Faster Work Stealing With Return Barriers

Vivek Kumar
Australian National University
vivek.kumar@anu.edu.au

Stephen M. Blackburn
Australian National University
steve.blackburn@anu.edu.au

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. However, work-stealing comes with substantial overheads. Our prior work demonstrates that using the exception handling mechanism of modern VMs and gathering the runtime information directly from the victim's execution stack can significantly reduce these overheads.

In this paper we identify the overhead associated with managing the work-stealing related information on a victim's execution stack. A return barrier is a mechanism for intercepting the popping of a stack frame, and thus is a powerful tool for optimizing mechanisms that involve scanning of stack state. We present the design and preliminary findings of using return barriers on a victim's execution stack to reduce these overheads. We evaluate our design using classical work-stealing benchmarks. On these benchmarks, compared to our prior design, we are able to reduce the overheads by as much as 58%. These preliminary findings give further hope to an already promising technique of harnessing rich features of a modern VM inside a work-stealing scheduler.

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

General Terms Design, Performance.

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

1. Introduction

Work-stealing [4, 6, 9, 13] is a widely used 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 various programming languages, such as Cilk [6] and X10 [4], and in application frameworks, such as the Java fork/join framework [9] and Intel Threading Building Blocks [13].

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. In our prior work [8] we analyzed the sources of sequential overhead in work-stealing schedulers and designed and implemented two optimized work-stealing runtimes that significantly reduce overheads by building upon existing runtime services of modern JVMs. Our 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.

The techniques that we use in our prior work are:

1. Using the victim's execution stack as an *implicit* deque.
2. Modifying the runtime to extract execution state directly from the victim's stack and registers.
3. Dynamically switching the victim to *slow* versions of code to reduce coordination overhead.

Once a thief finds a potential victim, it exploits the runtime's existing yieldpoint mechanism to force the victim to yield. The victim is stopped while the steal is performed. However, when steals are frequent, forcing the victim to yield at each steal becomes costly.

The contributions of this paper are: 1) a detailed study of the cost associated with stealing from a victim in our prior work; 2) a detailed design for reducing this overhead and 3) evaluation of our new design using classical work-stealing benchmarks.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 provides the relevant background. Section 4 discusses our evaluation methodology. Section 5 discusses the motivation for this work. Section 6 explains the design of our new system. Section 7 discusses the performance evaluation of our new design and finally section 8 concludes the paper.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Related Work

The ideas behind work-stealing have a long history which includes lazy task creation [12] and the MIT Cilk project [6], which offered both a theoretical and practical framework. In [8], we exploit rich features of modern virtual machines and build a new work-stealing framework for X10 language's finish-async programming model [16]. We steal only one task at a time, just as in Java's fork/join framework [9].

Though stealing one task at a time has been shown to be sufficient to optimize computation along the 'critical path' to within a constant factor [1, 3], several authors have argued that the scheme can be improved by allowing multiple tasks to be stolen at a time [2, 5, 7, 10, 14]. In this work, we explore a different approach to minimizing steal overheads. We exploit the return barrier mechanism to optimize the steal process. The return barrier mechanism is not a new, it is used in various garbage collectors [15, 17], however, to our knowledge, it has not been applied to work-stealing until now.

3. Background

This section provides a brief overview of work-stealing runtimes in the context of the X10 (Try-Catch) implementation from our prior work [8].

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.

Our try-catch work-stealing framework is currently designed for the X10 **finish**-**async** style programming model. In the section below, we will discuss this model briefly.

3.1 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.

```
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;
}
```

Figure 1. X10's **finish**-**async** style programming model

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. Figure 1 shows X10's **finish**-**async** style programming model.

We have modified the X10 compiler to compile to X10 (Try-Catch) and hence, X10 (Try-Catch) represents a new backend for work-stealing.

3.2 X10 (Try-Catch) Java implementation

3.2.1 Leveraging Exception Handling Support

Most JVMs, including Jikes RVM, have very efficient support for exception handling. Because exceptions are important and potentially expensive, JVM implementers have invested heavily in optimizing the mechanisms. We leveraged these optimized mechanisms to efficiently implement the peculiar control flow requirements of work-stealing. The X10 (Try-Catch) system annotates **async** and **finish** blocks by wrapping them with **try**/**catch** blocks with special work-stealing exceptions. 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 work-stealing implementation consists of following basic phases, each of which require special support from the runtime or library:

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).
3. Termination. (Join tasks and ensure correct termination).

3.2.2 Initiation

X10 (Try-Catch) avoids maintaining an explicit deque for workers. Instead, marker try/catch blocks are used to communicate the current deque state to the work-stealing runtime. The execution stack of a thread is used as an implicit deque.

A thief identifies its potential victim by checking the *steal flag* maintained by each worker thread. The steal flag is set as the first action within an **`async`**. This flag is cleared when the worker or a thief determines that there is no more work to steal. 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. Only one thief is allow to steal from one victim at any given time. Different thieves can steal from different victims concurrently.

The head of the task deque corresponds to the top of the execution stack. The list of continuations (from newest to oldest) is established by walking the set of **`catch`** blocks that wrap the current execution state. Each worker has a `stealToken` that acts as a *tail* for the deque. None of the continuations found after this point is stolen. This `stealToken` also helps in guaranteeing atomicity. It acts as a *roadblock* for the worker and thieves to prevent either running or stealing continuations past that point.

3.2.3 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. This aspect of our X10 (Try-Catch) implementation is discussed in more detail below.

3.2.4 Termination

Control must only return from a **`finish`** context when all tasks within the context have terminated. To support this, a singly linked list is maintained. 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 the partial results to be passed between threads. 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 special exception at the appropriate point, allowing the thread to extract partial results and continue out from the **`finish`**.

3.3 Return Barriers

A return barrier, like a write barrier, allows the runtime to intercept a common event, and (conditionally) interpose special semantics. In the case of a write barrier, a runtime typically interposes itself on pointer field updates, conditionally remembering updates of pointers in certain conditions. On the other hand, a return barrier [15, 17], interposes special semantics upon the return from a method (which corresponds to the popping of a stack frame). One use for a return barrier is to keep track of a 'low water mark' for each stack since some particular event, such as the last garbage collection. In a language where pointers into the stack are not permitted, there is a guarantee that no part of the stack below the low water mark has been changed since the low water mark was set. This information can be used to reduce the overhead of stack scanning. In our work, we use a return barrier to 'protect' the victim from stumbling upon an active thief. We do this by installing a return barrier above the stealable frames, allowing the victim to ignore all steal activty that occurs below the low water mark. Only when the frame above the return barrier is popped does the victim need to consider the activity of thief.

A naive implementation of a return barrier would require some (modest) code to be executed upon every return, just as a write barrier is typically executed upon every pointer update. In our implementation we insert a trampoline that hijacks the return of a particular frame (the return is redirected to our trampoline). The trampoline executes the return barrier semantics (which may include re-installing the return barrier at a lower frame), before returning to the correct frame (whose address was remembered in a side data structure). This barrier has absolutely no overhead in the common case, and only incurs a modest cost when the frame targeted by the return barrier is popped.

We now motivate the work presented in the remainder of the paper with an analysis of the cost of stealing in a well-tuned work-stealing runtime.

4. Methodology

4.1 Benchmarks

Because the primary goal of our work is to reduce the cost of steal operations, we have intentionally selected benchmarks with high steal rates We have used a collection of three benchmarks, which are briefly described below. In each case we ported the benchmark to plain Java (for the sequential case), as well as to our JavaWS (Try-Catch) system, described below.

The three benchmarks we have are:

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 [9]).

LUD Decomposition of 1024 × 1024 matrices of doubles (based on algorithm from Cilk-5.4.6 [11]). 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.

4.2 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 was configured with 16 GB of memory.

4.3 Software Platform

Jikes RVM Version 3.1.2. We used the production build.

4.4 Measurements

For each benchmark, 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 experiments. Other than this, we preserve the default settings of Jikes RVM.

All of our benchmarks make extensive use of arrays. The sequential versions of the benchmarks use Java arrays directly. However, 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 avoid the overhead associated with this indirect array accesses, we use a system that we call JavaWS (Try-Catch), which uses try-catch work-stealing but operates directly on Java arrays without X10.

5. Motivating Analysis

As we noted in our prior work, although work-stealing is a very promising mechanism for exploiting software parallelism, it can bring with it formidable overheads to the simple sequential case. In our prior work, we exploited rich features that pre-exist within the JVM implementation to significantly reduce these overheads. We heavily rely on the yieldpoint mechanism of the JVM to stop the victim so that the thieves can extract the required information directly from the execution stack. However, when the steals become frequent, stopping the victim inside a yieldpoint at each steal may amount to a significant overhead. In our prior work, we performed a study to understand steal ratios across a range of benchmarks. That analysis shows that steals are generally infrequent, ranging from 1/10 to $1/10^5$ steals/task (see Figure 6(a)).

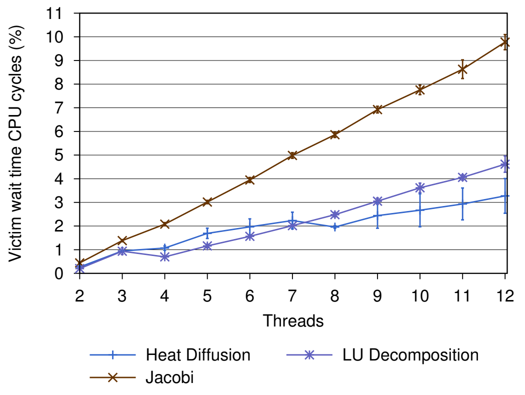


Figure 2. Percentage of CPU cycles lost by victims waiting for the steals to finish in default JavaWS (Try-Catch).

To further motivate our designs, we now measure: 1) the steal rate (steals/sec); 2) the overhead imposed by the steal mechanism upon the victims.

5.1 Steal Rate

The steal ratio is only one dimension of the steal overheads. We now measure the steal rate (steals/sec), which is shown in Figure 7(a). Steal rate is calculated by dividing the total number of steals by the benchmark execution time. This gives an indication of how frequently we are forcing the victim to execute the yieldpoint. The results obtained for the Jacobi in both these studies clearly indicates that a benchmark with a low steal *ratio* may have a high steal *rate*.

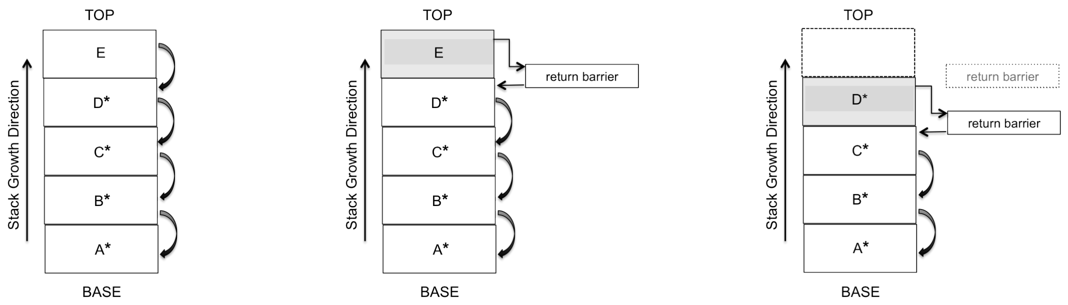
5.2 Stealing Overhead

Next we measured the cost of a steal as imposed upon the victim by the thief. We did this by measuring the percentage of CPU cycles lost by the victim while waiting for the thief to finish accessing its execution stack. We measure the CPU cycles required for the entire execution of the benchmark using hardware performance counters. We used Jikes RVM's existing mechanisms for measuring the number of cycles spent waiting for the thief to access the victim's stack. We start a timer when a victim is forced to execute yieldpoint. The timer is stopped when the thief finishes accessing the victim's execution stack and unlocks the victim from the yieldpoint. These cycles are summed for all the steals over the benchmark execution. At the end of execution we get the total CPU cycles lost to the thief. These are cycles which the victim could have utilized for carrying out its execution, had it not been forced to yield to the thief. We calculate this percentage of lost cycles and plot in Figure 2.

These numbers show that the steal overhead is as much as 10% for Jacobi.

6. Implementation

This section discusses the modifications made inside the JavaWS (Try-Catch) runtime [8] to use return barriers for



(a) Initial state of stack. (b) Thief installs the return barrier on victim's execution stack. (c) Thief installs the return barrier on victim's execution stack.

Figure 3. The victim's stack, installation, and movement of the return barrier.

assisting the steal process from a victim. We use the JavaWS (Try-Catch) runtime because it is the best performing work-stealing runtime.

6.1 Installing the First Return Barrier.

Figure 3(a) depicts a typical snapshot of a victim's execution stack. The stack frames having a stealable continuation are marked with a * in this figure. The newly executed methods occupy the stack frame slots on the top of execution stack. Each stack frame is recognized with the help of a frame pointer. The value stored inside this pointer is the frame pointer of the last executed method. The other important information, which is of interest to us, is the return address, which normally forms part of a stack frame. It holds the address where the control should be transferred after unwinding to the caller frame.

Once a thief has decided to rob this victim and discovers that the victim does not yet have a return barrier installed, the thief halts the victim by forcing it to execute the yieldpoint mechanism. After the victim has stopped, the thief starts scanning the stack frames to identify the oldest stealable continuation. In this case it is the frame *A*. However, before the thief unwinds down to the frame pointer for *A*, it notices that the first (newest) available continuation is *D*. It installs a return barrier to intercept the return from method *E*. The return address stored inside *E* is modified and this new address is now that of the return barrier trampoline method. Thus upon returning from *E*, the victim will find itself inside the return barrier trampoline. The trampoline maintains the address of frame *D*, and will return execution to *D* one the trampoline has been executed. The return address from the barrier is now the old value from frame *E*. Figure 3(b) depicts the victim's modified execution stack.

The victim holds a private boolean field *stealInProgress*, which is now marked as *true* by the thief. The thief then creates a clone of the entire stack of the victim and only then allows its victim to continue. The victim continues the rest of its computation, oblivious to the activity of the thief, while the thief proceeds with the steal process using the cloned stack. The cloned stack is also used for offline manipulation of the callee save registers.

6.2 Synchronization Between Thief and Victim During Steal Process.

When the victim finishing executing method *E*, it will return via the trampoline method of the return barrier. Before unwinding to *D*, the return barrier code checks the *stealInProgress* flag to determine whether a steal is in progress and whether the continuation being stolen is *D*. If *D* is not being stolen, the victim will reinstall its return barrier at the next available continuation after *D*. This is the method *C* in this case. The victim then returns to *D* and continues computation until it hits the return barrier again, at which point it repeats the process. Figure 3(c) shows this newly modified stack frame of the victim.

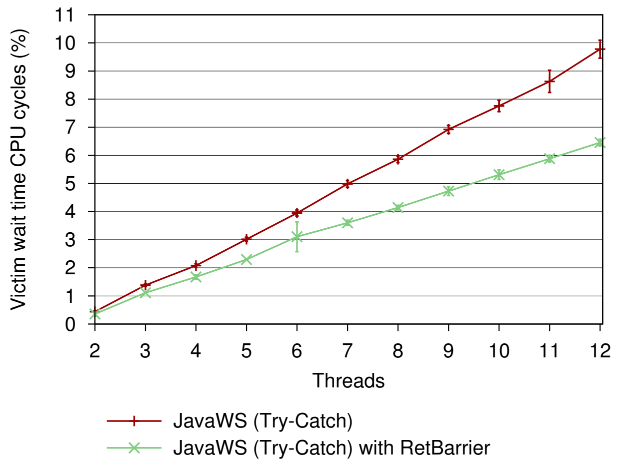
In the case where the victim discovers that the frame below the return barrier is being currently being stolen, it will wait on a condition variable. Once the steal is complete, the thief resets the victim's field *stealInProgress* to false and signals the victim. The victim now unwinds to the stolen frame and becomes a thief.

6.3 Stealing From a Victim with Return Barrier Pre-installed.

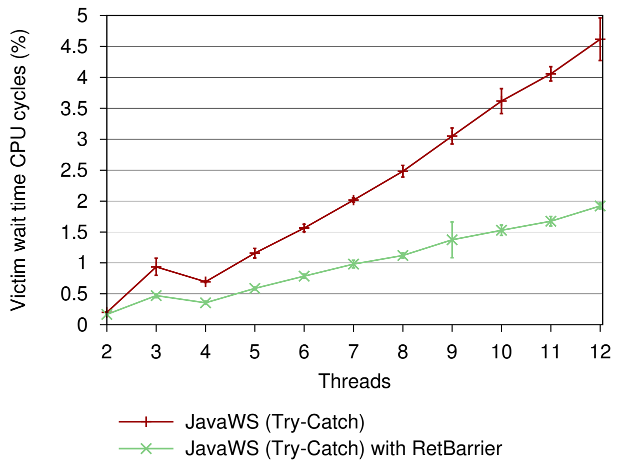
If, upon identifying a potential victim, the thief finds that there is already a return barrier installed on the victim's execution stack, it does not force the victim to execute the yieldpoint, but marks the victim's field *stealInProgress* as *true*. The steal and the victim's computation then happen concurrently, with the victim oblivious to the steal. The previously cloned stack of the victim is reused for the steal process and offline manipulation of the callee save registers.

7. Performance Evaluation

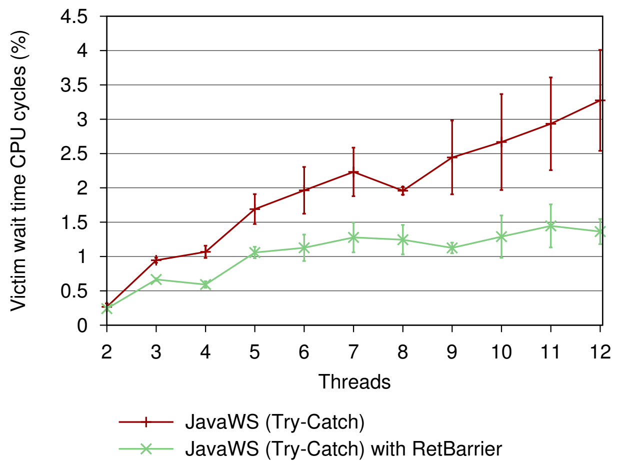
We start with measuring the percentage of CPU cycles lost by victims while waiting for the steal to finish. We then measure the speedup on both the modified and unmodified



(a) Jacobi

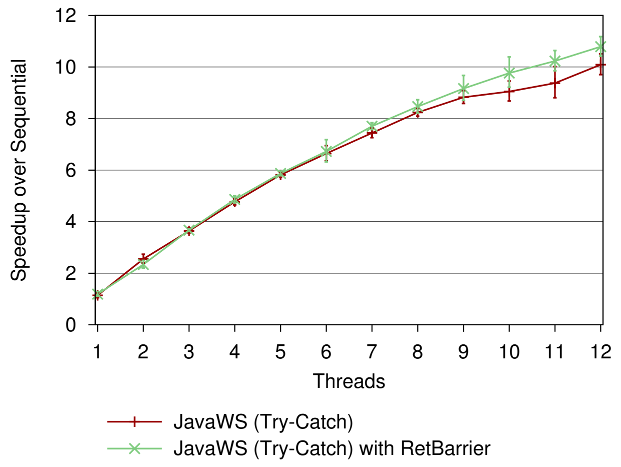


(b) LUD

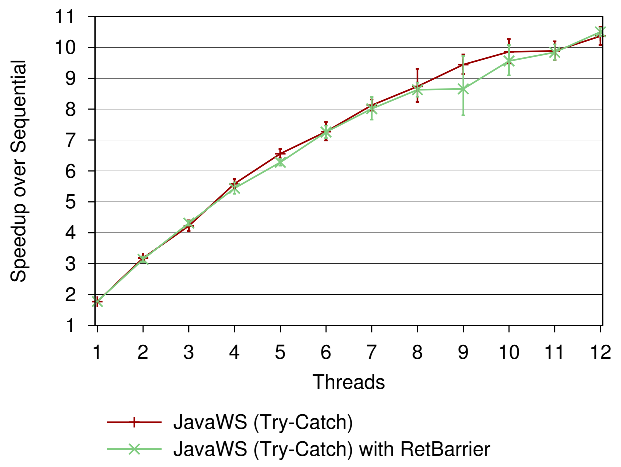


(c) Heat Diffusion

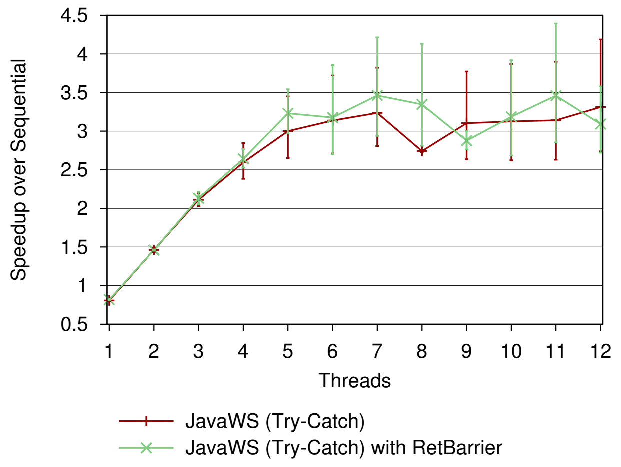
Figure 4. Percentage of CPU cycles lost by victims waiting for the steals to finish.



(a) Jacobi



(b) LUD



(c) Heat Diffusion

Figure 5. Speedup relative to sequential Java.

JavaWS (Try-Catch). We finish by examining the effect of the return barrier on the steal ratio and steal rate.

7.1 Cost of Stealing

We measure the percentage of CPU cycles lost by the victims while waiting for the thieves to gather the required information from its execution stack. Figure 4 shows this percentage for both the modified and unmodified systems. These results illustrate that using return barrier on a victim's execution stack can reduce the stealing overhead significantly, and by as much as 58%.

7.2 Work-Stealing Performance

Figure 5 shows the scalability of the benchmarks both on the default JavaWS (Try-Catch) and JavaWS (Try-Catch) with return barriers. The results show that scalability is not statistically significantly affected by the use of return barriers.

7.3 Steal Ratio and Steal Rate

To ensure that our design did not change the steal ratio and the steal rate, we measured them on our new design. The results are the figure 6 and figure 7. The results exactly match with the results on the unmodified system.

7.4 Summary

These results demonstrate that our approach is extremely effective at reducing the overhead associated with managing the work-stealing related information on a victim's execution stack. However, we do not notice increased speedup or increased steal rate even though our new design almost halved the stealing overhead. This is because the actual overhead is not large and hence there is no noticeable increase in performance of the system. However, for the cases where the cost of steals is significant, our new design will be helpful.

8. Conclusion

We believe that work-stealing will be an increasingly important approach for effectively exploiting software parallelism on parallel hardware. As an extension to our prior work, here we analyzed the overhead associated with stealing from the victim's execution stack. We designed a return barrier-based victim execution stack and evaluated it using a set of classical work-stealing benchmarks. Our preliminary results demonstrate that we can significantly reduce the overhead of the stealing process.

As future work, we plan to evaluate the steal-*N* strategy, which steals more than one frame at a time, and integrate it with our current work. We also plan to continue exploring ways in which JVM runtime mechanisms can be adapted to further improve work-stealing.

References

[1] N. Arora, R. Bolumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM, 1998.

[2] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 178–187. IEEE, 2001.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46 (5):720–748, 1999. ISSN 0004-5411.

[4] 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.

[5] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[6] 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.

[7] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289. ACM, 2002.

[8] V. Kumar, D. Frampton, S. Blackburn, D. Grove, and O. Tardieu. Work–stealing without the baggage. In *Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, Tucson, Arizona, USA, Oct. 2012. ACM.

[9] 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.

[10] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 164–172. ACM, 1993.

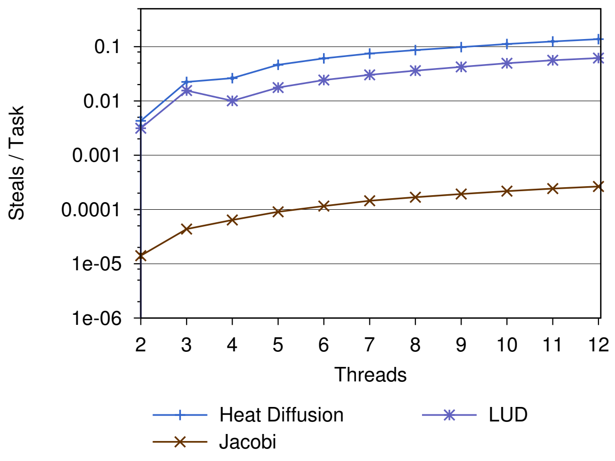
[11] MIT. The Cilk project. URL `http://supertech.csail.mit.edu/cilk/index.html`.

[12] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(3): 264–280, 1991.

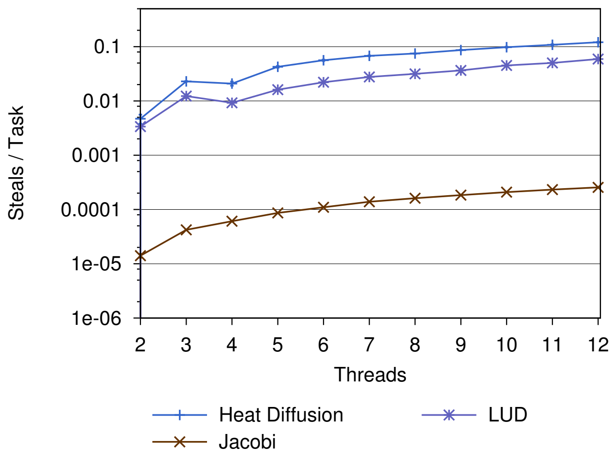
[13] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[14] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245. ACM, 1991.

[15] H. Saiki, Y. Konaka, T. Komiya, M. Yasugi, and T. Yuasa. Real-time gc in jerty vm using the return-barrier method. In *Object-Oriented Real-Time Distributed Computing, 2005.*

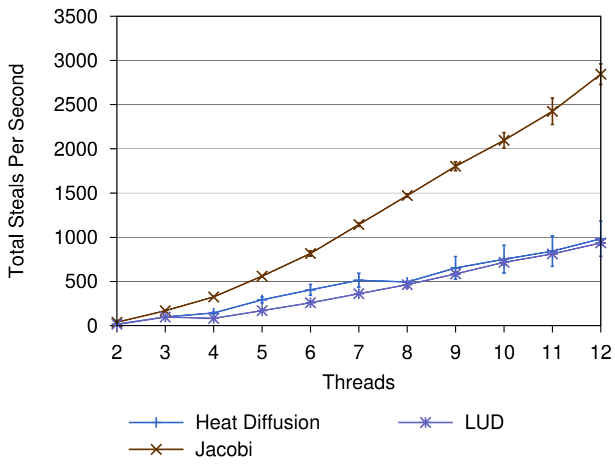


(a) Steal ratio on default JavaWS (Try-Catch)

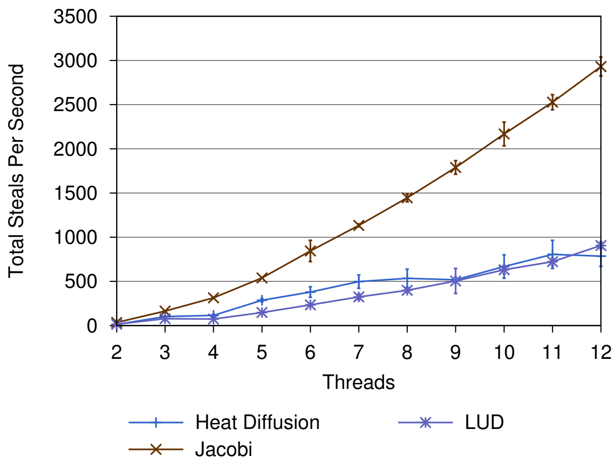


(b) Steal ratio on JavaWS (Try-Catch) with return barrier

Figure 6. Steal ratio comparison with our new implementation.



(a) Steal rate on default JavaWS (Try-Catch)



(b) Steal rate on JavaWS (Try-Catch) with return barrier

Figure 7. Steal rate comparison with our new implementation.

ISORC 2005. Eighth IEEE International Symposium on, pages 140–148. IEEE, 2005.

[16] 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.

[17] T. Yuasa. Real–time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.