A Unified Framework for Schedule and Storage Optimization*

William Thies†, Frédéric Vivien§, Jeffrey Sheldon†, and Saman Amarasinghe†

†Laboratory For Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{thies,jeffshel,saman}@lcs.mit.edu

§ICPS/LSIIT
Université Louis Pasteur
Strasbourg, France
vivien@icps.u-strasbg.fr

Abstract

We present a unified mathematical framework for analyzing the tradeoffs between parallelism and storage allocation within a parallelizing compiler. Using this framework, we show how to find a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid for all legal schedules. We 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 schedule. Our technique combines affine scheduling techniques with occupancy vector analysis and incorporates general affine dependences across statements and loop nests. We formulate the constraints imposed by the data dependences and storage mappings as a set of linear inequalities, and apply numerical programming techniques to efficiently solve for the shortest occupancy vector. We consider our method to be a first step towards automating a procedure that finds the optimal tradeoff between parallelism and storage space.

1 Introduction

It remains an important and relevant problem in computer science to automatically find an efficient mapping of a sequential program onto a parallel architecture. Though there are many heuristic algorithms in practical systems and partial or suboptimal solutions in the literature, a theoretical framework that can fully describe the entire problem and find the optimal solution is still lacking. The difficulty stems from the fact that multiple inter-related costs and constraints must be considered simultaneously to obtain an efficient executable.

While exploiting the parallelism of a program is an important step towards achieving efficiency, gains in parallelism are often overwhelmed by other costs relating to data locality, synchronization, and communication. In particular, with the widening gap between clock speed and memory latency, and with modern memory systems becoming increasingly hierarchical, the amount of storage space required by a program can have a drastic effect on its performance. Nonetheless, parallelizing compilers often employ varying degrees of array expansion [9, 5, 1] to eliminate element-level anti and output dependences, thereby adding large amounts of storage that may or may not be justified by the resulting gains in parallelism.

Thus, compilers must be able to analyze the tradeoffs between parallelism and storage requirements in order to arrive at an efficient executable. In this paper, we introduce a unifying mathematical framework that incorporates both schedule constraints (restricting *when* statements can be executed) and storage constraints (restricting *where* their results can be stored). We 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 schedule. Our technique incorporates general affine dependences across statements and loop nests, making it applicable to many scientific applications

Using this technique, we present solutions to three important scheduling problems. Namely, we show how to determine 1) a good storage mapping for a given schedule, 2) a good schedule for a given storage mapping, and 3) a good storage mapping that is valid for all legal schedules. Our method is precise and practical in that it reduces to a linear program that can be efficiently solved with standard techniques. We believe that these solutions represent the first step towards automating a procedure that finds the optimal compromise between parallelism and storage space.

The rest of this paper is organized as follows. In Section 2 we motivate the problem abstractly, and in Section 3 we define it concretely. Section 4 formulates the method abstractly, and Section 5 illustrates the method with examples. Experiments are described in Section 6, related work in Secion 7, and we conclude in Section 8.

2 Abstract Problem

To motivate our approach, we consider simplified descriptions of the scheduling problems faced by a parallelizing compiler. We are given a directed acyclic graph $G = (V, E)$. Each vertex $v \in V$ represents a dynamic instance of an instruction; a value will be produced as a result of executing v. Each edge $(v_1, v_2) \in E$ represents a dependence of v_2 on the value produced by v_1. Thus, each edge (v_1, v_2) imposes the *schedule constraint* that v_1 be executed before v_2, and the *storage constraint* that the value produced by v_1 be stored until the execution time of v_2.

*This research was done while Frédéric Vivien was a Visiting Professor in the MIT Laboratory for Computer Science. More information on this project can be found at http://compiler.lcs.mit.edu/aov.

PLDI 2001, Snowbird, Utah, USA.

```
A[][] = new int[n][m]
...
for j = 1 to m
  for i = 1 to n
    A[i][j] = f(A[i-2][j-1], A[i][j-1], A[i+1][j-1])
```

Figure 1: Original code for Example 1.

Our task is to output (Θ, m), where Θ is a function mapping each operation $v \in V$ to its execution time, and m is the maximum number of values that we need to store at a given time. Parallelism is expressed implicitly by assigning the same execution time to multiple operations. To simplify the problem, we ignore the question of how the values are mapped to storage cells and assume that live values are stored in a fully associative map of size m. How, then, might we go about choosing Θ and m?

2.1 Choosing a Store Given a Schedule

The first problem is to find the optimal storage mapping for a given schedule. That is, we are given Θ and choose m such that 1) (Θ, m) respects the storage constraints, and 2) m is as small as possible.

This problem is orthogonal to the traditional loop parallelization problem. After selecting the instruction schedule by any of the existing techniques, we are interested in identifying the best storage allocation. That is, with schedule-specific storage optimization we can build upon the performance gains of any one of the many scheduling techniques available to the parallelizing compiler.

2.2 Choosing a Schedule Given a Store

The second problem is to find an optimal schedule for a given size of the store, if any valid schedule exists. That is, we are given m and choose Θ such that 1) (Θ, m) respects the schedule and storage constraints, and 2) Θ assigns the earliest possible execution time to each instruction. Note that if m is too small, there might not exist a Θ that respects the constraints.

This is a very relevant problem in practice because of the stepwise, non-linear effect of storage size on execution time. For example, when the storage required cannot be accommodated within the register file or the cache, and has to resort to the cache or the external DRAM, respectively, the cost of storage increases dramatically. Further, since there are only a few discrete storage spaces in the memory hierarchy, and their size is known for a given architecture, the compiler can adopt the strategy of trying to restrict the store to successively smaller spaces until no valid schedule exists. Once the storage is at the lowest possible level, the schedule could then be shortened, having a more continuous and linear effect on efficiency than the storage optimization. In the end, we end up with a near-optimal storage allocation and instruction schedule.

2.3 Choosing a Store for all Schedules

The final problem is to find the optimal storage mapping that is valid for all legal schedules. That is, we are given a (possibly infinite) set $\Psi = \{\Theta_1, \Theta_2, \dots\}$, where each Θ in Ψ respects the schedule constraints. We choose m such that 1) $\forall \Theta \in \Psi$, (Θ, m) respects the storage constraints, and 2) m is as small as possible.

```
A[] = new int[n]
...
for j = 1 to m
  for i = 1 to n
    A[i] = f(A[i-2], A[i], A[i+1])
```

Figure 2: Transformed code for Example 1. The occupancy vector is (0,1).

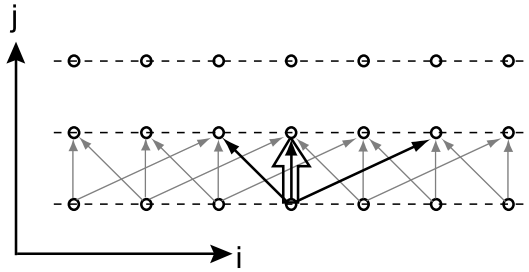

Figure 3: Iteration space diagram for Example 1. Given the schedule where each row is executed in parallel, our method identifies $(0, 1)$ as the shortest valid occupancy vector.

A solution to this problem allows us to have the minimum storage requirements without sacrificing any flexibility of our scheduling. For instance, we could first apply our storage mapping, and then arrange the schedule to optimize for data locality, synchronization, or communication, without worrying about violating the storage constraints.

Such flexibility could be critical if, for example, we want to apply loop tiling [10] in conjunction with storage optimization. If we optimize storage too much, tiling could become illegal; however, we sacrifice efficiency if we don't optimize storage at all. Thus, we optimize storage as much as we can without invalidating a schedule that was valid under the original storage mapping.

More generally, if our analysis indicates that certain schedules are undesirable by any measure, we could add edges to the dependence graph and solve again for the smallest m sufficient for all the remaining candidate schedules. In this way, m provides the best storage option that is legal across the entire set of schedules under consideration.

3 Concrete Problem

Unfortunately, the domain of real programs does not lend itself to the simple DAG representation as presented above. Primarily, loop bounds in programs are often specified by symbolic expressions instead of constants, thereby yielding a parameterized and infinite dependence graph. Furthermore, even when the constants are known, the problem sizes are too large for schedule and storage analysis on a DAG, and the executable grows to an infeasible size if a static instruction is generated for every node in the DAG.

Accordingly, we make two sets of simplifying assumptions to make our analysis tractable. The first concerns the nature of the dependence graph G and the scheduling function Θ. Instead of allowing arbitrary edge relationships and execution orderings, we restrict our attention to affine dependences and affine schedules. The second assumption concerns our approach to the optimized storage mapping. Instead of allowing a fully associative map of size m, as above, we employ the *occupancy vector* as a mechanism of storage reuse. In the following sections, we discuss these assumptions in the context of an example.

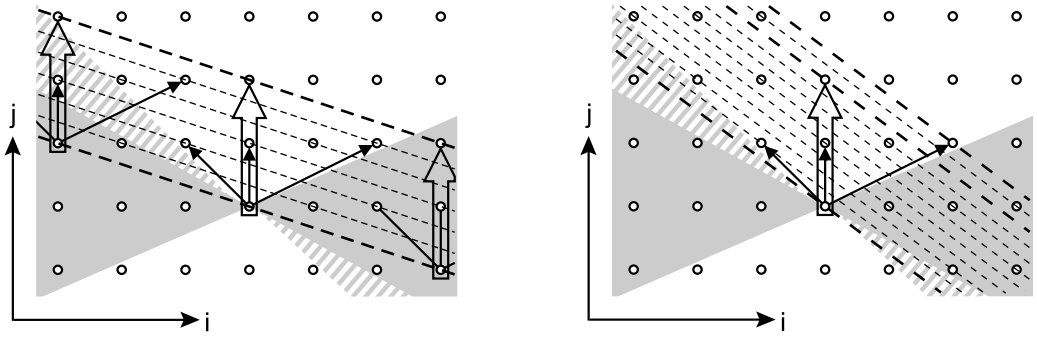

Figure 4: Iteration space diagram for Example 1. Given an occupancy vector of $(0, 2)$, our method identifies the range of valid schedules. An affine schedule will sweep across the space, executing a line of iterations at once. If this line falls within the gray region (as on the left), then the schedule is valid for the occupancy vector of $(0, 2)$. If this line falls within the striped region (as on the right) then the schedule is valid for some occupancy vector other than $(0, 2)$. The schedule at right is invalid because the operation at the tip of the occupancy vector $(0, 2)$ overwrites a value before the operation at $(2, 1)$ can consume it.

3.1 Program Domain

Primarily, we require an affine description of the dependences of the program. This formulation gives an accurate description of the dependences of programs with static control flow and affine index expressions [6] and can be estimated conservatively for others. As will become clear below, restricting our attention to affine dependences allows us to model the infinite dependence graph as a finite set of parameters, which is central to the method.

In this paper, we further assume a single-assignment form where the iteration space of each statement exactly corresponds with the data space of the array written by that statement. That is, for array references appearing on the left hand side of a statement, the expression indexing the i'th dimension of the array is the index variable of the i'th enclosing loop (this is formalized below). While techniques such as array expansion [5] can be used to convert programs with affine dependences into this form, our analysis will be most useful in cases where an expanded form was obtained for other reasons (e.g., to detect parallelism) and one now seeks to reduce storage requirements.

We will refer to the example in Figure 1, borrowed from [17]. It clearly falls within our input domain, as the dependences have constant distance, and iteration (i, j) assigns to $A[i][j]$. This example represents a computation where a one-dimensional array $A[i]$ is being updated over a time dimension j, and the intermediate results are being stored. We assume that only the element $A[n][m]$ is used outside the loop; the other values are only temporary.

3.2 Occupancy Vectors

To arrive at a simple model of storage reuse, we borrow the notion of an *occupancy vector* from Strout et al. [17]. The strategy is to reduce storage requirements by defining equivalence classes over the locations of an array. Following a storage transformation, all members of a given equivalence class in the original array will be mapped to the same location in the new array. The equivalence relation is:

$$R_{\vec{v}} = \{(\vec{l}_1, \vec{l}_2) \mid \exists k \in \mathbb{Z} \; s.t. \; \vec{l}_1 = \vec{l}_2 + k \cdot \vec{v}\}$$

and we refer to $\vec{v}$ as the *occupancy vector*. We say that A' is the result of *transforming* A under the occupancy vector $\vec{v}$ if, for all pairs of locations $(\vec{l}_1, \vec{l}_2)$ in A:

$$R_{\vec{v}}(\vec{l}_1, \vec{l}_2) \iff \vec{l}_1 \textit{ and } \vec{l}_2 \textit{ are stored in same location in } A'$$

We say that an occupancy vector $\vec{v}$ is *valid* for an array A with respect to a given schedule Θ if transforming A under $\vec{v}$ everywhere in the program does not change the semantics when the program is executed according to Θ.

Given an occupancy vector, we implement the storage transformation using the technique of [17] in which the original data space is projected onto the hyperplane perpendicular to the occupancy vector. If an occupancy vector intersects multiple (integral) points of the data space, then modulation must be used to distinguish these points in the transformed array.

Occupancy vector transformations are useful for reducing storage requirements when many of the values stored in the array are temporary. Generally, shorter occupancy vectors lead to smaller storage requirements because more elements of the original array are coalesced into the same storage location. However, the shape of the array also has the potential to influence the transformed storage requirements. Throughout this paper, we assume that the shapes of arrays have second-order effects on storage requirements, and we refer to the "best" occupancy vector as that which is the shortest.

We are now in a position to consider our occupancy vector analysis as applied to Example 1. First, assume that we have chosen to execute each row in parallel so as to have the shortest schedule. What is the best storage mapping for this schedule? Our method can identify $(0, 1)$ as the shortest occupancy vector for this schedule (see Figure 3), yielding the code in Figure 2.

Secondly, consider the case where we become interested in adding some flexibility to our scheduling. If we lengthen the occupancy vector to $(0, 2)$, what is the range of schedules that we can consider? As illustrated in Figure 4, our method can identify all legal affine schedules for the occupancy vector of $(0, 2)$. We could then use affine scheduling techniques [7] to choose amongst these schedules according to other criteria.

3.3 Affine Occupancy Vectors

Finally, we might inquire as to the shortest occupancy vector that is valid for all affine schedules in Example 1. An affine

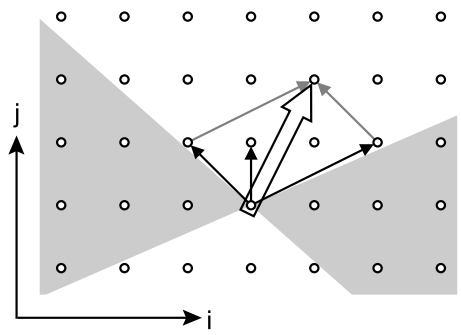

Figure 5: Iteration space diagram for Example 1. Here the hollow arrow denotes an Affine Occupancy Vector that is valid for all legal affine schedules. The gray region indicates the slopes at which a legal affine schedule can sweep across the iteration domain.

```
A[] = new int[2*n+m]
...
for j = 1 to m
  for i = 1 to n
    A[2*i-j+m] = f(A[2*(i-2)-(j-1)+m],
                   A[2*i-(j-1)+m],
                   A[2*(i+1)-(j-1)+m])
```

Figure 6: Transformed code for Example 1. The AOV is (1,2).

schedule is one where each dynamic instance of a statement is executed at a time that is an affine expression of the loop indices, loop bounds, and compile-time constants. To address the problem, then, we need the notion of an Affine Occupancy Vector:

Definition 1 *An occupancy vector $\vec{v}$ for array A is an Affine Occupancy Vector (AOV) if it is valid with respect to every affine schedule Θ that respects the schedule constraints of the original program.*

Note that, in contrast to the Universal Occupancy Vector of [17], an AOV need not be valid for *all* schedules; rather, it only must be valid for affine ones. Almost all the instruction schedules found in practice are affine, since any FOR loop with constant increment and bounds defines a schedule that is affine in its loop indices. (This is independent of the *array references* found in practice, which are sometimes non-affine.) In this paper, we further relax the definition of an AOV to those occupancy vectors which are valid for all *one-dimensional*[1] affine schedules.

We also observe that, if tiling is legal in the original program, then tiling is legal after transforming each array in the program under one of its AOV's. This follows from the fact that two loops are tilable if and only if they can be permuted without affecting the semantics of the program [10]. Since each permutation of the loops corresponds to a given affine schedule and the AOV is valid with respect to both schedules, the AOV transformation is also valid with respect to a tiled schedule.

Returning to our example, we find using our method that $(1, 2)$ is a valid AOV (see Figure 5). Any affine one-dimensional schedule that respects the dependences in the original code will give the same result when executed with the transformed storage.

[1] A one-dimensional affine schedule assigns a scalar execution time to each operation as an affine function of the enclosing loop indices and symbolic constants. Multi-dimensional schedules assign vector-valued execution times, which are ordered lexicographically; certain programs require multi-dimensional schedules. See [7, 8, 4] for details.

4 The Method

4.1 Notation

We adopt the following notation:

- An iteration vector $\vec{i}$ contains the values of surrounding loop indices at a given point in the execution of the program.
- The structural parameters $\vec{n}$, of domain $\mathcal{N}$, represent loop bounds and other parameters that are unknown at compile time, but that are fixed for any given execution of the program.
- There are n_s statements $S_1 \ldots S_{n_s}$ in the program. Each statement S has an associated polyhedral domain $\mathcal{D}_S$, such that $\forall \vec{i} \in \mathcal{D}_S$, there is a dynamic instance $S(\vec{i})$ of statement S at iteration $\vec{i}$ during the execution of the program.
- With each statement S is associated a scheduling function θ_S which maps the instance of S on iteration $\vec{i}$ to a scalar execution time. By assumption, θ_S is an affine function of the iteration vector and the structural parameters: $\theta_S(\vec{i}, \vec{n}) = \vec{a}_S \cdot \vec{i} + \vec{b}_S \cdot \vec{n} + c_S$. The schedule for the entire program is denoted by $\Theta \in \mathcal{E}$, where $\mathcal{E}$ is the space of all the scheduling parameters $(\vec{a}_{S_1}, \vec{b}_{S_1}, \vec{c}_{S_1}), \ldots, (\vec{a}_{S_{n_s}}, \vec{b}_{S_{n_s}}, \vec{c}_{S_{n_s}})$.
- There are n_p dependences $P_1 \ldots P_{n_p}$ in the program. Each dependence P_j is a 4-tuple $(R_j, T_j, \mathcal{P}_j, h_j)$ where R_j and T_j are statements, h_j is a vector-valued affine function, and $\mathcal{P}_j \subseteq \mathcal{D}_{R_j}$ is a polyhedron such that:

  $$\forall \vec{i} \in \mathcal{P}_j, R_j(\vec{i}) \textit{ depends on } T_j(\vec{h}_j(\vec{i}, \vec{n})) \qquad (1)$$

  The dependences P_j are determined using an array dataflow analysis, e.g., [6] or the Omega test [15].
- There are n_a arrays $A_1 \ldots A_{n_a}$ in the program, and $A(S)$ denotes the array assigned to by statement S. Our assumption that the data space corresponds with the iteration space implies that for all statements S, $S(\vec{i})$ writes to location $\vec{i}$ of $A(S)$, and S is the only statement writing to A. However, each array A may still appear on the right hand side of any number of statements, where its indices can be arbitrary affine expressions of $\vec{i}$ and $\vec{n}$.
- With each array A we will associate an occupancy vector $\vec{v}_A$ that specifies the storage reuse within A. The locations $\vec{l}_1$ and $\vec{l}_2$ in the original data space of A will be stored in the same location following our storage transform if and only if $\vec{l}_1 = \vec{l}_2 + k * \vec{v}_A$, for some integer k. Given our assumption about the data space, we can equivalently state that the values produced by iterations $\vec{i}_1$ and $\vec{i}_2$ will be stored in the same location following our storage transform if and only if $\vec{i}_1 = \vec{i}_2 + k * \vec{v}_A$, for some integer k.

4.2 Schedule Constraints

According to dependence P_j (Equation (1)), for any value of $\vec{i}$ in $\mathcal{P}_j$, operation $R_j(\vec{i})$ depends on the execution of operation $T_j(\vec{h}_j(\vec{i}, \vec{n}))$. Therefore, in order to preserve the semantics of the original program, in any new order of the computations, $T_j(\vec{h}_j(\vec{i}, \vec{n}))$ must be scheduled at a time strictly

earlier than $R_j(\vec{i})$, for all $\vec{i} \in \mathcal{P}_j$. We express this constraint in terms of the scheduling function. We must have, for each dependence P_j, $j \in [1, n_p]$:

$$\forall \vec{n} \in \mathcal{N}, \forall \vec{i} \in \mathcal{P}_j, \theta_{R_j}(\vec{i}, \vec{n}) - \theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}), \vec{n}) - 1 \geq 0 \quad (2)$$

These dependence constraints can be solved using Farkas' lemma as shown by Feautrier [7, 8, 4]. The result can be expressed as a polyhedron $\mathcal{R}$: the set of all the legal schedules Θ in the space of scheduling parameters $\mathcal{E}$. Note that Equation (2) does not always have a solution [7]. In such a case, one needs to use multidimensional schedules [8]. However, in this paper, we assume that Equation (2) has a solution.

Refer to Section 5.1.1 for an example of the schedule constraints.

4.3 Storage Constraints

The occupancy vectors induce some storage constraints. We consider any array A. Because we assume that the data space corresponds with the iteration space, and by definition of the occupancy vectors, the values computed by iterations $\vec{i}$ and $\vec{i} + \vec{v}_A$ are both stored in the same location $\vec{l}$. For an occupancy vector $\vec{v}_A$ to be valid for a given data object A, every operation depending on the value stored at location $\vec{l}$ by iteration $\vec{i}$ must execute *no later than* iteration $\vec{i} + \vec{v}_A$ stores a new value at location $\vec{l}$. Otherwise, following our storage transformation, a consumer expecting to reference the contents of $\vec{l}$ produced by iteration $\vec{i}$ could reference the contents of $\vec{l}$ written by iteration $\vec{i} + \vec{v}_A$ instead, thereby changing the semantics of the program. We assume that, at a given time step, all the reads precede the writes, such that an operation consuming a value can be scheduled for the same execution time as an operation overwriting the value. (This choice is arbitrary and unimportant to the method; under the opposite assumption, we would instead require that the consumer execute at least one step before its value is overwritten.)

Let us consider a dependence $P = (R, T, h, \mathcal{P})$. Then operation $T(\vec{h}(\vec{i}, \vec{n}))$ produces a value which will be later on read by $R(\vec{i})$. This value will be overwritten by $T(\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)})$. The storage constraint imposes that $T(\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)})$ is scheduled no earlier than $R(\vec{i})$. Therefore, any schedule Θ and any occupancy vector $\vec{v}_{A(T)}$ respects the dependence P if:

$$\forall \vec{n} \in \mathcal{N}, \forall \vec{i} \in \mathcal{Z}, \theta_T(\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)}, \vec{n}) - \theta_R(\vec{i}, \vec{n}) \geq 0 \quad (3)$$

where $\mathcal{Z}$ represents the domain over which the storage constraint applies. That is, the storage constraint applies for all iterations $\vec{i}$ where $\vec{i}$ is in the domain of the dependence, and where $\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)}$ is in the domain of statement T. Formally, $\mathcal{Z} = \{\vec{i} \mid \vec{i} \in \mathcal{P} \wedge \vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)} \in \mathcal{D}_T\}$. This definition of $\mathcal{Z}$ is not problematic, since the intersection of two polyhedra is defined simply by the union of the affine inequalities describing each, which obviously is a polyhedron. Note, however, that $\mathcal{Z}$ is parameterized by both $\vec{v}_{A(T)}$ and $\vec{n}$, and not simply by $\vec{n}$.

Equation (3) expresses the constraint on an occupancy vector for a given dependence and a given schedule. For an occupancy vector to be an AOV, however, it must respect all dependences across all legal schedules. Thus, the following constraint defines a valid AOV $\vec{v}_A$ for each object A in the program:

$$\forall \Theta \in \mathcal{R}, \forall \vec{n} \in \mathcal{N}, \forall j \in [1, n_p], \forall \vec{i} \in \mathcal{Z}_j,$$
$$\theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) - \theta_{R_j}(\vec{i}, \vec{n}) - 1 \geq 0 \quad (4)$$

See Section 5.1.1 for an illustration of the storage constraints.

4.4 Linearizing the Constraints

Equations (3) and (4) represent a possibly infinite set of constraints, because of the parameters. Therefore, we need to rewrite them so as to obtain an equivalent but finite set of affine equations and inequalities, which we can easily solve. Meanwhile, we seek to express the schedule (2) and storage (4) constraints in forms affine in the scheduling parameters Θ. This step is essential for constructing a linear program that minimizes the length of the AOV's.

Section 5.2 contains an illustrative example of the constraint linearization.

4.4.1 Reduction using the vertices of polyhedra

Any nonempty polyhedron is fully defined by its vertices, rays and lines [16], which can be computed even in the case of parameterized polyhedra [13]. The following theorem explains how we can use these vertices, rays and lines to reduce the size of our sets of constraints.

Theorem 1 *Let $\mathcal{D}$ be a nonempty polyhedron. $\mathcal{D}$ can be written $\mathcal{D} = P + C$, where P is a polytope (bounded polyhedron) and C is a cone. Then any affine function h defined over $\mathcal{D}$ is nonnegative on $\mathcal{D}$ if and only if 1) h is nonnegative on each of the vertices of P and 2) the linear part of h is nonnegative (resp. null) on the rays (resp. lines) of C.*

Although the domain of structural parameters $\mathcal{N}$ is an input of this analysis and may be unbounded, all the polyhedra produced by the dependence analysis of programs are in fact polytopes, or bounded polyhedra. Therefore, in order to simplify the equations, we now assume that all the polyhedra we manipulate are polytopes, except when stated otherwise. Then, according to Theorem 1, an affine function is nonnegative on a polyhedron if and only if it is nonnegative on the vertices of this polyhedron. We successively use this theorem to eliminate the iteration vector and the structural parameters from Equation (3).

4.4.2 Eliminating the Iteration Vector

Let us consider any fixed values of Θ in $\mathcal{R}$ and $\vec{n}$ in $\mathcal{N}$. Then, for all $j \in [1, n_p]$, $\vec{v}_{A(T_j)}$ must satisfy:

$$\forall \vec{i} \in \mathcal{Z}_j, \theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) - \theta_{R_j}(\vec{i}, \vec{n}) - 1 \geq 0 \quad (5)$$

which is an affine inequality in $\vec{i}$ (as $\vec{h}_j$, θ_{T_j}, and θ_{R_j} are affine functions). Thus, according to Theorem 1, it takes its extremal values on the vertices of the polytope $\mathcal{Z}_j$, denoted by $\vec{z}_{1,j}, \ldots, \vec{z}_{n_z,j}$. Note that $\mathcal{Z}_j$ is parameterized by $\vec{n}$ and $\vec{v}_{A(T_j)}$. Therefore, the number of its vertices might change depending on the domain of values of $\vec{n}$ and $\vec{v}_{A(T_j)}$. In this case we decompose the domains of $\vec{n}$ and $\vec{v}_{A(T_j)}$ into subdomains over which the number and definition of the vertices do not change [13], we solve our problem on each of these domains, and we take the "best" solution.

Thus, we evaluate (5) at the extreme points of $\mathcal{Z}_j$, yielding the following:

$$\begin{aligned}\forall k \in [1, n_z], \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) - 1 \geq 0\end{aligned} \tag{6}$$

According to Theorem 1, Equations (5) and (6) are equivalent. However, we have replaced the iteration vector $\vec{i}$ with the vectors $\vec{z}_{k,j}$, each of which is an affine form in $\vec{n}$ and $\vec{v}_{A(T_j)}$.

4.4.3 Eliminating the Structural Parameters

Suppose $\mathcal{N}$ is also a bounded polyhedron. We eliminate the structural parameters the same way we eliminated the iteration vector: by only considering the extremal vertices of their domain $\mathcal{N}$. Thus, for any fixed value of Θ in $\mathcal{R}$, j in $[1, n_p]$, and k in $[1, n_z]$ we must have:

$$\begin{aligned}\forall \vec{n} \in \mathcal{N}, \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) - 1 \geq 0\end{aligned} \tag{7}$$

Denoting the vertices of $\mathcal{N}$ by $(\vec{w}_1, \ldots, \vec{w}_{n_w})$, the above equation is equivalent to:

$$\begin{aligned}\forall l \in [1, n_w], \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) + \vec{v}_{A(T_j)}, \vec{w}_l) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) - 1 \geq 0\end{aligned} \tag{8}$$

Case of unbounded domain of parameters. It might also be the case that $\mathcal{N}$ is not a polytope but an unbounded polyhedron, perhaps corresponding to a parameter that is input from the user and can be arbitrarily large. In this case, we use the general form of Theorem 1. Let $\vec{r}_1, \ldots, \vec{r}_{n_r}$ be the rays defining the unbounded portion of $\mathcal{N}$ (a line being coded by two opposite rays). We must ensure that the linear part of Equation (8) is nonnegative on these rays. For example, given a single structural parameter $n_1 \in [5, \infty)$, we have the following constraint for the vertex $n_1 = 5$:

$$\begin{aligned}\theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 5), 5) + \vec{v}_{A(T_j)}, 5) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 5), 5) - 1 \geq 0\end{aligned}$$

and the following constraint for the positive ray of value 1:

$$\begin{aligned}\theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 1), 1) + \vec{v}_{A(T_j)}, 1) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 1), 1) \\ -\theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 0), 0) + \vec{v}_{A(T_j)}, 0) \\ +\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 0), 0) \geq 0\end{aligned} \tag{9}$$

Though this equation may look complicated, in practice it leads to simple formulas since all the constant parts of Equation (7) are going away. We assume in the rest of this paper that $\mathcal{N}$ is a polytope. This changes nothing in our method, but greatly improves the readability of the upcoming systems of constraints!

4.5 Finding a Solution

After removing the structural parameters, we are left with the following set of storage constraints:

$$\begin{aligned}\forall j \in [1, n_p], \forall k \in [1, n_z], \forall l \in [1, n_w], \\ \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) + \vec{v}_{A(T_j)}, \vec{w}_l) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) - 1 \geq 0\end{aligned} \tag{10}$$

which is a set of affine inequalities in the coordinates of the schedule Θ, with the occupancy vectors $\vec{v}_{A(T_j)}$ as unknowns. Note that the vertices $\vec{z}_{k,j}$ of the iteration domain, the vertices $\vec{w}_l$ of the structural parameters, and the components $\vec{h}_j$ of the affine functions, all have fixed and known values.

Similarly, we can linearize the schedule constraints to arrive at the following equations:

$$\begin{aligned}\forall j \in [1, n_p], \forall k \in [1, n_y], \forall l \in [1, n_w], \\ \theta_{R_j}(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l) - \theta_{T_j}(\vec{h}_j(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l), \vec{w}_l) - 1 \geq 0\end{aligned} \tag{11}$$

Where $y_{1,j}, \ldots, y_{n_y,j}$ denote the vertices of $\mathcal{P}_j$.

4.5.1 Finding an Occupancy Vector Given a Schedule

At this point we have all we need to determine which occupancy vectors (if any) are valid for a given schedule Θ: we simply substitute into the simplified storage constraints (10) the value of the given schedule. Then we obtain a set of affine inequalities where the only unknowns are the components of the occupancy vector. This system of constraints fully and exactly defines the set of the occupancy vectors valid for the given schedule. We can search this space for solutions with any Linear Programming solver.

To find the shortest occupancy vectors, we can use as our objective function the sum of the lengths[2] of the components of the occupancy vector. This metric minimizes the "Manhattan" length of each occupancy vector instead of minimizing the Euclidean length. However, minimizing the Euclidean length would require a non-linear objective function.

We improve our heuristic slightly by minimizing the difference between the lengths of the occupancy vector components as a second-order term in the objective function. That is, the objective function is

$$obj(\vec{v}) = k * \sum_{i=1}^{dim(v)} |v_i| + \sum_{i=1}^{dim(v)} \sum_{j=1}^{dim(v)} |v_i - v_j|$$

where k is large enough that the first term dominates, thereby selecting our vector first by the length of its components and then by the distribution of those lengths across its dimensions (a more "even" distribution having a shorter Euclidean distance.) It has been our experience that this linear objective function also finds the occupancy vector of the shortest Euclidean distance.

For an example of this procedure, refer to Section 5.1.2.

4.5.2 Finding a Schedule Given an Occupancy Vector

At this point, we also have all we need to determine which schedules (if any) exist for a given set of occupancy vectors. Given an occupancy vector $\vec{v}_A$ for each array A in the program, we substitute into the linearized storage constraints (10) to obtain a set of inequalities where the only unknowns are the scheduling parameters. These inequalities, in combination with the linearized schedule constraints (11) completely define the space of valid affine schedules valid for the given occupancy vectors. Once again, we can search this space for solutions with any Linear Programming solver, selecting the "best" schedule as in [7].

See Section 5.1.3 for an example.

[2] To minimize $|x|$, set $x = w - z$, $w \geq 0$, $z \geq 0$, and then minimize $w + z$. Either w or z will be zero in the optimum, leaving $w + z = |v|$.

```
A[][] = new int[n][m]
B[][] = new int[n][m]
...
for i = 1 to n
  for j = 1 to m
    A[i][j] = f(B[i-1][j])              (S1)
    B[i][j] = g(A[i][j-1])              (S2)
```

Figure 7: Original code for Example 2.

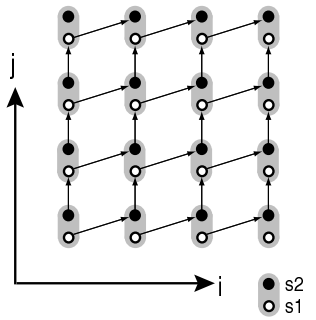

Figure 8: Dependence diagram for Example 2.

4.5.3 Finding the AOV's

Solving for the AOV's is more involved (follow Section 5.1.4 for an example.) To find a set of AOV's, we need to satisfy the storage constraints (10) for any value of the schedule Θ within the polyhedron $\mathcal{R}$ defined by the schedule constraints. To do this, we apply the Affine Form of Farkas' Lemma [16, 7, 4].

Theorem 2 *(Affine Form of Farkas' Lemma) Let $\mathcal{D}$ be a nonempty polyhedron defined by p affine inequalities*

$$\vec{a}_j \cdot \vec{x} + b_j \geq 0, \;\; j \in [1, p],$$

in a vector space $\mathcal{E}$. Then an affine form Ψ is nonnegative everywhere in $\mathcal{D}$ if and only if it is an affine combination of the affine forms defining $\mathcal{D}$:

$$\forall \vec{x} \in \mathcal{E}, \;\; \Psi(\vec{x}) \equiv \lambda_0 + \sum_j (\lambda_j (\vec{a}_j \cdot \vec{x} + b_j)), \;\; \lambda_0 \ldots \lambda_p \geq 0$$

The nonnegative constants λ_j are referred to as Farkas multipliers.

To apply the lemma, we note that the storage constraints are affine inequalities in Θ which are nonnegative over the polyhedron $\mathcal{R}$. Thus, we can express each storage constraint as a nonnegative affine combination of the schedule constraints defining $\mathcal{R}$.

To simplify our notation, let $STORAGE$ be the set of expressions that are constrained to be nonnegative by the linearized storage constraints (10). That is, $STORAGE$ contains the left hand side of each inequality in (10). Naively, $|STORAGE| = n_p \times n_z \times (n_w + n_r)$; however, several of these expressions might be equivalent, thereby reducing the size of $STORAGE$ in practice.

Similarly, let $SCHEDULE$ be the set of expressions that are constrained to be nonnegative by the linearized schedule constraints (11). The size of $SCHEDULE$ is at most $n_p \times n_y \times (n_w + n_r)$.

Then, the application of Farkas' Lemma yields these identities across the vector space $\mathcal{E}$ of scheduling parameters in which Θ lives:

```
A[] = new int[m+n]
B[] = new int[m+n]
...
for i = 1 to n
  for j = 1 to m
    A[i-j+m] = f(B[(i-1)-j+m])          (S1)
    B[i-j+m] = g(A[i-(j-1)+m])          (S2)
```

Figure 9: Transformed code for Example 2. Each array has an AOV of (1,1).

$$STORAGE_i(\vec{x}) = \lambda_{i,0} + \sum_{j=1}^{|SCHEDULE|} (\lambda_{i,j} \cdot SCHEDULE_j(\vec{x}))$$

$$\lambda_{i,j} \geq 0, \quad \forall \vec{x} \in \mathcal{E}, \forall i \in [1, |STORAGE|]$$

These equations are valid over the whole vector space $\mathcal{E}$. Therefore, we can collect the terms for each of the components of x, as well as the constant terms, setting equal the respective coefficients of these terms from opposite sides of a given equation (cf. [7, 4] for full details). We are left with $|STORAGE| \times (3 \times n_s + 1)$ linear equations where the only variables are the λ's and the occupancy vectors $\vec{v}_A$.

The set of valid AOV's is completely and exactly determined by this set of equations and inequalities. To find the shortest AOV, we proceed as in Section 4.5.1.

5 Examples

We present four examples to illustrate applications of the method described above.

5.1 Example 1: Simple Stencil

First we derive the solutions presented earlier for the 3-point stencil in Example 1.

5.1.1 Constraints

Let θ denote the scheduling function for the statement writing to array A. We assume that θ is an affine form as follows:

$$\theta(i, j, n, m) = a * i + b * j + c * n + d * m + e$$

There are three dependences in the stencil, each from the statement unto itself. The access functions describing the dependences are $\vec{h}_1(i, j, n, m) = (i-2, j-1)$, $\vec{h}_2(i, j, n, m) = (i, j-1)$, and $\vec{h}_3(i, j, n, m) = (i+1, j-1)$. Because these dependences are uniform–that is, they do not depend on the iteration vector–we can simplify our analysis by considering the dependence domains to be across all values of i and j. Thus, the schedule constraints are:

$$\begin{aligned} &\theta(i, j, n, m) - \theta(i-2, j-1, n, m) - 1 \geq 0 \\ &\theta(i, j, n, m) - \theta(i, j-1, n, m) - 1 \geq 0 \\ &\theta(i, j, n, m) - \theta(i+1, j-1, n, m) - 1 \geq 0 \end{aligned}$$

However, substituting the definition of θ into these equations, we find that i, j, n, and m are eliminated. This is because the constraints are uniform. Thus, we obtain the following simplified schedule constraints, which are affine in the scheduling parameters:

$$\begin{aligned} &2 * a + b - 1 \geq 0 \\ &b - 1 \geq 0 \\ &-a + b - 1 \geq 0 \end{aligned}$$

```
imax = a.length
jmax = b.length
kmax = c.length
D[][][] = new int[imax][jmax][kmax]
...
for i = 1 to imax
  for j = 1 to jmax
    for k = 1 to kmax
      if (i==1) or (j==1) or (k==1) then
        D[i][j][k] = f(i,j,k)                              (S1)
      else
        D[i][j][k] =                                       (S2)
          min(D[i-1][j-1][k-1] + w(a[i],b[j],c[k]),
              D[i][j-1][k-1] + w(GAP,b[j],c[k]),
              D[i-1][j][k-1] + w(a[i],GAP,c[k]),
              D[i-1][j-1][k] + w(a[i],b[j],GAP),
              D[i-1][j][k] + w(a[i],GAP,GAP),
              D[i][j-1][k] + w(GAP,b[j],GAP),
              D[i][j][k-1] + w(GAP,GAP,c[k]))
```

Figure 10: Original code for Example 3, for multiple sequence alignment. Here f computes the initial gap penalty and w computes the pairwise alignment cost.

```
imax = a.length
jmax = b.length
kmax = c.length
D[][] = new int[imax+jmax][imax+kmax]
...
for i = 1 to imax
  for j = 1 to jmax
    for k = 1 to kmax
      if (i==1) or (j==1) or (k==1) then
        D[jmax+i-j][kmax+i-k] = f(i,j,k)                   (S1)
      else
        D[jmax+i-j][kmax+i-k] =                            (S2)
          min(D[jmax+(i-1)-(j-1)][kmax+(i-1)-(k-1)] + w(a[i],b[j],c[k]),
              D[jmax+i-(j-1)][kmax+i-(k-1)] + w(GAP,b[j],c[k]),
              D[jmax+(i-1)-j][kmax+(i-1)-(k-1)] + w(a[i],GAP,c[k]),
              D[jmax+(i-1)-(j-1)][kmax+(i-1)-k] + w(a[i],b[j],GAP),
              D[jmax+(i-1)-j][kmax+(i-1)-k] + w(a[i],GAP,GAP),
              D[jmax+i-(j-1)[kmax+i-k] + w(GAP,b[j],GAP),
              D[jmax+i-j][kmax+i-(k-1)] + w(GAP,GAP,c[k]))
```

Figure 11: Transformed code for Example 3, using the AOV of (1,1,1). The new array has dimension [imax+jmax][imax+kmax], with each reference to [i][j][k] mapped to [jmax+i-j][kmax+i-k].

Now let $\vec{v}_A = (v_i, v_j)$ denote the AOV that we are seeking for array A. Then the storage constraints are as follows:

$$\theta(i-2+v_i, j-1+v_j, n, m) - \theta(i,j,n,m) \geq 0$$
$$\theta(i+v_i, j-1+v_j, n, m) - \theta(i,j,n,m) \geq 0$$
$$\theta(i+1+v_i, j-1+v_j, n, m) - \theta(i,j,n,m) \geq 0$$

Simplifying the storage constraints as we did the schedule constraints, we obtain the linearized storage constraints:

$$a*v_i + b*v_j - 2*a - b \geq 0$$
$$a*v_i + b*v_j - b \geq 0$$
$$a*v_i + b*v_j + a - b \geq 0$$

5.1.2 Finding an Occupancy Vector

To find the shortest occupancy vector for the schedule that executes the rows in parallel, we substitute $\theta(i,j,n,m) = j$ into the linearized schedule and storage constraints. Minimizing $|v_i + v_j|$ with respect to these constraints gives the occupancy vector of $(0,2)$ (see Figure 3).

5.1.3 Finding a Schedule

To find the set of schedules that are valid for the occupancy vector of $(0,2)$, we substitute $v_i = 0$ and $v_j = 2$ into the linearized schedule and storage constraints. Simplifying the resulting constraints yields:

$$b \geq 1 - 2*a$$
$$b \geq 1 + a$$
$$b \geq 2*a$$

Inspection of these inequalities reveals that the ratio a/b has a minimum value of $-1/2$ and a maximum value that asymptotically approaches $1/2$, thus corresponding to the set of legal affine schedules depicted in Figure 5 (note that in the frame of the figure, however, the schedule's slope is $-a/b$.)

5.1.4 Finding an AOV

To find an AOV for A, we apply Farkas' Lemma to rewrite each of the linearized storage constraints as a non-negative affine combination of the linearized schedule constraints:

$$\begin{bmatrix} a*v_i + b*v_j - 2*a - b \\ a*v_i + b*v_j - b \\ a*v_i + b*v_j + a - b \end{bmatrix} =$$

$$\begin{bmatrix} \lambda_{1,1} & \lambda_{1,2} & \lambda_{1,3} & \lambda_{1,4} \\ \lambda_{2,1} & \lambda_{2,2} & \lambda_{2,3} & \lambda_{2,4} \\ \lambda_{3,1} & \lambda_{3,2} & \lambda_{3,3} & \lambda_{3,4} \end{bmatrix} \begin{bmatrix} 1 \\ 2*a+b-1 \\ b-1 \\ -a+b-1 \end{bmatrix}$$

$$\lambda_{i,j} \geq 0, \ \forall i \in [1,3], \forall j \in [1,4]$$

Minimizing $|v_i + v_j|$ subject to these constraints yields an AOV $(v_i, v_j) = (1,2)$, which is smaller than the shortest UOV of $(0,3)$ [17].

To transform the data space of array A according to this AOV $\vec{v}$, we follow the approach of [17] and project the original data space onto the line perpendicular to $\vec{v}$. Choosing $\vec{v}_\perp = (2,-1)$ so that $\vec{v} \cdot \vec{v}_\perp = 0$, we transform the original indices of (i,j) into $\vec{v}_\perp \cdot (i,j) = 2*i - j$. Finally, to ensure that all data accesses are non-negative, we add m to the new index, such that the final transformation is from $A[i][j]$ to $A[2*i-j+m]$. Thus, we have reduced storage requirements from $n*m$ to $2*n+m$. The modified code corresponding to this mapping is shown in Figure 6.

5.2 Example 2: Two-Statement Stencil

We now consider an example adapted from [12] where there is a uniform dependence between statements in a loop (see Figures 7 and 8). Letting θ_1 and θ_2 denote the scheduling functions for statements 1 and 2, respectively, we have following schedule constraints:

$$\theta_1(i,j,n,m) - \theta_2(i-1,j,n,m) - 1 \geq 0$$
$$\theta_2(i,j,n,m) - \theta_1(i,j-1,n,m) - 1 \geq 0$$

and the following storage constraints:

$$\theta_2(i-1+v_{B,i}, j+v_{B,j}, n, m) - \theta_1(i,j,n,m) \geq 0$$
$$\theta_1(i+v_{A,i}, j-1+v_{A,j}, n, m) - \theta_2(i,j,n,m) \geq 0$$

We now demonstrate how to linearize the schedule constraints. We observe that the polyhedral domain of the iteration parameters (i,j) has vertices at $(1,1),(n,1),(1,m),(n,m)$,

```
A[][] = new int[n][m]
B[] = new int[n]
...
for i = 1 to n
  for j = 1 to n
    A[i][j] = B[i-1]+j                (S1)
  B[i] = A[i][n-i]                    (S2)
```

Figure 12: Original code for Example 4.

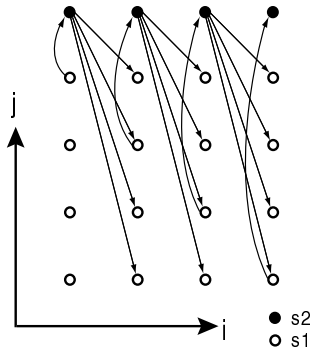

Figure 13: Dependence diagram for Example 4.

so we evaluate the schedule constraints at these points to eliminate (i, j):

$$\begin{aligned}
&\theta_1(1,1,n,m) - \theta_2(0,1,n,m) - 1 \geq 0\\
&\theta_2(1,1,n,m) - \theta_1(1,0,n,m) - 1 \geq 0\\
&\theta_1(n,1,n,m) - \theta_2(n-1,1,n,m) - 1 \geq 0\\
&\theta_2(n,1,n,m) - \theta_1(n,0,n,m) - 1 \geq 0\\
&\theta_1(1,m,n,m) - \theta_2(1-1,m,n,m) - 1 \geq 0\\
&\theta_2(1,m,n,m) - \theta_1(1,m-1,n,m) - 1 \geq 0\\
&\theta_1(n,m,n,m) - \theta_2(n-1,m,n,m) - 1 \geq 0\\
&\theta_2(n,m,n,m) - \theta_1(n,m-1,n,m) - 1 \geq 0
\end{aligned}$$

Next, we eliminate the structural parameters (n, m). Assuming n and m are positive but arbitrarily large, the domain of these parameters is an unbounded polyhedron: $(n, m) = (1, 1) + j * (0, 1) + k * (1, 0)$, for positive integers j and k. We must evaluate the above constraints at the vertex $(1, 1)$, as well as the linear part of the constraints for the rays $(1, 0)$ and $(0, 1)$. Doing so yields 24 equations, of which we show the first 3 (which result from substituting into the first of the equations above):

$$\begin{aligned}
&\theta_1(1,1,1,1) - \theta_2(0,1,1,1) - 1 \geq 0\\
&\theta_1(1,1,1,0) - \theta_2(0,1,1,0) - \theta_1(1,1,0,0) + \theta_2(0,1,0,0) \geq 0\\
&\theta_1(1,1,0,1) - \theta_2(0,1,0,1) - \theta_1(1,1,0,0) + \theta_2(0,1,0,0) \geq 0
\end{aligned}$$

Expanding the scheduling functions as $\theta_x(i, j, n, m) = a_x + b_x * i + c_x * j + d_x * n + e_x * m$, the entire set of 24 equations can be simplified to:

$$\begin{aligned}
&d_1 = d_2\\
&e_1 = e_2\\
&a_1 + b_1 + c_1 - a_2 - c_2 + (b_1 - b_2)n - 1 \geq 0\\
&a_1 + 2b_1 + c_1 - a_2 - b_2 - c_2 - 1 \geq 0\\
&a_2 + b_2 + 2c_2 - a_1 - b_1 - c_1 - 1 \geq 0\\
&a_2 + 2c_2 - a_1 - c_1 + (b_2 - b_1)n - 1 \geq 0
\end{aligned}$$

These equations constitute the linearized schedule constraints. In a similar fashion, we could linearize the storage constraints, and then apply Farkas' lemma to find the shortest AOV's of $\vec{v}_A = \vec{v}_B = (1, 1)$. Due to space limitations, we do not derive the entire solution here. The code that results after transformation by these AOV's is shown in Figure 9.

```
A[] = new int[n]
B = new int
...
for i = 1 to n
  for j = 1 to n
    A[i] = B+j                        (S1)
  B = A[i]                            (S2)
```

Figure 14: Transformed code for Example 4. The AOV's for A and B are (1,0) and 1, respectively.

5.3 Example 3: Multiple Sequence Alignment

We now consider a version of the Needleman-Wunch sequence alignment algorithm [14] to determine the cost of the optimal global alignment of three strings (see Figure 10). The algorithm utilizes dynamic programming to determine the minimum-cost alignment according to a cost function w that specifies the cost of aligning three characters, some of which might represent gaps in the alignment.

Using θ_1 and θ_2 to represent the scheduling functions for statements 1 and 2, respectively, we have the following schedule constraints (we enumerate only three constraints for each pair of statements since the other dependences follow by transitivity):

$$\begin{aligned}
&\theta_2(i,j,k,x,y,z) - \theta_1(i-1,j,k,x,y,z) - 1 \geq 0\\
&\quad for\ i = 2, j \in [2,y], k \in [2,z]\\
&\theta_2(i,j,k,x,y,z) - \theta_1(i,j-1,k,x,y,z) - 1 \geq 0\\
&\quad for\ i \in [2,x], j = 2, k \in [2,z]\\
&\theta_2(i,j,k,x,y,z) - \theta_1(i,j,k-1,x,y,z) - 1 \geq 0\\
&\quad for\ i \in [2,x], j \in [2,y], k = 2\\
&\theta_2(i,j,k,x,y,z) - \theta_2(i-1,j,k,x,y,z) - 1 \geq 0\\
&\quad for\ i \in [3,x], j \in [2,y], k \in [2,z]\\
&\theta_2(i,j,k,x,y,z) - \theta_2(i,j-1,k,x,y,z) - 1 \geq 0\\
&\quad for\ i \in [2,x], j \in [3,y], k \in [2,z]\\
&\theta_2(i,j,k,x,y,z) - \theta_2(i,j,k-1,x,y,z) - 1 \geq 0\\
&\quad for\ i \in [2,x], j \in [2,y], k \in [3,z]
\end{aligned}$$

Note that each constraint is restricted to the subset of the iteration domain under which it applies. That is, S_2 depends on S_1 only when i, j, or k is equal to 2; otherwise, S_2 depends on itself. This example illustrates the precision of our technique for general dependence domains.

The storage constraints are as follows:

$$\begin{aligned}
&\theta_2(i-1+v_i, j+v_j, k+v_k, x,y,z) - \theta_2(i,j,k,x,y,z) \geq 0\\
&\quad for\ i \in [3,x], j \in [2,y], k \in [2,z]\\
&\theta_2(i+v_i, j-1+v_j, k+v_k, x,y,z) - \theta_2(i,j,k,x,y,z) \geq 0\\
&\quad for\ i \in [2,x], j \in [3,y], k \in [2,z]\\
&\theta_2(i+v_i, j+v_j, k-1+v_k, x,y,z) - \theta_2(i,j,k,x,y,z) \geq 0\\
&\quad for\ i \in [2,x], j \in [2,y], k \in [3,z]
\end{aligned}$$

There is no storage constraint corresponding to the dependence of S_2 on S_1 because the domain $\mathcal{Z}$ of the constraint is empty for occupancy vectors with positive components, and occupancy vectors with a non-positive component do not satisfy the above constraints. That is, for the first dependence of S_2 on S_1, the dependence domain is $\mathcal{P} = \{(2, j, k) \mid j \in [2, y] \wedge k \in [2, z]\}$ while the existence domain of S_1 is $\mathcal{D}_{S_1} = \{(i, j, k) \mid i \in [1, x] \wedge j \in [1, y] \wedge k \in [1, z] \wedge (i = 1 \vee j = 1 \vee k = 1)\}$. Then, the domain of the first storage constraint is $\mathcal{Z} = \{(i, j, k) \mid (i, j, k) \in \mathcal{P} \wedge (i-1, j, k) + \vec{v}_A \in \mathcal{D}_{S_1}\}$. Now, $\mathcal{Z}$ is empty given that $\vec{v}_A$ has positive components, because if $(i, j, k) \in \mathcal{P}$ then $i = 2$, but if $(i-1, j, k) + \vec{v}_A \in \mathcal{D}_{S_1}$ then $i - 1 + v_{A,i} = 1$, or equivalently $i + v_{A,i} = 2$. Thus for $\mathcal{Z}$ to be non-empty, we would have $2 + v_{A,i} = 2$, which contradicts the positivity assumption on $v_{A,i}$. The argument is analogous for other dependences of S_2 on S_1.

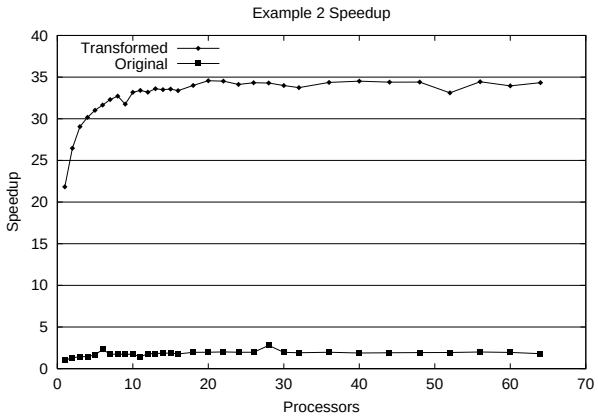

Figure 15: Speedup vs. number of processors for Example 2.

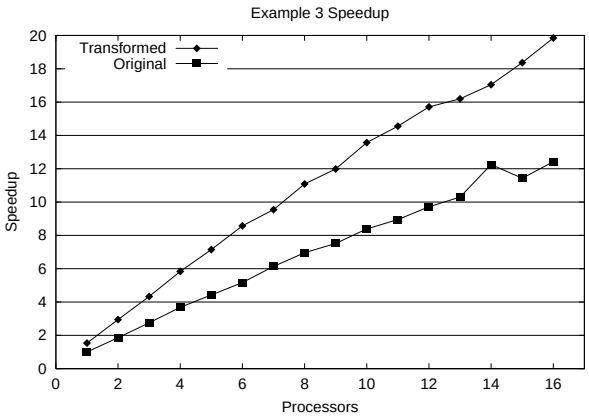

Figure 16: Speedup vs. number of processors for Example 3.

Applying our method for this example yields an AOV of $(1, 1, 1)$. The transformed code under this occupancy vector is just like the original, except that the array is of dimension [imax+jmax][imax+kmax] and element [i][j][k] is mapped to [jmax+i-j][kmax+i-k].

5.4 Example 4: Non-Uniform Dependences

Our final example is constructed to demonstrate the application of our method to non-uniform dependences (see Figures 12 and 13). Let θ_1 and θ_2 denote the scheduling functions for statements S_1 and S_2, respectively. Then we have the following schedule constraints:

$$\theta_1(i, j, n) - \theta_2(i - 1, n) - 1 \geq 0$$
$$\theta_2(i, n) - \theta_1(i, n - i, n) - 1 \geq 0$$

and the following storage constraints:

$$\theta_2(i - 1 + v_B, n) - \theta_1(i, j, n) \geq 0$$
$$\theta_1(i + v_{A,i}, n - i + v_{A,j}, n) - \theta_2(i, n) \geq 0$$

Applying our method to these constraints yields the AOV's $\vec{v}_A = (1, 0)$ and $v_B = 1$. The transformed code is shown in Figure 14.

6 Experiments

We performed preliminary experiments that validate our technique as applied to two of our examples. The tests were carried out on an SGI Origin 2000, which uses MIPS R10000 processors with 4MB L2 caches.

For Example 2, the computation was divided into diagonal strips. Since there are no data dependences between strips, the strips can be assigned to processors without requiring any synchronization [12]. Figure 15 shows the speedup gained on varying numbers of processors using both the original and the transformed array. Both versions show the same trend and do not significantly improve past 16 processors, but the transformed code has an advantage by a sizable constant factor.

Example 3 was parallelized by blocking the computation, and assigning rows of blocks to each processor. As shown in Figure 16, the transformed code again performs substantially better than the original code. With the reduced working set of data in the transformed code, the speedup is superlinear in the number of processors due to improved caching.

7 Related Work

The work most closely related to ours is that of [17], which considers schedule-independent storage mappings using the Universal Occupancy Vector (UOV). While an AOV is valid only for affine schedules, a UOV is valid for any legal execution ordering. Consequently, sometimes there exist AOV's that are shorter than any UOV since the AOV must be valid for a smaller range of schedules. While the analysis of [17] is limited to a stencil of dependences involving only one statement within a perfectly nested loop, our method applies to general affine dependences across statements and loop nests. Moreover, our framework goes beyond AOV's to unify the notion of occupancy vectors with known affine scheduling techniques.

Another related approach to storage management for parallel programs is that of [3, 2, 11]. Given an affine schedule, [11] optimizes storage first by restricting the size of each array dimension and then by combining distinct arrays via renaming. This work is extended in [3, 2] to consider storage mappings for a *set* of schedules, towards the end of capturing the tradeoff between parallelism and storage.

However, these techniques utilize a storage mapping where, in an assignment, each array dimension is indexed by a loop counter and is modulated independently (e.g. `A[i mod n][j mod m]`). This is distinct from the occupancy vector mapping, where the data space of the array is projected onto a hyperplane before modulation (if any) is introduced. The former mapping–when applied to all valid affine schedules–does not enable any storage reuse in Examples 2 and 3, where the AOV did. However, with a single occupancy vector we can only reduce the dimensionality of an array by one, whereas the other mapping can introduce constant bounds in several dimensions. In the future, we hope to extend our method to find multiple occupancy vectors, thereby enabling storage reuse along multiple array dimensions.

Memory reuse in the context of the polyhedral model is also considered in [18]. This approach uses yet another storage mapping, which utilizes array transformations on the data space to achieve the effect of multiple occupancy vectors applied at once. However, the mapping does not have any modulation, so it could not duplicate the effect of the $(2, 0)$ occupancy vector we found (for a given schedule) in Example 1.

8 Conclusion

We have presented a mathematical framework that unifies the techniques of affine scheduling and occupancy vector analysis. Within this framework, we showed how to determine a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid for all legal schedules. Our technique is general and precise, allowing inter-statement affine dependences and efficiently solving for the shortest occupancy vector using standard numerical programming methods.

We consider this research to be the first step towards automating a procedure that finds the optimal tradeoff between parallelism and storage space. This question is very relevant in the context of array expansion, where the cost of extra array dimensions must be weighed against the scheduling freedom that they provide. Additionally, our framework could be applied to single-assignment functional languages where all storage reuse must be orchestrated by the compiler. In both of these applications, and even for compiling to uniprocessor systems, understanding the interplay between scheduling and storage is crucial for achieving good performance.

However, since finding an exact solution for the "best" occupancy vector is a very complex problem, our method relies on several assumptions to make the problem tractable. We ignore the shape of the data space and assume that the shortest occupancy vector is the best; further, we minimize the Manhattan length of the vector, since minimizing the Euclidean length is nonlinear. Also, we restrict the input domain to programs where 1) the data space matches the iteration space, 2) only one statement writes to each array, 3) the schedule is one-dimensional and affine, and 4) there is an affine description of the dependences. It is with these qualifications that our method finds the "best" solution.

In future work, we aim to relax some of the assumptions about the input domain. Perhaps most relevant is the case of arbitrary affine references on the left hand side, since it would not only widen the input domain, but would allow the reduction of multiple array dimensions via application of successive occupancy vectors. Many of these extensions are difficult because, in their straightforward formulations, the constraints become nonlinear. We consider it to be an open question to formulate these extensions as linear programming problems.

It will also be valuable to consider more general storage mappings. The occupancy vector method as it stands now can only decrease the dimensionality of an array by one, and the irregular shape of the resulting data space could be hard to embed in a rectilinear array in a storage-efficient way. However, other storage mappings [11, 18] we discussed also have their limitations. The perfect storage mapping would allow variations in the number of array dimensions, while still capturing the directional and modular reuse of the occupancy vector and having an efficient implementation; it should also lend itself to efficient storage reuse between distinct arrays.

9 Acknowledgements

We would like to thank Kath Knobe for her helpful comments and suggestions. We appreciate the support of the MARINER project at Boston University for giving us access to its Scientific Computing Facilities. This work was partly supported by NSF Grant CCR0073510, DARPA grant DBT63-96-C-0036, and a graduate fellowship from Siebel Systems.

References

[1] D. Barthou, A. Cohen, and J. Collard. Maximal static expansion. In *Principles of Programming Languages*, pages 98–106, San Diego, CA, Jan. 1998.

[2] A. Cohen. Parallelization via constrained storage mapping optimization. *Lecture Notes in Computer Science*, 1615:83–94, 1999.

[3] A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. Technical Report 1998/46, PRiSM, U. of Versailles, 1988.

[4] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser Boston, 2000.

[5] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[6] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–51, 1991.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem. part I. one-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–347, Oct. 1992.

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

[9] P. Feautrier, J.-F. Collard, M. Barreteau, D. Barthou, A. Cohen, and V. Lefebvre. The interplay of expansion and scheduling in paf. Technical Report 1998/6, PRiSM, U. of Versailles, 1988.

[10] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Prog. Languages*, pages 319–329, San Diego, CA, Jan. 1988.

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

[12] A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages*, Jan. 1997.

[13] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6):525–549, Dec. 1997.

[14] S. B. Needleman and C. D. Wunsch. A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[15] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

[16] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[17] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Architectural Support for Programming Languages and Operating Systems*, pages 24–33, 1998.

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