A practical approach to DSP code optimization using compiler/architecture

B. Varnagiryte[1], A. Zemelis[1], O. Olsen[1], P. Koch[1], O. Wolf[2], E. Kazanavicius[3]

(1) Embedded Systems Group, CPK, Aalborg University, Denmark, E-mail: {bv, za, oo,pk}@kom.auc.dk

(2) DSP Assist, Denmark, E-mail: wolf@dspassist.com

(3) DSP Laboratory, Studentu 50 – 214c, Kaunas University of Technology ,LT-3031Kaunas,Lithuania, E-mail: ekaza@dsplab.ktu.lt

Introduction

Real-time digital signal processing algorithms such as filters, FFTs, speech coding algorithms, etc. can be realized on Digital Signal Processors (DSPs) using assembly language. As the size of DSP applications increase programming such algorithms in assembler is a time and cost consuming task, because it requires a thorough knowledge of both the processor architecture and the algorithm to write an efficient assembly code.

To reduce programming cost and to increase reusability of the code high-level languages (e.g., C and C++) and their compilers can be used. However, it is a well-known fact that in most cases a compiler-generated code for DSP processors has lower performance in terms of execution time and memory usage than a hand-optimized code. Consequently, to increase code performance selected portions of the compiler generated code may have to be rewritten in assembler. Therefore, a unified performance evaluation of the DSP processor and the associated compiler becomes relevant.

The evaluation may consider different aspects such as speed (i.e., execution time), reliability, utilization of memory and functional units or power consumption.

The maintainability (or reuse) aspect is also very important if the same piece of code should by executed on different processors [1].

In the past benchmarking of digital signal processing hardware was conducted almost only by the chip vendors themselves [2,3]. Within the last decade independent DSP analysts such as BDTi [12], EEMBC [13] and DSPstone [2,3] employed a benchmarking methodology for DSP compilers that compares the performance of the compiled C-code to the hand-optimized assembly code in terms of program/data memory consumption and execution time. The hand-optimized code can be considered optimal, and by analyzing the generated assembly code it is possible to identify the parts of the C-code that were interpreted differently by the compiler.

A revised C compiler benchmarking methodology was proposed in [4]. This methodology, applied to three different types of DSP architectures, such as enhanced conventional DSP processors, superscalar DSP processors, and VLIW DSP processors, shows that in many cases efficient implementation of a particular DSP application depends on the selected architecture as well as on the complexity of the DSP application. The methodology also helps to analyze how much the compiler-generated code differs from optimal (i.e., assembly code), to identify the strong and weak parts of the compiler, and to investigate, which features of the architecture may influence the performance of the C compiler.

The evaluation of C compilers is a widely discussed topic of recent scientific investigations. In [5] instruction scheduling and register allocation are discussed as important compiler efficiency influencing factors. This is because parallel execution of multiple instructions requires correct instruction combination as well as appropriate operands and data types.

In this paper we analyze the C-compiler for the Texas Instruments TMS320C55x (C55x) processor using two well-known computationally intensive algorithms (FIR filter and LMS algorithm). We present an approach consisting of a sequence of directed experiments that helps to analyze and exploit some important features of the TMS320C55x architecture such as dual multiply accumulate unit (dual MAC) and instruction level parallelism. In [11] Texas Instruments recommend non – standard C coding guidelines that help to exploit the dual MAC unit using a direct form I of block FIR filter kernel as an example. In our experiments we use a direct form I transposed implementation of block FIR filter and propose a different approach to dual MAC exploitation while recording code size and cycle count for the experiments.

The remainder of the paper is organized as follows: Section II gives a brief overview of relevant features of the C55x architecture, Section III describes some of the optimization techniques used in our investigation, Sections IV and V present results and an analysis of these results and final conclusions are presented in Section VI.

Architecture description

The Texas Instruments TMS320C55x is a multi-issue 16-bit fixed-point DSP processor family. The main features of C55x are as follows:

- Five functional units: Instruction Buffer Unit (I-Unit), Program Flow Unit (P-Unit), Address-Data Flow Unit (A-Unit), Data Computation Unit (D-Unit), Memory Interface Unit (M-Unit);
- Two Multiply Accumulate Units (dual MAC), two ALUs, and four accumulators;
- Five 16-bit data busses: three for data read and two for data write;
- Variable instruction width architecture. The width of the instruction word can vary between 8 and 48 bits;
- Multi issue: executes up to two instructions in parallel.

The exploitation of variable instruction width could lead to smaller program memory usage, while employment of the dual MAC architecture as well as dual ALUs and four accumulators could lead to parallel execution of instructions. However, there is no hardware-based scheduling of instructions; therefore the parallelism has to be detected and scheduled at compile time.

The main requirements for parallelism exploitation are in general that the resource constraints of the architecture are respected. This implies in particular that [6]:

- Two instructions can be executed in parallel if assembled they do not exceed 48 bits (6 bytes);
- Each instruction makes no more than a single data memory access;
- Memory, cross unit and constant busses do not compete for access;
- Parallelism is allowed between operations executed within the A-unit, the D-unit or the P-unit;
- Parallelism between subunits within the A-unit, D-unit or P-unit is allowed.

Optimization techniques

Algorithm kernels are used for compiler benchmarking in all references [2,3,4,7,8]. Such kernels as FFTs, FIR filter, etc. are the building blocks of many DSP applications. A large percentage of the execution time is spent in these computationally intensive kernels, which can significantly influence the overall execution time of the program. The advantages of using kernels are:

- their simplicity for implementation and optimization in assembly language,
- the possibility for reuse in many different types of applications,
- the possibility to measure the processor performance in these kernels.

In our approach we analyze the compiler-generated code, and estimate the compiler's ability to exploit the features of the given architecture.

As discussed in [10], compiler optimization techniques operate on three levels: *machine dependent*, describing the instruction-level sensitivities of a compiler (Coding Style Transformations), *architecture dependent*, denoting those parts of a program that relate to the general hardware implementation, but not to a specific machine (e.g. Multiple MAC Units, Parallel Instruction Execution or Multiple Data Streams) and *architecture independent*, related to those aspects of program formulation that do not depend on a particular computer system or even on a type of implementation, like pipeline processing (Common Sub-expression Elimination, Dead Variable Elimination, Code Motion or Constant Propagation). Our investigation covers optimization techniques that allow dual MAC and instruction level parallelism exploitation.

Consider an FIR filter with impulse response h and input x, where the output y is given by Eq.1.

$$y(n) = \sum_{m=0}^{N-1} h(m)x(n-m)\,. \qquad (1)$$

A parallel execution of an FIR filter on the 'C55x can be performed using 3 data busses and produces two outputs as shown in Fig.1. This is done by computing two sequential filter iterations (e.g., outputs $y(n)$ and $y(n\text{-}1)$) in parallel when both MAC units utilize a single coefficient, $h(m)$ as shown in Equations (2) through (5):

$$y(n) = h(0)*x(n) + h(1)*x(n\text{-}1) + \ldots + h(N\text{-}1)*x(n\text{-}N\text{-}1). \qquad (2)$$

$$y(n\text{-}1) = h(0)*x(n\text{-}1) + h(1)*x(n\text{-}2) + \ldots + h(N\text{-}1)*x(n\text{-}N\text{-}2). \qquad (3)$$

$$y(n\text{-}2) = h(0)*x(n\text{-}2) + h(1)*x(n\text{-}3) + \ldots + h(N\text{-}1)*x(n\text{-}N\text{-}3). \qquad (4)$$

$$y(n\text{-}3) = h(0)*x(n\text{-}3) + h(1)*x(n\text{-}4) + \ldots + h(N\text{-}1)*x(n\text{-}N\text{-}4). \qquad (5)$$

This approach is called time-based loop unrolling, [14].

For the first term in each of these two rows, the first MAC unit could compute the $h(0)*x(n)$, while the second MAC unit could compute $h(0)*x(n\text{-}1)$. For these two computations only three different values are required, i.e., $h(0)$, $x(n)$, and $x(n\text{-}1)$. Three available data-read busses (DB, CB, BB) permit reading these three data values from separate memory units in one instruction cycle. In the next cycle, $h(1)*x(n\text{-}1)$ and $h(1)*x(n\text{-}2)$ are computed similarly and added to the previous result, until both of the output vector samples are computed. In this way, DSP performance at two MAC operations per clock cycle should be maintained.

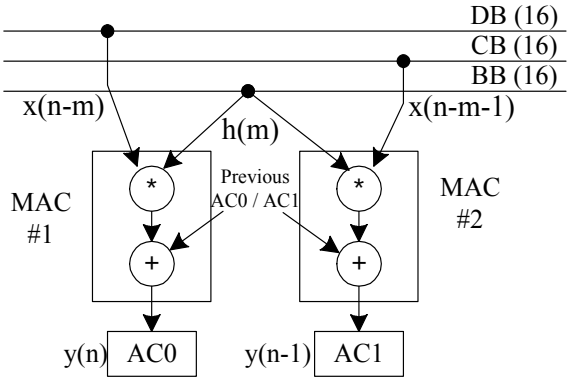

Fig. 1. **FIR filter on C55x**

A similar implementation is expected for the LMS algorithm kernel as it is based on FIR filtering.

To evaluate the compiler/architecture interaction the chosen algorithm should reveal the correspondence between language features expressing:

- Data access,
- Loop constructs,
- Arithmetic,

and architectural features such as:

- Addressing,
- Zero overhead looping,
- Computational resource utilization (dual MAC),
- Parallel instruction execution.

These optimizations applied to the FIR filter kernel are discussed in the following section. Metrics for the selected experiments are presented.

Results and analysis

Figure 2 shows one possible sequence of experiments leading to parallel instruction execution.

The experiments were based on the execution time profiling information from the selected FIR and LMS kernels. Both kernels are implemented as functions with appropriate parameters. This is natural considering usage of these kernels in larger applications.

Functions are declared as `void`, variables are declared using native processor data types in order for compiler to select the optimal instructions. Thereby the compiler could map an operation into single instruction, or execute instructions in parallel if possible.

Experiments were performed with a combination of manual, automatic and pragma specified C code transformations using the Texas Instruments Code Composer Studio, Version 2.0. The full symbolic debug and interlisting modes were turned off during the experiments.

Experiments E1-E4 denote the most successful order of optimizations to exploit instruction level parallelism. C code and generated assembly code obtained for E1-E4 experiments are explained afterwards. All the results are presented for a 16 tap FIR filter, producing 40 output samples. Some observations for a smaller number of filter taps are also made.

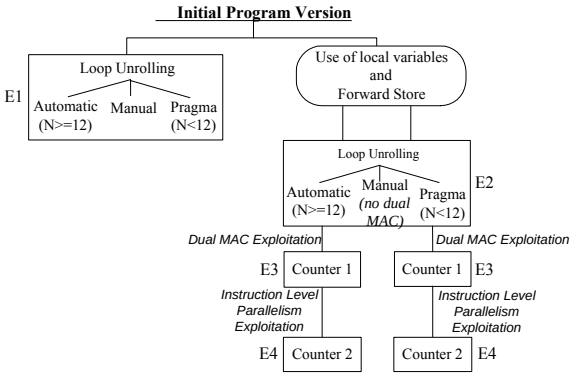

Fig. 2. **Directed sequence of experiments**

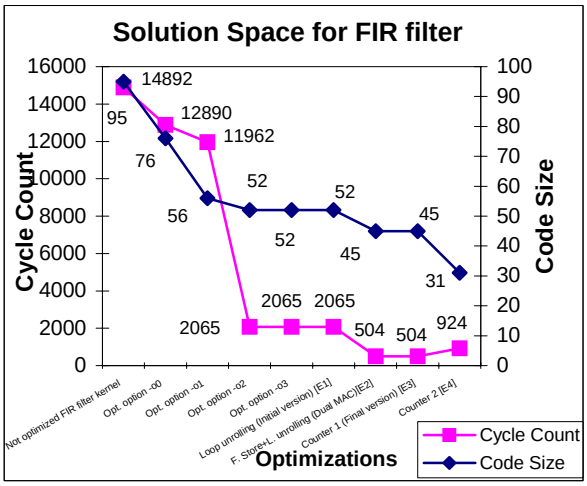

Fig. 3. **Results for FIR filter**

Figure 3 shows the results obtained. As seen, the compiler optimizer is capable of reducing the cycle count from 14,893 to 2,066, just by using built-in optimizations supplied by –o2 and -o3 compiler options, [9].

E1: Initial version with Loop Unrolling optimization

In order to generate dual MAC instruction, compiler maximum optimization options –o3 -mb should be enabled. After applying these options to the FIR filter kernel, this version was named "initial", and was used as a starting point for the optimizations in Experiments E2 through E4 shown in Figures 5 through 7.

The C code for the "initial" version of FIR filter kernel as well as the generated assembly code are shown in Figure 4. In this version of code, filter coefficients *h*[ ], input samples *x*[ ], and output vector *y*[ ] were defined as global variables.

C Code

```
void fir(const
         int h[],
         int x[],
         int y[])

{
int n,m;

n=nSamples-1;
while (n >= 0)
 {
   m = 0;
   while (m <nTaps )
    {
     y[n]+=h[m]*x[n-
m];
      m++;
    }
  n--;
 }
}
```

Assembly Code

```
MOV #((_y+38)&0xffff),AR3
 MOV #39, T1
 MOV #19, BRC0
 MOV #15, BRC1
 RPTBLOCAL L6-1
            ; loop starts
L3:
 ADD #(_x &0xffff),T1,AR4
  MOV #(_h & 0xffff), AR2
  SUB #1, AR4, AR1
   RPTBLOCAL L5-1
            ; loop starts
L4:
  MOV *AR3(short(#1)),AC0
  MACM *AR4-,*AR2,AC0,AC0
  MOV AC0,*AR3(short(#1))
  MOV *AR3, AC0
 MACM *AR1-,*AR2+,AC0,AC0
  MOV AC0, *AR3
            ; loop ends
L5:
      SUB #2, T1
      SUB #2, AR3
           ; loop ends
L6:
    return
            ; return
```

Fig. 4. **Experiment E1 (Initial version with Loop Unrolling)**

The generated assembly code reflects that the compiler recognizes the possibility for the **while**( ) loops to be translated into a block repeat instructions (**RPTBLOCAL)** using two branch counters, BRC0 and BRC1. It understands that the += (accumulate) and * (multiply) operators can be translated into one multiply-add instruction.

The compiler also performs automatic loop unrolling. The loop unrolling does not help the compiler generate dual MAC instructions, however, and the compiler instead replicates the inner loop twice producing two sequential MAC instructions. In addition, on each iteration of the inner loop the compiler loads and stores elements of *y*[ ] that do not change until the next iteration of the loop. It is especially these unnecessary loads and stores that cause additional cycle counts in this experiment.

It may not be surprising that the compiler makes these choices, because the input array *x*[ ] is not specified as a **const** array. Omitting the **const** qualifier from the input

array *x*[ ] implies that the compiler should assume that elements of *x*[ ] may be overwritten during the execution of the function, for example if *y*[ ] points to somewhere in *x*[ ]. Hence, the compiler should assume that elements of *x*[ ] may be overwritten during the each iteration of the inner loop.

An attempt to use the **const** qualifier for both *x*[ ] and *h*[ ] resulted in a compiler error in experiment E1. The compiler evidently realized that loop unrolling and dual MAC was possible, because it generated the following assembly language code for the inner loop:

```
     MACM *AR1-, *AR2+, AC0, AC1  ||  MACM *AR4-,
*AR2, AC1, AC0
```

Unfortunately, this combination of instructions is illegal, as two instructions combined in parallel cannot consume more than six bytes of memory for decoding. Thus the compiler aborts with an error. Therefore experiment E1 was performed without qualifying *x*[ ] with the **const** keyword to avoid this compiler defect.

The compiler understands the index arithmetic used to select elements in *x*[ ], *h*[ ], and *y*[ ] arrays, and translates this into efficient indirect addressing with post-increment and post-decrement operations.

E2: Forward Store

As mentioned, initially a global variable for the summation of *y*[ ] was used. While using global variables, the compiler is obligated to perform an intervening store or load to memory during the execution of the loop.

In Forward Store optimization, [7], store to a global variable *y*[ ] is done by performing the summation in a local variable *accum* and storing the final result to *y*[ ] outside the loop afterwards.

According to [15], 'C54x and some other DSP compilers are not performing Forward Store optimization. Our investigations have shown that the same optimization is absent in the 'C55x compiler, and it is therefore necessary to do it explicitly.

As seen in Figure 5, the compiler understands that the local variable *accum* does not change until the next iteration of the inner loop, and eliminates the unnecessary loads and stores that hampered performance in experiment E1.

Thus leaving only a multiply-accumulate operation inside of the inner loop, the compiler also realizes that loop unrolling enables dual MAC performance in the inner loop. Hence, with a simple forward store optimization, the compiler rewrites the inner loop to a single-instruction dual MAC loop.

Experiments also showed that automatic loop unrolling is not observed when the inner loop is repeated less than 12 times. When unrolling the loop manually in C and applying all the optimizations discussed so far the desirable performance is not obtained either. In both cases the compiler generates a one-to-one translation of the code and does not recognize the computational grouping necessary for dual MAC exploitation and hence the possibility to generate the dual MAC instruction.

C Code

```
void fir(const
         int h[],
   const int x[],
         int y[])
{
int n,m,accum;

n=nSamples-1;
while (n >= 0)
 {
  m = accum = 0;
  while(  m  <  nTaps
)
   {
   accum+=h[m]*x[n-
m];
   m++;
   }
  y[n]=accum;
  n--;
 }
}
```

Assembly Code

```
MOV#((_y+38)&0xffff),AR4
 MOV #39, AR1
 MOV #19, BRC0
  RPTBLOCAL L6-1
            ; loop starts
L3:
  ADD #(_x&0xffff),AR1,AR3
  SUB #1, AR3, AR2
  MOV #(_h & 0xffff), CDP
  MOV #0, AC0
   RPT #15
|| MOV AC0, AC1

            ; loop starts
L4:
MAC  *AR2-,  *CDP+,  AC1 :: MAC
*AR3-, *CDP+, AC0
            ; loop ends
L5:
  MOV AC1, *AR4
  MOV AC0, *AR4(short(#1))
  SUB #2, AR4
  SUB #2, AR1
        ; loop ends
L6:
        return
```

Fig. 5. **Experiment E2 (Forward Store)**

For a number of taps less than 12, the compiler generates a **MACM** instruction. Therefore, in order to generate a dual MAC instruction, the pragma UNROLL has to be applied in this case.

The compiler also arranges housekeeping instructions so that pairs of instructions can be executed in parallel.

E3: Counter 1 (Final version)

In the experiment E3 in Figure 6, the compiler does not take advantage of our attempt to make it explicit to the compiler where the indexes are to be updated.

C Code

```
void fir(const
         int h[],
   const int x[],
         int y[])
{
int n,m,accum;

n=nSamples-1;
while (n >= 0)
 {
  m = accum = 0;
  while(m<nTaps)
   {
accum     +=h[m]*x[n-
m++];
   }
  y[n--]=accum;
 }
}
```

Assembly Code

```
MOV #((_y+38)&0xffff),AR4
 MOV #39, AR1 ;
 MOV #19, BRC0
  RPTBLOCAL L6-1
            ; loop starts
L3:
 ADD #(_x&0xffff),AR1,AR3
 SUB #1, AR3, AR2
 MOV #(_h & 0xffff), CDP
 MOV #0, AC0
   RPT #15
|| MOV AC0, AC1
            ; loop starts
L4:
MAC  *AR2-,  *CDP+,  AC1 :: MAC
*AR3-, *CDP+, AC0
            ; loop ends
L5:
 MOV AC1, *AR4
  SUB #2, AR4
|| MOV AC0,*AR4(short(#1))
  SUB #2, AR1
            ; loop ends
L6:
  return  ; return occurs
```

Fig. 6. **Experiment E3 (Counter 1 (Final version))**

This is not surprising, because the compiler has already translated the index arithmetic into efficient addressing in the preceding experiments, and experiment E3 does not yield lower cycle counts than experiment E2.

E4: Counter 2

A common optimization technique is to simplify array index operations as shown in Experiment E4 in Figure 7.

C Code

```
void fir(const
         int h[],
   const int x[],
         int y[])
{
int n,m,t,nstart;

n=nSamples-1;
while (n >= 0)
 {
  m = accum = 0;
  nstart = n;
  #pragma UNROLL(1);
  while( m < nTaps )
   {
   accum += h[m++] *
        x[nstart--];
   }
  y[n--]=accum;
 }
}
```

Assembly Code

```
MOV #((_y+39)&0xffff),AR4
MOV #((_x+39)&0xffff),AR3
MOV #39, BRC0
 RPTBLOCAL L6-1
        ; loop starts
L3:
 MOV #(_h&0xffff),AR2
 MOV #0, AC0
 RPT #15
        ; loop starts
L4:
MACM *AR3-,*AR2+,AC0,AC0
        ; loop ends
L5:
 ADD #15, AR3
 MOV AC0, *AR4-
        ; loop ends
L6:
   return
        ; return occurs
```

Fig. 7. **Experiment E4 (Counter 2)**

We therefore simplified the indices in the inner loop as follows:

```
m = accum = 0;
nstart = n;
while( m < N )
{
    t += h[ m++ ] * x[ nstart-- ];
}
```

To our surprise, this optimisation apparently confused the compiler, which refuses to use dual-MAC instructions and instead executes the inner loop as a single MAC instruction. Oddly, the compiler also refuses to unroll the loop, even faced with the pragma UNROLL meta-instruction, which should force the compiler to perform loop unrolling.

General observations

The compiler selected invalid parallel MACM instructions when the "const-correct" fir(**const int** *x*[ ], **const int** *h*[ ], **int** *y*[ ]) function was called in the initial version of the C code. We believe that the compiler's attempt to perform a single-instruction repeat loop using a parallel dual-MAC instruction indicates that the compiler is capable of recognizing latent optimisations in the C code, and that the invalid instruction selection may be a simple defect that is unrelated to the compiler's code optimisation algorithms.

The experiments indicate that by emulating a "const-correct" **fir** function by virtue of the forward store optimisation, the compiler generates code with strong performance, using a single-instruction, dual-MAC inner loop.

The cycle count for this particular FIR filter algorithm can be computed according to Eq. 6, where nTaps and nSamples denote the number of filter taps and the number of samples respectively:

$$CC = Init + OuterLoop + InnerLoop =$$

$$= 4 + \left(\frac{nSamples}{2}\right) * 6 + \left(\frac{nSamples}{2}\right) * (3 + nTaps) =$$

$$= 4 + \left(\frac{nSamples}{2}\right) * (9 + nTaps). \quad (6)$$

Cycle counts for BDTi[1] and TI[2] block FIR filter benchmarks are shown in Eq. 7 and 8 respectively. Note, that there will not be perfect correspondence between the assembly code and the C code benchmarks, since the assembly code implementations follow the TI and BDTI Benchmark specification, while the C implementations do not.

BDTi:

$$CC_{BDTi} = 18 + nTaps + \left(\frac{nSamples}{2}\right) * (2 + nTaps). \quad (7)$$

TI:

$$CC_{TI} = 32 + \left(\frac{nSamples}{2}\right) * (4 + nTaps). \quad (8)$$

Results for LMS algorimh

The LMS algorithm consists of two inner loops: a FIR filter loop as described in section IV and a coefficient update loop plus an error calculation statement.

It is observed that presence of the second loop in the algorithm and overwriting the coefficient array prevents compiler from generating dual MAC instructions as should be expected from the results in the preceding section.

Another observation made during experiments with LMS algorithm is the *loop invariant code motion* optimization. Although this optimization is included in compiler optimization package *–o3*, in our LMS algorithm case the compiler only does this optimization partially.

As seen in Figure 8 a), the expression in brackets (*x*[n] - *accum*) does not change inside the loop, therefore could be moved out of it. The value of *mu* is also constant, so the multiplication by *mu* could be also moved out. However, in this case only the subtraction is moved out of the loop.

In Figure 8 b), the redundant multiplication together with subtraction is moved out from the loop manually. In this case the compiler detects the possibility for parallel instruction execution.

Conclusion

Concluding our experiments, we will now attempt to evaluate the results of our approach in a broader perspective.

The algorithm examined is a simple but fundamental example used in many DSP applications. Consequently only preliminary observations for the C55x platform analysis can be made.

[1] BDTI Benchmark (TM) results provided courtesy of Berkeley Design Technology, Inc. (BDTI). Copyright (c) 2002 BDTI.
[2] TI benchmarks are given in [16] and [17].

a) C Code

```
for (j = 0; j < nTaps;
     j++)
{


w[j]+=mu*(x[n]-
      accum)*x[n-j];
}
```

a) Assembly Code

```
ADD #(_x&0xffff),T3,AR4
 MOV SP, AR3
 SUB AC0, *AR2, T2
 RPTBLOCAL L9-1
            ; loop starts


 MPYM *AR4-, T2, AC1
 MOV AC1, T1
 MOV *AR3, AC2
 MACK T1, #33, AC2, AC1
 MOV AC1, *AR3+
            ; loop ends
 SUB #1, T3
 SUB #1, AR2
```

b) C Code

```
error=mu*(x[n]-accum);

for (j = 0; j < nTaps;
     j++)
{


 w[j] += error*x[n-j];
 }
```

b) Assembly Code

```
   SUB AC1, *AR4, AR2
   MOV SP, AR3
    ADD #(_x&0xffff),T2,AR2
|| MOV AR2, HI(AC0)

    MPYK #33, AC0, AC0

   RPTBLOCAL L9-1
|| MOV AC0, T1
            ; loop starts


   MOV *AR3, AC2
   MACM *AR2-, T1, AC2, AC2
   MOV AC2, *AR3+
            ; loop ends
    SUB #1, T2
    SUB #1, AR4
```

Fig. 8. **Loop invariant code motion for LMS algorithm: a - performed by compiler, b - performed manually**

First we noted that, as expected, the compiler was insensitive to alternate C code formulations such as index or pointer specification for array references and for- or while- specification of loops.

Then from the cycle counts reported it was obvious that optimisation levels -o2 and –o3 encompassing the automatic loop unrolling yield the first big leap in performance and that the use of forward store/local variables accounts for the next reduction. This was also to be expected.

Eq. 6 shows that only 4 cycles are used for initialisation and 6 cycles for (each) outer loop. This leaves limited room for improvement, and using this result it should be possible to identify when further optimisation might not be warranted, i.e., to identify the point of diminishing returns.

In future we will present a comparison of our results with published BDTi and TI benchmark results.

The LMS experiment shows that the compiler does not produce consistent results when asked to reproduce optimisations on sub-problems. The cause of this inconsistency remains unknown.

The proposed approach should be compared to other options such as using third-party software or processor specific compiler features. In such a comparison not only code size and cycle count are relevant; development time, cost, and risks as well as the potential for reuse, that is, the constraints imposed by real applications, should also be considered.

It seems fair to conclude that the proposed approach has a potential for shorter development time while maintaining low cycle counts and uncompromising code portability.

Acknowledgement

We want to thank Jeff Bier, a founder and General Manager of BDTi, for supplying information about BDTi benchmark results.

References

1. **Rajan S. P., Fujita M., Sudarsanam A., Malik S.** Development of an optimizing compiler for a Fujitsu fixed-point digital signal processor. Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99). 1999. P.2–6.
2. **Zivojnovic V., Velarde J. M. and Schlager C.** DSPstone: A DSP-oriented benchmarking methodology. Proceedings of ICSPAT'94, October 1994.
3. **Zivojnovic V., Velarde J. M. and Schlager C.** DSPstone: A DSP-oriented benchmarking methodology. Aachen University of Technology. Technical report. August 1994.
4. **Frederiksen A., Christiansen R., Bier J. and Koch P.** An Evaluation of Compiler - Processor Interaction for DSP Applications. Signals, Systems and Computers. 2000. Vol. 2. P 1684-1688.
5. **Hwang Y. and Hwang J.** Efficient code generation for digital signal processors with parallel and pipelined instructions. IEEE Workshop on Signal Processing Systems Design and Implementation. 1997. P.243-252.
6. Texas Instruments. TMS320C55x DSP Mnemonic Instruction Set Reference Guide. April 2001. Spru374e.pdf. P.29-40.
7. Forward Store optimization http://www.nullstone.com/htmls/category/\\fstore.htm
8. **Eyre J.** DSP Benchmarking Methodologies. Embedded Systems, March 1998.
9. Texas Instruments. TMS320C55x Optimizing C/C++ Compiler User's Guide. June 2001. Spru103f.pdf. P.66-67.
10. **Schneck P. B.** A survey of compiler optimization techniques. ACM: Association for Computing Machinery. 1973. P.106–113.
11. Texas Instruments. TMS320C55x DSP Programmer's Guide. July 2001. Spru376a.pdf. P.64-70.
12. Berkeley Design Technology, Inc. web page, www.bdti.com
13. The Embedded Microprocessor Benchmark Consortium web page, www.eembc.org
14. Texas Instruments. TMS320C55x DSP Programmer's Guide. April 2000. Spru376.pdf. P.87-89.
15. **Levy M.** C compilers for DSPs flex their muscles. EDN Access. 1997.
16. Texas Instruments. TMS320C55x DSP Library Programmer's Reference. August 2000. Spru422a.pdf. P.62-65.
17. C5000TM Platform Overview – Power Efficient DSPs: Benchmarks, Texas Instruments web page, http://dspvillage.ti.com/docs/dspvillagehome.jhtml

B. Varnagirytė, A. Žemelis, O. Olsen, P. Koch, O. Wolf, E. Kazanavičius

Praktinis DSP kodo optimizavimo naudojant architektūrinį kompiliatorių būdas

Reziumė

Nagrinėjamas diskretinių signalų procesorių (DSP) C-kodo kompiliatoriaus naudojimo efektyvumas ir jo generuojamo kodo optimizavimas, taip pat kodo generavimo technikos tinkamumas konkrečiai architektūrai. Darbe pateikti įvairiems DSP uždaviniams spręsti atliktų eksperimentų rezultatai.

Pateikta spaudai 2002 06 27

DOI: 10.5755/j01.u.43.2.8119