A Rule-based Optimal Placement of Scaling Shifts in Floating-point to Fixed-point Conversion for a Fixed-point Processor

Sanghyun Park, Doosan Cho, Taesong Kim, and Yunheung Paek

Abstract—**In the past decade, several tools have been developed to automate the floating-point to fixed-point conversion for DSP systems. In the conversion process, a number of scaling shifts are introduced, and they inevitably alter the original code sequence. Recently, we have observed that a compiler can often be adversely affected by this alteration, and consequently fails to generate efficient machine code for its target processor. In this paper, we present an optimization technique that safely migrates scaling shifts to other places within the code so that the compiler can produce better-quality code. We consider our technique to be safe in that it does not introduce new overflows, yet preserving the original SQNR. The experiments on a commercial fixed-point DSP processor exhibit that our technique is effective enough to achieve tangible improvement on code size and speed for a set of benchmarks.**

Index Terms—**Compiler, Conversion, Floating-point, Fixed-point, DSP**

I. INTRODUCTION

Fixed-point processors are generally cheaper than their floating-point counterparts. Thus, most high-volume, low-end DSP systems use fixed-point processors since the priority is low energy and cost. However, dynamic range and precision of a fixed-point processor are often strictly limited [5]. As a result, programming fixed-point processors is usually more painful since programmers must spend much time to maintain proper numeric accuracy and performance with the limited dynamic range and precision. So, the common practice is that programmers first employ floating-point processors to verify their designs and algorithms, and later implement the verified algorithms on fixed-point processors by converting floating-point data types into equivalent fixed-point ones.

As a first step in this floating-point to fixed-point conversion (FFC) process, they must find the dynamic range and precision needs of each variable in the code. Based on their findings, they insert shift operations to scale variables in the code. The integral part of this conversion process is to decide adequate places where to insert these scaling shifts because this decision deeply affects the two key factors, the signal-to-quantization noise ratio (SQNR) and overflow, which determine the numeric accuracy of the resulting fixed-point code. Therefore, in the FFC process, programmers must perform rigorous static analysis or simulation to compute exact run-time value ranges of all the variables, which will be used to obtain the accurate dynamic ranges and precisions for the variables.

As can be expected, processing the whole conversion by hand would be quite a time-consuming and error-prone task. According to empirical studies [1], the manual process accounts for roughly a third of the total implementation time. To relieve programmers from this burdensome task, many researchers have developed various FFC tools such as Autoscaler and FRIDGE [2, 6,

Manuscript received Sep 9, 2006; revised Nov. 9, 2006.
Software Optimizations and Restructuring Group School of Electrical Engineering and Computer Science Seoul National University, #58 San 56-1, Sillim-dong, Kwanak-ku Seoul 151-744, Korea
E-mail : shpark@compiler.snu.ac.kr

4] which automate the FFC process efficiently. However, to the best of our knowledge, all these tools do not fully consider detrimental effects of newly added scaling shifts in the fixed-point code on compiler code generation. Our recent experience reveals that such lack of consideration often hinders the compiler from matching *composite instructions* such as multiply-add(MAC) or dot operations which are very efficient in DSP processors. There is thus an imminent need to eliminate the adverse effect of additional scaling shifts that obfuscates compiler to generate good quality code. In this paper, we present an efficient technique that safely migrates scaling shifts to other places within the code so that the compiler can produce better-quality code. The rest of the paper is organized as follows. In Section 2 we show that conventional FFC has an adverse effect on code generation. In Section 3 we present our optimization technique that transforms IR (intermediate representation) using algebraic transformation. We show the experimental results in Section 4 and conclude the paper in Section 5.

II. Motivation

To illustrate the need of our technique, consider the ordinary floating-point C code segment in Fig. 1(a) which implements a popular DSP filter, called IIR. We used the Autoscaler tool [2] to convert this code into the fixed-point one in Fig. 1(b) where we see that many scaling shifts have been inserted during the conversion. Fig. 1(c) shows the assembly code for the ZSP400 processor [7] generated directly from the code of Fig. 1(b). This compiler output suggests that the compiler failed to utilize several nice operation patterns which can be easily translated into some DSP-specific instructions. In fact, as demonstrated in Fig. 1(d), the compiler should be able to further reduce the code size if it could exploit the ZSP **mac/nmac** instructions.

In this work, to address this issue, we use a method, commonly known as algebraic transformations, which helps the compiler to find a better chance of optimal code generation by restructuring the original DAG IR into a more desirable form. For algebraic transformations, we define a set of rules according to the properties of polynomial algebra, and use them to

```
w(n) = x(n)-ai1*w(n-1) - ai2*w(n-2)
y(n) = bi0*w(n)+ bi1*w(n-1) + bi2*w(n-2)
```

(a) Floating-point code for IIR Filter

```
w(n)=(x(n)–multf(ai1,w(n-1))>>1
        – multf(ai2,w(n-2))>>2)<<3
y(n)=(multf(bi0,w(n))>>1+multf(bi1,w(n-1))>>2
      + multf(bi2,w(n-2)))<<1
```

(b) Fixed-point code for IIR Filter

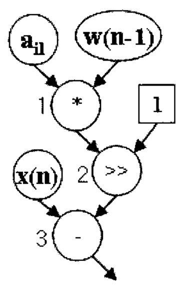
```
mul.a r4, r6
shra r0, 1;
sub r2, r0;
mul.a r5, r7;
shra r0,2;
sub r2, r0;
shla r2, 3;
mul.a r2,r8;
shra r0, 1;
mul.b r6, r9;
shra r2, 2;
add r0, r2;
shra r0, 1;
mac.a r7, r10;
shla r0, 1;
```

(c) Original code for ZSP400

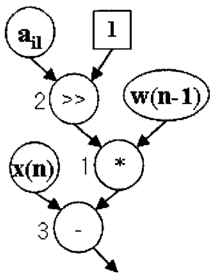
```
shra r4, 1;
nmac.b r4, r6;
shrla r2, 2;
nmac.b r5, r7;
shrla r2, 1;
mul.a r2, r8;
shra r6, 1;
mac.a r6, r9;
shra r0, 2;
mac.a r7, r10;
shla r0, 1;
```

(d) Improved code for ZSP400

Fig. 1. IIR Filter codes.



(a) Original IR



(b) Transformed IR

Fig. 2. DAG IRs for IIR Filter Code.

rewrite the expressions in the DAG. These rules guide our compiler [9] to automatically transform a given IR into some functionally-equivalent, yet more favorable form for code generation. For example, our technique may transform the original DAG IR in Fig. 2(a) into an equivalent form like the one in Fig. 2(b), where the two nodes 1 and 3 now become adjacent to each other. From this new IR, it is trivial for a compiler to produce a **mac** for the two nodes. The code in Fig. 1(d) in fact shows the output assembly code of our compiler generated from transformed IR, attaining 20% reduction in code size.

III. Algebraic Transformation

In this section we discuss how algebraic transformations can be applied to give a DAG IR so as to move the scaling shifts inserted as described in Section 2.

1. Rewriting Rules for Transformation

Algebraic transformations have been used in many domains such as compiler optimization [8] and high-level synthesis [3]. Given an arbitrary DAG, finding its optimal transformation subject to certain conditions is a well-known intractable problem. So in practice, the problem is approximated by a series of local pattern matching problems where a predetermined set of *rewriting rules* are applied subsequently to varied subgraphs of the DAG in order to gradually form an (near-)optimal structure. Note that there can be different rewriting rules for one source pattern as shown in Fig. 3. Although those rules bring the same effect on code generation, they usually have different effects on the SQNR (or precision) and overflow within the output code. Therefore, when we define new rules, we must predict their exact effects and exclude any rules with undesirable effects.

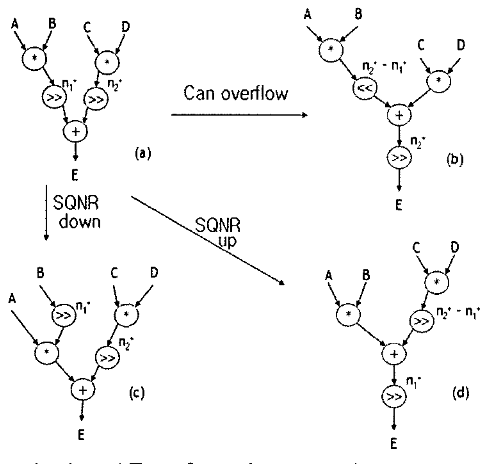


Fig. 3. Rules-based Transformation Example : n_1^+ and n_2^+ are positive integers such that $n_2^+ > n_1^+$.

Fig. 4 shows all the rules defined for our work, each of which contains scaling operations. Given a subject DAG, the complexity of algebraic transformations grows rapidly as the number of rewriting rules increases [8]. The number of rules is exponentially proportional to the size of patterns in each rule. Therefore, as can be seen from Fig. 4, the pattern is restricted to encompass the operators at the distance of at most two from the scaling shift at the center. The rationale for this is that composite instructions are normally generated by the compiler from at most three operations on neighboring nodes in the IR.

As displayed in Fig. 4, we divide the arithmetic operators in a pattern into three classes: additive ⊕, multiplicative ⊗ and scaling shift operators. In the Fig., the symbol ⊙ denotes an arbitrary arithmetic operator including ⊕ and ⊗. We also divide the patterns in the Fig. roughly into three cases, depending on the relative positions of these operators. The first case is when two scaling shifts are adjacent, as shown in rule 1 of Fig. 4 (1). Ordinary shifts for other than scaling cannot always be merged since they are usually used for masking their operands. But, we find that any adjacent scaling shifts can be safely merged without detrimental effects on the SQNR and overflow. So, in our transformations, an expression $\mathbf{B=(A<<n_x)>>n_y}$ would be simplified to $\mathbf{B=A<<(n_x-n_y)}$, according to the rule 1.

The second cases can be found from Fig. 4 (2.1) to (2.4), where a scaling shift is adjacent to an ⊗ operator, intervening between the operator and another one ⊙. If the processor has a composite instruction consisting of ⊙ and ⊗, we may want to move this scaling shift out of this place by the four rules 2.1, 2.2, 2.3 and 2.4, thereby allowing the compiler to generate the composite instruction. Note that rules 2.1, 2.2 and 2.3 contain a right shift, and rule 2.4 contains a left shift. We can see that the two operators ⊙ and ⊗ are neighboring in the target patterns, facilitating code generation of a composite instruction [⊙,⊗]. As an example, the patterns $\mathbf{C=A\otimes(B>>n_x)}$ and $\mathbf{C=(A\otimes B)>>n_x}$ are functionally-equivalent. So the first pattern can be transformed to the second one by rule 2.2, or inversely by rule 2.3. If B is ⊙, then we will apply rule 2.2. If C is ⊙, we will apply rule 2.3. As explained above with Fig. 3, rule 2.2 improves the SQNR while rule 2.3 does the opposite. Lastly, the remaining ten rules in Fig. 4 correspond to the third case where a scaling shift is adjacent to an ⊕ operator and intervenes between ⊕ and ⊙. We can easily prove by well-known algebraic properties that all rules perform valid transformations between functionally-equivalent expression DAGs.

2. Priority-based Rule Application

In this subsection, we discuss how we apply the rules in Fig. 4 to solve a local pattern matching problem in our transformations. We use a conventional DAG pattern matching algorithm for our problem [8]. To reduce the complexity of the pattern matching, we prioritize all the rules in the following sequence.

The priority is given according to the two metrics: precision and computation. The precision is evaluated by the values of SQNR, and the computation is by the number of nodes in the pattern. When two rules are simultaneously applicable, the one with the higher priority will be used to transform the subject DAG. For example, in Fig. 2(a), nodes 1 and 3 can be combined and translated to a mac instruction if node 2 is removed from between the two nodes via both rules 2.3 and 3.8. However, in this case, we prefer 3.8 since it has a higher priority over 2.3 as shown in Table 1.

Our pattern matching is priority-based peephole optimization. This means that a rule is applied only when its target pattern is found to be *useful* for the code generation on our fixed-point processor. The usefulness is determined by either machine-independent or machine-dependent properties. Each rule is iteratively applied to the subject DAG until no more rules are applicable.

Table 1. Priorities of Rules in Fig. 4.

priority	rules	changes in precision and computation
1	3.6	precision ↑ , computation ↓
2	1	computation ↓
3	2.2, 3.7, 3.8	precision ↑
4	2.1, 3.2, 3.3	no change
5	2.3, 2.4, 3.1, 3.4, 3.5	precision ↓

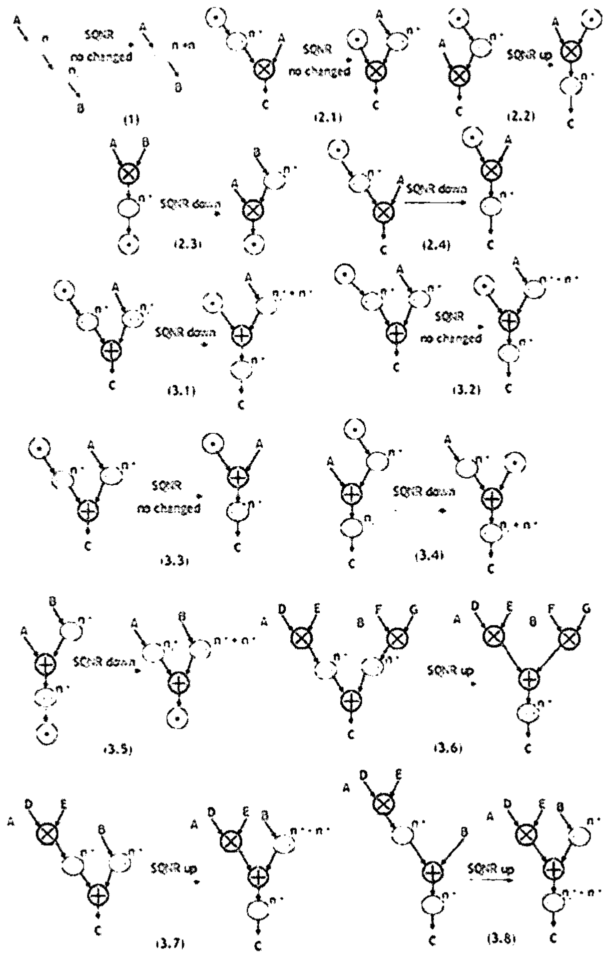


Fig. 4. Rewriting Rules.

IV. Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of the proposed technique, which is implemented for ZSP400 compiler backend. The experimental input is a set of floating point code from DSPstone. In order to isolate the impacts on performance and code size purely from our techniques, two sets of executables for the ZSP400 processor are produced for the benchmark codes; **ORGIN**: floating point to fixed point conversion with original Autoscaler and **TRANS**: floating point to fixed point conversion with Autoscaler included the algebraic transformation. With these two sets of executables, we measured (1) cycle counts with simulator and (2) code size with utility tool. The performance improvements and code size reduction due to proposed technique are measured in percentage, using the formula ((ORGIN-TRANS)/ORGIN)*100.

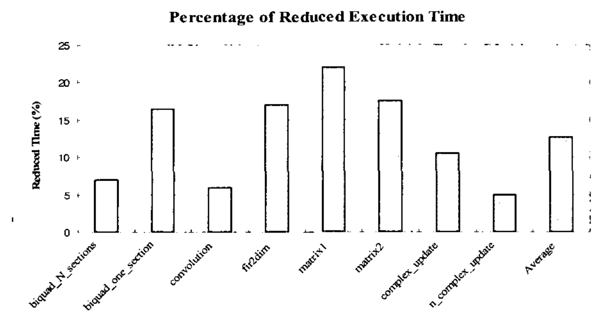


Fig. 5. Reduced Execution Time.

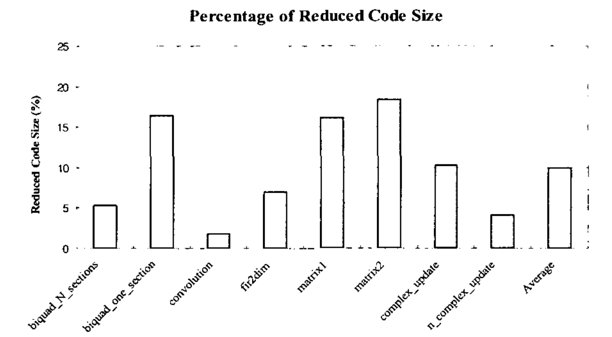


Fig. 6. Reduced Code Size.

Fig. 5 reports the performance improvements, which is based on the proposed technique. The graph shows that there is up to 21.5% and average 12.7% performance improvement by using our technique.

Fig. 6 demonstrates that we can reduce the code size by helping the compiler to select DSP-specific instructions. The graph show that there is up to 16.7% and average 10% code size reduction. by using our technique.

V. Conclusions

For DSP systems, there have been many techniques to convert the floating-point to fixed-point. However, existing techniques do not consider the side effect of scaling shifts on code generation. Such ignorance often raises a critical performance issue on fixed-point DSP processors because these processors mostly aim to gain the performance via DSP-specific CISC instructions. In this paper, we propose a rule-based algebraic transformation to alleviate the side effect of scaling shifts. As a special case, we applied our transformation technique for ZSP400 processor using priority-based algorithm. We observed substantial improvement on code size and execution time.

Acknowledgments

This work was partially funded by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), KRF contract D00191, MIC under Grant A1100-0501-0004 and IT R&D Project, the Korea Ministry of Science and Technology(MoST) under Grant M103BY010004-05B2501-00411, Nano IP/SoC promotion group of Seoul R&BD Program in 2006.

References

[1] T. Grötker, E. Multhaup, and O.Mauss, "Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis," *ICSPAT'96, Boston*, October 1996.

[2] S. Kim, K. Kum, and S. Wonyong, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Transactions on Circuits and Systems II*, 45(11), November 1998.

[3] A. Chandrakasan, et. al., "Optimizing Power Using Transformations," *IEEE Transactions on CAD,* Vol. 14, No. 1, 12–31, 1995.

[4] C. Shi and R. Brodersen, "Automated Fixed-point Data-type Optimization Tool for Signal Processing and Communication Systems," *In Design Automation Conference*, 2000.

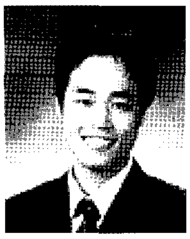
[5] P. Lapsely, J. Bier, A. Shoham and E. Lee, "DSP Processor Fundamentals: Architectures and Features," *IEEE Press* 1997.

[6] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design And Simulation Environment," *Design, Automation and Test in Europe*, 1998.

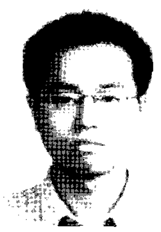
[7] ZSP 400 Digital Signal Processor Technical Manual, http://www.zsp.com.

[8] S. Muchinick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997.

[9] M. Ahn and Y. Paek, "A New ADL-based Compiler for Embedded Processor Design," *Technical Report*, SO&R Research Group, Seoul National University, 2005.


Sanghyun Park received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 2004. He is in Ph.D course in the same university. He was a Visiting Researcher for six months at University California, Irvine, in 2005. His current interestes lie in the interface between the compiler and architecture, and low power design methodology, especially for leakage power reduction.

Doosan Cho received the B.S. degree in digital information engineering from Hankuk University of Foregin Studies, Korea, in 2001, M.S. degree in electrical engineering from Korea University in 2003. Since 2003, he has been work on Software Optimization and Restructuring Lab. in Seoul National University, as an Ph.D student. His current research interests are design space exploration of memory system for application specific domain processors and retargetable optimizing compiler.


Taesong Kim received the B.S. degree in information and computing science from Ease China Normal University, Shanghai, China, in 2003, M.S. degree in electrical engineering from Seoul National University in 2006. Since 2006, he has been work on PULSUS Technology Inc. His current research interests are audio processing and fixed point programming.

Yunheung Paek earned a B.S. and an M.S. in Computer Engineering from Seoul National University. Then, he graduated from the Korea 3rd Military Academy to complete 6-month's mandatory military service. He gained a national scholarship from the Ministry of Education and entered the University of Illinois at Urbana-Champaign (UIUC) where he received a Ph.D. degree in Computer Science. In the spring of 2003, he joined SNU as an associate professor in the School of Electrical Engineering. Before he came to SNU, he had been in the Department of Electrical Engineering at Korea Advanced Institute of Science and Technology (KAIST) as an associate professor for one year and an assistant professor for two and a half years. His current research interests are embedded software, embedded system development tools, retargetabel compiler, and MPSoC.