Trading Off Memory For Parallelism Quality

Nicolas Vasilache[1], Benoit Meister[1], Albert Hartono[2], Muthu Baskaran[1], David Wohlford[1], and Richard Lethin[1]

[1]{vasilache,meister,baskaran,wohlford,lethin}@reservoir.com
[1]Reservoir Labs Inc., New-York, NY, USA
[2]{albert.hartono}@intel.com
[2]Intel Labs, Santa Clara, CA, USA (work performed while at Reservoir Labs Inc.)

ABSTRACT

We detail an algorithm implemented in the R-Stream compiler[1] to perform controlled array expansion and conversion to partial single-assignment form, which consists of (1) allowing our automatic code optimizer to selectively ignore false dependences in order to extract a good tradeoff between locality and parallelism, (2) detecting exactly all the causes of semantics violations in the relaxed schedule of the program and (3) incrementally correcting violations by minimal amounts of renaming and expansion. In particular, our algorithm may ignore all false dependences and extract the maximal available parallelism in the program given a limit on the amount of expansion. The spectrum of memory consumption then varies between no expansion and total single assignment, with many steps between those extremes. The exposed parallelism can be incrementally reduced to fit more tightly the number and organization of processing elements available in the targeted hardware, and, by the same token, to reduce the program's memory footprint. We extend our correction scheme in an iterative algorithm to tailor the mapping of the program for a good tradeoff between parallelism, locality and memory consumption. We demonstrate the power of our technique by optimizing a radar benchmark comprising a sequence of BLAS calls. By applying our technique and optimizing at a global level, we reach significant performance improvements over an implementation based on vendor optimized math library calls. Our technique also has implications on algorithm selection.

1. INTRODUCTION

The tension between parallelism and locality of memory references is an important topic in the field of compiler optimization. More parallelism allows more concurrent execution of the parallel portions of a program. Additional parallelism implicitly relates to more available computational operations per second. Increasing locality directly translates into communication reduction between memories and processing elements. Increasing parallelism may decrease locality and vice-versa. In the context of many-core computing and locality-aware programing, our goal is to help solve the complex tradeoffs in a generalized framework for classes of applications suitable for traditional high-level loop transformation optimizations. R-Stream is based on the polytope model [5, 13, 17]. In this setting, it is known that useful scheduling properties of programs can be optimized such as maximal fine-grained parallelism using Feautrier's algorithm [13], maximal coarse-grained parallelism using Lim and Lam's algorithm [22] or maximal parallelism given a (maximal) fusion/distribution structure using Bondhugula et al.'s algorithm [6].

In this paper we present a fully-automated iterative method to perform controlled array expansion as a means to correct violations arising from aggressive scheduling. We discuss differences with other approaches and present a correction algorithm that trades off memory expansion for parallelism and degrades gracefully. We demonstrate the strength of R-Stream by significant performance improvements in two use cases related to dense linear algebra and radar applications.

[1]The technologies described in this paper are Patent Pending, Reservoir Labs, Inc. This work was funded in part by US Army Space and Missile Defense Command, Code: W9113M, P.O. 1500, Huntsville, AL 35807.

IMPACT 2012
Second International Workshop on Polyhedral Compilation Techniques
Jan 23, 2012, Paris, France
In conjunction with HiPEAC 2012.

http://impact.gforge.inria.fr/impact2012

2. RELATED WORK

We rely on well-established traditional terminology of compiler analysis and data dependence analysis [3, 25]. We assume the reader is familiar with the notions of true dependences (i.e. *raw*) and memory based dependences (i.e. *war* or *waw*). To preserve program semantics, code transformations must guarantee all the true dependences are kept consistent with the order they appear in the original execution of the program. In practice, only dataflow dependences must be preserved and various solutions have been explored to reduce the parallelism-limiting effects of memory location reuse. Feautrier [10] described the first algorithm for array expansion in the polyhedral model using a variant of parametric integer programming [11]. Improvements to the Feautrier algorithm include [4]. One of the crucial problems in array expansion is the ability to compute the last write [7, 12, 23]. Unfortunately, the memory requirements of such approaches may grow prohibitively and various techniques have been researched to reduce the consumption once parallelization occurred [9, 26]. Closely related to the problem of array expansion and contraction is *array privatization* [16, 23, 30, 31].

Other advanced contributions have looked at reducing the memory footprint assuming the knowledge of a static specification of the schedule. Thies et al. "consider storage mappings that collapse one dimension of a multi-dimensional array, and programs that are in a single assignment form with a one-dimensional affine schedule" [28]. In the systolic community, Wilde and Rajopadhye proposed a technique for "transforming scheduled single-assignment code to multiple assignment code" [36]. These techniques assume the program is in single assignment form and try to reduce the storage using schedule information. Note that conversion to single assignment entails potentially complex code duplications. This is caused by the dataflow propagation algorithm which has to properly modify all the reads to a renamed, written array. An illustration of this behavior can be found in Figure 9 of Feautrier's original work [12]. Reducing the memory usage alone does not remove these duplications when they are avoidable. For space considerations we omit the discussion on static single assignment (SSA) form [8], Array SSA [19] and Region Array SSA [27]. These contributions and their interplay with the technique we present in this paper will be the topic of future research. Our current implementation is limited to static affine control loops. We have extended it with support for data-dependent conditionals [2] as well as arbitrary code (even precompiled) through the use of blackboxing. The closest contributions to our work are those of Lefebvre [20], Trifunovic [29] and Vasilache [33]. We discuss them in more detail in the next section.

3. OUR SOLUTION

We first define the terminology. The polyhedral model is a mathematical abstraction to represent and reason about programs in a compact representation. We assume the reader is familiar with basic concepts of the polyhedral model. We only establish terminology and describe more advanced notions. Our representation is based on a hierarchical generalized dependence graph (GDG)-based IR. The nodes of our graph are statements that represent operations grouped together in our internal representation. A statement S can be simple or arbitrarily complex (i.e. external precompiled object). Each statement has an iteration domain D^S, which is a union of disjoint convex polyhedra[2]. Each operation within the domain is denoted by $i^S \in D^S$. For each individual array accessed by S in read or write mode, a memory reference A with its associated affine access function is created. If A is injective, only distinct memory locations are touched and there is no temporal memory reuse. Otherwise, temporal memory reuse on reference A exists and may be exploitable depending on the scheduling function chosen. A scheduling function Θ^S maps each iteration in D^S to its actual execution time and thus defines a partial order on the iterations of various statements. We use the notion and conventions of Girbal et al. [14].

Loop Types : We extend our scheduling representation with information pertaining to the kind of parallelism available in a loop. This information corresponds to common knowledge in the compiler community and we use traditional terminology [3]: (1) *doall* loops do not carry any dependence and can be executed in parallel; (2) *permutable* bands of loops carry forward-only dependences and may be safely interchanged and tiled; (3) *sequential* loops must be executed in the specified order (not necessarily by the same processor) and (4) *reduction* loops are assumed to be executable in any sequential order.

Placement : A placement function $\mathcal{P}^S$ is a function that maps the iterations of S to hierarchies of processing elements. Its application to the iteration domain dictates (or provide hints at runtime) what iterations of a statement execute where. There is an implicit relation between the type of loop and the placement function.

Dataflow Dependence : We use the notation $\{T \rightarrow S\}$ to express that T "depends on" S. With this notation, the arrow can also be interpreted as "after". In the case of a *raw* dependence, T is a read that comes "after" S, a corresponding write. A dataflow dependence $\{T \rightarrow S\}_{\mathbf{d}}$ ($\mathbf{d}$ subscript for "dataflow") is a special kind of *raw* dependence. It conveys additional *last-write* information. When it is exact, it does not carry any redundancy (i.e. each read memory value has at most 1 producer). Array dataflow analysis is a global process involving all the statement in the considered portion of the program [12, 16].

Violated Dependence : A violated dependence is a relationship that mixes dependences and scheduling [32]. It occurs when dependent iterations of the source and the target statements are scheduled in different order. Formally, $\{(i^T, i^S) \in \{T \rightarrow S\} \mid \Theta^T \cdot i^T \ll \Theta^S \cdot i^S\}$, where $\ll$ denotes lexicographic ordering. Violated dependences between statements are represented by edges of a special type in the GDG. We write $\{T \rightarrow S\}_{\mathbf{v}}$ to denote the violation information. A *raw* violation *must* be corrected by rescheduling. A *war* of *waw* violation may be corrected *either* by rescheduling *or* by using new memory locations for intermediate storage. A violated dependence is a compact representation of a localized problem in the expression parallelism and can be resolved precisely.

Liveness Violated Dependence : A violated memory-based dependence does not necessarily violate the program semantics. A simple example is as follows: consider a sequence of 4 simple statements $a = b; c = a; a = d; e = a;$. The sequence $a = d; e = a; a = b; c = a;$ contains at least a *waw* violation on a. The final values of c and e are unchanged. Depending on whether a is live on exit (resp. or not), the semantics of this small program is preserved (resp. is violated). We define a liveness violation as a violated dependence that transitively translates into a semantics violation; we write $\{T \rightarrow S\}_{\mathbf{lv}}$. Note that any flow dependence violation automatically translates to a liveness violation. Using a set notation, we derive the following property of liveness violations.

THEOREM 3.1. *Let $\{T \rightarrow S\}_v$ be a violated false dependence. A subset of $\{T \rightarrow S\}_v$ is a liveness violation if and only if there exists a dataflow dependence $\{R \rightarrow T\}_d$ such that: $\{(i^R, i^T, i^S) | (i^R, i^T) \in \{R \rightarrow T\}_d \wedge (i^T, i^S) \in \{T \rightarrow S\}_v \wedge \Theta^T \cdot i^T \ll \Theta^s \cdot i^S \ll \Theta^R \cdot i^R\}$ is non-empty.*

PROOF. Consider a dataflow dependence $\{R \rightarrow T\}_{\mathbf{d}}$ such that R reads array A. No single covering write may intervene between R and T. By construction of the schedule, $\{R \rightarrow T\}_{\mathbf{d}}$ is still valid in the transformed program. Now consider

[2]More precisely, R-Stream provides these representations and implementations optimizations in terms of Z-Domains.

a statement S that also writes array A such that $\{T \rightarrow S\}_{\mathtt{v}}$ is a *waw* violation (i.e. it is a *waw* dependence in the original program that becomes a violation after scheduling). We have the following properties:

- by definition of $\{R \rightarrow T\}_{\mathtt{d}}$, no iteration of S covers the dataflow dependence in the original program,
- therefore no iteration of S should cover $\{R \rightarrow T\}_{\mathtt{d}}$ in the transformed program,

If some iteration of S covers $\{R \rightarrow T\}_{\mathtt{d}}$, then there is a liveness violation. This last condition is written $(\Theta^T \cdot i^T \ll \Theta^s \cdot i^S \ll \Theta^R \cdot i^R)$. $\square$

High-Level Mapping: : R-Stream performs high-level automatic mapping to heterogeneous architectures, a process that includes parallelism extraction, task-formation, locality improvement, processor assignment, data layout management, memory consumption management, explicit data movements generation (as well as their reuse optimization and pipelining with computations) and explicit synchronization generation [21]. Most high-level optimizations in our R-Stream take a GDG as input and generate a new GDG with additional or altered information. Low-level optimizations occur on a different SSA-based IR, after high-level transformations have been applied. The output is generally C extended with annotations and target-specific communication and synchronization library calls (OpenMP, pthreads, DMAs, CUDA, Mitrion Parallel Assembly ...).

In the rest of the paper, we detail our contributions to increase the amount of parallelism and its quality.

3.1 Preliminary Discussion

3.1.1 Partial Expansion

Lefebvre introduced an interesting method to manage storage for parallel programs [20]. His first contribution is a schedule-aware partial expansion that expands arrays as needed for the parallel schedule to respect program dependences. His second contribution examines the conditions under which different arrays can share the same memory space and is akin to memory contraction. We have our own advanced mechanisms to perform array compaction, array contraction and privatization; we consider these are related but different topics. In R-Stream, multiple optimizations occur between expansion and privatization. Our method is lazier: we first compute the violations on the original false dependences of the program. This process does not require array dataflow analysis and serves as a pruning step. Then, we incrementally correct the program by performing only the necessary renaming, expansion and index-set splitting to correct the violated dependences. Correction happens only when the value liveness properties of the original program are violated too (i.e. if a write happens to the memory location before the value has been consumed by all reads). Index-set splitting allows more precise targeting of the memory violations. This is particularly useful when only a subset of the writes are offending and may drastically reduce the amount of expansion needed.

3.1.2 Correction of Loop Transformations

Our work relates to automatic correction of loop transformations [33]. Although we use the authors' ideas to track violations in the program [32], the differences are multiple. By construction our algorithm respects the dataflow dependences in the program; we consider combinations of expansion, renaming and index-set splitting as correction mechanisms; to limit expansion, we consider rescheduling the program to precisely target the causes of the prohibitive memory consumption; when we reschedule, we use the full power of our state-of-the-art polyhedral scheduler and lastly, our method is guaranteed to produce correct solutions. In contrast, the aforementioned contribution violates flow dependences that are hard to correct without a powerful scheduler; only considers loop shifting and index-set splitting as correcting transformations; and will fail if more advanced affine transformations are needed.

3.1.3 Lazy Expansion

Although our work has been developed independently [34] and has not been published until now, it has multiple similarities with the work of Trifunovic [29]. Both works build on the notion of violated dependences [32], perform an iterative lazy expansion scheme to correct liveness dependences and may fail at expanding. As far as differences are concerned, our work always guarantees the proper scheduling of dataflow dependences. Our method allows the setting of an upper bound on the admissible level of memory increase. When the limit is reached, we precisely determine what dependence is the cause of the biggest increase and perform a callback to the scheduler. The scheduler then incorporates the offending false dependence in its constraints set and reschedules the program with potentially less parallelism. By iterating between expansion and scheduling, we reach a fixed point with good parallelism quality given a guaranteed bound on the memory consumption. By varying the amount of admissible expansion, we can trade off memory usage for parallelism quality. Our work allows index-set splitting to be performed on the source of a violation, which can greatly reduce the required increase in memory consumption. We also account for placement considerations to tailor expansion to the dimension and shape of the processor grid. We use loop type informations to further construct liveness violations. The handling of permutable loops is non-trivial and necessary to guarantee tiling will still be legal (see 4.1).

3.2 Traditional Array Expansion

Array expansion is sometimes necessary to enable parallelization. For example, consider the following sequential matrix-vector loop kernel:

```
for (i=0; i<N; i++) {                doall (i=0; i<N; i++) {
  s = 0;                               s[i] = 0;
  for (j=0; j<N; j++) {                red (j=0; j<N; j++) {
   s+=A[i][j] * B[j];                   s[i] += A[i][j] * B[j];
  }                                    }
  C[i] = s;                            C[i] = s[i];
 }                                    }
```

In the left code, the scalar variable may be mapped directly into a machine register, reducing the number of memory accesses. However, it becomes a storage bottleneck when trying to parallelize the loop. `s` may be expanded into an array and yield the code on the right. Later in the mapping process, once placement on the processor space is decided, `s` can be privatized and a single copy is made for each processor. In the particular case of OpenMP, declaring the variable `s` private is sufficient. Note that if conversion to single assignment form is performed [12], `s` will be expanded to a 2-D array and additional control flow will be needed.

3.3 Placement Aware, Iterative Algorithm

Our algorithm (Figure 1) initializes a list of false dependences that must always be preserved by the scheduler in Step 4. If the memory limit M is set to infinity, F_{dep} will never be incremented and the scheduler will never be forced to respect false dependences. In that case, our algorithm may produce a total static expansion. In some cases, F_{dep} may start initialized and the scheduler behaves conservatively with respect to those dependences. This may happen when dataflow dependence analysis can not be computed exactly (for instance when weak-writes are involved.

Inserting Copy-Out Operations : Step 2 of our algorithm corresponds to the *static last-value assignment* described in earlier work [31]. Since corrective array expansion solves conflicts by writing data to new memory locations, it can change the location of memory values that are visible outside of the optimization scope. To handle such cases, we introduce idempotent copies to the liveout memory locations. The dataflow propagation phase (Step 21) of the algorithm treats these copies as regular statements and properly performs the substitution of the reads. Eventually, the portions that have not be renamed are easily removed during a post-processing phase (Step 24). Consider the following example: in Step 2, our algorithm introduces the following copies corresponding to the locations written by references `B` and `C`. Our algorithm may schedule the copies with the original computations or leave them at the end. The latter is illustrated in this example. After dataflow propagation, the copies have been modified on the right code:

```
// Original code                  // Final parallelized code
for (i = 0; i <= N; i++) {        doall (i = 0; i <= N; i++) {
 for (j = 0; j <= N; j++) {        doall (j = 0; j <= N; j++) {
  C[i] = i+j+1;                      C_e[i][j] = i+j+1;
  B[i][1+j] = B[1+i][j]*C[i];        B_r[i][1+j] =
}}                                     B[1+i][j] * C_e[i][j];
doall (i = 0; i <= N; i++) {      }}
 doall (j = 1; j<= N+1; j++) {    doall (i = 0; i <= N; i++) {
     B[i][j] = B[i][j];            doall (j = 1; j <= N+1; j++) {
}}                                   B[i][j] = B_r[i][-1 + j];
doall (i = 0; i <= N; i++) {      }}
 C[i]=C[i];                       doall (i = 0; i <= N; i++) {
}                                  C[i] = C_e[N][i]);
                                  }
```

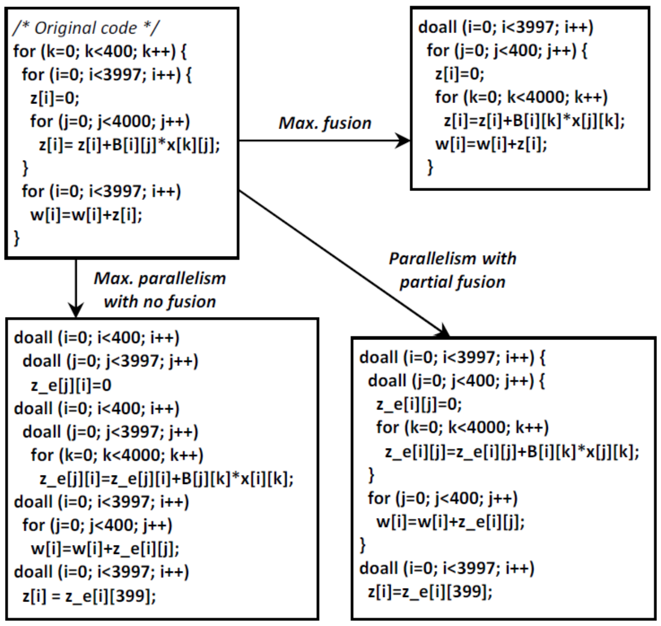
Scheduling And Placement : Step 4 and Step 5 are enabling technologies on which this paper relies. We have discussed the high-level properties of the scheduling and placement algorithms implemented in R-Stream. Figure 2 shows the interplay between these concepts. Additionally, there is a tension between parallelism and locality that interplays with expansion. Parallelism and locality requirements clearly dictate the need for expansion. We also believe that proper memory increase limits can guide the scheduler towards a good tradeoff between parallelism and fusion. Exploitation of such a technique is left for future work; this paper focuses on the ideas and algorithm to enable this exploitation.

Loop Type Information : Loop type information degradation is the means by which our current implementation controls the tradeoff between memory expansion and placement to match the physical resources available.

Computing Violations : Step 7 compute violated dependence information [32] It is done on a written memory reference by reference basis. This step is also the place we

```
Input: A $GDG$ with only nodes, a memory limit $M$
Output: A scheduled $GDG$ fitting within $M$

 1. $F_{dep} \leftarrow \emptyset$
 2. $GDG.nodes \leftarrow insert\_copy\_out\_operations\ ()$
 3. $GDG.edges \leftarrow array\_dataflow\_analysis\ ()$
 4. $GDG.schedule \leftarrow schedule\ (\{GDG.deps \cup F_{dep}\})$
 5. $GDG.placement \leftarrow place\_pe\_grid\ (GDG.schedule)$
 6. $GDG.loop\_info \leftarrow$
      $compute\_loop\ (GDG.schedule,\ GDG.placement)$
 7. $GDG.edges \leftarrow \{GDG.edges \cup$
          $violations\ (GDG.edges,\ GDG.schedule,$
                    $GDG.loop\_info)\}$
 8. $foreach\ (A \in GDG.nodes.written\_references)\ \{$
 9.   $V\_writes \leftarrow \emptyset$
10.   $foreach\ (w =\ \{T \rightarrow S\}_{\mathbf{v}} \in GDG.edges.violations)\ \{$
11.     $foreach\ (r =\ \{T' \rightarrow S'\}_{\mathbf{d}} \in GDG.edges.dataflow)\ \{$
12.       $V\_writes \leftarrow V\_writes\ \cup\ liveness\_pb\ (w,\ r)$
13.   $\}\}$

14.   $GDG.index\_set\_splitting\ (V\_writes)$
15.   $GDG.expand\ (V\_writes)$
16.   $if\ (GDG.memory\_consumption\ () > M)\ \{$
17.     $F_{dep} \leftarrow F_{dep} \cup GDG.get\_expensive\_violation\ ()$
18.     $GDG.reset\ ()$
19.     $goto\ step\ 4$
20.   $\}$

21.   $GDG.dataflow\_propagation\ ()$
22.   $GDG.update\_graph\ ()$
23. $\}$

24. $GDG.remove\_dead\_code\ ()$
```

Figure 1: Placement-Aware, Iterative, Corrective Array Expansion Algorithm


Max. fusion
Max. parallelism with no fusion
Parallelism with partial fusion

```
/* Original code */
for (k=0; k<400; k++) {
 for (i=0; i<3997; i++) {
  z[i]=0;
  for (j=0; j<4000; j++)
   z[i]= z[i]+B[i][j]*x[k][j];
 }
 for (i=0; i<3997; i++)
  w[i]=w[i]+z[i];
}
```

```
doall (i=0; i<3997; i++)
 for (j=0; j<400; j++) {
  z[i]=0;
  for (k=0; k<4000; k++)
   z[i]=z[i]+B[i][k]*x[j][k];
  w[i]=w[i]+z[i];
 }
```

```
doall (i=0; i<400; i++)
 doall (j=0; j<3997; j++)
  z_e[j][i]=0
doall (i=0; i<400; i++)
 doall (j=0; j<3997; j++)
  for (k=0; k<4000; k++)
   z_e[j][i]=z_e[j][i]+B[j][k]*x[i][k];
doall (i=0; i<3997; i++)
 for (j=0; j<400; j++)
  w[i]=w[i]+z_e[i][j];
doall (i=0; i<3997; i++)
 z[i] = z_e[i][399];
```

```
doall (i=0; i<3997; i++) {
 doall (j=0; j<400; j++) {
  z_e[i][j]=0;
  for (k=0; k<4000; k++)
   z_e[i][j]=z_e[i][j]+B[i][k]*x[j][k];
 }
 for (j=0; j<400; j++)
  w[i]=w[i]+z_e[i][j];
}
doall (i=0; i<3997; i++)
 z[i]=z_e[i][399];
```

Figure 2: Tradeoff fusion/parallelism/expansion

consider the loop types computed in Step 6 that are schedule and placement dependent. In particular, if a loop in the final space-time order is a doall, it may be executed in parallel. To guarantee correctness under synchronization-free parallel execution, no violation should occur under doall loop semantics. Intuitively, this requires more memory to store temporary values than if the loop semantic were sequential. The decision on conservative handling of the loop types information (irrespective of runtime decisions) is done by conservative violation computation. We use the characterization of loop types semantics. We abuse the notations to order loop types by their impact on the amount of expansion (the higher in the order, the more impact on memory): *doall* > *reduction* > *permutable* > *sequential*. Note that, in the following example, even sequential loops can force expansion.

```
                      // Scheduled (poorly), needs expansion
// Original           for (i=0; i<N; i++)
for (i=0; i<N; i++)     a = B[i];
  a = B[i];           for (j=0; j<N; j++)
  A[i] = a;             A[i] = a;
```

Consider a false dependence $\{T \rightarrow S\}$ in the original program where T is a write. To compute violations with loop type information, we determine the iteration subset where the schedule for S is greater or equal to T after transformation. Under the loop type semantics, the following cases define violation subsets of the original dependence:

- portions of *doall* loops that are strictly reversed,
- portions of *reduction* loops ordered at the same time,
- portions of *sequential* loops that are ordered at the same time step or that are reversed,
- *permutable* loops are trickier; they must be considered by groups of fully permutable bands. For K permutable loops, up to K portions in violation may be generated (lexicographic order reversal along K).

This complex case disjunction is necessary to support all loop type semantics. Support for permutable loops is crucial for tiling [18, 37].

Gathering Liveness Violations : Steps 10 to 13 compute liveness violations, which must be corrected. Step 12 is clear: for each of the false dependence violation on A, iterate over the dataflow dependences reading the reference A and determine if the dataflow dependence is covered by the violation. The problem reduces to writing the proper ordering constraints using set operations and the schedules in the transformed program [32]. From this point onwards, violation refers to liveness violations that must be corrected by expansion.

Index-set splitting : In the context of false dependence violations, it has been shown that index-set splitting can reduce the volume of iterations of a statement that needs correction [33]. In our context, this idea translates directly into fewer memory duplications. The idea is to determine whether violation occurs on a relatively smaller (i.e. at least 2x lower volume or strictly lower dimensionality) subportion of the iteration domain of the write statement. This is done by projecting the violation set on the iteration domain of the offending write statement using standard operations on iteration domains. Splitting too much may be undesirable because it increases the code size by a-priori unpredictable amounts. Our heuristic is based on counting points in parametric polyhedra [24]. It has a parametrized threshold at which to trigger splitting. At this point, any user input on parameter context is very useful.

Dataflow Propagation and Side Effects : Dataflow propagation is called in Step 21 but also helps explain the expansion process of Step 15, which is the reason we discuss it now. The need for dataflow propagation occurs when a written memory location is expanded. The program must be updated so that statements touching modified memory locations properly reference the new locations. This information is directly available from the flow dependences in the GDG. Propagating this information creates a need for index-set splitting in the statements that correspond to the read portion of the dependences. Here, the split is not an optimization but is needed for semantic correctness. The determination of the split is done by projecting the flow dependence on the iteration domain of the reading statement using standard operations on iteration domains. In the following example, suppose `c` is live on exit. After renaming, a portion of T must refer to the original memory `c` and another (disjoint) portion of T must refer to the renamed memory `c_r`. We assume `c` is live on exit:

```
// Before renaming      // After renaming
c = 0;                  c = 0;
for (i=0; i<=n; i++) {  for (i=0; i<=n; i++) {
  B[i] = c;              B[i] = (i==0) ? c : c_r;
  c = c + A[i];          c_r  = (i==0) ? c+A[i] : c_r+A[i];
}                       }}
                        c = c_r;
```

In general, index-set splitting may create very complex control flow, especially in the context of Z-Polyhedra where the lattice is not the identity. We developed heuristics to keep the splitting under control. Modeling the control flow increase to degrade expansion and by extension parallelism will be the topic of future work.

Renaming and Expansion : The reader may assume we are using Lefebvre's method [20] to perform expansion and renaming. Eventually, our algorithm iterates on all nodes that are still in violation at a given step. It gathers all dependences creating the considered violation and computes the new renamed or expanded array. Dataflow propagation is then called to update all the depending nodes. Consider the following simple illustrative example and its aggressively parallelized version with maximal parallelism and without considerations for correctness:

```
// Original code          // Maximal parallelism
for (i=0; i<=N; i++) {    doall (i=0; i<=N; i++) {
 for (j=0; j<=N; j++) {    doall (j=0; j<=N; j++) {
  C[i]=i+j+1;                C[i]=i+j+1;
  B[i][1+j]=                 B[i][1+j]=
      B[1+i][j]*C[i];            B[1+i][j]*C[i];
}}                        }}
```

There is a violation $\{S_1 \rightarrow S_0\}_{\mathtt{lv}}$ for all values of j because all iterations (i, j) occur (semantically) at the same time. However, a simple renaming of C[i] into D[i] will not suffice because of the dataflow dependence $\{S_1 \rightarrow S_0\}_{\mathtt{d}}$. Expansion is necessary as seen in the transformed code below.

```
// Renamed                 // Expanded
doall (i=0; i<=N; i++) {   doall (i=0; i<=N; i++) {
 doall (j=0; j<=N; j++) {   doall (j=0; j<=N; j++) {
  D[i]=i+j+1;                 C_e[i][j]=i+j+1;
  B[i][1+j]=                  B[i][1+j]=
    B[1+i][j]*D[i];             B[1+i][j]*C_e[i][j];
}}                         }}
```

In this form, the code is not yet correct since a violation $\{S_1 \rightarrow S_1\}$ remains on array `B`. The next step of our algorithm properly performs the renaming of `B` into `B_r`. Note that renaming does not modify the access subscripts. Our analysis shows there is no dataflow dependence between the renamed iterations and renaming is sufficient. In this case, renaming has the effect of enabling the parallelism in the loops by differentiating the read array from the written array.

```
// Final code
doall (i=0; i<=N; i++) {
  doall (j=0; j<=N; j++) {
    C_e[i][j]=i+j+1;
    B_r[i][1+j]=B[1+i][j]*C_e[i][j];
}}
```

Recovering From Memory Expansion Limit : Steps 16 to 20 describe our algorithm's behavior in case the limit on memory increase M is reached. The behavior is simple. We augment F_{dep} with a well-chosen dependence that entails the most violations. This is currently chosen using a heuristic based on counting the number of violations [24]. The algorithm jumps back to Step 4 and restarts. This behavior is not optimal since progress on expansion is lost but it is guaranteed to terminate and works well in practice.

Removing Dead Code : Step 2 of the algorithm inserts idempotent copy-out operations and modify them accordingly during dataflow propagation. If portions of such copies are not modified at all during the whole corrective array expansion algorithm, they end up as useless copies. Copies that read and write the same data at the end of the algorithm are deleted from the transformed program in Step 23. In addition, we should also mention that the dataflow propagation of our algorithm has a nice property, as it may exhibits dead code in the original program: portions of the program that write to memory locations that are overwritten before being read or that are never read are removed.

4. PERFORMANCE EVALUATION

4.1 An Example: Givens QR Decomposition

This section demonstrates the interplay with array expansion. In the pseudo-codes presented, $> a$ means write to a, $< a$ means read from a, $<> a$ means first read then write a in the same iteration. This notation allows us to be concise and to abstract from the operational details of the algorithm.

Because of scalar dependences parallelization is traditionally prohibited, except on the innermost k-loop. Total array expansion converts all scalars to 2-D arrays with a memory increase of 400%. Depending on the type of the loops after scheduling, corrective array expansion generates either no increase (Version 0), a 0.5% increase (Version 1), a 1% increase (Version 3) or a 400% increase (Version 2)[3].

```
// Original {seq,seq,doall}  // Version 1 {perm,perm,doall}
for (i=0; i<=N-2; i++){      for (i=0; i<=N-2; i++){  // p
 for (j=0; j<=N-i-2; j++){    for (j=i; j<=N-2; j++){ // p
  S0(<A[N-1-j,i], >a);          S0(<A[N-1+i-j,i], >a[i]);
  S1(<A[N-2-j,i], >b);          S1(<A[N-2+i-j,i], >b[i]);
  S2(<a, <b, >c, >d);           S2(<a[i], <b[i], ...);
```

[3]The $\{perm, perm, perm\}$ loop structure is only obtainable when S_0, S_1 and S_2 are sunk at the same level as S_3. This is done by adding a dummy $k = 0$ dimension in the IR that does not change the input code. This enables finer pipelining with S_3 by the scheduler.

```
  doall (k=0; k<=N-i; k++){     doall (k=0; k<=N-1-i; k++){
   S3(<>A[N-2-j,i+k],             S3(<>A[N-2+i-j, i+k],
      <>A[N-1-j,i+k],                <>A[N-1+i-j, i+k],
      <c, <d);                       <c[i], <d[i]);
}}}                              }}}
// Version 2 {perm,perm,perm}     // Vers. 3 {seq,doall,doall}
for (i=0; i<=N-2; i++) {  // p    for (i=0; i<=2*N-4; i++){
 for (j=i; j<=N-2; j++) { // p     doall (j=..; j<=..; j++){
  for (k=i; k<=N-1; k++) {// p      S0(<A[N-1+i-j,i],>a[i-j]);
   if (k==i) {                      S1(<A[N-2+i-j,i],>b[i-j]);
    S0(<A[N-1+i-j,i],>a[i,j-i]);    S2(<a[i-j],<b[i-j],...);
    S1(<A[N-2+i-j,i],>b[i,j-i]);    doall (k=..; k<=..; k++) {
    S2(<a[i,j-i],<b[i,j-i],          S3(<>A[N-2+i-j,i+k],
        >c[i,j-i],>d[i,j-i]);           <>A[N-1+i-j,i+k],
   }                                    <c[i-j],<d[i-j]);
   S3(<>A[N-2+i-j,i+k],           }}}
      <>A[N-1+i-j,i+k],
      <c[i,j-i],<d[i,j-i]);
}}}
```

We omit the upper and lower bounds for the `j` and `k`-loops in Version 3. They are non-trivial expressions involving min, max and division operations that would clutter the code.

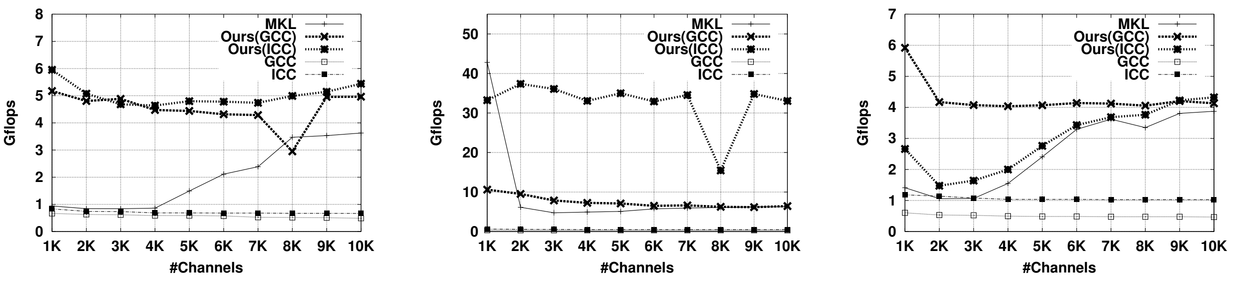
This has implications on the shapes of parallelism that R-Stream discovers. Our whole mapping process supports generation of code for Cell and SMP. Permutable loops are tiled creating coarse-grained tasks with less synchronizations. We observe target-specific behaviors. Version 0 does not have permutable loops; its execution is slow on both targets. It is the base of our experiments without our optimizations, Version 1 allows the creation of coarse-grained tasks and performs the best in our Xeon trial. This is not true on Cell where the footprint of the innermost k loop is too large to fit the scratchpad memory. On Cell, Version 1 executes slower than the original version. This is because there is no gain in parallelism granularity, hence no reuse across lines of A. Additionally, we have to communicate `a`, `b`, `c` and `d` which are now 1-D arrays. On SMP, the cache hierarchy saves the day. Version 2 has the nice property of exhibiting three permutable loops which can create two coarse-grained parallel loops after 3-D tiling. This comes at the cost of the global memory footprint of the application. The much higher memory cost is better for Cell because the local footprint resulting from the deep tiling of the loops fits on the scratchpads. Therefore, the local footprints exhibit reuse that reduces the number of overall communications. Interestingly, a much bigger memory footprint results in much fewer communications thanks to better exploitation of locality. Version 3 is an intermediate version that fits both Cell and SMP. It is easily obtained from Version 1 by skewing the `j`-loop into `i`. We can describe it as a medium-grained parallel version with many outermost sequential iterations. It performs decently on both targets.

Target	Ver. 0	Ver. 1	Ver. 2	Ver. 3
Xeon E5405(GCC-4.4)	1	**11.08**	10.46	5.38
Cell QS22 (XLC-10.1)	1	0.58	**16.77**	5.27

We run in single precision mode, on a 1024x1024 matrix and we obtain the performance table above. Numbers in each row are normalized execution speeds relative to the original version without our compiler optimizations (base 1).

4.2 Radar Benchmark

We demonstrate the benefits of our algorithm on a signal processing application. Adaptive beamforming is an algorithm to eliminate interference and clutter in a phased



(a) MVDR-SER (b) CSLC-LMS (c) CSLC-RLS

Figure 3: Parallel Radar Performance.

array antenna. Multiple different beamforming algorithms exist [35], we study three of them (MVDR-SER, CSLC-RLS and CSLC-LMS) in the context of corrective array expansion. We have implemented two versions of these algorithms: an Intel MKL library based on BLAS calls and a simple textbook C version. R-Stream optimizes the C code and produces OpenMP: we compare C code optimized to OpenMP to a sequence of MKL calls. We evaluated performance on a dual socket quad-core E5405. We used R-Stream , GCC 4.3.0[4], ICC 11.0[5][6] and Intel MKL 10.2.1. We report performance numbers on eight threads in single precision mode. All experiments were run ten times and then averaged. The performance results for the three beamforming algorithms are provided in Figure 3. The performance of ICC and GCC is low; they are unable to parallelize the textbook codes. Our performance is better than MKL for most of problem instances. We obtain up to 7x speedup over MKL on CSLC-LMS. This is because we are able to parallelize the outermost loop iterating over CSLC-LMS computation, whereas in the MKL version, such an outermost loop can not be parallelized due to loop-carried dependences. For MVDR-SER and CSLC-RLS, effective exploitation of data locality dominates the performance as the outermost loops of the most compute-intensive parts must be executed sequentially because of loop-carried dependences. Obtaining the results in this section did not necessitate the use of a memory limit. Figure 2 is an extremely simplified view of accesses to one of the arrays in the hotspot of the CSLC-LMS algorithm, the one on which our solution performs best. The original code can be represented with a few dozen lines of C code (or 5 MKL calls and other statements enclosed in a sequential outer loop). The final code is more than 1,000 lines long, after scheduling, correction and tiling are applied. Back to Figure 2, R-Stream's affine scheduling algorithm automatically finds the bottom-right schedule (parallelism with partial fusion) once false dependences are ignored. Writes to z are then corrected into expanded writes to z_e and 2-dimensional arrays are sufficient to enable the exploitation of 2 degrees of outermost parallelism. These 2 outermost parallel loops also enable tiling. The performance of the top-right code variant (max. fusion) is significantly lower. Even if the memory footprint is significantly smaller, tiling cannot be exploited (i.e. only 1 doall loop). The performance of the bottom-left code (max. parallelism with no fusion) is also significantly lower because fusion and reuse opportunities are lost. In terms of algorithm variants, our solution allows us to determine that CSLC-LMS is an algorithm that contains more parallelism than the other two. To be complete, our solution should also include index-set splitting [15] based on true dependences to uncover the absolute maximal available parallelism. Nevertheless, it can be readily used to help with algorithm selection.

[4]”-O6 -fno-trapping-math -ftree-vectorize -msse3 - fopenmp”
[5]”-fast -openmp” flags for the code we generate
[6]”-fast -parallel” flags for the MKL code.

5. DISCUSSION AND FUTURE WORK

An important class of storage optimization techniques are based on optimizing storage given a fixed schedule [1]. Such techniques could be integrated directly in our iterative algorithm to reduce the actual footprint to a minimum before deciding whether to reintegrate a liveness violation and to trigger rescheduling. Other clear improvements would consist in integrating array privatization at the same stage, bounding expanded buffers to sizes multiples of the actual tile sizes and taking advantage of OpenMP lastprivate semantics to remove unnecessary copy-backs. One of the challenges in performing contraction and privatization within the algorithm presented in this paper is the complex interplay with hierarchical scheduling. It is known that schedule-independent storage optimization is less successful than schedule-aware storage optimization [28]. Contracting too early in the compilation flow would prevent certain schedules at the next level of the hierarchy. On the other hand, applying the algorithm we propose, especially after multiple levels of tiling, is very unlikely to scale. Therefore, the currently preferred approach is to handle these additional optimization opportunities separately, in a latter phase when schedules and placements are fully determined. R-Stream is not limited to a single scheduling decision and has mechanisms to devise different schedules at various levels of the architecture hierarchy. For instance, this allows exploitation of coarse-grained parallelism at the highest level and fine-grained SIMD parallelism at the lowest level. The expansion we describe in this paper lies somewhere in between schedule-independent and schedule-aware expansion. It consists in performing expansion that allows a controlled subset of loop transformations (i.e. the ones needed for tiling). This is achieved by supporting permutable loop semantics and devising expansions that are valid under any tiling configuration using a given schedule. Without this special support, it would not be possible to allow tiling and parallelism exploitation for Version

1 and Version 2 of the QR decomposition for instance.

It is not clear that contraction techniques would easily be integrated in this philosophy. This is an interesting research prospect. In the future, it will also be interesting to examine the results of our algorithm in the context of auto-tuning. Areas with great potential include understanding how memory usage limitation influences parallelism and vice-versa and to examine tradeoffs with control flow increase on various architectures. Other opportunities include further extending the behavior of our algorithm to support runtime generated information and some form of dynamic single assignment.

6. CONCLUSION

We have introduced an iterative algorithm for array expansion that corrects liveness violations of an aggressively scheduled program. By combining placement decisions on the physical processing elements available with loop type semantics, our algorithm supports subsequent application of tiling. We have developed and integrated this algorithm in R-Stream and put particular emphasis on applicability of the technique. We have showed simple examples of new mapping tradeoffs between quality of parallelism and memory usage that our algorithm discovers. Additionally, we have uncovered significant parallelism in *only one out of three* algorithms that performs interference elimination in a radar application. The other two algorithms do not possess this kind of parallelism which limits their applicability to a large number of antennas. We demonstrated up to 7x speedup over a sequence of MKL calls. We believe this contribution is an important step in better harnessing the difficult tradeoffs between parallelism, locality, communication and memory usage.

7. REFERENCES

[1] C. Alias, F. Baray, and A. Darte. Bee+cl@k: an implementation of lattice-based array contraction in the source-to-source translator Rose. In *Languages Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 73–82, New York, NY, USA, June 2007. ACM Press.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures : a dependence-based approach.* Morgan Kaufmann, 2002.

[4] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–106, January 1998.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04*, pages 7–16, Juan-les-Pins, September 2004.

[6] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, Tucson, Arizona, June 2008.

[7] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 92–101, Santa Barbara, California, July 1995.

[8] R. Cytron, J. Ferrante, B. K. Rosen, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Computers*, 54(10):1242–1257, December 2005.

[10] P. Feautrier. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*, St. Malo, France, June 1988.

[11] P. Feautrier. Parametric integer programming. *RAIRO-Recherche Opérationnelle*, 22(3):243–268, 1988.

[12] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[15] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28:607–631, July 1999.

[16] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, March 1995.

[17] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *PPoPP'07*, pages 237–248, New York, NY, USA, June 2007. ACM Press.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, January 1988. ACM Press.

[19] K. Knobe and V. Sarkar. Array-SSA form and its use in parallelization. In *ACM SIGPLAN POPL*, pages 107–120, January 1998.

[20] V. Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, May 1998.

[21] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. Wohlford. Final report on the R-Stream 3.0 compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008.

[22] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *PoPL'97*, pages 201–214, Paris, France, January 1997.

[23] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[24] B. Meister and S. Verdoolaege. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems*, April 2008.

[25] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan-Kaufmann, 1997.

[26] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *Transactions on Programming Languages and Systems*, 22(5):773–815, September 2000.

[27] S. Rus, G. He, C. Alias, and L. Rauchwerger. Region array-SSA. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 43–52, New York, NY, USA, January 2006. ACM.

[28] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN PLDI*, pages 232–242, June 2001.

[29] K. Trifunovic, A. Cohen, R. Ladelsky, and F. Li. Elimination of memory-based dependences for loop-nest optimization and parallelization. In *3rd GCC Research Opportunities Workshop (GROW'11)*, Chamonix, France, April 2011.

[30] P. Tu. Automatic array privatization and demand-driven symbolic analysis. Technical Report UIUCDCS-R-95-1911, University of Illinios at Urbana-Campaign, May 1995.

[31] P. Tu and D. A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.

[32] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *ICS*, pages 335–344, June 2006.

[33] N. Vasilache, A. Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*, pages 292–304, Brasov, Romania, September 2007. IEEE Computer Society Press.

[34] N. T. Vasilache, A. K. Leung, B. Meister, and R. A. Lethin. System, method and apparatus for aggressive program scheduling. In *U.S. Provisional App. No. 61/371,126*, August 2010.

[35] P. Vouras and B. Freburger. Application of adaptive beamforming techniques to HF radar. In *IEEE Radar Conference*, pages 1–6, Roma, May 2008.

[36] D. Wilde and S. V. Rajopadhye. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters*, 7(2):203–215, June 1997.

[37] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, December 1989. Society for Industrial and Applied Mathematics.