Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis

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

For the majority of computation-intensive application systems, off-chip memory bandwidth is a critical bottleneck for both performance and power consumption. The efficient utilization of limited on-chip memory resources plays a vital role in reducing the off-chip memory accesses. This paper presents an efficient approach for optimizing the on-chip memory allocation by loop transformations in the imperfectly nested loops. We analytically model the on-chip buffer size and off-chip bandwidth after affine loop transformation, loop fusion/distribution and code motion. Branch-and-bound and knapsack reuse techniques are proposed to reduce the computation complexity in finding optimal solutions. Experimental results show that our scheme can save 40% of on-chip memory size with the same bandwidth consumption compared to the previous approaches.

Categories and Subject Descriptors: B.5.2 [**Hardware**]: Design Aids – *optimization*

General Terms: Algorithms, Design, Experimentation

Keywords: High-Level Synthesis, Loop Transformation, Memory Hierarchy Optimization, Data Reuse

1. INTRODUCTION

Off-chip memory bandwidth is a dominant bottleneck for performance and power consumption in digital hardware systems. On-chip memories have sufficient bandwidth but limited sizes due to implementation cost [1]. Allocating a portion of large arrays in on-chip buffers has proven to be an efficient technique to reduce the off-chip memory accesses in digital system designs. High-level synthesis tools enable these optimizations to be performed at the C-code level by traditional compiler techniques. A great deal of attention has been paid over the past two decades to optimizing the off-chip memory bandwidth by improving data reuse and locality [2-25]. The research can be classified into two categories.

The work in the first category focuses on improving data locality and date reuse by code transformation, especially loop transformation [2-11]. By changing the accessing order of array references in the loop nests, the co-located references become temporally "closer", which means a smaller data reuse buffer. Specific loop transformations, such as loop interchange, loop skewing, loop merging and loop tiling, were studied one by one in [2, 3], including the feasibility and profitability of the transformations, and the sequential combination of them to form complex transformations. But in practice, it is much harder to analytically model the sequential combinations of these loop transformations. Polyhedral-based loop transformation 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 for a unified representation of loop interchange, loop reversal and loop skewing transformations. To support more general transformations and objectives, affine transformation frameworks were established based on parametric integer linear programming [6, 7]. 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. Beside affine transformation, loop fusion/distribution, code motion and tiling for imperfectly nests loops have also been studied in recent work [8-11]. However, these models [2-11] use simple platform-independent objective functions, which cannot accurately model the impact of memory hierarchy allocation in hardware synthesis. For example, memory accesses to on-chip and off-chip memories will have significantly different cost models.

The work in the second category optimizes the allocation of the reuse buffers in the memory hierarchy for a fixed loop order. The data transfer and storage exploration (DTSE) methodology [1, 12] established an integrated design flow for the memory hierarchy optimization for customized memory systems. The optimization flow first analyzes the data reuse graph which presents all the possible data reuse buffer candidates of the array references at each loop level in the source program [13, 14]. Then, heuristics based on reuse buffer size and bandwidth reduction are applied to decide the allocation of the reuse candidates and their memory hierarchy [15-18]. In contrast to the heuristic approaches, an optimal allocation was proposed by formulating the problem into a mixed linear programming optimization problem [19]. For all these hierarchy allocation approaches, an independent loop transformation preprocessing is assumed to optimize data locality. The final result of memory hierarchy optimization may be greatly affected by this preprocessing.

Recently, researchers noticed the importance of considering platform-dependent cost modeling in optimizing the loop transformation [20]. Loop transformation and memory hierarchy allocation are loosely coupled by introducing fast hierarchical memory size estimators [21, 22] to evaluate the promising transformations. But the search process lacks analytic model for

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

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

guidance which makes it inefficient to search a large transformation space. Other researchers use analytic optimization formulations to optimize the loop tiling parameters and memory hierarchy allocation simultaneously [23, 24]. Their formulations are solved using non-linear optimization, such as sequential quadratic programming [23] and geometric programming [24]. However, these schemes [23, 24] still need affine transformations as a preprocessing procedure to improve data locality and enable tiling. Different from previous approaches, the recent study in [25] combines affine transformation and hierarchy allocation in an analytic and systematic way. A model-guided searching-based approach is used to find the optimal affine loop transformation and hierarchy allocation. But the work is limited to perfectly nested loops, so it is not applicable to real applications.

In this paper we propose an efficient approach for optimizing the on-chip memory allocation with loop transformations for imperfectly nested loops. The contributions of this work are:

- We propose an analytical modeling of hierarchy allocation problem with loop transformations for imperfectly nested loops. Buffer size is calculated after loop transformations such as affine loop transformations, loop fusing/distribution and code motion.
- We develop an efficient and optimal solution to the combined problem, which uses the branch-and-bound approach to prune the sub-optimal transformation space and the knapsack reuse technique to reduce the complexity of each transformation.

The remainder of this paper is organized as follows. Section 2 demonstrates a motivation example to show the benefits of combined optimization. Section 3 describes some preliminaries and the formulation of our combined optimization problem. Section 4 proposes an efficient solution to the formulated optimization problem. Section 5 gives the experimental results, and is followed by conclusions in Section 6.

2. MOTIVATION EXAMPLE

The work in [25] made a good case for combining loop transformation and memory hierarchy, but it was limited to perfectly nested loops. Here we use an example with two loop nests in Fig. 1(a) to show the necessity for supporting imperfectly nested loop and loop fusion/distribution. Data reuse between array references can be exploited by allocating on-chip buffers. For example, Fig. 1(b) shows the data reuse from reference *A[i,j]* to *A[i-2,j]*. The on-chip reuse buffer size is 2*N*, because the data fetched by reference *A[i,j]* will be used by reference *A[i-2,j]* after two loop *i* iterations, and the 2*N* data elements accessed during this period (two loop *i* iterations) need to be stored in the reuse buffer for continuous data reuse. After the data in the buffer is reused, it can be replaced to store new reusable data. The modulo operation in the reuse buffer addressing indicates that the buffer is accessed and updated in a cyclic way. By allocating the reuse buffer, off-chip memory accesses by reference *A[i-2,j]* are saved*.

Loop transformation can be used to reduce the buffer size by improving the data locality of array accesses. T0 in Fig. 1(c) is the result of the traditional loop optimizers, which minimizes the size of the largest reuse buffer. The largest reuse buffer in the original code is from *S1:B[i,j]* to *S2:B[i,j-3]*, whose size is N^2 because S1 and S2 are in different loop nests. In T0 the largest buffer size (from *S1:A[i,j]* to *S2:A[i-3,j]*) is reduced to 3*N* because reuse buffer can be cyclically reused every three loop *i* iterations.

However, traditional loop transformation do not consider the impact of memory hierarchy allocation (the selection of the reuse buffers to be allocated on-chip). T0 is the optimal transformation when all the data reuse buffers are allocated. But in practical cases, on-chip memory may not be sufficient for all these buffers. Trying to optimize the locality of off-chip accesses does not have much benefit because off-chip memories always have high density, but will generate over-constraints for the locality optimization of on-chip accesses. The results of traditional loop transformation may not be optimal when a certain amount of bandwidth needs to be traded for on-chip memory requirement. In our example, T1 shown in Fig1. (d) has smaller on-chip buffer size compared to T0 if we allow two reusable references to stay in off-chip memory.

* In Fig.1(b), bandwidth of *A[i-1,j]* can also be saved using the same reuse buffer, but we do not show it in the figure for the clarification of data reuse from *A[i,j]* to *A[i-2,j]*.

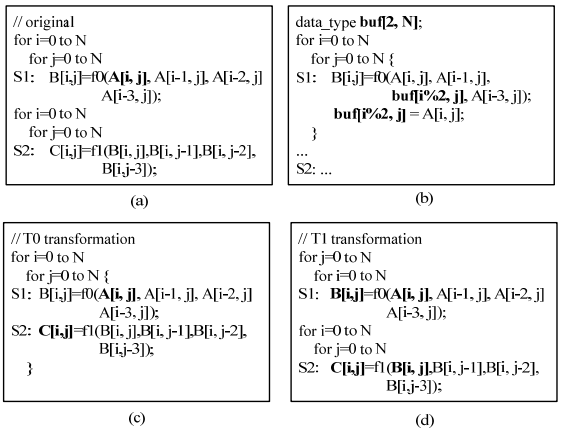

Figure 1. (a) Original code. (b) Buffer allocation. (c) Transformation T0 with a fused loop. (d) Transforamtion T1 with separately optimized loops.

TABLE I. BUFFER SIZES FOR LOOP TRANSFORMATIONS UNDER BANDWIDTH REQUIREMENTS FOR THE EXAMPLE IN FIGURE 1

Given AC	Original	T0	T1
$2N^2$	N^2	**3*N***	N^2
$4N^2$	3*N*	*N*	**6**

Table I shows the comparison of total on-chip buffer size (BS) for two loop transformations (T0 and T1 in Fig. 1) under different off-chip memory bandwidth requirements which are expressed as off-chip access counts (AC). When the given AC is $2N^2$, references *S1:A[i,j]* and *S2:C[i,j]* have to access off-chip memory, so all the reuse buffers are allocated on-chip. T0 has minimal buffer size in this case. When the given AC increases to $4N^2$, we can select two reusable references to be allocated off-chip. For T1, after two largest buffers (*S1:B[i,j]* and *S2:B[i,j]*) are removed, all the data reuse occurs in the inner loops. But for T0, at least one reuse buffer is carried on the outer loop. So T1 has smaller total buffer size than T0 in this case. The detailed explanation of Table I is given in [26].

We can see from Table I that loop transformation is important to improve the final result of memory hierarchy allocation, and inversely the trade-off between buffer size and bandwidth in hierarchy allocation impacts the optimality of the loop transformation. This paper investigates the interactions between loop transformation and hierarchy allocation in imperfectly nested loops, and optimizes these two steps simultaneously to achieve the optimal result.

3. PROBLEM FORMULATION

The challenge of combined loop transformation and memory allocation is the modeling of links between the design space and the overall physical design metrics, such as off-chip memory

bandwidth and on-chip memory utilization. Our problem is specified as follow: Given the high-level program with affine loop bounds and memory accesses, find the optimal loop transformations and two-level memory hierarchy allocation to minimize the on-chip buffer size under a specified off-chip bandwidth constraint. The dual problem, minimizing bandwidth with given buffer size, can be optimized by solving a sequence of the primal problems using binary searching.

3.1 Loop Transformation

We use the polyhedral model [6] to represent the program in the linear form. A program consists of a set of statements $P = \{S_n \mid n = 0..N-1\}$. Each statement describes a set of array references and the computation between these references. A statement in the loops has multiple instances which are indexed by the iteration vector. The iteration vector specifies the iterations of the loops surrounding the statement from the outermost level to the innermost level: $\vec{i}^{\,S} = (i_0, i_1, ..., i_{L_S-1})^T$, where L_S is the number of loops surrounding S. For example, in Fig. 1(d) iteration vectors for S1 and S2 are $\vec{i}^{\,S1} = (j,i)^T$ and $\vec{i}^{\,S2} = (i,j)^T$ respectively. The access instance in a statement can be indexed by the iteration vector of the statement as well.

For a program with affine array accesses, standard data flow analysis [2] can derive the dependence between array references. The set of all the read-after-write dependence is defined as D^{wr}, each element in D^{wr} is a pair of iteration vectors of the dependent access instances. For example, in Fig. 1(a) references *S1:B[i,j]* and *S2:B[i,j-1]* have true data dependence, so all the dependent iteration vector pairs $\{(\vec{i}^{\,S1}, \vec{i}^{\,S2}) \mid \vec{i}^{\,S1} = \vec{i}^{\,S2} - (0,1)^T\}$ are in D^{wr}. We can similarly define data reuse set D^{rr} by considering the read-after-read relations. Other kinds of dependence are not considered because they can be eliminated by renaming techniques [2].

The $2d$+1-dimensional representation $\vec{\phi}$ for affine schedules (d being the maximal loop depth in the program) is widely used to model loop fusion/distribution, code motion and affine loop transformations [27, 28, 11]. The even components of schedule are constants representing positions of the statement at different loop levels, and the odd components are the linear combinations of loop iterators which models the affine transformations. By labeling the position constants as $\vec{c}^{\,S} = (c_0^S, c_1^S, ..., c_d^S)^T$, and the affine transformation matrix as $\mathbf{\Theta}^S = (\vec{\varphi}_0^S, \vec{\varphi}_1^S, ..., \vec{\varphi}_{d-1}^S)^T$, we have $\phi_{2l}^S = c_l^S$ and $\phi_{2l+1}^S = \vec{\varphi}_l^S \cdot \vec{i}^{\,S}$.

Fig. 2 shows the schedule representation of the loop transformations in Fig. 1 (c) and Fig. 1(d). For transformation T0, S1 and S2 has two levels of outer common loops (loop i and j), the first two components of the position vectors are the same ($c_0^{S1} = c_0^{S2} = c_1^{S1} = c_1^{S2} = 0$); within loop j, the relative order of the two statements are determined ($c_2^{S1} = 0, c_2^{S2} = 1$). In T1, S1 and S2 are in the distributed loops at top level, so $c_0^{S1} = 0, c_0^{S2} = 1$. For the odd levels, T0 has the same loop iteration scanning order with the original code, which is iterator i in the outer level and then iterator j in the inner level for both statements. So matrices $\mathbf{\Theta}$ for both statements are identity matrices. But in T1, the loops surrounding S1 are permutated, so $\mathbf{\Theta}^{S1}$ is a permutation matrix and $\phi_1^{S1} = j, \phi_3^{S1} = i$.

This schedule representation can model all the combinations of affine loop transformations, fusion/distribution, and code motion.

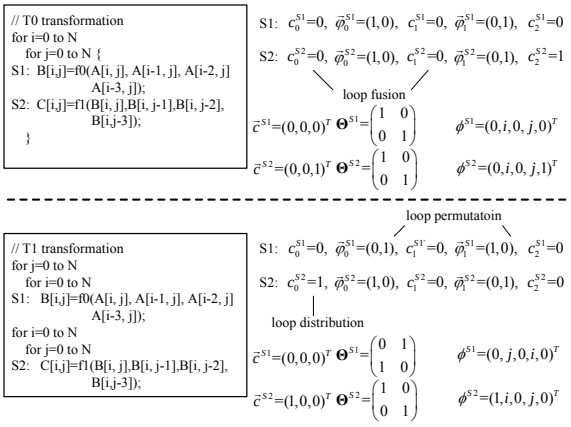

Figure 2. Loop transformations and interleaved schedule vectors.

And legality condition of a loop transformation can be expressed as $\forall(\vec{i}^{\,x}, \vec{i}^{\,y}) \in D^{wr}, \vec{\phi}^{\,y}(\vec{i}^{\,y}) \succ_{lex} \vec{\phi}^{\,x}(\vec{i}^{\,x})$, which preserves the order of the dependent statement instances after transformations. The proof of the expressiveness and legality condition of loop transformations are provided in [26].

3.2 Reuse Graph and Hierarchy Allocation

Data reuse graph (DRG) is widely used to represent data reuse candidates [14, 17, 18] as Fig. 3(a). Nodes of the graph are array references, and edges are the data reuse between the nodes. Nodes are weighted by the access count (AC) of the reference, and edges are weighted by the reuse buffer size (BS).

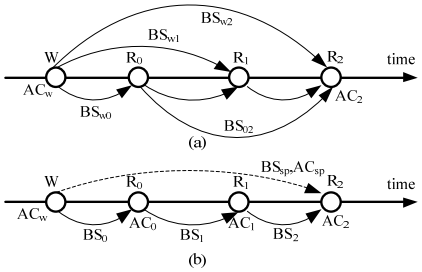

Figure 3. (a) Full data reuse graph. (b) Simplified data reuse graph.

We extend the standard DRG in the following two aspects. First, while traditional DRG only considers read nodes, we also model write nodes in DRG. Thus, it is possible to save the bandwidth of the first read node (in access order), because it can reuse data from the write node. And bandwidth of the write node can even be saved if all the read nodes of the same array are reused and the data is not the primary output of the design. Second, we simplify the DRG by pruning sub-optimal buffer allocations. In most affine programs, the reuse distance [16] for each array reference is a constant, or can be converted to a constant by array partitioning. By ignoring the boundary data elements, we can assume that all the data of each node are reused as a whole. For each node, same AC saving can be obtained by data reuse from different nodes. So we only consider the reuse from the nearest neighboring node as Fig. 3(b), which has the minimal buffer size. The hierarchy allocation is modeled as binary variables $\{b_y\}$ where b_y indicates whether node y is reused from its nearest neighbor. To model the case where write accesses are saved, we introduce a special edge as dashed line in Fig. 3(b) weighted by the total AC and BS of the array. If the special edge b_w^{γ} of array γ is allocated, no other edges (b_x) of the same

array γ need to be allocated because those buffers are already allocated by the special edge. Using the simplified DRG and binary variables, we can model and evaluate each hierarchy allocation candidate:

$$BS = \sum_y b_y BS_y, \quad AC = AC_{total} - \sum_y b_y AC_y$$
$$\forall x \in R_\gamma, b_w^\gamma \cdot b_x = 0$$

where constant AC_{total} is the total AC without data reuse, and R_γ is the set of read references to array γ.

3.3 Reuse Buffer Size Calculation

As shown in the motivation example, cyclic reuse buffers are used to reduce the buffer size. But seldom previous work has addressed the analytic calculation of cyclic reuse buffer size in imperfectly nested loops. Furthermore, various loop transformations make it harder to calculate the accurate buffer size.

We analytically calculate the size of transformed reuse buffer for node *y* in three steps. First, determine the access order of the nodes, and find the nearest neighbor reference *x* which has the same array with y and accesses before node *y*. Second, calculate the reuse distance between nodes *x* and *y* using standard data flow analysis [2], and get the loop level carrying the reuse (l_r) and the distance carried on the loop level (srd_{xy}). Third, calculate the number of array elements accessed by *y* in one iteration of level l_r (notated as $Q(l_r)$), and then the buffer size is $srd_{xy} \times Q(l_r)$.

For example we calculate the buffer size for reference S1:A[i-2,j] in Fig 1(c). The nearest neighbor node is S1:A[i-1,j]. By data flow analysis, we can know the reuse between the two nodes is carried at the outer loop *i*, and the distance at this level is 1. Within each iteration of *i*, there are *N* data accessed by both references, so the total buffer size is $1 \times N = N$. The analytic calculation of $Q(l_r)$ is performed by building linear constraints for the accessed data elements and counting the integer points in the polytope by the Barvinok library [29]. The detailed derivation is given in [26].

3.4 Formulated Optimization Problem

From the discussion above, we can summarize our formulation as Problem 1 stated below. Eqn. 1 and Eqn. 2 sum up the total BS and AC where $D = D^{wr} \bigcup D^{rr}$. Eqn. 3 and Eqn. 4 model legal loop transformations. Eqn. 5 defines the constraints of allocation variables for the special edges in Section 3.2. Finally, Eqn. 6 calculates the buffer sizes where analytic form of $Q_{l_r}(\vec{\phi}^x, \vec{\phi}^y)$ is given in [26].

PROBLEM 1. *Given an affine program P with array accesses, and a bound of off-chip access count, find the optimal loop transformation* $(\vec{c}, \mathbf{\Theta})$ *for each statement and memory hierarchy allocation* b_y *for each reuse edge to*

$$\text{Minimize} \quad BS = \sum_{y \in D} b_y BS_y(\vec{\phi}^x, \vec{\phi}^y) \tag{1}$$
$$\text{Subject to} \quad AC = AC_{total} - \sum_{y \in D} b_y AC_y \le AC_{required} \tag{2}$$
$$\phi_{2l}^y = c_l^y, \phi_{2l+1}^y = \vec{\varphi}_l^y \cdot \vec{i}^y, l = 0..L-1 \tag{3}$$
$$\forall (\vec{i}^x, \vec{i}^y) \in D^{wr}, \vec{\phi}^y \succ_{lex} \vec{\phi}^x \tag{4}$$
$$\forall x \in R_\gamma, b_w^\gamma \cdot b_x = 0 \tag{5}$$
$$BS_y(\vec{\phi}^x, \vec{\phi}^y) = Q_{l_r}(\vec{\phi}^x, \vec{\phi}^y) \cdot srd_y(\vec{\phi}^x, \vec{\phi}^y) \tag{6}$$

4. EFFICIENT SOLUTION

The buffer size calculation is non-convex, which means a enumeration-based approach as used in [25] is needed. But the design space pruning in [25] is no longer applicable because of the much larger design space for imperfectly nested loops.

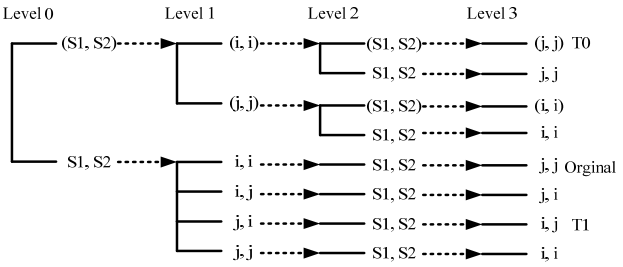

Figure 4. Enumeration tree of loop transformation.

4.1 Enumeration with Branch-and-Bound

Instead of enumerating all legal transformations in a brute-force way, the features of buffer allocation are used to prune sub-optimal partial transformations to greatly reduce the computation complexity of finding optimal solutions for large designs.

Fig. 4 shows the enumeration tree of the motivation example in Fig. 1. For each even level *l*, the statements are partitioned into ordered groups where c_l indicates the group index of each statement. For the loop levels, we limit the matrix coefficients $\vec{\varphi}_l^S$ as -1, 0 or 1. These matrices can model the most useful affine transformations such as permutation, reversal and skewing as in [25]. Illegal branches are pruned according to Eqn. 4.

Branch-and-bound approach was adopted in [27] to find good loop transformations for parallelism, but they did not consider memory hierarchy allocation. We propose an efficient branch-and-bound scheme for our problem considering interaction between partial schedule and minimal buffer size. For each loop transformation candidates $(\vec{c}, \mathbf{\Theta})$, we can calculate $\vec{\phi}^x$ for each statement to eliminate Eqn. 3, Eqn. 4 and Eqn. 6. The remaining problem is optimized the hierarchy allocation to minimize total BS subjects to AC constraints. Because a reuse candidate carried on an outer loop will generate a much larger buffer than that carried on an inner loop. We travel the enumeration tree branch by branch from outer levels to inner levels. For a branch *B*, we calculate the lower and upper bounds of the minimal total buffer size for the branch according to the partial transformation of the outer loop levels. The determined buffer set D_B^d contains all the edges carried on the outer levels that are determined by the branch, and non-determined buffer set D_B^n contains the remaining edges. The lower bound (LB_B) of the buffer size at branch *B* is calculated by optimizing the allocation of buffers in D_B^d and assuming the bandwidth of D_B^n are saved without BS cost. The upper bound (UB_B) of buffer size at branch *B* is calculated by adding the maximal possible sizes of all the buffers in D_B^n into LB_B. We can prove that the BS range of a child branch is always covered by that of its parent branch: $LB_{Parent} \le LB_{Child} \le UB_{Child} \le UB_{Parent}$. If two branches have non-overlapped BS ranges, we can prune the branch with larger BS and all its sub-branches without losing optimality.

4.2 Knapsack Reuse Technique

In addition to pruning the searching branches, we also reduce the complexity of hierarchy allocation for each branch. In general, the hierarchy allocation problem can be solved by integer linear

programming with an exponential computation complexity. But we observe that hierarchy allocation problem consisting of Eqn. 1, Eqn. 2 and Eqn. 5 can be converted into an extended knapsack problem. Each reuse buffer is an item to be put into the knapsack. BS_y and AC_y are the value and weight of item y respectively, and $t_y = 1 - b_y$ indicates whether item y is taken into the knapsack.

If we first ignore the constraint of Eqn. 5, the standard knapsack problem can be solved by dynamic programming. Let $BS[i, AC]$ be the maximum value that can be attained with weight no more than AC using up to first i items, and we have

$$BS[i, AC] = max(BS[i-1, AC], BS[i-1, AC - AC_i] + BS_i) .$$

The complexity of the dynamic programming is O(mn), where m is number of reuse buffers, and n is total AC which can be greatly reduced by normalizing the AC of reuse buffers to the loop iteration count.

The extended knapsack problem cannot be directly solved in polynomial time because the solution of the sub-problem is dependent on the decisions of its super-problem. Fortunately, the original dynamic programming does not specify the order of the items, and we can make the sub-problem independent by reordering the items. [26] shows the details of the reordering approach and proves that the complexity remains O(mn) for the extended knapsack problem.

Another feature of the hierarchy allocation problem is that the determined reuse buffer set of the child branch will always contain that of the parent branch, which means the intermediate knapsack results of the parent branch can be reused by its child branches. By reusing the knapsack results, the average computation complexity can be reduced to O(Δmn), where Δm is number of newly determined reuse buffers in the current branch, and n is the total AC. The extended knapsack problem can also reuse intermediate data from parent branches [26]. The reordering process will invalidate the intermediate results because the first i items are changed after reordering. But this kind of computation complexity overhead is small because only one reordering process is needed for each array, and only a part of the intermediate results are invalidated.

5. EXPERIMENTAL RESULTS

Our memory hierarchy optimization algorithm is performed as a source-to-source preprocessing step to a high-level synthesis tool. 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 using the ROSE compiler infrastructure [30]. The core optimizer finds the optimal loop transformation and on-chip buffer allocation. In the code generator, loop transformation is performed by ClooG [31], and the on-chip buffer is generated by the ROSE infrastructure. The optimized loop kernels are synthesized into VHDLs and then circuit netlists by the high-level synthesis tool AutoPilot [32, 33] and the FPGA implementation tool Xilinx ISE [34]. Our test designs include a set of real-life data-intensive loop kernels: FDTD and JACOBI are stencil codes chosen from polybench 3.0 [35]; DENOISE smooths a 3D image by averaging 13 neighboring pixels [36]; REG is one of the major parts of a 3D medical image registration algorithm [36], and SEG is a two-phase image segmentation algorithm [36]. We include the whole programs of the benchmarks with multiple loop nests, instead of only one main loop nest in [25]. The proposed combined loop transformation (LT) and memory hierarchy allocation (HA) scheme for imperfectly nested loops (LTHA-INL) is compared with two reference points in our experiments. The first reference point is the combined LT and HA scheme for perfectly nested loops (LTHA-PNL) in [25], in which the loop nests are optimized independently. And the second point is the separate LT and HA scheme (LT+HA) that was done in [11, 19].

Experimental results of the three approaches are reported in Table II. The second column (*AC*) shows the normalized access count per loop iteration for each design, which is calculated from the given bandwidth and performance requirement. We set the clock frequency as 10ns, and all design implementations satisfy the timing constraint. The FPGA implementation results of the three approaches in the Xilinx Virtex-6 xc6vlx365t platform are compared, such as the utilization of logic slice and on-chip Block RAM (BRAM), the execution latency in cycles, and the power consumption in mW. We also list the runtime (in seconds) of our proposed algorithm in the last column. We normalize the four metrics to the values of the LTHA-PNL scheme, and calculate the geometric mean of the normalized data in the last row of Table II.

From the results, it is clear that loop fusion/distribution and code motion are important to the results of hierarchy allocation. Compared to the only LTHA-PNL scheme, the separated LT+HA scheme can save the on-chip memory size by 60%. And our LTHA-INL scheme gains an additional 40% memory reduction and 19% power saving compared to the separated LT+HA scheme. In some cases, for example JACOBI_3D, LT+HA get worse results than LTHA-PNL because the original distributed loops are good enough for relative sufficient bandwidth. And in other cases such as FDTD_2D, LTHA-PNL has a much large buffer size because of the data reuse between different distributed loop nests. To isolate the impact of performance, we keep the initiation interval (II) unchanged for the loop pipelining in high-level synthesis. The II of a fused loop is set as the sum of the II of the original loops. The logic slice saving mainly comes from the better resource and data sharing after loop fusion. By our efficient pruning techniques, over 100x speed-up is achieved, and the overall execution time is within several seconds for the real-life benchmarks with four levels of loops and tens of array references.

TABLE II. EXPERIMENT RESULTS

Design	*AC*	LTHA-PNL [25]				Separated LT+HA [11]+[19]				Proposed LTHA-INL				
		Slice (#)	BRAM (#)	Latency (cycle)	Power (mW)	Slice (#)	BRAM (#)	Latency (cycle)	Power (mW)	Slice (#)	BRAM (#)	Latency (cycle)	Power (mW)	Runtime (s)
JACOBI_3D	3	487	203	4.86E+8	133	504	347	4.86E+8	163	378	203	4.86E+8	112	5.52
JACOBI_4D	3	-	6054*	1.22E+11	-	548	757	1.22E+11	244	520	427	1.22E+11	179	0.28
FDTD_2D	4	398	422	6.08E+6	148	281	18	6.08E+6	57	271	10	6.08E+6	56	0.31
FDTD_3D	6	-	1218*	3.64E+8	-	390	406	3.64E+8	150	378	204	3.64E+8	112	1.34
DENOISE	4	1074	609	2.62E+8	325	836	407	2.62E+8	240	836	306	2.62E+8	221	0.32
REG	12	1569	610	1.22E+9	645	1209	636	1.22E+9	582	1170	306	1.22E+9	393	0.14
SEG	10	-	882*	8.54E+8	-	932	714	8.54E+8	315	932	586	8.54E+8	291	2.50
Geomean		1.00	1.00	1.00	1.00	0.44	0.40	1.00	0.74	0.41	0.24	1.00	0.60	

*The maximum number of BRAMs in the experimental FPGA is 832, so there are no final implementation results for these cases.

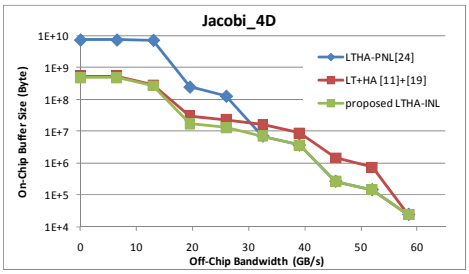

Figure 5. Design space exploration.

Fig. 5 investigates the trade-off between bandwidth and buffer size. When the bandwidth is low, LTHA-PNL is not good because loop fusion is needed to optimize large buffers. When the bandwidth is high, separate LT+HA performs poorly because it has over-constraints for small buffers. However, Our LTHA-INL can consistently attain optimal solutions in all cases.

6. CONCLUSION

This paper presents the first in-depth study that combines loop transformation and hierarchy allocation for imperfectly nested loops. It gains 40% reduction of the on-chip buffer usage under the same off-chip bandwidth constraint with no performance overhead. The proposed space pruning techniques are shown to be highly effective to speed up the execution of our algorithm. Our future work will integrate loop tiling transformation and tiling size selection into our memory hierarchy optimization.

7. REFERENCES

[1] F. Catthoor, E. d. Greef, and S. Suytack, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[2] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[3] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, pp. 424–453, July 1996.

[4] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in PLDI '91. New York, NY, USA.

[5] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 452–471, Oct. 1991.

[6] P. Feautrier, "Some efficient solutions to the affine scheduling problem: Part II. multidimensional time," *International Journal of Parallel Programming*, vol. 21, pp. 389–420, 1992.

[7] A. W. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in ICS '99. New York, Aug. 1999.

[8] N. Ahmed, N. Mateev, and K. Pingali, "Synthesizing transformations for locality enhancement of imperfectly-nested loop nests," *IJPP*, vol. 29, pp. 493–544, Oct. 2001.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI '08*, pp. 101–113, New York, NY, USA,.

[10] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *PACT '10*, pp. 343–352, Sept. 2010,.

[11] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. "Loop transformations: convexity, pruning and optimization," in *POPL '11*, pp. 549-562, Jan. 2011.

[12] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. v. Achteren, and T. Omnes, *Data access and storage management for embedded programmable processors*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[13] T. Van Achteren, G. Deconinck, F. Catthoor, and R. Lauwereins, "Data reuse exploration techniques for loop-dominated applications," in *DATE'02*, pp. 428–435, Mar. 2002.

[14] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Des. Autom. Electron. Syst.*, April 2007.

[15] S. Wuytack, J.-P. Diguet, F. V. M. Catthoor, and H. J. De Man, "Formalized methodology for data reuse: exploration for low-power hierarchical memory mappings," *IEEE Trans. on VLSI*, 1998.

[16] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *DAC'02*, pp. 628—633, June 2002.

[17] E. Brockmeyer, M. Miranda, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organisations," in *DATE'03*, pp. 1070–1075, Mar. 2003.

[18] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching scheme for scratchpad memory," in *DAC'11*, pp. 960–965, 2011.

[19] I. Issenin, E. Brockmeyer, B. Durinck, and N. D. Dutt, "Data-reuse-driven energy-aware cosynthesis of scratch pad memory and hierarchical bus-based communication architecture for multiprocessor streaming applications," *IEEE trans. on CAD*, vol. 27, no. 8, pp. 1439–1452, 2008.

[20] M. Palkovic, F. Catthoor, and H. Corporaal, "Trade-offs in loop transformations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 22:1–22:30, April 2009.

[21] P. R. Panda, N. D. Dutt, and A. Nicolau, "Local memory exploration and optimization in embedded systems," *IEEE Trans. on CAD*, vol. 18, no. 1, pp. 3–13, 1999.

[22] Q. Hu, P. G. Kjeldsberg, A. Vandecappelle, M. Palkovic, and F. Catthoor, "Incremental hierarchical memory size estimation for steering of loop transformations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, September 2007.

[23] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in PPoPP'08, pp. 1–10, Feb. 2008.

[24] Q. Liu, G. A. Constantinides, K. Masselos, and P. Cheung, "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework," *IEEE Trans. on CAD*, vol. 28, no. 3, 2009.

[25] J. Cong, P. Zhang, and Y. Zou, "Combined loop transformation and hierarchy allocation in data reuse optimization," in *ICCAD'11*, pp. 185–192, Nov. 2011.

[26] J. Cong, P. Zhang, and Y. Zou, "Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis," *Technical Report, Computer Science Department, UCLA*, TR200019, 2012.

[27] Kelly, W. A. *Optimization within a Unified Transformation Framework*, Ph.D. dissertation, University of Maryland, 1996.

[28] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies," *IJPP*, vol. 34, no. 3, June 2006.

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

[30] ROSE compiler infrastructure. http://rosecompiler.org/

[31] The ClooG code generator. http://www.cloog.org/

[32] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGA: From prototyping to deployment," *IEEE Trans. on CAD*, vol. 30, no. 4, 2011.

[33] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A Platform-Based ESL Synthesis System," *High-Level Synthesis: From Algorithm to Digital Circuit*, ed. P. Coussy and A. Morawiec, Springer Publishers, 2008.

[34] Xilinx ISE Design Suite. http://www.xilinx.com/products/design-tools/ise-design-suite/.

[35] Polyhedral benchmark suite v3.1. http://www.cse.ohio-state.edu/~pouchet/software/polybench/ .

[36] J. Cong, V. Sarkar, G. Reinman and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test of Computers*, vol. 28, no. 2, pp. 5-15, March/April 2011.