Extracting Concurrency via Refactoring in X10

Shane A. Markstrum
Bucknell University
Computer Science Department
Lewisburg, PA USA
shane.markstrum@bucknell.edu

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

Abstract

In this paper, 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 into a program by arranging for selected code in a loop body to run in parallel with future iterations of the loop. We discuss the mechanisms and algorithms required to implement the refactoring; provide insights on how development of this refactoring aids designing future concurrency refactorings; and outline new strategies for determining the effectiveness of concurrent refactorings.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.6 [*Software Engineering*]: Programming Environments—Integrated 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 (11; 2), UPC (4) and Titanium (12) 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 the rest of 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.[1]

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

[1] The transformation can be thought of as a fine-grained variant of loop distribution (8) where only a portion of the loop body is executed in parallel with other iterations.

WRT'09, October 25, 2009, Orlando, Florida, USA.

```
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.

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 possible 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 `futures` 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. In practice, this transformation is more widely applicable to multiple statements or expressions. We present here only the single statement or expression case.

Before:

```
for(...) {
    ... target ...
}
```

After:

```
for(...){
    ...
    targetFuture = future { target };
    ...
}

for(...) {
    ... targetFuture.force() ...
}
```

Figure 3. The transformation schema for futurizing an expression.

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. With a traditional model, this same transformation might also require an analysis to determine how much dependent data would need to be copied for asynchronous execution and where that execution should take place. The example highlights a perceived benefit of refactoring X10: data locality and asynchronous execution are separable, or *lateral*, concerns. Thus, a programmer may manipulate the amount of program concurrency while keeping the data distribution fixed, or manipulate the distribution while keeping perceived program concurrency relatively unchanged.

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. It is our hope that the analysis itself will prove useful in implementing future refactorings, and although the present implementation targets X10, we believe that the transformation is readily applicable to other languages with a PGAS programming model.

3. Algorithmic Details

The goal of the *extract concurrent* refactoring is to allow users to (somewhat) safely introduce parallelism/concurrency to array accesses and updates that are done within a `for` loop. Since loops tend to be execution hotspots, and X10 supports explicit asynchronous event constructs, this refactoring is the type that X10 users should expect to find in an IDE like Eclipse.

The general schema for the transformation is illustrated by the two code skeletons shown in Figure 3, which depict the original and the transformed code.

A basic assumption of this transformation is that it will introduce no activities that remain unfinished outside the scope of the parent construct containing the target distributed array. As a result, the developer need not expend mental effort to determine when the results of the activity will be available. In this case, overlap between the two loops can be easily permitted, since the `force()` calls in the second loop perform all of the necessary synchronization.

3.1 Pre-conditions

In general, the code for a candidate `for` loop will look as in Figure 4. In this case, the user would be interested in potentially making the function `f(Target[])` representing the targeted expres-

```
Loop (InductionVars ...){
  Block1
  Result = (Exp1 op f(Target[])) op Exp2;
  Block2
}
```

Figure 4. The generic loop structure for the *extract concurrent* refactoring.

sion – which is not necessarily a method call – into a futurized expression. The candidate expression or statement will be called the *target*. Here, `Block1` and `Block2` are both arbitrary blocks of code

As with all parallelizing algorithms, the most important information to track is loop-carried dependencies. In this case, all loop-carried dependencies must be reproducible on all executions of the loop. As a result, the following pre-conditions must be met in order to refactor the code:

- No loop-carried dependencies are allowed on the target distributed array.
- The target must not produce side-effects.
- There must be no loop-carried dependencies on user input or the evaluation of any chaotic functions (e.g., random number generators).
- Loop-carried dependent statements that occur semantically after the target may not have side-effects that directly affect the target.

3.2 Interactions with X10 Constructs

In this section, the interaction of the transformation with various X10 constructs will be discussed. Because the transformation is aimed at targets residing within loops, any errors that are a result of poorly formed code outside of the loop will be omitted from this discussion. It is noted, however, that any asynchronous activity that remains unfinished or unforced before the loop is entered may create data races within the program.

Asyncs – Unfinished asynchronous activities that have (possibly indirect) side-effects on the target and occur semantically before the target in the loop should be must stop the transformation from occurring. All other finished asynchronous activity is allowed if it conforms with the X10 language rules and meets the pre-conditions stated in the previous subsection. Note that choosing to extract concurrent on a target within a finished async block effectively precludes introduction of new parallelism.

Futures – Futures, in general, are treated the same as asyncs. However, code inserted between the two constructed loops must not have side-effects on any futurized expression that is used in the second loop

Atomics – As long as the target does not fall within an atomic block, there are no restrictions on the use of atomic in the loop. The target may contain an atomic block itself.

Clocks – Because the transformation creates new paths in the program, `next` and `drop` may not be called on a clock in the first half of the loop or in any statement which has loop-carried dependencies. After execution of the target can be guaranteed to have completed (i.e., at any non-loop-carried dependent statement after the target), it is safe to perform any of the clock operations.

Exceptions – If the target is within a try block, then as a conservative measure, we force the transformation to fail. This is based on the semantics of X10: uncaught exceptions thrown by asynchronous activities are accumulated by the collecting `finish`. Thus, exceptions thrown by activities spawned inside a try block are not propagated to the try block's catch clauses.

```
Loop (InductionVars ...){
  Block1
  Result = Exp1 op
    future(Target.dist[]){f(Target[])}.force() op
    Exp2;
  Block2
}
```

Figure 5. The basic candidate transformation loop with explicit support for data distribution.

4. Evaluation

4.1 Methods of Evaluating Transformed Code

There are two vectors through which we can measure the effectiveness of our extract concurrent transformation: running time and communication overhead. If the running time of the transformed code is reduced and the amount of additional communication is not made substantially worse, then the refactoring would be considered a success.

In particular, comparing the performance of the transformation to a program which only uses Java arrays – which exist solely in one place – would not be a perfectly valid comparison in terms of running time or interprocess communication. Distributed arrays inherently involve at least some interprocess communication and, as a result, may require more time to access or update elements of the array. As a result, transformed code should be measured against loops similar to the one seen in Figure 5.

Most current X10 implementations do not have compiler optimizations that significantly speed up concurrent data access, although some do support true multiple processor data distribution and communication. Due to this, the time measurements on both the basic asynchronous loop and the transformed asynchronous loops will most likely exhibit much worse behavior than standard Java. It is also possible that clock time between the basic and transformed loops will not be accurate and comparisons on these results might be skewed (e.g., the transformed loops might exhibit worse clock time because of unoptimized thread spawning and thread communication). However, this is simply a result of the current state of X10 implementation and not an inherent shortcoming of the X10 language or other PGAS model languages. At least one X10 implementation supports measurements that separate out overheads associated with the current implementations from those of the runtimes of X10 programs. It is with these tools that we hope to effectively measure the costs and benefits associated with applying this refactoring.

4.2 A Formal Cost-Benefit Analysis of the Transformation

In lieu of testing numbers, we present a more formal analysis of the running time of the algorithm for best, worst, and average case scenarios. To perform the analysis, we must first define a few variables. Let m be the running time of `f(Target[])`, n be the size of `Target[]`, C_{fut} be the overhead associated with starting a `future`, C_{for} be the overhead associated with calling `force`, p by the amount of available concurrency (i.e., number of places or threads) and $p \leq n$, and b be $O(\texttt{Block1}+\texttt{Exp1}+\texttt{Exp2})+\texttt{Block2})$.

For the code in Figure 5, the running time would be

$$O(n \cdot (b + C_{fut} + C_{for} + m))$$

To analyze the `future` transformation, we will break it down into best case, worst case, and average case and ignore any thread overhead.

Best case – In the best case, the running time of the targeted expression completely disappears. Thus, the running time in the best case is

$$O(n \cdot (b + C_{fut}) + n \cdot (b + C_{for})) \\ = O(n \cdot (2b + C_{fut} + C_{for})$$

Thus the transformation is at least as good as the standard loop if $m \leq b$.

Worst case – In the worst case, the full running time of the targeted expression is observed every time at every `force` expression. The running time in the worst case is

$$O(n \cdot (b + C_{fut}) + n \cdot (b + C_for + m)) \\ = O(n \cdot (2b + C_{fut} + C_{for} + m))$$

In this case, there is no way for the transformed code to match the performance of the standard loop unless $b = 0$. This case is also equivalent to the case of having distributed data but having no apparent concurrency.

Average case – In the average case analysis, $2 \leq p$, or the worst case behavior is exhibited. We shall consider two subcases for the average case analysis: with and without consideration of time spent executing each loop when calculating the amount time of evaluating `f(Target[])`.

In the case of loop execution time not being factored, assume for the sake of simplicity, but without loss of generality, that for every p processes encountered, m is accrued. Then the running time is

$$O(n \cdot (2b + C_{fut} + C_{for} + \frac{m}{p}))$$

In this case, the transformation provides a performance boost if $m > \frac{bp}{p-1}$. In the limit on p, this case is equivalent to the best case.

If loop execution time is factored into the concurrent running time, the analysis is slightly more complicated. First, the total running time for the loop code that executes simultaneously with execution of `f(Target[])` is

$$O(n \cdot (b + C_{fut}) - \tfrac{1}{2}b + n \cdot (b + C_{for}) - \tfrac{1}{2}b) \\ = O(n \cdot ((2 - \tfrac{1}{n})b + C_{fut} + C_{for}))$$

The total running time for the concurrent execution of `f(Target[])`, assuming a similar perfect concurrent pipeline as in the previous case, is $O(\frac{nm}{p} + p(b + C_{fut}))$. This is equivalent to the best case when

$$m \leq ((2 - \frac{p+1}{n})b + (1 - \frac{p+1}{n})C_{fut} + C_{for})$$

If the above inequality does not hold, running time is

$$O(\frac{n}{p}((p-1)(2b + C_{fut} + C_{for}) + 2m) + (1 + \frac{1}{p})(b + C_{fut}))$$

This is at least as good as the basic loop if

$$m \geq (1 + \frac{p+1}{n(p-2)})b + \frac{n-p-1}{n(p-2)}C_{fut} - \frac{1}{p-2}C_{for}$$

In the limit of p, this is equivalent to the best case.

Except in the worst case scenario, this formal analysis indicates that the extract concurrent refactoring *should* provide beneficial results as long as there are a large number of places and the data associated with the targeted expression is relatively well distributed among the places. It will be interesting to see how often this will be the case when applying the refactoring to real X10 code.

5. 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. Such tooling support is crucial since complete automatic parallelization of programs is now generally regarded as infeasible. Thus, refactoring and transformation tools allow programmers to remain aware of, and control to a certain degree, the level of concurrency in their code.

The SUIF Explorer (7), for the SUIF compiler system (10), combines automatic parallelization techniques with a dynamic dependence analyzer to assist programmers in parallelizing their code, but does not feature integrated refactoring support. The ParaScope Editor (6; 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 (9) 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. Another difference between this work and SUIF, ParaScope, Photran, and TSF, is our focus on PGAS model languages as opposed to the more traditional memory models of the procedural C and Fortran languages.

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] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: an interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993. URL `citeseer.ist.psu.edu/hall93experiences.html`.

[6] 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.

[7] 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.

[8] Yoichi Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971.

[9] 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.

[10] SUIF compiler system. The SUIF compiler system. http://suif.stanford.edu/suif/suif2/.

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

[12] 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.