A Work-Stealing Scheduler for X10's Task Parallelism with Suspension

Olivier Tardieu
IBM T.J. Watson Research Center,
Yorktown Heights, NY, USA
tardieu@us.ibm.com

Haichuan Wang
University of Illinois at
Urbana-Champaign, Urbana, IL, USA
hwang154@illinois.edu

Haibo Lin
IBM Research - China, Beijing, China
linhb@cn.ibm.com

Abstract

The X10 programming language is intended to ease the programming of scalable concurrent and distributed applications. X10 augments a familiar imperative object-oriented programming model with constructs to support light-weight asynchronous tasks as well as execution across multiple address spaces. A crucial aspect of X10's runtime system is the scheduling of concurrent tasks. Work-stealing schedulers have been shown to efficiently load balance fine-grain divide-and-conquer task-parallel program on SMPs and multicores. But X10 is not limited to shared-memory fork-join parallelism. X10 permits tasks to suspend and synchronize by means of conditional atomic blocks and remote task invocations.

In this paper, we demonstrate that work-stealing scheduling principles are applicable to a rich programming language such as X10, achieving performance at scale without compromising expressivity, ease of use, or portability. We design and implement a portable work-stealing execution engine for X10. While this engine is biased toward the efficient execution of fork-join parallelism in shared memory, it handles the full X10 language, especially conditional atomic blocks and distribution.

We show that this engine improves the run time of a series of benchmark programs by several orders of magnitude when used in combination with the C++ backend compiler and runtime for X10. It achieves scaling comparable to state-of-the art work-stealing scheduler implementations—the Cilk++ compiler and the Java fork/join framework—despite the dramatic increase in generality.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.4 [*Processors*]: Code generation, Run-time environments

General Terms Languages, Performance

Keywords Scheduling, Task Parallelism, Work-Stealing, X10

1. Introduction

The X10 programming language [5, 6, 22] is intended to ease the programming of scalable concurrent and distributed applications, targeting modern multicore and clustered architectures. X10 augments a familiar imperative object-oriented programming model with constructs to support light-weight asynchronous tasks as well as execution across multiple address spaces. It is a strongly-typed and class-based language much like Java or Scala [19]. It supports two levels of concurrency. The first level corresponds to concurrency within a single shared-memory process, which is represented by an X10 place. The second level supports parallelism across places, i.e., across processes that do not share memory. In each place, X10 encourages programmers to decompose computations into loosely-synchronized light-weight asynchronous tasks—asyncs—with the promise that theses tasks will run in parallel on parallel hardware. Fulfilling this promise however is hard. How can the runtime system efficiently allocate tasks to parallel execution units? What if the tasks are too small or too many? What about ordering dependencies?

Work-stealing schedulers [4] have emerged as the approach of choice to tackle these issues. A work-stealing scheduler uses a pool of worker threads to run a task-parallel program. Each worker maintains a queue of pending jobs[1] and pushes new jobs to its own queue. When a worker completes a job, it pops a pending job from its own queue, or, if empty, attempts to steal a job from another worker's queue. Work-stealing queues are double-ended: workers push and pop from the bottom of the queue, but steal from the top. Work-stealing schedulers typically perform well because they minimize contention among workers.

Work-stealing schedulers have acquired their reputation in the context of several programming models (Cilk [9], Java fork/join [16], Habanero [20], PFunc [14], Intel Threading Building Blocks [21], Microsoft Task Parallel Library [17]). In order to make work-stealing effective, these models are very constrained, and programmers have to renounce a lot of the power and flexibility of modern programming languages. For instance, parallel scopes in Cilk do not extend beyond procedure boundaries: a procedure cannot return if it has outstanding children. Cilk also adopts a weak exception semantics. Habanero's work-stealing scheduler only handles async-finish task graphs. Library-based frameworks offer less flexible scheduling policies and lack compiler support to statically rule out unsupported task dependencies. Java fork/join tasks may only use synchronization classes that are advertised to cooperate with fork-join scheduling. These restrictions are deemed necessary to make work-stealing effective and its implementation tractable.

In contrast, X10 permits arbitrary task synchronizations, adopts a determinate exception semantics, and makes no connection between task and method boundaries. While it is unrealistic to expect work-stealing to be as effective on arbitrary programs, we would like Cilk-like performance for Cilk-like codes and, at the same time, full language support with limited and predictable overhead.

[1] The scheduler may divide each source-level task into multiple jobs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
Copyright © 2012 ACM 978-1-4503-1160-1/12/02. . . $10.00

This work tackles these challenges. We design and implement an efficient portable work-stealing execution engine for the full X10 language. Its contributions are:

- *Dynamic load balancing at each place.* The engine supports multi-place programs and balances the computation in each place, automatically mapping tasks to worker threads. We do not consider load balancing across places, that is, task migration from place to place.
- *Full language support.* The engine is designed to handle task suspension by means of compiler-generated continuations, exceptions, asynchronous initialization, etc.
- *Portability.* Because X10 is intended to be available on a wide variety of platforms ranging from Systems-on-a-Chip to supercomputers, portability is a prime concern. The engine combines a compiler plugin and a runtime library. The compiler plugin implements an X10-source-to-X10-source program transformation that generates code artifacts (frame classes and continuation methods) required by the runtime scheduler. The runtime library written in X10 implements the work-stealing scheduler. Thanks to this X10-centric approach, we can plug this engine into both the C++ and Java backend compilers and runtimes for X10 and use any C++ compiler or JDK supported by X10.
- *Performance (C++ backend).* We develop a series of optimizations to obtain a scalable scheduler with low overhead. Some of these optimizations—lazy frame initialization and migration—require extending the backend compilers for X10. We extend the C++ backend compiler accordingly.

We evaluate performance on a series of shared-memory fork-join benchmarks translated from Cilk (Problem-Based Benchmark Suite [2]) and Java fork/join.

We discuss related work in Section 2. We give a brief introduction to X10 in Section 3. Sections 4 to 6 describe the architecture of our work-stealing scheduler and its implementation. We report on our experimental evaluation in Section 7 and conclude in Section 8.

2. Related Work

Work-stealing runtimes are increasingly popular to handle the scheduling of dynamic task parallelism.

Languages and libraries. These runtimes are often directly exposed to the programmer as libraries. Java's fork/join framework [16], Intel's Threading Building Blocks [21], and Microsoft's Task Parallel Library [17] follow this approach. The XWS library for X10 [7] implements a work-stealing scheduler dedicated to graph algorithms. Tasks correspond to vertices; programs submit vertices to the scheduler; the scheduler dynamically partition vertices across processing units to balance the load using a work-stealing policy.

Other work-stealing runtimes hide inside implementations of new language constructs and require matching compiler support to transparently map constructs to runtime routines. The Cilk-5 runtime [9] and the Habanero runtime [11] belong this category. Thanks to its built-in constructs for fine-grain concurrency, X10 is an ideal candidate for the latter approach, which we adopt here.

Shared beliefs. These many runtimes share common principles. A work-stealing scheduler uses a pool of threads (OS threads or VM threads) called workers. Each worker maintains a double-ended queue—a deque—of pending things to do. A worker primarily operates on its own deque, pushing content to the deque when a task is spawned and going back to the deque every time it finishes its current task. The deque is also the mechanism by which work is made available to other workers: if a worker empties its own deque, it then attempts to steal work from the deque of another worker. Usually, workers push and pop work from the bottom of their deque, but steal from the top, further reducing contention.

Work-stealing deques. X10's standard library provides a `Deque` class that is essentially a replica of the deque of Java's fork/join framework [15, 16]. Our engine is implemented using this deque.

Scheduling policies. Work-stealing algorithms first differ in what they push to the deque when a task is spawned. Under the work-first policy promoted by Cilk, the worker pushes the continuation of the parent task to the deque, executing the spawned task first. Under the help-first policy typical of library-based schedulers, the worker pushes the spawned task to its deque, while continuing the execution of the parent task. Previous research has looked into the pros and cons of these scheduling policies and how to combine them [4, 7, 11, 12, 18]. In this work, we implement a pure work-first scheduling policy. Our goal is to enlarge the class of languages and programs amenable to work-stealing rather than improving performance for a specific subclass. With work, our scheduler could be made more flexible using the recipes developed for Habanero.

Fork-join schedulers. Library-based schedulers are primarily targeted at fork-join parallelism. Fork-join tasks may only suspend to wait for subtasks to complete. As a consequence, when a task suspends, a fork-join scheduler can safely assign to the same worker a subtask of the suspended task, without bothering with context switches or continuations. The state of the suspended parent task simply remains on the thread stack underneath the state of the subtask. Obviously the parent task cannot be returned to until after the subtasks have completed. But this is just fine for fork-join tasks.

The standard X10 runtime, a.k.a XRX, has to support arbitrary suspension. It therefore adopts an hybrid approach: when a task suspends on a finish construct (waiting for subtasks) the worker starts running subtasks, but if a task suspends on a when construct (conditional atomic block) the scheduler suspends the worker thread and allocates or wakes another thread in its pool to compensate for the decrease in parallelism.

Among others, Java's fork/join framework goes beyond pure fork-join tasks. It ships with a few compatible synchronization classes. PFunc permits synchronization barriers as long as the number of tasks involved is less than the number of workers. By nature, these frameworks cannot handle arbitrary synchronization patterns without reverting to a one-to-one mapping from tasks to threads.

Scheduling with continuations. In contrast, compilers such as the Cilk compiler, the Habanero compiler, and our augmented X10 compiler can synthesize code artifacts that makes it possible for a runtime to queue a continuation for later execution without sacrificing a thread. As a consequence, our runtime uses a fixed number of threads (intended to match the number of available cores) and a unified approach, where continuations are used to implement both finish and when. Incidentally, this means our runtime makes it possible to run X10 programs on computer systems without support for dynamic thread creation, such as IBM's BlueGene/P, whereas XRX can only run async-finish programs on such architectures.

The drawback of this approach is known: compiler-generated continuations have overheads (code size, run time). Carefully engineered fork-join schedulers are capable of lower overhead. But Cilk has already established that aggressive compiler and runtime optimizations can make the approach successful. In this work, we cannot be as aggressive in our optimizations as the Cilk++ compiler as we intend to interface with a wide variety of backend compilers and runtimes for X10. We do however take advantage of C++-backend-only features and we suspect that a deeper integration with the C++ compiler itself would enable further reduction of our overhead.

Beyond shared-memory fork-join parallelism. Our compiler plugin is inspired by the work on compiler support for work-first

work-stealing in the Habanero project [20]. Habanero handles a larger class of task graphs than Cilk—async-finish graphs—by permitting arbitrary nesting of finish and async constructs. Various scheduling policies have been proposed to handle several classes of task graphs with more flexible synchronization constructs, e.g., synchronization variables [3] or futures [24]. Our work takes compiler support much further by also permitting conditional atomic blocks and distributed code. Thanks to the former, we can handle any kind of synchronization: cyclic barriers, futures, FIFOs, etc.

We handle distributed programs but only provide dynamic mapping from tasks to workers in each place. X10 requires programs to specify the place of each task. The X10 runtime is not permitted to migrate tasks across places. It is however possible for an application or library to interface with the runtime so as to dynamically choose where to spawn tasks in an attempt to balance the load across places [23]. These two levels of load balancing could be combined.

3. The X10 Language

This section briefly describes the context for the X10 project and introduces the key programming language concepts that will be discussed in later sections of the paper. This work is done in the context of the most recent revision of the X10 language: X10 2.2.

The genesis of the X10 project was the DARPA High Productivity Computing Systems (HPCS) program. As such, X10 is intended to be a programming language that achieves "Performance and Productivity at Scale." The primary hardware platforms being targeted by the language are clusters of multicore processors linked together into a large scale system via a high-performance network. Therefore, supporting both concurrency and distribution are first class concerns of the X10 language design and implementation.

X10 is a familiar strongly-typed, imperative, class-based, object-oriented programming language much like Java or Scala. Like functional languages, X10 supports first-class functions and encourages using immutable state. X10 emphasizes statically-checked guarantees by means of a rich type system with generics, constraints (i.e., dependent types), structs, and type definitions.

A computation in X10 consists of one or more asynchronous activities (light-weight tasks). A new activity is created by the statement `async S`. To synchronize activities, X10 provides the statement `finish S`. An activity that executes a finish statement will not execute the statement after the finish until all activities spawned within the finish's body have terminated.

Every activity executes in a single `Place` (address space). While executing in this place, it may freely access any object that also resides in the place. It may manipulate remote references (`GlobalRef`s) to objects that reside in other places, but is not able to actually access the state of any remote object. Therefore computations must sometimes "shift" from one place to another to access the data they need. When this happens, the compiler and runtime system collaborate to ensure that the necessary data and control information are communicated from one place to another. The fundamental X10 construct for "place-shifting" is `at (p) S`. An at statement shifts execution of the current activity from the current place to place p and executes S at the remote place. For instance, the program below prints one message from each place.

```
class HelloWorld {
  public static def main(Array[String]) {
    finish for(p in Place.places())
      async at(p) Console.OUT.println("Hello from place " + p);
  }
}
```

The set of available places (`Place.places()`) and the mapping from places to nodes in a cluster is decided by the user at launch time.

X10 includes an unconditional atomic block construct `atomic S` and a conditional atomic block construct `when (E) S`. An atomic block is executed by an activity as if in a single step during which all other concurrent activities in the same place are suspended. Execution of `when (E) S` suspends until a state is reached in which the condition E is true. In this state, the statement S is executed atomically.

X10 defines a "rooted" exception model in which a finish acts as a collection point for any exceptions thrown by activities that are executing under the control of the finish. Only after all such activities have terminated (normally or abnormally) does the finish propagate exceptions to its enclosing environment by collecting them into a single `MultipleException` object.

A great deal more information on X10 can be found online at `http://x10-lang.org`. In particular, the language specification [22], programmer's guide [5], and a collection of tutorials and sample programs are available.

4. A Scheduler for Single-place Async-Finish X10

We organize the discussion of our work-stealing scheduler into three sections. We start with an unoptimized scheduler for single-place async-finish programs in the current section, discuss optimizations in the next, and add full language support in Section 6. An async-finish program only uses async and finish to spawn and synchronize tasks (as opposed to at and when).

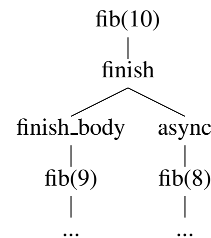
4.1 Principles

Using a simplified divide-and-conquer Fibonacci example method, we first explain informally how our scheduler works. In the second half if this section, we take a closer look at the required compiler and runtime support and show snippets of the generated code and runtime code.

```
static def fib(n:Int):void {
  if (n<=1) return;
  finish {
    async fib(n-2);
    fib(n-1);
  }
}
```

For simplicity, this method does not return anything (see Section 6.4 for the real thing).

Frames and cactus stacks. We decompose each method into a series of scopes and associate a *frame* with each scope. This example method has four scopes for (1) the method itself, (2) the finish construct, (3) the finish body, (4) the async construct. The work-stealing scheduler builds the *cactus stack* of the running program by chaining these frames into trees, such as:



Observe the finish frame has two children. When the execution enters the method the "fib(n)" frame is pushed to the top of the stack. Assuming n is greater than 1, it then reaches the finish construct. The corresponding frame is pushed to the stack shortly followed by the finish_body frame. It now reaches the async construct. An async frame however is not pushed to the top of the stack but to the top of the topmost finish on the stack *next* to any other child frame the finish frame might already have.

This cactus stack captures the current state of the execution and is shared and maintained by all the worker threads collaborating to execute the program. Each worker owns one top of the stack, i.e., one leaf of the tree. For instance, the worker running the finish_body frame will eventually push frame "fib(n-1)" on top of it; the worker handling the async frame will push frame "fib(n-2)" on top of it. This may be the same worker at a different point in time or a distinct worker.

The frames are regular X10 objects, that is, instances of X10 classes. They are created by means of constructor invocations. They are linked together just like object graphs are usually constructed. Each frame object is an instance of a subclass of the root `Frame` class. The `Frame` class has an `up` field of type `Frame` which holds a reference to the parent frame in the cactus stack. This reference is `null` for the root stack frame.

The cactus stack is intended to fully encapsulate the state of the execution. Each frame class has a field for each local variable declared in the corresponding scope. In addition, a method frame has a field for each parameter of the method. Most frames also have a `pc` field of integer type, which encodes the next position in the frame, that is, the index of the instruction in sequence after the one currently processed.

In summary, the state of the execution is encoded as an explicit cactus stack. The cactus stack preserves the caller-callee relationship, keeps track of return addresses (saved pc), and store the values of the variables.

In the remainder of this paper, when ambiguous, we use the term *cactus stack* to refer to this data structure and the term *thread stack* to refer to the pthread- or VM-level stack of a worker thread.

Deques and continuations. Our scheduler implements a work-first policy. Each worker maintains a deque of pending continuations. A continuation is simply a frame with valid pc, which specifies the point of reentry.

When a worker is about to enter an async it pushes the continuation of the async onto its deque. Concretely, it saves the index of the statement in sequence after the async in the current frame's pc field and pushes a reference to that frame onto its deque. In the previous example, just before entering the async scope, the worker stores the index of the "fib(n-1)" method invocation in the finish_body frame and pushes it onto the deque.

The worker then processes the async. When done with the async, it attempts to pop the continuation from the deque. If successful, the continuation has not been stolen and the execution continues. Another worker however might have stolen the continuation in the meantime and taken charge of its execution. In that case, the victim is left with nothing to do and in turn becomes a thief.

In summary, the leaves of the cactus stack are dynamically split into two sets. Some leaves correspond to frames that workers are currently processing (one per worker). The other leaves constitute the elements of the worker deques, that is, the pending continuations (not currently processed by any worker).

Scheduling. The first worker gets to execute the application main method. Other workers are idle. An idle worker attempts to steal a continuation from a random worker again and again until it finds something to run.

When a worker pops a completed frame to find a finish frame below, it checks how many child frames still point back to this finish frame. If none, it means all the tasks governed by this finish have completed, therefore the worker pops the finish frame and continues with its parent frame. Otherwise, the worker just abandons the current computation. This worker or another will eventually consider the finish frame again when another branch of the finish subtree gets completed.

4.2 Implementation

We implement this scheduling policy by rewriting an X10 program with finish and async constructs into a new X10 program that only uses a few long-lived asyncs. Each one of these asyncs is a worker of our work-stealing scheduler. They all run the same top-level loop, alternating between finding a continuation to run, and running this continuation. As part of their executions, these worker asyncs build the cactus stack of the program and operate the deques.

We run the transformed program using the standard X10 runtime. It maps each worker async to its own runtime thread for the duration of the execution.

Of course there is no point in implementing a standalone program rewriting engine from scratch given the existing compiler infrastructure for X10—parser, type checker, Abstract Syntax Tree (AST) representation and traversal facilities, pretty-printer, etc. We implement a typed-AST-to-typed-AST transformation in the style of many intermediate passes of the X10 compiler and add a new `-WORK_STEALING` option to the X10 compiler that enables the transformation.

Moreover, part of the code we need to generate is independent of the particulars of one specific X10 program. We therefore write it once and for all as a runtime library.

4.2.1 Runtime Support

We add a package to the X10 runtime named `x10.compiler.ws`. This package contains a series of frame classes and a `Worker` class.

The frame classes all inherit from a root `Frame` class and are intended to declare and/or implement the mechanisms by which frame objects can be managed by the scheduler to compose cactus stacks and encode continuations. The `Frame` class declares a field `up` of type `Frame` intended to hold a reference to the frame above the current frame object in the stack. In addition to the `Frame` class, we declare the `FinishFrame`, `AsyncFrame`, and `RegularFrame` classes. `FinishFrame`s are the only frames with potentially multiple children in the cactus stack. They maintain a count of these children. `RegularFrame`s may be pushed to the deques. The parent frames of `AsyncFrame`s are `FinishFrame`s.

The `Worker` class implements the worker of the work-stealing scheduler. Each worker alternate between the idle state—looking for a continuation to steal—and the running state—executing application code. Moreover, static methods of the `Worker` class handle the creation and destruction of the pool of workers.

Each worker more or less runs this loop:

```
var k:Frame;
while ((k = findContinuation()) != null) {
  try {
    while (k != null) {
      k._run(k.pc);
      k = k.up;
      if (k != null && k instanceof FinishFrame) {
          val f = k as FinishFrame;
          f.decreaseChildrenCount();
          if (f.hasOustandingChildren()) break;
    }
  } catch (Stolen) {}
}
```

When an idle worker finds a continuation k, it invokes k, that is, resumes the execution of the stolen frame at the saved pc. If the frame's execution completes, the worker pops the frame from the cactus stack and proceeds with the parent frame. If the parent frame is a finish frame, the execution continues up the stack only if the finish frame has no outstanding children.

The findContinuation method implements the usual infinite loop: pseudo-random selection of a victim, attempt to steal from the victim's deque, break if successful, continue if not.

The pop method invocation in async frames may either succeed or throw the STOLEN exception (unique instance of the Stolen exception type). In the latter, the computation of the worker is cancelled and the worker returns to idle state.

4.2.2 Compiler Support

The compiler generates the representation of the frames of the cactus stack and instruments the source code to (1) operate on the cactus stack and (2) push and pop continuations to the deques.

Our compiler pass divides methods into scopes and synthesizes a frame class for each scope. It identifies the local variables and method parameters in the program and add fields to frames accordingly. It indexes the statements in each scope. It splits methods by generating one helper method in each frame, which implements the code fragment of the corresponding scope. It synthesizes a replacement method for the original method, which constructs an instance of the method frame class and invokes the generated helper method of that frame.

The compiler also instruments the code of the helpers methods as the following:

- It replaces accesses to local variables and method parameters with field accesses.
- Before each method call, it saves the pc of the next instruction in the current frame. The method call itself is rewritten into an invocation of the corresponding replacement method.
- It replaces each nested scope of the method in the same way. First, the pc of the next instruction is saved in the current frame. Then, a frame object for the scope is constructed. Finally, the helper method of the new frame is invoked.
- It inserts push-to-the-deque commands just before entering async scopes, pop-from-the-deque commands just before leaving async scopes.
- It adds to each helper method a top-level switch statement, which permits entering the method in the middle so as to resume its execution at a chosen instruction index.

Finally, the compiler generates the main method of the transformed program. This method initializes the worker pool then invokes the transformed application main method. Once the application main method completes, it destructs the worker pool and exits.

To illustrate these tasks, here is a skeleton of the code we generate for the example fib method. For brevity, we only show the pc-related code in the `_fib_finish_body` class.

```
class _fib extends RegularFrame {
  val n:Int;
  def this(up:Frame, n:Int) {super(up); this.n=n;}
  def _run(pc:Int) {
    if (this.n<=1) return;
    val _frame1 = new _fib_finish(this);
    _frame1._run(0);
  }
}
class _fib_finish extends FinishFrame {
  def this(up:Frame) {super(up);}
  def _run(pc:Int) {
    val _frame1 = new _fib_finish_body(this);
    _frame1._run(0);
  }
}
class _fib_finish_body extends RegularFrame {
  def this(up:Frame) {super(up);}
  def _run(pc:Int) {
    switch(pc) {
    case 0:
      this.pc = 1;
      deque().push(this);
      val _frame1 = new _fib_async(this.up);
      _frame1._run(0);
    case 1:
      this.pc = 2;
      _fib(this, this.up.up.n-2);
    }
  }
}
class _fib_async extends AsyncFrame {
  def this(up:FinishFrame) {super(up);}
  def _run(pc:Int) {
    _fib(this, this.up.up.n-1);
    deque().pop();
  }
}
static def _fib(up:Frame, n:Int):void {
  new _fib(up, n)._run(0);
}
```

There are four generated classes corresponding to the four identified scopes, four _run methods corresponding to the fragment of code in each scope, plus a _fib method to replace the original fib method.

We now review the main additional tasks undertaken by the compiler not illustrated by our naive example.

Class hierarchy. A replacement method always takes the callee's frame as its first parameter. The rest of its signature is the same as the original method. Therefore visibility, overriding, and overloading are not affected by the transformation. We rewrite the signatures of abstract and interface method in the same manner.

Instance methods. The frames classes obtained from a method of class C are generated inside class C. Occurrences of *this* inside instance methods are replaced by qualified C.*this* expressions.

Control-flow constructs. We handle loops and conditionals by dividing the code into more scopes. For example, we create a frame class for each branch of an `if` statement. When a worker encounters a `return`, `break`, or `continue` statement, it starts popping and discarding stack frames until it finds the target frame for the construct.

Return values. The frame of a non-void method has a field intended to hold the returned value. Upon return, the caller obtains the value from the callee's frame just before discarding the reference to that frame.

5. An Optimized Scheduler

The scheduler of the previous section would not perform very well. In particular, it would not scale beyond a few workers and it would have high sequential overhead. In other word, transformed sequential code would be much slower than the original. While some of the overhead is unavoidable, many reasons for this poor performance can be addressed.

In this section, we discuss performance improvements. These improvements are compatible with the extended language support we discuss in the next section. More precisely, the extensions of the next section are engineered to be compatible with these optimization and to only marginally affect their effectiveness on single-place async-finish programs.

The main issues we are trying to address are:

- The scheduler constantly allocates, constructs, and collects frame objects.
- The generated methods are too many and too tiny. Too much time is spent transferring from callers to callees and back.
- Large chunks of code are sequential but the compiler transforms them anyway.
- Steals are infrequent. Most of time a task and all of its subtasks get processed by the same worker one after the other. We should be taking advantage of this.

5.1 Selective Transformation

Before applying our program transformation, we build the call graph of the target program. A method m is transformed by our compiler pass iff m or any method reachable from m contains a concurrency construct (async, at, when). Otherwise, it is left unchanged.

We say a local variable is *ephemeral* if no task may be spawned while it is alive, that is, if there is no async, at, or when construct between a definition and a use of the variable. There is no need to store ephemeral variable values in the cactus stack. Our compiler pass leave them alone. For now, we work around the lack of live variable analysis in the X10 tool chain by manually annotating variables `@x10.compiler.Ephemeral`.

5.2 Lazy Frame Initialization

X10 local variables and fields have different initialization semantics. A local variable can only be read after it has been definitely initialized. A field on the other hand is initialized to the default value of its type if not explicitly initialized in a constructor. Therefore, by substituting fields for local variables, we incur the overhead of field initialization to no avail since this default value will never be accessed. To eliminate this overhead, we annotate the fields of the frame classes `@x10.compiler.Uninitialized` and teach the C++ backend compiler not to zero annotated fields.

5.3 Inlining

We systematically inline frame constructors at call sites. In order to avoid a method call to lookup the current deque for push and pop operations, we add a worker parameter to every generated method and use that reference to access the deque. The push and pop codes themselves are inlined. We also inline helper methods (i.e., _run methods) into transformed methods (e.g., _fib method). Moreover, we eliminate the switch statements from the transformed methods since the helper methods are always invoked with pc equals 0.

Ultimately, we end up with two copies of each source method in the transformed program. The first copy, traditionally named *fast clone* is the transformed method itself. Thanks to inlining, each transformed method ends up having the same structure as the corresponding source method. The second copy of the code—the *slow clone*—consists of the collection of helper methods in frame classes. The slow clone is seldom used, that is, by thieves to resume the execution of an existing call stack. Workers spend most of their time running fast clones, so the small size of slow clone methods does not hurt performance.

5.4 Finish Out-degree

By construction, the worker who creates a finish frame must process all of the non-stolen subtasks of this finish before returning to the finish frame. For simplicity, in the previous section, we suggested keeping track of how many frames are pointing to a finish frame. But this is an overkill. Instead, we count how many subtasks of the finish have been stolen so far and not yet completed, adding one to that count if one or more of the non-stolen subtasks has not completed yet. In other words, we aggregate all the non-stolen subtasks as one for counting purposes.

While this might seem like a minor change, the performance implications are profound. Indeed, this count is shared across workers. Therefore, it must be updated atomically. Thanks to this change, these atomic operations are only required from thieves and victims. In particular, if a single worker executes a finish body in its entirety, then no counting ever takes place.

5.5 Speculative Stack Allocation

Most often, a frame will be constructed and discarded by the same worker without ever being seen by another worker. Therefore, we speculatively allocate frames on the thread stack. A thief is responsible for copying the stolen continuation frame as well as all the frames on top of it from the victim's stack to the heap (except of course for those frames that have already been copied to the heap by a prior thief).[2]

Thieves and victims must synchronize to ensure the integrity of the copy. A thief grabs a lock (tryLock) on the victim's deque before attempting to steal. If the lock is acquired and the steal is successful, it keeps the lock while copying the frames. This prevents another thief from interfering with the copy. The victim when it fails to pop the stolen continuation acquires and releases the lock, hence ensuring that the copy is complete before trashing its stack.

While making the copy, the thief inserts a pointer in each source finish frame to the destination finish frame. This pointer is used by the victim to decrement the count of remaining tasks under the finish after it discovers that the continuation has been stolen.

Heap allocated frames objects are explicitly deallocated when popped from the cactus stack (using C++ free).

In the end, although we replaced locals by fields the data remains on the thread stack. We incur hardly any memory management cost for frames.

5.6 Miscellaneous Field Optimizations

In order to avoid the need for traversing the stack to identify the finish frame which to attach an async frame, we add a finish frame field to each `RegularFrame`. We only add a `pc` field to frames with multiple points of reentry.

Discussion. Our choice of creating many small frame objects at first might seem a fatal mistake. However, thanks to these careful optimizations, we alleviate most of the overhead while retaining the benefit of a straightforward code generation. Even path expressions such as "this.up.up.n" in the Fibonacci example could be eliminated from the fast clones by a trivial constant propagation.

Several of the optimizations we implement deviate from X10's official semantics, thus requiring changes in the C++ backend compiler for X10. While uninitialized fields or stack-allocated objects have no place in the Java specification, such things could be achieved in Jikes RVM [1] with essentially the same benefits, thanks to its stack-walking API [8].

6. A Scheduler for the Full X10 Language

In this section, we augment the scheduler of the previous section to account for the X10's conditional atomic blocks, remote tasks, exceptions, and asynchronous variable initialization. We also discuss the current limitations of our implementation.

6.1 Suspension

The execution of `when (E) S` suspends until a state is reached in which the condition E is true. In this state, the statement S is executed atomically. Moreover, changes to E outside of atomic sections might not trigger the execution of S.

In order to cope with when constructs, our scheduler maintains in each place a queue of suspended continuations. Concretely, when a worker encounters a when statement whose condition evaluates to false, it saves the index of the when statement in the pc field of the current frame and pushes this frame to the extra queue. It then aborts its computation and switches to the idle state in which it will look for something else to run in the usual way. Dually, when a worker exits an atomic block, it grabs a lock on the extra queue and

[2] To be exact, a thief always copies the stolen frames up to the first finish frame, then only copies those frame that are not on the heap yet.

moves its entire content to its own deque, which will eventually lead to the re-evaluation of the condition of each suspended task.

Speculative stack allocation. In an async-finish program, a worker only aborts its computation if it finds the continuation it is about to execute has been stolen from it. By construction, this means all the frames on the thread stack have already been copied to the heap by one or several thieves. It is therefore ok for the victim to discard the content of its thread stack by means of the STOLEN exception.

In contrast, when a worker aborts because of a false condition in a when statement, its deque may be full of pending continuations and its thread stack filled with frames essential to continued execution of the program. Hence, we need to take extra care to make the scheme for when constructs work with speculative stack allocation.

We add an extra deque per worker. Both the old and the new deques contain continuations. But while the continuations of the old deque may point to stack-allocated frames, continuations of the new deque can only refer to heap-allocated frames. Before a worker aborts because of a false when condition, it "steals" all of the tasks from its normal deque and pushes them to its extra deque using the exact same protocol thieves use to steal tasks. Then it can safely abort since there remain no pointer to its thread stack.

We update the findContinuation algorithm accordingly. When a worker is idle it first processes the continuations of its extra deque if any before trying to steal from others. When worker A decides to steal from worker B it first accesses the extra deque of B then if empty its normal deque. Of course, if it acquires a continuation from the extra deque, it does not need to migrate frames to the heap since the stolen frames are already there.

6.2 Distribution

The execution of `at (p) S` suspends the execution of the caller while S is executed at place p.

When a worker encounters `at (p) S`, it instantiates a special at frame. Then it migrates the content of its normal deque to its extra deque in the manner of the previous section and sends the at frame to place p.[3] Finally it aborts. At this point, the parent frame of the at frame only exists as a remote reference in the up field of the at frame at place p.

We augment findContinuation so that idle workers not only try to steal from collocated workers but also listen for incoming continuations from other places.

When a worker at place p picks up the at frame, it starts executing S. When done with S, it sends back the reference to the parent of the at frame to the place of origin. There a worker can pick up the frame and continue its execution.

Remote finish. One problem with that scheme is that asyncs spawned by S at place p may need to update finish subtask counts at the place of origin. In order to avoid a stream of costly increment and decrement messages between the two places, we create a proxy finish object in place p in the manner of the standard X10 runtime, which effectively coalesce these messages into much fewer remote updates.

Async at. While `async at (p) S` is truly the combination of an async and an at construct, the pattern deserves a dedicated implementation. Our scheduler simply creates a frame for S and sends it to place p.

We treat this frame S as a stolen continuation. Indeed there are now two workers at least—one local, one remote—working concurrently in the finish scope containing this async. Therefore, the worker who encountered the async statement in the first place aborts and immediately resumes the continuation of the async, which happens to be the first thing on its deque. In other words it stops executing the fast clone of the code and starts executing the slow clone of the same code. This ensures that the actions of the local and remote workers will be properly synchronized.

[3] For lack of space, we cannot discuss here the detail of the serialization operations.

6.3 Exceptions

Exceptions require ubiquitous changes (Frame classes, Worker class, codegen). Basically, we add to the state of each worker an exception field. When an exception is thrown by the user code, it gets stored in this field. While the field value is not null, the worker keeps popping frames from the cactus stack until in reaches either a try-catch frame, an async frame, or a finish frame. If the exception is caught by the try-catch frame, the field is cleared. Exceptions are accumulated in a list field of the finish frame. When an exception reaches a finish frame it is added to the list. If it reaches an async frame it is added to the list of the parent finish frame of the async frame. In both cases, the exception field of the worker is cleared.

When a finish frame has no outstanding children, the exceptions of the list are combined into a MultipleException object which is loaded in the worker exception field.

Unfortunately, the scheduler itself makes use of the STOLEN exception. This exception must not be affected by the scheme we just described. Moreover, the STOLEN exception should not trigger the execution of finally blocks. After our work-stealing compiler pass, we schedule an additional compiler pass to rewrite all try-catch-finally blocks in the generated code to "ignore" this exception.

6.4 Asynchronous Initialization

The X10 compiler extends a Java-like definite assignment analysis to permit initializing final variables (X10's *val*s) from asynchronous tasks. For instance, here is how to actually compute Fibonacci numbers using this feature:

```
static def fib(n:Int):Int {
  if (n<=1) return 1;
  val u1:Int; val u2:Int;
  finish {
    async u2 = fib(n-2); // async init of u2
    u1 = fib(n-1);
  }
  return u1 + u2;
}
```

Thanks to this language feature, asynchronous tasks can "return" values without going through expensive heap-allocated objects. Moreover, the compiler guarantees race freedom.

Asynchronous initialization makes it possible for a worker to write into an inner frame of the cactus stack, possibly shared among multiple workers. This is not a concern per se as the compiler guarantees that reads happen after writes and that writes are unique. But this becomes an issue in the context of speculative stack allocation of frames. In short, a worker may write to a stale frame that has been migrated to the heap by a thief. We could of course prevent this but only by means of costly synchronization that would hinder performance irrespective of steals.

Instead, we let a worker update stack frames (not knowing whether these have been stolen or not), but migrate the values to the replacement frames when the worker finally recognizes it has been mugged. Because of the definite assignment analysis, the compiler knows statically which task is initializing which final variables. We can therefore generate as part of our code transformation a method in each async frame to propagate the right set of values. If the continuation of the async frame is stolen, the worker invokes this method before aborting. For instance, in the Fibonacci example, if the continuation of the async is stolen we know statically that u2's value must be propagated.

6.5 Current Limitations

While our scheduler is designed from the ground up to support the full X10 language, its current implementations is not 100% complete yet and suffers a few temporary limitations, primarily:

- Closure literals and constructor bodies must be sequential. Concurrent instances may be replaced by anonymous classes and factory methods.
- X10's call graph construction is buggy. As a result, our compiler pass may incorrectly assume a method is sequential and fail to process it. This can be worked around by annotating missed methods with `@x10.compiler.WS`.
- Clocks are a form of distributed cyclic barriers with the convenience of a dedicated syntax. Our scheduler supports cyclic barriers but cannot handle the clock syntax yet.

7. Experimental Results

In this section, we evaluate the performance of our scheduler by comparing against the Cilk++ runtime and the Java fork/join framework when applied to single-place async-finish programs. While these runtimes have some support for limited forms of synchronization beyond fork-join, they are incapable of the equivalent of handling X10's when and at constructs.

Our compiler and runtime extensions are distributed as part of the X10 distribution [25] under the Eclipse license. Benchmark codes are available upon request.

7.1 Benchmarks

We consider two sets of benchmarks. The first one is composed with three micro benchmarks, including *Fibonacci*, *Integrate*, and *QuickSort*. These benchmarks have small to absurdly small tasks, making it possible to measure overheads at their worst. *Fibonacci* computes Fibonacci numbers as seen in Section 6.4. *Integrate* was suggested by Doug Lea. It computes the numerical integration of a polynomial function of degree 3 using Gaussian quadrature. *QuickSort* is a parallel quick sort implementation of a randomly generated data set. To be fair, we use the same random input generator across all implementations. We implement these three benchmarks in X10, Cilk++, and Java using the fork/join framework.

The second set of benchmarks is derived from the Problem-Based Benchmark Suite [2], which is a collection of 19 fine-grain task-parallel graph algorithms implemented in Cilk++. We translated 6 of them (randomly selected) to X10. We have no Java fork/join implementation of these benchmarks.

7.2 Porting Methodology

Our X10, Cilk++, and Java fork/join micro benchmark implementations follow the style of the respective programming paradigms. Neither implementation tries to aggregate tasks either statically or dynamically. Our goal with these benchmarks is to measure the overhead of each runtime. If we were to for instance dynamically bound the number of spawned tasks, then all of the compared runtimes would have close to zero overhead and close to perfect scaling, which is not very interesting.

Our translation of the PBBS benchmarks preserves the original algorithms as much as possible: we want to spawn the same tasks, with the same granularity and the same dependencies. Here is a list of the non-trivial changes to the code:

- PBBS codes make ubiquitous use of type-unsafe idioms: unchecked arrays, pointer arithmetic, and function pointers. We replace pointer arithmetics with explicit offsets. We disable array bound checks via the `-NO_CHECKS` option of the X10 compiler. We replace function pointers with closures. We mitigate the higher cost of closure invocation in X10 by hoisting interface lookups out of critical loops.[4]
- PBBS codes are memory-intensive. We turn off the garbage collector of the X10 runtime and preserve the free calls of the original code. Otherwise, execution time would be dominated by GC pauses.
- PBBS codes use compare-and-swap instructions on array cells. We extend the X10 standard library to support these.
- X10 structs are immutable. PBBS codes use mutable C++ structs. We replace them with classes.
- PBBS codes use the *cilk_for* construct. The Cilk runtime converts a cilk_for loop into an efficient divide-and-conquer recursive traversal over the loop iterations. We extend our code transformation to generate helper methods to do the same.

7.3 Evaluation Environment and Methodology

We measure performance on a 16-way x86_64 blade with four AMD Quad-Core Opteron 8347 HE at 1.9GHz, 24GB of RAM, and running RHELS 5.6.

Cilk++ configuration. We use Intel's cilk++ SDK preview (build 8503) [13]. We compile with `-O2 -finline-functions` optimizations, which is what the C++ backend compiler for X10 is invoked with.

Java fork/join configuration. We use the Java fork/join library from the Concurrency jsr-166 interest site [15] atop an Oracle(Sun) Java SE build 1.6.0_22-b04 32bit VM.

X10 configuration. Our experiments are based on the X10 open-source distribution at revision 22646 [25]. Our C++ compiler is g++ (GCC) 4.4.4 (Red Hat 4.4.4-13). We compile the X10 runtime with flags `-DOPTIMIZE=true -DNO_CHECKS=true -DDISABLE_GC=true`. We compile the benchmarks with options `-WORK_STEALING -O -NO_CHECKS`. We link the generated code with the thread-caching malloc memory allocator of the google performance tools [10].

Methodology. We measure the run time of the benchmarks using 1 to 16 workers. We also measure the sequential performance of selected benchmarks by removing all concurrency related constructs in these codes.

All the performance data for X10 and Cilk++ were collected from the average of 10 runs. We do 5 warm-up iterations for Java fork/join benchmarks and report the average of the next 10 runs.

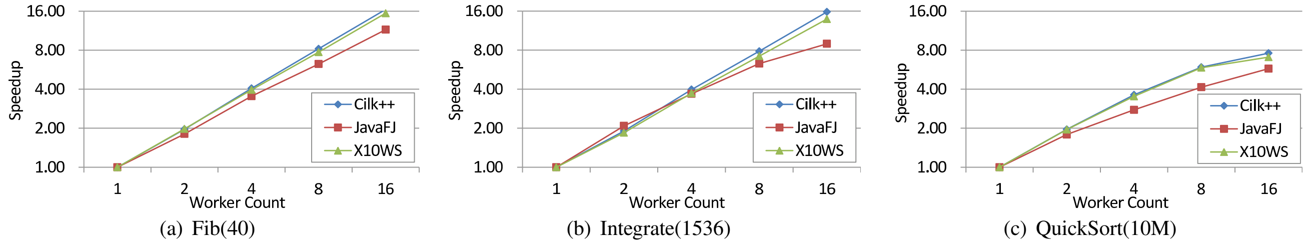
7.4 Micro Benchmark

Figure 1 shows the speedups for each micro benchmark when increasing the number of workers. Execution time is normalized to 1 for 1 worker. We can see X10's work-stealing scheduler scales nearly as well as Cilk++ and better than Java fork/join. Integrate has the largest spread. Cilk++ speedup is 15.8, X10 is 13.9 and Java fork/join only 9.0.

Figure 2 shows the run times for all benchmarks and all implementations with 16 workers. X10 is faster. We also ran the X10 codes using the unmodified X10 tool chain and 16 runtime threads. Compared to X10 with work-stealing, Fib(40) is 700x slower, Integrate(1536) is 300x slower, and QuickSort(10M) is 60x slower.

In Figure 3 we compare sequential overheads, that is, the execution time of the sequential version of each program with the execution time of the parallel version running with a single worker. X10 achieves the best result with overheads ranging from 7x to 28%.

[4] Function types in X10 are interface types. Invoking a closure literal requires a lookup of the function type in the interface table of the closure litteral, which has non-trivial cost.



(a) Fib(40) (b) Integrate(1536) (c) QuickSort(10M)

Figure 1. Micro Benchmark Speedups

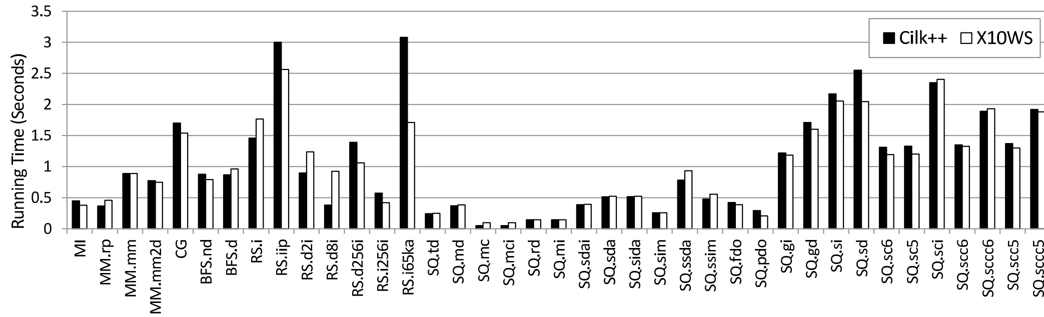
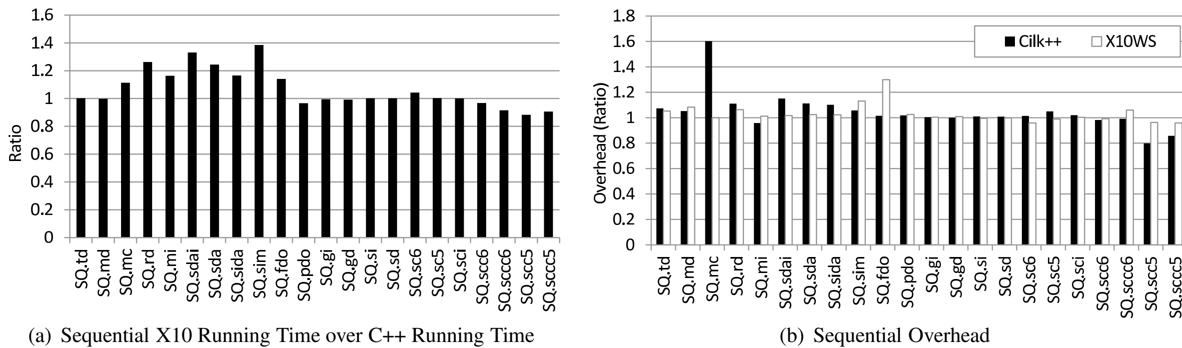


Figure 4. PBBS Running Time(s) (16 workers)



(a) Sequential X10 Running Time over C++ Running Time (b) Sequential Overhead

Figure 5. PBBS Overhead Analysis

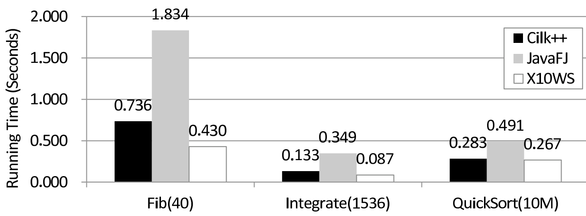


Figure 2. Micro Benchmark Running Time (16 workers)

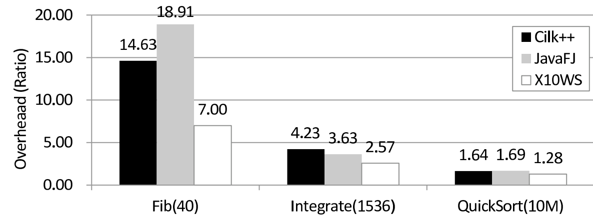


Figure 3. Micro Benchmark Sequential Overhead

If we disable exception support in the work-stealing transformation, we measure performance improvements never exceeding 10%. The cost of correct exception handling is therefore marginal.

7.5 PBBS Benchmark

We translated 6 of the 19 PBBS codes to X10. Most contain a series of measurements. In short, the Sequence tests (SQ) are unit tests. They measure the performance of building blocks used to construct the graph algorithms in the MaxIndSet (MI), MaxMatching (MM), ColorGraph (CG), BreadthFirstSearch (BFS), and RadixSort (RS) programs.

Figure 4 shows the execution time for every benchmark using 16 workers. Overall, X10 and Cilk++ performance are comparable and often very close. Neither runtime wins the comparison.

To better understand these results we plot a series of ratio in Figure 5 for the unit tests:

- The execution time of the sequential X10 code over the execution time of the sequential C++ code.

- The execution time of the single-worker Cilk++ test run over the execution time of the sequential C++ code.
- The execution time of the single-worker X10 test run over the execution of the sequential X10 code.

These numbers clearly identify two orthogonal contributing factors to the difference in performance between Cilk++ and our scheduler.

First, Figure 5(a), the raw sequential performance of the C++ code and the X10 code are not always the same due to the different nature of the languages and compilers. X10 is up to 40% slower than C++. X10 classes are slower than C++ mutable structs, closures are slower than function pointers, etc.

Second, in Figure 5(b), the X10 work-stealing code transformation overhead is often negligible, but in a few cases it slows down the code by up to 30%. Cilk++ is the same, with an overhead often but not always negligible. But affected benchmarks are different.

Overall, the relative sequential performance of the two languages matters more than the work-stealing code transformations.

8. Conclusion

We design and implement a portable work-stealing scheduler for the full X10 language.

We demonstrate it is possible to dramatically increase the coverage of work-stealing policies beyond single-place (shared-memory) async-finish (fork-join) programs while matching the performance of dedicated schedulers on this important class of programs. In particular, our scheduler handles task suspension in its full generality, hence data-dependent synchronization and remote task invocation.

To achieve performance we implement a series of low-level optimizations, which require support from the backend compilers. We implemented such support in the C++ backend compiler for X10. In the future, we plan to do the same with the Java backend compiler for X10 by taking advantage of non-standard capabilities of Jikes RVM.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, pages 211–238, 2000.

[2] G. E. Blelloch. The problem-based benchmark suite. `http://www.cs.cmu.edu/~guyb/PBBS.html`.

[3] G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. Matias. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA'97, pages 12–23, Newport, RI, USA, 1997.

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

[5] J. Brezin, S. J. Fink, B. Bloom, and C. Swart. An introduction to programming with X10. `http://dist.codehaus.org/x10/documentation/guide/pguide.pdf`.

[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 *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'05, pages 519–538, San Diego, CA, USA, 2005.

[7] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Processings of the 37th International Conference on Parallel Processing*, ICPP'08, pages 536–545, Portland, OR, USA, 2008.

[8] 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, San Francisco, CA, USA, 2003.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI'98, pages 212–223, Montreal, QC, Canada, 1998.

[10] S. Ghemawat and P. Menage. TCMalloc : Thread-caching malloc. `http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

[11] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Processings of the 23th IEEE International Parallel & Distributed Processing Symposium*, IPDPS'09, pages 1–12, Rome, Italy, 2009.

[12] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'10, pages 1–12, Atlanta, GA, USA, 2010.

[13] Intel. Intel cilk++ sdk. `http://software.intel.com/en-us/articles/download-intel-cilk-sdk/`.

[14] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: Modern task parallelism for modern high performance computing. In *Proceedings of the ACM/IEEE conference on Supercomputing*, SC'09, Portland, OR, USA, 2009.

[15] D. Lea. Concurrency jsr-166 interest site. `http://g.oswego.edu/dl/concurrency-interest/`.

[16] D. Lea. A java fork/join framework. In *Proceedings of the ACM conference on Java Grande*, JAVA'00, pages 36–43, San Francisco, CA, USA, 2000.

[17] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09, pages 227–242, Orlando, FL, USA, 2009.

[18] 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, Nice, France, 1990.

[19] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[20] R. Raman. Compiler support for work-stealing parallel runtime systems. Master's thesis, Rice University, Houston, Texas, 2009.

[21] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[22] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`.

[23] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP'11, pages 201–212, San Antonio, TX, USA, 2011.

[24] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the twenty-first annual ACM symposium on Parallelism in algorithms and architectures*, SPAA'09, pages 91–100, Calgary, AB, Canada, 2009.

[25] The X10 team. The X10 distribution. `http://sourceforge.net/projects/x10/`.