Proceedings of the 5th WSEAS/IASME Int. Conf. on SYSTEMS THEORY and SCIENTIFIC COMPUTATION, Malta, September 15-17, 2005 (pp266-271)

Optimization on array bound check and Redundancy elimination

Dr V. Vijay Kumar
CSE Department
JNTU ,
School of Information Technology,
Kukatpally,
Hyderabad,
India

Prof. K.V.N.Sunitha
CSE Department
JNTU ,
G.N.I.T.S,
Shaikpet,
Hyderabad,
India -500008

Abstract

Compiler optimization technology has been steadily advancing as more sophisticated processors hit the market. For high end embedded processors, today's most advanced compilers take advantage of both CPU specific and application specific information to optimize code intelligently for optimal result. This paper discusses one such sophisticated optimization technique which combines array bound check optimization together with redundancy elimination combined with folding. The array bound check optimization described in this paper reduces the run time overhead by eliminating unnecessary bound checks. The results and comparisons demonstrate number of advantages of this method over the existing methods.

Keywords: computation, conditional statement, optimization, redundancy elimination, array bound check,folding.

1. Introduction

1.1 Array Bound Check Optimization

Program statements that access an array outside of its declared array bounds introduce errors that are difficult to detect. To aid in the debugging of programs under development, many compilers generate run-time checks to detect dynamic errors due to array bound violations. Since array bound violations occur quite frequently and can involve a memory load of array length and two compare operations, the overhead of these checks is very high, resulting in slow execution of programs. Because this is expensive in practice, most compilers allow the programmer to control the generation of run–time checks through a switch to be specified at compile time. Hence even if the software appears to execute normally, it may be providing incorrect results due to the run time errors.

To ensure high reliability, the run time checks should not be omitted from the software in general. This is becoming a trend for newer programming languages. For example, in Java, Array bound checks are mandatory. Since doing run time checks is expensive, compiler optimizations that reduce the execution overhead of array bounds is expensive, compiler optimizations that reduce the overhead of array bounds checks

without compromising safety are extremely useful. A bound check is a conditional statement that checks the lower and upper bounds of a subscript expression. If this conditional statement is true, then there is no array bound violation. If it evaluates false, then it terminates and reports error.

1.2 Redundant sub expression elimination using Value Numbering

Value numbering is a compiler-based program analysis technique with a long history in both literature and practice. Although the name was originally applied to a method for improving single basic blocks, it is now used to describe a collection of optimizations that vary in power and scope. The primary objective of value numbering is to assign an identifying number (a value number) to each expression in a particular way. The number must have the property that two expressions have the same number if the compiler can prove they are equal for all possible program inputs. The numbers can then be used to find redundant computations and remove them. This paper uses the technique of Hash based value numbering for handling redundancies.

2. Background

Suzuki and Ishihata discussed [4] the implementation of a system that performs array bound checks on a program.. Such techniques are significantly more expensive than the techniques based on dataflow analysis. Suzuki and Ishihata's approach cannot reduce the run-time. The elimination and propagation algorithms developed by Markstein et al. [3] were developed in conjunction with a traditional code optimizer. Chow has designed and built a machine-independent global optimizer [2].Cocke and Schwartz describe a local technique [6] that uses hashing to discover redundant computations and fold constants. But our algorithm is more efficient because it is using a combined approach.

3. Methodology

The optimization described in this paper combines elimination of array bound check with redundant instruction elimination using hash based value numbering and folding.

This paper describes a methodology which first identifies arithmetic statements and conditional statements that correspond to bound checks. If it is a conditional statement, first the indexes are assigned to the variables based on the values in hash table HT[B] maintained for every basic block. C_OUT[B] is another array used to store all the statements containing conditional checks within the basic block. This information is required to facilitate the propagation of redundant bound checks. At every entry of a new block the array 'C_IN[B]' of current block is updated with the values of its previous block which provides the required information for propagating the bound checks. 'Ref_array' is an array which is maintained for storing the information of conditional statements including its various points of reference.

Each unique expression is identified by its value number. Two computations in a basic block have the same value number if they are provably equal. In the literature, this technique and its derivatives are called "value numbering.". The algorithm is relatively simple. In practice, it is very fast. For each instruction from top to bottom in the block it hashes the operator and the value numbers of the operands to obtain the unique value number that corresponds to the computations value. If it has already been computed in the block, the expression will already exist in the table. The recomputation can be replaced with a reference to an earlier computation. Any operator with known constant arguments is evaluated and the resulting value used to replace subsequent references. This algorithm is extended to account for commutativity and simple algebraic identities without affecting its complexity. As the variables are renamed with new values, the compiler must carefully keep track of the location of each expression in the hash table.

The entire algorithm is described in two passes stepwise in the following section.In pass 1, lexicographic ordering is done and constant folding is applied. Then Value Numbering is carried out for all variables and propagation of array bound limits is Performed. In pass 2, by using value numbers

redundant computations are identified and eliminated.

3.1 Algorithm for Optimization on Array bound check and Redundancy elimination

Pass 1:

1).If it is first block, C_IN[B] and HT[B] are set to 0 else if B has more than one predecessor , there will be a point where the different paths may merge to one. So that at this point the flag bit indicates the initialization values of C_IN[B] and HT[B] taken from one of its predecessor. else
if B has single predecessor then C_IN[B] = C_OUT[B-1]
HT[B] = HT[B-1].
2).if the statement is an arithmetic statement ,rearrange expression using associativity.
3.)apply folding to evaluate the constant expression .
4)If the statement contains conditional checks
a)Assign a new index number to the variables as the current index number in hash table HT[B].
b)Store the expression in C_OUT[B]. Else
a)First assign index number to variables on the right side of '=' based on values in the hash table HT[B].
b)Assign a new index number to variable on the left hand side of '=' as current index number in hash table +1. (HT[B]+1).
Update the hash index number with new values assigned.
5)Repeat step 4 for all arithmetic and conditional statements.
6) The C_OUT[B] is checked , if the variable used in conditional statement is with same index then propagate the lower and upper limits as per the following Conditions.
a)If the lower limits and upper limits are different for the various options following conditional statement then the lower value of lower limits and upper value of upper limits are propagated to other expressions, and also the corresponding reference is updated in Ref_array.
b)If lower and upper limits are in sequence without modifying the variable in consideration, then upper value of lower limits and lower value of upper limits are propagated and the corresponding reference is updated in Ref_array.

Pass 2:

1)If the statement is conditional statement then do step 2 otherwise step 3..2)Check in C_IN[B], if there is any conditional check applied more than once on a variable with same lower limit or upper limit indicated by Ref_array. Eliminate the second and subsequent checks retaining the first check.
3)If the statement is assignment statement then do step 4 to 7.
4)Take the right hand side of string of assignment operator &Store it in an array which holds both left variable and right expression.
5)Take next statement, Compare the right hand side of '=' with the existing string in the array.
6)If it matches with any, then replace its L.H.S variable in the new location indicating that this computation is not required.
7)If it is not matched then store the expression along with L.H.S variable in the array.
8)Repeat 1 to 8 till it reaches end of statement.
9)Replace all the variables which are indexed during pass1 by their true variable names.

4. Experimental results :The above algorithm is applied on the following code and resultant code after every pass is listed in the following figure 1.

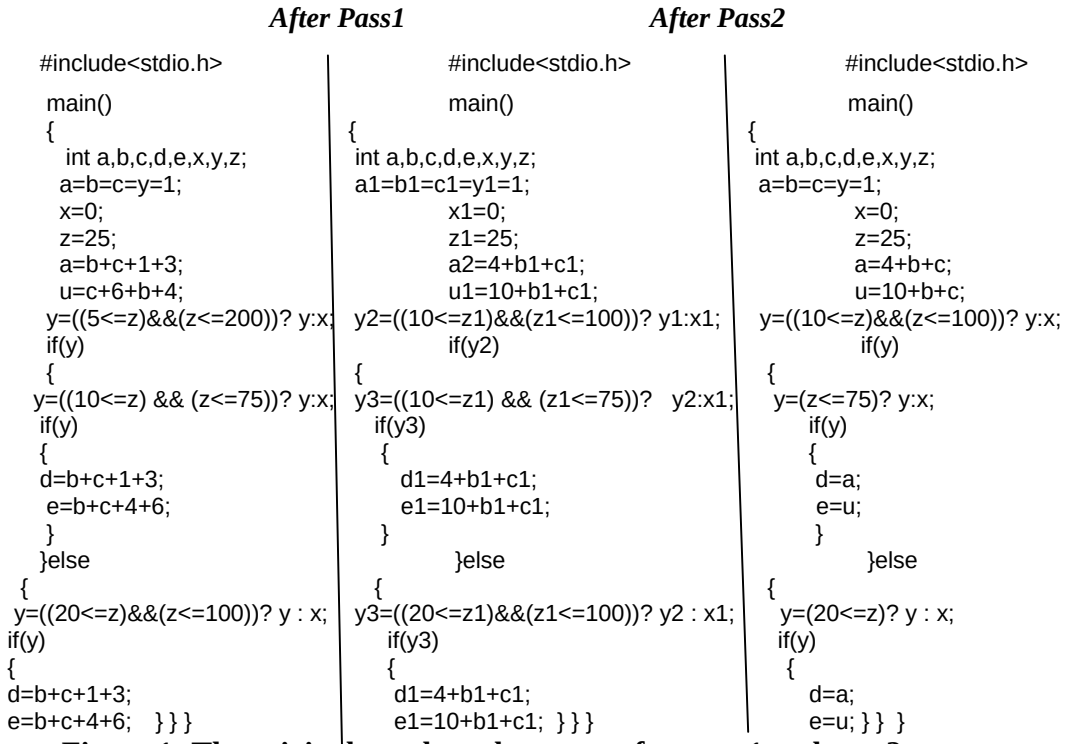

Figure 1: The original c code and outputs after pass1 and pass2

5. Conclusion

Bound checks reduce the runtime overhead. Eliminating redundant instructions will speed up the program by reducing number of instructions. By applying these two together with constant folding we have obtained reduction of 10% of source code which is a significant improvement at high level languages.

References:

[1] CHOW,F. 1983. A portable machine-independent global optimizer—Design and measurements.Tech. Rep. 83-254, Ph.D. thesis, Computer Systems Labs, Stanford Univ., Calif.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass.

[3] MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. In Proceedings of SIGPLAN 82 Symposium on Compiler Construction. (Boston, Mass., June 23–25). ACM, New York, 114–119.

[4] SUZUKI, N., AND ISHIHATA, K. 1977. Implementation of array bound checker. In Proceedings 4th ACM Symposium on Principles of Programming Languages. ACM, New York, 132-143.

[5] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[6] Steven S. Muchnick. Advanced Compiler Design and Implementation.