Software-controlled Processor Stalls for Time and Energy Efficient Data Locality Optimization

Philippe Clauss
INRIA CAMUS, ICube Lab, CNRS
University of Strasbourg, France
Email: philippe.clauss@inria.fr

Imen Fassi
INRIA CAMUS, ICube Lab, CNRS
University of Strasbourg, France
University El Manar, Tunisia
Email: imen.fassi@inria.fr

Alexandra Jimborean
Department of Information Technology
Uppsala University, Sweden
Email: alexandra.jimborean@it.uu.se

***Abstract*—Data locality optimization is a well-known goal when handling programs that must run as fast as possible or use a minimum amount of energy. However, usual techniques never address the significant impact of numerous stalled processor cycles that may occur when consecutive load and store instructions are accessing the same memory location. We show that two versions of the same program may exhibit similar memory performance, while performing very differently regarding their execution times because of the stalled processor cycles generated by many pipeline hazards. We propose a new programming structure called "xfor", enabling the explicit control of the way data locality is optimized in a program and thus, to control the amount of stalled processor cycles. We show the benefits of xfor regarding execution time and energy saving.**

I. Introduction

Designing optimizations for reducing the execution time or the energy consumption of compute-intensive codes relies predominantly on an analysis of the performed memory accesses. Indeed, it is well-known that memory accesses play a crucial role in the execution time of a program due to memory latencies that make the processor stall while waiting for requested data. Standard optimization objectives are to reduce the number of cache misses to reduce these latencies. Regarding energy consumption, the optimization goals are very similar since requests to memory units are responsible for a significant part of the total energy expenditure of a program, and stalls due to memory latencies cause the processor to waste energy. When programming, a common approach to reduce cache misses and memory latencies is to improve the temporal and spatial locality of the data accessed by the program.

However, even if a program exhibits "good" data locality with few generated cache misses, it may happen that it still provides weak time performance. A main reason can be numerous pipeline stalls due to WAW (Write-After-Write), RAW (Read-After-Write) or WAR (Write-After-Read) hazards, or to functional unit contentions. Thus, two different versions of a program may generate similar numbers of cache misses and memory accesses, while still performing very differently regarding execution time.

On the other hand, codes exhibiting good memory performance while generating a huge amount of pipeline stalls have a particular interest regarding energy saving. Indeed, since they are not generating more memory accesses than faster code versions, their energy consumption rate, *i.e.*, the consumed power, is significantly lower. Thus, it is possible to reduce the power consumed by a running code at a given threshold, at the price of an execution-time increase.

In this paper, we exhibit that handling the amount of pipeline stalls by software is as important as handling data locality. We propose a new programming structure, *xfor*, which enables explicit control of the data reuse distances between statements of a loop nest, thanks to two parameters: offset and grain. We show that the adjustment of these parameters results in code versions exhibiting good data locality and similar memory performance, but generating varying numbers of pipeline stalls. This new programming structure is supported by a source-to-source compiler named IBB for Iterate-But-Better, which automatically translates any C code containing xfor-loops into an equivalent code where xfor-loops have been translated into for-loops.

We conducted our experiments on an Intel Ivybridge processor and measured, in addition to the number of cache misses and stalled cycles, the energy dissipated by the target loop nests by accessing the dedicated processor counter MSR_PKG_ENERGY_STATUS.

The paper is organized as follows. In the next Section, we first introduce the xfor programming structure with a code example and then present its syntax and semantics. In Section III, we explain how to control the data reuse distances with xfor structures and highlight the general problem of stalled processor cycles significantly impacting the execution time. Section IV describes several strategies for saving energy thanks to the use of xfor programs. Section V proposes an iterative search to find the best combination of xfor offset values reaching improved data locality and controlled number of stalled processor cycles. Related work is addressed in Section VII. In Section VI, we show the benefits of xfor programming regarding execution time and energy consumption on a set of benchmark programs extracted from the polybench benchmark suite [12]. Conclusions are given in Section VIII.

II. The XFOR programming structure

A. Illustrative example

Consider the loop nest in Figure 1 which was extracted from the seidel program of the polybench benchmark suite [12]. This code is a classic stencil computation where the eight neighbors of each grid point are accessed to update the point with their average. Thus, each array element is reused eight

```
for (t = 0 ; t <= tsteps - 1 ; t++)
 for (i =1 ; i<= n-2 ; i++)
  for (j=1 ; j<=n-2 ; j++)
   A[i][j] = (A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]
             +A[i][j-1]+A[i][j]+A[i][j+1]
             +A[i+1][j-1]+A[i+1][j]+A[i+1][j+1])/9.0;
```

Fig. 1. Main for-loop nest of the seidel code

times to compute eight different averages. One important issue is that following the canonical lexicographic order of the 2-level loop nest, each point is updated using four neighbors that are already updated, while the four others are still assigned their initial values.

To minimize the data reuse distances, we split the statement into five elementary statements: four statements consisting of adding neighbors that were not yet updated, and one last statement consisting of adding all the remaining updated neighbors and of computing the average. The resulting loop body is shown in Fig. 2 as the body of an xfor loop structure nest. In this version of the loop body, accesses to a given element of array A made by statements 0 to 3 are all made once and for all at each iteration, while successively taking part of the computation of four stencil computations in which it appears. The code required to implement such a schedule by using standard for-loops would be tedious and complex to program.

The new xfor loop structure proposed in this paper enables programmers to schedule statements separately by defining their own loop iterators and two additional parameters, the offset and the grain, devoted to synchronize the statements among each other. Let us focus on the offset parameters of the example, appearing at the very end of the xfor headers. For instance, the first 0 in the outer loop list and the first 2 of the inner loop list – noted (0,2) in the following – tell that the first statement is shifted by 0 in the outer loop direction and by 2 in the inner loop direction, thus resulting in an execution behavior similar to the one of the corresponding statement in comments. Note that array elements that are actually accessed inside an xfor-loop iteration are given by subtracting the offset values to the indices inside the array reference functions, so resulting in the code in comments.

Dependent statements must be conveniently scheduled thanks to their respective offset values. The final computation of the average (statement 4), using array elements that were already updated, has to be performed after the last element update (statement 3). Thus statement 4 has been assigned the greatest couple of offset values (1,2), which is the same as for statement 3. Since statement 3 is appearing before statement 4 in the loop body, they are executed in this order and dependences are respected.

Both the other lists of integer numbers are defining the grain parameter values. Their role is described in the next subsection.

```
for (t = 0 ; t <= tsteps-1 ; t++)
xfor (i0=1,i1=1,i2=1,i3=1,i4=1 ;
      i0<=n-2,i1<=n-2,i2<=n-2,i3<=n-2,i4<=n-2 ;
      i0++,i1++,i2++,i3++,i4++ ;
      1,1,1,1,1 ; /* grains */
      0,1,1,1,1 ) /* offsets */ {
xfor (j0=1,j1=1,j2=1,j3=1,j4=1 ;
      j0<=n-2,j1<=n-2,j2<=n-2,j3<=n-2,j4<=n-2 ;
      j0++,j1++,j2++,j3++,j4++ ;
      1,1,1,1,1 ; /* grains */
      2,0,1,2,2 ) /* offsets */ {

      0: A[i0][j0] += A[i0][j0+1] ;
      1: A[i1][j1] += A[i1+1][j1-1] ;
      2: A[i2][j2] += A[i2+1][j2] ;
      3: A[i3][j3] += A[i3+1][j3+1] ;
      4: A[i4][j4] = (A[i4][j4]+A[i4-1][j4-1]
                     +A[i4-1][j4]+A[i4-1][j4+1]
                     +A[i4][j4-1])/9.0 ;

/*    0: A[i][j-2]     += A[i][j-1] ;
      1: A[i-1][j]     += A[i][j-1] ;
      2: A[i-1][j-1]   += A[i][j-1] ;
      3: A[i-1][j-2]   += A[i][j-1] ;
      4: A[i-1][j-2]   = (A[i-1][j-2]+A[i-2][j-3]
                         +A[i-2][j-2]+A[i-2][j-1]
                         +A[i-1][j-3])/9.0 ; */ }}
```

Fig. 2. Main xfor-loop nest of the seidel code

B. XFOR syntax and semantics

The xfor syntax is defined by:

```
xfor ( index = expr, [index = expr, ...] ;
       index relop expr, [index relop expr, ...] ;
       index += incr, [index += incr, ...] ;
       grain, [grain, ...] ;
       offset, [offset, ...] )      {
       label: {statements}
      [label: {statements}, ...]   }
```

The first three elements in the xfor header are similar to the initialization, test, and increment parts of a traditional C for-loop, except that all these elements describe two or more loop indices. The last two components provide the grain and offset for each index: these values are constants, and the grain must be positive. All domains must be affine: “expr” denotes affine combinations of enclosing loop indices, “relop” is one of ==, !=, <, <=, > or >=, and “incr” must be an integer. Every index in the set must be present in all components of the header, and (sequences of) statements are labelled with the rank of the corresponding index (0 for the first index, 1 for the second, and so on).

The list of indices defines several for-loops whose respective iteration domains are all mapped onto a same global “virtual referential” domain. The way iteration domains of the for-loops are overlapped is defined solely by their respective offsets and grains, and not by the values of their respective indices, which have their own ranges of values. The grain defines the frequency in which the associated loop has to run, relatively to the referential domain. For instance, if the grain equals 2, then one iteration of the associated loop will run for every second iteration of the referential. The offset defines the gap between the first iteration of the referential and the first iteration of the associated loop. For instance, if the offset equals 3, then the first iteration of the associated loop will run at the fourth iteration of the referential loop.

The size and shape of the referential domain can be deduced from the for-loop domains composing the xfor-loop. Geometrically, the referential domain is defined as the union of the for-loop domains, where each domain has been shifted according to its offset and dilated according to its grain.

The relative positions of the iterations of the individual for-loops composing the xfor-loop depend on how individual domains overlap. Iterations are executed in the lexicographic order of the referential domain. On portions of the referential domain where at least two domains overlap, the corresponding statements are run in the order implied by their label (which is also the order with which indices are listed in the xfor header).

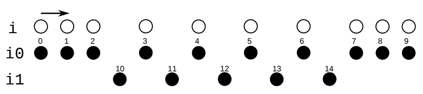
On a sub-domain where one or more loops actually execute their statements, it can happen that some iterations have no statement to execute, when the individual loops involved all have grains larger than 1. In such cases, that particular sub-domain is compressed, by a factor equal to the greatest common divisor of all grains.

The bodies of the for-loops composing the xfor-loop can be any C-based code. However, their statements can only access their respective loop indices, and not any other loop index whose value may be incoherent in their scope. Moreover, indices can only be modified in the loop header by incrementation, and never in the loop body. Let us illustrate this definition with a few examples.

Example 1. Consider the following xfor-loop:

```
xfor (i0=0, i1=10 ; i0<10, i1<15 ;
      i0++, i1++   ; 1, 1 ; 0, 2) {
      0: loop_body0
      1: loop_body1                  }
```

In this example, the offset of index i0 is zero, and the one of index i1 is 2. Thus, the first iteration of the i0-loop will run immediately, while the first iteration of the i1-loop will run at the third iteration of the xfor, but with index value i1=10. On the sub-domain where both for-loop domains overlap, the loop bodies are run in interleaved fashion starting with loop_body0. This behavior is illustrated by the figure below:

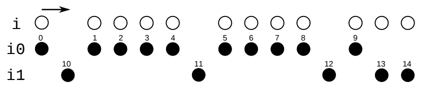


Notice that the index values have no effect on the relative positions of the iteration domains, which are uniquely determined by their respective grains and offsets.

Example 2. Another example is:

```
xfor (i0=0, i1=10 ; i0<10, i1<15 ;
      i0++, i1++   ; 1, 4 ; 0, 0) {
      0: loop_body0
      1: loop_body1                  }
```

Now, the i0-grain is 1 and the i1-grain is 4. In this case, for one iteration of the i1-loop, four iterations of the i0-loop will be run on the sub-domain on which they overlap. The last two iterations on i1 occur after the end of the i0-loop: their domain can be compressed by a factor of 4, as it is illustrated below:

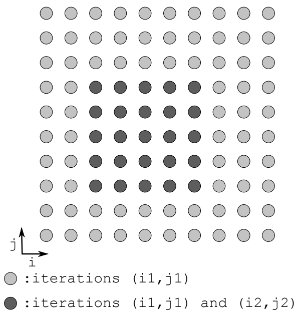


Nested xfor-loops are behaving like several nested for-loops which are synchronized according to the common referential domain. Nested for-loops are defined according to the order in which their respective indices appear in the xfor headers. For instance, in a 2-level xfor nest, the first index variable of the outermost loop is linked to the first index variable of the inner loop, the second to the second, and so on. Hence the same number of indices have to be defined at each level of any xfor nest. This is not a strong restriction. The syntax enables shorter specifications of indices which are not used inside statements. Let us illustrate this notion with some examples of nested xfor headers.

Example 3. Consider the following xfor-loop nest:

```
xfor (i1=0, i2=0 ; i1<10, i2<5 ;
      i1++, i2++ ; 1, 1 ; 0, 2)
xfor (j1=0, j2=0 ; j1<10, j2<5 ;
      j1++, j2++ ; 1, 1 ; 0, 2)
```

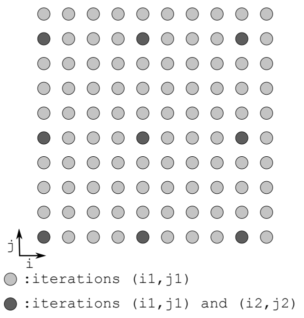
The second for-loop nest has an offset of two at each loop depth. Hence it is delayed in each dimension of the referential domain as it is represented below:



Example 4. Another example is:

```
xfor (i1=0, i2=0 ; i1<10, i2<3 ;
      i1++, i2++ ; 1, 4 ; 0, 0)
xfor (j1=0, j2=0 ; j1<10, j2<3 ;
      j1++, j2++ ; 1, 4 ; 0, 0)
```

In this example, the second for-loop nest has a grain of four at each loop depth. Hence its iterations are spaced by four points in each dimension of the referential domain as shown below:



C. XFOR code generation: the IBB compiler

Source code containing xfor loop-structures is translated by the IBB source-to-source compiler into a semantically equivalent C code made of "regular" for-loop structures. This is done in two steps. First, index domains are turned into polytopes over a common referential domain, and second, scanning code is generated for the union of these polytopes.

Each iteration domain of each for-loop nest composing a xfor nest defines a Z-polytope, *i.e.*, a lattice of integer points delimited by a finite polyhedron. Respective grains and offsets must also be considered in order to define the union of Z-polytopes corresponding to the overlapping of the domains that are defined by the xfor structure. The referential domain is implicitly created by expressing the original indices of the xfor structure into a common basis of referential indices, ref_index, according to the relation:

$$incr \times ref\_index = grain \times original\_index + offset \times incr$$

Increments greater than one yield additional constraints of the form:

$$original\_index = incr \times k,$$

for every integer k. Finally, lower and upper bounds of the referential domain indices are defined by shifting the original bounds to 0.

The second step is performed using the CLooG library [1] devoted to generate code for scanning unions of Z-polytopes. In this model, loop bounds are either constants or affine functions of the enclosing loop indices, and memory instructions are referencing scalars or array elements, also through affine functions of the enclosing loop indices. CLooG generates for-loops that reaches each integral point of one or more parameterized polyhedra. CLooG is designed to avoid control overhead and to produce very effective code. It is used by the GNU GCC compiler for loop optimizations. CLooG accepts input in the OpenScop format [2], which is an open specification defining a file format and a set of data structures to represent loop nests fitting the polyhedral model [3]. This format makes it easy to replace the original indices with their referential domain equivalent.

As an example, the code generated by IBB from the xfor seidel code of Figure 2 is shown in Figure 3. Note the length of this code compared to the xfor version, as well as the number of loops, the duplicated instructions and the various array references. Even this simple example shows the improvement in term of productivity that the xfor structure and the IBB compiler may provide for writing efficient code.

III. Data locality vs. stalled processor cycles

A. Setting data reuse distances with xfor-loops

We consider reuse distance as being the number of iterations between two successive accesses to a same memory location. Statements that share common data can be explicitly brought closer thanks to the grain and offset parameters of the xfor structure.

When handling several loop statements, the first step is to identify data dependences that occur between them, *i.e.*, relations between iteration points of different domains of types Read-After-Write, Write-After-Read, Write-After-Write, but also Read-After-Read, since it also implies data reuse. Convenient offset and grain values regarding two dependent statements can be determined with the support of distance vectors.

```
for (t=0 ; t<=tsteps-1 ;t++) {
 for (j=2 ; j <=n-1; j++)
  A[1][j-1]+=A[1][j];
 for (i=1 ; i<=n-3 ; i++) {
  A[i][1]+=A[i+1][0];
  A[i][1]+=A[i+1][1];
  A[i][1+1]+=A[i+1][1];
  for (j=2 ; j<=n-3 ; j++) {
   A[i][j-1]+=A[i+1][j];
   A[i][j-1]=(A[i][j-1]+A[i-1][j]+A[i-1][j-1]
            +A[i-1][j]+A[i][j])/9.0;
   A[i][j]+=A[i+1][j];
   A[i][j+1]+=A[i+1][j];
   A[i+1][j-1]+=A[i+1][j]; }
  A[i][n-3]+=A[i+1][n-2];
  A[i][n-3]=(A[i][n-3]+A[i-1][n-4]+A[i-1][n-3]
           +A[i-1][n-2]+A[i][n-4])/9.0;
  A[i][n-2]+=A[i+1][n-2];
  A[i+1][n-3]+=A[i+1][n-2];
  A[i][n-2]+=A[i+1][n-1];
  A[i][n-2]=(A[i][n-2]+A[i-1][n-3]+A[i-1][n-2]
           +A[i-1][n-1]+A[i][n-3])/9.0;
  A[i+1][n-2]+=A[i+1][n-1]; }
 A[n-2][1]+=A[n-1][0];
 A[n-2][1]+=A[n-1][1];
 A[n-2][2]+=A[n-1][1];
 for (j=2 ; j<=n-3 ; j++) {
  A[n-2][j-1]+=A[n-1][j];
  A[n-2][j-1]=(A[n-2][j-1]+A[n-3][j]+A[n-3][j-1]
             +A[n-3][j]+A[n-2][j])/9.0;
  A[n-2][j]+=A[n-1][j];
  A[n-2][j+1]+=A[n-1][j]; }
 A[n-2][n-3]+=A[n-1][n-2];
 A[n-2][n-3]=(A[n-2][n-3]+A[n-3][n-4]+A[n-3][n-3]
            +A[n-3][n-2]+A[n-2][n-4])/9.0;
 A[n-2][n-2]+=A[n-1][n-2];
 A[n-2][n-2]+=A[n-1][n-1];
 A[n-2][n-2]=(A[n-2][n-2]+A[n-3][n-3]+A[n-3][n-2]
            +A[n-3][n-1]+A[n-2][n-3])/9.0; }
```

Fig. 3. Code automatically generated by IBB from the xfor seidel code

Let $a[f(I_0)]$ and $a[g(I_1)]$ be two array references appearing in two dependent statements S_0 and S_1 inside an xfor loop nest, where I_k denotes the vector of the loop indices enclosing S_k, f and g denote affine functions. Let O_0 and O_1 be their respective vectors of offset values, G_0 and G_1 their respective vectors of grain values. The distance vector d from S_0 to S_1 is defined by:

$$d = \frac{g(I_1) + O_1}{G_1} - \frac{f(I_0) + O_0}{G_0}$$

The offsets and grains have to be set in order to get a lexicographically positive vector to ensure semantic correctness of the schedule. Null vectors are allowed if the dependent statements are written in a correct dependence-aware order inside the loop body. Data reuse distance minimization is performed by minimizing the components of d, and by minimizing primarily the outermost indices, since it defines the longest reuse distances carried by the outermost loops.

The minimization of data reuse distances can yield opposite effects regarding execution-time performance. Groups of successive read and write instructions that are referencing the same data may generate numerous stalls due to pipeline hazards, and thus significantly degrading execution time performance. This is illustrated by a code example in the next subsection.

```
for (t = 0 ; t <= tsteps-1 ; t++)
xfor (i0=1,i1=1,i2=1,i3=1,i4=1 ;
      i0<=n-2,i1<=n-2,i2<=n-2,i3<=n-2,i4<=n-2 ;
      i0+=2,i1+=2,i2+=2,i3+=2,i4+=2 ;
      1,1,1,1,1 ; /* grains */
      0,0,0,0,0 ) /* offsets */ {
xfor (j0=1,j1=1,j2=1,j3=1,j4=1 ;
      j0<=n-2,j1<=n-2,j2<=n-2,j3<=n-2,j4<=n-2 ;
      j0++,j1++,j2++,j3++,j4++ ;
      1,1,1,1,1 ; /* grains */
      1,0,2,0,1 ) /* offsets */ {

      0: { A[i0][j0] += A[i0][j0+1] ;
           A[i0+1][j0] += A[i0+1][j0+1] ; }
      1: { A[i1][j1] += A[i1+1][j1-1] ;
           A[i1+1][j1] += A[i1+2][j1-1] ; }
      2: { A[i2][j2] += A[i2+1][j2] ;
           A[i2+1][j2] += A[i2+2][j2] ; }
      3: { A[i3][j3] += A[i3+1][j3+1] ;
           A[i3+1][j3] += A[i3+2][j3+1] ; }
      4: { A[i4][j4] = (A[i4][j4]+A[i4-1][j4-1]
                      +A[i4-1][j4]+A[i4-1][j4+1]
                      +A[i4][j4-1])/9.0 ;
           A[i4+1][j4] = (A[i4+1][j4]+A[i4][j4-1]
                      +A[i4][j4]+A[i4][j4+1]
                      +A[i4+1][j4-1])/9.0 ; } }}
```

Fig. 4. Unrolled and jammed xfor-loop nest of the seidel code

B. Generating or avoiding pipeline hazards

Consider the seidel code of Fig. 4, which is the result of the initial version in Fig. 2 after unrolling and jamming the outer xfor loop one time, in order to increase the number of computations and memory accesses per iteration. Let us consider two versions defined by different offset values: a first version called V_1 with outer loop offsets (0,0,0,0,1) and inner loop offsets (0,0,0,0,0), and a second version V_2 with offsets (0,0,0,0,0) and (1,0,2,0,1). Both versions are run on an Intel Core i5-3470 Ivybridge CPU at 3.2GHz running Linux 3.11.0, and compiled using GCC 4.8.1 with flags -O3 -march=native -fno-tree-vectorize –param max-completely-peeled-insns=0. The two latter flags are used to avoid automatic vectorization and loop unrolling in order to measure the actual impact of data locality on execution times. In addition to the execution time, the energy consumption (MSR_PKG_ENERGY_STATUS), the number of stalled cycles, and the number of L1 and L2 cache misses have also been measured:

	V_1	V_2
exec. time (sec.)	1.294815	3.204376
cons. energy (joules)	21.170624	47.409576
L1 misses	120,190,581	80,150,636
L2 misses	40,090,106	40,086,200
stalled cycles	3,732,635,854	10,285,801,229

Surprisingly, both code versions exhibit similar numbers of cache misses, while performing very differently regarding their execution times, their numbers of stalled cycles, and obviously their energy consumptions. Thanks to Intel Vtune profiling tool, a precise view of the CPU time spent by each assembly instructions composing the main loop body of V_2 was obtained. It is shown in Figure 5. It clearly highlights excessive times used by instructions and groups of instructions of version V_2, while version V_1 is not showing such similar times. Groups of instructions spending up to hundreds of milliseconds are exhibiting dependences due to accesses to common registers and pressure on the floating-point unit simultaneously.

```
vaddsd %xmm2, %xmm3, %xmm1              |
lea (%r8,%rdx,1), %r11                  |
add $0x1, %ecx                          |   47
vaddsd %xmm12, %xmm3, %xmm3             |
add $0x8, %rdx                          |
vaddsd %xmm4, %xmm5, %xmm5              |
vmovsdq  %xmm1, -0x10(%rax)             |   41
vmovsdq  %xmm3, -0x18(%rdx)             |
vmovsdq  (%rax), %xmm0                  |
vaddsd %xmm10, %xmm0, %xmm2             |
vmovsdq  %xmm2, -0x8(%rax)              |   57
vaddsd %xmm13, %xmm2, %xmm2             |
vmovsdq  %xmm5, -0x10(%rdx)             |   43
vmovsdq  (%r11,%rsi,1), %xmm8           |
lea (%rdi,%rax,1), %r11                 |
add $0x8, %rax                          |
vaddsd %xmm7, %xmm2, %xmm2              |   35
vaddsd %xmm8, %xmm2, %xmm2              |   91
vaddsd %xmm2, %xmm1, %xmm2              |  142
vaddsd %xmm5, %xmm1, %xmm1              |  126
vdivsd %xmm9, %xmm2, %xmm2              |
vaddsd %xmm1, %xmm2, %xmm1              |  889
vmovsdq  %xmm2, -0x10(%rax)             |  120
vaddsd %xmm1, %xmm0, %xmm1              |
vaddsd %xmm1, %xmm3, %xmm3              |  121
vaddsd %xmm6, %xmm4, %xmm1              |  148
vdivsd %xmm9, %xmm3, %xmm3              |
vaddsd %xmm3, %xmm0, %xmm0              |  913
vmovsdq  %xmm3, -0x10(%rdx)             |  155
vmovsdq  %xmm0, -0x8(%rax)              |
vmovsdq  (%rdx), %xmm4                  |   49
vmovsdq  %xmm1, -0x8(%rdx)              |    5
vaddsd %xmm4, %xmm0, %xmm0              |
vmovsdq  %xmm0, -0x8(%rax)              |   76
vmovsdq  0x8(%r11,%r9,1), %xmm10        |   43
vaddsd %xmm10, %xmm1, %xmm5             |    4
vmovsdq  %xmm5, -0x8(%rdx)              |
```

Fig. 5. Total aggregated CPU time per instructions for version V_2 (ms)

In order to relate these observations to the xfor source codes of versions V_1 and V_2, the loop bodies are rewritten to use only one unique index domain by subtracting the statements respective offsets. They are shown in Fig. 6 and 7. It can be observed that version V_2 successively accesses elements belonging to two different rows of array A, while version V_1 accesses elements belonging to three different rows. Thus, version V_2 generates more dependent reuses between neighboring instructions, occurring either at the current iteration or at the preceding or the following iterations. More precisely, array elements of row `i` are updated two times and read two times inside each of three successive j-iterations, as referenced elements `A[i][j]`, `A[i][j-1]` and `A[i][j-2]`. Thus they are globally updated five times and read six times by closely executed instructions. Similarly, during four successive j-iterations, elements of row `i+1` are successively read as `A[i+1][j+1]`, updated two times and read as `A[i+1][j]`, updated two times and read two times as `A[i+1][j-1]`, and finally updated and read two times as `A[i+1][j-2]`. They are also globally updated five times and read six times by closely executed instructions. A similar analysis of version V_1 shows that successive accesses to array elements are slightly reduced while being distributed among elements of three different rows. Write and read counts are summarized in the following table:

```
0: { A[i][j] += A[i][j+1] ;
      A[i+1][j] += A[i+1][j+1] ; }
1: { A[i][j] += A[i+1][j-1] ;
      A[i+1][j] += A[i+2][j-1] ; }
2: { A[i][j] += A[i+1][j] ;
      A[i+1][j] += A[i+2][j] ; }
3: { A[i][j] += A[i+1][j+1] ;
      A[i+1][j] += A[i+2][j+1] ; }
4: { A[i-1][j] = (A[i-1][j]+A[i-2][j-1]
                 +A[i-2][j]+A[i-2][j+1]
                 +A[i-1][j-1])/9.0 ;
     A[i][j]   = (A[i][j]+A[i-1][j-1]
                 +A[i-1][j]+A[i-1][j+1]
                 +A[i][j-1])/9.0 ; }
```

Fig. 6. Loop Body of version V_1 into a unique index domain

```
0: { A[i][j-1] += A[i][j] ;
      A[i+1][j-1] += A[i+1][j] ; }
1: { A[i][j] += A[i+1][j-1] ;
      A[i+1][j] += A[i+2][j-1] ; }
2: { A[i][j-2] += A[i+1][j-2] ;
      A[i+1][j-2] += A[i+2][j-2] ; }
3: { A[i][j] += A[i+1][j+1] ;
      A[i+1][j] += A[i+2][j+1] ; }
4: { A[i][j-1] = (A[i][j-1]+A[i-1][j-2]
                 +A[i-1][j-1]+A[i-1][j]
                 +A[i][j-2])/9.0 ;
     A[i+1][j-1] = (A[i+1][j-1]+A[i][j-2]
                   +A[i][j-1]+A[i][j]
                   +A[i+1][j-2])/9.0 ; }
```

Fig. 7. Loop Body of version V_2 into a unique index domain

	accessed rows	succ. writes & reads
V_1	`i-1`	5 reads + 1 write
	`i`	3 reads + 5 writes
	`i+1`	4 reads + 4 writes
V_2	`i`	6 reads + 5 writes
	`i+1`	6 reads + 5 writes

Although version V_2 exhibits better data locality than version V_1, the numerous dependences between consecutive instructions induces a huge amount of stalls significantly impacting the execution time. Version V_1, which performs approximately 2.5x better, shows that to reach the best performance, a subtle balance between data locality and number of dependencies between instructions has to be found. The proposed xfor construct facilitates the finding of this balance thanks to its explicit control of the data reuse distances.

However, version V_2 may have a special interest regarding energy consumption: since V_2 is performing similarly to V_1 regarding its memory accesses, its power consumption, *i.e.*, $E_2 =$ (energy)/(execution time) $= 14.8$ watts, is significantly lower than $E_1 = 16.35$ watts. It means that one second spent in the execution of V_2 is about 9.5% cheaper than one second spent in the execution of V_1 in terms of energy consumption.

IV. Energy savings provided by the xfor construct

Benefits provided by xfor constructs for energy savings are twofold: either xfor enables the programmer to write very fast codes performing a reduced number of memory accesses, thanks to good data locality, and a reduced number of stalled cycles, and thus globally using a limited amount of energy; or xfor enables one to program codes performing a reduced number of memory accesses and a higher number of stalled cycles, and thus globally using less power.

Dynamic Voltage and Frequency Scaling (DVFS) is a common technique for saving energy, by scaling down voltage and frequency. Thus, since xfor codes are performing generally better than standard equivalent codes, xfor codes may be run at lower voltage and frequency, while still providing similar time performance than standard codes run at maximum frequency, but with a significantly lower energy consumption.

V. XFOR iterative compilation for data locality and controlled stalled cycles

While offset and grain parameters enable programmers to optimize locality of accessed data on the source code, it may be difficult to directly find out the best combination of their values inducing minimum (or maximum) stalled cycles, since it is strongly dependent on the compiler and the target processor. A convenient approach is to iteratively search this combination by successive experiments. The strategy is to initialize the process using the combination promoting the best data locality, and then to make each offset value vary in the small interval $[-2, 2]$, so as not to alter data locality too much. In order to reduce the search space, it is sufficient to proceed from the outermost to the innermost loop level. As soon as a good offset combination has been found for a given loop depth, the iterative process is reapplied for the inner loop depth, and so on. Note also that to accelerate the process, the code may be run using a small problem size, sufficient to highlight the processor behavior regarding stalls. Note also that any offset combination used must be valid regarding data dependences. Application of this iterative search on our target Ivybridge processor for the seidel code resulted in the offsets of versions V_1 and V_2 presented in Section III.

VI. Experiments

Experiments have been conducted on an Intel Core i5-3470 CPU at 3.20GHz (Ivybridge) processor running Linux 3.11.0. Our set of benchmarks has been built from the Polyhedral Benchmark suite [12], from which representative codes exhibiting data reuse have been selected. Every code has been rewritten using the xfor structure. Original and xfor versions have been compiled using GCC 4.8.1 with flags -O3 -march=native -fno-tree-vectorize –param max-completely-peeled-insns=0, and their outputs have been compared to ensure correctness of the xfor codes. In the following tables, execution times of the main loop kernels, original and rewritten as xfor loops, are given in seconds, energy consumptions are given in joules and have been obtained thanks to the MSR_PKG_ENERGY_STATUS processor counter.

VII. Related work

Improving energy efficiency is attracting increasing research interest, thus a wide palette of approaches have been proposed. Many of these rely either on adapting the hardware to match the code's behaviour [15] or vice-versa, compile-time

Code	**Pb size**	**Offset**	**Orig. time + Energy cons.**	**XFOR time + Energy cons.**	**Speed -up + Energy saving**
2mm	1024	0,0 0,nj 0,0	14.592 237.204	3.498 57.379	**4.17** **75.81%**
3mm	1024	0,ni,2*ni 0,0,0 0,0,0	21.875 354.900	4.692 76.382	**4.66** **78.48%**
jacobi-2d	16K	0,1 0,1	72.990 503.272	22.937 167.256	**3.18** **66.77%**
jacobi-1d	200M	0,2	0.902 14.535	0.558 9.594	**1.71** **33.99%**
fdtd-2d	10K	0,0,0,0 0,0,0,0	0.921 15.535	0.503 8.679	**1.83** **44.13%**
fdtd-apml	256	0,0,0,0 0,0,1,1 0,Cxm,0,Cxm	0.439 6.614	0.242 3.877	**1.81** **41.38%**
reg-detect	256 × 700	0,0,0,0,0,0,0,5 0,0,0,0,0,0,0,0 0..0,lgth-1,lgth,lgth	0.197 3.254	0.173 2.901	**1.14** **10.85%**
correlation	700	0,1,m+1,n+m+2 0,0,0,0 0,0,0,0	1.442 22.846	0.195 3.294	**7.39** **85.58%**
covariance	700	0,m,n+m 0,0,0 0,0,0	1.440 22.892	0.198 3.395	**7.27** **85.17%**
mvt	10K	0,0 0,0	1.570 24.743	0.222 3.625	**7.07** **85.35%**
gemver	10K	0,N,N,2N 0,0,N,0	1.857 28.849	0.490 8.070	**3.79** **72.03%**
seidel	4K	0,0,0,0,1 0,0,0,0,0	3.425 50.164	1.297 21.320	**2.64** **57.50%**
syrk	1024	0,0 0,0	1.757 27.760	1.699 26.854	**1.03** **3.26%**
syr2k	1024	0,0 0,0	2.671 43.727	2.429 38.515	**1.10** **11.92%**

TABLE I. CODE MEASUREMENTS AND COMPARISONS

transformations were dedicated to transforming the software to meet the hardware capabilities [10], [9]. In contrast, our proposal transfers the control to the programmer, who has now explicit ways to control data locality, pipeline stalls, and, thus, energy consumption, thanks to the offset and grain parameters of xfors. Since until now the only software approaches to enable energy saving optimizations relied on compilers, in what follows, we review works of this type.

In some cases, the most energy efficient solution is to complete the task as fast as possible, which causes higher power, but decreases the execution time, thus reducing the energy. In other situations, transforming the code to execute within a power budget and save energy, may harm performance. Significant efforts have been made to achieve a better balance between the two [15], [11], [7], [14], [8], [6], [5], [9].

Dynamic Voltage and Frequency Scaling (DVFS) is a common technique for saving energy, by scaling down voltage and frequency. To avoid considerable performance penalties, DVFS is traditionally applied on applications exhibiting memory bound phases [15], [11], [7], [8], given that the granularity of these phases is coarse enough to overcome the inherent overhead of frequency scaling. Jimborean et al. [9] propose compile-time code transformations that decouple memory accesses and computation in order to adapt the code for more efficient DVFS. Yet, as proved by Yuki et al. [17], frequency scaling is mostly suitable for memory bound codes, whereas compute-bound codes exhibit significant performance degradations when run at lower frequencies. Compute-bound applications are generally optimized for performance, which is commonly known as "race to sleep" [17], yielding energy savings as positive side-effects. In contrast, Saputra et al. [14] apply traditional compiler optimizations (loop fusion, tiling, etc) and scale down frequency to a lower value, which reduces the energy, while maintaining the performance of the original (unoptimized) code under maximum frequency.

Apart from DVFS techniques, other proposals target compiler-architecture collaborations, which enable a wiser use of the micro-architecture based on static information [6], [5], [16]. In particular, Finlayson et al. [6], [5] focus on improving the processor pipeline and propose an entirely statically pipelined processor, relying on an optimizing compiler to insert control information for each instruction. Thus, redundant operations performed by traditional processor pipelines are eliminated (e.g., unnecessary register reads and writes, when their consumers retrieve such information from forwarding). Whereas this approach requires a modified architecture, we rely on commodity hardware and exploit pipeline hazards for recording energy savings. The xfor programming structure is a knob that enables the programmer to easily create multiple code versions and control the amount of energy, while meeting the performance demands.

Previous works targeting data locality have as consequence improvements in the energy consumption [4], [13]. However, xfor can go beyond improving data locality, and not only it enables the programmer to accurately control the trade-off between execution time and energy, but it also proves that with appropriate language extensions an expert can create high-performing code, which cannot be generated by automatic tools.

VIII. CONCLUSION

We have described the new xfor programming construct that gives the programmer explicit control of the data reuse distance through the use of xfor's offset and grain parameters. It has been shown that it eases execution-time and energy optimizations. It also highlights the impact of stalled processor cycles on codes that still exhibit good data locality, and enables the programmer to control the amount of stalled cycles either for execution time or for power optimizations.

REFERENCES

[1] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16.

[2] ——, "Openscop: A specification and a library for data exchange in polyhedral compilation tools," Paris-Sud University, France, Tech. Rep., September 2011.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI '08*. ACM, 2008, pp. 101–113.

[4] ——, "A practical automatic polyhedral parallelizer and locality optimizer," in *2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2008, pp. 101–113.

[5] I. Finlayson, B. Davis, P. Gavin, G.-R. Uh, D. Whalley, M. Själander, and G. Tyson, "Improving processor efficiency by statically pipelining instructions," in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '13. New York, NY, USA: ACM, 2013, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2465554.2465559

[6] I. Finlayson, G.-R. Uh, D. Whalley, and G. Tyson, "Improving low power processor efficiency with static pipelining," in *Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures*, ser. INTERACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 17–24. [Online]. Available: http://dx.doi.org/10.1109/INTERACT.2011.7

[7] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application transformations for energy and performance-aware device management," in *Parallel Architectures and Compilation Techniques*, 2002, pp. 121–130.

[8] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," *SIGPLAN Not.*, pp. 38–48, 2003.

[9] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 262:262–262:272. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544161

[10] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards More Efficient Execution : A Decoupled Access-Execute Approach Categories and Subject Descriptors," in *ICS'13*, 2013.

[11] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," *SIGMETRICS Perform. Eval. Rev.*, 2001.

[12] "The Polyhedral Benchmark suite." [Online]. Available: http://www.cse.ohio-state.edu/ pouchet/software/polybench

[13] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: convexity, pruning and optimization," in *38thACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2011, pp. 549–562.

[14] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer, "Energy-conscious compilation based on voltage scaling," in *Joint Conf. on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002, pp. 2–11.

[15] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling and power management for portable systems," in *Design Automation Conf.*, 2001, pp. 524–529.

[16] S. Tavarageri and P. Sadayappan, "A compiler analysis to determine useful cache size for energy efficiency," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 923–930.

[17] T. Yuki and S. Rajopadhye, "Folklore confirmed: Compiling for speed = compiling for energy," in *26th Int'l Workshop on Languages and Compilers for Parallel Computing*, 2013.