Available online at www.elixirpublishers.com (Elixir International Journal)

Computer Science and Engineering

Elixir Comp. Sci. & Engg. 38 (2011) 4477-4480

ISSN: 2229-712X

Learning-based compiler level optimization of branching statement layout using execution patterns and dynamic code reordering

Sarath Chandran, K.R., Tesslyn Antony, Sruti and S.Mathuri, G

Department of Computer Science and Engineering, Sri Sivasubramania Nadar College of Engineering, Kalavakkam 603110, TN, India.

ARTICLE INFO

Article history:
Received: 22 August 2011;
Received in revised form:
26 August 2011;
Accepted: 31 August 2011;

Keywords
Branching,
Reordering,
Layout,
Optimization,
Compiler,
Execution patterns.

ABSTRACT

Code layout is an important factor that determines the performance of any application. For branching intensive loops where decisions have to be made among several branching paths (as in real time systems), an optimized layout of the conditional statements can increase the performance largely. Current methods can predict branches dynamically using speculative execution which can be resource intensive. Static branch prediction techniques are not as accurate. In this work a compiler based optimization for branching instructions by code reordering has been proposed. The proposed design consists of a code reordering component that along with the compiler can dynamically generate layout-optimized code, by reordering the conditions in the source program. The reordering is done base on dynamic run-time execution patterns. Based on the current execution pattern and the history, the most optimal program can be run, minimizing evaluation of conditions.

Introduction

Fetch performance broadly depends on three factors: the number of instruction cache misses, the width of instructions fetched each cycle, and the branch prediction accuracy. The first two factors determine the speed at which instructions are provided to the processor, the third determines the quality of the instruction provided.

Code reordering techniques are a known approach to the first two factors. The number of instruction cache misses depends on the code layout [6]. The performance loss due to branch instructions was first approached with static branch predictors, which always predicted the same outcome (always taken, or always as not taken) for a given branch. This prediction was obtained using very simple heuristics[1], static analysis[2], or profile information[3,4] and then moved on to dynamic predictors [1,5].

The current methods use speculative prediction to initially set up the supporting database (branch history table). This involves considerable resources. Also, the above said methods do not guarantee that unnecessary branches are not evaluated at the runtime, and cache miss rate is still considerably high. Further, when consequent executions of the branch statements form a pattern, the pattern behavior is not completely exploited. There is still scope left for dynamically changing the source code or the object code generated by the compiler based on the prediction history and the patterns derived from the execution.

This paper proposes an improvement at the source code level for optimizing the code layout. A global data structure maintains the history of branch execution. This branch execution history is used to reorder the source code dynamically. At regular intervals, the efficiency of the currently running program in the immediate past few runs is being checked. If the efficiency is not in the acceptable range, the source program is reordered. This is done based on the execution statistics, such that the most frequently taken branches are at the top. This new source program is then compiled and then the new program starts running instead of the old (unordered) one. This can improve the performance by minimizing the evaluation of not-taken branches.

An example application scenario

The application considered to demonstrate the concept is a condition-based equipment maintenance system. The input program is a system that monitors industrial equipments and detects faults in them. This program continuously keeps monitoring the equipments for any faults. When an equipment is found to be at fault, the module within the equipment that has faulted has to be located.

In such a scenario, all the equipments and all modules have to be monitored, but the most frequently faulting ones need to be monitored first. This makes fault isolation efficient. This requires reordering of the conditions periodically, based on the previous execution of the conditions, assuming that the equipment and the module that has faulted frequently in the past is likely to fault again.

A specific case of mutually independent branches that need to be executed in a sequence is considered here, but the idea can be extended to all kinds of branching statements.

Paper Structure

The reminder of this paper is structured as follows. Section 2 contains a brief outline of the existing work in both branch prediction and code layout optimizations. Section 3 provides the system design and explains each step of the process in detail. Section 4 deals with results and performance evaluation. Section 5 deals with conclusion and provides future directions and enhancements to the proposed system.

Existing Work

The related work can be classified into two regions, one on branch prediction and the other on layout optimization. Code

Tele:
E-mail addresses: tesslyn7104@cse.ssn.edu.in

layout optimizations mainly target the optimal layout of basic blocks while branch prediction techniques aim to improve the fetch performance.

Branch Prediction Schemes

To determine the outcome of a branch before it is known, processor may employ a branch prediction scheme. Branch prediction schemes are classified either as static or dynamic depending on when the branch's predicted outcome is determined.

In a static branch prediction scheme, the compiler predicts the same outcome for a branch, for example, as always taken [12]. Since this decision is made when a program is compiled, the branch's predicted outcome does not change. Two common variations of the static branch prediction scheme use profile-based and program-based heuristics. [1, 2, 3]

In a dynamic branch prediction scheme, the processor predicts the outcome of a branch at run-time based on its recent behavior. Unlike static prediction schemes, dynamic branch prediction schemes can adapt to changes in the behavior of a particular branch.

A common implementation of a dynamic branch prediction scheme is one that employs a branch prediction table (BPT) whose entries are n-bit counter values indexed by the lower portion of the branch's address. This scheme works by incrementing the value of the corresponding n-bit counter every time the branch is taken, and decrementing the counter every time the branch is not taken. The counter is neither incremented nor decremented when it reaches its maximum or minimum value, respectively. When using an n-bit counter to predict the outcome of the branch, if the most-significant bit of the value represented by the counter is 1, the branch is predicted as taken; otherwise, the branch is predicted as not taken.

Two-level adaptive branch predictors keep two levels of data about the branch behavior. The Level 1 table or the Branch History Table (BHT) keeps information about the past branch outcomes. This table indexes into the Level 2 table, composed of two-bit saturating counters managed as in the bimodal predictor. The Level 2 table is usually referred to as the Pattern History Table (PHT). A combination of both global and local execution information yields better accuracy.

Prediction Methods for Branch Instructions of Different Behaviors

The main idea here is that various prediction methods can be used for branch instructions of different behaviors. The compiler profiles and classifies all branch instructions into four different categories according to the behavioral history of the program during the profiling. The compiler encodes the classifications in 2-bit "hints". Then, at runtime, the hardware treats the branch instructions in different ways according to the compiler's hints.

In profiling, the compiler analyzes and characterizes the behavioral history of every branch instruction. For branch instructions with behavioral histories of almost always "taken" or almost always "not-taken", the hardware branch prediction mechanism will not be employed. The processor will use the compiler's static predictions to direct the branches.

The branch instructions other than the above two categories are classified into two categories: those with regular history patterns, and those with irregular history patterns. A hardware prediction scheme called switch counter has been proposed to predict such instructions with regular history patterns. The Switch-Counter can remember how many times the continuous "taken" or the continuous "not-taken" occurred when the last pattern switch happened. The branch predictions are made dynamically by the Switch Counter based on the values of the counters.

For instructions that do not follow such a regular pattern, a two-level adaptive predictor can be used.

Code Layout Optimization

Layout optimization aim to efficiently layout the routines and blocks in a program to effectively use the instruction cache. [6, 7, 8, 9, 10, 11]

One code layout optimization method is the Software Trace Cache (STC). The STC maps basic blocks so that sequentially executed basic blocks tend to be in consecutive memory positions, building basic block chains than may span multiple routines [13].

The effect of laying out basic blocks in a certain way and its effect on branch prediction has been dealt in great depth in [14]. However, this technique just uses the results from the initial profiling by running the program to obtain statistics, but does not take into consideration the dynamics of the system or program as it is running. This may provide valuable prediction hints in real-time systems.

Proposed Methodology

Here we propose a branch prediction and optimization component that reorganizes the source code such that the more probable conditions are evaluated before the ones with a lower probability. This can reduce the evaluation of not-taken branches thereby improving the performance.

A preprocessor component creates a global data structure which maintains the branching history for a user-specified (application-specific) part of a program. This can be specified using a delimiter. This component scans the input program and creates the necessary data structures based on the input program. The preprocessor is specific for a programming language. This component also expands the input source program so that it can update the data structure that keeps track of the branch execution. The probability of execution of these branches is updated dynamically as they get executed.

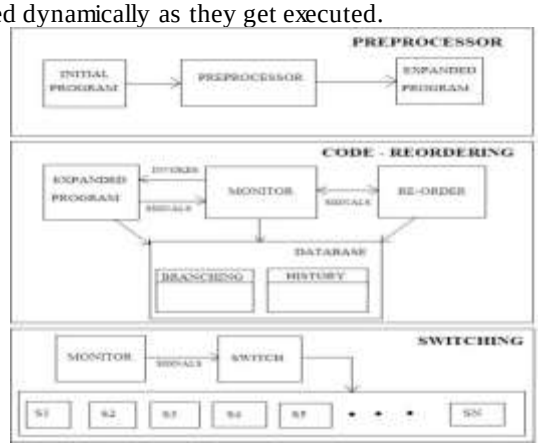

Figure 1: Architecture

At specified intervals, the current execution profile is compared against the global history data structure. If the currently running object code is not optimal, the source program is reordered based on the execution of the conditions and is recompiled. This reordered program now replaces the currently running program. A monitor component takes care of making the decision whether reordering, or just switching to another program needs to take place. The reorder module takes as input the expanded C program (output of the preprocessor) and creates a new expanded program with the conditions in the decreasing

order of the probability of execution. The reordering is done at each level of nesting. This reordering is done when the monitor signals this module to reorder. On completion of the reordering this module signals back the monitor module.

New executables are generated from the source program each time reordering is done. Hence an object code (executable file) for each condition execution pattern is obtained.

Over a period of time, as the program runs for a considerably large number of times, we can also arrive at the most optimal ordering of conditions. A pool of various source programs and the compiled object programs corresponding to the various execution profiles are obtained and a list of possible branching profiles get accumulated in the History table. As a result, the need for reordering minimizes after considerable time and just switching to the correct program happens.

Data Structures Used

The proposed system maintains two data structures. One to maintain the earlier patterns and the matching program for that pattern. This database table is termed the History and is created by the preprocessor. A new record gets inserted into this table whenever a new pattern of branching is obtained from the running program.

The other data structure, here called the branching table, is initially created by the system developer. It keeps track of the count of each branch being taken as the program is running. This table, in real time scenarios could be the register file of the processor. The preprocessor allocates the space for each condition it encounters. The value of the count field is initialized to 0. This gets updated as the expanded program runs.

The criteria for switching

The reordering is done whenever the pattern in the most recent run differs from the earlier pattern beyond a certain threshold value. This difference calculation that has been implemented is the sum of the differences obtained for each branch. This is demonstrated below.

If B_1, B_2, $B_3 \ldots B_n$ are n non-exclusive branching instructions, then the count that each of these branches are taken is recorded as the branches are evaluated. If at the end of a run, these values are $c_1, c_2, c_3 \ldots c_n$ and the corresponding values in the History table for the currently running program are $h_2, h_3, h_4, \ldots h_n$, the difference is given by

$$\text{Difference} = \sum |h_i - c_i| \quad \text{for } i=1, 2 \ldots n \qquad \text{(Eq. 1)}$$

If this difference computed from equation Eq. 1 is beyond a fixed threshold value, the currently running program is not the most efficient. So, the monitor module looks for a match for a program from the History table. If found, the monitor switches to that program, otherwise reordering is done. Then, we switch to the newly reordered and compiled program. If the difference is below the threshold, the currently running program has an acceptable efficiency and it continues running. Here, the choice of the threshold value plays a crucial role. There are also other factors discussed in the next section.

Evaluation and Results

To evaluate the proposed technique as a measure of time the time taken for evaluating the conditions, each condition evaluation has been fixed to consume a time of 1 second. Hence, the time taken for running is directly proportional to the number of conditions evaluated. The earlier the desired condition is met, the lesser the number of unnecessary conditions that get evaluated and hence lesser time. This method provides a quantitative method to analyze the performance of the system.

RUN #	RUNNING PGM	TIME TAKEN	DIFFERENCE (threshold : 35)	Improvement %
1	sample	379		
2	sample1	315	0	16.89
3	sample1	326	30	13.98
4	sample1	400	60	-5.54
5	sample2	329	49	13.19
6	sample1	315	30	16.89
7	sample1	326	0	13.98
8	sample1	398	48	-5.01
9	sample2	396	214	-4.49
10	sample3	309	0	18.47
11	sample3	312	60	17.68
12	sample3	350	48	8.07
13	sample4	350	182	8.33
14	sample5	354	98	6.60
15	sample6	297	0	21.64
16	sample6	303	30	20.05
17	sample6	326	38	13.98
18	sample6	292	43	22.96
19	sample7	306	48	19.53
20	sample6	379	98	0.26
	OVERHEAD (REORDER & SWITCHING)	42		
	Total time	6812		

Figure 2: Results from 20 runs of the program with threshold 35

The table in Figure 2 shows the results obtained from the proposed system for a program with 3 independent conditions each with 3 (or 4) nested independent conditions after 20 runs with constrained random input.

With the crude measure for reordering as specified by equation Eq.1 and the time interval chosen as 100 iterations of the program, the performance showed slight variations.

It can be inferred from the table in Figure 2 that the proposed systemdoes not predict right and layout the conditions in the right way every time. On some occasions, there can be a downfall in the performance, as indicated by the negative values of performance in Figure 2. However an overall gain of about 8% was achieved. This gain has been obtained in spite of the overhead in reordering and switching. The running of the original program for the same input for 10 runs took 7405 seconds as against the proposed system, which took 6812 seconds. It can thus be seen that he system can intelligently reorder the conditions to achieve an overall gain. The names sample1, sample2, etc are the names of the various reordered versions of the original program.

Factors affecting the performance of the proposed technique

The performance of this proposed methodology is governed by 1) The reordering and switching criteria 2) how often the efficiency of the currently running program is checked and the decision on reordering/switching made and 3) the threshold value.

The reordering and switching criteria

The formula for computing the difference between the currently running program and the pattern in the History database is given in Equation Eq.1. However, this formula for computation of the difference is crude and has its limitations in terms of performance and validity. The nesting of conditions has to be taken into consideration in deciding this measure. This measure has to be optimized to arrive at more accurate predictions. The more accurate this formula is, the more efficient the system. However, too much of complexity in the calculation of this measure can affect the performance in the adverse manner.

The frequency of decision-making

The running program's efficiency has to be monitored at regular intervals. However this value can't be too low or too high. Frequent evaluation can increase the overhead inadvertently. Doing it rarely destroys the whole aim of optimization. This depends totally on the behavior of the application and the size of the program.

The threshold value

The monitor module makes a decision of whether the currently running program is efficient or not, based on the

difference between the pattern in the History table and the most recent run. However, difference can be acceptable if it is below a certain threshold level.

If this threshold level is too large, the system may be too lenient and if the threshold value is too low, it becomes idealistic. Also lower threshold values may mean frequent switching and reordering hence may not be desirable.

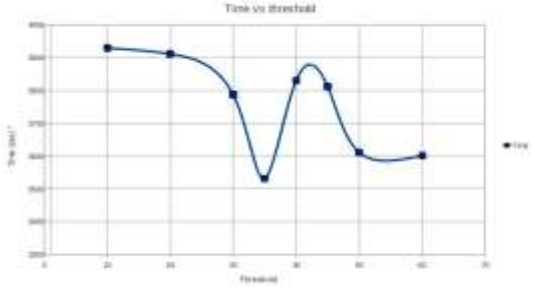

Figure 3: Time Vs Threshold

A graph showing the variation in performance with the threshold value is depicted in Figure 2. This has been obtained when the reordering frequency is 100 iterations of the program's conditions. The following inferences can be made from the graph shown in Figure 2.

• For low values of threshold, the time taken for execution is high, i.e., the performance is low. This is because of the high overhead incurred in reordering and switching. The system behaves idealistic.

• For high values, the performance is low; this is because the system is too lenient and allows for large deviations from the pattern in the History table.

• For very high values of the threshold, there is a substantial decrease in time; this is because the overhead in switching or reordering is reduced. The system always predicts that the program running is the most efficient.

• It can be seen that at a certain point (here at threshold value=35), the time taken is the least.

The choice of the threshold value has to be arrived at after careful experimentation and from domain expertise for which the application is being run.

However other techniques could be used to fix this value. This is discussed in the next section.

Future Directions

In this paper, a novel method has been suggested for improving the performance of a branching intensive system. The performance gain thereby recorded has been discussed in the previous sections. Here we suggest some possible enhancements that can be made to the method proposed.

As stated in earlier section, the threshold value, the criterion for reordering and switching can further be optimized. For a given application, suitable learning algorithms can be adopted to learn the way the branches are taken.

The computation of the difference for the reordering criteria can be replaced by other statistical measures.

Also, for huge real-time systems, training can be incorporated using fuzzy systems that optimize this reordering criterion.

Similarly, the threshold value can be fixed by swarm intelligence techniques, making the system an intelligent one. This also increases the accuracy in the prediction and reordering.

The frequency at which the efficiency of the currently running program is evaluated can also be fixed using similar techniques.

References

[1] J. E. Smith. A study of branch prediction strategies. Proceedings o the 8th Annual Intl. Symposium on Computer Architecture, pages 135-148, 1981.

[2] T. Ball and J. R. Lams. Branch prediction for free. Proceedings of the ACM SIGPLAN Conference on Programming Language Design And Implementation, pages 300-313, June 1993.

[3] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 85-95, 1992.

[4] B. Calder, D. Grunwald, and D. Lindsay. Corpus-based static branch prediction. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 79-92, 1995.

[5] T. Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. Proceedings o the 24th Annual ACMIIEEE Intl.Symposium on Microarchitecture, pages 51-61, 1991.

[6] K. Pettis and R. C. Hansen. Profile guided code positioning. Proc. ACM SIGPLAN Conf on Programming Language Design and Implementation, pages 16-27, June 1990.

[7] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. Proc. ACM SIGPLAN Conf on Programming Language Design and Implementation, pages 171-182, June 1997

[8] N. Cloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. Proceedings of the 30th Annual ACMIIEEE Intl. Symposium on Microarchitecture, pages 303-313, Dec. 1997.

[9] W.-M. Hwu and P. P. Chang. Achieving high instructioncache performance with an optimizing compiler. Proceedings of the 16th Annual Intl. Symposium on Computer Architecture, pages 242-25 l, June 1989.

[10] J. Torrellas, C. Xia, and R. Daigle. Optimizing instructioncache performance for operating system intensive workloads. Proceedings o the 1st Intl. Conference on High PerformanceComputer Architecture, pages 360-369, Jan. 1995

[11] A. Ramirez, J. L. Larriba-Pey, C. Navarm, J. Torrellas, and. Valero. Software trace cache. Proceedings of the 13thIntl. Conference on Supercomputing, June 1999.

[12] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pages 242-25 1, Oct. 1994.

[13] A. Ramirez, J. L. Larriba-Pey and M. Valero, The Effect ofCode Reordering on Branch Prediction, Proceedings of the2000 International Conference on Parallel Architectures And Compilation Techniques, pages 189-198, 2000.