Improved Loop Tiling Based on the Removal of Spurious False Dependences

RIYADH BAGHDADI, ALBERT COHEN, SVEN VERDOOLAEGE,
and KONRAD TRIFUNOVIĆ, École Normale Supérieure and INRIA

To preserve the validity of loop nest transformations and parallelization, data dependences need to be analyzed. Memory dependences come in two varieties: true dependences or false dependences. While true dependences must be satisfied in order to preserve the correct order of computations, false dependences are induced by the reuse of a single memory location to store multiple values. False dependences reduce the degrees of freedom for loop transformations. In particular, loop tiling is severely limited in the presence of these dependences. While array expansion removes all false dependences, the overhead on memory and the detrimental impact on register-level reuse can be catastrophic.

We propose and evaluate a compilation technique to safely ignore a large number of false dependences in order to enable loop nest tiling in the polyhedral model. It is based on the precise characterization of interferences between live range intervals, and it does not incur any scalar or array expansion. Our algorithms have been implemented in the Pluto polyhedral compiler, and evaluated on the PolyBench suite.

Categories and Subject Descriptors: D.3.4 [**Programming languages**]: Processor—Compilers, Optimization

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Tiling, false dependences, memory-based dependences, expansion, compiler

ACM Reference Format:
Baghdadi, R., Cohen, A., Verdoolaege, S., and Trifunović, K. 2013. Improved loop tiling based on the removal of spurious false dependences. ACM Trans. Architec. Code Optim. 9, 4, Article 52 (January 2013), 26 pages.
DOI = 10.1145/2400682.2400711 http://doi.acm.org/10.1145/2400682.2400711

1. INTRODUCTION AND RELATED WORK

To harness the computing resources of multiple cores with complex memory hierarchies, the need for powerful compiler optimizations and especially loop nest transformations is high. Loop transformations operate on the fine-grain schedule of the statement instances (iterations) executed in a loop nest. To guarantee that loop nest transformations are correct, the compiler needs to preserve data dependences among different statement instances. Two statements are in dependence if they access the same memory location and at least one of them is a write. Two types of data dependences exist, true (a.k.a. data-flow) and false (a.k.a. memory-based, output-, and anti-) dependences [Kennedy and Allen 2002]. False dependences are induced by the reuse of temporary variables across statement instances. These false dependences eliminate degrees of freedom that may be essential to the expression of effective loop

This work is supported by the European Commission through the FP7 project CARP id. 287767.
Authors' address: R. Baghdadi (corresponding author), A. Cohen, S. Verdoolaege, and K. Trifunović, École Normale Supérieure and INRIA; email: baghdadi.mr@gmail.com.

DOI 10.1145/2400682.2400711 http://doi.acm.org/10.1145/2400682.2400711

nest transformations. They prevent the compiler from performing loop nest tiling in particular.

Scalar and array expansion removes false dependences at the expense of memory footprint [Feautrier 1988; Kennedy and Allen 2002]. Expansion creates a private copy of scalars for each loop iteration by transforming scalars into arrays and transforming arrays into higher-dimensional arrays. Although this technique eliminates false dependences and enables loop tiling, it incurs high footprint on memory. Besides memory footprint, scalar and array expansion also degrades temporal locality [Thies et al. 2001]. Scalar expansion is particularly harmful as it converts register arguments into memory operations [Callahan et al. 1990].

Note that the terms privatization and expansion are sometimes used by the community interchangeably to refer to expansion. In this article we make a clear distinction between the two terms. In privatization, a private copy of the variables is created for each thread. In expansion, a private copy is created for each statement instance (syntactic occurrence and iteration), which induces the declaration of new data structures, the generation of more complex array indexing schemes, and data-flow restoration code [Feautrier 1988]. This usage is consistent with the use of privatization and expansion in Midkiff [2012] and Maydan et al. [1993].

In particular, privatization creates, for each thread cooperating on the execution of the loop, private copies of scalars (or arrays) [Gupta 1997; Tu and Padua 1994; Li 1992]. The number of private copies is equal to the number of parallel threads. Although privatization is required to enable parallelism, it is not sufficient to enable tiling, because privatization only creates private copies of scalars and arrays for each thread, eliminating false dependences crossing the boundary of data-parallel blocks of iterations. Classical tiling techniques require the elimination of all (false) negative-distance dependences [Irigoin and Triolet 1988; Griebl 2004; Kennedy and Allen 2002; Bondhugula et al. 2008], which is the domain of scalar or array expansion. We are looking for a characterization of false dependences that are compatible with loop tiling. We will achieve this through a dedicated criterion, adapted from both the traditional tiling correctness and privatization correctness criterion; false dependences satisfying this criterion will not need to be eliminated through expansion to enable tiling.

A family of array contraction techniques attempts to reduce the memory footprint without constraining loop nest transformations [Lefebvre and Feautrier 1998; Quilleré and Rajopadhye 2000; Darte and Huard 2005]: the compiler performs a maximal expansion, looks for transformations, and then attempts to contract the arrays. By performing a maximal expansion, false dependences are eliminated. Yet contraction is not always possible when unrestricted loop transformations have been applied, as the set of simultaneously live values may effectively require high-dimensional arrays to store them. Loop distribution is an example of a widely applied loop nest transformation that prevents array contraction and thus disables the effectiveness of this technique as we show in an example in Section 8.

Several alternative approaches have been proposed to constrain the expansion a priori. Maximal static expansion (MSE) restricts the elimination of dependences to the situations where the data flow can be captured accurately at compilation time [Cohen and Lefebvre 1998]. It is important when generalizing array-dependence analyses and loop transformations to dynamic control flow, and it can be combined with array contraction [Cohen 1999]. A priori constraints on memory footprint can also be enforced, up to linear volume approximations [Thies et al. 2001], and more generally, trade-offs between parallelism and storage allocation can be explored. These approaches are particularly interesting when adapting loop nests for execution on hardware accelerators and embedded processors with local memories.

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
S1:  C[i][j] = C[i][j] * beta;
      for (k = 0; k < nk; ++k)
S2:     C[i][j] += alpha * A[i][k] * B[k][j];
  }
```

Fig. 1. *Gemm* kernel.

In this article, we propose and evaluate a technique allowing compilers to tile and parallelize loop nests aggressively, even in the presence of false dependences that would violate existing validity criteria for loop tiling. The technique enables the compiler to decide which false dependences can be safely ignored, in the context of a given combination of enabling loop transformations and loop tiling. The proposed technique does not incur any costs of scalar or array expansion.

Section 2 presents the possible sources of false dependences. Section 3 shows that existing tiling algorithms are too restrictive and miss safe tiling opportunities. Section 4 introduces a new tiling test that avoids this problem and allows the tiling of kernels with false dependences. Section 5 proves that, by ignoring false dependences, tiling before 3AC transformation is equivalent to tiling after 3AC transformation. Sections 6, 7, and 8 show an overview of an implementation in Pluto, the benchmarks used to evaluate the proposed technique, and experimental results.

2. SOURCES OF FALSE DEPENDENCES

One common source of false dependences is found in temporary variables introduced by programmers in the body of a loop. A second source of false dependences is the compiler itself. Even in source programs that do not initially contain scalar variables, compiler passes may introduce false dependences when upstream transformations introduce scalar variables. This is a practical compiler construction issue, and a high priority for the effectiveness of loop optimization frameworks implemented in production compilers [Trifunovic et al. 2011]. Among upstream compiler passes generating false dependences, we will concentrate on the most critical ones, as identified by ongoing development on optimization frameworks such as the GRAPHITE polyhedral loop nest optimization framework in GCC [Trifunovic et al. 2010; Pop et al. 2006].

—Transformation to Three-Address Code (3AC). GRAPHITE is an example of a loop optimization framework operating on low-level three-address instructions. It is affected by the false dependences that result from a conversion to three-address code.
—Partial Redundancy Elimination (PRE) [Knoop et al. 1994; Muchnick 1997] applied to array operations removes invariant loads and stores, promoting array accesses into scalars. Figures 1 and 2 show an example of this optimization. Variants and extensions such as constant subexpression elimination or value numbering also introduce false dependences [Muchnick 1997].
—Loop-invariant code motion is a common compiler optimization that moves loop-invariant code outside the loop body eliminating redundant calculations. It can be seen as a special case of PRE where code motion degrades the perfect nesting of loop nests, carrying along dependences that used to be enclosed in inner loops towards outer loops. An example is shown in Figures 3 and 4, where tiling of the inner two loops is inhibited by the extra dependences on `t`. Notice that even the outer two loops may not be tilable with classical tiling algorithms, due to the false dependences on `t`.

In the case of GCC or LLVM, providing loop nest optimizations directly on three-address code provides many benefits. Since the representation is also in Static

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
S1:  t = C[i][j] * beta;
     for (k = 0; k < nk; k++)
S2:    t += alpha * A[i][k] * B[k][j];
S3:  C[i][j] = t;
    }
```

Fig. 2. *Gemm* kernel after applying PRE.

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++)
    for (k = 0; k < nk; k++)
      A[i][j][k] = B[i][j] * C[i][j] * D[i][j];
```

Fig. 3. Three nested loops with loop-invariant code.

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
    t = B[i][j] * C[i][j] * D[i][j];
    for (k = 0; k < nk; k++)
      A[i][j][k] = t;
  }
```

Fig. 4. Three nested loops after loop-invariant code motion.

Single Assignment form (SSA), it supports a rich set of highly efficient optimization algorithms, scaling to very large and complex programs. Loop transformations on three-address code enables a tight integration of loop optimization techniques with downstream compilation passes, including an automatic vectorization, parallelization, and memory optimizations. The ideal case for an optimization framework is to operate on a representation that is low enough so that this optimization framework benefits from all the passes operating on three-address code, and at the same time the optimization framework should be able to ignore spurious false dependences generated by these passes. Note that the SSA representation removes some false dependences due to internal reuse of scalars, but does not remove loop-carried dependences.

Loop tiling, and affine transformations enabling loop tiling (e.g., loop distribution, shifting, or skewing) are of utmost importance to coarsen the grain of parallelism and to exploit temporal locality [Irigoin and Triolet 1988; Bondhugula et al. 2008]. Unfortunately, these techniques are also highly sensitive to the presence of false dependences. Although it is possible to perform PRE and loop-invariant code motion after tiling in some compiler frameworks such as LLVM (yet this is not possible in GCC), transformation to 3AC must happen before any loop nest optimization in order to benefit from all the passes operating on three-address code, and this limits the chances to apply tiling.

Finally, enabling advanced loop transformations and parallelization methods on 3AC automatically extends their applicability to "native 3AC languages" and environments, such as Java bytecode or GPU shader and general-purpose accelerator languages such as PTX (CUDA, from NVIDIA), or FASIL (OpenCL, from AMD). Such techniques may contribute to offering some level of performance portability to GPGPU programming; they are commonly seen in source-to-source compilers such as PGI Accelerator [Portland Group 2010] or CAPS HMPP [HMPP], but would see a much wider impact when integrated in the just-in-time compilers of the device drivers themselves [Cohen and Rohou 2010].

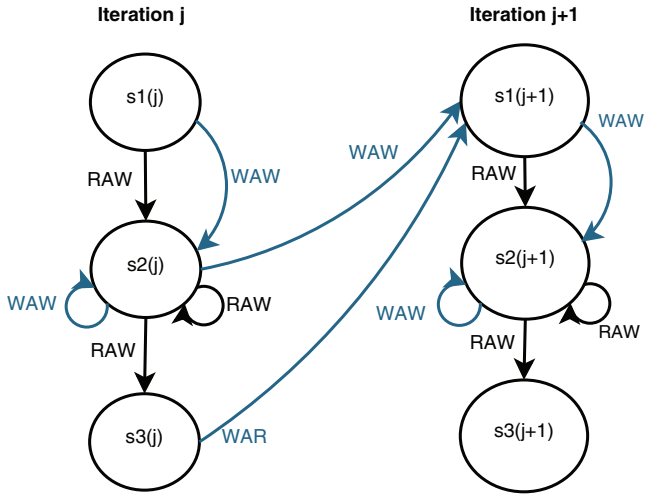

Fig. 5. Dependences in *gemm* kernel.

```
#define B 32
for (t1=0; t1<=floor((ni−1)/B); t1++)
  for (t2=0; t2<=floor((nj−1)/B); t2++)
    for (t3=B*t1; t3<=min(ni−1,B*t1+B−1); t3++)
      for (t4=B*t2; t4<=min(nj−1,B*t2+B−1); t4++)
      {
   S1:  t = C[t3][t4] * beta;
        for (t6=0; t6<nk; t6++)
   S2:    t = t + alpha * A[t3][t6] * B[t6][t4];
   S3:  C[t3][t4] = t;
      }
```

Fig. 6. Tiled version of the *gemm* kernel.

Our goal is to propose and evaluate a technique allowing compilers to escape this loophole, providing a more relaxed criterion for loop tiling in the presence of false dependences.

3. MOTIVATING EXAMPLE

Figure 5 shows data dependences on the scalar `t` for the *gemm* kernel presented in Figure 2. It contains both true (read-after-write) dependences and false (write-after-write and write-after-read) dependences. The figure shows a simplified version of the dependences between statements in two iterations `j` and `j+1`. Many false dependences stem from the fact that the same temporary scalar t is overwritten in each iteration of the loop. These dependences enforce a sequential execution.

The usual criterion to enable tiling is that none of the loops involved should carry a dependence that has a negative distance. On the example, this *forward* dependence condition prohibits the application of tiling, although the transformation is clearly valid as we show in Figure 6.

Our goal is to enable the compiler to perform tiling on codes that contain false dependences, by ignoring false dependences when this is possible, and not by eliminating them through array and scalar expansion.

```
 for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
S1:  t = A[i][j] * y_1[j];
S2:  x1[i] = x1[i] + t;
   }
```

Fig. 7. One loop from the *mvt* kernel.

In the next section, we introduce the concept of live range noninterference. We use this concept later to justify the correctness of the proposed technique and to decide when is it safe to ignore false dependences.

4. LIVE RANGE NONINTERFERENCE

In this section we present the concept of *live range noninterference*, which is then used to establish a relaxed validity criterion for tiling.

4.1. Live Ranges

We borrow some terminology from the register allocation literature where live ranges are widely studied [Chaitin et al. 1981; Hack et al. 2006; Bouchez et al. 2006], but we extend the definition of *live ranges* to capture dynamic statement instances instead of the static statements in the program. In a given sequential program, a value is said to be *live* in the range of statement instances between its definition instance and its last use instance. Since tiling may change the order of statement instances, we conservatively consider live ranges between a definition and *any* use, which includes the last use under any reordering. The definition instance is called the *source* of the live range, marking its beginning, and the use is called the *sink*, marking the end of the live range.

$(S_k(I), S_{k'}(I'))$ defines an individual *live range* beginning with an instance of the write statement S_k and ending with an instance of the read statement $S_{k'}$. I and I' are iteration vectors identifying the specific instances of the two statements. For example, the scalar `t` of the *mvt* kernel shown in Figure 7 induces several live ranges, including

$$(S_1(0, 0), S_2(0, 0)).$$

This live range begins in the iteration $i = 0$, $j = 0$ and terminates in the same iteration.

Since the execution of a given program generally gives rise to a statically nondeterminable number of live range instances, we do not want to store them individually, but rather in "classes" of live ranges that share some properties. For example, the live range above is an instance of the live range class

$$(S_1(i, j), S_2(i, j)) \quad \text{s.t. } 0 \leq i < n,\ 0 \leq j < n.$$

That is, for each iteration of i and j, there is a live range that begins with the write statement S_1 and terminates with the read statement S_2. To be able to describe such classes using affine constraints, we are interested in loop nests with affine bounds and conditional expressions (static control program parts). As live ranges are special cases of dependences, we will use the same notation for dependences in general.

4.2. Construction of Live Ranges

Live ranges form a subset of the read-after-write dependences. We construct these using the array data-flow analysis described in Feautrier [1991] and implemented in the `isl` library [Verdoolaege 2010] using parametric integer linear programming. Array data-flow analysis answers the following question: given a value v that is read from a memory cell m at a statement instance r, compute the statement instance w that is the source for the value v. The result is a (possibly nonconvex) affine relation mapping read

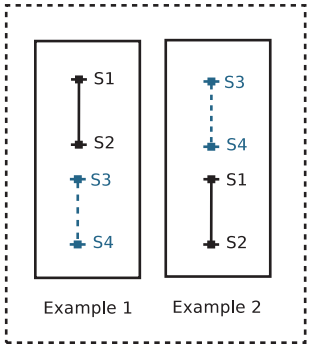

Fig. 8. Examples where the two live ranges (S_1, S_2) and (S_3, S_4) do not interfere (correct schedules).

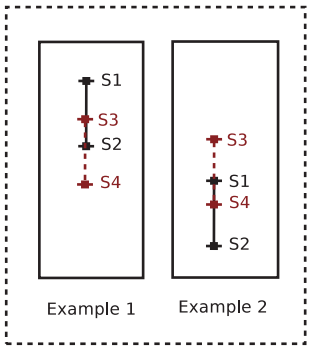

Fig. 9. Examples where the two live ranges (S_1, S_2) and (S_3, S_4) interfere (incorrect schedules).

accesses to their source. The live ranges are then computed by reversing this relation, mapping sources to all their reads.

This systematic construction needs to be completed with the special treatment of live-in and live-out array elements. Array data-flow analysis provides the necessary information for live-in ranges, but inter-procedural analysis of array regions accessed outside the scope of the static control part is needed for live-out properties. This is an orthogonal problem, and for the moment we conservatively assume that the program is manually annotated with live-out arrays (the live-in and live-out scalars are explicit from the SSA form of the loop nest).

4.3. Live Range Noninterference

Any loop transformation is correct if it respects data-flow dependences and does not lead to live range interference (no two live ranges overlap) [Vasilache 2007; Trifunovic et al. 2010, 2011]. If we want to guarantee the noninterference of two live ranges, we have to make sure that the first live range is scheduled either before or after the second live range. Figure 8 shows two examples where pairs of live ranges do not interfere. Figure 9 shows two examples where these pairs of live ranges do interfere. We will now take a closer look at the conditions for preserving noninterference across affine loop transformations.

4.4. Live Range Noninterference and Tiling

Tiling is possible when a *band* of consecutively nested loops is fully permutable. Using state-of-the-art tiling algorithms, the compiler looks for a band with *forward*

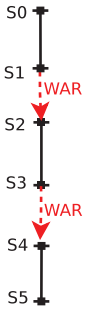

Fig. 10. An example of false dependences (dashed lines) adjacent to (S_0, S_1), (S_2, S_3) and (S_4, S_5).

dependences only, i.e., dependences with nonnegative distance for every nesting level within the band [Wolf and Lam 1991; Bondhugula et al. 2008]. This can be done by calculating dependence directions for all dependences and for each loop level. A tilable loop band is composed of consecutive loop levels with no negative loop dependence. When a loop band is identified, dependences that are strongly satisfied by this band are dropped before starting the construction of a new inner loop band (a dependence is strongly satisfied if the sink of the dependence is scheduled strictly after the source of the dependence in one of the loop levels that are a part of that band). Loop levels that belong to the identified loop bands are permutable: they can be freely permuted among themselves (a.k.a. loop interchange), which enables tiling.

The main contribution of our article is a relaxation of this permutability criterion, based on a careful examination of which false dependences can be ignored. In particular, our criterion ignores antidependences (write-after-read dependences) that are adjacent to iteration private live ranges. We say that a live range is *iteration private at level k* if it begins and ends in the same iteration of a loop at nesting level k. We say that an antidependence and a live range are *adjacent* if the source of one of them is equal to the sink of the other and if the antidependence and live range derive from the same memory access in the shared statement iteration. For example, in Figure 10, the dependence (S_1, S_2) is adjacent to the live ranges (S_0, S_1) and (S_2, S_3) but is not adjacent to (S_4, S_5). The dependence (S_3, S_4) is adjacent to (S_2, S_3) and to (S_4, S_5) but is not adjacent to (S_0, S_1).

Relaxed permutability criterion. A band of consecutively nested loops is fully permutable if and only if for every dependence one (or several) of the following conditions hold(s):

(1) the dependence is forward for each level within the band;
(2) it is an output dependence between statement instances that only define values that are local to the region of code being analyzed, in the sense that the values defined *are* used *inside* the region and are *not* used *after* the region;
(3) it is an antidependence, and it is adjacent only to live ranges that are iteration private within the band.

The first condition is inherited from the classical tiling criterion. The writes involved in the output dependences eliminated by the second condition are also involved in live ranges and are therefore guaranteed not to interfere with each other because of the third condition, as explained shortly. Moreover, the output dependences that are needed to ensure that values assigned by the last write to a given location (live-out values) will not be overwritten are *not* eliminated by the second condition. The third condition stems from the following observations.

(1) Any affine transformation—including tiling—is valid if it preserves data-flow dependences and if it does not introduce live range interferences.
(2) Tiling is composed of two basic transformations: loop strip-mining and loop interchange.
   —Strip-mining is always valid because it does not change the execution order of statement instances and thus it can be applied unconditionally.
   —Loop interchange changes the order of iterations, hence may introduce live range interferences. But a closer look shows that it never changes the order of statement instances within one iteration of the loop body.
(3) If live ranges are private to one iteration of a given loop (i.e., live ranges begin and terminate in the same iteration of that loop), changing the order of iterations of the present loop or any outer loop preserves the noninterference of live ranges.
(4) We can ignore an anti-dependence only when it is adjacent to iteration-private live ranges (i.e., not adjacent to any noniteration-private live range). This is correct for the following reason: an antidependence δ between two live ranges (S_1, S_2) and (S_3, S_4) is used to guarantee the noninterference of the two live ranges during loop nest transformations. If the two live ranges (S_1, S_2) and (S_3, S_4) are iteration private, then they will not interfere when tiling is applied, regardless of the presence of the antidependence δ. In other words, δ is useless in this case. And thus ignoring this antidependence during tiling is safe as long as all statements in this live range are subject to the same affine transformation. If one of the live ranges is noniteration private, then there is no guarantee that they will not interfere during tiling, and thus, conservatively, we do not ignore the antidependence between the two live ranges in this case.

Comparison with the criterion for privatization. Reformulated in our terminology, a loop can be privatized if all live ranges are iteration private within that loop [Maydan et al. 1993]. This condition is the same as the third condition of our relaxed permutability criterion, except that we only impose this condition on live ranges that are adjacent to backward antidependences. Our relaxed permutability criterion can therefore be considered as a hybrid of the classical permutability criterion and the privatization criterion. In particular, our criterion will allow tiling if either of these two criteria would allow tiling or privatization, but also if some dependences are covered by one criterion and the other dependences are covered by the other criterion.

4.5. Illustrative Examples

Let us illustrate the preceding observations on a few additional examples.

4.5.1. The mvt *and* gemm *Kernels.* We consider again the *mvt* kernel of Figure 7. The false dependences created by the `t` scalar on the `J` loop inhibit the compiler from applying tiling, although tiling for *mvt* is valid.

Figure 11 represents the live ranges of `t` and `x[i]` for a subset of the iterations. The figure shows the statements that are executed in each iteration (S_1 and S_2) and the live ranges generated by these statements. The original execution order is as follows: live range Ⓐ of `t` is executed first (this is the live range between S_1 and S_2 in the iteration $i = 0$, $j = 0$), then live range Ⓑ is executed (this is the live range for iteration $i = 0$, $j = 1$), then Ⓒ, Ⓓ, Ⓔ, Ⓕ, Ⓖ, etc.

We notice that each live range of `t` is private to one iteration. Each of these live ranges starts and terminates in the same iteration (no live range spans more than one iteration). The order of execution after applying a 2×2 tiling is as follows: (Ⓐ, Ⓑ, Ⓔ Ⓕ), (Ⓒ, Ⓓ, Ⓖ, Ⓗ). Tiling changes the order of execution of live ranges, but it does not break any live range of `t`.

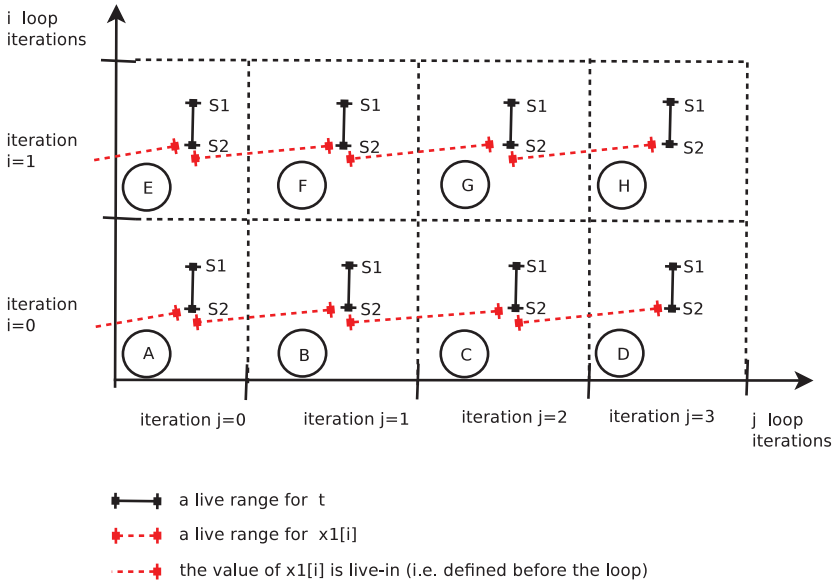

Fig. 11. Live ranges for `t` and `x1[i]` in *mvt*.

The false dependences generated by the array access `x1[i]` do not inhibit loop tiling as all of these false dependences are forward on both levels. The live ranges of `x1[i]` satisfy the first condition of the relaxed permutability criterion (they are all forward) and therefore also do not inhibit tiling.

Although tiling is valid in the case of *mvt*, the compiler fails to apply tiling because the classical tiling algorithm is too restrictive on `t`. Note that privatization of the inner loop is also not allowed. The problem for privatization is that the live ranges of `x1[i]` are not iteration private. A similar reasoning applies to tiling of the *gemm* kernel shown on Figure 2: tiling is valid because the scalar `t` is private to the `J` iteration.

4.5.2. An Example where Tiling Is Not Valid. Consider the example in Figure 12. Similarly to the previous examples, the false dependences created by the scalar `t` on the j loop prevent the compiler from applying loop tiling. But is it correct to ignore this false dependence?

Figure 13 shows the live ranges of `t`. The original execution order is: Ⓐ, Ⓑ, Ⓒ, Ⓓ, Ⓔ, Ⓕ, Ⓖ, Ⓗ. After a 2×2 tiling, the new execution order is: Ⓐ, Ⓑ, Ⓔ, Ⓕ, etc. This means that the value written in Ⓑ will be overwritten by the write statement in Ⓔ which introduces interference. This tiling breaks the live range $(S_2(I), S_2(I'))$.

Tiling in this case is not valid because the live ranges are not private to the J iteration; we should not ignore false dependences on `t`.

4.5.3. The Importance of Adjacency. Note that it is not safe to ignore an antidependence that is adjacent to an iteration-private live range but that is also adjacent to a noniteration-private live range. Tiling is not valid in this case.

4.6. Parallelism

Besides tiling, we are also interested in the extraction of data parallelism. In order to be data parallelized, a loop carrying false dependences needs the variables inducing these dependences to be privatized.

```
for (i = 0; i <= 1; i++)
  for (j = 0; j <= 3; j++) {
    if (j==0)
 S1:  t = 0;
    if (j>0 && j<=2)
 S2:  t = t + 1;
    if (j==3)
 S3:  A[i] = t;
  }
```

Fig. 12. Example where tiling is not possible.

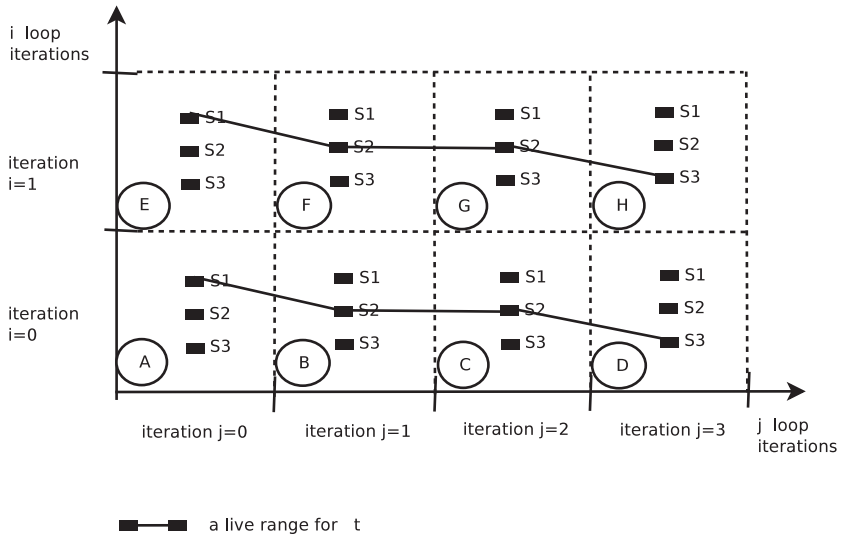

Fig. 13. Live ranges for the example in Figure 12.

For example, scalar variables that carry interfering ranges that live and die within a single iteration of a parallel loop may be safely ignored when detecting parallelism, but they must be marked as thread private in OpenMP. In principle, this reasoning also applies to array variables, but the OpenMP private clause does not support the declaration of thread private array elements, therefore we currently only ignore false dependences involving scalar variables when expressing parallelism.

The privatization of variables for parallelism is much less intrusive than actual scalar or array expansion. The former only makes variables thread private, not impacting address computation and code generation inside the loop body, while the latter involves generating new array data declarations, rewriting array subscripts, and recovering the data flow across renamed privatized structures [Feautrier 1991]. In addition, making scalar variable thread private does *not* result in the conversion of register operands into memory accesses, unlike the expansion of a scalar variable along an enclosing loop.

In practice, support for privatization is implemented as a slight extension to the state-of-the-art violated dependence analysis [Vasilache et al. 2006; Trifunovic et al. 2011].

Figure 14 compares the number of private copies that have to be created for a given scalar (or array) in order to enable parallelization, to enable tiling, or to enable both. N is the number of loop iterations and T is the number of parallel threads that execute the loop. In general T is significantly smaller than N.

	number of private scalar and arrays	
	when expansion is applied	when false dependences are ignored
parallelization only	T	T
tiling only	N	0
tiling + parallelization	N	T

Fig. 14. Comparing the number of private copies of scalars and arrays created to enable parallelization, tiling, or both.

Note that the proposed technique, unlike expansion, enables tiling without any overhead on memory. Expansion has the advantage of enabling tiling in more cases, but has the drawback of a high memory overhead. We compare between the performance of the two methods in the Section 8.

Interestingly, the transformation of source code to three-address code does not impact parallelism detection as it only induces intra-block live ranges. Section 5 proves a similar result about the impact of our technique on the applicability of loop tiling.

5. ON THE TILING POTENTIAL OF 3AC

This section shows that, by ignoring selected false dependences, the transformation of source code to 3AC form does not harm loop tiling. It was obviously a strong intuition when starting the design and implementation of the first low-level polyhedral optimization framework, GRAPHITE, in 2006 [Pop et al. 2006].

In other words, we show that by ignoring false dependences, whether tiling is applied before or after three-address lowering, we always get the same result, which is not the case with classical tiling algorithms.

Consider a 3AC statement S_1.

```
for (i1=...; i1<n1; i1++) {
  for (i2=...; i2<n2; i2++) {
    ...
    for (im=...; im<nm; im++) {
      ...
 S1:  x = expr_1 + expr_2 + expr_3;
      ...
    }
  }
}
```

Let Δ be the set of all dependences induced by S_1. After transforming this statement to 3AC, we get two new statements: S_2 and S_3.

```
for (i1=...; i1<n1; i1++) {
  for (i2=...; i2<n2; i2++) {
    ...
    for (im=...; im<nm; im++) {
      ...
 S2:  x1 = expr_2 + expr_3;
 S3:  x = expr_1 + x1;
      ...
    }
  }
}
```

Variable x1 introduced by the three-address transformation is a new variable and thus it does not have any dependence with the other variables of the program (the variables that were already in the program before three-address transformation), but it induces three new dependences between the two newly introduced statements. These dependences are as follows:

—a set of flow dependences: $\delta_{S_2 \to S_3}$;
—a set of write-after-write dependences: $\delta_{S_2 \to S_2}$;
—a set of write-after-read dependences: $\delta_{S_3 \to S_2}$.

Before three-address transformation the set of dependences was Δ. After three-address transformation the set of dependences is $\Delta \cup \delta_{S_2 \to S_3} \cup \delta_{S_2 \to S_2} \cup \delta_{S_3 \to S_2}$.

The set of flow dependences $\delta_{S_2 \to S_3}$ constitutes live ranges that are iteration private (they begin and terminate in the same iteration) and thus, by applying the technique proposed in this article, we can ignore the dependences $\delta_{S_2 \to S_2}$ and $\delta_{S_3 \to S_2}$ during tiling. The dependences that remain are $\Delta \cup \delta_{S_2 \to S_3}$. Since the dependences $\delta_{S_2 \to S_3}$ are iteration private (because their dependence distance is zero), they will have no effect on tiling,[1] and thus applying tiling on the transformed code is exactly equivalent to applying tiling on the original source-level code (since in both cases the tiling will be applied on the set of dependences Δ).

If the right side of a statement has more than 3 expressions (i.e., after transformation to 3AC, the statement is split into two statements or more) we can provide the proof by induction on the other new statements.

6. DESIGN AND IMPLEMENTATION

Our technique has been implemented in the Pluto polyhedral source-to-source compiler [Bondhugula et al. 2008]. The polyhedral model is an algebraic representation and abstraction of programs for reasoning about loop transformations [Feautrier 1991, 1992]. It allows to model and apply complex loop nest transformations addressing most of the parallelism and locality-enhancing challenges.

Figure 15 details the implementation of the proposed technique (steps 3 and 4 are the new steps introduced to the compiler). After calculating dependences, Pluto applies the forward-communication-only (FCO) algorithm [Griebl 2004; Bondhugula et al. 2008] to build a schedule that maximizes data locality and outermost loop parallelism (step 2 in the figure). This algorithm finds and applies the most appropriate combination of loop nest transformations, combining loop distribution, fusion, skewing, shifting, interchange, etc. FCO also identifies tilable bands, but does not perform tiling itself. Tiling (strip-mining and interchange) is performed in a follow-up pass.

We did not modify the FCO algorithm. It still takes into account all false dependences, attempting to convert them into forward dependences through an affine transformation. Since some antidependences may fail to be converted into forward dependences, we use our technique as a postpass to recognize outermost permutable bands larger than those recognized by the standard FCO algorithm. It works as follows.

—Step 3 marks false (output- and anti-) dependences that may be ignored.
—Step 4 calculates dependence directions for all loop levels in order to identify tilable bands. If a false dependence is ignored for a given loop level, its direction for that loop level is set to be zero. A tilable loop band is composed of consecutive loop levels with no negative loop dependence.

[1]Because tiling is possible in two cases: either if the dependence distance is zero (weakly satisfied) or if it is positive (strongly satisfied). The dependence distance in this case is zero so the tiling possibility does not depend on $\delta_{S_2 \to S_3}$ but on other dependences in the kernel.

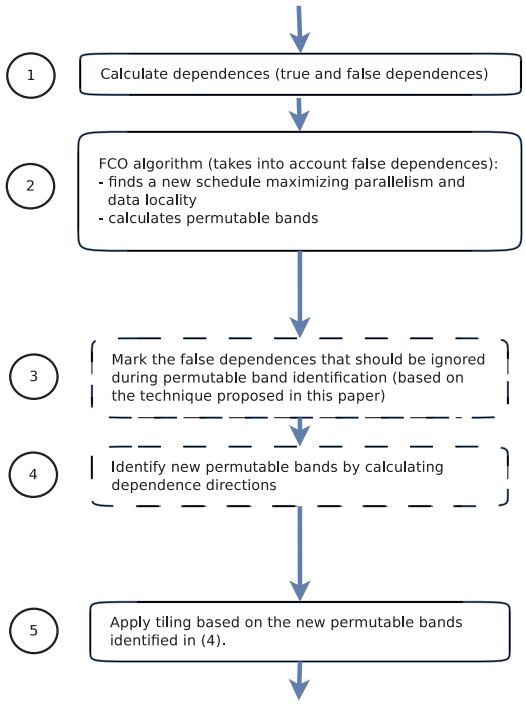

Fig. 15. Implementation of the proposed technique in Pluto.

—Step 5 applies tiling on permutable loop bands identified by step 4.

Although we implemented our technique in Pluto, the method relies on relaxing the correctness conditions for tiling and any compiler that implements some form of array data-flow analysis [Feautrier 1991] can benefit from it.

7. BENCHMARKS

The original PolyBench suite[2] was not initially designed to expose the problem of false dependences. Among all of the benchmarks of the suite, scalar variables are used in only 5 benchmarks, the remaining benchmarks do not use scalars.

To stress the proposed method on realistic false dependences, we chose to introduce scalar variables in each one of the kernels of PolyBench instead of designing a new benchmark suite. These scalar variables were introduced either by transforming the source program into Three-Address Code (3AC) or by applying Partial Redundancy Elimination (PRE). Both transformations (3AC and PRE) have been manually applied to PolyBench. Note that Pluto is a source-to-source compiler and that it does not convert the code to three-address form internally, it does not apply PRE, nor does it privatize or rename variables to eliminate false dependences.

In order to transform the benchmarks into 3AC, each instruction that has more than three operands is transformed into a sequence of instructions, where each one of these new instructions has at most three operands. Temporary scalar variables are used to hold the values calculated by new instructions, and each temporary scalar is assigned

[2]http://www.cse.ohio-state.edu/~pouchet/software/polybench.

```
for (i = 0; i < N; i++) {
  tmp[i] = 0;
  y[i] = 0;
  for (j = 0; j < N; j++) {
    tmp[i] = A[i][j] * x[j] + tmp[i];
    y[i] = B[i][j] * x[j] + y[i];
  }
  y[i] = alpha * tmp[i] + beta * y[i];
}
```

Fig. 16. Original *gesummv* kernel.

```
for (i = 0; i < N; i++) {
  tmp[i] = 0;
  y[i] = 0;
  for (j = 0; j < N; j++) {
    temp1 = A[i][j] * x[j];
    tmp[i] = temp1 + tmp[i];
    temp2 = B[i][j] * x[j];
    y[i] = temp2 + y[i];
  }
  temp3 = alpha * tmp[i];
  temp4 = beta * y[i];
  y[i] = temp3 + temp4;
}
```

Fig. 17. *gesummv* in 3AC form.

only once and used only once. Figures 16 and 17 show an example of transforming the *gesummv* kernel into 3AC.

The second transformation is PRE. We apply PRE to eliminate expressions that are redundant. We apply it whenever it is possible, ignoring any cost/benefit analysis to decide whether applying PRE is profitable for performance or not. Note that we do not apply 3AC when we apply PRE. Figures 18 and 19 show an example of applying PRE on the *gemm* kernel.

Throughout the following sections, PolyBench-3AC is used to refer to PolyBench in three-address form, PolyBench-PRE to refer to PolyBench after applying PRE, and PolyBench-original to refer to the original PolyBench.

To compare the proposed technique with expansion, we apply expansion on PolyBench-3AC manually. This manual expansion is performed as follows: each scalar in the benchmarks of PolyBench-3AC is transformed into an array. The dimension of the resulting array is the minimal dimension that makes tiling possible. Figures 20 and 21 show an example of applying expansion on the three-address form of the *matmul* kernel. We call the resulting benchmark PolyBench-expanded. Expansion is not applied on PolyBench-PRE, because in the case of PRE, the expanded benchmark is equivalent to PolyBench-original.

The manually transformed benchmarks (PolyBench-3AC, PolyBench-PRE, and PolyBench-expanded) are publicly available for reference.[3]

[3]http://www.di.ens.fr/~baghdadi/benchmarks.html

```
/* C := alpha*A*B + beta*C */
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++) {
    C[i][j] = C[i][j] * beta;
    for (k = 0; k < NK; ++k)
      C[i][j] += alpha*A[i][k]*B[k][j];
  }
```

Fig. 18. Original *gemm* kernel.

```
/* C := alpha*A*B + beta*C */
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++) {
    temp0 = C[i][j];
    temp0 = temp0 * beta;
    for (k = 0; k < NK; ++k)
      temp0 += alpha*A[i][k]*B[k][j];
    C[i][j] = temp0;
  }
```

Fig. 19. *gemm* after applying PRE.

```
/* Scalar version of 2mm */
/* D := alpha*A*B*C + beta*D */
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    tmp0 = 0;
    for (k = 0; k < N; k++) {
      tmp1 = alpha*A[i][k];
      tmp2 = tmp1*B[k][j];
      tmp0 += tmp2;
    }
    E[i][j]=tmp0;
  }
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    D[i][j] *= beta;
    for (k = 0; k < N; k++) {
      tmp3 = E[i][k]*C[k][j];
      D[i][j] += tmp3;
    }
  }
```

Fig. 20. *2mm*-3AC (*2mm* in 3AC form).

8. EXPERIMENTAL RESULTS

The experiments were performed on a dual-socket AMD Opteron (Magny-Cours) blade with 2×12 cores at 1.7 GHz, 2×12MB L3 cache and 16GB of RAM running with Linux kernel 2.6.32. The baseline is compiled with GCC 4.4, with optimization flags `-O3 -ffast-math`. This version of GCC performs no further loop nest optimization on these benchmarks, and does not perform any tiling, but it succeeds in automatically vectorizing the generated code in many cases. We use OpenMP as the target of automatic transformations. We compare the original Pluto implementation with our

```
/* Expanded version of 2mm */
/* D := alpha*A*B*C + beta*D */
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    tmp0[i][j] = 0;
    for (k = 0; k < N; k++) {
      tmp1[i][j] = alpha*A[i][k];
      tmp2[i][j] = tmp1[i][j]*B[k][j];
      tmp0[i][j] += tmp2[i][j];
    }
    E[i][j]=tmp0[i][j];
  }
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    D[i][j] *= beta;
    for (k = 0; k < N; k++) {
      tmp3[i][j] = E[i][k]*C[k][j];
      D[i][j] += tmp3[i][j];
    }
  }
```

Fig. 21. Expanded version of *2mm*-3AC.

modified version, reporting the median of the speedups obtained after 30 runs of each benchmark. The pass that implements the proposed technique is activated in the modified Pluto through the option `-ignore-false-dependences`. The options `-tile` and `-parallel` are used to enable tiling and parallelization in Pluto (OpenMP code generation). All the tests are performed using the big datasets in Polybench.

8.1. Evaluating the Effect of Ignoring False Dependences on Tiling

To evaluate the proposed technique, we perform a set of experiments. The results of these experiments are summarized in Figure 22. We report for each experiment whether Pluto succeeded to perform tiling or not. The first column lists the different benchmarks, classified into 4 classes. The second column shows the result of optimizing Polybench-original with Pluto (`pluto -tile -parallel`). It shows that all the kernels are tiled, except 4 kernels (*symm*, *cholesky*, *ludcmp* and *durbin*). *symm*, *cholesky*, and *ludcmp* are not tiled because the original version of the benchmarks already contains scalar variables that induce false dependences and prevent tiling. These benchmarks are examples where the original code written by the programmer contains scalar variables even before any 3AC or PRE code transformations. In the third column, we show the results of optimizing Polybench-3AC with Pluto (`pluto -tile -parallel`). In this experiment, Pluto systematically reports the presence of negative dependences, due to the extra scalar variables introduced by 3AC transformation, and fails to tile the Polybench-3AC benchmarks. None of these benchmarks is tiled.

When we repeat the previous experiment but with the exception of adding the option `-ignore-false-dependences` together with the options `-tile -parallel`, Pluto ignores the false dependences introduced by the transformation to 3AC, recovering the ability to tile Polybench-3AC (the results are presented in the fourth column). Most of the benchmarks are tiled except *lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*, *ludcmp*, and *durbin* (the explanation comes later). In the last experiment (fifth column), Polybench-expanded is compiled with `pluto -tile -parallel`. The

	pluto –tile –parallel			
	Polybench-original	Polybench-3AC	Polybench-3AC	Polybench-expanded
	-	-	–ignore-false-deps	-
2mm	yes	no	yes	yes
3mm	yes	no	yes	yes
gemm	yes	no	yes	yes
gemver	yes	no	yes	yes
gesummv	yes	no	yes	yes
mvt	yes	no	yes	yes
gramschmidt	yes	no	yes	yes
dynprog	yes	no	yes	yes
fdtd-2d	yes	no	yes	yes
trisolv	yes	no	yes	yes
trmm	yes	no	yes	yes
syr2k	yes	no	yes	yes
syrk	yes	no	yes	yes
covariance	yes	no	yes	yes
correlation	yes	no	yes	yes
atax	yes	no	yes	yes
fdtd-apml	yes	no	yes	yes
bicg	yes	no	yes	yes
lu	yes	no	no	yes
jacobi-2d-imper	yes	no	no	yes
jacobi-1d-imper	yes	no	no	yes
seidel	yes	no	no	yes
symm	no	no	yes	yes
cholesky	no	no	yes	yes
ludcmp	no	no	no	yes
durbin	no	no	no	no

Fig. 22. A table showing which kernels were tiled (classified for ease of reference). Different columns show different benchmarks, all of them compiled with `pluto -tile -parallel`. The `-ignore-false-dependences` option is used additionally in the fourth column.

expansion succeeded in enabling the tiling of all the kernels, except the *durbin* kernel. Tiling for *durbin* is not possible, because it would lead to an incorrect schedule.

We repeat later another set of experiments where we focus on performance and show that although expansion enables tiling, it leads in many cases to worse performance due to its high memory footprint compared to the technique of ignoring false dependences.

Similar results are found when the same experiments are applied on Polybench-PRE (Figure 23). Since PRE is only effective for some loop nests, this figure does not show all Polybench kernels. The figure shows that 5 kernels out of 9 are tiled. In the kernels that were tiled (*2mm*, *3mm*, *gemm*, *syr2k*, and *syrk*), the original loop depth is greater than 2, and thus after applying PRE on statements in the innermost loop, the outermost two loops remain free of false dependences, and this enables Pluto to apply tiling on these two outermost loops. *Gemm*, a representative example of these kernels, is presented in Figure 19.

	pluto –tile –parallel		
	Polybench-original	Polybench-PRE	Polybench-PRE
	-	-	–ignore-false-deps
2mm	yes	no	yes
3mm	yes	no	yes
gemm	yes	no	yes
gemver	yes	no	no
gesummv	yes	no	no
mvt	yes	no	no
syr2k	yes	no	yes
syrk	yes	no	yes
correlation	yes	no	no

Fig. 23. A table showing which kernels were tiled. Different columns show different benchmarks, all of them compiled with `pluto -tile -parallel`. The `-ignore-false-dependences` option is used additionally in the fourth column.

```
for (i = 0; i < _PB_N; i++) {
  temp0 = 0;
  temp1 = 0;
  for (j = 0; j < _PB_N; j++) {
    temp0 = temp0 + A[i][j] * x[j];
    temp1 = temp1 + B[i][j] * x[j];
  }
  y[i] = alpha * temp0 + beta * temp1;
}
```

Fig. 24. *gesummv* kernel after applying PRE.

In the other kernels that were not tiled (*gemver*, *gesummv*, *mvt*, and *correlation*) the original loop depth is exactly 2 and applying the PRE optimization introduces a temporary scalar variable (to hold invariant data across the iterations of the inner loop); this variable induces false dependences preventing Pluto from applying tiling. *Gesummv*, a representative example of kernels with a loop depth equal to 2, is presented in Figure 24. Note that PRE also makes the loops imperfectly nested in the latter case, but this does not in itself impact the ability of Pluto to tile the loops.

8.2. Performance Evaluation for Polybench-3AC

The following experiments focus on performance. In all of the experiments, the benchmark is first compiled with the Pluto source-to-source compiler, and then the resulting optimized code is compiled with GCC. The figures show different speedups compared to the baseline. Each speedup is the execution time of the kernel being tested normalized to the execution time of the baseline. The baseline is compiled with GCC 4.4, with optimization flags `-O3 -ffast-math`. Values greater than 1 indicate good speedups, values smaller than 1 indicate slowdowns (the speedup of the baseline is 1).

Figure 25 shows the difference in performance between the optimization of Polybench-original with `pluto -parallel -tile` and the optimization of Polybench-3AC with the same options. As presented in Figure 22, Pluto fails to tile Polybench after three-address code transformation.

Figure 26 shows the difference in performance between optimizing Polybench-3AC with and without the `-ignore-false-dependences` option. The figures compare also between ignoring false dependences and applying array and scalar expansion.

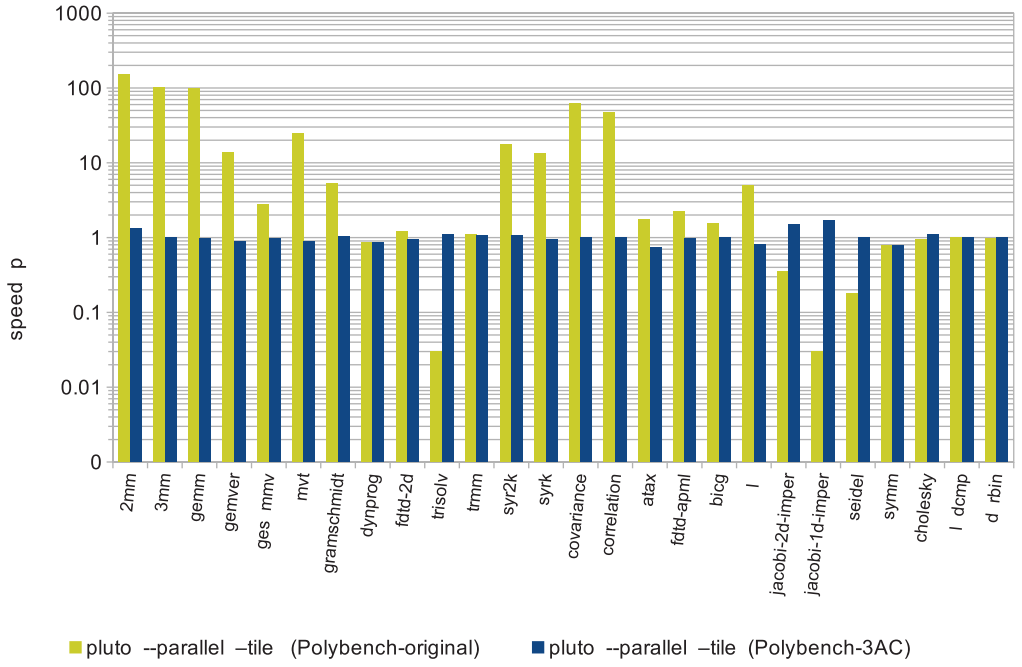

Fig. 25. The effect of 3AC transformation on tiling.

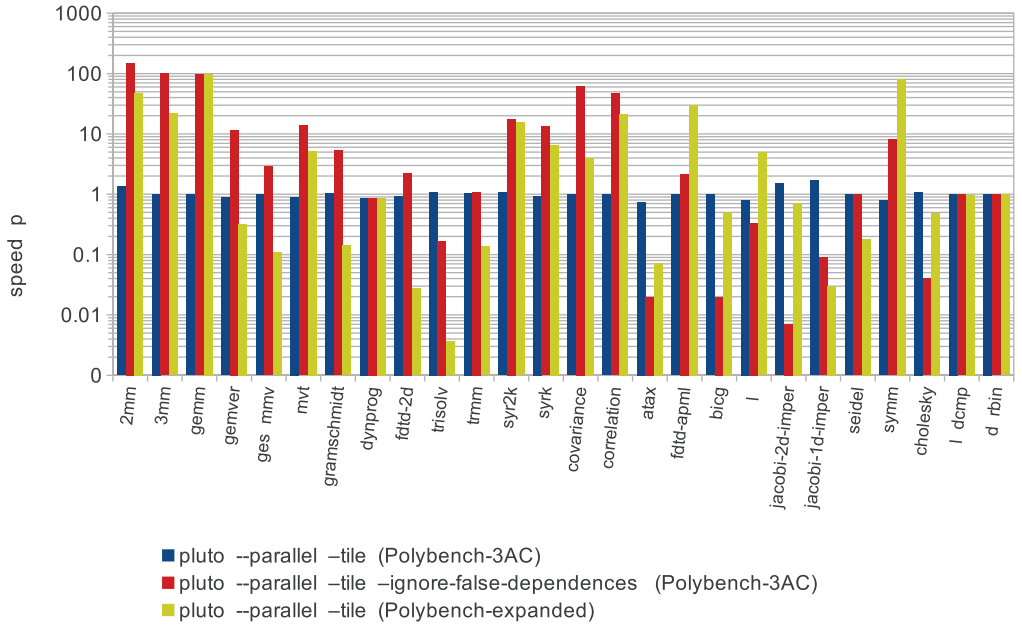

Fig. 26. Speedup against nontiled sequential code for PolyBench-3AC (first part).

```
#define B 32
#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N−1)/B);t2++)
  for (t3=0;t3<=floor((N−1)/B);t3++)
    for (t4=B∗t2;t4<=min(N−1,B∗t2+B−1);t4++)
      for (t5=B∗t3;t5<=min(N−1,B∗t3+B−1);t5++) {
        tmp0=0;
        for (t8=0;t8<=N−1;t8++) {
          tmp1=alpha∗A[t4][t8];
          tmp2=tmp1∗B[t8][t5];
          tmp0+=tmp2;
        }
      E[t4][t5]=tmp0;
    }

#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N−1)/B);t2++)
  for (t3=0;t3<=floor((N−1)/B);t3++)
    for (t4=B∗t2;t4<=min(N−1,B∗t2+B−1);t4++)
      for (t5=B∗t3;t5<=min(N−1,B∗t3+B−1);t5++)
        for (t8=0;t8<=N−1;t8++) {
          tmp3=E[t4][t5]∗C[t5][t8];
          D[t4][t8]+=tmp3;
        }
```

Fig. 27. *2mm*-3AC optimized ignoring false dependences.

The results shown in Figure 22 are important to understand these figures. Four classes of kernels can be identified by analyzing the figures. While some of the kernels show slowdowns, many kernels show a speedup. This speedup is obtained only when false dependences are ignored or when expansion is applied. The explanation for the slowdowns is presented in Section 8.3.

(1) The first and largest class is represented by *2mm*, *3mm*, *gemm*, *gemver*, *gesummv*, *mvt*, *covariance*, *correlation*, *syrk*, and *syr2k*. The `-ignore-false-dependences` option, applied to 3AC, restores the full tiling potential of Pluto. While expansion was effective in enabling tiling, the performance obtained with expansion is lower compared to the performance obtained when false dependences are ignored. The performance when expansion is applied is lower, because, with expansion, Pluto gets more freedom in applying loop nest transformations, and thus it applies transformations that on one hand enable better vectorization and parallelism but on the other hand prevent array contraction.[4] We explain this in the example presented in Figures 20, 21, 27, and 28. Figure 20 shows *2mm* in 3AC form, Figure 21 shows *2mm* after expansion (expansion is applied on the 3AC by transforming the scalars `tmp0`, `tmp1`, `tmp2`, and `tmp3` into arrays). Figures 27 and 28 show *2mm* after applying loop nest transformations with Pluto. To optimize the expanded *2mm*, Pluto chose to apply loop distribution and to reschedule statements moving the statement `tmp0` into a separate loop. Although loop distribution enables better vectorization and parallelism in general, in this example, it prevents the compiler to contract the `tmp0` array and thus the memory overhead created by expansion is not eliminated.

[4]Contraction is used to transform the expanded arrays back into scalars. This transformation eliminates the memory overhead created by expansion but is not always possible.

```
#define B 32
#pragma omp parallel for private(t3,t4,t5)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++) {
        D[t4][t5]*=beta;
        tmp0[t4][t5]=0;
      }
#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++) {
        for (t8=0;t8<=N-1;t8++) {
          tmp1[t4][t5]=alpha*A[t4][t8];
          tmp2[t4][t5]=tmp1[t4][t5]*B[t8][t5];
          tmp0[t4][t5]+=tmp2[t4][t5];
        }
        E[t4][t5]=tmp0[t4][t5];
        for (t8=0;t8<=N-1;t8++)
          tmp3[t4][t8]=E[t4][t5]*C[t5][t8];
        for (t8=0;t8<=N-1;t8++)
          D[t4][t8]+=tmp3[t4][t8];
      }
```

Fig. 28. Optimizing the expanded version of *2mm*-3AC.

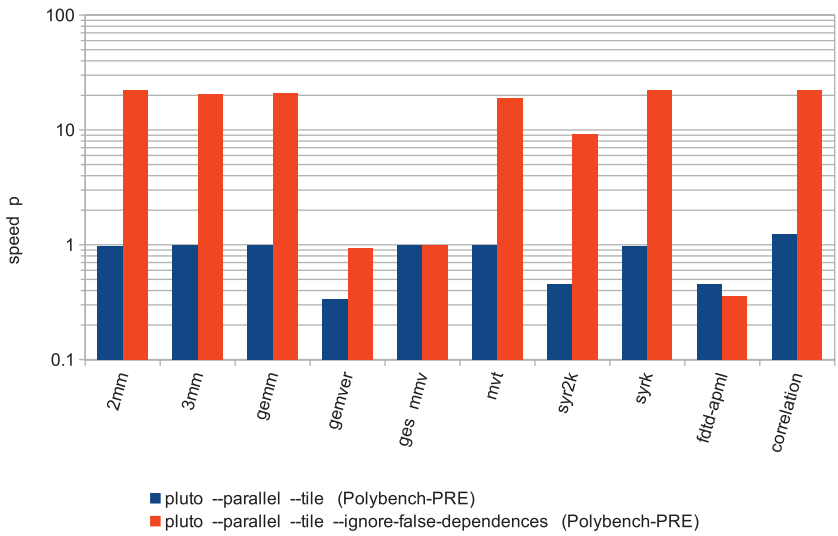

Fig. 29. Speedup against nontiled sequential code for PolyBench-PRE.

Loop distribution was also applied on the last loop of *2mm* leading to a scheduling of the statements manipulating `tmp3` into two separate loops which prevents the contraction of `tmp3`.

(2) The second class contains 7 kernels (see Figure 22): *atax*, *fdtd-apml*, *bicg*, *lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, and *seidel*. The kernels in this class are either kernels where we have better performance for expansion over ignoring false dependences (*atax*, *fdtd-apml*, and *bicg*) or where ignoring false dependences does not enable tiling (*lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*). These kernels are a part of the same class because in all of these kernels, the FCO algorithm does not select the right loop transformation due to false dependences. As explained in Section 6, the FCO algorithm performs many loop nest transformations that are needed to enable tiling and to enable outermost parallelism. Skewing is one example of these code transformations. Since loop skewing happens during FCO and since we do not ignore false dependences during FCO, skewing is not performed and thus tiling is not possible. The expanded code has no false dependences which gives much freedom for the FCO algorithm to perform the adequate affine transformations including skewing. *jacobi-2d-imper* is a good representative of the kernels where false dependences limit the ability of the FCO algorithm to perform loop transformations. To be able to tile the code and to extract outermost parallelism in *jacobi-2d-imper*, the FCO algorithm has to perform loop skewing. Due to false dependences in three-address code, FCO could not apply skewing and thus tiling was not applied and only innermost loops could be parallelized which led to a loss in performance. This class of benchmarks motivates future work towards the integration of noninterference constraints into the FCO algorithm itself.

(3) In the third class, represented by *cholesky* and *symm*, the original code contains scalar variables, and thus Pluto fails to find tilable loop bands (Figure 22). Ignoring false dependences enables Pluto to find tilable bands, resulting in performance improvements. Although the proposed technique enables loop tiling in *symm*, the performance obtained with expansion is better than the performance obtained with ignoring false dependences, because expansion enables the compiler to perform loop distribution which helps Pluto to find more profitable, outermost parallelism on the newly created loops instead of a less profitable, innermost parallelism on the original loop. Finding an outermost parallelism in the case of *symm* is not possible without expansion. This is an interesting example that shows that expansion can, in fact, enable aggressive loop nest transformations and that the gain from these transformations can bypass the loss of performance induced by memory footprint after expansion. This benchmark motivates future work towards studying the synergy between using expansion and ignoring false dependences to reach higher orders of performance gains.

(4) In the fourth class, represented by *ludcmp*, and *durbin*, tiling is not possible as tiling is not a valid transformation for these kernels (tiling will break live ranges). The original code contains negative dependences that cannot be ignored even by using our technique, because the relaxed permutability criterion defined in Section 4.4 does not hold (in Polybench-original and in Polybench-3AC). In *durbin*, even after expansion, the application of tiling remains invalid.

8.3. Reasons for Slowdowns

In kernels such as *atax*, *fdtd-2d*, *trisolv*, *bicg*, *lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*, and *cholesky*, the overhead of parallelization is higher than the benefit of parallelization. In *trmm*, for example (presented in Figure 30), the overhead of synchronization at the end of each iteration of the loop `t2` (the loop with induction variable `t2`) is higher than the benefit of the parallel execution especially in the case

```
for (t1=1;t1<=NI−1;t1++)
  for (t2=ceil((t1−31)/32);t2<=floor((t1+NI−2)/16);t2++) {
    lb1=max(ceil(t2/2),ceil((t1−31)/32));
    ub1=min(floor((t1+NI−1)/32),floor((t1+32*t2+31)/64));
    #pragma omp parallel for shared(t1,t2,lb1,ub1) private(t3,t4,t5,t6)
    for (t3=lb1; t3<=ub1; t3++)
      for(t4=max(−t1+32*t3,32*t2−32*t3);t4<=min(min(32*t3+30,t1+NI−2),32*t2−32*t3+31);t4++)
        for(t5=max(max(t1,32*t3),t4+1);t5<=min(min(t1+t4,32*t3+31),t1+NI−1);t5++) {
          temp1[t1][−t1+t5]=A[t1][t1+t4−t5]*B[−t1+t5][t1+t4−t5];;
          B[t1][−t1+t5]+=alpha*temp1[t1][−t1+t5];;
        }
  }
```

Fig. 30. Innermost parallelism in the expanded version of *trmm*.

of a small number of iterations in the inner loop nest. Although the parallelization is not profitable, Pluto parallelized the inner loop nest because it lacks an adequate profitability model. In some other kernels, the FCO algorithm performs partial loop distribution creating many parallel inner loops nested in an outer sequential loop. Parallelizing many inner loops impacts performance negatively, and leads to more slowdowns. Having a model that helps Pluto to decide when the parallelization is profitable and when it is not would enable Pluto to avoid such slowdowns, but this is orthogonal to the contribution of this article.

8.4. Performance Evaluation for Polybench-PRE

Figure 29 shows the effect of optimizing the Polybench-PRE with and without the `-ignore-false-dependences` option. This figure, when analyzed together with Figure 23, shows the performance enhancement obtained when tiling is enabled.

Tiling for the kernels *gemver*, *gesummv*, and *fdtd-apml* is not valid, consequently these kernels do not show any speedup. For the other kernels, the tests show that the use of the option `-ignore-false-dependences` restores Pluto's ability to identify tilable bands despite the application of PRE.

9. CONCLUSION AND FUTURE WORK

Loop tiling is one of the most profitable loop nest transformations. Due to its wide applicability, any relaxation on the safety criterion for tiling will impact a wide range of programs. The proposed technique allows the compiler to ignore false dependences between iteration-private live ranges, allowing the compiler to discover larger bands of tilable loops. Using our technique, loop tiling can be applied without any expansion or privatization, apart from the necessary marking of thread-private data for parallelization. The impact on the memory footprint is minimized, and the overheads of array expansion are avoided.

We have shown that ignoring false dependences for iteration-private live ranges is particularly effective to enable tiling on three-address code, or after applying scalar optimizations such as partial redundancy elimination and loop-invariant code motion.

We are investigating how to integrate noninterference constraints into the FCO algorithm itself in order to ignore false dependences on a more general class of affine transformations and avoid the performance degradation observed when converting a few benchmarks (such as *lu* and *jacobi-2d)* to three-address code. We are also interested in combining this technique with on-demand array expansion, to enable the maximal extraction of tilable parallel loops with a minimal memory footprint.

REFERENCES

BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM Press, New York, 101–113.

BOUCHEZ, F., DARTE, A., GUILLON, C., AND RASTELLO, F. 2006. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? Or revisiting register allocation: Why and how. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*. Lecture Notes in Computer Science. Springer.

CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI'90)*.

CHAITIN, G. J., AUSLANDER, M. A., COCKE, A. K. C., HOPKINS, M. E., AND MARKSTEIN, W. P. 1981. Register allocation via coloring. *Comput. Lang. 6*, 47–57.

COHEN, A. 1999. Parallelization via constrained storage mapping optimization. In *Proceedings of the International Symposium on High Performance Computing*, K. J. A. Fukuda, S. Tomita, and C. Polychronopoulos, Eds. Lecture Notes in Computer Science, vol. 1615. Springer, 83–94.

COHEN, A. AND LEFEBVRE, V. 1998. Optimization of storage mappings for parallel programs. In *Proceedings of the International European Conference on Parallel Processing (Euro-Par'98)*. Lecture Notes in Computer Science, vol. 1685. Springer, 375–382.

COHEN, A. AND ROHOU, E. 2010. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Proceedings of the Design Automation Conference (DAC'10)*. IEEEE.

DARTE, A. AND HUARD, G. 2005. New complexity results on array contraction and related problems. *J. VLSI Signal Process. Syst. 40*, 1, 35–55.

FEAUTRIER, P. 1988. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*. ACM Press, New York, 429–441.

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program. 20*, 1, 23–53.

FEAUTRIER, P. 1992. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. Parallel Program. 21*, 6, 389-420. See also Part I, one dimensional time *Int. J. Parallel Program. 21*, 5, 315–348.

GRIEBL, M. 2004. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis, Facultät für Mathematik und Informatik, Universität Passau.

GUPTA, M. 1997. On privatization of variables for data-parallel execution. In *Proceedings of the 11th International Parallel Processing Symposium*. IEEE, 533–541.

HACK, S., GRUND, D., AND GOOS, G. 2006. Register allocation for programs in ssa-form. In *Proceedings of the International Conference on Compiler Construction (CC'06)*. 247–262.

HMPP. 2012. Hmpp workbench: A directive-based multi-language and multi-target hybrid programming model. http://www.caps-entreprise.com/hmpp.html.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'88)*. 319–328.

KENNEDY, K. AND ALLEN, J. R. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann.

KNOOP, J., RUTHING, O., AND STEFFEN, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16*, 4, 1117–1155.

LEFEBVRE, V. AND FEAUTRIER, P. 1998. Automatic storage management for parallel programs. *Parallel Comput. 24*, 649–671.

LI, Z. 1992. Array privatization for parallel execution of loops. In *Proceedings of the 6th International Conference on Supercomputing*. ACM Press, New York, 313–322.

MAYDAN, D. E., AMARASINGHE, S. P., AND LAM, M. S. 1993. Array-Data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*. 2–15.

MIDKIFF, S. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

POP, S., COHEN, A., BASTOUL, C., GIRBAL, S., SILBER, G.-A., AND VASILACHE, N. 2006. GRAPHITE: Polyhedral analyses and optimizations for gcc. In *Proceedings of the GCC Developers Summit*.

QUILLERE, F. AND RAJOPADHYE, S. 2000. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst. 22*, 5, 773–815.

THE PORTLAND GROUP. 2010. *PGI accelerator programming model for Fortran & C v.1.3*. The Portland Group.

THIES, W., VIVIEN, F., SHELDON, J., AND AMARASINGHE, S. 2001. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*.

TRIFUNOVIC, K., COHEN, A., EDELSOHN, D., LI, F., GROSSER, T., JAGASIA, H., LADELSKY, R., POP, S., SJODIN, J., AND UPADRASTA, R. 2010. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*.

TRIFUNOVIC, K., COHEN, A., LADELSKI, R., AND LI, F. 2011. Elimination of memory-based dependences for loop-nest optimization and parallelization. In *3rd GCC Research Opportunities Workshop*.

TU, P. AND PADUA, D. 1994. Automatic array privatization. In *Languages and Compilers for Parallel Computing,* U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 768. Springer, 500–521.

VASILACHE, N. 2007. Scalable program optimization techniques in the polyhedral model. Ph.D. thesis, University of Paris-Sud, INRIA.

VASILACHE, N., BASTOUL, C., COHEN, A., AND GIRBAL, S. 2006. Violated dependence analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing*. ACM Press, New York, 335–344.

VERDOOLAEGE, S. 2010. Isl: An integer set library for the polyhedral model. In *Proceedings of the International Conference on Mathematical Software (ICMS'10)*, J. V. D. Hoeven, M. Joswig, N. Takayama, and K. Fukuda, Eds. Lecture Notes in Computer Science, vol. 6327. Springer, 299–302.

WOLF, M. E. AND LAM, M. S. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst. 2*, 4, 452–471.

Received June 2012; revised November 2012; accepted November 2012