# Optimal Code Size Reduction for Software-Pipelined and Unfolded Loops [*]

Qingfeng Zhuge, Bin Xiao, Zili Shao, Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083

Chantana Chantrapornchai
Faculty of Science
Silpakorn University
Nakorn Pathom, Thailand 73000

## ABSTRACT

Software pipelining and unfolding are commonly used techniques to increase parallelism for DSP applications. However, these techniques expand the code size of the application significantly. For most DSP systems with limited memory resources, code size becomes one of the most critical concerns for the high-performance applications. In this paper, we present the code size reduction theory based on retiming and unfolding concepts. We propose a code size reduction framework to achieve the optimal code size of software-pipelined and unfolded loops by using conditional registers. The experimental results on several well-know benchmarks show the effectiveness of our code size reduction technique in controlling the code size of optimized loops.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-Time and Embedded Systems*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*; D.3.4 [**Programming Languages**]: Procssors—*Compilers*

## General Terms

Design

## Keywords

Software pipelining, Retiming, Unfolding, Rotation scheduling

## 1. INTRODUCTION

Many real-time or high-performance DSP applications, such as telecommunication and image processing, exhibit intensive computations in a repetitive pattern such as loops. Software pipelining and unfolding are widely used in various compilers [2, 6, 8] to expose the parallelisms in these loops. The compiler of TI's TMS320C6000, a family of high-performance VLIW processors targeted toward Digital Signal Processing (DSP), is an example of using software pipelining to exploit the multiple functional units [4]. However, the drawback of using these optimization techniques is the expanded code size. Software pipelining introduces prologue and epilogue sections, which are the codes executed before entering and after leaving the new loop body. Unfolding expands a loop body and may result in codes that are outside of the unfolded loop body. For example, unfolding a loop with 10 instructions for 2 times, and then performing software pipelining with pipeline depth of 3, will result in a new code with size of 60 instructions. For some kind of architectures such as embedded systems, memory resources are limited. The program code size becomes a major concern for these systems. Two critical requirements, performance and code size, are conflicting with each other when using software pipelining and unfolding techniques to increase parallelisms. The difficult task is then left to the designers to decide the acceptable performance with code size constraint.

Some ad-hoc code size reduction techniques were used to reduce the prologue/epilogue produced by software pipelining. However, the quality of their techniques could not be guaranteed [4, 8]. The code size control technique such as code collapsing technique presented in [4] is developed to be applied on TMS320C6000 family. However, there is no theoretical framework presented in literature for code size control techniques of software-pipelined and unfolded loops.

In this paper, we present the theoretical foundation as well as code size reduction framework when software pipelining and unfolding techniques are used. The performance of software-pipelined applications after applying code size reduction can be further improved by some optimization techniques considering memory constraints and data prefetching [3, 10].

Our contributions in this paper are as follows:

1. Present the code size reduction theory based on fundamental understanding of retiming, software pipelining and unfolding.

2. Propose the code size reduction framework for controlling the code size of the software-pipelined and unfolded loops by using conditional registers without jeopardizing the performance.

3. Give the required number of conditional registers to achieve the optimal code size.

[*]This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, USA, and by NECTEC, NT-B-06-4D-16-509, Thailand.

4. Show that the approach which applies retiming first and then unfolding to a loop results in a smaller code size than the approach that applies unfolding first and then retiming.

The experiments show that our techniques can reduce the code size of the software-pipelined and unfolded loops up to 61%.

In Section 2, we introduce several basic concepts and principles used in the code size reduction technique, such as data flow graph, retiming and unfolding. In Section 3, we illustrate the code size reduction framework by using conditional registers. Then, Section 4 presents the code size reduction theorems that consider the code size requirement and the number of conditional registers needed to achieve the optimal code size. Section 5 presents the experimental results and Section 6 concludes the paper.

## 2. BASIC PRINCIPLES

In this section, we give an overview of some necessary concepts related to our code size reduction techniques.

### 2.1 Data Flow Graph

A data flow graph (DFG) $G = \langle V, E, d, t\rangle$ 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, $d$ is a function from $E$ to a set of non-negative integers, representing the number of delays between any two nodes, and $t$ is a function from $V$ to a set of positive integers, representing a computation time of each node.

The inter-iteration data dependencies are represented by edges with delays, indicated by edges with bar lines in the graph. An edge $e(u \rightarrow v)$ with delay $d(e)$ means the input data of node $v$ is generated by the computation of node $u$ which is in $d(e)$ iterations earlier. Iterative applications can be represented by cyclic data flow graphs. The dependencies within the same iteration are represented by edges with $d(e) = 0$, called intra-iteration dependency. A legal static schedule must obey both intra and inter-iteration dependencies. The *cycle period* of a DFG is defined as the computation time of the longest path without delay in the graph. The cycle period of a DFG corresponds to the minimum schedule length of one iteration when there is no resource constraint. In this paper, we assume the computation time is 1 time unit without special notation. The *iteration period* is defined as an average computation time it takes to compute one iteration of a loop body. Each cycle of a DFG imposes the lower bound on the iteration period. This lower bound is called *iteration bound* of a DFG. A schedule is said to be *rate-optimal* if its iteration period equals to its iteration bound.

### 2.2 Retiming and Unfolding

The *retiming* technique [7] can be applied on a data flow graph to improve the cycle period by evenly distributing the delays in the graph. For a retimed DFG $G = \langle V, E, d, t\rangle$, retiming operation is represented by retiming function $r : V \rightarrow Z$. The retiming function value $r(u)$ represents the number of delays pushed through a node $u \in V$. The number of delays of an edge $e(u \rightarrow v)$ after applying retiming $r$ is $d_r(e) = d(e) + r(u) - r(v)$. For any legal retiming $r$, we have $d_r(e) \geq 0$, and the total number of delays remains constant for any cycle in the graph.

For example, consider node A in the DFG in Figure 1(a). Since $r(A) = 1$, one delay is drawn from all incoming edges of $A$ and pushes through all outgoing edges of $A$. Figure 1(b) shows the retimed graph where the bar line represents a delay on the edge.

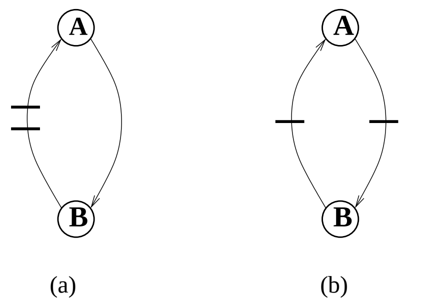

**Figure 1: (a) A simple DFG. (b) The retimed DFG with $r(A) = 1$ and $r(B) = 0$.**

Consider the retimed graph in Figure 1(b). When a delay is pushed through node A to its outgoing edge, the actual effect on the schedule of the new DFG is that the $i^{th}$ copy of A is shifted up and is executed with $(i-1)^{th}$ copy of node B. The schedule length of the new loop body is then reduced from two control steps to one control steps. Hence, *every retiming operation corresponds to a software pipelining operation*. When one delay is pushed forward through a node $u$, every copy of this node is shifted up by one iteration, and the first copy of the node is shifted out of the first iteration into the prologue.

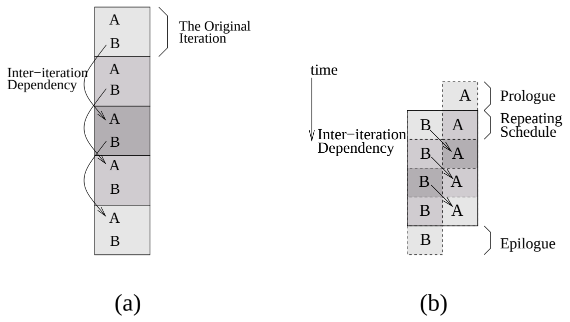

**Figure 2: (a) A static schedule of original loop. (b) The pipelined loops.**

We can measure the size of prologue and epilogue if the minimum retiming value in the graph is 0. This can be done by simply subtracting all the retiming values by the minimum retiming value, which is called normalized retiming function. When $r(v)$ delays are pushed forward through node $v$, there are $r(v)$ copies of node $v$ appeared in the prologue. Let the maximum retiming value in the data flow graph be $max_u r(u)$, then the number of copies of node $v$ appeared in the epilogue is $max_u r(u) - r(v)$. After retiming, the new loop body is repeated for $n - max_u r(u)$ times where $n$ is the number of iterations in the original loop.

Unfolding technique is popularly used in a compiler design. For a give data flow Graph $G = (V, E, d, t)$, given an *unfolding factor* $f$, the unfolded graph $G_f = (V_f, E_f, d_f, t)$ is obtained by unfolding $G$ for $f$ times. In particular, all nodes $v \in V$ are duplicated into $f$ copies in $V_f$ and the edges and their delays are adjusted properly [2]. Unfolding exposes more parallelism among different iterations. To generate a rate-optimal schedule, a commonly used method is to unfold the loop body and then do software pipelining. Obviously, the code size grows proportionally with the unfolding factor. Besides, if the number of iterations in the original loop program $n$ is not divisible by the unfolding factor $f$, the last $n(mod)f$ iterations need to be executed out of loop body. These remaining iterations add more code size in an unfolded program. Due to the significant code size expansion resulted from performance optimization, controlling code size of optimized loops becomes even more critical for embedded systems with limited memory space.

## 3. CODE SIZE REDUCTION FRAMEWORK

In this section, we propose the code size reduction framework by using *conditional register* and *conditional operation* to implement code size reduction technique. We will show that this technique can achieve the optimal code size of software-pipelined and/or unfolded loops without jeopardizing the performance by and large.

### 3.1 Conditional Register and Conditional Operation

Conditional registers and conditional operations are commonly implemented in many architectures, such as TI's TMS320C6000 family [4,9] and IA64 architecture [5]. Conditional registers are sometimes called predicate registers. Generally speaking, these registers only hold boolean values which is the result of a conditional test. In some architecture, conditional registers are a subset of general register file [4]. The execution of a conditional instruction depends on the result of the conditional test. If the condition test is true, the instruction is executed. Otherwise, the instruction is disabled or nullified. Such guarded instructions are also called conditional operations.

Using the conditional register, an instruction in the loop body may be guarded in a way that it starts execution in a later iteration. For example, the following code,

```
p1 = 1;
for i = 1 to n do
   (p1) A[i] = E[i-4] + 9;
        p1 = p1 - 1;
end
```

indicates that the computation of node $A$ can be executed from the second iteration. In other words, when $p1 > 0$, the instruction at node $A$ is disable. Conditional test is conducted at the beginning of the execution stage in many implemented architectures [5,9]. Then, the value of conditional register $p1$ is decreased every iteration.

### 3.2 Code Size Reduction for Retimed Loops

The conditional operation can be used to reduce the code in the prologue and epilogue introduced by the retiming technique. We set the initial value of conditional registers as the maximum retiming value minus the retiming value of the guarded computation node $v$, i.e. $p = max_u r(u) - r(v)$. We have to specify that the instruction is executed only when $0 \geq p > -LC$. In other words, the instruction is disabled when $p > 0$ or $p \leq -LC$, where $LC$ represents the original loop counter. Hence, the loop boundary must be specified in the conditional register.

We propose a new instruction to set the initial value and boundary of a conditional register.

```
setp p1 = 3 : -LC
```

This instruction sets the initial value of $p1$ to 3. Also, the guarded instruction will be disabled when $p1 > 0$ or $p1 \leq -LC$. The value of conditional register $p$ is decreased by 1 in every iteration using an explicit decrement instruction. If $r(A) < max_u r(u)$, the execution of the operation will be disabled in the first $(max_u r(u) - r(A))$ iterations,

```
A[1] = E[-3] + 9;
A[2] = E[-2] + 9;
B[1] = A[1] * 5;
C[1] = A[1] + B[-1];
A[3] = E[-1] + 9;
B[2] = A[2] * 5;
C[2] = A[2] + B[0];
D[1] = A[1] * C[1];
for i = 1 to n-3 do
   A[i+3] = E[i-1] + 9;
   B[i+2] = A[i+2] * 5;
   C[i+2] = A[i+2] + B[i];
   D[i+1] = A[i+1] * C[i+1];
   E[i] = D[i] + 30;
end
E[n] = D[n] + 30;
D[n] = A[n] * C[n];
E[n-1] = D[n-1] + 30;
B[n] = A[n] * 5;
C[n] = A[n] + B[n-2];
D[n-1] = A[n-1] * C[n-1];
E[n-2] = D[n-2] + 30;
```

(a)

```
p1 <-- 0;  // setp p1 = 0 : -n
p2 <-- 1;
p3 <-- 2;
p4 <-- 3;
for i = -2 to n do
   (p1) A[i+3] = E[i-1] + 9;
        p1 = p1 - 1;
   (p2) B[i+2] = A[i+2] * 5;
   (p2) C[i+2] = A[i+2] + B[i];
        p2 = p2 - 1;
   (p3) D[i+1] = A[i+1] * C[i+1];
        p3 = p3 - 1;
   (p4) E[i] = D[i] + 30;
        p4 = p4 - 1;
end
```

(b)

| | | | | | |
|---|---|---|---|---|---|
| ~~(2)D[−1]~~ | ~~(1)B[0]~~ | ~~(3)E[−2]~~ | ~~(1)C[0]~~ | (0)A[1] | |
| ~~(1)D[0]~~ | (0)B[1] | ~~(2)E[−1]~~ | (0)C[1] | (−1)A[2] | Prologue |
| (0)D[1] | (−1)B[2] | ~~(1)E[0]~~ | (−1)C[2] | (−2)A[3] | |
| (−1)D[2] | (−2)B[3] | (0)E[1] | (−2)C[3] | (−3)A[4] | Static Schedule |
| (−2)D[i+1] | (−3)B[i+2] | (−1)E[i] | (−3)C[i+2] | (−4)A[i+3] | |
| (−3)D[n−1] | (−4)B[n] | (−2)E[n−2] | (−4)C[n] | ~~(−4)A[n+1]~~ | |
| (−4)D[n] | ~~(−5)B[n+1]~~ | (−3)E[n−1] | ~~(−4)C[n+1]~~ | ~~(−4)A[n+2]~~ | Epilogue |
| ~~(−5)D[n+1]~~ | ~~(−6)B[n+2]~~ | (−4)E[n] | ~~(−4)C[n+2]~~ | ~~[−4]A[n+3]~~ | |

(c)

**Figure 3: (a) Software-pipelined code. (b) Code after removing prologue/epilogue. (c) Execution sequence.**

until the conditional register value is decremented downto $-LC$. The comparison between the value of conditional register and the negative loop counter (-LC) is implemented by hardware.

Hence, we can use conditional registers and the retiming function to eliminate *all* the code in prologue and epilogue. Figure 3(b) shows the code after removing prologue/epilogue of the code in Figure 3(a). The conditional registers p1, p2, p3 and p4 are used for different retiming values for nodes A, B and C, D as well as E respectively. Each of them is initialized to a different value depending on its retiming value, and is decremented for every iteration. Since there are four different retiming values, we need to use four conditional registers to completely remove prologue and epilogue. A decrement instruction needs to be inserted into the loop body for each register. For VLIW architecture, the inserted instructions can be put into a slot of the long instruction word wherever possible after all the guarded instructions are issued. The decrement instructions can also be parallelized with other instructions by software pipelining. In most cases, code size reduction does not hurt the performance of an optimized loop. Note that the loop will now be executed for $n - 3 + 3 + 3 = n + 3$ times, since it first decreases 3 iterations by software pipelining, which is $max_u r(u)$ in this example, and then adds 3 iterations of prologue and the other 3 of epilogue. Figure 3(c) shows the execution sequence of the conditional operations in our implementation. The numbers in parentheses are the values of conditional registers.

```
for i = 1 to n do
      A[i] = B[i-3] * 3;
      B[i] = A[i]+ 7;
      C[i] = B[i] * 2;
end
```

**Figure 4: A Simple loop**

## 3.3 Code Size Reduction for Unfolded Loops

Consider the code in Figure 4. After unfolded three times, we have the code in Figure 5(a). If $n$ is not divisible by 3, in the unfolded code, the last two iterations of the original loop is performed outside the loop in Figure 5(a). Figure 5(b) shows the new code that the execution of the these remaining iterations are smaller. We obtain the minimal code size by using only *one* conditional register $p1$. For any unfolded loops, we need only one conditional register, namely $R$ to completely remove the expanded code outside the new unfolded loop body. The initial value of $R$ is set to 0 and -LC. Value inside $R$ is decremented by unfolding factor $f$ for each iteration. When $R$ equals to -LC, the loop will stop execution. Suppose the original code size of the loop is $L_{orig}$, we can totally reduce $(n \bmod f) * L_{orig} - 2$ instructions if $n$ is not divisible by unfolding factor $f$.

```
for i = 1 to 3⌊n/3⌋ do by 3
      A[i] = B[i-3] * 3;
      B[i] = A[i]+ 7;
      C[i] = B[i] * 2;
      A[i+1] = B[i-2] * 3;
      B[i+1] = A[i+1]+ 7;
      C[i+1] = B[i+1] * 2;
      A[i+2] = B[i-1] * 3;
      B[i+2] = A[i+2]+ 7;
      C[i+2] = B[i+2] * 2;
end
A[n-1] = B[n-3] * 3;
B[n-1] = A[n-1]+ 7;
C[n-1] = B[n-1] * 2;
A[n] = B[n-2] * 3;
B[n] = A[n]+ 7;
C[n] = B[n] * 2;
```

(a)

```
p1 = 0;
for i = 1 to 3⌊n/3⌋ +1 do by 3
      A[i] = B[i-3] * 3;
      B[i] = A[i]+ 7;
      C[i] = B[i] * 2;
      (p1) A[i+1] = B[i-2] * 3;
      (p1) B[i+1] = A[i+1]+ 7;
      (p1) C[i+1] = B[i+1] * 2;
      (p1) A[i+2] = B[i-1] * 3;
      (p1) B[i+2] = A[i+2]+ 7;
      (p1) C[i+2] = B[i+2] * 2;
      p1 = p1 - 3;
end
```

(b)

**Figure 5: (a) Unfolded loop when $f = 2$. (b) New unfolded code.**

## 3.4 Code Size Reduction for Retimed and Unfolded Loops

For the loops optimized by both retiming and unfolding, the previous technique can reduce the expanded code size. We call the loop that gets retimed first and then unfolded a *retimed unfolded* loop and the loop that unfolded first and then retimed a *unfolded retimed* loop. We will show the code reduction technique for retimed unfolded loops does not require more conditional registers.

Consider the original loop body shown in Figure 4. Figure 7(a) shows the new code after reducing the code size of Figure 6(a). We use two conditional registers to guard statements $A$, $C$ and $B$, called $p1$ and $p2$. The registers are initially set to 1 and 0, respectively. Figure 7(b) shows the new code when the size of the retimed unfolded program in Figure 6(b) is reduced. We can use the same set of registers to completely remove the code outside the loop body. The idea is to hide the expanded code into unfolded iterations. Since the number of iterations in prologue is $max_u r(u)$, we have $\lceil \frac{max_u r(u)}{f} \rceil$ unfolded iterations for prologue. For the codes after the loop body, we have $\lceil \frac{max_u r(u) \% f + max_u r(u)}{f} \rceil$ iterations. By doing this, the lower bound of iteration body is adjusted to be $k - \lceil \frac{\max_u r(u)}{f} \rceil$, where $k$ is the original lower bound. Also, the initial value of a conditional register is adjusted to be $R + (f - max_u r(u) \% f)$.

For example, if $n = 9$, and $i$ run from 0 to 8. The index in the new loop starts from -2 and runs until 9. The initial value of registers p2 and p1 are set to 3 and 2 respectively. Figure 7(c) shows that in the first iteration when $i = -2$, only A[0] and C[0] are computed. All other operations are disabled. And the last iteration computes the remaining codes of unfolding and A[8] and C[8] as epilogue. Since each copy of a computation node uses the same conditional register in unfolded loop body, we use the same number of conditional registers for a retimed loop and a retimed unfolded loop with the same maximum retiming value.

For the case when unfolding is performed before retiming, each copy of the node may be retimed as a distinct node, therefore, the number of conditional register may be more than retimed unfolded loop. The code size reduction technique applied on unfolded retimed loop is straight forward after we giving the technique for retimed loop, we will not describe it in detail due to the limited space.

## 4. CODE SIZE REDUCTION THEOREMS

In this section, we present the theory of code size reduction technique based on the retiming and unfolding concepts. The code size reduction technique is a code transformation that attempts to remove the code expansion produced by software pipelining and unfolding, so that the total code size of a loop program is minimized. The theorems show that our code size reduction technique can achieve the optimal code size for retimed and/or unfolded loops, while preserving the correctness of the execution of the expanded code, i.e. the prologue/epilogue produced by retiming and the remaining iterations produced by unfolding.

```
B[1] = A[1]+ 7;
for i = 1 to n-1 do
      A[i] = B[i-3] * 3;
      B[i+1] = A[i+1]+ 7;
      C[i] = B[i] * 2;
end
A[n] = B[n-2]*3;
C[n] = B[n] * 2;
```

(a)

```
B[1] = A[1]+ 7;
for i = 1 to 3⌊(n-1)/3⌋ do by 3
      A[i] = B[i-3] * 3;
      B[i+1] = A[i+1]+ 7;
      C[i] = B[i] * 2;
      A[i+1] = B[i-2] * 3;
      B[i+2] = A[i+2]+ 7;
      C[i+1] = B[i+1] * 2;
end
A[n-2] = B[n-5] * 3; //from unfolding
B[n-1] = A[n-1]+ 7;
C[n-2] = B[n-2] * 2;
A[n-1] = B[n-4] * 3;
B[n] = A[n]+ 7;
C[n-1] = B[n-1] * 2;
A[n] = B[n-2]*3; //from retiming
C[n] = B[n] * 2;
```

(b)

**Figure 6: (a) Retimed loop when $r(B) = 1$ (b) The new loop after unfolded by a factor of 3.**

In the following two theorems, we show the correctness of the code size reduction technique for a software-pipelined loop. Due to the limited space, we omit the proofs of the theorems.

THEOREM 4.1. *Let $G_r = \langle V, E, d_r, t \rangle$ be a retimed DFG with a given retiming function r, and $M_r = max_u r(u), \forall u \in V$. Let n be the number of iterations of the original loop. The prologue can be correctly replaced by conditionally executing the loop body of $G_r$ for $M_r$ times, where a node $v \in V$ with retiming value $r(v)$ is executed for $r(v)$ times starting from the $(M_r - r(v) + 1)^{th}$ iteration.*

For example, if $r(v) = 3$ and $M_r = 5$, then node $v$ will not be executed in the first and the second iterations. Instead, it will start its execution from the third iteration. Similar situation can be applied to the epilogue.

THEOREM 4.2. *Let $G_r = \langle V, E, d_r, t \rangle$ be a retimed DFG with a given retiming function r, and $M_r = max_u r(u), \forall u \in V$. Let n be the number of iterations of the original loop. The epilogue can be correctly executed by conditionally executing the loop body of $G_r$ for $M_r$ times, where a node $v \in V$ with retiming value $r(v)$ is executed for $M_r - r(v)$ times in the last $M_r$ iterations starting from the $n^{th}$ iteration.*

Theorem 4.1 and Theorem 4.2 imply that the code in prologue or epilogue can be removed by conditionally executing the schedule of loop body. Therefore, the minimal code size required for a correct execution is only the code size of the loop body.

Based on the retiming concept, the number of copies of the nodes in prologue/epilogue are determined by their retiming functions as we discussed in Section 2. This property helps us to develop the code size reduction technique by using conditional register to guard the instructions with a certain retiming value, as shown in Section 3. The following theorem shows that we can *completely* remove the code in prologue/epilogue by using a certain number of conditional registers.

THEOREM 4.3. *(Total Code Reduction for Retimed Loop) Let $P_r$ be the number of available conditional registers, and $N_r$ the set of distinct*

```
p1 = 1; p2 =0;
for i = 0 to n do
      (p1) A[i] = B[i-3] * 3;
      (p2) B[i+1] = A[i+1]+ 7;
      (p1) C[i] = B[i] * 2;
      p1 = p1-1;
      p2 = p2-1;
end
```

(a)

```
p1 = 3; p2 = 2;
for i = -2 to 3⌊(n-1)/3⌋ + 2 do by 3
      (p1) A[i] = B[i-3] * 3;
      (p2) B[i] = A[i]+ 7;
      (p1) C[i] = B[i] * 2;
      p1 = p1-1;
      p2 = p2-1;
      (p1) A[i+1] = B[i-2] * 3;
      (p2) B[i+1] = A[i+1]+ 7;
      (p1) C[i+1] = B[i+1] * 2;
      p1 = p1-1;
      p2 = p2-1;
      (p1) A[i+2] = B[i-1] * 3;
      (p2) B[i+2] = A[i+2]+ 7;
      (p1) C[i+2] = B[i+2] * 2;
      p1 = p1-1;
      p2 = p2-1;
end
```

(b)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ~~A[-3]~~ | ~~B[-2]~~ | ~~C[-3]~~ | ~~A[-2]~~ | ~~B[-1]~~ | ~~C[-2]~~ | ~~A[-1]~~ | B[0] | ~~C[-1]~~ | prologue |
| A[0] | B[1] | C[0] | A[1] | B[2] | C[1] | A[2] | B[3] | C[2] | |
| A[3] | B[4] | C[3] | A[4] | B[5] | C[4] | A[5] | B[6] | C[5] | |
| A[6] | B[7] | C[6] | A[7] | B[8] | C[7] | A[8] | ~~B[9]~~ | C[8] | remaining code & epilogue |

(c)

**Figure 7: (a) New code for Figure 6(a). (b) New code for Figure 6(b). (c) Execution sequence when $n = 9$.**

*retiming values. The optimal code size of a retimed loop program can be achieved by completely removing prologue and epilogue with $P_r$ conditional registers, if $P_r \geq |N_r|$, where $|N_r|$ is the cardinality of set $N_r$.*

For example, if we have 3 distinct retiming values, {0,3,4}, we need at least 3 conditional registers to remove all the codes in prologue and epilogue. Without code size reduction, prologue and epilogue each contains code of 4 iterations, since the maximum retiming value is 4.

By using unfolding, there are remaining iterations produced out of unfolded loop body. For example, if the number of iteration is 98, and unfolding factor is 3, the last two iterations need to be put out of the loop body as additional codes. Let $Q_f$ represent the code size of the remaining iterations, we have $Q_f = (n(mod)f) * L_{orig}$, where $n$ is the number of iterations in the original loop, $f$ is the unfolding factor and $L_{orig}$ is the code size of original loop body. Our technique can be used to remove the code of remaining iterations *completely* by using one conditional register as we showed in Section 3.

Retiming and unfolding are two commonly used techniques to improve performance by exploiting the parallelism existing among different iterations. Retiming, or software pipelining, can achieve the optimal execution rate when iteration period is integer. For a DFG with a non-integral iteration period, only retiming cannot achieve optimal execution rate. There are two possible orders of applying retiming and unfolding to a loop: first, which is the commonly used approach, applying retiming first and then unfolding and applying unfolding first and then retiming. Previous work by Chao and Sha [1] showed that these two approaches can achieve the same minimum iteration period by carefully choose the retiming values. In the following theorems, DFG $G_{f,r}$ is generated by unfolding the original DFG $G$, and then retiming the unfolded graph; DFG $G_{f,r}$ is generated by directly retiming the original DFG $G$, and then unfolding it. we show that the code sizes of the second approach, retiming first and then unfolding, may produce smaller code size than the first one.

THEOREM 4.4. *Let $G = \langle V, E, d, t \rangle$ be a DFG and $G_{f,r} = \langle V_{f,r}, E_{f,r}, d_{f,r}, t \rangle$ be the unfolded retimed $G$ with unfolding factor $f$ and retiming function $r$. Let the maximum retiming value $M_r = max_u r(u)$, $\forall u \in V_{f,r}$ and $L_{orig}$ the code size of the original loop body. The code size of $G_{f,r}$ is $S_{f,r} = (M_r + 1) * L_{orig} * f + Q_f$, where $Q_f$ represents the remaining iterations produced by unfolding.*

THEOREM 4.5. *Let $G = \langle V, E, d, t \rangle$ be a DFG and $G_{f,r} = \langle V_{f,r}, E_{f,r}, d_{f,r}, t \rangle$ be the unfolded retimed $G$ with unfolding factor $f$ and retiming function $r$. Let $u_i \in V_{f,r}$ be the $i^{th}$ copy of node $u \in V$, where $0 \leq i < f$. Let $G_{r,f} = \langle V_{r,f}, E_{r,f}, d_{r,f}, t \rangle$ be the corresponding retimed unfolded $G$ with retiming function $r_f = \sum_{i=0}^{f-1} r(u_i)$, $u_i \in V_{f,r}$ and unfolding factor $f$, that achieves the same minimum cycle period as $G_{f,r}$. Let $L_{orig}$ be the code size of $G$. The code size of $G_{r,f}$ is $S_{r,f} = (max_u(r_f(u)) + f) * L_{orig} + Q_f$, where $Q_f$ represents the remaining iterations produced by unfolding.*

Based on the Theorem 4.4 and Theorem 4.5, we have $S_{f,r} = (max_u r(u) + 1) * L_{orig} * f$ and $S_{r,f} = (max_u(\sum_{i=0}^{f-1} r(u_i)) + f) * L_{orig}$. Since $max_u(\sum_{i=0}^{f-1} r(u_i)) \leq max_u r(u) * f$, one obtains $S_{r,f} \leq S_{f,r}$. That is, the approach that creates an unfolded retimed graph produces larger code size than the one that creates a retimed unfolded graph.

The code size reduction technique can be effectively applied to data flow graphs produced by retiming and unfolding. The following theorem states that the correct execution sequence of prologue can be preserved by only executing the code of unfolded loop body. It follows from Theorem 4.1 and Theorem 4.2, considering the unfolding factor. The idea is to hide the iterations performed in prologue $M_r$ in the new unfolded loop. If $M_r$ is not divisible by unfolding factor $f$, dummy iterations are added to fill the first iteration, which can be computed by $Q_{head} = (f - M_r \bmod f) \bmod f$

THEOREM 4.6. *Let $G = \langle V, E, d, t \rangle$ be a DFG and $G_{r,f} = \langle V_{r,f}, E_{r,f}, d_{r,f}, t \rangle$ be the retimed unfolded $G$ with given retiming function $r$ and unfolding factor $f$, and $M_r = max_u r(u), \forall u \in V$. The prologue can be correctly replaced by executing the loop body of $G_{r,f}$ for $\lceil \frac{M_r}{f} \rceil$ times, where node $v \in V$ with retiming value $r(v)$ is executed for $r(v)$ times starting from $(M_r - r(v) + Q_{head} + 1)^{th}$ iteration of $G$.*

We can do similar transformation to the epilogue and the remaining iterations by hiding the code in the new unfolded loop. The conditional registers that guard these nodes will ensure each node is executed for exactly $n$ times. For the retiming and then unfolding approach with retiming function $r$ and unfolding factor $f$, there are $\lceil \frac{max_u r(u)}{f} \rceil$, additional iterations for completely removing the epilogue and remaining iterations of the unfolded loop body. For a node $v$ with retiming value $r(v)$, it will be executed for $((n - max_u r(u)) \bmod f + max_u r(u) - r(v)$ times in these additional iterations.

According to Theorem 4.7, we can *completely* remove the expanded code size of a retimed unfolded loop by using conditional registers. An elegant property of our code size reduction technique is that the number of consumed conditional registers is not increased.

THEOREM 4.7. *(Total Code Reduction for Retimed and Unfolded Loop) Let $G_r = \langle V, E, d_r, t \rangle$ be a retimed DFG and $G_{r,f} = \langle V_{r,f}, E_{r,f}, d_{r,f}, t \rangle$ be the unfolded $G_r$. Let $r$ be a given retiming function and $f$ a given unfolding factor. The optimal code size of a retimed unfolded loop program can be achieved by completely removing prologue/epilogue and remaining iterations with $P_{r,f}$ conditional registers, such that $P_{r,f} = P_r$, where $P_r$ is the number of conditional registers used for removing the prologue and epilogue of $G_r$.*

Because the code size is increased proportionally with the unfolding factor, we can compute the maximum unfolding factor by given the retiming value and code size requirement. Let $L_{req}$ denote the code size requirement, and $L_{orig}$ denote the code size of original loop body. The maximum unfolding factor given the retimed loop body is $M_f = \lfloor \frac{L_{req}}{L_{orig}} \rfloor - M_r$. On the other hand, by given the unfolding factor and code size requirement, we can also obtain the maximum retiming value by $M_r = \lfloor \frac{L_{req}}{L_{orig}} \rfloor - f$. Based on the fundamental understanding of code size reduction for retimed and/or unfolded loop program, we can explore the trade-off space for the code size and the performance achieved by software pipelining and unfolding, which is important for choosing an appropriate software pipelining degree and unfolding factor in code optimization.

| **Benchmarks** | Code Size | | | Rgs | % Red. |
|---|---|---|---|---|---|
| | Orig | Ret. | **CR** | | |
| IIR Filter | 8 | 16 | **12** | 2 | 25.0 |
| Differential Equation | 11 | 33 | **17** | 3 | 48.5 |
| All-pole Filter | 15 | 60 | **23** | 4 | 61.7 |
| Elliptical Filter | 34 | 68 | **40** | 3 | 41.2 |
| 4-stage Lattice Filter | 26 | 78 | **32** | 3 | 59.0 |
| Voltera Filter | 27 | 54 | **31** | 2 | 42.6 |

**Table 1: Code size after retiming and registers needed.**

## 5. EXPERIMENTAL RESULTS

In Table 1, Column "Code size" shows the original code size and after retimed for various benchmarks to obtain the optimal cycle period. The code size is measured as the number of nodes in a schedule including prologue and epilogue parts. We assume that the computation time of each node is one time unit. Under Column "Rgs", we show the number of registers that are needed to totally remove all these prologue and epilogue incurred by retiming. Column "Retime" shows the code size after retiming is applied to achieve the rate-optimal cycle period. Column "CR" shows the code size when the code size reduction is applied to the graphs obtained in Column "Ret.". Column "% Red" presents the percentage of code size reduction after these codes are removed. Similarly,

| **Benchmarks** | Code Size | | Rgs | % Red. |
|---|---|---|---|---|
| | R-U | **CR** | | |
| IIR Filter | 48 | **32** | 2 | 33.3 |
| Differential Equation | 77 | **45** | 3 | 41.6 |
| All-pole Filter | 120 | **61** | 4 | 49.2 |
| Elliptical Filter | 238 | **114** | 3 | 52.1 |
| 4-stage Lattice Filter | 182 | **90** | 3 | 50.5 |
| Voltera Filter | 168 | **89** | 2 | 47.0 |

**Table 2: Code size after retiming and unfolding and registers needed, with unfolding factor 3 and loop counter 101.**

Table 2 shows the code size of the benchmarks in Table 1 after unfolding are applied. Column "R-U" shows the code size after unfolding is applied to these graphs. We can see that our technique can significantly reduce the code size while maintaining the performance of the code.

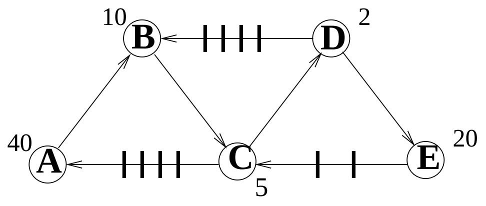

**Figure 8: An example from [1] where nodes computation times are not unit-time.**

Tables 3–4 shows the code size when applying unfolding and retiming in a different order with different unfolding factors. In the row "unfold-retime", is the case where unfolding is performed first and then retimed and vice versa in Row "retime-unfold". Row "retime-unfold-CR" show the code size when our proposed technique is used. In Table 3 we fixed the performance of the optimization by setting an iteration period for each unfolding factor to make a fair comparison. As stated in Theorems in Section 4, we suggest that performing retiming first and unfolding give less code size. When applying our technique using conditional registers, the code size is further reduced.

We can see that our approach can be used to effectively reduce code size when retiming and/or unfolding is performed while maintaining parallelism as in the original code. Further, our approach can be integrated in a design exploration in many aspects, for example, to find the maximum performance when the number of conditional registers are limited or explore different architectures (eg. memory size and conditional registers) to achieve the maximum performance.

| Approach | uf=2 | uf=3 | uf=4 |
|---|---|---|---|
| unfold-retime | 20 | 30 | 40 |
| retime-unfold | 20 | 30 | 30 |
| **retime-unfold-CR** | **14** | **19** | **24** |
| iteration period | 20 | 19 | 13.5 |

**Table 3: Comparison of code size and iteration period for DFG in Figure 8.**

## 6. CONCLUSION

Retiming and unfolding technique are commonly used techniques to exploit instruction-level parallelism and achieve performance gain in DSP systems. However, both techniques can enlarge code size by adding prologue and epilogue sections to the original code. The more parallelism is exposed, the more code size is expanded. For some systems with a very limited memory resource, the code size expansion can be a critical concern. In this paper, we present theoretical foundations and code size reduction framework while the parallelism can still be explored as in the original code. Our study shows that the approach that applies retiming and then unfolding will achieve a smaller code size than applying unfolding first and then retiming. The experimental results show that our technique can achieve optimal code size without degrading the performance by using conditional registers.

| Approach | uf=2 | uf=3 | uf=4 |
|---|---|---|---|
| unfold-retime | 156 | 312 | 416 |
| retime-unfold | 130 | 156 | 182 |
| **retime-unfold-CR** | **61** | **90** | **119** |

**Table 4: Comparison of code size for 4-stage lattice when cycle period is fixed to 8.**

## 7. REFERENCES

[1] 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.

[2] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, Dec. 1997.

[3] F. Chen, T. W. O'Neil, and E. H.-M. Sha. Optimizing overall loop schedules using prefetching and partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 11:604–614, Jun. 2000.

[4] 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 *Proceedings of the 3rd Workshop on Media and Streaming Processorsin conjunction with 34th Annual International Symposium on Microarchitecture*, pages 29–38. ACM, Dec. 2001.

[5] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, Dec. 2001.

[6] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328. ACM, June 1988.

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

[8] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169. ACM, Dec. 1992.

[9] Texas Instruments, Inc. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

[10] Z. Wang, T. W. O'Neil, and E. H.-M. Sha. Minimizing average schedule length under memory constraints by optimal partitioning and prefetching. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 27:215–233, Jan. 2001.