20-2-R01

Retargetable Compilation Technique for Optimal Placement of Scaling Shifts in a Fixed-point Processor

Sanghyun Park, Minwook Ahn, Doosan Cho, Jonghee Yoon, Yunheung Paek
Software Optimizations and Restructuring Group
School of Electrical Engineering and Computer Science
Seoul National University
(shpark; mwahn; dscho; jhyoon)@optimizer.snu.ac.kr ypaek@snu.ac.kr

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, they first determine the integer/fractional word lengths for each fixed-point variable, and attempt to optimize the SQNR of the fixed-point code while precluding overflows. In this attempt, a number of scaling shifts need to be inserted into the code, and inevitably they alter the original code sequence. Recently, we have observed that a compiler can often be adversely affected by this alteration of the source code, and consequently fails to generate efficient machine code for its target processor. In this paper, we discuss how we circumvent this problem with a simple peephole optimization technique that safely migrates scaling shifts to other places within the code so that the compiler can have a higher chance to produce better code. We consider our technique to be safe in that it does not introduce new overflows, yet preserving (sometimes even improving) the original SQNR. We implemented this technique on our retargetable compiler, Soargen. The experimental results 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.

1. 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 [10]. 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.

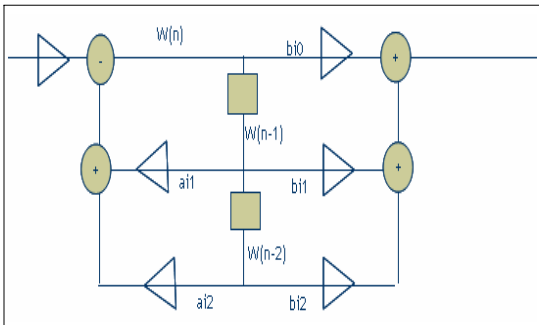
As a first step in this floating-point to fixed-point conversion (FFC) process, they must find the dynamic range and precision needs of each variable in the code. Based on their findings, they insert shift operations to scale variables in the code. The integral part of this conversion process is to decide 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 [3], 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 [4][6][8] 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 result in substantial degradation of the quality of the output machine code.

2. Motivation

To illustrate the need of our technique, consider the ordinary floating-point C code segment in Figure 1(b) which implements a popular DSP filter, called IIR, displayed in Figure 1(a). We used the Autoscaler tool [4] to convert this code into the fixed-point one in Figure 1(c) where we see that many scaling shifts have been inserted during the conversion. Figure 1 (d) shows the assembly code for the ZSP400 DSP processor [11] generated directly from the code of Figure 1 (c). As can be noticed from Figure 1 (a) and (b), the IIR filter originally contains several nice operation patterns which should be easily translated by the compiler into some DSP-specific instructions (e.g., multiply - accumulate and dot-product). However, the compiled output in Figure 1 (d) suggests that the compiler failed to utilize those instructions when it compiled the code of Figure 1 (c). Actually as demonstrated in Figure 1 (e), the compiler should be able to further reduce the code size if it could exploit the ZSP mac/nmac instructions. In this example, the main cause that hinders the efficient code generation is the scaling shifts inserted between the add and multiply operations in Figure 1 (c).

To explain this more clearly, consider Figure 1 (f) where the code of Figure 1 (c) is represented in a DAG, the common intermediate representation (IR) form adopted by many compilers. The IR in the figure has been automatically constructed from



(a) IIR Biquad N Sections Filter

```
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)
```

(b) 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
```

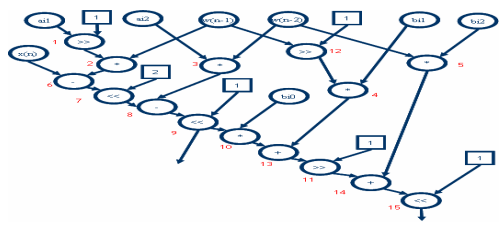
(c) Fixed-point code for IIR Filter

```
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,
```

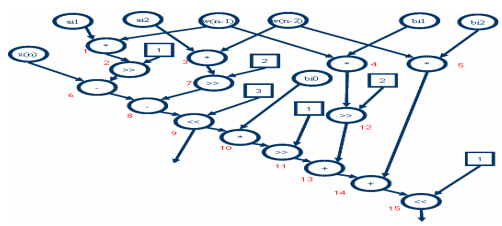
(d) Original code for ZSP400

```
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;
```

(e) Improved code for ZSP400



(f) Original IR



(g) Algerabraic Transformed IR

Figure 1

the fixed-point code by our compiler. From this IR, the compiler may recognize a multiply operation in node 5 immediately followed by an add operation in node 14, so it can translate them together to a mac in the assembly. To the contrary, in the case of the multiply-add pair in nodes 10 and 13, it is not straightforward for a compiler to generate a mac because the shift operation in node 11intervenes between the two operations. In consequence, a compiler would translate the pair into two separate multiply and add instructions along with a shift instruction in-between. As seen from Figure 1 (b), this shift operation was in fact not part of the original IIR filter code, but later inserted for scaling between the multiply and add operations during the FFC process. From our recent experience with several FFC techniques, we have learned that when they insert a scaling shift between two fixed-point operations, they normally ignore whether their compilers can translate the two operations later as

part of a single efficient machine instruction. 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, each of which is typically a composite instruction that encodes multiple operations in a single word [10].

In this paper, we discuss how we complement existing FFC techniques through algebraic transformations to facilitate better code generation. And we present how the rules for algebraic transformation are easily described in Architecture Description Language (ADL), and how our retargetable compiler use them to generate efficient code. For this, we start our discussion with the description of a typical FFC process in Section 3. Then in Section 4, we describe our optimization technique that transforms IR (intermediate representation) using algebraic transformation. We introduce our retargetable compiler, Soargen[14], and how the rules for algebraic transformation are described in SoadDL, which is our ADL, in Section 5. Section 6 shows the experimental results and we conclude the paper in Section 7.

3. Floating-point to Fixed-point Conversion

Typically, a fixed-point data format D consists of three fields of bits: a sign bit, integer bits and fractional bits. The integer word length (IWL) and fractional word length(FWL) represent the number of integer bits and that of fractional bits, respectively [1]. The word length (WL) of D can be defined as 1+IWL+FWL. As an example consider a variable x in Figure 2, which stores a binary value 01010110 in an 8-bit data format with IWL = 3 and FWL = 4. It represents a positive binary number 101.011.So its value is interpreted as 5.375. Similarly, the value of y in the example is interpreted as 1.5625.

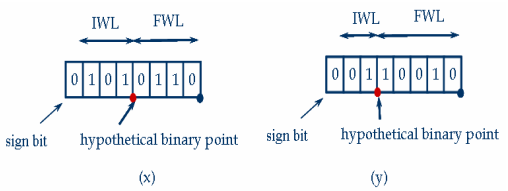


Figure 2. an example for fixed point data format

FFC techniques are elaborated to maximize the SQNR of the application while preventing new overflows from being introduced through the conversion process. The SQNR may be improved by minimizing quantization error, which is the numeric error occurred when a value requiring a data format with longer word length is stored to a shorter word. For a fixed-point format, the precision of the format is identical to its WL since the amount of quantization error is inversely proportional to the WL [10]. In theory, the longer WL the format has, the higher precision we have. But in practice, the WL is limited by hardware constraints. In fixed-point DSP processors, it is typically 16 bits for float-type formats and 32 bits for double-type ones. Thus, many FFC techniques employ a simple heuristic that assigns 16-bit integers for float-type variables and 32-bit integers for double-type variables.

Once the WL of a fixed-point data format is determined for each variable v, the IWL and FWL in the format are to be carefully selected to prevent overflows. This decision is contingent on the maximum value |vmax| that v can have at run time. Clearly, the IWL must be no less than $\lceil \log_2 |vmax| \rceil$ to avoid overflows. To find |vmax|, the

run-time value ranges of all variables in the code must be evaluated during the FFC process. For the evaluation, there have been two approaches [2]. In the first one, the value range of a variable is estimated from its statistical parameters obtained with a floating-point code simulation. One advantage of this approach is that it can obtain an accurate estimation of the range for specific signal input patterns. Another advantage is that it ensures a low probability of overflow for signal with the same input patterns. However, for different input patterns, the estimated results may be incorrect. The other approach [6][7] uses interval analysis [5] to estimate the value analytically. The estimated results are conservative so they are always safe and ensure no overflow. But for some cases, this approach may take too conservative stands to have the useful range information for effective FFC.

Using the value range of each variable v, we can identify |vmax| from which we determine IWLv and FWLv for v. For IWLv |vmax|, we can guarantee no overflow of v. Notice that particularly when IWLv = |vmax|, the precision of v is maximized. However, if we choose a naïve policy that uniformly assigns all variables their maximum values for their IWLs, we would have many different variables with many different IWLs. Previous studies indicate that this policy generally results in many scaling shifts inserted in the final code. Therefore, often in practice, a heuristic is additionally applied to reduce the number of scaling shifts by assigning the same IWL to the two variables with different maximum values, despite some precision loss.

Table 1 displays fixed-point arithmetic rules which direct when and where to insert scaling shifts inside the fixed-point code during the FFC process. In the table, Iv denotes the IWL of a fixed-point variable v. To briefly explain the rules, suppose that we have Ix>Iy for two variables x and y. According to the rules, for assignment x=y, we should perform y>>(Ix-Iy) to align the radix point of y to that of x before y is assigned to x. Likewise, if Ix<Iy, we should perform y << (Ix-Iy) before the assignment. In reality, floating-point arithmetic operations are either additive or multiplicative.

We use the notation to denote the additive operations and the notation the multiplicative operations. As shown in Table 1, an additive floating-point operation z=xy can be converted into either of three fixed-point operation patterns containing scaling shifts. Note hereby that two scaling shifts are always added in any situation.

	floating point	fixed-point			IWL
		Ix>Iy, Iz	Iy>Ix, Iz	Iz>Ix,Iy	
assign	x = y	x=y>>(Ix-Iy)	x=y<<(Iy-Ix)		No change in Ix
add/sub	z = xy	z = (x(y>>(Ix-Iy)))<<(Ix-Iz)	z = ((x>>(Iy-Ix))y)<<(Iy-Iz)	z = (x>>(Iz-Ix))(y>>(Iz-Iy))	Iz = max (Ix,Iy,Iz)
mult	z = xy	multf(x,y)			Iz = Ix +Iy+1

Table 1. fixed point arithmetic rules.

Fixed-point processors commonly provide dedicated functions for fixed-point multiplication as well as ordinary integer multiplication. This is because integer multiplication stores the lower half of the product while fixed-point multiplication needs to access the upper half [1]. For instance, the ZSP fixed-point processor supports two intrinsics: N_mul and N_extract. The first one performs 16-bit fixed-point multiplication and returns the result in 32 bits, and the second returns the upper half of the 32-bit result. In our work, we define a C function multf (see Table 1) for fixed-point multiplication on our target processor. The function can be implemented on the ZSP fixed-point processor as follows.

```
inline long multf(int a, long b){
long z;  // 32 bits
int x,y;  // 16 bits
x = a;
N_extract(y,b);
N_mul(z,x,y);
return z;
}
```

Notice from Table 1 that the IWL for the product of two variables is the sum of their IWLs with an extra 1-bit extension. Using the rules in the table, the code in Figure 1 (b) has been converted to the one in Figure 1 (c). Based on the range analysis, the IWLs for all variables in Figure 1 (b) are estimated as below.

```
Iw(n) = 2; Ix(n) = 5; Iai1 = 1; Iai2 = 0;
Ibi0 = 0; Ibi1 = -1; Ibi2 = 1; Iy(n) = 3.
```

After substituting a multf for each floating-point multiplication, we will have for the first line of Figure 1 (b),

w(n)=x(n)-multf(ai1,w(n-1))-multf(ai2,w(n-2))

Accoring to the rules, the IWLs of the two multfs would evaluate to 4 and 3, respectively. Using these IWLs, we insert scaling shifts into the code as guided by the fixed-point arithmetic rules, and consequently produce the fixed-point code on the first line of Figure 1 (c).

4. Algebraic Transformation

In this section we discuss how algebraic transformations can be applied to a give DAG IR so as to move the scaling shifts inserted as described in Section 3.

4.1 Rewriting Rules for Transformations

Algebraic transformations have been used in many domains such as compiler optimizations [12] and high-level synthesis [13]. 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. A rewriting rule, pspt, consists of a pair of patterns ps and pt, which we call the source pattern and target pattern, respectively. When ps matches a subgraph of the subject DAG, the rule is applied by substituting pt for the subgraph in the DAG. Figure 3 lists three rewriting rules with the same source pattern p0 and its three functionally-equivalent target patterns (p1, p2, p3); that is, p0→p1, p0→p2 and p0→p3.

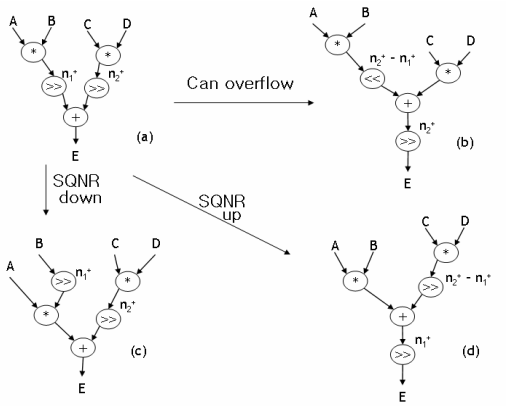


Figure 3. Rule based algebraic transformation

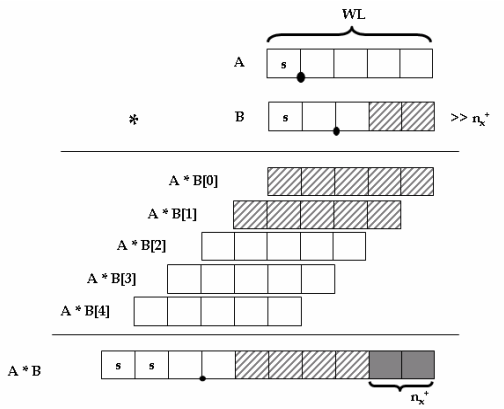


Figure 4 SQNR analysis

example where the operands of scaling shifts n1+ and n2+ are positive integers such that without loss of generality, n2+ > n1+.

Although different 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 bad effects. For instance, all three rules in Figure 3 basically lead the original DAG to the forms that a compiler finds an operation pattern for the mac instruction from. However, notice that unlike the other target patterns, the pattern p1 makes the code more vulnerable to fixed-point overflows than the source pattern p0. Thus, we will disregard the rule p0→p1 inour transformations.

As for p0→p2, we find that the SQNR is degraded if the rule is applied. To explain this, consider Figure 4 where we multiply two 5-bit integers A and B. The main difference between p0 and p2 is whether the right shift by nx+ bits is applied before or after the multiplication. According to Figure 4, when nx+ = 2, the result of (A*B)>>nx+ would contain error only at the least significant 2 bits. However, in the case of A*(B>>nx+), the error would contaminate the result up to the last 6 bits. As a rule of thumb, if we lose 1-bit information in a fixed-point format, the SQNR is degraded by 6 dB. Therefore, in this case, the SQNR will be degraded roughly by 36 dB. Unlike overflow, the SQNR is not a critical factor that determines whether a rule is included or not. In our work, this rule still will be considered for transformation unless the deterioration of the SQNR by 36 dB is beyond the allowable limits which have been predetermined by the programmer.

In case of p0→p3, we see that the original SQNR is improved since the product A*B is used immediately (without right shifts in-between) by the subsequent add operator, thereby preserving the data at the least significant n1+ bits. Also, there will be no overflow even if the product is directly given to the add operator without scaling down. This is because it has two sign bits at the most significant bits, as shown in Figure 4.

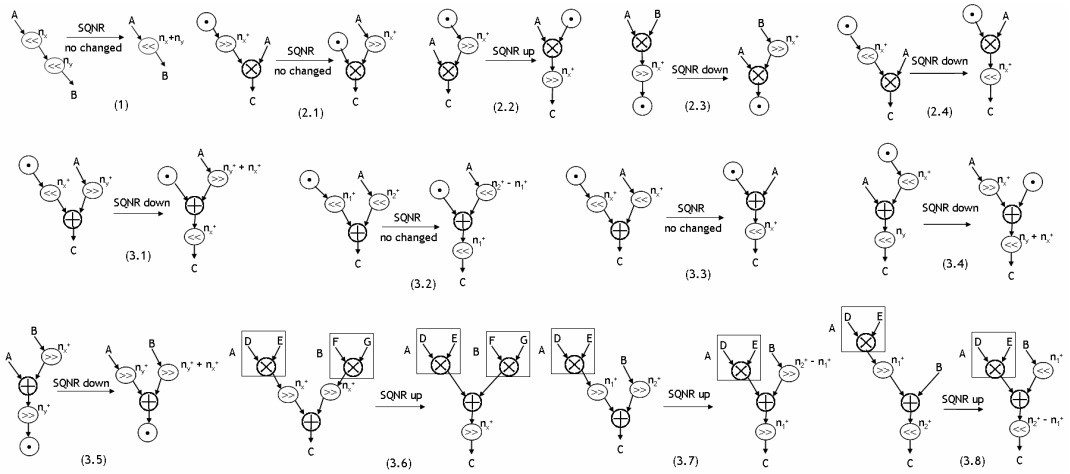


Figure 5. Rewriting Rules

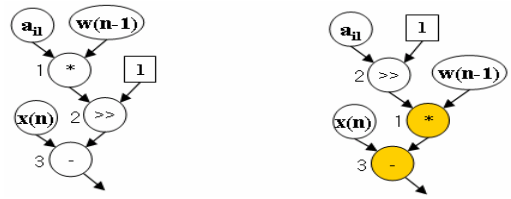
Figure 5 shows all the rules defined for our work, each of which contains scaling operations. These rules were built according to the arithmetic rules in Table 1. Given a subject DAG, the complexity of algebraic transformations grows rapidly as the number of rewriting rules increases [12]. The number of rules is the exponentially proportional to the size of patterns in each rule. Therefore, as can be seen from Figure 5, 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 Figure 5, we divide the arithmetic operators in a pattern into three classes: additive , multiplicative and scaling shift operators. In the figure, the symbol ⊙ denotes an arbitrary arithmetic operator including and . We also divide the patterns in the figure 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 Figure 5 (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

B=(A<<nx)>>ny would be simplified to
B=A<<(nx-ny), according to the rule 1.

The second cases can be found from Figure 5 (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 rules2.1, 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 C=A(B>>nx) and C=(AB)>>nx 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 Figure 4, rule 2.2 improves the SQNR while rule 2.3 does the opposite. Lastly, the remaining ten rules in Figure 5 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.



(a) Original IR (b) Transformed IR

Figure 6. DAG IRs for IIR Filter Code

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 ↓

Table 2. Priorities of Rules in Figure 5

4.2 Priority-based Rule Application

In this subsection, we discuss how we apply the rules in Figure 5 to solve a local pattern matching problem in our transformations. We use a conventional DAG pattern matching algorithm for our problem [12]. 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 Figure 6(a), nodes 1 and 3 can be combined and translated to a mac instruction if node 2 is removed from 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 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 2.

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-independent properties. Each rule is iteratively applied to the subject DAG until no more rules are applicable.

5. ADL-based Compilation Framework

In this section, we first discuss the overall structure of our retargetable compiler, and then describe our ADL with examples to demonstrate how a given ISA is described in this language and the description is used to target the compiler at the ISA. Finally, we

show that the rules for algebraic transformation are easily described in our ADL so that our compiler can recognize the rules and use it to select and generate instructions.

5.1 Overview of the compiler

Figure 7 shows our compiler infrastructure where retargetability can be achieved by enabling the users to describe their target architectures in our ADL. The ADL characterizes an architecture by specifying its structural and behavioral information. Although it is still being extended at present, structural information in the current implementation only describes register and memory architecture. Behavioral information contains a set of machine instructions and addressing modes.

As can be seen in Figure 7, the compiler is implemented with several C++ modules such as

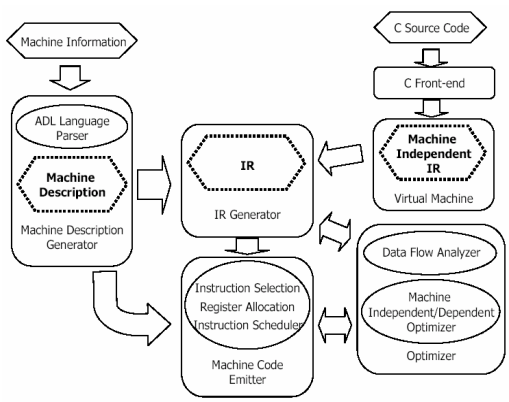


Figure 7. Our ADL-based Compiler Infrastructure

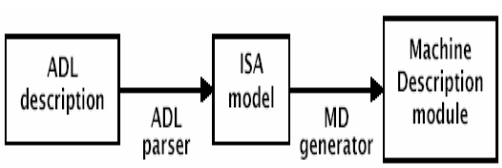


Figure 8. ADL Translation Process for the Code Generation

MachineDescription (MD), IRGen, VirtualMachine (VM), CodeGen, GlobalRegAllocator and Optimizer. The MD module contains a collection of C++ routines that carry all the machine specific information necessary for the compiler. As shown in Figure 8, from the ADL description for an architecture, an MD module is automatically generated and given as input to the CodeGen module which uses the module as a set of machine instruction templates in the phases of instruction selection, register allocation and instruction scheduling.

The ISA model is a group of C++ data structures all constituting an ISA template that will be used to build MD routines. It consists of two major components, resource and operation. The resource component represents storage elements such as registers and memory. The operation component abstracts the ISA of the target machine. Each address mode and instruction description is converted into an instruction template in operation of the ISA model. Among the attribute of an instruction template, the action template represents register transfer level behavior directly. The action template is a list of tree-shaped register transfers and the tree shaped templates can be directly used in the CodeGen module for instruction selection.

The MD generator builds a machine description module from an ISA model. Because our compiler needs various parameters for efficient code generation, the MD generator analyzes the ISA model and extracts necessary information such as register classes or register transfer graphs. The VM module provides a generic interface between the C front-end and our compiler. The virtual machine is an imaginary

machine with virtual assembly as its ISA. The virtual assembly is a simultaneous composition of a list of register transfer expressions (RTEs), each of which corresponds to a single instruction of the form (set lvalue rvalue) where the rvalue expression is evaluated and stored to the lvalue location. Operators in the expression are unary or binary depending on their types, and operands can be either symbolic registers or memory locations. The following shows an example of virtual assembly code:

```
L8:
  (set (SI: r2) (SI: 12(fp)))
  (set (SI: r3) (SI: 0(r2)))
  (set (SI: r4) (SI: 4(fp)))
  (set (SI: r2) (SI: 0(r4)))
  (set (SI: r2) (mult:SI (SI: r3) (SI: r2)))
  (set (SI: r3) (SI: 16(fp)))
  (set (SI: r3) (ss_plus:SI (SI: r3) (SI: r2)))
  (set (SI: 16(fp)) (SI: r3))
```

The virtual assembly is very intuitive. For instance, the first line means ‘load from memory located at fp + 12 to register r2’. All possible machine independent optimizations are performed by the front-end on virtual assembly. When the user compiles application code, virtual assembly code is first produced, and then converted to a graph-structural intermediate representation (IR) through common sub-expression elimination and control flow analysis. Our IR has a hierarchical graph structure. Each basic block node is a forest containing trees or DAGs, each of which represents a set of interdependent RTEs. Several basic block nodes in the same function form a control flow graph (CFG) to represent a function node. Finally, all function

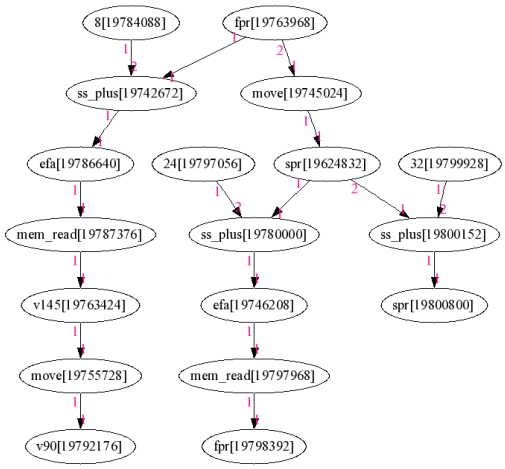


Figure 9. Visualization of the IR for the Kernel Code of convolution

nodes in a program forms a call graph representing the whole program. To visualize the entire hierarchical structure, we developed a visualization tool, called GraphViz. As demonstrated in Figure 9, this tool has been greatly helpful for us to analyze source code and debug our compiler modules when they are targeted to a new processor. The IR glues all compiler modules through a uniform interface. For instance, data flow analysis techniques such as reaching definition and live range analysis in the Optimizer module are performed on the code in the IR.

A DAG containing RTEs at the lowest level of the IR hierarchy is called an Expression DAG (EDAG). The EDAG represents data dependency between operators and values that the operators produces and consumes. In this sense, the nodes in an EDAG can be largely classified into two types: operator and value nodes. The value nodes are further broken down into four - that is, symbolic variable, memory location, effective address and constant. When the application code is transformed to

an IR, the CodeGen and GlobalRegAllocator modules sequentially take the IR and generate the target code using the routines in the MD module.

5.2 ADL for Machine Description

As stated earlier, the main purpose of an ADL is to provide a formal method to describe a target processor as is necessary to verify the completeness and correctness of the ISA. To attain this purpose, our ADL has been rigorously built on the formal definition of ISADesc.

Definition 1. ISADesc=<IS,AM,ST,RIA,RAS>, where
IS : a hierarchical structured set for the instructions of target architecture
AM : a set of addressing modes of target architecture
ST : a set of the storages of target architecture
RIA $\subseteq$ IS X AM^n, where n > 0
RAS $\subseteq$ AM X ST^n, where n > 0.

The primitive section defines primitive operations and types. Each primitive operation stands for an atomic behavior of the target machine. For instance, primitives ss_minus and div stand for subtraction and division operations in hardware. Type information is also represented in the primitive section. For instance, the prefix ss_here represents a signed single precision type. The storage section gives abstract resource structure of the target machine such as memory and registers. Each storage element has two fields: number and mode. Figure 10 shows an example of primitive and storage descriptions.

```
primitive {
   type { qi 8; hi 16; si 32; byte 8; halfword 16; word 32; }
   operation {
      ss_plus; ss_minus; mult; div; ... mem_write; jump;
    }
}
storage {
   memory qi dmem {
        latency 1 @[0x0000..0x9999];
   }
   register si reg[16];
   programCounter r15 PC;
   stackPointer r13 SPR;
}
```

Figure 10. Primitive Operations and Storage declared in our ADL

The address mode and instruction sections describe the machine instructions and addressing modes in the target ISA. In ISADesc, both instructions and addressing modes have three fields: name, action and syntax. They describe abstract behavioral level actions of the target processor. They specify the instruction semantics explicitly and hide the hardware details. As an example, Figure 11 presents an add-shift-left instruction described in our ADL.

To effectively support a top-down design methodology, each description in these sections are hierarchically defined; that is, each description can include several lower level descriptions. Its hierarchical property makes it easy to manage the ISA, and allows us to independently describe instructions, addressing modes and storage, thereby maximizing reusability of the architecture description. At the bottom of this hierarchical structure lies primitives and storage elements defined in the primitive and storage sections. They play role of basic building blocks for the action field in addressing mode and instruction descriptions. This is an example description of displacement addressing mode defined by the register storage type and the primitive ss_plus.

action { efa = ss_plus(Rd, imm5); }

The address mode section defines how to access hardware resource like memory or registers. When we need a memory reference for a load/store instruction, the address mode can be used to represent the effective address for the access. Instructions are described in a similar way. The following shows a multiply and accumulate instruction.

action { rd = mult(ss_plus(rs, rm)); }

When an instruction is described, its operands are usually of the register or memory type directly designated by the storage section. However, the user defines more complex addressing modes and uses them as the operands in an instruction description. For instance, in Figure 11, addressing mode dataAddrMode1 is defined in the address mode section and used as an operand in the description for instruction addsl1.

```
addressModeSet   dataAddrMode1 : dataAddressMode {
     regAddrMode1;     imm4AddrMode;
}
addressModeSet   dataAddrMode2 : dataAddressMode {
     regAddrMode2;     imm8AddrMode;
}
addressMode   regAddrMode1 : dataAddrMode1 {
     reg r3;     action {  r3;  }     syntax {  r3;  }   cost(0);
}
addressMode   imm4AddrMode : dataAddrMode1 {
     integer(4) int4;     action  {  int4; }      syntax  {  int4; }
cost(0);
}
addressMode   regAddrMode2 : dataAddrMode2 {
     reg r3;     action {  r3;  }     syntax {  r3;  }   cost(0);
}
addressMode   imm8AddrMode : dataAddrMode2 {
     integer(8) int8;      action  {   int8; }     syntax  {  int8; }
cost(0)
}
instruction   addsl1: MultipleOps {
        reg  r1;
        dataAddrMode1 opn1;
        dataAddrMode2 opn2;
        action {   r1 = ashift(ss_plus(r1, opn1), opn2));   }
        syntax {   " addsl "::r1::", "::opn1::", "::opn2;   }
        cost(1);
}
instruction   addsl2 : MultipleOps {
        reg  r1, r2, r3;
        integer(8)  imm8;
        action {   r1 = ashift(ss_plus(r2, r3), imm8));   }
        syntax {   " addsl "::r1::", "::r2::", "::r3::", "::imm8;   }
        cost(1);
}
```

Figure 11. Our ADL Description of Two add-shift Instructions

5.3 Specifying Rule in ADL

The rewriting rules described in Section 4.1 can be specified in our ADL, enabling our compiler to perform the transformation before selecting instructions. Thanks to its retargetability, we can make rules for any target architecture. In this section, we describe how we can describe the rules in ADL, and how the compiler exploits this information during instruction selection phase. We retargeted our compiler to ZSP400 processor, and described one simple pattern for MAC instruction in ADL. Figure 12 shows the rewriting rule to utilizeMAC instruction. And Figure 13 shows the instruction set description of our ADL.

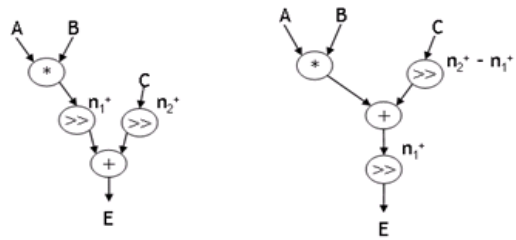


Figure 12. Rewriting Rule for MAC Instruction in ZSP400

```
instruction move shift up
: find MAC with moving shift
{
   uinteger(4) n1;
   uinteger(4) n2;
   GPR A, B;
   r0 C;

   action {
      C = ss plus(shift(\
                mult(A,B),n1, shift(C,n2));
   }

   syntax {
      "shla"::"  "::C::","::\
                   "("::n2::"-"::n1::")"
      "mac.a"::"  "::A::","::B;
      "shla"::"  "::C::","::n1;
   }
}
```

Figure 13. Example of ADL description for the rewriting rules.

6. Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of the proposed technique, which is implemented for Soargen compiler which is a retargetable compiler being developing in our group. 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

Figure 14 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.

Figure 15 demonstrates that we can reduce the code size by helping the compiler to select

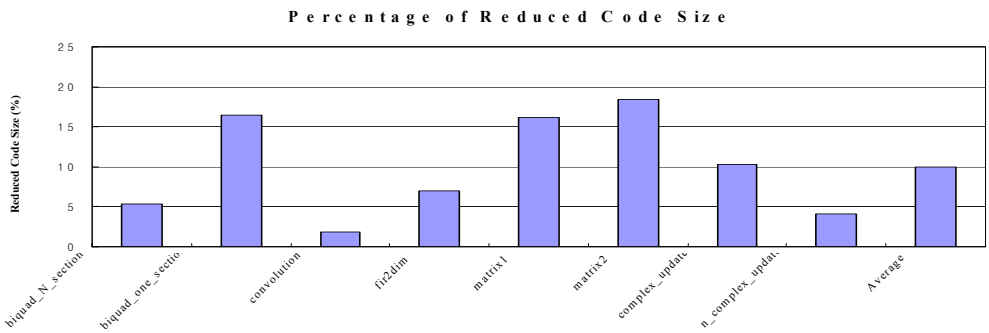


Figure 14. Reduced Execution Time

Figure 15. Reduced Code Size

DSP-specific instructions. The graph show that there is up to 16.7% and average 10% code size reduction. by using our technique.

7. Conclusion

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 retargetable compilation framework for 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 our ADL and compiler. We observed substantial improvement on code size and execution time.

Acknowledgement

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] Ki-Il Kum, Jiyang Kang, Wonyong Sung, "autoscaler for C: An optimizing floating-point to Integer C Program Converter For fixed-Point Digital Signal Processors". IEEE Transactions on Circuits & Systems II -Analog and Digital Signal Processing, 47:840 – 848,September 2000

[2] Daniel Menard, Daniel Chillet, Francois Charot, Olivier Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation" CASES 2002.

[3] T. Gr¨otker, E. Multhaup, and O.Mauss. Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis. In ICSPAT'96, Boston, October 1996.

[4] 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.

[5] R. Kearfott. Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin 2, 2(1): 95-112, 1996.

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

[7] M. Willems, V. Bursgens, H. Keding, and H. Meyr. System Level Fixed-Point Design Based On An Interpolative Approach. In Design Automation Conference, 1997.

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

[9] T. Parks and C. Burrus. Digital Filter Design. Jhon Willey and Sons Inc, 1987.

[10] P. Lapsely, J. Bier, A. Shoham and E. Lee, DSP Processor Fundamentals: Architectures and Features, IEEE Press 1997.
[11] ZSP 400 Digital Signal Processor Technical Manual, http://www.zsp.com.
[12] S. Muchinick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997.
[13] A. Chandrakasan, et. al, Optimizing Power Using Transformations. IEEE Transactions on CAD, Vol. 14, No. 1, 12–31, 1995
[14] M. Ahn, J. Cho, and Y. Paek, Using a H/W ADL-based Compiler for Fixed-point Audio Codec Optimization thru Application Specific Instructions. 정보처리학회논문지 제 13-A권 제4호, 2006.

박 상 현
2004년 서울대학교 전기공학부 (학사)
2004년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 저전력 설계.

안 민 욱
2003년 서울대학교 전기공학부 (학사)
2003년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 컴퓨터 시스템 설계

조 두 산
2001년 한국외국어대학교 전자정보공학부(학사)
2003년 고려대학교 전기공학과 (석사)
2003년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 컴퓨터 시스템 설계

윤 종 희
2005년 KAIST 전기및전자공학과(학사)
2005년~ 현재 서울대학교 전기컴퓨터공학부 석사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 재구성 가능 프로세서

백 윤 흥
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과학(박사)
1997년~1999년 NJIT 조교수
1999년~2003년 KAIST 전자전산학과 부교수
2003년~현재 서울대학교 전기컴퓨터공학부 부교수
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC