Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization

Jason Cong, Peng Zhang, Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, pengzh, zouyi}@cs.ucla.edu

***Abstract*—External memory bandwidth is a crucial bottleneck in the majority of computation-intensive applications for both performance and power consumption. Data reuse is an important technique for reducing the external memory access by utilizing the memory hierarchy. Loop transformation for data locality and memory hierarchy allocation are two major steps in data reuse optimization flow. But they were carried out independently. This paper presents a combined approach which optimizes loop transformation and memory hierarchy allocation simultaneously to achieve global optimal results on external memory bandwidth and on-chip data reuse buffer size. We develop an efficient and optimal solution to the combined problem by decomposing the solution space into two subspaces with linear and nonlinear constraints respectively. We show that we can significantly prune the solution space without losing its optimality. Experimental results show that our scheme can save up to 31% of on-chip memory size compared to the separated two-step method when the memory hierarchy allocation problem is not trivial. Also, run-time complexity is acceptable for the practical cases.**

High-level synthesis; loop transformation; memory hierarchy optimization; data reuse

I. INTRODUCTION

Memory systems play an increasingly important role in modern computation system design, for both general-purpose processors and application-specific accelerators. External memory bandwidth is a dominant bottleneck for system performance and power consumption, and the total amount of on-chip memory is a main component of implementation cost [1]. Data reuse has proved to be an efficient technique for reducing external memory access with a relatively small on-chip memory cost. A great deal of attention has been paid over the past two decades to optimizing the external memory bandwidth by improving the data reuse and locality [2-20]. The research can be classified into two categories.

The work in the first category focuses on improving the data locality and date reuse by code transformation, especially loop transformation [2-9]. By changing the accessing order of array references in the loop nests, the co-located references become temporally "closer" and the data locality is improved. Various loop transformations, such as loop interchange, loop skewing, loop merging and loop tiling, were studied extensively in [2] and proved to be beneficial for data reuse. Reference [3] also presented a survey on the feasibility and profitability of these specific loop transformations and the sequential combination of them to form complex loop transformations. But this approach suffers from the complexity of enabling transformations for complex loop structures and the mismatch of different objective functions.

To overcome these limitations, polyhedral-based affine loop transformation framework is widely used to unify the combination of a sequence of specific loop transformations into one single affine transformation matrix [4-9]. The pioneering work [4, 5] used unimodular transformation matrices to have a unified representation of loop interchange, loop reversal and loop skewing transformations. Constructive solutions were given for finding the maximal number of parallel innermost and outermost loops. To support more general transformations and objectives, affine transformation framework was established based on parametric integer linear programming [6-8]. Data dependence and transformation legality constraints are expressed with a polyhedral model in a linear form. To improve data locality, iteration distances between dependent array instances are formulated in the objective function to be minimized. However, current loop transformations do not take the memory platform information into consideration due to the difficulty of modeling it directly in the cost functions.

The work in the second category optimizes the allocation of the reuse buffers in the memory hierarchy [10-15]. The data transfer and storage exploration (DTSE) methodology [1, 10] established an integrated design flow for the memory hierarchy optimization for customized memory systems. The optimization flow first analyzes the possible data reuse copies (the candidates of reuse buffers) of the array references at each loop level in the source program [11, 12]. Then, heuristics based on reuse buffer size and bandwidth reduction are applied to decide which reuse copies (buffer candidates) are selected to be implemented as an intermediate level in the memory hierarchy [13, 14]. In contrast to the heuristic approach, optimal allocation was also proposed by formulating the problem into a mixed linear programming optimization problem [15]. For all of these memory hierarchy allocation design flows, an independent loop transformation steering preprocessing is needed to optimize the data locality. And the final result of memory hierarchy optimization is greatly affected by this preprocessing.

This work was supported in part by the Semiconductor Research Corporation under Contract 2009-TJ-1879, and in part by the National Science Foundation under the Expeditions in Computing Program CCF-0926127.

Recently, researchers noticed the importance of considering memory platform characteristics in optimizing the loop transformation [16]. Loop transformation and memory hierarchy allocation are loosely coupled by introducing fast hierarchical memory size estimators [17, 18] to evaluate the promising transformations. But the authors do not provide a systematic way to find those promising transformations. Other researchers use analytic optimization formulations to optimize the loop tiling parameters and memory hierarchy allocation simultaneously [19, 20]. Their formulations are solved using non-linear optimization, such as sequential quadratic programming [19] and geometric programming [20]. However, these schemes still need affine transformations as a preprocessing procedure to improve data locality and enable tiling.

To the best of our knowledge, this is the first work to provide a systematic and efficient approach to optimizing the affine loop transformation and memory hierarchy allocation simultaneously to obtain the optimal data reuse implementation in memory system design. The contributions of this work are twofold.

- We formulate the global optimization problem in a concise mathematic form. In formulating bandwidth requirements, bi-directional binary allocation variables are used to address the problem that the reuse direction is unknown before the transformation is determined. In formulating reuse buffer size requirements, the calculation of the number of transformed loop iterations is converted into an internal integer point counting problem for a polytope.
- We also propose an efficient and optimal solution to the combined optimization problem. The solution space S is decomposed into a subspace (L) with linear constraints and a subspace (N) with non-linear constraints such that S=N×L. Our early feasibility determination and equivalent matrix elimination techniques further prune many suboptimal solutions in N so that the execution time is only seconds for most of our test cases. Experimental results show that for the given memory bandwidth constraints, up to 31% on-chip memory saving can be achieved compared to previous local optimal schemes.

The remainder of this paper is organized as follows. Section II demonstrates a motivation example to show the benefits of combined global optimization. Section III describes some preliminaries and the formulation of our combined optimization problem. Section IV proposes an efficient solution to the formulated non-linear optimization problem. Section V gives the experimental result and the discussions of the proposed scheme, and is followed by conclusions in Section VI.

II. A MOTIVATION EXAMPLE

As an example to demonstrate the necessity of combining affine loop transformation and memory hierarchy allocation, we first focus on the simple stencil code in Fig. 1(a). Four references of the same array A are located in the innermost loop of the loop nest, expressed as A0 (A[i, j, k]), A1 (A[i-3, j, k]), A2 (A[i, j-2, k]), and A3 (A[i, j, k-1]). An array element accessed by one reference in some iteration may be reused by other references in other iterations.

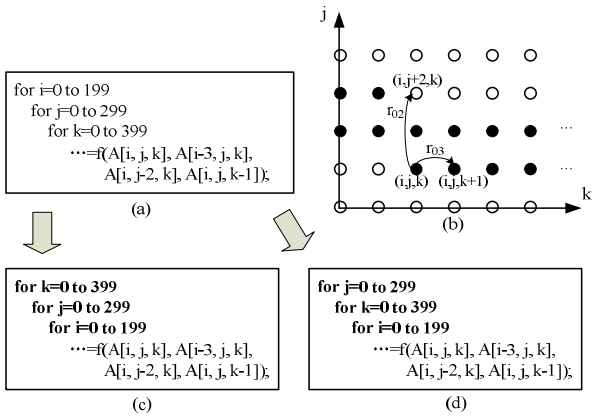

Figure 1. (a) Original loop nest. (b) Iterations of original loop nest. (c) Transformation T0 for locality of all data references. (d) Transformation T1 for locality of selected data references.

Reuse distance (vector) [21] is used to describe the difference of iteration vectors of two reusable array references. For example, the reuse distance from A0 to A1 is (3, 0, 0), which means the array element accessed by A0 in iteration (i, j, k) can be reused by A1 in iteration (i+3, j, k). If we allocate a reuse buffer in on-chip memory to store the data accessed by A0 until these data are used by A1, external memory bandwidth is saved for A1 access. To simplify our discussion, the loop bounds are far greater than the reuse distances. And the throughput of the loop nest measured by the number of innermost iterations executed per second is fixed. Hence, the total bandwidth is proportional to the number of references that do not reuse data from other references in one innermost iteration. The dead data in reuse buffers (which will no longer be reused) will be replaced by the recently accessed active data. The reuse buffer size is calculated as the maximal number of active data in the reuse buffer at each loop iteration, and is determined by reuse distance and the scanning order of the loop iterations. Reuse distance from A0 to A3 is (0, 0, 1), and only one buffer space is needed to realize the data reuse r_{03} in Fig. 1(b). But for reuse candidate r_{02} (from A0 to A2), the data reuse relation is carried by loop j. To realize the data reuse from A0 to A2, all solid-point data in Fig. 1(b) needs to be stored in the reuse buffer, and the buffer size is 2×400=800.

We adopt the data reuse analysis approach in [11, 12] to evaluate the reuse buffer size for data reuse at each loop level. For the program in Fig. 1(a), at innermost loop k, data reuse within this loop is (A0→A3). After realizing this data reuse, total bandwidth (BW) is reduced from the original 4 to 3 (normalized to the bandwidth for one reference), and the buffer size (BS) is 1. At intermediate loop j, data reuse (A0→A2) and (A0→A3) can be realized, and the corresponding BW and BS are 2 and 800 respectively. Loop transformation is used to improve the data locality and reduce the size of reuse buffers. We take two transformations T0 and T1 into consideration as shown in Fig. 1(c) and Fig. 1(d). If the comparison of two vectors is performed in the lexicographic order, T0 minimizes the maximal distance vector for all reuse reference pairs, and this is the result of traditional locality improving transformation. T1 only minimizes the distance of (A0→A1) and (A0→A3). We also evaluate all the BW vs. BS candidates for the transformed code in Fig. 1(c) and Fig. 1(d) respectively. The comparison is shown in Table I.

In this paper the terms bandwidth and buffer size are used specifically for external memory bandwidth and on-chip reuse buffer size respectively. We can see from Table I that for different bandwidth requirements, the optimal loop transformation that has the minimum on-chip memory for data reuse is different. Traditional locality-aware loop transformation (T0) minimizes all the possible reuse distances which can get the best result if the given bandwidth is small. But when more bandwidth is given, the selection of the data reuse distances to be optimized greatly impacts the loop transformation optimization. Trying to optimize the reuse distance of the references that are allocated in external memory will not help to reduce bandwidth, but will over-constrain the optimization of the reuse distance of the references that are allocated in on-chip reuse buffers. In our example, we select transformation T1 which only optimizes the reuse distances for (A0→A1) and (A0→A3). When the given bandwidth is 2, T1 has the minimal on-chip memory buffer size. In fact, if even more bandwidth is given as 3, only the innermost loop level data reuse is needed. In this case, the original code has the minimum buffer size instead.

TABLE I. BANDWIDTH AND BUFFER SIZE OF VARIOUS REUSE SCHEMES

Loop level of data reuse	Original		T0		T1	
	BW	***BS***	***BW***	***BS***	***BW***	***BS***
Innermost	3	1	3	3	3	3
Intermediate	2	800	2	400	2	200
Outermost	1	360k	1	60k	1	160k

As shown in this motivation example, the best reuse-aware loop transformation is highly dependent on the available off-chip bandwidth and on-chip buffer size. But it is hard to directly model these platform dependent requirements by the loop transformation alone. Memory hierarchy allocation can help to determine the selection of reuse distances to be optimized in loop transformation. Instead of previous loosely coupled methods which may get suboptimal results for various bandwidth and buffer size requirements, this paper proposes a fully integrated approach to optimize affine loop transformation and memory hierarchy allocation simultaneously to obtain the global optimal results.

III. PROBLEM FORMULATION

To simplify the formulation for the optimization problem, we make the following assumptions for the applications. The optimization is performed on perfectly nested loops. For imperfectly nested loops, we can also adopt the embedding approach proposed in [9] to convert the imperfectly nested loops into perfectly nested loops. In addition, loop bounds are constant or have a constant estimation. Array references are in the affine form of iteration variables, and the affine coefficients are far less than the loop bounds. The true data dependence distance and data reuse distance of two specific array references are constant (uniform distance). The majority of real-life computation-intensive applications satisfy these assumptions. Our problem is specified as: Given the high-level program accordant to our assumptions, find the optimal affine loop transformation with bounded coefficients and two-level memory hierarchy allocation which minimize the on-chip reuse buffer size under a specified bandwidth constraint. The dual problem, given buffer size constraint minimizing bandwidth, can be optimized by solving a sequence of the primal problems in a binary searching way.

A. Polyhedral Representation

Imperative programming language imposes total ordering for the execution of each statement, which is an over-constraint for our memory system optimization. Polyhedral representation only models the essential information of the application and presents it in the linear form.

DEFINITION 1 (ITERATION DOMAIN [6]). *The iteration vector of a m-level loop nest is a vector of iteration variable,* $\vec{i} = (i_0, i_1, ..., i_{m-1})^T$, *where* $i_0 \ldots i_{m-1}$ *are the iteration variables from outermost to innermost loop. Iteration domain* $\mathcal{D} \subset \mathbb{Z}^m$ *is the set of iteration vectors of the loop nest, and is expressed by a set of linear inequalities* $\mathcal{D} = \{\vec{i} \mid \mathbf{P}_{m\times m} \cdot \vec{i} \geq \vec{b}\}$.

EXAMPLE 1. *Consider the loop nest in Fig. 1(a):*

$$\mathcal{D} = \{(i_0, i_1, i_2) \mid \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i_0 \\ i_1 \\ i_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ -199 \\ 0 \\ -299 \\ 0 \\ -399 \end{pmatrix}, i_0, i_1, i_2 \in \mathbb{Z}\}$$

DEFINITION 2 (LEXICOGRAPHIC ORDER [6]). *Lexicographic order relation* $\succ_l$ *of two vector iteration vectors* $\vec{i}$ *and* $\vec{j}$ *is defined as:*

$$\vec{i} \succ_l \vec{j} \Leftrightarrow (i_0 > j_0) \vee (i_0 = j_0 \wedge i_1 > j_1) \vee (i_0 = j_0 \wedge i_1 = j_1 \wedge i_2 > j_2) \vee \ldots \vee (i_0 = j_0 \wedge \ldots i_{m-2} = j_{m-2} \wedge i_{m-1} > j_{m-1})$$

DEFINITION 3 (ACCESS FUNCTION [6]). *For a k-dimensional array reference, its access function* $\mathcal{H} \subset \mathbb{Z}^m \to \mathbb{Z}^k$ *is the mapping from iteration vector* $\vec{i}_m$ *to the access index* $\vec{h}_k$ *:*

$$\mathcal{H} = \{\vec{i}_m \to \vec{h}_k \mid \vec{h}_k = \mathbf{H}_{k\times m} \cdot \vec{i}_m + \vec{b}_k\}$$

Using iteration domain and access function, loop iterations and the array access are concisely described as polytopes in the space of integers. Data dependence and reuse possibility of array references can be analyzed from this polyhedral representation and expressed as dependence/reuse distance vectors [22, 23].

DEFINITION 4 (DEPENDENCE DISTANCE VECTOR [2]). *A_0 is a write reference and A_1 is a read reference of the same array. If the data element A_0 writes in iteration $\vec{i}$ is read by A_1 in iteration $\vec{j}$, the dependence distance vector is $\vec{d} = \vec{j} - \vec{i}$.*

DEFINITION 5 (REUSE DISTANCE VECTOR [2]). *A_0 and A_1 are two read references of the same array. If the data element A_0 reads in iteration $\vec{i}$ is the same as the one A_1 reads in iteration $\vec{j}$, the reuse distance vector from A_0 to A_1 is $\vec{r} = \vec{j} - \vec{i}$.*

EXAMPLE 2. *The reuse distance from A0 to A1 in Fig. 1(a) is a constant vector (3, 0, 0)^T.*

DEFINITION 6 (AFFINE LOOP TRANSFORMATION [6]). *Affine loop transformation changes the loop execution order by performing an affine transformation* $\mathbf{T} \in \mathbb{Z}^{m\times m}$ *on the loop iteration vector:* $\vec{i}\,' = \mathbf{T} \cdot \vec{i}$

EXAMPLE 3. *Consider the transformations T0 in Fig. 1(c) and T1 in Fig. 1(d):*

$$\mathbf{T}0=\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix},\quad \mathbf{T}1=\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}$$

The transformed distance vector can be calculated directly as $\vec{d}'=\mathbf{T}\cdot\vec{d}$ and $\vec{r}'=\mathbf{T}\cdot\vec{r}$. Not every matrix generates a legal loop transformation that preserves the semantics of the original program. Assuming read-to-write and write-to-write data dependence has been removed by a renaming preprocessing [2], we only consider true data dependence constraints in this paper.

THEOREM 1 (LEGALITY OF AFFINE LOOP TRANSFORMATION [6]). *An affine transformation* $\mathbf{T}$ *is legal iff for every data dependence distance vector* $\vec{d}$ *in the loop nest, the transformed distance vector is lexicographically positive:* $\forall\vec{d}\in\mathscr{R},\mathbf{T}\cdot\vec{d}\succ_l\vec{0}$, *where* $\mathscr{R}$ *is the set of dependence distance vectors.*

B. Reuse Graph and Hierarchy Allocation

Data reuse graph [13, 14] that represents data reuse candidates, is always used as the input for memory hierarchy allocation optimization. Nodes of the graph are reuse copies (array references in our simplified case), and edges weighted by the reuse distance are the possible data reuse between two nodes. Binary decision variables are allocated for each edge to indicate whether the reuse is realized in the on-chip buffer. Bandwidth and buffer size can be calculated using these binary allocation variables to formulate the optimization problem [15].

In our formulation the reuse direction between two references is unknown before the loop transformation is determined. This makes it difficult to calculate bandwidth consumption from allocation variables. To consider all the possible data reuse directions, two binary variables are used to indicate the data reuse of two references from two directions. Using these binary variables and reuse distances in the reuse graph, we can calculate the bandwidth and buffer size.

DEFINITION 7 (BIDIRECTIONAL DATA REUSE GRAPH). *For a loop nest L, the bidirectional data reuse graph is a directed graph* $G=(V,E)$, *where V is the read references in L, and for every reuse relation* $v_s\rightarrow v_t$ *with distance* $\vec{r}_{st}$ (v_s *and* v_t *in V), two edges* e_{st} ($v_s\rightarrow v_t$) *and* e_{ts} ($v_t\rightarrow v_s$) *are in E weighted by* $\vec{r}_{st}$ *and* $-\vec{r}_{st}$.

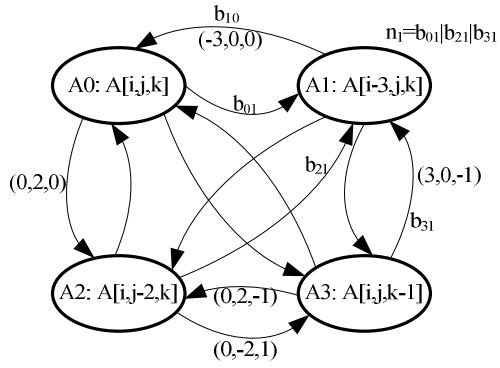

Figure 2. Example of bidirectional data reuse graph.

Fig. 2 shows the proposed data reuse graph for the program in Fig. 1(a). The binary variable b_{xy} indicates whether data reuse from node $x\in V$ to node $y\in V$ is realized in the on-chip reuse buffer. We also introduce an intermediate binary variable n_y for each node $y\in V$ as $n_y=\bigvee_{x\in V,x\neq y}b_{xy}$, indicating whether node y reuses data from other nodes. Because loop bounds are assumed to be far greater than reuse distance, we can consider a node with n_y=1 as totally reused, and ignore marginal un-reusable access on the boundaries. If V has n nodes, the normalized bandwidth is calculated as:

$$BW=n-\sum_{y\in V}n_y$$

C. Buffer Size Calculation

The buffer size of one reuse edge is the number of iterations within the iteration distance. The value is equal to the inner product of data reuse distance and the loop iteration vector.

DEFINITION 8 (ACCUMULATED ITERATION VECTOR, AIV). *For an m-level loop nest, its AIV is* $\vec{l}=(l_0,l_1,...,l_{m-1})^T$, *where* $l_{m-1}=1$, *and* $\forall k=0..m-2$, l_k *is equal to the maximum number of the total iterations of m-k-1 inner loops from k+1 to m-1 within one iteration of loop k.*

EXAMPLE 4. *The AIV of the loops in Fig. 1(a) is* $\vec{l}$ *=(120000, 400, 1)*^{T}. *If the reuse distance is* $\vec{r}$ *=(2, 0, 0), the buffer size is the total number of iterations between iteration (i, j, k) and (i+2, j, k), which is equal to the inner product of* $\vec{l}$ *and* $\vec{r}$ *(240000).*

PROBLEM 1. *Given iteration domain* $\mathscr{D}$ *and affine transformation* $\mathbf{T}$, *calculate transformed AIV* $\vec{l}\,'(\mathbf{T})$.

Since each iteration in the transformed loops corresponds to one iteration in the original loops, we can calculate $\vec{l}\,'(\mathbf{T})$ by counting the number of original iterations mapped into the inner levels of the transformed loop nest when iterations of outer loops are fixed.

$$l_k'(\mathbf{T})=\underset{\vec{i}}{Max}\left(\left|\{\vec{i}\,'\mid\vec{i}\,'\in\mathscr{D}',\ \vec{i}_k'=\vec{t}\}\right|\right)=\underset{\vec{i}}{Max}\left(\left|\{\vec{i}\mid\vec{i}\in\mathscr{D},\ \mathbf{T}_k\cdot\vec{i}=\vec{t}\}\right|\right),$$

where $\mathscr{D}'$ is the transformed iteration domain, $\vec{i}_k'$ is iteration vector of the outer k+1 transformed loops, $\vec{t}\in\mathbb{Z}^{k+1}$ is an intermediate vector, and $\mathbf{T}_k$ is the upper k+1 rows of matrix $\mathbf{T}$. Integer points in a polytope can be counted by polylib [24] and Barvinok [25] libraries.

It seems unnecessary that the total buffer size is equal to the sum of the sizes of all separate realized reuse buffers, because overlapped reuse buffers may share data. However, the following theorem ensures that data sharing between reuse buffers does not need to be specially considered in the formulation.

THEOREM 2 (NON-OVERLAP REUSE). *For the problem minimizing the sum of the allocated reuse buffer sizes under bandwidth constraints, no same data will be allocated in two reuse buffers in the optimal solution.*

PROOF. We first assume an optimal solution of the problem has data overlap as in Fig. 3(a). The relative order of the reused array instances *a, b, c, d* is determined after transformation. If we remove the overlapped part from one reuse buffer as in Fig. 3(b), the sum of the total buffer size will reduce and bandwidth will not increase. Then, we get a better solution than Fig. 3(a), which contradicts the assumption that Fig. 3(a) is optimal. □

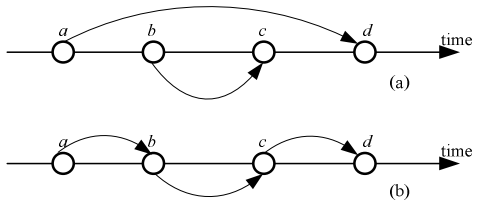

Figure 3. (a) A supposed optimal solution with data overlap. (b) A contradictory case with smaller buffer size

D. Combined Optimization Problem

From the discussion above, we can summarize our formulation as Problem 2. Eqn. 1 and Eqn. 2 are responsible for the legality of transformation and direction of data reuse respectively. Eqn. 3 and Eqn. 4 calculate the bandwidth. Eqn. 5 and Eqn. 6 calculate the buffer size, and Eqn. 7 ensures that the ordering of all the iterations is determined for code generation after transformation.

PROBLEM 2. *Given an original iteration domain $\mathscr{D}$ with m level of loops, a set of dependence distance vectors $\mathscr{R}$, a set of array references V and their reuse distance vectors $\mathscr{S} = \{\vec{r}_{xy} \mid x, y \in V\}$, and a bound of normalized bandwidth N, find the optimal loop transformation* $\mathbf{T}$ *and memory hierarchy allocation* $\{b_{xy}\}$ *to*

Mininize	BS	
Subject to	$\forall \vec{d} \in \mathscr{R}, \mathbf{T} \cdot \vec{d} \succ_l \vec{0}$	(1)
	$\forall (x, y \in V \wedge x \neq y), b_{xy}(\mathbf{T} \cdot \vec{r}_{xy}) \geq_l \vec{0}$	(2)
	$\forall y \in V,\ n_y = \bigvee_{x \in V, x \neq y} b_{xy}$	(3)
	$BW = n - \sum_{y \in V} n_y \leq N$	(4)
	$BS = \sum_{x,y \in V \wedge x \neq y} b_{xy}(\mathbf{T} \cdot \vec{r}_{xy})^T \cdot \vec{l}\,'(\mathbf{T})$	(5)
	$l_k'(\mathbf{T}) = \underset{\vec{i}}{Max}(\lvert\{\vec{i} \in \mathscr{D},\ \mathbf{T}_k \cdot \vec{i} = \vec{t}\}\rvert)$	(6)
	$rank(\mathbf{T}) = m$	(7)

IV. EFFICIENT SOLUTION

In general, Problem 2 has a non-convex form in Eqn. (6) and Eqn. (7), but we can have an efficient solution by utilizing the problem characteristics and mathematic transformations.

A. Enumeration-Based Method

In Problem 2, lexicographic ordering in Eqn. 1 and 2 can be converted into linear form by introducing additional variables to indicate the positions of positive components. Nonlinear operations on binary variables in Eqn. 2, 3 and 5 can also be linearized using the approaches in [15].

But the non-linear terms in Eqn. 6 and 7 are not easy to remove. We notice that these two equations are only related to the loop transformation. In practical cases, the loop level associated with data reuse is relatively small, and the coefficient value of the optimal transformation matrix is also small (always in [-1, 1]). If we enumerate the space of the transformation matrix, Problem 2 is converted into a set of linear programming problems.

We propose an enumeration-based two-step method to solve Problem 2 as in Fig. 4. Instead of a brute-force search, we present a set of efficient space pruning techniques to speed up the search, while maintaining the optimality of results.

DEFINITION 9 (REUSE-FREE LOOP). *A reuse-free loop is a loop that does not carry realized reuse after transformation.*

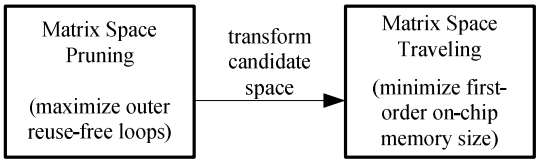

Figure 4. Enumeration-based solving method.

Since the loop bound is large enough, the optimal solution must have the maximal number of outer reuse-free loops, which means solutions with fewer outer reuse-free loops can be pruned. The number of outer reuse-free loops will be efficiently maximized by the techniques in Sections IV.B and IV.C.

B. Partial Feasibility Test

Outer p+1 reuse-free loops are related to the upper p+1 rows of transformation matrix $\mathbf{T}_p$ and allocation variables $\{b_{xy}\}$.

PROBLEM 3. *Problem 2 with additional constraints:*

$$\forall (x, y \in V \wedge x \neq y), b_{xy}(\mathbf{T}_p \cdot \vec{r}_{xy}) = \vec{0}. \quad (8)$$

If a matrix $\mathbf{T}_p$ satisfies Eqn. 8, its sub-matrix containing rows of $\mathbf{T}_p$ must also satisfy it. So we can enumerate the maximal-ranked $\mathbf{T}_p$ in a row-by-row incremental way as Algorithm 1. Only feasible sub-matrices are used to enumerate larger ranked matrices. Row candidates in line 6 are all possible rows for p=0, and then all feasible one-row sub-matrices instead for $p \geq 1$. CheckRank in line 8 checks whether a matrix is full-row-ranked. CheckNormalForm in line 9 and CheckDependence in line 16 are described later in Section IV.C.

CheckBandwidth tests the feasibility of Problem 3 for the given upper p rows of the transformation matrix. Eqn. 5 and 6 are ignored. We cannot simply ignore Eqn. 2 even though we have Eqn. 8. Because $\mathbf{T} \cdot \vec{r}_{xy}$ will never be zero, the direction of data reuse is constrained in Eqn. 2. But in Eqn. 8, the reuse direction may be not constrained, and both variables b_{xy} and b_{yx} may be one. This reuse cycle makes the bandwidth calculation incorrect. However, for calculating the bandwidth for a sub-matrix, we can arbitrarily give a relative order for the

references x and y if $\mathbf{T}_p \cdot \vec{r}_{xy} = \vec{0}$. This is because if we swap these two references, (1) Eqn. 3 and 4 will be unchanged because of their symmetry; (2) Eqn. 8 will also have the same form because $\mathbf{T}_p \cdot \vec{r}_{zy} = \mathbf{T}_p \cdot \vec{r}_{zx}$. In our CheckBandwidth, the bandwidth feasibility is directly checked by Eqn. 9, 3, and 4.

$$\forall (x, y \in V \wedge x \neq y),\ b_{xy} = (\mathbf{T}_p \cdot \vec{r}_{xy} == \vec{0}) \wedge (x < y) \qquad (9)$$

Algorithm 1 Outer Reuse-Free Sub-matrix Space Pruning

```
1: S     → set of feasible matrices
2:
3: add an abstract zero-row matrix into S
4: for p = 0 to m do
5:    for all p row matrices T_{p-1} in S do
6:       for all one-row candidates r do
7:          append row r to T_{p-1} to form Tr
8:          feasible  = CheckRank (Tr)
9:          feasible &= CheckNormalForm (Tr)
10:         feasible &= CheckBandwidth (Tr)
11:         if (feasible) then add p+1 row matrix Tr to S
12:      end for
13:   end for
14:   if no p+1 row matrix in S then break
15: end for
16: delete each T_p from S if CheckDependence (T_p) fails
17: return all matrices with maximal rows in S
```

C. RT-Equivalent Matrix Pruning

Investigating Eqn. 8 and 9 further, we find that a set of sub-matrices will generate the same constraints on $\{b_{xy}\}$.

DEFINITION 10 (RT-EQUIVALENT MATRIX). *Matrices* **A** *and* **B** *are row-transformed (RT) equivalent matrix if* **B** *can be generated from* **A** *by a sequence of elementary row transformations.*

THEOREM 3. *Two RT-equivalent sub-matrices impose the same constraints on* $\{b_{xy}\}$ *in Eqn. 8.*

PROOF. One elementary row transformation is equivalent to left-multiplying an elementary row transformation matrix.

$$\therefore\ \mathbf{A} \cdot b_{xy}\vec{r}_{xy} = 0 \Leftrightarrow \mathbf{E}_0 \cdot \ldots \cdot \mathbf{E}_l \cdot \mathbf{A} \cdot b_{xy}\vec{r}_{xy} = 0 \Leftrightarrow \mathbf{B} \cdot b_{xy}\vec{r}_{xy} = 0\ .\ \square$$

DEFINITION 11 (RT-NORMAL FORM). *A matrix is in the RT-normal form iff it has the following properties:*

(1) The leading position of each row increases. The leading position of a row is the position of the leftmost nonzero element.
(2) The coefficients in the leading positions are positive.
(3) The column of those leading positions has only one nonzero.

EXAMPLE 5. *A typical RT-normal matrix looks like:*

$$\begin{pmatrix} 0 & + & * & 0 & * & 0 & * \\ 0 & 0 & 0 & + & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & + & * \end{pmatrix}$$

The RT-normal form can be achieved by procedures similar to the Gaussian elimination. We can just enumerate the RT-normal sub-matrices for the outer reuse-free loops, because other non-RT-normal sub-matrix will not get better results than the RT-normal ones. Then, the form of Eqn. 7, 8 and 9 is unchanged, but Eqn. 1 is updated to:

$$\exists \mathbf{E}_{p\times p}, \det(\mathbf{E}_{p\times p}) \neq 0, \forall \vec{d} \in \mathscr{R}, \mathbf{E}_{p\times p} \cdot \mathbf{T}_p \cdot \vec{d} \succ_l \vec{0}\ . \qquad (10)$$

In Algorithm 1, CheckNormalForm checks whether a sub-matrix is in RT-normal form using Definition 11. And CheckDependence checks Eqn. 10 using Fourier-Motzkin elimination. Because dependence check (Eqn. 10) is not incremental as bandwidth check and normal form check, CheckDependence is performed on the output of reuse-free loop maximization. Finally, Algorithm 1 finds the maximal feasible outer reuse-free sub-matrices which include all the optimal solutions of the original Problem 2, while largely reducing the search space by the sub-matrix pruning.

D. First-Order Buffer Size Optimization

The dominant (first-order) part of the buffer size is allocated at the outermost loop which carries the realized data reuse. In this section we enumerate one more row which is appended to the outer reuse-free sub-matrix to minimize the dominant part of the buffer size. For a fixed **T**, the buffer size for each reuse distance is determined: $S_{xy} = (\mathbf{T} \cdot r_{xy})^{\mathrm{T}} \cdot l'(\mathbf{T})$. To simplify Eqn.2, we introduce $t_{xy} = (\mathbf{T}_{p+1} \cdot \vec{r}_{xy} \prec_l \vec{0}) \vee (\mathbf{T}_{p+1} \cdot \vec{r}_{xy} = \vec{0} \wedge x > y)$ to specify the reuse direction between reuse nodes. The allocation optimization can be expressed in a linear programming form as Problem 4.

PROBLEM 4. *Given* t_{xy}, *n, N, and* S_{xy}, *find optimal* $\{b_{xy}\}$ to

$$\begin{aligned} &\textit{Minimize} && BS = \sum_{x,y\in V \wedge x\neq y} b_{xy} S_{xy} && \\ &\textit{Subject to} && \forall x, y \in V \wedge x \neq y \wedge t_{xy}, b_{xy} = 0 && (11) \\ & && \forall y \in V, \sum_{x\in V, x\neq y} b_{xy} - n_y \geq 0 && (12) \\ & && \sum_{y\in V} n_y \geq n - N && (13) \\ & && \forall (x, y \in V \wedge x \neq y), 0 \leq b_{xy} \leq 1 \wedge 0 \leq n_y \leq 1 && (14) \end{aligned}$$

THEOREM 4. *The optimal solution for the linear programming Problem 4 is always integral.*

We omit the proof of Theorem 4 due to page limitations. Theorem 4 ensures that we can find optimal integer solutions for the allocation variables by solving a linear programming which has efficient polynomial-time solving algorithms.

V. EXPERIMENTAL RESULTS

Our data reuse optimization algorithm is performed as a source-to-source preprocessing step to a high-level synthesis tool for evaluation, as shown in Fig. 5. Our design flow takes loop kernels in high-level specifications like C/C++ as input, and analyzes the polyhedral intermediate representation (IR) with dependence and reuse distances by the LLVM-Polly framework [26]. Data reuse optimization finds the optimal loop transformation and on-chip buffer allocation. In the code generator, loop transformation is performed by ClooG [27], and the on-chip buffer is generated as in [12, 19]. The optimized loop kernels are synthesized into VHDLs and then circuit netlists by the high-level synthesis tool AutoPilot [28] and the ASIC synthesis tool Design Compiler [29].

Table II. EXPERIMENT RESULTS

Design	*n*	*N*	Only HA				Separated LT+HA				Combined LT+HA				
			Logic	SRAM	Latency	Power	Logic	SRAM	Latency	Power	Logic	SRAM	Latency	Power	Time
FDTD_3D	4	2	60	76	2.43E+8	12.6	59	30	2.43E+8	2.8	56	25	2.43E+8	2.6	0.12
JACOBI_3D	7	5	61	129	3.65E+8	23.3	60	86	3.65E+8	14.6	58	74	3.65E+8	12.0	0.21
DENOISE	13	5	33	224	6.08E+8	36.3	32	149	6.08E+8	26.3	28	124	6.08E+8	21.1	0.36
SEG	27	3	224	761	1.22E+8	116.0	224	300	1.22E+8	18.0	221	254	1.22E+8	17.0	0.32
ME	64	8	63	149	1.32E+7	26.5	61	124	1.32E+7	21.4	61	124	1.32E+7	21.4	0.32
FDTD_4D	5	3	75	76	3.6E+10	12.9	71	25	3.6E+10	2.9	72	13	3.6E+10	2.6	0.66
JACOBI_4D	9	3	80	129	6.1E+10	23.7	75	74	6.1E+10	12.3	76	20	6.1E+10	2.9	2.4
Geomean			1.00	1.00	1.00	1.00	0.97	0.52	1.00	0.39	0.94	0.36	1.00	0.29	

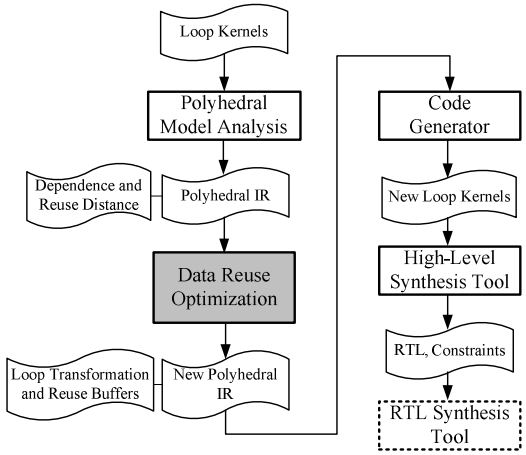

Figure 5. Implementation flow.

Our test designs include a set of real-life data-intensive loop kernels: FDTD and JACOBI are stencil codes chosen from polybench 2.0 [30]; ME is the 8×8 block matching algorithm for motion estimation in video encoding [31]; DENOISE smoothes a 3D image by averaging neighboring 13 pixels[32]; and SEG is a two-phase image segmentation algorithm [32]. The input data size is 352×288×600 and 352×288×600×100 for 3D and 4D kernels respectively. The proposed combined loop transformation and memory hierarchy allocation scheme (combined LT+HA) is compared with two reference points in our experiments. The first reference point is the optimal memory hierarchy allocation with original source code (only HA). And the second point separately optimizes the LT and HA (separated LT+HA) as was done in [12, 15].

A. Result and Analysis

Experimental results of the three approaches are reported in Table II. The second column (n) in Table II shows the number of read references in each design. And the third column (N) is the bandwidth requirement (normalized to the bandwidth of one read reference in the loop nest) for each design, which is calculated from the available external memory bandwidth (1GB/s) and the design performance requirements. We set the clock frequency as 5ns, and all design implementations satisfy the timing constraint. The ASIC implementation results of the three approaches in 65nm process technology are compared, such as the area of logic standard cells and on-chip SRAM in 103μm2, the execution latency in cycles, and the power consumption in mW. We also list the execution time (in seconds) of our combined optimization algorithm in the last column. We normalize the four metrics to the values of the only HA scheme, and calculate the geometric mean of the normalized data as shown in the last row of Table II.

From the results, it is clear that loop transformation is important to data reuse optimization in memory hierarchy. Compared to the only HA scheme, the separated LT+HA scheme and the combined LT+HA scheme can save the on-chip memory size by 48% and 64% respectively, and also reduce the power consumption by 61% and 71% respectively. The saving of on-chip memory is achieved by shortening the lifetime of the reused data using loop transformation. And our combined LT+HA scheme gains an additional 31% memory reduction compared to the separated LT+HA scheme, because loop transformation has more precise objective functions to optimize when considering memory hierarchy allocation simultaneously. The gain in ME design is small because the optimization space for this 2-level loop ME design is relatively small and the separated LT+HA scheme also gets the optimal result. Because the external memory bandwidth is fixed, the power reduction mainly comes from the leakage power saving because there is less on-chip SRAM allocated. The experimental results also show that the logic cells area and the execution latency have slight differences between the three approaches, which are less than 6% and 1% respectively.

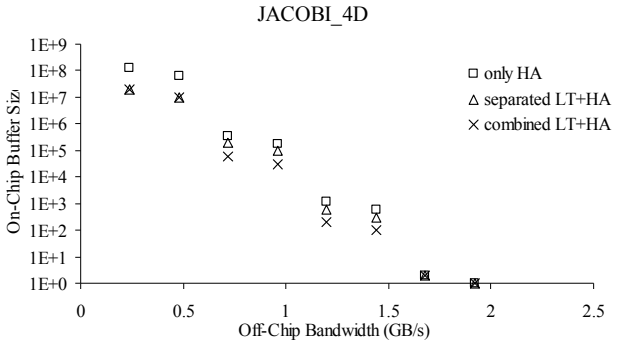

Figure 6. Design space exploration.

Fig. 6 investigates the design space for the trade-off between bandwidth and buffer size. When the bandwidth is too high or too low, the optimization space of memory hierarchy allocation is relatively small, and the separated LT+HA scheme can also get optimal results. For the bandwidth in-between, when the memory hierarchy allocation is not trivial, our combined LT+HA scheme outperforms the other schemes significantly.

B. Run-Time Complexity

In our iterative algorithm, the complexity of the feasibility check procedures and memory hierarchy allocation are in polynomial time, but the pruned search space size for loop transformation is still exponential to the loop level. In practice, the run-time is less than three seconds for our experimental cases. In addition, we run random tests for more loop nests and more references, and report the maximal run-time in Table III. Table III shows that our scheme is efficient in a large range of real-life cases which have less than 5 loops or 50 reuse references.

Table III. EXECUTION TIME ON RANDOM CASES (IN SECONDS)

#reuse reference	3-level loop	4-level loop	5-level loop
10	0.13	0.65	10
20	0.53	5.6	57
50	3.5	40	463
100	20.4	306	4662

VI. CONCLUSION

Loop transformation and memory hierarchy allocation are two coupled steps in the data reuse optimization flow. In this work we present a combined optimization algorithm to optimize loop transformation and memory hierarchy allocation simultaneously to obtain global optimal results in on-chip reuse buffer size and external memory bandwidth. A series of efficient space pruning techniques are proposed to speed up the execution of our algorithm by considering the characteristics of the loop transformation matrix and memory allocation constraints.

Our future work will concentrate on integrating more optimizations such as loop tiling and inter-loop-nest optimization, and further reducing the algorithm complexity by introducing intelligent heuristics which can achieve nearly optimal results for complex applications.

REFERENCES

[1] F. Catthoor, S. Wuytack, G. E. Greef, F. Banica, L. Nachtergaele, A. Vandecappelle, Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design. Kluwer Academic, Dordrecht, Netherlands, 1998.

[2] R. Allan, K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kauffman Publishers, 2002.

[3] K. S. McKinley, S. Carr, and C. W. Tseng, "Improving data locality with loop transformations," ACM Trans. Program, Language and Systems, 18(4), pp. 424-453, 1996.

[4] M. E. Wolf, M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," IEEE Transaction on Parallel and Distributed Systems, 2(4), October 1991.

[5] M. Wolf and M. S. Lam, "A data locality optimizing algorithm," ACM SIGPLAN PLDI '91, pages 30–44, 1991.

[6] P. Feautrier, "Some efficient solutions for the affine scheduling problem, part I, one dimensional time," International Journal of Parallel Processing, 21(6), December 1992.

[7] P. Feautrier, "Some efficient solutions for the affine scheduling problem, part II, multi-dimensional time," International Journal of Parallel Processing, 21(6), December 1992.

[8] A. W. Lim, G. I. Cheong, M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," Proceedings of the 13th International Conference on Supercomputing, ICS 1999.

[9] N. Ahmed, N. Mateev, and K. Pingali, "Synthesizing transformations for locality enhancement of imperfectly-nested loops," International Journal of Parallel Processing, 29(5), Oct. 2001.

[10] F. Catthoor, K. Danckaert, C. Kulkarni, Data Access and Storage Management for Embedded Programmable Processors. Kluwer Academic Publishers, 2002.

[11] T. V. Achteren, F. Catthoor, R. Lauwereins, G. Deconinck, "Data reuse exploration techniques for loop-dominated applications," IEEE/ACM Des. Autom. and Test Conf. (DATE), Paris, 2002.

[12] I. Issenin, E. Brockmeyer, M. Miranda, N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," ACM Trans. Des. Autom. Electron. Syst., 12(2), p. 15, 2007

[13] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organizations," IEEE/ACM Design Automation and Test Conference (DATE), pages 1070–1075, 2003

[14] R. Baert, E. d. Greef, E. Brockmeyer, "An automatic scratch pad memory management tool and MPEG-4 encoder case study," 45th Design Automation Conference (DAC), 2008

[15] I. Issenin and N. Dutt, "Data reuse driven energy-aware MPSoC co-synthesis of memory and communication architecture for streaming applications," 4th Int. Conf. CODES+ISSS, 2006, pp. 294–299

[16] M. Palkovic, H. Corporaal, F. Catthoor, "Trade-offs in loop transformations," ACM Transactions on Design Automation of Electronic Systems, 14(2), March 2009.

[17] P. R. Panda, N. D. Dutt, A. Nicolau, "Local memory exploration and optimization in embedded systems," IEEE Trans. Comput.-Aided Des., 18(1), page 3–13, January 1999.

[18] Q. Hu, P. G. Kjeldsberg, A. Vandecappelle, M. Palkovic, F. Catthoor, "Incremental hierarchical memory size estimation for steering of loop transformations," ACM Transactions on Design Automation of Electronic Systems, 12(4), 2007

[19] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08), 2008

[20] Q. Liu, G. A. Constantinides, K. Masselos, P. Cheung, "Combining data reuse with data-level parallelization for FPGA targeted hardware compilation: a geometric programming framework," IEEE Trans. Comput.-Aided Des., 28(3), page 305–315, March 2009.

[21] M. Kandemir, A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," Design Automation Conference (DAC), 2002

[22] Omega project, http://www.cs.umd.edu/projects/omega

[23] Piplib, http://www.piplib.org

[24] Polylib, http://icps.u-strasbg.fr/polylib

[25] Barvinok library, http://freshmeat.net/projects/barvinok

[26] LLVM-polly, https://llvm.org/svn/llvm-project/polly

[27] ClooG, http://www.cloog.org

[28] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst., 30(4), pp. 473-491, April 2011

[29] Synopsys Website, http://www.synopsys.com

[30] Polybench, http://www.cse.ohio-state.edu/~pouchet/software

[31] H.264 video coding reference software, http://iphome.hhi.de/suehring/tml

[32] CDSC medical image benchmarks, http://www.cdsc.ucla.edu