Friendly Barriers: Efficient Work-Stealing With Return Barriers *

Vivek Kumar†, Stephen M. Blackburn†, David Grove‡

†Australian National University ‡IBM T.J. Watson Research

Abstract

This paper addresses the problem of efficiently supporting parallelism within a managed runtime. A popular approach for exploiting software parallelism on parallel hardware is task parallelism, where the programmer explicitly identifies potential parallelism and the runtime then schedules the work. Work-stealing is a promising scheduling strategy that a runtime may use to keep otherwise idle hardware busy while relieving overloaded hardware of its burden. However, work-stealing comes with substantial overheads. Recent work identified *sequential* overheads of work-stealing, those that occur even when no stealing takes place, as a significant source of overhead. That work was able to reduce sequential overheads to just 15% [21].

In this work, we turn to *dynamic* overheads, those that occur each time a steal takes place. We show that the dynamic overhead is dominated by introspection of the victim's stack when a steal takes place. We exploit the idea of a low overhead return barrier to reduce the dynamic overhead by approximately *half*, resulting in total performance improvements of as much as 20%. Because, unlike prior work, we attack the overheads directly due to stealing and therefore attack the overheads that grow as parallelism grows, we improve the scalability of work-stealing applications. This result is complementary to recent work addressing the sequential overheads of work-stealing. This work therefore substantially relieves work-stealing of the increasing pressure due to increasing intra-node hardware parallelism.

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, Languages, Performance.

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

* 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.

VEE '14 March 01 - 02 2014, Salt Lake City, UT, USA

http://dx.doi.org/10.1145/2576195.2576207

1. Introduction

This paper is concerned with the efficient support for dynamic task parallelism within managed runtimes. Parallelism is a critical concern as improvements in on-chip performance are now delivered through hardware parallelism rather than clock scaling — single nodes can now scale to over one hundred cores.

Dynamic task parallelism is a popular strategy for exposing software parallelism to the underlying hardware. The programmer exposes the parallelism and the problem of scheduling that work is delegated to a supporting library or runtime. Work-stealing has emerged as a popular strategy for scheduling task parallel work [8, 12, 22, 28]. However, there are significant overheads associated with work-stealing, both *sequential* ones, that manifest whether or not stealing takes place, and *dynamic* ones, that manifest when stealing occurs. Our paper addresses dynamic work-stealing overheads, with the goal of improving the efficiency and scalability of intra-node parallelism.

Work-stealing has a long history which includes lazy task creation [26] and the MIT Cilk project [12], which offered both a theoretical and practical framework. It has also been adopted by more recent languages including X10 [8], which we use as the context for the work we present here. X10 is designed to ease programming of scalable concurrent and distributed systems, with explicit constructs for parallelism and data distribution. X10 uses a *finish/async* idiom to capture software parallelism, and its runtimes use work-stealing to efficiently schedule the work. Although the work we present here is evaluated in the context of X10, work-stealing has much broader application, and we hope that our insights will be applicable beyond this specific context.

Kumar et al. demonstrate that work-stealing overheads can be as high as 4×. They evaluate work-stealing overheads in X10 and attack the problem of *sequential* overheads; those that manifest independent of the level of actual parallelism. They reduced the sequential overheads due to work-stealing in X10 from around 4× to 15%. They achieved this through three principal means: a) using the victim's execution stack as an *implicit* deque; b) modifying the runtime to extract execution state directly from the victim's stack and registers; and c) using the exception handling mechanism to dynamically switch to different versions of code between the thief and victim.

In this work, we attack the *dynamic* overheads of work-stealing; those that manifest as steal rates grow, and are thus most evident when parallelism is greatest. As core counts increase, dynamic overheads are an increasingly important factor in the performance of work-stealing runtimes. We identify walking the victim's execution stack at every steal as the major dynamic cost. We address this problem by using a *return barrier* [36] to reduce the time spent scanning the stack. This reduces dynamic overheads by around 50%, leading to total performance improvements of up to 20%.

The system we improve upon is already high performance. In Section 6.5 we compare our system directly against the Fork-Join and Habanero-Java frameworks, two other widely used frameworks that use work-stealing, evaluating all three against a straightforward sequential Java baseline. We show that our system is highly competitive. It consistently performs well, and in three out of six workloads it substantially outperforms the other systems (from 50% to 6× better).

The principal contributions of this paper are as follows: a) a detailed study of the *dynamic* costs of work-stealing — costs associated with stealing work from victims; b) an approach for reducing this overhead and c) evaluation of our new design using classical work-stealing benchmarks.

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

2. Background

This section provides a brief overview of key background material, including return barriers, work-stealing and X10.

2.1 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 [36], 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 a thief introspecting the victim's stack.

```
1 foo() {
2   val X:Int;
3   val Y:Int;
4   finish {
5     async X = S1();
6     Y = S2();
7   }
8 }
```

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

2.2 Work-stealing

Work-stealing is a strategy for efficiently distributing work in a parallel system. The runtime maintains a pool of *worker threads*, each of which maintains a set of *tasks*. When local work runs out, the worker becomes a *thief* and seeks out a *victim* thread from which to *steal* work. 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 ensures that work is executed exactly once and that the state of the program reflects the contributions of all workers.

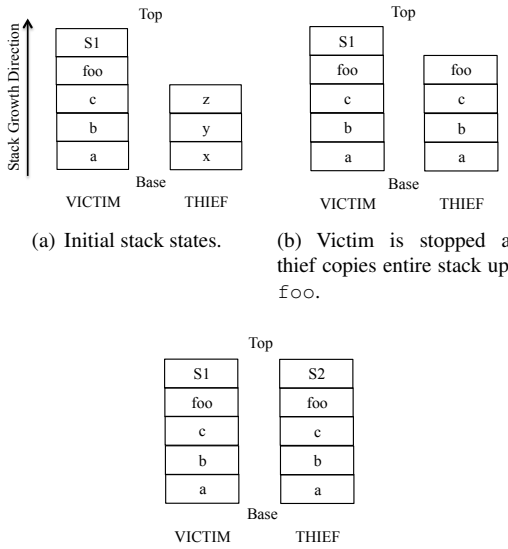
2.3 X10

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, `S`, 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. Figure 1 shows X10's **`finish`**–**`async`** programming model. X10 is implemented via compilation to either C++ (*native X10*) or Java (*managed X10*).

2.4 DefaultWS work-stealing framework

We use as our baseline the low-overhead work-stealing framework, JavaWS (Try-Catch), developed by Kumar et al. [21]. As demonstrated in that previous work, this framework achieves both good scalability and good absolute performance and is therefore a strong foundation for our work. The baseline framework supports managed X10 and uses the Jikes RVM [2] Java runtime. As in [21] the benchmark programs operate directly on Java arrays to avoid the sequential array access overhead of managed X10. In the rest of the paper we simply refer to this system as DefaultWS.

DefaultWS relies on: yieldpoints [3], on-stack replacement [11], dynamic code-patching [32], and exception handling. These fundamental mechanisms are already available in most production JVMs. The key engineering challenge the DefaultWS solves is how to represent the unusual code structure and control flow implied by the **`finish`**–**`async`**



(a) Initial stack states.

(b) Victim is stopped and thief copies entire stack up to `foo`.

(c) Victim is released and thief throws special exception to start `S2`.

Figure 2. Stack states during a steal procedure.

programming model in a way that facilitates efficient work-stealing. The DefaultWS system does this by re-writing **`finish`**–**`async`** into regular Java and exploiting the semantics Java offers for exception handling, which is very efficiently implemented in most modern JVMs. The result is that the runtime can walk a victim's stack and identify all **`async`** and **`finish`** contexts, resulting in a reduction in overhead from 4× to just 15%.

When a thief attempts to steal a task, it first requests the runtime to stop the victim so that it may safely walk the victim's execution stack. If the thief finds a steal-able task, it duplicates the victim's stack before allowing the victim to resume. The thief then runs a modified version of the runtime's exception delivery code to start this stolen task. Figure 2 describes this steal process in DefaultWS. The focus in this paper is on reducing the overheads arising from the thief interrupting the victim, which are incurred every time there is an attempt to steal.

We now conduct a quantitative analysis to characterize the dynamic overheads of workstealing.

3. Methodology

In Section 4 we conduct an analysis to motivate the problem we address. Before presenting that analysis, we briefly outline our experimental methodology, which is also used in the analysis of our solution, presented in Section 6.

3.1 Benchmarks

Because the primary goal of our work is to reduce the cost of steal operations, we have intentionally selected some of our benchmarks with high steal rates (they are available at `http://cs.anu.edu.au/~vivek/ws-vee-2014/`). DefaultWS almost completely eliminates the sequential overheads. This avoids the need to control task granularity and enhances programmer's productivity. The programmer only need to expose parallelism without worrying about the sequential per-task overhead. Most of our benchmarks follow this approach.

In each case we ported the benchmark to plain Java (for the sequential case). This sequential version does not have any work-stealing specific calls and also does not have any synchronization constructs. The managed X10 compiler automatically generates the Jikes RVM work-stealing runtime calls from the X10 version of benchmarks. Our six benchmarks are:

Jacobi Iterative mesh relaxation with barriers: 10 steps of nearest neighbor averaging on 1024×1024 matrices of doubles (based on an algorithm taken from Fork-Join).

FFT This is a Cooley-Tukey Fast Fourier Transform algorithm (adopted from Cilk). Input size is 1024×1024.

CilkSort A divide and conquer variant of mergesort (adopted from Cilk) for sorting 10 million integers.

Barnes-Hut A n-body algorithm to calculate gravitational forces acting on a galactic cluster of 100000 bodies. Adopted from Lonestar benchmark suite [20].

UTS The unbalanced tree search benchmark designed in [27]. We have used their tree type T2.

LUD Decomposition of 1024×1024 matrices of doubles (adopted from Cilk).

3.2 Hardware Platform

All experiments were run on a dual-socket machine with two Intel Xeon E5-2450 Sandy Bridge processors. Each processor has eight cores running at 2.10 GHz sharing a 20 MB L3 cache. The machine was configured with 47 GB of memory.

3.3 Software Platform

Jikes RVM Version 3.1.3. We used the production build. This is used as the Java runtime for managed X10. The command line arguments we used are: -Xms1024M -X:gc:variableSizeHeap=false -X:gc:threads=1.

OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode).

Fork-Join Java Fork-Join work-stealing framework [22]. Version 1.7.0.

Habanero-Java A work-stealing framework from Rice University, which uses X10's **`finish`**–**`async`** style in the Java programming language [7]. Version 1.3.1. We were unable to compile the benchmarks with the adaptive runtime of Habanero-Java. We build with both work-first and help-first policies and report the time from the policy which performs best for a particular benchmark.

We ported JavaWS (Try-Catch) of Kumar et al. from version 3.1.2 of Jikes RVM to version 3.1.3. This also includes one bug fix. In the original system the thief performs a small pause in the case when it fails to find a victim from any worker. After this pause, the thief reiterates searching for a victim. The downside of the pause is minimal in the case of infrequent steals, however even this small pause becomes a measurable overhead in frequent stealing. Hence, we modified JavaWS (Try-Catch) and allow the thief to continuously spin, searching for victims. This same setting is used in default work-stealing implementation of X10.

3.4 Measurements

For each benchmark, we ran twenty invocations, with fifteen iterations per invocation where each iteration performed the kernel of the benchmark. We report the mean of the final five iterations, 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 as one. We report the mutator time only in all the experiments.

4. Motivating Analysis

Although work-stealing is a very promising mechanism for exploiting software parallelism, it can bring with it formidable overheads to the simple sequential case. Kumar et al. [21] exploited rich features that pre-exist within the JVM implementation to significantly reduce these overheads from around 4× to 15% [21]. We use their system to further attack the problem of *dynamic* overheads — those associated with the cost of each steal — which increase as parallelism increases. Our approach is also to exploit highly optimized features within the runtime.

The principal sequential costs relate to organizing normal computation in such a way as to facilitate movement of a task to another thread if a steal should happen to occur. On the other hand, the principal cost in the dynamic case lies in synchronizing victim and thief threads at the time of a steal to ensure that the thief is able to take the victim's work without tripping upon each other.

Kumar et al. leveraged the runtime's yieldpoint mechanism to yield the victim while each steal took place. The yieldpoint mechanism is designed precisely for preemption of threads and has been highly optimized. When a thief initiates a steal, it sets a yield bit in the victim's runtime state. The next time the victim executes a yieldpoint, it will see the yield bit and yield to the thief. The JVM's JIT compiler injects yieldpoints on method prologues and loop back edges, tightly bounding the time it takes the victim to yield. Notwithstanding the efficiency of the yieldpoint mechanism, this approach nonetheless requires the victim to yield for the duration of the steal, whether or not the steal is successful.

To shed light on the dynamic costs due to stealing and further motivate our design, we now measure 1) the steal rate (steals/msec), and 2) the overhead imposed by the steal mechanism upon the victims.

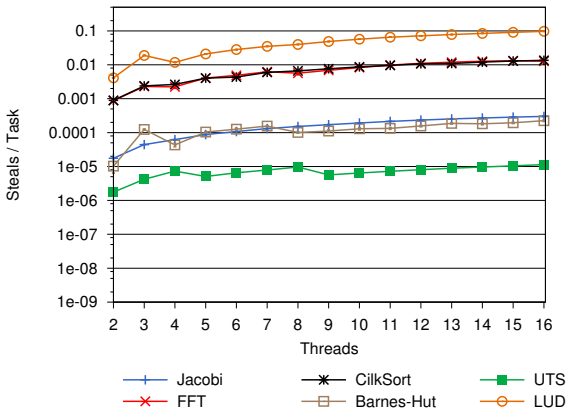


Figure 3. Steal *ratio*, as a function of thread count.

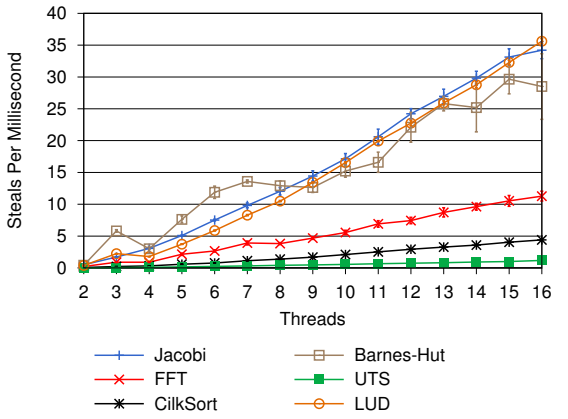


Figure 4. Steal *rate* as a function of thread count.

4.1 Steal Rate

The steal *ratio* (Figure 3) is only one dimension of the steal overheads. We also measure the steal *rate* (steals per millisecond), which is shown in Figure 4. Steal rate is calculated by dividing the total number of steals by the benchmark execution time. This indicates how frequently we are forcing the victim to execute the yieldpoint.

From Figure 3, we can notice that the steal *ratio* for Jacobi at 16 threads is as low as 0.0004. However, out of all the benchmarks, Jacobi has the highest *rate* of almost 35 steals per millisecond with same number of threads. This result shows that even a benchmark with very low steal ratio can still have a very high steal rate. In our next study we will explore how the high steal rates can affect the overall performance of the benchmarks.

4.2 Steal Overhead

In this study, we measure the cost of steals as imposed upon the victim by the thief. We measure this by calculating the

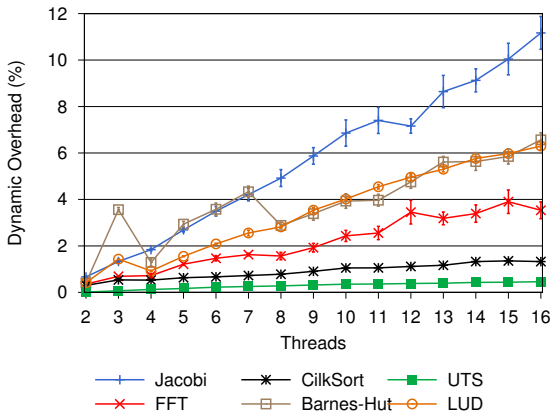


Figure 5. Dynamic overhead as a function of thread count.

percentage of CPU cycles lost by the victim while waiting for the thief to release it from the yieldpoint. For measuring the CPU cycles utilized by the work-stealing threads, we use hardware performance counters. We use the time stamp counter (TSC) [18] for measuring the cycles lost by the victim waiting to be released from yieldpoint. These cycles are summed for all the steals over the benchmark execution. The result in Figure 5 is calculated by dividing these cycles by total program execution cycles obtained from hardware performance counters as mentioned above.

By comparing this overhead in Figure 5 with the steal rate in Figure 4, we can see that higher steal rates correlate with higher overheads. The steal overhead can even be as much as 11.2% (Jacobi with 16 threads). This study clearly shows that forcing the victim to wait inside a yieldpoint at every steal is not an efficient strategy.

5. Design and Implementation

The previous sections identified the problem of dynamic overhead in a work-stealing runtime, highlighting the inefficiency of forcing the victim to wait inside a yieldpoint each time an attempt is made to steal work from it. We approach the problem by using a return barrier [36], to 'protect' the victim from any thief, which may be performing a steal lower down on the victim's stack. The insight is that the cost of the barrier is only incurred each time the victim unwinds past the barrier. So long as the victim remains above the protected frame, it sees no cost at all, and yet is fully protected from any thief stealing work lower down on the stack.

We now discuss the design and implementation of our return barrier, and the modifications made to DefaultWS (section 2.4).

5.1 Return Barrier Implementation

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 activity that occurs below the frame in which the barrier is installed. Only when the frame above the return barrier is unwound does the victim need to consider the possibility of an active thief.

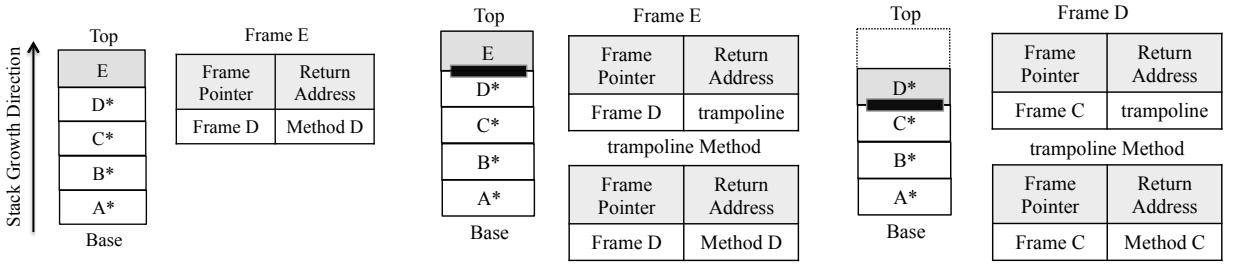
A naive implementation of a return barrier would require some (modest) code to be executed upon every return, just as a write barrier is typically executed upon every pointer update. Instead we use an approach similar to that of Yuasa [36]. We hijack the return address for a given frame, redirecting it to point to a special return barrier trampoline method, remembering the original return address is a separate data structure. When the affected frame is unwound, the return takes execution to our trampoline method rather than the caller of the returning frame. The trampoline method executes the return barrier semantics (which may include reinstalling the return barrier at a lower frame), before returning to the correct calling 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 unwound.

We can use the return barrier trampoline to protect the victim from active thieves — ensuring that the victim never unwinds to a frame a thief is actively stealing. We now discuss the general process of stealing work before detailing how we use the return barrier to perform efficient work-stealing.

5.2 Overview of Conventional Steal Process

Before describing our return barrier-based implementation, we outline the steps used to perform a steal in the prior implementation. In this process the thief steals the *oldest unstolen* continuation from the victim.

1. The *thief* **initiates** a steal.
2. The *victim* **yields** execution at the next yieldpoint.
3. The *thief* performs a **walk** of the victim's stack to find the oldest *unstolen* continuation frame.
4. The *thief* adjusts the return addresses of the callee of the stolen continuation to ensure the unstolen callee is correctly **joined** with the stolen continuation upon return.
5. The *thief* **copies** the frame of the stolen continuation and those of each of its callers onto a secondary stack in the following steps:
   - The *thief* **links** the copied frame on the secondary stack.
   - The *thief* **scans** callee frames to capture any references pertinent to the stolen frames. This is necessary due to the callee-save calling conventions used by many compilers.
6. The *victim* **resumes** execution.
7. The *thief* **throws** a special exception, which has the effect of resuming its execution on the secondary stack (which is now its primary stack).



(a) Initial state of victim's stack. (b) Thief installs return barrier on frame E. (c) Victim moves the return barrier on frame D.

Figure 6. The victim's stack, installation, and movement of the return barrier.

Notice that the victim must yield to the thief throughout steps 2 to 6. We now discuss how the return barrier can be used to avoid such yields when possible.

5.3 Installing the First Return Barrier

Figure 6(a) depicts a typical snapshot of a victim's execution stack. The stack frames with stealable continuations are marked with a * in this figure. The newly executed methods occupy the stack frame slots on the top of the 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 information of interest to us is the return address, which holds the address where the control should be transferred after unwinding to the caller frame.

Once the thief has decided to rob this victim, it first checks whether a return barrier is already installed on the victim's execution stack. If it discovers that there is no return barrier installed, the thief then stops the victim by forcing it to execute the yieldpoint mechanism (steal step 2). Once the victim has stopped, the thief starts walking the stack frames to identify the oldest unstolen continuation (steal step 3). In our example, it is the frame *A*. However, before the thief reaches frame *A*, it notices that the first (newest) available continuation is *D*. It installs a return barrier to intercept the return from method *E* to *D*. The return address and return frame pointer in *E* is hijacked by the return barrier trampoline. The return address stored in *E* is changed to that of the return barrier trampoline method. Figure 6(b) depicts the victim's modified execution stack.

The victim holds two boolean fields *stealInProgress* and *safeToUnwindBarrier*, which are now marked as *true* and *false* respectively by the thief. The flag *stealInProgress* is marked as *false* at the end of steal step 5, whereas *safeToUnwindBarrier* is marked *true* at the end of steal step 3. After installing the return barrier, the thief clones the *entire* stack of the victim and then allows the victim to continue. The victim continues the rest of its computation (i.e. frame *E*), oblivious to the activity of the thief, while the thief proceeds further with the stack walk in steal step 3. However, the thief now switches to the cloned stack of the victim.

5.4 Synchronization Between Thief and Victim During Steal Process

When the victim finishes executing method *E*, it returns via the trampoline method of the return barrier. It checks whether *safeToUnwindBarrier* is *true*. In this example, we assume it is still *false*.

Apart from the above two boolean flags, the victim also has a fixed size address array (we used size 20) to store frame pointers of its unstolen continuations. During the stack walk up to frame *A*, the thief updates the victim's frame pointer address array with the frame pointers of *C* and *B* (unstolen continuations). However, in reality there could be some unstealable frames in between stealable frames *D*–*A*. To make the description simpler, we have chosen this layout. In cases where there are more continuations than the victim's address array size, the thief starts inserting the surplus addresses from the middle index. After completing steal step 3 the thief marks the flag *safeToUnwindBarrier* as *true*. The victim is now ready to unwind to frame *D*.

There are situations when the frame *D* is the only unstolen continuation remaining on victim's stack. In this case the flag *safeToUnwindBarrier* will be marked as *true* only at the end of steal step 5. In this case, the victim cannot continue in parallel to the steal procedure so must wait on a condition variable until *stealInProgress* is *false*.

5.5 Victim Moves the Return Barrier

In our running example, the victim is now inside the return barrier trampoline method. The thief has finished steal step 3 and marked *safeToUnwindBarrier* as *true*. Frame *C* is the first frame pointer inside victim's frame pointer address array. Victim changes the position of the return barrier and reinstalls on frame *C*. After this the victim safely unwinds to frame *D* and starts execution of the method *D*. Figure 6(c) shows this newly modified stack frame of the victim. It keeps on changing the return barrier position until the last available frame pointer in its address array.

Once the steal is complete, the thief sets the victim's field *stealInProgress* to *false* and signals the victim. The victim is now ready to branch to join its part of the computation and become a thief itself. Hence, the return barrier helps the

victim continue its computation in parallel to thief's steal steps 3–5.

5.6 Stealing From a Victim with Return Barrier Pre-installed

Once installed, the return barrier removes the need for the yieldpoint mechanism in the steal step 2. Any thief that attempts to steal from a stack with the return barrier installed simply marks the victim's field *stealInProgress* as *true* and continues the rest of the steal steps 3–5 concurrently with the victim's computation. The thief uses the cloned stack of the victim (from the previous thief) to complete rest of its steal phases. We call this type of steal a *free steal*. There is no overhead imposed on the victim (unless the victim waits inside trampoline).

6. Results

We begin our evaluation of return barriers by measuring the reduction in dynamic overhead before evaluating the overall performance gain.

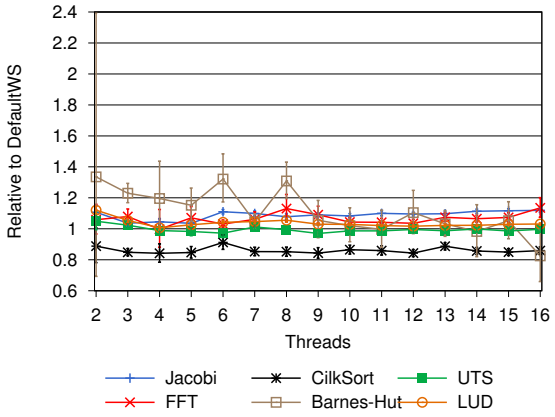
6.1 Dynamic Overhead

We measure the dynamic overhead of work-stealing for both the default system and our system using the return-barrier. Our methodology remains same as that used in Section 4.2; we use the TSC to accurately measure the cycles spent waiting for steals and express that as a percentage of total execution time. Figure 7 shows the dynamic overhead in both the systems as a function of the number of worker threads. With 16 worker threads, the dynamic overhead reduces in Jacobi by 29% (i.e. from 11.2% to 8%), in FFT by 40%, in CilkSort by 46%, in UTS by 60%, in LUD by 24%, and in Barnes-Hut by 30%.

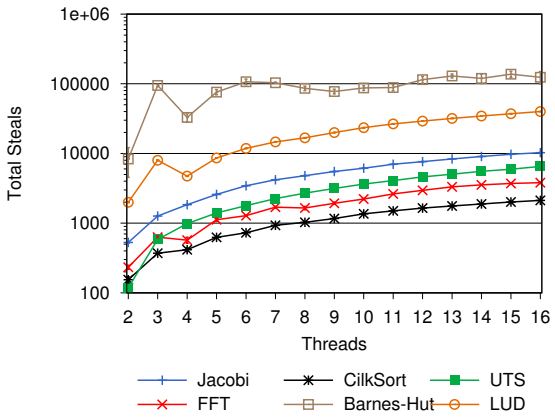
Now we explore how the use of the return barrier affects steal rates. Figure 8 compares the total number of steals in ReturnBarrierWS relative to those in DefaultWS. Values above 1.0 represent higher number of steals in ReturnBarrierWS than DefaultWS and vice versa. We observe from these figures that with the exception of CilkSort (at all thread counts) and Barnes-Hut (low thread counts), all other benchmarks exhibit very similar steal rates in both systems. The higher steal rates at low thread counts in Barnes-Hut show up in Figure 7(d) as nullifying the return barrier advantage. On the other hand, the 15% reduction in steals in CilkSort is consistent with the good result seen in Figure 7(c). Aside from these two outliers, ReturnBarrierWS sees a similar steal rate and consistently reduces the dynamic overhead in all other benchmarks.

6.2 Overhead of Executing Return Barrier

We now examine the cost to each thread of using the return barrier. Recall that this cost is only encountered when the stack unwinds to the point where a trampoline is installed. The trampoline is executed and it will either: a) reinstall itself on the next unstolen continuation frame further down



(a) Relative to DefaultWS.



(b) Absolute steal count.

Figure 8. Total steals in ReturnBarrierWS.

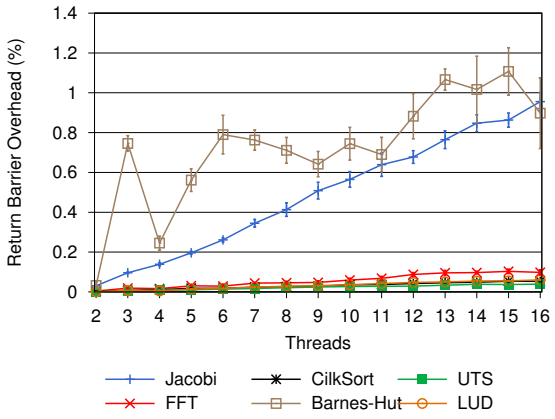


Figure 9. Overhead of executing return barrier in victims.

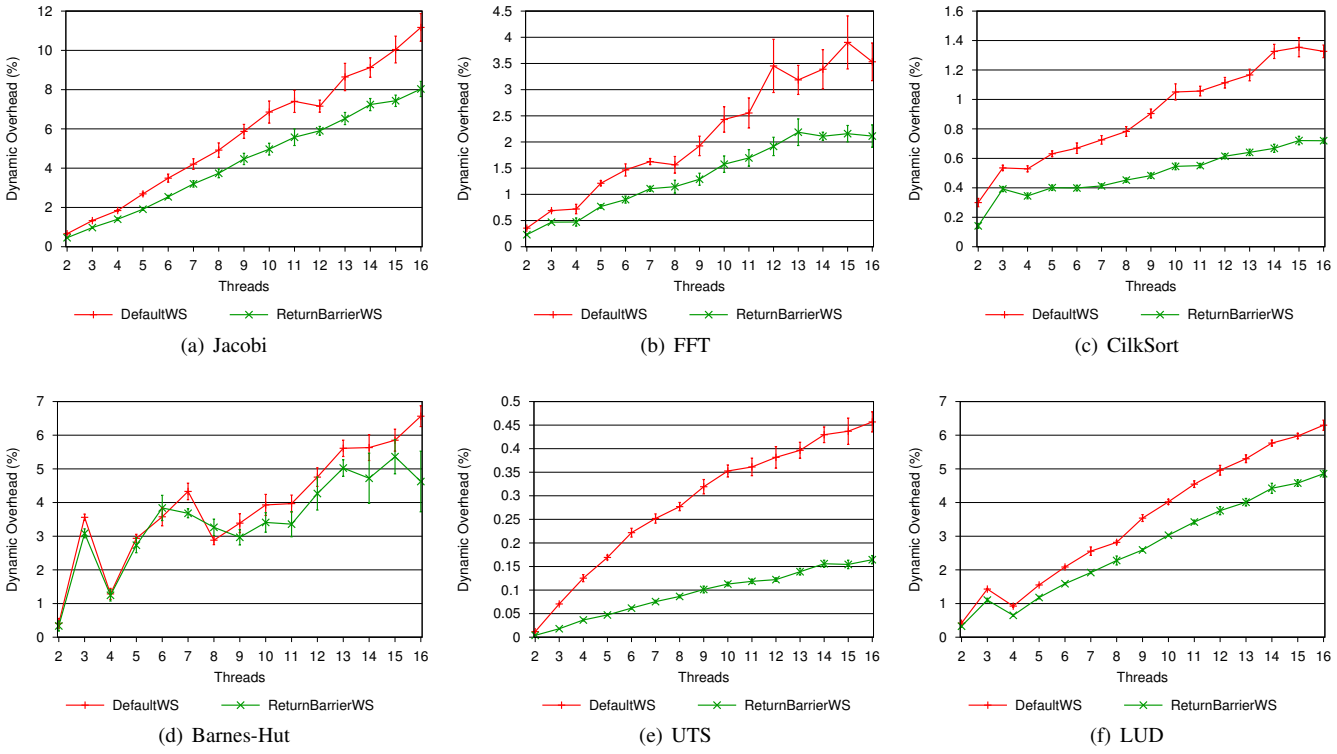


Figure 7. Dynamic overhead in our old and new systems.

before returning to the hijacked frame; or b) wait on a condition lock if there are no more unstolen continuations left and the steal is still in progress. We measure this overhead by using a high-resolution timer and measuring the time spent performing this operation. Figure 9 shows this overhead as a percentage of total program execution. With sixteen worker threads, the maximum overhead is around 0.95% in Jacobi and the minimum is 0.04% in UTS. However, Barnes-Hut shows an overhead of 0.7% even with just 3 threads.

Figure 8(b) shows that Barnes-Hut has the highest number of steals. Even with just three threads, there are around 95000 steals, whereas the closest, LUD, has just 8000 steals. More steals means more frequent trampoline visits by victims. This combined with a shallow stack (Section 6.3) leads to Barnes-Hut showing the highest overhead for the return barrier (max 1.1%).

6.3 Free Steals From Return Barrier

Recall from Section 5.6 that return barriers allow thieves to perform some steals for free. Figure 10 shows the percentage of steals that are free. UTS and CilkSort shows the maximum number of free steals. As the thread count increases, the percentage of free steels for these benchmarks converge close to 30%. FFT has 10% and Jacobi has 7%. Barnes-Hut and LUD have the lowest (3.5% and 1.5% respectively).

A higher free steal count reflects the return barrier staying longer on the victim's stack. This tends to reflect the depth of

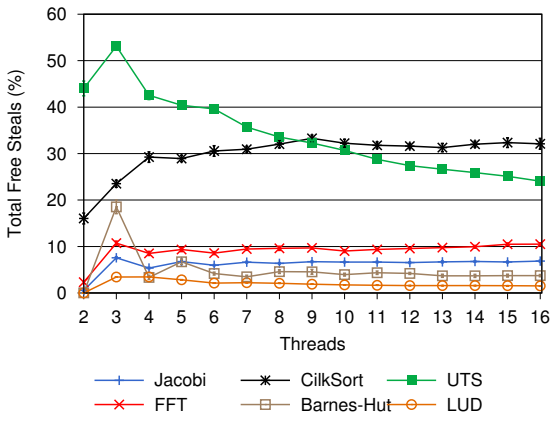


Figure 10. Free steals.

victim's stack. A shallow stack will mean that the victim will tend to more often unwind past the return barrier, meaning that the thief tends to more often require the victim to execute the yieldpoint mechanism. This reduces opportunities for the return barrier to reduce the dynamic overhead. Figure 7 supports this conjecture. LUD and Barnes-Hut benefit the least, whereas UTS and CilkSort benefits the most.

6.4 Overall Work-Stealing Performance

The goal of this work was to reduce dynamic overheads, which are naturally most evident when the level of paral-

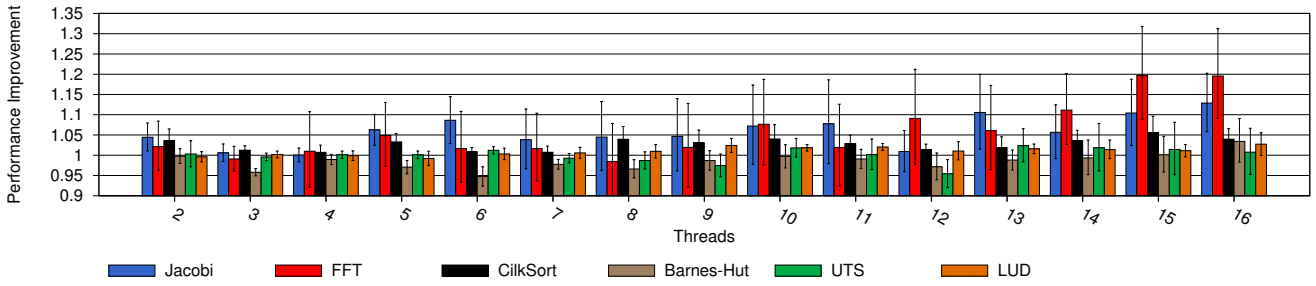


Figure 11. ReturnBarrierWS performance relative to DefaultWS.

lelism is high. Now we explore how the use of the return barrier affects performance when the number of threads grows to 16. In Section 6.5 we add a speedup comparison to the Fork-Join framework and Habanero-Java.

Figure 11 shows speedup relative to DefaultWS for each of the benchmarks on our 16 core machine. The time with n worker threads in ReturnBarrierWS is normalized to the time for n worker threads in DefaultWS. Values above 1.0 reflect a benefit. We can expect benefits at high steal rates, which also happens as parallelism increases. Jacobi reaches the 10% mark with 13 threads. With 16 threads, Jacobi is 13% faster and FFT is 20% faster. CilkSort gets a maximum benefit of 5% (15 threads). UTS, LUD and Barnes-Hut almost remains unchanged.

The absence of a performance improvement in LUD and Barnes-Hut is despite the fact that their maximum dynamic overheads (close to 6%) are even higher than that of FFT (3.5%). The reason for this is little improvement in their dynamic overheads due to the presence of a shallow stack (Section 6.3). On the other hand, UTS already has very low dynamic overhead in DefaultWS (max 0.5%). This results in no benefit in performance even by reducing its dynamic overhead by 60%.

6.5 Comparison to Fork-Join and Habanero-Java

The Fork-Join framework [22], which is now a part of Java 7, is a widely used work-stealing framework. Habanero-Java is another such project, which is inspired from X10 and is actively being used [7, 13, 14, 23, 34, 37]. To determine the performance of our system, we now do a speedup comparison with Fork-Join and Habanero-Java on all our benchmarks. We measure speedup relative to the sequential Java version of each benchmark.

From the speedup graph in Figure 12, we can see that ReturnBarrierWS achieves significantly better speedup on Jacobi, FFT, CilkSort and Barnes-Hut than both the other systems. For UTS, ReturnBarrierWS performs similar to Fork-Join but better than Habanero-Java (threads 2 to 10). LUD performs better in Habanero-Java than both ReturnBarrierWS and Fork-Join. However, the gap in LUD speedup across all the three systems is not very wide.

One interesting point to notice is the poor speedup of Barnes-Hut and Jacobi in Fork-Join and Habanero-Java. Compared to the sequential Java version, single threaded Barnes-Hut is 83% slower in Fork-Join, 95% slower in Habanero-Java and 30% slower in ReturnBarrierWS. Similarly, Jacobi is 70% slower in Fork-Join, 90% slower in Habanero-Java and 10% slower in ReturnBarrierWS. The large slowdown in Fork-Join and Habanero-Java is because of the sequential overheads of work-stealing. Kumar et al.'s work, which we build upon, successfully attacked sequential overheads, leading to the good underlying performance we see here. To verify our findings, we also ran Barnes-Hut and Jacobi on both Fork-Join and Habanero-Java with OpenJDK JVM and observe the same behavior. This verified our belief that its the sequential overhead that is taking the toll on these two benchmarks.

6.6 Summary

These encouraging results demonstrate that our approach is effective at reducing the dynamic overhead and also improves scalability. Our approach therefore promises improved performance with increasing parallelism. However, even for benchmarks with very small dynamic overhead, our approach does not negatively affect performance.

7. Related Work

Stealing overheads In our work-stealing implementation, we steal only one task at a time as in native X10, Cilk, Fork-Join etc. 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 [4, 6], several authors have argued that the scheme can be improved by allowing multiple tasks to be stolen at a time [5]. Dinan et al. [10] demonstrate that a 'steal-half' policy gave the best performance in their distributed setting. Stealing multiple tasks in a distributed setting has proven to be better in several other studies [24, 25, 29]. Guo et al. [13] introduce and evaluate the *help first* scheduling policy for the scalability of depth first search algorithms. Cong et al. [9] explore the idea of adaptive task batching for irregular graph algorithms. The thieves steal a batch of tasks at a time, where the batch size is determined adaptively. Several other authors have also ar-

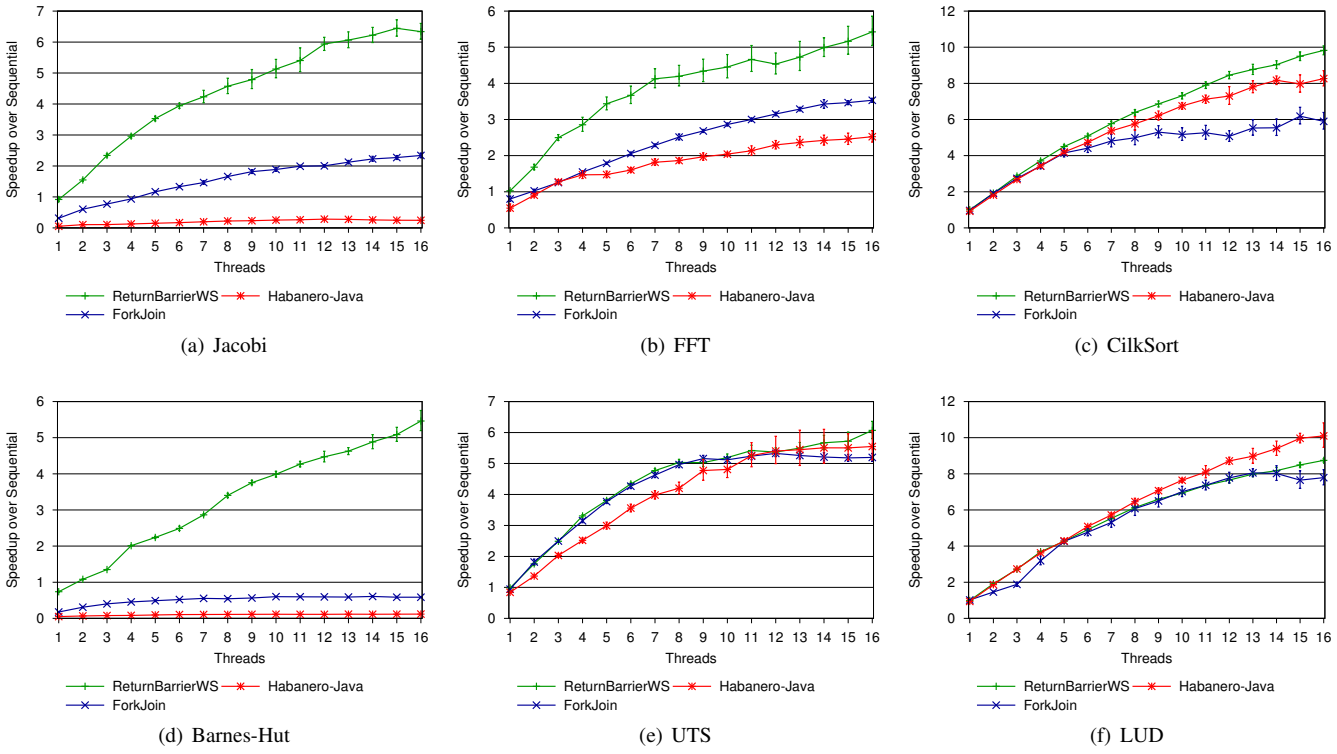


Figure 12. Speedup over sequential Java.

gued that stealing multiple tasks works best in irregular algorithms [1, 15, 16, 31].

These studies show that stealing multiple tasks is better in two cases: a) when performing work-stealing over a distributed setting, where the cost of stealing from remote node is substantial and hence stealing multiple tasks amortizes the communication overhead; and b) in irregular problems, such as depth-first-search algorithm, that do not fit into the divide-and-conquer model. However, not all the workloads are irregular in design nor do all follow the divide-and-conquer style algorithm, where the steal one approach always works better in non-distributed setting. Though we have targeted divide-and-conquer style algorithms, our insight of using a return barrier to reduce the cost of stealing will perform well in irregular algorithms as well.

Our return barrier mechanism for work-stealing and the framework by Kumar et al. on which it is built have some similarity with Umatani et al.'s work [33]. The fundamental insight is the same in both systems: work-stealing overheads can be significantly reduced by deferring operations that most other work-stealing systems perform eagerly. However, the systems are different in a number of ways including: a) In their system a thread starts with stack allocated activation frames. They consider steals to be very infrequent and hence at every steal, the victim heap allocates all of its continuations and stores on its deque. This greatly differs from our system. Frames in our system are always part of execution stack and simply copied from victim to thief's execution stack (not heap allocated). As reported in Section 4, steals are not always infrequent and hence Umatani et al.'s assumption would lead to large overheads in those cases. b) To start a stolen task, explicitly saved states have to be restored from heap. In DefaultWS, states are already a part of an execution stack and our thief merely throws a special exception to launch the stolen task. c) In their system, when the victim's stack is empty, it tries to get an unstolen heap frame from its deque. This transition between stack to deque can be thought of as a return barrier. For the first steal, victim required synchronization and then again while fetching a heap allocated frame. However, we use explicit return barriers as we treat the victim's execution stack as its implicit deque. d) They used only two microbenchmarks (Fibonacci and Matmul) to evaluate their implementation. We don't have access to their versions of these two tests, but the data presented in their paper suggests that the sequential overhead on their improved system for these two tests is about 57% and 20% respectively. For the same microbenchmarks, Kumar et al. see sequential overheads on JavaWS (Try-Catch) of 35% and 18% respectively. Like Kumar et al., we too use more benchmarks including several real world benchmarks.

Return barriers The return barrier mechanism was first used in [17] in the context of debugging optimized code, to allow lazy dynamic deoptimization of the stack. It has also

been used in various garbage collector algorithms [19, 30, 35, 36]. In this work, we exploit the return barrier mechanism to optimize the steal process. To our knowledge, return barriers have not been applied to work-stealing until now.

8. Conclusion

Effectively exposing software parallelism to underlying hardware is a pressing issue, a problem addressed by task-based scheduling offered by systems such as the Fork-Join framework and languages such as X10 and Habanero-Java. These systems use work-stealing to schedule work across the underlying parallel hardware. Unfortunately work-stealing comes with significant overheads. In this work, we identify victim yields as a major source of remaining overhead, and show that a return barrier can be used to greatly reduce this source of overhead. The return barrier allows a thief to steal from a victim's stack without requiring the victim to yield in the common case. Our technique enables improvements of as much as 20% in total execution time. The extremely low dynamic overhead should enable increasing speedups with the increasing parallelism in modern hardware.

References

[1] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442538.

[2] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2010. ISSN 0018-8670. doi: 10.1147/sj.391.0211.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. doi: 10.1145/353171.353175.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM. ISBN 0-89791-989-0. doi: 10.1145/277651.277678.

[5] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5): 1260–1279, May 2003. ISSN 0097-5397. doi: 10.1137/S0097539701399551.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999. ISSN 0004-5411. doi: 10.1145/324133.324234.

[7] 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.

[8] 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.

[9] 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.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654113.

[11] 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.

[12] 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.

[13] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161079.

[14] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693504.

[15] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM. ISBN 1-58113-485-1. doi: 10.1145/571825.571876.

[16] R. Hoffmann and T. Rauber. Adaptive task pools: efficiently balancing large number of tasks on shared-address spaces. *International Journal of Parallel Programming*, 39(5):553–581, 2011.

[17] 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: 10.1145/143095.143114.

[18] Intel Corporation. Using the RDTSC instruction for performance monitoring, 1997. URL http://www.intel.com.au/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[19] G. Kliot, E. Petrank, and B. Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508296.

[20] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76, 2009. doi: 10.1109/ISPASS.2009.4919639.

[21] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384639.

[22] 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.

[23] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 885–902, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048133.

[24] 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*, SPAA '93, pages 164–172, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2. doi: 10.1145/165231.165252.

[25] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 212–221, New York, NY, USA, 1998. ACM. ISBN 0-89791-989-0. doi: 10.1145/277651.277687.

[26] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91631.

[27] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: an unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72520-6. URL http://dl.acm.org/citation.cfm?id=1757112.1757137.

[28] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2010.

[29] 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*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM. ISBN 0-89791-438-4. doi: 10.1145/113379.113401.

[30] H. Saiki, Y. Konaka, T. Komiya, M. Yasugi, and T. Yuasa. Real-time GC in JeRTy™ VM using the return-barrier method. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, pages 140–148, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2356-0. doi: 10.1109/ISORC.2005.45.

[31] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736055.

[32] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a Java Just-In-Time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.16.

[33] S. Umatani, M. Yasugi, T. Komiya, and T. Yuasa. Pursuing laziness for efficient implementation of modern multithreaded languages. In A. Veidenbaum, K. Joe, H. Amano, and H. Aiso, editors, *High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20359-9. doi: 10.1007/978-3-540-39707-6_13.

[34] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. Practical permissions for race-free parallelism. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 614–639, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_27.

[35] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, Mar. 1990. ISSN 0164-1212. doi: 10.1016/0164-1212(90)90084-Y.

[36] T. Yuasa, Y. Nakagawa, T. Komiyay, and M. Yasugiy. Return barrier. In *Proceedings of the International Lisp Conference*, 2002.

[37] J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Isolation for nested task parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 571–588, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509534.