Revisiting Loop Transformations with X10 Clocks

Tomofumi Yuki
Inria / LIP / ENS Lyon, France
tomofumi.yuki@inria.fr

Abstract

Loop transformations are known to be important for performance of compute-intensive programs, and are often used to expose parallelism. However, many transformations involving loops often obfuscate the code, and are cumbersome to apply by hand. The goal of this paper is to explore alternative methods for expressing parallelism that are more friendly to the programmer. In particular, we seek to expose parallelism without significantly changing the original loop structure. We illustrate how clocks in X10 can be used to express some of the traditional loop transformations, in the presence of parallelism, in a manner that we believe to be less invasive. Specifically, expressing parallelism corresponding to one-dimensional affine schedules can be achieved without modifying the original loop structure and/or statements.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

Keywords X10, parallel programming, clocks, synchronization, loop transformation, affine schedule, unimodular framework

1. Introduction

With increasing amount of parallelism and heterogeneity thrown at the performance programmers, achieving high performance in a productive manner is an extremely challenging task. `X10` provides built-in parallel constructs to express parallelism of various forms as an attempt to ease this task.

Loops have been, and will continue to be, the hotspot of many applications. In the world of compilers and high-performance computing, loop transformations have been a topic of interest for a long time, with many transformations and frameworks developed over the years [1]. With the ability to spawn and synchronize with threads through loosely constrained syntax of `finish/async`, loop transformations in `X10` are even more challenging and interesting.

One classical use of loop transformations is to expose parallelism, which may involve applying sequences of complicated transformations. The resulting code often looks very distant from the original, and the process can be error-prone. The key question in this paper is how to express some of the transformations using `X10` constructs in a manner that is more friendly to the programmer. Specifically, our goal is to avoid modifying loop bounds or expressions involving loop indices that typically accompanies advanced loop transformations such as loop skewing.

We present a few loop transformations involving parallel loops that can be expressed using clocks in `X10`. We believe that our proposed way of expressing the transformations are easier for the programmer to understand and manipulate, especially for fine-grained parallelism. However, it does require that the programmer goes through the learning curve of subtleties associated with synchronizations via clocks.

One potential application of our observation is to recognize such patterns for specifying parallelism and then to replace them with more efficient implementations based on traditional loop transformations. Separation of concerns is a topic that has been raised a number of times in many different contexts, equally applicable to parallel programming.

Moreover, it can also be used to better communicate the result of compiler analysis to the user. When a sophisticated IDE applies and/or suggests refactoring transformations, it is important for the programmer to be able to establish the correspondence between the original and the transformed code. Even if a compiler is capable of automatically transforming a loop nest to expose parallelism, the resulting code may be complicated and different from the original, making it difficult to understand and/or to maintain for the programmers. We hope that our observation can contribute to making the link easier by retaining most of the original structure intact.

The rest of this paper is organized as follows. Section 2 introduces the necessary background about `X10` clocks, and notations used in the paper. Our observation is illustrated through examples in Section 3, followed by discussions in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. Background

In this paper, we extensively use *clocks* in `X10`, a synchronization mechanism that resembles barriers. We introduce the minimal set of `X10` constructs needed for our examples. The main idea of clocks is that all activities *registered* to a clock blocks on a call to `advance` until all other activities registered to the same clock also call `advance`. This separates the operations within an activity into phases, adding orderings between operations in addition to those given by `finish`. Clocks may be dropped (unregistered) by an activity at any time, and are automatically dropped when an activity terminates.

The example in this paper uses the following constructs:

- `clocked finish S`: `finish` waits for all activities spawned in `S` to terminate; `clocked finish` is a variant that create an unnamed clock, and registers itself to it.
- `async S`: `async` spawns a new activity to execute `S`; `clocked async` is a variant where the spawned activity is registered to the clocks that the parent activity is registered with. In this paper, all `async`s are clocked, and the `clocked` keyword is omitted.

- `advance`: In this paper, each code fragment will have at most one clock. We use `advance` to denote the call to `advance` to the corresponding implicit clock.
- `for` loops: While the full language supports loops over other expressions, we limit ourselves to loops with integer iterators.

In addition, we omit the outermost `clocked finish` that surrounds the code fragment in our examples. The parenthesization of blocks may also be omitted when indentation can serve as a clear separation of code blocks.

Formalisms for Loop Transformations

This paper also uses some concepts and notations from the literature of loop transformation frameworks. Both the unimodular framework [2] and the polyhedral model [3] reason about dynamic instances of statements within a loop as a set of integer points. For example, the following loop nest:

```
for (i = 1 .. N)
   for (j = 1 .. M)
      S
```

is represented as a statement over an integer set, characterized by a polyhedron: $\{i, j | 1 \leq i \leq N \wedge 1 \leq j \leq M\}$.

Loop transformations are expressed as a function of the loop iterators. Permuting the two loops, for instance, is expressed by the function $(i, j \rightarrow j, i)$. These functions can also be viewed as the following affine transformation matrix: $A.\vec{z} + \vec{b}$ where A is a matrix representing linear combinations of $\vec{z}$, and $\vec{b}$ is the constant offsets.

Unimodular framework focus on loop transformations, and restricts the A matrix to be unimodular (i.e., the function is bijective). In the polyhedral model, these functions are also used to represent *schedules*, a mapping from iterations to time stamps, which is not bijective if there is parallelism.

3. Examples

In this section, we present examples of traditional loop transformations expressed with clocks.

3.1 Loop Skewing

Loop skewing is perhaps the most illustrative transformation of our ideas. The main goal of this transformation is to expose wave-front parallelism as a `forall` loop. Figure 1 shows a classical application of loop skewing. Note that the dependences are expressed as dataflow arrows from the producer to the consumer.

Although it may not seem overly complicated with this example, and systematic methods for applying loop skewing is well known, it still involves projection (i.e., Fourier-Motzkin elimination), and updates to all expressions involving the loop indices. When applied to higher dimensional loops with many operations, the resulting code may look very different from the original one.

In Figure 2, we illustrate how clocks can be used to express the same parallelism, with far less changes to the code. The only significant transformation is the loop interchange needed to make the `j` loop outermost. This interchange cannot be avoided, since the parallelization by `forall` loops place the "time" loop that governs synchronizations outside of the parallel loop, while clocks work in the exact opposite manner; outer parallel, synchronize inside.

Once the loops are interchanged, the skewing can easily be realized by placing `advance` statements to control the rate in which the activities are spawned. Figure 2a delays the spawn of the `j`-th activity by waiting on an `advance` after each `async`. The main activity must wait for an operation of the `i` loop to be performed by the already spawned activities to spawn another one. This effectively adds a delay of `j` to each activity, which corresponds to the affine transformation $(i, j \rightarrow i + j, j)$.

When the code is written in this form, skewing by different functions is only a matter of placing more/less `advance` statements. For instance, adding one extra `advance` implements skewing by $(i, j \rightarrow i + 2j, j)$ as shown in Figure 2b.

If the goal is to express the same wave-front parallelism, it is also possible to keep the loop structure intact as illustrated in Figure 3a. The transformation $(i, j \rightarrow i, i + j)$ exposes the same wave-front but the parallelism is associated with the outer loop, preventing its parallelization by `forall` loops. The outer parallelism suits more naturally with clocks, avoiding the need for loop interchange.

3.2 Loop Fusion/Fission

Merging and separation of loops is another common loop transformation. Although applying these transformations may improve performance, the program before fusion/fission may correspond better to the natural view of the computation. For instance, if two different filters are applied to the same image, loop fusion may improve locality, but now the implementation of the two filters are mixed into one loop.

Furthermore, fusing loops may accompany additional transformations for legality reasons. The example depicted in Figure 4 requires shifting the latter loop by one, which necessitates duplication of statements for prologue of the merged loop as well as updates to indexing expressions. Although the extent of modification may not seem significant with this example, the amount of required changes quickly increases for more complicated programs. For example, fusing two loop nests that implement two different image filters is not a trivial task.

It is possible to express the merged order of execution using clocks. The main idea is to execute the two loops in "parallel" with tight synchronizations that effectively implements a sequential schedule. Figure 4c is an example that correspond to applying loop fusion to Figure 4a. The sequence of two `advance` statements are necessary to enforce sequential execution of the two loops. If both of them are replaced by a call to `advance`, the two statement instances, $S1\langle i \rangle$ and $S2\langle i - 1 \rangle$, will be in parallel.

Loop fission is a natural application of clocks, as it splits a body of a loop into multiple phases. Figure 5 illustrates loop fission with clocks. This may be the most common use of clocks (and barriers) that we have today.

This example is similar to the loop fusion example by Zhao et al. [10] where a program of the form Figure 5b is transformed into that of Figure 5c. The motivation in their case was to reduce the amount of task creation and its related overhead, by replacing them with synchronizations that is expected to be cheaper.

3.3 Software Pipelining

Another important optimization involving loops is software pipelining. It can be used at a coarse granularity for pipelined parallelism of tasks, and fine-grained pipelining is one of the major sources of performance for hardware accelerators (e.g., FPGAs), which is receiving increased attention due for its low-power capabilities. While it is typically left to the compiler to exploit fine-grained parallelism, it is also interesting to be able to precisely specify the pipelining a designer seeks.

Figure 6 illustrates how clocks can simplify the specification of pipelined parallelism. It can be seen as a special case of loop skewing, where the different stages of the pipeline are in a fully unrolled loop. The number of `advance` statements can control the delay of each stages or the rate in which new instructions are fed.

4. Discussion

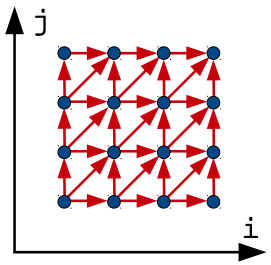
In this section we first make a preliminary attempt towards formalizing and characterizing the idea behind our observations. We then discuss several directions where the idea may further be extended.

```
for (i = 1 .. N)
   for (j = 1 .. N)
      h[i][j] = foo(h[i-1][j],
                    h[i-1][j-1],
                    h[i  ][j-1]);
```

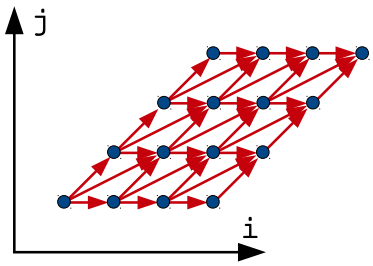
(a) Original loop nest. Neither `i` nor `j` loop is parallel.

```
for (i = 1 .. 2N-1)
   forall (j = max(1,i-N) .. min(N,i-1))
      h[i][j] = foo(h[(i-j)-1][j],
                    h[(i-j)-1][j-1],
                    h[(i-j)  ][j-1]);
```

(b) After skewing. `j` loop is now parallel.



(c) Geometrical view of the original iteration space. There are dependences along `i` and `j` loops preventing parallelization.



(d) Geometrical view of the "skewed" iteration space. There is no vertical dependence, allowing `forall` parallelization.

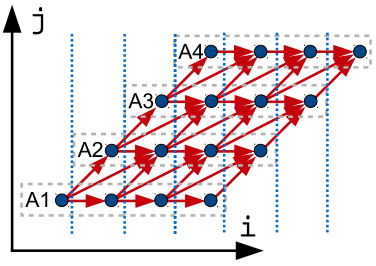
Figure 1: Example of skewing with traditional loop transformation. The transformation can be expressed as $(i, j \to i + j, j)$ in the unimodular/affine framework. Skewing requires the loop bounds as well as the array accessing expressions to be modified.

```
for (j = 1 .. N) {
   async
      for (i = 1 .. N) {
         h[i][j] = foo(h[i-1][j],
                       h[i-1][j-1],
                       h[i  ][j-1]);
         advance;
      }
   advance;
}
```

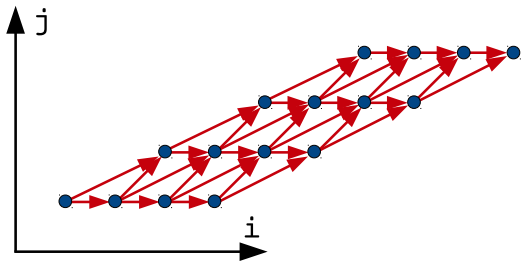
(a) Skewing realized by clocks. Note that the loops bounds and statements stay exactly the same as the original code.

```
for (j = 1 .. N) {
   async
      for (i = 1 .. N) {
         h[i][j] = foo(h[i-1][j],
                       h[i-1][j-1],
                       h[i  ][j-1]);
         advance;
      }
   advance; advance;
}
```

(b) Skewing for a different wave-front. The only difference is an extra `advance` at the end of the `j` loop.



(c) Visualization of execution with clocks. Horizontal boxes denote operations by an activity, and vertical bars depict clock synchronizations.

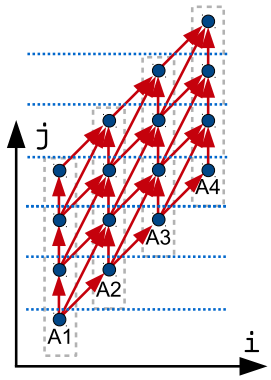


(d) The result of adding one additional `advance`. This corresponds to skewing by $(i, j \to i + 2j, j)$.

Figure 2: Example of skewing the code in Figure 1a with clocks. Clocks are used to delay the start of activities to implement the execution order that corresponds to skewing. The loops are interchanged since the parallelization is now outer-parallel with synchronizations instead of parallelizing the innermost loop.

```
for (i = 1 .. N) {
   async {
      for (j = 1 .. N) {
         h[i][j] = foo(h[i-1][j],
                       h[i-1][j-1],
                       h[i  ][j-1]);
         advance;
      }
   }
   advance;
}
```

(a) Another realization that preserves loop structure.



(b) The execution order of the code in Figure 3a.

Figure 3: Realizing skewing without the need to interchange the loops. This implementation preserves the same wave-front parallelism, but corresponds more closely to affine transformation by $(i, j \rightarrow i, i+j)$.

```
for (i = 1 .. N)
   x[i] = S1(i);
for (j = 1 .. N-1)
   ...  = S2(x[j+1]);
```

(a) Original loops.

```
x[1] = S1(1);
for (i = 2 .. N)
   x[i] = S1(i);
   ...  = S2(x[i]);
```

(b) Loop fusion after shifting the latter loop.

```
async
   for (i = 1 .. N)
      x[i] = S1(i);
      advance; advance;
async
   advance;
   for (j = 1 .. N-1)
      ...  = S2(x[j+1]);
      advance; advance;
```

(c) The same order of execution via clocks.

Figure 4: Loop fusion with clocks. The original loops can be fused if the latter loop is shifted by one. This creates a peeled statement for prologue, and renaming the indexing expression for `S2` in Figure 4b. By using clocks, the original loop structure can be maintained, while the same order of execution is realized through tight synchronizations.

```
forall (i = 1 .. N)
   S1;
   S2;
```

(a) Original loop.

```
forall (i = 1 .. N)
   S1;
forall (i = 1 .. N)
   S2;
```

(b) Traditional loop fission.

```
for (i = 1 .. N)
   async
      S1;
      advance;
      S2;
```

(c) Loop fission via clocks.

Figure 5: Loop fission with clocks. The original loop is separated into two loops in Figure 5b. The same effect can be achieved by splitting the two statements into different phases of the clock as shown in Figure 5c.

```
for (i = 1 .. N)
   S1(i);
   S2(i);
   S3(i);
```

(a) Original loop.

```
//prologue
S1(1)
S1(2); S2(1)

for (i = 1 .. N-2, 3)
   S1(i+2); S2(i+1); S3(i);
   S1(i+3); S2(i+2); S3(i+1);
   S1(i+4); S2(i+3); S3(i+2);

//epilogue
S2(N); S3(N-1)
       S3(N)
```

(b) Traditional software pipelining.

```
for (i = 1 .. N) {
   async S1;
   async {advance; S2}
   async {advance; advance; S3}
   advance;
}
```

(c) Pipelining with clocks.

Figure 6: Software pipelining with clocks. The traditional transformation employs loop unrolling, considerably complicating the code. With clocks, expressing pipelined parallelism can be made very concise, and details such as initiation intervals can be tweaked with little effort.

4.1 Composition of Transformations

One of the strengths of the well studied frameworks for loop transformations is the ability to compose transformations before applying them. The transformations are reasoned in terms of its representations as unimodular matrices or affine functions, and the result of the composition of these representations are finally applied. It is therefore important to characterize the family of transformations and its compositions that can be expressed with clocks.

4.1.1 One-Dimensional Schedules

As a first step, we show that it is possible to express all transformations that corresponds to a subset of one-dimensional affine schedules that consists of non-negative coefficients. More precisely, it is the loop transformations that is a result of unimodular completion [5] of such class of one-dimensional schedules.

Under this constraint, the schedules can be characterized by a vector in the first orthant of the iteration space. The time steps denoted by the schedule correspond to the hyper-planes normal to this vector. For instance, the 45 degrees wave-front of a 2-dimensional iteration space, $\langle i, j \rangle$, can be expressed as a schedule $(i, j \rightarrow i+j)$ or by the vector $[1\ 1]$, which is normal to $i+j$.

The problem can be seen as generating a code that respects a one-dimensional schedule that is a non-negative combination of the loop indices. Then the systematic approach is to first add `async`s for each loop that covers its body. With a triply nested loop, the result is the following:

```
for (i = ...)
  async for (j = ...)
    async for (k = ...)
      async S;
```

Then you may delay the spawn of the activities by $c \times x$ phases by adding c calls to `advance` in the body of loop x where c is the corresponding element of the vector. The combination of such calls to `advance` at all levels corresponds to the linear function. For example, the following code implements the schedule $(i+j+k)$, which is characterized by the vector $[1\ 1\ 1]$:

```
for (i = ...)
  async for (j = ...)
    async for (k = ...)
      async S;
      advance;
    advance;
  advance;
```

Since the newly spawned activity starts from the current clock phase, each term of the linear schedule can be implemented separately at each loop level. The dynamic nature of the clocks makes deadlocks not an issue. The number of `advance` statements at each activity can be different (which would case a deadlock in the case of barriers) simplifying the argument for validity of such a translation.

The above systematic method applies to all one-dimensional schedules with non-negative coefficients. This is because the corresponding vector is also guaranteed to be non-negative, and thus can be expressed by adding `advance` statements which will always increase the phase count. There are multiple unimodular completions for a schedule as illustrated earlier in Figure 3. In such cases, there exists at least one completion of the schedule that preserves the original loop structure, which is the one derived by the approach above. Other completions corresponding to other loop transformations that use the same wave-front (schedule) may require permutations as shown in Figure 2.

We do not fully treat imperfectly nested loops and multiple statements, but some of such cases are merely an unrolled version of a loop nest with constant bounds. The pipelining example in Figure 6 can also be viewed as a loop nest of the following form:

```
for (i = 1 .. N)
   for (s = 1 .. 3)
      async S(i,s)
      advance;
   advance;
```

The non-negative constraint clearly limits some loop transformations such as loop reversal. For instance, loop reversal is not possible. This is because the clocks are only capable of *delaying* the operations, and are not adequate for reverting the order. It is possible to add an extra loop to mimic the reversed order:

```
for i = 1 .. N
   async {
     for (j = N-i .. N) advance;
     S;
   }
```

However, this does not meet our goal to keep the original structure.

Note that the systematic approach is presented as part of the argument to show that (non-negative) one-dimensional affine schedules can always be expressed with clocks. The result of the presented approach is likely to be much more complicated than needed, and is not intended to be used in practice.

4.1.2 Multi-Dimensional Schedules

Although clocks can be used to express certain types of one-dimensional schedules, it is not sufficient to express multi-dimensional schedules without considerably altering the loop structure.

There are some cases where transformations of loops that do not have a 1D schedule can be expressed in degenerate cases (e.g., inner loop nest has 1D schedule, surrounded by sequential loops). The fusion example (Figure 4) is actually an instance of a program with multi-dimensional schedule. It is still possible for this code fragment to be captured with clocks, since the second dimension of the schedule is of constant length.

It may still be possible to express multi-dimensional schedules of some form by using multiple clocks, but the complexity of such representation may undermine the whole idea of having simpler and less invasive way to express loop transformations.

4.2 Region Arrays

X10 provides `Region` as a way to represent set of points to iterate over as a separate data structure not directly tied with loops or arrays. Using regions, one can easily iterate over triangular regions or bands of matrices, and even more complicated iteration spaces by applying operations on regions.

All the points in a region is ordered by the lexicographic order of the dimensions. Some loop transformations (e.g., permutation) can easily be specified as a different order of traversing the set of points in a region. The Chapel[1] language supports user-defined iterators for visiting points, which may also be interesting for `X10` regions. Permutation of a 2D sequential loop requires multi-dimensional schedules, and is difficult to express with clocks, but is simple to express as a different order of traversing the region.

Combined with orthogonal specification of the execution order of the regions, it may be possible to further broaden the space of transformations that can be expressed without changing the original loop structure.

[1] http://chapel.cray.com/

4.3 "Equivalence" of the Expressed Parallelism

One remark needs to be made regarding the "equivalence" of the parallelism. In our terms, the following two code fragments express the "same" parallelism.

```
for i                          for j
   finish for j                   async for i
      async S;                       S; advance;
```

The difference is that one has the parallelism as the inner loop, and the other has it as the outer loop. One might argue that the continuity of the activities is an important part of the input specification, which is lost in the inner parallel version.

This distinction is not made in the presented observations. We consider the parallelism in Figure 2c and Figure 3b to be equivalent, although they can be argued to have different parallelism based on the mapping of iterations to activities (e.g., the work by Zhao et al. [10]). This is certainly another instance of the "concerns" that is better if expressed orthogonally to the specification itself.

5. Related Work

Barrier synchronizations in commonly used APIs for parallelism are typically much less flexible than clocks. OpenMP barriers cannot be placed within work sharing constructs (e.g., `parallel for`), and MPI barriers operates over pre-determined set of processes. The dynamicity of `X10` clocks, where the set of participating activities changes at run-time, is an important element that enables our approach. In this section, we discuss the related work on transformations of parallel programs written in `X10`.

5.1 Efficiency Oriented Transformations

Shirako et al. [9] and Zhao et al. [10] have explored loop transformations of parallel programs written in `X10`. Their goal is to improve the efficiency of the parallel program through transformations including traditional loop transformations, which poses new challenges such as legality of transformations.

Feautrier et al. [4] showed that, with current implementation of the `X10`, the overhead of clocks may exceed that of repeated `finish/async` blocks and proposed a technique to remove clocks.

Our proposed way of expressing traditional loop transformations with clocks is certainly not designed for efficiency. We expect that the performance to be significantly worse than that of an `X10` implementation that closely follows traditional implementations. These work on improving the efficiency of parallel `X10` programs, especially the work on clock removal [4], complements our work by establishing the necessary link between program structures suited for the programmers and efficient but cryptic implementations.

5.2 Refactoring of Parallel Loops

The need for Integrated Development Environment for parallel programming, and in particular for sophisticated refactoring transformations, has been argued by several groups in the past [6, 8]. Although `X10` provides an IDE based on Eclipse, it is not yet equipped with many refactoring transformations specific to `X10`.

Markstrum et al. [7] explored some initial steps for concurrency refactoring in `X10`. Our work is aimed at such interactions with the programmer by providing alternative ways to express the traditional loop transformations.

6. Conclusion

We have presented a perspective on how `X10` clocks may ease the expression of certain forms of (fine-grained) parallelism. It is based on our observation that some loop transformations can be expressed with clocks without modifying the original loop structure and statements. When familiarized with the subtleties of `X10` clocks, we believe that it enables much cleaner and/or easier expression of some of the traditional loop transformations. Although this paper is primarily motivated by fine-grained parallelism, some of the transformations (e.g., skewing) is equally important for coarse-grained, inter-node, parallelism.

This paper merely presents an observation and the ideas behind, leaving the question if the proposed approach is indeed easier and/or more programmer-friendly as we claim unanswered. The clocks in `X10` seems to be a feature that is not extensively used for the moment, and it takes some "twisting" of heads even for experts in loop transformations to understand the idea. We hope that this paper can provide a fresh/unique (and perhaps wild and crazy) look at loop transformations as well as `X10` clocks.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[2] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[3] P. Feautrier and C. Lengauer. Polyhedron model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, 2011.

[4] P. Feautrier, É. Violard, and A. Ketterlin. Improving the performance of X10 programs by clock removal. In *Proceedings of the 23rd International Conference on Compiler Construction*, CC '14, pages 113–132, Apr. 2014.

[5] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 106–111, 1998.

[6] K. Kennedy, K. S. Mckinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[7] S. A. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for X10. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 303–304, 2009.

[8] J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for fortran and high-performance computing. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 37–39, 2005.

[9] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 181–192, 2009.

[10] J. Zhao, J. Shirako, V. K. Nandivada, and V. N. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 169–180, 2010.