International Journal of Engineering Science Invention
ISSN (Online): 2319 – 6734, ISSN (Print): 2319 – 6726
www.ijesi.org ||Volume 4 Issue 4 || April 2015 || PP.46-51

Type Conversion Elimination by Dominant Flow Analysis

Dae-Hwan Kim
(Department of Computer and Information, Suwon Science College, 288 Seja-ro, Jeongnam-myun, Hwaseong-si, Gyeonggi-do 445-742, Rep. of Korea)

ABSTRACT*: Various data types are used in a program such as character and integer. Operand type is explicitly converted into another type or is implicitly converted when it is operated with another operand type. Additional conversion occurs when the size of a data type is less than that of an integer type, which promotes the value of the smaller type into an integer before computation in many high level programming languages such as C and C++ conforming to the standards. Type conversion often requires a sign or zero extension, which leads to code size increase and performance degradation. When multiple references of a variable need to be converted, it may not be necessary to convert all the conversions because the conversion of a previous reference may be live and hold the same value as the current conversion. Therefore, the proposed approach analyzes the flow of the definitions and uses of a variable, and minimizes the number of type conversions by inserting conversions only at dominant locations. Experimental results show that the proposed approach eliminates an average of 71.5% of type conversions.*

KEYWORDS – *Compiler, code generation, flow analysis, optimization, type conversion*

I. INTRODUCTION

Various data types such as character, short, and integer are supported in a program, each of which has its own size. The compiler writer often determines the size of each type based on the machine architecture. In a 32-architecture where the width of register is 32 bits, the integer type is also 32-bit long while the sizes of short and character types are 16 bits and 8 bits, respectively. Type sizes are different in 8-bit architecture processors where a register is 8-bit wide. Character is 8-bit long while the widths of both integer and short types are 16 bits.

Type conversions are more frequent than a programmer's expectation due to the implicit type conversions. ANSI C standard enforces the value of smaller type than integer should be converted to the integer value before arithmetic operations [1]. Thus, even if all the variables are of character types, they are converted integers before computation. Note that in most compilers, the widths of character and short types are shorter than that of the integer type.

Type conversion is accompanied by the sign/zero extension overhead. To change the value of a smaller signed type into a larger one, the sign bit of the smaller type is propagated to the remaining upper parts of a larger type. This conversion causes degradation in performance, and increases both code size and power consuming. Thus, the number of conversions should be minimized as much as possible.

Reducing power consumption is one of the dominant and crucial paradigms for portable digital devices where power is dissipate mainly by the switching activity in the circuits. The sign or zero extended value is the input to an ALU. If the previous conversion is live at the current conversion whose value is the same as the previous one, we can eliminate the current conversion while preserving the semantics. This can avoid the switching of the upper bits, and accordingly, reduce power consumption.

Often, the propagation of a sign/zero requires a pair of left shift and right shift instructions. To reduce such overhead, some architectures support type conversion instructions [2-3] such as the cbw (convert byte to word) instruction [2] in X86. This instruction automatically performs the sign bit propagation on the destination operand. In addition, many machines provide the instruction that loads a smaller value from memory into a register with a sign or zero extension. For example, ARM7TDMI [4] supports the loadsb instruction that performs a sign extension by loading a byte value from memory and extends the sign bit into the register. Even with the designated sign/zero extension instructions, type conversions still increase both code size and power consumption while degrading system performance.

In many compilers, type conversion follows the path from the source type to the destination type. Fig. 1 shows the type conversion path in the lcc compiler [5]. For example, an unsigned short type is converted into a signed char type as follows. Unsigned short is first converted into unsigned, and then, into integer, and finally integer is converted to a signed short type.

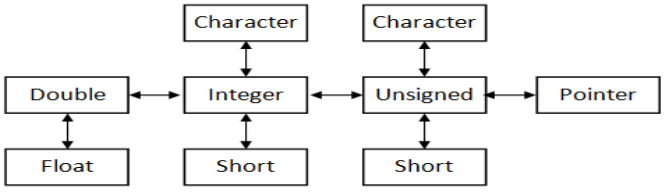

Figure 1. Type conversion path

When several type conversions need to be performed for multiple references of a variable, it may not be necessary for all the references to be converted. This is because the previous conversion may have the value for the current conversion. Thus, the flow analysis is performed to identify the necessary conversions. To improve performance and code density, some commercial compilers such as RealView C compiler [6] for ARM processors and Keil compiler for C8051 [7] optimize type conversions or provide the directive for the type conversion. However, the algorithms are not known because they do not publish their techniques. Authors in [8-9] briefly introduced the type conversion idea, but no details are described there.

The rest of this paper is organized as follows. Section II presents the proposed algorithm with examples. Section III provides experimental results, and conclusions are presented in Section IV.

II. PROPOSED ALGORITHM

The proposed approach inserts type conversions at the dominant locations, and deletes all the conversions that have the same value and are reachable from the dominant positions. We say node ‘x’ of a flow graph dominates node ‘y’, if every path from the initial node to ‘y’ goes through ‘x’ [10]. Dominance analysis is widely used in a flow analysis, and one typical use of a dominator tree is the loop detection. After the dominator tree of the basic blocks is constructed, the loop can be easily detected by finding the backward edge in the control flow. Fig. 2 shows a control flow of basic blocks and its dominator tree.

A web is a maximal union of definition-use chains (du-chains) that have a use in common, which is the basic unit for a register allocation and the proposed type conversion analysis. By analyzing the conversions in definitions and uses of a web, we can determine the dominant positions for the conversions. Then, we can delete other conversions that can be reachable from the dominant ones.

Though type conversions can be classified in widening and narrowing, there is conceptually no discrimination between them. Both conversions can be performed by the similar sign/zero extension. For example, the type widening from character to integer extends the sign bit of character value into the upper parts of integer value. In a 32-bit processor, the data sizes are 8 bits and 32 bits for character and integer, respectively. When a signed character value is converted into an integer, the bits from 31 down to 8 are filled with the bit 7, which is the sign bit of the character value. Similarly, the narrowing from integer to character propagates the bit 8 to the upper bits of the character type. Fig. 3 illustrates this property when ‘c’ and ‘i’ are of character and integer types, respectively.

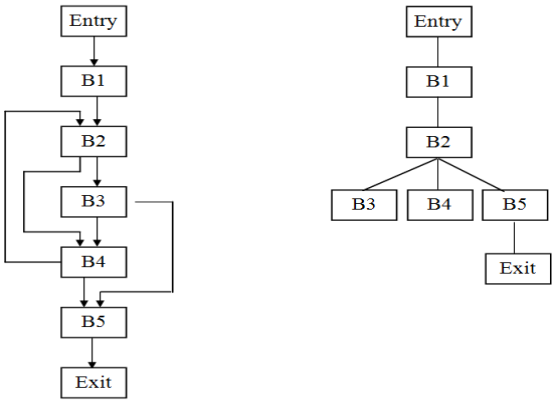

Figure 2. Control flow and its dominator tree

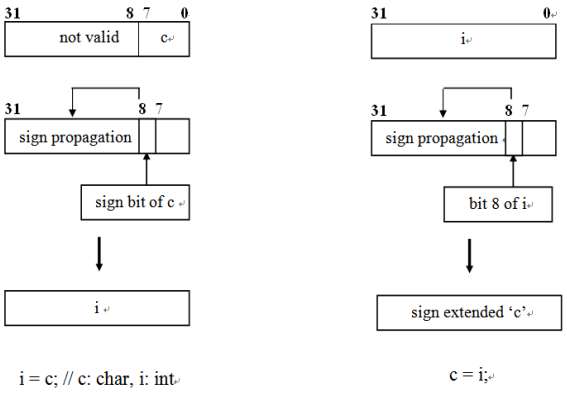

Figure 3. Type widening and narrowing

The proposed algorithm runs in two steps. At the first stage, it identifies the necessary and irremovable conversions among the conversions at definitions and uses of a web. The first stage simply traverses all the intermediate code. Fig. 4 shows the code segment and its intermediate code structure in lcc compiler [5]. The intermediate code is represented as a forest of trees. While traversing, this stage propagates the type of root node to the children nodes. Thus, if the type of a root operation is an integer, the children of smaller types should be extended into integer to preserve the program semantics. If the root operator is of character type, integer promotions can be eliminated in the children because uses in the children already have the sing/zero extended value. This simple mechanism works correctly with the additional consideration for the operators that cat not be eliminated. For example, the character value in the explicit right shift operation should be extended into the integer value to keep the semantics.

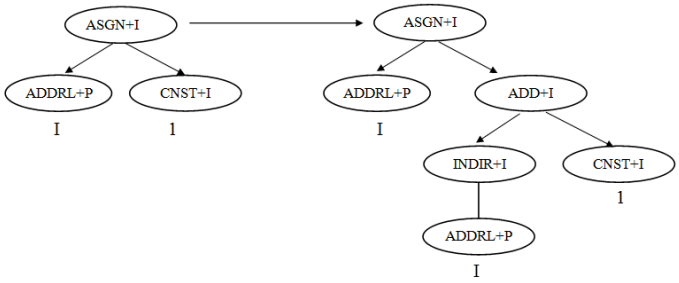

Figure 4. lcc intermediate code for 'I = 1; I = I + 1;'

The second stage optimizes the conversion locations by moving the conversions detected at the first stage into the minimal cost positions, which can be determined by the dominator tree for a web. Fig. 5 shows the control flow for the references of variables. Both INST1 and INST2 convert integer types to character types whereas INST3 and INST4 are nodes for changing types from characters to integers. Now, INST3 and INST4 require the type conversions, but the insertion of conversion at INST3 is enough because it dominates INST4.

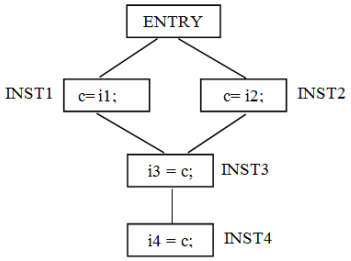

Figure 5. Example of dominance relation

To minimize the number of conversion operations, the proposed approach first constructs the dominator tree for references of a web. The dominance relations between uses of a web are constructed by using pre-built basic block dominator tree. Then, the definitions that can reach dominant uses are included as the parent of each dominant use. Fig. 6 (a) shows the control flow of basic blocks and accesses of variables, and (b) shows the dominator tree for web 'i'.

Node in a dominator tree is annotated by the overhead cost. Cost can be given considering for the target architecture and optimization purpose. Each cost is given as one for the code size optimization when the processor provides the sign extension instruction, and is given as two when the extension should be implemented by a pair of sequential left and right shift instructions. For the speed optimization, the cost is given as the execution time of an instruction for a node.

After the annotation, the proposed approach identifies dominant conversion locations. For this detection, the dominator tree of each web is traversed in a bottom-up order until all the roots are visited. When the cost of a node is less than or equal to the sum of all the costs of the children, the type conversion code is generated at that node while no conversion code is emitted at all the conversion nodes in the children. Fig. 7 shows the general algorithm of the proposed approach.

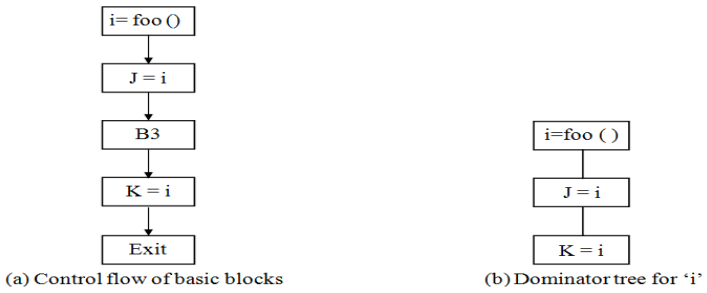

Figure 6. Dominator tree of references of a variable

Step 1. For each intermediate node, identify the necessary conversions by propagating the type of the root operator into the children.

Step 2. For each web which has type conversion operations do the following

1) make a dominator tree for the web.
2) annotate the tree with the conversion cost.
3) detect the valid conversion nodes in a tree.
   By traversing tree in a bottom-up order, move conversions into parents and remove the conversions at the children if the cost of the parent is less than the sum of the costs of the children.
4) detect the necessary conversions by analyzing the dominance relations.
5) emit conversion instructions for the valid and dominant nodes.

Figure 7. Proposed algorithm

Fig. 8 shows the example of the proposed detection. The effects of other optimizations such as copy propagation are not considered to simplify the description. At INST1, INST2, and INST3, type narrowing is performed while the instructions from INST4 to INST8 perform type widening. In the dominator tree, INST4 dominates all the positions from INST5 to INST8. Thus, for all the uses of variable 'c', only one conversion at INST4 is enough.

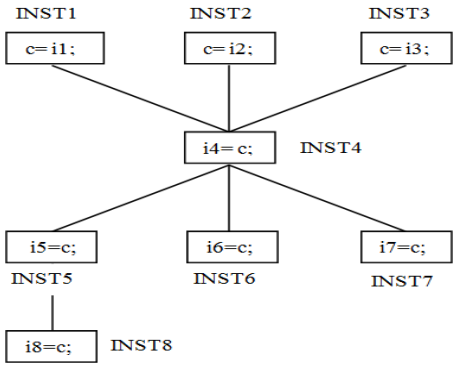

Figure 8. Dominator tree example

III. EXPERIMENTAL RESULTS

To evaluate the efficiency, the proposed type conversion elimination is implemented in lcc [5] targeting ARM7TDMI processor [4], which is one of the most widely used 32-bit embedded processors. The benchmarks are stanford, dhrystone, mpeg2, adpcm, g721, pgp, gsm and runlength programs. Stanford consists of a series of small real-world algorithms developed by John Hennessey of Stanford University. Dhrystone is one of the most widely used computing benchmark program to measure computer and compiler efficiency. G721, gsm and adpcm (adaptive differential pulse-code modulation) are three standard speech and voice codecs for the communication in the telecommunications network. Mpeg2 implements the standard for the generic coding of moving pictures and associated audio information. Pgp (pretty good privacy) is a public key encryption algorithm, and runlength is the implementation of a simple form of data compression.

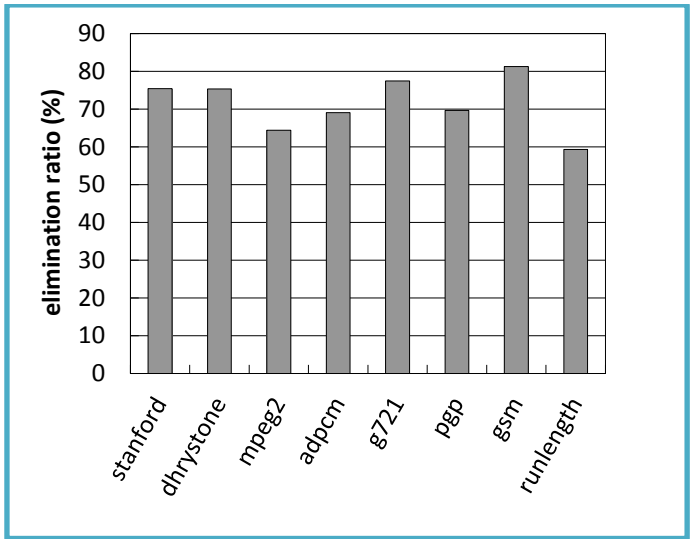

Figure 9. Type conversion elimination ratio

Fig. 9 shows the ratio of the eliminated type conversion achieved by the proposed approach. In eight benchmarks, an average of 71.5% of type conversions can be eliminated by the proposed approach. Note that the reduction ratio ranges from 59.3% to 81.2%. The reduction ratio is high when the dominant conversions can replace the subsequent conversion operations. Mpeg2, adpcm, and runlength are such examples.

IV. CONCLUSION

In this paper, a new technique is proposed to eliminate type conversions, which are generated by the explicit casting, the use of different types for the operator, and the implicit conversion such as integer promotion. The propose approach constructs a dominator tree for the references of a variable. Then, the approach inserts the conversions only at the dominant locations, and removes reachable conversions from the dominant points. The dominant flow analysis of a variable can be easily performed by the traditional flow analysis technique. The elimination of the redundant type conversions can significantly reduce code size and power consumption while improving performance.

V. ACKNOWLEDGEMENTS

This research was supported by Business for Cooperative R&D between Industry, Academy, and Research Institute funded Korea Small and Medium Business Administration in 2014 (Grants No. C0213097).

REFERENCES

[1] H. Schildt, *The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990*(Berkeley, CA: Osborne/McGraw-Hill, 1990).
[2] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2011.
[3] ARM Ltd., *Cortex-M3 technical reference manual*, 2010.
[4] S. Segars, K. Clarke, and L. Goudge, Embedded control problems, Thumb, and the ARM7TDMI , *IEEE Micro, Vol. 15, No. 5*, 1995, 22-30.
[5] C. W. Fraser, and D. R. Hanson, *A retargetable C compiler: design and implementation* (Redwood City, CA: Benjamin/Cummings, 1995).
[6] RVCT, Realview Compilation Tools. http://www.keil.com/arm/realview.asp.
[7] C. E. Nunnally , Teaching Microcontrollers, *Proc. The 26th Frontiers in Education Annual Conference*, Salt Lake City, Utah, 1996, 434-436.
[8] D. H. Kim, Advanced Compiler Optimization for Calm RISC8 Low-End Embedded Processor, *Proc. 9th Int. Conf. on Compiler Construction*, Berlin, Germany, 2000, 173-188.
[9] N. E. Johnson, *Code size optimization for embedded processors*, Technical Report UCAM-CL-TR-607, (University of Cambridge Computer Laboratory, 2004)
[10] S. S. Muchnick, *Advanced compiler design and implementation* (San Francisco, CA: Morgan Kaufmann, 1997).