http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 9 (5): 984-988, 2010
ISSN 1812-5638

An If-Conversion Algorithm Based on Predication Execution

[1]Zuwei Tian and [2]Guang Sun
[1]Department of Information Science and Engineering,
Hunan First Normal University, Changsha, 410205, China
[2]Department of Information Management,
Hunan Financial and Economic College, Changsha, 410205, China

Abstract: In this paper, we analyzed the if-conversion algorithms based on IA-64's predicated execution in GCC, and pointed out some existing problems of GCC if-conversion. And then an amended algorithm is proposed which applies if-conversion selectively by considering multi-aspects of factors, such as critical path length, resource usage height, misprediction rates, misprediction cost and instruction number. The experiment results show that the proposed algorithm can improve the effect of if-conversion and promote the performance of programs.

Key words: Instruction level parallelism, predication execution, if-conversion, compiler optimization

INTRODUCTION

Human beings have always pursued high performance of microprocessors. Exploiting as much parallelism as possible is still the base to implement high performance computer systems. For any microprocessor with the ability of instruction-level parallelism (Mahlke *et al*., 1992; Yan and Zhang, 2008a, b; Wang *et al*., 2009), to take full advantage of CPU's performance, the key technique is to make the CPU issue uninterruptedly multiple instructions, that is, to maintain a high instruction issue rate. The performance of modern processors is becoming highly dependent on the ability to execute multiple instructions per cycle. But no matter what architecture it is, branch instruction is an important factor of affecting the continuity. Especially, in terms of modern high-performance microprocessors with wide-issue and deep pipeline architecture, branch instruction is an expensive instruction. The jump of control flow led by branch interrupts the continuity of the instruction pipeline and results in instruction cache failure and program execution pause. The deeper the number of pipeline stations, the longer branch delay, the greater its impact will be. According to statistics (Bringmann *et al*., 1993), if program's branch frequency is 20%, assuming that the processor has a 2 cycle branch delay, it implies 40% of the machine time being wasted on the branch delay, the three cycle delay will waste 60% of the machine time. More importantly, according to statistics, approximately 10 to 30% of instructions in a program are branches, an average of one branch for every three to five instructions. Therefore, branch optimization is a very important compiler optimization, especially, for the instruction-level parallelism microprocessor with deep pipeline and multi-issue. On the other hand, branch instruction causes the program to be divided into a number of basic blocks. Instruction scheduling can not be performed in parallel with the original instructions. In order to reduce the execution costs of a branch instruction, modern high performance microprocessors adds the hardware itself with the ability of branch prediction and provides the following three aspects of support for branch optimization: providing predicated execution mechanism supporting the implementation of branch elimination and basic block merging optimization; providing special support for software pipelining and the export of counting loop, supporting loop pipeline optimization; providing the software hint for the branch, improving the accuracy of hardware branch prediction and the efficiency of pre-fetched resources, reducing the cost of branches.

Not only can if-conversion eliminate the branch and provide continuous instruction stream, it can also extend the length of basic block and improve the effective parallelism of code. It provides further opportunities and convenience for compiler to implement other optimization. For example, if-conversion can be used to implement a number of key compiler optimization, such as loop optimization, software pipelining scheduling, instruction scheduling and register allocation and so on. It brings the

Corresponding Author: Zuwei Tian, Department of Information Science and Engineering, Hunan First Normal University, No. 1015, Third Road of Fenglin, Yuelu District, Changsha, 410205, China
Tel: 86-731-82841011 Fax: 86-731-82841005

potential benefits to them. The formation of the hyperblock and treegion compiler areas also needs the support of if-conversion. Mahlke *et al.* (1994) studied the impact of if-conversion on branch prediction. It is shown that an average of 27% of branches and 56% of mispredictions are eliminated at runtime with if-conversion over a mix of SPEC92 benchmarks and UNIX utilities (Mahlke *et al.*, 1995). Mantripragada and Nicolau (2000) presented a new algorithm which selectively performs if-conversion by making use of profile data. Simulation experiment results indicate, on 4-issue processor, decreases branch misprediction rate from 1.2 to 55%.

IF-CONVERSION

If-conversion (Allen *et al.*, 1983; Mahlke *et al.*, 1994; August *et al.*, 1997; Bruel, 2009) is a compilation technique that takes advantage of predicated execution provided by modern processors to promote the performance of programs. Each instruction has an additional source operand called a guarding predicate. If an instruction's guarding predicate is true, then after we execute it, we commit it-allowing its result to update processor state. If an instruction's guarding predicate is false, then after we execute it, we nullify its result preventing it from making any change to processor state. There are many benefits from if-conversion. Firstly, if-conversion can convert branch instructions in program to sequential predicated execution instructions, thus control dependence be turned into data dependence and instructions in multiple paths can be merged into a basic block after if-conversion, which increases ILP in basic blocks by making basic blocks contain more instructions and extend instruction scheduling space. On the other hand, because of the reduction of branch instructions and simplicity in hardware design, performance loss due to branch mispredication is reduced as well.

Figure 1 is an example of If-conversion. Original control flow graph is shown in Fig. 1a. In Fig. 1b, original codes are converted into predicated codes. The branch is removed by if-conversion and the original four basic blocks are merged into a single basic block, extending the range of instruction scheduling.

What branch should be if-converted and when the if-conversion should be applied are two key problems needed to be solved.

If-conversion works by removing branches and combining multiple paths of control into a single path of conditional instructions. However, excessive if-conversion does not necessarily improve program performance and even a few if-conversions do not necessarily improve performance. The conditional code

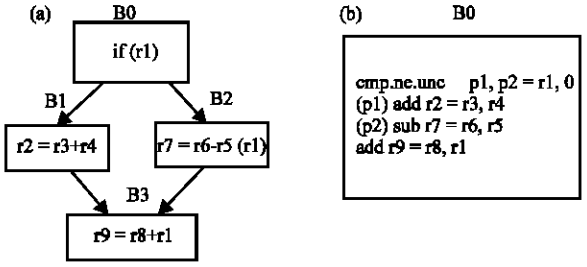

Fig. 1: An example of if-conversion; (a) Original control flow graph and (b) Predicted code

will be converted into predicated execution code by if-conversion, which will eliminate the problems posed by the branch. It will make the two branch instructions use instruction slot at the same time and have the same execution cycle. Therefore, improper conversion may cause other problems. For example, it requires a longer execution time than that of the original code in the same path, because it contains another long path. On the other hand, the range of basic block is expanded, thus enlarging the number of the register allocation. If register allocation (Quinones *et al.*, 2006, 2007) is not sensitive to the predicate, it could put the pressure on register allocation. Therefore, it is necessary to select branch for if-conversion, this choice needs to consider various factors.

Mahlke *et al.* (1994) firstly presented the concept of hyperblock. A hyperblock is a structure created to facilitate optimization and scheduling for predicated architectures. A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations. Hyperblocks are formed by applying tail duplication and if-conversion over a set of carefully selected paths. Inclusion of a path into a hyperblock is done by considering its profitability. The profitability is determined by the following factors: resource utilization, invalid instructions, branch instructions inherent instruction dependence, branch execution frequency, the balance of branch execution and so on.

In general, if-conversion may be applied early in the compiler backend or delayed to occur in conjunction with scheduling. Applying if-conversion early enables the full use of the predicate representation by the compiler to facilitate ILP optimizations and scheduling. However, the decision-making of if-conversion may dynamically change with the compiler optimization code, three problems exist that are difficult to solve when applying if-conversion early. Three such problems presented here are optimizations that change code characteristics, unpredictable resource interference and partial path inclusion. August *et al.* (1997) presented a framework for

balancing control flow and predication. This framework overcomes limitations of other schemes by utilizing two phases of predicated code manipulation to support predicated execution. Aggressive if-conversion is applied in an early compilation phase to create the predicate representation and allow flexible application of predicate optimizations throughout the compiler backend. Big hyperblocks are formed by applying if-conversion, it forms hyperblocks which are larger than the target architecture can handle. The if-converter relies on later compilation phases to ensure that this hyperblock is efficient. The large hyperblocks increase the scope for optimization and scheduling, further enhancing their benefits. Partial reverse if-conversion is implemented during instruction scheduling. According to the target processor feature, the algorithm adjusts the amount of predicated code in each hyperblock, balances control flow and the number of predicates in code. Partial reverse if-conversion may be repeatedly applied to the same hyperblock until the resulting code is desirable. To improve performance, a delicate balance between control flow and predication must be created by the compilation framework.

ANALYSIS OF IF-CONVERSION ALGORITHM IN GCC

The if-conversion in GCC is done in the compiler backend, aiming at the RTL intermediate code. The main function of the if- conversion algorithm in GCC is defined in the file ifcvt.c, its main entrance is if_conver () function. In the if_convert ( ) function firstly finishes a series of initialization work, such as setting loop exit signs, removing the flag of all the basic blocks, calculating the post dominator and so on and then all of the basic blocks in the function blocks are processed one by one.

By calling find_if_header (bb, pass) function to determine whether the current basic block bb is just only two successors, if so, then find_if_block (and ce_info) function is invoked to determine whether the block bb is a simple IF_THEN or the head block of the IF_THEN_ELSE structure, if not, then the function returns. Otherwise, the function transforms these simple IF_THEN or IF_THEN_ELSE structure into a sequence structure and thus removes the corresponding branch. The find_if_block ( ) function invokes process_if_block() function to complete the conversion work. In the process_if_block() function, the actual conversion is done by noce_process_if_block() function and cond_exec_process_if_block() function. If the conversion is successful, find_if_header() function returns a pointer to the converted basic block, namely, the new_bb points to the new converted block. If the condition of while loop is true, it will invoke find_if_header() function with new_bb as parameter to process similarly. If the conversion is unsuccessful, then exit while loop, the function continues to execute for loop to deal with the successor block of the bb block. When all the basic blocks of the function are processed, if successfully convert IF_THEN or IF_THEN_ELSE into a conditional execution statement in the conversion process, then the value of flag variable cond_exe_changed_p is true, the conditions of do loop is true. Then, in order to discover new opportunities for conversions, all basic blocks in the function are re-processed uniformly.

IMPROVEMENT OF IF-CONVERSION ALGORITHM BASED ON GCC

The if-conversion algorithm in the GCC, mainly considers the accuracy and portability during conversion process and all IF blocks that meet the transition conditions are converted. After the IF block is identified, the algorithm considers only the largest number of the target machine supporting predicate instruction. The programs only consider simply whether the number of the instruction in the IF block is less than the target machine defined MAX_CONDITIONAL_EXECUTE or not (a constant is defined in the ia64.h file, the size will be double if it is IF_THEN_ELSE block). To determine whether the current block is suitable for if-conversion, the conversion did not take into account IF block THEN part and ELSE part of the characteristics of instruction, the length of the critical path, THEN block and ELSE blocks balance, the issue of resource conflict in incorporated basic block and not taking full advantage of the new feature of IA-64, such as the use of parallel comparison instructions of IA-64 to simplify the definition of predicate instruction, reducing the height of control. If-conversion has both advantageous and disadvantageous aspects, in order to maximize the benefits of if-conversion, the chosen IF block should be converted. It must have a better evaluation system to weigh the advantages and disadvantages. In some cases, not only if-conversion does not improve the code quality, but the code performance will deteriorate, for example, there is such a case, before if-conversion, 90% of probability, the program take the left path, only two cycles, only 10% of the cases, 20 cycles. After if-converted, we need 20 cycles each time. In this case, even if one of the branch instructions are deleted, the performance of the entire code will deteriorate. The if-conversion algorithm in the GCC, without considering the actual situation of IF block and unconditionally carry out the conversion, this will definitely affect the quality of the converted code.

What branch should be if-converted is a key problem needed to be solved in if-conversion process. As mentioned above, when selecting a branch to be converted in if-conversion process based on GCC, we only consider the accuracy and portability of conversion process, without considering whether it can enhance the quality of converted code or not, this will led to some of branch in the program which is obviously not suitable for if-conversion carried out the conversion and will seriously affect the quality of generated code. In order to choose the conversion branch more reasonably, we have improved the original if-conversion algorithm. We add a heuristic function worth_if_convert ( ) to regard as an approximate criteria of branch selection, that is, if the execution time of converted predicate code is longer than the execution time of the average code, we think that the IF block is not suitable for conversion, the function return FALSE, otherwise, the current IF block after conversion can improve the performance, be worthy of conversion and return TRUE. The heuristic function considers the length of the critical path, resource use, misprediction rate, the cost of misprediction , the number of instructions and other factors.

The improved if-conversion algorithm is described as follows:

Step 1: Considering only the use of resources, the length of critical path, the probability of implementing the various branches, misprediction rates, the cost of misprediction, calculating the non-predicate code execution time

Step 2: Calculate the execution time of the predicate code after if-converted

Step 3: Decide whether it is worthwhile to if-conversion, that is, it will be converted if the predicate code execution time is less than the non-predicate code execution time, or not to convert

Step 4: Finally, we modify the cond_exec_process_ if_block() function and add determine statement to the function

EXPERIMENT AND CONCLUSION

Experimental environment: a server for HP's Itanium 2, dual Itanium 2 processors, 1000 MHZ frequency, 2G RAM, running the operating system for Linux version 2.4.18-e.12 SMP. Based on GCC 3.4.0 version, has improved its if-conversion algorithm. To test the presented the improved algorithm, we have tested in two cases: 1, GCC 3.4.0 of the if-conversion; 2, based on an improved if-conversion algorithm.

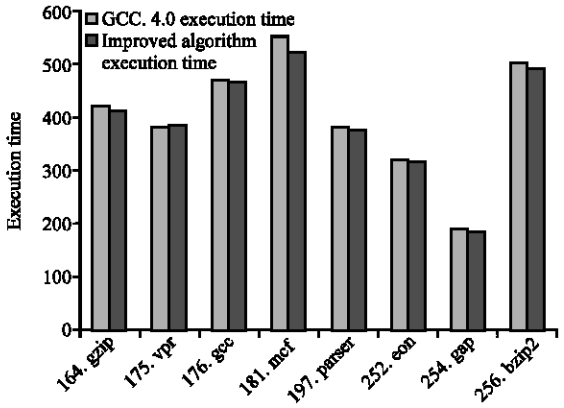

Fig. 2: Performance comparison of GCC3.4.0 and improved algorithm

We can see that, when using the GCC3.4.0 to compile, IF block is converted to predicate instructions, in this way, no matter how the conditions of branch are changed, it will implement all of the instructions after being incorporated, while using improved if-conversion algorithms, due to the branch path is imbalance and the transfer probability is heavily tilted towards ELSE block, worth_if_invert() function will return FALSE, not to implement if- conversion, so that only a few instructions in the THEN block are implemented, which can improve application performance. We will put the above program segment into a sufficient number of cycles for testing its execution time, experiments show that the improved algorithm can significantly improve program performance, lead to program execution time the reduction of 12.4%. Some of the results are obtained by testing with the SPEC CPU2000.

When compared with GCC 3.4.0, the improved algorithm leads to the overall performance improvement of average 1.12%, the performance of fixed-point programs are on the rise, the overall performance of fixed-point program is increased by 1.09%. The detailed results are shown in Fig. 2 and it shows that the proposed improvement of the if-conversion algorithm can improve program performance, especially the fixed-point program performance for IA-64 architecture.

ACKNOWLEDGMENTS

The study is supported by Scientific Research Fund of Hunan Provincial Education Department of China (08B014) and Scientific Research Fund of Department of Science and Technology of Hunan Provincial, China (2008GK3134).

REFERENCES

Allen, J.R., K. Kennedy, C. Porterfield and J. Warren, 1983. Conversion of control dependence to data dependence. Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Jan. 24-26, Austin, Texas, pp: 177-189.

August, D.I., W.W. Hwu and S.A. Mahlke, 1997. A framework for balancing control flow and predication. Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, Dec. 1-3, Research Triangle Park, North Carolina, USA., pp: 92-103.

Bringmann, R.A., S.A. Mahlke, R.E. Hank, J.C. Gyllenhaal and W.W. Hwu, 1993. Speculative execution exception recovery using write-back suppression. Proceedings of the 26th Annual International Symposium on Microarchitecture, Dec. 1-3, Austin, Texas, USA., pp: 214-223.

Bruel, C., 2009. If-conversion for embedded VLIW architectures. Int. J. Embedded Syst., 4: 2-16.

Mahlke, S.A., D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, 1992. Effective compiler support for predicated execution using the hyperblock. Proceedings of the 25th Annual International Symposium on Microarchitecture, Dec. 1-4, Portland, Oregon, USA., pp: 45-54.

Mahlke, S.A., R.E. Hank, R.A. Bringmann, J.C. Gyllenhaal, D.M. Gallagher and W.W. Hwu, 1994. Characterizing the impact of predicated execution on branch prediction. Proceedings of the 27th Annual International Symposium on Microarchitecture, Nov. 30-Dec. 2, San Jose, California, USA., pp: 217-227.

Mahlke, S.A., R.E. Hank, J.E. McCormick, D.I. August and W.W. Hwu, 1995. A comparison of full and partial predicated execution support for ILP processors. Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 22-24, S. Margherita Ligure, Italy, pp: 138-150.

Mantripragada, S. and A. Nicolau, 2000. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. Proceedings of the 14th International Conference on Supercomputing, May 8-11, Santa Fe, New Mexico, USA., pp: 206-214.

Quinones, E., J.M. Parcerisa and A. Gonzalez, 2006. Selective predicate prediction for out-of-order processors. Proceedings of the 20th Annual International Conference on Supercomputing, June 28-July 01, Cairns, Queensland, Australia, pp: 46-54.

Quinones, E., J.M. Parcerisa and A. Gonzailez, 2007. Improving branch prediction and predicated execution in out-of-order processors. Proceedings of the IEEE 13th International Symposium on High-Performance Computer Architecture, Feb. 10-14, Arizona, USA., pp: 75-84.

Wang, F.Q., Y. Li and Z.X. Zhang, 2009. A register allocation algorithm based on predicate analysis system. Proceedings of the WASE International Conference on Information Engineering, July 10-11, Taiyuan, Shanxi, China, pp: 78-81.

Yan, J. and W. Zhang, 2008a. A time-predictable VLIW processor and its compiler support. Real-Time Syst., 38: 67-84.

Yan, J. and W. Zhang, 2008b. Exploiting virtual registers to reduce pressure on real registers. ACM Trans. Architecture Code Optimization, 4: 1-18.