SOURCE-LEVEL LOOP OPTIMIZATION FOR DSP CODE GENERATION

Bogong Su

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

Jian Wang

Real Time Speech Systems
Nortel Montreal Lab.
Montreal, QC, Canada, H3E 1H6
email: jiwang@nortel.ca

Andrew Esguerra

Dept. of Computer Science
The William Paterson University
of New Jersey
Wayne, NJ 07470, USA
email: njd44@gw.wilpaterson.edu

ABSTRACT

The performance of current C compilers for DSP is almost unacceptable. One of the most important reasons is the lack of implementing software pipelining. This paper presents a remedy called source-level loop optimization. DSP programmers can use source-level loop optimization first then input its result to the DSP compiler to obtain better assembly code. The implementation of source-level loop optimization is easier than that of software pipelining. The preliminary result with the DSP compiler-challenge C code shows that source-level loop optimization is a portable and efficient approach. The detailed method and working examples are presented.

1. INTRODUCTION

Instruction-level parallelism(ILP) [6] has been used extensively in general purpose microprocessors. It is a set of design techniques that speed up a program by executing in parallel several operations [3]. Recently a VLIW-like DSP, TMS320C6x, has been developed, its hardware can offer ILP for two multiplications, six ALU operations and several memory access operations. It also adopts some software optimization technologies such as software pipelining from general purpose optimizing compilers [2]. On the other hand, many of the popular DSPs' hardware provides some sort of ILP, so-called parallel instruction. For example, the Motorola MC 56300 allows parallel move operations. The scheduling and coalescing phase in its compiler can take advantage of this parallelism and produce the better code.

Since the sizes of DSP applications are getting bigger and reducing the time-to-market a new DSP product becoming more important, the more efficient high level language compilers, such as C compilers, are urgently expected. However, the implementation of some complicated optimization techniques such as software pipelining in the backend of DSP compilers, has been a challenge due to the small size of register files of almost all DSP processors. The performance of current C compilers for DSPs is almost unacceptable. The overhead of compiled codes, in terms of clock cycles and code space, falls typically in the range of 2 to 8 [4,5]. To solve this problem, besides improving DSP compilers, it is necessary to study the methods in which to write a better C program so that the current DSP compilers can generate more efficient codes. [1] provides some useful suggestions. [4] shows an example of Vector Multiply that Analog Devices uses software pipelining at the source level, however it is just a tuning for an individual program and there is no detailed method presented. In this paper we present a so-called source level loop optimization approach for DSP compilers to produce more efficient codes. We also analyze the result of the preliminary experiment on the DSP compiler-challenge code.

2. BASIC CONCEPT

Software pipelining is extensively used in optimizing compilers for microprocessors as an efficient instruction level loop optimization technique[6]. Figure 1 illustrates its basic idea. Fig.1(1) and Fig.1(2) are a loop body and its data dependence graph(DDG) respectively. The critical path is the longest path through the DDG. The length of critical path determines the minimum length of the result of scheduling and coalescing. In this simple example it equals three. For further optimization, we can unroll the loop body and then pipeline the unrolled bodies as shown in Fig.1(3) and Fig.1(4) respectively. Fig.1(5) is the new DDG for the operations in the frame in Fig.1(4). There is only data anti-dependency, shown by dotted lines. Using this method all data dependencies are removed. Data anti-dependencies can be eliminated when reading a register earlier than writing to the same register or register renaming technique is adopted. Therefore, the critical path length is reduced to one and the body length of the software pipelining result is also one as shown in Fig.1(6).

From this example, we know that software pipelining can be used to remove data dependency in the loop body. Hence reducing the critical path length and the loop body length. However, most current DSP C compilers have not implemented software pipelining due to the above-mentioned reasons. Fig.2(1)is the typical structure of a DSP compiler. Our approach is illustrated in Fig.2(2): there is a software pipelining based loop transformation at the source level to get the improved C code, i.e. some data dependency in the loop body can be removed. It provides more opportunity to the scheduling and coalescing phase in DSP compilers.

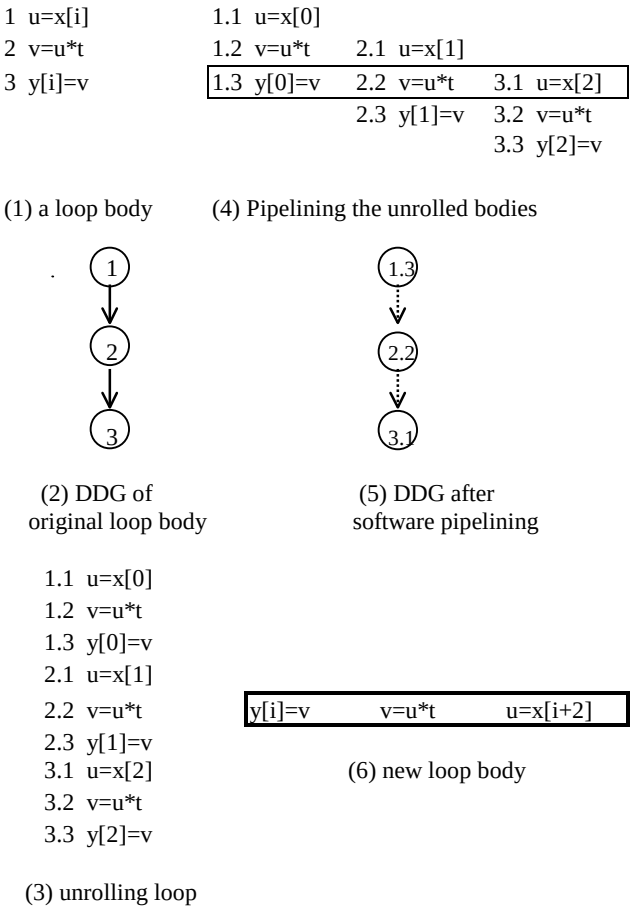


Figure 1 Software Pipelining

3. SOURCE-LEVEL LOOP OPTIMIZATION

The basic idea of source-level loop optimization is illustrated in Figure 3. Fig.3(1) is the original C source code which is a loop taken from the FIR program.

(1) Re-writing the loop body of C source code.
Some C statements include several operations which can not be executed at the same time due to data dependency, e.g. there is only one statement in the loop body in Fig.3(1) which includes two load operations from memory into registers and at least one arithmetic operation. This statement can not be executed within one cycle. These kind of statements must be decomposed into several operations as shown in Fig.3(2).

(2) Building the DDG of the re-written code.
From the DDG in this example as shown in Fig.3(3), it is obvious that the critical path length is two.

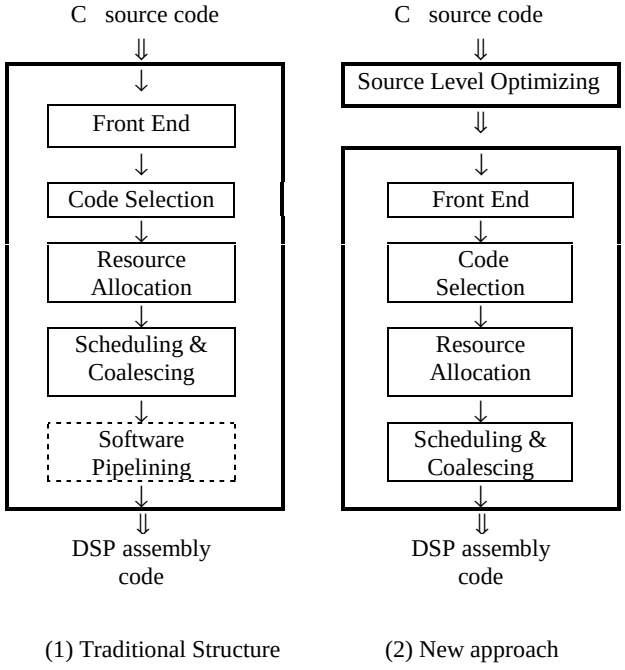


Figure 2 DSP Code Generation

(3) Software pipelining the re-written loop body.
We use a simple software pipelining technique [7] called URPR: unrolling the loop body first, then pipelining the unrolled loop bodies, rerolling the result of pipelining to obtain a new loop body finally. The second step, pipelining the unrolled loop bodies is much simpler than that of regular software pipelining in optimizing compilers. We consider the inter-body data dependency only and disregard the resource conflict problem. There are two bodies to be pipelined in Fig.3(4). The operations within the frame consist of the new loop body. It is easily ascertained that the data dependency has been removed in the new loop body, i.e. the critical path length is only one.

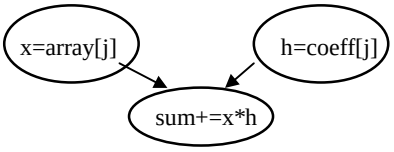
The operations before and after the new loop body are called prelude and postlude respectively.

```
fir_inner(const short
array[], const short
coeff[], short output[])
{
int ,j,sum;
sum=0;
for(j=0;j<100;j++)
{
sum+=array[j]*coeff[j]
}
}
```

(1) a sample program

```
x=array[j];
h=coeff[j];
sum+=x*h;
```

(2) Re-written loop body



(3) DDG of re-written loop body

```
x=array[j]  h=coeff[j]
sum+=x*h            x=array[j]   h=coeff[j]
                    sum+=x*h
```

(4) New loop code after source level optimization

```
{
int j,sum,x,h;
sum=0;
x=array[0];
h=coeff[0];
for(j=1;j<100;j++)
{
sum+=x*h;
x=array[j];
h=coeff[j];
}
sum+=x*h;
}
```

(5) Improved Source code

```
move y:(r0)+,x0
asr #23,a,a  y:(r4)+,y0
mac +x0,y0,a
asl #23,a,a
```

(6) DSP assembly code of original loop

```
asr #23,a,a
mac +x1,x0,a     y:(r0)+,x1
asl #23,a,a      y:(r4)+,x0
```

(7) DSP assembly code of improved loop body

Figure 3 Example of Source-Level Loop Optimization

(4) Obtaining the improved C source code.
Since C language is not a parallel programming language, we must re-write the new loop body in Fig.3(4) into a regular sequential C code, shown in Fig.3(5). However, the C compiler can determine that there is no data dependency in the loop body. Finally it is necessary to combine the prelude and postlude parts into the improved C code and adjust the index to maintain the semantic correctness.

By examining Fig.3(7), the DSP assembly code produced by the C compiler for the Motorola DSP 56300 we find that the loop body length is three instead of one due to the extra shift operations before and after mac operation to avoid overflow. Compared with the DSP assembly code of the original loop body shown in Fig.3(6), a 25% speed increase has been attained by using source-level loop optimization.

4. PRELIMINARY RESULTS

To verify our approach we have chosen all the DSP compiler-challenge C codes from [4], except the FIR Filter with Redundant Load Elimination which is a loop-unrolled version of FIR Filter. Their assembly code generated by current commercial DSP C compilers carry a big overhead. Fig. 4 presents the example of Dot Product, one of the compiler-challenge programs.

Table 1 uses the length of the innermost loops of the assembly code to compare the performance. The table lists the results generated by Motorola DSP 56300 compiler, the results by using source-level loop optimization before Motorola compiler and the "Optimum" code picked from[4] which is produced by the Tasking compiler from modified C source code. Tasking applies some machine dependent modifications, beside re-assigning the memory space allocation it also changes the variable type from short to fract to eliminate shift operations and uses pointers instead of arrays.

Table 1 Performance Comparison
(Length of Innermost Loop)

PROGRAMS	Motorola Compiler	Source -level Optimization + Motorola Compiler	"Optimum" Code
Dot Product	11	6	3
Vector Multiply	7	6	3
FIR Filter	12	12	10
Lattice Synthesis	17	16	7
IIR Filter	31	28	12
Vector Codebook Search	20	15	16
JPEG	25	20	10

Table 1 shows that the source-level loop optimization can obtain an average of 17% performance improvement for those seven compiler-challenge programs. There is no improvement for FIR program because the register allocation in Motorola DSP compiler is quite poor. It generates new data dependence in the assembly code of the source-level optimization result, i.e. the critical path length can not be reduced. In Lattice Synthesis and IIR Filter programs, source-level optimization eliminates

some data dependence. However, the result length of the source-level optimized assembly code is longer than that of the original assembly code due to the poor register allocation problem[8], therefore limiting the performance improvement of these two programs.

Though there is still an average of 82% overhead for these seven compiler-challenge programs by comparing the results of source-level optimization with the "Optimum" codes it is possible to get further improvement by applying some other source level optimization approaches e.g. using pointers instead of arrays.

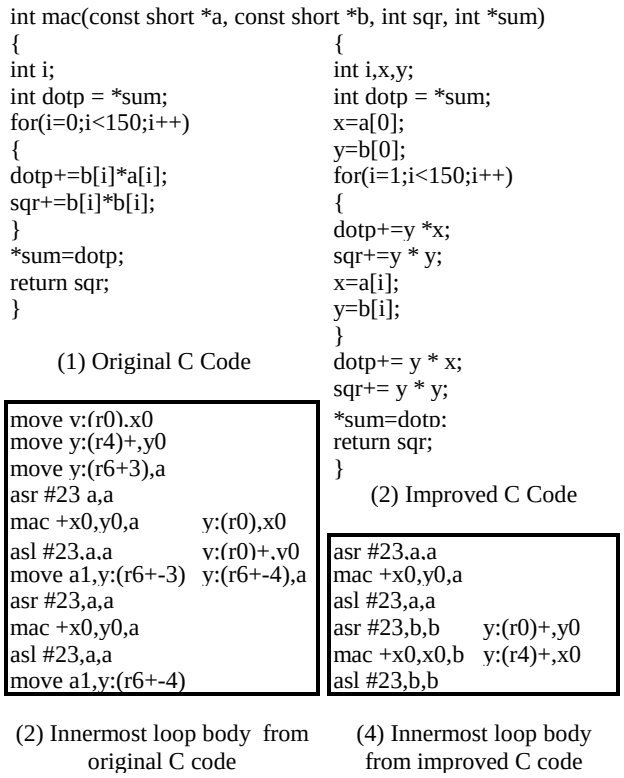


Figure 4 Dot Product Program

5. CONCLUSION

Source-level loop optimization is a portable and efficient approach to improve the DSP assembly code generated by current commercial DSP compilers which do not implement software pipelining. DSP programmers can develop a machine dependent preprocessor or conduct the optimization manually to obtain code improvement. Compared with conducting the software pipelining manually on the assembly code level, the source-level loop optimization is far easier because the source code is simpler and resource conflict checking can be omitted.

We can combine source-level optimization with other optimizing approaches to obtain further performance improvement, and extend this idea to the optimization of nested loops[9].

6. ACKNOWLEDGMENTS

This work was partially supported by the Faculty Summer Research Award 1998 from the Center for Research, College of Science and Health, The William Paterson University of New Jersey.

7. REFERENCES

[1] A. Davis, "Tips for Writing More Efficient DSP C Code", *EDN*, June 5, 1997, p.98.

[2] T. Dillon, "The Use of Software Pipelining in Developing DSP Algorithms for the TMS320C6x", *Proc. of ICSPAT97*, Sept. 1997, 834-837.

[3] P. Faraboschi, G. Desoli, and J. Fisher, "The Latest Word in Digital and Media Processing", *IEEE Signal Processing*, vol. 15, No. 2, March 1998, 59-85.

[4] M. Levy, "C Compilers for DSPs - Flex Their Muscles", *EDN*, June 5, 1997, 93-107

[5] P. Marwedel, "Code Generation for Core Processors", *Proc. of DAC-97*, 1997,232-237.

[6] B. Rau and J. Fisher, "Instruction-level Parallel Processing: History, Overview, and Perspective", *The Journal of Supercomputing*, 7(1), Jan. 1993.

[7] B. Su et al, "URPR - an Extension of URCR for Software Pipelining", *Proc. of MICRO-19*, 1986, 104-108.

[8] B. Su et al, "A Study of Source-level Loop Optimization for DSP Code Generation", Technical Report, College of Science and Health, William Paterson University, Aug. 1998.

[9] J. Wang and B. Su, "Software Pipelining of Nested Loops for Real-time DSP Applications", *Proc. of ICASSP98*, May, 1998.