Optimally Maximizing Iteration-Level Loop Parallelism

Duo Liu, Yi Wang, Zili Shao, *Member*, *IEEE*, Minyi Guo, *Senior Member*, *IEEE*, and Jingling Xue, *Senior Member*, *IEEE*

Abstract—Loops are the main source of parallelism in many applications. This paper solves the open problem of extracting the maximal number of iterations from a loop to run 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 nonzero 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 nonoptimal algorithms using a set of benchmarks frequently used in prior work and a set of graphs generated by TGFF.

Index Terms—Loop parallelization, loop transformation, retiming, data dependence graph, iteration-level parallelism.

1 Introduction

CHIP multiprocessors, such as Intel Dual-Core processors, AMD Phenom processors, and ARM11 MPCore 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 overemphasized. While programmers can utilize multiple-threaded application development environments to generate coarse-grain parallel programs with thread-level parallelization in practice [2], [3], 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 [4], [5], [6], [7], [8], [9]. 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 [4], [10], [11]. Most of these techniques, however, focus on instruction-level parallelism. In this paper, we propose an iteration-level loop parallelization technique that supplements the previous work by enhancing loop parallelism. We target at iteration-level parallelism [12] by which different iterations from the same loop kernel can be executed in parallel.

- *D. Liu, Y. Wang, and Z. Shao are with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, and D. Liu is also a lecturer at the School of Computer Science and Technology, Southwest University of Science and Technology, Mianyang, P.R. China. E-mail: cszlshao@comp.polyu.edu.hk.*
- *M. Guo is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, P.R. China.*
- *J. Xue is with the Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Australia.*

Manuscript received 14 May 2010; revised 21 Jan. 2011; accepted 11 May 2011; published online 13 June 2011.
Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-05-0293.
Digital Object Identifier no. 10.1109/TPDS.2011.171.

At the iteration level, based upon the degree of parallelism, loops can be mainly classified into three categories: serial loops, parallel loops (DOALL) [13], and partially parallel loops (DOACROSS) [14], [15]. Without any loop transformations, all iterations in a serial 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 interiteration dependences. In the case of a DOACROSS loop, its successive iterations can be partially overlapped because of interiteration data dependences. In this paper, we focus on maximizing loop parallelism for serial and DOACROSS 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 [16].

There have been numerous studies to enhance loop parallelism by exploiting data dependences [17] of dependence cycles [4], [10], [18], [19], [20], [21]. In [18], 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. Cycle shrinking [19], [20], [22] is a loop tiling technique by which consecutive dependence-free iterations are grouped to form the innermost loop kernel of a new set of nested loops. In [4] and [10], node splitting is used to eliminate anti- or output-dependences by adding new copy statements. In [21], 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 interiteration data dependences so optimal parallelism can be achieved by the proposed technique in this paper.

For nested loops with a single statement each, the minimum distance method [23], [24] is proposed to maximize loop parallelism by partitioning a loop into multiple independent execution sets and then map these execution sets to different processors. In this method, a dependence matrix formed from the original loop is transformed into an upper triangular matrix, which is then used to identify the

independent execution sets. This method is extended to solve a special case for nested loops with multiple statements, in which the loop dependences among all variables could be transformed and derived as if they were from only one variable, such that the dependence matrix is simplified and can be used to obtain maximum partitions. However, such an extension was discussed only briefly and illustrated by examples. No systematic method is given to solve the multistatement problem, in general. This paper focuses on a more general case for one-level loop with multiple statements, i.e., the loop dependences among all variables may or may not be transformed and derived from one variable. Our method can solve this class of loops optimally for the first time in a systematic manner. A general algorithm is given and illustrated by examples and experimental evaluation.

In our proposed technique, loop transformation is modeled by retiming [25], [26], [27]. 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 [28]. For loop transformation, retiming is used under the name of "index shift method" for parallelizing nested loops [29]. The basic idea of the index shift method is to defer or advance the execution steps of some statements in such a way that we can increase the parallelism of a loop based on the hyperplane method. In [30], a loop optimization method is proposed to optimize nested loops by combining the index shift method [29] and the generalized cycle shrinking [22]. In [31], 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 none considers iteration-level parallelism for serial and DOACROSS loops. 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 interiteration data dependences by regrouping statements of a loop kernel in such a way that the number of consecutive independent iterations is always maximized. In our technique, a dependence graph is constructed to model data dependences among statements in a loop, and then retiming is used to model dependence migration among 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 nonzero 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 Directed Acyclic Graph (DAG), 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 (ILP) formulation that can be solved efficiently for the dependence graphs found in real code. Finally, we give a loop transformation algorithm that can generate the optimized code for a given loop, including its prologue, loop kernel, and epilogue based on the retiming values of 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 serial or DOACROSS loop.
- We demonstrate the effectiveness of our technique by comparing with a number of representative (non-optimal) techniques using a set of benchmarks frequently used in prior work and a set of graphs generated by TGFF [32].

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 this paper 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. First, the notion of dependence graph is introduced in Section 2.1. Then, in Section 2.2, we examine briefly how to use retiming to model dependence migration among the edges in a dependence graph. A brief discussion on iteration-level parallelism is presented in Section 2.3. Finally, the problem addressed is defined 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 *intraiteration dependence* if $w(u, v) = 0$ and an *interiteration 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 dependences are further explained as follows:

- Intraiteration dependence $w(u, v) = 0$. Such a dependence occurs in the same iteration between a pair of statements. If there exists an intraiteration dependence between two statements u and v within the same iteration, then statement v reads the results generated by statement u.
- Interiteration dependence $w(u, v) > 0$. Such a dependence occurs when two statements from different iterations are dependent on each other. If there exists an interiteration dependence between u and v, then the execution of statement v in

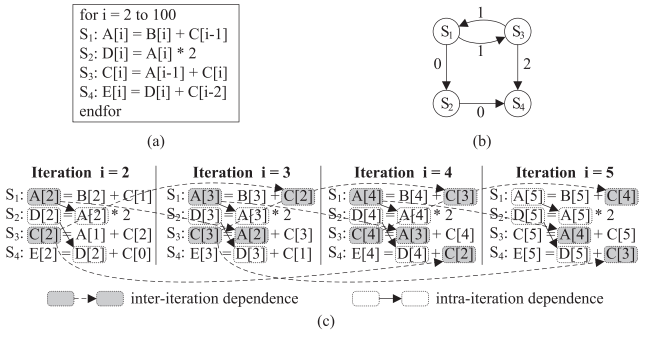


Fig. 1. A loop kernel from [33] and its dependence graph. (a) The loop kernel. (b) The dependence graph. (c) Intra- and interiteration dependences.

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.

We use a real loop application from [33] to show how to use a dependence graph to model a loop. The loop kernel is shown in Fig. 1a and its corresponding dependence graph in Fig. 1b. This loop has both intraiteration and interiteration dependences. For example, the weight of edge (S_1, S_2) is zero, indicating an intraiteration dependence between S_1 and S_2. The weight of edge (S_3, S_4) is 2, indicating an interiteration dependence between S_3 and S_4 with a distance of 2.

Fig. 1c illustrates how iteration-level loop parallelism can be constrained by dependences with the first four iterations shown. Let us examine intraiteration 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 intraiteration dependence (due to $D[i]$). In general, intraiteration dependences are confined to the same iteration and thus do not inhibit iteration-level parallelism. Let us next examine interiteration 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 one iteration in parallel since every iteration requires results from the preceding two iterations. Hence, interiteration dependences are the major obstacle to iteration-level parallelism.

2.2 Retiming and Dependence Migration

Retiming [25] 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 $G_r = (V, E, w_r)$ 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.*

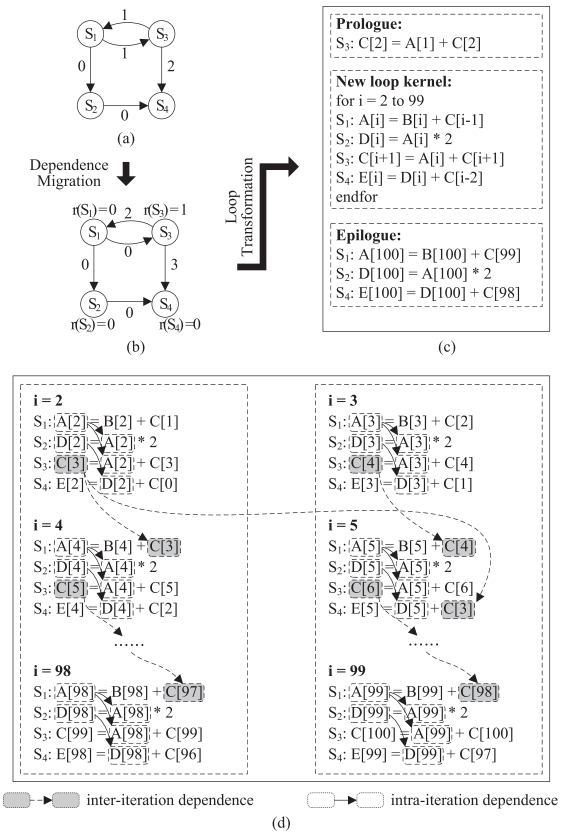


Fig. 2. Loop transformation for Fig. 1b. (a) The original dependence graph (with the minimum nonzero edge weight being 1). (b) The transformed dependence graph with the minimum nonzero edge weight being 2. (c) The new loop kernel after loop transformation. (d) A graphical representation of parallelism in the transformed loop.

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 nonnegative. An illegal retiming function occurs when one of the retimed edge weights becomes negative, and this situation implies a reference to nonavailable 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 [25].

For simplicity, we normalize a retiming r such that the minimum retiming value(s) is always zero [34]. A retiming function r can be normalized by subtracting $\min_v r(v)$ from $r(v)$ for every v in V [35].

As an example shown in Fig. 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 interiteration dependence

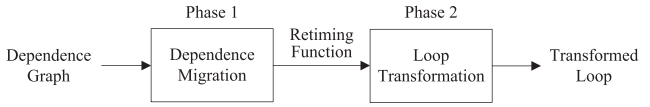


Fig. 3. Optimal retiming-based loop parallelization.

$(w(S_1, S_3) = 1)$ between S_1 and S_3 is transformed into an intraiteration dependence $(w_r(S_1, S_3) = 0)$. Fig. 2c shows the new loop kernel obtained based on this retiming function. Fig. 2d 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 are executed in parallel. However, loop iterations must be executed in accordance with their interiteration dependences. Thus, interiteration dependences inhibit iteration parallelization. For example, in Fig. 1c, S_3 and S_4 have an interiteration dependence with a distance of 2. As a result, the ith 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 interiteration dependence between S_1 and S_3 with a distance of 1. Therefore, the minimum interiteration dependence distance in a loop (i.e., the minimum nonzero 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 nonzero edge weight with dependence migration and loop transformation. Given a dependence graph for a loop, its minimum nonzero edge weight, β, represents the parallelism degree of the loop, which implies the absence of interiteration 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 Fig. 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 nonzero edge weight β of the transformed dependence graph $G_r = (V, E, w_r)$ is maximized.

Existing solutions [4], [10], [18], [19], [20], [21] to this problem are all approximate for serial and DOACROSS loops. Our solution, as outlined in Fig. 3, solves the problem optimally (for the first time) in two phases. In the first phase, we introduce a *dependence migration algorithm (DMA)* 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.

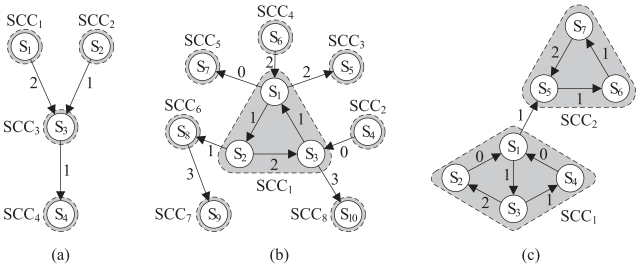


Fig. 4. Three types of dependence graphs. (a) DAG. (b) CGSC. (c) CGMC.

3 Dependence Migration Algorithm

In this section, we introduce our dependence migration algorithm given in Algorithm 3.1 to find a retiming function for a given dependence graph so that the minimum nonzero 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)
Input: A dependence graph $G = (V, E, w)$ of a loop L_G.
Output: A retiming function of G.

```
1: SNode_SCC ← 0; SCycle_Num ← 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 && E_Num == N_Num then
7:         SCycle_Num ++.
8:         Let C_G[SCycle_Num] be the SCC (V_SCC, E_SCC, w).
9:     end if
10: end for
11: if SNode_SCC == SCC_Num then
12:     Let α be the parallelism degree (any positive
        number) we want to achieve.
13:     DAG_Migration(G, α).
14: else if SCycle_Num == 1 && SNode_SCC ==
    SCC_Num − 1 then
15:     Single_Cycle_Migration(G, C_G[SCycle_Num]).
16: else
17:     Multi_Cycle_Migration(G).
18: end if
```

We classify a dependence graph into one of the three types: a DAG, a Cyclic Graph with Single Cycle, and a Cyclic Graph with Multiple Cycles, based on the structure of the Strongly Connected Components (SCCs) in the graph. The SCCs in a dependence graph can be obtained by using the Tarjan's Algorithm [36]. For DAGs and CGSCs, two algorithms are given to find their optimal retiming functions polynomially, respectively. For CGMCs, an optimal algorithm is given based on an integer linear programming formulation. The three different types of dependence graphs, as illustrated in Fig. 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 Fig. 4a.

- **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 Fig. 4b.
- **CGMCs.** In this general case, a graph G has more than one cycle. Multi_Cycle_Migration comes into play to retime G. Fig. 4c gives a dependence graph with multiple cycles.

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, which means that a loop can be fully parallelized if its dependence graph is a DAG. 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) [25].

Let $|V|$ and $|E|$ be the node and edge numbers of G, respectively. In Algorithm 3.2, the number of all retiming constraints is $|E|$ and the number of all unknown variables is $|V|$ (for each node we want to know its retiming value). Therefore, the node number of the constraint graph is $|V| + 1$ and the edge number of the constraint graph is $|E| + |V|$ (for each constraint, there is one edge in the constraint graph and for the pseudonode there are $|V|$ edges connecting to all other nodes). So, it takes $O((|V|+1)(|E|+|V|))$ to compute the retiming function by using the Bellman-Ford algorithm. Thus, the time complexity of Algorithm 3.2 is $O(|V|^2 + |V||E|)$.

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'', in which for a node u ($u \neq s_0$), $r(u)$ is the shortest distance from s_0 to u.
7: Let r' be a retiming function normalized from r.
8: Return r'.

An example is shown in Fig. 5. The original DAG G is shown in Fig. 5a, in which the minimum nonzero 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 each edge (u,v) in G, we obtain a system of retiming constraints given in Fig. 5b. Based on this system, the constraint graph G' is built as shown in Fig. 5c. This gives rise to the single-source constraint graph G'' shown in Fig. 5d, 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 Fig. 5e 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 Fig. 5f. It can be seen that the edge weights remain unchanged after the normalization, and the minimum nonzero edge weight is $\beta = \alpha = 3$. Therefore, the parallelism of the transformed loop is $\beta = 3$.

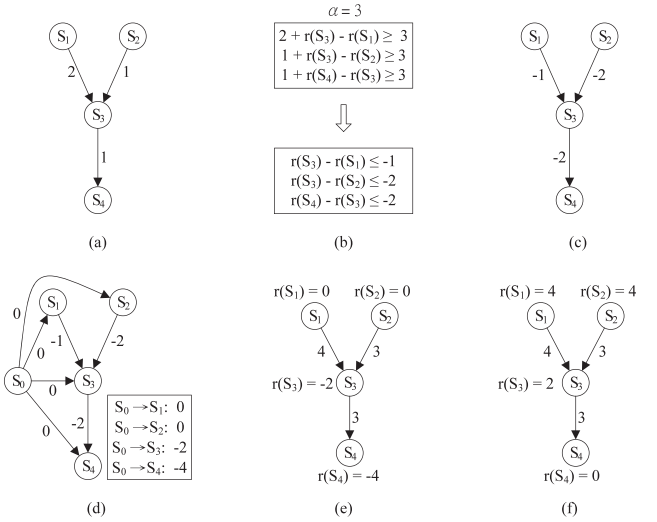


Fig. 5. An illustration of DAG_Migration. (a) The original DAG. (b) The retiming constraints. (c) The constraint graph. (d) The single-source constraint graph and retiming function (inside the box) obtained by Bellman-Ford. (e) The transformed DAG by the retiming function. (f) The transformed DAG by the normalized retiming function.

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.*

(The proof can be found in Supplementary File S1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

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. For node v, $r(v) = 0$; for all other nodes in the cycle, the retiming value of a node equals 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 each edge 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 nonzero edge weight in the transformed dependence graph based on is equal to the cycle weight θ. Therefore, the transformed loop is *θ-parallelizable*.

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: $Adj[k] \leftarrow$ the 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 such that all nodes of the cycle can be fused into a single node and G becomes a DAG.
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.

Let $|V|$ and $|E|$ be the node and edge numbers of G, respectively. In Algorithm 3.3, the node number of the single cycle is bounded by $|V|$. Therefore, it takes $O(|V|)$ to finish the retiming value assignment in the single cycle from Steps 1 to 10. Then, Algorithm 3.2 is called to obtain the retiming values of all other nodes in G. Thus, the time complexity of Algorithm 3.3 is $O(|V|^2 + |V||E|)$ as well.

An example is shown in Fig. 6. The original CGSC dependence graph is shown in Fig. 6a, in which the minimum nonzero edge weight is 1. Thus, the parallelism of the original loop is 1. The dependence migration of the cycle is shown in Fig. 6b, in which all weights in the cycle are migrated to edge (S_2, S_3). Then, by fixing the retiming values of every node in the cycle $C_G(S_1, S_2, S_3)$, we apply DAG_Migration algorithm to obtain the retiming values of all other nodes not in C_G by making their edge weights be no less than 4 as shown in Fig. 6c. The normalized dependence graph is shown in Fig. 6d. Since the minimum nonzero 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 L_G, if G is a graph with single cycle C_G, and the cycle weight of C_G (the summation of the edge weights of C_G) is $\theta(\theta \geq 0)$, then the best loop parallelization degree we can obtain from L_G is θ.*

(Please find the proof in Supplementary File S2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

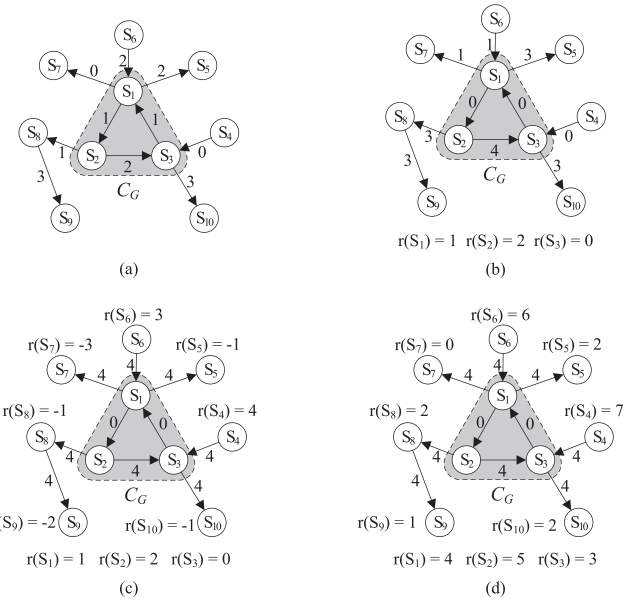


Fig. 6. An illustration of Single_Cycle_Migration. (a) The Original CGSC. (b) The edge weights in the cycle have all been migrated to (S_2, S_3) with $r(S_1) = 1$, $r(S_2) = 2$, and $r(S_3) = 0$. (c) The transformed CGSC with the minimum nonzero edge weight 4. (d) The transformed dependence graph with the normalized retiming values.

Algorithm 3.3 can be used to solve one extended case for a cyclic graph with multiple disjoint single cycles (CGMDSC). For this type of graph, we can first obtain the retiming function of each single cycle by invoking Algorithm 3.3. Then, similar to Algorithm 3.3, we fix the retiming value of each node in all the cycles, and let the weight of each edge not belonging to the cycles be greater than or equal to the minimum cycle weight among all the cycles by invoking Algorithm 3.2. As a result, the parallelism of the transformed dependence graph equals the minimum cycle weight among all the cycles. (The complete algorithm and one example for this case are given in the Supplementary File S3, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

3.3 Multi_Cycle_Migration

In this section, the Multi_Cycle_Migration algorithm is proposed to construct an integer linear programming formulation, by which the optimal solution can be obtained for the dependence graph with multiple cycles. As shown in Algorithm 3.4, we first obtain δ, the minimum SCC weight, and φ, the maximum SCC weight, where an SCC weight is the summation of all edge weights of an SCC. Then, for each edge, we add a set of retiming constraints so its retimed edge weight will become either 0 or T. Finally, we set the objective function as maximizing T where $T \in [0, \delta]$.

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.

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

(The proof and an example can be found in Supplementary File S4, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

4 Loop Transformation Algorithm

The loop transformation algorithm generates the optimized code for a given loop, including the new loop kernel, prologue and epilogue, based on the retiming values obtained by the DMA algorithm in the first phase. In our loop transformation algorithm, based on the retiming value of each node, some copies of the node are put into prologue or epilogue, and its loop index is changed accordingly as well. In addition, the execution sequence of nodes in the new loop kernel is revised based on intraiteration dependences 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. (For details, please refer to Supplementary File S5, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

5 Evaluation

In this section, we first evaluate our technique with a set of benchmarks frequently used in iteration-level loop parallelization. Then, we conduct a series of experiments with a set of graphs randomly generated by TGFF [32] to evaluate the scalability of our integer linear programming formulation.

We use real benchmarks to test the effectiveness of our loop parallelization technique. The benchmarks include six loop kernels with single cycle or multiple cycles which are obtained from [10], [20], [33], [37], [38], [39]. For different type of benchmarks, we use corresponding algorithm introduced in Section 3 to solve. The basic information of these loops is shown in Table 1. We compare our approach with the previous work in [18] and [19].

As shown in Table 1, the original parallelism of SHRINKING is 2 while the parallelism of other loops is 1. We apply both cycle shrinking [19] and partitioning [18] to these loops. It can be seen that both cycle shrinking and partitioning 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. The detailed description of how our technique improves loop parallelism of these six benchmarks is given in Supplementary File S6, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.

TABLE 1
Loop Parallelism of DMA versus Cycle Shrinking and Partitioning [18], [19]

Benchmarks	Cycle Num.	Iteration-Level Parallelism		
		Original	Cycle-Shrinking & Partitioning [19], [18]	Our Tech.
SHRINKING [20]	1	2	2	5
BREAKING [10]	1	1	1	6
LPDOSER [33]	1	1	1	2
REORDER [37]	1	1	1	4
BMLA [38]	2	1	1	3
EDDR [39]	2	1	1	3

To further evaluate the Multi_Cycle_Migration algorithm given in Algorithm 3.4, we have conducted a series of experiments with a set of CGMCs. Our experiments are performed on a 3.0 GHz quad-core Intel Xeon machine running Redhat Enterprise Linux 5 (kernel version is 2.6.18) with 16 GB main memory and a 1TB hard disk. In our experiments, each generated CGMC is modeled based on the ILP formulation in Algorithm 3.4 and solved by ILOG CPLEX 11.2 [40], a commercial linear programming solver.

We design a graph generation program to generate CGMCs, and the program works as follows: a DAG is initially generated by TGFF and then augmented by randomly adding multiple edges in the DAG. A test is made to see whether there exists more than one cycle in the DAG. If the number of cycles is smaller than two, more edges should be added in the DAG until the number of cycles is greater than one. In this way, a cyclic graph with multiple cycles can be generated.

In our experiments, we use this graph generation program to randomly generate 12 graph sets, and each graph set contains 210 CGMCs with various node counts and edge counts. For the graphs in each set, the number of nodes varies from 5 to 100 with an interval of 5, and the number of edges varies from 10 to 200 with the same interval. Each edge weight in a graph is set within the range of 0-6. So, distinct CGMCs can be generated with different combinations of node counts and edge counts.

Table 2 reports the statistical results for all graphs in each graph set by the Multi_Cycle_Migration algorithm. As shown, in each graph set, most of the graphs (more than 92.00 percent) are solved optimally within 60 seconds, and only a small part of the graphs (less than 8.00 percent) are solved optimally by taking more than 60 seconds. Among all graphs, the maximum time spent is 15,000 seconds when a graph with 44 nodes and 99 edges is processed. The

TABLE 2
Loop Parallelism and Computation Times of our Technique on the Graphs in 12 Graph Sets

Graph Set Name	Original Loop Parallelism	Computation Time (s)				Average Loop Parallelism
		< 1s	1s-5s	5s-60s	≥ 60s	
SET1	1	88.57%	3.33%	2.38%	5.71%	5
SET2	1	88.10%	3.81%	2.38%	5.71%	4
SET3	1	90.95%	2.86%	1.90%	4.29%	5
SET4	1	91.90%	3.33%	1.43%	3.33%	5
SET5	1	87.14%	3.33%	1.90%	7.62%	5
SET6	1	90.48%	2.86%	0.00%	6.67%	4
SET7	1	91.90%	3.33%	0.95%	3.81%	5
SET8	1	92.38%	1.90%	0.48%	5.24%	5
SET9	1	91.90%	4.29%	0.00%	3.81%	4
SET10	1	87.14%	4.76%	2.86%	5.24%	5
SET11	1	88.57%	2.86%	1.43%	7.14%	5
SET12	1	87.62%	5.24%	1.43%	5.71%	5

average transformed loop parallelism for all graphs in each graph set is 4 or 5, while the original loop parallelism of each graph is 1. Therefore, our technique can effectively enhance the parallelism. (For details about the graph sets and the experimental results, please refer to Supplementary File S6, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.171.)

6 Conclusion and Future Work

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 solved the dependence migration problem that is to find a retiming value of each node in a given dependence graph such that the minimum nonzero 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 and a set of graphs generated by TGFF. 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 combine this approach with loop tiling to solve iteration-level parallelism of nested loops where more synchronization may occur. Third, we will study how to integrate our technique into a compiler and evaluate it based on a multicore processor.

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), the Innovation and Technology Support Programme of Innovation and Technology Fund of the Hong Kong Special Administrative Region, China (ITS/082/10), and the Hong Kong Polytechnic University (A-PJ17 and 1-ZV5S). This version is a revised version. A preliminary version of this work appears in the Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2009) [1].

References

[1] D. Liu, Z. Shao, M. Wang, M. Guo, and J. Xue, "Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*, pp. 67-76, Oct. 2009.

[2] P.-C. Yew, "Is There Exploitable Thread-Level Parallelism in General-Purpose Application Programs?," *Proc. 17th Int'l Symp. Parallel and Distributed Processing (IPDPS '03)*, p. 160.1, 2003.

[3] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda, "Generalized Multipartitioning of Multi-Dimensional Arrays for Parallelizing Line-Sweep Computations," *J. Parallel and Distributed Computing*, vol. 63, no. 9, pp. 887-911, 2003.

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

[5] J. Xue, *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

[6] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien, "Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation," *Parallel Computing*, vol. 24, nos. 3/4, pp. 421-444, 1998.

[7] A. Aiken and A. Nicolau, "Optimal Loop Parallelization," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 308-317, 1988.

[8] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[9] D.-K. Chen and P.-C. Yew, "Redundant Synchronization Elimination for Doacross Loops," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 5, pp. 459-470, May 1999.

[10] W.-L. Chang, C.-P. Chu, and M. Ho, "Exploitation of Parallelism to Nested Loops with Dependence Cycles," *J. Systems Architecture*, vol. 50, no. 12, pp. 729-742, 2004.

[11] N. Manjikian and T.S. Abdelrahman, "Fusion of Loops for Parallelism and Locality," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 2, pp. 193-209, Feb. 1997.

[12] L. Bic, J.M.A. Roy, and M. Nagel, "Exploiting Iteration-Level Parallelism in Dataflow Programs," *Proc. 12th Int'l Conf. Distributed Computing Systems*, pp. 376-381, June 1992.

[13] D.A.P. Haiek, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," PhD dissertation, Dept. of Computer Science, Univ. of Illinois, Nov. 1979.

[14] R.G. Cytron, "Compile-Time Scheduling and Optimizations for Multiprocessor System," PhD dissertation, Univ. of Illinois, Sept. 1984.

[15] D.-K. Chen, J. Torrellas, and P.-C. Yew, "An Efficient Algorithm for the Run-Time Parallelization of Doacross Loops," *Proc. Conf. Supercomputing (Supercomputing '94)*, pp. 518-527, 1994.

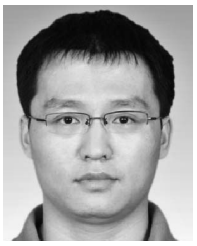
[16] C.-P. Chu and D.L. Carver, "Reordering the Statements with Dependence Cycles to Improve the Performance of Parallel Loops," *Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS '97)*, pp. 322-328, 1997.

[17] Z. Li, P.-C. Yew, and C.-Q. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 26-34, Jan. 1990.

[18] D. Padua, D. Kuck, and D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers*, vol. C-29, no. 9, pp. 763-776, Sept. 1980.

[19] C.D. Polychronopoulos, "Advanced Loop Optimizations for Parallel Computers," *Proc. First Int'l Conf. Supercomputing*, pp. 255-277, Mar. 1988.

[20] C.D. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. Computers*, vol. 37, no. 8, pp. 991-1004, Aug. 1988.
[21] C.-M. Wang and S.-D. Wang, "Compiler Techniques to Extract Parallelism within a Nested Loop," *Proc. 15th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '91)*, pp. 24-29, Sept. 1991.
[22] W. Shang, M.T. O'Keefe, and J.A.B. Fortes, "On Loop Transformations for Generalized Cycle Shrinking," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 2, pp. 193-204, Feb. 1994.
[23] J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," *IEEE Trans. Computers*, vol. 38, no. 8, pp. 1203-1211, Aug. 1989.
[24] J.K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," PhD dissertation, Univ. of Illinois, 1986.
[25] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5-35, 1991.
[26] A. Darte, G.-A. Silber, and F. Vivien, "Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling," *Parallel Processing Letters*, vol. 7, no. 4, pp. 379-392, 1998.
[27] E.H.-M. Sha, C. Lang, and N.L. Passos, "Polynomial-Time Nested Loop Fusion with Full Parallelism," *Proc. Int'l Conf. Parallel Processing (ICPP '96)*, vol. 3, pp. 9-16, Aug. 1996.
[28] T. O'Neil and E.H.-M. Sha, "Retiming Synchronous Data-Flow Graphs to Reduce Execution Time," *IEEE Trans. Signal Processing*, vol. 49, no. 10, pp. 2397-2407, Oct. 2001.
[29] L.-S. Liu, C.-W. Ho, and J.-P. Sheu, "On the Parallelism of Nested for-loops Using Index Shift Method," *Proc. Int'l Conf. Parallel Processing (ICPP '90)*, pp. 119-123, 1990.
[30] Y. Robert and S.W. Song, "Revisiting Cycle Shrinking," *Parallel Computing*, vol. 18, no. 5, pp. 481-496, 1992.
[31] A. Darte and G. Huard, "Complexity of Multi-Dimensional Loop Alignment," *Proc. 19th Ann. Symp. Theoretical Aspects of Computer Science (STACS '02)*, pp. 179-191, 2002.
[32] R.P. Dick, D.L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," *Proc. Sixth Int'l Workshop Hardware/Software Codesign (CODES/CASHE '98)*, pp. 97-101, 1998.
[33] D.N. Jayasimha and J.D. Martens, "Some Architectural and Compilation Issues in the Design of Hierarchical Shared-Memory Multiprocessors," *Proc. Sixth Int'l Parallel Processing Symp.*, pp. 567-572, Mar. 1992.
[34] L.-F. Chao, "Scheduling and Behavioral Transformations for Parallel Systems," PhD dissertation, Dept. of Computer Science, Princeton Univ., 1993.
[35] T.W. O'Neil, S. Tongsima, and E.H.-M. Sha, "Extended Retiming: Optimal Scheduling via a Graph-Theoretical Approach," *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP '99)*, pp. 2001-2004, 1999.
[36] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing*, vol. 1, no. 2, pp. 146-160, 1972.
[37] D.-K. Chen and P.-C. Yew, "Statement Re-Ordering for Doacross Loops," *Proc. Int'l Conf. Parallel Processing (ICPP '94)*, vol. 2, pp. 24-28, Aug. 1994.
[38] A. Fraboulet and A. Mignotte, "Source Code Loop Transformations for Memory Hierarchy Optimizations," *Proc. Workshop Memory Access Decoupled Architecture*, pp. 8-12, Sept. 2001.
[39] K. Okuda, "Cycle Shrinking by Dependence Reduction," *Proc. Second Int'l Euro-Par Conf. Parallel Processing (Euro-Par '96)*, pp. 398-401, 1996.
[40] *ILOG CPLEX 11.2 Users Manual*, ILOG SA, Gentilly, 2008.



Duo Liu received the BE degree in computer science from the Southwest University of Science and Technology, Sichuan, China, in 2003, and the ME degree from the Department of Computer Science, University of Science and Technology of China, Hefei, China, in 2006, respectively. He is currently working toward the PhD degree in the Department of Computing at the Hong Kong Polytechnic University. His current research interests include emerging memory techniques for embedded systems and high-performance computing for multicore processors.

Yi Wang received the BE and ME degrees in electrical engineering from Harbin Institute of Technology, China, in 2005 and 2008, respectively. He is currently working toward PhD degree in the Department of Computing at the Hong Kong Polytechnic University. His research interests include embedded systems and real-time scheduling for multicore systems.

Zili Shao received the BE degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the MS and PhD degrees from the Department of Computer Science, University of Texas at Dallas, in 2003 and 2005, respectively. He has been an associate professor with the Department of Computing, Hong Kong Polytechnic University, since 2010. His research interests include embedded software and systems, real-time systems, and related industrial applications. He is a member of the IEEE.

Minyi Guo received the BSc and ME degrees in computer science from Nanjing University, China, and the PhD degree in computer science from the University of Tsukuba, Japan. He is currently distinguished professor and head of the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU), China. Before joined SJTU, he had been a professor and department chair of School of Computer Science and Engineering, University of Aizu, Japan. He received the national science fund for distinguished young scholars from NSFC in 2007. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, pervasive computing, and cloud computing. He has more than 250 publications in major journals and international conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, *the IEEE Transactions on Nanobioscience*, t*he IEEE Transactions on Computers*, *the ACM Transactions on Autonomous and Adaptive Systems*, *the Journal of Parallel and Distributed Computing*, INFOCOM, IPDPS, ICS, CASES, ICPP, WWW, PODC, etc. He received five best paper awards from international conferences. He is on the editorial board of *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*. He is a senior member of the IEEE, member of The ACM, IEICE IPSJ, and CCF.

Jingling Xue received the BSc and MSc degrees in computer science from Tsinghua University, Tsinghua, China, in 1984 and 1987, respectively, and the PhD degree in computer science from Edinburgh, United Kingdom, in 1992. He is currently a professor with the School of Computer Science and Engineering, University of New South Wales, Sydney, Australia. His current research interests include programming languages, compiler optimizations, program analysis, high-performance computing and embedded systems. He is a senior member of the IEEE.