Xfor: Semantics and Performance

Eric Violard, Philippe Clauss, Imen Fassi

Xfor: Semantics and Performance

Éric Violard[1], Philippe Clauss[1], and Imen Fassi[1,2]

[1] Team CAMUS, INRIA - University of Strasbourg, France,
[2] University of Tunis El-Manar, Tunisia
{Eric.Violard,Philippe.Clauss,Imen.Fassi}@inria.fr

Abstract. This paper introduces a new programming control structure called "xfor" as an extension of the classical "for" construct in C. It is designed to help one programmer to improve data locality on multi-core architectures by allowing him to express the schedule of instructions in an abstract way. This schedule is defined geometrically by mapping the iteration domains relatively to each other onto a unique referential by using specific parameters called grain and offset. A semantic framework is presented which associates a precise meaning with this syntactic construct and serves as a base for applying reliable xfor code transformations and programming strategies. These issues are illustrated with the Red-Black algorithm. Performance measurements carried out with benchmarking programs rewritten by using the xfor construct show significant execution times speed-ups.

1 Introduction

While parallel architectures are more and more complex and unpredictable, many researches focus in proposing new languages supposed to facilitate their programming [1–4]. However, most of these languages imply to change drastically programmers habits and have weak chances to be adopted by the software industry [5]. An interesting idea to bypass this problem is to gradually extend mainstream languages with new programming control structures that are derived from already existing ones, thus giving developers the opportunity to enlarge their way of reasoning while programming.

Efficient implementations for challenging programs require a combination of high-level algorithmic insights and low-level implementation details. Deriving the low-level details is a natural job for the compiler-computer couple (CCC), but it can not replace the human insight. Therefore, one of the central challenges for programming languages is to establish a synergy between the programmer and the CCC, exploiting the programmer's expertise to reduce the burden on the CCC. However, programmers must be able to provide their insight effortlessly, using programming structures they can understand. In order to reach simultaneously low-level efficiency and programmability, a solution is to assist programmers with automatic code transformations that translates into an efficient program what they express at their level of understanding. In the last decades, many efforts have been made in providing automatic program optimization and parallelization software tools that analyze and transform source codes. However, such approaches address three major challenges: fully automatic analysis and transformation is highly complex and can never surpass the human insight, programming control structures of current mainstream languages are almost never addressing directly the main performance issues of current computers, while super effective codes cannot reasonably be written by programmers due to their very convoluted shapes.

Following this idea, we propose a computer-assisted control structure called "xfor", allowing programmers to address directly and accurately a main issue regarding performance: the locality of data accessed in compute-intensive loops. Two parameters in this structure, the "offset" and the

"grain", afford to adjust precisely data reuse distances in a natural programming context, while a source-to-source translator, called IBB for Iterate-But-Better, is in charge of generating the final convoluted, but very efficient, code. The IBB xfor support tool takes additionally advantage of optimizations implemented in the polyhedral code generator CLooG [6] which is used by IBB to generate from xfor-loops equivalent for-loop code. However, such an assisted programming approach requires complete confidence on the performed translation that must express exactly what programmers have in mind while writing their source codes.

Thus, computer-assisted control structures, as xfor, must be associated with a precise and unambiguous meaning. Obviously, reasoning on such a syntactic construct, i.e., transforming it in order to obtain an equivalent but more efficient version, suppose there exists a way to prove that two versions have the same effect where as a different behaviour. Therefore, a semantics definition is presented that resolves all such semantic issues.

The paper is organized as follows. Section 2 provides an intuitive description of the xfor construct. The underlying concepts are presented and illustrated with the Gauss-Seidel Red-Black problem implemented using the xfor loop structure. This section also introduces the general syntax definition of well-formed xfor statements. Section 3 presents the mathematical meaning of these well-formed statements by using a denotational approach. This denotational semantics formally defines the instructions' scheduling which is used in a xfor code. Section 4 presents the proof of semantic equivalence for the xfor codes of section 2. Based on this semantics, we describe the relevant xfor-programming strategies in section 5. Significant performance improvements are also reported in this section using a set of benchmarks built from re-writing codes of the polybench benchmark suite [7]. Conclusions are given in section 7.

2 An intuitive description

The xfor control structure allows one programmer to gather several nested for loops. Since any loop nest of a given depth can be transformed into a loop nest of deeper level by inserting some one-iteration loops, without loss of generality, we consider from now on a set of loop nests having all the same depth: later on we refer to as k, the number of used loop nests and as n, the depth of these loop nests. Instead of writing such k loop nests one after the other, thus defining a schedule by default of the instructions inside the loop nest bodies, the programmer can write a single xfor statement and then focus on expressing in abstract way an adequate schedule amongst all the ones that can be defined by using xfor specific parameters called *grain* and *offset*.

This schedule is defined geometrically: each instruction is mapped onto a point in a n-dimensional geometric space such that this point can be specified as an element of $\mathbb{Z}^n$. The elements of $\mathbb{Z}^n$ corresponding to instructions compose a domain called the *referential domain*. The schedule of the instructions is induced from the lexicographic total order on $\mathbb{Z}^n$: the instructions are executed in the lexicographic increasing order of the corresponding elements of $\mathbb{Z}^n$.

The instructions are mapped onto $\mathbb{Z}^n$ in two steps and for each original loop nest separately: first, the instructions of the loop nest are mapped onto its *iteration domain*, which is a subset of $\mathbb{N}^n$; then, the elements of the iteration domain are "moved" across $\mathbb{Z}^n$, i.e, scattered and shifted – according to the geometric interpretation – by using the grain and offset in order to define their final locations and consequently, their schedules.

Let us consider the two loop nests on the left of Fig. 1. They compose the standard code for the Red-Black Gauss-Seidel problem and describe the computation of the elements of a $N \times N$ matrix

(omitting the border elements initialization). We will next refer to these loop nests as the first and the second loop nest implemented by the xfor structure shown on the right of Fig. 1.

```
// Red phase
for(i0=1;i0<N-1;i0+=1) {
  for(j0=1;j0<N-1;j0+=1) {
    if((i0+j0)%2==0) u[i0][j0] = f(u[i0][j0+1],
       u[i0][j0-1],u[i0-1][j0],u[i0+1][j0]); }}
// Black phase
for(i1=1;i1<N-1;i1+=1) {
  for(j1=1;j1<N-1;j1+=1) {
    if((i1+j1)%2==1) u[i1][j1] = f(u[i1][j1+1],
       u[i1][j1-1],u[i1-1][j1],u[i1+1][j1]); }}
```

```
// code xfor1 (Red and black together)
xfor(i0=1,i1=1; i0<N-1,i1<N-1; i0+=1,i1+=1; 1,1; 0,1) {
 xfor(j0=1,j1=1; j0<N-1,j1<N-1; j0+=1,j1+=1; 1,1; 0,0) {
  0 : if((i0+j0)%2==0) u[i0][j0] =
        f(u[i0][j0+1],u[i0][j0-1],
          u[i0-1][j0],u[i0+1][j0]);
  1 : if((i1+j1)%2==1) u[i1][j1] =
        f(u[i1][j1+1],u[i1][j1-1],
          u[i1-1][j1],u[i1+1][j1]); }}
```

Fig. 1. Loop nests implementing the Red-Black algorithm (left) embedded into one xfor code (right)

Fig. 2 shows a geometric representation of this standard code in which each instruction of one loop nest is represented by a point in a sub-domain of $\mathbb{N}^2$, i.e., the *iteration domain*. In this interpretation, the point of coordinates (i, j) identifies the instruction executed at iteration (i, j), i.e., at the $(i+1)$-th iteration of the outer loop and the $(j+1)$-th iteration of the inner one. We will later denote an instruction of the first (resp. the second) loop nest by $S_1[(i, j)]$ (resp. $S_2[(i, j)]$). Each red (resp. black) point represents an instruction of the first (resp. second) loop nest. An instruction depicted as a light point is equivalent to a NOP instruction, i.e., no operation, whereas any of the other instructions depicted as a dark point performs actual computations: it computes a value and assigns it to element $(i+1, j+1)$ of matrix `u` after having read its North-South-East-West neighbors.

Note that, for each of the two loop nests, the iteration domain, i.e, $\{(i, j) \mid 0 \leq i, j < N-2\}$, is different from the domain of the index variables' values, i.e, $\{(i, j) \mid 1 \leq i, j < N-1\}$. Indeed, the iteration domain does not depend on the values of the index variables. It only depends on the number of iterations in each of the nested loops. We thus have two domains denoted by D_1 and D_2,

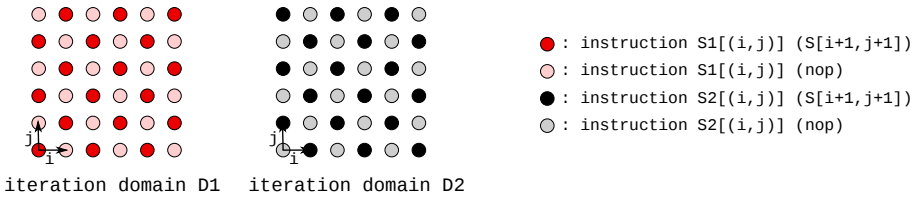

Fig. 2. Iteration domains for $N = 8$

one for each of the loop nests. It is then possible to define a schedule of the instructions by indicating how these two geometric domains are located one relatively to the other. This can be achieved by mapping these domains onto a common domain called the *referential domain*. For example, the black domain could be placed right beside the red one as drawn on the left of Fig. 3. In this case, the instructions are executed in the same order as the standard code since this order corresponds to the lexicographic order on $\mathbb{Z}^2$ represented by a dotted line. For example, for $N = 8$, instruction $S_2[(0, 0)]$ at point $(6, 0)$ in the referential domain is executed just after instruction $S_1[(5, 5)]$ at point $(5, 5)$. Another more interesting arrangement – because it has an impact on correction and performance as well – is obtained from the previous one by moving the black domain to the left so that it is shifted by one position to the right relatively to the red domain as drawn on Fig.3 (right).

In this case, a red instruction and a black one may be mapped onto the same point in the referential domain and we state, as the xfor semantics formally says, that the instruction of the first loop nest, i.e., the red one, is executed before the other one.

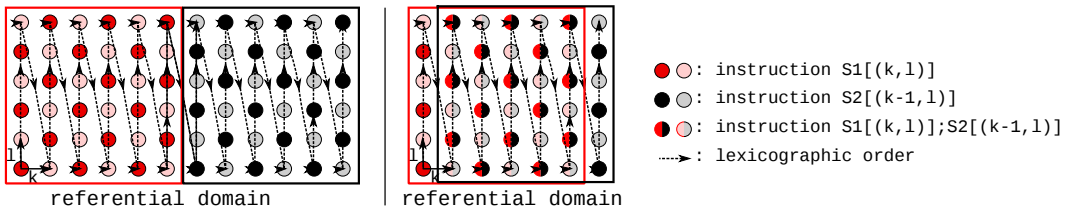

Fig. 3. Referential domains and lexicographic order

The xfor allows one programmer to specify such a mapping of each iteration domains onto the referential domain by using one grain and one offset for each dimension. The grain and offset denote integer numbers that are respectively multiplied by and then added to the coordinate of each point in the iteration domain to give the coordinate in the referential domain. A positive grain dilates the iteration domain and an offset shifts it onto the referential domain as depicted on Fig. 4.

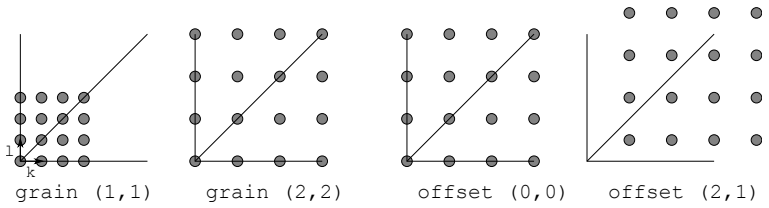

Fig. 4. Grain and offset

The mapping of Fig. 3 (right) relates to the xfor code in Fig. 1, while the other mapping on Fig. 3 (left) relates to the same code except that underlined $\underline{1}$, i.e., the i-axis offset, has to be replaced by `N-2`. The xfor syntactic construct matches the following general form:

```
xfor( (v_i^1=a_i^1)_{i=1..k};(v_i^1<b_i^1)_{i=1..k};(v_i^1+=c_i^1)_{i=1..k};(g_i^1)_{i=1..k};(o_i^1)_{i=1..k}) {
    xfor( (v_i^2=a_i^2)_{i=1..k};(v_i^2<b_i^2)_{i=1..k};(v_i^2+=c_i^2)_{i=1..k};(g_i^2)_{i=1..k};(o_i^2)_{i=1..k}) {
        ...
            xfor( (v_i^n=a_i^n)_{i=1..k};(v_i^n<b_i^n)_{i=1..k};(v_i^n+=c_i^n)_{i=1..k};(g_i^n)_{i=1..k};(o_i^n)_{i=1..k}) {
                0    : S_1
                1    : S_2
                    ...
              <k-1> : S_k } ... } }
```

where the index range $1..k$ is the same all across the construct, $k, n \in \mathbb{N}^*$ are respectively the number of original loop nests and the loop nest depth, v_i^l ($i = 1..k$, $l = 1...n$) are the *index variables*. They are all distinct and of type `int`, while a_i^l, b_i^l, c_i^l, g_i^l and o_i^l are expressions of type `int`. Each S_i is any statement – eventually containing a xfor construct – not containing a `break` or `continue` statement, and whose nearest enclosing statement is a xfor. Statement S_i is labelled by an integer literal denoted by $\langle i \rangle$ which indicates that S_i is the body of the so called i-th loop nest. Each g_i^l

(resp. o_i^l) is called a *grain* (resp. an *offset*). Since the xfor construct is designed as an extension of C, the current typing mechanism of C is used to check that all these sub-expressions have the right type. Note that, for now, the IBB compiler does not accept non-affine offsets or grains, i.e., each of these expressions must be a linear (affine) function of the surrounding index variables.

Now, let us show how the xfor construct can be used. When handling several iteration domains, the first step is to identify dependences that are occurring between them. The Red-Black example obviously defines two dependent iteration domains: the red and the black domains. Each black point depends on its four North-South-East-West red neighbors. Both domains can be scheduled such that any black point can be computed as soon as all four red points from which it depends have been computed. It comes that according to the lexicographic order, any black point can be computed as soon as its eastern neighbor is available. Hence, the corresponding xfor code, i.e., `xfor1` in Figure 1, is equivalent to the standard code where data reuse distances have been minimized. However, the xfor code of Fig. 1 contains useless guards to test the parity of $(i + j)$. These conditionals yield empty iterations that can be removed by translating the conditionals into 2-grain parameters, 2-increments and convenient offsets, and by splitting each of the red and black domains into two red and two black domains, defined respectively by i *modulo* $2 = 0$ and i *modulo* $2 = 1$. The resulting xfor code, i.e., `xfor2`, is shown in Figure 5 (left), where statement blocks 0 and 1 are associated with the red domain.

```
// code xfor2 (without guards)
xfor(i0=1,i1=2,i2=1,i3=2; i0<N-1,i1<N-1,i2<N-1,i3<N-1;
       i0+=2,i1+=2,i2+=2,i3+=2; 2,2,2,2; 0,1,1,2) {
  xfor(j0=1,j1=2,j2=1,j3=2; j0<N-1,j1<N-1,j2<N-1,j3<N-1;
         j0+=2,j1+=2,j2+=2,j3+=2; 2,2,2,2; 0,1,0,1) {
    0 : u[i0][j0] = f(u[i0][j0+1],u[i0][j0-1],
          u[i0-1][j0],u[i0+1][j0]);
    1 : u[i1][j1] = f(u[i1][j1+1],u[i1][j1-1],
          u[i1-1][j1],u[i1+1][j1]);
    2 : u[i2][j2] = f(u[i2][j2+1],u[i2][j2-1],
          u[i2-1][j2],u[i2+1][j2]);
    3 : u[i3][j3] = f(u[i3][j3+1],u[i3][j3-1],
          u[i3-1][j3],u[i3+1][j3]); }}
```

```
// code xfor3 (flexible offset)
xfor(i0=1,i1=2,i2=1,i3=2; i0<N-1,i1<N-1,i2<N-1,i3<N-1;
       i0+=2,i1+=2,i2+=2,i3+=2; 2,2,2,2; 0,1,k+1,k+2) {
  xfor(j0=1,j1=2,j2=1,j3=2; j0<N-1,j1<N-1,j2<N-1,j3<N-1;
         j0+=2,j1+=2,j2+=2,j3+=2; 2,2,2,2; 0,1,0,1) {
    0 : u[i0][j0] = f(u[i0][j0+1],u[i0][j0-1],
          u[i0-1][j0],u[i0+1][j0]);
    1 : u[i1][j1] = f(u[i1][j1+1],u[i1][j1-1],
          u[i1-1][j1],u[i1+1][j1]);
    2 : u[i2][j2] = f(u[i2][j2+1],u[i2][j2-1],
          u[i2-1][j2],u[i2+1][j2]);
    3 : u[i3][j3] = f(u[i3][j3+1],u[i3][j3-1],
          u[i3-1][j3],u[i3+1][j3]); }}
```

Fig. 5. A xfor code without useless guards (left) and the same one but with variable offset (right)

For sake of performance, one programmer can also make his code more flexible, thus allowing the compiler to balance different goals according to specific architecture features. Since the programmer knows that programs are equivalent whatever is the mapping, provided the black domain is located shifted by at least one position to the right, he can rewrite its code and get the xfor code in Fig. 5 (right) in which a positive variable `k` is used to define the i-axis offset.

Thanks to the abstraction it provides, a xfor code is easier to transform than its equivalent code written using classical for-loops. For example, the length of the for-loop code generated by the source-to-source translator IBB from the xfor code in Figure 5 (left) is about 30 lines of code to be compared with the 6 lines in the xfor version.

3 Xfor denotational semantics

Amongst the approaches to define the meaning of programs using mathematics (operational or axiomatic semantics as examples), denotational semantics [8, 9] is particularly well-suited for designing code transformations. From our point of view, although an operational semantics could give

some insight, a denotational semantics is better suited to the xfor construct because the purpose of this construct is mainly to define a function that maps instructions onto a geometric domain. It is clear that the operational behaviour consists in executing these instructions in the lexicographic order on $\mathbb{Z}^n$. Moreover, this denotational approach does not require to specify the steps for building this mapping, which enables various implementations.

3.1 Some preliminaries

Memory states We define memory states in a classic way, excepting that we introduce an *access permission*: a memory state is defined by an *environment* that maps each declared identifier to a memory location, and a *store* that maps each used location to a value. We assume some properties about the store that make it an abstraction of the physical memory of a computer: namely, each location is large enough to contain any storable value and the set of locations is infinite, which means that we do not deal with the edge-case of running out of memory. Formal semantics is usually not impacted by implementation restrictions. In addition, we associate each location with an access permission indicating if it is read-only. This is used to restrict accesses to index variables and to prevent some instructions inside a loop nest body to access the memory locations corresponding to an index variable of another loop nest inside a xfor structure.

In the following, we denote by ID the non-empty set of all identifiers, LOC the non-empty and totally ordered set of locations[3], VAL the non-empty set of values, ENV the set of environments, $STORE$ the set of stores, $ACCESS$ the set of access permissions, and finally $STATE$ the set of memory states. The sets are defined as follows:

$$ENV \equiv ID \rightharpoonup LOC \quad STORE \equiv LOC \rightharpoonup VAL \times ACCESS$$
$$STATE \equiv ENV \times STORE \quad ACCESS \equiv \{ro, rw\}$$

where $\rightharpoonup$ builds a set of partial functions.

A memory state, say s, is then a pair (env_s, sto_s) where the environment env_s is a partial function from ID to LOC whose domain, $dom(env_s)$, is the set of declared identifiers. The store sto_s is a partial function from LOC to $VAL \times ACCESS$ whose domain, $dom(sto_s)$, is the set of used locations. If the store maps a location to the access permission ro, it means that this location is being marked as read-only.

Moreover, the following shorthands are used for describing some memory states. We denote:

- $s_\emptyset$, the empty memory state with no declared identifiers, i.e., $(\bot, \bot)$ where $\bot$ is the well-known nowhere defined function,
- $s \oplus id$, the memory state obtained from state s by declaring a new read-only variable id of type `int`, i.e., $(env_s[id \mapsto l], sto_s[l \mapsto (\textit{uninitialized value}, ro)])$, where l is the smallest location that is not in domain of sto_s. [4]
- $s \ominus id$, the memory state obtained from state s by freeing the variable named id, i.e., (env'_s, sto'_s), where env'_s (resp. sto'_s) is the partial function whose domain is $dom(env_s) \backslash \{id\}$ (resp. $dom(sto_s) \backslash \{env_s(id)\}$) and is equal to env_s (resp. sto_s) on this domain,
- $(s \mid id \mapsto v)$, the same memory state as s except that id is associated with value v (irrespective of the access permission), i.e., $(env_s, sto_s[env_s(id) \mapsto (v, ro)])$,

[3] We restrict the set of locations to be totally ordered because we want to uniquely determine what the used locations will be after a memory allocation.

[4] $f[x \mapsto y]$ stands for the partial function whose domain is $dom(f) \cup \{x\}$, and that maps x to y, and any $z \in dom(f) \setminus \{x\}$ to $f(z)$.

– $(s \ |_D \ s')$, where $D \subseteq ID$, the same memory state as s, except that it equals to state s' for every identifier in D (for any such identifier, its corresponding location, its value and its access permission in the resulting state are the same as in s'), i.e., $(env_s[D\ env_{s'}], sto_s[env_{s'}(D)\ sto_{s'}])$.[5]

Semantic function Our semantic definition of the xfor statement is built from semantic functions that we assume to be known for all the other constructs of the C programming language. We classically denote by $[\![S]\!] : STATE \rightharpoonup STATE$, the denotational semantics of any statement S, and $[\![e]\!] : STATE \rightharpoonup VAL \times STATE$, the denotational semantics of any C expression e. Note that this latter function returns a memory state, since an expression in C may change the memory state as a side effect. Moreover, since statements or expressions may use previously declared functions, all these semantic functions are parametrized by a global environment. However, since this global environment is not altered during the run of a xfor, it remains implicit.

Domain of the semantic function Here, we informally describe what are the necessary conditions for the partial function $[\![M]\!]$, the semantics of the xfor construct M, to result in a defined memory state. These conditions refer to the syntactic elements defined on page 4. The resulting memory state is defined only if all these conditions hold:

– expressions a_i^l, b_i^l, c_i^l, g_i^l and o_i^l ($i = 1..k$, $l = 1..n$) do not have any side effect, i.e., each of these expressions, say e, is such that $\forall s \in dom([\![e]\!]), \exists v \in VAL, [\![e]\!](s) = (v, s)$. In this case, we will denote by $[\![e]\!](s)$ the value of e (i.e., v) in memory state s,
– expressions a_i^l, b_i^l, c_i^l, g_i^l and o_i^l ($i = 1..k$, $l = 1..n$) do not depend on any index variable of inner loops, *i.e.,* any of these expressions does not depend on a $v_i^{l'}$ with $l' \geq l$ (but may depend on the other index variables of the i-th loop nest),
– expressions c_i^l and g_i^l do not evaluate to zero whatever are the values of the index variables,
– statement S_i and expressions a_i^l, b_i^l, c_i^l, g_i^l and o_i^l ($l = 1..n$), do not contain any occurrence of an index variable v_j^l where $j \neq i$ ($j = 1..k$, $l = 1..n$),
– no index variables are modified within statements S_i ($i = 1..k$).

3.2 Semantics definition

Let s_0 be the so called *initial memory state* given as argument to function $[\![M]\!]$. Let V be the set of all index variables, i.e., $\{v_i^l, i = 1..k, l = 1..n\}$. We also need to define the memory state, say s'_0, obtained from s_0 by declaring all the index variables. It can be defined by $s'_0 \stackrel{\text{def}}{=} ((s_0 \ |_V \ s_\emptyset) \oplus V)$ where $(s \oplus V)$ is a shortcut for $(\ldots((((\ldots((s \oplus v_1^1) \oplus v_2^1)\ldots) \oplus v_k^1) \oplus v_1^2) \oplus v_2^2)\ldots) \oplus v_k^n$.

Iteration domains We first define the iteration domain D_i of each original loop nest ($i = 1..k$). This domain is a subset of $\mathbb{N}^n$, where n is the loop nest depth. It can be defined incrementally, starting from the outermost level, i.e, $l = 1$, to the innermost one, i.e., $l = n$, of the loop nest, and adding a dimension at each level. We denote by D_i^l ($l = 1..n$) the subset of $\mathbb{N}^l$ related to level l. The domain D_i is thus defined as $D_i \stackrel{\text{def}}{=} D_i^n$.

Each element z of the domain D_i^l ($l = 1..n$) is obtained by evaluating expressions a_i^l, b_i^l and c_i^l in a memory state where the outer index variables $v_i^{l'}$ (with $l' < l$) are declared and associated with the values corresponding to z. We denote this memory state by $s_i^{l-1}[z]$. It comes that, in this memory state, the other index variables, i.e., the inner loops index variables (with $l' \geq l$) and the index

[5] $f[D\ g]$ stands for the partial function whose domain is $(dom(f) \setminus D) \cup (D \cap dom(g))$, and that maps x to $f(x)$ if $x \notin D$, and to $g(x)$ if $x \in D$.

variables of the other loop nests, are thus not associated with any location. Therefore, expressions a_i^l, b_i^l and c_i^l cannot use any of these other index variables. Memory states $s_i^l[z]$ ($z \in D_i^l$, $l = 1..n$) are defined by recurrence as follows, from memory state $s_i^0[()] \stackrel{\text{def}}{=} (s'_0 \ |_V \ s_\emptyset)$ where some locations are used to store the values of the index variables, however these variables are not associated with their location:

$$s_i^l[z] \stackrel{\text{def}}{=} ((s' \ |_{\{v_i^l\}} \ s'_0) \mid v_i^l \mapsto [\![a_i^l]\!](s') + z_l \times [\![c_i^l]\!](s')) \text{ with } s' = s_i^{(l-1)}[(z_1, ..., z_{l-1})]$$

where $()$ is the unique element of $\mathbb{N}^0$ and z_ℓ ($\ell = 1..l$) stands for the ℓ-th component of the l-dimensional vector z of $\mathbb{N}^l$. In the previous definition, memory state $s_i^l[(z_1, ..., z_l)]$ is obtained from $s_i^{(l-1)}[(z_1, ..., z_{l-1})]$ by associating one more index variable, i.e., v_i^l, with its location in memory state s'_0, and by assigning the value corresponding to element $(z_1, ..., z_l)$ in the iteration domain to this variable. This value is expressed by using z_l and the values of expressions a_i^l and c_i^l. Domains D_i^l ($i = 1..k$, $l = 1..n$) can then be defined by:

$$D_i^l \stackrel{\text{def}}{=} \{z \in \mathbb{N}^l \mid (z_1, ..., z_{l-1}) \in D_i^{l-1} \wedge 0 \leq z_l < \lceil \tfrac{[\![b_i^l]\!](s') - [\![a_i^l]\!](s')}{[\![c_i^l]\!](s')} \rceil\} \text{ with } s' = s_i^{(l-1)}[(z_1, ..., z_{l-1})]$$

where $D_i^0 \stackrel{\text{def}}{=} \{()\}$ and $\lceil x \rceil$ is the integer ceiling of x. As said in section 3.1, $[\![M]\!](s_0)$ is not defined if there exists a previously defined memory state s' such that $[\![c_i^l]\!](s') = 0$.

Instructions To each element, say z, of the iteration domain D_i ($i = 1..k$), we attach an *instruction* denoted by $S_i[z]$. Note that we are not interested here in defining the syntax of this instruction, but its semantics only. By definition, this instruction is equivalent to the statement S_i in the case where the index variables are declared and associated with the values corresponding to z, i.e., in a memory state where the index variables have the same locations and values as in $s_i^n[z]$. Therefore, assuming that, in a memory state s, no identifier is associated with the same location as an index variable in s'_0, then

$$[\![S_i[z]]\!](s) \stackrel{\text{def}}{=} ([\![S_i]\!](s \ |_V \ s_i^n[z]) \ |_V \ s)$$

and the assumption about the memory state s formalizes as $dom([\![S_i[z]]\!]) \stackrel{\text{def}}{=} \{s \in dom([\![S_i]\!]) \mid env_s(ID) \cap env_{s'_0}(V) = \emptyset\}$.

Note that this definition means that the statement S_i executes in an environment where only the index variables of the i-th loop nest can be accessed to get their value. This statement cannot access the other index variables. Moreover, since the locations of the index variables are marked as read-only, the instruction semantics is undefined if S_i contains an assignment of an index variable.

Schedule The schedule of instructions is defined by using grains and offsets, namely g_i^l and o_i^l ($i = 1..k$, $l = 1..n$). These expressions define an injection from iteration domain D_i to $\mathbb{Z}^n$. This injection, called map_i, can be formalized as follows, for any $z \in D_i$:

$$map_i(z) \stackrel{\text{def}}{=} (z_l \times [\![g_i^l]\!](s_i^{l-1}[z]) + [\![o_i^l]\!](s_i^{l-1}[z]))_{l=1..n}$$

where z_l is the l-th component of the n-dimensional vector z of $\mathbb{N}^n$. By definition, the *referential domain* is the domain $D \stackrel{\text{def}}{=} \cup_{i=1}^k map_i(D_i)$. The lexicographic order on $\mathbb{Z}^n$, denoted by $\ll$, induces the schedule of instructions. Instruction $S_i[z]$ is executed before $S_j[z']$ if $map_i(z) \ll map_j(z') \vee (map_i(z) = map_j(z') \wedge i < j)$.

The semantics of the xfor construct is the composition of all the instructions' semantics in the previously mentioned order. Formally, let us denote by $\Delta[z]$ ($z \in D$) the partial function obtained

by composition of all the partial functions $[\![S_i[z']\!]$, such that $i \in \{1..k\}$, $z' \in D_i$ and $map_i(z') = z$, in the decreasing order of i. Then,

$$[\![M]\!](s_0) \stackrel{\text{def}}{=} (((\Delta[z^m] \circ \Delta[z^{m-1}] \circ ... \circ \Delta[z^1])(s'_0) \ominus V) \ |_V \ s_0)$$

where $z^1, ..., z^m$ is the sequence of all elements of D sorted by the lexicographic order on $\mathbb{Z}^n$ and $(s \ominus V)$ is a shortcut for $(\dots((((\dots((s \ominus v_k^n) \ominus v_{k-1}^n)\dots) \ominus v_1^n) \ominus v_k^{n-1}) \ominus v_{k-1}^{n-1})\dots) \ominus v_1^1$.

Note that, after the xfor statement has been executed, the index variables are deallocated and the environment of the initial memory state is restored so that an identifier having the same name as an index variable retrieve the value it had just before the xfor execution.

4 Equivalence proof

In this section, we use the denotational semantics to show that the Red-Black xfor codes in Fig. 1 (right) and Fig. 5, i.e., `xfor1`, `xfor2` and `xfor3`, are equivalent. For sake of conciseness, we only write the main steps of the proof. In order to distinguish the different instructions and domains, we put the xfor code number in superscript: for example, the referential domain for `xfor1` will be denoted by D^1. We also denote by S the statement `u[i][j]=f(u[i][j+1],u[i][j-1],u[i-1][j],u[i+1][j]);` and $S[i,j]$ the statement S in which variables `i` and `j` have been substituted by literals, i.e., $\langle i \rangle$ and $\langle j \rangle$ respectively. For example, $S[1,1]$ is the statement `u[1][1]=f(u[1][2],u[1][0],u[0][1],u[2][1]);`.

Proof (Equivalence between `xfor1` and `xfor2` codes). The instructions and the referential domains are not the same in `xfor1` and `xfor2`, but we show that the referential domain of the `xfor2` code, i.e., D^2, is included in D^1, and that at every point z in D^2, $\Delta^1[z] = \Delta^2[z]$, and for any other point in D^1, $\Delta^1[z]$ is an identity, i.e., a neutral element with respect to function composition, which is a sufficient condition for the two codes to be equivalent.

For `xfor1`, we have $n = 2$, $k = 2$ and thus there are two iteration domains. By definition:

$$D_1^1 = \{(i,j) \in \mathbb{N}^2 \mid () \in \{()\} \wedge 0 \leq i < \lceil \tfrac{[\![\texttt{N-1}]\!](s') - [\![\texttt{1}]\!](s')}{[\![\texttt{1}]\!](s')} \rceil \wedge 0 \leq j < \lceil \tfrac{[\![\texttt{N-1}]\!](s'') - [\![\texttt{1}]\!](s'')}{[\![\texttt{1}]\!](s'')} \rceil\}$$

with $s' = s_1^0[()]$ and $s'' = s_1^1[(i)]$. It comes that $D_1^1 = D_2^1 = \{(i,j) \in \mathbb{N}^2 \mid 0 \leq i,j < N-2\}$ where $N = [\![\texttt{N}]\!](s_0)$. The injections into the referential domain are $map_1^1(i,j) = (i,j)$ and $map_2^1(i,j) = (i+1,j)$. The referential domain D^1 is thus equal to $map_1^1(D_1^1) \cup map_2^1(D_2^1) = \{(k,l) \in \mathbb{Z}^2 \mid 0 \leq k < N-1 \wedge 0 \leq l < N-2\}$.

For `xfor2`, we have $n = 2$ and $k = 4$, and there are four iteration domains, i.e., D_1^2, D_2^2, D_3^2, D_4^2. The injections into the referential domain are $map_1^2(i,j) = (2i, 2j)$, $map_2^2(i,j) = (2i+1, 2j+1)$, $map_3^2(i,j) = (2i+1, 2j)$ and $map_4^2(i,j) = (2i+2, 2j+1)$. The referential domain D^2 is thus equal to $\{(k,l) \in \mathbb{Z}^2 \mid 0 \leq k < N-1 \wedge 0 \leq l < N-2 \wedge (k+l) \mod 2 = 0\}$.

Hence, we have $D^2 \subseteq D^1$. Moreover, assuming s is a memory state such that no identifier is associated with the same location as an index variable in s'_0, i.e., $env_s(ID) \cap env_{s'_0}(V) = \emptyset$, then for any (i,j) in the iteration domain, i.e, D_1^1 or D_2^1, $[\![S_1^1[(i,j)]]\!](s)$ is equal to $[\![S[i+1, j+1]]\!](s)$ if $(i+j) \mod 2 = 0$, and $[\![\{\}]\!](s)$ $(= s)$ else, and similarly $[\![S_2^1[(i,j)]]\!](s)$ is equal to $[\![S[i+1, j+1]]\!](s)$ if $(i+j) \mod 2 = 1$, and $[\![\{\}]\!](s)$ else. Hence, for any $(k,l) \in D^1$,

$$\Delta^1[(k,l)](s) = \begin{cases} [\![S[k+1, l+1]]\!](s), \text{if } (k+l) \mod 2 = 0 \wedge k = 0 \\ ([\![S[k+1, l+1]]\!] \circ [\![S[k, l+1]]\!])(s), \text{if } (k+l) \mod 2 = 0 \wedge 0 < k < N-2 \\ [\![S[k, l+1]]\!](s), \text{if } (k+l) \mod 2 = 0 \wedge k = N-2 \\ [\![\{\}]\!](s), \text{otherwise} \end{cases}$$

Similarly, we have, for any (i,j) in the iteration domain, i.e, D_1^2 or D_2^2 or D_3^2 or D_4^2, $[\![S_1^2[(i,j)]]\!](s) = [\![S[2i+1, 2j+1]]\!](s)$, $[\![S_2^2[(i,j)]]\!](s) = [\![S[2i+2, 2j+2]]\!](s)$, $[\![S_3^2[(i,j)]]\!](s) = [\![S[2i+1, 2j+1]]\!](s)$, and $[\![S_4^2[(i,j)]]\!](s) = [\![S[2i+2, 2j+2]]\!](s)$. Hence, for any $(k,l) \in D^2$, $\Delta^1[(k,l)](s) = \Delta^2[(k,l)](s)$. □

Proof (Equivalence between ***xfor2*** *and* ***xfor3*** *codes).* In this case, instructions are the same but their schedules are different. Thus the proof consists in showing that the composition of instructions' semantics in each of the two orders gives the same result. We prove a sufficient condition that falls into these three parts: **(1) The instructions are the same in the two codes**: only the position of the instructions in the sequence is different. This is obvious because the two codes only differ from their respective offsets. Thus, the instruction set and the iteration domains are the same, i.e., $D_i^3 = D_i^2$ and $S_i^3[z] = S_i^2[z]$ ($i \in \{1..4\}$). Moreover, grains and offsets define injections, i.e., map_i^2 and map_i^3 ($i \in \{1..4\}$) are injections onto the referential domain. Therefore, no instruction can be deleted or duplicated. **(2) In the instructions' sequence of `xfor3`, red (resp. black) instructions are set in the same order as with `xfor2`**. This is also trivial because modifying offsets does not change this order. The reason comes from the following property of the lexicographic order on $\mathbb{Z}^n$: $\forall o \in \mathbb{Z}^n, \forall z, z' \in \mathbb{Z}^n, z \ll z' \Rightarrow (z + o) \ll (z' + o)$. **(3) The dependencies between red and black instructions are preserved**, i.e., given two instructions, a red one and a black one that depends on the red one, if this black instruction is located after this red one in the sequence of `xfor2`, then it is the same for `xfor3`. The proof of this assertion can be done by considering every case, since a red instruction is either $S_1[(i, j)]$ or $S_2[(i, j)]$, and a black instruction is either $S_3[(i', j')]$ or $S_4[(i', j')]$. For example, let us consider the case where the red instruction is $S_1[(i, j)]$ and the black one is $S_3[(i', j')]$ and assume that $S_3[(i', j')]$ is after $S_1[(i, j)]$ in `xfor2`, i.e., $map_1^2(i, j) = (2i, 2j) \ll map_3^2(i', j') = (2i' + 1, 2j')$. By definition of the lexicographic order on $\mathbb{Z}^n$, if $k \geq 0$ where $k = [\![\mathtt{k}]\!](s_0)$, then we have $(2i' + 1, 2j') \ll (2i' + k + 1, 2j')$. Since $\ll$ is an order relation, it is transitive and therefore $(2i, 2j) \ll (2i' + k + 1, 2j')$, which means that $S_3[(i', j')]$ is after $S_1[(i, j)]$ in `xfor3`. Note that, in this case, the proof does not require to use the assumption saying that $S_3[(i', j')]$ depends on $S_1[(i, j)]$. If required, this assumption could be formalized in the previously presented semantic framework as: $\mid (2i + 1) - (2i' + 1) \mid + \mid (2j + 1) - (2j' + 1) \mid \leq 1$. This formula can be deduced from Bernstein's conditions [10] which state when two instructions depend on each other. From the three previous properties, the sequence of instructions is the same for `xfor2` and `xfor3`, except that in `xfor3` a black instruction can be set after a red one when they do not depend on each other. In this case, it is possible to swap these instructions in the sequence without changing the semantics, i.e., the partial functions commute under composition. Starting from the sequence of `xfor3` and performing this exchange whenever possible terminates and gives exactly the same sequence as for `xfor2`. □

5 Xfor programming

In this section, we describe programming strategies for inter and intra data reuse distance minimization, i.e., reuse between statements belonging originally to separated loop bodies, and reuse between statements belonging originally to a single loop body, respectively. We consider reuse distance as being the number of iterations between two successive accesses to a same memory location.

Minimizing inter-data-reuse distance: This strategy is the one used for the Red-Black problem. As previously said, the first step is to identify data dependences, i.e., relations between points of the different iteration domains of types RAW (Read-After-Write), WAR, WAW, but also RAR, since it also implies data reuse. All data-dependent domains have then to be scheduled by overlapping them using shifting (offset) and dilatation (grain), with the goal of minimizing data reuse distances while respecting data dependences, according to the lexicographic order on the referential domain. The final schedule can then be described by a xfor-loop nest.

Minimizing intra-data-reuse distance: In this strategy, an iteration domain is split into several domains, each being associated with a subset of the original loop body, or a partial computation of an original arithmetic expression, if such a decomposition is allowed regarding mathematical properties and arithmetic precision. Thus, statements can be re-scheduled by overlapping these domains as described in the previous strategy.

Experiments Experiments have been conducted on an Intel Xeon X5650 6-core processor 2.67GHz running Linux 3.2.0-35. Our set of benchmarks has been built from the the Polyhedral Benchmark suite [7], from which representative codes exhibiting data reuse and allowing simple transformations have been selected. These are most of the codes included in the benchmark suite. Every code has been rewritten using the xfor structure and the programming strategies previously described, and is available following the link indicated in [11]. Original and xfor versions have been compiled using GCC 4.6.3 with options O3 and march=native, and their outputs have been compared to ensure correctness of the xfor codes. In the graphs of Figure 6, speed-ups for the main loop kernels and obtained thanks to the xfor codes are given (original time/xfor time). The grains are always set to 1, excepting for the Red-Black code whose grain is 2.

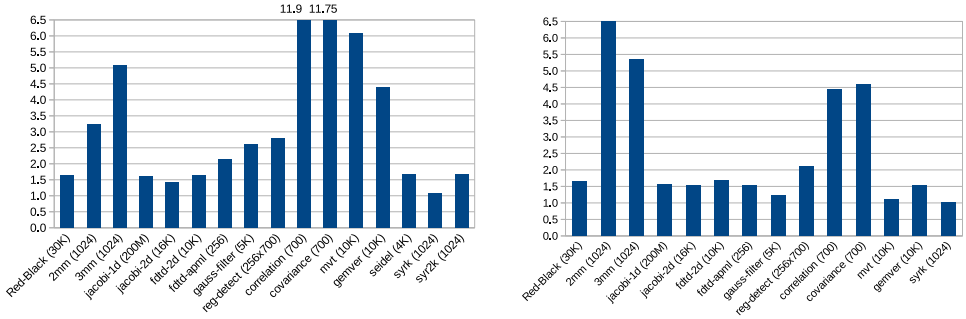

Fig. 6. Speed-ups of sequential (left) and OpenMP-parallel (right) xfor codes compared with the original sequential and parallel versions

For the first set of measurements regarding sequential executions (Figure 6, left), automatic vectorization has been applied when allowed by GCC, and by setting the offsets of the innermost xfor-loops conveniently when necessary. Some benchmark codes could not take advantage of vectorization due to access patterns detected as too complicated by GCC. Notice that both original and xfor codes are always simultaneously affected since they have similar access patterns. OpenMP parallelization has been turned on in GCC using option fopenmp. According to the dependences, the outermost possible xfor-loops and original for-loops have been parallelized. Some xfor codes have been slightly modified in order to exhibit slices of parallel outermost xfor-loops, by increasing offset values and tiling the outermost xfor-loops. Codes have been run using 12 parallel threads mapped on the 6 hyperthreaded processor cores of the Xeon X5650 processor. Speed-ups represented in Figure 6 (right) are given by comparison between the parallel original and the parallel xfor code versions. Every benchmark code has been easily parallelized, excepting seidel which both xfor and original code versions could not be parallelized due to dependences carried by every loop. A more advanced loop transformation is required by this code (skewing). One can observe that xfor codes are always faster than original codes, mostly providing significant, and sometimes dramatic, execution times speed-ups.

6 Related work

To our knowledge, there is no similar proposal in the literature. New looping features have been proposed in PGAS (Partitioned Global Address Space) languages, such as zippered iterators in Chapel [12] or sequential iteration over points in region in canonical lexicographic order in X10 [13]. Regions in X10 can be defined from composing arrays which are 2-dimensional at maximum, and whose composed shapes are limited to rectangles and triangles. Some xfor structures may be translated into PGAS languages' loop structures. However, it would require the programmer to define domains compositions and handle related code modifications, as indices substitutions and scheduling of the statements. Moreover, these languages compilers do not take advantage of polyhedral modeling and optimizations, and weak performance of the code generated by their compilers has been reported, compared to standard parallel languages as OpenMP or MPI.

7 Conclusion

We have presented a new programming control structure, xfor, whose purpose is to provide programmers a concise and direct way to control the locality and reuse distances of accessed data, while preventing them of writing complicated and convoluted code thanks to the automatic translator and code generator IBB. Thanks to the given denotational semantics, reliable transformations of xfor statements can be defined. The xfor construct provides one programmer with an abstract geometrical viewpoint. Experiments carried out with the Polyhedral Benchmark suite show the benefits from using it in order to obtain significant speed-ups in practice. The xfor construct appears to be a convenient tool allowing to pass down know-how between programmers and compilers.

References

1. Leiserson, C.E.: The Cilk++ Concurrency Platform. In: Proceedings of the 46th Annual Design Automation Conference. DAC '09, New York, NY, USA, ACM (2009) 522–527
2. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: Better Strategies for Parallel Haskell. In: Proc. of the 3rd ACM SIGPLAN Symp. on Haskell, Baltimore, MD, USA, ACM Press (September 2010) 91–102
3. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Series. Artima Press (2011)
4. Sagonas, K.F.: Using Static Analysis to Detect Type Errors and Concurrency Defects in Erlang Programs. In: FLOPS. (2010) 13–18
5. Christadler, I., Erbacci, G., Simpson, A.D.: Performance and productivity of new programming languages. In Keller, R., Kramer, D., Weiss, J.P., eds.: Facing the Multicore-Challenge II. Springer-Verlag (2012) 24–35
6. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques. PACT '04, IEEE Computer Society (2004) 7–16
7. PolyBench: The Polyhedral Benchmark suite. `cse.ohio-state.edu/~pouchet/software/polybench`
8. Scott, D.S.: Outline of a Mathematical Theory of Computation. Technical Report PRG–2, Programming Research Group, Oxford, England (November 1970)
9. Tennent, R.D.: The denotational semantics of programming languages. Com of ACM **19**(8) (1976)
10. Bernstein, A.J.: Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers **EC-15** (1966) 757–763
11. IBB: The IBB Compiler. `www.team.inria.fr/camus/ibb`
12. Chamberlain, B.L., Choi, S.E., Deitz, S.J., Navarro, A.: User-Defined Parallel Zippered Iterators in Chapel. In: PGAS 2011: Fifth Conf. on Partitioned Global Address Space Programming Models. (2011)
13. Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O., Grove, D.: X10 Language Specification V2.2 (2012)