Variable Liberalization

SANYAM MEHTA and PEN-CHUNG YEW, University of Minnesota, Twin Cities

In the wake of the current trend of increasing the number of cores on a chip, compiler optimizations for improving the memory performance have assumed increased importance. Loop fusion is one such key optimization that can alleviate *memory* and *bandwidth wall* and thus improve parallel performance. However, we find that loop fusion in interesting memory-intensive applications is prevented by the existence of dependences between temporary variables that appear in different loop nests. Furthermore, known techniques of allowing useful transformations in the presence of temporary variables, such as privatization and expansion, prove insufficient in such cases.

In this work, we introduce *variable liberalization*, a technique that selectively removes dependences on temporary variables in different loop nests to achieve loop fusion while preserving the semantical correctness of the optimized program. This removal of extra-stringent dependences effectively amounts to variable expansion, thus achieving the benefit of an increased degree of freedom for program transformation but without an actual expansion. Hence, there is no corresponding increase in the memory footprint incurred. We implement *liberalization* in the Pluto polyhedral compiler and evaluate its performance on nine hot regions in five real applications. Results demonstrate parallel performance improvement of 1.92× over the Intel compiler, averaged over the nine hot regions, and an overall improvement of as much as 2.17× for an entire application, on an eight-core Intel Xeon processor.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Compilers**

Additional Key Words and Phrases: Polyhedral compiler, dependence refinement, scheduling, liberalization, loop fusion, parallelization

ACM Reference Format:
Sanyam Mehta and Pen-Chung Yew. 2016. Variable liberalization. ACM Trans. Archit. Code Optim. 13, 3, Article 23 (August 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2963101

1. INTRODUCTION

With the increase in the number of cores on a chip (or processors in a node) and the consequent worsening of the existing problems of *memory* and *bandwidth wall*, there is a renewed focus on the optimization capabilities of a parallelizing compiler. A parallelizing compiler should not only help to exploit the available parallelism in the host hardware but also alleviate the problems of memory and bandwidth wall by performing memory optimizations such as *loop tiling, data prefetching, loop fusion*, and other supporting optimizations such as *loop shifting* and *loop interchange*. While these important responsibilities of a parallelizing compiler are well recognized, it is also well known that compilers often fall short in capitalizing on the optimization opportunities provided by a target application.

Authors' addresses: S. Mehta, Cray Plaza, 380 Jackson Street, St Paul, MN 55101; email: sanyam.mehta@gmail.com; P.-C. Yew, Department of Computer Science, University of Minnesota Twin Cities, 200 Union St SE, Minneapolis, MN 55455; email: yew@cs.umn.edu.

DOI: http://dx.doi.org/10.1145/2963101

```
for(i1=0;i1<Nx;i1++) {
  for(j1=0;j1<Ny;j1++) {

    for(k1=0;k1<Nz;k1++) {
      S1: a = rho[i1][j1][k1];
      S2: tmp[k1] = y[i1][j1][k1] + a;
      S3: x[i1][j1][k1] = x[i1][j1][k1] + a*c;
    }

    for(k1=1;k1<Nz-1;k1++) {
      S4: z[i1][j1][k1] = x[i1][j1][k1] + tmp[k1+1] - tmp[k1-1];
    }

  }
}

for(i2=0;i2<Nx;i2++) {
  for(j2=0;j2<Ny;j2++) {

    for(k2=0;k2<Nz;k2++) {
      S5: a = rho[i2][j2][k2];
      S6: tmp[k2] = y[i2][j2][k2] * 0.5 + a;
      S7: x[i2][j2][k2] = x[i2][j2][k2] + a*b;
    }

    for(k2=1;k2<Nz-1;k2++) {
      S8: z[i2][j2][k2] = x[i2][j2][k2] + tmp[k2+1] - tmp[k2-1];
    }

  }
}
```

(a)

```
parfor(i=0;i<Nx;i++) {  // fused loop
  for(j=0;j<Ny;j++) {  // fused loop

    for(k1=0;k1<Nz;k1++) {
      a = rho[i][j][k1];
      tmp[k1] = y[i1][j1][k1] + a;
      x[i][j][k1] = x[i][j][k1] + a*c;
    }

    for(k1=1;k1<Nz-1;k1++) {
      z[i][j][k1] = x[i][j][k1] + tmp[k1+1] - tmp[k1-1];
    }

    for(k2=0;k2<Nz;k2++) {
      a = rho[i][j][k2];
      tmp[k2] = y[i1][j1][k1] * 0.5 + a;
      x[i][j][k2] = x[i][j][k2] + a*b;
    }

    for(k2=1;k2<Nz-1;k2++) {
      z[i][j][k2] = x[i][j][k2] + tmp[k2+1] - tmp[k2-1];
    }

  }
}
```

(b)

Fig. 1. (a) Original code and (b) Optimized (or transformed) code after *a* and *tmp* are privatized.

One key reason for this shortfall is that most existing production compilers built on traditional wisdom often limit themselves to optimizing only small scopes in the entire program [Rauchwerger and Padua 1999; Ding and Kennedy 2004; Vandierendonck et al. 2010] or kernels [Bondhugula et al. 2008; Pouchet et al. 2007, 2008; Girbal et al. 2006]. Even in some recent work, such as Johnson et al. [2013], the authors only analyze individual hot loops in target applications to mark those loops as parallel or determine the validity of loop distribution. However, our experiments with several scientific applications from the SPEC benchmark suite reveal that there are many opportunities for improvement in memory (and parallel) performance of those benchmarks through global program optimizations (or transformations) such as applying loop fusion across a sequence of such hot loops. Such loop fusion across multiple loop nests saves the cost of fork-join synchronization between loops and, more importantly, significantly improves *temporal locality* and thus saves costly off-chip memory accesses. Both these benefits are of increasing importance in the era of multi-/many cores.

In the past, there has been work on loop fusion [Kennedy and McKinley 1994; Megiddo and Sarkar 1997; Singhai and McKinley 1997; Ding and Kennedy 2004], but the existing production compilers still prove insufficient in fusing multiple loop nests. We find that, in addition to some of the other limitations such as non-conformable loop bounds or loop orders in different nests, and the existence of imperfect nests, it is the existence of (extra-)stringent memory dependences on temporary variables (scalars or low-dimensional arrays as shown in Figure 1(a)) in different nests that is the key factor behind the dismal performance of existing compilers with regard to global program optimizations. Such memory dependences severely limit the degrees of freedom for loop transformations such as loop fusion and loop interchange. *It is important to note that we cannot simply get rid of all such dependences to enable loop fusion, because fusion merges multiple definitions and uses in the fused loop's body that can potentially violate*

program correctness. Such dependences thus require specific treatment. In this work, we make the following contributions,

(1) We analyze an important cause of the weakness of existing compilers in achieving global program transformations—the (extra-)stringent data dependences caused by temporary variables that appear in different loop nests and have the same variable name.
(2) We propose *variable liberalization*,[1] a technique that strategically removes dependences on temporary variables (and, consequently, liberalizes scheduling of statements containing temporary variables) in different nests to release the degrees of freedom needed to express effective loop transformations such as loop fusion and loop interchange while preserving program correctness. Variable liberalization differs from *variable privatization* because (1) it removes dependences on multiple outer loops *across loop nests* instead of just the outermost loop in *a single nest* and (2) it is performed to also enable fusion and not just coarse-grained parallelization. Liberalization differs from *expansion* because there is no real expansion of a variable's dimensions that takes place in order to accomplish fusion and parallelization. That is, there is no change in memory allocation, but the same effect is achieved through selective removal of dependences. It thus provides the benefits of both privatization and expansion and avoids their drawbacks.
(3) We implement our work in the state-of-the-art Pluto polyhedral compiler and evaluate the improvement obtained in terms of parallel performance on nine hot regions in five real scientific applications from the SPEC and NAS Parallel Benchmark suites that have not been effectively optimized by current production compilers even though they contain significant opportunity. With the recent advances in the scalability of polyhedral compilers [Mehta and Yew 2015] that has rendered the compile time of large programs comparable to current production compilers, we believe that this work will further motivate scientific programmers to leverage polyhedral optimizations for real applications.

The rest of the article is organized as follows. Section 2 reinforces the motivation for this work through a simple example program that exposes the limitation of existing compilers in global program transformation and shows the potential benefits from overcoming that limitation. Section 3 provides a background on dependence analysis with focus on temporary variables and also introduces key concepts to explain our proposed variable liberalization optimization. This is followed by a detailed discussion of our approach and its application in the different cases seen in real applications in Section 4. Section 5 puts all previous discussion together in the form of an algorithm that implements liberalization. In Section 6, we evaluate our approach against state-of-the-art compilers and discuss results in each case. This is followed by a discussion on related work in Section 7. Finally, Section 8 presents the conclusions from this work.

2. MOTIVATION

Figure 1(a) shows an example program that assumes some of the features that are characteristic of real scientific applications. These features include extensive use of temporary variables such as the scalar variable a and the array variable *tmp*; excellent opportunities for data reuse across loop nests such as that in arrays *rho*, x, and z; and imperfectly nested loops. Figure 1(b) shows a transformed program where the two loop nests in the original program (Figure 1(a)) are fused into a single nest and the

[1]Liberalization literally means removing barriers to allow freer interaction between two parties. Variable liberalization similarly removes the extra-stringent data dependences caused by temporary variables in two loop nests to allow global optimizations such as loop fusion between them.

outermost loop has been parallelized (after marking variables *a* and *tmp* as *private* to each thread) . Clearly, the transformed program is equivalent to the original program since the transformed program satisfies all memory dependences.

In order to evaluate the strength of existing compilers with regard to performing loop optimizations on real scientific application code, we compiled the code in Figure 1(a) with ICC (v14, a state-of-the-art production compiler) and Pluto (0.11.4, a state-of-the-art polyhedral compiler). Our experiments led to the following interesting findings.

(1) Neither ICC nor Pluto (+ ICC, since Pluto is a source-to-source compiler and needs ICC as the backend compiler) were able to either achieve loop fusion, even though there is reuse, or mark the outermost loop as parallel for coarse-grained parallelization.
(2) The transformed program in Figure 1(b) fuses both the loop nests and thus achieves a sequential performance improvement of 1.19× over the original program in Figure 1(a) (compiled using ICC). This demonstrates the efficacy of exploiting data reuse across loop nests. Pluto performs even worse than ICC (0.48× as compared to ICC) in this case because it fuses the two k1-loops in the first loop nest and the two k2-loops in the second. Although this helps to gain some reuse, there is also loss of vectorization in the innermost loops.
(3) The transformed program in Figure 1(b) achieves a parallel performance improvement of 4.47× over the original program (compiled using ICC) when running eight threads in parallel, clearly revealing the optimization potential. Even after the two loop nests in Figure 1(a) are explicitly marked parallel (and variables *a* and *tmp* privatized), parallel performance improvement of the transformed program in Figure 1(b) is still 1.48× over this manually parallelized version (compiled using ICC). This is because of the combined savings of off-chip memory accesses and fork-join synchronization through loop fusion.

The key reason for the poor performance of ICC and Pluto is that the use of same temporary variables in the two loop nests introduces fusion-preventing backward dependences on the outer loops of these nests. These dependences are not false since their removal allows a perfect fusion of the two nests, which may violate program correctness due to incorrect *definitions* reaching certain *uses* (see Figure 4 for an example). The following section (Section 3), however, shows that while all dependences involving temporary variables *cannot* be removed, many can be removed to achieve liberalization. In other words, those dependences are more stringent than needed and artificially lead to a reduced degree of freedom for transformations such as fusion and parallelization. Section 3 then introduces concepts that lead us to variable liberalization, the technique that removes dependences selectively to generate the transformed code as in Figure 1(b).

2.1. Why Not Scalar and Array Expansion (Perhaps, Followed by Contraction)?

Scalar or array expansion involves transforming the scalar or low-dimension array variables (such as *a* and *tmp*[] in our motivating example) into full-dimension arrays (such as a[][][] and tmp[][][]). Expansion essentially creates a new memory location for the temporary variable for each iteration of the loop nest. It thus removes loop-carried dependences between different references of the variable, leading to effective loop transformations. However, this technique (1) increases the memory footprint significantly and thus degrades temporal locality both in cache [Thies et al. 2001] and registers [Callahan et al. 1990] and (2) requires declaration of new data structures and corresponding changes in the source code. The authors in Lefebvre and Feautrier [1998], Quilleré and Rajopadhye [2000], and Darte and Huard [2005] propose to perform array contraction to reduce the memory footprint after an initial expansion step.

Their rationale is that expansion will enable aggressive optimizations, and array contraction will help to recover the loss in temporal locality at a later stage. Similarly, Cohen [1999] proposes maximal static expansion, followed by finding parallelism (or possibly other transformations) and then performing contraction. Thies et al. [2001], on the other hand, simultaneously consider the goals of reducing memory footprint (or storage) and achieving good schedules in a single mathematical framework. These works all require us to contract the expanded temporary variables, and the following problem results when trying to do so.

If we perform an expansion of scalar a in the example 1 of Figure 1(a), then it will be converted into a three-dimensional (3D) array. As a result, the statement S3 containing array x will become free to separate out of the first $k1$ loop. It can thus be fused with statement S4 (and distributed with statements S1 and S2) in the second $k1$ loop. As a result, this transformation is completely feasible (and likely, because it is not detrimental to reuse). Now that the use and definition of array a are in different loops (i.e., a loop distribution has been performed), array a cannot be contracted back to a scalar, losing the opportunity for optimizing storage. Thies et al., however, do point out the technique of adding certain additional dependence edges to prevent certain schedules. This could be used to prevent the undesired distribution as above, but the nature of those additional dependences for the purpose of preventing distribution is not discussed.

2.2. Why Not Scalar and Array Renaming Across Loop Nests?

Renaming scalar and array variables in different loop nests could also result in disappearance of the transformation-limiting loop-carried dependences across loop nests and will facilitate loop fusion and parallelization. But, there are three disadvantages of such a strategy that preclude us from implementing it.

(1) The key limiting factor is that, after fusion of loop nests with (multiple) renamed temporary variables, the opportunity for reuse of data (accessed by temporary variables) in the higher levels of the memory hierarchy is lost. In addition, the working set in those higher (and smaller) levels of the memory hierarchy expands by a factor of the number of temporary variables and the number of merged nests. This degrades program performance, especially in the presence of temporary array variables. For example, the transformed *zeusmp* benchmark application program from the SPEC suite (containing 24 temporary arrays in a loop nest) runs 7% slower when loop fusion is performed after renaming as compared to that performed without renaming (i.e., through variable liberalization). The performance degradation may be even larger for certain other scientific applications that use even more temporary variables.

    In addition, a proposal to revert the temporary variables back to their original names after fusion to contract the working set may not be feasible in many cases. Consider, for example, the code in Figure 2(a). If we rename the temporary array x in the second loop nest to y, then the two loops nests become fusable, and the code in Figure 2(b) is a valid fused code that results. Clearly, the temporary array y cannot be renamed back to x to contract the working set and achieve reuse after fusion because that would hurt program correctness. In this case, however, fusion could still be achieved by selective removal of dependences as in variable liberalization.

(2) As a result of this substantial increase in the number of temporary variables, the number of hardware prefetch streams also increase in the same proportion as each temporary array triggers one of those prefetch streams. Since every processor has a limited number of these streams, the transformed program can easily fall short of the needed prefetch streams, leading to performance degradation.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    S1: x[j] = a[i][j]; }
  for (j=1; j<N-1; j++) {
    S2: b[i][j] = x[j+1] – x[j-1]; }
}

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    S3: x[j] = c[i][j]; }
  for (j=1; j<N-1; j++) {
    S4: d[i][j] = x[j+1] – x[j-1]; }
}
```

(a)

```
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
    S1: x[j] = a[i][j];
    S3: y[j] = c[i][j]; }
  for (j=1;j<N-1;j++) {
    S2: b[i][j] = x[j+1] – x[j-1];
    S4: d[i][j] = y[j+1] – y[j-1]; }
}
```

(b)

Fig. 2. (a) Original code and (b) fused code after renaming.

(3) Another drawback is that renaming will require generation of new variable declarations and also modification of all references to the temporary variable in all loop nests.

Our framework, on the other hand, does not require any of those changes to the original source code and, more importantly, does not cause any unnecessary increase in the number of variables in the program. This promotes efficient data reuse in higher levels of cache and effective prefetching by the hardware.

3. BACKGROUND

Instancewise dependence analysis [Feautrier 1988] employed in state-of-the-art polyhedral compilers [Trifunovic et al. 2010; Bondhugula et al. 2008; Grosser et al. 2011] precisely tells us which iterations of the involved statements are dependent (rather than merely revealing which statements are dependent) and is thus more effective in reasoning about the feasibility of loop transformations. In this section, we use instancewise dependence analysis to show how dependences between temporary variables artificially suppress fusion of multiple loop nests. We first show how loop interchange (one of the fusion-enabling transformations) is hampered by dependences on temporary variables within the same strongly connected component (SCC), followed by an example of how loop fusion is prevented by similar dependences across different SCCs or loop nests. In the polyhedral model, the dependence relation between individual instances of statements connected by a dependence edge in the data dependence graph is captured by an affine relation between the iterations and accessed data, called the *dependence polyhedron*. Through the rest of the article, we refer to dependence relations between individual instances of statements as dependences, and the set of between two references in the source and destination statements as dependence polyhedron.

Instancewise dependence analysis could be used to refer to two kinds of analysis: memory based and dataflow or value based. In memory-based dependence analysis, a statement instance is considered to depend on any previous statement instance accessing the same data element provided one of the two accesses is a write. On the other hand, in dataflow analysis, a statement instance performing a read only depends on the last preceding statement instance performing a write to the same data element. In this work, we use ISL [Verdoolaege 2010] for the purpose of dependence analysis in Pluto. ISL implements both memory-based dependence analysis and also another form of dependence analysis that generalizes between the two more traditional forms

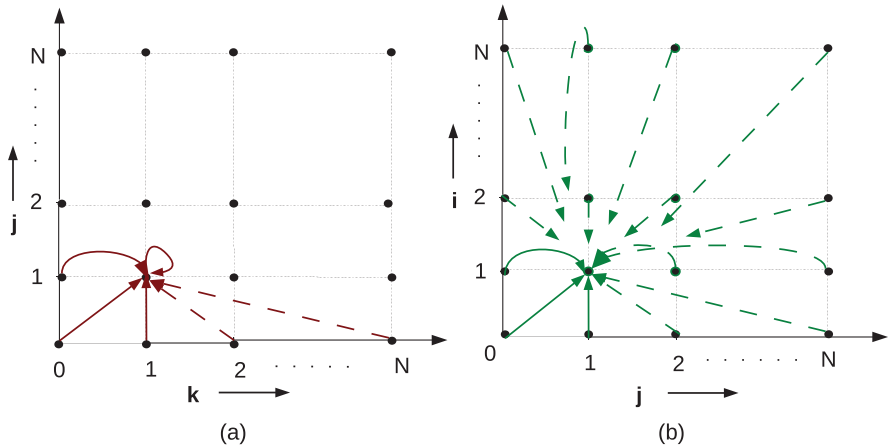

Fig. 3. Instancewise *memory-based* RAW dependences between statements (a) S1 and S2 and (b) S2 and S8 of Figure 1(a); the dashed arrows in the figure indicate a backward (RAW) dependence.

of analysis. We refer to this form of dependence analysis as "lastwriter" through this article. The lastwriter Read After Write (RAW) dependences are the same as dataflow dependences. For WAR (, WAW) dependences, there is a dependence only between a write and the last preceding read (, write) to the same element. That is, an intermediate write to the same element nullifies the WAR (, WAW) between the previous read (, write) and a following write all to the same element. In this section, the figures and the accompanying explanatory text assumes *memory-based* dependence analysis, but we also explain how the concepts apply to the *lastwriter* dependence analysis.

Figure 3 shows the (memory-based) dependences between specific statement instances involving temporary variables to demonstrate the transformation limiting nature of such dependences. Since the same memory locations are accessed in multiple iterations of the loop nest, backward dependences (i.e., dependences with negative dependence distance) are introduced in multiple loops of the loop nest. For example, consider the read to the temporary scalar variable a in Statement S2 in Figure 1(a) at the instance (j1=1, k1=1); it is the sink of a RAW dependence from not just the instances (j1=1, k1=0) and (j1=1, k1=1) but also from instances (j1=0, k1=0 .. N-1) of statement S1 as shown in Figure 3(a). Thus, it leads to loop-carried dependence in not just loop $k1$ but also loop $j1$.

It is important to note that the dependence distance for the dependences involving the scalar a whose source instance is (j1=0, 1<k1<N) is negative along loop $k1$ (i.e., they are backward dependences), and such dependences are thus marked by dashed arrows in Figure 3 for emphasis. Since loop interchange requires all dependences to be either loop independent or forward directed on the involved loops, interchanging loops $j1$ and $k1$ is rendered infeasible in this case in the presence of the above-mentioned loop-carried dependences. When using the *lastwriter* dependence analysis (that only reports the dependence between a read to the first write to prevent redundant WAR dependences), a dependence between the read instance (j1, k1=Nz-1) of S2 and the write instance (j1+1, k1=0) of S1 involving the scalar a is reported. This is again a backward dependence, which prevents the interchange of loops $j1$ and $k1$ as in the case of memory-based dependence analysis. The interchange between loops $i1$ and $j1$ is prevented for similar backward dependences occurring with both forms of dependence analysis.

We next show how dependences on temporary variables in different loop nests preclude fusion of the involved loop nests. Figure 3(b) shows the instancewise dependences between statements S2 and S8 involving the temporary array variable, *tmp*. Since S2 and S8 are in different nests, for a given $k = k1 = k2$, every write to *tmp*[k1] in S2 will be visible to the read *tmp*[k2] in S8 since the same memory location is involved each time. Thus, there is a RAW dependence from the definition in the first loop nest to the use in the second for every instance of loops i and j. This is depicted in Figure 3(b) for the instance (i=1, j=1) in the second loop nest, which becomes the sink for RAW dependences whose sources (0≤i≤N-1, 0≤j≤N-1) lie in the first loop nest. We call the dependence polyhedron capturing these RAW as being *all-to-all* in loops i and j, since it represents dependences from all iterations of these loops in the source-nest to all iterations of the corresponding loop in the destination-nest. Similarly, we can see that the polyhedron capturing the dependences involving scalar a is all-to-all in loop k as shown in Figure 3(a). Thus, if loop i or loop j were fused for both nests, then this would lead to backward (negative distance) dependences in loop i or loop j. Such a fusion will be invalid, and the compiler restricts itself from performing it.

Similarly, the presence of temporary scalar variables in different nests leads to all-to-all dependence polyhedra on all loops in the loop nest, which proves to be similarly fusion restricting.

When using the *lastwriter* dependence analysis, there is a WAR dependence from the last read to any temporary variable in the first loop nest to its first write in the second loop nest. For example, there is a dependence between the read instance (i1=Nx-1, j1=Ny-1, k1=Nz-1) of S3 and the write instance (i2=0, j2=0, k2=0) of S5 involving the temporary scalar a. This is again a backward dependence that prevents fusion of the two loop nests as in the case of memory-based dependence analysis.

3.1. Some Key Concepts and Definitions

We next describe some key concepts and definitions to aid the understanding of our approach to loop fusion by selectively removing dependences involving temporary variables.

3.1.1. Live Ranges. In the context of polyhedral compilers, a live range is appropriately defined in terms of precise statement instances [Baghdadi et al. 2013] instead of static statements in the program, as studied in earlier literature [Hack et al. 2006]. We define live ranges as in Baghdadi et al. [2013]. A live range of a value is the range of statement instances between its definition instance and any use (Baghdadi et al. [2013] explain that they consider all uses instead of the last use since loop transformations may change the last use instance). However, since any source-sink pair of references can lead to as many unique live ranges as the number of iterations in enclosing loops, we represent live ranges generated by any given source-sink pair in terms of live-range "classes." For example, the live-range "class," which the live range of the scalar a defined in the first loop nest of the example program in Figure 1(a) belongs to, is:

$$[\text{S1(i, j, k)} \rightarrow \text{S3(i, j, k)}], \text{ s.t. } 0 \leq \text{i} < \text{Nx}, 0 \leq \text{j} < \text{Ny}, 0 \leq \text{k} < \text{Nz}.$$

The above notation implies that there is a live range that begins at the write in statement S1 in every iteration of the loop nest (comprising loops i, j and k) and lasts until statement S3 in the very same iteration. In terms of dependence and dependence polyhedron, each individual live range corresponds to a RAW dependence and a live-range class corresponds to the dependence polyhedron associated with the RAW dependence edge in the data dependence graph.

3.1.2. Iteration-Private Live Ranges. For the above live range, we note that the live range begins and ends in the same iteration of the innermost loop k of the loop nest. We call

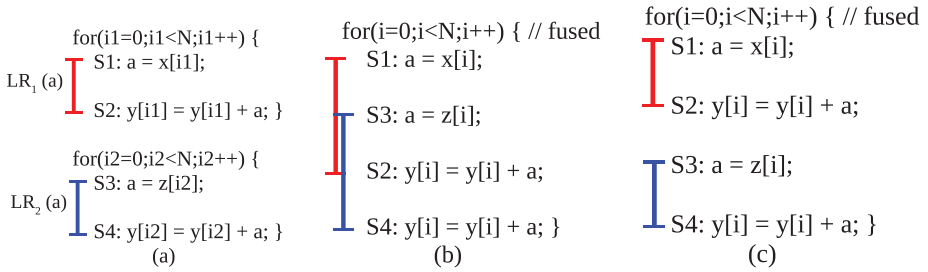

Fig. 4. Live-range interference.

such a live range as *iteration private* in loop k. Intuitively, the same live range is also iteration private in the outer loops, i and j. Past work [Maydan et al. 1993] has shown that if all live ranges are iteration private in a loop, then the temporary variables can be marked as private in that loop. In other words, the criteria for privatization requires that there be no loop-carried dependences (both true and false dependences) at the loop level concerned. For example, the live range for the scalar a in our example program in Figure 1(a) is iteration private in all three loops, and, hence, a can be marked as private in all three loops. However, we only mark a as private in only the outermost loop (as noted in Section 2) since that allows us to benefit from coarse-grained parallelization, an important use-case of privatization. We use this concept of iteration-private live ranges in our proposed variable liberalization optimization.

3.1.3. Live Ranges and Loop Fusion. As a result of loop fusion (after a possible selective removal of dependences involving temporary variables), loop bodies of the fused nests merge. Consequently, multiple definitions and uses of temporaries with the same name end up in the same loop. Thus, to ensure validity of fusion, each use must see the same definition as in the original program, or, in other words, *fusion of loop nests should preserve non-interference of live ranges*. For example, the program in Figure 4(a) shows two live ranges for the scalar variable a in the two loop nests. Figure 4(b) shows a fused nest where the two live ranges interfere with each other as shown, and the first use of the scalar a does not see the same definition as in the original program; it is thus an incorrectly transformed program. This interference results because all dependences (including the anti-dependence between statements S2 and S3 that could prevent the reordering of S2 and S3) are removed in Figure 4(b). *Thus, dependences between temporary variables in different nests cannot be all removed or else an incorrect transformation as in Figure 4(b) may result.* Figure 4(c) preserves the non-interference of live ranges and is thus a correctly transformed program. Therefore, any removal of dependences must preserve this non-interference of live ranges to ensure correctness. *This (1) non-interference of live ranges and (2) preservation of dataflow dependences form the criteria for validity of program transformation* as proved in previous work [Trifunovic et al. 2011; Baghdadi et al. 2013]. We use this criteria to reason about correctness of the transformed program in the wake of our proposed refinement (selective removal) of dependences involving (just) temporary variables.

4. OUR APPROACH

Our approach of achieving a valid fusion by removing the extra-stringent dependences on temporary variables is based on the following key insight:

4.1. Key Insight

The temporary variables, by virtue of their functionality of storing partial results temporarily in a program, are mostly defined in one of the inner loops of loop nests and

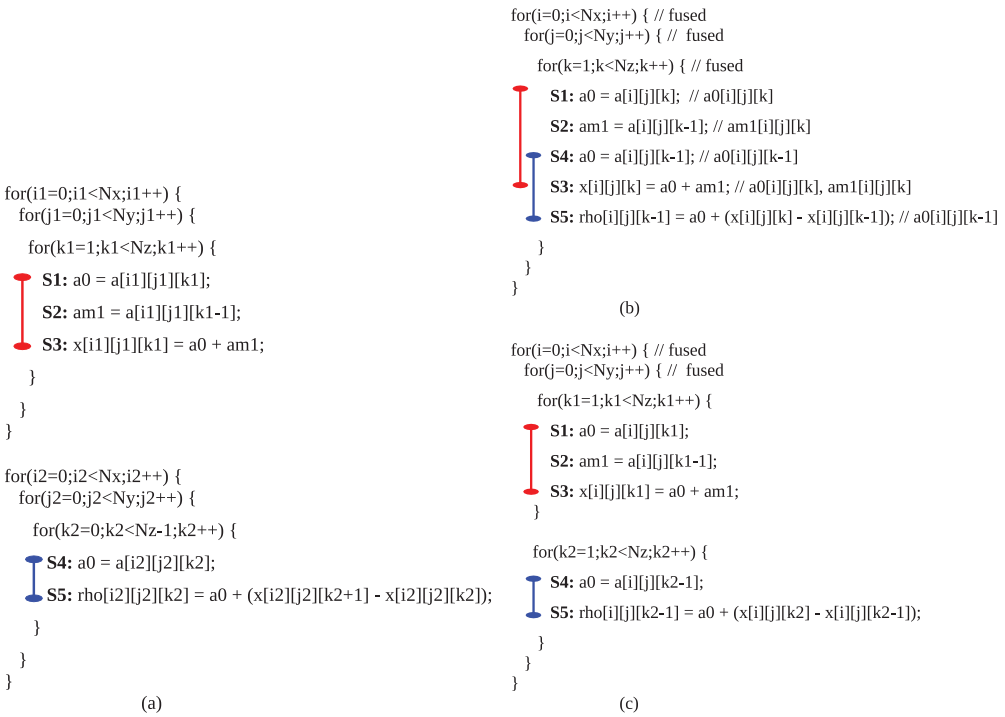

Fig. 5. (a) Example program, (b) (incorrectly) transformed program after dependence relaxation, and (c) correctly transformed program.

are used in the same loop. In other words, they are iteration private in one of the inner loops and, consequently, in all outer loops. For example, in the program in Figure 1(a) that has a three-level loop nest, the temporary scalar a is iteration private in the innermost loop k (and also loops i and j). The temporary 1D array, *tmp*, is iteration private in the next-to-innermost loop j (and loop i). If there was a temporary 2D array, then it would be iteration private in the outermost loop i.

In the event of fusion of loop nests containing temporary variables with the same name, interference of live ranges is only possible at loops within and including the innermost loop with iteration-private live ranges. Thus, *the dependences on the outer loops can be removed to allow fusion* of outer loops up to the innermost loop with iteration-private live ranges. This "removal" of dependences implies that the transformation-restricting all-to-all dependence polyhedra in outer loops are converted into *one-to-one* polyhedra (i.e., every instance or iteration of the source statement in the source nest depends on the exact same instance of the destination statement in the destination nest) as if the polyhedra now represents dependences between fully expanded arrays and not temporaries. Effectively, this is expansion and releases the necessary degrees of freedom for loop transformations. It is important to note that despite the removal of certain dependences, there still exist both RAW and one-to-one WAR dependences at the innermost loop with iteration-private live ranges. In particular, the one-to-one WAR dependence between the sink of a live range to the source of a following live range in the fused loop body preserves the non-interference of live ranges and hence program correctness. For example, the existence of a one-to-one (or, in this case, loop-independent) WAR dependence between S3 and S4 in Figure 5(c) ensures correctness.

With our proposed dependence refinement, loop fusion is enabled by converting all-to-all dependence polyhedra into one-to-one polyhedra. However, a polyhedral compiler being capable of composing multiple loop transformations simultaneously may perform other transformations such as loop shifting, skewing,[2] interchange, and reversal (enabled recently in Pluto+ [Acharya and Bondhugula 2015]). For correctness, therefore, it is important that these transformations do not cause live-range interference. We observe that, apart from shifting, these transformations when composed with fusion do not change the one-to-one WAR dependence at the innermost loop with iteration-private live ranges. This WAR dependence on fusion becomes a loop-independent dependence and ensures the non-interference of live ranges. However, in the event of loop shifting (alongside fusion), this WAR dependence may be converted to a loop-carried dependence and allow interference of live ranges. This is explained through the following example.

Figure 5(b) shows a transformed program with fused nests—note that statements S3 and S4 are reordered, which is a correct reordering under the proposed refinement. In this example, the cross-nest all-to-all dependence polyhedra are converted to one-to-one polyhedra to enable fusion. The net effect of this is that from the compiler's perspective, the scalar variables are expanded to become 3D arrays like other arrays such as x in the loop nest. This is shown in the comments at the end of each statement.

In this example program, although we remove dependences to enable loop fusion, fusion is ultimately accomplished by the compiler with the help of shifting as an enabling transformation. This is because the statements in the second nest have to be shifted by one iteration in the innermost loop (k) to prevent backward dependences on array x in loop k. As a result of shifting, the (possible) backward dependence on array x is converted into a forward dependence in the transformed program. Thus, we see that there is a forward loop-carried WAR dependence from S3 to S4 on the scalar $a0$ in loop k (i.e., $a0[i][j][k]$ read in statement S3 is written in the next iteration of the k-loop in statement S4) and a forward loop-carried RAW dependence between statements S1 and S5 on the same scalar $a0$ in loop k (i.e., $a0[i][j][k]$ written in statement S1 is read in the next iteration of the k-loop in statement S5). As a result, the schedule of statements within the fused nest as shown in Figure 5(b) is completely valid from the point-of-view of data dependences (some of which have been purposely removed). However, the transformed program in Figure 5(b) is incorrect because the non-interference of live-ranges property is violated—the live ranges for the two def-use pairs involving the scalar $a0$ interfere with each other as shown. This happened precisely because shifting converts the WAR dependence between statements S3 and S4 to a loop-carried dependence in the fused loop. The same is possible for the other loops in the nest. *Thus, the sufficient condition to preserve non-interference of live ranges and hence correctness of the transformed program when removing dependences is that the loops participating in fusion should not be shifted with respect to each other.* We next discuss exactly which dependences are removed in variable liberalization and how we ensure that the above-mentioned sufficient condition is maintained for correctness.

4.2. Dependence Refinement

Dependencies involving temporary variables are of two kinds—those between statements that belong to the same SCC (at the outermost loop) and those between statements belonging to different SCCs (different loop nests). As discussed earlier in Section 3, dependences of the first kind (such as the RAW dependence between statement

[2]It is important to note that, in general, those loops participate in skewing that have loop-carried dependences (such as the time-tilable stencils where there is reuse and hence loop-carried dependences in the outermost time loop), and thus those loops do not contain iteration-private live ranges. Such loops are, therefore, not affected by variable liberalization.

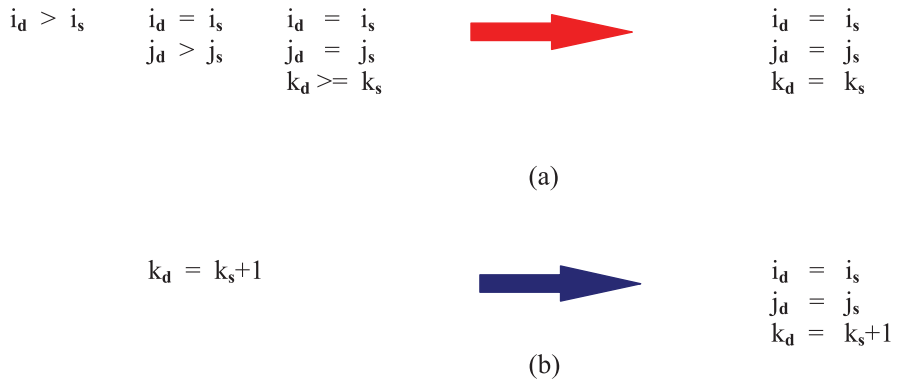

Fig. 6. Dependence refinement when statements belong to (a) the same SCC (statements S1 and S2) and (b) different SCCs (statements S2 and S8) in Figure 1(a).

S1 and S2 in Figure 1(a) involving the scalar variable a) prevent loop interchange which is an enabling transformation for fusion. Dependencies of the second kind (such as the RAW dependence between statements S2 and S8 in Figure 1(a) involving the temporary variable *tmp*, where S2 and S8 belong to two different SCCs) prevent fusion of the involved loop nests.

In variable liberalization, we (selectively) remove both of these kinds of dependences on temporary variables to enable more transformations for improved performance, particularly, fusion, interchange, and parallelization (coarse and fine grained). This is done in both cases as follows.

The dependences between statements S1 and S2 in Figure 1(a) that both belong to the same SCC is shown in Figure 6(a) by three sets of equalities/inequalities called dependence polyhedra (note that inequalities capturing loop bounds have been ignored). Note that the dependences representing these polyhedra were shown earlier in Section 3. Clearly, these involve loop-carried backward edges preventing interchange. In liberalization, we remove all loop-carried dependences in loops with iteration-private live ranges (loops i, j, and k, in this case) as shown in the dependence polyhedron on the right in Figure 6(a). This refinement is feasible since the source and destination statements in a dependence involving a temporary variable often belong to the same SCC in the loops[3] that have iteration-private live ranges (we explicitly check for this in the implementation). For example, statements S1 and S2 belong to the same SCC in loops i, j, and k. This ensures that statements S1 and S2 will not be shifted with respect to each other in these loops since the presence of cyclical dependences involving such statements (in an SCC) prevent any relative shifting in the loops concerned. This is sufficient to guarantee the correctness of liberalization. Similarly, when using the *lastwriter* dependence analysis, the WAR dependences between statements S2 and S1 in Figure 1(a) are removed since they are all loop-carried in loops i, j, and k. Furthermore, the existence of the loop-independent RAW and WAR dependences ensures the non-interference of live ranges within the SCC.

The dependences between statements S2 and S8 that both belong to different SCCs is shown in Figure 6(b) by a dependence polyhedron. The backward dependence edges in this all-to-all dependence polyhedron prevent fusion of the two loops containing S2

[3]Note that we use the terminology, "SCC in loop i" to denote SCCs computed at loop level i. This is because an SCC is always defined on a per-loop basis and not on a per-loop-nest basis, and after each loop hyperplane (or a band of hyperplanes) is found, SCCs in the SCoP are recomputed since some dependence edges now disappear from the dependence graph.

```
for(i1=0;i1<Nx;i1++)
  for(j1=0;j1<Ny;j1++)
    for(k1=0;k1<Nz;k1++)
      tmp = tmp + a[i1][j1][k1];

for(i2=0;i2<Nx;i2++)
  for(j2=0;j2<Ny;j2++)
    for(k2=0;k2<Nz;k2++)
      tmp = tmp + b[i2][j2][k2];
```

(a)

```
for(i=0;i<Nx;i++)
  for(j=0;j<Ny;j++)
    a[i] += b[i][j] + c[i][j];
```

(b)

Fig. 7. Example programs where dependence removal is infeasible.

and S8 as explained earlier in Section 3. As in the first case, we remove the dependence edges on the loops outside of and including the innermost loop with iteration-private live ranges. We convert the all-to-all dependence polyhedron to a one-to-one polyhedron as shown on the right in Figure 6(b). The same approach is applicable when *lastwriter* dependence analysis is used. Unlike the first case, however, statements S2 and S8 do not belong to the same SCC and thus correctness in the wake of this dependence removal is contingent on the non-occurrence of loop shifting in loops i and j. We ensure this in our framework by *forcing such statements as S2 and S8 to have the same transformation* (i.e., no shifting with respect to each other) if they are fused together by the compiler. That is, if the scheduler finds different transformations for any of the loops outside of and including the innermost loop with iteration-private live ranges (since we only consider loops with iteration-private live ranges, the same approach applies to both perfect and imperfect nests), then we force a distribution of the two SCCs that contain the two statements into different loop nests. This loop distribution was not needed to generate the correct fused code for Figure 1(a) (as shown in Figure 1(b)) but was needed to obtain the correct fused code for Figure 5(a) (as shown in Figure 5(c)) where loop shifting was involved in loop k.[4] Furthermore, the existence of loop-independent WAR dependence between statements S4 and S6 in Figure 1(b) (and between statements S3 and S4 in Figure 5(c)) ensures the non-interference of live ranges within the fused loop nest. Thus, we see that this dependence refinement by selective removal of dependences in variable liberalization satisfies the sufficient condition for non-interference of live ranges and hence program correctness.

4.3. Cases in Which Dependence Refinement Is Not Feasible

In this section, we mention two examples where dependences on temporary variables in different loop nests cannot be removed. In particular, these examples show example cases where scalars or 1D arrays may be used but their live range is not iteration private, that is, they are are used as temporary variables in these cases. As a result, it is not safe to remove dependences through our proposed variable liberalization.

Figure 7(a) shows the first example that contains a temporary scalar variable *tmp* in the two loop nests. Removing dependences on the outer loops i and j would result in fusion of the two nests, but such a fusion is clearly invalid and our framework desists from performing it. The reason is that the live range involving *tmp* is not iteration private in the outer loops of the loop nest, that is, a value written in *tmp* in each iteration of loop j is read in the next iteration of loop j. The same holds true for the outermost loop i as well. The net result is that *tmp* is live-in in the second nest, which clearly indicates the infeasibility of loop fusion in this case.

[4]It is important to note that although a perfect fusion could be achieved in this case, loop distribution is helpful since it aids inner-loop vectorization. In such cases, one could also wait until the entire schedule is computed to actually witness live-range interference and only then choose to distribute the loops, but it would require re-computation of the entire schedule.

Figure 7(b) shows the second example. The subscript of the temporary array a is the outermost loop variable i as opposed to the previous examples where temporary array's subscript is an innermost loop variable. Clearly, the live range of a is not iteration private in any loop in the loop nest, which also indicates its use later in the program. It is therefore not marked for *liberalization*. Also, in such a case, liberalization is not needed to allow fusion (and parallelism) of the outer loop since there are no backward dependences (represented by an all-to-all dependence polyhedron) on the outer loop.

5. IMPLEMENTATION: PUTTING IT ALL TOGETHER

This section describes the algorithm used to implement variable liberalization. Our proposed dependence refinement involving temporary variables as discussed in the previous sections relies on determining loops with iteration-private live ranges so selective dependences in such loops could be identified for removal. For this purpose, we use an array, *i-p_depth*, that stores the depth of the innermost loop with iteration-private live ranges for all SCCs in the program. Since we apply Algorithm 1 only once before the scheduling phase in Pluto, the SCCs at this point are those computed at the outermost loop level.

As discussed in Section 3.1.1, a live range is the same as a dataflow or a RAW dependence (in *lastwriter*). Thus, we know that a live range (and, therefore, the live-range class) is iteration private in a given loop if the corresponding RAW dependence (or dependence polyhedron) is loop independent in that loop. We thus obtain the depth of the innermost loop with iteration-private live ranges in Step 1 of the algorithm. However, since different temporary variables may have live ranges that are iteration private up to different depths in an SCC, we take the minimum depth over all live ranges. This ensures that when dependences on such loops with iteration-private live ranges are removed, no useful dependences are removed.

In Step 2, we remove the extra-stringent dependences. In Step 2a, we remove those dependences involving temporary variables whose source and destination statements belong to the same SCC (again, at the outermost loop level). In this case, as discussed in Section 4, we prune the dependences that are loop-carried on loops outside of and including the innermost loop with iteration-private live ranges. Furthermore, only those dependences are pruned whose source and destination statements belong to the same SCC in all such outer loops (as indicated in lines 21–27 in the algorithm). This ensures that the removal of those dependences will not result in different schedules (and consequently, no loop shifting) for the statements involved, in the concerned outer-loops. In our algorithm, we consult the original program schedule (and the SCCs computed thereof at each loop level for the original program schedule) to determine all the outer loop levels in which the source and destination statements of a given dependence belong to the same SCC.

Step 2b results in removing dependences involving temporary variables whose source and destination statements belong to different SCCs. In this case, as discussed in Section 4, we explicitly add equalities in the dependence polyhedron (representing dependences involving a temporary variable across SCCs) to convert it from all-to-all to one-to-one. In other words, all dependences that were loop-carried (including backward dependences) in loops outside of and including the innermost loop with iteration-private live ranges are pruned, allowing for fusion of two nests.

5.1. Validation of Relaxation Criteria

In polyhedral compilation, violated dependence analysis [Vasilache et al. 2006] has been proposed in the past. Using violated dependence analysis, the polyhedral compiler can reason the correctness of the proposed transformation in the wake of either relaxed validity checks [Vasilache et al. 2007] or reduced dependences [Trifunovic et al.

ALGORITHM 1: Variable Liberalization

INPUT:
deps : Copy of the list of dependence polyhedra
SCCs : List of SCCs at the outermost loop (i.e., before scheduling begins)

STEP 1:
for each $scc \in SCCs$ **do**
  $i\text{-}p\_depth[scc] = -1$
for each dependence polyhedron $d \in deps$ s.t. ($d.src\_scc\_id{=}d.dest\_scc\_id$) $\wedge$ IsRAW(d.type) **do**
  Set *depth* to 0
  for each loop l in $d.src\_scc\_id$ starting from the outermost loop **do**
    if d is loop-independent in l **then**
      Increment *depth*
  if $depth < i\text{-}p\_depth[d.src\_scc\_id] \vee i\text{-}p\_depth[d.src\_scc\_id]{=}-1$ **then**
    $i\text{-}p\_depth[d.src\_scc\_id]{=}depth$

STEP 2a:
for each dependence polyhedron $d \in deps$ s.t. $d.src\_scc\_id{=}d.dest\_scc\_id$ **do**
  Set *depth* to 0
  for each loop $l \in d.src\_scc\_id$ starting from the outermost loop **do**
    if d is loop-carried in l **then**
      Break out of the for-loop
    Increment *depth*
  if $depth < i\text{-}p\_depth[d.src\_scc\_id]$ **then**
    Set *loop_level* to 0
    for each hyperplane h in the original program schedule **do**
      if IsTYPE($d.src$, h) = H_LOOP **then**
        Increment *loop_level*
        if !IsSameOrigSCC($d.src$, $d.dest$, h) $\wedge$ ($depth < loop\_level$) **then**
          Remove d from *deps*
          Break out of the for-loop

STEP 2b:
for each dependence $d \in deps$ s.t. ($d.src\_scc\_id$!= $d.dest\_scc\_id$) $\wedge$ ($i\text{-}p\_depth[d.src\_scc\_id] > 0$) $\wedge$ ($i\text{-}p\_depth[d.dest\_scc\_id] > 0$) **do**
  for each loop l in $d.src\_scc\_id$ starting from the outermost loop upto a depth given by $min(i\text{-}p\_depth[d.src\_scc\_id], i\text{-}p\_depth[d.dest\_scc\_id])$ **do**
    Add an equality in d to convert d from being all-to-all in loop l to one-to-one

OUTPUT: *Refined set of dependence polyhedra, deps*

2011; Vasilache et al. 2012]. In such an approach, one has to wait until a transformation is performed and re-iterate the process possibly several times in the case of an incorrect transformation or take a corrective action. In this work, we introduce *violated transformation analysis*, where we can reason about the correctness of the computed transformation during the process of finding transformation itself, and the recovery is executed immediately to yield a correct (although weaker) transformation.

As discussed in Section 4, we can reason about the validity of dependence refinement by checking for the satisfaction of the *sufficient* condition. The sufficient condition to validate dependence refinement states that the loops participating in fusion should not be shifted with respect to each other during the transformation. In the polyhedral framework, the transformation process consists of finding a schedule row at a time. This schedule row determines the transformation performed on each statement at that

Table I. Summary of Fusion Models in Different Compilers (Note That *Ifort* Is Used as the Backend Compiler with the "Smartfuse," "Explicit," and "Var-Lib" Models

Fusion Model	Description
gfortran	GNU Fortran Compiler (baseline); flags = '-O3 -ftree-loop-parallelize=n'
ifort	The Intel Fortran Compiler; flags = '-O3 -parallel'
smartfuse	The default fusion model in Pluto.
	It uses heuristics to determine a good fusion schedule
explicit	Explicit parallelization or parallelization by hand
var-lib	Variable liberalization in Pluto (our work)

loop level.[5] In variable liberalization, in order to satisfy the sufficient condition, we check if all source and destination statements of dependences involving temporary variables across SCCs undergo the same transformation. In other words, we check for the occurrence of loop shifting in any of the common (fused) loops. If we find that any such source and destination statements undergo different transformation at a particular loop level, then we discard (only) that schedule row and force distribution of the SCCs containing the respective source and destination statements. Thus, this ensures that the sufficient condition is held and the correctness is guaranteed without the need to re-iterate the process.

6. EXPERIMENTAL EVALUATION

6.1. Setup

The test programs were compiled and run on an Intel Xeon processor (E5-2650 v2) with eight Sandy Bridge-EP cores operating at 2.0GHz. The processor has private L1 (32KB per core) and L2 (256KB per core) caches and a 20MB shared L3 cache and 16GB DDR3 memory. Since we implement the variable liberalization optimization in the Pluto polyhedral compiler, we compare our performance results with the fusion model within Pluto, in addition to the popular production compilers, the GNU and the Intel compilers. A summary of the different fusion models used for comparison is given in Table I. It is important to note that Pluto itself cannot parse Fortan code (and the applications we use for experiments are written in Fortran). We thus use PolyOpt/Fortran [OSU 2012], a tool that uses a ROSE compiler [LLNL 2015] frontend to parse Fortran code and relies on Pluto (version 0.5.4) for loop optimizations. We have also tested the C versions of most of these applications with our variable liberalization optimization implemented in the latest version of Pluto (version 0.11.4) with the options '–lastwriter –parallel/–tile' (note that we do not rely on scalar privatization performed by the dependence analyzer). It is now available as a git branch of Pluto, called *scalable-fuse*. The transformed source code generated is then compiled using the Intel compiler v14 (ifort or icc) as the backend compiler. The compile time options used with the Intel compiler are '-O3' and '-parallel'.

6.2. Benchmarks

The experiments were run on five real application programs used in the scientific community and in other published research. A brief description of these programs is given in Table II; we choose these applications because we identify them to contain opportunity for significant data reuse through loop fusion. In these five applications, we identify nine hot regions that form the computationally intensive portions of the application. Each identified hot region constitutes a Static Control Part (SCoP), or the maximal syntactic program segment that contains sequences of loop nests with

[5]More details on the process of computing transformations using a polyhedral compiler can be found in Bondhugula et al. [2008].

Table II. Summary of the Benchmarks

Benchmark	Benchmark Suite	Category	Problem Size
applu	NPB/OMP2012	Computational Fluid Dynamics (CFD)	N=102; CLASS B
bt	NPB/OMP2012	”	”
sp	NPB	”	”
zeusmp	CPU2006	Simulation of astrophysical phenomena	Reference Input
swim	OMP2012	Weather prediction	Reference Input

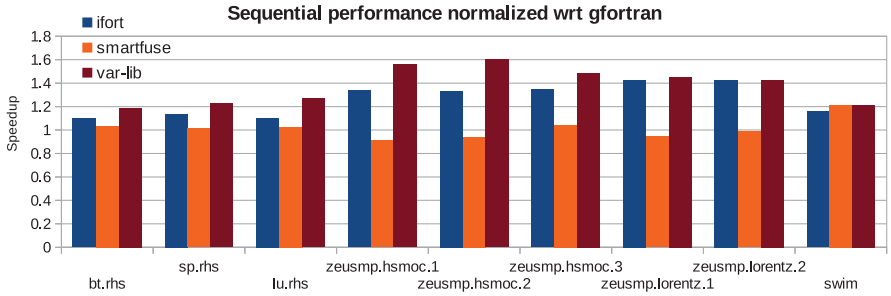

Fig. 8. Sequential kernel speedups wrt gfortran.

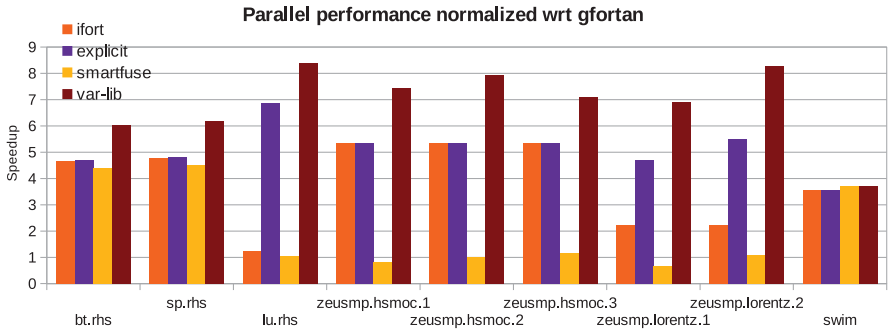

Fig. 9. Parallel kernel speedups wrt gfortran (using 8 cores).

constant strides and affine bounds. As a result, all chosen hot regions are amenable for optimizations by a polyhedral compiler (other applications from these suites were either not amenable to polyhedral optimizations or did not contain opportunity to benefit from fusion). It is important to note that each of these nine SCoPs contain multiple large loop nests with the number of statements in each SCoP ranging from 48 to 121. Such large sequences of statements are known to be hard for the compilers to optimize. In particular, previous work has not shown the strength of polyhedral model on such large SCoPs.

6.3. Results and Discussion

Figures 8 and 9 show the performance results for the nine SCoPs using different compilers for comparison. Figure 8 shows results for sequential performance, while Figure 9 shows parallel performance. Among the five benchmarks, the entire *compute_rhs* subroutine in *bt*, *sp*, and *lu* benchmarks forms a single SCoP, the *hsmoc* subroutine within *zeusmp* benchmark consists of three SCoPs, each separated by a procedure call (with possible side-effects and thus recognized as a non-affine component), whereas the *lorentz* subroutine consists of a single SCoP but divided into two in order to limit the memory requirement for optimizing it using polyhedral compilation. In *swim*, the

three large loops that contain the bulk of the computation together form a single SCoP. Each of the chosen benchmarks contains multiple loop nests and thus offers considerable opportunities for loop optimizations—a characteristic of most scientific application codes.

In particular, some of the chosen benchmarks contains loop nests with different loop-orders and thus are more challenging from the point of view of the loop fusion optimization. From the figures, we can find that *var-lib*, that is, the compiler optimization proposed in this work, outperforms the other compilers in almost all cases. The sequential performance of *var-lib* outperforms *ifort* by as much as 1.2×, and the performance improvement is significantly larger for parallel versions of the benchmarks with *var-lib* outperforming *ifort* by as much as 6.8× in *lu*. Even against the explicitly parallelized versions, *var-lib* performs considerably better, as shown in Figure 9. We next discuss the performance results for each compiler in more detail.

6.3.1. gfortran and ifort. *gfortran* or the GNU Fortran compiler was chosen as the baseline in our experiments. In addition, we also show results with the Intel Fortan compiler. In all cases, *gfortran* proved to be worse than *ifort* due to the latter being much more effective at vectorization. However, both these production compilers are equally poor in performing loop fusion. As a result, neither of them fused any large loop nests for any of the SCoPs listed in the figure. *gfortran* performed the worst of all compilers because it could not recognize parallel loops in any benchmark. *ifort* could recognize parallel loops in *bt.rhs*, *sp.rhs*, *zeusmp.hsmoc*, *zeusmp.lorentz.1*, and *swim*.

From these results (and other experiments using our test kernels), we find that *ifort* is capable of parallelizing the loop nest in the presence of temporary scalar variables but not in the presence of temporary array variables. It is for this reason that *ifort* could not parallelize *lu*. Although temporary array variables exist in *hsmoc* as well, we believe that *ifort* relies on recognizing certain specific patterns in this case to achieve parallelization because the inability to recognize temporary array variables in a computationally less intensive subroutine, *lorentz*, in the same benchmark hurts parallelization opportunity. In any case, we can conclude that existing production compilers are limited in their capability of detecting parallelism in scientific application codes that contain temporary variables and much more so in performing the important loop fusion optimization.

6.3.2. Pluto's Smartfuse. Pluto is a state-of-the-art polyhedral compiler that has shown significant promise in achieving automatic loop parallelization [Bondhugula et al. 2008; Mehta et al. 2014]. Pluto uses three different heuristics for fusion, *min*-, *max*-, and *smartfuse*. *maxfuse* and *smartfuse* are practically equivalent for SCoPs with many statements, and *minfuse* performs maximal distribution and almost always performs sub-optimally. Thus, we only show results for *smartfuse*. Pluto is also capable of performing scalar privatization that empowers it to perform coarse-grained parallelization in the presence of scalar temporary variables. It cannot, however, privatize temporary array variables. As a result, *smartfuse* can identify parallel loops in *bt*, *sp*, and *swim* but not in any of the other SCoPs that contain temporary array variables. Also, since Pluto does not remove any dependences (or perform variable liberalization) on such temporary variables across loop nests, it is deprived of the opportunity to perform loop fusion. This amounts to reduced performance as compared to *var-lib* even for the benchmarks where it could achieve parallelization.

6.3.3. Variable Liberalization (Var-Lib). When using a single thread to run benchmarks, *var-lib* outperforms all other compilers on account of fusing multiple loop nests. The improvement is proportional to the fusion opportunity available in the benchmarks. For example, both the large nests in *zeusmp.hsmoc.2* are fused to enable data reuse,

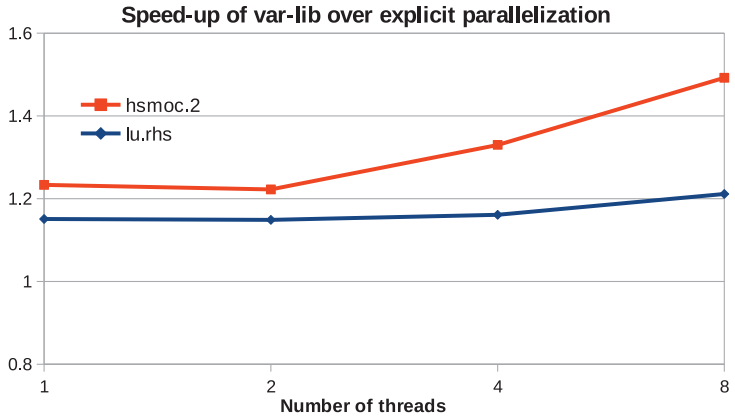

Fig. 10. Parallel kernel speed-up of *var-lib* over *explicit parallelization*.

Table III. Performance Counters Indicating Reduced Pipeline Stalls and Effective Memory Latency through *Var-Lib*

	$\frac{\text{explicit-varLib}}{\text{explicit}} * 100(\%)$	
Hardware Event	lu	zeusmp
Load_latency_gt_512	24.03	12.88
Resource_stalls.any	7.68	13.8
Uops_dispatched.stall_cycles	8.8	19.94

whereas in the other cases, such as *lu*, only two of the three large nests could be fused together. This is because the common outer loop of the first two nests in *lu* is the innermost loop of the third nest, and the innermost loop cannot participate in loop interchange because of the presence of temporary arrays and imperfect nesting. It is important to note that although, unlike other benchmarks, *swim* does not contain temporary variables and does not thus benefit from *var-lib*, *var-lib* does not hurt its performance either, when compared to *smartfuse*. In other words, the optimization does not hurt when not applicable. This is because *var-lib* does not modify any dependencies in the original program other than those on temporary variables.

Interestingly, the performance improvement for all benchmarks surges upon parallelization even for benchmarks that are successfully parallelized by other compilers (including explicitly or manually parallelized code) such as *bt*, *sp*, and *zeusmp.hsmoc*. This is due to two reasons: (1) reduction of fork-join synchronization points and, more importantly, (2) saving of off-chip memory accesses, and thus the bandwidth, which is a source of contention among parallel threads in such memory-intensive applications. Both of these benefits are direct consequences of effective loop fusion achieved as a result of variable liberalization. In addition, *var-lib* significantly outperforms all other compilers when they cannot identify outer-parallel loops due to the presence of temporary array variables and imperfect nests as in *lu* and *zeusmp.lorentz*.

Figure 10 shows the speedups achieved by *var-lib* over *explicitly* parallelized code for two of the eight SCoPs, *lu.rhs* and *zeusmp.hsmoc.2*, when using different numbers of threads. The speedup achieved in the case of *zeusmp.hsmoc.2* is larger than that in *lu.rhs*. This is because of a greater opportunity for fusion in the former case, as explained earlier. Furthermore, the speedup achieved increases with the increase in the number of threads. This is explained with the help of performance counters shown in Table III as obtained from Intel's VTune Performance Profiler [Intel 2015].

Table IV. Overall (Application) Speedup (S) Achieved by *Var-Lib (+ Ifort)* Over *Ifort*

Benchmark	Subroutine	# statements	# deps	% Ex. time	S = $\frac{\text{Ex. Time}_{\text{ifort}}}{\text{Ex. Time}_{\text{var-lib}}}$
bt	rhs	48	1419	16.8	1.06x
sp	rhs	50	1428	33.7	1.08x
lu	rhs	106	3033	63.1	2.17x
zeusmp	hsmoc.1	121	2660	24.1	1.36x
	hsmoc.2	118	2493		
	hsmoc.3	120	2296		
	lorentz.1	98	2227	27.3	
	lorentz.2	92	2149		
swim	main	52	472	100	1.05x

Both benchmarks witness a considerable reduction (24.03% and 12.88%, respectively) in the number of memory loads whose latency is greater than 512 cycles after variable liberalization. Since the memory latency on the test processor (SandyBridge microarchitecture) is roughly 200 cycles, such high latency corroborates bandwidth contention. Thus, clearly, lower number of such high latency loads confirms the efficacy of effective loop fusion performed by *var-lib*. These high-latency loads result in an increase in the number of stall cycles due to resource contention (measured by the counter *Resource_stalls.any* that includes stalls due to fully occupied load Buffer, Store Buffer, Reorder Buffer, and Reservation Stations) and also due to reservation stations waiting on operands (measured by the counter *Uops_dispatched.stall_cycles*). The reduction in loads with high latency are larger for *lu* than *zeusmp* since the optimized subroutine, *rhs*, in *lu* is its most memory-intensive part and thus benefits more from loop fusion. However, reduction in stalls is more significant in case of *zeusmp* since the optimized subroutines, *hsmoc* and *lorentz*, together contribute more than 50% of the execution time in *zeusmp*, as compared to *rhs* that contributes only 20% of the total execution time in the explicitly parallelized version of the *lu* benchmark.

Last, Table IV shows the overall application speedups achieved by *var-lib* over *ifort* for the five applications. The performance improvement depends on both, the contribution of the optimized portion to the overall application and also the opportunity for optimization. For example, the speedup ranges from 1.06× in *bt* where the optimized subroutine (*compute_rhs*) contributes only 16.8% to the execution time, to 2.17× in *lu* where the optimized subroutine contributes 63.1% to the overall execution time. On the other hand, *swim* benefits by 1.05× only because of little opportunity to improve locality through fusion.

6.4. Scope of This Work

Until recently, polyhedral compilers had been largely restricted to tiny scopes with few statements. This is primarily due to two reasons, (1) undiscovered large affine codes and (2) unscalability of the algorithms used for programs with many statements and dependences. As a result, the potential for performance improvement through a polyhedral compiler in real application codes with hundreds of lines of SCoPs had remained unknown.

This work shows that the polyhedral compilers are, in fact, useful for real applications also by showing significant improvement for such codes. To the best of our knowledge, this is the first time that results on such large SCoPs have been shown using a polyhedral compiler.

While variable liberalization opens up opportunities for loop fusion, it might hurt program performance for three reasons. They are as follows: (1) it may increase the

number of prefetch streams in the loop that the hardware cannot detect; (2) it may cause pronounced bandwidth contention; (3) it may cause pressure on the data cache. However, we argue that none of these actually prove to be detrimental in practice. This is due to the following reasons.

(1) The latest processors are equipped with hardware to detect many more streams than needed in almost all cases in practice. For example, the Intel SandyBridge can prefetch 32 streams, and we found it to be over-provisioned for all but one (*cactusADM*) benchmark in the SPEC suites. Even after aggressive loop fusion with *var-lib*, the same was observed. This is because, the fused loops use the same arrays (note that we do not do renaming) in most cases and therefore do not contribute different streams. In *cactusADM* also, even though there are 81 prefetch streams, the prefetcher proves unhelpful because of the large amount of computation on those arrays.
(2) Loop fusion, in most cases, leads to reuse of data between at least a few common arrays in the fused loops, which alleviates bandwidth contention. For example, in *rhs* subroutine of *bt*, *sp*, and *lu*, there is considerable reuse in the last-level cache since many common arrays are referenced in each loop nest. However, even in the worst-case scenario, when there are no common arrays in the fused loops, although no reuse is achieved, the amount of computation in the loop usually increases in proportion to the arrays. For example, in the *zeusmp* benchmark, there is much less reuse but we do not observe a performance degradation upon fusion, even for a single thread. This is because the memory to computation ratio in the loop does not change. Thus, fusion does not lead to pronounced bandwidth contention in most cases.
(3) Loop fusion does cause increase in pressure at the last-level cache (LLC) because of greater number of array references in the loop. As a result, reusable data may be evicted from the LLC before the program control enters the next loop. For example, in *zeusmp*, fusion definitely exerts more pressure at LLC, especially since there is not much reuse available. However, we find that the serial performance does not degrade and the parallel performance increase due to the slight reuse exploited and the reduction of the synchronization points. Also, our framework prevents loop fusion in the complete absence of data reuse, thereby further reducing chances of performance degradation upon fusion.

It is for the above reasons that we argue that *var-lib* is more generally applicable than may seem from the results on five real applications. Having said that, we concede that it would certainly be interesting to test *var-lib* on more programs, and we consider it as our on-going effort. We also hope that this work will motivate scientists to write code that is amenable to polyhedral optimizations.

7. RELATED WORK

Past work has well recognized the importance of enabling important optimizations through variable privatization and expansion. Variable privatization [Li 1992; Maydan et al. 1993; Tu and Padua 1994] involves creating multiple copies of a temporary variable (scalar or a low dimension array) that is defined and used in one of inner loops of a loop nest (i.e., has iteration-private live ranges in that loop). Consequently, each thread is assigned a private copy of the variable, thus eliminating races between threads and allowing coarse-grained parallelism at an outer loop. From a data-dependence analysis perspective, privatization involves pruning loop-carried dependencies on the temporary variables at the outermost loop. However, privatization only affects a single loop nest; the data dependencies between temporary variables in different nests continue to be stringent and, thus, transformation restricting.

While privatization is a supporting optimization for parallelism, expansion [Feautrier 1988] is another enabling optimization for many other transformations such as loop fusion, shifting, and distribution. However, it has a known drawback of significantly increasing the memory footprint. To counteract this, the authors in Lefebvre and Feautrier [1998], Quilleré and Rajopadhye [2000], and Darte and Huard [2005] propose to perform a maximal expansion to enable important optimizations and then attempt to contract the expanded variables as much as possible. However, this contraction of the optimized program is not only difficult but is sometimes not possible in the wake of transformations such as distribution that can potentially distribute the definition and use of expanded variables to different loops. In our work, we propose variable liberalization that achieves the benefits of both privatization and expansion. In liberalization, we remove dependences on temporary variables in different nests as in privatization, thus effectively expanding those variables but not through an actual expansion in their dimensions.

In the past, there has also been considerable work on loop fusion using different algorithms to find the best loops to fuse with the objective of maximizing data reuse [Megiddo and Sarkar 1997; Kennedy and McKinley 1994], minimizing synchronization [Kennedy and McKinley 1994], reducing register pressure [Singhai and McKinley 1997], and saving off-chip bandwidth [Ding and Kennedy 2004]. Interestingly, none of these techniques have been used in existing production compilers for various pragmatic reasons in addition to the increased compile time. These pragmatic reasons include loops with different bounds, orders, or those that are imperfectly nested and are hard for the compiler to optimize in real application codes. However, we show in this work that even if these limitations were removed (most of which are non-existent in polyhedral compilers due to exact dependence analysis), effective fusion and consequent parallelization could still be prohibited due to the occurrence of transformation-restricting dependences on temporary variables in different nests. Thus, this work proposes variable liberalization to fill this gap.

The authors in Trifunovic et al. [2011] and Vasilache et al. [2012] propose lazy expansion where they ignore WAW and WAR dependences when finding transformations and then reason correctness using an extension to the violated dependence analysis [Vasilache et al. 2006], called live-range violation analysis. In the event of a violation, variables are expanded as needed (lazily) to ensure correctness. Since they do not violate true (RAW) dependences, they guarantee transformation correctness, albeit at the cost of higher memory footprint. However, these works do not focus on loop fusion, which is crucial for performance in large SCoPs. In fact, since there also exist RAW dependences across loop nests that are fusion-restricting, considering them for transformations without relaxation would disallow fusion in the first place in the presence of temporary variables in different nests. It is also important to note that even partial expansion and renaming can significantly increase (potentially double with the fusion of just two nests) the working set in the higher levels of the memory hierarchy by increasing the amount of data accessed in the inner loops of the nest and thus hurt performance. In this work, we extend the violated dependence analysis to our proposed violated transformation analysis where we can reason if the transformation has gone wrong during the time it is being found and immediately take corrective measures by imposing stricter constraints. This thus prevents subsequent passes to find a correct transformation and does not require any renaming or expansion.

The work by Baghdadi et al. [2013] that prunes spurious dependences for the purpose of tiling comes closest to our work. We use the authors' idea of live-range noninterference to argue the validity of our proposed variable liberalization. However, there are important differences. In polyhedral compilers, the actual creation of dimensions corresponding to tiled loops/tiled space is implemented as a post-pass

optimization. Therefore, removing certain dependences such as the WAR dependence between two live ranges (as proposed) to enable tiling is helpful and does not hurt other optimizations. However, if such WAR dependences were completely removed to enable fusion, it may lead to incorrect code. Therefore, we only selectively remove such dependences to enable fusion. Among other polyhedral tools, Polly and Graphite are both capable of performing fusion in the presence of temporary scalars but not temporary array variables. This is because these tools map scalars to registers and not to memory at the IR level, and therefore scalars do not induce any dependences to restrict fusion. Also, Verdoolaege and Cohen [2016] have recently incorporated similar ideas as presented in this work into their PPCG compiler to achieve fusion of loop nests in the presence of temporary variables. Their approach relies on finding *adjacent* live ranges and anti-dependencies; a live range and an anti-dependence are adjacent to each other if the source of one is the sink of the other. Their basic premise is that either live ranges must be local to the band (i.e., both elements in the live range are assigned the same values by the members of a band) being computed or the anti-dependencies adjacent to the live ranges concerned must be satisfied to ensure correctness. In effect, their approach ignores the anti-dependencies on the outer loops that have iteration-private live ranges (i.e., live ranges that are local to the loops) and is thus similar to our approach in this work. However, we find that, in some cases, when it is not possible to force the live ranges to be local to the band (particularly, in certain imperfect nests), then the adjacent anti-dependence is enforced in its entirety, thereby precluding fusion in its entirety. In such cases, *var-lib* can still achieve partial fusion whenever possible, which is very useful to reduce reuse distance by an order of magnitude.

Last, our recent work on the scalability of polyhedral compilers [Mehta and Yew 2015] nicely complements this work in showing the merit of polyhedral optimizations for real applications, which had not been revealed in previous work. It may also be noted that a preliminary version of this work has been available in Mehta [2014].

8. CONCLUSION

In this work, we propose variable liberalization, a compiler technique that strategically removes dependences on temporary variables that impede useful optimizations such as fusion of loop nests and coarse-grained parallelization. Unlike variable expansion, variable liberalization does not cause an actual expansion of variables while enabling fusion, thus further improving the memory performance of transformed programs. In variable liberalization, effective loop fusion is accomplished by a novel method of selective removal of dependences involving temporary variables. This technique differs from (and complements) privatization in that privatization only removes dependences involving temporary variables that appear in the same nest with the objective for coarse-grained parallelization. Experimental results on real applications demonstrate its effectiveness in achieving fusion of nests and subsequent parallelization of the fused nest. On an eight-core Intel Xeon processor, variable liberalization achieves a geomean parallel speedup of 1.92× over the Intel compiler for nine hot regions in five scientific applications.

REFERENCES

Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, 54–64.

Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. 2013. Improved loop tiling based on the removal of spurious false dependences. *ACM Trans. Archit. Code Optim.* 9, 4, Article 52 (Jan. 2013), 26 pages.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 101–113.

David Callahan, Steve Carr, and Ken Kennedy. 1990. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. ACM, 53–65.

Albert Cohen. 1999. Parallelization via constrained storage mapping optimization. In *Proceedings of the International Symposium on High Performance Computing*, Vol. 1615. Springer, 83–94.

Alain Darte and Guillaume Huard. 2005. New complexity results on array contraction and related problems. *J. VLSI Sign. Process. Syst. Sign. Image Vid. Technol.* 40, 1 (2005), 35–55.

Chen Ding and Ken Kennedy. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC* (2004).

Paul Feautrier. 1988. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*. ACM, 429–441.

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.* 34, 3 (2006), 261–317.

Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011.

Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register allocation for programs in SSA-form. In *Proceedings of the 15th International Conference on Compiler Construction*. Springer-Verlag, 247–262.

Intel. 2015. Intel's VTune. (2015). Available at www.intel.com/Software/Products.

Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August. 2013. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 39–50.

Ken Kennedy and Kathryn McKinley. 1994. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*. Lecture Notes in Computer Science, Vol. 768. 301–320.

Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel Comput.* 24, 3–4 (1998), 649–671.

Zhiyuan Li. 1992. Array privatization for parallel execution of loops. In *Proceedings of the 6th International Conference on Supercomputing*. ACM, 313–322.

LLNL. 2015. ROSE Compiler. (2015). Available at http://rosecompiler.org/.

Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. 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*. ACM, 2–15.

Nimrod Megiddo and Vivek Sarkar. 1997. Optimal weighted loop fusion for parallel programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, NY, 282–291.

Sanyam Mehta. 2014. *Scalable Compiler Optimizations for Improving the Memory System Performance in Multi- and Many-core Processors*. Ph.D. Dissertation. University of Minnesota.

Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. 2014. Revisiting loop fusion in the polyhedral framework. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 233–246.

Sanyam Mehta and Pen-Chung Yew. 2015. Improving compiler scalability: Optimizing large programs at small price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 143–152.

OSU. 2012. PolyOpt: a Polyhedral Optimizer for the ROSE compiler. (2012). Available at http://www.cse.ohio-state.edu/pouchet/software/polyopt/.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 90–100.

Louis-Noel Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE, 144–156.

Fabien Quilleré and Sanjay Rajopadhye. 2000. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.* 22, 5 (2000), 773–815.

L. Rauchwerger and D. A. Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.* 10, 2 (Feb 1999), 160–180.

S. K. Singhai and K. S. McKinley. 1997. A parametrized loop fusion algorithm for improving parallelism and cache locality. *Comput. J.* 40, 6 (1997), 340–355.

William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. 2001. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, 232–242.

Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. 2010. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *GCC Research Opportunities Workshop (GROW)*.

Konrad Trifunovic, Albert Cohen, Razya Ladelsky, and Feng Li. 2011. Elimination of memory-based dependences for loop-nest optimization and parallelization. In *GCC Research Opportunities Workshop (GROW)*.

Peng Tu and David A. Padua. 1994. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 500–521.

Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 389–400.

Nicolas Vasilache, Cédric Bastoul, Albert Cohen, and Sylvain Girbal. 2006. Violated dependence analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing*. ACM, 335–344.

Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. 2007. Automatic correction of loop transformations. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 292–304.

Nicolas Vasilache, Benoit Meister, Albert Hartono, Muthu Baskaran, David Wohlford, and Richard Lethin. 2012. Trading off memory for parallelism quality. In *International Workshop on Polyhedral Compilation Techniques, IMPACT*.

Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Mathematical Software ICMS 2010*. Lecture Notes in Computer Science, Vol. 6327. 299–302.

Sven Verdoolaege and Albert Cohen. 2016. Live range reordering. In *6 Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*. Prag, Czech Republic.

Received March 2016; revised June 2016; accepted June 2016