Timing Optimization of Nested Loops Considering Code Size for DSP Applications *

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

Software pipelining for nested loops remains a challenging problem for embedded system design. The existing software pipelining techniques for single loops can only explore the parallelism of the innermost 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 use MD retiming to model the software pipelining problem of nested loops. We show that the computation time and code size of a software-pipelined loop nest is affected by execution sequence and retiming function. The algorithm of Software PIpelining for NEsted loops technique (SPINE) is proposed to generate fully parallelized loops efficiently with the overheads as small as possible. The experimental results show that our technique outperforms both the standard software pipelining and MD retiming significantly.

1. Introduction

With the advance of the technology, embedded systems with multiple cores or VLIW-like architectures, such as TI's TMS320C6x, Philips' TriMedia, and IA64, etc., become necessary to achieve the required high performance for the applications with growing complexity. To exploit multiple functional units or processors in parallel embedded systems, software pipelining is widely used to explore the instruction-level parallelism in loops [6, 13]. However, software pipelining can greatly expand the code size [14, 16]. Code size is one of the most critical concerns for many embedded processors because the capacity of on-chip memory modules is still very limited due to the chip size, cost and power considerations. The code size problem has a dramatic effect on nested loops because code size can be expanded in multiple loop levels. Our research shows that the code size of software-pipelined loop nests is greatly affected by the execution sequence and the software pipelining degree chosen by an optimization technique. An example of software pipelining on a nested loop is shown in Figure 1. Figure 1(a) shows the original loop nest. The schedule length of the loop body is 3 control steps. Figure 1(b) and Figure 1(c) are software pipelined loops with the same schedule length of loop body which is 1 control step. However, the code in Figure 1(c) is much more complicated than that in Figure 1(b). The optimization technique generating the code in Figure 1(b) uses a row-wise execution sequence, while the technique generating the code in Figure 1(c) uses a skewed execution sequence which introduces large overheads in code size and loop indexes computation. Because of the space limitation, we do not show the code sections of epilogues which are about the same size as the prologues in this case. Optimization techniques on nested loops need to consider both timing and code size requirements, therefore, becomes a great challenge for parallel compiler.

While software pipelining of single loops has been extensively studied and implemented [2, 3, 6, 13, 14, 16], there is very few work 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 parallelism in a nested loop or do not consider the overheads such as loop indexes and loop bounds computation, and code size expansion due to transformation. The standard software pipelining techniques for single loops focuses on one-dimensional problems. When applied to nested loops, it only optimizes the innermost loop [1, 2, 6, 13]. While nested loops usually exhibit dependencies cross loop dimensions. Therefore, the performance improvement that can be obtained by the standards software pipelining techniques is very limited. Hyperplane schedul-

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

```
for i=0 to m do
   for j=0 to n do
      d(i,j) = c(i,j) + 2.;
      a(i,j) = d(i–1,j) * .5;
      b(i,j) = a(i,j) + 1.;
      c(i,j) = b(i,j–1) + 2.
   endfor
endfor
```

(a)

```
for i=0 to m do
   a(i,0) = d(i–1,0) * .5;  \
   a(i,1) = d(i–1,1) * .5;   |
   b(i,0) = a(i.0) + 1.;     |-- Prologue
   c(i,0) = b(i,0) + 2.;    /
   for j=0 to n–2 do
      d(i,j) = c(i,j) + 2.;
      a(i,j+2) = d(i–1,j+2) * .5;
      b(i,j+1) = a(i,j+1) + 1.;
      c(i,j+1) = b(i,j) + 2.;
   endfor
   ...... Epilogue ......
endfor
```

(b)

```
d(0,0) = c(0,0) + 2.;       \
a(0,0) = d(–1,0) * .5;       |-- Outer loop
b(0,0) = a(0,0) + 1.;        |   Prologue
c(0,0) = b(0,–1) + 2.;      /
for i'=2–n to m do
   lj = max(0,2–i');        \-- Loop bounds
   uj = min(n–2,m–i');      /   computation
   a(i'+lj,lj) = d(i'+lj–1,lj) * .5;  \
   c(i'+lj,lj) = b(i'+lj,lj–1) + 2;    |
   if (lj > 0)                          |-- Inner loop
      a(m,lj) = d(m–1,lj) * .5;         |   Prologue
      c(m,lj) = b(m,lj–1) + 2.;         |
   endif                               /
   for j'=lj to uj do
      d(i'+j'–2,j'+2) = c(i'+j'–2,j'+2) + 2.;
      a(i'+j'–1,j'+1) = d(i'+j'–2,j'+1) * .5;
      b(i'+j',j') = a(i'+j',j') + 1.;
      c(i'+j'–2,j'+2) = b(i'+j'–2,j'+1) + 2.;
   endfor
              ......
      ......  <Epilogue>  ......
              ......
```

(c)

Figure 1. (a) The original code. (b) The software-pipelined code generated by the SPINE technique. (c) The software-pipelined code generated by the chained MD retiming.

ing [5, 12] tries to convert a nested loop into a single loop using loop unrolling and skewing to reduce the execution time. However, this technique makes code generation extremely difficult, and introduces large overhead in loop index generation. 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 exploited. 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 [10]. MD retiming can achieve full parallelism of an multi-dimensional problem. That is, all the computations in an MD problem can be executed in parallel. However, MD retiming does not consider some critical issues for nested loop optimization, such as loop index generation and code size. To the authors' knowledge, there is no existent technique that can effectively solve the software pipelining problem for nested loops in embedded systems.

In this paper, we use retiming concept [7] to model the software pipelining on nested loops. The theory of software pipelining for nested loops is derived based on the understanding of the relationship among execution rate, execution sequence, and the retiming function of nested loops. We prove that the minimum computation time of an iteration without unfolding [4, 9] for a nested loop with given execution sequence can be achieved by choosing an appropriate retiming function (Theorem 3.3). We do not consider the relationship between execution rate and unfolding factor for nested loops in this paper because unfolding duplicates the code size of loop body. Based on the theoretical foundation of software pipelining for nested loops, we propose a Software PIpelining for NEsted loops (SPINE) technique with an efficient algorithm, the SPINE-FULL algorithm. It fully parallelizes a nested loop with the minimal overheads. We conduct experiments on a set of two-dimensional DSP benchmarks to compare the code quality generated by SPINE with that generated by the other two techniques: the standard software pipelining technique, Modulo scheduling [13], and the standard MD retiming technique, the chained MD retiming [11]. Our experimental results show that SPINE out-performs or ties both of the other two techniques on all of our benchmarks.

The rest of the paper is organized as follows: Section 2 gives an overview for the graph representation of nested loops and multi-dimensional retiming model. The limitations of the existing techniques are discussed in Section 2.1. Section 3 presents the theory of software pipelining for nested loops. Section 4 presents the SPINE algorithms with an illustrative example. The experimental results are provided in Section 5. Finally, we conclude the paper in Section 6.

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 *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 $\mathbf{i} = (i_1, i_2, \ldots, i_n)$, starting from $(0, 0, \ldots, 0)$, where elements are ordered from the outermost loop to innermost loop. Inter-iteration dependencies are represented by edges weighted by delay vectors. For any iteration $\mathbf{j}$, an edge $e(u \to v)$ with delay $\mathbf{d}(e)$ indicates

that the computation of node v at iteration $\mathbf{j}$ requires data produced by node u at iteration $(\mathbf{j} - \mathbf{d}(e))$. Intra-iteration dependencies, i.e. dependencies within the same iteration, are represented by edges without delay. A legal MDFG must have no zero-delay cycle. A static schedule must obey intra-iteration dependencies. The *cycle period* of a data flow graph G, denoted by $\Phi(G)$, is defined as the computation time of the longest zero-delay path, which corresponds to the minimum schedule length without resource constraint. We assume the computation time of a node is 1 time unit in this paper. The longest zero-delay paths of the MDFG in Figure 2(a) are from $D \to A \to B$ or $D \to A \to C$. Thus, the cycle period is $\Phi(G) = 3$.

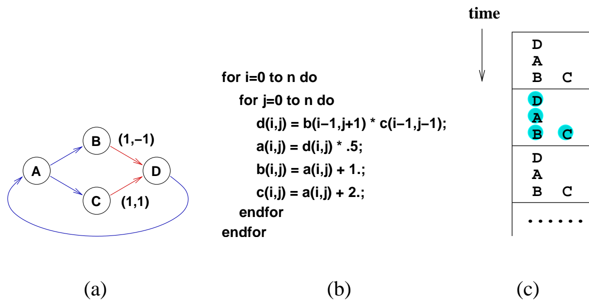

Figure 2. (a) An MDFG. (b) Code of the nested loop. (c) The static schedule.

The retiming technique [7] can be applied on a data flow graph to minimize the cycle period in polynomial time by evenly distributing the delays in the graph. The delays are moved around in the graph in the following way: a delay unit is drawn from *each* of the incoming edges of v, and then added to *each* of the outgoing edges of v, or vice verse. Note that the retiming technique preserves data dependencies of the original DFG. The multi-dimensional retiming (MD retiming) function $\mathbf{r} : V \to \mathbb{Z}^n$ represents the number of delay units moved through node $v \in V$. Figure 3(a) shows the retimed MDFG of Figure 2(a) with MD retiming function $\mathbf{r}(D) = (0, 1)$. Consider a retimed DFG $G_r = (V, E, \mathbf{d}_r, t)$ computed by retiming $\mathbf{r}$. The number of delays of any edge $e(u \to v)$ after retiming can be computed as $\mathbf{d}_r(e) = \mathbf{d}(e) + \mathbf{r}(u) - \mathbf{r}(v)$. And the total number of delays remains constant for any cycle in the graph.

When a delay is pushed through node D to its outgoing edge as shown in Figure 3(a), the actual effect on the schedule of the new MDFG is that the $\mathbf{i}^{th}$ copy of D is shifted up and is executed with $(\mathbf{i} - (0, 1))^{th}$ copy of nodes A, B, and C. Because there is no dependency between node D and nodes A, B, and C in the new loop body, node D and node A can be executed in parallel in the new schedule. The schedule length of the new loop body is then reduced from three control steps to two control steps as shown in Figure 2(c) and Figure 3(c). In fact, every retiming operation corresponds to a software pipelining operation [16]. Some nodes are shifted out of the loop body to become prologue and epilogue. We can measure the exact code size of prologue and epilogue [16] using retiming function $\mathbf{r}$.

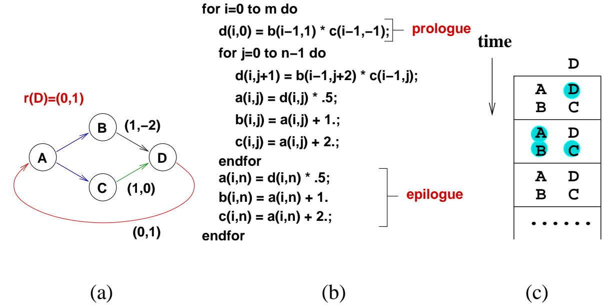

Figure 3. (a) The retimed MDFG G_r of MDFG in Figure 2(a) with $r(D) = (0, 1)$. (b) Code of the retimed MDFG.

2.1. Limitations of Existing Techniques

There are several techniques can be applied to optimize nested loops. However, they all have limitations in terms of performance improvement, code size, or complexity of code generation. By analyzing the limitations of the existing techniques, the issues of nested loop optimization becomes clear.

The standard software pipelining techniques for one-dimensional loops are well developed [13]. However, when they are applied to optimize nested loops, the performance improvement is very limited. Consider the MDFG in Figure 4(a) with cycle period $\Phi(G) = 3$. The best result can be achieved by the standard software pipelining techniques, such as Modulo scheduling, is $\Phi(G) = 2$, as shown in Figure 4(b). Actually, the MDFG can be fully-parallelized. Figure 4(c) shows the retimed MDFG by using our SPINE technique which exploit the outer loop dependencies. The detailed SPINE algorithms will be presented later.

Another interesting example is shown in Figure 5(a) which has two orthogonal delay vectors $\mathbf{d}(B \to C) = (0, 1)$ and $\mathbf{d}(D \to A) = (1, 0)$. A cell dependency graph (CDG) clearly shows the execution sequence of the loop nest [10] as shown in Figure 5(b). The cell dependency graph of an

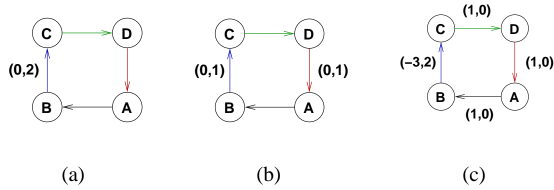

Figure 4. (a) An MDFG. (b) The retimed MDFG using the standard software pipelining technique. (c) A fully-parallelized MDFG using SPINE technique.

MDFG is a directed acyclic graph showing the dependencies between copies of nodes representing the MDFG. A node in CDG is a computational cell that represents a complete iteration. The CDG of a nested loop is bounded by the loop indexes. A *schedule vector* $\mathbf{s}$ is a normal vector for a set of parallel hyper-planes that defines a sequence of execution in a CDG. A legal MDFG $G = \langle V, E, \mathbf{d}, t \rangle$ is realizable if there exists a schedule vector $\mathbf{s}$ for the cell dependency graph with respect to G, i.e., $\mathbf{s} \cdot \mathbf{d}(e) \geq 0$, for all $e \in E$, and no cycle exists in the CDG [10]. The CDG shown in Figure 5(b), can be executed by a row-wise execution sequence, i.e., the schedule vector $\mathbf{s} = (1, 0)$.

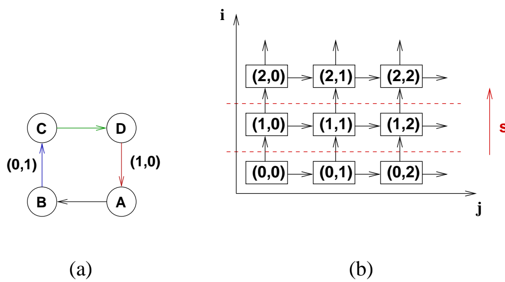

Figure 5. (a) An MDFG. (b) The cell dependency graph.

For the MDFG shown in Figure 5(a), the standard software pipelining techniques can do nothing to optimize the loop. However, it can be fully parallelized using MD retiming techniques, such as the chained MD retiming [10]. The chained MD retiming [10] can fully parallelize a MDFG $G = \langle V, E, \mathbf{d}, t \rangle$ by selecting a legal schedule vector $\mathbf{s}$ such that $\mathbf{s} \cdot \mathbf{d}(e) > 0$, for all $e \in E$, and a retiming vector $\mathbf{r}$ such that $\mathbf{r} \perp \mathbf{s}$. To achieve a realizable MDFG after retiming, the legality condition, $\mathbf{s} \cdot \mathbf{d_r}(e) \geq 0$, has to be satisfied, and there should not exist any cycle in the cell dependency graph of the MDFG. For the example in Figure 5(a), a fully-parallelized solution using the chained MD retiming is shown in Figure 6(a). However, the original row-wise execution sequence is skewed by using the chained MD retiming. The schedule vector becomes $\mathbf{s} = (1, 1)$ as shown in Figure 6(b). The skewed execution sequence generates the code with large overheads as shown in Figure 1(c). The dramatically large overheads are produced due to the following reasons: First, prologue and epilogue are produced for multiple loop levels. Second, the computation of new loop bounds and loop indexes becomes much more complicated than using a row-wise execution sequence. Extra computation for executing the iterations along the loop boundrary makes the final code even more complicated as indicated by the "Inner Loop Prologue" in Figure 1(c). And third, the original data locality is disrupted because of the skewed execution sequence. As a result, address generation becomes very complicated. Therefore, MD retiming is not suitable for software pipelining of nested loops.

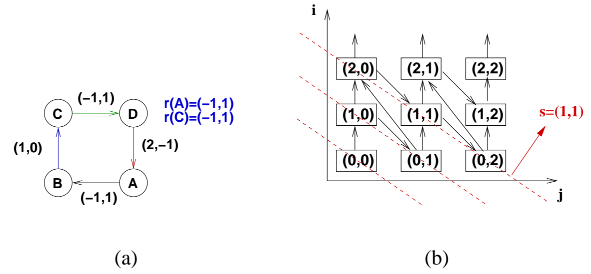

Figure 6. (a) A fully-parallelized MDFG using chained MD retiming. (b) The cell dependency graph.

3. Theory of Software Pipelining for Nested Loops

In this section, we show that the lower bound of the computation time of a nested loop is affected by schedule vector and retiming function. We present the theoretical foundation of software pipelining for nested loops with two loop levels based on retiming concept. Although the theorems are derived for two-dimensional, unit-time MDFGs, they can be generalized to multi-dimensional, general-time cases. First, we will introduce definitions and assumptions that are necessary for the understanding of the theorems.

Definition 3.1. *Given a schedule vector* $\mathbf{s}$ *and a retiming function* $\mathbf{r}$*. Let* ℓ *be a cycle in an MDFG* G*. The minimum*

cycle time of cycle ℓ, $C_{min}(\ell)$, is the minimum execution time of ℓ with schedule vector **s** *via MD retiming.*

We also make the following assumptions: 1) We assume all the elements of a schedule vector are nonnegative integers. 2) If there are delay vectors orthogonal to **s**, we assume that only the ones on the right hand-side of **s** are the legal ones in order to avoid cycles in cell dependency graph. For instance, the given schedule vector is $\mathbf{s} = (0, 1)$. Delay vectors $\mathbf{d} = (1, 0)$ and $\mathbf{d} = (-1, 0)$ produce cycles in cell dependency graph. According to our assumption, only the delay vector $\mathbf{d} = (1, 0)$ is legal, which is corresponding to a loop counter with an increasing value.

We start with the simplest case with schedule vector $\mathbf{s} = (1, 0)$ and retiming function $\mathbf{r} = (0, 1)$. The following theorem derives the achievable minimum cycle time, and show that the minimum cycle time $C_{min}(\ell) = 1$ is not always achievable with given **s** and **r**.

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

1. *The minimum cycle time $C_{min}(\ell) = 1$ can be achieved via MD retiming, if $\sum_{e \text{ in cycle } \ell} \mathbf{d}(e) = (i, j)$, for $i > 0$,*
2. *If $\sum_{e \text{ in cycle } \ell} \mathbf{d}(e) = (0, k)$, for $k > 0$, then the minimum cycle time $C_{min}(\ell) = \lceil T(\ell)/k \rceil$ can be achieved, where $T(\ell) = \sum_{v \text{ in cycle } \ell} t(v)$.*

Theorem 3.1 states that a cycle can always be fully parallelized using a row-wise execution sequence when the total delay of the cycle is $\mathbf{d}(\ell) = (i, j)$, $i > 0$. The total delay vector after retiming is $\mathbf{d_r}(\ell) = \mathbf{d}(\ell) + \mathbf{r}(u) - \mathbf{r}(v) = (i, j) + (0, k) - (0, l) = (i, j + k - l)$, and $i > 0$. Therefore, the legality condition of MD retiming is always satisfied. In other words, a nested loop with outer-loop carried dependency can always be fully parallelized via MD retiming using a row-wise execution sequence.

For the nested loops without outer-loop carried dependency, the total delay of a cycle has a formula of $(0, k)$. We call this kind of cycle a (0,k)-cycle. Then, the full parallelism may not be achievable. For example, the minimum cycle time of the (0,2)-cycle in Figure 4(a) is 2, as shown in Figure 4(b), using a row-wise execution sequence.

Theorem 3.1 indicates that we need to consider the total delay of a cycle to decide whether a cycle can be fully parallelized with given schedule vector and retiming vector. We propose an idea of merging delays via MD retiming based on this theorem. Let's re-consider the example in Figure 5(a). The standard software pipelining technique such as Module scheduling cannot do any optimization for this kind of loop. While the chained MD retiming can fully parallelize the loop nest, but use a skewed schedule vector $\mathbf{s} = (1, 1)$, which expands the code size dramatically. While we suggest that the MDFG needs to be

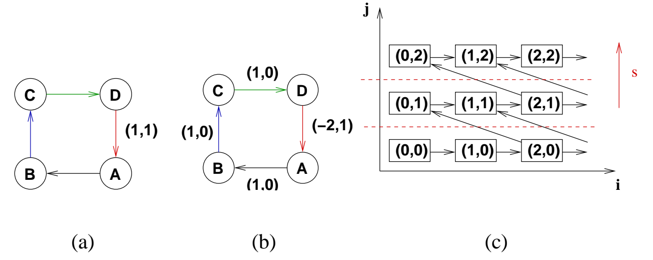

Figure 7. (a) Merging delays. (b) The fully parallel MDFG via retiming $\mathbf{r}(A) = (3, 0)$, $\mathbf{r}(B) = \mathbf{r}(C) = (2, 0)$, and $\mathbf{r}(D) = (1, 0)$. (c) The cell dependency graph.

transformed before software pipelining by merging the delays in a cycle. The resultant MDFG shown in Figure 7(a) has only one delay vector $\mathbf{d}(D \to A) = (1, 1)$, using retiming $\mathbf{r}(C) = \mathbf{r}(D) = (0, 1)$. Then, we can fully parallelize the loop via another MD retiming with schedule vector $\mathbf{s} = (1, 0)$ and retiming vector $\mathbf{r} = (0, 1)$. Thus, the row-wise execution sequence can be preserved, which is critical for obtaining the final code with small overheads. The final code of the retimed MDFG in Figure 7(b) is shown Figure 1(b). The final code size is substantially reduced. Loop bounds and loop indexes can be easily computed according to retiming functions. The data locality is also maintained. The cell dependency graph with row-wise execution sequence is shown Figure 7(c).

To further understand the timing property of an MDFG with given schedule vector and retiming function, Theorem 3.2 derives the achievable minimum cycle time by generalizing the result of Theorem 3.1 to any given schedule vector **s** and a retiming **r** that is orthogonal to **s**.

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

1. *The minimum cycle time $C_{min}(\ell) = 1$ can be achieved via MD retiming if $\exists e \in \ell$, $\mathbf{d}(e) \nparallel \mathbf{r}$.*
2. *If $\mathbf{d}(e) \parallel \mathbf{r}$, $\forall e \in \ell$, then, $C_{min}(\ell) = \lceil T(\ell)/D(\ell) \rceil$, where $T(\ell) = \sum_{v \text{ in cycle } \ell} t(v)$, and $D(\ell) \times \mathbf{r} = \sum_{e \text{ in cycle } \ell} \mathbf{d}(e)$.*

The result of Theorem 3.1 is generalized based on the understanding that if only there exists a delay vector in a cycle that is not parallel to retiming **r**, then the total delay of the cycle will not be parallel to retiming **r**. Thus, the MDFG can be fully parallelized using retiming **r**. On the other hand, if the summation of the delay vectors in a cycle is parallel to **r**, all the delay vectors in the cycle must

be parallel to $\mathbf{r}$. In this case, the full parallelism may not be achievable.

To fully understand the timing property of any loop nest, we further derive the minimum cycle time for any given $\mathbf{s}$ and $\mathbf{r}$ such that $\mathbf{s} \not\perp \mathbf{r}$. The minimum cycle time can be computed as $C_{min}(\ell) = \lceil T(\ell)/D(\ell) \rceil$, where $T(\ell) = \sum_{v \ in\ cycle\ \ell} t(v)$, and $D(\ell) = min(g(\ell), T(\ell))$, where $g(\ell) = \lceil \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}} \rceil + \delta$, and δ can be computed as follows: If $\lceil \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}} \rceil \neq \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}}$, $\delta = 0$. If $\lceil \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}} \rceil = \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}}$, and $\mathbf{d}(\ell) \cdot \mathbf{x} - \lceil \frac{\mathbf{s} \cdot \mathbf{d}(\ell)}{\mathbf{s} \cdot \mathbf{r}} \rceil \cdot (\mathbf{r} \cdot \mathbf{x}) \leq 0$, $\delta = 0$.
Otherwise, $\delta = 1$. Due to the space limitation, we do not show the formal theorem and its proof. They can be easily verified on MDFGs.

The above theorems show that the minimum cycle time $C_{min}(\ell)$ of a cycle ℓ is affected by retiming function $\mathbf{r}$ and schedule vector $\mathbf{s}$. In the following theorem, we show that a retiming function $\vec{r}$ that is orthogonal to $\vec{s}$ is the best choice for timing optimization. Due to the space limitation, we omit the formal proof of the theorem, but the correctness of the theorem can be easily verified.

Theorem 3.3. *Given MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, and a schedule vector $\mathbf{s}$. Let ℓ be a cycle in G. Let $\mathbf{r}$ and $\mathbf{r}'$ be two retiming functions such that $\mathbf{r} \perp \mathbf{s}$ and $\mathbf{r}' \not\perp \mathbf{s}$. The minimum cycle time of ℓ via MD retiming using retiming function $\mathbf{r}$ is always less than or equal to that using $\mathbf{r}'$.*

Based on the theory of software pipelining for nested loops, we propose a Software PIpelining for NEsted loops (SPINE) technique. The principals of SPINE technique can be summarized as follows: (1) There must exist a schedule vector $\mathbf{s}$, and a retiming function $\mathbf{r}$ orthogonal to $\mathbf{s}$, such that the retimed loop is fully parallelized. (2) We should pick $\mathbf{r} \perp \mathbf{s}$, so that it gives the shortest schedule. (3) If we fix $\mathbf{s} = (1, 0)$ and use $\mathbf{r} = (0, 1)$, SPINE will find the retimed MDFG of the input program with the minimum computation time. The computation cost due to loop transformation and the code size overhead of the software-pipelined loop is minimal.

4. SPINE Algorithm

In this section, we present the algorithm of software pipelining for nested loops. The SPINE-FULL algorithm generates fully-parallelized nested loops with computation and code size overheads as small as possible. It will be interesting to see that the chained MD retiming becomes a special case of the SPINE-FULL algorithm. Although the algorithm is presented for two-dimensional loops, multi-dimensional problems can be solved in a similar way.

We employ the idea of delay merge but transform it into a more efficient technique in our SPINE algorithm. According to retiming theory, DAG can always be retimed such that any edge has at least one delay. After the (i,j)-delay edges are removed, a cycle with (i,j)-delay edges becomes a DAG. Thus, we can directly apply one-dimensional retiming to minimize the cycle period without actually doing delay merge.

Algorithm 4.1 SPINE-FULL Algorithm

Input: MDFG $G = \langle V, E, \mathbf{d}, t \rangle$
Output: A fully parallel MDFG $G_r = \langle V, E, \mathbf{d}_r, t \rangle$ with the smallest possible overheads.
if SPINE((1,0), G, 1) == 1 **then**
  Return: $\mathbf{s} = (1, 0)$
else if SPINE((0,1), G, 1) == 1 and $\mathbf{d}(e) \cdot (0, 1) \geq 0$ **then**
  Return: $\mathbf{s} = (0, 1)$
else if SPINE((1,1), G, 1) == 1 and $\mathbf{d}(e) \cdot (1, 1) \geq 0$ **then**
  Return: $\mathbf{s} = (1, 1)$
else
  Choose a $\mathbf{s}$ such that SPINE($\mathbf{s}$, G, 1) == 1, $\mathbf{s} \cdot \mathbf{d}(e) \geq 0, \forall e \in E$, and $|\mathbf{s}_x| + |\mathbf{s}_y|$ is as small as possible.
  Return: $\mathbf{s}$
end if

Algorithm 4.2 Procedure of SPINE(G, $\mathbf{s}$, c)

Input: MDFG $G = \langle V, E, \mathbf{d}, t \rangle$, schedule vector $\mathbf{s}$, a desired cycle period c.
Output: Retiming function $\mathbf{r}$, retimed graph G_r, and cycle time c if it is feasible; otherwise, return 0.
/* Check the legality of schedule vector $\mathbf{s}$. */
if $\mathbf{s} \cdot \mathbf{d}(e) \not\geq 0, \forall e \in E$. **then**
  Error: Illegal schedule vector.
end if
Choose retiming vector such that $\mathbf{r} \perp \mathbf{s}$
/* Remove the edges that are not parallel to $\mathbf{r}$. */
Construct graph $G' = \langle V, E', d', t \rangle$, such that $E' \longleftarrow E - \{e \mid \mathbf{d}(e) \not\parallel \mathbf{r}, \forall e \in E\}$, and $d'(e) \longleftarrow \mathbf{d}(e)/\mathbf{r}, \forall e \in E'$
/* The feasible clock-period test algorithm */
Compute $D(u, v)$ and $T(u,v)$ for any two nodes $u, v \in V$.
Construct the constraint graph with the desired cycle period c.
Use the shortest path algorithm to fi nd a solution.
if There exists a retiming solution $r'(v)$ such that $\Phi(G'_{r'}) \leq c$ **then**
  /* Update the retimed graph */
  Retime G with $r'(v) \cdot \mathbf{r}, \forall v \in V$.
  Return: c
else
  Error: Unfeasible cycle period.
  Return: 0
end if

Algorithm 4.1 shows the procedure of SPINE-FULL Algorithm. The major function of the algorithm is SPINE(G, $\mathbf{s}$, c) which is shown in Algorithm 4.2. The inputs of SPINE function are schedule vector $\mathbf{s}$, MDFG G, and a desired cycle period c. The function produces a retiming of G with cycle period at most c, and returns c, if such a retiming exists; otherwise, it returns 0. The procedure first checks the legality of a given schedule vector $\mathbf{s}$, and chooses a retiming $\mathbf{r}$ that is orthogonal to $\mathbf{s}$. Then, it removes all the edges with delay vectors not parallel to $\mathbf{r}$. In order to apply 1-D

retiming, the delay vectors are converted to the delay units with respect to **r**. To apply the feasible cycle-period test algorithm [7], the total number of delays of the critical path between any two nodes $D(u, v)$, and the total computation time of the critical path $T(u, v)$ are computed. If the desired cycle period c is achievable, the MD retiming will be performed on the MDFG G, and the value of c is returned. Otherwise, the function reports error and returns 0.

Given an MDFG $G = \langle V, E, d, t\rangle$ and a desired cycle period c, this algorithm produces a retiming r of G such that the cycle period of the retimed graph $\Phi(G_r) \leq c$, if such a retiming exists. The shortest path algorithm, such as Bellman Ford algorithm, can be used to find the solution of a set of constraint inequalities which is formulated from the following legality and feasibility conditions of retiming:

1. Legality: $d_r(e) = d(e) + r(u) - r(v) \geq 0 \Longrightarrow r(v) - r(u) \leq d(e), \forall e(u \to v) \in E$.
2. Feasibility: $r(v) - r(u) \leq D(u, v) - 1, \forall u, v \in V$ such that $T(u, v) > c$.

where $D(u, v)$ is the minimum number of delays on any path from node u to node v. The path $u \rightsquigarrow v$ with the delay number $= D(u, v)$ is called the critical path from u to v. The value of $T(u, v)$ is the maximum total computation time on any critical path from u to v.

The SPINE-FULL Algorithm shown in Algorithm 4.1 aims to achieve a fully parallelized loop. Therefore, the desired cycle period c is set to be 1 for unit-time MDFG. First, the algorithm tries to use row-wise execution sequence, i.e. schedule vector $\mathbf{s} = (1, 0)$, to fully parallelize the MDFG G. If it is not achievable, the algorithm tries the schedule vector $\mathbf{s} = (0, 1)$ if it is a legal schedule vector. The schedule vector $\mathbf{s} = (0, 1)$ represents an column-wise execution and can be transformed to row-wise execution by loop index interchange. The schedule vectors (1,0) and (0,1) are the ones that produce the smallest code size expansion and the least loop index and bounds computation. If these two schedule vectors are not achievable, the algorithm chooses a legal schedule vector such that $|\mathbf{s}_x| + |\mathbf{s}_y|$ is minimal. In fact, the schedule vector $\mathbf{s} = (1, 1)$ is always tried before choosing other candidates. When the chosen schedule vector satisfies $\mathbf{s} \cdot \mathbf{d}(e) > 0$, i.e., $\mathbf{s}$ is not parallel to any delay vectors, the final result is equivalent to that produced by the chained MD retiming which becomes a special case of SPINE-FULL algorithm.

Consider the example shown in Figure 4(a), SPINE algorithm uses a column-wise execution sequence to optimize the computation time as shown in Figure 4(c). While the chained MD retiming can choose schedule vector $\mathbf{s} = (1, 1)$. Remember that the chained MD retiming has to use a schedule vector such that $\mathbf{s} \cdot \mathbf{d}(e) > 0$ for any non-zero delay vector in the MDFG. Therefore, the schedule vector $\mathbf{s} = (0, 1)$ is not even considered because of the existence of delay vector (0,2). By using schedule vector $\mathbf{s} = (1, 1)$ the row-wise execution sequence cannot be maintained anymore, and the final code will have large overheads.

5. Experiments

In our experiments, we compare software-pipelined loops generated by three different approaches: the standard software pipelining technique, Modulo scheduling ("Modulo"), the standard MD retiming technique, chained MD retiming ("Chained"), 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.

Bench	Node	Cycle Period			Execution Time		
		SPINE	Modulo	Improv	SPINE	Modulo	Improv
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 Modulo scheduling and the SPINE-FULL algorithm. Column "Node" shows the number of nodes in a loop. Column "Improve" shows the percentage of the improvement on cycle period of loops by using the SPINE-FULL algorithm. 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 for all the cases. It indicates that the potential parallelism that can be explored by the standard software pipelining techniques on nested loops is very limited. The improvement on

cycle period by using SPINE is 73.8% on average. The improvement on execution time of the whole loop is 71.7%.

Bench	Retime	Code Size			Execution Time		
		SPINE	Chained	Improv	SPINE	Chained	Improv
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.

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, SPINE achieves full parallelism using schedule vector $\mathbf{s} = (1, 0)$, while the chained MD retiming needs to skew the execution sequence using schedule vector $\mathbf{s} = (1, 1)$. For the DPCM program, (0,k)- 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. The code size is reduced by 69.5% on average by using SPINE. The average improvement on execution time is 27.3%.

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 only explore the parallelism in one-dimension. Multi-dimensional retiming can fully parallelize a nested loop, but does not consider timing and code size overheads due to loop transformation. In this paper, we present the theory of Software Pipelining for NEsted loops (SPINE) based on the fundamental understanding of the properties of the software-pipelined nested loops using retiming concept. An efficient SPINE algorithm is developed to fully parallelize a nested loop with the minimal overheads. The experimental results show that our SPINE technique outperforms the standard software pipelining techniques, such as Modulo scheduling, for all our benchmarks. It is also superior to MD retiming technique for most of the cases in terms of the execution time and the code size of software-pipelined loop nests.

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. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing*, 10:207–223, 1995.

[4] 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.

[5] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, Aug. 1994.

[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] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.

[10] 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.

[11] N. L. Passos, E. H.-M. Sha, and S. C. Bass. Loop pipelining for scheduling multi-dimensional systems via rotation. In *Proc. 31st ACM/IEEE Design Automation Conf. (DAC)*, pages 485–490, 1994.

[12] J. Ramanujam. Optimal software pipelining of nested loops. In *Proc. 8th Int'l Parallel Processing Symposium*, pages 335–342, Apr. 1994.

[13] 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.

[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] Q. Zhuge, Z. Shao, and E. H.-M. Sha. Optimal code size reduction for software-pipelined loops on dsp applications. In *Proc. IACC Int'l Conf. on Parallel Processing (ICPP)*, pages 613–620, Aug. 2002.