Memory Allocations for Tiled Uniform Dependence Programs *

Tomofumi Yuki
Colorado State University
Fort Collins
Colorado, U.S.A.
yuki@cs.colostate.edu

Sanjay Rajopadhye
Colorado State University
Fort Collins
Colorado, U.S.A.
Sanjay.Rajopadhye@colostate.edu

ABSTRACT

In this paper, we develop a series of extensions to schedule-independent storage mapping using Quasi-Universal Occupancy Vectors (QUOVs) targeting tiled execution of polyhedral programs. By quasi-universality, we mean that we restrict the "universe" of the schedule to those that correspond to tiling. This provides the following benefits: (i) the shortest QUOVs may be shorter than the fully universal ones, (ii) the shortest QUOVs can be found without any search, and (iii) multi-statement programs can be handled. The resulting storage mapping is valid for tiled execution by any tile size.

1. INTRODUCTION

In this paper, we discuss storage mappings for tiled programs, especially for the case when tile sizes are not known at compile time. When the tile sizes are parameterized, most techniques for storage mappings [6, 13, 15, 20] cannot be used due to the non-affine nature of parameterized tiling. However, we cannot combine parametric tiling with memory re-allocation if we cannot find a legal allocation for all legal tile sizes.

One approach that can find storage mappings for parametrically tiled programs is the Schedule-Independent Storage Mapping proposed by Strout et al. [19]. For programs with uniform dependences, schedule-independent memory allocation finds storage mappings that are valid for any legal execution of the program, including tiling by any tile size. We present a series of extensions to schedule-independent mapping for finding legal and compact storage mappings for polyhedral programs with uniform dependences.

Schedule-Independent Storage Mapping is based on what are called Universal Occupancy Vectors (UOVs), that characterize when a value produced can safely be overwritten. As originally defined by Strout et al, UOVs are *fully universal*, where the resulting allocations are valid for *any* legal schedule. In this paper, we restrict the UOVs to smaller universes and exploit their properties to efficiently find good UOVs. In the remainder of this paper, we call such UOVs that are not fully universal *Quasi-UOVs* (QUOVs) to distinguish them from fully universal ones. Using QUOVs, we can find valid mappings for tiled execution by any tile size, but not necessarily valid for other legal schedules. This leads to more compact storage mappings for cases when fully universal allocation is an overkill.

*This work was funded in part by the National Science Foundation, Award Numbers: 1240991 and 0917319

The restriction on the universality leads to the following:

- QUOVs may be shorter than the shortest UOV. Since the universe of possible schedules is restricted, valid storage mappings may be more compact. We use Manhattan distance as the length of UOVs as a cost measure when we describe optimality of a projective memory allocation.
- The shortest QUOV for tiled loop programs can be analytically found, and the dynamic programming algorithm presented by Strout et al. is no longer necessary.
- Imperfectly nested loops can be handled. The original method assumed single statement programs (and hence perfectly nested loop programs.) We extend the method by taking statement orderings, often expressed in the polyhedral model as constant dimensions, into account. This is possible because we focus on tiled execution, where tiling applies the same schedule (except for ordering dimensions) to all statements.

The input to our analysis is a program in polyhedral representation, which can be obtained from loop programs through array data-flow analysis [7, 14]. Our methods can be used for loop programs in single assignment form; an alternative view used in some of the prior work [19, 20].

In addition, we present a method for isolating boundary cases based on index-set splitting [9]. UOV-based allocation assumes that every a dependence is active at all points in the iteration space. In practice, programs have boundary cases where certain dependences are only valid at iteration space boundaries. We take advantage of the properties of QUOVs we develop to guide the splitting.

2. BACKGROUND

In this section, we present the background necessary for this paper. We first introduce the terminology used, and then present an overview of Universal Occupancy Vectors [19].

2.1 Polyhedral Representations

Polyhedral representations of programs primarily consist of statement domains and dependences. Statement domains represent the set of iteration points where a statement is executed. In polyhedral programs, such sets are described by a finite union of polyhedra.

In this paper, we focus on programs with uniform dependences, where the producer and the consumer differ by a constant shift. We characterize these dependences using a

vector, which we call *data-flow vector*, drawn the producer to the consumer. For example, if a value produced by an iteration $[i, j]$ is used by another iteration $[i + 1, j + 2]$, the corresponding data-flow vector is $[1, 2]$.

2.2 Schedules and Storage Mappings

The schedules are represented by affine mappings that map statement domains to a common dimensional space, where the lexicographic order denotes the order of execution. In the polyhedral literature, statement orderings are commonly expressed as constant dimensions in the schedule. Furthermore, one can represent arbitrary orderings of loops and statements by adding $d + 1$ additional dimensions [8], where d is the dimensionality of statement domains in the programs[1]. To make the presentation consistent, we assume such schedules are always used, resulting in a $2d + 1$ dimensional schedule.

The storage mappings are often represented as a combination of affine mappings and dimension-wise modulo factors [13, 16, 19].

2.3 Tiling

Tiling is a well known loop transformation that was originally proposed as a locality optimization [12, 17, 18, 22]. It can also be used to extract coarser grained parallelism, by partitioning the iteration space to tiles (blocks) of computation, some of which may run in parallel [12, 17].

Legality of tiling is a well established concept defined over contiguous subsets of the schedule dimensions (in the range of the scheduling function; scheduled space), also called *bands* [3]. These dimensions of the schedules are tilable, and are also known to be fully permutable [12].

The range space of the schedules given to statements in a program all refers to the common space, and thus have the same number of dimensions. Among these dimensions, a dimension is tilable if all dependences are not violated (i.e., the producer is not scheduled after the consumer, but possibly be scheduled to the same time stamp,) with a one-dimensional schedule using only the dimension in question. Then any contiguous subset of such dimensions forms a legal tilable band.

We call a subset of dimensions in an iteration space to be tilable, if the identity schedule is tilable for the corresponding subset. The iteration space is *fully-tilable* if all dimensions are tilable.

2.4 Universal Occupancy Vectors

A Universal Occupancy Vector (UOV) [19] is a vector that denotes the "distance after which" when a value may safely be overwritten in the following sense. When an iteration z is executed, its value is stored in some memory location. Another iteration z' can reuse this same memory location if all iterations that use the value produced by z have been executed. When a storage mapping is such that z and z' are mapped to the same memory location, the difference of these points, $z' - z$, is called the occupancy vector.

Universal Occupancy Vector is a specialization of such vectors, where all iterations k that use z are guaranteed to be executed before z' in any legal schedule. Since most machine models assume that, within a single assignment statement, reads happen before writes in a given time step, z' may also use the value produced by z.

Additionally, we introduce a notion of scoping to UOVs. We call a vector v to be an UOV with respect to a set of dependences $\mathcal{I}$, if the vector v satisfies the necessary property to be an UOV for a subset of all points k that use the value produced by z with one of the dependences in $\mathcal{I}$.

Once the UOV is computed, the mapping that corresponds to the projection of the statement domain along the UOV is a legal storage mapping. If the UOV crosses more than one integer points, then an array that corresponds to a single projection is not sufficient. Instead, multiple arrays are used in turn, implemented using modulo factors. The necessary modulo factor is the GCD of elements of the UOV.

The trivial UOV; a valid, but possibly suboptimal UOV is computed as follows.

1. Construct the set of data-flow vectors corresponding to all the dependences that use the result of a statement.
2. Compute the sum of all data-flow vectors in the constructed set.

The above follows from a simple proposition shown below, and an observation that the data-flow vector of a dependence is a legal UOV with respect to that dependence.

PROPOSITION 1 (SUM OF UOVS). *Let u and v be, respectively, the UOVs for two sets of dependences $\mathcal{U}$ and $\mathcal{V}$. Then $u + v$ is a legal UOV for $\mathcal{U} \cup \mathcal{V}$.*

PROOF. The value produced at z is dead when $z + u$ can legally be executed with respect to the dependences in $\mathcal{U}$, and similarly for $\mathcal{V}$ at $z + v$. Since there is a path from z to $z + u + v$ by following the edges $z + u$ and $z + v$ (in either order), the value produced at z is guaranteed to be used by all uses, $z + u$ and $z + v$, when $z + u + v$ can legally be executed. □

The optimality of UOVs without any knowledge of size or shape of the iteration space is captured by the length of the UOV. However, the length that should be compared is not the Euclidean length, but the Manhattan distance. We discuss the optimality of UOV-based allocations and other projective allocations in Section 7.

3. OVERVIEW OF OUR APPROACH

UOV-based allocation give legal mappings even for schedules that cannot be implemented as loops. For example, even a run-time work stealing scheduler can use UOV-based allocation. However, this is obviously an overkill if we only consider schedules that can be implemented as loops.

The important change in perspective is that we are not interested in schedule-independent storage mappings, although the concept of UOV is used. We are only interested in using UOV-based allocation in conjunction with tiling. Thus, our allocation is partially *schedule-dependent*. The overview of our storage mapping strategy is as follows:

1. Extract polyhedral representations of programs (array expansion.)
2. Perform schduling and apply the schedules as transformations to the iteration space[2]. After the transformation, lexicographic scan of the resulting iteration space

[1]Note that for uniform dependence programs, all statement domains have the same number of dimensions.

[2]This can be viewed as pre-processing to code generation [2].

respects the schedules. The resulting space should be (partially) tilable to take advantage of our approach.

3. Apply UOV-guided index-set splitting (Section 6.) This step attempts to isolate boundaries of statement domains that negatively influence storage mappings.

4. Apply QUOV-based allocation (Section 4.) Our proposed storage mapping based on extensions to the UOVs are applied to each statement after the splitting. Although inter-statement sharing of arrays may be possible, such optimization is beyond the scope of this paper.

The order of presentation does not follow the above for two reasons. One is that the UOV-guided splitting is an optional step that can further optimize memory usage. In addition, splitting introduces multiple statements to the program, and requires our extension to handle multiple statements presented in Section 5.

4. QUOV-BASED ALLOCATION FOR TILED PROGRAMS

In this section, we present a series of formalism to analytically find the shortest QUOV. We first develop a lemma that can eliminate dependences while constructing UOVs. We then apply the lemma to find the shortest QUOVs in different contexts.

4.1 Relevant Set of Dependences for UOV Construction

The trivial UOV, which also serves as the starting point for finding the optimal UOV, is found by taking the sum of all dependences. However, this formulation may lead to significantly inefficient starting points. For example, if two dependences with data-flow vectors $[1,0]$ and $[2,0]$ exist, the former dependence may be ignored during UOV construction since a legal UOV-based allocation using only the latter dependence is also guaranteed to be legal for the former dependence.

We may refine both the construction of the trivial UOV and the optimality algorithm by reducing the set of dependences considered during UOV construction. The optimality algorithm presented by Strout et al. [19] searches a space bounded by the length of trivial UOV using dynamic programming. Therefore, reducing the number of dependences to consider will improve both the trivial UOV and the dynamic programming algorithm.

The main intuition is that if a dependence can be transitively expressed by another set of dependences, then it is the only dependence that needs to be considered. This is formalized in the following lemma.

LEMMA 1 (DEPENDENCE SUBSUMPTION). *If a dependence f can be expressed as compositions of dependences in a set $\mathcal{G}$, where all dependences in $\mathcal{G}$ are used at least once in the composition, then a legal UOV with respect to f is also a legal UOV with respect to all elements of $\mathcal{G}$.*

PROOF. Given a legal UOV with respect to a single dependence f, the value produced at z is preserved at least until z' defined by $f(z') = z$, can be executed. Let the set of dependences in $\mathcal{G}$ be denoted as g^x, $1 \leq x \leq |\mathcal{G}|$. Since composition of uniform functions is associative and commutative, there is always a function g^* obtained by composing dependences in $\mathcal{G}$, such that $f = g^* \circ g^x$ for each x. Thus, all points z'', $g^x(z'') = z$, are executed before z' for all x. Therefore, a legal UOV with respect to f is guaranteed to preserve the value produced at z until all points that directly depend on z by a dependence in set $\mathcal{G}$ have been executed. □

Finding a composition in the above can be implemented as an integer linear programming problem. The problem may also be viewed as determining if a set of vectors are linearly dependent when restricted to positive combinations. The union of all sets $\mathcal{G}$, called subsumed dependences, found in the initial set of dependences can be ignored when constructing the UOV.

Applying Lemma 1 may significantly reduce the number of dependences to be considered. However, the trivial UOV of the remaining dependences may still not be the shortest UOV. For example, consider data-flow vectors $[1,1], [1,-1], [1,0]$. Although the vectors are independent by positive combinations, the trivial UOV $[3,0]$ is clearly longer than another UOV $[2,0]$. Further reducing the set of dependences to consider requires a variation of Lemma 1 that allows f also to be a composition of dependences. This leads to complex operations, and the dynamic programming algorithm for finding optimal UOV by Strout et al. [19] may be a better alternative. Instead of finding the shortest UOV in the general case, we show that such UOV can be found very efficiently for a specific context, namely tiling.

4.2 Finding the Shortest QUOV for Tiled Programs

When UOV-based allocation is used in the specific context of tiling, the shortest QUOV can be analytically found. If we know that the program is to be tiled, we can add *dummy* dependences to restrict the universality of the storage mapping, while maintaining tilability. In addition, we may assume that the dependences are all non-positive (for the tilable dimensions) as a result of pre-scheduling step to ensure the legality of tiling. For the remainder of this section, the "universe" of UOVs is one of the following restricted universes: fully tilable, fully sequential, and mixed sequential and tilable.

Note that the expectation is that "tilable" iteration spaces are tiled in a later phase. We analyze iteration spaces that will be tiled using QUOV, and then apply tiling. We also assume that the iteration points are scanned in the lexicographic order within a tile.

THEOREM 1 (SHORTEST QUOV, FULLY TILABLE). *Given a set of dependences $\mathcal{I}$ in a fully tilable space, the shortest QUOV u for tiled execution is the element-wise maxima of data-flow vectors of all dependences in $\mathcal{I}$.*

PROOF. Let the element-wise maxima of all data-flow vectors be the vector m, and f_m be a dependence with data-flow vector m. For unit vectors u^d in each of the d dimensions, we introduce *dummy* dependences f^d with data-flow vector u^d. Because these dependences have non-negative components, the resulting space is still tilable. For all dependences in $\mathcal{I}$ there exists a sequence of compositions with the dummy dependences to transitively express f_m. Using Lemma 1, the only dependence to be considered in UOV construction can therefore be reduced to f_m, which has the trivial UOV of m.

It remains to show that no QUOV shorter than m exists for the set of dependences $\mathcal{I}$. The shortest QUOV is defined

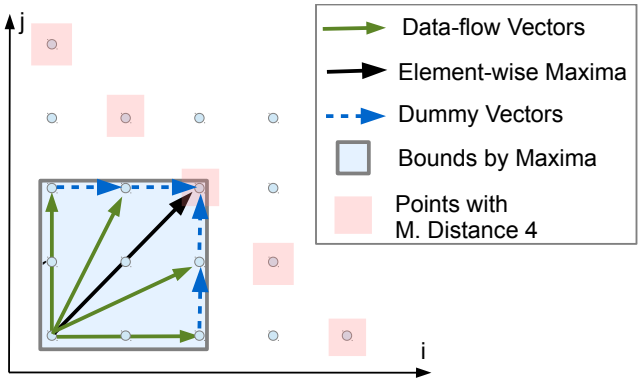


Figure 1: Illustration of Theorem 1 for the set of dependences with data-flow vectors $[2,0]$, $[2,1]$, $[1,2]$, and $[0,2]$. The element-wise maxima of the data-flow vectors correspond to the shortest UOV. The value produced by the bottom left iteration is used by the destination of the data-flow vectors. The data-flows induced by dummy dependences guarantees that the iteration pointed by the element-wise maxima is only executed after all iterations that depend on the bottom left. None of the other iterations with the same Manhattan distance (4) can be reached, since it requires backward data-flow along at least one of the axes.

by the closest[3] point from z that can be reached from all uses of z by following the dependences. Since the choice of z does not matter, let us use the origin, $\vec{0}$, to simplify our presentation. This allows us to use the data-flow vectors interchangeably with coordinate vectors.

Then, the hyper-rectangle with diagonal m includes all $\mathcal{I}$, and all bounds of the hyper-rectangle are touched by at least one dependence. Since all dependences in a fully tilable space are restricted to have non-negative data-flow vectors, no points within the hyper-rectangle can be reached by following dependences. Thus, it is clear that m is the closest common point that can be reached by those that touch the bounds. □

The theorem is illustrated in Figure 1, and is contrasted with the trivial UOV used by Strout et al. [19] in Figure 2.

The basic idea of inserting dummy dependences to restrict the possible schedule can be used beyond tilable schedules. One important corollary for sequential execution is the following.

Corollary 1 (Sequential Execution). *Given a set of dependences $\mathcal{I}$ in an n-dimensional space where lexicographic scan of the space is a legal schedule. Let m be the lexicographic maximum of the data-flow vectors of all dependences in $\mathcal{I}$. Then the shortest QUOV u for lexicographic execution is either m or the vector $[m_1+1, 0, \cdots, 0]$ where m_1 is the first element of m.*

Proof. For sequential execution, we may introduce dummy dependences to any lexicographically preceding point. Then the dependence, with a data-flow vector whose first element is m_1 can can subsume other dependences with lower values in the first element according to Lemma 1 by introducing appropriate dummy dependences.

[3]Shortest and closest are both in terms of Manhattan distance.

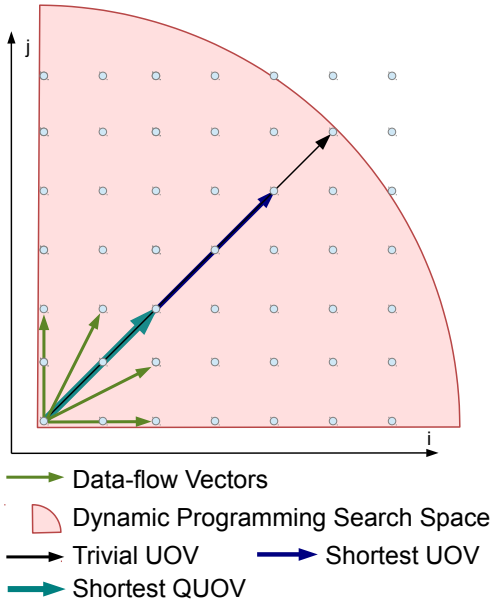


Figure 2: Comparison against the trivial UOV computed as proposed by Strout et al. [19] for the same set of dependences as in Figure 1. The trivial UOV is $[5,5]$ and it becomes the radius on the bounds of the search space for the dynamic programming algorithm proposed by Strout et al. [19]. In contrast, Theorem 1 gives the shortest QUOV by simply computing the element-wise maxima of the data-flow vectors. Furthermore, the shortest fully universal UOV is twice as long as the shortest QUOV, since the unit length dummy dependences cannot be assumed. The search space is bounded by a sphere since Euclidean distance is used by Strout et al. [19], but can be adapted to Manhattan distance.

For the remaining dependences there are two possibilities:

- We may use dummy dependences of the form $[1, *, \cdots, *]$ to let a dependence with data-flow vector $[m_1+1, 0, \cdots, 0]$ subsume all remaining dependences.
- We may use m as the QUOV following Theorem 1.

It is obvious that when the former options is used, $[m_1+1, 0, \cdots, 0]$ is the shortest. The optimality of the latter case follows from Theorem 1. Thus, the shortest UOV is the shortest among these two options. □

Note that m can be shorter than $[m_1+1, 0, \cdots, 0]$ only when $m = [m_1, 0, \cdots, 0]$. In addition, although the above corollary can be applied to tilable iteration spaces, the tilability may be lost due to memory-based dependences introduced by the allocation.

The allocation given by the above corollary is not schedule-independent at all. It is an analytical solution to the storage mapping of uniform dependence programs, where the schedule is the lexicographic scan of the iteration space.

The following corollary can trivially be established by the combination of the above.

Corollary 2 (Sequence of Tilable Spaces). *Given a set of dependences $\mathcal{I}$ in a space where a subset of*

```
for (i=0:N)
   S1[i] = foo();
for (j=0:N)
   S2[j] = bar(S1[j]);
```

(a) When $\theta_{S1} = (i \to 0, i, 0)$ and $\theta_{S2} = (j \to 1, j, 0)$.

```
for (i=0:N)
   S1[i] = foo();
   S2[i] = bar(S1[i]);
```

(b) When $\theta_{S1} = (i \to 0, i, 0)$ and $\theta_{S2} = (j \to 0, j, 1)$.

Figure 3: Two possible schedules for statements `S1` and `S2`. Note that statement `S1` in Figure 3a requires $O(N)$ memory whereas it only requires a scalar in Figure 3b, although the code shown is still in single assignment.

the dimensions are tilable, and lexicographic scan is legal for other dimensions, the shortest QUOV u for tiled execution of the tilable space, and sequential execution of the rest is the combination of vectors computed for each contiguous subset of either sequential or tilable spaces.

Note that the above corollary only takes effect when there are sequential subsets with at least two contiguous dimensions. When a single sequential dimension is surrounded by tilable dimensions, its element-wise maxima and lexicographic maxima are equivalent.

Using the above, the shortest QUOV for sequential, tiled, or a hybrid combination, can be computed very efficiently.

5. HANDLING OF PROGRAMS WITH MULTIPLE STATEMENTS

In many programs, there are multiple statements depending on each other. The original formulation of UOVs are for single statement programs [19]. In this section, we show that the concept can be adapted to multi-statement programs, knowing that we restrict the universe to tiled execution of the iteration space.

5.1 Limitations of UOV-based Allocation

Allocations based on UOVs have a strong property that they are valid for any legal schedule. Here, the schedule is not limited to affine schedules in the polyhedral model, and time stamps to each operation can be assigned arbitrarily, as long as they respect the dependences. The concept of UOV applies to reuse among writes to a single common space, and relies on the fact that every iteration writes to the same space (or array.) Different statements may write to different arrays, in programs with multiple statements, and hence UOV cannot be directly used.

For example, consider a program with two statements `S1` and `S2`:

- $\mathcal{D}_{S1} = \{i | 0 \leq i \leq N\}$
- $\mathcal{D}_{S2} = \{j | 0 \leq j \leq N\}$

where the dependence is such that iteration x of `S1` must executed before x of `S2`.

Figure 3 illustrates two possible schedules and its implications on memory usage. Note that we do not discuss storage mapping for $S2$, since it is not used within the code fragment above. A *fully universal* UOV-based allocation would have to take account for such variations of a schedule, but such extension may not even make sense. When two statements are scheduled differently, the dependence between two statements in the scheduled space may no longer be uniform.

When we apply the concept of UOV for tiling, we are no longer interested in arbitrary schedules. We first apply all non-tiling scheduling decisions before performing storage mapping. Therefore, the only change in the execution order comes from a tiling transformation viewed as a post-processing, so the same "schedule" applied to all statements involved. This allows us to extend the concept of UOVs to imperfectly nested loops.

5.2 Handling of Statement Ordering

When the ordering dimensions are represented as constant dimensions, the elements in the UOV require special handling. In the original formulation there is only one statement, and thus every iteration point writes to the same array. When multiple statements exist in a program, the iteration space of a statement is a subset of the combined space, and are made disjoint by statement ordering dimensions. Thus, not all points in the common space correspond to a write, and this affects how the UOV is interpreted.

Consider the program in Figure 3b. The only dependence involving `S1` has data-flow $[0, 0, 1]$, and since it is the only dependence, it is the shortest UOV. Literal interpretation of this vector as UOV means that the iteration $z + [0, 0, 1]$ can safely overwrite the value produced by z. However, this interpretation does not make sense in the context of by-statement allocation, since statement `S2` at $z+[0, 0, 1]$ writes to a different array.

We handle multi-statement programs by removing d out $d + 1$ constant dimensions for the purpose of UOV-based analysis. We apply the following rule to remove constant dimensions by transferring the information carried by these dimensions to others. Let v be a data-flow vector of a dependence in the scheduled space. We first apply the following rule to the vector v:

- For each constant dimension $x > 0$, where $v_x > 0$, set $v_{x-1} = max(v_{x-1}, 1)$

Once the above rule is applied, all the constant dimensions, except for the first one, are removed to obtain $d + 1$ dimensional data-flow vector. We repeatedly apply the above to all dependences in the program, resulting with a set of data-flow vectors with $d + 1$ dimensions.

We justify the above rule in the following. When the constant dimension of the data-flow is greater than 0, it means that some textually later statement uses the produced value. With respect to this particular dependence, the memory location may be reused once this textually later statement has been executed. However, there is always exactly one iteration of a specific statement in a constant dimension, since it is an artificial dimension for statement ordering. Therefore, the earliest possible iteration that can overwrite the memory location in question is in the next iteration of the immediate surrounding loop dimension.

For example, the only dependence of `S1` in Figure 3b is $[0, 0, 1]$. The value produced by `S1` at i is used by `S2` at i, but only overwritten by another iteration of `S1` at $i + 1$. Therefore, we transfer the dependence information to a

```
for (t=0:T)
   for (i=0:N)
      A[i] = foo(A[i]);              //S1
   for (i=1:N)
      A[i] = bar(A[i-1], A[i]);      //S2
```

Figure 4: Example to illustrate influences of boundary cases. Note that the value produced by `S1` is last used by `S2` of the same outer loop iteration, except for when $i = 0$, in which case it is used again by `S1` at $[t + 1, 0]$.

preceding dimension.

Projecting a $d+1$ dimensional domain along a vector does indeed produce a domain with d dimensions. This is because in the presence of imperfect nests, d dimensional storage may be required for d dimensional statements even with uniform dependences. Consider the code in Figure 3a. The only use of `S1` is by `S2` with data-flow $[1, 0, 0]$ in the scheduled space. Applying the rule described above removes the last dimension, yielding $[1, 0]$ as the QUOV. Projecting the statement domain of `S1` (in the scheduled space with d constant dimensions removed) along the vector $[1, 0]$ gives a two-dimensional domain: $\{i, x | 0 \leq i \leq N \wedge x = 0\}$. Although the domain is two-dimensional, it is effectively one-dimensional because of the equality in the constant dimension.

It is also important to note a special case when the UOV takes the form: $[0, \cdots, 0, 1]$. When the last constant dimension is the only non-zero entry, it is obvious that the statement requires only a scalar, since its immediately consumed.

6. UOV GUIDED INDEX SET SPLITTING

In the polyhedral representation of programs there are usually boundary cases that behave differently from the rest. For instance, the first iteration of a loop may read from inputs, whereas successive iterations use values computed by previous iterations.

In the polyhedral model, storage mapping is usually computed for each statement. With pseudo-projective allocations, the same allocation must be used for all points in the statement domain. Thus, dependences that only exist at the boundaries influence the entire allocation.

For example, consider the code in Figure 4. The value produced by `S1` at $[t, i]$ is last used by `S2` at $[t, i+1]$ for $i > 0$. However, the value produced by `S1` at $[t, 0]$ is last used by `S1` at $[t + 1, 0]$. Thus, the storage mapping for `S1` must ensure that a value produced at $[t, i]$ is live until $[t + 1, i]$ for all instances of `S1`. This clearly leads to wasteful allocations, and our goal is to avoid them.

One solution to the problem is to apply a form of *index set splitting* [9] such that the boundary cases and common cases have different mappings. In the example above, we wish to use piece-wise mappings for `S1` at $[t, i]$ where the mapping is different for two disjoint sets $i = 0$ and $i > 0$. This reduces the memory usage from an array of size $N + 1$ to 2 scalars.

Once, the *pieces* are computed, application of piece-wise mappings can done through splitting the statements (nodes) as defined by the pieces, and then applying a separate mapping to each of the statements after split. Thus, the only interesting problem that remain is finding meaningful splits.

In this section, we present an algorithm for finding the split with the goal of minimizing memory usage. The algorithm is guided by Universal Occupancy Vectors and works best with UOV-based allocations. The goal of our index set splitting is to isolate boundaries that require longer lifetime than the main body. Thus, we are interested in a subdomain of a statement with a different dependence pattern than that in the rest of the statement's domain. We focus on boundary domains that contain at least one equality. The approach may be generalized to boundary planes of constant thickness using Thick Face Lattices [11].

The original index-set splitting [9] aimed at finding better schedules. The quality of a split is measured by its influence on possible schedules: whether different scheduling functions for each piece yields a better schedule.

In our case, the goal is different. Our starting point is the program after affine scheduling, and we are now interested in finding storage mappings. When the dependence pattern is the same at all points in the domain, splitting cannot improve the quality of the storage mapping. Since the dependence pattern is the same, the same storage mapping will be used for all pieces (with a possible exception of the cases when the split introduces equalities or other properties related to shape of the domains). Because two points that may have been in the nullspace of the projection may now be split into different pieces, the number of points that can share the same memory location may be reduced as the result of splitting.

Thus, as a general measure of quality, we seek to ensure that a split influences the choice of storage mapping for each piece. The obvious case when splitting is useless is when a dependence function at a boundary is also in the main part. We present Algorithm 1 based on this intuition to reduce the number of splits.

The intuition of the algorithm is that we start with all dependences with equalities in their domain as candidate pieces. Then we remove some of the dependences where splitting does not improve the allocation from candidate pieces. The obvious case is when the same dependence function exists in the non-boundary cases (i.e., dependences with no equalities in their domain). In addition, more sophisticated exclusion is performed using Theorem 1.

The algorithm can also be easily adapted for non-uniform programs. It may also be specialized/generalized by adding more dependence elimination rules to Step 2. This requires a method similar to Lemma 1 for other memory allocation methods.

Example

Let us describe the algorithm in more detail with an example. The statement $S1$ in Figure 4 has three data-flow dependences:

- $\mathcal{I}_1 = S1[t, i] \rightarrow S2[t, i]$ when $0 \leq i \leq N$
- $\mathcal{I}_2 = S1[t, i] \rightarrow S2[t, i + 1]$ when $i < N$
- $\mathcal{I}_3 = S1[t, i] \rightarrow S1[t + 1, i]$ when $i = 0$

The need for index-set splitting does not arise until some prior scheduling fuses the two inner loops. Let the scheduling functions be:

- $\theta_{S1} = (t, i \rightarrow 0, t, 0, i, 0)$
- $\theta_{S2} = (t, i \rightarrow 0, t, 0, i, 1)$

Algorithm 1 UOV-Guided Split

Input:

$\mathcal{I}$: Set of uniform dependences that depend on a statement S. A dependence is a pair $\langle f, D\rangle$ where f is the dependence function and D is a domain. The domain is the constraints on the producer statement.

Output:

$\mathcal{P}$: A partition of D_S, the domain of S, where each element defines a piece of the split.

Algorithm:
We first inspect the domain of dependences in $\mathcal{I}$ to detect equalities.
Let

$\mathcal{I}_b$ be the set of dependences with equalities, and

$\mathcal{I}_m$ be the set of those without equalities.

Then,

1. **foreach** $\langle f, D\rangle \in \mathcal{I}_b$,

   if $\exists \langle g, E\rangle \in \mathcal{I}_m; f = g$ **then** remove $\langle f, D\rangle$ from $\mathcal{I}_b$

2. Further remove dependences from $\mathcal{I}_b$ using the following *if applicable*:

   (a) Theorem 1 and its corollaries. The following steps are only for Theorem 1.

   Let m be the element-wise maxima of dataflow vectors in $\mathcal{I}_m$.
   foreach $\langle f, D\rangle \in \mathcal{I}_b$
   - Let v be the dataflow vector of f.
   - **if** $\forall_i : v_i \leq m_i$ **then** remove $\langle f, D\rangle$ from $\mathcal{I}_b$.

3. Group the remaining dependences in $\mathcal{I}_b$ into groups $\mathcal{G}_i$, $0 \leq i \leq n$, where $\forall X, Y \in \mathcal{G}_i; \mathcal{D}_X \cap \mathcal{D}_Y \neq \emptyset$. In other words, group the dependences with overlapping domains in the producer space.

4. **foreach** $i \in 0 \leq i \leq n$, $\mathcal{P}_i = \bigcup_{\forall X \in \mathcal{G}_i} \mathcal{D}_X$

5. **if** $n \geq 0$ **then** $\mathcal{P}_{n+1} = \mathcal{D}_S \setminus \bigcup_{i=0}^{n} \mathcal{P}_i$ **else** $\mathcal{P}_0 = \mathcal{D}_S$

The data-flow vectors in the scheduled space, after removing constant dimensions (we also remove the outer-most one since there is only one loop at the outer-most level) are:

- $\mathcal{I}_1^{'} = [0, 1]$ when $0 \leq i \leq N$
- $\mathcal{I}_2^{'} = [0, 1]$ when $i < N$
- $\mathcal{I}_3^{'} = [1, 0]$ when $i = 0$

As the pre-processing step, we separate the dependences into two sets based on the equalities in the domain:

- $\mathcal{I}_b = \{\mathcal{I}_3^{'}\}$; those with equalities, and
- $\mathcal{I}_m = \{\mathcal{I}_1^{'}, \mathcal{I}_2^{'}\}$; those without equalities.

Then we use Step 1 to eliminate identical dependences. If the same dependence function is both in the boundary and the main body, separating the boundary does not reduce the number of distinct dependences to be considered. Therefore, splitting such domain does not positively influence the storage mapping, and hence is removed from further consideration. Since $\mathcal{I}_3^{'}$ is different from the other two, this step does not change the set of dependences for this example.

Step 2a is the core of this algorithm. The general idea is the same as Step 1, but we use additional properties of UOV-based allocation to identify more cases where splitting does not positively influence the storage mapping.

Theorem 1 states that if all elements of a data-flow vector are less than the corresponding element of the element-wise maxima of all data-flow vectors under consideration, the dependence does not influence shortest QUOV. Therefore, if the candidate dependence to split does not contribute to the element-wise maxima, the split is useless in terms of further shortening the QUOV. As an illustration, consider the rectangle in Figure 1 defined by the element-wise maxima. If separating a dependence does not shrink the size of the rectangle, the length of QUOV cannot be shortened.

In this example, the element-wise maxima of dependences in $\mathcal{I}_m$ is $[0, 1]$. However, the boundary dependence has data-flow vector $[1, 0]$, and when combined, the element-wise maxima becomes $[1, 1]$. Therefore, the boundary dependence does contribute to the element-wise maxima, and is not removed from the candidate set in Step 2a. The dependences that are left in the set $\mathcal{I}_b$ after this step are the set of dependences that will be split from the main body.

Step 3 is a grouping step, where dependences to be split are grouped into those that have overlap in their domains.

Two sub-domains of a statement cannot be split into separate statements, unless the computation is duplicated. Although computing the values redundantly may be an option, we enforce the two statements to be jointly split as one additional statement. This step is irrelevant for our example, since our example only has one dependence in set $\mathcal{I}_b$.

The last step is a cleanup step, which adds the remainder of splits to the set of partitions.

Let `S1b` be the statement after splitting the domain of $\mathcal{I}_3$ from `S1`, and `S1a` be the remainder of the main body of `S1`. The domain of statements in the program are now:

- $\mathcal{D}_{S1a} = \{t, i | 0 \leq t \leq T \wedge 1 \leq i \leq N\}$
- $\mathcal{D}_{S1b} = \{t, i | 0 \leq t \leq T \wedge i = 0\}$
- $\mathcal{D}_{S2} = \{t, i | 0 \leq t \leq T \wedge 1 \leq i \leq N\}$

and the dependences are:

- $\mathcal{I}_1^* = S1a[t, i] \rightarrow S2[t, i]$ when $1 \leq i \leq N$
- $\mathcal{I}_2^* = S1a[t, i] \rightarrow S2[t, i+1]$ when $i < N$
- $\mathcal{I}_3^* = S1b[t, i] \rightarrow S1a[t+1, i]$ when $i = 0$

The QUOV for `S1a` is $[0, 0, 0, 0, 1]$ in the scheduled space with constant dimensions, which is the aforementioned special case, and only a scalar is required for `S1a`. The QUOV for `S1b` is $[1, 0]$ after removing the constant dimensions. The projection of its domain along this vector is also a constant, due to the equality in its domain. Thus, the storage requirement for `S1` in the original program becomes two scalars, which is much smaller than what is required without the splitting.

7. RELATED WORK

There is a lot of prior work on storage mappings for polyhedral programs [1, 4, 5, 13, 15, 19, 20, 21]. Most approaches focus on the case when the schedule for statements are given.

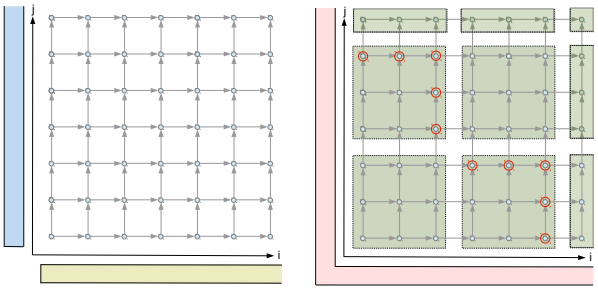
7.1 Efficiency of UOV-based Allocation

By the nature of its strategy, UOV-based allocation, including those using QUOVs, cannot yield more compact storage mappings compared to alternative strategies for a specific schedule. However, UOV-based allocation may not be as inefficient as one might think for programs that require $d-1$ dimensional storage. The misconception is (at least partially) due to the trade-off between memory usage and parallelism that is often overlooked. Consider the following code fragment with Smith-Waterman(-like) dependences.

```
for (i=1:N)
   for (j=1:M)
       H[i,j] = foo(H[i-1,j], H[i,j-1]);
```

As illustrated in Figure 5, UOV-based allocation, even with QUOVs, gives a storage mapping that use $O(N+M)$ memory for this program. However, the program cannot be tiled if $O(N)$ or $O(M)$ storage mappings are used due to memory-based dependences. One approach to still accomplish tiled execution of this program is to transform the program such that the iteration space is tilable even when the memory-based dependences are under consideration.

For the above program, this requires a skewing as depicted in Figure 6. Once the skewing is applied so that $O(M)$ storage mapping can be tiled, the UOV-based allocation for the same program will also yield a storage mapping with $O(M)$ memory. Note that the skewed iteration space has less parallelism (longer critical path length) when compared to the original rectangular iteration space. The parallelism is effectively traded off with decreased memory usage.



(a) Iteration space and possible storage mappings

(b) Set of live values in tiled execution

Figure 5: Iteration space of Smith-Waterman(-like) code and possible storage mappings. Figure 5a shows two possible storage mappings, either horizontal or vertical projection of the iteration space. The size of memory is $O(M)$ or $O(N)$, depending of the direction of the projection. Figure 5b shows the iteration space after tiling, and the values that are live after executing two tiles on the diagonal. Observe that neither projection (horizontal or vertical) can store all the live values. One example of a valid projective storage mapping is the projection along $[1, 1]$ that use $O(N+M)$ memory.

For this example, when the other condition (amount of parallelism) is equal, the allocation using UOVs is no worse than what is considered a more efficient allocation. This observation can be generalized to other instances of uniform dependence programs, such as Jacobi/Gauss-Seidel stencils. How to jointly find schedule and storage mappings to explore such trade-off is still an open problem.

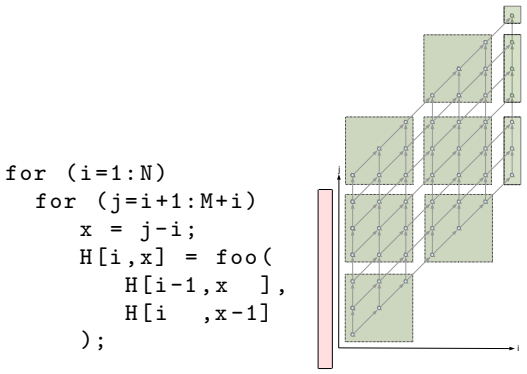
7.2 Optimality of Projective Allocations

There is also an upper bound on dimension-wise optimality. Quilleré and Rajopadhye [15] show that the number of linearly independent projection vectors can be viewed as the primary criterion for optimality of storage mappings.

UOV-based allocation, as originally defined, was limited to allocations with one projection vector by its nature, and therefore, is limited to finding $d-1$ dimensional storage for d dimensional iteration space. The additional optimizations we describe in Section 5 allow us to overcome this limitation, but for many, if not most, uniform dependence programs, the lower bound on the number of memory dimensions is $d-1$. Therefore, UOV-based allocations are no more than a constant fold more expensive.

Now, the constant factor can become important, but Quilleré and Rajopadhye [15] give a number of examples to illustrate that the problem is subtle when the size of the iteration domain is parameterized. For some programs Lefebvre-Feautrier [13] gives a better memory footprint than the Quilleré-Rajopadhye method, while for others, it is worse.

It is easy to show that any multiple, by some integer greater than one, of a legal UOV uses more memory. Two



(a) Code (b) Iteration space

Figure 6: The code and iteration space after skewing the original program. It is easy to see that $[1, 1]$ is the shortest UOV for this program. With this UOV, the amount of memory used is $O(M)$.

UOVs that are not constant multiples of each other are often difficult to compare. For example, memory usage of two allocations based on UOVs $[1, 1]$ and $[2, 0]$ are only parametrically comparable. With $N \times M$ iteration space, the former use $N + M$ and the latter use $2N$. The optimal allocation in such case depends on the values of N and M that are not known until run-time.

Informally, increasing the Manhattan distance will always increase memory usage by either increasing the GCD, and hence increasing the mod factor, or by increasing the angle of the projection, and hence increasing the size of the projected space.

7.3 Parametric Tile Sizes

Our main motivation for QUOVs is to find storage mappings that are valid for tiled execution by any (legal) tile sizes. Schedule-dependent approaches cannot provide storage mappings for parametric tile sizes due to its non-affine nature.

It is possible to provide tile coordinates and tile sizes as additional parameters to the polyhedral representation, and then apply polyhedral storage mappings for each tile individually. Although this approach makes sense in certain contexts (e.g., [10]), it is not suitable for others (e.g., shared memory parallelization.) As shown in Figure 6, UOV-based allocation maps iterations from different tiles to a single memory location, allowing inter-tile reuse of storage. There is no need to transfer data from one tile to another in UOV-based allocation.

7.4 Affine Occupancy Vectors

Thies et al. [20] present an extension to the concept of UOV to affine schedules, named Affine Occupancy Vectors. They restrict the universality to affine scheduling, rather than the full universe. Although the idea of restricting the universality has some similarities with our work, the restricted universe is still the entire affine scheduling space. In addition, they only handle one-dimensional affine schedules.

8. CONCLUSIONS

We have presented a series of extensions to the Schedule-Independent Storage Mapping. Although UOVs were originally used for schedule-independent mappings, our extensions restrict the universality of the occupancy vectors to analyze a specific class of schedules; tiling.

For such a restricted universe, Quasi-UOVs can be shorter than fully universal ones, leading to more compact memory. We can also take advantage of its properties to directly find the shortest QUOV.

Although UOV-based allocations are limited to uniform dependence programs, storage mappings that are legal for a class of schedules is an interesting alternative to most memory allocation methods that require schedules to be given.

Our extensions aim to make UOV-based allocations more practical by providing efficient method for finding the shortest UOV for a smaller, but an important universe, tilable programs.

9. 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 *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Language, Compiler and Tool Support for Embedded Systems*, volume 13, pages 73–82, 2007.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th IEEE International Conference on Parallel Architecture and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[4] Y. Bouchebaba and F. Coelho. Tiling and memory reuse for sequences of nested loops. In *Proceedings of the 8th International Euro-Par Conference*, volume 2400, page 255, 2002.

[5] A. Cohen. Parallelization via constrained storage mapping optimization. In *Proceedings of the International Symposium on High Performance Computing*, pages 83–94, 1999.

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

[7] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

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

[9] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[10] S. Guelton, A. Guinet, and R. Keryell. Building retargetable and efficient compilers for multimedia instruction sets. In *2011 International Conference on*

Parallel Architectures and Compilation Techniques, pages 169–170, 2011.

[11] G. Gupta and S. Rajopadhye. Simplifying reductions. In *Proceedings of the 33rd ACM Conference on Principles of Programming Languages*, PoPL '06, pages 30–41, New York, NY, USA, Dec 2006. ACM.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '88, pages 319–329. ACM, 1988.

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

[14] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

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

[16] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[17] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2 of *ICPP '90*, pages 179–186, 1990.

[18] R. Schreiber and J. J. Dongarra. Automatic blocking of nested loops. Technical report, 1990.

[19] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. *ACM SIGOPS Operating Systems Review*, 32(5):24–33, 1998.

[20] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation*, PLDI '01, pages 232–242. ACM, 2001.

[21] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Proceedings of the 2nd International Euro-Par Conference*, pages 389–397, 1996.

[22] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.