work.



(a) X10 (Try-Catch) steal ratio

(b) X10 (Try-Catch) failure rate

(c) X10WS (OffStack) steal ratio

(d) X10WS (OffStack) failure rate

Figure 11. Steals to task ratio and steal failure rates for X10 (Try-Catch) and X10WS (Default).

pilation [7]. To support on-stack replacement, Jikes RVM's compilers generate machine code mapping information for selected program points that enable extraction of the Java-level program state from the machine registers and thread stack and the transfer of this state to newly created stack frames. We use these existing mechanisms inside Jikes RVM to walk a victim's Java thread stack and extract all the program state. The thief uses this to establish the necessary context for it to be able to execute stolen work.

The C++ implementation of X10 performs speculative stack allocation [19]. The victim starts by allocating the frames on a stack. The thief is responsible for copying the stolen frames from the victim's stack to the heap. This is not possible in the Java X10 implementation since Java does not support stack allocation. However we are able to leverage the runtime's stack walking mechanism to achieve an even simpler result—the thread state is not preprocessed. There are no frame objects on either the stack or the heap. Instead, by using the virtual machine's internal thread stack walking capability, we extract the state directly from the stack when a steal occur. Our approach radically lowers the memory management load of work-stealing. We are not aware of such functionality in any work-stealing scheduler.

9. Conclusion

We believe that work-stealing will be an increasingly important approach for effectively exploiting software parallelism on parallel hardware. In this work, we analyzed the sources of sequential overhead in work-stealing schedulers and designed and implemented two optimized work-stealing runtimes that significantly reduce them by building upon existing runtime services of modern JVMs. Our empirical results demonstrate that we can almost completely remove the sequential overhead from a work-stealing implementation and therefore obtain performance improvements over sequential code even at modest core counts.

We plan to continue exploring ways in which JVM runtime mechanisms can be adapted to further improve work-stealing. One avenue of exploration is to investigate techniques for reducing the stop time on the victim threads by applying ideas from concurrent garbage collection. For example, stack barriers could be employed to enable state extraction by the thief to happen mostly concurrently with victim execution. This should reduce stealing overheads and enable the system to tolerate much higher steal ratios without losing performance.

References

[1] Intel® Cilk™ Plus sdk. URL http://software.intel.com/en-us/articles/intel-cilk-plus.

[2] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093165.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852.

[4] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3374-2. doi: 10.1109/ICPP.2008.88.

[5] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79560-X, 978-3-540-79560-5. URL http://dl.acm.org/citation.cfm?id=1789826.1789838.

[6] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0. doi: 10.1109/ICPP.2009.64.

[7] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776288.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277725.

[9] M. Frigo, H. Prokop, M. Frigo, C. Leiserson, H. Prokop, S. Ramachandran, D. Dailey, C. Leiserson, I. Lyubashevskiy, N. Kushman, et al. The Cilk project. *Algorithms*, 1998.

[10] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143103.143114.

[11] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference*

on High Performance Computing Networking, Storage and Analysis, SC '09, pages 43:1–43:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654103.

[12] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465.

[13] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106.

[14] H.-W. Loidl and K. Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 International Conference on Functional Programming*, FP'95, pages 135–144, Swinton, UK, UK, 1995. British Computer Society. URL http://dl.acm.org/citation.cfm?id=2227330.2227343.

[15] J. Mellor-Crummey. Cilk++, parallel performance, and the cilk runtime system. URL http://www.clear.rice.edu/comp422/lecture-notes/comp422-2012-Lecture5-Cilk++.pdf.

[16] MIT. The Cilk project. URL http://supertech.csail.mit.edu/cilk/index.html.

[17] E. Mohr, D. Kranz, Halstead, and J. R.H. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical report, Cambridge, MA, USA, 1991.

[18] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.

[19] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145850.

[20] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An adaptive task creation strategy for work-stealing scheduling. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 266–277, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772992.

[21] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 307–324, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048092.