

Optimal Code Size Reduction for Software-Pipelined Loops on DSP Applications *

Qingfeng Zhuge, Zili Shao, Edwin H. -M. Sha
qfzhuge@utdallas.edu, zxs015000@utdallas.edu, edsha@utdallas.edu
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083

Abstract

Code size expansion of software-pipelined loops is a critical problem for DSP systems with strict code size constraint. Some ad-hoc code size reduction techniques were used to try to reduce the prologue/epilogue produced by software pipelining. This paper presents the fundamental understanding of the relationship between code size expansion and software pipelining. Based on the retiming concept, we present a powerful Code-size REDuction (CRED) technique and its application on various kinds of processors. We also provide CRED algorithms integrated with software pipelining process. One advantage of our algorithms is that it can explore the trade-off space between “perfect” software pipelining and constrained code size. That is, the software pipelining process can be controlled to generate a schedule concerned with code size requirement. The experiment results show the effectiveness of our algorithms in both reducing the code size for software-pipelined loops and exploring the code size/performance trade-off space.

Keywords: Retiming, DSP processors, Software pipelining, Scheduling

1 Introduction

Many real-time or high-performance DSP applications, such as telecom and image processing, exhibit intensive computations in loops. Software pipelining is widely used to improve the execution rate of these applications by exploiting the instruction-level parallelism in loops [4, 8]. The compiler of TI’s TMS320C6000, a family of high-performance VLIW processors targeted toward Digital Signal Processing (DSP), is a good example of using software pipelining to exploit the multiple functional units [2]. While software pipelining helps to achieve compact schedules, its disadvantage is the introduction of prologue and epilogue sections, which cause code size expansion. This code size overhead is a significant concern for embedded systems because of the limitation of on-chip memory size. Furthermore, the size of prologue and epilogue grows proportionally as more iterations of the loop

get overlapped in the pipeline [8]. For embedded systems, these two critical requirements, performance and code size, are conflict with each other when using software pipelining to optimize loop schedule. The difficult task of making trade-off between them is left to the compiler.

A simple *for* loop and its transformation after applying software pipelining are shown in Figure 1(a) and Figure 1(b). We can see that the size of prologue and epilogue size is about three times of the kernel code size when the loop schedule length is reduced from four control steps to one control step. Figure 2(a) shows the execution flow of the original loop. Figure 2(b) illustrates the loop pipeline where the execution of the nodes from different iterations are overlapped. The shaded region represents one iteration of the original loop. We can see that a new iteration is initiated in every clock cycle. In the steady state of the pipeline, all the operations can be executed simultaneously within one control step. But the price paid for this significant performance gain is the same significant code size overhead in prologue and epilogue.

```
for i = 1 to n do
  A[i] = E[i-4] + 9;
  B[i] = A[i] * 5;
  C[i] = A[i] + B[i-2];
  D[i] = A[i] * C[i];
  E[i] = D[i] + 30;
end
```

(a)

```
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;
```

(b)

Figure 1. (a) Original loop. (b) The loop after applying software pipelining.

Previous work on code size reduction of software-pipelined loop such as code collapsing technique was developed particu-

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

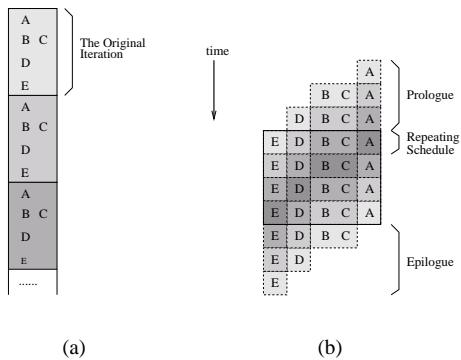


Figure 2. (a) A static schedule of original loop. (b) The model of loop pipeline.

larly for TI’s TMS320C6000 family [2]. The quality of this approach, however, cannot be guaranteed. Kernel-only code generation schema presented in [8] is specially applied to IA64 processor. It requires special hardware support, and this kind of hardware support is not found in DSP processors. There’s no theoretical framework presented in literature for code size control of pipelined loops. More importantly, we believe that by casting a retiming view on the software pipelining, the code size control problems can be unified into one problem of rebuilding the kernel code in prologue/epilogue by using only the retiming functions. We build up a theoretical base to expose the relations between code size of prologue/epilogue and the retiming functions of the computation nodes in the loop.

Our algorithms are developed to remove the iterations of prologue/epilogue by conditionally executing the kernel code. The values of conditional registers are determined by retiming function, which can be known during the software pipelining process. Furthermore, the underlying relationship between the software pipeline depth and the code size increment can be exploited to help compiler exploring the decision space between performance gain and code size control. As we know, even if the loop can be highly pipelined, the generated loop schedule may not be used in a real situation, if it’s found that the code size requirement cannot be satisfied. So it is necessary that the software pipelining process can be controlled beforehand to meet a certain code size requirement. We show that our technique can control the software pipelining “degree” to achieve acceptable performance improvement, and at the same time, to cope with the code size requirement.

In this paper, we establish the theory and technique of code size reduction based on retiming concept. The input code description is transformed into a data flow graph, whose nodes represent operations to be performed, and edges represent precedent relations. The cyclic data flow graph can represent iterative algorithms or *for/while* loops in description language. We also depict a loop pipeline as a three-component object: prologue, repeating schedule and epilogue, to reflect the actual repeating pattern of the pipeline schedule. The length of the repeating schedule represents the execution rate of the pipeline at a stable state. We use the

retiming to model software pipelining process, such that the prologue, new loop body and epilogue can be derived directly from retiming functions. A flexible scheduling technique presented by Chao and Sha, Rotation Scheduling [1], is used to generate the pipelined schedule. Because it applies retiming operations implicitly to achieve pipelined schedule with resource constraint.

Our contributions are:

1. Establish the theoretical foundation of code size reduction technique for software-pipelined loops. The retiming concept is used to interpret the software pipelining in our model.
2. Present a general Code-size REDuction (CRED) technique which can be implemented on different types of processors.
3. Explore the trade-off space between the perfect software pipelining and the satisfaction of code size requirement.
4. Present the algorithms of totally or partially removing the iterations in prologue and epilogue.

The experimental results show the effectiveness of our CRED technique in reducing code size of software-pipelined loop. For example, for the 4-stage lattice filter, the software-pipelined code size is 24. But it is significantly reduced to 7 after our CRED technique is applied. The improvement of code sizes is ranged from 57% to 84% from our experiments. We also show the experiments using our algorithms to explore the opportunities in making code size/performance trade-offs.

The rest of the paper is organized as follows: In Section 2, we introduce several basic concepts and principles used in CRED technique. In Section 3, we illustrate the applications of general CRED technique on various types of processors. Section 4 presents the code size reduction theory. Section 5 proposes CRED algorithms. Section 6 provides experimental results.

2 Basic Principles

In this section, we provide an overview of the basic principles related to our code size reduction technique. We demonstrate that retiming and software pipelining are essentially the same concept. In fact, we will use retiming to model software pipelining, and use retiming function to derive the code size of software pipelined loops. First of all, we briefly introduce the data flow graph.

2.1 Data Flow Graph

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

The dependencies within the same iteration are represented by edges without delay. The inter-iteration data dependencies are represented by weighted edges. For an edge $e(u \rightarrow v)$ with delays, $d(e)$ means the input data of node v at j^{th} iteration is generated by node u at $(j - d(e))^{\text{th}}$ iteration. A loop can be represented by a cyclic DFG. The delay count for any cycle in the cyclic DFG is

positive for a legal program. The DFG in Figure 3(a) represents the loop in Figure 1(a). Each iteration corresponds to the execution of each node of DFG for exactly once.

The *cycle period* of a DFG is defined as the computation time of the longest path without delay. The cycle period of a DFG corresponds to the minimum schedule length of one iteration when there are no resource constraint.

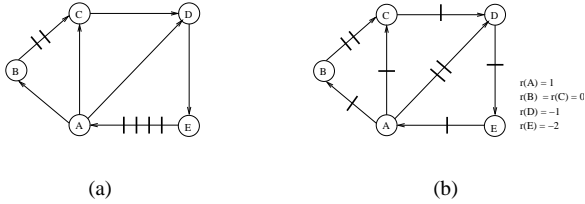


Figure 3. (a) A DFG G . (b) A retimed DFG G_R .

2.2 Retiming Technique and Retimed Graph

The *Retiming* technique has been effectively used to obtain the minimum cycle period for a DFG by evenly distributing the delays [1, 5].

On a DFG $G = (V, E, d, t)$, a retiming moves delays around in the following way: a delay is drawn from *each* of the incoming edges of v , and then it is pushed to *each* of the outgoing edges of v , or vice versa. Note that retiming preserves the data dependencies of the original DFG. Only the initial assignments need to be modified after a retiming. The modification of initial assignments is carried out by a *prologue* of statements.

A retiming function $r : V \rightarrow Z$ of a DFG G represents the number of delays moved through node v . Figure 3(b) shows the retimed DFG G_R , where two delays were pushed backward through node E . Therefore, the retiming function $r(E) = -2$.

Let $G_r = (V, E, d_r, t)$ be the DFG retimed by retiming r . A retiming is *legal* iff the delay count d_r is non-negative for any edge in E . That is, $d_r(e) = d(e) + r(u) - r(v) \geq 0$. Also for any cycle l in G , a legal retiming produces $d_r(l) = d(l)$. Since retiming preserves the number of delays in a cycle, it also preserves the iteration bound of a DFG.

We only consider the *normalized* retiming in this paper, which can be obtained by subtracting $\min_v r(v)$ from $r(v)$, $\forall v \in V$.

2.3 Retiming and Software Pipelining

Let's use the simple example in Figure 4(a) and Figure 4(b) to have a clearer picture on the effect of retiming. We can retiming the DFG by evenly distributed the delays over the cycle such that the cycle period of the retimed graph is minimized. When a delay is pushed through node A to its outgoing edge, it actually pushed the i^{th} copy of A to be executed with $(i - 1)^{\text{th}}$ copy of node B , as the dashed rectangular boxes shown in Figure 5(a). And since there's no dependency between A and B within the new iteration, these two nodes can be executed simultaneously in the schedule

of the new iteration if there's no resource constraint. The effect of this new schedule is the same as pipeline the nodes from different iterations as shown in Figure 5(b).

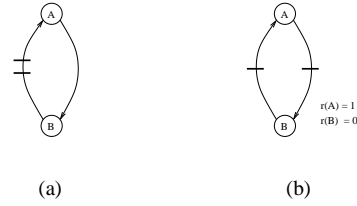


Figure 4. (a) A simple DFG. (b) The retimed DFG.

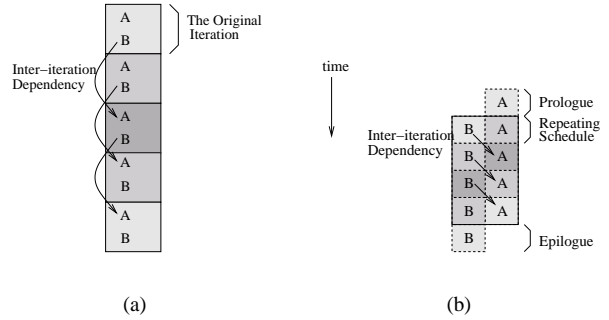


Figure 5. (a) A static schedule of original loop. (b) The pipelined loops.

In fact, each retiming corresponds to a software pipelining operation. When a delay is pushed from the incoming edges of node A to its outgoing edges, every copy of node A is shifted up by one iteration, and the first copy of A is shifted out of the first iteration into the prologue. The operations of retiming regroups computation nodes from the different iterations of the original loop body into new iterations.

With normalized retiming function, we can measure the sizes of prologue and epilogue naturally. When $r(v)$ delays are pushed forward through node v , there are $r(v)$ copies of node v appeared in the prologue. The number of copies of a node in the epilogue can also be derived in a similar way. If the maximum retiming value in the data flow graph is $\max_u r(u)$, there are $\max_u r(u) - r(v)$ copies of node v appeared in epilogue.

For a software-pipelined loop, if the nodes in a static schedule are from k different iterations, we call such k the *depth* of the loop pipeline. From the retiming point of view, if there are k different retiming values, k iterations are pipelined in the static schedule.

2.4 Rotation Scheduling

Rotation scheduling is a flexible technique for scheduling cyclic DFGs with resource constraints [1]. In each rotation phase, it implicitly applies retiming operations on a set of nodes, then these nodes are rescheduled to obtain a software-pipelined schedule.

Figure 6(a) to Figure 8(b) illustrate the rotation scheduling progress by simply rotating the first row of the schedule in each rotation phase. For example, in the first rotation phase, node A is rotated and rescheduled as shown in Figure 6(b) and Figure 6(c). The effect on the schedule is the same as pushing the first copy of node A into prologue and the last copy of the other nodes into epilogue. The resulted schedule is optimal. The 4-cycle loop is transformed into a 1-cycle loop. The pipeline depth is four. The italic letters in the schedule show how the second copy of the original loop body are merged with other copies in a new iteration.

In the process of rotation scheduling, the state of rotation can be recorded by retiming functions. For instance, after the last rotation, the retiming functions are $r(A) = 3$, $r(B) = r(C) = 2$, $r(D) = 1$ and $r(E) = 0$. That is, the node A of i^{th} copy, node B and C from $(i + 1)^{\text{th}}$ copy, node D from $(i + 2)^{\text{th}}$ copy and node C from $(i + 3)^{\text{th}}$ copy are overlapped in one iteration of the pipelined loop.

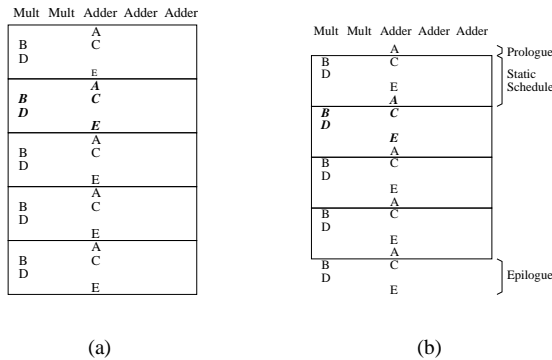


Figure 6. (a) The original loop schedule. (b) The first phase rotation. (c) Rescheduling after rotation.

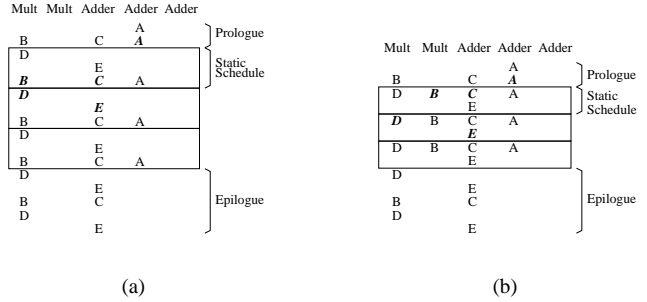


Figure 7. (a) The second phase rotation. (b) Rescheduling.

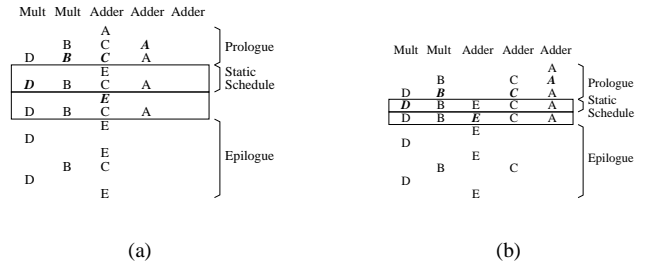


Figure 8. (a) The third phase rotation. (b) The resulted pipeline schedule.

In this paper, we use rotation scheduling to illustrate the Code-size REDuction (CRED) technique, because it clearly shows the relationship between software pipelining and retiming function, which is a key factor of this technique.

3 Application to Various Processors

Our Code-size REDuction (CRED) technique can be generally applied to various types of processors. We can classify the processors into four classes. **Processor class 1** is the processors without conditional registers, such as Motorola/Agere's StarCore. **Processor class 2** supports conditional execution with conditional registers, such as Philips' TriMedia. Each instruction can be *guarded* by a binary bit. **Processor class 3** implements conditional registers with counters such as TI's TMS320C6000. **Processor class 4** implements special hardware support for conditional execution for software pipelining, such as HP/Intel's IA64. Because of the space limit, we will only focus on the implementation for Processor class 3. The implementations for other classes are similar which are briefly explained in the end of the section.

In this paper, we use the architecture similar to TI's TMS320C6000 DSP processor to illustrate the implementation details. Conditional register and conditional execution are com-

only implemented in many architectures [3, 7, 9]. Conditional register is also called predicate register when it holds boolean values, or “guard” register. An instruction guarded by a conditional register is conditionally executed, depending on the value in the conditional register. If it is “true”, the instruction is executed. Otherwise, the instruction is disabled.

In TMS320C6000, the conditional register is implemented as a counter [2, 9]. 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. $\max_u r(u) - r(v)$. We also specify that the instruction is executed only when $0 \geq p > -LC$, and it is disabled when $p > 0$ or $p \leq -LC$, where LC represents the original loop counter. And the value of conditional register is decreased by 1 in every iteration. We use the notation of $[]$ surrounding the conditional register name to indicate the conditional execution. For example,

```

p = 2;
[p] A[i] = B[i-1] * 5;
p = p - 1;

```

indicates that the computation of A is not executed in the first and second iteration. Instead, it starts to be executed in the third iteration when the value of p is decreased down to 0. Also, the computation of A will not be executed any more when the value of p is less than $-LC$, suppose that $-LC$ is stored in a special register and the value checking of p is done by hardware. The example in Figure 9 illustrates the results after applying CRED on the program in Figure 1(b).

Figure 9(a) shows the code after removing the prologue/epilogue in Figure 1(b). The conditional registers p , q , r and s are used for different retiming values. Each of them starts a backward counting from a different initial value to control the guarded operation. Since there are four different retiming value, we need to use four registers to totally remove prologue and epilogue. A decrement instruction needs to be inserted into the loop body for each register. Suppose that the results of these decrement instructions do not affect the operations out of the loop, these decrements don't need to be guarded. Note that the loop will now be executed for $n - 3 + 3 + 3 = n + 3$ times, since it includes 3 iterations from prologue and the other 3 for epilogue. Figure 9(b) shows the execution sequence after applying CRED technique. Note that each iteration now executes the same static schedule. The numbers in brackets are the values of conditional registers. Figure 9(c) shows that the new code size is the same as kernel code size.

CRED technique can also be applied to the other types of processors. For the processors in class 1, it's obvious that we can use an *if-then* clause to control the execution of the nodes with the same retiming value [6]. Each retiming value needs a counter and a branch control. To implement CRED, class 1 processor also needs a counter and some instructions to manipulate the counter, such as comparison and decrement instructions.

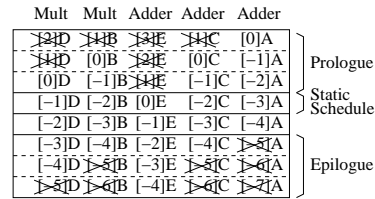
For the processors in class 2, each instruction is guarded by a predicate bit; thus the boolean assignments of predicate bits can be used to control conditional executions [7]. Thus, the performance penalty related to the branches, such as, branch misprediction, and branch delay, can be eliminated. A counter and some counter related instructions should be included to implement CRED. Processor in class 3 have been discussed previously. It uses the counter as a conditional register [9], so the number of inserted

```

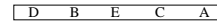
p = 0;
q = 1;
r = 2;
s = 3;
for i = 1 to n+3 do
  [p] A[i] = B[i-4] + 9;
  p = p - 1;
  [q] B[i-1] = A[i-1] * 5;
  [q] C[i-1] = A[i-1] + B[i-3];
  q = q - 1;
  [r] D[i-2] = A[i-2] * C[i-2];
  r = r - 1;
  [s] E[i-3] = D[i-3] + 30;
  s = s - 1;
end

```

(a)



(b)



(c)

Figure 9. (a) Code after totally removing prologue/epilogue. (b) The execution sequence after applying CRED. (c) The new code size.

instructions are reduced.

Processor in class 4, such as IA64, supplies special hardware support. The conditional register is a 64-bit rotating register, which is controlled by a set of special loop control instructions, such as *brtop*. These instructions are actually the hardware implementations of control logic that deals with the rotating register and the loop counter. For this kind of processor, only one instruction, such as *brtop*, needs to be insert into the loop body. Also, The instructions with the same retiming value are guarded by a one-bit predicate. The number of inserted instructions is the smallest among 4 classes of processors; however, it needs special hardware support that is not yet commonly found in most of processors [3].

4 Code Size Reduction Theorems

In Section 2, we have already demonstrated that software pipelining can be modeled by retiming functions. In this section, we formally present the theorems of code size reduction based on retiming concept. The code size reduction technique is a code transformation that attempts to remove the iterations in prologue and epilogue, so that the code size requirement, i.e., the number

of instruction words for a VLIW processor, can be satisfied. The code size reduction technique preserves the correct execution of the removed iterations in prologue/epilogue by conditional operations.

Lemma 4.1. *Given a retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop with repeating schedule S . Suppose that there is only one retiming value $r(v) = k$, where $k > 0$ for all the retimed nodes $v \in V$, then, the prologue can be correctly executed by only executing node v in schedule S for k times.*

Lemma 4.1 can be extended to a general case when multiple iterations are overlapped in the steady stage of pipeline, that is, there are multiple retiming values in the retimed DFG. The following theorem states that all the dependencies in the original code can be preserved by only executing the repeating schedule if we know the retiming value of each node.

Theorem 4.2. *Given a retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop with repeating schedule S . The prologue can be correctly executed by:*

- Executing only S .
- Executing node $u \in V$ with retiming value $r(u) = k$ for k times starting from the $(\max_u r(u) - k + 1)^{\text{th}}$ iteration, where $k \geq 0$.

For example, if $r(v) = 3$ and $\max_u r(u) = 5$, then node v will not be executed in the first and the second iterations. Instead, it will start to be executed at the third iteration.

Theorem 4.3. *Given a retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop with repeating schedule S . Let n be the number of iterations in the original loop. The epilogue can be correctly executed by:*

- Executing only S .
- Executing node $u \in V$ with retiming value $r(u) = k$ for $(\max_u r(u) - k)$ times in the last $\max_u r(u)$ iterations starting from the n^{th} iteration, where $k \geq 0$.

Theorem 4.2 and Theorem 4.3 establish the theoretical foundation for code size reduction of a software-pipelined loop. They indicate that the code size overhead in prologue or epilogue can be removed by only executing the kernel code after software pipelining, if the computation nodes in the kernel code can be conditionally executed. Figure 9(b) shows the execution sequence of the conditional executions of the repeating schedule.

With the architectural support described in Section 3, the code size reduction technique can be implemented easily with conditional registers and very few hardware support. The following theorem states the number of conditional registers needed for totally removing the prologue and epilogue.

Theorem 4.4. *(Total Code Size Reduction) Given the retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop with repeating schedule S . Let R be the number of different retiming values in G_r . Let P be the number of available conditional registers. If $P \geq R$, then all the codes in prologue/epilogue can be removed by conditionally executing S . That is, the optimal code size can be achieved.*

Theorem 4.4 defines the maximum software pipelining “degree” allowed for totally removing prologue and epilogue for a given number of conditional registers. For the example, if we have 4 conditional registers, we can remove all three iterations in prologue and epilogue. In the other words, we can produce 3 retiming values greater than zero during software pipelining. To make it clear, we define the software pipelining “degree” as the number of different retiming values produced in the software pipelining. Since the number of different retiming values equals to the pipeline depth, these three terms actually state the same concept from different views.

Since the retiming values in a data flow graph may not be continuous integer sequence, we have the number of different retiming values $R \leq \max_u r(u)$. That is, the number of conditional register P needed is no more than the number of iterations to be removed. So, we use less conditional registers than kernel only method, which use one register for each iteration [8]. Also we use less conditional registers than code collapsing method, which needs two conditional registers for the same computation node, one for removing the copy in prologue, the other for epilogue [2].

Since the resource of conditional registers is very limited in DSP processors, we may not have enough conditional registers to remove all the iterations in prologue/epilogue. The following theorem states the partial code size reduction technique. For example, we have 3 different retiming values $\{0, 3, 4\}$ in a software-pipelined loop, while only 2 conditional registers. Originally, prologue and epilogue each has 4 iterations, since the maximum retiming value is 4. By using partial code size reduction, the nodes in the innermost 3 iterations in prologue and epilogue with retiming value 3 can be removed.

Theorem 4.5. *(Partial Code Size Reduction) Given the retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop with repeating schedule S . Let R be the number of different retiming values in G_r , and r_P the P^{th} smallest retiming value, where $P < R$ is the number of available conditional registers. Then, the innermost r_P iterations of prologue and epilogue can be removed by conditionally executing S .*

Suppose that there is a node u in these r_P iterations with retiming value $r(u) > r_P$, we can use the same conditional register for the nodes with retiming value r_P . The initial values of the conditional registers are set as $r_P - r(v)$ for node v with $r(v) \leq r_P$. Figure 10(a) shows the code after removing part of the iterations in prologue/epilogue. Figure 10(b) shows the execution sequence. Figure 10(c) shows the new code size with only the first iteration remains in prologue/epilogue, which usually has a very small number of computation nodes.

Because of the limited number of predicate registers, it’s necessary to control the software pipelining “degree” when the code size requirement cannot be satisfied for some deeply pipelined loops. A commonly applied solution for this situation is going back to suppress software pipelining, or simply using another version of the code without pipelining [2, 8]. With CRED-Partial we can avoid the cost of re-doing software pipelining or the brute force solution. As a matter of fact, there is a gray area to be explored between two extreme solutions, that is, the “perfect” software pipelining with the largest prologue/epilogue and no pipelining with the smallest code size. This exploration benefits the compiler in making code

crement instructions, so the computation nodes will be disabled when the value of conditional register is greater than 0.

Comparing the effect of prologue/epilogue only CRED technique and that of Code Collapsing technique [2], it's interesting to see that the code size reduced by these two techniques are equal. That is, code collapsing becomes a special case of our CRED technique. Because our technique is based on the fundamental understanding of retiming and code size expansion, it can be generally applied to reduce the code size of any software-pipelined loops.

6 Experiment Results

We have experimented with CRED algorithms on several well-known benchmarks. In most cases, we can use less than three or fewer conditional registers to completely remove all the iterations in prologue and epilogue without incurring performance penalty. The code size improvements are very promising.

Table 1 shows the experimental results of CRED-Total. All the schedules are generated on a simulated architecture with 2 multipliers and 3 adders with conditional registers described in Section 3, assuming that the computation time of each functional unit is one time unit. The data in the second column are the numbers of conditional registers used to remove all the iterations in prologue and epilogue, i.e., the number of different retiming values. The third and fourth columns are the code size of "perfectly" software pipelined schedules and the code size after total code size reduction. Here the code size is measured as the number of instruction words. We can see the remarkable code size improvement percentages in the fourth column. The last two columns show the schedule lengths before and after applying CRED-Total. In most cases, except for the all-pole lattice filter, the schedule lengths are the same as pipelined schedule length. The pipelined schedule of all-pole lattice filter has four different retiming values, which means four decrement instructions need to be inserted. Its schedule length is increased by one cycle because its schedule is already very compact. On the other hand, the improvement on the all-pole lattice filter's code size is the largest, since all three iterations in prologue/epilogue are reduced. If the code size requirement of this application can be relaxed to larger than six instruction words, both CRED-Partial and Prologue/Epilogue Only CRED can be applied to achieve the code size target while maintain the schedule length of the tightest pipelined.

Benchmarks	Reg #	Code Size			Sch. Len.	
		SP	CRED	%	SP	CRED
IIR Filter	2	6	2	66.7	2	2
Differential Eq.	3	14	3	78.6	3	3
All-pole Filter	4	39	6	84.6	5	6
Elliptic Filter	2	26	11	57.7	11	11
4-stage Filter	3	24	7	70.8	7	7
Voltera Filter	2	22	9	59.1	9	9

Table 1. The results of CRED-Total.

Table 2 shows the trade-offs between code size and performance for all-pole lattice filter by given two conditional registers. We use CRED-Partial to remove the innermost two iterations of

the pipelined loop. The first column shows the software pipeline depth, which is also the software pipeline degree. The second and third columns show the code sizes of software pipelined loop and that after applying CRED-Partial. The fourth column shows the improvement percentage on code size, which is limited by the number of available conditional registers. The last two columns show the schedule lengths before and after applying CRED. In most cases, the schedule lengths produced by CRED are the same as tightest pipelined ones. We can see that when the pipeline depth increases, the code size is increased and the schedule length is decreased. The experimental results show the "gray" area existing between the tightest software pipelining with a large code size and the shallow pipeline with a relatively small code size. By using CRED technique, a compiler can efficiently explore the "gray" area to make code size/performance trade-offs

Pipeline Depth	Code Size			Sch. Len.	
	SP	CRED	%	SP	CRED
2	23	11	52.2	11	11
3	30	19	36.7	7	7
4	39	29	25.6	5	6

Table 2. Experimental Results for All-pole Lattice Filter with 2 conditional registers.

References

- [1] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229–239, Mar. 1997.
- [2] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiaciak. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proceedings of the 3rd Workshop on Media and Streaming Processors in conjunction with 34th Annual International Symposium on Microarchitecture*, pages 29–38. ACM, Dec. 2001.
- [3] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, Dec. 2001.
- [4] 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.
- [5] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, Aug. 1991.
- [6] Motorola Digital DNA & Agere Systems. *StarCore SC140 DSP Core Reference Manual*, Nov. 2001.
- [7] Philips, Inc. *TM-1300 Media Processor Data Book*, May 2000.
- [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.