A STUDY of LOOP UNROLLING for VLIW-BASED DSP PROCESSORS

Suleyman Sair David R. Kaeli Waleed Meleis

Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

Abstract - With the growing popularity of DSPs and their associated applications, cost-effective software development has become a major issue. High-level language compilers are becoming more commonplace in the DSP world. While these compilers can generate correct code for DSP architectures, there remains considerable room for performance improvements. This paper addresses issues related to DSP compilation, focusing specifically on unrolling techniques proposed for VLIW-based DSP architectures.

INTRODUCTION

The increasing complexity of embedded applications has made code generation for embedded processors a challenging problem. Strict requirements on code size and execution performance are difficult to satisfy when using current compiler technology for DSPs. High performance typically translates to handcoding, which places more pressure on development time and costs. This issue has been addressed by many researchers. As discussed in [1],[2],[3],[4], conventional code generation techniques for RISC architectures produce inefficient code for a number of DSPs (e.g., AD2101, M56K, NEC 77016, TMS320C5x). However, more recent DSPs (e.g., TMS320C6x) are superscalar VLIW architectures. Thus, techniques used to address code generation for current VLIW-based pipelines should produce good code schedules for this new generation of DSPs.

This paper addresses loop unrolling based optimizations, targeting DSP applications as compiled and run on DSP processors. The rest of this paper is organized as follows. In Section 2 we review the issues related to loop unrolling. In Section 3 we present a discussion on the Initiation Interval and provide an example on how to compute the unrolling factor. In Section 4 we present results of our unrolling scheme, and compare them to the commercial compiler-generated codes. In Section 5 we conclude, suggesting directions for future work.

LOOP UNROLLING

Loops are localized areas where programs spend most of their execution time. To improve overall performance, one must utilize the hardware as efficiently as possible, overlapping successive iterations of a loop to achieve sufficient instruction-level parallelism. As Lavery and Hwu have pointed out [5], unrolling can achieve this goal through improving the efficiency of software pipelining. Basically, loop unrolling reorganizes and reschedules the loop body. It involves the following steps [6]:

1. replicating the loop body n times (n is generally between 2–8, rarely greater than 8),
2. removing index register increments/decrements from all blocks but the last,
3. multiplying the increment/decrement by n and adjusting offsets,
4. removing loop counter increments/decrements from all blocks but the last, and
5. eliminating branches from all blocks but the last.

As has been shown by Weiss and Smith [6], loop unrolling provides the following benefits:

1. eliminates a significant number of branches,
2. reduces the total number of issued instructions by eliminating unnecessary increments and decrements between adjacent blocks, and
3. allows for a better schedule by increasing the basic block size.

This last benefit may be the most powerful of the three, but it also presents a second-order effect. Since embedded processors have limited memory, expanding the code size by duplicating the loop body may result in poor cache performance. Moreover, the unrolled program may not even fit in the available amount of memory. Besides memory constraints, unrolling introduces additional register pressure on the compiler, which already has to deal with the small number of registers present on most DSPs.

The following resources need to be taken into account when rescheduling a loop using unrolling [5]:

1. execution resources (e.g., registers, pipeline stages, cache, memory and functional units (FUs)), and
2. dependence patterns and associated instruction latencies.

There are a number of techniques to perform unrolling [5],[7],[8],[9]. In this paper we attempt to explore the various options to produce an improved execution of signal processing applications.

THE INITIATION INTERVAL

As in most microprocessor-based unrolling algorithms, we begin by determining the *Minimum Initiation Interval* (*MII*). The Initiation Interval (*II*) is the number of cycles between the start of successive iterations. There are two lower bounds on the *MII* :

1. *Resource Minimum Initiation Interval* (*ResMII*) : This bound is derived from the resource constraints within the loop body. If each iteration of the loop occupies a FU for C cycles, and there are N such units in the architecture, then *ResMII* is given by C/N. The FU with the maximum C/N ratio determines the *ResMII*.

2. *Recurrence Minimum Initiation Interval* (*RecMII*) : This bound is derived from the dependence patterns present in the loop body. Cycles in the dependence graph (DG) determine the *RecMII*. Each arc on the DG has two values. The first represents the minimum number of cycles between the two dependent instructions. The second represents the number of loop iterations separating the dependencies (0 is equivalent to an intra-loop dependency, 1 is a loop-carried dependency).

```
        LD      0X14,B0
L2:     ST      B5,*B4++        [1]
        SUB     B0,1,B0         [2]
        BNZ     B0,L2           [3]
```

Figure 1: Pseudocode for a loop

```
        LDW   .D1   0X14,B0
L2:     SUB   .L2   B0,1,B0
[B0]    B     .S1   L2
        NOP         4
        STW   .D2   B5,*B4++
```

Figure 2: Code generated by the TI compiler

The vertices labeled 1, 2, and 3, in Figure 3, correspond to the three lines of pseudocode in Figure 1. Each edge represents either a data or control dependency. For instance, the edge between node 3 and node 1 has a value (5,1), indicating that there is a control dependence between the branch and the top of the loop, that the distance between these instructions is 5 cycles,

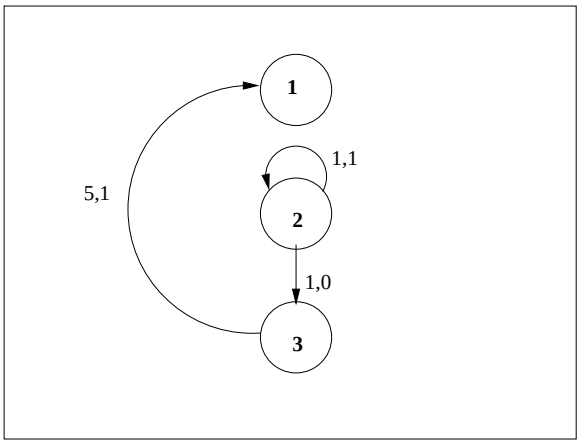

Figure 3: Dependence graph for the given code fragment

and that their dependency is across two iterations of the loop (i.e. loop-carried). For each cycle in the DG, an *MII* value is determined by the C/D ratio, where C is the total execution time of the cycle (the sum of the first number in the edge pairs), and D is the sum of the iteration distances (the sum of the second number in the edge pairs) . For the DG shown in Figure 3, the *RecMII* is 1.

The *MII* is the maximum of *ResMII* and *RecMII*. Once the *MII* is determined, the *optimal unrolling factor* (OUF) can be computed. As the *MII* is a rational number, the OUF is the smallest integer that will make *MII* an integer [5],[9],[10]. For instance, if the *MII* was 3/2, then OUF will be 2. The OUF can be obtained by dividing the denominator of the *MII* by the greatest common divisor of the numerator and the denominator of the *MII*.

For loops whose iteration count is not an exact factor of the OUF, we find the excess number of iterations with a modulo operation with respect to OUF, and add those to the end of the loop.

Iteration counts for some loops are not known until run-time. We can still unroll those loops, provided that we check the loop counter against the OUF to see if we can go into the unrolled loop and provide a *fix-up routine*. When we cannot enter, we will branch to the fix-up routine which contains a single iteration of the original loop body. For the loops whose entry to the loop body is controlled by a boolean condition, we can not perform unrolling easily. Alternatively, profile-based information can be used to unroll those loops, though a more elaborate fix-up routine will be needed to ensure correct execution.

Instruction	*Number of cycles*
Branch	5
Load	4
Multiply	1

Table 1: Delay slots for the TMS320C62xx

EVALUATION

To evaluate the effects of unrolling on signal processing workloads, we compiled and ran five applications on the Texas Instruments C Source Debugger and Simulator Tool, Version 1.00 for Windows 95 and Windows NT. This tool faithfully emulates a TMS320C62xx (C6x), which is a VLIW architecture. We compiled these benchmarks with software pipelining turned on and off using the -o3 option for optimizations and the -mu switch for software pipelining. The C6x has the following features [11]:

1. uses a 200 MHz clock,
2. executes up to eight instructions per cycle,
3. has 32 32-bit general purpose registers,
4. provides eight functional units (6 ALUs, 2 multipliers), and
5. can conditionally execute all instructions.

This last feature gives the C6x the ability to avoid the high cost of resolving branches, and potentially increase parallelism. For example, in the compiler-generated code in Figure 2, the branch instruction (B) is conditionally taken based on the contents of general purpose register B0. The branch is executed if the contents of B0 is not zero [12]. The number of delay slots associated with different types of instructions is provided in Table 1.

Simulation Results

We unrolled the DSPstone kernel benchmarks [3] listed in Table 2 with the algorithm just described. In all the filtering benchmarks, the number of taps used was 16. For *IIR_N_BIQUADS*, the number of sections simulated was 4.

We present performance results obtained from the emulator. Tables 3 and 4 provide results for the TI compiler-generated code. The first column identifies the benchmark. The second column presents the performance of the code in cycles, and the last column lists the code size in bytes. Tables 5 and 6 provide results for our code (which we will refer to as the *OUF* code).

Benchmark	*Description*
Convolution	This benchmark implements the convolution operation as a sum of products without a state update.
FIR	Digital filtering is one of the central applications using DSPs. This benchmark multiplies an array of state variables x by an array of coefficients and accumulates the result in the output variable y.
IIR_N_BIQUAD	This benchmark filters input values through a biquad Infinite Impulse Response filter. The IIR filter consists of N serial, second order subsystems, named biquads, so that the output of each biquad corresponds to the input of next biquad.
LMS (Least Mean Square)	This benchmark implements an adaptive filter. It performs a finite impulse response filtering where the tap coefficients are adapted iteratively.
Matrix1x3	This benchmark multiplies a 3 by 3 matrix with a 1 by 3 vector to generate a 3 by 1 vector.

Table 2: DSP benchmark suite used in this study

Name Of Benchmark	*Execution Time (in CPU cycles)*	*Code Size in bytes*
Convolution	314	3682
FIR	332	3683
Biquad_N_Sections	435	4383
LMS	673	4531
Matrix1x3	147	3317

Table 3: The performance of the TI compiler generated code w/o software pipelining

Name Of Benchmark	*Execution Time (in CPU cycles)*	*Code Size in bytes*
Convolution	128	3906
FIR	134	4151
Biquad_N_Sections	168	5055
LMS	312	5011
Matrix1x3	56	3733

Table 4: The performance of the TI compiler generated code with software pipelining

Name Of Benchmark	*Execution Time (in CPU cycles)*	*Code Size in bytes*
Convolution	210	3737
FIR	216	3921
Biquad_N_Sections	211	4401
LMS	449	4613
Matrix1x3	61	3381

Table 5: The performance of the *OUF* code w/o software pipelining

Name Of Benchmark	*Execution Time (in CPU cycles)*	*Code Size in bytes*
Convolution	107	4173
FIR	125	4583
Biquad_N_Sections	148	5145
LMS	237	5959
Matrix1x3	56	3788

Table 6: The performance of the *OUF* code with software pipelining

The results show that we can significantly improve the performance of DSP kernels using loop unrolling. However this does not reflect the amount of improvement on complete applications. The DSP benchmarks used consist of an initialization section and a main body. We have only been focusing on the most inner loop in the code body.

If we look at the code given in our example, we see it contains four NOP instructions (due to the delay slot associated with the branch). Our implementation of that portion of the code is shown in Figure 4.

```
        LDW   .D1   0X14,B0
L2:     SUB   .L2   B0,5,B0
 [B0]   B     .S1   L2
        STW   .D2   B5,*B4++
        STW   .D2   B5,*B4++
        STW   .D2   B5,*B4++
        STW   .D2   B5,*B4++
        STW   .D2   B5,*B4++
```

Figure 4: Our implementation of filling the delay slot

Referring back to the code in Figure 2, for 20 iterations, the TI compiler-generated code will take $20 * 7 = 140$ cycles (7 is derived from the number

of cycles for one loop iteration). On the other hand, the *OUF* code will only consume $4 * 7 = 28$ cycles. This simple example shows the effect which loop unrolling can have on the execution performance of a DSP application.

In terms of code size, since the benchmarks are small, the expansion in size seems insignificant. However, we believe the significance will grow when we optimize for complete applications. This effect may be critical if special care is not taken when unrolling with an *OUF* of 8 or more.

The effects of software pipelining has been investigated by unrolling the source code. We fed the resulting C code to the TI compiler using the same switches we compiled the original source code with, and obtained the presented results. The LMS benchmark produced the best percentage gain as it was the only benchmark where we could unroll all of its' loops four times. The rest were unrolled with smaller factors.

SUMMARY

We have implemented unrolling techniques proposed for RISC architectures and compared our results with codes compiled with TI s C6x compiler. We have found that we could make significant improvements in the overall performance for the DSP applications studied, in some cases decreasing the runtime by an order of magnitude. Moreover, our results showed that unrolling improves the effect of software pipelining. For future work we will apply our techniques to complete applications, as well include additional benchmarks from UTDSP Suite [13]. We will also implement Ebcioglu's proposed unrolling technique for VLIW architectures [7],[8],[14]. This would enable us to handle multiple basic block loops, which we weren't able to unroll because of the large number of loop carried dependencies..

References

[1] C.H. Gebotys and R.J. Gebotys. Complexities in DSP Software Compilation : Performance, Code Size, Power, Retargetability. *IEEE Computer Magazine*, pages 51–58, December 1997.

[2] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[3] V. Zivojnovic, J.M. Velarde, C. Schlager, and H. Meyr. DSPtone, a DSP-Oriented Benchmarking Methodology - Final Report. Technical report, Aachen University, Germany, August 1994.

[4] C.H. Gebotys. An Efficient Model for DSP Code Generation : Performance, Code Size, Estimated Energy. In *Proceedings of the 10th International Symposium on System Synthesis (ISSS '97)*, September 1997.

[5] L.M. Lavery and W.W. Hwu. Unrolling Based Optimizations for Modulo Scheduling. Technical report, University of Illinois, 1995.

[6] S. Weiss and J.E. Smith. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. *ACM Transactions On Mathematical Software*, pages 223–245, 1990.

[7] M. Lam. Software Pipelining : An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN'88*, pages 318–328, 1988.

[8] R.B. Jones and V.H. Allan. Software Pipelining : An Evaluation of Advanced Pipelining. *ACM Press*, pages 82–92, 1991.

[9] J. Cortadella, F. Sanchez, and R.M. Badia. A Mathematical Formulation of the Loop-Pipelining Problem. Technical Report UPC-DAC-1995-36, Department of Computer Architecture, Universitat Politecnica de Catalunya, Spain, 1995.

[10] J. Cortadella and F. Sanchez. Maximum Throughput Software Pipelining. Technical Report CICYT TIC-95-0419, Department of Computer Architecture, Universitat Politecnica de Catalunya, Spain, 1995.

[11] Texas Instruments Inc. *TMS320C62xx CPU and Instruction Set.* 1997.

[12] Texas Instruments Inc. *TMS320C6x Optimizing C Compiler.* 1997.

[13] C.G. Lee. UTDSP Benchmark Suite. Technical report, University of Toronto, Canada, 1992.

[14] R.B. Jones and V.H. Allan. Software Pipelining : A Comparison and Improvement. In *Proceedings of the International Symposium on Microarchitecture (MICRO-23 '90)*, pages 46–56, 1990.