

CRED: Code Size Reduction Technique and Implementation for Software-Pipelined Applications *

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

Chantana Chantrapornchai
ctana@su.ac.th
Department of Mathematics,
Silpakorn University
Nakorn Pathom, Thailand 73000

Abstract

Software pipelining technique is extensively used to explore the instruction level parallelism in loops. However, this performance optimization technique results in code size expansion. For embedded systems with very limited memory resources, the code size becomes one of the most important optimization concerns. In this paper, we propose the theoretical foundation for code size reduction based on the relationship between retiming function and code size expansion. A general Code-size REDuction technique (CRED) for software-pipelined loops are presented on various kind of processors. Our algorithms integrate the code size reduction procedure with scheduling to produce optimal code size for a target schedule length. The experiments on a set of well-known benchmarks show the effectiveness of this technique on both code size reduction and code size exploration.

Keywords: Retiming, DSP processors, Software pipelining, Scheduling

1 Introduction

Software pipelining is extensively used to exploit instruction level parallelism in loops [4, 8]. Although this performance optimization technique helps to achieve a compact schedule, it expands the total code size by introducing prologue and epilogue sections, i.e., the codes executed before entering and after leaving the new loop body. Furthermore, the size of prologue and epilogue grows proportionally as more iterations of the loop get overlapped in the pipeline [8]. For embedded processors with very limited on-chip memory resources, the code size expansion becomes a major concern. Consequently, making trade-off between code size and performance for software-pipelined applications becomes an important task for compilers targeting at embedded systems.

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

A simple *for* loop and its code after applying software pipelining are shown in Figure 1(a) and Figure 1(b). The loop schedule length is reduced from four control steps to one control step for software-pipelined loop. However, the code size of software-pipelined loop is three times larger than the original code size.

```
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[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)

(b)

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

Some ad-hoc code size control techniques were used to reduce the prologue/epilogue produced by software pipelining. For example, code collapsing technique is developed for TI's TMS320C6000 processors [2]. However, the effectiveness of their techniques cannot be guaranteed and quite limited. Kernel-only code generation schema presented in [8] is specially applied to IA64 [3]. However, it requires special architectural support which is not found in DSP processors. There is no theoretical framework presented in literature for a code size reduction of software-pipelined loops.

In our research work, we study the underlying relationship between *retiming* and software pipelining, and show that the

size of code expansion is closely related to the retiming function. Based on this understanding, we present a Code-size Reduction (CRED) technique, which can be generally applied to different kinds of processors with or without conditional registers. 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. We classify the processors into four classes. **Processor class 1** is the processors without conditional registers, such as Motorola/Agere’s StarCore [6]. **Processor class 2** supports conditional execution with predicate registers, such as Philips’ TriMedia [7]. Each instruction can be *guarded* by a binary bit. **Processor class 3** implements conditional registers with counters such as TI’s TMS320C6000 [9]. **Processor class 4** implements special hardware support for conditionally executing software-pipelined loops, such as HP/Intel’s IA64 [3]. Our CRED technique can be applied to all these four classes of processors and achieve minimal code size.

Our contributions are:

1. Establish the theoretical foundation of code size reduction technique for software-pipelined loops based on retiming concept.
2. Design the CRED technique for achieving minimal code size for software-pipelined applications.
3. Show that CRED technique is general enough to be applied to any type of processors.
4. CRED reaches the minimum code size for processor class 3 and 4.
5. Explore the code size/performance trade-off space to generate best schedule length for a given code size requirement.

Our experimental results show the effectiveness of our techniques in reducing code size of a software-pipelined loop. For example, for the differential equation, the software-pipelined code size is 33. But it is significantly reduced to 17 after our CRED technique is applied. The improvement of code sizes is ranged from 25% to 61% for our experiments on processor class 3 (modified TMS320 architecture). We also show the experiments to explore the opportunities in making code size/performance trade-offs by using our algorithm.

The rest of the paper is organized as follows: In Section 2, we introduce necessary backgrounds related to CRED technique. In Section 3, we present the application of CRED technique on various processors. Section 4 presents the CRED theories. Section 5 provides CRED algorithms. Section 6 presents the experimental results. The last section, Section 7, concludes the paper.

2 Basic Principles

In this section, we give an overview of basic concepts and principles related to CRED technique. These include data flow graphs, retiming, software pipelining and rotation scheduling.

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 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, which is 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. The dependencies within the same iteration are represented by edges without delay ($d(e) = 0$). A static schedule must obey these intra-iteration dependencies. Programs with loops can be represented by cyclic DFGs as shown in Figure 2(a). The *cycle period* of a DFG is defined as the computation time of the longest path without delay in the graph, which corresponds to the minimum schedule length when there is no resource constraint. We assume the computation time of a node is 1 time unit in this paper.

2.2 Retiming and Software Pipelining

The retiming technique [5] can be applied on a data flow graph to improve the cycle period by evenly distributing the delays in the graph. The delays are moved around in the graph in the following way: a delay is drawn from *each* of the incoming edges of v , and then added to *each* of the outgoing edges of v , or vice versa. Note that the retiming technique preserves data dependencies of the original DFG.

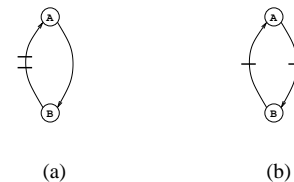


Figure 2: (a) A simple DFG. (b) The retimed DFG with $r(A) = 1$ and $r(B) = 0$.

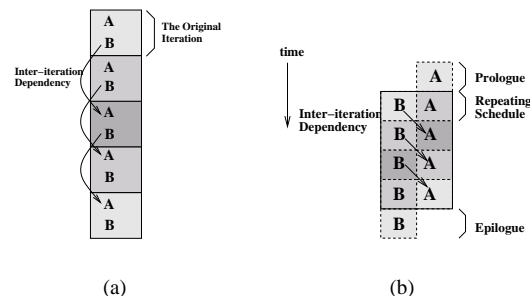


Figure 3: (a) A static schedule of original loop. (b) The pipelined loops.

The retiming function $r : V \rightarrow Z$ is the number of delays moved through node $v \in V$. Figure 2(b) shows the retimed DFG

of Figure 2(a) with retiming functions $r(A) = 1$, $r(B) = 0$. We use the *normalized* retiming function in computing the expanded code size, which simply subtracts $\min_v r(v)$ from $r(v)$ for every node v in V .

Consider a retimed DFG $G_r = (V, E, d_r, t)$ computed by retiming r . The number of delays of any edge $e(u \rightarrow v)$ after retiming can be computed as $d_r(e) = d(e) + r(u) - r(v)$. For any legal retiming r , we have $d_r(e) \geq 0$ for every edge, and the total number of delays remains constant for any cycle in the graph.

When a delay is pushed through node A to its outgoing edge as shown in Figure 2(b), 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 $(i - 1)^{\text{th}}$ copy of node B. The schedule length of the new loop body is then reduced from two control steps to one control steps. This transformation is illustrated in Figure 3(a) and Figure 3(b).

In fact, 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 moved up by one iteration, and the first copy of the node is shifted out of the first iteration into the prologue. With retiming function r , we can measure the size of prologue and epilogue. 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 the epilogue.

2.3 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. The effect of this retiming on the static schedule is that the nodes are moved to a different iteration. The schedule of the program in Figure 1(b) generated by rotation scheduling is shown in Figure 4.

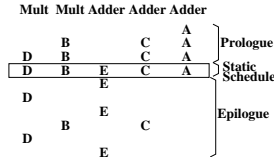


Figure 4: The schedule of the program in Figure 1(b).

3 Application of CRED to Various Processors

In this section, we show that the CRED technique can be applied to any types of processors for software-pipelined applications with code size constraints. We use the 4 classes of processors to illustrate that the applicability of this technique is independent of architectures. We also show that the processors of class 3 and 4 can achieve better code size reduction results than those of class 1 and 2. While processor class 4 obtains the smallest code

size, it depends on the special hardware support for conditional execution of software-pipelined loops as in IA64, which is not yet found in DSP processors. Processor class 3 is practical for DSP processors, because it's based on a similar architecture with TI's TMS320C6000, and it also achieves minimal code size for this kind of architecture.

CRED technique use the retiming function to control the execution order of the computation nodes in a software-pipelined loop. The relative values are stored in a counter to set the "life-time" of the nodes with the same retiming value. For node v with retiming value $r(v)$, its counter is set as the maximum retiming value minus the retiming value of node v , i.e. $p = \max_u r(u) - r(v)$. We specify that the instruction is executed only when $0 \geq p > -LC$. The value of p is decreased by 1 in every iteration. Therefore, the instruction can not be executed when $p > 0$, instead, it is delayed until the value of p is decremented down to 0, then it starts to be executed. On the other hand, this instruction will be disabled once the value of p is less than or equal to the negative loop counter of the original loop.

Processor Class 1

For the processors in class 1, the conditional execution defined by CRED can be directly translated to *if-then* clauses. Each retiming value needs a counter and a branch control. To implement CRED, class 1 processor also needs some instructions to manipulate the counter, such as comparison and decrement instructions. The following code gives the code after applying CRED for the example shown in Figure 1(b):

```

p1 = 0; p2 = 1; p3 = 2; p4 = 3;
for i = 1 to n+3 do
  if p1 <= 0 and p1 > -n then
    A[i] = E[i-4] + 9;
  if p2 <= 0 and p2 > -n then
    B[i-1] = A[i-1] * 5;
    C[i-1] = A[i-1] + B[i-3];
  if p3 <= 0 and p3 > -n then
    D[i-2] = A[i-2] * C[i-2];
  if p4 <= 0 and p4 > -n then
    E[i-3] = D[i-3] + 30;
  p1--; p2--; p3--; p4--;

```

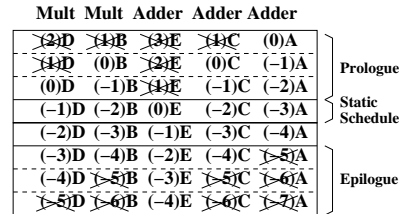


Figure 5: The execution sequence after CRED.

By doing so, the instruction computing A starts execution from the 1st iteration and stops at the n^{th} iteration, while the instructions computing B and C start execution from the 2nd iteration and stops at $(n + 1)^{\text{th}}$ iteration, and so on. Figure 5 shows the execution sequence of the conditional operations in our implementation.

Processor Class 2

For the processors in class 2 that support conditional executions with predicate registers, the *if-then* control branch can be removed. Instead, the computation nodes with the same retiming

values are guarded by one predicate. The boolean assignments of the predicate control the execution of these nodes. This mechanism is called “guarding” [7]. By using the predicate registers, the performance penalty related to the branches, such as, branch mis-prediction, and branch delay, can be eliminated. A part of the loop body after CRED is shown in Figure 6.

```

r2 = 1;
.....
for i = 1 to n+3 do
.....
  \* set predicate register p2 *\
  p2 <-- (r2 <= 0);
  p <-- (r2 > -LC);
  p2 <-- p2 & p;
  (p2) B[i-1] = A[i-1] * 5;
  (p2) C[i-1] = A[i-1] + B[i-3];
  r2 = r2 - 1;
.....
end

```

Figure 6: CRED on TriMedia.

Processor Class 3

Processor in class 3 implements the conditional registers with the functionality of counters. The representative processor in this class is TI’s DSP processor TMS320C6000.

Figure 7 shows a portion of the new code for the program in Figure 1(b). A prefix (p1) means the guarded instruction is executed when p1 is non-zero, while a prefix (!p1) indicates the instruction can only be executed when p1 is zero [2, 9].

```

r2 = LC + 1;
p2 = 1;
.....
for i = 1 to n+3 do
.....
  (!p2) B[i-1] = A[i-1] * 5;
  (!p2) C[i-1] = A[i-1] + B[i-3];
  (p2) p2 = p2 - 1;
  r2 = r2 - 1;
  (!p2) p2 <-- (r2 < 0);
.....
end

```

Figure 7: CRED on TMS320C6000.

We propose an architecture similar to TMS320 to further reduce the inserted instructions for implementing CRED. A new instruction is proposed 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, and instruct the hardware to disable the guarded instruction when $p1 > 0$ or $p1 \leq -LC$. Figure 8 shows the code after applying CRED on the modified TMS320 processor.

Processor Class 4

Processor in class 4, such as IA64, supplies special hardware support for conditional execution of software-pipelined loops [3, 8]. The conditional register is implemented as a 64-bit rotating register, where each bit is a predicate. The rotating register 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

```

p1 <-- 0; // setp p1 = 0 : -n
p2 <-- 1;
p3 <-- 2;
p4 <-- 3;
for i = 1 to n+3 do
  (p1) A[i] = E[i-4] + 9;
  p1 = p1 - 1;
  (p2) B[i-1] = A[i-1] * 5;
  (p2) C[i-1] = A[i-1] + B[i-3];
  p2 = p2 - 1;
  (p3) D[i-2] = A[i-2] * C[i-2];
  p3 = p3 - 1;
  (p4) E[i-3] = D[i-3] + 30;
  p4 = p4 - 1;
end

```

Figure 8: Code after totally removing prologue/epilogue.

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 processors. Figure 9 illustrates a portion of code after we remove prologue and epilogue.

```

for i = 1 to n+3 do
.....
  (p2) B[i-1] = A[i-1] * 5;
  (p2) C[i-1] = A[i-1] + B[i-3];
.....
  brtop;
end

```

Figure 9: CRED on IA64.

4 Code Size Reduction Theorems

In this section, we present the theoretical foundation of CRED technique based on the retiming concept. It is a code transformation that attempts to remove the code in prologue and epilogue, so that the total code size requirement can be satisfied. The theorems show the correctness of this code transformation.

Theorem 4.1. *Let $G_r = \langle V, E, d_r, t \rangle$ be the retimed data flow graph of a loop with retiming function r . The prologue can be correctly executed by:*

1. Executing only the repeated loop body and
2. Executing node u whose $r(u) = k$ for k times starting from the $(\max_u r(u) - k + 1)$ -th iteration, $\forall u \in V$ and $k \geq 0$.

For example, if $r(v) = 3$ and $\max_u r(u) = 5$, then node v will be disabled in the first and the second iterations, and start to be executed in the third iteration. Similar situation can be applied to the epilogue, except that the loop body needs to be executed for $(\max_u r(u) - k)$ times in the last $\max_u r(u)$ iterations.

Theorem 4.1 indicates that the code in prologue or epilogue can be removed by conditionally executing the schedule of loop body. Due to the limited space, we omit the proof. Based on the retiming value, we can decide the conditional registers required for achieving the minimal code size as in the following theorem.

Theorem 4.2. (CRED-Total) Let P be the number of available conditional registers, and R the number of different retiming values. If $P > R$, then all the codes in prologue/epilogue can be removed.

Code size reduction technique can also be applied to remove a part of prologue and epilogue when there are insufficient conditional registers. For example, suppose we have 3 different retiming values $\{0, 3, 4\}$. Originally, prologue and epilogue each contains codes of 4 iterations, since the maximum retiming value is 4. In the following theorem, we show that the innermost 3 iterations can be safely removed from both prologue and epilogue with only 2 conditional registers. That is, the nodes with retiming values 0 and 3 can be removed from prologue and epilogue.

Theorem 4.3. (CRED-Partial) Let P be the number of available conditional registers. Let R be the number of different retiming values, and r_p be the p^{th} smallest retiming value. If $P < R$, then the innermost r_p iterations can be safely removed in both prologue and epilogue.

For node u whose whose retiming value $r(u) > r_p$. We can use the conditional register for the nodes with retiming value r_p to guard node u . For the pipelined schedule shown in Figure 4, if we have only 3 available conditional registers, the last two iterations performed in the prologue and the first two iterations performed in the epilogue can be removed. Since the largest retiming value of the nodes whose $r(v) \leq r_p$ is $r(B) = r(C) = 2$, the initial value of the conditional register is set to $2 - r(v)$. Figure 10(a) shows the code of the loop after reducing part of the iterations performed in prologue and epilogue. After this reduction, only the first iteration of node A remains in the prologue and the last iteration of node E in the epilogue. Figure 10(b) shows the reduced schedule.

```

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

```

(a)

					A	} Prologue
D	B	E	C	A	A	
		E				} Epilogue

(b)

Figure 10: (a) The code after reducing part of prologue and epilogue. (b) Reduced schedule.

From the previous theorems we can see that the code size of prologue/epilogue is proportional to software pipelining “degree”, which is also the number of different retiming values. Next, we give the theorems for computing the expanded code size after software pipelining.

Theorem 4.4. Given the retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop Q . Let $\max_u r(u)$ be the maximum retiming value for all the nodes $u \in V$. The number of instructions in Q is $\mathcal{N} = R * |V|$.

We have shown the applications of CRED-Total in Section 3. The following theorem concludes the computation of the code size after applying CRED-Total on different classes of processor. For processor class 3, we use the modified TMS320 architecture.

Theorem 4.5. Given the retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop Q . Let R be the number of different retiming values in G_r . Then, the number of instructions in Q after applying CRED-total is

1. processor class 1 is $\mathcal{N}_{cred} = R * 6 + |V|$;
2. processor class 2 is $\mathcal{N}_{cred} = R * 5 + |V|$;
3. processor class 3 is $\mathcal{N}_{cred} = R * 2 + |V|$;
4. processor class 4 is $\mathcal{N}_{cred} = |V| + 1$.

5 Code Size Reduction Algorithms

Our CRED algorithms are integrated with rotation scheduling to become a code size-aware software pipelining procedure. The advantage of integrating code size reduction with software pipelining is to achieve the required code size with the least possible schedule length increment.

5.1 Total Code Size Reduction Algorithm

Algorithm 5.1 CRED-Total Procedure for Modified TMS320 Architecture

Input: Initial schedule S , DFG $G = \langle V, E, d, t \rangle$.

Output: New schedule S_{opt} , the number of conditional registers used j and the new loop counter LC .

```

j ← 1;
for all i = 0, ..., S.length do
  /* Step 1: Rotate nodes. */
  Q ← First_Row(S);
  S_opt ← ReSchedule(S, Q);
  /* Step 2: Guard the nodes v with r(v) = 0. */
  p_0 ← max_u r(u), ∀ u ∈ V;
  Insert the decrement instruction of p_0;
  /* Step 3: Guard the nodes with new retiming values. */
  if there's a new retiming value r(v) then
    p_j ← max_u r(u) - r(v);
    Insert the decrement instruction of p_j;
    j ← j + 1;
    LC ← LC + 2;
  end if
  /* Step 4: Update the inserted nodes */
  Update the decrement instructions in S_opt;
end for
return S_opt, j, LC;

```

Algorithm 5.1 is used to totally remove the prologue and epilogue, assuming there are sufficient conditional registers. Note that the inserted decrement instructions can also be rescheduled by rotation scheduling process to reduce the total schedule length.

5.2 Partial Code Size Reduction Algorithm

According to Theorem 4.3, CRED-Partial algorithm can be obtained by some modifications of Algorithm 5.1. Two more inputs will be added, i.e., the number of available conditional registers CR and the code memory requirement W_{req} . Besides the new schedule S_{opt} , the output will also report the number of used conditional registers CR_{use} , the new code memory W_{new} and the maximum retiming value $\max_u r(u)$. In step 2, the if statement will check if there are available registers, $CR_{use} < CR$. If so, CR_{use} is increased by one in this step. If not, we do the following.

else

Guard all the nodes v whose $r(v) > p_j$
with conditional register p_j ;

We also add Step 5 to check code memory requirement:

if $W_{new} \geq W_{req}$
 $S_{opt} = S$;
Exit the loop;

CRED-Partial algorithm produces the shortest schedule satisfying the code memory requirement in terms of the number of instruction words, and produces the maximum retiming value. Thus, the software pipelining degree can be controlled by compiler when the code memory is limited. In the traditional approach, when the resulting code size cannot be fit in the memory, the compiler may give up the software pipelining [2], and use an unoptimized version of the code. By using our approach, the cost of redoing software pipelining can be reduced. The CRED-Partial can also be used effectively to explore the trade-off space between code size and software pipeline depth, since the pipeline depth and the code memory can be easily computed in each step of rotation scheduling.

6 Experimental Results

We have experimented the CRED algorithms on a set of well-known benchmarks with various processors. In most cases, we can use only three or fewer conditional registers to completely remove all the iterations performed in prologue and epilogue without incurring performance penalty in most cases. The code size is measured as the number of nodes in a schedule including prologue and epilogue parts. The schedules are generated on a simulated architecture with 3 adders and 2 multipliers, assuming the computation time of each functional unit is one time unit. The experiments show the promising results for code size improvements.

Table 1 displays the code size results by using Algorithm 5.1 on modified TMS320 architecture. The data in the second column show the numbers of conditional registers used to remove all the iterations performed in prologue and epilogue, which is equivalent to the number of distinct retiming values. The third column displays the code size of software pipelined schedules including prologue and epilogue. The fourth column displays the code size after performing the code size reduction. We can see that the remarkable code size reduction (shown in column %) is achieved. The last two columns show the schedule lengths of the loop body before and after applying code

size reduction. In most cases, the schedule lengths are the same as the original pipelined schedule lengths.

Benchmarks	Reg #	Code Size			Sch. Len.	
		SP	CRED	%	SP	CRED
IIR Filter	2	16	12	25.0	2	2
Diff. Eq.	3	33	17	48.5	3	3
All-pole Filter	4	60	23	61.7	5	6
Elliptic Filter	2	68	38	44.1	11	11
4-stage Filter	3	78	32	59.0	7	7
Voltera Filter	2	54	31	42.6	9	9

Table 1: The results of CRED-Total on modified TMS320.

Table 2 shows the resulted code sizes after applying CRED-Total technique on four different types of processors, processor class 1 to 4, starting from column 4 in that order. The second column shows the code size of original loops, and the third column shows the code size of the expanded code after software pipelining. For the percentages of reduced code size shown in “%” columns, most of them, except for the first cell for processor class 1 and 2, show the impressive improvement on the code size of pipelined loops. The negative percentage only happens on processor class 1 and 2 for the applications with very few nodes in the original loops, because the number of inserted instructions are over-numbered the number of reduced instructions. According to Theorem 4.4 and Theorem 4.5, the code size can be reduced when $\mathcal{N} < \mathcal{N}_{cred}$. For the extra small applications, processor class 3 or 4 are more effective for code size reduction than processor class 1 and 2. The experiments also shows that the best code size can be achieved in IA64 with special hardware support and loop control instructions. However, this kind of special hardware support is not found in DSP processors. For DSP processors without special architectural features as in IA64, the modified TMS320 architecture can achieve the best code size. The experimental results show that the CRED technique can be generally applied to various kinds of processors to reduce the code size. The last row of the table shows the average code size improvement on four classes of processors.

Given the number of available conditional registers, we can explore the trade-off between code size and schedule length. Table 3 illustrates several design choices in code size/performance trade-off space for the all-pole lattice filter. We use the CRED-Partial algorithm in Section 5.2 to remove the innermost two iterations of the pipelined loop. The software-pipelined schedule length(sch len), the code size(opt) and the reduced code size(CRED-P) produced on the modified TMS320 architecture are shown along with the software pipeline depth. For example, for a pipelined depth of 2, we get a schedule length of 11 control steps, and code size of 23 instruction words, which can be reduced to 11 with 2 conditional registers. When the pipeline depth increases, the code size is increased and the schedule length is decreased. We can see that CRED technique helps generate a compact schedule with the much smaller code size than the original code size.

Benchmarks	Orig	SP	4 Types of Processors							
			w/o CR (StarCore)		CR (TriMedia)		CR-CNT (TMS)		h/w (IA64)	
			code size	%	code size	%	code size	%	code size	%
IIR Filter	8	16	20	-25.0	18	-13.0	12	25.0	9	43.8
Diff. Eq.	11	33	29	12.1	26	21.2	17	48.5	12	63.6
All-pole Filter	15	60	39	35	35	41.7	23	61.7	16	73.3
Elliptic Filter	34	68	46	32.4	44	35.3	38	44.1	35	48.5
4-stage Filter	26	78	44	43.6	41	47.3	32	60.0	27	65.4
Voltera Filter	27	54	39	27.8	37	31.5	31	42.6	28	48.1
Ave. Improv.				21.0		27.3		39.5		57.1

Table 2: The results of CRED-Total on various types of processors.

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

Table 3: Code size exploration for all-pole lattice filter using 2 conditional registers.

7 Conclusion

Software pipelining is a commonly used optimization technique for exploiting instruction-level parallelism and achieving performance gain in DSP systems. However, this performance optimization technique expands the code size, which is a major concern for embedded system with very limited on-chip memory size. In this paper, we build the theoretical foundation for a general code size reduction technique, CRED, based on the understanding of the relationship among retiming, software pipelining and code size expansion. It can easily be integrated in a compiler optimization scheme for the architectures with or without conditional registers. We propose the implementation of the technique on various processors. The algorithms to achieve an optimized schedule while controlling the code size are presented. Our technique can also be used to explore the trade-off space between code size and performance efficiently. Experimental results show that CRED technique can be effectively implemented on various type of architectures while maintaining an optimized performance.

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. Zbiciak. 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.