The New Trends in Compiler Analysis and Optimizations

Article *in* International Journal of Emerging Trends & Technology in Computer Science · May 2017

CITATIONS
0

READS
4,500

1 author:

Emmanuel Okon
University of Lagos
8 PUBLICATIONS **8** CITATIONS

SEE PROFILE

The New Trends in Compiler Analysis and Optimizations

Enyindah P.[1],Okon E. Uko[2]

Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria.
Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria.

Abstract—*Compiler construction primarily comprises of some standard phases such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and target code generation but due to the improvement in computer architectural designs, there is a need to improve on the code size, instruction execution speed, etc.*
Hence, today better and more efficient compiler analysis and optimization techniques such as advanced dataflow analysis, leaf function optimization, cross-linking optimizations, etc. are adopted to meet with the latest trend in hardware technology and generate better target codes for recent machines.

Keywords —*Compiler, analysis, optimizations, new, advanced, data-flow, control-flow, loop, function-calls, memory, processors, power, consumption, inter-procedural, alias.*

I. INTRODUCTION

As technology in computer architectural design advances into very large instruction word (VLIW) architecture, embedded processors and parallel execution of instructions, there is a dire need to adopt a more efficient way of analysing and optimizing codes for high performance computing in these modern[1] machines. This paper focuses on showcasing most recent compilation techniques applicable today for effective compiler analysis and backend optimization in advanced machines today.

Conventional[2] compiler analysis and optimization has always been in use before now. This method were efficiently generating target machine codes until recent changes in the development of computer architecture and design (necessitated by user convenience and computing efficiency) was introduced as a product of researches. This change in hardware development has necessitated machine independent compilation techniques against machine dependent analysis and optimization techniques applied in conventional compiler construction. Also, compilation of source codes for embedded processors has drawn programmer's attention to the need for controlled power or energy consumption, real-time execution and portability of code.

Conventional compilers were designed using common control flow analysis technique such as the data flow analysis. While advance compiler designs also employs data flow analysis among other methods, their mode of application is different because while the conventional design focuses on machine dependent analysis, the modern/ advanced design concentrates on machine independent analysis.

In this paper, we discuss new and sophisticated compiler analysis and optimization techniques for modern systems against the mere redundancy elimination of the conventional compilers. We also highlight key modifications to conventional compiler techniques (such as data-flow analysis, parallel optimization, loop optimization, etc.) that are still applicable to new computing technology.

Section II covers conventional compiler analysis and optimization techniques, Section III discussnew compiler analysis and optimization techniques.

II. CONVENTIONAL COMPILER ANALYSIS AND OPTIMIZATION TECHNIQUES

The target of conventional compiler analysis and optimization techniques is to eliminate all forms of inefficiencies in a computer program, eliminate redundant operations (especially in loops and recursive evaluations), and manage resources (by reordering operations and data to better map to the target machine).

In order to meet these targets efficiently, methods such as data-flow analysis, peep-hole optimization, sub-expression elimination, jump-to-jump elimination, index-check elimination, loop/function-call optimization are employed. We consider data-flow analysis as well as the above mentioned optimization techniques which are categorized under local, global and loop optimization.

A. Data-flow Analysis

In conventional compilers, data-flow analysis tries to discover how information flows through a program. An analysis could attempt to approximate a collection of positive values that a variable can hold, and hence the flow is from assignment to uses.

[1]Recent.
[2]traditional

It is the process of collecting information about the way the variables are used, defined in the program.

Usually, it is enough to gather this information at the boundaries of basic blocks[3], since computing the information at each point in the basic block is easy.

When Forward flow analysis is applied, the exit state of a code block is determined by its entry state.

Figure 1 illustrates the data-flow equation used to enhance the code block relationship:

For each block c:

$\mathbf{Out_c = trans_c(in_c)}$

$\mathbf{in_c = join_{p\epsilon}pred_c(out_p)}$

Fig. 1 Data flow equation for determining entry and exit relationship in forward flow analysis.

In the above equation, **transfer function** for block b is denoted by $\mathbf{trans_c}$. It operates on the entry state $\mathbf{in_c}$and yields the exit state $\mathbf{Out_c}$. The entry state of c $\mathbf{in_c}$is determined by performing a join operation **join** on the exit state of the predecessor$_{p\epsilon}$**pred**$_c$ of block c.

This equation when solved can be used to determine the properties of the program at the block boundaries. There are different types of dataflow analysis for conventional compiler design and each of them has their own unique join and transfer operations.

B. Local Optimization

Local optimization is often applied in conventional compiler design to help enable code improvement within a local block of codes. Example of local optimization techniques include:

- **Local sub-expression elimination:**This transforms common multiple assignment expressions (in the intermediate code of the compiler) into equivalent binary expressions on the right hand side. For instance, the code in figure 2 below will be transform into figure 3 using local sub-expression elimination.

```
a := b + 5;
.
.
.
b := 5+b
```

Fig. 2 intermediate code with multiple assignment expression

```
t1 := b + 5;
a := t1
.
.
.
b := t1
```

Fig. 3 intermediate code transformed by localsub-expression elimination

[3]Sequence of instructions with a single point of entry and a single point of exit.

- **Local Constant Propagation/Folding:**
Constant folding requires that the compiler be optimized for alertness in handling user-defined constants and allocating arrays to these constants.

- **Local Strength Reduction:**
Local strength reduction is often applied in conventional compiler optimization to enable the replacement of an expression by its meaning within each basic block of the intermediate code. This method of optimization reduces compilation time greatly because if computers performs addition faster than multiplication, then

X*Y=X+X+X+…(Y times)

can ensure quicker compilation of source codes.

C. GlobalOptimization

It is during global optimization that data-flow analysis is applied to enable optimization of codes between blocks. Global optimization unlike local optimization operates on a global[4] scope. Global optimization can extend its usefulness to reduction of codes within a loop. Several techniques are used to achieve global optimization but among the commonest are the following:

- **Redundant (common) Sub-expression elimination:**

Quite unlike the local common sub-expression elimination technique, redundant sub-expression elimination technique seeks to globally determine common sub-expression in the entire program and compute then at once in each path.

This is demonstrated in Figure 4 below.

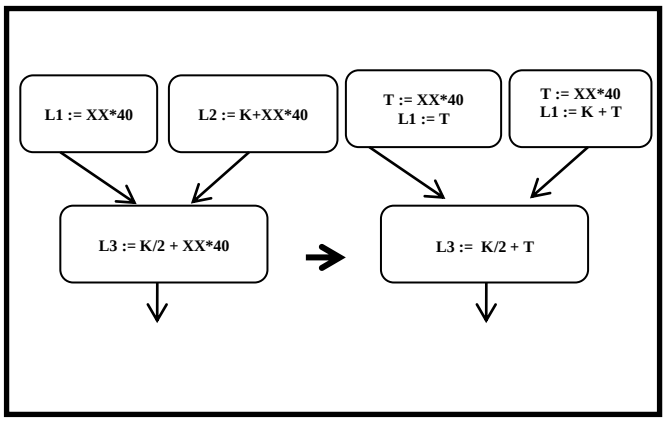

Fig. 4 intermediate code transformed by localsub-expression elimination

- **Global Constant Folding and Propagation:**

[4]Non-local or wider.

This technique constructs a use-definition chain (ud-chain)[5]. Constant propagation is implemented using ud-chain. Definitions are matched to their subsequent uses in each block and in situations where there are no definitions within the block; all definitions are match with the use as the reach the block.

- **Dead Code Elimination:**

After the elimination of sub-expression, and other optimization operations, the intermediate code contains unnecessary codes or even blocks. This is where dead-code elimination is applicable.

Other types of global optimization for conventional compiler optimization include conditional re-ordering, GOTO casing, alias analysis, Array temporary elimination, conditional pruning and assignment elimination due to equality.

D. Loop Optimization

One of the objectives of conventional compiler design is to increase the speed of compilation. We discuss loop-variant code motion and induction variable detection/elimination as the two major conventional loop optimization techniques in this article.

- **Loop-invariant code motion:**

A statement within a loop in a program is considered to be loop-invariant if it produces the same value each time the loop is executed. Scenarios like this could cause delay in compile time. Hence, when this sort of statement is spotted, code motion is carried out using the algorithm below:

Algorithm:

- Compute definition reaching header and dominator for each given node in a loop.
- Identify the loop-invariant statements.
- Identify exits of the loop; the nodes with a successor outside the loop.
- Check for statements that are;

1. Loop-variant
2. In blocks that dominates exit
3. In blocks that dominates all blocks in the loop which use their computed values
4. Assigned to variables not assigned to elsewhere in loop

- Perform depth-first search. Use depth-first order to verify each block.
- Move statement selected in the search process to the pre-header.

- **Induction variable detection and elimination:**

When each loop is accessed in an intermediate code, there may be some variables whose value per loop iteration is a linear function of the iteration index. Strength reduction or code elimination are often carried out on such values or expressions. Figure 5 shows a typical induction variable:

```
X := 0
FOR I :=1 to N DO
        …
        X := X+1
        Y := 2+X
        Z :=  3+Y
        …
        A(Z) :=
ENDFOR
```

Fig. 5 Intermediate codes with induction variables X, Y, Z and I (a trivial induction variable).

Other examples of loop optimization that are common in conventional compiler design includes the following; loop jamming or fusion, loop unrolling, count up to zero and un-switching.

III. New Compiler Analysis and Optimization Techniques

Conventional compiler analysis and optimization considered most importantly the correctness of an executed code, the efficiency in execution of programs irrespective of the professionalism of the programmer and lastly the speed of compilation of a program.

Today, as we embrace new technology with great changes in architectural design (we now have even smaller devices with embedded processors and micro-chips), highly parallel processing, registers allocations, cache heirarchy and high processing speed demands, new compiler analysis and optimization needs to be adopted in order to match with the increasing demand for better compilers.

E. Data-Flow Analysis

Data-flow analysis information in the conventional compiler analysis seems to be too Boolean since it places priority on bit-vectors bzut modern compilers are beginning to incorporate some level of fuzziness in its data information. For instance an expression can be *may-be-available.* Hence, incorporating the values that fall between such values(assuming a *must-be-available* and*is-not-available expression).*

- **Alias Analysis:** During liveness analysis in conventional compiler analysis, the two common functions applied are the *Gen* and *Kill* function but most recent techniques in advanced compiler analysis also applies the *Aliases* in determining the liveness of the references at the exit of the block.

F. Reverse-Inlining (Procedural Abstraction):

Reverse-inlining also considered as procedural abstraction is one of the recent method in compiler optimization whose aim is to achieve code size reduction. Reverse inlining

[5]A collection of variable definitions linked with their uses.

achieves this by using function calls to replace code patterns that are in the entire program.

The name reverse-inlining is typical due to the counter operation of this technique to code inlining (where function calls are replaced with function bodies). This technique is noted to reduce code size by 30%.

G. Leaf Function Optimization:

Leaf functions are those functions who do not directly call functions in a program. Leaf optimization achieves code reduction by creatingthese types of functions. When represented in a call graph, leaf functions forms the leaves of the call graph.

It is easier to inline leaf functions. Hence, function entry/exit code is not required and this reduces code size greatly. During leaf optimization, register constraints are placed as function calls need to be controlled (this also reduces code size). After leaf optimization, there is further opportunity for optimization as the body of the inlined function is within the context of the parent function.

Leaf function optimization can be applied to functions that do not resemble leaf function, such as the functional programming code in figure 6 below.

Fig. 6Fuctional programming syle factorial function that can be transformed into a loop using leaf fuction optimization.

H. Combined code motion and register allocation:

Combined code motion and register allocation uses two conventional compiler phases: code motion and register allocation. Codemotion aims to place instructions in less frequently executedbasic blocks, while instruction scheduling within blocks orregions arranges instructions such that independentcomputations can be performed in parallel. The RegisterAllocation and Code Motion (RACM) algorithm aims toreduce register pressure firstly by moving code (CodeMotion), secondly by Live-range splitting (Code-Cloning),and thirdly by spilling.

This optimization is applied to theVSDG intermediate code, which greatly simplifies the task ofcode motion. Data dependencies are explicit within the graph,and so moving an operation node within the graph ensuresthat all relationships with dependent nodes are maintained.

Also, it is trivial to compute the live range of variables (edges) within the graph; computing register requirements at anygiven point (called a cut) within the graph is a matter ofenumerating all of the edges (live variables) that areintersected by that cut.

I. Cross Linking optimization:

This method is commonly used in search engine optimization. Today, this method has also been applied in compiler optimization. Cross-linking can be applied locally or globally to functions that contains switch statements with similar tail codes.

Since recent computer architectures are focused on code reduction, cross-linking has this as its major goal. When tail codes are spotted in a switch statement, cross-linking optimization algorithm is used to factor out this codes thereby reducing the actual size of the code.

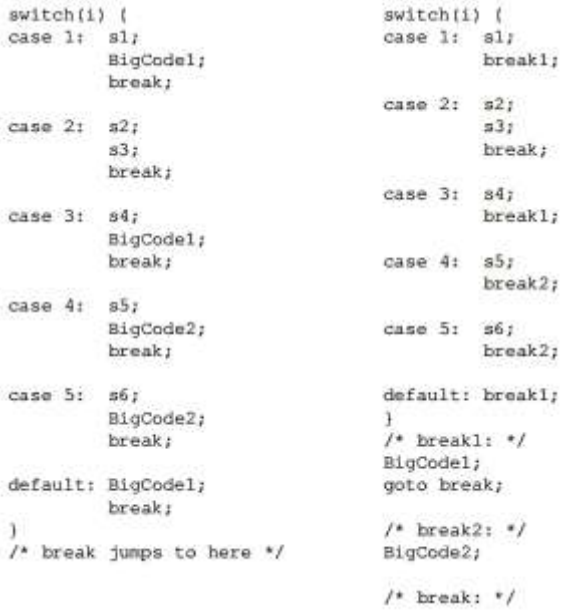

Fig. 7 code segment showing application of cross linking optimization to an un-optimized switch case statement

J. Address Code Optimization:

Address code optimization uses simple offset and general offset assignment techniques to the number of address computation code by reordering variables in memory. The main aim of address code optimization is to speed-up instructions or execution time.

As data processing expression increases in number, memory access instructions and address computation codes also increases. Hence, there is need for rearrangement of layout of data in memory to reduce and simplify address computation.

K. Type Conversion Optimization:

Data processing also involves the physical size of the data itself, and the semantics of data processing operations. Support for a variety of data types is the target of most compilers; as such compilers insert many implicit data type conversions, such as sign or zero extension. For instance, the C programming language specifies that arithmetic operators operate

on integer-sized values (typically 32-bit). For example, given the code below;

```
char a, b, c;
    ...
c = a + b;
    ...
```

Fig. 8 code segment showing application of type conversion optimization

The code above promotes char (say, signed 8-bit) variables a and b to integer types before doing the addition. The result of the addition is then demoted to char type before being stored in c. If a and b are held in memory, then most processors have some form of load-with-sign-extend operation, so the above expression would indeed compile into four instructions (two loads, add, and store).

However, if any of a or b are held in a register, then the compiler must generate code that behaves as if they were loaded from memory with sign extension. The effect of this is that a naıve compiler will generate code that sign-extends a and b before doing the addition. For a processor without explicit sign-extend instructions the compiler will generate two instructions for each sign extension (a left-shift to move the char sign bit into the int sign bit, and then a right shift to restore the value and propagate the sign bit into the upper 24 bits of the register).

L. Multiple Memory Access Allocation:

Multiple Memory Allocation (MMA) is one of the newest optimization techniques. It involves loading and storing each instruction in multiple registers. Microprocessors today use this method to reduce the code size. For example, in a ARM7 processor, the LDM instruction uses only sixteen bitsto encode up to sixteen register loads (512 bits without the useof the LDM instruction).). Effective use of LDM and STMinstructions in a ARM7 processor can save up to 480 bitsopcode space.

IV. CONCLUSIONS

New generation compilers are designed to effectively generate target codes for embedded processors, parallel processing devices and very large instruction word (VLIW) architecture. The objective of new compilers is not just to generate efficient codes but to reduce the code size, properly utilize memory and increase program execution speed.

This objective is evident in the implementation of the new compiler analysis and implementation techniques.

REFERENCES

[1] A. K. Sarma, *New trends and challenges in source code optimization*. Journal Article. Delhi, India: International Journal of Computer Application, December2015, vol. 131-No.16.

[2] A. Krall and S. Lelait. Compilation techniques for multimedia processors. Intl. Journal ofParallel Programming, 28(4):347-361, Aug. 2000.

[3] A. V.Aho,J . E. Hopcroft, and J . D.Ullman, *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[4] Assmann, U. *Graph Rewrite Systems for Program Optimization.* ACM Trans. Programming Languages and Systems 22, 4 (July 2000), 583– 637.

[5] B.Alpern, M. N. Wegman, and F. K.zadeck, *Detecting Equality of Variables in Programs*. In Proc. Conf. Principles of Programming Languages (January 1988), vol. 10, ACM Press, pp. 1– 11.

[6] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. *Adapting instruction level parallelism for optimizing leakage in VLIW architectures.* In Proc. of ACM SIGPLAN Conf. on Language, Compiler, and Tool for Embedded Systems, pages 275–283, San Diego, CA, June 2003.

[7] M. K. Jha, *Compiler Construction: An Advanced Course*, 3rd ed., Delhi, India: DhanpatRai and Co. Ltd, 2011.

[8] N. Johnson, *Code Size Optimization for embedded processors*. TechnicalReport. Cambridge, United Kingdom: University of Cambridge, December2004, vol. 131-No.16.

[9] R. Sinha and A. Dewanga, Transmutation of Regular Expression to Source Code using Code Generators. IJCTT Journal, 2012, vol.3 Issue 6.

[10] T. E. Mogensen, “Basics of Compiler Design” 2nd ed., Copenhagen, Denmark: University of Copenhagen,2010.

[11] U. P. Khedker and A. Sanyal. *Improving garbage collection through static analysis.* Technicalreport TR-CSE-004-02, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 2002.

[12] V. Garg and Anu, A Review of DFA Minimizing State Machine using Hash-Tables, IJCTT Journal, April 2013, Vol. 4 Issue 4, 2231-2803.