A Unified Framework for Schedule and Storage Optimization

by

William Frederick Thies

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Unified Framework for Schedule and Storage Optimization

by

William Frederick Thies

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

We present a unified mathematical framework for analyzing the tradeoffs between parallelism and storage allocation within a parallelizing compiler. Using this framework, we show how to find a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid for all legal schedules. We consider storage mappings that collapse one dimension of a multi-dimensional array, and programs that are in a single assignment form with a one-dimensional 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.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

There are a number of people who made fundamental contributions to the work described in this thesis. I am extremely grateful to my advisor, Saman Amarasinghe, for his unparalleled enthusiasm and technical insight, both during the course of this project and beyond. I am also especially indebted to Frédéric Vivien, who provided critical assistance with the technique (in particular the formulation of the storage constraints and the application of the vertex method) and wrote some parts of Chapter 3; thanks also for patiently correcting my mistakes! Thanks are due to Jeffrey Sheldon, who single-handedly implemented all of the code experiments described in Chapter 5, and at very short notice, too. I am also very grateful to Paul Feautrier for considering some of our optimization problems and discussing directions for future work.

Outside the scope of this project, there are many people that provided invaluable support and encouragement. I owe special thanks to Kath Knobe, who first stimulated my interest in compilers and invested a huge amount in me during my first research experiences at Compaq's Cambridge Research Lab. I also thank the members of the Commit compilers group at MIT for feedback and companionship (and suffering through all my practice talks), and Matt Frank in particular for valuable technical insights. Finally, I thank my parents for their continued love and support–I wouldn't have made it without you.

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

Chapter 1

Introduction[1]

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

While exploiting the parallelism of a program is an important step towards achieving efficiency, gains in parallelism are often overwhelmed by other costs relating to data locality, synchronization, and communication. In particular, with the widening gap between clock speed and memory latency, and with modern memory systems becoming increasingly hierarchical, one can often achieve drastic performance improvements by decreasing the amount of storage space that is required by a program. However, storage optimization has a drawback: it can impose constraints on the order in which instructions can execute, thereby limiting the parallelism of the original program. Striking the right balance between storage space and parallelism is an important unsolved problem in the compilers community.

[1]For the current status of this project, please visit `http://compiler.lcs.mit.edu/aov`. Also, a condensed version of this work can be found in [24].

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

Figure 1-1: Original code.

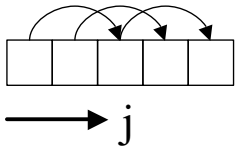

Figure 1-2: Original data space.

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

Figure 1-3: Code following array expansion.

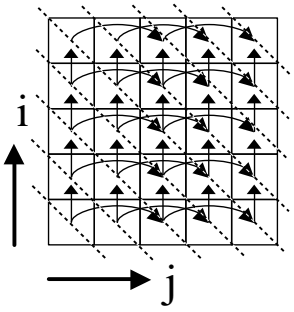

Figure 1-4: Data space after array expansion. The dotted lines indicate elements that can now be written in parallel.

1.1 Motivating Example

We illustrate the tradeoff between parallelism and storage space by means of an example. Figure 1-1 shows a simple loop nest that performs a successive relaxation on the elements of array A. The j loop adjusts each element $A[j]$ as some function of $A[j]$ and $A[j-2]$, while the i loop repeats this update process for a total of n iterations. Figure 1-2 shows the *data space* of A–that is, the layout of the array elements in memory (for the case when $n = 5$). The arrows denote dependences between elements of the array; on each iteration of i, the data stored at the base of an arrow is used to update the element at the arrow's tip. Given this data layout, it is natural to execute the loop sequentially, in which case it will require 25 time steps (or $O(n^2)$ in the general case) to complete the program.

However, one can improve the execution time of this program by performing a transformation known as *array expansion* [5]. Array expansion allocates extra dimensions for a given array, usually to match the number of enclosing loops in the program. Figure 1-3 illustrates our example after A has been expanded across the i

loop; as is evident from the figure, the transformation adjusts the index expressions of A so as to preserve the semantics of the original program. Feautrier developed a general technique [5] that expands arrays to the dimensionality of their enclosing loop nest and applies to any program with static control flow (see Chapter 2).

The benefit of array expansion is that it can eliminate "false dependences" in the original program. Here a brief review of terminology is in order. A *true* dependence (read after write) denotes the actual flow of data in a program; an *anti* dependence (write after read) refers to a write operation that must wait for a true dependence to be satisfied before it can overwrite a given value; and an *output* dependence (write after write) denotes an ordering of write operations that must be preserved so that the final value is correct upon exiting the program. Often, anti- and output-dependences are termed "false dependences" because they are consequences of the storage mapping rather than a fundamental aspect of the algorithm.

In the example above, there is a true dependence of iteration (i, j) on iteration $(i, j-2)$ because the array element $A[j-2]$ is used to update the array element $A[j]$. As the program is written in Figure 1-1, there is also an anti-dependence from iteration (i, j) to iteration $(i+1, j-2)$ because the value $A[j-2]$ must be read on iteration i before it is overwritten on iteration $i+1$. However, this anti-dependence is eliminated by array expansion, as each iteration in the transformed code (Figure 1-3) writes to a distinct memory location, and there is no overwriting of values.

The absence of anti-dependences in the transformed code allows us to execute the loop in a parallel schedule that would have been illegal in the original version. As illustrated in Figure 1-4, this schedule writes to all the diagonals of A in parallel, and completes the entire loop in just 9 time steps (or $\sqrt{2}n$ in the general case). If we had executed this schedule with the original data layout, then the overwriting would cause iteration (i, j) to read the value written by $(i+1, j-2)$ instead of $(i, j-2)$–the anti-dependence would have been violated.

Thus, it is for the purpose of exposing parallelism that many compilers expand arrays in their internal representation of programs [10, 1]. However, expansion comes at a cost, as well, since larger data structures can ruin performance if they cause an

overflow of the register set, cache, or DRAM. 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, then 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 the following section, we consider the abstract questions that one might expect such a framework to address. We then go on to restrict these questions to a practical domain, and to develop a mathematical framework for solving them efficiently.

1.2 Abstract Problem

We now consider a more 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 a dependence of v_2 on the value produced by v_1. Thus, each edge (v_1, v_2) imposes the *schedule constraint* that v_1 be executed before v_2, and the *storage constraint* that the value produced by v_1 be stored until the execution time of v_2.

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

fully associative map of size m. How, then, might we go about choosing Θ and m?

1.2.1 Choosing a Store Given a Schedule

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

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

1.2.2 Choosing a Schedule Given a Store

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

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

1.2.3 Choosing a Store for all Schedules

The final problem is to find the optimal storage mapping that is valid for all legal schedules. That is, we are given a (possibly infinite) set $\Psi = \{\Theta_1, \Theta_2, \dots\}$, where each Θ in Ψ respects the schedule constraints. We choose m such that 1) $\forall \Theta \in \Psi,\ (\Theta, m)$ respects the storage constraints, and 2) m is as small 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 our storage mapping, and then arrange the schedule to optimize for data locality, synchronization, or communication, without worrying about violating the storage constraints.

Such flexibility could be critical if, for example, we want to apply loop tiling [11] in conjunction with storage optimization. If we optimize storage too much, tiling could become illegal; however, we sacrifice efficiency if we don't optimize storage at all. Thus, we optimize storage as much as we can without invalidating a schedule that was valid under the original storage mapping.

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

1.3 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 constants are known, the problem sizes are too large for schedule and storage analysis on a DAG, and the executable grows to an infeasible size if a static instruction is generated for every node in the DAG.

Accordingly, we make two sets of simplifying assumptions to make our analysis tractable. The first concerns the nature of the dependence graph G and the scheduling function Θ. Instead of allowing arbitrary edge relationships and execution orderings, we restrict our attention to affine dependences and affine schedules [7]. The second assumption concerns our approach to the optimized storage mapping. Instead of allowing a fully associative map of size m, as above, we employ the *occupancy vector* as a mechanism of storage reuse [23]. Chapter 2 contains more background information on affine schedules and occupancy vectors.

In this context, we show how to determine 1) a good storage mapping for a given schedule, 2) a good schedule for a given storage mapping, and 3) a good storage mapping that is valid for all legal schedules. We consider storage mappings that collapse one dimension of a multi-dimensional array, and programs that are in a single assignment form with a one-dimensional affine schedule. Our technique incorporates general affine dependences across statements and loop nests, making it applicable to several scientific applications. Our method is precise and practical in that it reduces to an integer linear program that can be solved with standard techniques.

The rest of this thesis is organized as follows. In Chapter 2 we review the mathematical background that forms the basis for our technique. Chapter 3 formulates our method abstractly, and Chapter 4 illustrates our method with examples. Experiments are described in Chapter 5, related work in Chapter 6, and we conclude in Section 7.

Chapter 2

Background

In this chapter, we present background material that is needed to understand our technique. In Section 2.1 we discuss the occupancy vector as a means of collapsing storage requirements, in Section 2.2 we review the polyhedral model for representing programs, and in Section 2.3 we discuss Feautrier's approach to the affine scheduling problem. Those who are already familiar with occupancy vectors, the polyhedral model, and affine scheduling can skip most of this chapter, but might want to review our notation in Sections 2.2.3 and 2.3.1.

2.1 Occupancy Vectors

To arrive at a simple model of storage reuse, we borrow the notion of an *occupancy vector* from Strout et al. [23]. The strategy is to reduce storage requirements by defining equivalence classes over the locations of an array. Following a storage transformation, all members of a given equivalence class in the original array will be mapped to the same location in the new array. The equivalence relation is:

$$R_{\vec{v}} = \{(\vec{l_1}, \vec{l_2}) \mid \exists k \in \mathbb{Z} \; s.t. \; \vec{l_1} = \vec{l_2} + k \cdot \vec{v}\}$$

and we refer to $\vec{v}$ as the *occupancy vector*. We say that A' is the result of *transforming*

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

Figure 2-1: Original code for Example 1.

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

Figure 2-2: Transformed code for Example 1 for an occupancy vector of (0, 1).

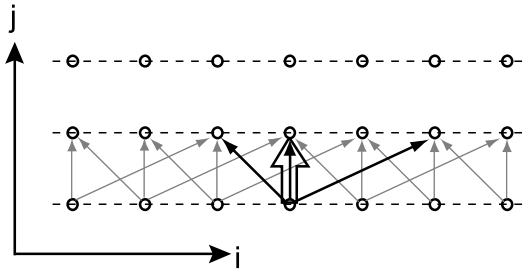

Figure 2-3: Iteration space diagram for Example 1. Given the schedule where each row is executed in parallel, the shortest valid occupancy vector is $(0, 1)$.

A under the occupancy vector $\vec{v}$ if, for all pairs of locations $(\vec{l_1}, \vec{l_2})$ in A:

$$R_{\vec{v}}(\vec{l_1}, \vec{l_2}) \iff \vec{l_1} \textit{ and } \vec{l_2} \textit{ are stored in same location in } A'$$

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

Given an occupancy vector, we implement the storage transformation using the technique of [23] in which the original data space is projected onto the hyperplane perpendicular to the occupancy vector, and modulation is further used to distinguish colliding points. For example, consider the example code given in Figure 2-1, which we borrow from [23]. Consider the schedule where all the rows of the program are executed in parallel. The shortest valid occupancy vector for this schedule is $(1, 0)$, as depicted in Figure 2-3. (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 $(0, 2)$.) To generate code for the collapsed storage mapping, we identify $(1, 0)$ as the hyperplane

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

Figure 2-4: Transformed code for Example 1 for an occupancy vector of (0, 2).

perpendicular to the occupancy vector, and project each array reference (i, j) using the scalar product: $(i, j) \cdot (1, 0) = i$. In this case, replacing each array reference (i, j) with i yields the transformed code, as shown in Figure 2-2.

The transformation needs one extra step in cases where the occupancy vector intersects multiple integral points of the iteration domain. For example, consider the occupancy vector of $(0, 2)$, which is also valid for the schedule described above. Following the projection as before, we identify the perpendicular hyperplane of $(2, 0)$ and obtain an index expression of $(i, j) \cdot (2, 0) = 2i$. However, this expression coalesces some elements that should actually be in different equivalence classes according to the occupancy vector–for example, the points $(0, 1)$ and $(0, 2)$. This occurs because the occupancy vector is *non-primal*–that is, the GCD of its coordinates is not a prime number, and thus there are multiple integral points of the iteration domain that lie along its path. In order to distinguish between these points, we need to add a "modulation term" to the index expression that returns a unique value for each point along the path. In this case, such an expression is $(j \ mod \ 2)$, and we arrive at the complete index transformation of $2i + (j \ mod \ 2)$, as shown in Figure 2-4.

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

2.2 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 2-1 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.

To address this problem, Feautrier paved the way by showing that a certain class of programs are mathematically equivalent to a System of Affine Recurrence Equations (SARE's) over polyhedral domains [7]. In Section 2.2.1, we give a brief overview of SARE's and explain why they are attractive from the standpoint of program analysis. In Section 2.2.2 we describe the class of programs that can be represented as a SARE, and in Section 2.2.3 we describe our notation for the polyhedral model.

2.2.1 System of Affine Recurrence Equations

The representation of computations as a system of recurrence equations was pioneered by Karp et. al [12], and the SARE representation has since found widespread use for scheduling and automatic parallelization. A formal description of a SARE can be found in many papers (e.g., [7]), so our description will be informal (we save our mathematical notation for Section 2.2.3).

A SARE consists of a set of recursive equations between some variables, each of which can be thought of as a multi-dimensional array. Each variable has a domain over which it is defined, and this domain must be polyhedral. A polyhedron is any section of space defined by the intersection of planes or, equivalently, by a system of

linear inequalities. A further requirement is that each array index must be an affine function [1] of the iteration that is being defined. A trivial example of a SARE is for the first n Fibonacci numbers:

$$F(i) = i \qquad \text{for } 0 \leq i \leq 1$$
$$F(i) = F(i-1) + F(i-2) \qquad \text{for } 2 \leq i \leq n$$

For any value of n, the above equations enumerate the value of the first n Fibonacci numbers. Though the domain of the variable F can be of arbitrary size, we can represent it with a finite number of points–namely, its endpoints 0 and n. In higher dimensions, we can always represent a polyhedron by the finite set of its vertices, or, alternatively, by its sides. In this way, the SARE representation provides exactly what we were seeking: a finite, parameterized representation of an infinite set.

2.2.2 Static Control Flow Programs

We now turn our attention to the task of representing a program in the polyhedral model, which is mathematically equivalent to a SARE. For a certain class of applications, known as *static control flow programs*, the polyhedral model provides an *exact* description of the flow dependences. For other programs, the dependences can be approximated conservatively; however, the primary gains will be had when there is static control flow. In order to put the model in context, then, we first list the characteristics of a static control flow program. They are as follows:

- All statements consist of assignments to array variables, with the right hand side being an arbitrary expression of array variables. (A scalar is considered to be a zero-dimensional array.)

- All array accesses have index expressions that are affine functions of the loop indices and the structural parameters. This applies to both array references (on the right hand side) as well as array assignments (on the left hand side).

[1] A function $f(\vec{x})$ is an affine function of $\vec{x}$ if it is of the form $f(\vec{x}) = \vec{a} \cdot \vec{x} + b$, where $\vec{a}$ and $\vec{b}$ are constants that do not depend on $\vec{x}$.

- All loop bounds are either 1) affine expressions of enclosing loop indices or the structural parameters, or 2) the MIN or MAX of a finite set of such expressions.

- All conditional statements depend only on an affine inequality involving the surrounding loop indices and the structural parameters.

Programs meeting these conditions can have all of their control flow analyzed at compile time. In other words, the compiler can statically analyze exactly which statements will be invoked, as well as the value of the loop indices for each invocation. As we will see below, this property allows an affine description of the dependences to exactly capture the flow of data between operations in the program. This is in contrast to other dependence representations–such as Allen & Kennedy's dependence levels, Wolf & Lam's direction vectors, and Darte & Vivien's uniformized dependences–which provide a less precise approximation of the dependence information [4].

2.2.3 Notation

We now present our notation for the polyhedral model. We will refer to this notation for the remainder of this thesis. It is as follows:

- 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 2-1.

- 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 2-1, we have $\vec{n} = (\mathrm{m}, \mathrm{n})$.

- 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 2-1 there is one assignment statement S, and its domain is the polyhedron $\mathcal{D}_S = \{(i,j) \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq m\}$.

- There are n_p dependences $P_1 \ldots P_{n_p}$ in the program. Each dependence P_j is a 4-tuple $(R_j, T_j, \mathcal{P}_j, h_j)$ where R_j and T_j are statements, h_j is a vector-valued affine function, and $\mathcal{P}_j \subseteq \mathcal{D}_{R_j}$ is a polyhedron such that:

$$\forall \vec{i} \in \mathcal{P}_j, R_j(\vec{i}) \textit{ depends on } T_j(\vec{h}_j(\vec{i}, \vec{n})) \tag{2.1}$$

  The dependences P_j are determined using an array dataflow analysis, e.g., [6] or the Omega test [20].

  In the case of Figure 2-1, there are three dependences, each corresponding to an array index 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 these pre-loop assignments for the sake of this example). The dependences are as follows:

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

This is all the notation we need to describe a program in the polyhedral model. We now turn to our description of the schedule.

2.3 Affine Scheduling

2.3.1 Representing a Schedule

In Section 1.2 we considered a scheduling function Θ that governs the execution of all the instructions in the program. We now consider a parameterized representation of the scheduling function, following Feautrier's approach in [6]. Rather than have a single function for the entire program, let us define a scheduling function θ_S for each statement S that maps the instance of S on iteration $\vec{i}$ to a scalar execution time. Sometimes we will refer to a statement-iteration pair $(S, \vec{i})$ as an *operation*. We assume that θ_S is an affine function of the iteration vector and the structural parameters:

$$\theta_S(\vec{i}, \vec{n}) = \vec{a}_S \cdot \vec{i} + \vec{b}_S \cdot \vec{n} + c_S$$

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

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. However, a number of recent works have built on this framework to include more sophisticated machine models (e.g. the affine partitioning approach of Lim & Lam [15]).

Unfortunately, not all programs can be executed according to a one-dimensional affine schedule. Rather, some programs require multi-dimensional schedules with vector-valued execution times that are executed in lexicographic order. For example, the following program cannot be executed with a one-dimensional schedule, since

Figure 2-5: Examples of Affine Schedules

Original
$\theta_{S1}(i,n) = i$
$\theta_{S2}(i,n) = n + i$
	for i = 1 to n
	  $S1$
	for i = 1 to n
	  $S2$

Loop Interchange
$\theta_{S1}(i,n) = n + i$
$\theta_{S2}(i,n) = i$
	for i = 1 to n
	  $S2$
	for i = 1 to n
	  $S1$

Loop Reversal
$\theta_{S1}(i,n) = n - i$
$\theta_{S2}(i,n) = 2n - i$
	for i = n downto 1
	  $S1$
	for i = n downto 1
	  $S2$

Loop Fusion
$\theta_{S1}(i,n) = 2i$
$\theta_{S2}(i,n) = 2i + 1$
	for i = 1 to n
	  $S1$
	  $S2$

Loop Parallelization
$\theta_{S1}(i,n) = 1$
$\theta_{S2}(i,n) = 2$
	for ALL i = 1 to n
	  $S1$
	for ALL i = 1 to n
	  $S2$

there is a dependence chain of length n^2:

```
for i = 1 to n
  for j = 1 to n
    x = f(x) + g(i, j)
```

If one tries to execute this loop with a one-dimensional schedule, the function might be something like $\theta(i,j) = n * i + j$. However, this is no longer affine in the iteration vector and the structural parameters, as there is a product of n and i in the schedule. What is needed is a multi-dimensional schedule such as $\theta(i,j) = (i,j)$ which expresses the sequential loop execution without multiplying terms. The reader can consult [7, 8, 4] for more details on multi-dimensional schedules; throughout the rest of this thesis, we restrict our attention to the one-dimensional case.

Affine schedules provides a very powerful and flexible means of reasoning about the ordering of instructions in a program. For instance, it has been shown that affine

scheduling techniques are a generalization of standard transformations such as loop permutation, skewing, and reindexing, as well as loop distribution and loop fusion (see Figure 2-5).

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 provides a direct solution method that supports various optimization metrics within a linear programming framework [7]. The technique has become known as the Farkas method because it relies on a form of Farkas' Lemma from linear algebra. In the remainder of this chapter, we present the foundations of the scheduling problem, and consider two alternative methods for solving it–the vertex method and the Farkas method. Both of these methods prove to be integral to our technique.

2.3.2 Finding a Schedule

A legal schedule is one that respects the dependences in the original program. Amongst 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 in [7], this can be done efficiently by formulating the set of legal schedules with a set of linear (in)equalities, and then choosing amongst them with 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 notation above.

According to dependence P_j (Equation (2.1)), for any value of $\vec{i}$ in $\mathcal{P}_j$, operation $R_j(\vec{i})$ depends on the execution of operation $T_j(\vec{h}_j(\vec{i},\vec{n}))$. Therefore, in order to preserve the semantics of the original program, in any new order of the computations, $T_j(\vec{h}_j(\vec{i},\vec{n}))$ must be scheduled at a time strictly *earlier* than $R_j(\vec{i})$, for all $\vec{i} \in \mathcal{P}_j$. We express this constraint in terms of the scheduling function. We must have, for each dependence P_j, $j \in [1, n_p]$:

$$\forall \vec{n} \in \mathcal{N}, \forall \vec{i} \in \mathcal{P}_j, \theta_{R_j}(\vec{i},\vec{n}) - \theta_{T_j}(\vec{h}_j(\vec{i},\vec{n}),\vec{n}) - 1 \geq 0 \tag{2.2}$$

The trouble with this equation is that–as before–we are faced with an infinite 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 non-linear, as θ contains two terms which 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 scheduling 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 sides 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 prefer one to another for practical considerations.

2.3.3 Farkas Method

The Farkas method is founded on the following theorem from linear algebra.

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

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

The nonnegative constants λ_j are referred to as Farkas multipliers.

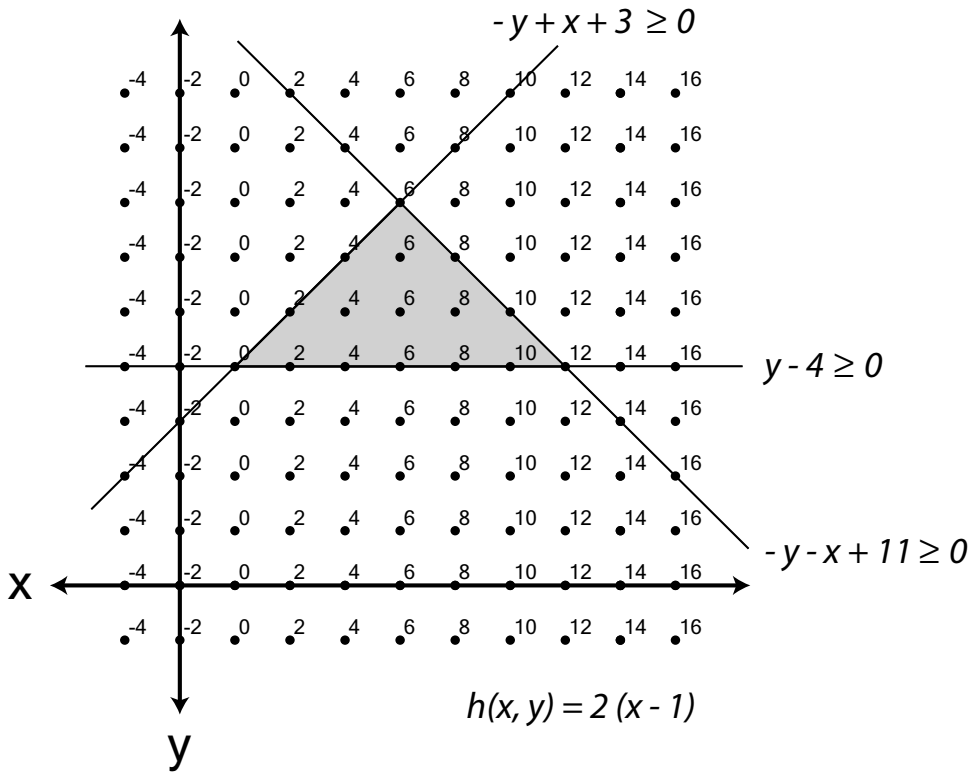

Figure 2-6: An illustration of Farkas' lemma. The affine field $h(x, y) = 2 \cdot (x - 1)$ is non-negative within the shaded polyhedron. Thus, it can be expressed as a non-negative affine combination of the sides of that polyhedron: $h(x, y) = 2 \cdot (-y + x + 3) + 2 \cdot (y - 4)$.

Admittedly, the notation 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 sides of that shape. Of course, the qualifiers on the "certain" are very important–the function must be affine, the shape must be a non-empty polyhedron, and the combination must be affine and applied to the inequalities that define the polyhedron's sides (see Figure 2-6). Since Feautrier's use of Farkas' lemma in [7], there have been several other applications within the realm of parallelizing compilers; for example, it provides the basis of the affine partitioning techniques developed by Lim & Lam [15].

The method proceeds via two applications of this lemma to the affine scheduling problem that we have formulated above. We give only the intuition here; for a more formal treatment, the user can refer to [7, 8, 4]. The first step of the method is to

express each scheduling function θ_S in terms of the parameters defining the domain $\mathcal{D}_S$ of the statement S. This can be done by assuming that each schedule is non-negative (since it must start somewhere, this is not restrictive) and then applying Farkas' Lemma to equate each scheduling function as a non-negative affine combination of the sides defining the domain $\mathcal{D}_S$. The lemma applies in this case by virtue of the fact that the schedules are affine in $\vec{i}$ and are non-negative for all values of $\vec{i}$ that fall within the polyhedral domain $\mathcal{D}_S$. This step does not linearize the scheduling constraint posed above; rather, it provides an alternate representation of the schedules in terms of the loop bounds instead of the scheduling parameters.

The second step involves another application of Farkas' lemma, this time in order to eliminate $\vec{i}$ and $\vec{n}$ from the constraints, thereby obtaining equations that are linear in the parameters. This time the non-negative affine form is the *delay* between each consumer and producer that are related by a data dependence. According to the causality constraint in Equation 2.2, this delay must be positive for all iterations that fall within the domain $\mathcal{P}$ of a given dependence. Thus, we can express the delay as a non-negative affine combination of the inequalities defining the dependence domain. The resulting equation will hold for *all* values of $\vec{i}$ and $\vec{n}$ (not just those within some polyhedron) and thus we can eliminate $\vec{i}$ and $\vec{n}$ from the equation by collecting like terms and setting their coefficients to be equal.

The resulting equations will be linear, and can be submitted to a linear programming solver with a variety of objective functions to guide the selection of a good schedule.

2.3.4 Vertex Method

The method above can also be posed in terms of a polyhedron's vertices, rather than its sides. Any nonempty polyhedron is fully defined by its vertices, rays and lines [22], which can be computed even in the case of parameterized polyhedra [17]. The following theorem explains how we can use these vertices, rays and lines to reduce the size of our sets of constraints.

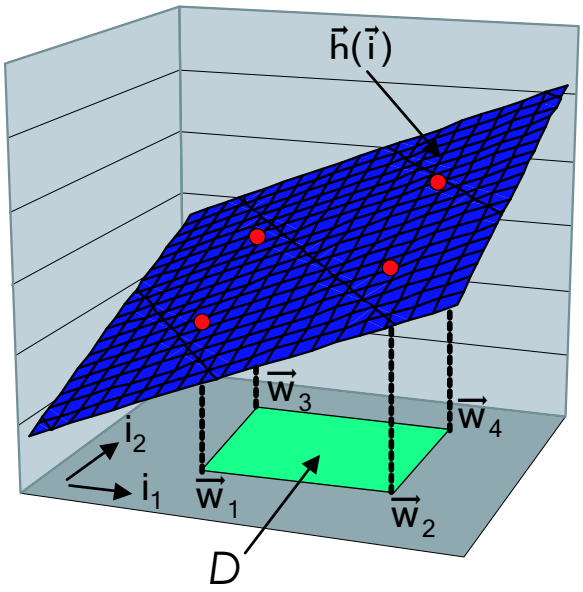

Figure 2-7: An illustration of the vertex method. The affine field $h(\vec{i})$ will be non-negative everywhere within $\mathcal{D}$ if and only if it is non-negative at the vertices $w_1 \ldots w_4$ of $\mathcal{D}$.

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

The intuition for this theorem is straightforward: if an affine function is non-negative at the vertices of a polyhedron, then it must be non-negative within that polyhedron, as well (see Figure 2-7). In theory, this technique is equally powerful as Farkas' lemma for eliminating variables and linearizing constraints such as the scheduling 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 [9]; the vertex method adds an equation for each vertex, while the Farkas method adds a variable for each side. In the realm of program analysis, it often turns out to be more efficient to use Farkas' lemma, as most of the polyhedrons correspond to an n-level loop nest which has $2n$ sides but 2^n vertices.

However, the vertex method is still useful in cases where there are not enough equations to solve the problem with the Farkas method alone; for instance, our analysis succeeds if one step is done with the vertex method, but we have found it to be infeasible when using only Farkas' lemma.

Chapter 3

The Method

In this chapter, we describe our technique for considering both schedule optimization and storage optimization within a single framework. We start with a description of our program domain and additional notation, and then describe our formulation of the 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 efficiently solve for one. Also, we show how to solve for a schedule that is valid for a range of occupancy vectors. Finally, we conclude with a high-level summary of our technique.

3.1 Program Domain

The basis for our program domain is that of static control flow programs (see Chapter 2). Additionally, we further assume a single-assignment form where the iteration space of each statement exactly corresponds with the data space of the array written by that statement. That is, for array references appearing on the left hand side of a statement, the expression indexing the i'th dimension of the array is the index variable of the i'th enclosing loop (this is formalized below). While techniques such as array expansion [5] can be used to convert programs with affine dependences into this form, our analysis will be most useful in cases where an expanded form was obtained for other reasons (e.g., to detect parallelism) and one now seeks to reduce storage requirements.

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

Figure 3-1: Original code for Example 1.

We will refer to the example in Figure 2-1 (which, for easy reference, is duplicated in Figure 3-1), borrowed from [23]. It clearly falls within our input domain, as the dependences have constant distance, and iteration (i, j) assigns to $A[i][j]$. This example represents a computation where a one-dimensional array $A[i]$ is being updated over a time dimension j, and the intermediate results are being stored. We assume that only the values $A[1..n][m]$ are used outside the loop; the other values are only temporary.

3.2 Notation

In addition to the notation described in Chapter 2, 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 all statements S, $S(\vec{i})$ writes to location $\vec{i}$ of $A(S)$, and S is the only statement writing to A. However, each array A may still appear on the right hand side of any number of statements, where its indices can be arbitrary affine expressions of $\vec{i}$ and $\vec{n}$.

- With each array A we 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.

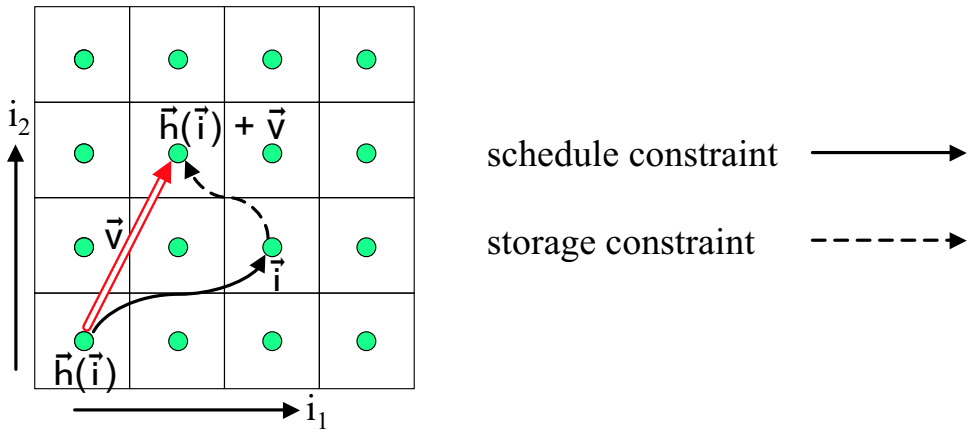

Figure 3-2: An illustration of the schedule and storage constraints. The schedule constraint requires that the producer $\vec{h}(\vec{i})$ execute before the consumer $\vec{i}$. The storage constraint requires that the consumer $\vec{i}$ execute before operation $\vec{h}(\vec{i}) + \vec{v}$, which overwrites the value of the producer $\vec{h}(\vec{i})$ according to the storage mapping imposed by the occupancy vector $\vec{v}$.

3.3 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 2.3.2 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. 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:

$$\forall \vec{n} \in \mathcal{N}, \forall \vec{i} \in \mathcal{P}_j, \theta_{R_j}(\vec{i}, \vec{n}) - \theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}), \vec{n}) - 1 \geq 0 \tag{3.1}$$

Following Feautrier [7], 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 4.1.1 for an example of the schedule constraints.

3.4 Storage Constraints

The occupancy vectors also induce some *storage constraints* (see Figure 3-2). We consider any array A. Because we assume that the data space corresponds with the

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

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

$$\forall \vec{n} \in \mathcal{N}, \forall \vec{i} \in \mathcal{Z}, \theta_T(\vec{h}(\vec{i}, \vec{n}) + \vec{v}_{A(T)}, \vec{n}) - \theta_R(\vec{i}, \vec{n}) \geq 0 \tag{3.2}$$

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 $\vec{n}$.

An example of the storage constraints can be found in Section 4.1.1.

3.5 Linearizing the Constraints

Equations (3.1) and (3.2) 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 (3.1) and storage (3.2) constraints in forms affine in the scheduling parameters Θ. This step is essential for constructing a linear program that minimizes the length of the occupancy vectors.

As discussed in Section 2.3.2, we could apply either the Farkas method or the vertex method to linearize our constraints. Though the Farkas method is usually preferred because it produces fewer constraints in practice, we choose the vertex method here in order to enable us to solve a more general problem in Section 3.6.3. Though the Farkas method and the vertex method are equivalent in power, they do not give identical constraints; we were unable to formulate our entire technique without using the vertex method. Thus, though one could use only Farkas to solve the problems of Sections 3.6.1 and 3.6.2, we present a solution with the vertex method in order to enable our technique in Section 3.6.3.

Section 4.2 contains an illustrative example of the constraint linearization.

3.5.1 Reduction using the vertices of polyhedra

Although the domain of structural parameters $\mathcal{N}$ is an input of this analysis and may be unbounded, all the polyhedra produced by the dependence analysis of programs are in fact polytopes, or bounded polyhedra. Therefore, in order to simplify the equations, we now assume that all the polyhedra we manipulate are polytopes, except when stated otherwise. Then, according to Theorem 2 (presented in Section 2.3.4), 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 (3.2).

3.5.2 Eliminating the Iteration Vector

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

$$\forall \vec{i} \in \mathcal{Z}_j, \theta_{T_j}(\vec{h}_j(\vec{i}, \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) - \theta_{R_j}(\vec{i}, \vec{n}) \geq 0 \tag{3.3}$$

which is an affine inequality in $\vec{i}$ (as $\vec{h}_j$, θ_{T_j}, and θ_{R_j} are affine functions). Thus, according to Theorem 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 $\vec{v}_{A(T_j)}$. Therefore, the number of its vertices might change depending on the domain of values of $\vec{n}$ and $\vec{v}_{A(T_j)}$. In this case we decompose the domains of $\vec{n}$ and $\vec{v}_{A(T_j)}$ into subdomains over which the number and definition of the vertices do not change [17], we solve our problem on each of these domains, and we take the "best" solution.

Thus, we evaluate (3.3) at the extreme points of $\mathcal{Z}_j$, yielding the following:

$$\begin{aligned} \forall k \in [1, n_z], \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) \geq 0 \end{aligned} \tag{3.4}$$

According to Theorem 2, Equations (3.3) and (3.4) 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 $\vec{v}_{A(T_j)}$.

3.5.3 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 extremal vertices of their domain $\mathcal{N}$. Thus, for any fixed value of Θ in $\mathcal{R}$, j in $[1, n_p]$, and k in $[1, n_z]$ we must have:

$$\begin{aligned} \forall \vec{n} \in \mathcal{N}, \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) + \vec{v}_{A(T_j)}, \vec{n}) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{n}), \vec{n}) \geq 0 \end{aligned} \tag{3.5}$$

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

$$\begin{aligned} \forall l \in [1, n_w], \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) + \vec{v}_{A(T_j)}, \vec{w}_l) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) \geq 0 \end{aligned} \tag{3.6}$$

Case of unbounded domain of parameters. It might also be the case that $\mathcal{N}$ is not a polytope but an unbounded polyhedron, perhaps corresponding to a parameter that is input from the user and can be arbitrarily large. In this case, we use the general form of Theorem 2. Let $\vec{r}_1, \ldots, \vec{r}_{n_r}$ be the rays defining the unbounded portion of $\mathcal{N}$ (a line being coded by two opposite rays). We must ensure that the linear part of Equation (3.6) is nonnegative on these rays. For example, given a single structural parameter $n_1 \in [5, \infty)$, we have the following constraint for the vertex $n_1 = 5$:

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

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

$$\begin{aligned} \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 1), 1) + \vec{v}_{A(T_j)}, 1) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 1), 1) \\ -\theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 0), 0) + \vec{v}_{A(T_j)}, 0) \\ +\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, 0), 0) \geq 0 \end{aligned} \tag{3.7}$$

Though this equation may look complicated, in practice it leads to simple formulas since all the constant parts of Equation (3.5) 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!

3.6 Finding a Solution

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

$$\begin{gathered}\forall j \in [1, n_p], \forall k \in [1, n_z], \forall l \in [1, n_w], \\ \theta_{T_j}(\vec{h}_j(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) + \vec{v}_{A(T_j)}, \vec{w}_l) \\ -\theta_{R_j}(\vec{z}_{k,j}(\vec{v}_{A(T_j)}, \vec{w}_l), \vec{w}_l) \geq 0\end{gathered} \tag{3.8}$$

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

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

$$\begin{gathered}\forall j \in [1, n_p], \forall k \in [1, n_y], \forall l \in [1, n_w], \\ \theta_{R_j}(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l) - \theta_{T_j}(\vec{h}_j(\vec{y}_{k,j}(\vec{w}_l), \vec{w}_l), \vec{w}_l) - 1 \geq 0\end{gathered} \tag{3.9}$$

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

3.6.1 Finding an Occupancy Vector Given a Schedule

At this point we have all we need to determine which occupancy vectors (if any) are valid for a given schedule Θ: we simply substitute into the simplified storage constraints (3.8) 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[1] of the components of the occupancy vector. This metric minimizes the "Manhattan" length of each occupancy vector instead of minimizing the Euclidean length. However, minimizing the Euclidean length would require a non-linear objective function.

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

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

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

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 non-integral, we have found that in most cases (including all examples in this paper), 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.

For an example of this solution procedure, refer to Section 4.1.2.

3.6.2 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. Given an occupancy vector $\vec{v}_A$ for each array A in the program, we substitute into the linearized storage constraints (3.8) to obtain

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

a set of inequalities where the only unknowns are the scheduling parameters. These inequalities, in combination with the linearized schedule constraints (3.9) completely define the space of valid affine schedules valid for the given occupancy vectors. Once again, we can search this space for solutions with any linear programming solver, selecting the "best" schedule as in [7].

See Section 4.1.3 for an example.

3.6.3 Finding a Store for a Range of Schedules

Affine Occupancy Vectors

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

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

Note that, in contrast to the Universal Occupancy Vector of [23], an AOV need not be valid for *all* schedules; rather, it only must be valid for affine ones. Almost all the instruction schedules found in practice are affine, since any FOR loop with constant increment and bounds defines a schedule that is affine in its loop indices. (This is independent of the *array references* found in practice, which are sometimes non-affine.) In this paper, we further relax the definition of an AOV to those occupancy vectors which are valid for all *one-dimensional* affine schedules.

We also observe that, if tiling is legal in the original program, then tiling is legal after transforming each array in the program under one of its AOV's. This follows from the fact that two loops are tilable if and only if they can be permuted without affecting the semantics of the program [11]. Since each permutation of the loops

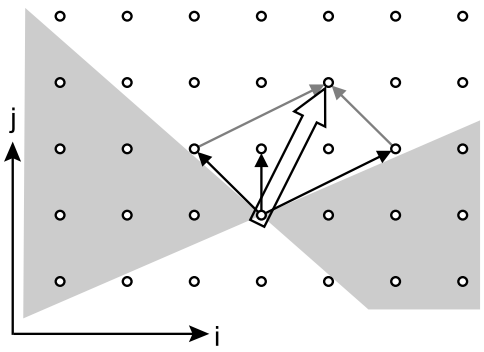

Figure 3-3: Iteration space diagram for Example 1. Here the hollow arrow denotes an Affine Occupancy Vector that is valid for all legal affine schedules. The gray region indicates the slopes at which a legal affine schedule can sweep across the iteration domain.

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

Figure 3-4: Transformed code for Example 1. The AOV is (1,2).

corresponds to a given affine schedule and the AOV is valid with respect to both schedules, the AOV transformation is also valid with respect to a tiled schedule.

Returning to our example, we find using our method that $(1, 2)$ is a valid AOV (see Figure 3-3), yielding the transformed code shown in Figure 3-4. Any one-dimensional affine schedule that respects the dependences in the original code will give the same result when executed with the transformed storage.

Finding the AOV's

Solving for the AOV's is somewhat involved (follow Section 4.1.4 for an example.) To find a set of AOV's, we need to satisfy the storage constraints (3.8) for any value of the schedule Θ within the polyhedron $\mathcal{R}$ defined by the schedule constraints. To do this, we apply the Affine Form of Farkas' Lemma (see Section 2.3.3).

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

To simplify our notation, let $STORAGE$ be the set of expressions that are constrained to be nonnegative by the linearized storage constraints (3.8). That is, $STORAGE$ contains the left hand side of each inequality in (3.8). Naively, $|STORAGE| = n_p \times n_z \times (n_w + n_r)$; however, several of these expressions might

be equivalent, thereby reducing the size of $STORAGE$ in practice.

Similarly, let $SCHEDULE$ be the set of expressions that are constrained to be nonnegative by the linearized schedule constraints (3.9). The size of $SCHEDULE$ is at most $n_p \times n_y \times (n_w + n_r)$.

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

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

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

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

The set of valid AOV's is completely and exactly determined by this set of equations and inequalities. To find the shortest AOV, we proceed as in Section 3.6.1.

3.6.4 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 AOV's: what is a good schedule that is valid for a given range of occupancy vectors? 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 Θ. Then, one can choose any storage mapping within the range and be guaranteed that it will be valid for Θ.

Let us denote a set of occupancy vectors by $\vec{V} \in \mathcal{Y}$, where $\mathcal{Y}$ is the space of all 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 occupancy vectors. Let us denote

this polyhedron by $\mathcal{Q}$, which is defined by the following q inequalities:

$$\vec{r}_j \cdot \vec{V} + \vec{s}_j \geq 0, \ j \in [1, q],$$

It is now the case that the storage constraints (3.8) must hold for all $\vec{V} \in \mathcal{Q}$. Also, we must satisfy the schedule constraints (3.9). This set of constraints is non-linear in its current form, because the storage constraints contain a product of the scheduling parameters Θ (which variables we are seeking) and the set of occupancy vectors $\vec{V}$ (which varies over $\mathcal{Q}$).

To linearize these constraints, we can apply Farkas' Lemma in the same way as in Section 3.6.3. We 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 notation as above for the storage constraints, we have:

$$STORAGE_i(\vec{x}) = \lambda_{i,0} + \sum_{j=1}^{q} (\lambda_{i,j} \cdot (\vec{r}_j \cdot \vec{x} + \vec{s}_j))$$

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

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 3.6.3 (see [7, 4] for details). We are left with a set of linear equations where the only variables are the λ's and the scheduling parameters Θ. These 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 [7].

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, it would seem desirable to find a schedule that is valid for *some* occupancy

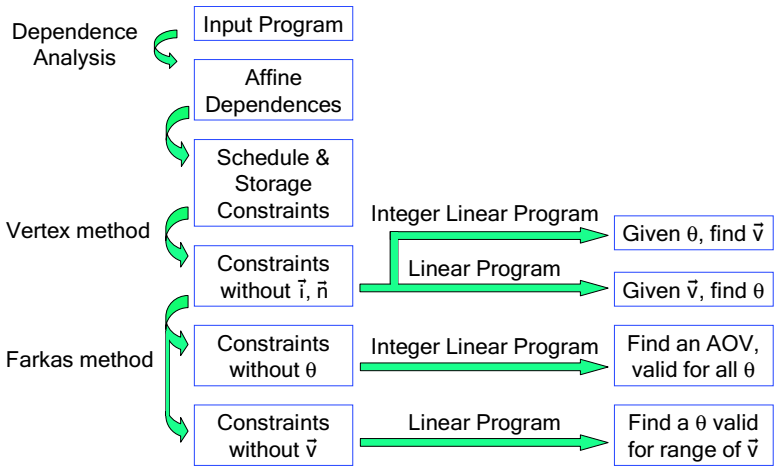

Figure 3-5: A block diagram of our solution technique.

vector of a given length, rather than *all* occupancy vectors in a range, although 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 the above solution for this problem is interesting from a theoretical standpoint, its usefulness is limited in its current form.

3.6.5 Summary

A high-level view of our method appears in Figure 3-5. 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 exact. Then, we formulate the schedule and storage constraints, which are non-linear 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 apply the vertex method and eliminate $\vec{i}$ and $\vec{n}$ from the constraints, thereby obtaining a linearized set of constraints that we can use 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 further manipulate the constraints by an application of Farkas' lemma to eliminate the scheduling parameters Θ, thereby obtaining a linear program that yields the shortest AOV–that is, an occupancy vector that is valid for

any legal affine schedule. Similarly, if we are given a range of candidate occupancy vectors, we can apply Farkas' lemma to 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.

Chapter 4

Examples

We present four examples to illustrate applications of the method described in the preceding chapter.

4.1 Example 1: Simple Stencil

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

4.1.1 Constraints

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

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

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

Thus, the schedule constraints are:

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

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

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

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

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

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

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

4.1.2 Finding an Occupancy Vector

To find the shortest occupancy vector for the schedule that executes the rows in parallel, we substitute $\theta(i,j,n,m) = j$ into the linearized schedule and storage constraints. Minimizing $|v_i + v_j|$ with respect to these constraints gives the occupancy vector of $(0,1)$ (see Figure 2-3).

4.1.3 Finding a Schedule

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

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

Inspection of these inequalities reveals that the ratio a/b has a minimum value of $-1/2$ and a maximum value that asymptotically approaches $1/2$, thus corresponding to the set of legal affine schedules depicted in Figure 3-3 (note that in the frame of the figure, however, the schedule's slope is $-a/b$.)

4.1.4 Finding an AOV

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

$$\begin{bmatrix} a * v_i + b * v_j - 2 * a - b \\ a * v_i + b * v_j - b \\ a * v_i + b * v_j + a - b \end{bmatrix} =$$
$$\begin{bmatrix} \lambda_{1,1} & \lambda_{1,2} & \lambda_{1,3} & \lambda_{1,4} \\ \lambda_{2,1} & \lambda_{2,2} & \lambda_{2,3} & \lambda_{2,4} \\ \lambda_{3,1} & \lambda_{3,2} & \lambda_{3,3} & \lambda_{3,4} \end{bmatrix} \begin{bmatrix} 1 \\ 2 * a + b - 1 \\ b - 1 \\ -a + b - 1 \end{bmatrix}$$
$$\lambda_{i,j} \geq 0, \ \forall i \in [1, 3], \forall j \in [1, 4]$$

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

To transform the data space of array A according to this AOV $\vec{v}$, we follow the

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

Figure 4-1: Original code for Example 2.

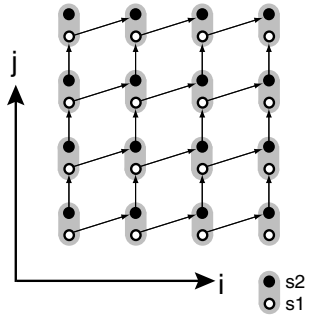

Figure 4-2: Dependence diagram for Example 2.

approach of [23] and project the original data space onto the line perpendicular to $\vec{v}$. Choosing $\vec{v}_\perp = (2, -1)$ so that $\vec{v} \cdot \vec{v}_\perp = 0$, we transform the original indices of (i, j) into $\vec{v}_\perp \cdot (i, j) = 2 * i - j$. Finally, to ensure that all data accesses are non-negative, we add m to the new index, such that the final transformation is from $A[i][j]$ to $A[2 * i - j + m]$. Thus, we have reduced storage requirements from $n * m$ to $2 * n + m$. The modified code corresponding to this mapping is shown in Figure 3-4.

4.2 Example 2: Two-Statement Stencil

We now consider an example adapted from [14] where there is a uniform dependence between statements in a loop (see Figures 4-1 and 4-2). Letting θ_1 and θ_2 denote the scheduling functions for statements 1 and 2, respectively, we have following schedule constraints:

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

and the following storage constraints:

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

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

Figure 4-3: Transformed code for Example 2. Each array has an AOV of (1,1).

We now demonstrate how to linearize the schedule constraints. We observe that the polyhedral domain of the iteration parameters (i, j) has vertices at $(1, 1)$, $(n, 1)$, $(1, m)$, and (n, m), so we evaluate the schedule constraints at these points to eliminate (i, j):

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

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

$$\theta_1(1, 1, 1, 1) - \theta_2(0, 1, 1, 1) - 1 \geq 0$$
$$\theta_1(1, 1, 1, 0) - \theta_2(0, 1, 1, 0) - \theta_1(1, 1, 0, 0) + \theta_2(0, 1, 0, 0) \geq 0$$
$$\theta_1(1, 1, 0, 1) - \theta_2(0, 1, 0, 1) - \theta_1(1, 1, 0, 0) + \theta_2(0, 1, 0, 0) \geq 0$$

Expanding the scheduling functions as $\theta_x(i, j, n, m) = a_x + b_x * i + c_x * j + d_x * n + e_x * m$,

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

Figure 4-4: Original code for Example 3, for multiple sequence alignment. Here f computes the initial gap penalty and w computes the pairwise alignment cost.

the entire set of 24 equations can be simplified to:

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

These equations constitute the linearized schedule constraints. In a similar fashion, we can linearize the storage constraints, and then apply Farkas' lemma to find the shortest AOV's of $\vec{v}_A = \vec{v}_B = (1, 1)$. The code that results after transformation by these AOV's is shown in Figure 4-3.

4.3 Example 3: Multiple Sequence Alignment

We now consider a version of the Needleman-Wunch sequence alignment algorithm [18] to determine the cost of the optimal global alignment of three strings (see Figure 4-4). The algorithm utilizes dynamic programming to determine the minimum-cost

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

Figure 4-5: Transformed code for Example 3, using the AOV of (1,1,1). The new array has dimension [imax+jmax][imax+kmax], with each reference to [i][j][k] mapped to [jmax+i-j][kmax+i-k].

alignment according to a cost function w that specifies the cost of aligning three characters, some of which might represent gaps in the alignment.

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

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

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:

$$\theta_2(i-1+v_i, j+v_j, k+v_k, x, y, z) - \theta_2(i, j, k, x, y, z) \geq 0$$
$$for\ i \in [3, x], j \in [2, y], k \in [2, z]$$

$$\theta_2(i+v_i, j-1+v_j, k+v_k, x, y, z) - \theta_2(i, j, k, x, y, z) \geq 0$$
$$for\ i \in [2, x], j \in [3, y], k \in [2, z]$$

$$\theta_2(i+v_i, j+v_j, k-1+v_k, x, y, z) - \theta_2(i, j, k, x, y, z) \geq 0$$
$$for\ i \in [2, x], j \in [2, y], k \in [3, z]$$

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

Applying our method for this example yields an AOV of $(1, 1, 1)$. The transformed code under this occupancy vector is just like the original, except that the array is of dimension [imax+jmax][imax+kmax] and element [i][j][k] is mapped to [jmax+i-j][kmax+i-k] (see Figure 4-5).

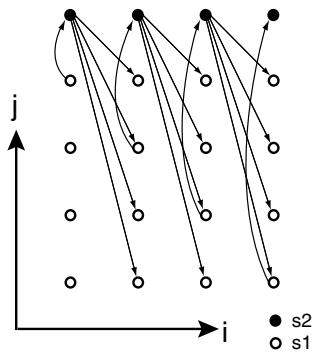

Figure 4-6: Dependence diagram for Example 4.

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

Figure 4-7: Original code for Example 4.

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

Figure 4-8: Transformed code for Example 4. The AOV's for A and B are (1,0) and 1, respectively.

4.4 Example 4: Non-Uniform Dependences

Our final example is constructed to demonstrate the application of our method to non-uniform dependences (see Figures 4-7 and 4-6). Let θ_1 and θ_2 denote the scheduling functions for statements S_1 and S_2, respectively. Then we have the following schedule constraints:

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

and the following storage constraints:

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

Applying our method to these constraints yields the AOV's $\vec{v}_A = (1, 0)$ and $v_B = 1$. The transformed code is shown in Figure 4-8.

Chapter 5

Experiments

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

For Example 2, the computation was divided into diagonal strips. Since there are no data dependences between strips, the strips can be assigned to processors without requiring any synchronization [14]. Figure 5-1 shows the speedup gained on varying numbers of processors using both the original and the transformed array. Both versions show the same trend and do not significantly improve past 16 processors, but the transformed code has an advantage by a sizable constant factor.

Example 3 was parallelized by blocking the computation, and assigning rows of blocks to each processor. As shown in Figure 5-2, the transformed code again performs substantially better than the original code. With the reduced working set of data in the transformed code, the speedup is super-linear in the number of processors due to improved caching.

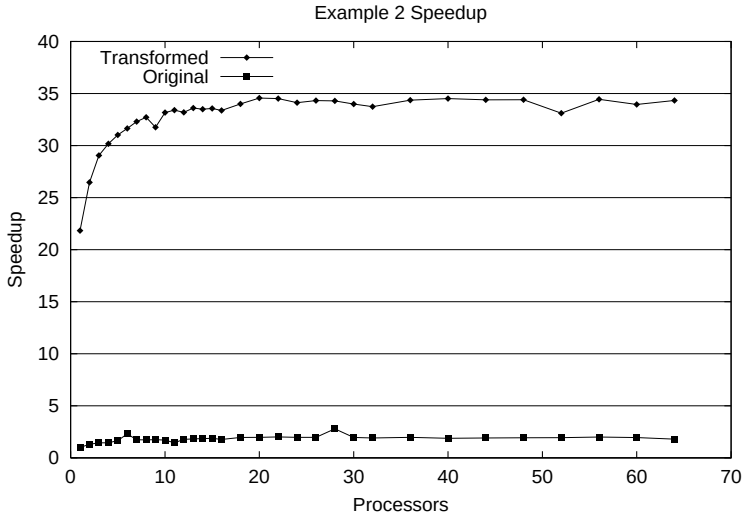

Figure 5-1: Speedup vs. number of processors for Example 2.

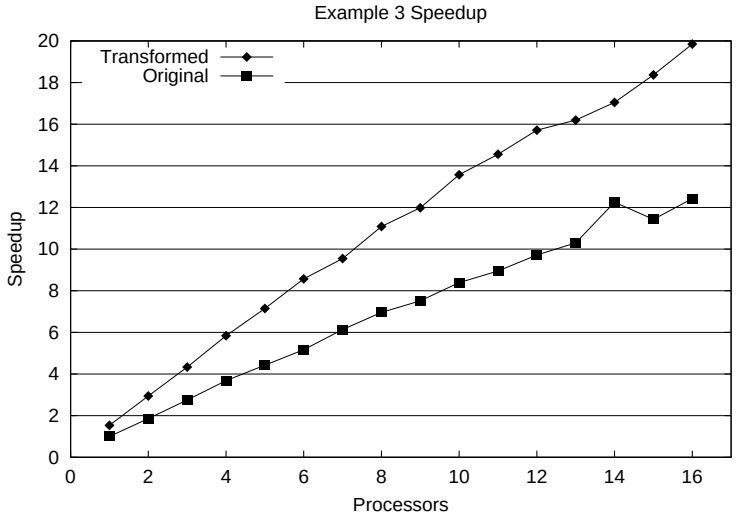

Figure 5-2: Speedup vs. number of processors for Example 3.

Chapter 6

Related Work

The work most closely related to ours is that of Strout et. al [23], which considers schedule-independent storage mappings using the Universal Occupancy Vector (UOV). While an AOV is valid only for affine schedules, a UOV is valid for any legal execution ordering. Consequently, sometimes there exist AOV's that are shorter than any UOV since the AOV must be valid for a smaller range of schedules; for example, the shortest UOV for Example 1 is $(0, 3)$ whereas our technique found an AOV of $(1, 2)$. While the analysis of [23] is limited to a stencil of dependences involving only one statement within a perfectly nested loop, our method applies to general affine dependences across statements and loop nests. Moreover, our framework goes beyond AOV's to unify the notion of occupancy vectors with known affine scheduling techniques.

The leading method for schedule-specific storage optimization in the context of the polyhedral model is that of Quilleré and Rajopadhye [21], which builds on that of Wilde and Rajopadhye [25]. 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 technique utilizes projective memory allocations that achieve the effect of several occupancy vectors applied at once; they introduce "pseudo-projective" allocation functions to deal with modulation in a general way. The method is optimal in the sense that it finds the maximum number of linearly independent projection vectors for col-

lapsing a given array. It is not clear how storage is minimized along each dimension of the final mapping.

We view our technique as being complementary to that of Quilleré and Rajopadhye. When the schedule is given, we believe that 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 related approach to storage management for parallel programs is that of Lefebvre, Feautrier, and Cohen [3, 2, 13]. Given an affine schedule, Lefebvre and Feautrier [13] 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 [3, 2] to consider storage mappings for a *set* of schedules, towards the end of capturing the tradeoff between parallelism and storage.

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

Lim et. al also address the problem of the interplay between scheduling and storage optimization [16]. Their technique encompasses affine partitioning [14], 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 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 [19]. 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.

Chapter 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 a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid for all legal schedules. Our technique is general and precise, allowing inter-statement affine dependences and solving for the shortest occupancy vector using integer linear programming.

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 very relevant in the context of array expansion, where the cost of extra array dimensions must be weighed against the scheduling freedom that they provide. Additionally, our framework could be applied to single-assignment functional languages where all storage reuse must be orchestrated by the compiler. In both of these applications, and even for compiling to uniprocessor systems, understanding the interplay between scheduling and storage is crucial for achieving good performance.

However, since finding an exact solution for the "best" occupancy vector is a very complex problem, our method relies on several assumptions to make the problem tractable. We ignore the shape of the data space and assume that the shortest occupancy vector is the best; further, we minimize the Manhattan length of the vector, since minimizing the Euclidean length is nonlinear. Also, we restrict the input domain to programs where 1) the data space matches the iteration space, 2)

only one statement writes to each array, 3) the schedule is one-dimensional and affine, and 4) there is an affine description of the dependences. It is with these qualifications that our method finds the "best" solution.

Clearly, it would 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 would not only 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 non-linear. 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 non-linear 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.

Bibliography

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

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

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

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

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

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

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

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

[9] P. Feautrier. The use of farkas lemma in memory optimization. Unpublished note, June 2001.

[10] P. Feautrier, J.-F. Collard, M. Barreteau, D. Barthou, A. Cohen, and V. Lefebvre. The Interplay of Expansion and Scheduling in PAF. Technical Report 1998/6, PRiSM, U. of Versailles, 1988.

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

[12] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

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

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

[15] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[16] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112. ACM Press, 2001.

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

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

[19] G. Pike. *Reordering and Storage Optimizations for Scientific Programs.* PhD thesis, University of California, Berkeley, 2002.

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

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

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

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

[24] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 232–242. ACM Press, 2001.

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