Towards Concurrency Refactoring for X10

Shane Markstrum
University of California, Los Angeles
Computer Science Department
Los Angeles, CA USA
smarkstr@cs.ucla.edu

Robert M. Fuhrer
IBM T.J. Watson Research Center
Hawthorne, NY USA
rfuhrer@us.ibm.com

Todd Millstein
University of California, Los Angeles
Computer Science Department
Los Angeles, CA USA
todd@cs.ucla.edu

Abstract

In this poster, we present our vision of refactoring support for languages with a partitioned global address space memory model as embodied in the X10 programming language. We examine a novel refactoring, extract concurrent, that introduces additional concurrency within a loop by arranging for some user-selected code in the loop body to run in parallel with other iterations of the loop. We discuss the mechanisms and challenges for implementing this refactoring and how development of this refactoring provides insight for designing future refactorings.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.6 [*Software Engineering*]: Programming Environments; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Languages

1. Introduction

The industry's shift to multicore processors has sparked a trend toward pushing parallel programming into the mainstream. This trend poses a significant challenge, since creating and maintaining parallel programs that are both efficient and reliable is notoriously difficult. One response to this challenge by the programming languages community has been to create new (and revisit old) language abstractions and programming models for parallel programming and to develop new languages based on these abstractions.

While new languages can greatly aid programmers in developing parallel programs, we believe that new languages cannot achieve mainstream success without associated tooling support. Modern integrated development environments (IDEs) such as Eclipse (3) provide many benefits that programmers have come to rely upon, helping them to more easily navigate through a program, understand dependencies among parts of a program, and safely evolve a program to improve its quality along some dimension. The latter benefit is typically provided through code *refactorings*.

We believe that specialized refactoring support will be a critical tool to help programmers improve the quality of parallel code. Such refactorings could be used to improve efficiency while preserving program behavior and key concurrency invariants (e.g., atomicity, deadlock-freedom).

Any set of refactorings will of necessity be tailored to the needs of a particular parallel programming model and language. We focus on providing refactoring support for the *partitioned global address space* (PGAS) memory model as embodied in the X10 (8; 2), UPC (4) and Titanium (9) programming languages. In this model, the programmer sees a uniform representation of data and data structures over distributed nodes, regardless of the physical location of the data. However, each piece of data is assigned to a fixed partition (or *place* in X10 terminology) and can only be accessed by *activities* that run at that place.

In this presentation, we focus on the X10 language, an object-oriented language providing first-class, high-level constructs for asynchronous activities, synchronization, phased computations, data distribution, and atomicity. By incorporating such abstractions as first-class constructs in the language, the burden of reasoning about various program properties is often reduced from global reasoning involving complex control flow to simple and modular reasoning about lexical containment. In fact, many interesting properties (e.g. deadlock freedom) can often be ensured statically. In this way, X10's constructs simplify both the programmer's task of understanding and tools' tasks of static program analysis.

The PGAS model provides particular opportunities and challenges for automated refactorings. On the one hand, code transformations are simplified since the code need not explicitly handle inter-processor communication. On the other hand, transformations must properly handle the asynchronicity that arises among activities and must respect the synchronization constraints imposed on these activities by the semantics of the various language constructs. In this paper we describe an initial concurrency refactoring for X10 that we have been developing inside the X10DT plugin for Eclipse.

2. Extract Concurrent

As a first step toward our vision, we are developing a refactoring called *extract concurrent* for the X10 language within the X10DT, our Eclipse-based IDE. The transformation introduces concurrency within a loop by arranging for some user-selected code in the loop body to run in parallel with other iterations of the loop.

As an example, consider the X10 code in Figure 1, an excerpt from an X10 implementation of the `ConcurrentHashMap` class from the Java standard utilities. In this snippet from the `containsValue` method, a snapshot of the modification status of each map segment is taken before determining whether a particular segment contains the desired value. This approach allows value lookup to occur without locking the entire map. Given the PGAS model for data distribution, the individual elements of the array `segments` could reside anywhere in the global address space, increasing their access cost. Further, each call to `modCount()` must block until the call completes, per X10 semantics. Thus it is possi-

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
ACM 978-1-60558-397-6/09/02.

```
int mcsum=0;
for ( point p : segments ) {
  mc[p] = segments[p].modCount();
  mcsum += mc[p];
  if(segments[p].containsValue(value))
    return true;
}
```

Figure 1. An excerpt from the X10 version of the Java library `java.util.concurrent.ConcurrentHashMap`.

```
int mcsum=0;
future<int>[.] f_segments =
                 new future<int>[segments.region];
for ( point p : segments ) {
  f_segments[p] =
    future(segments[p]){segments[p].modCount()};
}

for ( point p : segments ) {
  mc[p] = f_segments[p].force();
  mcsum += mc[p];
  if(segments[p].containsValue(value))
    return true;
}
```

Figure 2. A transformation of the program excerpt from Figure 1 introducing additional concurrency via the X10 `future` construct.

ble that asynchronously executing the `modCount` method for each array element will speed up the overall execution of the loop.

One way to introduce this concurrency, shown in Figure 2, is to execute each of the `modCount` invocations as `future`s in a new loop and only synchronize with those executions via a `force` operation when their results are actually needed. This is in fact the transformation that was manually applied to the code in our example after the initial translation from Java to X10. Our *extract concurrent* refactoring automates this transformation and ensures that the transformation preserves program behavior. Our refactoring also supports a generalization of this transformation that allows a block of statements to be safely executed asynchronously, but we do not discuss it here for brevity's sake.

Our refactoring involves two main components:

1. *Loop dependence analysis.* Since introducing parallelism in the middle of a loop might affect the ability of other statements in a loop to evaluate properly, it is important that loops do not depend on the results of any asynchronously executed statements. We have developed a set of analyses to determine whether *extract concurrent* will adversely affect the execution of the code and violate its perceived sequential consistency.
2. *Transformation pattern.* We have developed a general pattern for the *extract concurrent* transformation on viable sequential loops. The pattern splits the loop in two, as shown in Figure 2: the first loop introduces the desired statement- and/or expression-level parallelism, while the second loop synchronizes with and utilizes the results of the asynchronous execution.

The concurrency constructs and the PGAS model used by X10 simplify the analysis required to determine when program transformations are safe. For example, the static analysis required to determine whether a statement may be executed asynchronously is reduced to determining local and loop-carried dependencies that prevent a statement from being asynchronously executed. The example highlights a perceived benefit of refactoring X10: data locality and asynchronous execution are separable, or *lateral*, concerns. Thus, a programmer may manipulate either the amount of program concurrency or the data distribution while keeping the other relatively fixed.

We have built a prototype of the *extract concurrent* refactoring and the supporting analysis in the X10DT, and we are in the process of refining the implementation and experimenting with it on some X10 applications. Although the present implementation targets X10, we believe that the transformation is readily applicable to other languages with a PGAS programming model.

3. Related Work

To our knowledge, ours is the first work to consider automated code refactorings in the context of the PGAS model. However, a number of IDEs and tools have been developed to aid parallel language users. We elide discussion here of automatic parallelization techniques and parallel program analysis to focus on related work in tooling support.

The SUIF Explorer (6) assists programmer parallelizing of code by providing a dynamic dependence analyzer, but does not feature integrated refactoring support. The ParaScope Editor (5) is an IDE that enables exploration and manipulation of loop-level parallelism in a Fortran-like language. It makes analysis results and a number of program transformations, including *loop interchange* and *loop distribution*, available to the user.

Photran (7) is an IDE that intends to, but does not yet, provide concurrency refactoring support for HPC applications in Fortran. TSF (1) is an IDE tool for writing transformation scripts and transforming parallel Fortran programs. Some of the transformations it provides have preconditions for verifying soundness, which is a feature we also integrate into the extract concurrent refactoring.

References

[1] F. Bodin, Y. Mével, and R. Quiniou. A user level program transformation tool. In *International Conference on Supercomputing*, pages 180–187, 1998.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[3] Eclipse home page. `http://www.eclipse.org`.

[4] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specifications v1.1.1, October 2003.

[5] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[6] S.-W. Liao, A. Diwan, J. Robert P. Bosch, A. Ghuloum, and M. S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, New York, NY, USA, 1999. ACM Press.

[7] J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and high-performance computing. In *SE-HPCS '05: Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications*, pages 37–39, New York, NY, USA, 2005. ACM Press.

[8] V. Saraswat. Report on the experimental language X10 v0.41. http://www.research.ibm.com/x10/.

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.