Code Size Reduction Technique and Implementation for Software-Pipelined DSP Applications

QINGFENG ZHUGE, BIN XIAO, and EDWIN H.-M. SHA
University of Texas at Dallas

Software pipelining technique is extensively used to exploit instruction-level parallelism of loops, but also significantly expands the code size. For embedded systems with very limited on-chip memory resources, code size becomes one of the most important optimization concerns. This paper presents the theoretical foundation of code size reduction for software-pipelined loops based on retiming concept. We propose a general Code-size REDuction technique (CRED) for various kinds of processors. Our CRED algorithms integrate the code size reduction with software pipelining. The experimental results show the effectiveness of the CRED technique on both code size reduction and code size/performance trade-off space exploration.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Signal processing systems*; D.3.4 [**Programming Languages**]: Processors—*Code generation*; *compilers*; *optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Retiming, DSP processors, software pipelining, scheduling

1. INTRODUCTION

Software pipelining is extensively used to exploit instruction level parallelism in loops [Chao and Sha 1995, 1997; Chao et al. 1997; Hennessy and Patterson 1995; Huff 1993; Kuck et al. 1981; Lam 1988; Rau 1994; Rau and Glaeser 1981; Rau and Fisher 1993; Rau et al. 1992]. Although this performance optimization technique helps to achieve a compact schedule, it expands the total code size by introducing prologue and epilogue sections, that is, 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 [Rau et al. 1992]. 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

This work is partly supported by TI University program, NSF EIA-0103709, Texas ARP 009741-0028-2001.
Author's address: Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083; email: {qfzhuge,bxiao,edsha}@utdallas.edu.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2003 ACM 1539-9087/03/1100-0590 $5.00

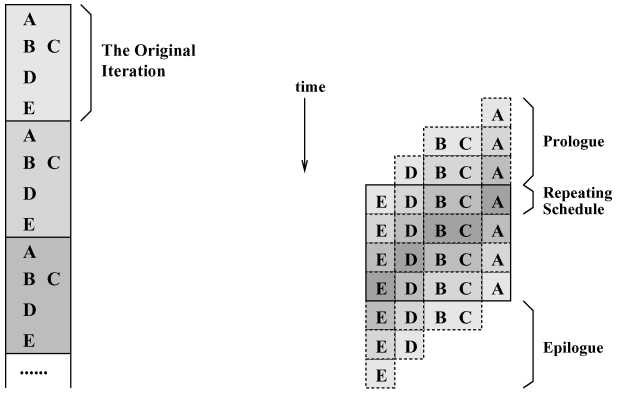
```
for i = 1 to n do
  A[i] = E[i-4] + 9;
  B[i] = A[i] * 5;
  C[i] = A[i] + B[i-2];
  D[i] = A[i] * C[i];
  E[i] = D[i] + 30;
end
```

(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)

Fig. 1. (a) The original loop. (b) The loop after applying software pipelining.



(a) (b)

Fig. 2. (a) Execution record of the original loop. (b) Execution record of the software-pipelined loop.

for software-pipelined applications becomes an important task for compilers targeting at embedded systems [Araujo et al. 1995; Lanneer et al. 1995; Texas Instruments, Inc. 2001a, 2001b].

A simple *for* loop and its code after applying software pipelining are shown in Figures 1(a) and 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. Figures 2(a) and 2(b) show the execution records of the original loop and the software-pipelined loop, respectively. In this paper, code size is defined as the number of basic instructions of the compiled code.

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 [Granston et al. 2001]. However, the effectiveness of their techniques cannot be guaranteed and quite limited. Kernel-only code generation schema presented in Rau et al. [1992] can only be applied to IA64 [Intel Corporation 2001]. It requires special architectural support that is not found in DSP processors. There is no theoretical framework presented in literature for the 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. As a result, the code size of a software-pipelined loop can be controlled by using only the retiming function. Based on this understanding, we present a Code-size REDuction (CRED) technique that attempts to remove the code in prologue and epilogue by conditionally executing the loop body. This code transformation technique can be generally applied to various 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 five classes. *Processor class 0* is the processors that do not have conditional register and do not fully support conditional execution with predication, such as Motorola/Agere's StarCore [Motorola Digital DNA & Agere Systems 2001]. *Processor class 1* is the processors that do not have conditional register but support conditional execution with predication by using "condition code" bits in the instruction, such as Intel's StrongARM [Seal 2000]. *Processor class 2* supports conditional execution with predicate registers, such as Philips' TriMedia [Philips Inc. 2000]. Each instruction can be guarded by a predicate. *Processor class 3* implements conditional registers with counters as in TI's TMS320C6000 processors [Texas Instruments Inc. 2000]. *Processor class 4* implements specialized hardware support for conditionally executing software-pipelined loops as in IA64 [Intel Corporation 2001]. Our CRED technique can be applied to all these five classes of processors and significantly reduce the code size.

Our contributions are

(1) Establish the theoretical foundation of code size reduction for software-pipelined loops based on retiming concept (Sections 2 and 4),
(2) Design the CRED technique for achieving the minimal code size of software-pipelined applications (Sections 2 and 5),
(3) Show that the CRED technique is general enough to be applied to any type of processors (Section 3),
(4) Obtain good code size reduction with accurate count of resulting code sizes for each class of processors (Theorem 4.6), and
(5) Explore the code size/performance trade-off space to generate the best schedule length for a given code size requirement. (Sections 5.2, 5.3, and 6).

Our experimental results show the effectiveness of our techniques in reducing the code size of a software-pipelined loop. For example, the software-pipelined

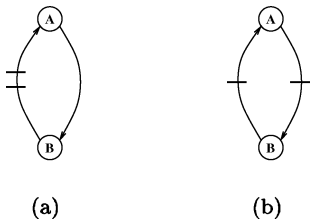


Fig. 3. (a) A simple DFG. (b) The retimed DFG with $r(A) = 1$ and $r(B) = 0$.

code size of elliptic filter is 68. But it is significantly reduced to 38 after our CRED technique is applied. The improvement of code sizes is ranged from 25.0% to 61.7% for our benchmarks experimented on processor class 3 (modified TMS320 processor). We also conduct the experiments to explore the opportunities in making code size/performance trade-off by using our algorithm. Our code size reduction technique can be easily combined with some optimization techniques considering memory constraints and data prefetching, such as those in Wang et al. [2001], and Chen et al. [1998, 2000].

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 theoretical foundation of code size reduction for software pipelined loops. 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 provide an overview of the basic principles related to our code size reduction technique. These include data flow graph, retiming, software pipelining, and rotation scheduling. We demonstrate that retiming and software pipelining are essentially the same concept. 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 a set of computation nodes, $E \subseteq V \times V$ is a set of edges, d is a function from E to a set of nonnegative integers, representing the number of delays between any two nodes, and t is a function from V to a set of positive integers, representing the computation time of each node.

Programs with loops can be represented by cyclic DFGs as shown in Figure 3(a). An *iteration* is the execution of each node in V exactly once. Iterations are identified by an index i starting from 1. Interiteration dependencies are represented by edges with delays, which is indicated by the edges with bar lines in the graph. In particular, an edge $e(u \rightarrow v)$ with delay count $d(e) > 0$ means that the computation of node v at jth iteration requires data produced by node u at $(j - d(e))$th iteration. The dependencies within the same iteration are represented by edges without delay ($d(e) = 0$). A static schedule must obey these intraiteration dependencies. The *cycle period* of a DFG is defined as

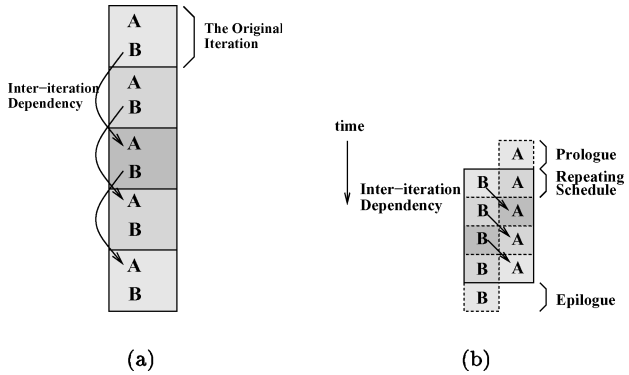


Fig. 4. (a) A static schedule of original loop. (b) The pipelined loops.

the computation time of the longest zero-delay path, which corresponds to the minimum schedule length when there is no resource constraint. We assume the computation time of a node is 1 time unit in this paper. Thus, the cycle period of the DFG in Figure 3(a) is 2.

2.2 Retiming and Software Pipelining

The retiming technique [Leiserson and Saxe 1991] 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.

The retiming function $r : V \rightarrow Z$ is the number of delays moved through node $v \in V$. Figure 3(b) shows the retimed DFG of Figure 3(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 3(b), the actual effect on the schedule of the new DFG is that the ith copy of A is shifted up and is executed with $(i - 1)$th copy of node B. Because there is not dependency between node A and B in the new loop body, these two nodes can be executed in parallel. The schedule length of the new loop body is then reduced from two control steps to one control steps. This transformation is illustrated in Figures 4(a) and (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

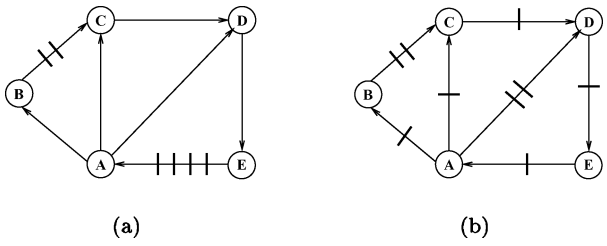


Fig. 5. (a) A DFG of the program in Figure 1(a). (b) The retimed DFG for the program in Figure 1(b).

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. For example, Figure 5(a) shows the DFG of the code in Figure 1(a). Figure 5(b) shows the retimed graph for software-pipelined loop in Figure 1(b) with $r(A) = 3$, $r(B) = r(C) = 2$, $r(D) = 1$, and $r(E) = 0$. We can see that there are exactly three copies of node A and two copies of node B and C in the prologue. Since the maximum retiming value is 3, there is no copy of node A in epilogue, and there is $3 - 2 = 1$ copy of node B and C in epilogue. The numbers of copies of the other nodes can also be obtained in a similar way.

From the retiming point of view, if there are k different retiming values, k iterations are pipelined in the static schedule. That is, the pipeline depth is k. In this paper, we also call the number of different retiming values k as the software pipelining "degree." The larger this value is, the deeper the pipeline is, and the shorter the schedule length can be achieved.

2.3 Rotation Scheduling

Rotation scheduling is a flexible technique for scheduling cyclic DFGs with resource constraints [Chao et al. 1997]. It produces a compact schedule iteratively. 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 the retiming on a static schedule is that the nodes are moved to a different iteration.

Figure 6(a) to Figure 8(b) illustrate the rotation scheduling progress on the program in Figure 1(b). In the first rotation phase, node A is rotated and rescheduled as shown in Figures 6(b) and (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. Figure 7(a) to Figure 8(b) show the second and the last rotation phases. The resulting schedule is optimal. The schedule length is only one control step. The pipeline depth is four. The italic letters in the schedule show how the second copy of the original loop body are pipelined with other copies in a new iteration. In the process of rotation scheduling, the state of rotation can be recorded by retiming functions. For example, the state of the last rotation is recorded as $r(A) = 3$, $r(B) = r(C) = 2$, $r(D) = 1$, and $r(E) = 0$.

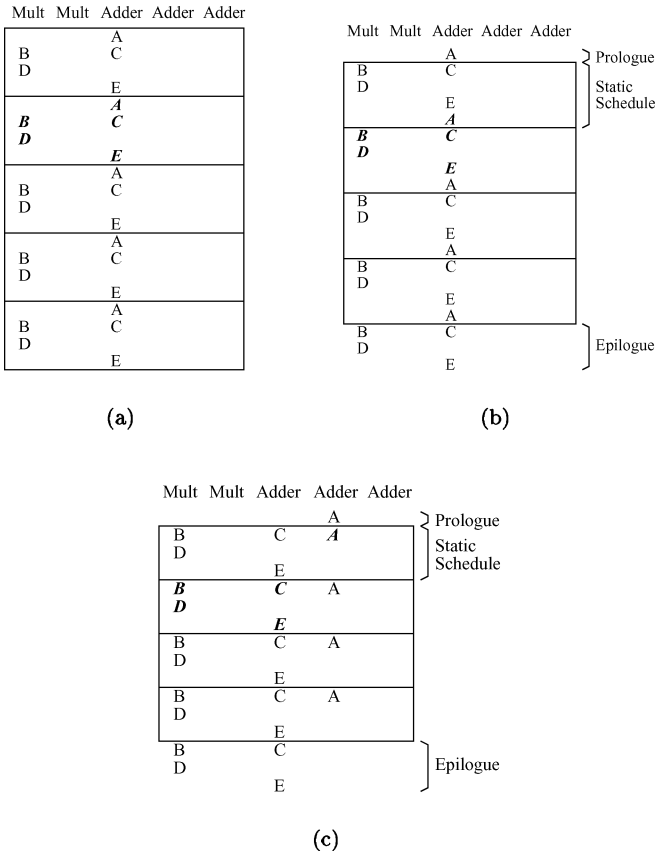


Fig. 6. (a) The original loop schedule. (b) The first phase rotation. (c) Rescheduling after rotation.

3. APPLICATION OF CRED TO VARIOUS PROCESSORS

In this section, we show that the CRED technique can be applied to any type of processors for software-pipelined applications with code size constraints. We use the five classes of processors introduced in Section 1 to illustrate that the applicability of this technique is independent of architectures. We also show that the code size is gradually reduced when the architectural support is increased from processor class 0 to 4. While processor class 4 obtains the smallest code size, it depends on the specialized hardware support for conditional execution of software-pipelined loops as in IA64. However, this kind of architectural support has not been found in DSP processors. We propose a modified TMS320 processor for processor class 3, which is practical and efficient for implementing CRED on DSP processors.

CRED technique uses 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 "lifetime" 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, that is, $p = \max_u r(u) - r(v)$.

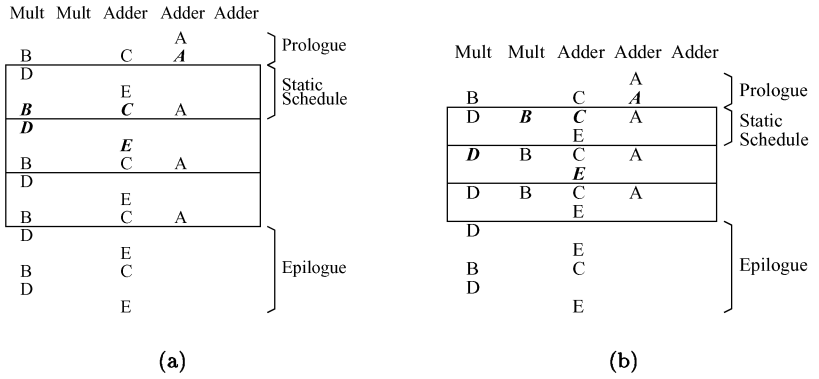


Fig. 7. (a) The second phase rotation. (b) Rescheduling.

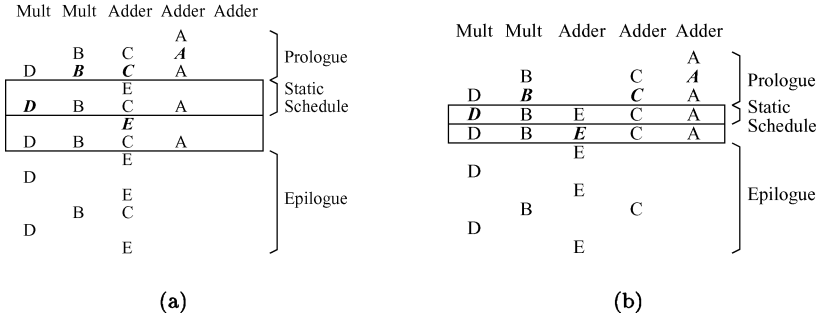


Fig. 8. (a) The third phase rotation. (b) The resulted pipeline schedule.

We also specify that the instruction is executed only when $0 \geq p > -n$. In other words, the instruction is disabled when $p > 0$ or $p \leq -n$, where n represents the original loop counter. In the following, we use the software-pipelined loop in Figure 1(b) to show the application of CRED technique on various processor classes.

3.1 Processor Class 0

Processor class 0 does not have conditional register, and does not support conditional execution with predication for all its instructions. For these processors, the conditional execution defined by CRED can be directly translated to *if–then* clauses. To implement CRED, each retiming value needs a counter and a branch. Thus, computations for updating the counter and controlling the branch need to be added in the loop. For processors in class 0, we can use conditional branches and retiming function to eliminate *all* the code in prologue and epilogue.

Figure 9(a) shows the code after removing prologue/epilogue of the code in Figure 1(b). The registers p1, p2, p3, and p4 are used for four different retiming values of 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 decreased by one for each iteration. Note that the loop is now executed for $n-3+3+3 = n+3$

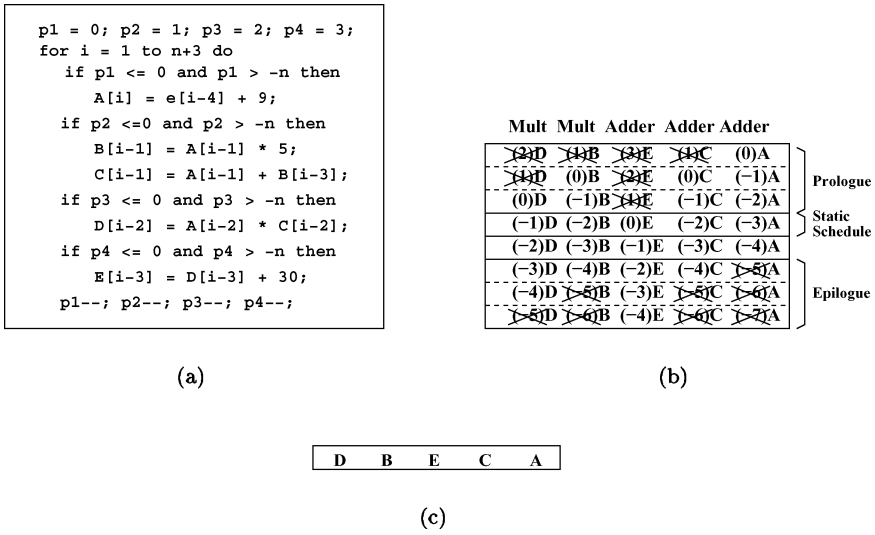


Fig. 9. (a) The code after applying CRED on processor Class 0. (b) The new execution sequence. (c) The reduced code size in memory.

times, since it first decreases three iterations by software pipelining, which is $\max_u r(u)$ in this example, and then adds three iterations from prologue and the other three from epilogue because of code size reduction. By doing so, the computation of node A starts from the 1st iteration and stops at the nth iteration, while the computations of nodes B and C start from the 2nd iteration and stops at $(n+1)$th iteration, and so on. Figure 9(b) shows the execution sequence of the conditional operations in our implementation when $n = 5$. The numbers in parentheses are the values of the counters. Note that each iteration executes only the static schedule of the loop body after applying CRED. Figure 9(c) illustrates the reduced code size for VLIW architecture with three adders and two multipliers. The effect of the additional computations of branches and counters on code size and performance will be discussed later.

3.2 Processor Class 1

Processor class 1 supports generalized predication. That is, all the instructions can be conditionally executed by checking "condition code" bits in the instruction [Seal 2000]. This kind of architectural support for predication can be found in ARM architecture. To implement the CRED technique on these processors, the *if–then* branch can be converted to a sequence of predicated instructions in the compiled code, as shown in Figures 10(a) and (b). Note that the branch in Figure 10(a) is equivalent to the branch in Figure 9(a). The interesting thing in the compiled code is that the second compare instruction (`cmplt`) is also predicated. The two compare instructions correctly set the value of register $p1$ without involving the and operation. The instruction count of this code is less than the compiled code of processor class 0 by two for the same branch.

Fig. 10. (a) A branch. (b) The compiled code in ARM.

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

Fig. 11. CRED on processor class 2.

3.3 Processor Class 2

For the class 2 processors 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 a predicate. The Boolean assignments of the predicate control the execution of these nodes. This mechanism is called "guarding" [Philips Inc. 2000]. By using the predicate registers, the performance penalty related to the branches, such as branch misprediction, can be eliminated. The code size of a loop after performing CRED on processor class 2 is the same as that on processor class 1. A part of the loop body after applying CRED on processor class 2 is shown in Figure 11.

3.4 Processor Class 3

Processor class 3 implements the conditional registers with the functionality of counters. The representative of this processor class is TI's DSP processor TMS320C6000. Figure 12 shows a portion of the new code for the program in Figure 1(b). The prefix `(p2)` means the guarded instruction is executed when `p2` is nonzero, while the prefix `(!p2)` indicates the instruction can only be executed when `p2` is zero [Granston et al. 2001; Texas Instruments Inc. 2000].

3.5 Modified TMS320 Processor

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.

$$\text{setp p1} = 3 : -\text{n}.$$

```
r2 = n + 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
```

Fig. 12. CRED on processor class 3.

```
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
```

Fig. 13. The code after totally removing prologue/epilogue on modified TMS320 processor.

This instruction sets the initial value of $p1$ to 3, and specifies that the guarded instructions will be disabled when $p1 > 0$ or $p1 \leq -n$. Figure 13 shows the code after applying CRED on the modified TMS320 processor.

For a VLIW processor, the computations of conditional register can be easily put into the available slots of an instruction word wherever possible after all the guarded instructions are issued. These inserted instructions can also be executed in parallel with other instructions through software pipelining. In most cases, code size reduction does not hurt the performance of an optimized loop.

The other option of CRED implementation can further reduce the initialization part of the code. We only need to initialize one conditional register `p1`. The other registers can be set up in the loop body by adding a value difference to `p1`. For example, if `p1 = 0` in initialization, we can set another conditional register `p2 = p1 + 1` in the loop body. Going further, we can even remove the initialization instruction of `p1`, if the loop counter `i` is decreased by 1 in each

```
p1 = 1; lc = n-1; ec = 4;
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
```

Fig. 14. CRED on processor class 4.

iteration, and p1 can be set as a function of i in the loop body. However, this option introduces the dependencies among the computations of conditional registers, which increases the difficulty for generating a schedule without increasing the schedule length. Therefore, we use the implementation in this paper, which allows the computations of conditional registers to be scheduled more freely.

3.6 Processor Class 4

Processors in class 4, such as IA64, provide special-purpose hardware support for conditional execution of software-pipelined loops [Intel Corporation 2001; Rau et al. 1992]. The conditional register is implemented as a rotating register, where each bit is a predicate. The rotating register is controlled by a set of special loop control instructions, such as brtop. Each of these instructions is actually a control logic updating the rotating register and the loop counter. The operations in the control logic is implemented by hardware. To implement CRED on this kind of processor, a 1-bit predicate in the rotating register is used to guard the instructions with the same retiming value. Also, only one loop control instruction, such as brtop, needs to be insert into the loop body. The number of inserted instructions for performing CRED on processor class 4 is the smallest among the four classes of processors; however, it needs specialized hardware support that is not found in DSP processors. Figure 14 illustrates a portion of code after removing prologue and epilogue on processor class 4. The initialization phase includes setting the first predicate in the rotating register and the other two counters (lc and ec) required by by the loop control instruction [Intel Corporation 2001].

4. CODE SIZE REDUCTION THEOREMS

In this section, we present the theoretical foundation of code size reduction based on retiming concept. It is a code transformation that attempts to remove the code in prologue and epilogue, so that the 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 $k \geq 0$.*

PROOF. Suppose that there is an edge $e(u \rightarrow v)$ and retiming function $r(u)$ and $r(v)$ for nodes u and v. Thus, there are $r(u)$ copies of node u and $r(v)$ copies of node v in prologue. We show that if the dependency represented by $e(u \rightarrow v)$ cannot be preserved by executing the static schedule as stated in the theorem, there must be at least one illegal retiming.

Part I. Edges with No Delay. Suppose that there is an edge $e(u \rightarrow v)$ with no delay before a retiming. Let u_i denote a copy of node u in the ith iteration. If u_i is not executed before node v_i by executing the static schedule after retiming, there must be $r(v) > r(u)$. That is, $d_r(e) = d(e) + r(u) - r(v) = 0 + r(u) - r(v) < 0$. Hence, the corresponding retiming on $r(v)$ is illegal.

Part II. Edges with Delays. Suppose that the delay count on edge $(u \rightarrow v)$ is j, and $j > 0$. This interiteration dependency defines that u_i needs to be executed before v_{i+j}. If this order cannot be preserved in static schedule after retiming, there must be $r(v) - r(u) > j$. We have $d_r(e) = d(e) + r(u) - r(v) = j + r(u) - r(v) < 0$. This is also an illegal retiming. □

Theorem 4.1 gives the correct execution sequence of prologue when we only execute the static schedule. 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. A similar execution 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 4.2. *Let $G_r = \langle V, E, d_r, t \rangle$ be the retimed data flow graph of a loop with retiming function r. Let n be the number of iterations in the original loop. The epilogue can be correctly executed by*

(1) *Executing only the repeated loop body and*
(2) *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 $(n+1)$th iteration, $\forall u \in V$ and $k \geq 0$.*

PROOF. The proof is similar to the proof of Theorem 4.1. □

Theorems 4.1 and 4.2 establish the theoretical foundation for code size reduction of a software-pipelined loop. They indicate that the code in prologue or epilogue can be removed by conditionally executing the schedule of loop body.

As we have presented in Section 3, conditional registers can be used to guard the execution of instructions in a static schedule. Then, the prologue and epilogue can be totally removed. In the following theorem, we decide the relationship between the number of conditional registers required for a total code size reduction and the number of distinct retiming values.

THEOREM 4.3 (CRED-TOTAL). *Let P be the number of available conditional registers, and R the number of different retiming values in a software-pipelined loop. If $P \geq R$, then all the codes in prologue and epilogue can be removed.*

PROOF. From Theorems 4.1 and 4.2, we know that the nodes in the static schedule need to be conditionally executed according to their retiming values. Since the nodes with the same retiming value can be guarded by one conditional register, it is clear that R conditional registers are needed to totally remove the prologue and epilogue. □

Theorem 4.3 actually defines the maximum software pipelining degree (the number of distinct retiming values) allowed for obtaining a software-pipelined loop without code size overhead in prologue and epilogue. For instance, if we want to obtain a pipelined loop without code size overhead by using four conditional registers, the maximum software pipelining degree performed on this loop should be less than or equal to 4. That is, there are at most three iterations in prologue and epilogue.

Since CRED technique uses retiming function to control the execution sequence of the nodes, it consumes less conditional registers than code-collapsing method presented in Granston et al. [2001], which needs two conditional registers for the same computation node, one for removing the copy in prologue, the other for epilogue.

For most DSP processors, the resource of conditional registers is very limited. TI's TMS320C6x, for example, can have up to six conditional registers [Texas Instruments Inc. 2000]. For some deeply software-pipelined applications, we may not have enough conditional registers to remove all the iterations in prologue/epilogue. It is obvious that we can add branches to simulate the function of conditional registers in the codes, but it introduces performance overhead. The following theorem states that CRED technique can also be applied to remove a part of prologue and epilogue when there are insufficient conditional registers. For example, suppose we have three different retiming values {0, 3, 4}. Originally, prologue and epilogue each contains codes of four iterations, since the maximum retiming value is 4. In the following theorem, we show that the innermost three iterations can be safely removed from both prologue and epilogue with only two conditional registers. That is, the nodes with retiming values 0 and 3 can be removed from prologue and epilogue.

THEOREM 4.4 (CRED-PARTIAL). *Let P be the number of available conditional registers. Let R be the number of different retiming values in a software-pipelined loop, and r_P be the Pth smallest retiming value. If $P < R$, then the innermost r_P iterations can be safely removed in both prologue and epilogue.*

PROOF. The proof of this corollary follows directly from Theorems 4.1–4.3. □

For node u whose retiming value $r(u) > r_P$, We can use the conditional register of the nodes with retiming value r_P to guard the node u. Consider the pipelined schedule shown in Figure 8(b), if we have only three 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 15(a) shows the loop after applying CRED-Partial on modified TMS320 processor. Figure 15(b) shows the

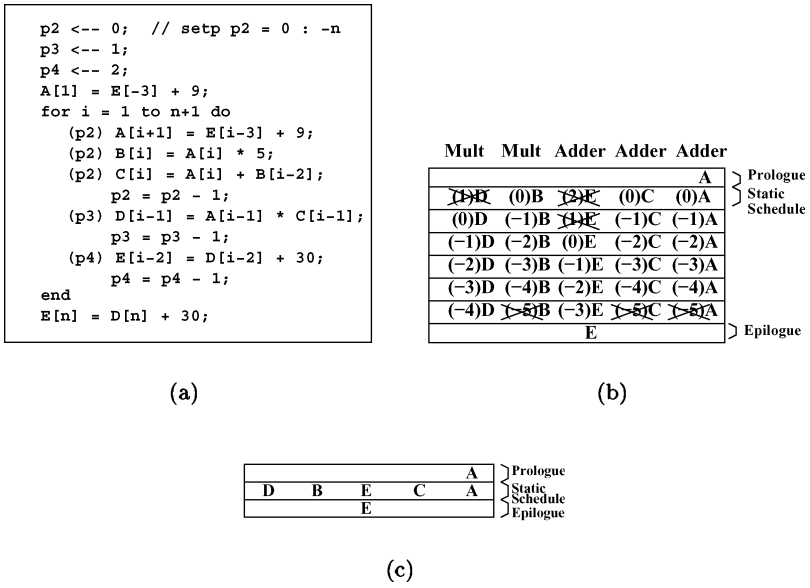


Fig. 15. (a) The code after reducing part of prologue and epilogue on modified TMS320 processor. (b) The execution sequence after applying CRED-Partial. (c) Reduced code size in memory.

execution sequence. After this reduction, there are only the first iteration with one node A remaining in the prologue and the last iteration with node E left in the epilogue. Figure 15(c) illustrates the reduced code size for a VLIW processor. In case that the innermost iterations have the most instructions of prologue and epilogue, CRED-Partial can be used to obtain smaller code size effectively. More importantly, Theorem 4.4 can be used in design process to explore the trade-off space of code size and software pipeline depth. The details of the algorithm will be presented in Section 5.2.

Based on the understanding of underlying relationship between retiming function and code size expansion for software-pipelined loops, we can accurately compute the expanded code size after software pipelining and the code size after applying CRED-Total on all five processor classes. In the following theorems, the code size is measured by the number of instructions in the compiled code.

THEOREM 4.5. *Given the retimed DFG* $G_r = \langle V, E, d_r, t \rangle$ *of a software-pipelined loop* $\mathcal{Q}$. *Let* $\max_u r(u)$ *be the maximum retiming value of* G_r. *The number of instructions in* $\mathcal{Q}$ *is* $\mathcal{N} = (\max_u r(u) + 1) * |V|$.

PROOF. For node v with retiming value $r(v)$, there are $r(v)$ copies of node v in prologue, and $\max_u r(u) - r(v)$ copies of node v in epilogue. Thus, totally there are $\max_u r(u)$ copies of node v out of the loop body for any node $v \in V$. It is obvious that there is exactly one copy of node v in the loop body. Hence, the total number of instructions in the software-pipelined loop is $\mathcal{N} = (\max_u r(u) + 1) * |V|$. □

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 various processors.

THEOREM 4.6. *Given the retimed DFG $G_r = \langle V, E, d_r, t \rangle$ of a software-pipelined loop $\mathcal{Q}$. Let R be the number of different retiming values in G_r. Then, the number of instructions in $\mathcal{Q}$ after applying CRED-total is*

—*processor class 0:* $\mathcal{N}_{\text{cred}} = R * 6 + |V|$;
—*processor class 1:* $\mathcal{N}_{\text{cred}} = R * 4 + |V|$;
—*processor class 2:* $\mathcal{N}_{\text{cred}} = R * 4 + |V|$;
—*processor class 3:* $\mathcal{N}_{\text{cred}} = R * 2 + |V|$;
—*processor class 4:* $\mathcal{N}_{\text{cred}} = |V| + 4$.

PROOF. It follows directly from the CRED technique discussed in Section 3. □

Consider a loop with 50 instructions, $\max_u r(u) = 1$, and $R = 2$. The code size of software-pipelined loop is expanded to $\mathcal{N} = 2 * 50 = 100$ instructions according to Theorem 4.5. After applying CRED on processor class 3, the code size is reduced to $\mathcal{N}_{\text{cred}} = 2 * 2 + 50 = 54$ instructions, according to Theorem 4.6. This result is very impressive for DSP processors without specialized architectural support as in IA64.

5. CODE SIZE REDUCTION ALGORITHMS

In this section, we present CRED algorithms. These algorithms can be used to meet various code size reduction requests for removing prologue and epilogue totally, partially, or only removing iterations in either prologue or epilogue. Our CRED algorithms are integrated with rotation scheduling to control the code size and software pipelining degree at the same time. The advantage of integrating code size reduction with software pipelining is to achieve the code size requirement with the least affect on schedule length. The algorithms are illustrated for modified TMS320 processor. The CRED algorithms on the other processor classes can be easily implemented according to our discussion in Section 3.

5.1 Total Code Size Reduction Algorithm

Algorithm 5.1 is used to totally remove the prologue and epilogue, assuming there are sufficient conditional registers. The code size reduction is performed during rotation scheduling. Rotation scheduling generates a software-pipelined schedule iteratively. Each rotation phase consists of four steps. The first step does normal rotation scheduling. It tries to find a more compact schedule by rotating the first row of the initial schedule and rescheduling the rotated nodes. The second step assigns one conditional register to guard the nodes with $r(v) = 0$, that is, the nodes not retimed. The third step detects the distinct retiming values produced by rotation scheduling, and assigns one conditional register for each retiming value. The loop counter and the number of consumed conditional registers are updated at the same time. Note that the inserted decrement instructions for updating conditional registers can also be rotated and rescheduled in the rotation scheduling procedure. Thus, our algorithm can produce the minimal code size with the least schedule length increment. For

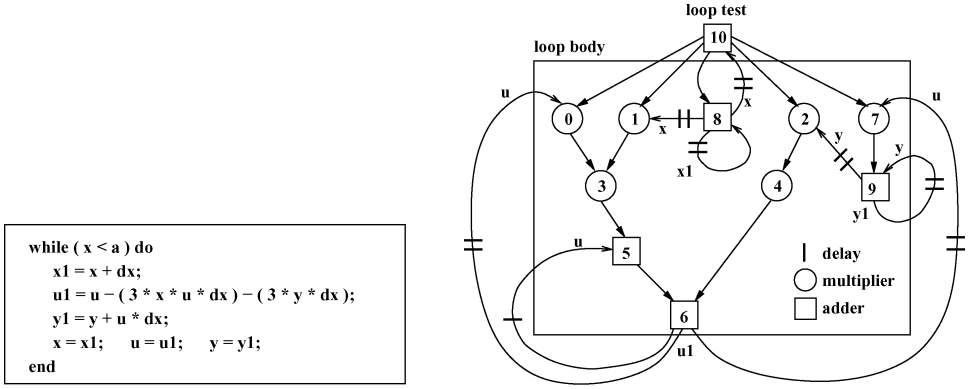


Fig. 16. (a) The differential equation solver. (b) The data flow graph.

a VLIW processor, these decrement instructions can be inserted into available slots of the instruction word without increasing the schedule length in most cases.

ALGORITHM 5.1 [CRED-TOTAL]

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.

$j \leftarrow 1$;
for $i = 0, \ldots, S.length$
  /* ***Step 1:*** Rotate nodes. */
  $Q \leftarrow First\_Row(S)$;
  $S_{opt} \leftarrow ReSchedule(S, Q)$;
  /* ***Step 2:*** Guard the nodes v with $r(v) = 0$. */
  $p_0 \leftarrow \max_u r(u), \forall u \in V$;
  Insert the decrement instruction of p_0;
  /* ***Step 3:*** Guard the nodes with new retiming values. */
  if there's a new retiming value $r(v)$
    $p_j \leftarrow \max_u r(u) - r(v)$;
    Insert the decrement instruction of p_j;
    $j \leftarrow j + 1$;
    $LC \leftarrow LC + 2$;
  endif
  /* ***Step 4:*** Update the inserted nodes */
  Update the decrement instructions in S_{opt};
endfor
return S_{opt}, j, LC;

We use the differential equation solver in Figure 16(a) as an example to illustrate the procedure of Algorithm 5.1. The DFG is shown in Figure 16(b). We use the boxes to represent additions and the circles to represent multiplications.

Figures 17 through 19(b) demonstrate the procedure of producing a compact schedule with the minimal code size by applying Algorithm 5.1. The final schedule on a processor with two multipliers and three adders has three control steps as shown in Figure 19(b). It has the same schedule length as the optimal

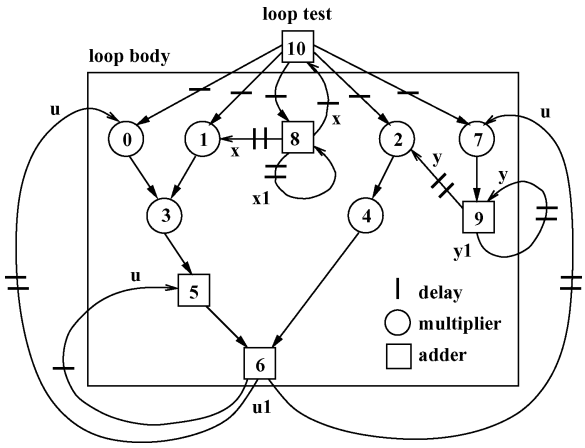


Fig. 17. Retiming node 10.

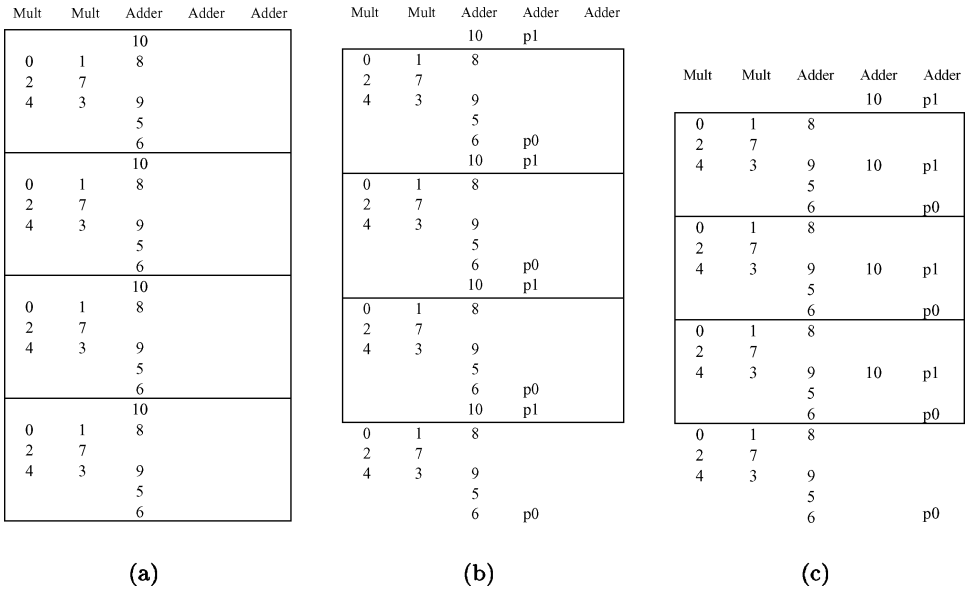
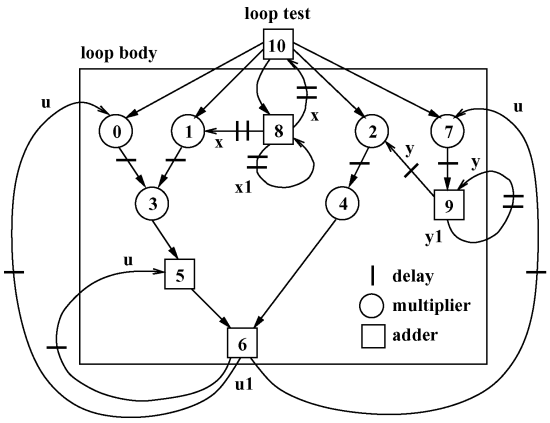


Fig. 18. (a) A global view of entire loop schedule for differential equation solver. (b) The first rotation. (c) Rescheduling.

schedule. Only two conditional registers are used to completely remove the prologue and epilogue.

5.2 Partial Code Size Reduction Algorithm

According to Theorem 4.4, CRED-Partial algorithm can be obtained after making some modifications on Algorithm 5.1. Two more input data need to be added, that is, the number of available conditional registers CR and the memory code size requirement W_{req}. For VLIW architecture, the code size requirement is represented by the number of instruction words for a particular processor



Mult	Mult	Adder	Adder	Adder
			10	p1
0	1		8	
2	7			
4	3	9	10	p1
0	1	5	8	
2	7	6		p0
4	3	9	10	p1
0	1	5	8	
2	7	6		p0
4	3	9	10	p1
0	1	5	8	
2	7	6		p0
4	3	9		
		5		
		6		p0

(a) (b)

Fig. 19. (a) Retiming nodes 0,1,2,7, and 8. (b) The final schedule.

configuration. The output will also report the number of used conditional registers CR_{use}, the new memory code size W_{new} and the largest retiming value of the nodes guarded by conditional registers in CRED-Partial, r_P. In Step 3, the *if* statement will check if there are available registers. If so, CR_{use} is increased by one in this step. If not, it guards all the other nodes with the last conditional register that contains the largest retiming value r_P as shown in Theorem 4.4.

/* **Step 3:** Guard the nodes with new retiming values. */
if there's a new retiming value $r(v)$ and $CR_{\text{use}} < CR$
  $p_j \leftarrow \max_u r(u) - r(v)$;
  Insert the decrement instruction of p_j;
  $j \leftarrow j + 1$;
  $LC \leftarrow LC + 2$;
  $CR_{\text{use}} \leftarrow CR_{\text{use}} + 1$;
else
  $r_P \leftarrow \max_u r(u) - p_j$;
  Guard all the nodes v whose $r(v) > r_P$
  with conditional register p_j;

We also add Step 5 to output the shortest schedule satisfying the memory code size requirement:

/* **Step 5:** Output the schedule satisfying memory code size requirement. */
if $W_{\text{new}} \geq W_{\text{req}}$
  $S_{opt} = S$;
  Exit the loop;

CRED-Partial algorithm captures the code size and software pipeline depth during software pipelining, and produces the shortest schedule satisfying the code size requirement. Thus, the software pipelining degree can be controlled by compiler when the program memory size 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, and use an unoptimized version of the code [Granston et al. 2001; Rau et al. 1992]. By using CRED-Partial, the compiler

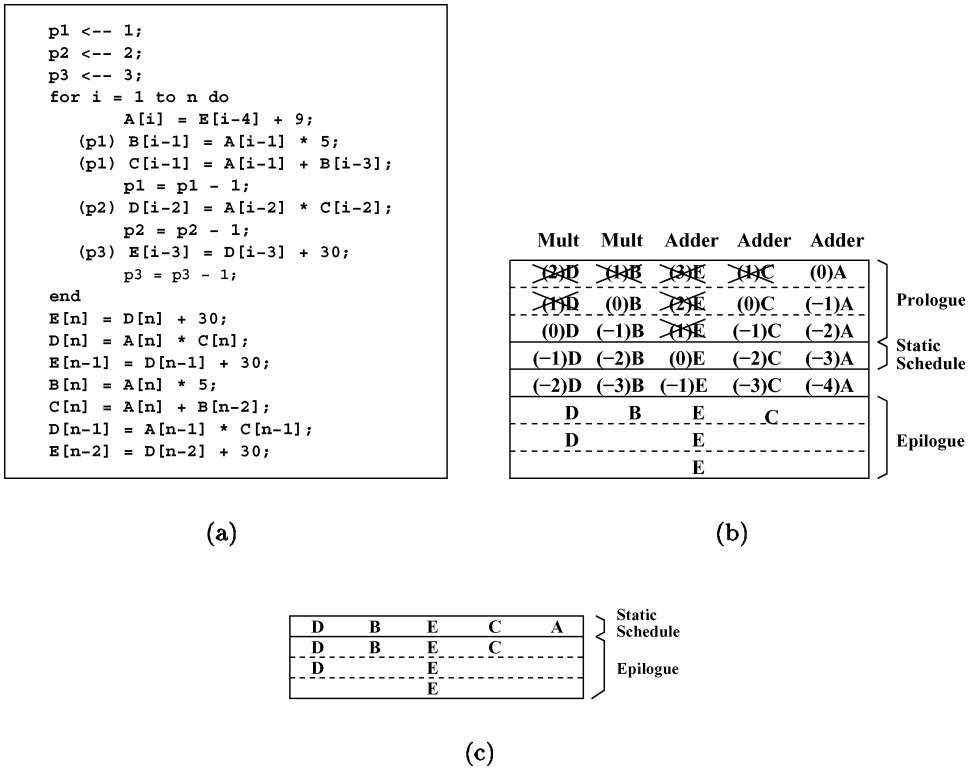


Fig. 20. (a) The code after applying Prologue-only CRED on modified TMS320 processor. (b) The execution sequence. (c) The reduced code size in memory.

is able to effectively explore the trade-off space between code size and software pipeline depth. Figures 15(a)–(c) illustrate the results of partial code size reduction.

5.3 Prologue/Epilogue Only Code Size Reduction Algorithm

In some cases, removing only prologue or epilogue, or part of prologue or epilogue can also achieve desired code size. Both CRED algorithms in Sections 5.1 and 5.2 can be modified to obtain prologue/epilogue-only CRED algorithm.

When only prologue is removed, the nodes with retiming value $\max_u r(u)$ do not need to be guarded, since there is still a complete epilogue section left in the program for the completion of the pipeline. Also, the loop counter register can be removed. Similarly, when only the code in epilogue is removed, the nodes with retiming value 0 do not need to be guarded. Note that the loop counter for this algorithm is still n when removing only the prologue or the epilogue. Similarly, CRED-Partial algorithm can be modified to remove part of iterations in either prologue or epilogue. The new loop counter is $n - i$, where i denotes the number of iterations left in prologue/epilogue after applying CRED. Figure 20(a) illustrates the loop after applying prologue-only CRED on modified TMS320 processor for the example shown in Figure 1(b). Figure 20(b) shows

Table I. The Results of CRED-Total on Modified TMS320

Benchmarks	Number of Registers	Code Size			Schedule Length	
		SP	CRED	%	SP	CRED
IIR	2	16	12	25.0	2	2
DEQ	2	22	15	31.8	3	3
All-pole	4	60	23	61.7	5	6
Elliptic	2	68	38	44.1	11	11
4-Stage	3	78	32	59.0	7	7
Voltera	2	54	31	42.6	9	9

the new execution sequence, and Figure 20(c) shows the reduced code size in memory.

Comparing the prologue/epilogue-only CRED technique with the code collapsing technique in Granston et al. [2001], it is interesting to see that the effect of code collapsing is the same as prologue/epilogue-only CRED. That is, code collapsing becomes a special case of CRED technique. Since CRED technique is based on the fundamental understanding of retiming and code size expansion of software-pipelined loops, the CRED technique can be generally applied to any software-pipelined application on any processor class.

6. EXPERIMENTAL RESULTS

We have experimented the CRED algorithms with a set of well-known benchmarks on various processors, including IIR filter (IIR), differential equation solver (DEQ), all-pole filter (All-Pole), fifth order elliptic filter (Elliptic), 4-stage lattice filter (4-Stage), and voltera filter (Voltera). In most cases, we can use only three or fewer conditional registers to completely remove all the iterations in prologue and epilogue without incurring performance penalty. The code size is measured as the number of instructions in a schedule including prologue, loop body, and epilogue. The schedules are generated on simulated processors with three adders and two multipliers, assuming the computation time of each functional unit is one time unit. The experiments show the promising results of code size improvement.

Table I displays the code size by applying CRED-Total algorithm (Algorithm 5.1) on modified TMS320 processor. The second column shows the number 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 the software-pipelined loops. The fourth column displays the code size after performing code size reduction. The code size reduction percentage ranges from 25.0% to 61.7%, as shown in column "%". 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 software-pipelined schedule lengths, except for All-Pole filter, whose schedule length is increased by one control step. The performance overhead introduced by additional computation for code size reduction is very small.

Table II shows the code size results after applying CRED-Total technique on five different types of processors, processor class 0 to 4, starting from column 4 in that order. The second column shows the code size of original loops, and the

Table II. The Results of CRED-Total on Various Types of Processors

			Various Types of Processors									
			Class 0 (StarCore)		Class 1 (StrongARM)		Class 2 (TriMedia)		Class 3 (Modified TMS)		Class 4 (IA64)	
Benchmarks	Original	SP	Size	%	Size	%	Size	%	Size	%	Size	%
IIR	8	16	20	−25.0	16	0	16	0	12	25.0	12	25.0
DEQ	11	22	23	−4.5	19	13.6	19	13.6	15	31.8	15	31.8
All-pole	15	60	39	35.0	31	48.3	31	48.3	23	61.7	19	68.3
Elliptic	34	68	46	32.4	42	38.2	42	38.2	38	44.1	38	44.1
4-Stage	26	78	44	43.6	38	51.3	38	51.3	32	60.0	30	61.5
Voltera	27	54	39	27.8	35	35.2	35	35.2	31	42.6	31	42.6
Average Improvement				18.2		31.1		31.1		44.2		45.6

Table III. Code Size Exploration for All-Pole Lattice Filter Using 2 Conditional Registers

Pipeline Depth	Schedule Length	Number of Instruction			Instruction Words		
		SP	CRED-P	%	SP	CRED-P	%
2	11	30	19	36.7	23	11	52.2
3	7	45	34	24.4	30	19	36.7
4	5	60	49	18.3	35	29	17.1

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 show the impressive improvement on the code size of pipelined loops. The code size reduction percentages on processor classes 3 and 4 are greater than the other two kinds of processors. Also, there are two negative percentages appearing in the column of processor class 0. These numbers indicate that the computations of the conditional execution is larger than the number of reduced instructions in prologue/epilogue. According to Theorems 4.5 and 4.6, the reduction can only be achieved when $\mathcal{N}_{\text{cred}} < \mathcal{N}$. Since Theorems 4.5 and 4.6 can accurately compute the code size of a software-pipelined loop and the code size after applying CRED, the designer can easily decide whether or not to apply CRED in optimization. The experimental results also show that the smallest code size can be achieved on processor class 4, which has specialized hardware support and loop control instructions as in IA64. For DSP processors without these special architectural features, the modified TMS320 architecture can achieve the minimal code size. The experimental results show that the CRED technique can be generally applied to various kinds of processors to reduce the code size, and impressive results can be achieved in most cases, especially for processor classes 3 and 4. The last row of the table shows the average code size improvement for each processor class.

Given the number of available conditional registers, we can explore the trade-off between code size and software pipeline depth by using the CRED-Partial algorithm in Section 5.2. Table III illustrates several design choices in code size/performance trade-off space with two conditional registers for All-pole filter. The column "Number Instruction #" shows the number of instructions for the software-pipelined loop (field "SP"), the number of instructions for the loop

after performing code size reduction (field "CRED-P"), and the reduction percentage (field "%"). The column "Instruction Words" shows the number of instruction words of the compiled code for the particular processor with three adders and two multipliers. For example, for a pipelined depth of 2, we get a schedule length of 11 control steps, and 23 instruction words, which can be reduced to 11 instruction words with three conditional registers. When the pipeline depth increases, the memory code size is increased and the schedule length is decreased. By using the CRED technique, designers are able to generate a compact schedule with much smaller code size than that of a software-pipelined loop, and explore the trade-offs between code size and performance effectively.

7. CONCLUSION

Software pipelining is widely used to exploit instruction-level parallelism and to improve the performance of applications in embedded systems. However, this performance optimization technique expands the code size, which is a major concern for embedded systems with very limited on-chip memory size. In this paper, we developed the theoretical foundation for a general code size reduction technique, CRED, based on the fundamental understanding of the relationship among retiming, software pipelining, and code size expansion. It can be easily integrated in a compiler to optimize the code size. We proposed the implementations of the CRED technique on various processor classes, with or without conditional registers. Our CRED algorithms can control the code size during software pipelining. The experimental results show that the CRED technique can be effectively applied on various types of processors while maintaining an optimized performance in most cases. The CRED technique can also be used to explore the trade-off space between code size and software pipeline depth efficiently.

REFERENCES

ARAUJO, G., DEVADAS, S., KEUTZER, K., LIAO, S., MALIK, S., SUDARSANAM, A., TJIANG, S., AND WANG, A. 1995. Challenges in code generation for embedded processors. In *Code Generation For Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publishers, Dordrecht, Ch. 1, 4–17.

CHAO, L.-F., LAPAUGH, A. S., AND SHA, E. H.-M. 1997. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16*, 3 (March), 229–239.

CHAO, L.-F. AND SHA, E. H.-M. 1995. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing 10*, 207–223.

CHAO, L.-F. AND SHA, E. H.-M. 1997. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems 8*, 12 (Dec.), 1259–1267.

CHEN, F., O'NEIL, T. W., AND SHA, E. H.-M. 2000. Optimizing overall loop schedules using prefetching and partitioning. *IEEE Transactions on Parallel and Distributed Systems 11*, 604–614.

CHEN, F., TONGSIMA, S., AND SHA, E. H.-M. 1998. Loop scheduling algorithm for timing and memory operation minimization with register constraint. In *Proceedings 1998 IEEE Workshop on Signal Processing Systems (SiPS)*, 579–588.

GRANSTON, E., SCALES, R., STOTZER, E., WARD, A., AND ZBICIAK, J. 2001. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proceedings 3rd IEEE/ACM Workshop on Media and Streaming Processors*, 29–38.

HENNESSY, J. AND PATTERSON, D. 1995. *Computer Architecture: A Quantitive Approach*, 2nd ed. Morgan Kaufmann, San Mateo, CA.

HUFF, R. A. 1993. Lifetime-sensitive modulo scheduling. In *Proceedings SIGPLAN'93 ACM Conference on Programming Language Design and Implementation*, 258–267.

Intel Corporation 2001. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*. Intel Corporation. (literature number 245317-003).

KUCK, D. J., KUHN, R. H., PADUA, D. A., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 207–218.

LAM, M. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings SIGPLAN'88 ACM Conference on Programming Language Design and Implementation*, 318–328.

LANNEER, D., PRAET, J. V., KIFLI, A., SCHOOFS, K., W.GEURTS, THOEN, F., AND GOOSSENS, G. 1995. CHESS: Retargetable code generation for embedded processors. In *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publishers, Dordrcht, Ch. 5, 85–296.

LEISERSON, C. E. AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 5–35.

Motorola Digital DNA & Agere Systems 2001. *StarCore SC140 DSP Core Reference Manual*. Motorola Digital DNA & Agere Systems.

Philips, Inc. 2000. *TM-1300 Media Processor Data Book*. Philips, Inc.

RAU, B. R. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, 63–74.

RAU, B. R. AND FISHER, J. A. 1993. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing 7*, 1/2 (July), 9–50.

RAU, B. R. AND GLAESER, C. D. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings 14th ACM/IEEE Annual Workshop on Microprogramming*, 183–198.

RAU, B. R., SCHLANSKER, M. S., AND TIRUMALAI, P. P. 1992. Code generation schema for modulo scheduled loops. In *Proc. 25th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, 158–169.

SEAL, D., Ed. 2000. *ARM Architecture Reference Manual*, 2nd ed. Addison-Wesley, Reading, MA.

Texas Instruments, Inc. 2000. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, Inc. (literature number SPRU189F).

Texas Instruments, Inc. 2001a. *Code Composer Studio IDE v2 White Paper*. Texas Instruments, Inc. (literature number SPRA004).

Texas Instruments, Inc. 2001b. *TMS320C6000 Optimizing Compiler User's Guide*. Texas Instruments, Inc. (literature number SPRU187).

WANG, Z., O'NEIL, T. W., AND SHA, E. H.-M. 2001. 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.

Received June 2002; revised January 2003; accepted February 2003