A Performance Model for X10 Applications

What's going on under the hood?

David Grove Olivier Tardieu David Cunningham Ben Herta Igor Peshansky Vijay Saraswat

IBM Research

groved,tardieu,dcunnin,bherta,igorp,vsaraswa@us.ibm.com

Abstract

To reliably write high performance code in any programming language, an application programmer must have some understanding of the performance characteristics of the language's core constructs. We call this understanding a *performance model* for the language. Some aspects of a performance model are fundamental to the programming language and are expected to be true for any plausible implementation of the language. Other aspects are less fundamental and merely represent design choices made in a particular version of the language's implementation.

In this paper we present a basic performance model for the X10 programming language. We first describe some performance characteristics that we believe will be generally true of any implementation of the X10 2.2 language specification. We then discuss selected aspects of our implementations of X10 2.2 that have significant implications for the performance model.

1. Introduction

Programmers need an intuitive understanding of the performance characteristics of the core constructs of their programming language to be able to write applications with predictable performance. We will call this understanding a *performance model* for the language. Desirable characteristics of a performance model include simplicity, predictive ability, and stability across different implementations of the language. The performance model should abstract away all non-essential details of the language and its implementation, while still enabling reasoning about those details that do have significant performance impact. Languages with straightforward mappings of language constructs to machine instructions usually have fairly straightforward performance models. As the degree of abstraction provided by the language's constructs and/or the sophistication of its implementation increase, its performance model also tends to become more complex.

In this paper, we describe a preliminary performance model for the X10 programming language. X10 is an object-oriented language designed specifically to enable the productive programming of multi-core and multi-node computers. In addition to the expected core language features of any modern object-oriented language, it contains additional constructs for expressing fine-grained concurrency and distributed computation.

X10 Workshop 2011 date, City.

Although the rate of change of the X10 language has significantly decreased from earlier stages of the project, the language specification and its implementations are still immature when compared to languages such as C++ and Java. As such, we consider some aspects of the X10 performance model to still be evolving. Therefore, we break our presentation into two logical sections: aspects that we believe are fairly fundamental to the language itself and aspects that are more closely tied to specific choices embodied in the X10 2.2 implementations.

We begin with a brief review of the X10 language in Section 2. Section 3 discusses those aspects of the performance model that arise from fundamental aspects of the language definition. Section 4 provides an overview of the X10 2.2 implementations. The second logical section of the performance model is presented simultaneously with a discussion of some of the central implementation decisions embodied in the X10 2.2 runtime system (Section 5) and compiler (Section 6).

2. Background

This background 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. A great deal more information can be found online at `http://x10-lang.org`. In particular, the language specification [8], programmer's guide [3], and a collection of tutorials and sample programs are available.

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 multi-core 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 programming language design. The language must also support the development and use of reusable application frameworks to increase programmer productivity; this requirement motivates the inclusion of a sophisticated generic type system, closures, and object-oriented language features. Finally, like any new language, to gain acceptance X10 must be able to smoothly interoperate with existing libraries written in other languages. This last requirement constrains both the design and the implementation of X10 in various ways.

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

(GlobalRefs) 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. To facilitate the return of a value from a remote computation, X10 also supports `at` expressions: `at (p) E`.

X10 also 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.

3. X10 Performance Model

The core language model of X10 is that of a type-safe object-oriented language. Thus much of the core performance model is intended to be similar to that of Java. We believe the Java performance model is generally well-understood. Therefore in this section we focus on areas where the performance models for X10 and Java diverge or where X10 has new constructs that do not trivially map to Java constructs.

3.1 X10 Type System

The type systems of X10 and Java differ in three ways that have important consequences for the X10 performance model. First, although X10 classes are very similar to Java's, X10 adds two additional kinds of program values: functions and structs. Second, X10's generic type system does not have the same erasure semantics as Java's generic types do. Third, X10's type system includes *constrained types*, the ability to enhance type declarations with boolean expressions that more precisely specify the acceptable values of a type.

Functions in X10 can be understood by analogy to closures in functional languages or local classes in Java. They encapsulate a captured lexical environment and a code block into a single object such that the code block can be applied multiple times on different argument values. X10 does not restrict the lifetime of function values; in particular they may escape their defining lexical environment. Thus, the language implementation must ensure that the necessary portions of the lexical environment are available for the lifetime of the function object. In terms of the performance model, the programmer should expect that an unoptimized creation of a function value will entail the heap allocation of a closure object and the copying of the needed lexical environment into that object. The programmer should also expect that trivial usage of closures (closures that do not escape and are created and applied solely within the same code block) will be completely eliminated by the language implementation via inlining of the function body at the application site.

Structs in X10 are designed to be space-efficient alternatives to full-fledged classes. Structs may implement interfaces and define methods and fields, but do not support inheritance. Furthermore structs are immutable: a struct's instance fields cannot be modified outside of its constructor. This particular design point was chosen specifically for its implications for the performance model. Structs can be implemented with no per-object meta-data and can be freely inlined into their containing context (stack frame, containing struct/object, or array). Programmers can consider structs as user-definable primitive types, that carry none of the space or indirection overheads normally associated with objects.

The X10 generic type system differs from Java's primarily because it was designed to fully support the instantiation of generic types on X10 structs without losing any of the performance characteristics of structs. For example, `x10.lang.Complex` is a struct type containing two double fields; `x10.util.ArrayList[T]` is a generic class that provides a standard list abstraction implemented by storing elements of type `T` in a backing array that is resized as needed. In Java, a `java.util.ArrayList[Complex]`, would have a backing array of type `Object[]` that contained pointers to heap-allocated `Complex` objects. In contrast, the backing storage for X10's `x10.util.ArrayList[Complex]` is an array of inline Complex structs without any indirection or other object-induced space overheads. This design point has a number of consequences for the language implementations and their performance model. Much of the details are implementation-specific so we defer them to Section 6 and to the paper by Takeuchi et al [9]. However, one high-level consequence of this design is generally true: to implement the desired semantics the language implementation's runtime type infrastructure must be able to distinguish between different instantiations of a generic class (since instantiations on different struct types will have different memory layouts).

Constrained types are an integral part of the X10 type system and therefore are intended to be fully supported by the runtime type infrastructure. Although we expect many operations on constrained types can be checked completely at compile time (and thus will not have a direct runtime overhead), there are cases where dynamic checks may be required. Furthermore, constrained types can be used in dynamic type checking operations (`as` and`instanceof`). We have also found that some programmers prefer to incrementally add constraints to their program, especially while they are still actively prototyping it. Therefore, the X10 compiler supports a compilation mode where instead of rejecting programs that contain type constraints that cannot be statically entailed it, will generate code to check the non-entailed constraint at runtime (in effect, the compiler will inject a cast to the required constrained type). When required, these dynamic checks do have a performance impact. Therefore part of performance tuning an application as it moves from development to production is reducing the reliance on dynamic checking of constraints in frequently executed portions of the program.

3.2 Distribution

An understanding of X10's distributed object model is a key component to the performance model of any multi-place X10 computation. In particular, understanding how to control what objects are serialized as the result of an `at` can be critical to performance understanding.

Intuitively, executing an `at` statement entails copying the necessary program state from the current place to the destination place. The body of the `at` is then executed using this fresh copy of the program state. What is necessary program state is precisely defined by treating each upwardly exposed variable as a root of an object graph. Starting with these roots, the transitive closure of all objects reachable by properties and non-transient instance fields is serialized and an isomorphic copy is created in the destination place. Furthermore, if the `at` occurs in an instance method of a class or struct and the body of the `at` refers to an instance field or calls an instance method, `this` is also implicitly captured by the `at` and will be serialized. It is important to note that an isomorphic copy of the object graph is created even if the destination place is the same as the current place. This design point was chosen to avoid a discontinuity between running a program using a single place and with multiple places.

Serialization of the reachable object graph can be controlled by the programmer primarily through injection of transient modifiers on instance fields and/or GlobalRefs. It is also possible to have a class implement a custom serialization protocol (`x10.io.CustomSerialization`) to gain even more precise control. An X10 implementation may be able to eliminate or otherwise optimize some of this serialization, but it must ensure that any program visible side-effects caused by user-defined custom serialization routines happen just as they would have in an unoptimized program. Thus, the potential of user-defined custom serialization makes automatic optimization of serialization behavior a fairly complex global analysis problem. Therefore, the base performance model for object serialization should not assume that the implementation will be able to apply serialization reducing optimizations to complex object graphs with polymorphic or generic types.

The X10 standard library provides the `GlobalRef`, `RemoteArray` and `RemoteIndexedMemoryChunk` types as the primitive mechanisms for communicating object references across places. Because of a strong requirement for type safety, the implementation must ensure that once an object has been encapsulated in one of these types and sent to a remote place via an `at`, the object will be available if the remote place ever attempts to spawn an activity to return to the object's home place and access it. For the performance model, this implies that cross-place object references should be managed carefully as they have the potential for creating long-lived objects. Even in the presence of a sophisticated distributed garbage collector[1], the programmer should expect that collection of cross-place references may take a significant amount of time and incur communication costs and other overheads.

Closely related to the remote pointer facility provided by `GlobalRef` is the `PlaceLocalHandle` functionality. This standard library class provides a place local storage facility in which a key (the `PlaceLocalHandle` instance) can be used to look up a value, which may be different in different places. The library implementation provides a collective operation for key creation and for initializing the value associated with the key in each place. Creation and initialization of a `PlaceLocalHandle` is an inherently expensive operation as it involves a collective operation. On the other hand cross-place serialization of a `PlaceLocalHandle` value and the local lookup operation to access its value in the current place are relatively cheap operations.

3.3 Async and Finish

The `async` and `finish` constructs are intended to allow the application programmer to explicitly identify potentially concurrent computations and to easily synchronize them to coordinate their interactions. The underlying assumption of this aspect of the language design is that by making it easy to specify concurrent work, the programmer will be able to express most or all of the useful fine-grained concurrency in their application. In many cases, they may end up expressing more concurrency than can be profitably exploited by the implementation. Therefore, the primary role of the language implementation is to manage the efficient scheduling of all the potentially concurrent work onto a smaller number of actually concurrent execution resources. The language implementation is not expected to automatically discover more concurrency than was expressed by the programmer. In terms of the performance model, the programmer should be aware that an `async` statement is likely to entail some modest runtime cost, but should think of it as being a much lighter weight operation than a thread creation.

As discussed in more detail in Section 5, the most general form of `finish` entails a distributed termination problem. Although programmers can assume that the language implementation will apply a number of static and dynamic optimizations, they should expect that if a `finish` needs to detect the termination of activities across multiple places, then it will entail communication costs and latency that will increase with the number of places involved in the finish.

[1] which is not available in X10 2.2

3.4 Exceptions

The X10 exception model differs from Java's in two significant ways. First, 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 which it will then throw. Second, the current X10 language specification does not specify the exception semantics expected within a single activity. Current implementations of X10 assume a non-precise exception model that enables the implementation to more freely reorder operations and increases the potential for compiler optimizations.

4. X10 2.2 Overview

X10 2.2 is implemented via source-to-source compilation to another language, which is then compiled and executed using existing platform-specific tools. The rationale for this implementation strategy is that it allows us to achieve critical portability, performance, and interoperability objectives. More concretely, X10 2.2 can be either compiled to C++ or Java. The resulting C++ or Java program is then compiled by either a platform C++ compiler to produce an executable or compiled to class files and then executed on a JVM. We term these two implementation paths *Native X10* and *Managed X10* respectively.

Portability is important because we desire implementations of X10 to be available on as many platforms (hardware/operating system combinations) as possible. Wide platform coverage both increases the odds of language adoption and supports productivity goals by allowing programmers to easily prototype code on their laptops or small development servers before deploying to larger cluster-based systems for production.

X10 programs need to be capable of achieving close to peak hardware performance on compute intensive kernels. Therefore some form of platform-specific optimizing compilation is required. Neither interpretation nor unoptimized compilation is sufficient. However, by taking a source-to-source compilation approach we can focus our optimization efforts on implementing a smaller set of high-level, X10-specific optimizations with significant payoff while still leveraging all of the classical and platform-specific optimization found in optimizing C++ compilers and JVMs.

Finally, X10 needs to be able to co-exist with existing libraries and application frameworks. For scientific computing, these libraries are typically available via C APIs; therefore Native X10 is the best choice. However, for more commercial application domains existing code is often written in Java; therefore Managed X10 is also an essential part of the X10 implementation strategy.

Using source-to-source compilation to bootstrap the optimizing compilation of a new programming language is a very common approach. A multitude of languages are implemented via compilation to either C/C++ and subsequent post-compilation to native code or via compilation to Java/C# (source or bytecodes) and subsequent execution on a managed runtime with an optimizing JIT compiler. An unusual aspect of the X10 implementation effort is that it is pursuing both of these paths simultaneously. This decision has both influenced and constrained aspects of the X10 language design (consideration of how well/poorly a language feature can be implemented on both backends is required) and provided for

an interesting comparison between the strengths and limitations of each approach. It also creates some unfortunate complexity in the X10 performance model because the performance characteristics of C++ and Java implementations are noticeably different.

5. X10 2.2 Runtime

The X10 runtime implements the primitive X10 constructs for concurrency and distribution (*async*, *at*, *finish*, *atomic*, and *when*). The X10 compiler replaces these constructs with calls to the corresponding runtime services. The X10 runtime library gets linked to the compiled X10 code (statically or dynamically) to perform these services.

The X10 runtime also defines and implements key APIs for concurrency and distribution such as *x10.util.Team* for multi-point communication or *x10.array.Array.asyncCopy* for large data transfers.

The X10 runtime is primarily written in X10 on top of a series of low-level APIs that provide a platform-independent view of processes, threads, primitive synchronization mechanisms (e.g., locks), and inter-process communication. For instance, the *x10.lang.Lock* class is mapped to *pthread_mutex* (resp. *java.util.concurrent.ReentrantLock*) by Native X10 (resp. Managed X10).

In this section, we review the specifics of the X10 2.2 runtime implementation focusing on performance aspects.

5.1 Distribution

The X10 2.2 runtime maps each place in the application to one process.[2] Each process runs the exact same executable (binary or bytecode).

Upon launch, the process for place 0 starts executing the main activity.

```
finish {
  finish run_static_field_initializers();
  main(args);
}
```

Static fields. Before entering the user main method, the main activity initializes all the static fields for all the classes in the application.

Both X10 2.2 backend compilers map static fields initialized with compile time constants to static fields of the target language. Other static fields are mapped to method calls. Each method triggers the evaluation of the static initializer, caches the result at place 0 for subsequent calls to the method, and broadcasts the result to the other places. Therefore, all but the simplest static initializers incur the cost of a broadcast.

X10RT. The X10 2.2 distribution comes with a series of pluggable libraries for inter-process communication referred to as X10RT libraries [10, 11]. The default X10RT library—*sockets*—relies on POSIX TCP/IP connections. The *standalone* implementation supports SMPs via shared memory communication. The *mpi* implementation maps X10RT APIs to MPI [7]. Other implementations support various IBM transport protocols (LAPI, DCMF, PAMI).

Each X10RT library has its own performance profile—latency, throughput, etc. For instance, the X10 2.2 *standalone* library is significantly faster than the *sockets* library used on a single host.

The performance of X10RT can be tuned via the configuration of the underlying transport implementation. For instance, the *mpi* implementation honors the usual MPI settings for task affinity, fifo sizes, etc.

Teams. The *at* construct only permits point-to-point messaging. The X10 2.2 runtime provides the *x10.util.Team* API for efficient multi-point communication.

Multi-point communication primitives—a.k.a. *collectives*—provided by the *x10.util.Team* API are hardware-accelerated when possible, e.g., *broadcast* on BlueGene/P. When no hardware support is available, the Team implementation is intended to make a reasonable effort at minimizing communication and contention using standard techniques such as butterfly barriers and broadcast trees.

AsyncCopy. The X10 2.2 tool chain implements *at* constructs via serialization. The captured environment gets encoded before transmission and is decoded afterwards. Although such an encoding is required to correctly transfer object graphs with aliasing, it has unnecessary overhead when transmitting immediate data, such as arrays of primitives.

As a work around, the X10 2.2 *x10.array.Array* class provides specific methods—*asyncCopy*—for transferring array contents across places with lower overhead. These methods guarantee the raw data is transmitted as efficiently as permitted by the underlying transport with no redundant packing, unpacking, or copying. Hardware permitting, they initiate a direct copy from the source array to the destination array using RDMAs.[3]

5.2 Concurrency

The cornerstone of the X10 runtime is the scheduler. The X10 programming model requires the programmer to specify the place of each activity. Therefore, the X10 scheduler makes per-place decisions, leaving the burden of inter-place load balancing to the library writer and ultimately the programmer.

The X10 2.2 scheduler assumes a symmetrical, fixed number of concurrent execution units (CPU cores) per process for the duration of the execution. This assumption is consistent with the HPCS context—job controllers typically assign concurrently running applications to static partitions of the available computing resources—but will be relaxed in subsequent releases of X10.

Work-Stealing scheduler. The X10 2.2 scheduler belongs to the family of *work-stealing* schedulers [2, 4] with a *help-first* scheduling policy [5]. It uses a pool of worker threads to execute activities. Each worker thread owns a double-ended queue of pending activities. A worker pushes one activity for each *async* construct it encounters. When a worker completes one activity, it pops the next activity to run from its deque. If the deque is empty, the worker attempts to *steal* a pending activity from the deque of a randomly selected worker.

Since each worker primarily interacts with its own deque, contention is minimal and only arises with load imbalance. Moreover, a *thief* tries to grab an activity from the top of the deque whereas the *victim* always pushes and pops from the bottom, further reducing contention.

In X10 2.2, the *thief* initially chooses a *victim* at random then inspects the deque of every worker in a cyclic manner until it manages to steal a pending activity.

The X10 scheduler borrows the deque implementation of Doug Lea's Fork/Join framework [6].

Life cycle. A worker may be in one of four states:

running one activity,

searching for an activity to execute,

[2] The X10 2.2 runtime may launch additional processes to monitor the application processes. These helper processes are idle most of the time.

[3] RDMA: remote direct memory access.

suspended because the activity it is running has executed a blocking construct, such as *finish* or *when*, or method, such as *System.sleep* or *x10.util.Team.barrier*,

stopped because there are already enough workers running or searching.

Suspended and stopped workers are idle. In X10 2.2, workers searching for pending activities are spinning (i.e., busy waiting). We expect a later X10 release to permit these workers to idle eventually.

Cooperative scheduler. The X10 2.2 scheduler never preempts a worker running user code. The X10 2.2 runtime is designed to enable achieving the highest possible performance on MPI-like distributed X10 applications where the programmer use matching send and receive instructions to achieve total control over the communication and execution schedule. If the user code never yields to the runtime then pending activities (local or remote) are not processed. In other words, the runtime does not make any fairness guarantee.

A worker may yield either by executing a blocking statement or by invoking the *Runtime.probe* method. The latter executes all the pending activities at the time of the call before returning to the caller. This includes all the pending remote activities—activities spawned here from other places—and all the activities already in this worker deque, but does not include activities in other deques.

Parallelism. The user can specify the number of workers in the pool in each place using the `X10_NTHREADS` environment variable.[4] The X10 2.2 scheduler may create additional threads during the execution. But it strives to maintain the number of *non-idle* workers close to the requested value.

- If a worker suspends, the scheduler wakes a stopped worker if available or allocates and starts a new worker if not.
- If a suspended worker resumes, the scheduler preempts and stops a searching worker if any.
- If there are more than `X10_NTHREADS` workers running then the scheduler preempts and stops the first one who empties its deque.

As a result, the current scheduler guarantees the following properties that are intended to hold for any X10 implementation.

1. If there are `X10_NTHREADS` pending activities or more then there are `X10_NTHREADS` or more workers processing them, that is, running them or searching for them.
2. If there are "$n <$ `X10_NTHREADS`" workers running user code then there are "`X10_NTHREADS` $- n$" workers searching for pending activities.
3. If there are `X10_NTHREADS` or more workers running then there are no workers spinning.

Property 1 is the goal of any work-stealing scheduler: assuming the effort of finding pending activities is negligible, parallel activities are processed in parallel using `X10_NTHREADS` parallel processing units.

Property 2 guarantees that available cores are used to find pending activities quickly.

Property 3 mitigates the penalty of busy waiting in the current implementation: spinning workers are never getting in the way of the application provided the user makes sure that `X10_NTHREADS` is at most equal to the number of hardware cores available to the runtime for each place. For instance, if running 8 places on a 32-core node, `X10_NTHREADS` must not be larger than 4 workers per place.

[4] Some X10RT libraries may internally use additional threads for network management. See documentation.

Joining. In order to minimize pool size adjustments, the scheduler implements one key optimization. If a worker blocks on a *finish* construct but its deque is not empty, it does not suspend but instead processes the pending activities from its deque. It only eventually suspends if its deque becomes empty or if it reaches some other blocking construct (different from *finish*). By design, the pending activities that get processed by the worker in this phase must have been spawned from the blocked *finish* body. In the X10 2.2 implementation, the worker will not attempt to steal activities from others if the *finish* construct is still waiting for spawned activities to terminate when the deque gets empty as this would require to carefully pick activities the *finish* construct is waiting for.

Thanks to this behavior, *finish* has much less scheduling overhead than other synchronization mechanisms, e.g., *when* constructs, and should be preferred when possible.

While this optimization is typically very effective at improving performance without observable drawbacks, it may lead to unbounded stack growth for pathological programs. Therefore, it may be disabled by setting the environment variable `X10_NO_STEALS`.[5]

Overhead. For each *async* statement, the current worker must make work available to other workers. In the best implementation and best case scenario (no contention) this requires at least one CAS instruction[6] per *async*. As a result, *async* constructs should only be used to guard computations that require (significantly) more resources than a CAS.

The X10 2.2 runtime also allocates one small heap object per *async*. Again, anything smaller than that should be executed sequentially rather than wrapped with an *async*. Moreover, memory allocation and garbage collection can become a bottleneck if vast amounts of activities are created concurrently. The runtime therefore exposes the *Runtime.surplusActivityCount* method that returns the current size of the current worker deque. Application and library code may invoke this method to decide whether or not to create more asynchronous activities, as in:

```
if (Runtime.surplusActivityCount() >= 3)
  m();
else
  async m();
```

5.3 Synchronization

Finish. Within a place, one only needs to count activity creation and termination events to decide the completion of a *finish* construct. The story is different across places as inter-process communication channels are likely to reorder messages so that termination events may be observed ahead of the corresponding creation events. The X10 2.2 implementation of *finish* keeps track of these events on an unambiguous, per-place basis.

In the worst-case scenario, with p places, there will be p counters in each place, that is, $p \times p$ counters for each *finish*. Moreover, there could be one inter-process message for each activity termination event. Messages could contain up to p data elements.

In practice however much fewer counters, fewer messages, and smaller messages are necessary thanks to various optimizations embedded in the X10 2.2 implementation. In particular, events are accumulated locally and only transmitted to the *finish* place when local quiescence is detected—all local activities for this *finish* have completed. Counters are allocated lazily. Messages use sparse encodings.

[5] The `X10_NO_STEALS` flag essentially turns deep stacks into large collections of mostly-idle threads with smaller stacks, avoiding stack overflow errors. But ultimately, this only matters to unscalable programs of little practical relevance.

[6] CAS: compare-and-swap.

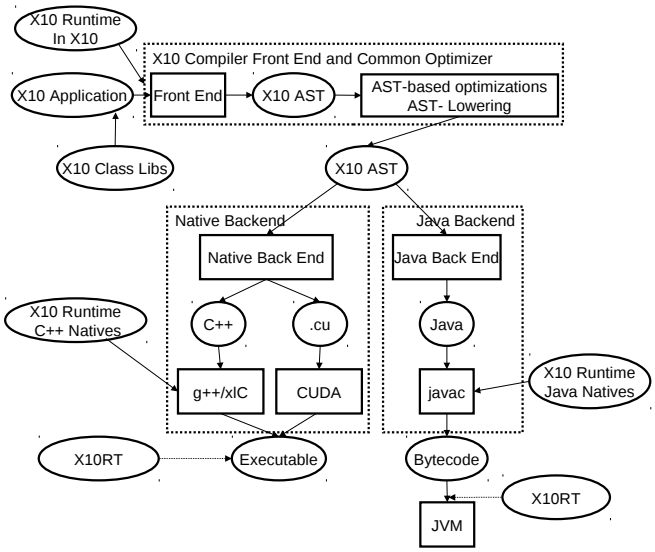

Figure 1. X10 Compiler Architecture

Atomic and When. The X10 2.2 implementation of the *atomic* construct uses a place-wide lock. The lock is acquired for the duration of the atomic section. The *when* construct is implemented using the same lock. Moreover, every suspended *when* statement is notified on every exit from an atomic section, irrespective of condition.

The per-place lock effectively serializes all atomic operation in a place whether they might inerfere or not. This implementation does not scale well beyond a few worker threads. Similarly, the *when* implementation does not scale well beyond a few occurrences (distinct condition variables).

The X10 standard library provides various atomic classes and locks that enable better scaling. Both the *collecting finish* idiom and the *x10.util.WorkerLocalStorage* API may be also used to minimize contention.

6. X10 2.2 Compilation

The overall architecture of the X10 compiler is depicted in Figure 1. This compiler is composed of two main parts: an AST-based front-end and optimizer that parses X10 source code and performs AST based program transformation; Native/Java backends that translate the X10 AST into C++/Java source code and invokes a post compilation process that either uses a C++ compiler to produce an executable binary or a Java compiler to produce bytecode.

6.1 Native X10

When an application programmer writes X10 code that they are intending to execute using Native X10, their base performance model should be that of C++. Unless discussed below, the expected performance of an X10 construct in Native X10 is the same as the corresponding C++ construct.

6.1.1 Classes and Interfaces

X10 classes are mapped to C++ classes and the compiler directly uses the C++ object model to implement inheritance, instance fields, and instance methods. Interfaces are also mapped to C++ classes to support method overloading, but the X10 implements relationship is not implemented using the C++ object model. Instead, additional interface dispatch tables (akin to ITables in Java[7]) are generated by the X10 compiler "outside" of the core C++ object model. The motivation for this design decision was to stay within the simpler, single-inheritance subset of C++ that minimizes per-object space overheads and also preserves the useful property that a pointer to an object always points to the first word of the object and that no "this pointer adjustment" needs to be performed on assignments or during the virtual call sequence.

Non-interface method dispatch corresponds directly to a C++ virtual function call. Interface method dispatch will involve additional table lookups and empirically is 3 to 5 times slower than a virtual function call. C++ compilers typically do not aggressively optimize virtual calls, and will certainly not be able to optimize away the dispatch table lookup used to implement interface dispatch. Therefore, as a general rule, non-final and interface method invocations will not perform as well in Native X10 as they will in Managed X10.

Unless specially annotated, all class instances will be heap allocated and fields/variables of class types will contain a pointer to the heap allocated object.

6.1.2 Primitives and Structs

The dozen X10 struct types that directly correspond to the built-in C primitive types (int, float, etc.) are implemented by directly mapping them to the matching primitive type. Any X10 level functions defined on this types are implemented via static inline methods. The performance characteristics of the primitive C++ types is exactly the performance of their X10 counterparts.

All other X10 structs are mapped to C++ classes. However, all of the methods of the C++ class are declared to be non-virtual. Therefore, the C++ class for a struct will not include a vtable word. Unlike object instances, struct instances are not heap allocated. They are instead embedded directly in their containing object or stack-allocated in the case of local variables. When passed as a parameter to a function, a struct is passed by value, not by reference. In C++ terms, a variable or field of some struct type S is declared to be of type S, not S*.

This implementation strategy optimizes the space usage for structs and avoids indirections. Programmers can correctly think of structs as taking only the space directly implied by their instance fields (modulo alignment constraints). However, passing structs, especially large structs, as method parameters or return value is significantly more expensive than passing/returning a class instance. In future versions of X10 we hope to be able to pass structs by reference (at the implementation level) and thus ameliorate this overhead.

6.1.3 Closures and Function Types

An X10 function type is implemented exactly the same as other X10 interface types. An X10 closure literal is mapped to a C++ class whose instance fields are the captured lexical environment of the closure. The closure body is implemented by an instance method of the C++ class. The generated closure class implements the appropriate function type interface. Closure instances are heap allocated. If the optimizer is able to propagate a closure literal to a program point where it is evaluated, the closure literal's body is unconditionally inlined. In many cases this means that the closure itself is completely eliminated as well.

6.1.4 Generics

Generic types in X10 are implemented by using C++'s template mechanism. Compilation of a generic class or struct results in the definition of a templatized C++ class. When the generic type is

[7] see the description of "searched ITables" in Alpern et al. [1]

instantiated in the X10 source, a template instantiation happens in the generated C++ code.

The performance of an X10 generic class is very similar to that of a similar C++ templatized class. In particular, instantiation based generics enable X10 generic types instantiated on primitives and structs to be space efficient in the same way that a C++ template instantiated on a primitive type would be.

6.1.5 Memory Management

On most platforms Native X10 uses the Boehm-Demers-Weiser conservative garbage collector as its memory manager. A runtime interface to explicitly free an object is also available to the X10 programmer. The garbage collector is only used to automatically reclaim memory within a single place. The BDWGC does not yield the same level of memory management performance as that of the memory management subsystem of a modern managed runtime. Therefore, when targeting Native X10 the application programmer may need to be more conscious of avoiding short-lived objects and generally reducing the application's allocation rate.

Because the X10 2.2 implementation does not include a distributed garbage collector, if a `GlobalRef` to an object is sent to a remote place, then the object (and therefore all objects that it transitively refers to) become uncollectable. The life-time of all multi-place storage must currently be explicitly managed by the programmer. This is an area of the implementation that needs further investigation to determine what mix of automatic distributed garbage collection and additional runtime interfaces for explicit storage control will result in the best balance of productivity and performance while still maintaining memory safety.

6.1.6 Other Considerations

In general, Native X10 inherits many of the strengths and weaknesses of the C++ performance model. C++ compilers may have aggressive optimization levels available, but rarely utilize profile-directed feedback. C++ compilers are generally ineffective at optimizing non statically-bound virtual function calls. Over use of object-oriented features, interfaces, and runtime type information is likely to reduce application performance more in Native X10 than it does in Managed X10.

The C++ compilation model is generally file-based, rather than program-based. In particular, cross-file inlining (from one .cc file to another) is performed fairly rarely and only at unusually high optimization levels. Since the method bodies of non-generic X10 classes are mostly generated into .cc files, this implies that they are not easily available to be inlined except within their own compilation unit (X10 file). Although for small programs, this could be mitigated by generating the entire X10 application into a single .cc file, this single-file approach is not viable for the scale of applications we need Native X10 to support.

6.2 Managed X10

When an application programmer writes X10 code that they are intending to execute using Managed X10, their base performance model should be that of Java. Unless discussed below, the expected performance of an X10 construct in Managed X10 is the same as the corresponding Java construct.

Because significantly more details on the Managed X10 implementation and its performance characteritics can be found in the workshop paper by Takeuchi et al [9], we only touch on a few key issues in this paper.

6.2.1 Classes and Interfaces

X10 classes and interfaces are mapped to Java classes and interfaces. As much as possible, the X10 object model is implemented by using the corresponding feature of the Java object model. As the JVM expends significant effort to optimize object-oriented features of Java, the same features in X10 are also able to benefit from state-of-the-art optimization.

6.2.2 Primitives and Structs

The eight X10 struct types that correspond to built-in Java primitive types are mapped directly to those types. All other X10 structs are mapped to Java classes, including the unsigned versions of the built-in primitives. This deviates from the intended X10 performance model in that most structs are no more space efficient than classes. In Managed X10 2.2, with the exception of the eight built-in Java primitives, instances of both structs and classes are heap allocated and incur the same indirection overhead.

6.2.3 Closures

X10 closures are mapped to Java classes that implement the appropriate function interface. Capture of the lexical environment is done explicitly by the X10 compiler just as in Native X10.

6.2.4 Generics

X10 generics cannot be implemented simply by mapping to Java generics because X10 supports a richer set of operations on a generic type. In particular, generic type parameters can be used in runtime type tests (`instanceof` and `as`) and in allocation operations. Therefore, Managed X10 must augment Java generics with additional runtime type information. Significantly more information about the implementation options and their performance characteristics is found in Takeuchi et al [9].

6.2.5 Memory Management

The memory management subsystem of the JVM tends to be highly tuned. Therefore, X10 programs executing in Managed X10 can expect to sustain a higher allocation rate than is possible in Native X10. However, just as in Native X10, there is no distributed garbage collection system implemented in X10 2.2 and therefore management of cross-place references is entirely manual.

6.2.6 Other Considerations

Just like Java, Managed X10 relies on Just-in-Time compilation and adaptive optimization to reach peak performance. Therefore it may take significant running time for an application to "warmup" and reach its peak performance. X10 application programmers need to consider this when trying to evaluate the performance of their code.

7. Conclusion

Clearly, the performance models described in this paper are not the final and definitive X10 performance model. However, we do believe that the language specification and its two implementations are well-enough understood that it is possible for significant X10 programs to be written and for programmers to obtain predictable and understandable performance behavior from those programs. As the language implementations continue to mature, we expect to be able to eliminate some of the less desirable features of the X10 2.2 performance models.

We hope that the open discussion of our design decisions in implementing X10 and their implications for its performance will be useful to the X10 programmer community and to the broader research community that is engaged in similar language design and implementation projects. Finally, we look forward to developing more detailed performance models for Native X10 and Managed X10 and more formally comparing the inherent trade-offs between implementing a language via static compilation and via a managed runtime with an optimizing JIT compiler.

Acknowledgments

The performance models described in this paper are the results of discussions with a large number of people on the X10 team and user community. We are especially grateful to all of the people who have written benchmarks and application programs in X10. The experience of working with them to tune their programs has been invaluable in helping us develop the ideas described in this paper.

We thank the anonymous reviewers for their valuable comments on how to improve the content and structure of this paper.

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

References

[1] ALPERN, B., COCCHI, A., FINK, S., AND GROVE, D. Efficient implementation of Java interfaces: Invokeinterface considered harmless. *ACM SIGPLAN Notices 36*, 11 (Nov. 2001), 108–124. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[2] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46* (September 1999), 720–748.

[3] BREZIN, J., FINK, S. J., BLOOM, B., AND SWART, C. An introduction to programming with X10. `http://dist.codehaus.org/x10/documentation/guide/pguide.pdf`.

[4] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (New York, NY, USA, 1998), PLDI '98, ACM, pp. 212–223.

[5] GUO, Y., BARIK, R., RAMAN, R., AND SARKAR, V. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–12.

[6] LEA, D. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (New York, NY, USA, 2000), JAVA '00, ACM, pp. 36–43.

[7] MPI FORUM. MPI: A Message-Passing Interface Standard. Version 2.2. `http://www.mpi-forum.org`, September 4th 2009.

[8] SARASWAT, V., BLOOM, B., PESHANSKY, I., TARDIEU, O., AND GROVE, D. X10 language specification. `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`.

[9] TAKEUCHI, M., MAKINO, Y., KAWACHIYA, K., HORII, H., SUZUMURA, T., SUGANUMA, T., AND ONODERA, T. Compiling x10 to java. X10 workshop, 2011.

[10] X10RT API specification. `http://dist.codehaus.org/x10/x10rt`.

[11] X10RT implementations. `http://x10.codehaus.org/X10RT+Implementations`.