XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance

Imen Fassi, Philippe Clauss

XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance

Imen Fassi[a,b]
[a]URAPOP, Faculté des Sciences de Tunis
University El Manar, Tunis, Tunisia
Email: imen.fassi@inria.fr

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

***Abstract*—We propose a new loop structure named *xfor*, offering programmers explicit control of the interactions between statements inside a loop nest. An xfor simultaneously represents several for-loops and several statements, and maps their respective iteration domains onto each other according to two parameters, called *grain* and *offset*. Grains and offsets basically "stretch" and "shift" iteration domains relative to an implicit, global referential domain. We show that such a programming structure allows to fill important optimization gaps remained by automatic loop optimizers. We highlight five important gaps filled by xfor which are: insufficient data locality optimization, excess of conditional branches in the generated code, too verbose code with too many machine instructions, data locality optimization resulting in processor stalls, and finally missed vectorization opportunities. We describe programming strategies where xfor-loops help produce efficient code and exhibit a set of benchmark programs rewritten with xfor, with significant, and sometimes dramatic, execution time speed-ups.**

I. Introduction

Loops and loop nests are main targets of the optimizations performed by compilers, and the literature related to loop optimizing transformations is substantial. Loops have their own analysis and transformation mathematical framework called the *polyhedral model* [1] which has generated an intense and prolific research activity for several decades. Perhaps the most well-known software tool that has emerged is the source-to-source polyhedral compiler *Pluto* [2], [3]. Pluto implements some of the most advanced loop optimizing strategies. However, even if it is probably the best automatic loop optimizer, it must inherently be based on heuristics. By definition, heuristics are sub-optimal and fallible. On the other hand, works on iterative and machine learning compilation frameworks [4], [5], [6], [7] are proof of the high complexity of code optimization, even when handling loops exhibiting seemingly simple shapes, as loops with linear bounds and linear memory references that are all in the scope of Pluto [4], [5]. Finally, the ever evolving hardware complexity and the nature of the codes generated by back-end compilers are also important issues preventing automatic optimizers of being wholly foolproof, since they can never address all the possible and forthcoming performance issues simultaneously.

Thus there will always be a significant gap between the runtime performance that may be reached thanks to the best automatic optimizers, and the peak performance that could be expected from an optimized code that is run on a given hardware, whose resources are still underused.

To fill this gap, we propose to make available programming structures to users, enabling them to apply, with relative ease, some advanced and efficient optimizing transformations to their codes, while alleviating the related burden thanks to the assistance of automatic code generators.

Following this idea, we propose a computer-assisted control structure called *xfor*, helping programmers in addressing directly and accurately three main issues regarding performance: well balanced data locality improvement through generalized loop fusion and loop fission, vectorization and loop parallelization. Two parameters in this structure, the *offset* and the *grain*, afford to adjust precisely the schedule of statements and their interactions regarding data reuse, 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 also benefit of optimizations implemented in the polyhedral code generator CLooG [8] which is invoked by IBB to generate for-loops from xfor-loops.

We show that the xfor structure helps in highlighting important performance issues, that could not have been clearly identified before for some of them. By comparing xfor-generated codes to Pluto-generated codes, and also xfor-codes among each other, we highlight five important gaps in the currently adopted and well-established code optimization strategies: *insufficient data locality optimization, excess of conditional branches in the generated code, too verbose code with too many machine instructions, data locality optimization resulting in processor stalls*, and finally *missed vectorization opportunities.*

We illustrate the importance of these issues in program optimization using eleven representative codes extracted from the Polybench benchmark suite [9]. Every code has been rewritten using the xfor structure and also optimized by the most recent version of Pluto with the combination of options generating the best performing code.

The paper is organized as follows. First, we present the xfor syntax and semantics in Section II, together with the xfor compiler IBB. Next in Section III, we focus on five important performance issues by relating them as being jointly the cause of wasted processor cycles. Each highlighted issue is addressed in a dedicated subsection using illustrative benchmark codes. In Section IV, we explain two dual xfor programming strategies related to inter-statement and intra-statement data-reuse distance minimization, and show that simple loop transformations yield even more efficient xfor

codes. In Section V, we specifically address OpenMP multi-threaded xfor-loop parallelization. Section VI is dedicated to experimentations in which Pluto-optimized and xfor versions are compared regarding sequential, vectorized, and loop-parallelized executions. Section VII addresses related work while conclusions are given in Section VIII.

II. XFOR SYNTAX AND SEMANTICS

In this section, we recall the xfor syntax and semantics initially presented in [10] and later updated in [11]. 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 in two iterations 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) and their order in the loop body (statements are interleaved according to this order).

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.

```
xfor (i0=1,i1=1,i2=1,i3=1,i4=1,i5=1,i6=1 ;
      i0<N,i1<N,i2<N,i3<N,i4<N,i5<N,i6<N ;
      i0++,i1++,i2++,i3++,i4++,i5++,i6++ ;
      1,1,1,1,1,1,1 ; /* grains */
      0,1,0,2,1,1,2)  /* offsets */ {
xfor (j0=1,j1=1,j2=1,j3=1,j4=1,j5=1,j6=1 ;
      j0<N,j1<N,j2<N,j3<N,j4<N,j5<N,j6<N ;
      j0++,j1++,j2++,j3++,j4++,j5++,j6++ ;
      1,1,1,1,1,1,1 ; /* grains */
      1,1,1,1,0,2,1)  /* offsets */ {

      0: b[i0][j0] =  0           ;
      1: b[i1][j1] += a[i1][j1]   ;
      2: b[i2][j2] += a[i2-1][j2] ;
      3: b[i3][j3] += a[i3+1][j3] ;
      4: b[i4][j4] += a[i4][j4-1] ;
      5: b[i5][j5] += a[i5][j5+1] ;
      6: b[i6][j6] =  b[i6][j6]/5 ; }}
```

Fig. 1. XFOR code example

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.

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.

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 polyhedra over a common referential domain, and second, scanning code is generated for their union. The second step is performed using the CLooG library [8] devoted to generate code for scanning unions of polytopes.

Example. The xfor loop nest of Figure 1 implements an optimized version of a stencil computing the average of every element of an array `a` and its four neighbors, and writing the result in array `b`. The code generated by IBB from this code is shown in Figure 2. 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. Without xfor, a user would have to write codes similar to the one of Figure 2 to reach the performance attainable with xfor.

```
for (j=1;j<=N-1;j++)
  b[1][j]=a[0][j];
b[1][1]+=a[1][0];
b[1][1]+=a[1][1];
b[1][2]+=a[1][1];
b[2][1]=a[1][1];
for (j=2;j<=N-2;j++) {
  b[1][j-1]+=a[1][j];
  b[1][j]+=a[1][j];
  b[1][j+1]+=a[1][j];
  b[2][j]=a[1][j];}
b[1][N-2]+=a[1][N-2+1];
b[1][N-1]+=a[1][N-1];
b[2][N-1]=a[1][N-1];
b[1][N-1]+=a[1][N];
for (i=2;i<=N-2;i++) {
  b[i][1]+=a[i][0];
  b[i-1][1]+=a[i][1];
  b[i-1][1]/=5;
  b[i][1]+=a[i][1];
  b[i][2]+=a[i][1];
  b[i+1][1]=a[i][1];
  for (j=2;j<=N-2;j++) {
    b[i-1][j]+=a[i][j];
    b[i-1][j]/=5;
    b[i][j-1]+=a[i][j];
    b[i][j]+=a[i][j];
    b[i][j+1]+=a[i][j];
    b[i+1][j]=a[i][j]; }
  b[i-1][N-1]+=a[i][N-1];
  b[i-1][N-1]/=5;
  b[i][N-2]+=a[i][N-1];
  b[i][N-1]+=a[i][N-1];
  b[i+1][N-1]=a[i][N-1];
  b[i][N-1]+=a[i][N]; }
```

```
/* continued */
b[N-1][1]+=a[N-1][0];
b[N-2][1]+=a[N-1][1];
b[N-2][1]/=5;
b[N-1][1]+=a[N-1][1];
b[N-1][2]+=a[N-1][1];
for (j=2;j<=N-2;j++) {
  b[N-2][j]+=a[N-1][j];
  b[N-2][j]/=5;
  b[N-1][j-1]+=a[N-1][j];
  b[N-1][j]+=a[N-1][j];
  b[N-1][j+1]+=a[N-1][j];}
b[N-2][N-1]+=a[N-1][N-1];
b[N-2][N-1]/=5;
b[N-1][N-2]+=a[N-1][N-1];
b[N-1][N-1]+=a[N-1][N-1];
b[N-1][N-1]+=a[N-1][N];
for (j=1;j<=N-1;j++) {
  b[N-1][j]+=a[N-1+1][j];
  b[N-1][j]/=5; }
```

Fig. 2. Code example automatically generated by IBB

III. Wasted processor cycles

The execution time of a program is obviously directly related to the total number of cycles spent by the CPU for running all of its instructions. Among these consumed cycles, some of them may be stalled, and some others may be spent uselessly in running a too verbose set of instructions that perform computations that could either have been achieved using a significantly smaller set of instructions, or by taking advantage of some accelerator processor units using the dedicated instructions. The latter issue is detailed in subsection III-E regarding vectorization, while the previous one is addressed in subsection III-C. It is shown that codes exhibiting a good data locality may be even slower than codes with weaker locality, just because of one of both issues.

Stalled processor cycles are cycles spent by the processor in waiting for the completion of some event on which the continuation of the current instruction sequence depends. Thus these cycles are wasted since they are uselessly consuming time and energy. Although such processor stalls can never be completely avoided, or may be partially hidden by simultaneous instruction executions, their amount should be minimized. For this purpose, their cause have to be handled specifically when optimizing programs. They can be classified into four main categories:

1) *stalls due to insufficient computing resources:* for example, the processor core is not embedding enough floating-point units while several floating point operations are ready to be performed simultaneously.

2) *stalls due to memory latency:* this issue is one of the most frequently handled issues in program optimization techniques, with goals like data locality improvement and minimization of cache misses.

3) *stalls due to dependences between instructions:* the executed code contains many sequences of dependent instructions, *i.e.*, instructions for which at least one operand is reused in some closely following instructions in a Read-After-Write fashion. Such a situation prevents superscalar microprocessors to launch simultaneously several instructions due to the unavailability of operands. This may potentially occur with codes resulting from aggressive data locality optimization, since data reuse distances are traditionally minimized by bringing as close as possible instructions referencing common data which may be dependent.

4) *stalls due to branch mispredictions:* When a branch prediction made by the CPU is incorrect, all the speculatively executed instructions are discarded as soon as the correct branch is determined, and the processor execution pipeline restarts with instructions from the correct branch destination. This halt while the new instructions work their way down the execution pipeline causes a processor stall, which is a major drain on performance.

While point 1 can be solved using more hardware, point 2 is handled by most compilers which implement data locality optimization techniques that are more or less efficient. Regarding linear loop nests, the Pluto source-to-source compiler [2], [3] implements some of the most advanced data locality optimization strategies based on the polyhedral model, e.g. some advanced tiling techniques, loop interchange, skewing, etc. However, the heuristics that are used necessarily miss some optimization opportunities that may be handled by an expert programmer, particularly when using the xfor structure, as it will be shown in the next subsection. All in all, the strategies used are not conscious of the other performance issues described below, and may have such a negative impact that they annihilate the gain provided by data locality improvement, as it will be shown in the following subsections.

Regarding points 3, this issue is never addressed explicitly by automatic optimizers since data locality optimization is always considered as a final goal. However, we show in subsection III-D that code versions that are all exhibiting similar and "optimized" memory performance (and similar performance regarding all the other points) may show very different execution times because of this issue. Additionally, the minimization of data reuse distances among instructions may prevent vectorization of these instructions (subsection III-E).

Regarding point 4, while branch predictors cannot be controlled by software, the potential risk with loop transformations regarding branch mispredictions is related to the kind of optimizing transformation that has been applied and the number of

#CPU cycles:	total number of CPU cycles, halted and unhalted.
#L1 data loads:	total number of data references to the L1 cache.
#Li misses:	total number of loads that miss the Li cache.
#TLB misses:	total number of load misses in the TLB that cause a page walk.
#branches:	total number of retired branch instructions.
#branch misses:	total number of branch mispredictions.
#Stalled cycles:	total number of cycles in which no micro-operations are executed on any port.
#Resource related stalls:	total number of allocator resource related stalls.
#Reservation Station stalls:	Number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly instructions dependent upon instructions further down the pipeline that have yet to retire). Regarding program analysis, a high count of this event most probably exhibits the effect of long chains of dependences between close instructions.
#Re-Order Buffer stalls:	Number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit. A high count for this event indicates that there are long latency operations in the pipe (possibly, load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). Regarding program analysis, a high count of this event most probably exhibits the effect of long latency memory operations and TLB or cache misses.
#instructions:	total number of retired instructions.

TABLE I. CPU EVENTS COLLECTED USING `libpfm`

	Pluto	XFOR	Ratios
mvt			
#CPU cycles	3,824M	2,425M	-36.58%
#L1 data loads	748M	451M	-39.71%
#L1 misses	45M	50M	+10.71%
#L2 misses	**29M**	**5.8M**	**-80.09%**
#L3 misses	**38M**	**14M**	**-63.77%**
#TLB misses	**3.8M**	**0.7M**	**-82.62%**
#branches	224M	212M	-4.89%
#branch misses	470K	439K	-6.58%
#instructions	2,469M	2,010M	-18.58%
syr2k			
#CPU cycles	7,005M	5,671M	-19.05%
#L1 data loads	4,322M	2,158M	-50.06%
#L1 misses	**299M**	**137M**	**-54.18%**
#L2 misses	**8.4M**	**3.6M**	**-55.94%**
#L3 misses	**10M**	**5.1M**	**-48.57%**
#TLB misses	**4.3M**	**3.2M**	**-25.78%**
#branches	1,072M	1,078M	+0.58%
#branch misses	1,072K	1,084K	+1.03%
#instructions	11,890M	13,946M	+17.29%
3mm			
#CPU cycles	17,557M	4,358M	-75.18%
#L1 data loads	4,226M	2,440M	-24.36%
#L1 misses	**815M**	**206M**	**-74.67%**
#L2 misses	**554M**	**5.4M**	**-99.02%**
#L3 misses	**174M**	**3M**	**-98.25%**
#TLB misses	**541M**	**3.2M**	**-99.41%**
#branches	1,625M	813M	-49.96%
#branch misses	2,704K	1,630K	-39.73%
#instructions	11,331M	8,941M	-21.09%
gauss-filter			
#CPU cycles	3,457M	2,963M	-14.28%
#L1 data loads	873M	843M	-3.45%
#L1 misses	**75M**	**46M**	**-38.97%**
#L2 misses	**4.2M**	**2.4M**	**-42.33%**
#L3 misses	**29.5M**	**24.8M**	**-15.91%**
#TLB misses	**1.5M**	**0.7M**	**-49.78%**
#branches	724M	572M	-20.92%
#branch misses	622K	689K	+10.78%
#instructions	5,026M	4,652M	-7.44%

TABLE II. XFOR SPEEDUPS ATTRIBUTABLE TO DECREASED TLB AND CACHE MISSES

branches resulting from it in the executable code. The classic tiling transformation may present such a potential risk due to the complicated control it requires, particularly when it involves non-rectangular shapes. This point is addressed in subsection III-B.

In the following subsections, we illustrate the importance of these issues in program optimization using eleven representative codes extracted from the Polybench benchmark suite [9]. Every code has been rewritten using the xfor structure and also optimized by the most recent version of the source-to-source Pluto polyhedral compiler [3] with the combination of options generating the best performing code among `--tile` (with the default tile size of 32 in each tilable dimension), `--l2tile`, `--smartfuse`, `--maxfuse` and `--rar` . Xfor and Pluto versions have been compiled using GCC 4.8.1 with options O3 and march=native, and are compared regarding several relevant processor performance counters whose values were collected using the `perf` linux tool and the `libpfm` library [12]. The collected CPU events are detailed and commented in Table I. Notice that the origins of stalls are generally difficult to classify using CPU events. Each performance counter related to stalled cycles monitors a particular hardware unit that may stall for many reasons, and several units may stall for a common reason. Thus the reported counters in the following subsections provide some hints about the origins of some stalls, but can never be exhaustive. Experiments have been conducted on an Intel Xeon X5650 6-core processor 2.67GHz (Westmere) running Linux 3.2.0.

Among the eleven benchmark codes, we identified the ones whose runtime behavior is more significantly impacted by one single performance issue among the five ones, even if in general, performance is a question of balance among the provided gains and overheads. Thus these eleven codes have been selected because they enable such discrimination for pedagogical purposes. Notice also that the highlighted issues are independent of the compiler. We have observed similar runtime behaviors with codes compiled with the Intel compiler ICC, excepting for automatic vectorization which is generally better handled by ICC.

A. Gap 1: Insufficient data locality optimization

Tables II and III show four codes whose best Pluto-optimized versions are compared to better performing xfor-optimized versions. By comparing their respective performance counters, one can observe that the number of stalled cycles and the number of TLB and data cache misses are showing important differences, while the other values do not show such significant disparity. From more than 25% up to 99% more TLB misses, and more than 15% up to 98% more L3 misses, have been observed with the Pluto codes, obviously yielding more stalled cycles associated to larger memory access latencies. The origin of this higher amount of stalls is specifically highlighted by the high count of re-order buffer stalls which are symptomatic of long latency memory operations.

Pluto's heuristics do not seem to promote temporal data reuse among different statements at all, despite the `--rar`

	Pluto	XFOR	Ratios
mvt - #stalled cycles	**2,742M**	**1,582M**	**-42.29%**
#Resource related stalls	**2,544M**	**1,347M**	**-47.05%**
#Reservation Station stalls	431M	447M	+3.63%
#Re-Order Buffer stalls	**2,008M**	**771M**	**-61.62%**
syr2k - #stalled cycles	**1,570M**	**1,346M**	**-14.27%**
#Resource related stalls	**1,495M**	**1,332M**	**-10.91%**
#Reservation Station stalls	327M	1,199M	+266.50%
#Re-Order Buffer stalls	**1,182M**	**132M**	**-88.80%**
3mm - #stalled cycles	**12,695M**	**524M**	**-95.87%**
#Resource related stalls	**12,392M**	**387M**	**-96.87%**
#Reservation Station stalls	**10,667M**	**379M**	**-96.44%**
#Re-Order Buffer stalls	**2,606M**	**38M**	**-98.52%**
gauss-filter - #stalled cycles	**1,351M**	**1,196M**	**-11.45%**
#Resource related stalls	**924M**	**824M**	**-10.82%**
#Reservation Station stalls	174M	150M	-13.88%
#Re-Order Buffer stalls	**171M**	**134M**	**-21.25%**

TABLE III. DECREASED STALLS ATTRIBUTABLE TO DECREASED TLB AND CACHE MISSES

and `--maxfuse` options. For example, with `mvt`, Pluto did not detect the opportunity of interchanging loops of the second loop nest before merging them. With `syr2k`, the xfor code promotes the inter-statement data reuse of elements of matrices A and B, while the Pluto code prioritizes only intra-statement data locality for each single access to the matrices. Similar situations occur with `3mm` and `gauss-filter`.

B. Gap 2: Excess of conditional branches

Codes `seidel`, `correlation` and `covariance`, are symptomatic cases where loop tiling is more penalizing than advantageous, despite the fact that it may provide a significantly better cache performance. Pluto's best performing versions for these three codes are tiled versions embedding many additional loop levels and complex loop bounds made with combinations of *min, max, floor* and *ceiling* functions invocations (see Figure 3). This additional control yields many more branches in the final generated code than in a version built without tiling, and thus more machine instructions. No tiling has been applied in the xfor codes. Consequently, Pluto's codes are more exposed to branch misses as exhibited by the performance counters (see Table IV). Moreover, branches resulting from complex combinations of *min, max, floor* and *ceiling* may be hardly predictable. Thus, the larger amount of instructions and the related branch misses completely annihilate the gain expected from the significantly lower number of TLB misses generated with `seidel` and `covariance` Pluto's versions. Notice that for `covariance`, the xfor code is even exhibiting more stalled cycles than the Pluto code, although it is still globally faster.

Complex loop control yields also many more instructions of various kinds in the final executable than with simpler control, as it is clearly highlighted by the number of retired instructions for `seidel` and `covariance`. This issue, which is specifically addressed in the next subsection, impacts also solely performance significantly.

C. Gap 3: Number of instructions

Both Pluto and `XFOR1` codes of Table V are implementing a similar transformation of the original `jacobi-2d` code which consists in fusing both original loop nests in order to promote inter-statement data reuse and minimize loop control cost. Even if `XFOR1` and `XFOR2` exhibit a better data locality

```
for (t1=0;t1<=floord(tsteps-1,32);t1++)
 for (t2=t1;t2<=min(floord(32*t1+n+29,32),
      floord(tsteps+n-3,32));t2++)
  for (t3=max(ceild(64*t2-n-28,32),t1+t2);
       t3<=min(min(min(min(floord(32*t1+n+29,16),
       floord(tsteps+n-3,16)),
       floord(64*t2+n+59,32)),
       floord(32*t1+32*t2+n+60,32)),
       floord(32*t2+tsteps+n+28,32));t3++)
   for (t4=max(max(max(32*t1,32*t2-n+2),16*t3-n+2),
        -32*t2+32*t3-n-29);
        t4<=min(min(min(min(32*t1+31,32*t2+30),
        16*t3+14),tsteps-1),-32*t2+32*t3+30);t4++)
    for (t5=max(max(32*t2,t4+1),32*t3-t4-n+2);
         t5<=min(min(32*t2+31,32*t3-t4+30),t4+n-2);
         t5++)
     for (t6=max(32*t3,t4+t5+1);
          t6<=min(32*t3+31,t4+t5+n-2);t6++) {
      A[-t4+t5][-t4-t5+t6] = ...;
```

Fig. 3. Tiled loop nest generated by Pluto for `seidel`

	Pluto	XFOR	Ratios
seidel			
#CPU cycles	15,721M	7,476M	-52.45%
#L1 data loads	3,099M	672M	-78.31%
#L1 misses	12M	83M	+569.40%
#L2 misses	3.7M	1.2M	-65.64%
#L3 misses	3.9M	3.4M	-12.69%
#TLB misses	78K	688K	+783.18%
#branches	**387M**	**179M**	**-53.88%**
#branch misses	**456K**	**132K**	**-70.97%**
#stalled cycles	**11,297M**	**4,499M**	**-60.18%**
#Resource related stalls	**11,030M**	**4,4281M**	**-59.85%**
#Reservation Station stalls	**3,017M**	**440M**	**-85.39%**
#Re-Order Buffer stalls	**9,466M**	**3,982M**	**-57.93%**
#instructions	**10,015M**	**7,857M**	**-21.55%**
correlation			
#CPU cycles	425M	426M	+0.22%
#L1 data loads	224M	186M	-17.10%
#L1 misses	3.7M	12M	+223.95%
#L2 misses	2.2M	1M	-50.77%
#L3 misses	635K	395K	-37.83%
#TLB misses	294K	306K	+4.27%
#branches	**120M**	**78M**	**-34.39%**
#branch misses	**549K**	**231K**	**-58.01%**
#stalled cycles	**115M**	**47M**	**-58.79%**
#Resource related stalls	**81M**	**24M**	**-69.49%**
#Reservation Station stalls	**47M**	**3.7M**	**-92.10%**
#Re-Order Buffer stalls	**16M**	**14M**	**-13.31%**
#instructions	**906M**	**934M**	**+3.03%**
covariance			
#CPU cycles	419M	320M	-23.71%
#L1 data loads	217M	117M	-46.19%
#L1 misses	3.5M	22M	+539%
#L2 misses	1.9M	9M	+366.65%
#L3 misses	744K	496K	-33.42%
#TLB misses	247K	501K	+102.87%
#branches	**119M**	**35M**	**-70.40%**
#branch misses	**721K**	**199K**	**-72.37%**
#stalled cycles	**61M**	**123M**	**+100.75%**
#Resource related stalls	**59M**	**117M**	**+98.54%**
#Reservation Station stalls	**44M**	**43M**	**-1.40%**
#Re-Order Buffer stalls	**17M**	**75M**	**+344.54%**
#instructions	**1,050M**	**506M**	**-51.86%**

TABLE IV. XFOR SPEEDUPS PARTIALLY ATTRIBUTABLE TO DECREASED BRANCH MISPREDICTIONS

than Pluto's code (less caches misses), they also execute a significantly greater amount of instructions making them slower. The small differences in the reservation station and re-order buffer stalls show that the execution times differences are not significantly influenced by differences regarding memory operations or dependences between instructions.

jacobi-2d	Pluto	XFOR1	XFOR2
#CPU cycles	12,136M	13,700M	12,641M
#L1 data loads	1,400M	1,530M	1,529M
#L1 misses	236M	206M	205M
#L2 misses	44M	6M	11M
#L3 misses	76M	68M	68M
#TLB misses	2.7M	2.8M	3M
#branches	657M	564M	650M
#branch misses	1,560K	1,448K	1,329K
#stalled cycles	9,265M	9,463M	8,673M
#Resource related stalls	8,317M	8,433M	7,606M
#Reservation Station stalls	1,123M	1,088	930M
#Re-Order Buffer stalls	5,435M	4,775M	4,740M
#instructions	**6,950M**	**9,370M**	**10,469M**

TABLE V. XFOR SLOWDOWNS ATTRIBUTABLE TO HIGHER INSTRUCTION COUNTS

```
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 */
      ?,?,?,?,? ) /* 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 */
      ?,?,?,?,? ) /* 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. The xfor seidel code used for register dependence analysis

D. Gap 4: Unaware data locality optimization

We have written three xfor code versions of the polybench `seidel` code which just differ by their offset values. The xfor code is shown in Figure 4 while the offset values are shown in Table VI. Notice that these codes have a different shape than the xfor `seidel` code addressed in subsection III-B, which explains the different counter values. One can observe from the performance counters that these three codes are behaving mostly similarly at runtime, while showing important execution time differences. The only performance counters showing significant differences are those related to stalled cycles. However, neither the amount of branch misses, instructions, nor cache misses can explain these differences. Some of these numbers seem even slightly more favorable for the slowest code.

This performance issue is probably the most surprising one among the five issues highlighted in this paper. It is generally difficult to identify since it is usually hidden by other performance issues. The xfor structure allows to isolate it, thanks to its explicit control of the data reuse distances, which enables the generation of several code versions which are all exhibiting a similar well-optimized data locality.

Thanks to the Intel Vtune profiling tool, a precise view of the CPU time spent by the respective groups of most

seidel	XFOR1	XFOR2	XFOR3
offsets-i	0,0,0,0,1	0,1,0,0,1	0,1,1,1,1
offsets-j	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
#CPU cycles	7,392M	11,393M	12,283M
#L1 data loads	986M	997M	837M
#L1 misses	123M	123M	103M
#L2 misses	1.9M	1.9M	1.6M
#L3 misses	3.5M	3.5M	3.5M
#TLB misses	725K	694K	693K
#branches	97M	94M	96M
#branch misses	74K	78K	78K
#stalled cycles	**5,100M**	**8,002M**	**9,367M**
#Resource related stalls	**5,076M**	**7,969M**	**9,334M**
#Reservation Station stalls	**1,543M**	**7,765M**	**9,130M**
#Re-Order Buffer stalls	**3,537M**	**170M**	**157M**
#instructions	6,131M	7,146M	6,503M

TABLE VI. INCREASED STALLS ATTRIBUTABLE TO INCREASED REGISTER DEPENDENCES FOR THREE XFOR VERSIONS OF `seidel`

time-consuming assembly instructions of `XFOR1`, `XFOR2` and `XFOR3` is shown in Table VII. It clearly shows excessive times spent by some instructions. Instructions spending up to hundreds of milliseconds are exhibiting dependences due to accesses to common registers that could not be resolved through register renaming. These dependences are typically Read-After-Write (RAW) dependences. These excessive latencies are particularly exacerbated by the use of the x86 `divsd` floating-point division instruction which is costly: its latency is about 24 CPU cycles on Westmere microprocessors as reported in the related documentations. Thus, any delay regarding its execution has a significant impact on depending instructions, and any delay regarding instructions on which it depends extends significantly its latency.

Typically, in this example in Table VII, each instruction following immediately instruction `divsd` exhibits a high latency due to its RAW register dependence with instruction `divsd`: `movsdq` and register `xmm2` for `XFOR1`, `movsdq` and register `xmm0` for `XFOR2`, `addsd` and register `xmm1` for `XFOR3`.

These code examples show that a "too good" data locality may introduce long chains of many short dependences making instructions so tightly coupled that despite register renaming, and despite out-of-order execution, the microprocessor cannot find any independent instructions to launch simultaneously. This issue is particularly highlighted by the higher counts of the reservation station stalls in Table VI.

E. Gap 5: Insufficient handling of vectorization opportunities

Table VIII shows three codes whose xfor versions are significantly faster than Pluto's versions, although their respective performance counters are not exhibiting great differences. Some counters are even in contradiction with the execution times (number of TLB and cache misses). In contrast to the previous issue regarding short dependences between instructions, these codes are representative of another issue related to vectorization: the compiler automatically vectorized kernel loops of the xfor codes, while it did not for Pluto's codes. This has been clearly observed thanks to the `--ftree-vectorizer-verbose` GCC option.

Vectorization is subject to two main parameters: data dependence and alignment. Processors' SIMD units require fixed-size vectors, say sv, of equally spaced data, *i.e.*, spaced

XFOR1	ms
addsd %xmm7, %xmm0	
addsd %xmm1, %xmm0	44
divsd %xmm3, %xmm0	
movsdq %xmm0, -0x8(%r8)	8
movsdq -0x8(%rcx), %xmm2	
movsdq (%r9), %xmm13	72
addsd %xmm1, %xmm2	
movapd %xmm13, %xmm1	
addsd %xmm9, %xmm1	
addsd %xmm0, %xmm2	12
addsd %xmm7, %xmm1	
addsd %xmm13, %xmm2	
addsd %xmm5, %xmm2	
divsd %xmm3, %xmm2	20
movsdq %xmm2, -0x8(%rcx)	796
movsdq (%rax), %xmm11	28

XFOR2	ms
addsd %xmm11, %xmm2	
addsd %xmm0, %xmm2	70
addsd %xmm4, %xmm0	108
divsd %xmm3, %xmm2	
movsdq %xmm2, (%rdi)	542
addsd %xmm2, %xmm0	48
movsdq 0x8(%r9), %xmm9	64
addsd %xmm9, %xmm0	
addsd %xmm1, %xmm0	40
movapd %xmm10, %xmm1	78
divsd %xmm3, %xmm0	
movsdq %xmm0, (%rax)	526
movsdq 0x8(%rcx), %xmm4	40

XFOR3	ms
addsd %xmm9, %xmm0	28
addsd %xmm7, %xmm0	
addsd %xmm8, %xmm0	60
divsd %xmm3, %xmm0	48
movsdq %xmm0, -0x8(%rcx)	602
addsd %xmm0, %xmm1	20
movsdq (%r9), %xmm2	124
addsd %xmm2, %xmm1	
addsd %xmm13, %xmm1	96
divsd %xmm3, %xmm1	42
addsd %xmm1, %xmm2	824
movsdq %xmm1, -0x8(%rdx)	74

TABLE VII. TOTAL AGGREGATED CPU TIME PER INSTRUCTIONS (MS) – SOURCE: INTEL VTUNE

by constant memory strides. Thus, sv iterations are run in parallel thanks to the SIMD unit. Mainstream compilers featuring automatic vectorization also require straightforward memory access patterns. Thus, the xfor programming strategy promoting vectorization is to build bodies of statements whose inter-statement data reuse distance is strictly greater than the SIMD vector size, and whose alignment of accessed data complies with the processor requirements. A convenient adjustment of the offset values allows easy compliance with these requirements. In the following, we illustrate this programming strategy using the xfor implementation of `jacobi-1d`.

As done in the xfor code, Pluto fuses appropriately both original loops into one unique loop where the second statement is shifted by one iteration in order to respect the Write-After-Read dependence regarding accesses to array A (see Figure 5). However, such a program construction does not promote vectorization since the CPU cannot run simultaneously both statements because of the simultaneous write and read of array element `A[t1-1]`. On the other hand, the xfor structure allows to set the reuse distance precisely such that the final generated loops are in favour of automatic vectorization. For `jacobi-1d`, a reuse distance set to 9 provides the best performance.

	Pluto	XFOR	Ratios
jacobi-1d			
#CPU cycles	9,711M	9,063M	-6.67%
#L1 data loads	895M	885M	-0.03%
#L1 misses	110M	110M	-0.53%
#L2 misses	4M	4.7M	+16.78%
#L3 misses	54M	57M	+5.34%
#TLB misses	2.3M	2M	-15.51%
#branches	508M	505M	-0.48%
#branch misses	1,031K	1,174K	+13.91%
#stalled cycles	7,465M	6,844M	-8.32%
#instructions	4,891M	4,924M	+0.69%
fdtd-2d			
#CPU cycles	7,631M	5,679M	-25.58%
#L1 data loads	950M	962M	1.25%
#L1 misses	130M	114M	-12.29%
#L2 misses	5.6M	11.3M	+103.02%
#L3 misses	39M	32M	-18.81%
#TLB misses	1.8M	1.4M	-25.64%
#branches	345M	249M	-27.85%
#branch misses	755K	636K	-15.79%
#stalled cycles	5,844M	3,871M	-33.77%
#instructions	3,936M	4,427M	+12.46%
fdtd-apml			
#CPU cycles	2,969M	1,871M	-36.96%
#L1 data loads	360M	333M	-7.56%
#L1 misses	27M	30M	+10.85%
#L2 misses	971K	1,127K	+16.11%
#L3 misses	9.6M	9.2M	-3.55%
#TLB misses	710K	925K	+30.31%
#branches	97M	81M	-17%
#branch misses	476K	572K	+20.31%
#stalled cycles	2,196M	1,190M	-45.81%
#instructions	1,581M	1,448M	-8.46%

TABLE VIII. NOT VECTORIZED/VECTORIZED CODES

```
/* Pluto code: A[t1] reuse distance = 1 */
B[2] = 0.33333 * (A[1] + A[2] + A[3]);;
for (t1=3;t1<=n-2;t1++) {
 B[t1] = 0.33333 * (A[t1-1] + A[t1] + A[t1 + 1]);;
 A[t1-1] = B[t1-1]; }
A[n-2] = B[n-2];

/* XFOR code: A[j] reuse distance = 9  */
xfor(j0=2,j1=2;j0<n-1,j1<n-1;j0++,j1++;1,1;0,9) {
 0 : B[j0] = 0.33333 * (A[j0-1] + A[j0] + A[j0+1]);
 1 : A[j1] = B[j1]; }
```

Fig. 5. Pluto and xfor codes for `jacobi-1d`

IV. XFOR PROGRAMMING

Xfor facilitates loop fusion while keeping from the user the burden of writing prologues and epilogues loops, complex bounds, etc. When handling several iteration domains scanned by successive loop nests exhibiting some data reuse, these nests may be carefully fused to be scheduled more efficiently. This is achieved by overlapping accurately their respective iteration domains through shifting (offset) and dilatation (grain). The offset and grain values must yield data reuse distances among the statements which promote simultaneously short data reuse distances and vectorization, while paying attention to data dependences. The final schedule can then be described by a xfor-loop nest.

Example. The Red-Black Gauss-Seidel algorithm is composed of two phases. The first phase updates the red elements of a grid, which are every second point in the i and j directions, starting from the first point at the bottom left corner, by using their North-South-East-West (NSEW) neighbors, which are black elements (see Figure 6, left). The second phase updates the black elements from the red ones using the same stencil f. For a 2D N×N problem, the standard code is of the form

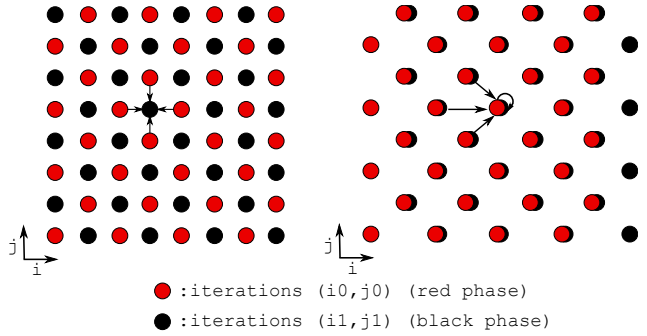

Fig. 6. Gauss-Seidel red and black domains, original (left) and black-shifted (right)

```
// Red phase
for (i=1 ; i < N-1 ; i++)
 for (j=1 ; j < N-1 ; j++)
  if ((i+j) % 2 == 0)
   u[i][j] = f(u[i][j+1], u[i][j-1],
               u[i-1][j],u[i+1][j]);
// Black phase
for (i=1 ; i < N-1 ; i++)
 for (j=1 ; j < N-1 ; j++)
  if ((i+j) % 2 == 1)
   u[i][j] = f(u[i][j+1], u[i][j-1],
               u[i-1][j],u[i+1][j]);
```

Fig. 7. Red-Black Gauss-Seidel code

shown in Figure 7 (the border elements initialization has been omitted). This code is not handled by Pluto which is unable to handle non-linear conditionals nor dilated domains.

This example obviously defines two dependent iteration domains: the red one which includes points (i, j) such that $(i+j)$ *modulo* $2 = 0$, and the black one with points such that $(i + j)$ *modulo* $2 = 1$. Each black point depends on its four NSEW red neighbors. Both domains can be scheduled such that any black point is computed as soon as all four red points from which it depends have been computed. This means that according to the lexicographic order, any black point can be computed as soon as its eastern neighbor is available, since it is the last computed point of the stencil. Hence, a shift of the black domain of one unit in direction east, *i.e.*, along the i axis, overlaps black points with their respective eastern red points (see Figure 6, right). Both red and black points define the body of the xfor-loop nest where the red statement precedes the black statement, in order to respect their dependences. The resulting xfor nest is shown in Figure 8.

Another strategy to improve data locality is to split an

```
xfor (i0 = 1, i1 = 1 ; i0 < N-1, i1 < N-1 ;
                        i0++,i1++ ; 1,1 ; 0,1)
xfor (j0 = 1, j1 = 1 ; j0 < N-1, j1 < N-1 ;
                        j0++,j1++ ; 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. 8. First Red-Black Gauss-Seidel xfor code

```
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=2, j3=1 ;
      j0 < N-1, j1 < N-1, j2 < N-1, j3 < N-1 ;
      j0+=2, j1+=2, j2+=2, j3+=2 ;
      2,2,2,2 ; 0,1,1,0) {
      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. 9. Second Red-Black Gauss-Seidel xfor code

iteration domain into several domains, each being associated with a subset of the original loop body, or a partial computation of an original arithmetic expression (similarly to the example of Figure 1), if such a decomposition is allowed regarding mathematical properties and arithmetic precision. Thus, each subset can be re-scheduled individually by overlapping the domains. This latter approach may also be useful to optimize codes containing conditionals with modulo expressions of the loop indices, as it is also illustrated with the Red-Black example in the following.

Example (continued). The xfor program of Figure 8 contains guards testing the parity of $(i+j)$. However, 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 is shown in Figure 9, where statements 0 and 1 are associated with the red domain.

Simple loop interchange and unroll-and-jam transformations combined with xfor take part of the winning programming strategy. An application example of unroll-and-jam is shown with the xfor seidel code in Figure 4, where each statement has been duplicated in their respective labeled code block. Such a transformation generally increases data reuse among the statements, and thus improves data locality. However, as shown in subsection III-D, it has to be used with moderation to avoid processor stalls. Loop interchange may be classically employed to improve intra-statement data locality by changing column-major into row-major accesses.

V. Multi-threaded parallel xfor

Beside data locality and vectorization, loop parallelization is obviously an important issue with current multicore processors. We show that xfor structures promotes better effectiveness of the parallel codes.

Since the xfor structure promotes the minimization of data reuse distances, successive iterations may be potentially strongly dependent, thus preventing loop parallelization. However, offsets may be increased at given xfor-loop levels to enlarge reuse distances and exhibit slices of independent iterations as soon as dependent memory references are performed

by domain-separated instructions. Each slice can then be parallelized and all the slices being run serially thanks to an enclosing for-loop. This approach results in a well balanced schedule among data locality and efficient parallelization.

The IBB compiler handles OpenMP pragmas for xfor-loop parallelization (#pragma omp [parallel] for). IBB copies them above every for-loop resulting from a parallelized xfor-loop in the output source code, while also preserving the convenient OpenMP clauses, as "shared" or "private". The new referential loop indices introduced by IBB are inserted inside the "private" clauses.

Example. For the Red-Black Gauss-Seidel example, as it can be observed on the right of Figure 6, for a fixed value of the i-index, all j-iterations may be performed in parallel, since no dependence occurs along the j-axis, and the dependence between body statements are not violated thanks to their preserved order. However, parallelization of the outer xfor-loop may provide better performance thanks to a larger parallelism grain and less synchronizations. Unfortunately, data dependences prevent this alternative. But it can be observed that a higher offset applied to the black domain increases dependence distances and enables the parallel execution of several successive iterations of the outer loop. More precisely, an offset incremented by $2 \times k$ allows k successive iterations of the outer xfor-loop to run in parallel. To implement this solution, an enclosing for-loop has to be inserted, which is devoted to scan iterations per groups of k iterations, while the outer xfor-loop scans the parallel iterations inside each group. This newly introduced for-loop requires some related modifications of the xfor-loop bounds and offsets, similarly to a classical strip-mining loop transformation. The resulting code is shown in Figure 10.

VI. Experiments

Although several experiments have already been presented in Section III, we show the execution times that were collected from running the best performing Pluto codes and xfor codes in some other scenarios. The Red-Black Gauss-Seidel xfor-code is compared to its original version since it is not handled by Pluto, while every other xfor-code is compared to the best Pluto-code (speed-up = (Pluto time)/(XFOR time)). Pluto and xfor versions have been compiled using GCC 4.8.1 and ICC 14.0.3 with options O3 and march=native, and their outputs have been compared to ensure correctness of the xfor codes. Execution times of the main loop kernels are given in seconds in the tables. The xfor grains are always set to 1, except for the Red-Black code whose grain is 2.

ICC was successful in vectorizing some codes while GCC was not. Typical examples are those exhibited in subsection III-E: for `jacobi-1d`, `fdtd-2d` and `fdtd-apml`, ICC is able to vectorize the Pluto codes, while GCC only handles the xfor codes. For the second set of measurements, OpenMP parallelization has been turned on in Pluto using option `--parallel`, in GCC using option `-fopenmp`, and in ICC using option `-openmp`. Codes have been run using 12 parallel threads mapped on the 6 hyperthreaded processor cores of the Xeon X5650 processor. `Seidel` does not appear in Table X because it requires skewing to afford parallelization.

Code	Pluto time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Pluto time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	N/A	1.92	**1.66** *over orig.*	N/A	1.92	**2** *over orig.*
mvt	0.71	0.18	**3.94**	0.44	0.15	**2.93**
syr2k	2.54	2.12	**1.20**	1.43	1.32	**1.08**
3mm	5.93	1.61	**3.68**	0.93	1.60	**0.58**
gauss-filter	1.14	0.94	**1.21**	0.91	0.83	**1.10**
seidel	5.28	2.56	**2.06**	4.71	3.17	**1.49**
correlation	0.15	0.10	**1.50**	0.17	0.09	**1.88**
covariance	0.15	0.12	**1.25**	0.15	0.12	**1.25**
jacobi-2d	0.71	0.74	**0.95**	0.71	0.75	**0.95**
jacobi-1d	0.66	0.44	**1.50**	0.44	0.44	**1.00**
fdtd-2d	0.61	0.42	**1.45**	0.33	0.33	**1.00**
fdtd-apml	0.91	0.50	**1.82**	0.76	0.55	**1.38**

TABLE IX. Vectorized code measurements

Code	Pluto time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Pluto time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	N/A	0.88	**1.18** *over orig.*	N/A	0.84	**1.09** *over orig.*
mvt	0.12	0.10	**1.2**	0.13	0.11	**1.18**
syr2k	0.28	0.17	**1.65**	0.25	0.16	**1.56**
3mm	1.75	*0.20*	**8.75**	0.27	*0.48*	**0.56**
gauss-filter	1.13	0.11	**10.27**	0.91	0.11	**8.27**
correlation	0.12	0.04	**3.00**	0.05	0.02	**2.50**
covariance	0.03	0.03	**1.00**	0.03	0.03	**1.00**
jacobi-2d	1.41	0.41	**3.44**	1.34	0.43	**3.12**
fdtd-2d	0.30	0.19	**1.58**	0.30	0.19	**1.58**
fdtd-apml	0.11	0.07	**1.57**	0.15	0.08	**1.88**

TABLE X. OpenMP parallel code measurements (12 threads)

```
#define k NUMBER_OF_THREADS
for(i=1 ; i < (N-1)/2*k ; i+=2*k)
#pragma omp parallel for private(i0,i1,i2,i3) \
                         private(j0,j1,j2,j3) \
                         firstprivate(i)      \
                         shared(u)
xfor (i0=i, i1=i+1, i2=i, i3=i+1 ;
      i0 < min(i+2*k,N-1), i1 < min(i+1+2*k,N-1),
      i2 < min(i+2*k,N-1), i3 < min(i+1+2*k,N-1) ;
      i0+=2, i1+=2, i2+=2, i3+=2 ;
      2,2,2,2 ; i-1,i,1+i+2*k,2+i+2*k)
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. 10. Parallelized Red-Black Gauss-Seidel xfor code

Parallelization of `jacobi-1d` does not provide any speed-up for both Pluto and xfor codes.

VII. Related work

New looping features have been proposed in PGAS languages, such as zippered iterators in Chapel [13] or sequential iteration over regions in X10 [14]. Regions in X10 can be defined from arrays which are 2-dimensional at maximum, and whose composed shapes are limited to rectangles and triangles. Translating xfor structures into PGAS languages' loop

structures would require the programmer to define domains compositions and handle related code modifications. 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 languages.

A framework utilizing the associativity and commutativity of operations in loops is presented in [15]. The xfor structure would allow users to write codes implementing explicitly and in a concise manner the proposed scatter-gather combinations for stencil computations. However, it focuses only on data reuse and vectorization, and does not address the other optimization gaps: processor stalls due to branch misses, register dependencies and instruction count.

Xfor takes its roots in the polyhedral model [1]. Many studies in the field target automatic parallelization [2]. However, fundamental complexity limits, difficult syntactic and semantic analysis, and the variety of possible optimization criteria make it difficult to automate program transformations [4]. The goal of our work is to bring sophisticated and efficient polyhedral techniques at the programming-language level. The shifting of statements made available by the xfor structure has been studied by Darte and Huard regarding its complexity in [16], where the authors differentiate between internal and external shifting and prove the NP-Completeness of the latter.

VIII. Conclusion

The highlighted performance issues confirm that program optimization is based on a careful balance between several concurrent goals. While data locality is obviously an important issue, it is not an isolated one and its improvement must be careful of the other four issues which are excessive number of branches, instruction counts, long chains of short RAW dependences and vectorization. Moreover, inter-statement data locality is as important as intra-statement data locality and should be handled accordingly.

However, inter-statement data reuse distances must be reduced carefully but not always at maximum, to still enable vectorization of close instructions accessing common data which are dependent. A convenient distance must be maintained between data that are read and written by the instructions which are vectorized simultaneously. At the same time, this distance must still stay small enough to take advantage of cache locality.

Tiling is often the *de facto* answer for improving data locality, although we have shown that better performance can be reached without tiling because of other performance issues that annihilate locality improvement. The main drawback of tiling is the code required for the additional loop levels and loop bounds which are often resulting from complex computations using functions *min*, *max*, *floor* and *ceiling*. Such a code may yield too many instructions with many branches which are potentially subject to branch misses.

All in all, we have shown that even when handling linear loops, which are often perceived as already well handled by compilers and current microprocessors, there is still a large gap to fill to reach extreme performance. Compilers must still be made conscious of more performance issues, hardware prefetchers do not compensate for bad data locality, even with linear accesses, and branch predictors are not infallible. The xfor structure is an antidote to help addressing these gaps, until the perfect compiler and microprocessor have been developed, if they ever will be in the future.

References

[1] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1581–1592. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_502

[2] 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.

[3] "PLUTO - An automatic parallelizer and locality optimizer for multicores," http://pluto-compiler.sourceforge.net.

[4] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: part ii, multidimensional time," in *Proc. of the ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, 2008, pp. 90–100. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375594

[5] E. Park, J. Cavazos, L. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *International Journal of Parallel Programming*, vol. 41, no. 5, pp. 704–750, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0241-1

[6] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *TACO*, vol. 11, no. 1, p. 2, 2014. [Online]. Available: http://doi.acm.org/10.1145/2579561

[7] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. V. Bonilla, J. Thomson, C. K. I. Williams, and M. F. P. O'Boyle, "Milepost GCC: machine learning enabled self-tuning compiler," *Int. J. of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10766-010-0161-2

[8] 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.

[9] "The Polyhedral Benchmark suite," http://sourceforge.net/projects/polybench.

[10] I. Fassi, P. Clauss, M. Kuhn, and Y. Slama, "Multifor for Multicore," in *IMPACT 2013, Third Int. Workshop on Polyhedral Compilation Techniques*, A. Grösslinger and L.-N. Pouchet, Eds. Berlin, Germany: Epubli, Jan. 2013, pp. 37–44.

[11] P. Clauss, I. Fassi, and A. Jimborean, "Software-controlled processor stalls for time and energy efficient data locality optimization," in *XIVth Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS*, 2014, pp. 199–206. [Online]. Available: http://dx.doi.org/10.1109/SAMOS.2014.6893212

[12] "perfmon2: improving performance monitoring on Linux," http://perfmon2.sourceforge.net.

[13] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-defined parallel zippered iterators in chapel," in *PGAS 2011: Fifth Conf. on Partitioned Global Address Space Programming Models*, October 2011.

[14] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification version 2.2," Mar. 2012.

[15] K. Stock, M. Kong, T. Grosser, L. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '14*, 2014, pp. 65–76. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594342

[16] A. Darte and G. Huard, "Complexity of multi-dimensional loop alignment." in *STACS*, ser. Lecture Notes in Computer Science, H. Alt and A. Ferreira, Eds., vol. 2285. Springer, 2002, pp. 179–191.