Combined ILP and Register Tiling: Analytical Model and Optimization Framework

Lakshminarayanan Renganarayana, U. Ramakrishna,
and Sanjay Rajopadhye

Computer Science Department
Colorado State University
{ln,ramakrsn,svr}@cs.colostate.edu

Abstract. Efficient use of multiple pipelined functional units and registers is very important for achieving high performance on modern processors. Instruction Level Parallelism (ILP) and register reuse (through register tiling) are two mechanisms for this, respectively. Program transformations that expose and exploit ILP and register reuse interact with each other in subtle ways. We study the combined problem of optimal ILP and register reuse. We consider the class of uniform dependence, fully permutable, rectangular loop nests. We develop an analytical model of the combined problem and formulate a mathematical optimization problem that chooses the parameters of the ILP-exposing transformation and register tiling so as to minimize the total execution time. We distinguish two cases: when loop permutation can and cannot expose a parallel loop. We show that the combined problem can be reduced to a single integer convex optimization problem for the former case, and to a set of integer convex optimization problems for the latter case, both of which can be solved to global optimality.

1 Introduction

It takes more than a good algorithm to achieve high performance: efficient use of the multiple pipelined functional units and registers are also important. Instruction level parallelism (ILP) allows a sequence of instructions derived from a sequential program to be parallelized for execution on multiple pipelined functional units in modern processors. Exploiting ILP and register reuse is critical for efficient use of execution resources. State-of-the-art compilers perform a variety of program optimizations to expose, enhance and exploit ILP and register reuse.

Loop nests are often the main sources for ILP and register reuse. The traditional approach uses unroll and jam [1] to expose ILP and scalar replacement to expose register reuse. However, this approach has the disadvantage of increased code size and register pressure. Further, it is hard to quantify the interactions [2] between unroll and jam, scalar replacement and software pipelining, the widely used loop scheduling technique [3,4,5].

Loop parallelizing techniques offer many transformations that can expose parallelism. Examples include, loop permutation, loop skewing [1], multi dimensional scheduling [6], etc. In addition, loop tiling [7] can be used enable register

reuse. We propose to use loop permutation and skewing to expose ILP, followed by tiling to enable register reuse. Our approach does not suffer from increased code size. However, enabling register reuse with tiling requires a register allocator for array variables as compared to the use of scalar register allocator in the scalar replacement approach.

Program transformations that expose ILP and those that enable register reuse interact with each other in subtle ways. For example, loop unrolling and loop skewing will expose ILP but might also increase the number of live values and hence the register pressure. On the other hand, register tiling will enable register reuse but might also limit the amount of ILP with the new order of execution of the tiled program. Quantifying and modeling these interactions between various program transformations is crucial for finding optimal (*w.r.t.* total program execution time) transformations. In this paper we seek to solve the combined problem of choosing the optimal parameters for the ILP exposing (loop skewing) transformation and register tiling. Our contributions are as follows.

- We give an analytical model that quantifies the interaction between the ILP exposing transformation (loop skewing) and register tiling.
- We formulate the optimal ILP and register tiling problem as a mathematical optimization problem. We present a globally optimal solution to this problem by reducing it to a convex optimization problem.
- We distinguish two cases: when loop permutation can and cannot expose a parallel loop. In the former case, we reduce the combined optimization problem to a single integer convex optimization problem. In the later case, when skewing is required to expose ILP, we show that the combined problem can be reduced to a set of integer convex optimization problems.

The solution to our combined problem will produce a loop nest in which the ILP and register reuse are exposed. The scheduling and register allocation phase is an important step in achieving good performance. This phase is beyond the scope of this paper. Our main observation is that this phase can be constructed by adapting well studied techniques like modulo scheduling [4] and register allocation for array variables.

In the next section we give an outline of our solution to the ILP and register tiling problem. In section 3, we define the program, tiling, and execution models and describe the basic building blocks of our analytical model. In section 4, we formulate the mathematical optimization problem that chooses the optimal skew and tile parameters. In section 5, we characterize the condition under which a permutation can expose a parallel loop and present an efficient algorithm to check this condition. In section 6, we characterize the space of valid skewing transformations. In section 7, we show how the optimal tiling problem can be reduced to a convex program and solved efficiently and in section 8, we present the strategy for finding the globally optimal solution to the combined ILP and register tiling problem. In section 9, we illustrate our solution method with an example. In section 10, we present related work and in section 11, we present a discussion and future work. A more detailed version of this paper can be found in the technical report [8].

2 Our approach to ILP and register tiling

Our approach is to use loop skewing to expose ILP, and register tiling to enable register reuse, and software pipelining to expose the ILP so exposed. Since we are using register tiling together with loop skewing, we require that after skewing, the resulting loop nest must admit rectangular tiling.

Software pipeliners look at the innermost loop[1] to find ILP among operations from different iterations of the loop. Hence, if we could transform the loop nest into one in which the inner most loop does not carry any dependences, i.e., all of its iterations can be executed in parallel, then the software pipeliner can find a schedule in which the performance is constrained only by the execution resources as opposed to dependencies. *When sufficient ILP exists and can be exploited, the performance is limited only by the available execution resources – or the execution bandwidth of the machine.* Such a schedule will exploit the maximum possible ILP and have maximum utilization of functional units.

Motivated by the above discussion, we seek a transformation that would transform the given fully permutable loop nest into one

- **(C1)**: for which rectangular tiling is valid for any given tile sizes $\boldsymbol{t} = (t_1, \ldots, t_n)$. This validity condition reduces to *non-negativity of all the components of all the dependences*, under the reasonable assumption that the tile size be larger than the dependence lengths, and the iteration space size be larger than the tile size [11].
- **(C2)**: in which there is at least one loop which does not carry any dependences (i.e., whose iterations are all parallel). We can always permute this loop to the inner most position, as full permutability (of the transformed loop nest) is necessary for condition (C1) to hold.

There are many classes of transformations that can produce a loop nest that would satisfy the above two conditions. Loop skewing is one such class and we have chosen it for the following reasons. First, for uniform dependence loops, we can always find a skewing transformation that will produce a loop that satisfies (C1) and (C2). Second, loop skewing is conceptually simple and easy to construct, and this allows us to develop an efficient algorithm for finding the optimal skew transformation parameters.

Our solution methodology is as follows. Using the performance model described in Section3, we formulate an optimization problem whose solution yields the skew factor and tile sizes that minimize the overall execution time. We check whether permutation can expose any parallel loop. If so, we permute, expose the parallelism, and then tile for registers. In this case, the combined problem reduces to the problem of finding the optimal tile sizes, which can be reduced to a single integer convex optimization problem. When loop permutation cannot expose a parallel loop, loop skewing is required to expose the ILP. In this case, we need to find the optimal skewing and tile sizes. We find these by solving a set of integer convex optimization problems.

[1] The two exceptions are the works of Rong et al. [9] and Ramanujam [10]. See the related work section for details.

3 An analytical model

In this section we develop an analytical model that quantifies the interaction between loop skewing and register tiling transformations. A similar model was previously used in the context of tiling for memory hierarchy [12].

3.1 Program and tiling model

The programs we consider belongs to the class of fully permutable rectangular loop nests with uniform dependence bodies. Note that this class of programs admit rectangular tiling and are also the class for which software pipelining is usually applied. We consider an n-dimensional loop nest with constant upper and lower bounds. The loop body contains statements with uniform dependences. Let $\mathcal{L} = [L_1, \ldots, L_n]$ be the given n-dimensional loop nest, where each L_i denotes a loop at depth i. Any n-D vector formed by the loop counters of $\mathcal{L}$ is called an iteration vector. Let $D = [d_1, \ldots, d_m]$ be a matrix whose columns are the (n-D) dependence vectors.

To expose ILP we use skewing and permutation. A skewing (transformation) matrix has the form of an upper triangular matrix with all the diagonal entries equal to 1. The non-diagonal entries are determined by the skewing factors. We denote the skewing matrix that we seek by S. Skewing a loop L_i with respect to a loop L_j, by an appropriate factor f, makes the loop L_i carry all the dependences that were originally carried by loop L_j. A permutation transformation that permutes the i^{th} loop with the j^{th} loop can be represented by an identity matrix (of appropriate size) in which the i^{th} and j^{th} rows are interchanged.

We consider *rectangular* (or *orthogonal*) loop tiling: tiling the loop nest with *hyper-rectangles whose boundaries are orthogonal to the canonic axes.* We assume that rectangular loop tiling is valid for the given loop nest [7]. Note that the tiled loops are fully permutable. The *tile graph* is the graph where each node represents a tile and each arc represents a dependency between tiles. In our case, each node of the tile graph represents a hyper-rectangle in the iteration space of size $t_1 \times t_2 \times \cdots \times t_n$. Note that though our iteration space is rectangular, after skewing, we will have hyper-parallelepiped shaped iteration space, and when we tile this with rectangular tiles, we will have some full rectangular tiles and some partial non-rectangular tiles.

It is well known that [13] if the t_i's are large compared to the elements of the dependency vectors, then the dependencies between the tiles are *unit vectors* (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). In general, the feasible value of each t_i is bounded from below by some constant. For the sake of notational simplicity, in this paper we assume that this is 1.

3.2 Architecture and Execution model

We use an atomic tile execution model. However, the parallelism available inside the tile is exploited with software pipelining. We first present the architectural

parameters used in the execution model and then introduce the functions that model various aspects of the execution time of the transformed loop nest.

Although we do not provide experimental validation of our execution time model in this paper, similar models of execution time have been used by Sarkar [14] (in the IBM XL Fortran compiler) and also by Wolf et al. [15], and they have been thoroughly validated.

We seek an abstraction of the architecture (processor and memory features) that is suitable for use in a cost model for tiling loop programs in our program class. Our model uses the following parameters:

- α – *cost of an iteration*: this is the cost of executing an instance of the loop body (in cycles per iteration). In our case, since the innermost loop is completely parallel, a modulo scheduler can always achieve the resource minimum initiation interval (ResMII) [4], and hence α is equal to ResMII.
- β – the cost (in cycles) for transferring a word from lowest level cache to the registers.
- η – *loop increment and test cost*: this is the cost for incrementing a loop variable and checking its bounds.
- NR – *number of registers available:* depending on the loop body, NR could be either the number of integer or floating point registers.

3.3 Fundamental measures

Computation volume. *The computation volume,* $\mathsf{TV}(\boldsymbol{t})$*, of a tile is the amount of computation done in a tile.* The computation volume of a tile $\boldsymbol{t} = (t_1, \ldots, t_n)$, is the number of integer points in the n-dimensional hyper-rectangle: $\mathsf{TV}(\boldsymbol{t}) = \prod_{i=1}^{n} t_i$. The tile volume, $\mathsf{TV}(t)$, represents the volume of full tiles. We approximate the volume of partial tiles with that of the full tiles, and hence use $\mathsf{TV}(\boldsymbol{t})$ as the volume for all the tiles.

Load store volume. *The load store volume,* $\mathsf{LS}(\boldsymbol{t}, D)$*, of a tile is the total amount of data that is loaded and stored when the tile is executed.* This quantity is also known as the tile foot-print. The dependences and data reuse patterns determine the load store volume. Our program model restricts dependences to be uniform (constant distance). A tile is compute bound if the amount of data accessed (input/output) during the computation of the tile is at least one dimension less than the computation; otherwise the tile is I/O-bound. It is easy to see that with uniform dependences, the load store volume of I/O-bound tiles is proportional to the tile volume $\mathsf{TV}(\boldsymbol{t})$. The interesting case, where tiling is really useful, is when the tile is compute bound.

For an n-dimensional compute bound tile, the input and output are $O(x^{n-1})$, where, $x = \max_{i=1}^{n} t_i$, where t_i is the tile size along dimension i. We consider the case in which the input and output are of $O(x^{n-1})$, other cases when the input or output is smaller than $O(x^{n-1})$ can be handled easily. Since our tile graph has dependence vectors that correspond to unit vectors, the $O(x^{n-1})$ input/output of a tile directly corresponds to the $(n-1)$ dimensional facets of the tile, and a constant multiple of every facet contributes to the load store volume of a

tile. The constant is determined by the dependence distances. There are n pairs of facets, and in rectangular tiling, each of these is potentially involved in a communication. The volume of the i^{th} facet, Δ_i, is given by $\prod_{j=1,j\neq i}^{n} t_j$. Now, the load store volume is $\mathsf{LS}(\boldsymbol{t}, D) = \sum_{i=1}^{n} a_i \Delta_i$, where a_i is a constant that denotes distance along the i^{th} facet that is involved in the communication and is determined by the longest i^{th} dimension component of any dependence vector in the dependence matrix D. Based on the schedule, some facets need not be stored and loaded again. There is at most one such facet, say f, and sharing of f can be captured by excluding it from the load store, i.e., $\mathsf{LS}(\boldsymbol{t}, D) = \sum_{i=1,i\neq f}^{n} a_i \Delta_i$. We can take care of multiple dependences to the same variable by considering the bounding box of the dependences to each variable and using the diagonal of this bounding box as the columns of D.

Number of tiles. *The number of tiles,* $\mathsf{NT}(\boldsymbol{t}, \boldsymbol{N}) = \frac{N_1 \times \cdots \times N_n}{t_1 \times \cdots \times t_n}$, *counts the total number of tiles after a rectangular tiling with tiles of sizes* $\boldsymbol{t} = (t_1, \ldots, t_n)$, *of the rectangular iteration space of size* $\boldsymbol{N} = (N_1, \ldots, N_n)$. After skewing, the iteration space may no longer be rectangular and counting the number of tiles in this case is complicated. We use the quantity (iteration space volume)/(tile volume), which is a lower bound on the actual number of tiles, as an approximation. Since we start with a rectangular iteration space and skewing is a volume preserving unimodular transformation, the quantity (iteration space volume)/(tile volume) is the same as[2] $\mathsf{NT}(\boldsymbol{t}, \boldsymbol{N})$.

Loop overhead. *The loop overhead of a loop is used to account for the cost of loop termination test and loop variable increment.* It is proportional to the number of times the loop body is executed. An n-dimensional rectangular loop nest after one level of tiling will have $2n$ loops. We call the outer n loops *inter-tile loops* and the inner n loops *intra-tile loops.* The i^{th} inter-tile loop is executed precisely $\frac{N_i}{t_i}$ times for *each* instance of the surrounding loop indices. The total overhead of the n inter-tile loops , $\mathsf{LoInterTile}(\boldsymbol{t}, \boldsymbol{N})$, is $\sum_{i=1}^{n} x_i$, where $x_i = \frac{N_1 \times \ldots \times N_i}{t_1 \times \ldots \times t_i}$. The i^{th} intra-tile loop is executed t_i times. The overhead of the set of n intra-tile loops, $\mathsf{LoIntraTile}(\boldsymbol{t}, \boldsymbol{N})$, is $\sum_{i=n+1}^{2n} y_i$, where $y_i = (t_1 \times \ldots \times t_i) \times \mathsf{NT}(\boldsymbol{t}, \boldsymbol{N})$, where $\mathsf{NT}(\boldsymbol{t}, \boldsymbol{N})$ is the total number of tiles and also equal to the number of times the n inter-tile loops surrounding the intra-tile loops will be executed. The total (intra plus inter tile) loop overhead, $\mathsf{LO}(\boldsymbol{t}, \boldsymbol{N}) = \mathsf{LoIntraTile}(\boldsymbol{t}, \boldsymbol{N}) + \mathsf{LoInterTile}(\boldsymbol{t}, \boldsymbol{N})$. Since after skewing the iteration space may not be rectangular, the rectangular tiling might leave some partial and full tiles. Treating partial tiles as full tiles and using the approximation for number tiles, developed above, we can approximate by $\mathsf{LO}(\boldsymbol{t}, \boldsymbol{N})$, the loop overhead of a skewed rectangular loop nest tiled with rectangular tiles.

When we use skewing to expose ILP, the shape of the iteration space, as well as the dependences change. The iteration space becomes a parallelepiped and the transformed dependences are given by SD, where S and D are the skewing and dependence matrices, respectively.

[2] Given that we are tiling for registers, the tile sizes are going to be very small and with small tile sizes, this approximation is better.

4 Optimization problem formulation

We now formulate an optimization problem that clearly captures and quantifies the interaction between the skewing and the register tiling transformations. The objective function is the total execution time and the unknowns are the tile sizes ($\boldsymbol{t}$) and the skewing matrix (S).

$$\begin{aligned} \text{minimize } & \eta \mathsf{LO}(\boldsymbol{t}, \boldsymbol{N}) + \mathsf{NT}(\boldsymbol{t}, \boldsymbol{N}) \times \max\left(\alpha \times TV(\boldsymbol{t}),\ \beta \times \mathsf{LS}(\boldsymbol{t}, \mathsf{bbox}(SD))\right) \\ \text{s.t.} \quad & \mathsf{LS}(\boldsymbol{t}, \mathsf{bbox}(SD)) \leq \mathsf{NR} \\ & \boldsymbol{N} \geq \boldsymbol{t} \geq 1, SD \geq 0, \boldsymbol{t} \in \mathbb{Z}^n, S \in \mathbb{Z}^{n \times n} \end{aligned} \tag{1}$$

where, $\boldsymbol{t}$ and S are the variables representing tile sizes and skew matrix, respectively, $\mathsf{NT}(\boldsymbol{t}, \boldsymbol{N})$ is the number of tiles, $\mathsf{TV}(\boldsymbol{t})$ is the tile volume, D is the dependence matrix, $\mathsf{LS}(t, \mathsf{bbox}(SD))$ is the load store volume, $\mathsf{LO}(\boldsymbol{t}, \boldsymbol{N})$ is the loop overhead, NR is the number of registers available, α, β and η are respectively the cost of an iteration, load store cost, and loop bounds check cost. All vector inequalities in the constraints are component-wise. The first constraint makes sure that the register foot print $\mathsf{LS}(t\ ,\ \mathsf{bbox}(SD))$ fits in the number of available registers, NR, and the second constraint $\boldsymbol{t} \geq 1$ makes sure that the tile sizes are positive and the third constraint $SD \geq 0$ ensures that the skewed loop nest is fully permutable and hence admits a rectangular tiling.

Once we choose a skew transformation S, substituting it in the combined problem gives an optimization problem with $\boldsymbol{t}$ as the only variable. Let $\widehat{D} = \mathsf{bbox}(SD)$. Then the resulting optimization problem is shown below (2). We call (2) the *optimal tiling problem (for a fixed skew).*

$$\begin{aligned} \text{minimize } & \eta\ \mathsf{LO}(\boldsymbol{t}, \boldsymbol{N}) + \mathsf{NT}(\boldsymbol{t}, \boldsymbol{N}) \times \max\left(\alpha \mathsf{TV}(\boldsymbol{t}),\ \beta\ \mathsf{LS}(\boldsymbol{t},\ \widehat{D})\right) \\ \text{s.t.} \quad & \mathsf{LS}(\boldsymbol{t}\ ,\ \widehat{D}) \leq \mathsf{NR}, \boldsymbol{N} \geq \boldsymbol{t} \geq 1, \boldsymbol{t} \in \mathbb{Z}^n \end{aligned}$$

Note that, though $\widehat{D}$ is shown as a parameter to the $\mathsf{LS}(\boldsymbol{t}, \widehat{D})$ function, it is here a given constant vector, and not a variable of the optimization problem.

5 Can permutation can expose a parallel loop

We will first introduce some notations (used only in this section) which will make the exposition clear and concise. For any vector x, $x(j)$ represents its j-th component. The *level* of a vector $\mathsf{level}(x)$ is j if $\forall i < j :\ x(i) = 0$ and $x(j) \neq 0$, i.e., $x(j)$ is the first non-zero component of x. A *zero-lead column* is a column vector of the form $(0, 0, \ldots, 0, c)^T$ for some $c \neq 0$. The j-th *unit vector* e_j is a vector with $e_j(j) = 1$ and $e_j(i) = 0, \forall i \neq j$. A *scaled unit vector*, $\mathsf{suv}(c, j)$ is a vector x of the form $\forall i \neq j :\ x(i) = 0$ and $x(j) = c$ for some non-zero constant c. In other words, $\mathsf{scv}(c, j)$ is an unit vector along j scaled by a non-zero factor c. The dimension of a scaled unit vector is often obvious from the context. An example (of dimension 4) is $\mathsf{suv}(2, 3) = (0, 0, 2, 0)$. Note that $\mathsf{level}(\ \mathsf{suv}(c, j)\ ) = j$. $\mathsf{diag}(c_1, c_2, \ldots, c_n)$ constructs a diagonal matrix with $c_1, \ldots, c_n$ as the diagonal entries. A loop is called *parallel* if it does not carry any dependences.

5.1 Existence of a loop with no carried dependences

We seek to characterize a condition under which there exists no permutation of $\mathcal{L}$ with at least one parallel loop. In other words, in every permutation of $\mathcal{L}$, all the loops carry dependences. We seek a characterization based on the dependences. Let us form a dependence (distance vector) matrix $D = [d_1\ d_2\ \ldots\ d_m]$ whose columns are the m dependences, $d_1, d_2, \ldots, d_m$ present in $\mathcal{L}$'s body. The effect of loop permutation on the dependences is completely captured by permuting the rows of D. In any permutation of $\mathcal{L}$, if there is a dependence d with $\mathsf{level}(d) = j$ then loop l_j, of the permuted loop nest, carries d.

Consider the two dependence matrices: $D_1 = \begin{matrix} d_1\ d_2\ d_3 \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 2 \\ 1 & 1 & 0 \end{pmatrix} \end{matrix}$ $D_2 = \begin{matrix} d_1\ d_2\ d_3\ d_4 \\ \begin{pmatrix} 1 & 0 & 0 & 3 \\ 1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{matrix}$.

In the matrix D_1, the dependence vectors d_2 and d_3 are scaled unit vectors: $d_2 = \mathsf{suv}(1,3)$ and $d_3 = \mathsf{suv}(2,2)$. Now, in this permutation, the dependences d_1, d_2 and d_3 have levels $1, 3$ and 2 respectively and are carried by the loops L_1, L_3 and L_2 respectively. However, we can see that by exchanging rows 1 and 3 of D_1 we can get an innermost loop (row 3 of permuted D_1) with no carried dependences. Now consider matrix D_2 : there exists no permutation of rows of D_2 which can create a parallel loop. What is the structure of the matrix D_2 that induces this property? We seek to characterize this structure in the following discussion leading to Theorem 1.

In any given permutation of the loops, all the n loops will carry dependences if and only if there are (at least) n dependence vectors with levels $1, 2, \ldots, n$. If we have dependence vectors of all levels $(1, 2, \ldots, n)$ in every permutation of the loops in $\mathcal{L}$, then we can say that there is no permutation that will expose a parallel loop.

Theorem 1: Every permutation of the rows of D will contain n columns with levels $1, 2, \ldots, n$ *if and only if* D contains a $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix, say $\mathsf{diag}(c_1, c_2, \ldots, c_n)$, where $c_1, \ldots, c_n$ are the scale factors of the n scaled unit vectors.

Proof: ($\Longrightarrow$) Assume that every permutation of the rows of D will contain n columns with levels $1, 2, \ldots, n$. Let $x_1, \ldots, x_n$ be these n columns with levels $1, 2, \ldots, n$ respectively. Given that we have exactly n vectors each having a different level, they all have to be linearly independent. If we show that these n columns are scaled unit vectors, then we can always permute these columns to form a $n \times n$ diagonal sub matrix of D. To show that $x_1, \ldots x_n$ are scaled unit vectors we will use proof by contradiction. Let us assume that they are (all) not scaled unit vectors. Note that the vector x_n with level n has to be a scaled unit vector. Let the $n - 1$ columns each have one more non-zero entry below their first non-zero entry. Without loss of generality we can assume that this entry is the next immediate entry. Then the matrix looks the matrix M given below.

Algorithm 1 Algorithm to check whether the input loop nest has any parallel loop.

1. **Input**: Dependence matrix D. **Output**: boolean value indicating whether the input loop nest has any parallel loop or not.
2. Pick all the columns of D which are scaled unit vectors. This can be done in $O(nm)$, where, n is the number of rows of D and m, the number of columns. There can be at most m such columns.
3. As we pick the columns in the previous step we can note their levels. Check whether there are n columns each of which is a scaled unit vector for a distinct j, i.e., $\mathsf{suv}(c_j, j)$ for $j = 1 \ldots n$. This can also be done in time $O(nm)$. If there are such n columns return a **true**; return a **false** otherwise.

$$M = \begin{pmatrix} x_{1,1} & 0 & \cdots & 0 & 0 \\ x_{2,1} & x_{2,2} & \cdots & 0 & 0 \\ \vdots & x_{2,2} & \cdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & x_{n-1,n-1} & 0 \\ \cdots & & & x_{n,n-1} & x_{n,n} \end{pmatrix} \quad \Longrightarrow \quad M' = \begin{pmatrix} x_{1,1} & 0 & \cdots & 0 & 0 \\ x_{2,1} & x_{2,2} & \cdots & 0 & 0 \\ \vdots & x_{2,2} & \cdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & x_{n,n-1} & x_{n,n} \\ \cdots & & & x_{n,n-1} & 0 \end{pmatrix}$$

Now we can interchange the last two rows of M to get M' in which there is no dependence of level n and hence loop l_n does not carry any dependence. But this is a contradiction to our assumption that every permutation of the rows of D contains n columns with all the levels. Hence the proof. ⊡

Proof: ($\Longleftarrow$) Now we assume that D contains a $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix say $\mathsf{diag}(c_1, c_2, \ldots, c_n)$. Let C be this $n \times n$ sub matrix of D whose columns can be permuted to form $\mathsf{diag}(c_1, \ldots, c_n)$. We need to show that every permutation of D will contain n column with levels $1, 2, \ldots, n$. It is obvious that after any set of row permutations of a diagonal matrix there exists a set of column permutations that will bring it back to diagonal matrix form. Hence, after any set of permutations of C we can column permute C to make it a diagonal matrix. This diagonal matrix form makes it obvious that the n columns have levels $1, \ldots, n$ respectively. Hence the proof. ⊡

Theorem 1 gives us an efficient way to check whether there exists at least one loop no carried dependences – we only need to check whether the dependence matrix D contains $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix $\mathsf{diag}(c_1, \ldots, c_n)$. This can be done in time linear in the size of the dependence matrix D. The outline of the algorithm is given in Algorithm 1.

6 Space of valid skewing transformations

When loop permutations alone cannot expose a parallel loop, we need to skew the loop nest. We make two observations regarding the skew matrix S that we seek in the combined optimization problem (1). These observations narrow down the search space of S.

- **Only positive skews produce loops that admit rectangular tiling.** We have two constraints: $D \geq 0$ (since our input loop nest admits rectangular tiling) and $SD \geq 0$ (since we require the skewed loop nest to admit rectangular tiling). From Theorem 1, we know that, if the input loop nest does not have any parallel loop, then the dependence matrix D has a $n \times n$ sub matrix whose columns are scaled unit vectors and which can be permuted to form a diagonal matrix, say $M = \mathsf{diag}(c_1, \cdots, c_n)$. Without loss of generality we can assume that that these n columns $c_1, c_2, \ldots, c_n$ have levels $1, 2, \ldots, n$ respectively. At least two of these columns should be made to have the same levels, only then we will have a loop with no carried dependences. Let us view the matrix D as a partitioned as $[M\ N]$, where $M = \mathsf{diag}(c_1, \ldots, c_n)$ is the $n \times n$ diagonal sub matrix and N is the sub matrix that contains rest of the columns of D. We claim that negative skew factors will lead to an invalid transformation by creating negative entries in the sub-matrix M. To see why, let us see what happens when we skew loop L_i with respect to a loop L_j with a negative skew factor $-f$ (cf. Section 3.1 for notation). Such a skew would add to the i-th row of M, the j-th row multiplied by $(-f)$. The new i-th row would have $-f \times c_j$ in its j-th entry. This negative entry is not permitted since we require that all the entries of the transformed matrix (SD) be non-negative. Hence, only positive skew factors are valid, since a zero skew factor is just an identity transformation.
- **Skewing any one loop with respect to just one other loop is sufficient and optimal.** We seek to transform the loop nest so that in the transformed loop nest there is one loop that carries no dependences, i.e., parallel. Given that the input loop nest is fully permutable, after skewing, we can permute this parallel loop to the inner most position to get our desired loop nest. To make any one loop, say L_i, parallel, it is sufficient to skew some other loop, say L_j, with respect to L_i. Also, given that (positive) skewing increases the length of the (positive) dependences, skewing with respect to more than one loop will always produce longer (when compared to skewing w.r.t. to just one loop) dependences. And, the longer the dependences, the larger the bounding box and hence, the greater the load store volume, $\mathsf{LS}(\boldsymbol{t}, \mathsf{bbox}(SD))$. So, skewing with respect to just one other loop is also optimal. By a similar argument, skewing by a factor larger than 1 to parallelize the loop only increases the load store cost and is sub-optimal.

Based on these two observations, we seek to find positive skews of one loop with respect to just one other loop. The number of choices for such skews is $d \times (d-1)$ where, d is the depth of the loop nest. This gives a list of $d(d-1)$ potentially optimal skews. For example, for a loop nest with depth 2 or 3 we will have 2 or 6 choices of skews, respectively.

7 Solving the optimal tiling problem

The optimal tiling problem (2) seeks to choose tile sizes that minimize some criteria and satisfy some constraints. The key insight is that *the variables of this*

optimization problem, tile sizes, are always positive. Based on this insight we can directly cast it as an *Integer Geometric Program* (IGP) [16]. Due to space constraints, we do not give the translation of the optimal tiling problem into an IGP. The techniques used to cast the optimal tiling problem as an IGP can be found in the technical report [8].

Geometric programs can be transformed into convex optimization problems using a variable substitution [17] and solved efficiently using polynomial time interior point methods [18]. Integer solutions can be found by using a branch-and-bound algorithm. We use YALMIP [19] – a tool that provides an high level symbolic interface in MATLAB to define and solve IGPs. The number of (tile) variables of our IGPs are related to number of dimensions tiled and hence are often small. In our experience with solving IGPs related to tiling, the integer solutions were found in few (less than ten) iterations of the branch-and-bound algorithm. The (wall clock) running time of this algorithm was just a few seconds, even with the overhead of using the symbolic MATLAB interface.

8 Solving the combined ILP and register tiling problem

Recall that, according to our solution strategy, we need skewing only when the input loop nest does not contain any parallel loop that can be exposed by permutation. Hence, first we check (using Algorithm 1 discussed in section 5) whether the input loop nest has any parallel loop that can be exposed by permutation. If it does, then just permuting the loop to the inner most position will achieve our goal. This permutation is always valid, since our input loop nest is fully permutable (since rectangular tiling is valid for it). In this case, we just permute the loop and do not skew (i.e., the skew matrix S becomes the identity matrix). Then the combined problem (1) reduces to the optimization problem for finding the optimal tile sizes (for the permuted loop nest), i.e., the optimal tiling problem (c.f. problem (2)) with $S = I$(the identity matrix) and hence $\widehat{D} = \mathsf{bbox}(D)$. This problem can now be directly solved as discussed in section 7. Note that when permutation alone is sufficient, it is globally optimal too, because any skewing will only increase the load store cost and hence the execution time.

When permutation cannot expose a parallel loop, we need skewing to expose ILP. In this case, as shown in Section 6, we have $d(d-1)$ choices for the skewing matrix (where d is the depth of the loop nest). We construct $d(d-1)$ optimal tiling problems (with fixed skewing matrices), one for each choice of the skewing matrix. The optimal skew and tile sizes are obtained by solving these $d(d-1)$ optimal tiling problems (2) and picking the one that has the smallest objective function value (i.e., the minimum execution time).

9 A complete example

```
1  for ( i1 = 1; i1 ≤ N1 ; i1++)
2      for ( i2 = 1; i2 ≤ N2; i2++)
3          A[i2] = A[i2-1] + A[i2];
```

Consider the above loop nest and its dependence matrix $D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. As inidicated by Theorem 1, there exists no permutation of the loops that can expose the parallelism to a software pipeliner. However, the loop has lots of parallelism that can be exposed to a software pipeliner by skewing. We have $d(d-1) = 2$ choices for skewing the loops, viz., skewing `i1` w.r.t to `i2` or vice-versa. But, due the symmetry of D, both skews will have the same effect on the bounding box. Let us consider skewing loop `i2` with respect to `i1`, and then permuting them to make the `i1` loop the innermost. Now, all the dependences are carried by outer loop (`i2`) and the inner loop (`i1`) is completely parallel. A software pipeliner can exploit this parallelism to construct a schedule which is constrained only the available execution resources (and not by the dependence constraints). We then tile this skewed-permuted loop nest to enable register reuse.

To determine the optimal tile sizes, we instantiate the combined optimization problem (1) with the optimal skew (and permute) matrix $S = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, as follows. Now, $\widehat{D} = \mathsf{bbox}(SD) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Instantiating the optimal tiling problem we get

$$\begin{array}{rl} \text{minimize} & \frac{N_1 \times N_2}{t_1 \times t_2} \times \max\left(\alpha \times t_1 \times t_2,\ \beta \times (t_1 + t_2)\right) \ + \\ & \eta\ \left(N_1 \times N_2 + \frac{N_1 \times N_2}{t_2} + \frac{N_1 \times N_2}{t_1 \times t_2} + \frac{N_1}{t_1}\right) \\ \text{s.t.} & t_1 + t_2 \leq \mathsf{NR}, \boldsymbol{t} \geq 1, \boldsymbol{t} \in \mathbb{Z} \end{array} \tag{2}$$

where, α is the cost per iteration and is equal to the II (initiation interval), β is the cost of moving a data item from the lowest level cache to the register and η is the cost of a loop bound check. NR is the number of (floating point) registers in the architecture.

10 Related work

Unroll and jam. Sarkar [14] addresses the same problem as ours and uses unroll and jam followed by scalar replacement [20] for exposing ILP and register reuse. He formulates the problem as a discrete optimization problem with unroll factors as variables, and proposes an exhaustive search with heuristics to solve it. Our formulation seeks both the skew matrix and the tile sizes, and is solved to global optimality via convex programming. The class of programs considered by Sarkar, loops with affine dependences, is larger than what is considered by ours, loop nests with uniform dependences. However for uniform dependence loop nests, by setting the skew matrix to identity, viewing the tile sizes as unroll factors, and adding the code size constraint, our method can be directly used to solve the problem addressed by Sarkar. In this sense, for this class of loop nests, the problem of solving for optimal unroll factors is a special case of our problem.

Carr and Kennedy [21] propose an algorithm to determine the unroll factors that balance the floating-point and memory access operations. This objective function is different from ours, as well as Sarkar's, viz., minimizing the execution time.

Hierarchical tiling. The work of Carter et al. [22], followed up by Mitchell et al. [23], uses tiling to expose the register reuse as well as ILP. They propose hierarchical tiling as a *hand tuning* technique to better exploit pipelined functional units and registers. Our work is similar to this work in spirit, however, we have proposed a completely *automatic method* to determine the tile sizes and skew factors.

Code generation for register tiling. Jiminez et al. [24] propose a code generation strategy for non-rectangular loop nests tiled for registers. Their strategy uses index set splitting to strip off the partial boundary tiles and the full tiles are completely unrolled. Hence, they assume that unroll and jam followed by scalar promotion is used for exposing ILP and register reuse. Sarkar [14] also proposes a code generation algorithm which takes the unroll factors as input and produces an unrolled loop nest.

Software pipelining of loop nests. Traditionally software pipeliners have only looked at innermost loop nests. Ramanujam [10] proposed a technique where an integer linear programming formulation is used to find a (software) pipelined schedule that exploits the parallelism available in the whole loop nest. However, he did not consider resource constraints. Rong et al. [25] have recently proposed a technique called *single dimension software pipelining for multi-dimensional loop*s. Their technique computes the initiation interval and (cache) locality of every loop in the given loop nest and picks the best. They do not consider any ILP exposing transformations like permutation or skewing, and hence, are limited in how ILP can be exploited. On the other hand, our approach, by the virtue of looking at skewing and permutation, will always be able to expose the available ILP. Rong et al. also propose a method for code generation [26] and recently have addressed the register allocation issue [9]. A similar problem in the context of ILP and caches has been addressed by Wolf et al. [15].

11 Discussion and future work

We have formulated the combined problem of choosing an ILP-exposing (skewing) transformation and register tiling. We have proposed an efficient way to check whether permutation can expose any parallel loops. We have distinguished two cases: when loop permutation can expose a parallel loop, and when it cannot. For the former case, we have reduced the combined problem to a single convex optimization problem and for the latter case we have reduced the combined problem to a typically small set of convex optimization problems. All these convex optimization problems can be solved efficiently using currently available tools (e.g., YALMIP [19]).

The formulation of the combined problem exposes the fact that the skewing transformation affects the dependences and which in turn affects the overall execution time of transformed loop nest. We see this formulation, and its analysis, as a first step in understanding the structure of this important complex problem. To the best of our knowledge, this is the first formulation and globally optimal solution of this combined problem.

Future work. We are currently working on adapting modulo scheduling techniques [4,5] to schedule the transformed loop nest. Note that the modulo scheduler is guaranteed to find the inner most loop nest parallel. Hence, we do not need any dependence analysis to determine the achievable initiation interval. We are also investigating array register allocation techniques to map all the array values accessed in a tile to registers. Note that from the constraints of the optimal tiling problem, we are guaranteed to have enough registers.

As a future work, we plan to extend the program class. One direction is to extend the work to include iteration spaces with parallelepiped shapes. Another direction is to permit non-uniform (affine) dependences in the loop body.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence Based Approach. Morgan Kaufman, San Francisco (2002)
2. Carr, S., Sweany, P.: An experimental evaluation of scalar replacement on scientific benchmarks. Software Practice and Experience **33**(15) (2003) 1419–1445
3. Lam, M.: Software pipelining: an effective scheduling technique for vliw machines. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM Press (1988) 318–328
4. Rau, B.R.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture, New York, NY, USA, ACM Press (1994) 63–74
5. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. **27**(3) (1995) 367–432
6. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser Boston (2000)
7. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers (2000)
8. Renganarayana, L., Ramakrishna, U., Rajopadhye, S.: Combined ILP and register tiling: Analytical model and optimization framework. Technical Report CS-05-102, Department of Computer Science, Colorado State University (2005) Available from `http://www.cs.colostate.edu/~ln/publications/TR-CS-05-102.pdf`.
9. Rong, H., Douillet, A., Gao, G.R.: Register allocation for software pipelined multidimensional loops. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM Press (2005) 154–167
10. Ramanujam, J.: Optimal software pipelining of nested loops. In: IPPS. (1994) 335–342
11. Xue, J.: On tiling as a loop transformation. Parallel Processing Letters **7**(4) (1997) 409–424
12. Renganarayana, L., Rajopadhye, S.: A geometric programming framework for optimal multi-level tiling. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2004) 18
13. Andonov, R., Balev, S., Rajopadhye, S.V., Yanev, N.: Optimal semi-oblique tiling. IEEE Trans. Parallel Distrib. Syst. **14**(9) (2003) 944–960
14. Sarkar, V.: Optimized unrolling of nested loops. International Journal of Parallel Programming **29**(5) (2001) 545–581

15. Wolf, M.E., Maydan, D.E., Chen, D.K.: Combining loop transformations considering caches and scheduling. In: Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, IEEE Computer Society TC-MICRO and ACM SIGMICRO (1996) 274–286
16. Duffin, R., Peterson, E., Zener, C.: Geometric Programming – Theory and Applications. John Wiley (1967)
17. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press. (Online version available at: `http://www.stanford.edu/~boyd/cvxbook.html`) (2004)
18. Kortanek, K.O., Xu, X., Ye, Y.: An infeasible interior-point algorithm for solving primal and dual geometric programs. Math. Program. **76**(1) (1997) 155–181
19. Löfberg, J.: YALMIP : A toolbox for modeling and optimization in MATLAB. In: Proceedings of the CACSD Conference, Taipei, Taiwan (2004) Available from `http://control.ee.ethz.ch/~joloef/yalmip.php`.
20. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1990) 53–65
21. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Trans. Program. Lang. Syst. **16**(6) (1994) 1768–1810
22. Carter, L., Ferrante, J., Hummel, S.F.: Hierarchical tiling for improved superscalar performance. In: Proceedings of the 9th International Symposium on Parallel Processing, Washington, DC, USA, IEEE Computer Society (1995) 239–245
23. Mitchell, N., Högstedt, K., Carter, L., Ferrante, J.: Quantifying the multi-level nature of tiling interactions. International Journal of Parallel Programming **26**(6) (1998) 641–670
24. Jiménez, M., Llabería, J.M., Fernández, A.: Register tiling in nonrectangular iteration spaces. ACM Trans. Program. Lang. Syst. **24**(4) (2002) 409–453
25. Rong, H., Tang, Z., Govindarajan, R., Douillet, A., Gao, G.R.: Single-dimension software pipelining for multi-dimensional loops. In: CGO '04: Proceedings of the international symposium on Code generation and optimization, Washington, DC, USA, IEEE Computer Society (2004)
26. Rong, H., Douillet, A., Govindarajan, R., Gao, G.R.: Code generation for single-dimension software pipelining of multi-dimensional loops. In: CGO '04: Proceedings of the international symposium on Code generation and optimization, Washington, DC, USA, IEEE Computer Society (2004)