SOFTWARE PIPELINING OF NESTED LOOPS FOR REAL-TIME DSP APPLICATIONS

Jian Wang

Speech Recognition Software
Nortel Montréal Lab.
Montréal, QC, Canada, H3E 1H6
email: *jiwang@nortel.ca*

Bogong Su

Dept. of Computer Science
The William Paterson University of New Jersey
Wayne, NJ 07470, USA
email: *bsuwpc@frontier.wilpaterson.edu*

ABSTRACT

Modern DSP Processors have been integrated with *Instruction-Level Parallelism(ILP)*, which presents a challenge to exploit ILP within DSP applications. *Software Pipelining* is an efficient technique used to expose ILP for loop programs and has been widely used for current microprocessors. It has been recently used in DSP compilers, but only for the innermost loops. This paper proposes a new approach which extends software pipelining from innermost loops to whole nested loops in DSP applications. Given a perfect loop, we apply an existing software pipelining approach for the innermost loops, then use the so-called *pipelining-dovetailing* transformation to extend software pipelining to the outer loops. We also present a transformation to convert a non-perfect nested loop into a perfect one. We have verified the above transformations with some nested loops selected from DSP compiler-challenge C code. The preliminary results are further presented in this paper.

1. INTRODUCTION

Modern DSP processors have been integrated with Instruction-Level Parallelism(ILP). For example, Motorola DSP56300 allows the parallel moves with MAC operation, which performs a multiply-accumulate, two data moves, and two pointer updates. This presents a challenge to exploit ILP within DSP applications for designers of DSP optimizing compilers and for programmers. Even though DSP compilers have been around for many years, their performance is less than acceptable. For example, the overhead of compiled code(in terms of clock cycles and code space) typically lays in the range from 2 to 8 [1, 2]. Therefore design of new optimization techniques for DSP compilers is in high demand.

Some efficient approaches in microprocessor compilers such as scheduling and software pipelining [3] have been applied on DSP compilers [4, 5, 6]; In [1], some problems of software pipelining in some commercial DSP compilers are mentioned. [4] uses dependence retiming to enhance the effectiveness of software pipelining. Since most of DSP programs are loop-intensive, software pipelining can greatly benefit from improved clock cycles and code space. However, only the optimization of the innermost loops has been focused on so far [1, 4, 5]. Since nested loops are frequently-seen program structure in real-time DSP applications, it is necessary to extend the optimization from the innermost loops to the nested loops, in particular for those nested loops whose innermost loops have small trip counts. On the other hand, almost all modern DSP processors except TIs C6x have small register files, a lack of orthogonal instructions and multiple function units. Therefore applying the normal software pipelining technology to the nested loops is impractical.

In this paper, we present a new approach to software pipeline the nested loops in DSP applications, which could be used for modern DSP processors. Our approach retains the existing mature framework of software pipelining for the innermost loops, but extends this framework from the innermost loops to the whole nested loops in a smooth and efficient fashion. Given a perfect nested loop, we apply an existing software pipelining approach on the innermost loop, then use the so-called pipelining-dovetailing transformation to extend software pipelining to the outer loops. We also present a transformation to convert a non-perfect nested loop into a perfect one.

To verify our approach, we chose some DSP compiler-challenge C codes which have nested loops from [1], and conducted the experiment on Motorola DSP56300. Preliminary results show that our approach can obtain an average performance improvement of 23% over software pipelining of innermost loops only.

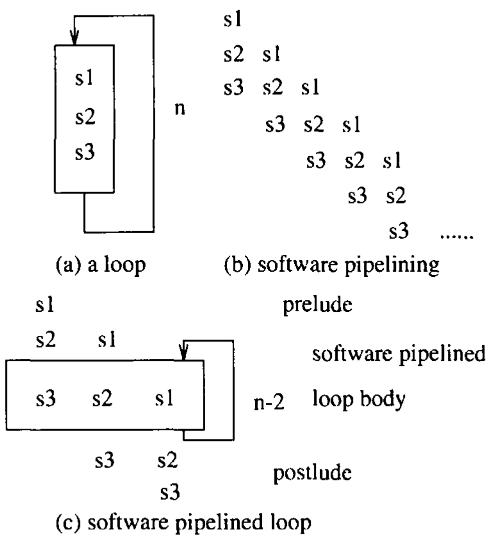


Figure 1 An example of software pipelining

2. SOFTWARE PIPELINING AND PIPELINING-DOVETAILING

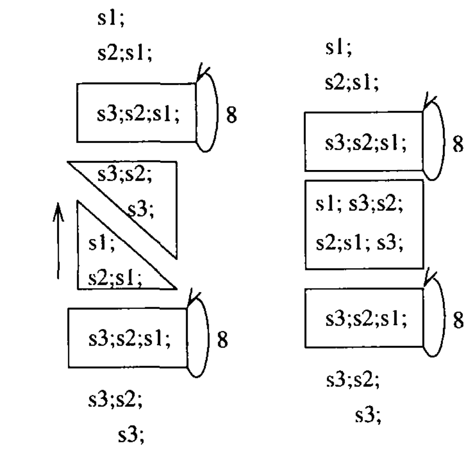
Software pipelining is an efficient instruction-level loop scheduling technique [3]. It tries to overlap the execution of operations from several consecutive iterations of a loop under the constraints of data dependences and hardware resources. A software pipelined loop consists of three parts – *prelude, postlude and the steady state.* Figure 1 gives a simple example where the operations will be issued for execution at the same machine cycle if they are on the same line. Software pipelining is traditionally applied to the innermost loops only.

```
for i1=1 to 2 do
  for i2=1 to 10 do
  {
   s1;
   s2;
   s3;
  }
```

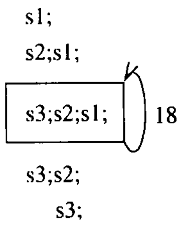
(1) a nested loop

```
for i1=1 to 2 do
 { s1;
   s2; s1;
  for i2=3 to 10 do
   {s3; s2; s1; }
    s3; s2;
     s3; }
```

(2) software pipelining of the innermost loop



(3) unrolling the outer loop (4) dovetailing



(5) merging

```
s1;
s2;s1;
for i1=1 to 2 do
  for i2=lb(i1) to 10 do
   { s3;s2;s1;}
s3;s2;
   s3;
```

(6) re-writing in the form of a nested loop

Figure 2 Pipelining-Dovetailing

Pipelining-dovetailing [7] is a simple but efficient loop transformation used to extend software pipelining from the innermost loops to the whole nested loop. The principle of pipelining-dovetailing is illustrated with a simple nested loop shown in Figure 2(1). First, assume we software pipeline the innermost loop as shown in Figure 2(2). Then imaging that we fully unroll the outer loop as shown in Figure 2(3). Now, we present a transformation, called *dovetailing*, to transform Figure 2(3) to Figure 2(4). In Figure 2(3), the prelude of the second software pipelined loop can be moved upward to fit together with the postlude of the first software pipelined loop, thus generating Figure 2(4). After dovetailing, the loop in Figure 2(4) can be *merged* as shown in Figure 2(5). We re-write the merged loop in the form of a nested loop as shown in Figure 2(6), where $lb(i1)$ means that $i2$ should count from 3 if $i1 = 1$, otherwise from 1. We call *"pipelining-dovetailing"* the transformation from Figure 2(2) to Figure 2(6).

3. OUR TRANSFORMATIONS FOR DSP

This section describes our new optimization approach used for nested loops in real-time DSP applications. The ideas behind our approach are to maintain the framework of the existing DSP compilers and software pipelining of the innermost loops and to insert two efficient but simple pre-transformations before or after code generation. The two transformations are: (1) convert a non-perfect nested loop into a perfect one and (2) apply pipelining-dovetailing to the converted loop. In a DSP compiler, our transformations can be implemented in front-end (high-level) or in back-end (instruction-level).

3.1. The Conditions

The conditions under which pipelining-dovetailing can be applied are as follows: (1) the nested loops must be perfect and (2) dovetailing can not break the data dependences among the software pipelined innermost loops.

A *perfect nested loop* is a loop where there is no code between different loop-levels. For example, the FIR Filter nested loop [1] in Figure 3 is not a perfect loop (called non-perfect nested loop) since there is code between the innermost level and the outer level (e.g. *sum=0;* and *output[i]= sum>>15;*).

```
for ( i=0; i<N-ORDER; i++)
{ sum=0;
for (j=0; j<ORDER; j++)
 { sum += array[i+j]*coeff[j]; }
   output[i] = sum>>15;
}
```

Figure 3 The nested loop in FIR Filter

For the second condition, we presented a theorem based on the concept of *distance vectors* in [7]. The theorem provides a method to verify the condition before dovetailing is applied. Actually, this condition is very weak especially for the nested loops in real-time DSP applications. In practice, we only need to check the data dependence between the postlude of a software pipelined innermost loop and the prelude of its next software pipelined innermost loop. In most cases, there are no data dependences found.

3.2. Renaming and Loop Distribution

The first condition is not always satisfied for DSP's nested loops. Therefore, we present *renaming* and *loop distribution* transformations to convert a non-perfect loop into a perfect one before applying pipelining-dovetailing.

Note that for most nested loops in DSP applications, the code between different loop-levels is the initialization part or the result-stored part. It is not difficult to remove this code with the renaming and loop distribution.

Let us take Figure 3 as an example. *sum* is a variable which is initialized for the loop body of the innermost loop, its value will be stored after the innermost loop is finished. For each iteration of the outer loop, *sum* is re-used. For the first step, we need to rename all re-used variables to make loop distribution feasible. To do so, for each re-used variable, we introduce an array whose dimension is the number of iterations of the outer loop (if this number is not available at compiler time, we have to take the maximum one in the worst case) and use the array element to replace the variable in the code. The nested loop in Figure 3 becomes the one in Figure 4(1) after *sum* is renamed.

After renaming is done, we can apply loop distribution to the loop. The loop body of the outer loop can be considered into three parts – the initialization part, the innermost loop and the result-stored part. Loop distribution will convert the original nested loop into three loops. The initialization part forms a single level loop as does the result-stored part. The main part of the code now is a perfect nested loop. After loop distribution is done, the nested loop in Figure 4(1) becomes the one in Figure 4(2).

```
int sum[N-ORDER];
for ( i=0; i<N-ORDER; i++)
 {  sum[i]=0;
 for (j=0; j<ORDER; j++)
  { sum[i] += array[i+j]*coeff[j]; }
    output[i] = sum[i]>>15;
 }
```

(1) Renaming

```
int sum[N-ORDER];

for ( i=0; i<N-ORDER; i++)
{  sum[i]=0;}

for ( i=0; i<N-ORDER; i++)
 for (j=0; j<ORDER; j++)
  { sum[i] += array[i+j]*coeff[j]; }

for ( i=0; i<N-ORDER; i++)
{output[i] = sum[i]>>15;}
```

(2) Loop distribution

Figure 4 Renaming and Loop distribution

Note, although we illustrate our transformation in C language level before code generation, the transformation can be also done in instruction level after code generation. We will discuss the tradeoff between these two cases in the next subsection.

3.3. The Optimization Framework

As a result, the general framework of the nested loop optimization for DSP applications can be described in the following steps:

1. check the first condition; if the loop is not a perfect one, for each re-used variable in the initialization part and the result-stored part, do the renaming; then do loop distribution;
2. software pipeline the innermost loop;
3. check the second condition; use the theorem in [7] or simply check the prelude and the postlude of the software pipelined innermost loop; if the second condition is not satisfied, return with not-dovetailing;
4. do the pipelining-dovetailing on the perfect nested loop;

The above framework can be implemented in high level before code generation. In high level, renaming and loop distribution are easily done. But, during code generation, some instructions (e.g. loop index register initialization) may be generated between loop levels and may degrade the performance of the final result. The advantage of this scheme is that the compiler implementation is easy.

We can also implement the above framework in instruction level after code generation. In order to perform renaming and loop distribution, we need to adjust register allocation and generate some new instructions for holding the program semantics. The advantage of the scheme is that the optimal code may be obtained, but the drawback is the complexity of its implementation. We suggest using this scheme if hand-craft code is needed. In section 4, we use this scheme to conduct our experiment.

```
for (i=0; i<100; i++)
 for (j=0; j<10; j++)
  C[i,j] +=A[i,j] * B[j];
```

(1) a nested loop

```
move #>FC,r1
move #>FA,r2
move #>FB,r3
move #9,m3
DO #100,L2
DO #10,L1
move X:(r1),a
move Y:(r2)+,x0
move Y:(r3)+,y0
mac x0,y0,a
move a,X:(r1)+
L1:
L2:
```

(2) assembly code

Figure 5 A Working Example (1)

4. AN EXAMPLE AND PRELIMINARY RESULTS

In order to show how we conduct the preliminary experiment, we first present a simple working example to illustrate the procedure of our new optimization approach.

The example nested loop is shown in Figure 5(1). We first generate its Motorola DSP56300 [8] assembly code(Figure 5(2)) where we use modular addressing mode for $B[j]$ and hardware loop for *"for (...)"*. It is a perfect nested loop so we can directly apply software pipelining and pipelining-dovetailing. Figure 6(1) gives the result after the innermost loop is software pipelined, while Figure 6(2) presents the result after pipelining-dovetailing is done on Figure 6(1). Assuming all data is stored in the internal memory, the execution time of Figure 6(1) is 2709 cycles and Figure 6(2) is 2011 cycles. Therefore, pipelining-dovetailing can obtain an improvement of 26% over software pipelining of innermost loops only.

```
move #>FC,r1
move #>FA,r2
move #>FB,r3
move #9,m3

DO #100,L2
move X:(r1),a Y:(r2)+,x0
move Y:(r3)+,y0
DO #9,L1
mac x0,y0,a X:(r1),a Y:(r2)+,x0
move a,X:(r1)+ Y:(r3)+,y0
L1: mac x0,y0,a
    move a,X:(r1)+
L2:
```

(1) software pipelining

```
move #>FC,r1
move #>FA,r2
move #>FB,r3
move #9,m3

move X:(r1),a Y:(r2)+,x0
move Y:(r3)+,y0

DO #999,L1
mac x0,y0,a X:(r1),a Y:(r2)+,x0
move a,X:(r1)+ Y:(r3)+,y0

L1: mac x0,y0,a
    move a,X:(r1)+
```

(2) software pipelining plus pipelining-dovetailing

Figure 6 A Working Example (2)

In the preliminary experiment, we chose three DSP compiler-challenge C codes from [1]. The FIR filter and FIR filter with redundant load elimination are two level nested loops. The JPEG Discrete Cosine Transform (JPEG DCT) is a three level nested loop. JPEG DCT contains a large amount of code between the innermost level and the mid-level loops, so it is not suitable for our optimization before some modifications are made. Since our experiment is conducted by hand, we only optimize the innermost part.

We apply the simplest software pipelining method URPR [9] on the innermost loops of these three loops. Since there are no branches and loop-carried dependencies in all of these three codes, URPR can produce the same results as other software pipelining approaches which are more complicated and unnecessary for our simple programs.

As a result, compared to software pipelining of innermost loops only, our approach can obtain an improvement of 30% for FIR Filter, 12% for FIR Filter with Redundant Load Elimination and 26% for the modified JPEG DCT.

5. CONCLUSION

This paper has presented a new optimization approach for the nested loop programs in real-time DSP applications. Based on the existing quite mature framework of software pipelining technique for the innermost loops, our approach integrates renaming, loop distribution and pipelining-dovetailing.

Nested loops are frequently-seen program structure in real-time DSP applications. According to the characteristics of DSP's nested loops, our approach first uses renaming and loop distribution transformations to convert non-perfect nested loops to perfect ones, then applies software pipelining on the innermost loops, and finally performs pipelining-dovetailing on the converted loops and extends the effect of software pipelining to the whole nested loops.

6. ACKNOWLEDGMENT

We would like to thank Ms. Jeanette Myers for proof-reading. Prof. Bogong Su was partially supported by the Faculty Summer Research Award 1997 from the Center for Research, School of Science and Health, The William Paterson University of New Jersey.

7. REFERENCES

[1] Markus Levy. C compiler for dsps: Flex their muscles. In *EDN, June 5*, 1997.

[2] Peter Marwedel. Code generation for core processors. In *proc. of DAC-97*, 1997.

[3] B. R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), January 1993.

[4] F. Sanchez and J. Cortadella. Timed-constrained loop pipelining. In *proc. of 8th ISSS*, 1995.

[5] Jr. Thomas J. Dillon. The Use of Software Pipelining in Developing DSP Algorithms for the TMS320C6x. In *proceedings of International Conference on Signal Processing Application and Technology*, Sept. 1997.

[6] S. Novack, A. Nicolau, and N. Dutt. A Unified Code Generation Approach Using Mutation Scheduling. In *Book "Code Generation for Embedded Processors"*, 1995.

[7] Jian Wang and Guang R. Gao. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *proceedings of the 1996 International Conference on Compiler Construction, Lecture notes in Computer Science, No.1060*, pages 1 - 17. Spring-Verlag, April 1996.

[8] *DSP56300 24-Bit Digital Signal Processor Family Manual.* Motorola, Inc., 1995.

[9] B. Su, S. Ding, and J. Xia. URPR - an extension of URCR for software pipelining. In *proceedings of the 19th International Symposium on Microprogramming and Microarchitectures (MICRO-19)*, pages 104 - 108, 1986.