Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism

Yi Guo Rajkishore Barik Raghavan Raman Vivek Sarkar
Department of Computer Science
Rice University
{yguo, rajbarik, raghav, vsarkar}@cs.rice.edu

Abstract

Multiple programming models are emerging to address an increased need for dynamic task parallelism in applications for multicore processors and shared-address-space parallel computing. Examples include OpenMP 3.0, Java Concurrency Utilities, Microsoft Task Parallel Library, Intel Thread Building Blocks, Cilk, X10, Chapel, and Fortress. Scheduling algorithms based on work stealing, as embodied in Cilk's implementation of dynamic spawn-sync parallelism, are gaining in popularity but also have inherent limitations. In this paper, we address the problem of efficient and scalable implementation of X10's async-finish task parallelism, which is more general than Cilk's spawn-sync parallelism. We introduce a new work-stealing scheduler with compiler support for async-finish task parallelism that can accommodate both work-first *and* help-first *scheduling policies. Performance results on two different multicore SMP platforms show significant improvements due to our new work-stealing algorithm compared to the existing* work-sharing *scheduler for* X10*, and also provide insights on scenarios in which the help-first policy yields better results than the work-first policy and vice versa.*

1 Introduction

The computer industry is entering a new era of mainstream parallel processing due to current hardware trends and power efficiency limits. Now that all computers — embedded, mainstream, and high-end — are being built using multicore chips, the need for improved productivity in parallel programming has taken on a new urgency. The three programming languages developed as part of the DARPA HPCS program (Chapel [5], Fortress [2], X10 [6]) all identified dynamic lightweight task parallelism as one of the prerequisites for success. Dynamic task parallelism is also being included for mainstream use in many new programming models for multicore processors and shared-memory parallelism, such as Cilk, OpenMP 3.0, Java Concurrency Utilities, Intel Thread Building Blocks, and Microsoft Task Parallel Library. In addition, dynamic data driven execution has been identified as an important trend for future multicore software, in contrast to past programming models based on the Bulk Synchronous Parallel (BSP) and Single Program Multiple Data (SPMD) paradigms. Scheduling algorithms based on Cilk's work-stealing scheduler are gaining in popularity for dynamic lightweight task parallelism but also have inherent limitations.

In this paper, we focus on a core subset of X10's concurrency constructs, consisting of the `async`, `atomic` and `finish` statements. We address the problem of efficient and scalable implementation of X10's dynamic *async-finish* task parallelism, which is more general than Cilk's *spawn-sync* parallelism and can easily be integrated into any of the programming models listed above. Our contributions include:

- A new work-stealing scheduling framework with compiler support for async-finish task parallelism that can accommodate both *work-first* and *help-first* scheduling policies.
- A non-blocking implementation of the help-first work-stealing scheduling policy. This is important for scalability in situations when the steal operations can become a serial bottleneck with the work-first policy.
- A study of the performance differences between work-first and help-first scheduling policies, and insights on scenarios in which the help-first policy yields better results than work-first policy and vice versa.
- Performance results that show significant improvements (up to $4.7\times$ on a 16-way Power5+SMP and $22.8\times$ on a 64-thread UltraSPARC II) for our work-stealing scheduler compared to the existing *work-sharing* scheduler for X10 [3].

The rest of the paper is organized as follows. Section 2 summarizes the similarities and differences between Cilk's spawn-sync parallelism and X10's async-finish parallelism, as well as past work on the Cilk work-stealing scheduler. Section 3 summarizes two key limitations of Cilk-style work-stealing schedulers in handling async-finish task parallelism, namely *escaping async's* and *sequential calls to parallel functions*, and our approach to addressing these limitations in a work-stealing framework. Section 4 introduces the help-first scheduling policy and describes our non-blocking implementation. Section 5 presents performance results on two different multicore SMP platforms. Section 6 discusses related work, and Section 7 contains our conclusions.

2 Background

In this section, we first compare the computations in Cilk spawn-sync task parallelism and the X10 finish-async task parallelism and then briefly summarize the Cilk work-stealing scheduler [10].

2.1 Cilk computation vs X10 computation

Blumofe et al. [4] defined the notion of *fully-strict* computation as follows. Each multithreaded computation can be viewed as a dag of dynamic instruction instances connected by dependency edges. The instructions in a task are connected by *continue* edges, and tasks form a spawn tree with *spawn* edges. *Join* edges are introduced to model the dependencies enforced by *sync* operations. A *strict* computation is one in which all join edges from a task go to one of its ancestor tasks in the spawn tree. A *fully-strict* computation is one in which all join edges from a task go to its parent task in the spawn tree. All computations generated by Cilk programs are fully-strict. A *terminally-strict* computation is one in which each join edge goes from the last (terminal) instruction of a task to an instruction in one of its ancestor tasks in the spawn tree [1].

As in the case of Cilk, an X10 computation can also be represented as a dag in which each node corresponds to a dynamic execution instance of an X10 instruction/statement, and each edge defines a precedence constraint between two nodes. Figure 1 shows an example X10 code fragment and its computation dag [1]. The first instruction of the main activity[1] serves as the *root* node of the dag (with no predecessors). Any instruction which spawns a new activity will create a child node in the dag with a *spawn* edge connecting the async instruction to the first instruction of that child activity. X10 activities may wait on descendant activities by executing a finish statement. We model these dependencies by introducing startFinish (l_2 in Figure 1) and stopFinish (l_8 in Figure 1) nodes in the dag for each instance of a finish

```
l1 S0;
l2 finish { //startFinish
l3     async {
l4         S1;
l5         async {
l6             S2;}
l7         S3;}
l8 } //stopFinish
l9 S4;
```

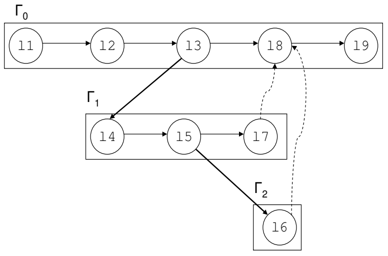

Figure 1. X10 **computation dag**

construct and then create *join* edges from the last instruction of each spawned activity within the scope of finish to the corresponding stopFinish instruction.

The async statement in X10 creates a new task in the computation dag and a spawn edge is created from the current task to the new task. In Cilk, the spawn statement has the same effect. Both the finish statement in X10 and sync statement in Cilk result in join edges being created from descendant tasks. However, there are interesting differences between the X10 and Cilk computations. For example, the direct join edge from activity Γ_2 to activity Γ_0 in Figure 1 is not allowed in Cilk because Γ_0 is not Γ_2's parent in the dag. The only way to establish such a dependence in Cilk is via Γ_1. In X10, it is possible for a descendant activity (e.g., Γ_2) to continue executing even if its parent activity (e.g., Γ_1) has terminated. This degree of asynchrony can be useful in parallel divide-and-conquer algorithms so as to permit sub-computations at different levels of the divide-and-conquer tree to execute in parallel without forcing synchronization at the parent-child level.

As shown in Figure 2, the class of Cilk's spawn-sync parallel computations is a subset of X10's async-finish parallel computations. Specifically, Cilk's spawn-sync computations must be fully-strict and terminally-strict, where as X10's async-finish computations must be terminally-strict but need not be fully-strict.

Blumofe et al. [4] proved that fully-strict computations can be scheduled with provably efficient time and space bounds using work-stealing. The same theoretical time and space bounds were extended to X10's async-finish parallel computations by Agarwal et al. [1], but no work-stealing implementations were designed and evaluated in that paper.

[1]The terms "activity" and "task" are used interchangeably in this paper.

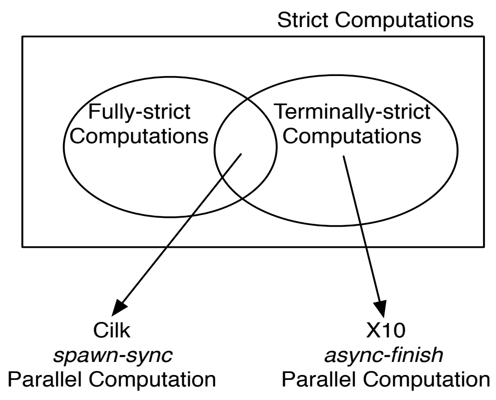

Figure 2. Classes of multithreaded computations

2.2 Cilk Work-Stealing Scheduler

The Cilk work-stealing runtime comprises of a set of workers, typically one per CPU or core. Each worker maintains a local *deque* of frames which represents work. The runtime starts with one worker executing the `main` function and the rest being idle with empty deques. Whenever a worker is idle, it becomes a thief and attempts to *steal* work from another worker's deque. On a spawn, the continuation is saved in a *frame* which is pushed onto the worker's deque so that other workers can steal it. Whenever the worker returns from a spawned task, it will first check if the frame that it pushed before the spawn is stolen. If so, the fully-strict model guarantees that there is no other useful work on the deque for the worker and hence it becomes a thief. Otherwise, it just executes the continuation of the spawn. Every Cilk function has two clones generated by the compiler: the *fast clone* and the *slow clone*. The fast clone is always invoked on a spawn, and is designed to have the smallest overhead possible. The slow clone is invoked when the thief steals a frame from a victim and then needs to resume the task at its apporpriate continuation. The slow clone contains operations trestore the execution context such as global and local variables etc.

3 Work-Stealing Extensions for X10 Computation

As discussed in Section 2, X10 computations form a non-trivial super-set of Cilk computations. In this section, we first identify two new features of X10 that demand compiler and runtime support for work-stealing, but have not been addressed by past work. Then we present our approach to supporting the two features. The approach described in this section represents a common framework that can be used by both the work-first and help-first policies described in the next section.

```
1 class V  {
2   V [] neighbors;
3   V parent;
4   V (int i) {super(i); }
5   boolean tryLabeling(V n) {
6       atomic if (parent == null)
7             parent = n;
8       return parent == n;
9   }
10  void compute() {
11     for (int i=0; i<neighbors.length; i++) {
12        V e = neighbors[i];
13        if (e.tryLabeling(this))
14          async e.compute(); //escaping async
15     }
16  }
17  void DFS() {
18     parent = this;
19     finish compute();
20  }}
```

Figure 3. Code for parallel DFS spanning tree algorithm in X10

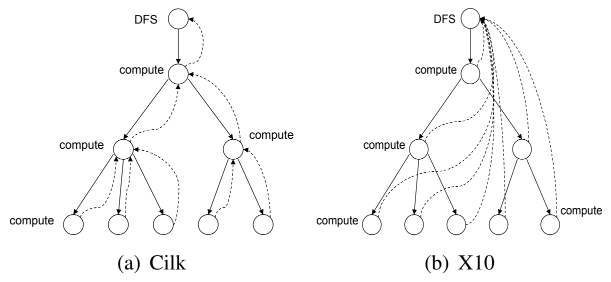

Figure 4. Cilk's and X10's spawn trees (solid lines) with join edges (dashed lines) for computation generated by program in Figure 3

3.1 Escaping Asyncs

The first feature is *escaping asyncs*, which is defined to be a task that outlives its parent and continues execution after its parent has terminated. As an example, consider the parallel-DFS spanning tree graph algorithm [9] shown in Figure 3. In a terminally-strict language like X10, a single finish scope at line 19 suffices for all descendant tasks spawned at line 14. It is possible in this X10 version for a call to `compute()` in a child task to outlive a call to `compute()` in a parent task. The only constraint is that all async calls to `compute()` must complete before the root task can continue execution past line 19. In contrast, a fully-strict computation model, like Cilk, will require the insertion of an implicit `sync` operation at the end of each task thereby ensuring that each parent task waits for all its child tasks to complete. In X10 terms, this would be equivalent to adding an additional finish scope that encloses the body of the `compute` function.

Figures 4(a) and 4(b) show the spawn trees for Cilk's fully-strict computation and X10's terminally-strict computation for the program in Figure 3. The solid and dashed arrows represent spawn and join edges respectively. Note that in Figure 4(b), all join edges go from a task to the root while in Figure 4(a), each join edge goes from a task to its parent. In terminally-strict computations, a task terminates without waiting for its descendants. This allows the runtime to collect the space used by those tasks earlier than in fully-strict computations. As discussed in Section 5, this is why it may be possible for a terminally-strict computation to use even less space than that of an equivalent sequential program (or of an equivalent fully-strict computation).

3.2 Sequential Calls to Parallel Functions

The second feature is that *sequential calls to parallel functions* are permitted in X10 but not in Cilk, as a result of which there is no need to distinguish between "parallel" and "sequential" functions in X10. In Cilk, a *parallel function* (also known known as a `cilk` function) is defined to be one that may spawn tasks. Sequential calls to parallel functions are not directly permitted in Cilk, but can be simulated by spawning the function and then performing a sync operation immediately thereafter. In addition to the overhead of extra spawns, this restriction has a significant software engineering impact because it increases the effort involved in converting sequential code to parallel code, and prohibits the insertion of sequential code wrappers for parallel code. In contrast, X10 permits the same function to be invoked sequentially or via an async at different program points.

The program shown in Figure 5 is valid in X10 but cannot be directly translated to Cilk. In Cilk, `C()` and `E()` would be *cilk functions* because they may spawn tasks. Thus `C()` and `E()` cannot be called sequentially in function `B()` and `D()` respectively.

3.3 Our Approach

3.3.1 Escaping Asyncs

To support escaping asyncs in X10, a finish scope is implemented as a bracketed pair of `startFinish` and `stopFinish` statements. Any task spawned within a finish scope is joined only at the `stopFinish` operation for the scope. In contrast, an implicit `sync` is inserted at the end of each `cilk` function to ensure fully-strictness.

We dynamically create a finish node data structure in the runtime for each `startFinish` statement. Various finish nodes are maintained in a tree-like structure with the parent pointer pointing to the node of its Immediately Enclosing Finish (IEF) scope. Apart from the parent pointer, each finish node keeps track of the number of workers that are working under its scope. When a worker is blocked at a `stopFinish`, the continuation after the finish scope is saved in the finish node. This continuation is subsequently picked up for execution by the *last* child in the finish scope (*i.e.,* the child which decrements the workers-counter to zero).

3.3.2 Sequential Call to a Parallel Function

The continuation after an async should contain enough context for the thief to resume the execution. In Cilk, each cilk function corresponds to a task in the computation dag. A parallel function cannot be called sequentially and must be spawned. For the work-stealing scheduler, this simplifies the continuation to contain only the activation frame of the current function because the thief that executes the continuation will never return to its caller as its caller must be its parent task whose continuation must have already been stolen.

In X10, when calling a parallel function sequentially, the activation frame of the caller still needs to be saved. This is because if stealing occurs in the parallel function, the thief may return from the call and execute the statements after the call. To support this feature, every continuation is extended to contain a stack of activation frames up to the previous `async` in the call chain. This stack of activation frames is managed at runtime at every sequential and parallel function call, but is only pushed to the deque at points where stealing may actually occur, i.e., the `async` program points. The thief that steals the frame is then responsible for unwinding the activation frame stack.

In Cilk, functions are distinguished as parallel or sequential using the `cilk` keyword. In our implementation, the compiler performs static interprocedural analysis to distinguish sequential and (potentially) parallel functions so as not to burden the programmer.

Consider the example in Figure 5. `C1` and `C2` label the points where stealing can actually occur. At `C1`, the frame pushed to the deque contains the stack of activation frames for `C1, L2, L1` in order. The thief that steals the frame is responsible for starting the continuation at `C1`. Upon returning from function `C`, the thief will resume the execution at `L2`, which will return to `L1`. At each return, the thief will find the activation frame required to resume the execution by popping the stack. The activation frame stack for `C2` contains the activation frames up to the previous spawn i.e., `C2` and `L3`.

As mentioned earlier, a naive way to support a sequential call to a parallel function in a work-stealing runtime is to enclose the sequential call in `finish-async`. This approach loses parallelism by disallowing the code after the sequential call to run in parallel with the task that escapes the callee of the sequential call. In contrast, our approach

```
A() {
  B();
L1: ...
}

B() {
  C();
L2: ...
}

C() {
  async D();
C1: ...
}
```

```
D() {
  E();
L3: ...
}

E() {
  if (...)
    async E();
C2: ...
}
```

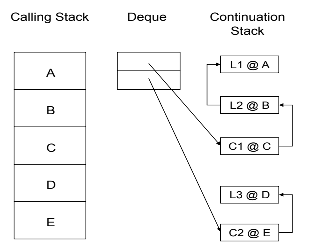

Figure 5. Support for sequential call to a parallel function

```
for (i=1 to SOR_ITERATIONS) {
      finish for (p = 0 to P-1) {
            // parallel for.
            // work partitioned into P chunks
            async {...}
      }
 }
```

Figure 6. `SOR` **code structure depicting iterative loop parallelism**

does not lose any parallelism.

Another approach to support sequential calls to parallel functions is to save the whole calling context before making an asynchronous call. This requires access to the runtime stack by user code, which is not allowed by many high-level programming language, such as Java, due to security reasons. Our approach instead saves local variables in *heap frames* that are manipulated with compiler and runtime support.

4 Work-first vs. Help-First Scheduling Policies

One of the design principles of Cilk is the *work-first* principle [10]. In a nutshell, the work-first policy dictates that a worker executes a spawned task and leaves the continuation to be stolen by another worker. In contrast, the *help-first* policy dictates that a worker executes the continuation and leaves the spawned task to be stolen. We use the "help-first" name for this policy because it suggests that the worker will ask for help from its peer workers before working on the task itself. Figure 8 shows the sketch of the code that the compiler will generate for both policies.

The work-first policy is designed for scenarios in which stealing is a rare event. When the number of steals is low, the worker will mostly execute the *fast clone* which has much lower overhead than the slow clone.

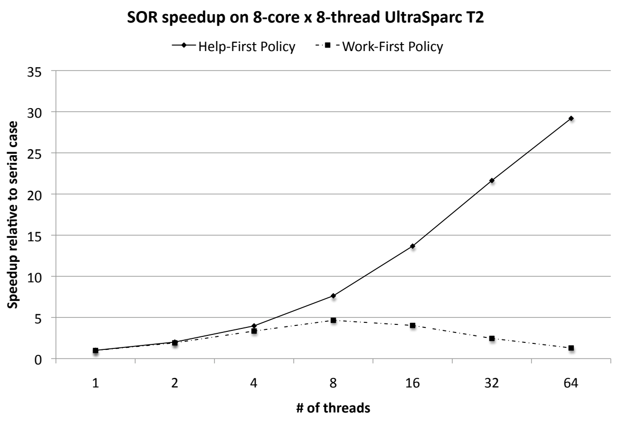

Figure 7. Speedup of `SOR` **over its serial version on 64-way UltraSparc T2 under help-first and work-first policies**

4.1 Limitations of Work-First Policy

In practice, we observe that the overhead of steals becomes increasingly significant as the number of workers increases. One example is iterative loop parallelism. Figure 6 shows the code structure of a wavefront algorithm implemented in an X10 version of the Java Grande `SOR` benchmark. It has a sequential outer loop and a parallel inner loop. The work in the inner parallel loop is divided evenly among P workers. Figure 7 shows the speedup of this `SOR` benchmark relative to its serial version on a 64-thread UltraSPARC T2 Niagara2 machine. If the asyncs are executed under the work-first policy, we observe a degradation in performance beyond 16 threads. The following analysis explains why.

Let the total amount of work in one iteration of the outer loop be T, the amount of time to migrate a task from one worker to another be t_{steal} and the time to save the continuation for each iteration be t_{cont}. Under the work-first policy, one worker will save and push the continuation onto the local deque and another idle worker will steal it. Therefore, distributing the P chunks among P workers will require $P-1$ steals and *these steals must occur sequentially*. The length of the critical path of the inner loop is bounded below by $(t_{steal} + t_{cont}) * (P - 1) + T/P$. According to the THE protocol [10], a push operation on the deque is lock-free but a thief is required to acquire the lock on the victim's deque before it can steal, thus $t_{cont} << t_{steal}$. As P increases, the actual work for each worker T/P decreases and hence, the total time will be dominated by the time to distribute the task, which is $O(t_{steal} * P)$.

Another example where the work-first policy suffers is the parallel Depth First Search (DFS) spanning tree algorithm shown in Figure 3. If the asyncs are scheduled using the work-first policy, the worker calls the `compute`

```
finish {
     async S1;
L1:  async S2;
L2:  S3;
}
```

```
startFinish();
push continuation after L1;
S1; // Worker executes eagerly
return if frame stolen
push continuation after L2;
S2; // Worker executes eagerly
return if frame stolen;
S3;
stopFinish();
```

Work-first Policy

```
startFinish();
push task S1 to local deque;
push task S2 to local deque;
S3;
stopFinish();
```

Help-first Policy

Figure 8. Handling async using work-first and help-first policy

function recursively and (like its equivalent sequential version) will overflow any reasonably-sized stack, for large graphs. Hence, one cannot use a recursive DFS algorithm if only a work-first policy is supported for asyncs. Since the stack-overflow problem also arises for a sequential DFS implementation, it may be natural to consider a Breadth First Search (BFS) algorithm as an alternative. However, Cong et al. [9] show that, for the same input problem, BFS algorithms are usually less scalable than DFS algorithms due to their additional synchronization requirements. As discussed in the next section, our approach in such cases is to instead use a *help-first* policy that has a lower space requirement for stack size than the work-first policy, for the DFS algorithm in Figure 3.

4.2 Help-First Scheduling Policy

As mentioned earlier, our work-stealing runtime also supports the help-first scheduling policy. Under this policy, upon executing an async, the worker will create and push the task onto the deque and proceed to execute the async's continuation. Once the task is pushed to the deque, it is available to be stolen by other workers and the stealing can be performed in parallel. Let t_{task} be the time to create and push the task to the deque. For the same reason mentioned in the last subsection, $t_{task} << t_{steal}$. As the stealing overhead is parallelized, the overhead of the steal operation is not a bottleneck any more. When asyncs are handled under help-first policy, the performance of the SOR benchmark scales well as shown in the Figure 7.

4.3 Non-blocking Steal Operation

As illustrated in the SOR example, steal operations for the same finish scope are serialized in a work-first policy. However, under the help-first policy, the steal operations can be performed in parallel by using a non-blocking algorithm. To support this, we extend the lock-free dynamic circular work-stealing deque proposed by Chase and Lev [7] and designed a non-blocking implementation for steal operations in help-first policy.

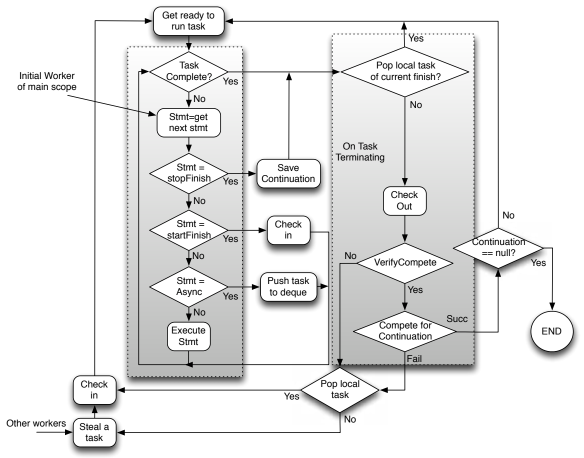

Figure 9. Flow chart for help-first work-stealing runtime

Figure 9 shows the main flow of the runtime and Figure 10 lists the pseudo-code of the related functions. Each finish scope has a global worker count (gc) and one local task counter (lc) for each worker. All counters are atomic and volatile. The global worker counter has a version number associated with it. The version number will increment every time the global worker counter is incremented from 0 to 1 (line 15).

After initialization, one worker will start executing the main function, which begins with the `startFinish`. Other workers will start stealing. The runtime terminates when the continuation after the main finish scope is executed.

We define the term *check in* and *check out* for a worker and a finish scope. A worker checks into a finish scope F if it enters F by calling `startFinish` or it begins to execute a task of F after stealing it from the other worker. A worker checks out of the finish scope F if it completes a task under F, or it cannot complete the current task because it is not safe to continue execution at the `stopFinish` of F while there is no local task under F on the deque. If there is still a local task under F, the worker will defer the check out until all local tasks under F are completed. Note that when a worker checks into a new finish scope F by calling `startFinish` of F, it will not check out of F's parent, but instead, will choose the worker that executes the continuation after F as its delegate to check out F's parent.

Now we argue that the runtime guarantees that for any finish scope F, the continuation after F is *safely* executed by *exactly* one worker:

- First, we argue that when `verifyComplete(F)` returns true at line 48, it is *safe* to execute the continuation after F, i.e., all tasks spawned within the finish scope F have been completed. When `verifyComplete` returns true, it verifies that all workers checked into the finish scope have been checked out and there is no worker that has stolen a task but not checked in yet. The former case is detected by verifying that the global worker counter is 0. The latter is detected by comparing the version number of the global worker counter before and after verifying all local task counters are 0. Note that in the steal function, the thief checks in (line 4) *before* the local task counter of the victim is decremented (line 5). If there is a worker that steals a task but has not yet checked in, the local task counter of the worker must be greater than 0. Observe that when a worker checks out of F, there is no task under the F on its local deque. So when the global counter of F is 0 and no stealing is happening, it is safe to execute the continuation after F.

- Second, we observe that `verifyComplete` will return true after the last worker decrements the global worker counter to 0. The `CAS` operation ensures that at most one worker will execute the continuation in case there are multiple workers competing for the continuation.

4.4 Discussion

An important theoretical advantage of the work-first policy is that it guarantees that the space used to run the parallel program is bounded by a constant factor of the space used in its sequential version. In the help-first policy, since we are creating tasks eagerly, the space bound is not guaranteed. As part of our future work, we plan to explore a hybrid approach that adaptively switches from a help-first policy to a work-first policy depending on the size of the local deques so as to establish a guaranteed space bound.

Another approach to creating a task eagerly is *work-sharing*, as in the current X10 runtime [3] which is

```
1   function Object help_first_steal () {
2      task = steal task from victim's deque;
3      finish = task's finish scope;
4      current worker checks in under finish;
5      finish.lc[victim]--;
6      return task;
7  }

8  function push_task_to_deque(task) {
9      finish = current finish scope;
10      finish.lc[this_worker]++;
11    this.deque.pushBottom(task);
12 }

13  function check_in(finish) {
14      if (finish.gc.getAndIncrement() == 0);
15          finish.gc.version++;
16  }

17  function check_out(finish) {
18      decrement finish.gc;
19  }

20  function startFinish() {
21      checks in new finish scope;
22  }

23  function stopFinish() {
24      finish = current finish scope;
25      save continuation after finish;
26      return to runtime;
27  }

28  function task OnTaskComplete() {
29     finish = current finish scope;
30     task = pop local deque;
31     if (task.finish == finish)
32         return task;
33     else
34         push task back to deque;
35     check_out finish;
36     if (verifyComplete(finish)) {
37        if (CAS(finish.gc, 0, -1)) {
38           return finish.continuation;
39     }
40     return null;
41 }

42  function boolean verifyComplete(finish) {
43      versionOld = finish.gc.version();
44      if (finish.gc != 0) return false;
45      if (not all lc of finish 0)
46         return false;
47      versionNew = finish.gc.version();
48      return versionOld == versionNew;
49  }
```

Figure 10. Pseudo code of help-first non-blocking steal protocols. Global counters (gc) and local counters (lc) are atomic and volatile.

based on the `ThreadPoolExecutor` class in the `java.util.concurrent` package. Unlike work-stealing, work-sharing uses a single task queue. Whenever an async is encountered, the task is created and inserted in the queue. All workers get their tasks from this global queue. The disadvantage of the work-sharing approach is that the global queue is likely to become a scalability bottleneck as the number of workers grows. In contrast, our work-stealing scheduler with the help-first policy maintains local deques for workers and adds newly created tasks onto these deques locally.

5 Experimental Results

In this section, we present experimental results to compare the performance of our portable implementation of work-stealing schedulers based on work-first and help-first policies. We also compare the work-stealing implementation with the existing work-sharing X10 runtime system. Section 5.1 summarizes our experimental setup. Section 5.2 compares the performance of the schedulers on a set of benchmarks that includes eight Java Grande Forum (JGF) benchmarks, two NAS Parallel Benchmark (NPB) benchmarks, and the Fibonacci, and Spanning Tree micro-benchmarks. The Java Grande Forum and NAS Parallel benchmarks are more representative of iterative parallel algorithms rather than recursive divide-and-conquer algorithms that have been used in past work to evaluate work-stealing schedulers. We use Fibonacci and Spanning Tree as microbenchmarks to compare the performance of the work-first and help-first policies described in Section 4. There are several advantages in using a new language like X10 to evaluate new runtime techniques, but a major challenge is the lack of a large set of benchmarks available for evaluation. To that end, we focused on X10 benchmarks that have been used in past evaluations [6, 3, 16, 18, 9].

5.1 Experimental Setup

The compiler and runtime infrastructure used to obtain the performance results in this paper is summarized in Figure 11. A Polyglot-based front-end is used to parse the input X10 program and produce Java class files in which parallel constructs are transformed to X10 runtime calls. To support our work-stealing based runtime, the front-end generated code needs to be modified to produce fast and slow clones for every method [10]. We achieve this in the *Code Gen* components in Figure 11 by using the Soot infrastructure [19] to transform the class files. The *work-sharing* runtime in the current X10 system does not need any special code generation for fast and slow clones. In an effort to achieve as close to an "apples-to-apples" comparison as possible, all paths use the same X10 source

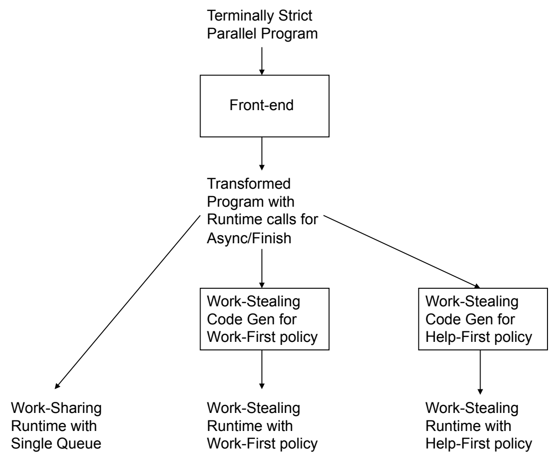
Terminally Strict Parallel Program
Front-end
Transformed Program with Runtime calls for Async/Finish
Work-Stealing Code Gen for Work-First policy
Work-Stealing Code Gen for Help-First policy
Work-Sharing Runtime with Single Queue
Work-Stealing Runtime with Work-First policy
Work-Stealing Runtime with Help-First policy

Figure 11. Compiler and runtime infrastructure for work-sharing and work-stealing schedulers.

program at the top of the figure and the same JVM at the bottom.

The work-sharing scheduler shown in the left hand side of Figure 11 represents the current scheduler in the open source X10 implementation. The work-sharing scheduler makes extensive use of the standard java.util.concurrent (JUC) library [15]. Details of the current X10 runtime can be found in [3].

The performance results were obtained on two multi-core SMP machines. The first is a 16-way 1.9 GHz Power5+ SMP[2] with 64 GB main memory; the runs on this machine were performed using IBM's J9 virtual machine for Java version 1.6.0. The second machine is a 64-thread 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory; the runs on this machine were performed using Sun's Hotspot VM for Java version 1.6. All results were obtained using the `-Xmx2000M -Xms2000M` JVM options to limit the heap size to 2GB, thereby ensuring that the memory requirement for our experiments was well below the available memory on all the machines. The main program of the benchmarks were extended with multiple-iterations within the same Java process for all JVM runs to reduce the impact of JIT compilation time in the performance comparisons. We also used the `-Xjit:count=0`, `optLevel=veryHot`, `ignoreIEEE`, `-PRELOAD_CLASSES=true` and `-BIND_THREADS=true` options for the Power5+ SMP runs. The `count=0` option ensures that each method is JIT-compiled on its first invocation. The `-PRELOAD_CLASSES=true` option causes all X10 classes referenced by the application to be loaded before

[2]Though the Power5+ SMP hardware has support for 2 hardware threads per core, we explicitly turned off SMT for all Power5+ runs.

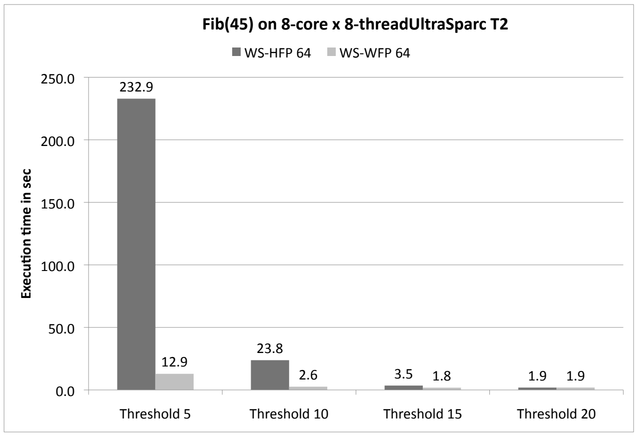

Figure 12. Execution times of `Fib(45)` using 64 workers for HFP (Help-First Policy) and WFP (Work-First Policy) with thresholds 5, 10, 15, and 20 on an UltraSparc T2 system.

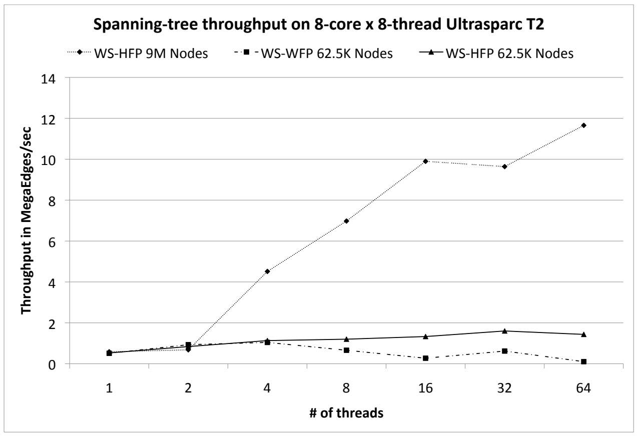

Figure 13. Spanning Tree throughput in terms of MegaEdges/sec for HFP (Help-First Policy) and WFP (Work-First Policy) on an UltraSparc T2 system. The maximum stack size we could use for WFP is 16MB.

the main program is executed. This approach is permitted for X10 (but not for Java in general), and allows for better code to be generated when each method is compiled on its first invocation. The `-BIND_THREADS=true` option prevents runtime migration of worker threads.

All JGF experiments were run with the largest data size provided for each benchmark except for the Series benchmark for which Size B was used instead of Size C. There are five available sizes for the NPB benchmarks (S, W, A, B, C), and we used the intermediate Size A for all runs in this paper. Since the focus of this paper is on task-level parallelism in an SMP, the `NUMBER_OF_LOCAL_PLACES` runtime option was set to 1 for all runs.

All performance measurements on the UltraSparc T2 machine followed the "Best of 30 runs" methodology proposed in [11], whereas a "Best of 5 runs" approach was used for the Power5+ SMP machine due to limited availability of the machine for our experiments.

5.2 Results

`Fib` is a classic example of recursive parallel divide and conquer algorithm, and is a useful micro-benchmark for work-stealing schedulers. Figure 12 compares the execution time of the work-first and help-first policies for `Fib(45)` using 64 workers and varying the *threshold* from 5 to 20. A threshold of T implies that all calls to `Fib()` with parameters $\leq T$ are implemented as sequential calls. Similar threshold based Fib performance measurements have been reported by D. Lea [13]. As predicted by past work (*e.g.,* [10]), the work-first policy (WS-WFP) significantly outperforms the help-first policy (WS-HFP) for smaller threshold values. This result re-establishes the fact that the work-first policy is well-suited for recursive parallel algorithms with abundant parallelism and small numbers of steals. However, if we increase the threshold to 20 (a value considered to be reasonable for Fib in [13]), the performance gap between WS-HFP and WS-WFP disappears.

In Figure 13, we compare the throughput of the work-first and help-first policies on the spanning tree microbenchmark for irregular large graphs. Throughput is shown in terms of million edges processed per second. Spanning tree is an important kernel computation in many graph algorithms. In general, large-scale graph algorithms are challenging to solve in parallel due to their irregular and combinatorial nature. There are three observations that can be drawn from Figure 13. First, WS-HFP outperforms WS-WFP for a given input graph (62.5K nodes). Second, as also observed in [9], WS-WFP is unable to run on large graphs due to stack size limitations. In our experiments, we were unable to get WS-WFP results for a graph larger than 62.5K nodes. Third, not only can WS-HFP process larger graphs (with 9M nodes in Figure 13), but it also achieves better scalability when doing so.

The Fib and Spanning-Tree microbenchmarks illustrated cases in which WS-WFP performs better than WS-HFP and vice versa, though the gap can be narrowed for Fib by increasing the threshold value for parallelism. We now turn our attention to the JGF and NPB benchmarks, with results presented for two SMP machines in Figures 14 and 15. For convenience, the speedup measurements in both figures are normalized with respect to a single-processor execution of the X10 work-sharing scheduler [3]. In previous work [16], we showed that the work-sharing scheduler can

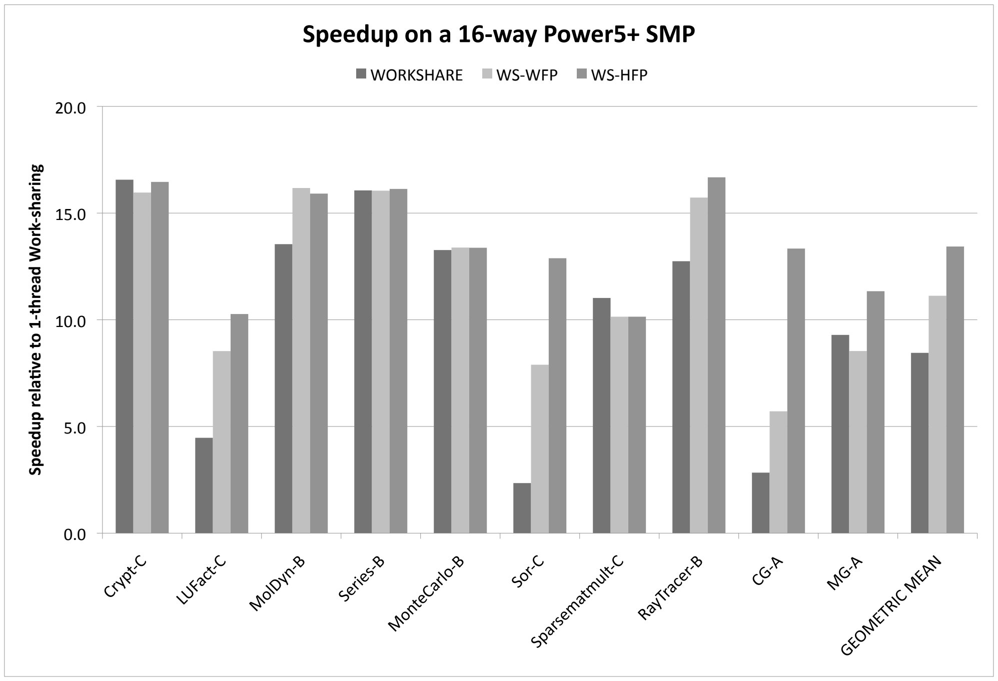

Figure 14. Comparison of work-sharing, HFP (Help-First Policy) and WFP (Work-First Policy) on a 16-way Power5+ SMP. The execution times were obtained using "Best of 5 runs" approach.

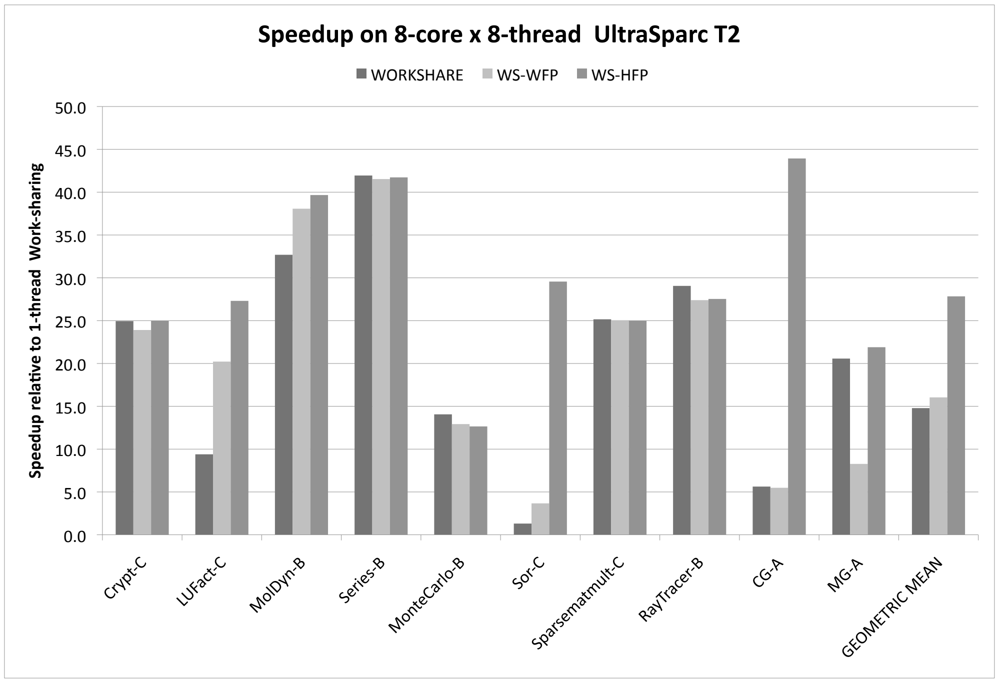

Figure 15. Comparison of work-sharing, HFP (Help-First Policy) and WFP (Work-First Policy) on a 64-thread UltraSparc T2 SMP. The execution times were obtained using "Best of 30 runs" approach.

deliver equal or better performance than the multithreaded Java version for these benchmarks. We observe that on average, work-stealing schedulers outperform the work-sharing scheduler and the help-first policy performs better than work-first policy for work-stealing. This is because the JGF and NPB benchmarks contain large amounts of loop-based parallelism. For the `SOR`, `CG`, and `LUFact` benchmarks, the help-first policy outperforms the work-first policy by a large margin. In summary, the performance results show significant improvements (up to $4.7\times$ on a 16-way Power5+SMP and $22.8\times$ on a 64-thread UltraSPARC II) for our work-stealing scheduler compared to the existing work-sharing scheduler for X10.

6 Related Work

The three programming languages developed as part of the DARPA HPCS program (Chapel, Fortress, X10) all identified dynamic lightweight task parallelism as one of the prerequisites for success. Dynamic task parallelism is also being included for mainstream use in many new programming models for multicore processors and shared-memory parallelism, such as Cilk, OpenMP 3.0, Java Concurrency Utilities, Intel Thread Building Blocks, and Microsoft Task Parallel Library. Our results on efficient and scalable implementation of terminally-strict async-finish task parallelism can easily be integrated into any of the programming models listed above.

Work-stealing schedulers have a long history that includes *lazy task creation* [14] and the theoretical and implementation results from the MIT Cilk project. Blumofe et al. defined the fully-strict computation model and proposed a randomized work stealing scheduler with provable time and space bounds [4]. An implementation of this algorithm with compiler support for Cilk was presented in [10]. In earlier work [1], we proved that terminally-strict parallel programs can be scheduled with a work-first policy so as to achieve the same time and space bounds as fully-strict programs. To the best of our knowledge, the work presented in this paper is the first work stealing implementation including compiler and runtime support for an async-finish parallel language which allows escaping asyncs and sequential calls to a parallel function[3].

The open source X10 v1.5 implementation [3] includes a work-sharing runtime scheduler based on the Java 5 Concurrency Utilities. This approach implements X10 activities as runnable tasks that are executed by a fixed number of Java threads in a `ThreadPoolExecutor`, compared to creating a separate Java thread for each X10 activity. As shown in our experimental results, the work-stealing based scheduler (for both work-first and help-first policies) presented in this paper performs better than this work-sharing implementation.

The X10 Work Stealing framework (XWS) is a recently released library [9] that supports help-first scheduling for a subset of X10 programs in which sequential and async calls to the same function and nesting of finish constructs are not permitted. The single-level-finish restriction leads to a control flow structure of alternating sequential and parallel regions as in OpenMP parallel regions, and enables the use of a simple and efficient global termination detection algorithm to implement each finish construct in the sequence. The library interface requires the user to provide code for saving and restoring local variables in the absence of compiler support. With these restrictions and an additional optimization for adaptive batching of tasks, the results in [9] show impressive speedups for solving large irregular graph problems. In contrast, our approach provides a language interface with compiler support for general nested finish-async parallelism, and our runtime system supports both work-first and help-first policies. In addition, we have implemented non-blocking *steal* operations for help-first scheduling that differs from the algorithm in [9]. An interesting direction for future research is to extend our compiler support to generate calls to the XWS library so as to enable performance comparisons for the same source code, and explore integration of the scheduling algorithms presented in this paper with the adaptive batching optimization from [9].

The Fork/Join framework [12] is a library-based approach for programmers to write divide-and-conquer programs. It uses a work-stealing scheduler that supports a help-first policy implemented using a variant of Cilk's THE protocol. Unlike the non-blocking steal operations in our approach, the thief needs to acquire a lock on the victim's deque when performing a steal in the Fork/Join framework.

7 Conclusions and Future Work

In this paper, we addressed the problem of efficient and scalable implementation of X10's async-finish task parallelism, which is more general than Cilk's spawn-sync parallelism. We introduced work-stealing schedulers with work-first and help-first policies for the async-finish task parallelism, and compared it with the work-sharing scheduler that was previously implemented for X10. Performance results on two different multicore SMP platforms show that the work-stealing scheduler with either policy performs better than the work-sharing scheduler. They also shed insight on scenarios in which work-stealing scheduler with work-first policy outperforms the one with help-first policy and vice-versa.

There are many interesting directions for future research. We see opportunities for optimizing the book-keeping code

[3]The recent Cilk++ release from Cilk Arts [8] allows the same function to be spawned and called sequentially from different contexts.

necessary for maintaining continuation frames. It is also important to extend the work stealing algorithm presented in this paper to be locality-conscious so that it can support the affinity directives embodied in X10 places [6], and also support broader classes of parallel programs with coordination and synchronization mechanisms that go beyond async-finish, such as phasers [18, 17].

Acknowledgments

This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

We would like to thank all X10 team members for their contributions to the X10 software used in this paper, and we gratefully acknowledge support from an IBM Open Collaborative Faculty Award. We would like to especially thank Vijay Saraswat and Igor Peshansky for discussions on the X10 Work Stealing framework (XWS), Bill Scherer and Rui Zhang for discussions on non-blocking algorithms. Finally, we would like to thank Doug Lea for access to the UltraSPARC T2 SMP system and the IBM Poughkeepsie Benchmark Center for access to the PowerPC SMP system, both of which were used to obtain the experimental results reported in this paper.

References

[1] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[2] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, Apr. 2005.

[3] R. Barik, V. Cave, C. Donawa, A. Kielstra, I. Peshansky, and V. Sarkar. Experiences with an smp implementation for X10 based on the java concurrency utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006, Sep 2006.*

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[6] 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 *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.

[8] Cilk Arts. *Cilk++ Programmer's Guide Version 1.0.2*.

[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 International Conference on Parallel Processing (ICPP'08)*, Sept. 2008.

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

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[12] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[13] D. Lea. Engineering Fine-Grained Parallelism in Java, November 2008. Presentation.

[14] E. Mohr, D. A. Kranz, and J. Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264280, 1991.

[15] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[16] J. Shirako, H. Kasahara, and V. Sarkar. Language extensions in support of compiler parallelization. In *The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, 2007.

[17] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. S. III. Phaser accumulators: a new reduction construct for dynamic parallelism. In *The 23rd IEEE International Parallel and Distributed Processing Symposium IPDPS'09 (to appear)*, May 2009.

[18] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point sync hronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomp uting*, pages 277–288, New York, NY, USA, 2008. ACM.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.