Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism

Duo Liu[†], Zili Shao[†], Meng Wang[†], Minyi Guo[‡] and Jingling Xue[§]

[†]Dept. of Computing
The Hong Kong Polytechnic Univ.
Hung Hom, Hong Kong
cszlshao@comp.polyu.edu.hk

[‡]Dept. of Comp. Sci. & Eng.
Shanghai Jiao Tong Univ.
Shanghai, P. R. China
guo-my@cs.sjtu.edu.cn

[§]School of Comp. Sci. & Eng.
The Univ. of New South Wales
Sydney, NSW 2052, Australia
jingling@cse.unsw.edu.au

ABSTRACT

This paper solves the open problem of extracting the maximal number of iterations from a loop that can be executed in parallel on chip multiprocessors. Our algorithm solves it optimally by migrating the weights of parallelism-inhibiting dependences on dependence cycles in two phases. First, we model dependence migration with retiming and formulate this classic loop parallelization into a graph optimization problem, i.e., one of finding retiming values for its nodes so that the minimum non-zero edge weight in the graph is maximized. We present our algorithm in three stages with each being built incrementally on the preceding one. Second, the optimal code for a loop is generated from the retimed graph of the loop found in the first phase. We demonstrate the effectiveness of our optimal algorithm by comparing with a number of representative non-optimal algorithms using a set of benchmarks frequently used in prior work.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Performance, Languages

Keywords

Loop parallelization, loop transformation, retiming, data dependence graph, iteration-level parallelism

1. INTRODUCTION

Chip multiprocessors, such as Intel Dual-Core processors, AMD Phenom processors, IBM Cell processors, ARM11 MPCore processors, TI TMS320DM6467 DaVinci processors and Freescale MSC812x Quad-Core DSP processors, are widely used in both general-purpose and embedded computing. The importance of harnessing parallelism in programs to fully utilize their computation power cannot be over-emphasized. While programmers can utilize multiple-threaded application development environments to generate coarse-grain parallel programs with thread-level parallelization in practice [29], loop parallelization at the granularities of loop iterations is generally too hard to be done manually. A lot of automatic loop parallelization techniques have been developed for parallel/vector compilers in the previous work [1, 31, 2, 17, 34, 32, 33]. Based on data dependence analysis, various techniques, such as scalar renaming, scalar expansion, scalar forward-substitution, dead code elimination and data dependence elimination, have been proposed [2, 4, 18]. Most of these techniques, however, focus on instruction-level parallelism. In this paper, we propose an iteration-level loop parallelization technique that supplements this previous work by enhancing loop parallelism. We target at iteration-level parallelism [3] by which different iterations from the same loop kernel can be executed in parallel.

At the iteration level, based upon the degree of parallelism, loops can be mainly classified into three categories: serial loops (DOSER), parallel loops (DOALL) [13], and partially parallel loops (DOACR) [10]. Without any loop transformations, all iterations in a DOSER loop must be executed sequentially due to the dependences between successive iterations. For a DOALL loop, all its iterations can be executed in parallel since it exhibits no inter-iteration dependences. In the case of a DOACR loop, its successive iterations can be partially overlapped because of inter-iteration data dependences. In this paper, we focus on maximizing loop parallelism for DOSER and DOACR loops. The main obstacle to their parallelization lies in the presence of dependence cycles, a dependence relation in a set of statements to which the statements are strongly connected via dependence relations [9].

There have been numerous studies to enhance loop parallelism by exploiting data dependences of dependence cycles [20, 23, 24, 2, 30, 4]. In [20], a partitioning technique is proposed to group all iterations of a loop together to form a dependence chain if the greatest common divisor of their dependence distances is larger than one. In [23, 24], cycle-shrinking groups consecutive dependence-free iterations to form the innermost loop kernel of a new set of nested loops. In [27], cycle-shrinking is generalized to multi-dimensional loops. In [2], node splitting is used to eliminate anti- or output-dependences by adding new copy statements. In [4], node splitting is extended to nested loops. In [30], cycle breaking is used to partition a loop into a series of small loops. Unlike the previous work, this work applies loop transformation to change inter-iteration data dependences so that better loop parallelism can be achieved. In

CASES'09, October 11–16, 2009, Grenoble, France.

fact, optimal parallelism is guaranteed by the proposed technique in this paper.

In our proposed technique, loop transformation is modeled by retiming [15, 22]. Retiming is originally proposed to minimize the cycle period of a synchronous circuit by evenly distributing registers. It has been extended to schedule data flow graphs on parallel systems in [7, 6, 21]. For loop transformation, retiming is used under the name of "index shift method" for parallelizing nested loops [16]. In [25], a loop optimization method is proposed to optimize nested loops by combining the index shift method [16] and the generalized cycle shrinking [27]. In [26], retiming is applied in loop transformation for nested loop fusion. In [11], a loop transformation technique is proposed in an attempt to fully parallelize an inner loop through retiming an outer loop. Most of the above work focuses on instruction-level parallelism and on loop parallelism. However, none considers DOACR parallelism. To our knowledge, this work is the first to optimally solve the iteration-level loop parallelization problem with dependence migration modeled by retiming.

In this paper, we propose an optimal iteration-level loop parallelization technique with loop transformation to maximize loop parallelism. Our basic idea is to migrate inter-iteration data dependences by regrouping statements of a loop kernel in such a way that the number of consecutive independent iterations is always maximized. We construct a dependence graph to model the data dependences among the statements in a loop and then use retiming to model dependence migration among the edges in the dependence graph. As a result, this classic loop optimization problem is transformed into a graph optimization problem, i.e., one of finding retiming values for its nodes so that the minimum non-zero edge weight in the graph is maximized. To solve the graph optimization problem incrementally, we classify a dependence graph into one of the three types: a DAG (Directed Acyclic Graph), a Cyclic Graph with Single Cycle (CGSC), and a Cyclic Graph with Multiple Cycles (CGMC). This allows us to present our technique in three stages. For DAGs and CGSCs, we give two polynomial algorithms to find their optimal solutions, respectively. For CGMCs, we find their optimal solutions based on an integer linear programming formulation, which can be solved efficiently for the dependence graphs found in real code. Finally, we give a loop transformation algorithm for generating the optimized code for a given loop, including its prologue, loop kernel, and epilogue based on the retiming values for the loop.

This paper makes the following contributions:

- We present for the first time an optimal loop parallelization technique for maximizing the number of concurrently executed loop iterations in a DOACR loop.
- We demonstrate the effectiveness of our technique by comparing with a number of representative (non-optimal) techniques using a set of benchmarks that are frequently used in prior work.

The rest of this paper is organized as follows. Section 2 presents some basic concepts about dependence graphs and retiming, formalizes the problem addressed, and gives an overview of our optimal loop parallelization technique. In Section 3, we present our technique incrementally by considering three different types of dependence graphs and presenting three algorithms to find their optimal retiming functions, with each being built on the preceding one. In Section 4, we give an algorithm for generating the optimal code for a loop based on a retiming function. Section 5 evaluates and analyzes the proposed technique against existing loop parallelization techniques. Section 6 concludes and discusses future work.

2. BASIC CONCEPTS AND MODELS

In this section, we introduce basic concepts and models that are used in the later sections. We introduce the notion of dependence graph in Section 2.1. In Section 2.2, we examine briefly how to use retiming to model dependence migration among the edges in a dependence graph. We include a brief discussion on iteration-level parallelism in Section 2.3. Finally, we define the problem addressed in Section 2.4.

2.1 Dependence Graph

Given a loop, its *dependence graph* $G = (V, E, w)$ is an edge-weighted directed graph, where V is the set of nodes with each node representing a statement in the loop, $E = \{(u, v) : u \rightarrow v \in V\}$ is the edge set that defines the dependence relations for all nodes in V with (u, v) denoting the edge from node u to node v, and $w : E \mapsto \mathbb{Z}$ is a function that associates every edge $(u, v) \in E$ with a nonnegative weight known as its *dependence distance*. By convention, an edge (u, v) represents an *intra-iteration dependence* if $w(u, v) = 0$ and an *inter-iteration dependence* otherwise (i.e., if $w(u, v) > 0$). In either case, $w(u, v)$ represents the number of iterations involved. These two kinds of dependence are further explained as follows:

- Intra-iteration dependence $w(u, v) = 0$. Such a dependence occurs in the same iteration between a pair of statements. If there exists an intra-iteration dependence between two statements u and v within the same iteration, then statement v reads the results generated by statement u.
- Inter-iteration dependence $w(u, v) > 0$. Such a dependence occurs when two statements from different iterations are dependent on each other. If there exists an inter-iteration dependence between u and v, then the execution of statement v in iteration $i + w(u, v)$ reads the results generated by u in iteration i. Thus, the earliest iteration in which v can be executed is $w(u, v)$ iterations later than the iteration in which u is executed.

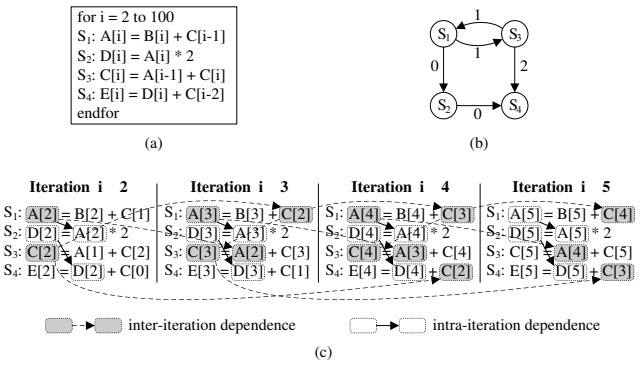

Figure 1: A loop kernel from [14]. (a) Loop kernel. (b) Dependence graph. (c) Intra- and inter-iteration dependences.

We use a real loop application from [14] to show how to use a dependence graph to model a loop. The loop kernel is shown in Figure 1(a) and its corresponding dependence graph in Figure 1(b). This loop has both intra-iteration and inter-iteration dependences. For example, the weight of (S_1, S_2) is zero, indicating an intra-iteration dependence between S_1 and S_2. The weight of (S_3, S_4) is 2, indicating an inter-iteration dependence between S_3 and S_4 with a distance of 2.

Figure 1(c) illustrates how iteration-level loop parallelism can be constrained by dependences with the first four iterations shown. Let us examine intra-iteration dependences first. In each iteration, S_2 must be executed after S_1 since $A[i]$ read by S_2 should be written by S_1 first. In addition, S_2 and S_4 also have an intra-iteration dependence (due to $D[i]$). In general, intra-iteration dependences are confined to the same iteration and thus do not inhibit iteration-level parallelism. Let us next examine inter-iteration dependences in the loop. S_4 in iteration 4 reads $C[2]$. Nevertheless, according to the execution order of loop iterations, $C[2]$ should be written first by statement S_3 in iteration 2. Thus, S_4 in iteration i can only be executed until after S_3 in iteration $i-2$ has been executed. Likewise, S_3 in iteration i can only be executed until after S_1 in iteration $i-1$ has been executed. As a result, we cannot execute more than each iteration in parallel since every iteration requires results from the preceding two iterations. Hence, inter-iteration dependences are the major obstacle to iteration-level parallelism.

2.2 Retiming and Dependence Migration

Retiming [15] is used to model dependence migration, and it is defined as follows.

DEFINITION 2.1. *Given a dependence graph $G = (V, E, w)$,* a retiming r of G *is a function that maps each node in V to an integer $r(v)$. For a node $u \in V$, the retiming value $r(u)$ is the number of dependence distances (edge weights) drawn from each of its incoming edges and pushed to each of its outgoing edges. Given a retiming function r, let $G_r = \langle V, E, w_r \rangle$ be the retimed graph of G obtained by applying r to G. Then $w_r(u, v) = w(u, v) + r(u) - r(v)$ for each edge $(u, v) \in E$ in G_r.*

As defined above, by retiming a node, dependences are moved from its incoming edges to its outgoing edges; thus, dependence relations can be changed. On the other hand, a retiming function can be directly mapped to a loop transformation by which we can obtain a new loop that has the corresponding dependence relations. How to perform this mapping is discussed in detail in Section 4. As retiming can be directly mapped to loop transformation, a retiming function must be legal in order to preserve the semantic correctness of the original loop. A retiming function r is *legal* if the retimed weights of all edges in the retimed graph G_r are non-negative. An illegal retiming function occurs when one of the retimed edge weights becomes negative, and this situation implies a reference to non-available data from a future iteration. If G_r is a retimed graph of G derived by a legal retiming function, then G_r is functionally equivalent to G [15].

For simplicity, we normalize a retiming r such that the minimum retiming value(s) is always zero [5]. A retiming function r can be normalized by subtracting $\min_v r(v)$ from $r(v)$ for every v in V.

As an example shown in Figure 2, the retiming value $r(S_3) = 1$ conveys that one unit of dependence distance is drawn from the incoming edge of node S_3, $S_1 \rightarrow S_3$, and pushed to both of its outgoing edges, $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_4$. Therefore, by applying $r(S_3) = 1$, the execution of S_3 is moved forward, and correspondingly, the original inter-iteration dependence between S_1 and S_3 is transformed into an intra-iteration dependence. Figure 2(c) shows the new loop kernel obtained based on this retiming function. Figure 2(d) illustrates the iteration-level parallelism in the new kernel, indicating that its two consecutive iterations can be executed in parallel since they are now independent.

2.3 Iteration-Level Parallelism

Iteration-level parallelism is achieved when different iterations from a loop can be executed in parallel. However, loop iterations

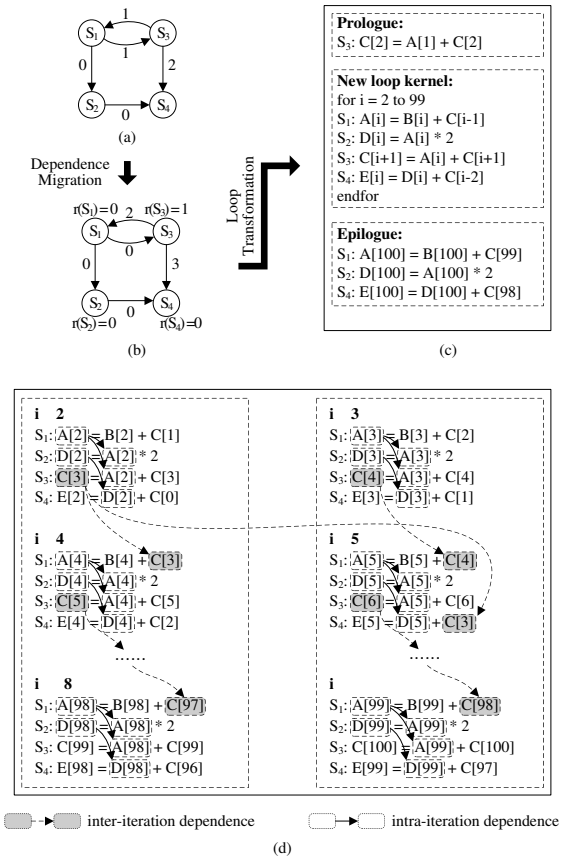

Figure 2: Loop transformation for Figure 1(b). (a) Original dependence graph (with the minimum non-zero edge weight being 1). (b) Transformed dependence graph with the minimum non-zero edge weight being 2. (c) New loop kernel after loop transformation. (d) A graphical representation of parallelism in the transformed loop.

must be executed in accordance with their inter-iteration dependences. Thus, inter-iteration dependences inhibit iteration parallelization. For example, in Figure 1(c), S_3 and S_4 have an inter-iteration dependence with a distance of 2. As a result, the i-th and $(i-2)$-th iterations cannot be executed in parallel. Moreover, every two consecutive iterations cannot be executed in parallel either as there is also an inter-iteration dependence between S_1 and S_3 with a distance of 1. Therefore, the minimum inter-iteration dependence distance in a loop (i.e., the minimum non-zero edge weight in its dependence graph) bounds the amount of parallelism exploitable in the loop from above.

The focus of this work is on maximizing the minimum non-zero edge weight with dependence migration and loop transformation. Given a dependence graph for a loop, its minimum non-zero edge weight, β, represents the parallelism degree of the loop, which implies the absence of inter-iteration dependences within β consecutive iterations. We say that this loop is *β-parallelizable*. If the loop can be fully parallelized, it is said to be *fully-parallelizable*. For example, the loop in Figure 2 is *2-parallelizable*, which can be obtained from the transformed dependence graph.

2.4 Problem Statement

For a dependence graph used to model a given loop, the problem of performing optimally iteration-level loop parallelization is

defined as follows:

Given a dependence graph $G = (V, E, w)$ of a loop, find a retiming function r of G such that the minimum non-zero edge weight β of the transformed dependence graph $G_r = \langle V, E, w_r \rangle$ is maximized.

Existing solutions [20, 23, 24, 2, 30, 4] to this problem are all approximate for DOACR loops. Our solution, as outlined in Figure 3, solves the problem optimally (for the first time) in two phases. In the first phase, we introduce a *dependence migration algorithm* to find a retiming function for a given dependence graph such that β in the graph is maximized. In the second phase, we apply a *loop transformation algorithm* to generate the optimal code for the given loop based on the retiming function found.

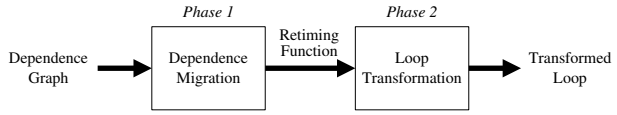

Figure 3: Optimal retiming-based loop parallelization.

3. DEPENDENCE MIGRATION ALGORITHM

In this section, we introduce our dependence migration algorithm (DMA) given in Algorithm 3.1 to find a retiming function for a given dependence graph so that the minimum non-zero edge weight in the retimed graph is maximized. For efficiency reasons and also to ease understanding, DMA finds an optimal retiming function by performing a case analysis based on the structure of the dependence graph.

Algorithm 3.1 DMA(G, L_G, α)

Input: A dependence graph $G = (V, E, w)$ of a loop L_G and a positive integer α.
Output: A retiming function of G.

1: $SNode\_SCC \leftarrow 0$; $MNode\_SCC \leftarrow 0$; $SCycle\_Flag \leftarrow 0$.
2: Let SCC_Num be the number of SCCs found in G.
3: **for** each SCC **do**
4: Let N_Num (E_Num) be its node (edge) count.
5: **if** $N\_Num == 1$ **then** $SNode\_SCC$ ++.
6: **if** $N\_Num > 1$ **then** $MNode\_SCC$ ++.
7: **if** $E\_Num == N\_Num$ **then**
8: $SCycle\_Flag \leftarrow 1$.
9: Let C_G be the cycle (V_{SCC}, E_{SCC}, w).
10: **end if**
11: **end for**
12:
13: **if** $SNode\_SCC == SCC\_Num$ **then**
14: DAG_Migration(G, α)
15: **else if** $MNode\_SCC == 1$ && $SCycle\_Flag == 1$ **then**
16: Single_Cycle_Migration(G, C_G).
17: **else**
18: Multi_Cycle_Migration(G).
19: **end if**

We classify a dependence graph into one of the three types: a DAG (Directed Acyclic Graph), a Cyclic Graph with Single Cycle (CGSC) and a Cyclic Graph with Multiple Cycles (CGMC), based on the structure of the SCCs (Strongly Connected Components) in the graph. For DAGs and CGSCs, two algorithms are given to find their optimal retiming functions polynomially. For CGMCs, an optimal algorithm is given based on an integer linear programming formulation. These three different types of dependence graphs, as illustrated in Figure 4, are classified as follows:

- **DAGs.** If every SCC in G has one node only, then G is a DAG. In this case, DAG_Migration is invoked to retime G. An example DAG with four singleton SCCs are shown in Figure 4(a).
- **CGSCs.** In such a graph G, one SCC is actually a single cycle and each other SCC has one node only. In this case, Single_Cycle_Migration is invoked to retime G. An example is given in Figure 4(b).
- **CGMCs.** In this general case, every graph G has more than one cycle. Multi_Cycle_Migration comes into play to retime G. Figure 4(c) gives a dependence graph with multiple cycles.

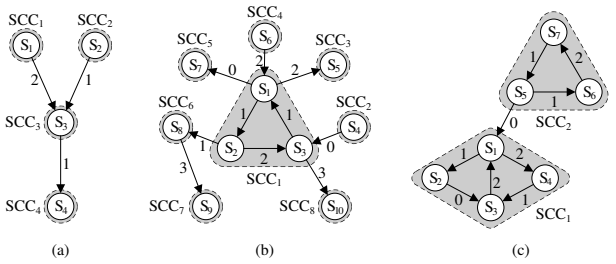

Figure 4: Three types of dependence graphs. (a) DAG. (b) CGSC. (c) CGMC

3.1 DAG_Migration

This algorithm given in Algorithm 3.2 finds a retiming function for a DAG so that the transformed loop is α-*parallelizable* for any positive integer α given. This implies that $w_r(u, v) = w(u, v) + r(u) - r(v) \geq \alpha$ for every edge (u, v) in G after retiming. Hence, in lines 1 – 3, these retiming constraints form a system of linear inequalities. A solution to the system found by the Bellman-Ford algorithm represents a retiming function of G as desired (lines 4 – 8) [15].

Algorithm 3.2 DAG_Migration(G, α)

Input: A DAG $G = (V, E, w)$ and a positive integer α.
Output: A retiming function r' of G.

1: **for** each edge (u, v) in G **do**
2: Generate a retiming constraint: $r(v) - r(u) \leq w(u, v) - \alpha$
3: **end for**
4: Build the constraint graph G' such that its node set is V and there is a directed edge from u to v with the weight $w(u, v) - \alpha$ if $r(v) - r(u) \leq w(u, v) - \alpha$ is a retiming constraint.
5: Let G'' be obtained from G' by adding a pseudo source node s_0 and a zero-weighted edge from s_0 to every node in G'.
6: Obtain a retiming function r by applying the Bellman-Ford algorithm to the single-source constraint graph G''.
7: Let r' be a retiming function normalized from r.
8: Return r'.

An example is shown in Figure 5. The original DAG G is shown in Figure 5(a), in which the minimum non-zero edge weight is 1. Thus, the parallelism of the original loop is 1. Suppose that α, the expected degree of parallelism, is 3. By imposing $w_r(u, v) \geq 3$ for

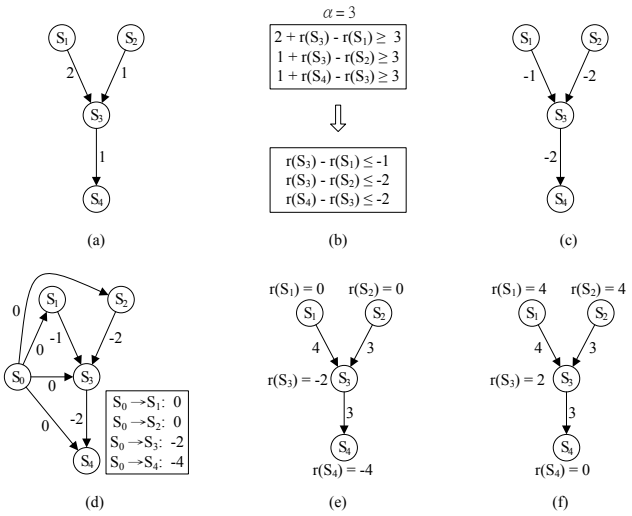

Figure 5: An illustration of DAG_Migration. (a) Original DAG. (b) Retiming constraints. (c) Constraint graph. (d) Single-source constraint graph and retiming function (inside the box) obtained by Bellman-Ford. (e) Transformed DAG by the retiming function. (f) Transformed DAG by the normalized retiming function.

each edge (u, v) in G, we obtain a system of retiming constraints given in Figure 5(b). Based on this system, the constraint graph G' is built as shown in Figure 5(c). This gives rise to the single-source constraint graph G'' shown in Figure 5(d), for which the solution found by Bellman-Ford is shown inside the box. For example, the weight of the shortest path from S_0 to S_1 is 0, i.e. $r(S_1) = 0$. The transformed dependence graph is shown in Figure 5(e) with the retiming function r shown. The retiming values of some nodes are negative. However, they are legal since there is no negative edge weight in the transformed dependence graph. The transformed dependence graph with the normalized retiming function r' is shown in Figure 5(f). It can be seen that the edge weights remain unchanged after the normalization, and the minimum non-zero edge weight is $\beta = \alpha = 3$. Therefore, the parallelism of the transformed loop is $\beta = 3$.

THEOREM 3.1. *Let α be a positive integer. If the dependence graph $G = (V, E, w)$ of a loop L_G is a DAG, then a* α-parallelizable *loop can be obtained from L_G by DAG_Migration in polynomial time.*

PROOF. First, DAG_Migration is a polynomial-time algorithm. Second, as shown in Algorithm 3.2, we can always obtain a retiming function such that the weight of each edge in the transformed dependence graph is greater than or equal to α. Hence, the transformed loop is *α-parallelizable*. □

3.2 Single_Cycle_Migration

A polynomial algorithm is proposed for a dependence graph with a single cycle. As shown in Algorithm 3.3, we first choose an edge (u, v) in the cycle such that the weights of all other edges in the cycle can be migrated to (u, v). According to the retiming definition, for an edge (u, v), the retimed edge weight $w_r(u, v) = w(u, v) + r(u) - r(v)$. In order to let the retimed weight of an edge in the cycle be 0 (i.e. $w_r(u, v) = 0$), $r(v)$ should equal $w(u, v) + r(u)$. Therefore, from node v to u along the path, we can repeatedly calculate the retiming value of each node in the cycle.

Algorithm 3.3 Single_Cycle_Migration(G, C_G)

Input: A CGSC $G = (V, E, w)$ with its single cycle $C_G \in G$.
Output: A retiming function of G.

1: $\theta \leftarrow$ the weight of the cycle C_G.
2: Select the edge (u, v) in cycle C_G with the biggest weight.
3: $k \leftarrow v$.
4: $Adj[k] \leftarrow$ adjacent node of k ($Adj[k] \in C_G$).
5: Let the retiming value of each node in G be 0.
6: **while** $Adj[k] \neq v$ **do**
7: $\quad r(Adj[k]) \leftarrow w(k, Adj[k]) + r(k)$.
8: $\quad k \leftarrow Adj[k]$.
9: **end while**
10: Fix the retiming value of each node in the cycle C_G.
11: Call DAG_Migration(G, θ) to let the weight of all edges not in C_G be greater than or equal to θ.
12: Return the retiming function obtained.

For node v, $r(v) = 0$; For all other nodes in the cycle, the retiming value of a node equals to the summation of the weight of its incoming edge and the retiming value of its parent. As a result, the weight of edge (u, v) equals the cycle weight θ while the weight of all other edges in the cycle becomes zero. Next, we fix the retiming values of all nodes in the cycle to guarantee the weight of the cycle remains constant. At last, we let the weight of all edges not belonging to the cycle be greater than or equal to θ by invoking Algorithm 3.2. After the processing of Algorithm 3.3, we can obtain a retiming function, and the minimum non-zero edge weight in the transformed dependence graph based on is equal to the cycle weight θ. Therefore, the transformed loop is *θ-parallelizable*. An example is shown in Figure 6.

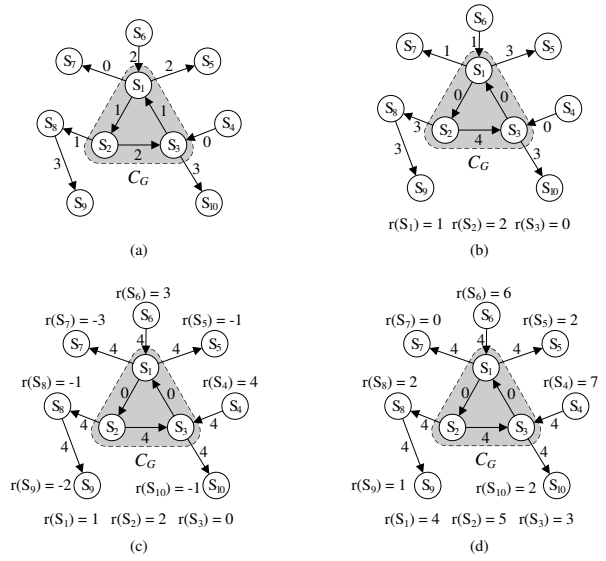

Figure 6: An illustration of Single_Cycle_Migration by an example loop. (a) Original dependence graph. (b) The edge weights in the cycle have all been migrated to (S_2, S_3) with $r(S_1) = 1$, $r(S_2) = 2$, and $r(S_3) = 0$. (c) Transformed dependence graph with the minimum non-zero edge weight 4. (d) Transformed dependence graph with the normalized retiming values.

The original dependence graph is shown in Figure 6(a), in which the minimum non-zero edge weight is 1. Thus the parallelism of

the original loop is 1. The dependence migration of the cycle is shown in Figure 6(b), in which all weights in the cycle is migrated to edge (S_2, S_3). Then, by fixing the retiming values of the nodes in the cycle (S_1, S_2, S_3), we apply DAG_Migration algorithm to obtain the retiming values of all other nodes by making their edge weights be no less than 4 as shown in Figure 6(c). The normalized dependence graph is shown in Figure 6(d). Since the minimum non-zero edge weight in the transformed dependence graph is 4, the transformed loop is *4-parallelizable*.

THEOREM 3.2. *Given a dependence graph $G = (V, E, w)$ of a loop, if G is a graph with single cycle C_G, and the weight of cycle C_G is $\theta(\theta \geq 0)$, then the transformed loop is* θ-parallelizable.

To prove this, we need Property 3.1 [15] below.

PROPERTY 3.1. *Let $G = (V, E, w)$ be a graph and r a retiming function of G. Then the summation of the edge weights in a cycle remains a constant after retiming.*

The proof of Theorem 3.2 is shown as follows.

PROOF. Based on Property 3.1, the cycle weight (the summation of the edge weights of a cycle) cannot be changed by retiming. Therefore, for the cycle in G, the possible maximum non-zero edge weight is the cycle weight θ . In order to achieve this, we need to move all dependences into one edge in such a way that the weight of one edge equals to the cycle weight, while the weight of all other edges in the cycle equals 0. At the same time, for all edges not belonging to the cycle, we need to make their non-zero edge weight be no less than θ in order to achieve *θ-parallelizable*. Algorithm 3.3 shows how to achieve this. □

3.3 Multi_Cycle_Migration

In this section, we propose an algorithm, Multi_Cycle_Migration, to construct an integer linear programming (ILP) formulation, by which the optimal solution can be obtained for the dependence graph with multiple cycles. The Multi_Cycle_Migration algorithm is shown in Algorithm 3.4.

Algorithm 3.4 Multi_Cycle_Migration(G)

Input: A CGMC $G = (V, E, w)$.
Output: A retiming function of G.

1: Let δ (φ) be the minimum (maximum) SCC weight.
2: **for** each edge $(u, v) \in E$ **do**
3: Add the following constraints into the ILP formulation:

$$\begin{cases} w(u,v) + r(u) - r(v) \geq 0 \\ w(u,v) + r(u) - r(v) + (1 - \varepsilon(u,v)) \times \varphi \geq T \\ w(u,v) + r(u) - r(v) - \varphi \times \varepsilon(u,v) \leq 0 \\ \varepsilon(u,v) = 0 \ \ or \ \ 1 \end{cases}$$

4: **end for**
5: Set the objective function: Maximize T where $T \in [0, \delta]$.
6: Find the feasible retiming values that satisfy the ILP formulation, and return the retiming function.

In Algorithm Multi_Cycle_Migration, we first find the minimum SCC weight δ and the maximum SCC weight φ in the given dependence graph. Then an integer linear programming formulation is constructed. The objective function of this ILP formulation is to maximize T which is an integer variable ($T \in [0, \delta]$). In the ILP model, T is the expected minimum non-zero edge weight that we attempt to achieve by retiming. And the minimum SCC weight δ is used as the upper bound of T since it is the possible maximum value that we can obtain for the minimum non-zero weight.

For each edge (u, v) in G, the constraints are added into the ILP formulation. In the constraints, $w(u, v)$ denotes the original edge weight; $w(u, v) + r(u) - r(v)$ denotes the weight of each edge (u, v) after retiming (i.e. $w_r(u, v) = w(u, v) + r(u) - r(v)$); $\varepsilon(u, v)$ is an integer variable that satisfies $0 \leq \varepsilon(u, v) \leq 1$. Thus $\varepsilon(u, v)$ can be 0 or 1 in our ILP model. The first constraint is obtained by $w_r(u, v) = w(u, v) + r(u) - r(v) \geq 0$, which is based on the legality of a retiming function we discuss in Section 2 (A retiming function r is legal if the retimed weights of all edges in the retimed graph G_r are non-negative). The second and third constraints are a pair to force that the retimed edge weight can only be either 0 or a value no less than T. When $\varepsilon(u, v) = 0$, this pair of constraints become:

$$\begin{cases} w(u,v) + r(u) - r(v) + \varphi \geq T \\ w(u,v) + r(u) - r(v) \leq 0 \end{cases}$$

Since φ is selected as the maximum weight and $w(u, v) + r(u) - r(v) \geq 0$, in this case, we can obtain $w(u, v) + r(u) - r(v) = 0$, which means that the retimed edge weight is zero.

Correspondingly, when $\varepsilon(u, v) = 1$, this pair of constraints become:

$$\begin{cases} w(u,v) + r(u) - r(v) \geq T \\ w(u,v) + r(u) - r(v) - \varphi \leq 0 \end{cases}$$

In this case, we can guarantee that the retimed edge weight is not less than T ($w(u, v) + r(u) - r(v) \geq T$). At the same time, we set $w_r(u, v) \leq \varphi(w(u, v) + r(u) - r(v) - \varphi \leq 0)$, that is, any retimed edge weight cannot be larger than φ, the maximum cycle weight. The reason is that for any edge in a cycle, based on Property 3.1 and the legality of retiming, its retimed edge weight cannot be larger than φ; for any edge not belonging to cycles, as shown in Section 3.1, their retimed edge weight can achieve any nonnegative integer. Therefore, we can use φ as the upper bound for retimed edge weights.

Figure 7 shows an example of Multi_Cycle_Migration algorithm. Figure 7(a) shows the original loop. Figure 7(b) shows the corresponding dependence graph with single SCC, in which there are two cycles. In this example, the minimum SCC weight equals the maximum SCC weight, i.e. $\delta = \varphi = 6$. The objective function of our ILP formulation is to maximize T whose upper bound is 6. Figure 7(c) shows the transformed dependence graph with the maximized minimum non-zero edge weight 3. So the parallelism of the transformed loop 3. Figure 7(d) shows the prologue, new loop kernel and epilogue. Figure 7(e) shows three consecutive iterations that can be executed in parallel.

In the following, we prove Multi_Cycle_Migration algorithm gives the optimal solution for a given dependence graph with multiple cycles.

THEOREM 3.3. *An optimal solution can be obtained by Algorithm Multi_Cycle_Migration if G is a dependence graph with multiple cycles.*

PROOF. As discussed above, from our ILP formulation, we can guarantee that any retimed edge weight can only be either 0 or a value no less than T. In the ILP model, we set up the upper bound of T as the minimum SCC weight, since it is the maximum value we can obtain for all possible minimum non-zero weights. The objective function of the ILP formulation is to maximize T. Therefore, using the ILP model, we maximize the minimum non-zero edge weight. □

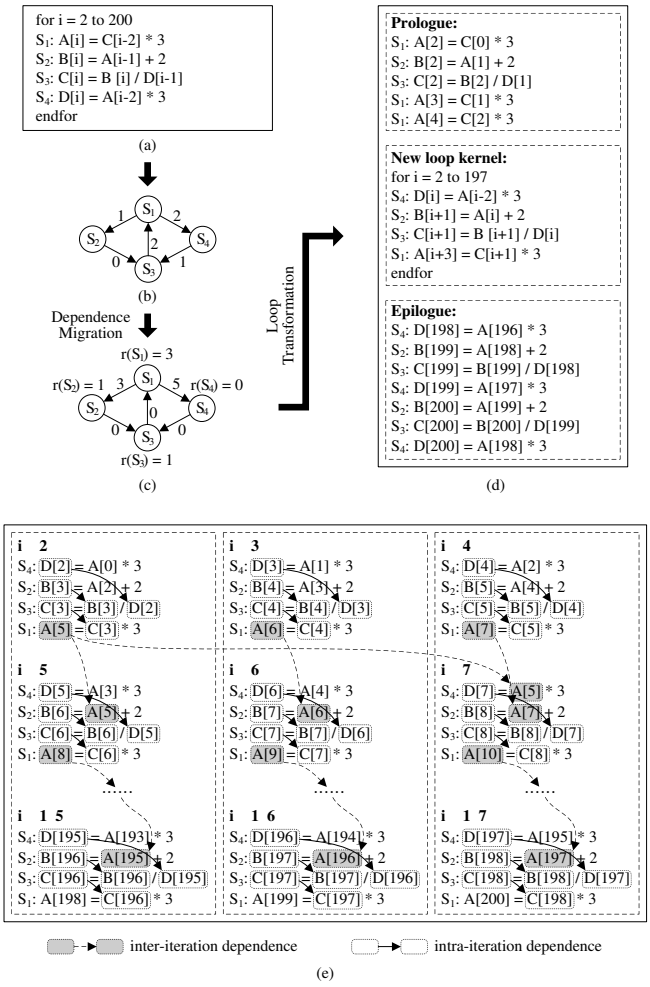

Figure 7: An illustration of Multi_Cycle_Migration by an example loop. (a) Original loop kernel. (b) Original dependence graph. (c) Transformed dependence graph with the minimum non-zero edge weight 3. (d) New loop kernel. (e) A graphic representation of the parallelism in the transformed loop.

4. LOOP TRANSFORMATION ALGORITHM

In this section, we propose a loop transformation algorithm for generating the optimized code for a given loop, including the loop kernel, prologue and epilogue, based on the retiming values obtained by the DMA algorithm in the first phase. The loop transformation algorithm is shown in Algorithm 4.1.

From Algorithm Loop_Transformation, we can see that the upper bound and lower bound of the original loop are assigned to M and L respectively. For example, in Figure 7(a), the upper bound $M = 200$ and the lower bound $L = 2$. Then some copies of a node u are put into prologue or epilogue according to M, L and the retiming value of each node. At the same time, the indexes of the new loop kernel are transformed. In addition, the execution sequence of each node in the new loop kernel must be sorted based on the intra-iteration dependences between any two nodes in the transformed dependence graph. Finally, we set the new upper bound of the loop index for the new loop kernel. As a result, the code of the transformed loop is produced. An example is given in Figure 7 to show how to obtain the new loop kernel, prologue, epilogue, and the loop indexes from the retiming values.

Algorithm 4.1 Loop_Transformation(G_r, $r(u)$, L_G)

Input: Transformed graph $G_r = (V, E, w_r)$, retiming r, and the original loop L_G.

Output: The transformed loop L_{G_r} with new loop kernel, prologue and epilogue.

1: Let M (L) be the upper (lower) bound of the loop index of L_G.
2: $r_{max} = \max_{u \in V}(r(u))$.
3: **for** each node $u \in V$ **do**
4: /* Generate the prologue of the loop L_{G_r} */
5: **for** $i = L$ to $(L + r(u) - 1)$ **do**
6: Put u into the prologue where the loop index equals i.
7: **end for**
8: /* Generate the epilogue of the loop L_{G_r} */
9: **for** $i = 1$ to $(r_{max} - r(u))$ **do**
10: Put u into the epilogue where the loop index equals $M - i + 1$.
11: **end for**
12: /* Generate the loop kernel of the loop L_{G_r} */
13: Increase the loop index of u by $r(u)$ in the new loop kernel.
14: **end for**
15: /* Sort the execution sequence in the new loop kernel */
16: **for** each edge $(u, v) \in E$ **do**
17: **if** $w_r(u, v) = 0$ **then**
18: Move u in front of node v in the new loop kernel.
19: **end if**
20: **end for**
21: Let $M - r_{max}$ be the upper bound of the loop index of L_{G_r}.
22: Return L_{G_r}

5. EVALUATION

In this section, we evaluate our technique with a set of benchmarks frequently used in iteration-level loop parallelization. The benchmarks include six loop kernels with single cycle or multiple cycles which are obtained from [24, 4, 14, 8, 12, 19]. We use these benchmarks to test the effectiveness of our loop parallelization technique. Our ILP model for graph with multiple cycles is implemented based on the open source linear programming, LP_solve_5.5[28]. We compare our approach with the previous work in [23, 20]. The basic information of these loops is shown in Table 1.

Table 1: Loop parallelism of DMA vs. cycle-shrinking and partitioning [23, 20].

Benchmarks	Cycle Num.	Iteration-Level Parallelism		
		Original	Cycle-Shrinking & Partition [23, 20]	Our Tech.
SHRINKING [24]	1	2	2	5
BREAKING [4]	1	1	1	6
LPDOSER [14]	1	1	1	2
REORDER [8]	1	1	1	4
BMLA [12]	2	1	1	3
EDDR [19]	2	1	1	3

As shown in Table 1, the original parallelism of SHRINKING is 2 while the parallelism of other loops is 1. We apply both cycle-

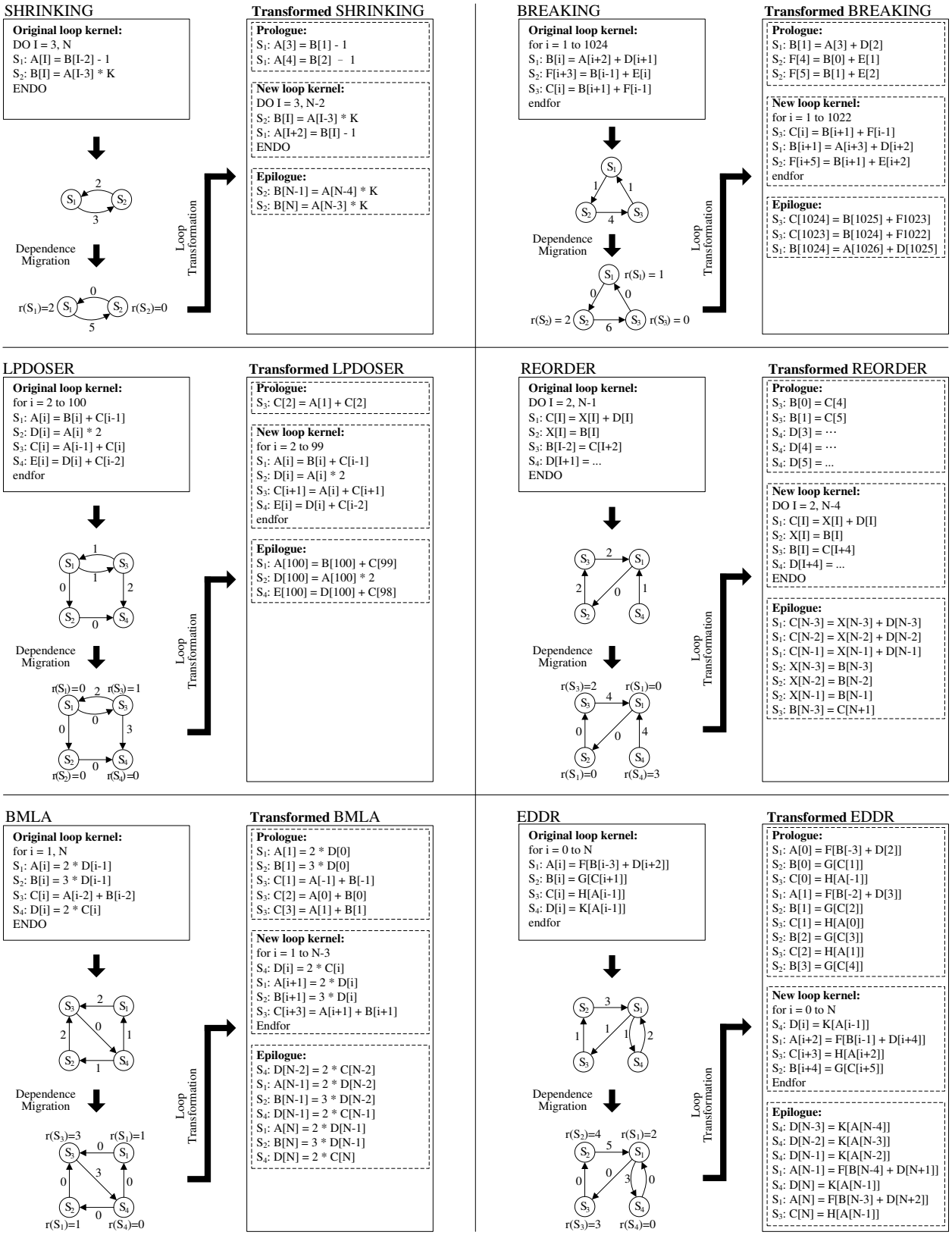

Figure 8: Optimal transformations obtained by our loop parallelization technique.

shrinking [23] and partitioning [20] to these loops. It can be seen that both cycle-shrinking [23] and partitioning [20] cannot improve the loop parallelism for these loops. The numbers in the last column show the parallelism achieved by applying our technique. The results show that our technique can remarkably enhance the parallelism for these loops. Taking loop BREAKING as an example, the best result by cycle-shrinking and partitioning is 1 while our technique can obtain 6. Therefore, by applying our technique, we can parallelize these loops and achieve various degrees of iteration-level parallelism.

Figure 8 shows how our technique works on each of these six loops and it confirms that our technique can effectively improve loop parallelism compared to the previous work. For each loop kernel in Figure 8, we can see that it will be first modeled by a corresponding dependence graph. Then a dependence graph is transformed by using the dependence migration algorithm introduced in Section 3. Finally, according to the retiming values obtained in the first phase, a new loop kernel with prologue and epilogue can be generated by using our loop transformation algorithm given in Section 4. In a transformed dependence graph, it is noticeable that the weight of some edges is zero while the weight of other edges is a non-zero integer. In addition, we can see that the minimum non-zero edge weight in a cycle within a transformed dependence graph equals to the cycle weight. This means that the minimum non-zero edge weight in the transformed dependence graph is maximized. Correspondingly, the parallelism of the original loop kernel is maximized.

6. CONCLUSION

In this paper, we have proposed an optimal two-phase iteration-level loop parallelization approach to maximize the loop parallelism. In the first phase, we solve the dependence migration problem that is to find a retiming value of each node in a given dependence graph such that the minimum non-zero edge weight in the dependence graph can be maximized. In the second phase, based on the retiming function obtained in the first phase, we proposed a loop transformation algorithm to generate the transformed loop kernel, prologue, and epilogue. We conducted experiments on a set of benchmarks frequently used in iteration-level loop parallelization in the previous work. The results show that our technique can efficiently obtain the optimal solutions and effectively improve loop parallelism compared to the previous work.

There are several directions for future work. First, in the paper, we only discuss how to apply our technique in iteration-level parallelism. In fact, after iterations are parallelized, they can be directly vectorized. How to combine this technique with loop vectorization is one direction for future work. Second, our technique can be applied to optimize the innermost loop for nested loops. It is an interesting problem to extend this approach to solve iteration-level parallelism of nested loops.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and improvements to this paper. The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF POLYU 5260/07E) and HK PolyU 1-ZV5S, National 863 Program of China (Grant No. 2006AA01Z172, Grant No. 2008AA01Z106), National Natural Science Foundation of China (Grant No. 60533040), National Science Fund for Distinguished Young Scholars (Grant No. 60725208) and Australian Research Council Grant (DP0987236).

8. REFERENCES

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. *ACM SIGPLAN Notices*, 23(7):308–317, 1988.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, first edition, 2001.

[3] L. Bic, J. M. A. Roy, and M. Nagel. Exploiting iteration-level parallelism in dataflow programs. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 376–381, June 1992.

[4] W.-L. Chang, C.-P. Chu, and M. Ho. Exploitation of parallelism to nested loops with dependence cycles. *Journal of Systems Architecture*, 50(12):729–742, 2004.

[5] L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Computer Science, Princeton University, 1993.

[6] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: a loop pipelining algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229–239, March 1997.

[7] L.-F. Chao and E. H.-M. Sha. Static scheduling of uniform nested loops. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 254–258, April 1993.

[8] D.-K. Chen and P.-C. Yew. Statement re-ordering for doacross loops. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP '94)*, volume 2, pages 24–28, August 1994.

[9] C.-P. Chu and D. L. Carver. Reordering the statements with dependence cycles to improve the performance of parallel loops. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)*, pages 322–328, 1997.

[10] R. G. Cytron. *Compile-time Scheduling and Optimizations for Multiprocessor System*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, September 1984.

[11] A. Darte and G. Huard. Complexity of multi-dimensional loop alignment. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS '02)*, pages 179–191, 2002.

[12] A. Fraboulet and A. Mignotte. Source code loop transformations for memory hierarchy optimizations. In *Proceedings of the Workshop on Memory Access Decoupled Architecture (MEDEA '01)*, pages 8–12, Barcelona, Spain, September 2001.

[13] D. A. P. Haiek. *Multiprocessors: Discussion of some Theoretical and Practical Problems*. PhD thesis, Deptartment of Computer Science, University of Illinois at Urbana-Champaign, November 1979.

[14] D. N. Jayasimha and J. D. Martens. Some architectural and compilation issues in the design of hierarchical shared-memory multiprocessors. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 567–572, March 1992.

[15] C. E. Leiserson1 and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[16] L.-S. Liu, C.-W. Ho, and J.-P. Sheu. On the parallelism of nested for-loops using index shift method. In *Proceedings of the International Conference on Parallel Processing (ICPP '90)*, pages 119–123, 1990.

[17] M. Liu, Q. Zhuge, Z. Shao, and E. H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '04)*, pages 190–201, September 2004.

[18] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[19] K. Okuda. Cycle shrinking by dependence reduction. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing (Euro-Par '96)*, pages 398–401, 1996.

[20] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[21] L. Passos and E. H.-M. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1150–1163, November 1996.

[22] N. L. Passos and E. H.-M. Sha. Full parallelism in uniform nested loops using multi-dimensional retiming. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP '94)*, pages 130–133, August 1994.

[23] C. D. Polychronopoulos. Advanced loop optimizations for parallel computers. In *Proceedings of the 1st International Conference on Supercomputing*, pages 255–277, March 1988.

[24] C. D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1988.

[25] Y. Robert and S. W. Song. Revisiting cycle shrinking. *Parallel Computing*, 18(5):481–496, 1992.

[26] E. H.-M. Sha, C. Lang, and N. L. Passos. Polynomial-time nested loop fusion with full parallelism. In *Proceedings of the Proceedings of the 1996 International Conference on Parallel Processing (ICPP '96)*, volume 3, page 9, August 1996.

[27] W. Shang, M. T. O'Keefe, and J. A. B. Fortes. On loop transformations for generalized cycle shrinking. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):193–204, February 1994.

[28] L. Solve. http://lpsolve.sourceforge.net/5.5/. 2009.

[29] X. Tang and G. R. Gao. Automatically partitioning threads for multithreaded architectures. *Journal of Parallel and Distributed Computing*, 58(2):159–189, 1999.

[30] C.-M. Wang and S.-D. Wang. Compiler techniques to extract parallelism within a nested loop. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC '91)*, pages 24–29, Tokyo, Japan, September 1991.

[31] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[32] C. Xue, Z. Shao, M. Liu, and E. H.-M. Sha. Iterational retiming: maximize iteration-level parallelism for nested loops. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '05)*, pages 309–314, September 2005.

[33] J. Xue, M. Guo, and D. Wei. Improving the parallelism of iterative methods by aggressive loop fusion. *Journal of Supercomputing*, 43(2):147–164, February 2008.

[34] J. Xue, Q. Huang, and M. Guo. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP '05)*, pages 107–115, 2005.