# A Step Towards Unifying Schedule and Storage Optimization

WILLIAM THIES
Massachusetts Institute of Technology
FRÉDÉRIC VIVIEN
INRIA
and
SAMAN AMARASINGHE
Massachusetts Institute of Technology

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 (one-dimensional affine) schedules. We consider storage mappings that collapse one dimension of a multidimensional array, and programs that are in a single assignment form and accept a one-dimensional affine schedule. Our method 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 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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilation; optimization; memory management*

General Terms: Languages

Additional Key Words and Phrases: Automatic parallelization, affine scheduling, storage optimization, occupancy vectors, polyhedral model, affine recurrence equations

**ACM Reference Format:**

Thies, W., Vivien, F., and Amarasinghe, S. 2007. A step towards unifying schedule and storage optimization. ACM Trans. Program. Lang. Syst. 29, 6, Article 34 (October 2007), 45 pages. DOI = 10.1145/1286821.1286825 http://doi.acm.org/10.1145/1286821.1286825

This work was partly supported by NSF grant CCR0073510, DARPA grant DBT63-96-C-0036, and a graduate fellowship from Siebel Systems.
Authors' address: W Thies; email: thies@alum.mit.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 0164-0925/2007/10-ART34 $5.00 DOI 10.1145/1286821.1286825 http://doi.acm.org/10.1145/1286821.1286825

## 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 interrelated 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 [Feautrier et al. 1998; Feautrier 1988; Barthou et al. 2000] 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. Ideally, one would like to consider all possible storage mappings and instruction schedules and to choose the combination that results in the optimal execution time. However, doing so is too complex to be practical—there are too many combinations to enumerate, and it is difficult to gauge the efficiency of each one.

Here we arrive at a classical phase ordering problem in compilers. Since it is too complicated to optimize storage and parallelism at the same time, there must be some sequence of phases that optimizes each, in turn. However, if we first optimize for storage space, we restrict the range of legal schedules that can be considered by the parallelizer. Alternately, first optimizing for parallelism will restrict the set of storage mappings that we can consider later. What is needed is an efficient framework that can consider both storage optimization and schedule optimization at the same time.

In this article, 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 multidimensional array, and programs that are in a single assignment form with a one-dimensional affine schedule. Our technique incorporates general affine dependences across statements and loop nests; it deals with both perfectly nested and nonperfectly nested loops.

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 (one-dimensional affine) schedules. Our method is precise and practical in that it reduces to an integer linear program (or, in the case of problem 2, a linear program) that can be solved with standard techniques. We believe that these solutions represent a first step towards automating a procedure that finds the optimal compromise between parallelism and storage space.

The rest of this article is organized as follows. Section 2 gives an abstract formulation of the problem, and Section 3 reviews the mathematical background that forms the basis for our technique. Section 4 formulates our method abstractly, and Section 5 illustrates our method with examples. Related work is described in Section 6 and we conclude in Section 7.

## 2. ABSTRACT PROBLEM

We now consider an abstract description of the scheduling problems faced by a parallelizing compiler. We start with 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 that $v_2$ consumes 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 $v_2$.

Our task is to output $(\Theta, \Lambda)$, where $\Theta$ is a function mapping each operation $v \in V$ to its time of execution, and $\Lambda$ is a function mapping each operation to a storage location (where its result is stored). Parallelism is expressed implicitly by assigning the same execution time to multiple operations. How, then, might we go about choosing $\Theta$ and $\Lambda$?

### 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 $\Theta$ and choose $\Lambda$ such that 1) $(\Theta, \Lambda)$ respects the storage constraints, and 2) $\Lambda$ uses as few storage locations as possible (i.e., the size of the set $\{\Lambda(v) \mid v \in V\}$ is minimized).

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.

Note that if the schedule is sufficiently regular (i.e., affine), then one can precisely estimate the minimum number of storage locations needed, even if the program is parameterized, using Ehrhart polynomials [Clauss 1996; Kouache 2002, 19–20]. However, arbitrary storage mappings can be very expensive to implement. In this paper, we restrict our attention to a regular class of storage mappings (collapsing a single array dimension) that is efficient to implement and easy to reason about.

### 2.2 Choosing a Schedule Given a Store

The second problem is to find an optimal schedule for a given storage mapping, if any valid schedule exists. That is, we are given $\Lambda$ and choose $\Theta$ such that 1) $(\Theta, \Lambda)$ respects the schedule and storage constraints, and 2) $\Theta$ executes all of the instructions in the minimal span of time. Note that if $\Lambda$ is too restrictive (storing many values in the same location), there might not exist a $\Theta$ that respects the constraints.

This is a very relevant problem in practice because of the stepwise, nonlinear 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 $\Psi = \{\Theta_1, \Theta_2, \ldots\}$, where each $\Theta$ in $\Psi$ respects the schedule constraints. We choose $\Lambda$ such that 1) $\forall \Theta \in \Psi$, $(\Theta, \Lambda)$ respects the storage constraints, and 2) $\Lambda$ uses as few storage locations as possible.

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 a storage mapping, and then arrange the schedule to optimize for data locality, synchronization, or communication, without worrying about violating the storage constraints.

We could also postpone the choice of a schedule until runtime, when the system could incorporate dynamic information (e.g., the values of loop bounds and the available processor resources) into the scheduling decision. The storage mapping guarantees that any of the candidate schedules is valid.

Such flexibility also could be critical if, for example, we want to apply loop tiling [Irigoin and Triolet 1988] 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 originally. (Even though our technique considers only one-dimensional affine schedules, we demonstrate in Section 4.6.3 that it still applies to some forms of tiling.)

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 most storage-efficient $\Lambda$. In this way, $\Lambda$ provides the best storage option that is legal across the entire set of schedules under consideration.

## 2.4 Approach

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

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

Fig. 1. Code for Example 1.

constants are known, the problem sizes are too large for schedule and storage analysis on a DAG, and the generated code grows to an infeasible size if a static instruction is generated for every node in the DAG. Finally, it is desirable to have the output of the analysis (the transformed source code) be in a compact form so that it can be further compiled and optimized using standard techniques.

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 $\Theta$. Instead of allowing arbitrary edge relationships and execution orderings, we restrict our attention to affine dependences and one-dimensional affine schedules [Feautrier 1992a]. The second assumption concerns our approach to the optimized storage mapping. Instead of allowing an arbitrary mapping from operations to locations, we employ the *occupancy vector* as a simple and efficient mechanism of storage reuse [Strout et al. 1998]. The following section contains more background information on affine schedules and occupancy vectors.

## 3. BACKGROUND

In this section, we present background material that is needed to understand our technique. In Section 3.1 we give our notations for the polyhedral model, in Section 3.2 we review Feautrier's approach to the affine scheduling problem, and in Section 3.3 we discuss the occupancy vector as a means of collapsing storage requirements. Those who are already familiar with the polyhedral model, affine scheduling, and occupancy vectors can skip most of this section, but might want to review our notations in Sections 3.1 and 3.2.1.

### 3.1 The Polyhedral Model

A fundamental difficulty in constructing optimized schedules for real scientific programs is that some of the program parameters are unknown at compile time. For example, the code in Figure 1 has symbolic loop bounds $m$ and $n$ which might be unknown to the compiler. Moreover, even if they were known, they might be too large for the compiler to emit an explicit execution time for each iteration of the loop. Consequently, it is imperative that the scheduler consider programs not as a static list of individual instructions, but as a *family* of programs that are structured by some parameters. In the case when there is no bound on the structural parameters, the size of each program family is infinite. The compiler needs a finite representation for this infinite set of programs, as well as a tractable way to parameterize the schedule so that it is valid for any possible input.

The polyhedral model offers exactly this kind of representation [Feautrier 1996; Darte 1998]. It represents an infinite family of programs as a set of parameterized polyhedra, each of which can be finitely generated using the following definition.

*Definition* 1. A polyhedron $\mathcal{D}$ is defined as the intersection of a finite set of closed linear half-spaces. Its representation is specified by a system of inequalities:

$$\mathcal{D} = \{\vec{x} \in \mathbb{Q}^n \mid A\vec{x} \leq \vec{b}\}, \tag{1}$$

where $n$ is the dimension of the space containing the polyhedron; $A$ is a $j \times n$ matrix, $\vec{b}$ is a $j$-vector, and $j$ is the number of inequalities.

In the polyhedral model, polyhedra are used to represent the domains of loop iterations and data dependences. For a given statement, the iteration domain (or just *domain*) consists of the set of values assumed by the enclosing loop indices at runtime. If a statement uses values produced by previous statements, then there are also data dependence domains specifying the values of the loop indices where the dependences exist. The dependence domains are (nonstrict) subsets of the iteration domain.

The translation of programs into the polyhedral model was pioneered by Feautrier, who showed that programs with static control flow[1] can be accurately represented in the polyhedral model [Feautrier 1992a]. Throughout this article, we will use the following notations to describe a program in the polyhedral model:

—An iteration vector $\vec{i}$ contains the values of surrounding loop indices at a given point in the execution of the program. For example, the two-dimensional iteration vector $\vec{i} = (5, 10)$ could represent an iteration where $i = 5$ and $j = 10$ in the program of Figure 1.

—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. For example, in the case of Figure 1, we have $\vec{n} = (m, n)$.

—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. For example, in Figure 1 there is one assignment statement $S$, and its domain is the polyhedron $\mathcal{D}_S = \{(i, j) \mid 1 \leq i \leq m \ \wedge \ 1 \leq j \leq n\}$. Note that $\mathcal{D}_S$ is parameterized by the structural parameters $\vec{n}$.

—There are $n_p$ dependences $P_1 \ldots P_{n_p}$ in the program. Each dependence $P_j$ is a 4-tuple $(R_j, T_j, \vec{h}_j, \mathcal{P}_j)$ where $R_j$ and $T_j$ are statements, $\vec{h}_j$ is a vector-valued

[1]A *static control flow program* is one in which: 1) all control flow is in the form of FOR loops and if/then/else blocks (no function calls or GOTOs), 2) each statement is an assignment to a scalar or array variable, with the right-hand side being an arbitrary expression of such variables, 3) each array access has index expressions which are affine functions of the loop indices and symbolic constants, 4) each upper (resp. lower) loop bound is also in this form, or is the MIN (resp. MAX) of a finite set of such expressions, and 5) each conditional governing an if/then/else block depends only on an affine inequality involving the surrounding loop indices or symbolic constants.

affine[2] function, and $\mathcal{P}_j \subseteq \mathcal{D}_{R_j}$ is a polyhedron such that:

$$\forall \vec{i} \in \mathcal{P}_j,\ R_j(\vec{i}\,) \text{ depends on } T_j(\vec{h}_j(\vec{i}, \vec{n})). \tag{2}$$

In the case of Example 1, there are three dependences, each corresponding to an array access on the right hand side. The dependence domains are governed by the boundary conditions, since some instances of the assignment statement $S$ refer to values of the array that were assigned before the loop (although we ignore the pre-loop assignments in this example for sake of simplicity). The dependences are as follows:

$P_1 = (S, S, \vec{h}_1, \mathcal{P}_1)$,
where $\vec{h}_1(i, j) = (i-1, j-2)$ and $\mathcal{P}_1 = \{(i, j) \mid 2 \le i \le m \ \wedge\ 3 \le j \le n\}$

$P_2 = (S, S, \vec{h}_2, \mathcal{P}_2)$,
where $\vec{h}_2(i, j) = (i-1, j)$ and $\mathcal{P}_2 = \{(i, j) \mid 2 \le i \le m \ \wedge\ 1 \le j \le n\}$

$P_3 = (S, S, \vec{h}_3, \mathcal{P}_3)$,
where $\vec{h}_3(i, j) = (i-1, j+1)$ and $\mathcal{P}_3 = \{(i, j) \mid 2 \le i \le m \ \wedge\ 1 \le j \le n-1\}$.

The representation of dependences is very compact; instead of enumerating the instances of $T_j$ that some instance of $R_j$ depends on, there is a single function $\vec{h}_j$ that spans across the entire dependence domain $\mathcal{P}_j$. The dependences $P_j$ are determined using an array dataflow analysis [Feautrier 1991, 2001a; Maydan et al. 1993; Pugh 1992].

Note that there are cases in which our representation of dependences is not exact. For static control flow programs, the exact dependences can be represented by a quasi-affine selection tree (or *quast*) which consists of a tree of conditionals with dependence functions at the leaves [Feautrier 1991]. Each conditional and dependence is expressed using a quasi-affine function—that is, an affine function that may contain integer division, as well—on the loop counters and structural parameters. We assume that the dependence functions are affine rather than quasi-affine, thereby implying that each dependence domain is a polyhedron (rather than a $\mathbb{Z}$-polyhedron, which is the intersection of a polyhedron and an integral lattice) and each $\vec{h}_j$ is an affine function (rather than a quasi-affine function). Note that we also represent the domain of structural parameters $\mathcal{N}$ and each iteration domain $\mathcal{D}$ as a polyhedron rather than a **Z**-polyhedron; as detailed by Feautrier [1992a, 17], this is a conservative approximation that might (in rare cases) rule out some possible solutions but will never yield invalid results.

These are all the notations we need to describe a program in the polyhedral model. We now turn to our description of the schedule.

## 3.2 Affine Scheduling

3.2.1 *Representing a Schedule.* In Section 2 we considered a scheduling function $\Theta$ that governs the execution of all the instructions in the program.

[2]A function $\vec{h}(\vec{x})$ is *affine* if it can be represented as $\vec{h}(\vec{x}) = A\vec{x} + \vec{b}$, where $A$ and $\vec{b}$ are constants that do not vary with $\vec{x}$.

We now consider a parameterized representation of the scheduling function, following the approach of Feautrier [1992a]. Rather than having the same function for all statements in the program, let us define a scheduling function $\Theta_S$ for each statement $S$. $\Theta_S$ maps the instance of $S$ on iteration $\vec{i}$ to a time of execution. Sometimes we will refer to a statement-iteration pair $S(\vec{i}\,)$ as an *operation*. We assume that $\Theta_S$ is an affine function of the iteration vector and the structural parameters:

$$\Theta_S(\vec{i}, \vec{n}) = \vec{a}_S \cdot \vec{i} + \vec{b}_S \cdot \vec{n} + c_S. \tag{3}$$

The schedule for the entire program, then, is denoted by $\Theta \in \mathcal{E}$, where $\mathcal{E}$ is the space of all the scheduling parameters $(\vec{a}_{S_1}, \vec{a}_{S_2}, \ldots, \vec{a}_{S_{n_S}}, \vec{b}_{S_1}, \vec{b}_{S_2}, \ldots, \vec{b}_{S_{n_S}}, \vec{c}_{S_1}, \vec{c}_{S_2}, \ldots, \vec{c}_{S_{n_S}})$.

This representation is referred to as a one-dimensional affine schedule; it assigns a scalar execution time to each operation as an affine function of the enclosing loop indices and symbolic constants. Given such a schedule, the execution model is simple: all the instructions that are scheduled for a given time step are executed in parallel, with synchronization barriers between one time step and the next. Thus, parallelism is implicit; two operations run in parallel if they are assigned to the same time. The basic execution model assumes that there are an unlimited number of processors, that all operations take unit time, and that there is no communication cost between processors.

Unfortunately, not all programs can be executed according to a one-dimensional affine schedule. Rather, some programs require multidimensional schedules with vector-valued execution times that are executed in lexicographic order. The reader can consult Feautrier [1992a, 1992b] and Darte et al. [2000] for more details on multidimensional schedules; throughout the rest of this paper, we restrict our attention to the one-dimensional case.

Affine schedules provide a very powerful and flexible means of reasoning about the ordering of instructions in a program. For instance, multidimensional affine scheduling techniques are a generalization of standard transformations such as loop permutation, reversal, and skewing, as well as loop distribution and loop fusion [Feautrier 1992b]. There are also techniques for generating efficient code from affine schedules [Quilleré et al. 2000]. However, note that affine scheduling captures only transformations that correspond to a multidimensional affine schedule; that is, the outermost loops are sequential and the innermost loops are parallel, where "parallel" means that 1) all iterations are independent, and 2) all computations within the loop body are independent. Affine schedules do not capture some codes with a more general definition of parallelism [Darte et al. 1997].

As attractive as affine schedules are for their expressivity and succinctness, the most convincing reason to use them as a representation is for the efficient methods that can be used to find, analyze, and optimize them. Feautrier [1992a] provides a direct solution method that supports various optimization metrics within a linear programming framework. In the remainder of this section, we present the foundations of the scheduling problem, and consider two alternative methods for solving it—the Farkas method [Feautrier 1992a] and the vertex method [Quinton 1987]. Our technique relies heavily on both of these methods.

3.2.2 *Finding a Schedule.* A legal schedule is one that respects the dependences in the original program. Among these schedules, our goal is to find the "best" one by some optimization metric, such as the length of the schedule or the delay between producers and consumers. As shown by Feautrier [1992a], this can be done efficiently by formulating the set of legal schedules with a set of linear (in)equalities, and then selecting one using a linear objective function. For a schedule to be legal, it must satisfy the causality condition—that is, each producer must be scheduled before the consumers that depend on it. We now present this schedule constraint in terms of our notations above.

According to dependence $P_j$ (Equation (2)), 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 $P_j$, $j \in [1, n_p]$:

$$\forall \vec{n} \in \mathcal{N}, \quad \forall \vec{i} \in \mathcal{P}_j, \quad \Theta_{R_j}(\vec{i}, \vec{n}) - \Theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}), \vec{n}) - 1 \geq 0. \tag{4}$$

The trouble with this equation is that—as before—we are faced with a parameterized number of constraints. That is, the consumer $R_j$ must be scheduled at least one step after the producer $T_j$ for every point $\vec{i}$ of the dependence domain $\mathcal{P}_j$ and for every possible value of the structural parameters $\vec{n}$. In other words, we are treating $\vec{i}$ and $\vec{n}$ as *variables* that can assume any value within their domains. However, this renders the constraints nonlinear, as $\Theta$ contains two terms that multiply $\vec{i}$ and $\vec{n}$ by the scheduling variables: $\vec{a}_S \cdot \vec{i}$ and $\vec{b}_S \cdot \vec{n}$.

Thus, we need to "linearize" the schedule constraints before we can consider the problem in a linear programming framework. There are two equivalent methods of accomplishing this, each of which relies on a different representation of the polyhedral domains over which the variables $\vec{i}$ and $\vec{n}$ are defined. As described below, the Farkas method uses the polyhedron's faces to perform the reduction, while the vertex method relies on the vertices of the polyhedron. Though both of these techniques are equally powerful in a theoretical sense, we often choose one over the other to match the practical concerns of a given situation.

3.2.3 *Farkas Method.* The Farkas method is founded on the following theorem from linear algebra [Schrijver 1986; Feautrier 1992a; Darte et al. 2000].

THEOREM 1 (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 $\mathcal{W}$. Then an affine form $h$ is nonnegative everywhere in $\mathcal{D}$ if and only if it is a nonnegative affine combination of the affine forms defining $\mathcal{D}$:*

$$\forall \vec{x} \in \mathcal{W}, \quad h(\vec{x}) \equiv \lambda_0 + \sum_{j=1}^{p} (\lambda_j(\vec{a}_j \cdot \vec{x} + b_j)), \quad \lambda_0 \ldots \lambda_p \geq 0.$$

*The nonnegative constants $\lambda_j$ are referred to as Farkas multipliers.*

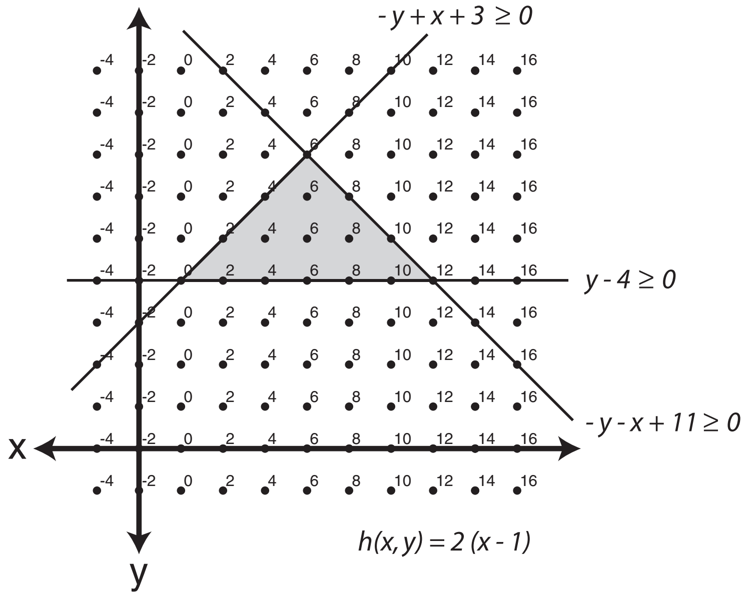

Fig. 2. An illustration of Farkas' lemma. The affine form $h(x, y) = 2 \cdot (x - 1)$ is nonnegative within the shaded polyhedron. Thus, it can be expressed as a nonnegative affine combination of the faces of that polyhedron: $h(x, y) = 2 \cdot (-y + x + 3) + 2 \cdot (y - 4)$.

On first glance, the notations for Farkas' lemma can be somewhat overwhelming. However, the intuition is simple: if a certain function is never negative anywhere inside a certain shape, then that function can be exactly represented as a certain combination of the faces of that shape. Of course, the qualifiers on the "certain" are very important—the function must be affine, the shape must be a nonempty polyhedron, and the combination must be affine, nonnegative, and applied to the inequalities that define the polyhedron's faces (see Figure 2). Note that the use of the identity sign ($\equiv$) indicates an equivalence between functions rather than an equality of values; as these functions are affine, the coefficients of each of the function arguments are identical pairwise.

The schedule constraints (4) can be solved using Farkas' lemma, as shown by Feautrier [Feautrier 1992a, 1992b; Darte et al. 2000]. The result can be expressed as a polyhedron $\mathcal{R}$: the set of all the legal schedules $\Theta$ in the space of scheduling parameters $\mathcal{E}$. Note that Equation (4) does not always have a solution [Feautrier 1992a], in which case multidimensional schedules are needed. However, in this paper, we assume that Equation (4) has a solution.

3.2.4 *Vertex Method.* This method uses Minkowski's representation of polyhedra.

*Definition* 2 (Minkowski's representation of polyhedra). A polyhedron can always be decomposed as the sum of a polytope (i.e., a bounded polyhedron) and a polyhedral cone (see Schrijver [1986] for details). A polytope is defined by its vertices, and any point of the polytope is a (nonnegative) convex combination of the polytope vertices. A polyhedral cone is finitely generated and is defined by its rays and lines. Any point of a polyhedral cone is the sum of a nonnegative combination of its rays and any combination of its lines. Therefore, a polyhedron $\mathcal{D}$ can be equivalently defined by a set of vertices, $\{\vec{v}_1, \ldots, \vec{v}_\omega\}$, a set of rays, $\{\vec{r}_1, \ldots, \vec{r}_\rho\}$, and a set of lines, $\{\vec{l}_1, \ldots, \vec{l}_\lambda\}$. Then $\mathcal{D}$ is the set of all vectors $\vec{p}$ such that

$$\vec{p} = \sum_{i=1}^{\omega} \mu_i \vec{v}_i + \sum_{i=1}^{\rho} \kappa_i \vec{r}_i + \sum_{i=1}^{\lambda} \xi_i \vec{l}_i \tag{5}$$

with $\mu_i \in \mathbb{Q}^+$, $\kappa_i \in \mathbb{Q}^+$, $\xi_i \in \mathbb{Q}$, and $\sum_{i=1}^{\omega} \mu_i = 1$.

The vertices, rays, and lines of a polyhedron can be computed even for parameterized polyhedra [Loechner and Wilde 1997]. The vertex method, introduced by Quinton [1987] for uniform dependences and generalized to affine dependences by Rajopadhye et al. [1986] and Quinton and Dongen [1989], uses Minkowski's representation to rewrite (in)equations which hold on a polyhedron.

THEOREM 2 (THE VERTEX METHOD). *Let $\mathcal{D}$ be a nonempty polyhedron defined by a set of vertices, $\{\vec{v}_1, \ldots, \vec{v}_\omega\}$, a set of rays, $\{\vec{r}_1, \ldots, \vec{r}_\rho\}$, and a set of lines, $\{\vec{l}_1, \ldots, \vec{l}_\lambda\}$. Let $\Phi$ be an affine form of linear part $\vec{a}$ and constant part $b$ ($\Phi(\vec{x}) = \vec{a} \cdot \vec{x} + b$). Then the affine form $\Phi$ is nonnegative over $\mathcal{D}$ if and only if* 1) *$\Phi$ is nonnegative on each of the vertices of $\mathcal{D}$ and* 2) *the linear part of $\Phi$ is nonnegative* (*respectively null*) *on the rays* (*resp. lines*) *of $\mathcal{D}$. This can be written*:

$$\forall \vec{p} \in \mathcal{D}, \quad \vec{a} \cdot \vec{p} + b \geq 0 \quad \Leftrightarrow$$
$$\forall i \in [1, \omega], \quad \vec{a} \cdot \vec{v}_i + b \geq 0, \quad \forall i \in [1, \rho], \quad \vec{a} \cdot \vec{r}_i \geq 0, \ \textit{and} \ \ \forall i \in [1, \lambda], \ \vec{a} \cdot \vec{l}_i = 0.$$

The intuition for this theorem is straightforward: if an affine function is nonnegative at the vertices of a polyhedron, then it is non-negative within that polyhedron, as well (see Figure 3). In theory, this technique is equally powerful as Farkas' lemma for eliminating variables and linearizing constraints such as the schedule constraint posed above. However, it is not the case that the equations yielded by an application of the vertex method are exactly the same as those introduced by the Farkas method [Feautrier 2001b]; the vertex method adds an equation for each vertex, while the Farkas method adds a variable for each face. In the realm of program analysis, it often turns out to be more efficient to use Farkas' lemma, as most of the polyhedra correspond to an $n$-level loop nest that has $2n$ faces but $2^n$ vertices. Balev et al. [1998] confirmed this notion experimentally, showing that the Farkas method is significantly faster than the vertex method for large problem sizes. However, the vertex method is still useful, as there are cases in which there seem to be too few equations to solve the problem with the Farkas method alone. For instance, in our analysis, we cannot linearize the equations further after an application of Farkas' lemma. Instead, we employ the vertex method for the first set of linearizations, and apply the Farkas method (if desired) for the last transformation only.

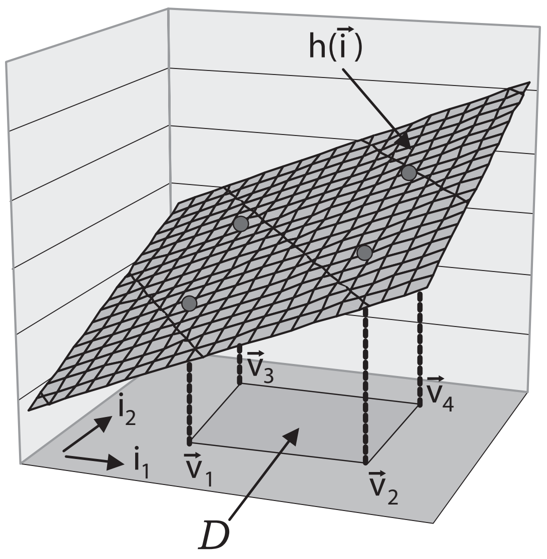

Fig. 3. An illustration of the vertex method. The affine form $h(\vec{i})$ will be non-negative everywhere within $\mathcal{D}$ if and only if it is nonnegative at the vertices $\vec{v}_1 \ldots \vec{v}_4$ of $\mathcal{D}$.

## 3.3 Occupancy Vectors

3.3.1 *Definitions.* To arrive at a simple model of storage reuse, we borrow the notion of an *occupancy vector* from Strout et al. [1998]. Intuitively, the technique works by collapsing all array locations that are separated by an integer multiple of the occupancy vector. For the transformation to be valid, no two collapsed locations can be live at the same time during execution of the original program.

Formally, we can describe an occupancy vector as 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. Letting $l_1$ and $l_2$ denote array locations, the equivalence relation is:

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

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

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

We say that an occupancy vector $\vec{v}$ is *valid* for an array $A$ with respect to a given schedule $\Theta$ if:

(1) transforming $A$ under $\vec{v}$ everywhere in the program does not change the semantics when the program is executed according to $\Theta$
(2) $\forall \vec{l} \in A, \vec{l}$ is written to before $\vec{l} + \vec{v}$ according to $\Theta$

The first part of this definition simply states that, for an occupancy vector to be valid, it must induce a semantics-preserving transformation. The second part states that the occupancy vector must be aligned with the order of computations in the schedule: the location at the tip of the vector must be written after the location at the base. While this is is not fundamental to the concept of an occupancy vector, it constrains the problem in a way that we can solve it. As detailed in Section 4.4, without this proviso we are unable to formulate constraints implied by the occupancy vector, because we do not know if the value overwriting location $\vec{l}$ will come from location $\vec{l}+\vec{v}$ or from location $\vec{l}-\vec{v}$. By fixing the order of computations, we resolve this ambiguity and make the problem solvable at the cost of overconstraining the problem. Note that this aspect of the definition also assumes that each array location is written at most once; as explained in Section 4.4, this is a fundamental limitation of our technique.

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, if one dimension of an array is much larger than another, it might be preferable to lengthen the occupancy vector if doing so allows one to collapse storage along the larger dimension of the array. Because array dimensions are often unknown at compile time, in this paper we refer to the “best” occupancy vector as that which has the shortest length (using the Manhattan distance or a related metric; see Section 4.6.1). We consider it to be a fruitful direction of future research to minimize the overall size of the data space resulting from an occupancy vector transformation.

3.3.2 *Calculating a Storage Mapping*. Given an occupancy vector, we implement the storage transformation using the technique of Quilleré and Rajopadhye [2000], who describe the general case of $n$ occupancy vectors. Our presentation is somewhat simpler because we consider only a single occupancy vector.

First consider the case in which the occupancy vector $\vec{v} \in \mathbb{Z}^n$ is *primitive*, that is, its components are relatively prime. In this case, the new storage cell for location $\vec{l} \in \mathbb{Z}^n$ is given by $\Pi_{\vec{v}}\,\vec{l}$, where $\Pi_{\vec{v}}$ is any $(n-1)\times n$ integral matrix whose null space is spanned by $\vec{v}$. That is, $\Pi_{\vec{v}}\,\vec{l}_1 = \Pi_{\vec{v}}\,\vec{l}_2$ if and only if $\vec{l}_1 - \vec{l}_2 = q\vec{v}$ for some rational number $q$. Note that this is a slightly different condition than that required by the occupancy vector: two locations are collapsed under the occupancy vector if and only if they are separated by an integral (not rational) multiple of $\vec{v}$. However, for primitive occupancy vectors, these conditions are equivalent, as no fraction $p\vec{v}$ (for $p \notin \mathbb{Z}$) of a primitive occupancy vector can also be integral.

There are many valid choices of $\Pi_{\vec{v}}$ for a given occupancy vector $\vec{v}$. A practical choice for $\Pi_{\vec{v}}$ is a matrix that admits a unimodular[3] completion; that is, one that can be augmented with additional rows to obtain a unimodular matrix.

[3]A matrix $M$ is *unimodular* if $M$ is square, integral and $\det M = \pm 1$.

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

Fig. 4. Original code for Example 1.

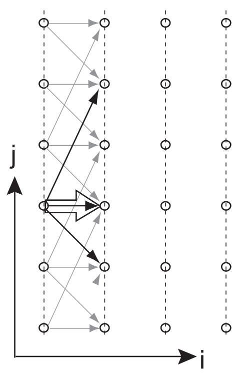

Fig. 5. Iteration space diagram for Example 1. The solid arrows represent dependences, while the dotted lines indicate a schedule in which the columns of the array are written to in parallel. Given this schedule, the shortest valid occupancy vector is (1, 0), depicted by the hollow arrow.

Unimodular transformations facilitate clean code generation, as the transformed data space does not contain any "holes." One procedure for building such a $\Pi_{\vec{v}}$ is as follows. Construct an $n \times n$ unimodular matrix $U_{\vec{v}}$ that contains $\vec{v}$ as the first column (see Darte [1991] for a detailed procedure). Then set $\Pi_{\vec{v}}$ to be rows $2 \ldots n$ of $U_{\vec{v}}^{-1}$. This $\Pi_{\vec{v}}$ obviously admits a unimodular completion because $U_{\vec{v}}^{-1}$ is also unimodular. Also, as $U_{\vec{v}}^{-1} U_{\vec{v}} = I$ and the first column of $U_{\vec{v}}$ is $\vec{v}$, it follows that:

$$U_{\vec{v}}^{-1} \vec{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

and thus $\Pi_{\vec{v}} \vec{v} = \vec{0}$. That is, $\vec{v}$ is in the null space of $\Pi_{\vec{v}}$.

For example, consider the code for Example 1 (duplicated in Figure 4, for easy reference), which we borrow from Strout et al. [1998]. Consider the schedule where all the columns of the array are written to in parallel (that is, iteration $(i, j)$ is executed at time $i$). The shortest valid occupancy vector for this schedule is $(1, 0)$, as depicted in Figure 5. (This calculation assumes that the underlying architecture provides support so that an element can be read and then written to during the same clock cycle; otherwise the shortest valid occupancy vector would be $(2, 0)$.) To generate code for the collapsed storage mapping, we construct:

$$U_{\vec{v}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad U_{\vec{v}}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Pi_{\vec{v}} = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

Each array reference $(i, j)$ is mapped to $\Pi_{\vec{v}} \cdot (i, j) = (0, 1) \cdot (i, j) = j$. Replacing $(i, j)$ by $j$ yields the transformed code shown in Figure 6.

The transformation is more complex for *nonprimitive* occupancy vectors, in which the *GCD* of the components is greater than 1. Such occupancy vectors intersect multiple integral points of the iteration domain (as many as the *GCD*). Due to this fact, it is not sufficient to use only a projection matrix $\Pi_{\vec{v}}$ whose null space is spanned by $\vec{v}$. For example, consider the occupancy vector of $(2, 0)$, which is also valid for the schedule described previously. Using $\Pi_{\vec{v}} = (0, 1)$ collapses points in the correct direction and results in a transformed index of $j$. However, this transformation is incorrect for the occupancy vector of $(2, 0)$, as it coalesces some elements that should actually be in different equivalence classes—for example, the points $(1, 0)$ and $(2, 0)$.

The general storage mapping for primitive and nonprimitive occupancy vectors contains a "modulation term" to distinguish points that are separated by nonintegral multiples of $\vec{v}$. The mapping for location $\vec{l}$ is given by $(\Pi_{\vec{v}_0}\vec{l},\ \vec{y} \cdot \vec{l} \bmod GCD_{\vec{v}})$, where:

—$GCD_{\vec{v}}$ is the *GCD* of the components of $\vec{v}$.
—$\vec{v}_0 = (1/GCD_{\vec{v}})\vec{v}$ is the primitive occupancy vector that is parallel to $\vec{v}$.
—$\vec{y}$ is any vector satisfying $\vec{y} \cdot \vec{v}_0 = 1$. A convenient choice for $\vec{y}$ is the first row of $U_{\vec{v}_0}^{-1}$, which satisfies this property by construction.

The $\Pi_{\vec{v}_0}$ component of the mapping collapses all cells in the direction of the occupancy vector. Note that we use $\Pi_{\vec{v}_0}$ instead of $\Pi_{\vec{v}}$ because it is impossible to construct $\Pi_{\vec{v}}$ in the manner described previously: a vector must have relatively prime components to admit a unimodular completion. The modulation term counts the multiples of the primitive occupancy vector $\vec{v}_0$ that a given location has accumulated in the direction of $\vec{v}$. As $\vec{v}$ intersects $GCD_{\vec{v}}$ integral points, the modulo operation collapses all points separated by integral multiples of $\vec{v}$. Note that this mapping works for both primitive and nonprimitive occupancy vectors, as primitive vectors have $GCD_{\vec{v}} = 1$ which causes the modulation term to remain constant.

To illustrate the storage transformation for nonprimitive occupancy vectors, consider the components of the mapping for $\vec{v} = (2, 0)$:

$$GCD_{\vec{v}} = 2 \quad \vec{v}_0 = (1, 0) \quad U_{\vec{v}_0} = U_{\vec{v}}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Pi_{\vec{v}_0} = (0, 1) \quad \vec{y} = (1, 0)$$

The storage mapping for location $(i, j)$ is thus $((0, 1) \cdot (i, j),\ (1, 0) \cdot (i, j) \bmod 2) = (j,\ i \bmod 2)$. The transformed code appears in Figure 7. Note that the expensive modulo operations can be eliminated using the techniques in Sheldon et al. [2001].

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

Fig. 6. Transformed code for Example 1 for an occupancy vector of (1, 0).

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

Fig. 7. Transformed code for Example 1 for an occupancy vector of (2, 0).

## 4. THE UNIFIED FRAMEWORK

In this section, we describe our technique for considering both schedule optimization and storage optimization within a single framework. While this framework does not simultaneously optimize for scheduling and storage (the optimizations must be done in sequence), it provides a single formulation of the constraints along with techniques that use this formulation to solve for 1) a schedule given a storage mapping, 2) a storage mapping given a schedule, or 3) a storage mapping that is valid for a range of schedules. We consider only one-dimensional affine schedules and storage mappings that are based on occupancy vectors.

The section starts with a description of our program domain and additional notations, and then describes our formulation of the schedule and storage constraints and their conversion into a set of linear inequalities. In order to consider storage mappings that apply to a *range* of schedules, we introduce the notion of an Affine Occupancy Vector and show how to solve for one efficiently. We also discuss how to solve for a schedule that is valid for a range of occupancy vectors. We conclude with a high-level summary of our technique.

### 4.1 Program Domain

The basis for our program domain is that of static control flow programs (see Section 3.1). Additionally, we make a very important and somewhat restrictive assumption: we require 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). As explained in Section 4.4, this assumption is central to our technique; our approach is infeasible without this requirement.

Note that our single-assignment assumption implies that there is no overwriting or anti-dependences in the program; we deal only with flow dependences. While techniques such as array expansion [Feautrier 1988] can be used to convert programs with affine dependences into this single-assignment 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 Example 1, shown previously in Figure 4. It clearly falls within our input domain, as the array accesses are affine, and iteration $(i, j)$ assigns to $A[i][j]$. This example represents a computation where a one-dimensional array $A[j]$ is being updated over a time dimension $i$, and the intermediate results are being stored. We assume that only the values $A[m][1..n]$ are used outside the loop; the other values are only temporary.

## 4.2 Notations

In addition to the notations described in Section 3, we will use the following definitions:

—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 each statement $S$, $S(\vec{i})$ 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 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 $k$.

## 4.3 Schedule Constraints

We will refer to the *schedule constraints* as those which restrict the execution ordering due to data dependences in the original program. Section 3.2.2 formulates the schedule constraints and reviews their solution via classical techniques. For the sake of completeness, we duplicate the mathematical formulation of the schedule constraints here. Due to our single-assignment assumption, all dependences in the program are flow dependences from a producer (writing a value) to a consumer (reading that value). Our technique cannot deal with anti-dependences, hence our single-assignment assumption. We have that, for each dependence $P_j$, $j \in [1, n_p]$, the consumer $R_j$ must execute at least one step after the producer $T_j$:

$$\forall \vec{n} \in \mathcal{N}, \quad \forall \vec{i} \in \mathcal{P}_j, \quad \Theta_{R_j}(\vec{i}, \vec{n}) - \Theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}), \vec{n}) - 1 \geq 0 \tag{6}$$

Following Feautrier [1992a], we can solve these constraints via application of Farkas' lemma to express the range of valid schedules as a polyhedron $\mathcal{R}$ in the space of scheduling parameters $\mathcal{E}$. The reader can refer to Section 5.1.1 for an example of the schedule constraints.

## 4.4 Storage Constraints

4.4.1 *Overview.* Before formalizing the storage constraints, we offer a high-level view of what the storage constraints represent and why we need to restrict our program domain in order to achieve a linear formulation. Consider that, in the original program, statement $T$ writes a value $x$ to location $\vec{l}$ that is later read by statement $R$. An occupancy vector and schedule are valid so long as $\vec{l}$ holds $x$ when $R$ executes. The schedule constraints guarantee that $T$ executes before $R$. The storage constraints guarantee that no other statement writes to location $\vec{l}$ between the execution of $T$ and $R$. Together, these constraints are enough to guarantee that a given occupancy vector and schedule are valid.

The challenge in formulating the storage constraints is prohibiting certain statements from executing between $T$ and $R$. We would like to say: all statements that write to location $\vec{l}$ must either: (a) execute before $T$, or (b) execute after $R$. However, such formulations directly lead to constraints that are quadratic in the scheduling parameters, as the schedule itself determines whether a statement should be subject to constraint (a) or constraint (b). As we seek a system of constraints that is linear in the scheduling parameters, we have chosen to restrict our attention to a class of programs in which the schedule does *not* influence the set of statements that are subject to constraint (a) or (b) above. That is, by placing restrictions on the program, we aim to enforce a fixed order on all of the statements that write to a given location.

The restrictions described previously are all for the sake of achieving this property. The assumption that the iteration space corresponds with the data space is needed for two reasons. First, it disallows any overwriting in the original program, as each location is written once; this implies that all overwriting is caused by the occupancy vector. Second, it allows us to deduce the set of iterations that will overwrite the value produced on iteration $\vec{i}$: by the definition of the occupancy vector, iterations $\vec{i} + k\vec{v}$ for $k \in \mathbb{Z}$ will all write to the same location.

Still, we need to impose an order on the operations that write to a given location. This requires three assumptions: 1) all schedules are one-dimensional and affine, 2) each array is written by a single statement, thereby ensuring a single affine schedule for operations writing to a given array, and 3) the occupancy vector is defined such that $\Theta(\vec{i}) < \Theta(\vec{i} + \vec{v})$. In combination with the assumption about the data space, these restrictions allow us to conclude that the results of iteration $\vec{i}$ will first be overwritten by iteration $\vec{i} + \vec{v}$, *regardless of the schedule*. Using this information, we can formulate the storage constraint that iteration $\vec{i} + \vec{v}$ must not execute before any statement that depends on the result of iteration $\vec{i}$.

We emphasize that all three restrictions in the previous paragraph are needed to obtain our desired result. If a multidimensional schedule was allowed, then the next iteration to overwrite $\vec{i}$ might not be $\vec{i} + \vec{v}$; rather, it would again depend on the schedule. If different portions of a given array were assigned by different statements, then different affine schedules would apply to each piece and the ordering of assignments to a given collapsed location would

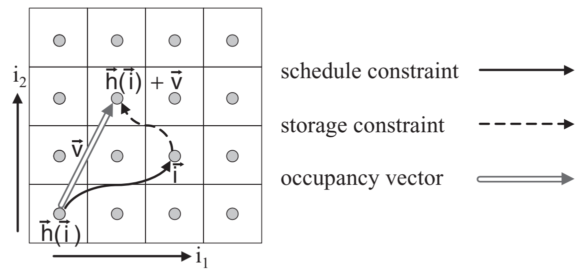

Fig. 8. An illustration of the schedule and storage constraints. The schedule constraint requires that the producer $\vec{h}(\vec{i}\,)$ execute before the consumer $\vec{i}$. The storage constraint requires that the consumer $\vec{i}$ execute before operation $\vec{h}(\vec{i}\,) + \vec{v}$, which overwrites the value of the producer $\vec{h}(\vec{i}\,)$ according to the storage mapping imposed by the occupancy vector $\vec{v}$.

depend on the schedules. The same problem arises for piece-wise affine schedules, as the direction of computations might vary across different portions of the iteration space; we require a single one-dimensional affine schedule per statement (and thus per array). Finally, the definition of the occupancy vector gives us the direction of the writing operations. Without this definition, we would be unable to tell if $\vec{i} + \vec{v}$ or $\vec{i} - \vec{v}$ is the next to overwrite the result of iteration $\vec{i}$.

4.4.2 *Formulating the Constraints.* We now undergo a formal treatment of the storage constraints (illustrated in Figure 8) in the context of our restricted program domain. We consider any array $A$. Thanks to the restrictions detailed above, the value computed by iteration $\vec{i}$ and stored in location $\vec{l}$ is first overwritten by iteration $\vec{i}+\vec{v}_A$. 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 this 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 $P = (R, T, \vec{h}, \mathcal{P})$. Then operation $T(\vec{h}(\vec{i}, \vec{n}))$ produces a value that will later be 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 $\Theta$ and any occupancy vector $\vec{v}_{A(T)}$ respects the dependence $P$ if:

$$\forall \vec{n} \in \mathcal{N}, \quad \forall \vec{i} \in \mathcal{Z}, \quad \Theta_T\big(\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)}, \vec{n}\big) - \Theta_R(\vec{i}, \vec{n}) \geq 0, \tag{7}$$

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 $\vec{n}$.

Note that the schedule (6) and storage (7) constraints might not yield a solution. This could correspond to a case in which the time between production and final consumption of a value is not constant, but grows with the parameters of the program, for example:

```
for i = 1 to N
  A[i] = A[i] + A[N-i+1]
```

In this case, there is no storage reuse that can be captured by the occupancy vector.

An example of the storage constraints can be found in Section 5.1.1.

## 4.5 Linearizing the Constraints

Equations (6) and (7) 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 (6) and storage (7) constraints in forms affine in the scheduling parameters $\Theta$. This step is essential for constructing a linear program that minimizes the length of the occupancy vectors.

As discussed in Section 3.2.2, we could apply either the Farkas method or the vertex method to linearize our constraints. However, the Farkas method can be applied only once due to the nested quantifiers in our equations. Like the vertex method, the Farkas method can yield an infinite set of constraints. However, unlike the vertex method, these constraints might not be affine due to the new variables (the Farkas multipliers) that are introduced. For this reason, we employ the vertex method in this section, and present an application of the Farkas method in Section 4.6.3. Despite its limited applicability, the Farkas method is valuable because it is generally more efficient in practice [Balev et al. 1998].

Section 5.2 contains an illustrative example of the constraint linearization.

4.5.1 *Focusing on Polytopes.* 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. We thus assume that all the polyhedra we manipulate are polytopes, except when stated otherwise. This changes nothing in our method, but simplifies the presentation. Then, according to Theorem 2 (presented in Section 3.2.4), 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 (7).

4.5.2 *Eliminating the Iteration Vector.* Let us consider any fixed values of $\vec{n}$ in $\mathcal{N}$ and $\Theta$ in $\mathcal{R}$ (the range of valid schedules, see Section 4.3). Then, for all

$j \in [1, n_p]$, $\vec{v}_{A(T_j)}$ must satisfy:

$$\forall \vec{i} \in \mathcal{Z}_j, \quad \Theta_{T_j}\big(\vec{h}_j\big(\vec{i}, \vec{n}\big) + \vec{v}_{A(T_j)}, \vec{n}\big) - \Theta_{R_j}\big(\vec{i}, \vec{n}\big) \geq 0, \tag{8}$$

which is an affine inequality in $\vec{i}$ (as $\vec{h}_j$, $\Theta_{T_j}$, and $\Theta_{R_j}$ are affine functions). Thus, according to Theorem 2, 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 $\vec{v}_{A(T_j)}$. Thus, the vertices of $\mathcal{Z}_j$ are also parameterized by $\vec{n}$ and $\vec{v}_{A(T_j)}$, and the number of vertices might change depending on the domain of values of $\vec{n}$ and $\vec{v}_{A(T_j)}$ [Loechner and Wilde 1997]. 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 [Loechner and Wilde 1997], we solve our problem on each of these domains, and we take the "best" solution.

Thus, we evaluate (8)at the extreme points of $\mathcal{Z}_j$, yielding the following:

$$\begin{aligned} \forall k \in [1, n_z], \quad & \Theta_{T_j}\big(\vec{h}_j\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{n}\big), \vec{n}\big) + \vec{v}_{A(T_j)}, \vec{n}\big) \\ & - \Theta_{R_j}\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{n}\big), \vec{n}\big) \geq 0. \end{aligned} \tag{9}$$

According to Theorem 2, Equations (8)and (9)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 $\vec{v}_{A(T_j)}$.

4.5.3 *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 vertices of their domain $\mathcal{N}$. For any fixed value of $\Theta$ in $\mathcal{R}$, $j$ in $[1, n_p]$, and $k$ in $[1, n_z]$ we must have:

$$\begin{aligned} \forall \vec{n} \in \mathcal{N}, \quad & \Theta_{T_j}(\vec{h}_j\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{n}\big), \vec{n}\big) + \vec{v}_{A(T_j)}, \vec{n}\big) \\ & -\Theta_{R_j}\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{n}\big), \vec{n}\big) \geq 0. \end{aligned} \tag{10}$$

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

$$\begin{aligned} \forall l \in [1, n_w], \, & \Theta_{T_j}\big(\vec{h}_j\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{w}_l\big), \vec{w}_l\big) + \vec{v}_{A(T_j)}, \vec{w}_l\big) \\ & - \Theta_{R_j}\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, \vec{w}_l\big), \vec{w}_l\big) \geq 0. \end{aligned} \tag{11}$$

*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 runtime parameter that can be arbitrarily large. In this case, we use the general form of Theorem 2. Let $\vec{r}_1, \ldots, \vec{r}_{n_r}$ and $\vec{l}_1, \ldots, \vec{l}_{n_l}$ be the rays and lines, respectively, defining the unbounded portion of $\mathcal{N}$. We must ensure that the linear part of Equation (11)is nonnegative on these rays and null on these lines. For example, given a single structural parameter $n_1 \in [5, \infty)$, we have the following constraint for the vertex $n_1 = 5$:

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

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

$$\begin{aligned} & \Theta_{T_j}\big(\vec{h}_j\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, 1\big), 1\big) + \vec{v}_{A(T_j)}, 1\big) - \Theta_{R_j}\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, 1\big), 1\big) \\ & \quad - \Theta_{T_j}\big(\vec{h}_j\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, 0\big), 0\big) + \vec{v}_{A(T_j)}, 0\big) + \Theta_{R_j}\big(\vec{z}_{k,j}\big(\vec{v}_{A(T_j)}, 0\big), 0\big) \geq 0. \end{aligned} \tag{12}$$

Though this equation may look complicated, in practice it leads to simple formulas since all the constant parts of Equation (10) 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!

## 4.6 Finding a Solution

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

$$\begin{aligned}&\forall j \in [1, n_p], \quad \forall k \in [1, n_z], \quad \forall l \in [1, n_w],\\ &\quad \Theta_{T_j}\left(\vec{h}_j\left(\vec{z}_{k,j}\left(\vec{v}_{A(T_j)}, \vec{w}_l\right), \vec{w}_l\right) + \vec{v}_{A(T_j)}, \vec{w}_l\right)\\ &\quad -\Theta_{R_j}\left(\vec{z}_{k,j}\left(\vec{v}_{A(T_j)}, \vec{w}_l\right), \vec{w}_l\right) \geq 0,\end{aligned} \tag{13}$$

which is a set of affine inequalities in the coordinates of the schedule $\Theta$, 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:

$$\begin{aligned}&\forall j \in [1, n_p], \quad \forall k \in [1, n_y], \quad \forall l \in [1, n_w],\\ &\quad \Theta_{R_j}(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l) - \Theta_{T_j}(\vec{h}_j(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l), \vec{w}_l) - 1 \geq 0,\end{aligned} \tag{14}$$

where $\vec{y}_{1,j}, \ldots, \vec{y}_{n_y,j}$ denote the vertices of $\mathcal{P}_j$.

4.6.1 *Finding an Occupancy Vector Given a Schedule.* At this point we have all we need to address the first abstract problem (2.1): choosing a storage mapping for a given schedule. To determine which occupancy vectors (if any) are valid for a given schedule $\Theta$, we simply substitute into the simplified storage constraints (13) 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[4] 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 nonlinear objective function.

We can improve our heuristic slightly by running a second linear program that minimizes the difference between the lengths of the occupancy vector components. That is, if we find by the above procedure that the minimum

[4]To minimize $|x|$, introduce a variable $y$ with the inequalities $y \geq x$, $y \geq -x$ (which are equivalent to $y \geq |x|$) and then minimize $y$.

Manhattan length is $k$, then we introduce the following constraint:

$$\sum_{i=1}^{dim(\vec{v})} |v_i| = k$$

and we minimize the following objective function:

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

In the case where there are multiple valid occupancy vectors with the same Manhattan length, this linear program will select the vector that has the most "even" distribution across its components, thereby leading to a shorter Euclidean distance. It has been our experience (on a handful of synthetic examples, analyzed using a Maple script) that this two-step procedure also finds the occupancy vector of the shortest Euclidean distance.

In solving for the shortest occupancy vector, we do need to ensure that each component of $\vec{v}$ is integral. Strictly speaking, this requires an integer linear program with each component constrained to be an integer. Though we have constructed cases for which the optimal rational value of $\vec{v}$ is nonintegral, we have found that in most cases (including all examples in this article), the optimal value of $\vec{v}$ is in fact integral. Thus, in our examples, one can obtain the optimum by solving a linear program instead of an integer linear program.

Note that the minimal value of $\vec{v}$ might be $\vec{0}$. This represents a case in which none of the values of a given array are used by the program. The storage for such an array (as well as the computations writing to the array) can be eliminated.

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

4.6.2 *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 (addressing the abstract problem of Section 2.2). Given an occupancy vector $\vec{v}_A$ for each array $A$ in the program, we substitute into the linearized storage constraints (13) to obtain a set of inequalities where the only unknowns are the scheduling parameters. These inequalities, in combination with the linearized schedule constraints (14) completely define the space of valid affine schedules compatible with the given occupancy vectors. Once again, we can search this space for solutions with any linear programming solver, selecting the "best" schedule as does Feautrier [1992a].

For instance, Figure 9 illustrates the range of valid schedules for Example 1, given the occupancy vector of $(2, 0)$. Section 5.1.3 contains a more detailed solution for this example.

4.6.3 *Finding a Store for a Range of Schedules.* Finally, we might inquire as to the shortest occupancy vector that is valid for all legal affine schedules (corresponding to the abstract problem of Section 2.3). Recall that an affine 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

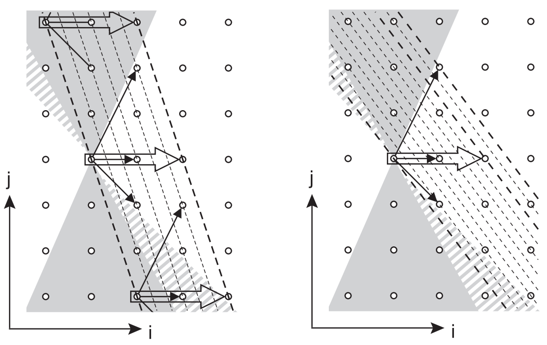

Fig. 9. Iteration space diagram for Example 1. Given an occupancy vector of (2, 0), our method identifies the range of valid schedules. An affine schedule sweeps across the space, executing a line of iterations at once. If this line falls within the gray region (as on the left), then the schedule is valid for the occupancy vector of (2, 0). If this line falls within the striped region (as on the right) then the schedule is valid for some occupancy vector other than (2, 0). The schedule at right is invalid because the operation at the tip of the occupancy vector (2, 0) overwrites a value before the operation at (1, 2) can consume it.

Occupancy Vector:

*Definition* 3. An occupancy vector $\vec{v}$ for array $A$ is an Affine Occupancy Vector (AOV) if it is valid with respect to every affine schedule $\Theta$ that respects the schedule constraints of the original program.

Note that, in contrast to the Universal Occupancy Vector of Strout et al. [1998], an AOV need not be valid for *all* schedules; rather, it only must be valid for affine ones. In this paper, we further relax the definition of an AOV to those occupancy vectors which are valid for all *one-dimensional* affine schedules.

Returning to our example, we find using our method that (2, 1) is a valid AOV (see Figure 10), yielding the transformed storage mapping shown in Figure 11. The code in Figure 11 (and subsequent examples in this article) adopts a one-dimensional affine schedule, as our technique only guarantees that the AOV transformation is valid when executed under such schedules. However, note that the schedule in Figure 11 is only an example; any one-dimensional affine schedule that respects the dependences in the original program will give the same result when executed with the transformed storage.

*Applications of AOVs.* A primary application of AOVs is tiling. Even though tiling utilizes a multidimensional affine schedule and AOVs are only guaranteed to be valid across one-dimensional affine schedules, in certain cases tiling is also legal following storage optimization with an AOV. To extend to a tiled schedule, the AOV needs to be calculated using a strict storage constraint in which operations cannot read a value at the same instant that another operation is overwriting the value (that is, the right-hand side of Equation (7) changes from 0 to 1). If the iteration space has $d$ dimensions, then a one-dimensional affine schedule will still contain $d - 1$ degrees of parallelism, executing several operations at each time step. However, under the strict storage constraint

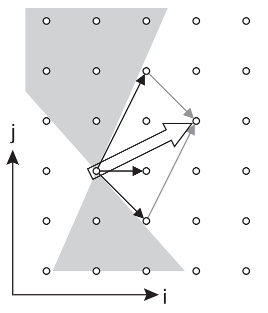

Fig. 10. Iteration space diagram for Example 1. Here the hollow arrow denotes an Affine Occupancy Vector that is valid for all legal one-dimensional affine schedules. The gray region indicates the slopes at which a legal one-dimensional affine schedule can sweep across the iteration domain.

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

Fig. 11. Transformed storage mapping for Example 1, in which the AOV is (2, 1). The loops illustrate an example one-dimensional affine schedule of $\Theta(i, j) = i$.

above, the parallel operations can always be serialized without violating the constraints. This follows from the fact that no operation could be overwriting a value needed by another operation occurring in parallel; otherwise the strict constraint would be violated.

Thus, in each valid one-dimensional affine schedule, one can serialize the remaining parallelism into any arbitrary order. In particular, one can arrange the parallel operations in the order specified by a different valid one-dimensional schedule (independent from the first). If there is still any parallelism remaining, yet another one-dimensional schedule can be used to order the parallel operations. The result of this process is a multidimensional affine schedule, each component of which represents a valid one-dimensional schedule. The multidimensional affine schedule formed in this way is valid; that is, it respects the schedule constraints and the strict storage constraints described previously.

It is easy to show that any multidimensional affine schedule formed in this way can be tiled. A multidimensional affine schedule can be viewed as a series of nested loops, with one loop for each dimension of the schedule. A series of nested loops can be tiled if the loops are fully permutable—that is, they can be freely interchanged while respecting the program constraints. In our construction of the multidimensional schedule, we can add the one-dimensional components in any order. Thus, all orderings are valid with respect to the schedule and storage constraints, which implies that the corresponding loops are fully permutable and thus tilable. Note that if there is parallelism remaining in the innermost loop (i.e., we chose to leave some parallelism in the multidimensional schedule), then additional inner loops may be built from any independent affine expressions; these additional loops may not be tilable, but they do not affect the tiling of the outer loops. The final result is as follows: we can interchange, and thus tile, outer loops built from independent, valid, one-dimensional affine schedules, even if the innermost loops are built from arbitrary affine expressions.

A second potential application of AOVs across one-dimensional affine schedules is in dynamic optimization. Loop bounds and array dimensions are often unknown at compile time, thereby obscuring the metric for the "best" schedule. For instance, consider the range of valid one-dimensional schedules for Example 1 (see Figure 10). A schedule of $\Theta_1(i, j) = i$ can utilize up to `n` (i.e., $j_{max}$) parallel units and will complete in `m` (i.e., $i_{max}$) execution steps. Meanwhile, a schedule of $\Theta_2(i, j) = 2i + j$ demands only $n/2$ parallel units and will complete in no more than $2(m + n)$ execution steps. It could be beneficial to choose one of these schedules (or something in between) at runtime based on the dynamic loop bounds and the available hardware resources. This calls for an AOV storage mapping, as the AOV offers the flexibility to choose between several scheduling alternatives. Though it might also be possible to choose between alternate data layouts at runtime, AOV-mapped storage would be important for global data structures that persist across many independently optimized subroutines.

The reader can verify that the choice of schedules can make an even larger difference in other cases; for example, consider the schedules $\Theta_1(i, j) = i$ vs. $\Theta_2(i, j) = j$ under a uniform data dependence $\vec{h}(i, j) = (i - 1, j - 1)$. While the occupancy vectors $(1, 0)$ and $(0, 1)$ are each valid for exactly one of the schedules, the AOV of $(1, 1)$ provides flexibility to choose either schedule at runtime depending on the relative size of $i_{max}$ and $j_{max}$.

*Finding the AOVs.* Solving for the AOVs is somewhat involved (follow Section 5.1.4 for an example.) To find a set of AOVs, we need to satisfy the storage constraints (13) for any value of the schedule $\Theta$ within the polyhedron $\mathcal{R}$ defined by the schedule constraints (see Section 4.3). To do this, we could use either the Farkas method (Section 3.2.3) or the vertex method (Section 3.2.4). We choose the Farkas method because it is likely to be more efficient in practice [Balev et al. 1998].

To apply Farkas' lemma, we note that the storage constraints are affine inequalities in $\Theta$ 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 notations, let *STORAGE* be the set of expressions that are constrained to be nonnegative by the linearized storage constraints (13). That is, *STORAGE* contains the left-hand side of each inequality in (13). Naively, $|STORAGE| = n_p * n_z * (n_w + n_r)$; 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 (14). The size of *SCHEDULE* is at most $n_p * n_y * (n_w + n_r)$.

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

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

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

These equations are valid over the whole vector space $\mathcal{E}$. Therefore, we can collect the terms for each of the components of $\vec{x}$, as well as the constant terms, setting equal the respective coefficients of these terms from opposite sides of a given equation (see Feautrier [1992a]; Darte et al. [2000] for full details). We are left with $|STORAGE| * (3 * n_s + 1)$ linear equations (recall that $n_s$ denotes the number of statements in the program) where the only variables are the $\lambda$s and the occupancy vectors $\vec{v}_A$.

The set of valid AOVs is completely and exactly determined by this set of equations and inequalities. To find the shortest AOV, we proceed as in Section 4.6.1.

4.6.4 *Finding a Schedule for a Range of Stores.* We note as a theoretical extension that our framework also allows one to solve a problem that is in some sense dual to that of the AOVs: what is a good schedule that is valid for a given range of occupancy vectors? This could also apply to dynamic optimization, as the savings associated with a given occupancy vector are a function of the array dimensions, which often are unknown until runtime. If one dimension turns out to be much larger than another, it would be preferable to collapse storage along the larger dimension (even if the occupancy vector is not the smallest of the alternatives). By choosing a schedule compatible with a range of storage mappings, the dynamic optimizer has the freedom to adjust the mapping at runtime.

This question is also relevant to the phase ordering problem, as one might wish to restrict one's attention to a given set of storage mappings before selecting a schedule $\Theta$. Then, one can choose any storage mapping within the range and be guaranteed that it will be valid for $\Theta$.

Let us denote a vector of occupancy vectors by $\vec{V} \in \mathcal{Y}$, where $\mathcal{Y}$ is the space of all vectors of the occupancy vectors $(\vec{v}_{A_1}, \vec{v}_{A_2} \ldots \vec{v}_{A_{n_A}})$. Our technique allows one to specify a range of storage mappings as a polyhedron of candidate vectors of occupancy vectors. Let us denote this polyhedron by $\mathcal{Q}$, which (by Definition 1) can be defined by $q$ inequalities:

$$\mathcal{Q} = \{\vec{V} \mid \forall j \in [1, q],\ \vec{r}_j \cdot \vec{V} + s_j \geq 0\}$$

It is now the case that the storage constraints (13) must hold for all $\vec{V} \in \mathcal{Q}$. Also, we must satisfy the schedule constraints (14). This set of constraints is nonlinear in its current form, because the storage constraints contain a product of the scheduling parameters $\Theta$ (which are the variables we are seeking) and the set of occupancy vectors $\vec{V}$ (which varies over $\mathcal{Q}$).

To linearize these constraints, we can apply either the Farkas method or the vertex method. We choose the Farkas method for efficiency reasons and proceed as in Section 4.6.3. Note that the storage constraints are affine inequalities in $\vec{V}$ which are nonnegative over the polyhedron $\mathcal{Q}$. Thus, we can express each storage constraint as a nonnegative affine combination of the constraints defining $\mathcal{Q}$. Using the same notations as above for the storage constraints, we have:

$$\mathit{STORAGE}_i(\vec{x}) = \lambda_{i,0} + \sum_{j=1}^{q} (\lambda_{i,j} \cdot (\vec{r}_j \cdot \vec{x} + s_j))$$
$$\lambda_{i,j} \geq 0, \quad \forall \vec{x} \in \mathcal{Y}, \quad \forall i \in [1, |\mathit{STORAGE}|].$$

These equations are valid over the entire vector space $\mathcal{Y}$. Thus, we can equate like terms on the components of $\vec{x}$ as we did in Section 4.6.3 (see Feautrier [1992a]; Darte et al. [2000] for details). We are left with a set of linear equations where the only variables are the $\lambda$s and the scheduling parameters $\Theta$. This set of equations must be considered in combination with the original schedule constraints to define the space of legal schedules. To find a desirable schedule from within this space, we can proceed as in Feautrier [1992a].

Though our framework provides an exact solution to this problem, we are skeptical as to its practical applications in its current form. In part, this is because we believe that a range of candidate occupancy vectors is not best described by a convex polyhedron. For instance, one might be interested in considering all occupancy vectors that are larger than a given length; however, this is not given by a convex shape. Moreover, in relation to the phase ordering problem, it would seem desirable to find a schedule that is valid for *some* occupancy vector of a given length, rather than *all* occupancy vectors in a range. However, this kind of quantification does not integrate with our technique.

It is for these reasons that we omit from our examples and summary statements the problem of finding a schedule for a range of occupancy vectors. Though our solution for this problem is interesting from a theoretical standpoint and could represent a new prospect for dynamic optimization, its usefulness is limited in its current form.

## 4.7 Summary

A high-level view of our method appears in Figure 12. Starting with an input program, we perform some dependence analysis to obtain an affine description of the program dependences. If the input is a static control flow program, this analysis will be accurate. Then, we formulate the schedule and storage constraints, which are nonlinear in their original form due to a product of the scheduling parameters with the iteration vector $\vec{i}$ and the structural parameters $\vec{n}$. To solve this problem, we can apply the vertex method to eliminate $\vec{i}$ and $\vec{n}$ from the constraints. The Farkas method can be substituted for the vertex method at any time, but it can be used only once: no further linearization is possible after the Farkas method has been applied. Once $\vec{i}$ and $\vec{n}$ have been eliminated, we have a linearized set of constraints that can be used to solve the first two problems we considered: finding a good schedule for a given store, and finding a good store for a given schedule. We can continue by eliminating the scheduling parameters $\Theta$, thereby obtaining an integer linear program that yields the shortest AOV—that is, an occupancy vector that is valid for any legal

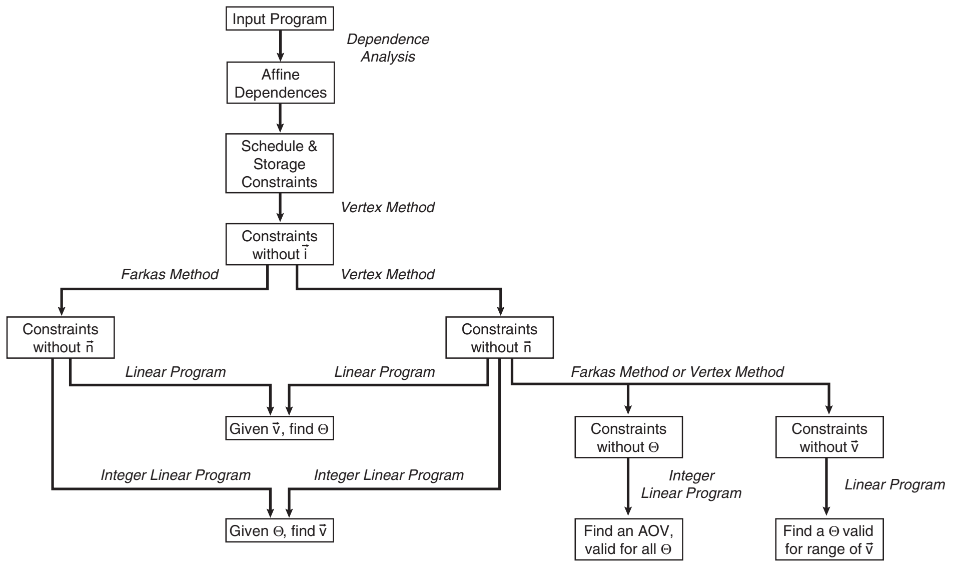
Input Program
Dependence Analysis
Affine Dependences
Schedule & Storage Constraints
Vertex Method
Constraints without $\vec{i}$
Farkas Method
Vertex Method
Constraints without $\vec{n}$
Constraints without $\vec{n}$
Linear Program
Linear Program
Farkas Method or Vertex Method
Given $\vec{v}$, find $\Theta$
Constraints without $\Theta$
Constraints without $\vec{v}$
Integer Linear Program
Integer Linear Program
Integer Linear Program
Linear Program
Given $\Theta$, find $\vec{v}$
Find an AOV, valid for all $\Theta$
Find a $\Theta$ valid for range of $\vec{v}$

Fig. 12. A block diagram of our solution technique.

one-dimensional affine schedule. Similarly, if we are given a range of candidate occupancy vectors, we can eliminate $\vec{v}$ and find a schedule that is valid for all occupancy vectors in the range. The key contribution of our technique is the flexibility that is gained via a single framework, and in particular the ability to solve for a storage mapping that is valid across a range of schedules.

## 5. EXAMPLES

We present four examples to illustrate the method described in the preceding section. The first example contains only a single statement, while the second contains two interdependent statements. The third example is an algorithmic kernel that requires precise dependence domains, and the final example illustrates nonuniform[5] dependences and imperfectly nested loops. Though we derive each equation by hand and use the shortcuts available in each case for the sake of readability, our method is systematic and could be fully automated in a parallelizing compiler.

### 5.1 Example 1: Simple Stencil

First we derive the solutions presented earlier for the 3-point stencil in Example 1 (see Figure 4).

5.1.1 *Constraints.* This example contains a single statement $S$. Its iteration domain is $\mathcal{D}_S$, which represents the set of runtime values for the loop indices that surround $S$:

$$\mathcal{D}_S = \{(i, j) \mid 1 \le i \le m \ \wedge \ 1 \le j \le n\}.$$

[5]A dependence $\vec{h}(\vec{i}) = A\vec{i} + \vec{b}$ is *uniform* if all entries of $A$ are equal to 1.

There are three data dependences, each from statement $S$ unto itself.

$P_1 = (S, S, \vec{h}_1, \mathcal{P}_1)$,
where $\vec{h}_1(i, j) = (i-1, j-2)$ and $\mathcal{P}_1 = \{(i, j) \mid 2 \leq i \leq m \ \wedge \ 3 \leq j \leq n\}$
$P_2 = (S, S, \vec{h}_2, \mathcal{P}_2)$,
where $\vec{h}_2(i, j) = (i-1, j)$ and $\mathcal{P}_2 = \{(i, j) \mid 2 \leq i \leq m \ \wedge \ 1 \leq j \leq n\}$
$P_3 = (S, S, \vec{h}_3, \mathcal{P}_3)$,
where $\vec{h}_3(i, j) = (i-1, j+1)$ and $\mathcal{P}_3 = \{(i, j) \mid 2 \leq i \leq m \ \wedge \ 1 \leq j \leq n-1\}$.

Let us review our notations for the dependence $P_1$ above. $P_1$ indicates that iteration $(i, j)$ of statements $S$ depends on iteration $\vec{h}_1(i, j) = (i-1, j-2)$ of statement $S$. The dependence exists for all $(i, j)$ in the dependence domain $\mathcal{P}_1$, which is the entire iteration domain $\mathcal{D}_S$ except for where $\vec{h}_1(i, j) \notin \mathcal{D}_S$. The other dependences are analogous.

Let $\Theta_S$ denote the scheduling function for statement $S$. Following Section 3.2.1, $\Theta_S$ can be written as follows:

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

Replacing $\Theta_S$ by its definition in dependence $P_1$, we have the following schedule constraint:

$$\begin{aligned} \forall (m, n) \in \mathcal{N}, \quad \forall (i, j) \in \mathcal{P}_1, \quad & (a * i + b * j + c * m + d * n + e) \\ & - (a * (i-1) + b * (j-2) + c * m + d * n + e) - 1 \geq 0. \end{aligned}$$

This can be rewritten as:

$$\forall (m, n) \in \mathcal{N}, \quad \forall (i, j) \in \mathcal{P}_1, \quad a + 2 * b - 1 \geq 0.$$

We can now observe that in this constraint, the domain quantifiers are meaningless, as both $\mathcal{N}$ and $\mathcal{P}_1$ are nonempty[6] but no component of $\vec{n}$ or $\vec{i}$ appears in the equation. Thus, we can eliminate the quantifiers and write the constraint as follows:

$$a + 2 * b - 1 \geq 0.$$

The variables $i$ and $j$ were eliminated from the equation because the dependence $P_1$ is uniform—that is, it does not vary with the loop iteration. Since the other dependences are also uniform, we can follow the same procedure to obtain the following set of inequalities:

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

These are the linearized schedule constraints. Note that since the dependences are uniform, we obtained the linearized schedule constraints without appealing to the vertex method or the Farkas method.

[6]If $\mathcal{N}$ was empty, the program would be meaningless as a loop bound is drawn from $\mathcal{N}$. $\mathcal{P}_1$ could be empty in the trivial case where $m \leq 1$ or $n \leq 2$, in which case we disregard the constraint.

Now let $\vec{v}_A = (v_i, v_j)$ denote the occupancy vector that we are seeking for array A. Using $\mathcal{Z}_1$, $\mathcal{Z}_2$, and $\mathcal{Z}_3$ to denote the storage constraint domains, the storage constraints are as follows:

$$\begin{array}{ll}
\forall(m,n) \in \mathcal{N}, & \forall(i,j) \in \mathcal{Z}_1,\ \Theta_S(i-1+v_i, j-2+v_j, m, n) - \Theta_S(i,j,m,n) \geq 0 \\
\forall(m,n) \in \mathcal{N}, & \forall(i,j) \in \mathcal{Z}_2,\ \Theta_S(i-1+v_i, j+v_j, m, n) - \Theta_S(i,j,m,n) \geq 0 \\
\forall(m,n) \in \mathcal{N}, & \forall(i,j) \in \mathcal{Z}_3,\ \Theta_S(i-1+v_i, j+1+v_j, m, n) - \Theta_S(i,j,m,n) \geq 0.
\end{array}$$

The domain $\mathcal{Z}$ of a storage constraint contains iterations $(i, j)$ where there is a schedule constraint (that is, $(i, j) \in \mathcal{P}$) and where the operation at the tip of the occupancy vector is in the domain of the statement (that is, $\vec{h}(i, j) + (v_i, v_j) \in \mathcal{D}_S$). Substituting the value of $\mathcal{D}_S$, as well as $\vec{h}$ and $\mathcal{P}$ (for each dependence $P$ above) gives the storage constraint domains:

$$\begin{aligned}
\mathcal{Z}_1 = \{(i,j) \mid\ & 2 \leq i \leq m\ \wedge\ 3 \leq j \leq n \\
& \wedge 1\ \leq i-1+v_i \leq m\ \wedge 1 \leq j-2+v_j \leq n\} \\
\mathcal{Z}_2 = \{(i,j) \mid\ & 2 \leq i \leq m\ \wedge\ 1 \leq j \leq n \\
& \wedge 1\ \leq i-1+v_i \leq m\ \wedge 1 \leq j+v_j \leq n\} \\
\mathcal{Z}_3 = \{(i,j) \mid\ & 2 \leq i \leq m\ \wedge\ 1 \leq j \leq n-1 \\
& \wedge 1\ \leq i-1+v_i \leq m\ \wedge 1 \leq j+1+v_j \leq n\}.
\end{aligned}$$

In this case, the details of these domain definitions turn out to be unimportant. In the nontrivial case, each $\mathcal{Z}$ is nonempty, as $m$ and $n$ are larger than 2, and $v_i$ and $v_j$ are significantly smaller than $m$ and $n$ (otherwise there would be no storage reuse with the occupancy vector.) Since each $\mathcal{Z}$ is nonempty, we can eliminate the domain quantifiers and rewrite the storage constraints as we did the schedule constraints. This gives us the linearized storage constraints:

$$\begin{array}{l}
(v_i - 1) * a + (v_j - 2) * b \geq 0 \\
(v_i - 1) * a + v_j * b \geq 0 \\
(v_i - 1) * a + (v_j + 1) * b \geq 0.
\end{array}$$

5.1.2 *Finding an Occupancy Vector.* To find the shortest occupancy vector for the schedule that writes to the rows of $A$ in parallel, we substitute $\Theta_S(i, j, m, n) = i$ into the linearized schedule and storage constraints. That is, we set $a = 1$, $b = 0$, $c = 0$, $d = 0$, and $e = 0$, reducing the constraints to the following:

$$v_i - 1 \geq 0.$$

Minimizing $|v_i| + |v_j|$ with respect to this constraint gives the occupancy vector of $(1, 0)$ (see Figure 5).

5.1.3 *Finding a Schedule.* To find the set of schedules that are valid for the occupancy vector of $(2, 0)$, we substitute $v_i = 2$ and $v_j = 0$ into the linearized schedule and storage constraints.

Simplifying the resulting constraints yields:

$$\begin{array}{l}
a - 2 * b \geq 0 \\
a + 2 * b - 1 \geq 0 \\
a - b - 1 \geq 0.
\end{array}$$

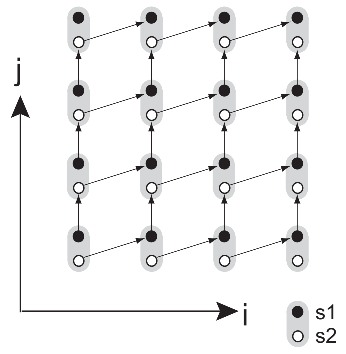

Fig. 13. Dependence diagram for Example 2.

Inspection of these inequalities reveals that the ratio $b/a$ has a maximum value of 1/2 and a minimum value that asymptotically approaches $-1/2$, thus corresponding to the set of legal affine schedules depicted in Figure 10 (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 nonnegative affine combination of the linearized schedule constraints:

$$\begin{bmatrix} (v_i-1)*a+(v_j-2)*b \\ (v_i-1)*a+v_j*b \\ (v_i-1)*a+(v_j+1)*b \end{bmatrix} =$$

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

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

Minimizing $|v_i|+|v_j|$ subject to these constraints yields an AOV $(v_i, v_j) = (2, 1)$.

To transform the data space of array $A$ according to this AOV $\vec{v} = (2, 1)$, we follow the procedure given in Section 3.3. As the components of $\vec{v}$ are relatively prime, it is a primitive occupancy vector and there will be no modulation term. We construct a unimodular matrix $U_{\vec{v}}$ with $\vec{v}$ as the first column and calculate its inverse:

$$U_{\vec{v}} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \quad U_{\vec{v}}^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}.$$

We set the projection matrix $\Pi_{\vec{v}}$ to the bottom rows of $U_{\vec{v}}^{-1}$: $\Pi_{\vec{v}} = [-1\ 2]$. We calculate the index transformation as $(-1,2)\cdot(i,j) = 2*j-i$. Finally, to ensure that all data accesses are nonnegative, we add $m$ to the new index, such that the final transformation is from $A[i][j]$ to $A[2*j-i+m]$. Thus, we have reduced storage requirements from $m*n$ to $m+2*n-1$. The modified code corresponding to this mapping is shown in Figure 11.

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

Fig. 14. Original code for Example 2.

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

Fig. 15. Transformed storage mapping for Example 2, in which each array has an AOV of $(1, 1)$. The loops illustrate the example one-dimensional affine schedules of $\Theta_{S_1}(i, j) = 2*j$, $\Theta_{S_2}(i, j) = 2*j+1$.

## 5.2 Example 2: Two-Statement Stencil

We now consider an example adapted from Lim and Lam [1998] where there is a uniform dependence between two statements in a loop (see Figures 13 and 14). The iteration domains for statements $S_1$ and $S_2$ are as follows:

$$\mathcal{D}_1 = \mathcal{D}_2 = \{(i, j) \mid 1 \le i \le m \ \wedge\ 1 \le j \le n\}.$$

Since $S_1$ and $S_2$ are enclosed in exactly the same loops, their iteration domains are equivalent. There are two data dependences,

$$P_1 = (S_1, S_2, \vec{h}_1, \mathcal{P}_1)$$
$$\text{where } \vec{h}_1(i, j) = (i-1, j) \text{ and } \mathcal{P}_1 = \{(i, j) \mid 2 \le i \le m \ \wedge\ 1 \le j \le n\}$$

$$P_2 = (S_2, S_1, \vec{h}_2, \mathcal{P}_2)$$
$$\text{where } \vec{h}_2(i, j) = (i, j-1) \text{ and } \mathcal{P}_2 = \{(i, j) \mid 1 \le i \le m \ \wedge\ 2 \le j \le n\}.$$

$P_1$ represents the dependence of $S_1$ on $S_2$, and $P_2$ represents the dependence of $S_2$ on $S_1$.

Letting $\Theta_{S_1}$ and $\Theta_{S_2}$ denote the scheduling functions for $S_1$ and $S_2$, respectively, we have the following schedule constraints:

$$\forall(m, n) \in \mathcal{N}, \quad \forall(i, j) \in \mathcal{P}_1, \quad \Theta_{S_1}(i, j, m, n) - \Theta_{S_2}(i-1, j, m, n) - 1 \ge 0$$
$$\forall(m, n) \in \mathcal{N}, \quad \forall(i, j) \in \mathcal{P}_2, \quad \Theta_{S_2}(i, j, m, n) - \Theta_{S_1}(i, j-1, m, n) - 1 \ge 0.$$

and the following storage constraints:

$$\begin{aligned}
&\forall(m,n) \in \mathcal{N}, \quad \forall(i,j) \in \mathcal{Z}_1, \quad \Theta_{S_2}(i-1+v_{B,i}, j+v_{B,j}, m, n) - \Theta_{S_1}(i,j,m,n) \geq 0 \\
&\quad \text{where } \mathcal{Z}_1 = \{(i,j) \mid 2 \leq i \leq m \ \wedge \ 1 \leq j \leq n \ \wedge \\
&\qquad\qquad 1 \leq i-1+v_{B,i} \leq m \ \wedge \ 1 \leq j+v_{B,j} \leq n\} \\
&\forall(m,n) \in \mathcal{N}, \forall(i,j) \in \mathcal{Z}_2, \ \Theta_{S_1}(i+v_{A,i}, j-1+v_{A,j}, m, n) - \Theta_{S_2}(i,j,m,n) \geq 0 \\
&\quad \text{where } \mathcal{Z}_2 = \{(i,j) \mid 1 \leq i \leq m \ \wedge \ 2 \leq j \leq n \ \wedge \\
&\qquad\qquad 1 \leq i+v_{A,i} \leq m \ \wedge \ 1 \leq j-1+v_{A,j} \leq n\}.
\end{aligned}$$

As in Example 1, the dependences are uniform and we can linearize the constraints without appealing to the vertex method or the Farkas method. However, for the sake of illustration, we will use the vertex method to linearize the constraints. We observe that the iteration domains $\mathcal{D}_1$ and $\mathcal{D}_2$ each have vertices $(1,1)$, $(m,1)$, $(1,n)$, and $(m,n)$, so we evaluate the schedule constraints at these points to eliminate $(i,j)$:

$$\begin{aligned}
&\Theta_{S_1}(1,1,m,n) - \Theta_{S_2}(0,1,m,n) - 1 \geq 0 \\
&\Theta_{S_2}(1,1,m,n) - \Theta_{S_1}(1,0,m,n) - 1 \geq 0 \\
&\Theta_{S_1}(m,1,m,n) - \Theta_{S_2}(m-1,1,m,n) - 1 \geq 0 \\
&\Theta_{S_2}(m,1,m,n) - \Theta_{S_1}(m,0,m,n) - 1 \geq 0
\end{aligned}$$

$$\begin{aligned}
&\Theta_{S_1}(1,n,m,n) - \Theta_{S_2}(0,n,m,n) - 1 \geq 0 \\
&\Theta_{S_2}(1,n,m,n) - \Theta_{S_1}(1,n-1,m,n) - 1 \geq 0 \\
&\Theta_{S_1}(m,n,m,n) - \Theta_{S_2}(m-1,n,m,n) - 1 \geq 0 \\
&\Theta_{S_2}(m,n,m,n) - \Theta_{S_1}(m,n-1,m,n) - 1 \geq 0.
\end{aligned}$$

Next, we eliminate the structural parameters $m$ and $n$. Assuming $m$ and $n$ are positive but might be arbitrarily large, the domain of these parameters is an unbounded polyhedron: $(m,n) = (1,1) + j*(0,1) + k*(1,0)$, 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):

$$\begin{aligned}
&\Theta_{S_1}(1,1,1,1) - \Theta_{S_2}(0,1,1,1) - 1 \geq 0 \\
&\Theta_{S_1}(1,1,1,0) - \Theta_{S_2}(0,1,1,0) - \Theta_{S_1}(1,1,0,0) + \Theta_{S_2}(0,1,0,0) \geq 0 \\
&\Theta_{S_1}(1,1,0,1) - \Theta_{S_2}(0,1,0,1) - \Theta_{S_1}(1,1,0,0) + \Theta_{S_2}(0,1,0,0) \geq 0.
\end{aligned}$$

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

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

```
imax = a.length
jmax = b.length
kmax = c.length
A[][][] = 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
        A[i][j][k] = f(i,j,k)                            (S1)
      else
        A[i][j][k] =                                     (S2)
          min(A[i-1][j-1][k-1] + w(a[i],b[j],c[k]),
              A[i][j-1][k-1] + w(GAP,b[j],c[k]),
              A[i-1][j][k-1] + w(a[i],GAP,c[k]),
              A[i-1][j-1][k] + w(a[i],b[j],GAP),
              A[i-1][j][k] + w(a[i],GAP,GAP),
              A[i][j-1][k] + w(GAP,b[j],GAP),
              A[i][j][k-1] + w(GAP,GAP,c[k]))
```

Fig. 16. Original code for Example 3, for multiple sequence alignment. Here $f$ computes the initial gap penalty and $w$ computes the pairwise alignment cost.

These equations constitute the linearized schedule constraints. In a similar fashion, we can linearize the storage constraints to obtain the following:

$$\begin{aligned}
&a_1 = a_2\\
&b_1 = b_2\\
&c_1 = c_2\\
&d_1 = d_2\\
&a_2 * (v_{B,i} - 1) + b_2 * v_{B,j} - e_1 + e_2 - 1 \geq 0\\
&a_2 * v_{A,i} + b_2 * (v_{A,j} - 1) + e_1 - e_2 - 1 \geq 0.
\end{aligned}$$

Using the linearized schedule and storage constraints, we can now apply Farkas' lemma to find the shortest AOVs of $\vec{v}_A = \vec{v}_B = (1, 1)$. The code that results after transformation by these AOVs is shown in Figure 15; memory requirements have been reduced from $2 * m * n$ to $2 * (m + n)$.

### 5.3 Example 3: Multiple Sequence Alignment

We now consider a version of the Needleman-Wunsch sequence alignment algorithm [Needleman and Wunsch 1970] to determine the cost of the optimal global alignment of three strings (see Figure 16). 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 $\Theta_{S_1}$ and $\Theta_{S_2}$ 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):

$$\forall(i, j, k) \text{ s.t. } 3 \le i \le imax \ \wedge\ 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax,$$
$$\Theta_{S_2}(i, j, k, imax, jmax, kmax) - \Theta_{S_2}(i-1, j, k, imax, jmax, kmax) - 1 \ge 0$$

$$\forall(i, j, k) \text{ s.t. } 2 \le i \le imax \ \wedge\ 3 \le j \le jmax \ \wedge\ 2 \le k \le kmax,$$
$$\Theta_{S_2}(i, j, k, imax, jmax, kmax) - \Theta_{S_2}(i, j-1, k, imax, jmax, kmax) - 1 \ge 0$$

$$\forall(i, j, k) \text{ s.t. } 2 \le i \le imax \ \wedge\ 2 \le j \le jmax \ \wedge\ 3 \le k \le kmax,$$
$$\Theta_{S_2}(i, j, k, imax, jmax, kmax) - \Theta_{S_2}(i, j, k-1, imax, jmax, kmax) - 1 \ge 0$$

$$\forall(j, k) \text{ s.t. } 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax,$$
$$\Theta_{S_2}(2, j, k, imax, jmax, kmax) - \Theta_{S_1}(1, j, k, imax, jmax, kmax) - 1 \ge 0$$

$$\forall(i, k) \text{ s.t. } 2 \le i \le imax \ \wedge\ 2 \le k \le kmax,$$
$$\Theta_{S_2}(i, 2, k, imax, jmax, kmax) - \Theta_{S_1}(i, 1, k, imax, jmax, kmax) - 1 \ge 0$$

$$\forall(i, j) \text{ s.t. } 2 \le i \le imax \ \wedge\ 2 \le j \le jmax,$$
$$\Theta_{S_2}(i, j, 2, imax, jmax, kmax) - \Theta_{S_1}(i, j, 1, imax, jmax, kmax) - 1 \ge 0.$$

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. First, there are the storage constraints based on the dependence of $S_2$ on $S_2$:

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 3 \le i \le imax \ \wedge\ 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1 \le i-1+v_i \le imax \ \wedge\ 1 \le j+v_j \le jmax \ \wedge\ 1 \le k+v_k \le kmax,\end{aligned}$$
$$\Theta_{S_2}(i-1+v_i, j+v_j, k+v_k, imax, jmax, kmax) - \Theta_{S_2}(i, j, k, imax, jmax, kmax) - 1 \ge 0$$

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 3 \le i \le imax \ \wedge\ 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1 \le i+v_i \le imax \ \wedge\ 1 \le j-1+v_j \le jmax \ \wedge\ 1 \le k+v_k \le kmax,\end{aligned}$$
$$\Theta_{S_2}(i+v_i, j-1+v_j, k+v_k, imax, jmax, kmax) - \Theta_{S_2}(i, j, k, imax, jmax, kmax) - 1 \ge 0$$

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 3 \le i \le imax \ \wedge\ 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1 \le i+v_i \le imax \ \wedge\ 1 \le j+v_j \le jmax \ \wedge\ 1 \le k-1+v_k \le kmax,\end{aligned}$$
$$\Theta_{S_2}(i+v_i, j+v_j, k-1+v_k, imax, jmax, kmax) - \Theta_{S_2}(i, j, k, imax, jmax, kmax) - 1 \ge 0.$$

Then, there are storage constraints that follow from the dependence of $S_2$ on $S_1$:

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1+v_i = 1 \ \wedge\ 1 \le j+v_j \le jmax \ \wedge\ 1 \le k+v_k \le kmax,\end{aligned}$$
$$\Theta_{S_1}(1+v_i, j+v_j, k+v_k, imax, jmax, kmax) - \Theta_{S_2}(2, j, k, imax, jmax, kmax) - 1 \ge 0$$

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1 \le i+v_i \le imax \ \wedge\ 1+v_j = 1 \ \wedge\ 1 \le k+v_k \le kmax,\end{aligned}$$
$$\Theta_{S_1}(i+v_i, 1+v_j, k+v_k, imax, jmax, kmax) - \Theta_{S_2}(i, 2, k, imax, jmax, kmax) - 1 \ge 0$$

$$\begin{aligned}\forall(i, j, k) \text{ s.t. } & 2 \le j \le jmax \ \wedge\ 2 \le k \le kmax \ \wedge \\ & 1 \le i+v_i \le imax \ \wedge\ 1 \le j+v_j \le jmax \ \wedge\ 1+v_k = 1,\end{aligned}$$
$$\Theta_{S_1}(i+v_i, j+v_j, 1+v_k, imax, jmax, kmax) - \Theta_{S_2}(i, j, 2, imax, jmax, kmax) - 1 \ge 0.$$

```
imax = a.length
jmax = b.length
kmax = c.length
A[][] = new int[imax+jmax][imax+kmax]
...
for iter = 3 to imax + jmax + kmax
  for ALL (i,j,k) s.t. i+j+k=iter, 1<=i<=imax, 1<=j<=jmax, 1<=k<=kmax
    if (i==1) or (j==1) or (k==1) then
      A[jmax+i-j][kmax+i-k] = f(i,j,k)               (S1)
    else
      A[jmax+i-j][kmax+i-k] =                        (S2)
        min(A[jmax+(i-1)-(j-1)][kmax+(i-1)-(k-1)] + w(a[i],b[j],c[k]),
            A[jmax+i-(j-1)][kmax+i-(k-1)] + w(GAP,b[j],c[k]),
            A[jmax+(i-1)-j][kmax+(i-1)-(k-1)] + w(a[i],GAP,c[k]),
            A[jmax+(i-1)-(j-1)][kmax+(i-1)-k] + w(a[i],b[j],GAP),
            A[jmax+(i-1)-j][kmax+(i-1)-k] + w(a[i],GAP,GAP),
            A[jmax+i-(j-1)][kmax+i-k] + w(GAP,b[j],GAP),
            A[jmax+i-j][kmax+i-(k-1)] + w(GAP,GAP,c[k]))
```

Fig. 17. Transformed storage mapping 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]. The loops illustrate an example one-dimensional affine schedule of $\Theta_{S_1}(i, j, k, imax, jmax, kmax) = \Theta_{S_2}(i, j, k, imax, jmax, kmax) = i + j + k$.

In practice, the domains of the three constraints above are empty—and thus no constraints are imposed—as each requires a component of $\vec{v}$ to be zero. However, if any component of $\vec{v}$ is zero, then $\vec{v}$ is not a valid occupancy vector because it violates one of the first three storage constraints (based on the dependence of $S_2$ on $S_2$). Intuitively, one can see this as follows. For any dimension of array $A$, each instance of $S_2$ references two distinct elements of $A$ whose indices differ only in this dimension; if the corresponding component of $\vec{v}$ was zero, then all elements along this dimension of $A$ would be collapsed to a scalar, leaving space for only one of the two elements referenced by $S_2$.

Applying our method for this example yields an AOV of (1, 1, 1). The transformed code under this occupancy vector appears in Figure 17; the transformed array is of dimension [imax+jmax][imax+kmax] and element [i][j][k] of the original array is mapped to [jmax+i−j][kmax+i−k] of the transformed array. Memory requirements have been reduced from $imax * jmax * kmax$ to $(imax + jmax) * (imax + kmax)$.

As described in Section 4.6.3, a tiled loop nest can be constructed using loops that correspond to valid, independent, one-dimensional affine schedules. In this case, such schedules can be constructed as $\Theta_{S_1}(i, j, k, imax, jmax, kmax) = \Theta_{S_2}(i, j, k, imax, jmax, kmax) = ai + bj + ck$ for any $a, b, c \geq 1$. Choosing three independent valuations for $(a, b, c)$ yields schedules corresponding to a three-level loop nest that can be tiled.

## 5.4 Example 4: Nonuniform Dependences

Our final example is constructed to demonstrate the application of our method to nonuniform dependences and imperfectly nested loops (see Figures 18 and 19). Let $\Theta_{S_1}$ and $\Theta_{S_2}$ denote the scheduling functions for statements $S_1$ and $S_2$,

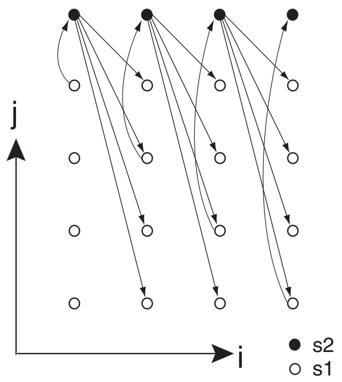

Fig. 18. Dependence diagram for Example 4.

```
A[][] = new int[n][n]
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)
```

Fig. 19. Original code for Example 4.

respectively. Then we have the following schedule constraints:

$$\forall (i, j) \text{ s.t. } 2 \leq i \leq n \ \wedge \ 1 \leq j \leq n, \quad \Theta_{S_1}(i, j, n) - \Theta_{S_2}(i-1, n) - 1 \geq 0$$
$$\forall i \text{ s.t. } 1 \leq i \leq n, \quad \Theta_{S_2}(i, n) - \Theta_{S_1}(i, n-i, n) - 1 \geq 0$$

and the following storage constraints:

$$\forall (i, j) \text{ s.t. } 2 \leq i \leq n \ \wedge \ 1 \leq j \leq n \ \wedge \ 1 \leq i - 1 + v_B \leq n,$$
$$\Theta_{S_2}(i - 1 + v_B, n) - \Theta_{S_1}(i, j, n) \geq 0$$

$$\forall i \text{ s.t. } 1 \leq i \leq n \ \wedge \ 1 \leq i + v_{A,i} \leq n \ \wedge \ 1 \leq n - i + v_{A,j} \leq n,$$
$$\Theta_{S_1}(i + v_{A,i}, n - i + v_{A,j}, n) - \Theta_{S_2}(i, n) \geq 0.$$

Applying our method to these constraints yields the AOVs $\vec{v}_A = (1, 0)$ and $v_B = 1$. The transformed code (shown in Figure 20) requires only $n + 1$ memory locations, compared to the original memory requirements of $n * (n + 1)$.

## 6. RELATED WORK

The work most closely related to ours is that of Strout et al. [1998], 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 AOVs that are shorter than any UOV since the AOV must be valid for a smaller range of schedules; for example, the shortest UOV for Example 1 is (3, 0) whereas our technique finds an AOV of (2, 1). While the analysis of Strout et al. [1998] is limited to a stencil of dependences involving only one statement within a perfectly

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

Fig. 20. Transformed storage mapping for Example 4. The AOVs for $A$ and $B$ are (1, 0) and 1, respectively. The loops illustrate example one-dimensional affine schedules of $\Theta_{S_1}(i, j) = 2 * i$, $\Theta_{S_2}(i, j) = 2 * i + 1$.

nested loop, our method applies to general affine dependences across statements and loop nests (both perfectly nested and nonperfectly nested). Moreover, our framework goes beyond AOVs to unify the notion of occupancy vectors with known affine scheduling techniques.

The leading method for schedule-specific storage optimization in the context of the polyhedral model is that of Darte et al. [2005]. They formulate modular memory allocations in terms of integer lattices, where the basis of the lattice is akin to a set of occupancy vectors that are applied simultaneously. A lattice admits a modular storage mapping that is a generalization of ours (it collapses multiple dimensions of storage). There is also a generalized storage constraint: a lattice is valid for a given array if it does not intersect a convex set $K$ that bounds the shape of the live data elements across all points in the schedule. For multidimensional affine schedules, $K$ can be calculated statically. The authors show that the size of the allocated storage is equivalent to the determinant of the lattice, and they propose several heuristics for minimizing this quantity.

The minimization heuristics generally operate in two stages: first by fixing the direction of the vectors which will form the basis for the lattice, and then by scaling the vectors until the lattice is valid. For certain orientations of the initial basis vectors, the minimization heuristics can be proven to perform within a factor of the optimum (though the factor grows exponentially with the dimensionality of the array). In manipulating the orientation of the lattice, this technique generalizes that of Lefebvre and Feautrier [1998] and possibly that of Quilleré and Rajopadhye [2000]. While the optimization metric is more precise than ours (it minimizes the exact size of the transformed storage instead of the length of the occupancy vector), the lattice technique seems to assume that all loop bounds are compiletime constants. Our technique (as well as Lefebvre and Feautrier [1998]; Quilleré and Rajopadhye [2000]) supports symbolic parameters in the program.

We view our technique as being complementary to that of Darte et al. [2005]. When the schedule is given, their method provides more powerful storage optimization than ours. However, our work addresses a different goal: that of adding flexibility to the scheduling process, such that some storage optimization can precede the final choice of schedules. Integrating our approach with their framework could provide an interesting avenue for future research.

Another approach to schedule-specific storage optimization in the polyhedral model is that of Quilleré and Rajopadhye [2000], which builds on that of Wilde and Rajopadhye [1997]. We also consider this technique to be more powerful than ours for finding a storage mapping given a schedule. Like our technique, their analysis targets static control flow programs in a single assignment form. However, they also support multi dimensional affine schedules, as well as multiple dimensions of storage reuse. The method is optimal in the sense that it finds the maximum number of linearly independent projection vectors for collapsing a given array. It is not clear how storage is minimized along each dimension of the final mapping.

Related approaches to storage management for parallel programs are due to Lefebvre, Feautrier, and Cohen [Cohen and Lefebvre 1999; Cohen 1999; Lefebvre and Feautrier 1998]. Given an affine schedule, Lefebvre and Feautrier [1998] optimize storage first by restricting the size of each array dimension and then by combining distinct arrays via renaming. This work is extended by Cohen and Lefebvre [1999] and Cohen [1999] 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.

De Greef, Catthoor, and De Man describe another approach to storage transformations in the polyhedral model, assuming the schedule is fixed [De Greef et al. 1997b]. In their model, each array is effectively flattened into a one-dimensional array, which is implemented as a circular buffer using a single modulo operation. In flattening a multidimensional array, each permutation and orientation of dimensions is evaluated (e.g., column-major vs. row-major) and the configuration yielding the smallest buffer size is selected. This storage mapping corresponds to a set of transformations that partially overlaps with ours. Both mappings can describe compression along a single dimension of the original array. Our mapping (but not De Greef et al.'s) can describe a projective mapping, compressing in a direction that spans multiple dimensions of the original array. The De Greef et al. mapping (but not ours) can describe complete compression of some array dimensions with partial compression of another, as well as "unaligned" compression in which the buffer size is not a multiple of any dimension size. The same authors describe a second pass in which arrays with non-intersecting lifetimes can reuse storage locations [De Greef et al. 1997a]; this pass uses a greedy layout algorithm that could be run following our optimizations. Murthy and Bhattacharyya [2001, 2004] also study the problem of reusing buffer space in the context of synchronous dataflow graph.

Lim et al. [2001] also address the problem of the interplay between scheduling and storage optimization. Their technique encompasses affine partitioning

[Lim and Lam 1998], array contraction, and generalized blocking to improve cache performance. The algorithm works by extracting independent threads from a program, eliminating array dimensions that are not live within an individual thread, and then interleaving and blocking certain threads to improve memory locality. In the context of phase ordering, this approach could be seen as a sequence of schedule optimization, storage optimization, and another round of schedule optimization (with extra storage adjustments possible to support blocking.) However, their array contraction is more limited than an occupancy vector transformation, as the only direction of collapse is along the original axis of the array, and no modulation is allowed.

Pike [2002] presents a set of storage optimizations for scientific programs, including a flexible array contraction algorithm that appears to preserve the semantics for a range of legal execution orderings. The algorithm targets array accesses that are unimodular functions of the enclosing loop indices. Thus, no statement can write the same element twice; however, multiple statements can write to the same array. The algorithm attempts to generate a different array contraction for each assignment statement, provided that each array reference has a unique source. This model of contraction is more general than occupancy vector methods, as different parts of the array can be collapsed to different extents. However, the program domain is different—it is more restrictive in requiring unimodular accesses, but more general in allowing multiple writes to the same array. A more direct comparison is difficult, as the technique is posed algorithmically instead of in terms of the polyhedral model.

Wong and Delosme [Wong and Delosme 1992; Wong 1989] and Saouter [1992] consider a problem in systolic array synthesis that is related to our storage optimizations. Given a linear schedule and fixed values for all structural parameters in the program, they derive a linear allocation function (mapping operations to processors) that minimizes the number of processors needed. The allocation function is analogous to a transformation by a single primitive occupancy vector. The approach is more precise than ours in that it considers the shape of the allocated region, minimizing the total number of allocated processors rather than minimizing the length of the occupancy vector. The solution is obtained via enumerative search. Wong and Delosme make the search practical by using a nontrivial upper bound on the length of the optimal occupancy vector, while Saouter employs a heuristic and restricts the search to vectors that are "close" to one of the longest diameters of the polyhedral domain (a diameter of a polyhedron is a vector linking two of its vertices). However, the techniques do not apply to the problem considered in this paper, due to a difference in the constraints. In systolic array synthesis, a processor allocation is valid if each processor performs at most one operation per time step; that is, the validity of the allocation depends only on the schedule. In contrast, our storage constraints specify that live values must not be overwritten; that is, the validity of the storage mapping depends on both the schedule and the program dependences. Consequently, the shortest occupancy vector satisfying our constraints can be longer than the upper bound derived by Wong and Delosme. In addition, our technique is not restricted to primitive occupancy vectors.

## 7. 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, under some restrictions explained below, a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid across a range of schedules. Our technique is general and precise, allowing interstatement affine dependences and solving for the shortest occupancy vector using integer linear programming. The analysis we describe could be fully automated within a parallelizing compiler.

We consider this research to be a first step towards automating a procedure that finds the optimal tradeoff between parallelism and storage space. This question is 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.

As the ultimate goal of a storage optimization is to reduce the absolute storage volume, it would be worthwhile to consider the shape of the data space rather than considering only the Manhattan length of the occupancy vector. Unfortunately, the shape of the data space may not be known at compile time; for example, an array of dimension $100 \times n \times m$ contains non-comparable parameters, and it is unclear which dimension is most important to collapse. While other techniques minimize the absolute storage volume by fixing the structural parameters (e.g., [Darte et al. 2005], [Wong and Delosme 1992], and [Saouter 1992]), our focus lies in deriving storage transformations that are valid for all values of the parameters. Future work could explore a hybrid approach, deriving a good storage mapping for various orderings of the parameters and choosing between them at runtime.

It would also be desirable to extend our analysis to a broader domain of programs. In particular, the method would be more general if it could deal with arbitrary affine references on the left-hand side; this not only would widen the input domain, but would allow for the reduction of multiple array dimensions via application of successive occupancy vectors. Also of primary importance is support for multi dimensional schedules, as there are many programs that do not admit a one-dimensional schedule. However, all of our attempts to formulate these extensions have resulted in a set of constraints that is nonlinear. We consider it to be an open question to formulate these problems in a linear programming framework.

In the long term, there are also other questions that one would like a framework such as ours to answer. For instance, what is the range of schedules that is legal for *any* valid occupancy vector? This would capture a notion of storage-independent scheduling, such that one could optimize a schedule without restricting the legal storage options. Perhaps a more natural question is to solve for a schedule that permits the shortest possible occupancy vectors for a given program. However, even in the case of our restricted input domain, both of these questions lead to nonlinear constraints that we cannot readily linearize via the techniques of this paper. We consider it to be an interesting open problem to develop a framework that can answer these questions in a practical way.

## ACKNOWLEDGMENTS

We are grateful to Paul Feautrier for helpful discussions regarding our optimization problems, and to the anonymous reviewers for their insightful comments.

## REFERENCES

BALEV, S., QUINTON, P., RAJOPADHYE, S., AND RISSET, T. 1998. Linear programming models for scheduling systems of affine recurrence equations, a comparative study. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*. 250–258.

BARTHOU, D., COHEN, A., AND COLLARD, J. 2000. Maximal static expansion. *Int. J. Parl. Program. 28*, 3, 213–243.

CLAUSS, P. 1996. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *Proceedings of the 10th ACM International Conference on Supercomputing*. 278–285.

COHEN, A. 1999. Parallelization via constrained storage mapping optimization. In *Proceedings of the 2nd International Symposium on High Performance Computing*. 83–94.

COHEN, A. AND LEFEBVRE, V. 1999. Storage mapping optimization for parallel programs. In *Proceedings of the 5th International Euro-Par Conference*. 375–382.

DARTE, A. 1991. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, VLSI J. 12*, 293–304.

DARTE, A. 1998. Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model. In *Algorithms for Parallel Processing*, M. H. Heath, A. Ranade, and R. S. Schreiber, Eds. IMA Volumes in Mathematics and its Applications, vol. 105. Springer-Verlag, 147–183.

DARTE, A., ROBERT, Y., AND VIVIEN, F. 2000. *Scheduling and Automatic Parallelization*. Birkhäuser, Boston, MA.

DARTE, A., SCHREIBER, R., AND VILLARD, G. 2005. Lattice-based memory allocation. *IEEE Trans. Comput. 54*, 10, 1242–1257.

DARTE, A., SILBER, G.-A., AND VIVIEN, F. 1997. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Paral. Process. Lett. 7*, 4, 379–392.

DE GREEF, E., CATTHOOR, F., AND DE MAN, H. 1997a. Array placement for storage size reduction in embedded multimedia systems. In *Proceedings of the 8th IEEE International Conference on Application-Specific Systems, Architectures and Processors*. 66–75.

DE GREEF, E., CATTHOOR, F., AND DE MAN, H. 1997b. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parall. Comput. 23*, 12, 1811–1837.

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

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Paral. Program. 20*, 1, 23–51.

FEAUTRIER, P. 1992a. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *Int. J. Paral. Program. 21*, 5, 313–347.

FEAUTRIER, P. 1992b. Some efficient solutions to the affine scheduling problem. Part II. Multi-dimensional time. *Int. J. Paral. Program. 21*, 6, 389–420.

FEAUTRIER, P. 1996. *The Data Parallel Programming Model*. LNCS Tutorial, vol. 1132 Chapter Automatic Parallelization in the Polytope Model. Springer Verlag, 79–103.

FEAUTRIER, P. 2001a. Array dataflow analysis. In *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, D. P. Agrawal and S. Pande, Eds. Lecture Notes in Computer Science, vol. 1808. Springer, 173–220.

FEAUTRIER, P. 2001b. The use of Farkas lemma in memory optimization. Unpublished note, June, 2001.

FEAUTRIER, P., COLLARD, J.-F., BARRETEAU, M., BARTHOU, D., COHEN, A., AND LEFEBVRE, V. 1998. The interplay of expansion and scheduling in PAF. Tech. rep. PRiSM, University of Versailles.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*. 319–329.

KOUACHE, R. 2002. Durées de vie et compression mémoire. M.S. thesis, Université Louis Pasteur, Strasbourg. (In French).

LEFEBVRE, V. AND FEAUTRIER, P. 1998. Automatic storage management for parallel programs. *Paral. Comput. 24*, 3–4, 649–671.

LIM, A. W. AND LAM, M. S. 1998. Maximizing parallelism and minimizing synchronization with affine partitions. *Paral. Comput. 24*, 3–4, 445–475.

LIM, A. W., LIAO, S.-W., AND LAM, M. S. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the 8th ACM Symposium on Principles and Practices of Parallel Programming*, 103–112.

LOECHNER, V. AND WILDE, D. K. 1997. Parameterized polyhedra and their vertices. *Int. J. Paral. Program. 25*, 6, 525–549.

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

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2001. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst. 20*, 2, 177–198.

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2004. Buffer merging–a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Elect. Sys. 9*, 2, 212–237.

NEEDLEMAN, S. B. AND WUNSCH, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol. 48*, 443–453.

PIKE, G. 2002. Reordering and storage optimizations for scientific programs. Ph.D. thesis, University of California, Berkeley.

PUGH, W. 1992. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM 8*, 102–114.

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

QUILLERÉ, F., RAJOPADHYE, S., AND WILDE, D. 2000. Generation of efficient nested loops from polyhedra. *Int. J. Paral. Program. 28*, 5, 469–498.

QUINTON, P. 1987. *Automata Networks in Computer Science*, Chapter The systematic design of systolic arrays, 229–260. Manchester University Press.

QUINTON, P. AND DONGEN, V. V. 1989. The mapping of linear recurrence equations on regular arrays. *J. VLSI Sign. Process. 1*, 2, 95–113.

RAJOPADHYE, S., PURUSHOTHAMAN, S., AND FUJIMOTO. 1986. Synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings of the 6th International Conference on Foundations of Software Technology and Theoretical Computer Science*. 488–503.

SAOUTER, Y. 1992. À propos de systèmes d'équations récurrentes. Ph.D. thesis, Université de Rennes 1.

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

Sheldon, J. W., Lee, W., Greenwald, B., and Amarasinghe, S. 2001. Strength reduction of integer divison and modulo operations. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*. 254–273.

Strout, M. M., Carter, L., Ferrante, J., and Simon, B. 1998. Schedule-independent storage mapping for loops. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 24–33.

Wilde, D. and Rajopadhye, S. 1997. Memory reuse analysis in the polyhedral model. *Paral. Proces. Lett. 7*, 2, 203–215.

Wong, Y. 1989. Algorithms for systolic array synthesis. Ph.D. thesis, Yale University.

Wong, Y. and Delosme, J.-M. 1992. Space-optimal linear processor allocation for systolic arrays synthesis. In *Proceedings of the 6th International Parallel Processing Symposium*. 275–282.

Received March 2003; revised February 2005, September 2006; accepted May 2007