Mind The Gap!
A study of some pitfalls preventing peak performance in polyhedral compilation using a polyhedral antidote

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

ABSTRACT

The polyhedral model is a wonderful but imperfect world. While many advanced and fully automatic program analysis and optimization techniques have been developed thanks to its accuracy and expressiveness, these techniques may fail in producing efficient codes in some circumstances. Recently, this has been identified more clearly through the proposition of a new programming structure called xfor (multifor). This structure eases the manual application of optimizing transformations on loop nests for expert programmers and allows to generate executable codes that may be significantly faster than those generated automatically using well-established polyhedral strategies. In this paper, we highlight five main gaps regarding these strategies and discuss some ideas on how to bridge them.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Performance

Keywords

Loop nest optimization, program execution performance, performance issues, data locality, data dependence, processor stalls

1. INTRODUCTION

The polyhedral model is a mathematical framework which has permitted the development of very advanced analysis and optimizing transformation techniques for Static Control Parts (SCoP), some of them being implemented in high quality software libraries and compilers [15, 4, 3, 2]. Recently, its underlying principles have even been adapted to speculative and dynamic optimization of more general loops exhibiting a polyhedral behavior at runtime [8, 14]. However, the actual runtime performance of the codes generated automatically using polyhedral techniques is still an uncontrolled issue. This situation has symptomatically lead to the development of iterative and machine learning compilation frameworks [12, 9, 16, 7]. Decisions taken by automatic tools are based on heuristics that can fail in some circumstances. Moreover, these heuristics may ignore some important performance issues or miss some alternative optimization techniques.

Recently, we have proposed a new programming control structure called "xfor", allowing users to explicitly schedule statements of a loop nest by shifting and stretching each statement's iteration domain [6, 5]. It has been shown that xfor programs often reach better performance than programs optimized by fully automatic polyhedral compilers like Pluto [4]. It has also been shown that different versions of codes may perform very differently, although their memory behaviors are very similar [5]. By analyzing further the origins of such performance differences, we noticed 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*.

In this paper, we highlight and try to explain these five gaps using benchmark codes. We also give some ideas for strategies to bridge these gaps.

The paper is organized as follows. First, we recall the xfor syntax and semantics in Section 2. Next in Section 3, 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. Section 4 discusses some ideas for the consideration of these performance issues when optimizing codes. Finally, conclusions are given in Section 5.

2. THE XFOR LOOP STRUCTURE

In this section, we recall the xfor syntax and semantics initially presented in [6] and later updated in [5]. 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

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

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.

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 [5] 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 using the CLooG library [3].

Notice that the xfor structure allows users to write for example codes implementing explicitly and in a concise manner the scatter-gather combinations for stencil computations described in [13].

3. 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 3.5 regarding vectorization, while the previous one is addressed in subsection 3.3. 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 affine loop nests, the Pluto source-to-source compiler [4, 1] 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 polyhedral optimizers since data locality optimization has always been seen as a final goal. However, we show in subsection 3.4 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 3.5).

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 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 3.2.

In the following subsections, we illustrate the importance of these issues in program optimization using eleven representative codes extracted from the Polybench benchmark suite [11]. 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 [1] 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 [10]. The collected CPU events are detailed and commented in Table 1. 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 ICC, excepting for automatic vectorization which is generally better handled by ICC.

#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 1: Collected CPU events

3.1 Gap 1: Insufficient data locality optimization

Tables 2 and 3 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 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

	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 2: Slowdowns mostly due to more TLB and cache misses

stalls which are symptomatic of long latency memory operations.

In `mvt`, from the first to the second loop nest, only array A is reused. Data-reuse distances between both iteration domains are large, since array A is accessed in row-major order in the first nest, and in column-major order in the second nest, which is also an unfavourable access order regarding intra-statement data locality. Since no data dependences prevent loop transformations, interchanging loops i and j in the second nest is obviously beneficial. Finally, overlapping both resulting iteration domains optimizes inter-statement data reuse distances and provides a significant speed-up. The best Pluto version is resulting from merging both loop nests and tiling the resulting nest, however without interchanging loops of the second nest.

With `syr2k`, the xfor code is built by splitting the iteration domain into two iteration domains respectively associated with one of the two statements, and by exchanging the loops of the second domain from i-j-k to j-i-k. Thus, each couple of accesses to elements of arrays A and B is reused consecutively by both statements. Temporal reuse was also promoted by using a temporary variable instead of `C[i][j]` for the second statement. Pluto just kept intact the original code which seemingly was evaluated exhibiting a good data locality.

In `3mm`, there are three successive matrix products $E = A \times B$, $F = C \times D$ and $G = E \times F$ for which the two first products are independent and thus could be perfectly overlapped. However, since there is no data reuse between them, such a program version would induce saturated memory bandwidth or numerous cache conflicts. The third product uses both computed matrices to compute matrix G. The loop nests can obviously take advantage of a common loop interchange from order i-j-k to i-k-j. However, to promote overlapping of the third nest onto the second nest, the order of the two first loop nests is inverted such that the reuse of matrix E in the third nest can occur row by row as soon as any E-row computation has been completed. The best Pluto version is not implementing any loop fusion and do not take advantage at all of data reuse among the statements.

	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 3: Stalls mostly due to more TLB and cache misses

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

3.2 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 1). 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. The code generated by the xfor compiler IBB for `seidel` is shown in Figure 2. Consequently, Pluto's codes are more exposed to branch misses as exhibited by the performance counters (see Table 4). 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

```
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(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] = ...;
```

Figure 1: Tiled loop nest generated by Pluto for `seidel`

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.

3.3 Gap 3: Number of instructions

Both Pluto and `XFOR1` codes of Table 5 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 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.

3.4 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 3 while the offset values are shown in Table 6. Notice that these codes have a different shape than the xfor `seidel` code addressed in subsection 3.2, 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

	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 4: Slowdowns partially due to more branch mispredictions

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 5: Slowdowns mostly due to higher instruction counts

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

Figure 2: Code generated by the xfor-IBB compiler for `seidel`

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 time-consuming assembly instructions of `XFOR1`, `XFOR2` and `XFOR3` is shown in Table 7. 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 7, 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`.

However it is difficult and tedious to understand precisely from assembly codes why some codes are slower than others regarding solely this issue, even if the cause is obviously related to the instruction schedule. Additionally, since the schedule window is too narrow for the CPU to dynamically improve the situation, a similar issue occurs for humans when trying to understand the weak performance by analyzing the assembly code.

```
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 ; } }}
```

Figure 3: The xfor seidel code used for register dependence analysis

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 6: Processor stalls due to register dependences for three xfor versions of `seidel`

Figure 4 shows the source codes of two successive unrolled iterations of the innermost loop in each of the three xfor codes. A rationale for the execution-time differences can be deduced from the memory accesses related to the floating-point divisions – translated as `divsd` instructions – which are the main bottlenecks. Accesses to array elements related by some dependences with an array access made in one of the division statements have been colored with the same color.

The `XFOR1` code shows four successive short RAW dependences regarding the updates of element `A[i][j]` – in lines 11, 13, 15, 17, 20 – and one RAW dependence regarding element `A[i-1][j]` – in lines 19, 20. All these dependences are impacting significantly the second division statement in line 20, as it can be observed in Table 7 with a total CPU time

of 796ms for the second `divsd` instruction, while the first statement in line 19 is not impacted by any short distance dependence. The RAW dependences prevent the superscalar CPU to launch simultaneously several `addsd` instructions, which consequently extends the time to be spent until the availability of all the operands of statement 20. Moreover, statement 19, which is a costly division, has also to be completed before statement 20 can be launched.

The `XFOR2` code is showing very similar short RAW dependences regarding the updates of element `A[i][j]` – in lines 11, 14, 15, 17, 20 – and the same RAW dependence. But additionally, since the statement in line 13 is delayed by the update of element `A[i][j-1]` in line 10 – which is a long latency division (RAW dependence) – the division in line 19 is transitively impacted since it reads and also updates the updated element `A[i-1][j]` of line 13 (RAW dependence). It can be observed in Table 7 that both `divsd` division instructions are now showing important latencies (542ms and 526ms CPU times).

The situation is even worse with `XFOR3` which is showing a succession of dependences preventing any simultaneous superscalar execution of instructions of the loop body. Lines from 11 to 20 are strongly sequentialized due to many short RAW dependences which explain the huge latencies of both `divsd` instructions.

Since `XFOR1` is the fastest version, an even faster version should be expected by extending the distances of dependences which impact statement 20. On the other hand, such extensions may alter the data locality, or increase the number of instructions and thus result in a slower code. An offset-j of 1 assigned to statement 20 extends significantly its dependence distance regarding statement 19. This can be directly observed with this new xfor code from the huge improvement in the number of reservation station stalls (119M). However, due to the specific schedule of statement 20, a larger amount of required instructions (7,354M) makes the code slightly slower (7,442M CPU cycles).

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 6.

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 7: Total aggregated CPU time per instructions (ms) – source: Intel VTune

3.5 Gap 5: Insufficient handling of vectorization opportunities

Table 8 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 by constant memory strides. Thus, *sv* iterations are run in parallel thanks to the SIMD unit. Mainstream compilers fea-

	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 8: Not vectorized/vectorized codes

```
/**** XFOR1 ****/
/* PREVIOUS ITERATION */
01  A[i][j-1] += A[i][j];
02  A[i+1][j-1] += A[i+1][j];
03  A[i][j-1] += A[i+1][j-2];
04  A[i+1][j-1] += A[i+2][j-2];
05  A[i][j-1] += A[i+1][j-1];
06  A[i+1][j-1] += A[i+2][j-1];
07  A[i][j-1] += A[i+1][j];
08  A[i+1][j-1] += A[i+2][j];
09  A[i-1][j-1] = (A[i-1][j-1] + A[i-2][j-2] + A[i-2][j-1] + A[i-2][j] + A[i-1][j-2])/9.0;
10  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;

/* CURRENT ITERATION */
11  A[i][j] += A[i][j+1];
12  A[i+1][j] += A[i+1][j+1];
13  A[i][j] += A[i+1][j-1];
14  A[i+1][j] += A[i+2][j-1];
15  A[i][j] += A[i+1][j];
16  A[i+1][j] += A[i+2][j];
17  A[i][j] += A[i+1][j+1];
18  A[i+1][j] += A[i+2][j+1];
19  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;
20  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;
```

```
/**** XFOR2 ****/
/* PREVIOUS ITERATION */
01  A[i][j-1] += A[i][i];
02  A[i+1][j-1] += A[i+1][j];
03  A[i-1][j-1] += A[i][j-2];
04  A[i][j-1] += A[i+1][j-2];
05  A[i][j-1] += A[i+1][j-1];
06  A[i+1][j-1] += A[i+2][j-1];
07   A[i][j-1] += A[i+1][j];
08  A[i+1][j-1] += A[i+2][j];
09  A[i-1][j-1] = (A[i-1][j-1] + A[i-2][j-2] + A[i-2][j-1] + A[i-2][j] + A[i-1][j-2])/9.0;
10  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; */

/* CURRENT ITERATION */
11  A[i][j] += A[i][j+1];
12  A[i+1][j] += A[i+1][j+1]; }
13  A[i-1][j] += A[i][j-1];
14  A[i][j] += A[i+1][j-1];
15  A[i][j] += A[i+1][j];
16  A[i+1][j] += A[i+2][j]; }
17  A[i][j] += A[i+1][j+1];
18  A[i+1][j] += A[i+2][j+1]; }
19  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;
20  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;
```

```
/**** XFOR3 ****/
/* PREVIOUS ITERATION */
01  A[i][j-2] += A[i][j-1];
02  A[i+1][j-2] += A[i+1][j-1];
03  A[i-1][j-2] += A[i][j-3];
04  A[i][j-2] += A[i+1][j-3];
05  A[i-1][j-2] += A[i][j-2];
06  A[i][j-2] += A[i+1][j-2];
07  A[i-1][j-3] += A[i][i-2];
08  A[i][j-3] += A[i+1][j-2];
09  A[i-1][j-3] = (A[i-1][j-3] + A[i-2][j-4] + A[i-2][j-3] + A[i-2][j-2] + A[i-1][j-4])/9.0;
10  A[i][j-3] = (A[i][j-3] + A[i-1][j-4] + A[i-1][j-3] + A[i-1][j-2] + A[i][j-4])/9.0; */

/* CURRENT ITERATION */
11  A[i][j-1] += A[i][j];
12  A[i+1][j-1] += A[i+1][j];
13  A[i-1][j-1] += A[i][j-2];
14  A[i][j-1] += A[i+1][j-2];
15  A[i-1][j-1] += A[i][j-1];
16  A[i][j-1] += A[i+1][j-1];
17  A[i-1][j-2] += A[i][j-1];
18  A[i][j-2] += A[i+1][j-1]; }
19  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;
20  A[i][j-2] = (A[i][j-2] + A[i-1][j-3] + A[i-1][j-2] + A[i-1][j-1] + A[i][j-3])/9.0;
```

Figure 4: Dependent memory accesses among two unrolled iterations represented in a common referential for XFOR1, XFOR2 and XFOR3

```
/* 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]; }
```

Figure 5: Pluto and xfor codes for `jacobi-1d`

turing 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. 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.

4. BRIDGING THE GAPS

One legitimate rhetoric of the polyhedral community is that source codes should be written by programmers in a form that is the simplest for the compiler, such that codes can be analyzed as precisely as possible to apply efficient optimizing transformations. In the same way, a similar rhetoric should be claimed about the shape of the codes generated by compilers for micro-processors.

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.

Regarding the latter aspect, the grain of reasoning should be the memory access, rather than the statement which may involve several memory accesses. According to this idea, it has been shown with some xfor codes that splitting a statement into several statements, where most of them involve one memory read and one memory write, enables more accurate scheduling also addressing inter-statement data reuse (e.g. `seidel`). The same has been highlighted in [13].

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 main-

tained 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.

A fully automatic and static strategy covering simultaneously and ideally all these issues seems difficult to define. Nevertheless, they should at least be considered since they are mostly ignored in the currently adopted optimization approaches. One unfortunate consequence is that proposals of new optimization techniques may even worsen the situation regarding these ignored issues. A collaborative static-dynamic approach, merging compile-time and runtime mechanisms, may constitute an effective response.

All in all, we have shown that even when handling polyhedral 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 a polyhedral antidote to help addressing these gaps, until the perfect compiler and microprocessor have been developed, if they ever will be in the future.

5. CONCLUSION

Apart from supplying advanced analysis and optimization techniques for affine loop nests, the polyhedral model has played the educational role of improving the understanding of some main issues related to program performance and code optimization. In the same way, beside being a new programming structure allowing users to write very fast codes, xfor helps in highlighting and overcoming important issues. Data locality optimization has played the role of the tree that hides the forest, while xfor allows to go behind this tree. The post-data-locality era of polyhedral optimization has started. *Mind the gap!* [1].

Acknowledgments

I would like to thank Imen Fassi, Alexandra Jimborean, Louis-Noël Pouchet and Aravind Sukumaran-Rajam for their assistance.

6. REFERENCES

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

[2] PolyLib - A library of polyhedral functions. `http://icps.u-strasbg.fr/polylib`.

[3] 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*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

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

[5] 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*, pages 199–206, 2014.

[6] I. Fassi, P. Clauss, M. Kuhn, and Y. Slama. Multifor for Multicore. In A. Grösslinger and L.-N. Pouchet, editors, *IMPACT 2013, Third Int. Workshop on Polyhedral Compilation Techniques*, pages 37–44, Berlin, Germany, Jan. 2013. Epubli.

[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*, 39(3):296–327, 2011.

[8] A. Jimborean, P. Clauss, J. Dollinger, V. Loechner, and J. M. M. Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[9] 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*, 41(5):704–750, 2013.

[10] perfmon2: improving performance monitoring on Linux. `http://perfmon2.sourceforge.net`.

[11] The Polyhedral Benchmark suite. `http://sourceforge.net/projects/polybench`.

[12] 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*, pages 90–100, 2008.

[13] 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*, pages 65–76, 2014.

[14] A. Sukumaran-Rajam, J. M. M. Caamaño, W. Wolff, A. Jimborean, and P. Clauss. Speculative program parallelization with scalable and decentralized runtime verification. In *Runtime Verification - 5th Int. Conf., RV 2014, Toronto, ON, Canada, LNCS 8734*, pages 124–139, 2014.

[15] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *Proc. of the Third Int. Conf. on Math. Software*, LNCS 6327, pages 299–302, 2010.

[16] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *TACO*, 11(1):2, 2014.

[1] Source: London subway