

Optimization of Nest-Loop Software Pipelining *

Qingfeng Zhuge, Zili Shao, Edwin H.-M. Sha

Department of Computer Science

University of Texas at Dallas

Richardson, Texas 75083, USA

{qfzhuge, zxs015000, edsha}@utdallas.edu

Abstract

Embedded systems have strict timing and code size requirements. Software pipelining is one of the most important optimization techniques to improve the execution time of loops by increasing the parallelism among successive loop iterations. However, little research has been done for the software pipelining problem on nested loops. The existing software pipelining techniques for single loops can only explore the innermost loop parallelism of a nested loop, so the final timing performance is inferior. While multi-dimensional (MD) retiming can explore the outer loop parallelism, it introduces large overheads in loop index generation and code size due to transformation. In this paper, we propose theory and algorithms of software pipelining for nested loops with minimal overheads based on the fundamental understanding of the properties of software pipelining for nested loops. We show the relationship among execution sequence, execution time of loop body, and software pipelining degree of a nested loop using retiming concepts. Two algorithms of Software Pipelining for NEsted loops (SPINE) are proposed: The SPINE-FULL algorithm generates fully parallelized loops with computation and code size overheads as small as possible. The SPINE-ROW-WISE algorithm generates the best possible parallelized loops with the minimal overheads. Our technique can be directly applied to imperfect nested loops. The experimental results show that the average improvement on the execution time of the pipelined loop generated by SPINE is 71.7% compared with that generated by the standard software pipelining approach. The average code size is reduced by 69.5% compared with that generated by MD retiming.

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, and NSF CCR-0309461

1 Introduction

Embedded systems usually have stringent requirements in timing and code size. With the advance of the technology, embedded systems with multiple cores or VLIW-like architectures, such as TI's TMS320C5x, Philips' TriMedia, and IA64, etc., become necessary to achieve the required high performance for the applications with growing complexity. Meanwhile, the capacity of on-chip memory modules is still very limited due to the chip size, cost and power considerations. The designers try their best to fit the code into the small on-chip memory to avoid slow (external) memory accesses. Hence, code size is a critical concern for many embedded processors.

To reduce the execution time of loops, software pipelining is widely used to explore the instruction-level parallelism in a loop by parallelizing the execution of successive iterations [6, 12]. However, software pipelining can dramatically expand the code size by adding code sections in prologue and epilogue ¹ [14, 17]. If the software-pipelined code cannot be fit into on-chip memory, a designer, without proper techniques, may have to give up using software pipelining, resulting in a design with a deteriorated timing performance. This awkward situation still exists in leading industry such as Texas Instruments [5] because there is no effective design tools to consider the code size issue along with timing optimization. Therefore, optimization with both timing and code size requirements becomes a great challenge for embedded system design.

Timing and code size issues are even greater concerns for software pipelining on nested loops. Unlike software pipelining of single loops which has been extensively studied and implemented [2, 6, 12, 17], very few work has been done for the software pipelining problem on nested loops. A few existing techniques that could be applied to nested loop optimization either cannot fully explore the embedded parallelism in a nested loop or do not consider the overheads such as loop index and loop bounds generation, and code size expansion due to transformation.

Software pipelining for single loops focuses on one-dimensional problems. When applied to nested loops, it only optimizes the innermost loop [1, 2, 6, 12]. While nested loops usually exhibit dependencies cross loop dimensions. They provide abundant opportunities to achieve full parallelism, that is, all the operations in one iteration can be executed in parallel [9]. Therefore, the performance improvement that can be obtained using software pipelining for single loops is very limited. For example, the execution time of the Floyd-Steinberg algorithm generated by the Modulo scheduling [12] is 25000 time units according to our experimental results, while it is only 2950 time units when generated by our Software Pipelining for NEsted loops (SPINE) technique. The improvement on the loop execution time is 88.2%. It indicates that a lot of potential parallelism cannot be explored by software pipelining for single loops.

¹The code sections added before or after loop body to preserved the correct execution of software-pipelined loop.

Another technique called hyperplane scheduling [4, 11] tries to convert a nested loop into a single loop using loop unrolling and skewing. However, this technique makes code generation extremely difficult, and results in large overheads due to transformations. The best effort existing in industry on nested loop pipelining is to overlap the executions of the prologue and epilogue of the innermost loop, called outer loop pipelining [8, 15]. In this method, the dependencies among the outer loop iterations are still not changed. Hence, the potential parallelism that can be explored is very limited.

The only existing method that can fully explore the potential parallelism in multi-dimensional problems is multi-dimensional (MD) retiming [9]. However, MD retiming aims to achieve full parallelism, but does not consider the overheads in loop index generation and code size. The regular row-wise execution sequence, which is implemented in most nested loops, can be altered unnecessarily in the final code. As a result, transformations need to be performed to compute new loop indexes and loop bounds due to a skewed execution sequence [9, 16]. The code size is greatly expanded. Data locality is also disrupted. According to our experimental results, the code size of Floyd-Steinberg algorithm generated by MD retiming is 1646 instructions, while it is only 169 instructions using our SPINE technique. The code size is reduced by 89.7%. The execution time is also reduced by 31.6%. Although MD retiming can benefit high-level synthesis with specialized hardware support, it is not suitable for software pipelining on nested loops.

The following example shows how a skewed execution sequence significantly affects the performance and code size of the generated code. Figure 1(a) shows the original code of a nested loop. Figure 1(c) shows the pipelined loop generated by the standard MD retiming technique [9]. It uses a diagonal execution sequence to achieve full parallelism. The code size grows dramatically not only because prologue and epilogue sections are produced in both loop levels, but also because the extra codes are generated for new loops bounds and indexes computation. These extra computations deteriorate the performance and code quality of the final code. Due to the space limitation, we cannot show the whole piece of the program. Figure 1(b) shows the software-pipelined code generated by our SPINE algorithm using row-wise execution. The loop is fully parallelized. Assuming that each computation in the loop body can be executed in one time unit, then, the execution time of one iteration of the pipelined loop is just one time unit. The code size is just slightly larger than the original code, and can be further reduced by code size reduction technique presented in [17].

In our research, we reveal the relationship among execution sequence, execution time of loop body, and software pipelining degree. We show that a full-parallel nested loop with minimal overheads can be generated by using our technique that is based on the fundamental understanding of the properties of nest-loop software pipelining. We make our contributions as follows:

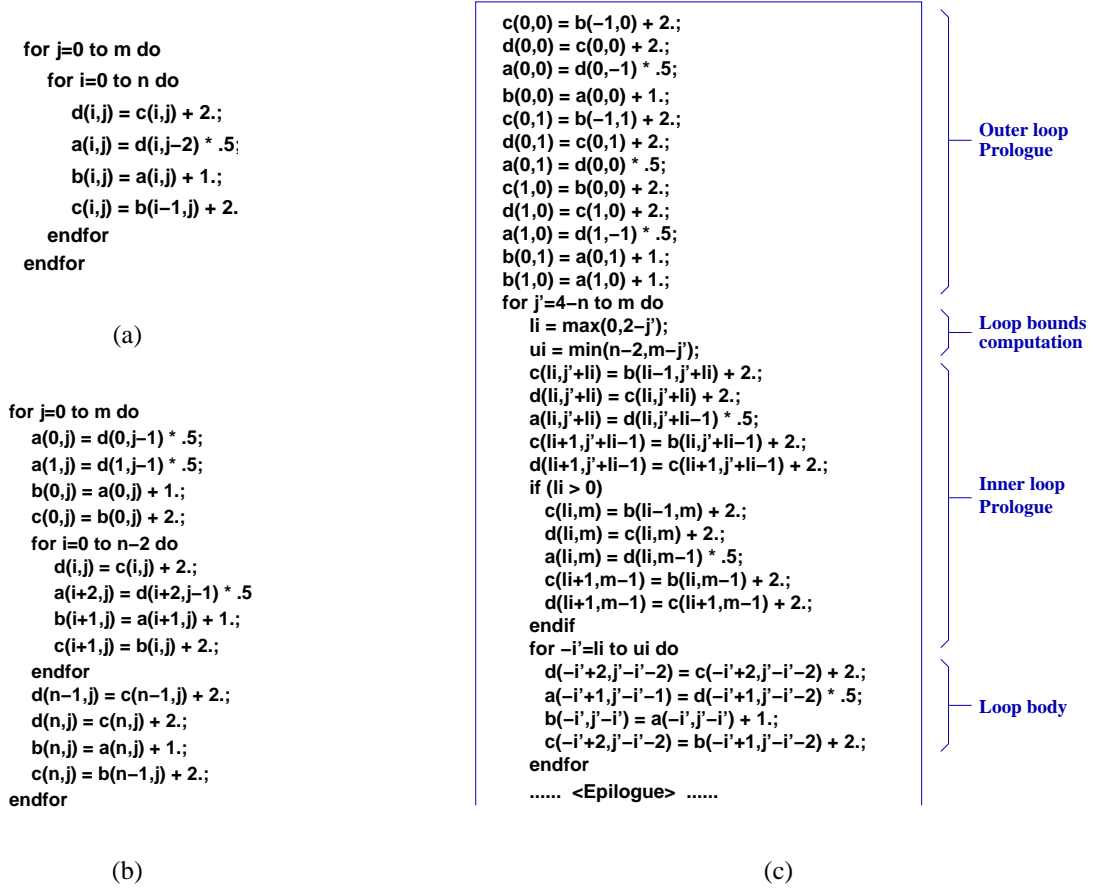


Figure 1: (a) The original code. (b) The fully-pipelined code generated by SPINE. (c) The fully-pipelined code generated by chained MD retiming.

1. We use multi-dimensional retiming concept to effectively model the software pipelining problem on nested loops. (Section 2.2).
2. Based on this model, we present the theory of software pipelining for nested loops to reveal the relationship among execution time of loop body, software pipelining degree, and execution sequence of loop iterations. (Section 3)
3. We propose a delay merge technique that identifies all the nested loops that are possible to be fully parallelized with a row-wise execution sequence. (Theorem 3.1)
4. We present the mathematical formula to compute the minimum execution time of the loop body in a nested loop with given execution sequence and retiming vector. (Theorem 3.2 and 3.3)

5. We prove that the best execution time of loop body can be achieved by using a retiming vector that is orthogonal to a given execution sequence. (Theorem 3.4)
6. We present two algorithms of Software Pipelining for NEsted loops (SPINE) technique: (Section 4)
 - The SPINE-FULL algorithm fully parallelizes a nested loop, and tries to use a row-wise execution sequence whenever it is possible to reduce the overheads.
 - The SPINE-ROW-WISE algorithm aims to achieve the best possible parallelized result of a nested loop with a fixed row-wise execution sequence. Therefore, transformation and code size overheads are minimal.
7. Our technique can be directly applied to loops with codes between loop levels (imperfect nested loops). (Section 4)

We conduct experiments on a set of two-dimensional benchmarks to compare the code quality generated by SPINE with that generated by the standard software pipelining, and MD retiming. Our experimental results show that SPINE out-performs or ties both of the other two techniques on all of our benchmarks. The average improvement on the execution time of the pipelined loop generated by SPINE is 71.7% compared with that generated by the standard software pipelining approach. The average code size is reduced by 69.5% compared with that generated by MD retiming. Based on the result of this paper, the future research can be extended to other optimization objectives of nested loops, such as low power scheduling, address register allocation, etc.

The rest of the paper is organized as follows: Section 2 overviews the necessary backgrounds related to SPINE. Section 3 presents the theory of software pipelining for nested loops. Section 4 presents the SPINE algorithms and an illustrative example. We also show that our technique can be applied to imperfect nested loops. Section 5 presents the experimental results. The last section, Section 6, concludes the paper.

2 Basic Principles

In this section, we give an overview of basic concepts and principles related to software pipelining problem for nested loops. These include multi-dimensional data flow graph, multi-dimensional retiming, and software pipelining. We demonstrate that retiming and software pipelining are essentially the same concept. A discussion of the limitations of the existing techniques for optimizing nested loops is also provided in Section 2.3.

2.1 Multi-Dimensional Data Flow Graph

A *multi-dimensional data flow graph (MDFG)* $G = \langle V, E, \mathbf{d}, t \rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \subseteq V * V$ is the set of edges representing dependencies, \mathbf{d} is a function from E to \mathbb{Z}^n , representing the multi-dimensional delays between two nodes, where n is the number of dimensions, and t is a function from V to a set of positive integers, representing the computation time of each node.

Programs with nested loops can be represented by an MDFG with cycles as shown in Figure 2(a). An *iteration* is the execution of each node in V exactly once. Iterations are identified by a vector index $\mathbf{i} = (i_1, i_2, \dots, i_n)$, starting from $(0, 0, \dots, 0)$, where elements are ordered from the innermost loop to outermost loop. Inter-iteration dependencies are represented by edges with delays. In particular, an edge $e(u \rightarrow v)$ with delay $\mathbf{d}(e)$ indicates that the computation of node v at \mathbf{j}^{th} iteration requires data produced by node u at $(\mathbf{j} - \mathbf{d}(e))^{\text{th}}$ iteration. An inter-iteration dependency with all zero elements but the first element is called inner-loop dependency, which is the same as the dependency of a single loop. Otherwise, it is called an outer-loop dependency. In the case of nested loops, The dependencies within the same iteration are represented by edges without delay ($\mathbf{d}(e) = (0, 0, \dots, 0)$). A static schedule must obey these intra-iteration dependencies. The *cycle period* of a DFG is defined as the computation time of the longest zero-delay path, which corresponds to the minimum schedule length when there is no resource constraint. We assume the computation time of a node is 1 time unit in this paper. Thus, the cycle period of the DFG in Figure 2(a) is 3.

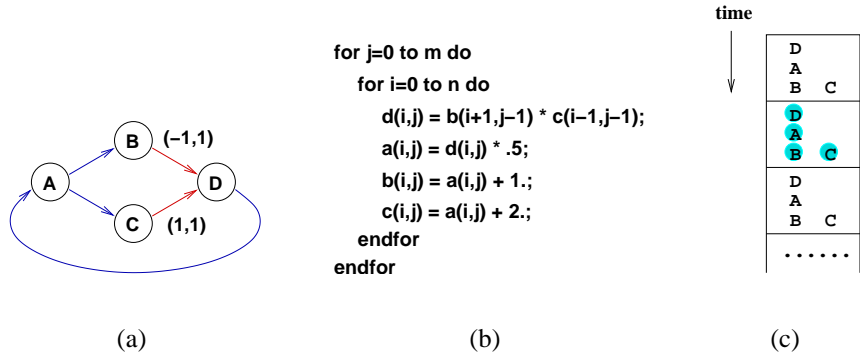


Figure 2: (a) An MDFG. (b) Code of the nested loop. (c) The static schedule.

The iteration space of the two-dimensional data flow graph show in Figure 2(a) is shown in Figure 3(a). The marked cell is iteration $(0, 0)$. The inter-iteration dependencies $B \rightarrow D$ and

When a delay is pushed through node D to its outgoing edge as shown in Figure 4(a), the actual effect on the schedule of the new MDFG is that the i^{th} copy of D is shifted up and is executed with $(i - (1, 0))^{\text{th}}$ copy of nodes A, B, and C. Because there is not dependency between node D and A in the new loop body, these two nodes can be executed in parallel. The schedule length of the new loop body is then reduced from three control steps to two control steps. This transformation is illustrated in Figure 2(c) and Figure 4(c). Note that the original zero-delay edge $D \rightarrow A$ in Figure 2(a) has a delay $(1, 0)$ after retiming. The cycle period is then reduced from 3 to 2 time units.

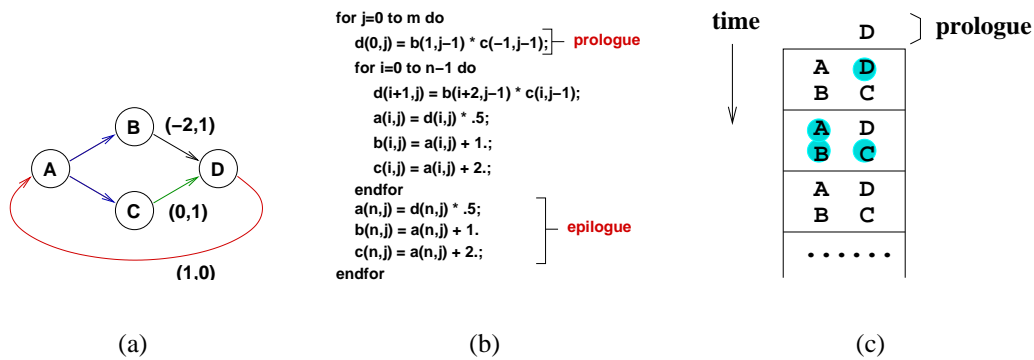


Figure 4: (a) The retimed MDFG G_r of MDFG in Figure 2(a) with $r(D) = (1, 0)$. (b) Code of the retimed MDFG.

In fact, every retiming operation corresponds to a software pipelining operation. When $r(u)$ delay units are pushed through a node u , every copy of this node is moved by $r(u)$ iterations. Hence, a new iteration consists of nodes from different iterations. Some nodes are shifted out of the loop body to be the *prologue* to provide the necessary data for the iterative process. Correspondingly, some nodes will be executed after loop body to complete the process. These are called *epilogue*. With MD retiming function r , we can trace the pipelined nodes and also measure the size of prologue and epilogue [17]. When $r(v) = (r_x, r_y)$ delays are pushed through node v , there are r_x copies of node v appeared in the prologue in x-dimension, and r_y copies of node v in the prologue in y-dimension. The number of copies of a node in the epilogue can also be derived in a similar way. For example, Figure 2(a) shows the MDFG of the code in Figure 2(b). Figure 4(a) shows the retimed graph for the software-pipelined loop in Figure 4(b) with $r(D) = (1, 0)$. We can see from the iteration space shown in Figure 5(a) that there is exactly one copy of node D in the prologue in x-dimension. The corresponding cell dependency graph is shown in Figure 5(b).

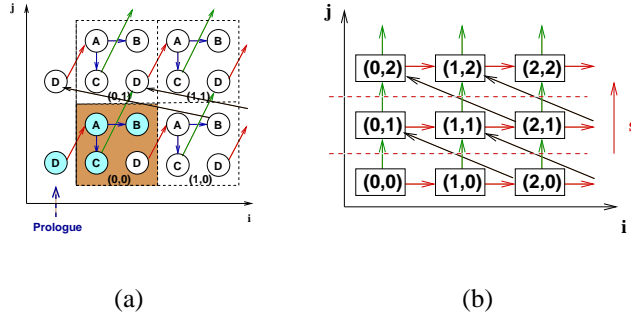


Figure 5: (a) The iteration space the retimed MDFG in Figure 4(a). (b) The cell dependency graph.

A MDFG can always be fully parallelized by using MD retiming [9], which means that all the nodes in the MDFG can be executed in parallel if there is no resource constraint. For example, Figure 6(a) and Figure 6(b) show the fully-parallelized MDFG and its static schedule, respectively, with retiming function $r(D) = (2, 0)$ and $r(A) = (1, 0)$.

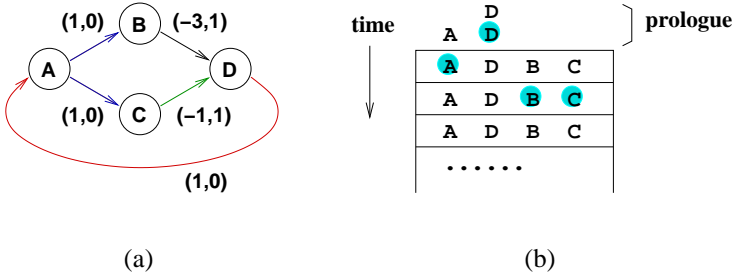


Figure 6: (a) A fully-parallelized graph of the MDFG in Figure 2(a). (b) The static schedule.

However, an arbitrarily chosen retiming may cause cycles in CDG of a multi-dimensional data flow graph, which is not executable. An illegal retiming is shown in Figure 7(a). By simple inspection of the CDG in Figure 7(b), we notice that a cycle is created by the dependencies $(1,0)$ and $(-1,0)$.

2.3 Limitations of Existing Techniques

The standard software pipelining techniques [12, 13] can only be used to optimize the innermost loop, or to optimize in one loop dimension when loop index exchange can be employed to switch

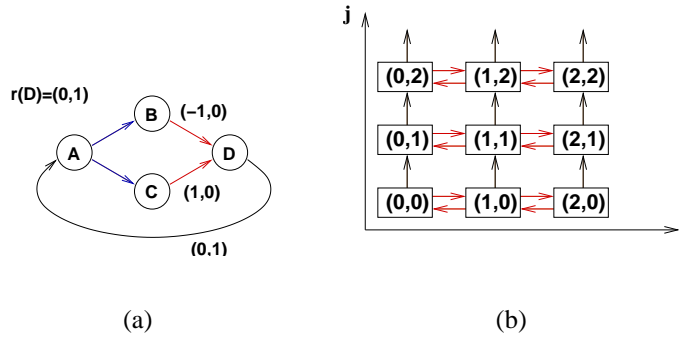


Figure 7: (a) An illegal retiming for the MDFG in Figure 2(a). (b) Cell dependency graph with cycles.

the index of any loop dimension to the index of the innermost loop. However, this approach produces inferior results when applied to nested loops because the parallelism to be explored in a single loop dimension is very limited. Consider the MDFG example in Figure 8(a) with a cycle period of three time units. The best result can be achieved by standard software pipelining is a cycle period of two time units, as shown in Figure 8(c). While a fully parallel MDFG can be generated by using MD retiming as shown in Figure 8(d).

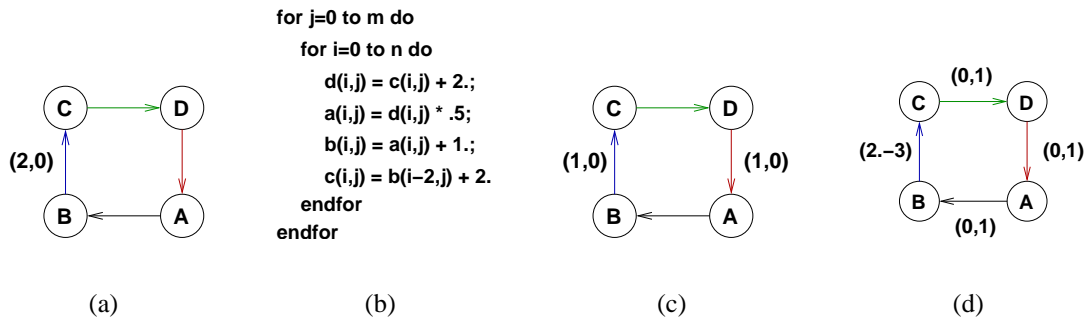


Figure 8: (a) An MDFG. (b) The nested loop. (c) Using the standard software pipelining. (d) A fully-parallelized MDFG.

Another interesting example is shown in Figure 9(a) which has two orthogonal delay vectors $\mathbf{d}(B \rightarrow C) = (1, 0)$ and $\mathbf{d}(D \rightarrow A) = (0, 1)$. The cell dependency graph is shown Figure 9(b). In this case, the standard software pipelining can do nothing to optimize the loop even with loop index interchange. However, it can still be fully parallelized by the standard MD retiming, such

as the chained MD retiming [9], as shown in Figure 10(a).

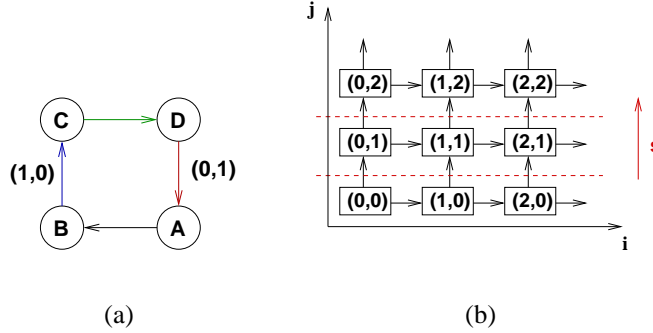


Figure 9: (a) An MDFG. (b) The cell dependency graph.

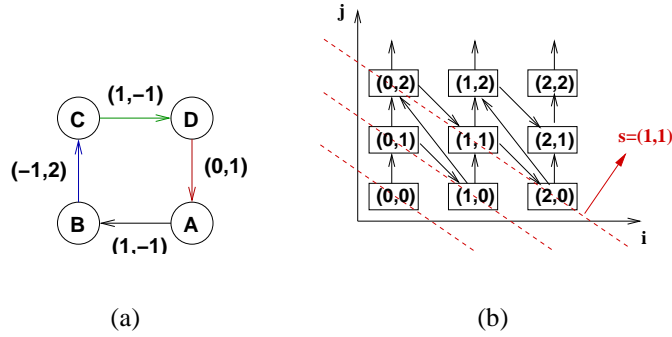


Figure 10: (a) A fully-parallelized MDFG using chained MD retiming. (b) The cell dependency graph.

The chained MD retiming algorithm selects a legal schedule vector \mathbf{s} of an MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, such that $\mathbf{s} \cdot \mathbf{d}(e) > 0, \forall e \in E$, and a unit retiming vector ² \mathbf{r}_u that is orthogonal to \mathbf{s} . Then, a topological sort is conducted on the subgraph of the MDFG that contains only the zero-delay edges, which is a directed acyclic graph (DAG). Each node is labeled by its level number l according to the topological sort. Then, it retimes each node v by retiming vector $(k - l) \cdot \mathbf{r}_u$, where k is the length of the longest path. For the example in Figure 9(a), chained MD retiming chooses schedule vector $\mathbf{s} = (1, 1)$ and unit retiming vector $\mathbf{r}_u = (1, -1)$. Therefore, retiming functions are $\mathbf{r}(A) = (1, -1)$ and $\mathbf{r}(C) = (1, -1)$.

²Elements of a unit retiming vector have no common divisor greater than 1.

By observing the CDG in Figure 10(b), we notice that row-wise execution cannot correctly execute the retimed MDFG any more. A legal schedule vector of the retimed graph is $\mathbf{s} = (1, 1)$, instead of the regular schedule vector $\mathbf{s} = (0, 1)$. No matter how slight the change is, a skewed schedule vector results in sever overheads for a software-pipelined loop. First, the code size is expanded drastically because the large code sections of prologue and epilogue produced in both loop dimensions as shown in the resulting code in Figure 1(c) and the iteration space shown in Figure 11. Compared with the original code size in Figure 1(a), the code size of pipelined loop generated by chained MD retiming is excessively large. Second, the computation of the new loop bounds introduces large computation overhead and extra codes. The extra code for loop bounds computation is indicated in the code shown in Figure 1(c). And third, the original data locality is disrupted because of the skewed execution sequence. As a result, address generation becomes much more complicated. Therefore, the existing MD retiming techniques, although achieves full parallelism in an iteration, are not suitable for software pipelining of nested loops because of high computation and code size overheads.

Now the question is if it is possible to keep the row-wise execution while achieving the optimal cycle period, i.e. one time unit, to execute the loop body. The answer is “yes”. And we will show the provable result in the next section.

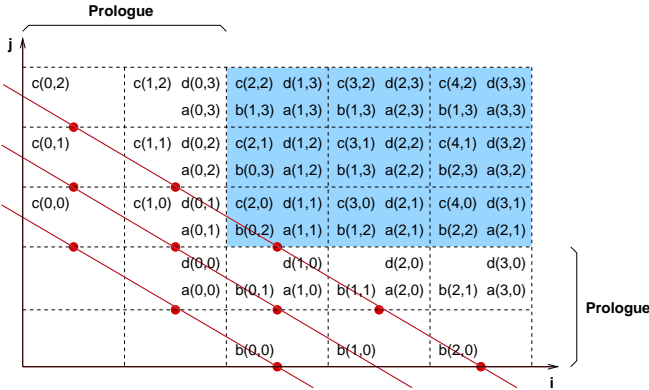


Figure 11: The iteration space of the nested loop in Figure 1(c).

3 Theory of Software Pipelining for Nested Loops

In this section, we present the theoretical foundation of software pipelining of nested loops with two loop levels based on retiming concept. We study the timing property of cycles in an MDFG

considering both schedule vector and unit retiming vector. Although the theorems are derived for two-dimensional, unit-time MDFGs, they can be easily generalized to multi-dimensional, general-time cases. Before the theorems are presented, it's necessary to give several definitions, assumptions.

Definition 3.1. *Given a schedule vector \mathbf{s} and a unit retiming vector \mathbf{r}_u . Let ℓ be a cycle in an MDFG. The minimum cycle time of cycle ℓ , $C_{\min}(\ell)$, is the minimum execution time of ℓ with schedule vector \mathbf{s} via MD retiming.*

Definition 3.2. *Given a schedule vector \mathbf{s} and a unit retiming vector \mathbf{r}_u . Let ℓ be a cycle in an MDFG G . The schedule bound of G is $SB(G) = \max_{\ell \in \text{in } G} C_{\min}(\ell)$.*

In addition, the following concepts are frequently mentioned in the theorems. The *total delay vector* of a cycle ℓ in an MDFG G is denoted by $\mathbf{D}(\ell) = \sum_{e \in \ell} \mathbf{d}(e)$, and the *total computation time* of a cycle is denoted by $T(\ell) = \sum_{v \in \ell} t(v)$. Therefore, *cycle period* $c(G) = \max_{\ell \in \text{in } G} T(\ell)$.

For sake of a clear presentation, we make the following assumptions without losing generality. First, all the elements of a schedule vector are nonnegative integers. Second, for delay vectors that are orthogonal to \mathbf{s} , only one of them is legal. For instance, for a loop with schedule vector $\mathbf{s} = (0, 1)$, the existence delay vectors $\mathbf{d} = (1, 0)$ and $\mathbf{d} = (-1, 0)$, causes cycles in cell dependency graph. Hence, we chose delay vector $\mathbf{d} = (1, 0)$ as a legal delay based on the fact that loop counters usually increase in a program.

Because the timing and code size of software-pipelined loops are greatly affected by schedule vector and unit retiming vector, we need to study the timing property of cycles in an MDFG with given schedule vector \mathbf{s} and unit retiming vector \mathbf{r}_u . We start with schedule vector $\mathbf{s} = (0, 1)$ and unit retiming vector $\mathbf{r}_u = (1, 0)$ because they generate code with the minimal overheads. The following theorem states that the minimum cycle time $C(\ell) = 1$ is not always achievable with this setting. Furthermore, the minimum cycle time can be computed before performing MD retiming.

Theorem 3.1. *Given a unit-time MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, a schedule vector $\mathbf{s} = (0, 1)$, and a unit retiming vector $\mathbf{r}_u = (1, 0)$. Let ℓ be a cycle in G , and e be the only weighted edge in the cycle with delay vector $\mathbf{d}(e)$.*

1. *If $\mathbf{d}(e) = (k, 0)$, for $k > 0$, the minimum cycle time $C_{\min}(\ell) = \lceil T_\ell/k \rceil$ can be achieved via MD retiming.*
2. *If $\mathbf{d}(e) = (i, j)$, $j > 0$, the minimum cycle time $C_{\min}(\ell) = 1$ can be achieved via MD retiming,*

The first statement of Theorem 3.1 indicates the MDFGs with only delay vectors in form of $\mathbf{d}(e) = (k, 0)$. Then, the minimum cycle time can be computed as $C_{\min}(\ell) = \lceil T_\ell/k \rceil$. When $k < T(\ell)$, we have $C_{\min}(\ell) > 1$, i.e., the schedule bound $SB(G) > 1$. In this case, the optimal cycle time $C_{\min}(\ell) = 1$ cannot be achieved, because the maximum retiming can be applied is $k \cdot \mathbf{r}_u$.

For example, the MDFG in Figure 8(a) has a $(2, 0)$ delay vector in the cycle. The minimum cycle time with schedule vector $\mathbf{s} = (0, 1)$ and unit retiming vector $\mathbf{r}_u = (1, 0)$ is $C_{\min}(\ell) = \lceil 3/2 \rceil = 2$. The maximum retiming is $2 \cdot \mathbf{r}_u$. That is, we cannot move more than two delay units through a node. Any retiming $m \cdot \mathbf{r}_u$ and $m > 2$ will generate cycles in cell dependency graph. An illegal retiming $3 \cdot \mathbf{r}_u$ is illustrated in Figure 12(a) and 12(b).

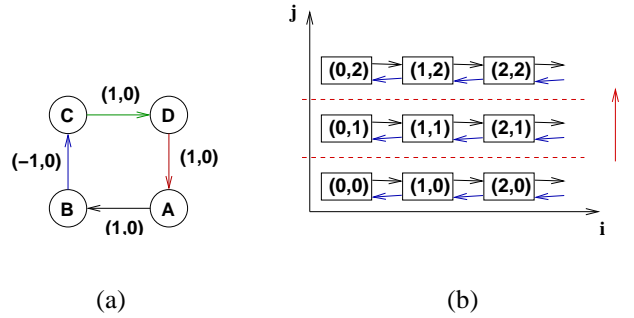


Figure 12: (a) An illegal retiming for MDFG in Figure 8(a) with $r(C)=(3,0)$, $r(D)=(2,0)$, and $r(A)=(1,0)$. (b) The cell dependency graph with cycles.

The second case of Theorem 3.1 states that a cycle can always be fully parallelized when there are delay vectors with positive y-dimension elements, because the legality rule can always be satisfied.

Theorem 3.1 inspires us to consider the total delay vector $D(\ell)$ of a cycle to decide whether the cycle can be fully parallelized using schedule vector $\mathbf{s} = (0, 1)$ and unit retiming vector $\mathbf{r}_u = (1, 0)$. We propose an idea of removing $(k, 0)$ delay vectors by merging delays via retiming. If $D(\ell) \neq (k, 0)$ after delay merge for cycle ℓ in an MDFG, it is always possible to fully parallelize a nested loop using schedule vector $\mathbf{s} = (0, 1)$. Hence, the optimal result of minimum cycle time $C_{\min}(\ell) = 1$ can be achieved.

Consider the example in Figure 9(a) again. Originally, chained MD retiming can only find a schedule vector $\mathbf{s} = (1, 1)$ and a unit retiming vector $\mathbf{r}_u = (1, -1)$ to fully parallelize the loop. The resulted code is deteriorated by huge computation overhead and code size expansion. Standard software pipelining for single loop cannot even do any optimization for this kind of

loop. However, after merging two delays in the MDFG via retiming $\mathbf{r}(C) = \mathbf{r}(D) = (1, 0)$, the retimed MDFG in Figure 13(a) only has one delay vector $\mathbf{d}(D \rightarrow A) = (1, 1)$. Now we can use schedule vector $\mathbf{s} = (0, 1)$ and retiming $\mathbf{r} = (1, 0)$ to retime the MDFG. The fully parallel MDFG is shown in Figure 13(b) which is retimed by retiming functions $\mathbf{r}(A) = (3, 0)$, $\mathbf{r}(B) = \mathbf{r}(C) = (2, 0)$, and $\mathbf{r}(D) = (1, 0)$. The cell dependency graph with row-wise execution sequence is shown Figure 13(c). It is clear that the execution time of one iteration is just one time unit. The loop transformation cost and code size expansion are minimal because only innermost loop produces prologue and epilogue as shown the code Figure 1(b). We can see that the final code size is substantially reduced compared with the code generated by chained MD retiming as shown in Figure 1(c). Besides, loop bounds and loop indexes can be easily computed according to retiming functions. The data locality is also maintained.

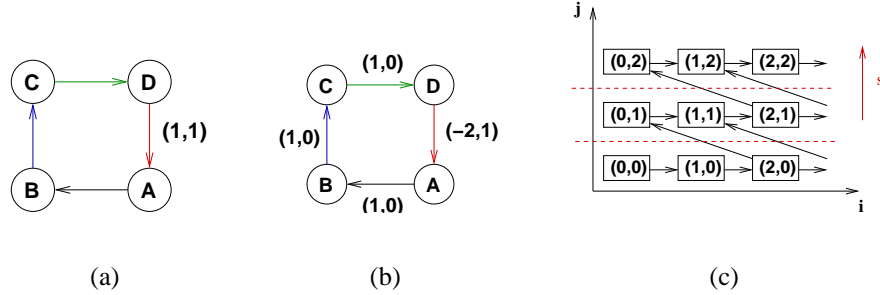


Figure 13: (a) Merging delays. (b) The fully parallel MDFG via retiming. (c) The cell dependency graph.

To fully understand the timing property of any given MDFG, schedule vector, and unit retiming vector, we further derive the following two theorems to compute the achievable minimum cycle time. Theorem 3.2 generalizes the result of Theorem 3.1 to any given schedule vector and a retiming vector that is orthogonal to \mathbf{s} . Theorem 3.3 further derives the minimum cycle time with any given \mathbf{s} and \mathbf{r}_u such that $\mathbf{s} \perp \mathbf{r}_u$.

Theorem 3.2. *Given a unit-time MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, a schedule vector \mathbf{s} , and a unit retiming vector \mathbf{r}_u such that $\mathbf{r}_u \perp \mathbf{s}$. Let ℓ be a cycle in G .*

1. *If there exists a weighted edge $e \in \ell$ such that the delay vector $\mathbf{d}(e) \parallel \mathbf{r}_u$, the minimum cycle time $C_{\min}(\ell) = 1$ can be achieved via MD retiming.*
2. *If $\mathbf{d}(e) \not\parallel \mathbf{r}_u, \forall e \in \ell$, then, the minimum cycle time $C_{\min}(\ell) = \lceil T(\ell) / (\mathbf{D}(\ell) / \mathbf{r}_u) \rceil$ can be achieved via MD retiming.*

Theorem 3.3. Given a unit-time MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, a schedule vector \mathbf{s} , and a unit retiming vector \mathbf{r}_u such that $\mathbf{s} \not\perp \mathbf{r}_u$. Let ℓ be a cycle in G . The minimum cycle time of ℓ via MD retiming is $C_{\min}(\ell) = \lceil T(\ell) / \min(g_\ell, T(\ell)) \rceil$ where $g_\ell = \lceil \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u} \rceil + \delta$, and δ can be computed as follows:

Case 1: if $\lceil \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u} \rceil \neq \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u}$, $\delta = 0$.

Case 2: if $\lceil \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u} \rceil = \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u}$, and $\mathbf{D}(\ell) \cdot \mathbf{x} - \lceil \frac{\mathbf{s} \cdot \mathbf{D}(\ell)}{\mathbf{s} \cdot \mathbf{r}_u} \rceil \cdot (\mathbf{r}_u \cdot \mathbf{x}) \leq 0$, $\delta = 0$.

Case 3: otherwise, $\delta = 1$.

Figure 14(a) shows an example with schedule vector $\mathbf{s} = (0, 1)$, unit retiming vector $\mathbf{r}_u = (1, 2)$, and a delay vector $\mathbf{d}(e) = (1, 3)$, the first case of Theorem 3.3 is applied. Then, we have $g_\ell = \lceil 3/2 \rceil = 2$. It turns out that there are at most two edges with delays in the cycle as shown in Figure 14(b).

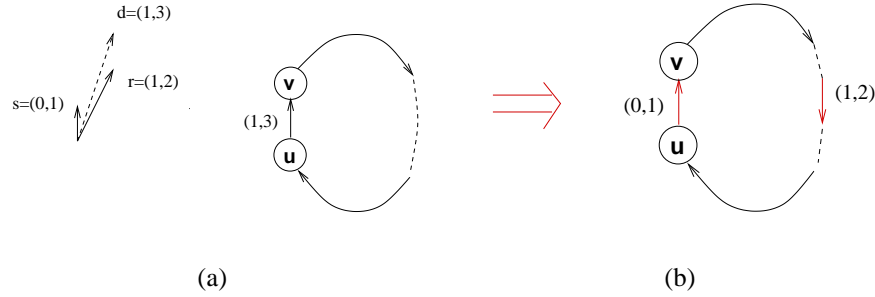


Figure 14: (a) A MDFG with given \mathbf{s} , \mathbf{r} , and $\mathbf{s} \not\perp \mathbf{r}$. (b) The retimed MDFG.

Due to the space limitation, we do not show the detail proofs of the above two theorems. They are easily verified on MDFGs.

From Theorem 3.1 to Theorem 3.3, we found that a unit retiming vector that is orthogonal to schedule vector \mathbf{s} can produce smaller minimum cycle time than those that are not orthogonal to \mathbf{s} . A proof is given for the following theorem that states the best choice of unit retiming vector.

Theorem 3.4. Given MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, and a schedule vector \mathbf{s} . Let ℓ be a cycle in G . Let \mathbf{r}_u and \mathbf{r}'_u be two unit retiming vectors such that $\mathbf{r}_u \perp \mathbf{s}$ and $\mathbf{r}'_u \not\perp \mathbf{s}$. The minimum cycle time of ℓ via MD retiming using unit retiming vector \mathbf{r}_u is always smaller than or equal to that using \mathbf{r}'_u .

Proof. Without loss of generality, assume that $\mathbf{s} = (0, 1)$.

Case 1. $\mathbf{D}(\ell) = (k, 0)$, $k > 0$.

For unit retiming vector $\mathbf{r}_u = (1, 0)$, $C_{\min}(\ell) = \lceil T(\ell)/k \rceil$.

For unit retiming vector $\mathbf{r}'_u \neq (1, 0)$, $C'_{\min}(\ell) = T(\ell)$.

Therefore, $C'_{\min}(\ell) \geq C_{\min}(\ell)$

Case 2. $\mathbf{D}_\ell = (i, j), j > 0$.

For $\mathbf{r}_u = (1, 0)$, $C_{\min}(\ell) = 1$.

For $\mathbf{r}'_u \neq (1, 0)$, $C'_{\min}(\ell) = \lceil T_\ell / \min(g_\ell, T_\ell) \rceil$, where $T_\ell \geq \min(g_\ell, T_\ell)$ (Theorem 3.3).

Therefore, $C'_{\min}(\ell) \geq C_{\min}(\ell)$

□

According to the theorems presented in this section, we can fully explore the potential parallelism in a nested loop with minimal overheads and maximum flexibility. We summarize Software Pipelining for NEsted loops (SPINE) technique as follows:

- There must exist a schedule vector \mathbf{s} , and a unit retiming vector \mathbf{r}_u orthogonal to \mathbf{s} , such that the retimed loop is fully parallelized.
- We should pick $\mathbf{r}_u \perp \mathbf{s}$, so that it gives the shortest schedule.
- If we fix $\mathbf{s} = (0, 1)$ and use $\mathbf{r}_u = (1, 0) \perp \mathbf{s}$, SPINE will find the best schedule with length equal to *schedule bound*. The computation cost due to loop transformation and the code size overhead of the software-pipelined loop is minimal.

4 SPINE Algorithms

In this section, we present two algorithms of nest-loop software pipelining. The SPINE-FULL algorithm generates fully-parallelized nested loops with computation and code size overheads as small as possible. It's interesting to see that chained MD retiming becomes a special case of the SPINE-FULL algorithm. The SPINE-ROW-WISE algorithm fixes row-wise execution sequence, while generates a software-pipelined loop schedule with length equal to the schedule bound. Although the algorithms are presented for two-dimensional loops, multi-dimensional problems can be solved in a similar way.

Algorithm 4.1 shows the procedure of SPINE-FULL. It first performs delay merge to remove as many $(k, 0)$ delays as possible. If there is no cycle with $(k, 0)$ delays after delay merge, the MDFG can be fully parallelized using schedule vector $\mathbf{s} = (0, 1)$. If there are cycles with $(k, 0)$ delays, but $T_\ell \leq k$, the loop can still be fully parallelized. Otherwise, it explores the opportunity of loop index interchange. When all the attempts fail, it chooses a schedule vector \mathbf{s} that generates the code size and computation overhead as small as possible. Therefore, chained MD retiming becomes a special case of the SPINE-FULL algorithm.

Algorithm 4.2 shows the general procedure of delay merge. It removes $(k, 0)$ delays by merging the delays via retiming. It carefully constructs a DAG composed of paths connecting one $(k, 0)$ delay edge and one (i, j) delay edge. Each round, it adds $(1, 0)$ a node via retiming. The algorithm quickly terminates when all the $(k, 0)$ delay edges have been considered.

Algorithm 4.1 Procedure of SPINE-FULL

Input: MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, schedule vector $\mathbf{s} = (0, 1)$.

Output: A fully parallel MDFG $G_r = \langle V, E, \mathbf{d}_r, t \rangle$ with the smallest possible execution time and code size.

Remove as many $(k, 0)$ delays as possible using delay merge (Algorithm 4.2).

if There is no cycle with $(k, 0)$ delay **then**

 Generate fully parallel MDFG via MD retiming. Exit.

end if

if There is cycle with $(k, 0)$ delay **then**

if $k \leq T(\ell)$ **then**

 Generate fully parallel MDFG via MD retiming. Exit.

else

 Perform loop index interchange.

if There is no cycle with $(k, 0)$ delay **then**

 Generate fully parallel MDFG via MD retiming. Exit.

end if

end if

end if

Perform chained MD retiming choosing a schedule vector \mathbf{s} such that $\mathbf{s} \cdot \mathbf{x} + \mathbf{s} \cdot \mathbf{y}$ is as small as possible.

The procedure of the SPINE-ROW-WISE algorithm is shown in Algorithm 4.3. The SPINE-ROW-WISE algorithm achieves the schedule bound of a nested loop with given schedule vector $\mathbf{s} = (0, 1)$ and unit retiming vector $\mathbf{r}_u = (1, 0)$. Because the schedule vector is fixed, we found a more efficient algorithm that even does not need to merge delays. It only consists of two steps. The first step constructs a subgraph by removing all the edges with delay $\mathbf{d} \neq (k, 0)$. Then, a one-dimensional retiming can be performed on the subgraph to achieve the schedule bound. The idea behind this efficient algorithm is as follows: If a cycle has no $(k, 0)$ delay edge after delay merge, there must exist (i, j) , $j > 0$, delay edges in the original cycle. After removing these edges, the cycle becomes a DAG with only $(k, 0)$ delay edges. The MD retiming on this kind of DAG is the same as one-dimensional retiming which can add delays on each edge in the DAG [7]. It is

Algorithm 4.2 Procedure of Delay Merge

Input: MDFG $G = \langle V, E, \mathbf{d}, t \rangle$.

Output: Retimed MDFG $G_r = \langle V, E, \mathbf{d}_r, t \rangle$ with the smallest possible number of $(k, 0)$ delay edges.

Step 1. Construct a DAG $G_D = \langle V_D, E_D, \mathbf{d}, t \rangle$ from $G = \langle V, E, \mathbf{d}, t \rangle$ by finding all zero-delay paths starts with node u and ends with node v , such that, $\mathbf{d}(w \rightarrow u) = (k, 0)$ and $\mathbf{d}(v \rightarrow x) = (i, j)$, for $u, v, w, x \in V, k > 0$, and $j > 0$.

Step 2. Retime node v by $\mathbf{r}(v) = \mathbf{r}(v) + (1, 0), \forall v \in V_D$.

Step 3. Update delays in G . Exit if there is no $(k, 0)$ delay edge any more.

Step 4. Repeat step 1-3 for $\sum_e k$ times, where $\mathbf{d}(e) = (k, 0), \forall e \in E$.

equivalent to fully parallelize the original MDFG. On the other hand, if a cycle has $(k, 0)$ delay edge after delay merge, the original cycle must have only $(k, 0)$ delay edges. Then, applying one-dimensional retiming can find the minimum cycle period for unit-time data flow graph [7] which is also the minimum cycle time by definition. Note that when there is no $(k, 0)$ delay edge after delay merge, the SPINE-ROW-WISE algorithm generates the same full parallel MDFG as SPINE-FULL, but with more efficiency. When there are $(k, 0)$ delay edges after delay merge, the SPINE-ROW-WISE algorithm generates a retimed MDFG with the minimum cycle period. The resulted schedule length of the retimed MDFG is smaller than or equal to that is generated by standard software pipelining for single loops [1].

Algorithm 4.3 Procedure of SPINE-ROW-WISE

Input: MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, schedule vector $\mathbf{s} = (0, 1)$.

Output: A retimed MDFG $G_r = \langle V, E, \mathbf{d}_r, t \rangle$ achieving the schedule bound.

Step 1. Remove all the edges with delay vector $\mathbf{d}(e) = (i, j)$, for $i \geq 0$ and $j > 0$.

Step 2. Do one-dimensional retiming in x-dimension.

return The retimed MDFG.

In the following, we use an example to illustrate SPINE procedure and compare the final result generated by the SPINE-FULL algorithm with that generated by the standard software pipelining and chained MD retiming. Consider the data flow graph in Figure 15(a). The execution time of the cycle is 4 time units. The code size can be measured by the number of nodes in MDFG, which is 6 instructions. Figure 15(b) shows the retimed MDFG using standard software pipelining approach, e.g. Modulo scheduling. The best cycle period can be obtain is 3 time units. However, the optimal execution rate is 1 time unit, which can be obtained by using either the SPINE-FULL algorithm (Figure 15(d)), or chained MD retiming (Figure 15(c)). Although the cycle periods are

the same, the execution sequences of the pipelined loop is different. The schedule vector used by SPINE is $\mathbf{s} = (0, 1)$, while it's $(1, 1)$ using chained MD retiming.

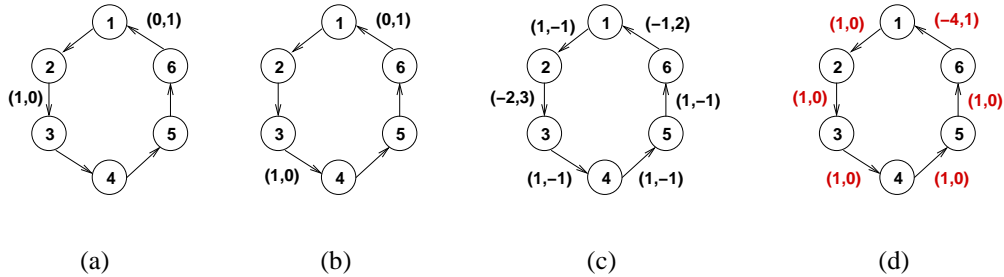


Figure 15: (a) An MDFG. (b) The retimed MDFG using the standard software pipelining. (c) The retimed MDFG using chained MD retiming. (d) The retimed MDFG using the SPINE-FULL algorithm.

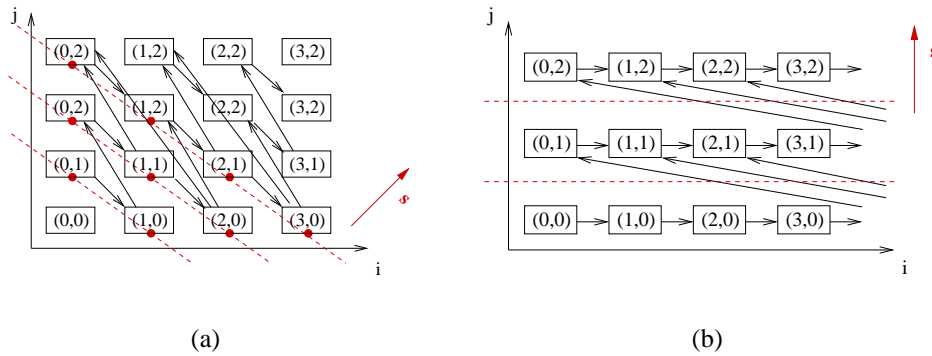


Figure 16: (a) The cell dependency graph of the pipelined loop using chained MD Retiming. (b) The cell dependency graph of the pipelined loop using SPINE.

Unimodular transformation [16] is performed to compute the new loop bounds and loop indexes for the pipelined loop generated by chained MD retiming. The code size of the compiled loop generated by MD retiming has 178 instructions. The resulting code size is 36 instructions using SPINE, which is only 20.2% of that generated by MD retiming. Assume that the loop bounds of the innermost loop and the outer loop are both 50. The execution time of the loop is 3916 time units, assuming a unit-time MDFG. The execution time of the fully pipelined loop generated by SPINE is 2750 time units, which is only 70.2% of the loop generated by MD retim-

ing. The cell dependency graphs of the software-pipelined loops are shown in Figure 16(a) and Figure 16(b), respectively.

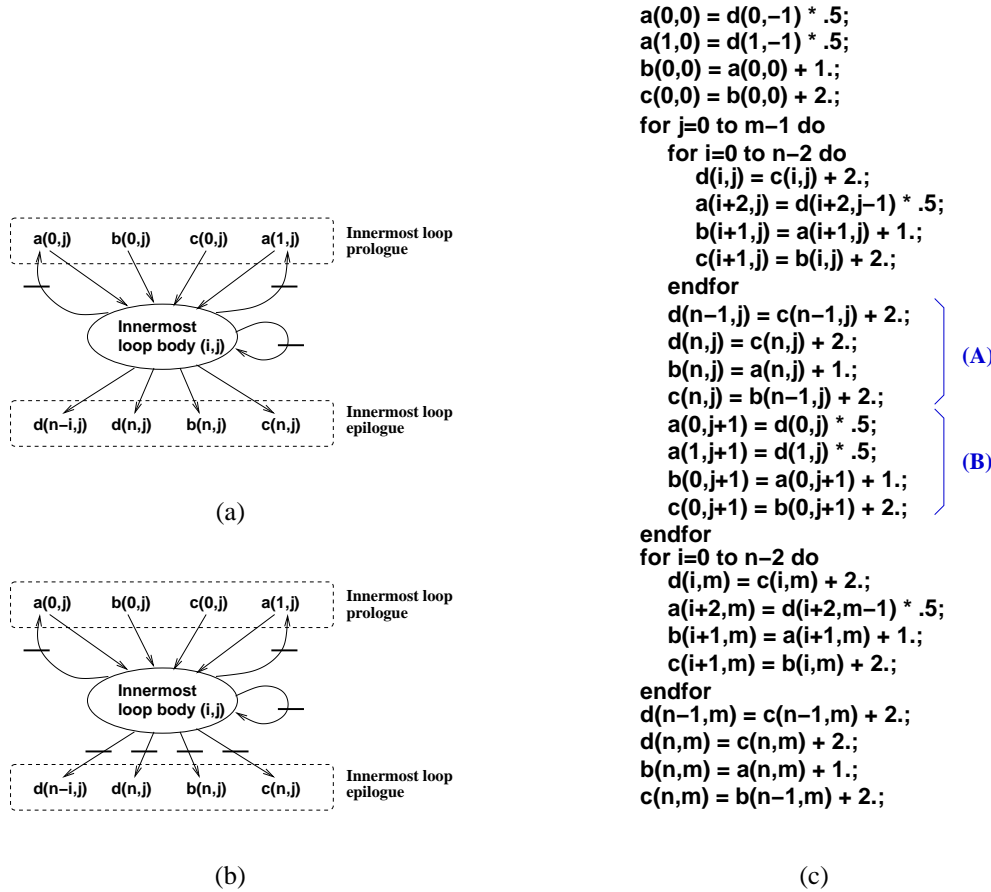


Figure 17: (a) The data flow graph of an imperfect loop shown in Figure 1(b). (b) The retiming view of outer loop pipelining. (c) The final code.

Software Pipelining on Imperfect Nested Loops

Our technique can be directly applied to loops with codes between loop levels (imperfect nested loops). The idea is to regard the whole innermost loop body as one node in a data flow graph. Then, we can apply retiming on the lower-dimensional data flow graph. For example, the two-level nested loop in Figure 1(b) can be modeled by the one-dimensional DFG in Figure 17(a) after representing the innermost loop body by a node in the DFG. The lines on edges represent delay units regarding to the outer loop iterations, that is, the dependency distance on y-dimension. Then, we can apply retiming on y-dimension alone [2, 3, 7] as shown in Figure 17(b). For this

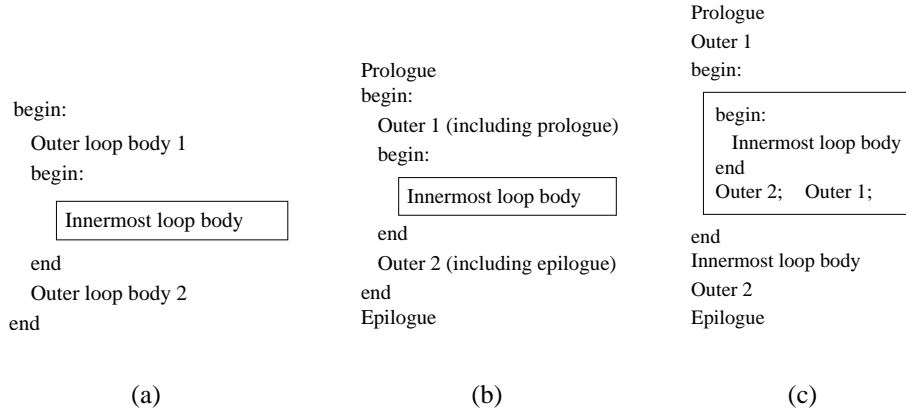


Figure 18: (a) The original code layout. (b) The software pipelined code layout. (c) The code after outer loop pipelining.

example, the retiming operation is equivalent to pipelining the prologue and epilogue sections of two successive outer loop iterations as shown in Figure 17(c). Note that the code section (A) and (B) in Figure 17(c) can be executed in parallel, as long as the dependency distance on x-dimension is preserved. We can also apply code size reduction technique on the resulted code to minimize the code size [17]. It is interesting to see that outer loop pipelining [8, 15] can be easily modeled using our approach. The code transformation of software pipelining and outer loop pipelining is illustrated in Figure 18 on simplified code layouts. Therefore, our SPINE technique can be effectively applied to multi-dimensional problems in general.

5 Experiments

In our experiments, we compare software-pipelined loops generated by the Modulo scheduling ("Modulo") [12, 14], the chained MD retiming ("Chained") [9, 10], and the SPINE-FULL algorithm ("SPINE") on a simulated system without resource constraint. Our benchmarks include a set of 2-D nested loops: Wave Digital filter ("WDF"), Differential Pulse-Code Modulation device ("DPCM"), Two Dimensional filter ("2D"), Floyd-Steinberg algorithm ("Floyd"), a small multi-dimensional data flow graph example ("MDFG"), and its 2-by-2 unfolded graph ("MDFG22"). Three metrics are compared in our experiments: cycle period of a MDFG, execution time of a software-pipelined loop, and code size of a pipelined loop. We use the count of instructions in the compiled code to measure the code size. Since the SPINE-ROW-WISE algorithm performs

the same as the SPINE-FULL algorithm when there is no $(k, 0)$ delay edge after delay merge, or performs the same as one-dimensional retiming when there is $(k, 0)$ delay edge after delay merge, we do not show the experimental results of the SPINE-ROW-WISE algorithm in this paper.

Benchmark	Node	Cycle Period			Execution Time		
		SPINE	Modulo	Improve	SPINE	Modulo	Improve
WDF(1)	4	1	3	66.7%	2600	7500	65.3%
WDF(2)	12	1	6	83.8%	2750	15000	81.7%
DPCM	16	1	4	75.0%	3838	10150	62.2%
2D(1)	4	1	4	75.0%	2650	10000	73.5%
2D(2)	34	1	3	66.7%	2850	7800	63.5%
MDFG	8	1	3	66.7%	2650	10000	73.5%
MDFG22	32	1	3	66.7%	2650	7650	65.4%
Floyd	16	1	10	90.0%	2950	25000	88.2%

Table 1: Compare cycle period and execution time of software-pipelined nested loops generated by SPINE and Modulo scheduling.

Table 1 compares the cycle period and execution time of the pipelined loops using the standard software pipelining and the SPINE-FULL algorithm. Column “Node” shows the number of nodes in a loop. Column “Improve” shows the improvement percentage of cycle period of loops generated by SPINE-FULL compared with that generated by the standard software pipelining. The experimental results show that the cycle period of the pipelined loop generated by SPINE is one time unit for all the benchmarks. While the Modulo scheduling cannot achieve full parallelism on any of them. The experimental results show that the potential parallelism that can be explore by the standard software pipelining in a nested loop is very limited. The average improvement on cycle period by using SPINE is 73.8%. The average execution time of the loops generated by SPINE is 71.7% smaller than that generated by the standard software pipelining.

Table 2 compares the code size and execution time of the software-pipelined loops generated by the SPINE-FULL algorithm and the chained MD retiming. Column “Retime” shows the retiming times. For most of the benchmarks, except for DPCM, SPINE achieves full parallelism using schedule vector $\mathbf{s} = (0, 1)$, while the chained MD retiming uses schedule vector $\mathbf{s} = (1, 1)$. The code size is reduced by 69.5% on average by using SPINE. The average improvement on execution time is 27.3%. For DPCM, $(k, 0)$ delay vectors in cycles of the MDFG cannot be removed. In this case, SPINE-FULL uses schedule vector $\mathbf{s} = (1, 1)$. Therefore, the code sizes and execution rates are the same as the chained MD retiming. Note that the chained MD retiming

Benchmark	Retime	Code Size			Execution Time		
		SPINE	Chained	Improve	SPINE	Chained	Improve
WDF(1)	2	21	70	70.0%	2600	3759	30.8%
WDF(2)	5	81	474	82.9%	2750	3996	31.2%
DPCM	3	312	312	0%	3838	3838	0%
2D(1)	3	25	98	74.5%	2650	3838	31.0%
2D(2)	7	281	2240	87.5%	2850	4194	32.0%
MDFG	3	41	166	75.3%	2650	3838	31.0%
MDFG22	3	137	574	76.1%	2650	3838	31.0%
Floyd	9	169	1646	89.7%	2950	4312	31.6%

Table 2: Compare code size and execution time of software-pipelined nested loops generated by SPINE and Chained MD Retiming.

can chose any schedule vector that satisfies $\mathbf{s} \cdot \mathbf{d} > 0$ without considering the computation and code size overheads. While using SPINE-FULL, the schedule vector $\mathbf{s} = (1, 1)$ is chosen on purpose to keep the computation and code size overheads as small as possible.

6 Conclusion

The existing techniques cannot optimize nested loops effectively for many embedded systems with strict timing and code size requirements. The standard software pipelining techniques can not fully explore the parallelism in nested loops. Multi-dimensional retiming, on the other hand, can fully parallelize a nested loop, but does not consider timing and code size overheads. Therefore, it is not suitable for software pipelining on nested loops. In this paper, we present theory and algorithms of nest-loop software pipelining with minimal overheads in loop index generation and code size based on the fundamental understanding of the properties of software pipelining problem on nested loops. Our SPINE-FULL algorithm tries to fully parallelize nested loops with a row-wise execution sequence whenever it is possible. For the loops cannot be fully parallelized using row-wise execution sequence, the algorithm quickly identifies them and generates full-parallel nested loop with overheads as small as possible. Our SPINE-ROW-WISE algorithm generates the best parallelized result with the minimal overheads using a row-wise execution sequence. The experimental results show that SPINE technique outperforms the Modulo scheduling in nest-loop software pipelining for all our benchmarks. It is also superior to MD retiming for most of the cases in terms of code size and execution time of the final code.

References

- [1] R. Bailey, D. Defoe, R. Halverson, R. Simpson, and N. Passos. A study of software pipelining of multi-dimensional problems. In *Proc. 13th Int'l Conf. on Parallel and Distributed Computing and Systems*, pages 426–431, Aug. 2000.
- [2] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229–239, Mar. 1997.
- [3] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. on Parallel and Distributed Systems*, 8(12):1259–1267, Dec. 1997.
- [4] A. Darté and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, Aug. 1994.
- [5] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiciak. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proc. 3rd IEEE/ACM Workshop on Media and Streaming Processors*, pages 29–38, Dec. 2001.
- [6] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. SIGPLAN'88 ACM Conf. on Programming Language Design and Implementation*, pages 318–328, June 1988.
- [7] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, Aug. 1991.
- [8] K. Muthukumar and G. Doshi. Software pipelining of nested loops. In R. Wilhelm, editor, *CC 2001*, LNCS 2027, pages 165–181. Springer-Verlag, Berlin Heidelberg, 2001.
- [9] N. L. Passos and E. H.-M. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Trans. on Parallel and Distributed Systems*, 7(11):1150–1163, Nov. 1996.
- [10] N. L. Passos and E. H.-M. Sha. Scheduling of uniform multi-dimensional systems under resource constraints. *IEEE Trans. on VLSI Systems*, 6(4):719–730, Dec. 1998.
- [11] J. Ramanujam. Optimal software pipelining of nested loops. In *Proc. 8th Int'l Parallel Processing Symposium*, pages 335–342, Apr. 1994.

- [12] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. 27th IEEE/ACM Annual Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov. 1994.
- [13] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th ACM/IEEE Annual Workshop on Microprogramming*, pages 183–198, Oct. 1981.
- [14] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. 25th IEEE/ACM Annual Int'l Symp. on Microarchitecture (MICRO)*, pages 158–169, Dec. 1992.
- [15] R. Scales. Nested loop optimization on the TMS320C6x. Application Report SPRA519, Texas Instruments, Feb. 1999.
- [16] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.
- [17] Q. Zhuge, E. H.-M. Sha, and C. Chantrapornchai. CRED: Code size reduction technique and implementation for software-pipelined applications. In *Proc. IEEE Workshop of Embedded System Codesign (ESCODES)*, pages 50–56, Sept. 2002.