Improving Semi-static Branch Prediction by Code Replication

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
andi@mips.complang.tuwien.ac.at

Abstract

Speculative execution on superscalar processors demands substantially better branch prediction than what has been previously available. In this paper we present code replication techniques that improve the accurracy of semi-static branch prediction to a level comparable to dynamic branch prediction schemes. Our technique uses profiling to collect information about the correlation between different branches and about the correlation between the subsequent outcomes of a single branch. Using this information and code replication the outcome of branches is represented in the program state. Our experiments have shown that the misprediction rate can almost be halved while the code size is increased by one third.

1 Introduction

Branch prediction forecasts the direction a conditional branch will take. It reduces the branch penalty in a processor and is a basis for the application of compiler optimization techniques. In this paper we are mainly interested in the latter use, since we will apply branch prediction to compiler based speculative execution and other code motion techniques. *Static branch prediction* relies only on information that is obtained by static analysis of the program. *Semi-static branch prediction* uses profiling [Wal91] to predict the branch direction. *Dynamic branch prediction* saves branch directions in hardware history registers and tables and uses this information to predict branches during run time. The misprediction rate of semi-static branch prediction strategies is about half that of the best static branch prediction strategies [BL93]. The misprediction rate of the best dynamic branch prediction strategies is about half that of semi-static branch prediction strategies [YN93].

Compile time optimizations like code motion and speculative execution rely on an accurate branch prediction strategy. For many optimizations existing branch prediction strategies are not sufficient. So we looked for a method to improve the accuracy of compile time branch prediction. Our approach replicates a piece of code, so that the branches in the replicated code pieces are more predictable than in the original code. This idea was inspired by the work of Pettis and Hanson [PH90], who use profiling for code positioning to improve cache behaviour, and by the work of Mueller and Whalley [MW92], who use code replication to avoid jumps.

Chapter 2 presents existing branch prediction methods. Chapter 3 contains a description of our profiling tool and presents the results of profiling our benchmark suite. Chapter 4 describes the techniques for compacting the collected history information in order to make it usable for semi-static branch prediction. Chapter 5 explains the code replication techniques and shows the effects on the code size.

2 Branch Prediction Strategies

2.1 Static Branch Prediction

Smith [Smi81] explored some simple heuristics and compared them with simple dynamic branch prediction strategies. He uses following static strategies:

- predict that all branches will be taken
- predict that only certain branch operation codes will be taken
- predict that all backward branches will be taken

The misprediction rate of these simple branch prediction strategies is about 30%, but some benchmark programs have a misprediction rate of 65%.

A more sophisticated implementation of static branch prediction has been done by Ball and Larus [BL93]. Their branch prediction strategy is based on a control flow analysis of the program to determine loops. Furthermore, the code surrounding a branch determines the kind of the branch. They applied different heuristics in different lexicograhic orders. The most successful order was:

Point	pointer comparison (predict not taken)
Call	avoid branches to blocks which call a subroutine
Opcode	decide the branch direction on the branch instruction opcode
Return	avoid branches to blocks which return from a subroutine
Store	avoid branches to blocks which contain a store instruction
Loop	predict that the loop branch will be taken
Guard	branch to a block which uses the operands of the branch

With this heuristics Ball and Larus reach an average misprediction rate of 20%, about twice the misprediction rate achieved by profile based branch prediction.

2.2 Semi-static Branch Prediction

Semi-static branch prediction is based on profiling [Wal91]. McFarling and Hennessy [MH86] were the first to suggest the use of profiling for branch prediction and gave some data. Fisher and Freudenberger [FF92] made a comprehensive study of branch prediction based on profiling. Instead of using the misprediction rate as a measure, they gave the average number of executed instructions per mispredicted branch for the programs of the SPEC benchmark suite. Furthermore they studied the influence of different datasets on the accuracy of the prediction. Fisher and Freudenberger report between 80% and 100% of the prediction rate for the best prediction using another dataset instead of the reference data set. The worst prediction varies between 50% and 100%.

2.3 Dynamic Branch Prediction

Dynamic branch prediction depends on information available only at run time. Simple strategies have been studied by Smith [Smi81]. Among other strategies he proposed the following strategies:

- predict that a branch will take the same direction as on its last execution
- associate a counter with a branch and decide on the counter value

The counter based strategy increments the counter using saturation arithmetic if the branch was taken and decrements it otherwise. If the counter value is in the upper half of the value range, the branch is predicted taken. If the value is in the lower half, the branch is predicted not taken. A two bit counter gives the best result. The misprediction rate is comparable to the profile based strategies.

Today the best dynamic branch prediction strategies are based on two levels of history information. One level of information represents the outcome of the last branches. It is usually implemented as a shift register in hardware. The second level of history information contains the different patterns of history register values combined with a two bit counter, which gives the branch prediction for this pattern. Yeh and Patt presented a strategy whith a history register for each branch and a pattern table for each branch [YN92]. Pan, So and Rahmeh presented a strategy with a single global history register and a pattern table for each branch [PSR92]. Since their strategy depends on the correlation of different branches, they called it branch correlation. Later Yeh and Patt studied all nine combinations of one global history register, a history register for a set of branches and a history register for each branch with one global pattern table, a pattern table for a set of branches or a pattern table for each branch [YN93]. The best strategy, a history register for each branch and a pattern table for a set of branches, achieved an average misprediction rate of about 3% having an implementation cost of 128K bits.

3 Collecting Branch Correlation Information

We are interested in a branch prediction strategy usable for predictions at compile time. So we tried to adapt the dynamic branch prediction strategies of [YN93] for semi-static branch prediction. In contrast to a dynamic branch strategy we are not restricted by the size of the history tables. So we used a pattern table for each branch. Furthermore, we used only history register schemes which are meaningful for code replication. A global history register means that a branch depends on other branches. We will call this strategy *correlated branch strategy*. A history register for each branch (local history register) means that this branch depends on previous executions of the same branch. We will call this scheme *loop branch strategy* and branches which use this scheme loop branches. For each pattern in the pattern table we predict the more frequent direction.

Since no existing profiling or trace tool fulfilled our needs we developed our own profiling and analysis tools. To get a trace of the executed branches we insert code in a program which writes trace information to a file. The trace information contains the branch number and the branch direction. In compressed form a trace of 5 million branches occupies about 1MB. In contrast to the QPT profiling and tracing tool [Lar93], which inserts trace instructions in the object code of a program, our tool inserts trace code in the assembly language source of a program. The advantage of our method is that address calculation and code relocation is done by the assembler. The disadvantage of our method is that we cannot trace system library functions. Furthermore, the trace tool does a control flow analysis and saves the description of branches, a control flow graph and loop information in a file. An analysis tool processes the trace and generates tables that describe the branch prediction accuracy and the effects on code size. Programs with tracing enabled are about three times slower than without. The analysis of the trace is done in a few seconds. A production version of the profiling tool will include the first part of the analysis tool which transforms the trace data into the pattern table. This enables profiling with an unlimited number of branches.

With these tools we evaluated a set of eight benchmark programs. Since especially integer programs need better branch prediction, we included only one floating point program. The programs are:

abalone	a board game employing alpha-beta search
c-compiler	the lcc compiler front end of Fraser & Hanson
compress	a file compression utility (SPEC)
ghostview	an X postscript previewer
predict	our profiling and trace tool
prolog	the minivip Prolog interpreter
scheduler	an instruction scheduler
doduc	hydrocode simulation (floating point) (SPEC)

These benchmark programs have been compiled with the C or Fortran compilers with optimization enabled. We traced the whole program up to a maximum of 10 million branch instructions. For the purpose of comparison we evaluated dynamic and semi-static branch prediction strategies:

last direction	a branch takes the same direction as the last time (dynamic)
2 bit counter	decide on the value of a 2 bit counter (dynamic)
two level 41K bit	a 1K entry 9 bit history register and a 16K entry pattern table with 2 bit counters (dynamic)
profile	predict the most frequent direction (semi-static)
6 bit correlation	predict using one global 6 bit history register (semi-static)
6 bit loop	use 6 bit history registers for every branch (semi-static)
9 bit loop	use 9 bit history registers for every branch (semi-static)
loop–correlation	the best of 6 bit correlation and 9 bit loop for each branch (semi-static)

Furthermore, we collected information about the static number of branches, the number of branches that were executed during the run of the program and the number of branches that could be improved by the loop-correlation strategy compared to profile branch prediction. Table 1 gives the results.

4 The Branch Prediction State Machine

Table 1 shows that branch prediction strategies using history information significantly improves the prediction accuracy. But how can this information be used in a semi-static branch prediction scheme? As an example, we consider a branch embedded in a loop that is alternating between taken and not taken. Figure 1 shows the flow graph of an example loop.

Basic block "1" contains the branch that can be improved by code replication. The loop is duplicated and the branch switches between the two copies. Each copy of the loop represents a state that remembers the branch direction of the previous execution of this branch. State "0" has the meaning that the last time the branch was not taken, State "1" has the meaning that the branch was taken. In both copies of the loop the branch is now predicted correctly 100% of the time. Basic blocks "2b" and "3a" are not replicated. Since there is no path to them they have been discarded.

This is the scheme we implement for 1 bit history information. Unfortunately, we cannot use this scheme for longer histories, since this would increase the program size too much. A replicated loop representing a 9 bit history would result in 512 copies of the loop. On the other hand information in the history tables is very sparse. Table 2 gives the percentage of the pattern table fill rate. Only between 0.6 and 21 percent of the 9 bit pattern table entrys of the executed branches are used. We therefore construct branch prediction state machines for loop branches and correlated branches that

	aba-lone	c-com-piler	com-press	ghost-view	pre-dict	pro-log	sche-duler	do-duc
last direction	22.9	18.4	18.0	1.27	11.8	14.5	13.8	7.53
2 bit counter	20.8	14.9	14.5	1.20	7.70	11.3	10.9	3.87
two level 41K bit	6.82	12.7	13.7	1.85	4.64	10.7	11.1	0.89
profile	18.7	13.5	17.2	1.27	7.97	11.3	13.6	3.99
6 bit correlation	11.6	8.67	14.2	0.31	6.30	7.53	9.28	1.64
6 bit loop	9.67	9.33	13.4	0.41	5.22	7.18	7.76	1.55
9 bit loop	6.89	7.79	13.0	0.36	4.12	5.72	5.97	1.33
loop–correlation	6.47	6.97	12.6	0.21	3.72	5.35	5.02	1.11
static branches	496	3645	170	1399	451	2324	490	665
executed branches	311	2183	70	517	345	819	431	487
improved branches	209	658	13	84	92	320	242	74

Table 1: misprediction rates of different branch prediction strategies in percent

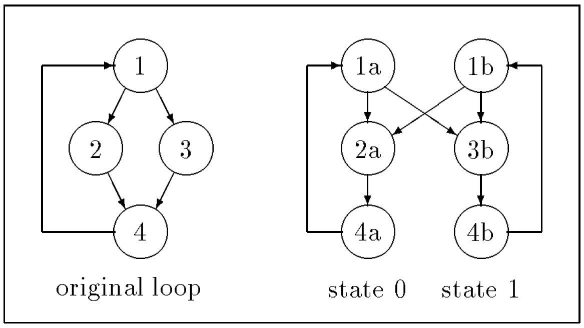

Figure 1: flow graph of an intra loop branch and a 2 state machine

	aba-lone	c-com-piler	com-press	ghost-view	pre-dict	pro-log	sche-duler	do-duc
1 bit history	92.6	84.2	67.9	68.1	82.9	85.0	86.5	84.1
2 bit history	84.6	66.1	46.1	42.9	64.6	68.6	75.6	60.9
3 bit history	74.4	49.5	31.4	25.0	48.2	53.8	65.8	40.4
4 bit history	62.9	36.1	21.8	14.1	35.8	41.3	56.8	25.2
5 bit history	52.1	25.8	16.0	7.85	26.7	31.3	48.4	15.1
6 bit history	42.5	18.2	12.3	4.27	20.0	23.2	39.6	8.79
7 bit history	34.3	12.7	10.1	2.30	14.8	16.6	30.8	5.05
8 bit history	27.1	8.70	8.77	1.22	10.6	11.5	22.7	2.85
9 bit history	21.0	5.89	8.00	0.65	7.28	7.56	15.8	1.58

Table 2: fill rate of the history tables in percent

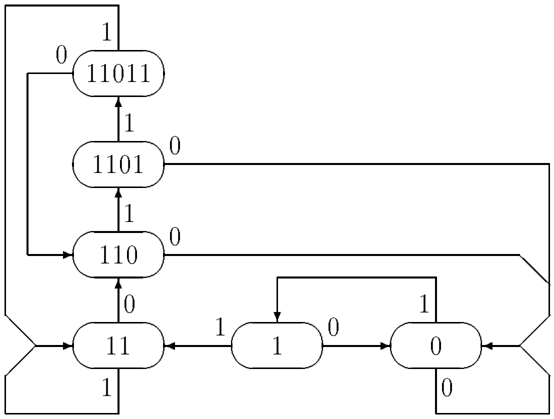

Figure 2: branch prediction state machine 5/1

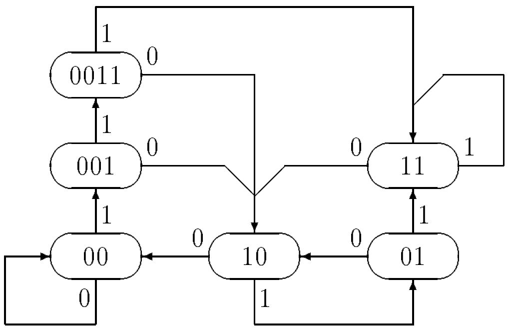

Figure 3: branch prediction state machine 4/2

use fewer states but have nearly the same prediction accuracy. Furthermore, we divide loop branches in *intra loop branches* that occur inside a loop, and *exit loop branches* which may leave the loop.

4.1 Intra Loop Branches

Intra loop branches do not leave the loop. For intra loop branches a state represents the last n branch directions of previous iterations of the loop. It is necessary that each state can be reached from another state and via other states from the initial state, i.e. the state used in the first iteration. An example for a valid intra loop state machine is the state machine 5/1 in figure 2. A state is a copy of the loop, e.g. the loop of figure 1. The digits in the state describe the directions of the last branches. "0" means that the branch was not taken, "1" means that the branch was taken. The rightmost digit represent the direction of the last iteration. State "1" and state "0" representing only one branch direction are used as catch-all states if more specialized states do not

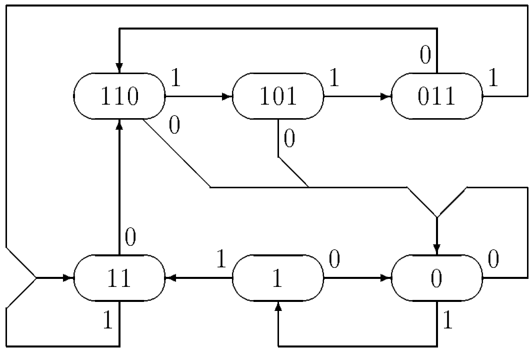

Figure 4: branch prediction state machine 3/3/3/1

match. An arc labeled "0" describes the state transition if a branch was not taken. An arc labeled "1" describes the transition if the branch was not taken. Each state can be used as an initial state for the first iteration.

It is also possible to construct state machines where the smallest state represents the last two directions of the branch. In this case there are four catch-all states which match when specialized states fail. Figure 3 shows such a state machine.

Another example is the state machine of figure 4 that has a loop of three states with size three (the states "110", "101" and "011"). This loop can be reached from the catch all state "1" via the state "11". Although a state machine as in figure 2 usually gives the best result for a 6 state machine, we make an exhaustive search in the pattern table to find the best state machine. For each 9 bit pattern we collected the number of taken and not taken branches. This information is used to compute the number of taken and not taken branches for all shorter patterns. Adding now the counts for the more frequent direction of all states of a branch prediction state machine and taking care that patterns are counted not more than once, we get the number of correct predicted branches for the state machine. The branch prediction state machine with the highest number of correct predicted branches is selected. Table 3 shows the misprediction rates of the benchmark programs, where only a reduced state machine is used instead of the complete pattern table. A state machine with 2 states implements exactly the 1 bit history scheme, so no data for a two bit state machine is presented. A state machine with n states usually needs the history information up to a length $n-1$. So we grouped always a history with n bits with a $n+1$ state machine to show the effect of accuracy loss.

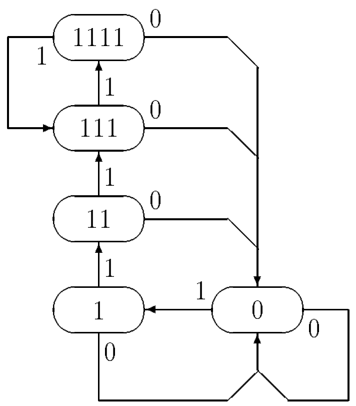

Figure 5: loop exit branch machine

4.2 Loop Exit Branches

Loop exit branches are those branches that go from inside the loop to the surrounding code. Therefore, they are more restricted than intra loop branches. Loop exit branch state machines have one initial state that represents the loop exit in the last execution (state "0"). The other states represent the loop iterations (the states with an increasing number of "1"). Figure 5 shows a loop exit branch machine. It is not necessary that a loop remains in the state with the longest history information. If a loop has a high probability of an even or odd number of iterations, the loop would change between the two states with the longest history information as shown in figure 5. Such a behavior can be detected using longer history information. The misprediction rates of loop exit state machines are shown in table 3, too.

4.3 Correlated Branches

The state machines for correlated branches are different from the state machines for loop branches. The states of a loop branch depend on each other, the states of a correlated branch are independent. A state in a correlated branch state machine represents a path from correlated branches to the branch to be predicted. The correlated branch state machine is the set of those paths which give the lowest misprediction rate. One state covers the case where the control flow matches none of the paths. The code replication for correlated branches is similar to [MW92]. The difference is that our aim was to save information about the branch direction, whereas the aim of Mueller and Whalley was to avoid unconditional jumps. Table 4 gives the misprediction rate of correlated branches. We used a maximum path length of $n-1$ for an n state machine to keep the size of the replicated code small. The table shows that the correlation information can be compacted with very small loss.

5 Branch Prediction and Program Size

To find intra loop, exit loop and other branches, a control flow analysis of the program is performed. The program is divided into basic blocks and a control flow graph is constructed. Natural loop analysis [ASU86] is performed. With this information the state machines for loop exit and intra loop branches are selected. For all branches all predecessors with a path length less than the size of the state machine are collected, and the correlated branch state machines are selected. The best available strategy for each branch is chosen. Branch prediction is deteriorated because only half of the branches belong to a loop and because the state machines reduce the accuracy of the branch prediction. Table 5 gives the misprediction rates of the benchmark programs for the different sizes of the state machine ignoring the effects on program size.

To study the effect of code replication on the code size, we added states to a program and measured how the code size increased and the misprediction rate was reduced. The states were added in such an order that the state that predicted the largest number of branches and that increased the code size by the smallest amount was choosen first. The first states reduce the misprediction rate substantially, later ones increase the code size considerably. The appendix shows graphs of code size versus misprediction rate. The asymptotic behaviour is explained by the dependences between branches. If branches are in different loops, the number of states is only added. If branches are in the same loop, the number of states must be multiplied. Some programs reach the best misprediction rate within a code size increase of a factor 1.5, other programs would increase the code size more than thousand times. With the exception of abalone every program comes close to the best achieveable misprediction rate by increasing the code size by less than 30%.

An optimizer using code replication for improving branch prediction will not improve the whole program, but only certain branches. In general, an optimization technique like branch aligning or speculative execution is not applied to a branch whose prediction accuracy is low. If code replication improves the accuracy of the prediction for this branch, such an optimization can be applied. A cost function will calculate whether the increase in code size (negative impact on instruction cache miss rate) is worth the gain in execution time.

	aba-lone	c-com-piler	com-press	ghost-view	pre-dict	pro-log	sche-duler	do-duc
profile	18.7	13.5	17.2	1.27	7.97	11.3	13.6	3.99
1 bit	16.9	12.4	16.0	0.63	7.36	10.0	10.4	2.27
2 bit	15.4	11.9	14.6	0.51	7.07	9.23	9.91	1.57
3 states loop	15.4	12.0	14.7	0.52	7.10	9.26	9.94	1.57
3 states exit	16.1	12.2	15.7	0.58	7.23	9.66	10.3	2.02
3 bit	13.6	10.8	14.2	0.48	6.72	8.67	9.67	1.56
4 states loop	14.4	11.3	14.2	0.48	6.83	8.84	9.73	1.56
4 states exit	15.4	12.0	15.7	0.57	7.08	9.49	10.2	2.02
4 bit	12.1	10.3	13.7	0.45	6.23	8.13	9.32	1.55
5 states loop	13.0	10.7	14.0	0.46	6.41	8.51	9.53	1.55
5 states exit	15.1	11.8	15.7	0.57	6.84	9.44	10.2	2.02
5 bit	10.7	9.81	13.5	0.42	5.68	7.67	9.02	1.55
6 states loop	12.3	10.5	13.8	0.44	6.26	8.21	9.42	1.55
6 states exit	14.7	11.7	15.7	0.56	6.75	9.39	10.2	2.02
6 bit	9.67	9.33	13.4	0.41	5.22	7.18	7.76	1.55
7 states loop	11.3	10.3	13.6	0.42	5.96	7.98	8.49	1.55
7 states exit	14.6	11.6	15.7	0.56	6.58	9.38	9.37	2.02
7 bit	8.76	8.78	13.2	0.40	4.74	6.74	7.21	1.55
8 states loop	10.7	10.1	13.6	0.42	5.76	7.84	8.32	1.55
8 states exit	14.4	11.6	15.6	0.55	6.48	9.37	9.30	2.02
8 bit	7.68	8.34	13.1	0.37	4.41	6.26	6.61	1.34
9 states loop	10.1	9.91	13.5	0.41	5.66	7.73	8.24	1.34
9 states exit	14.1	11.6	15.6	0.53	6.46	9.36	9.29	2.02
9 bit	6.89	7.79	13.0	0.36	4.12	5.72	5.97	1.34
10 states loop	9.64	9.71	13.4	0.39	5.51	7.61	8.17	1.34
10 states exit	13.8	11.6	15.6	0.53	6.46	9.35	9.28	2.02

Table 3: misprediction rates of loop and loop exit branches in percent

	aba-lone	c-com-piler	com-press	ghost-view	pre-dict	pro-log	sche-duler	do-duc
profile	18.7	13.5	17.2	1.27	7.97	11.3	13.6	3.99
1 bit	18.6	12.8	17.1	1.22	7.75	10.6	12.0	2.52
2 states	18.6	12.8	17.1	1.22	7.75	10.6	12.0	2.52
2 bit	16.2	11.7	17.1	1.05	7.37	9.89	11.1	1.95
3 states	16.2	11.7	17.1	1.05	7.37	9.90	11.1	1.95
3 bit	15.1	10.8	15.9	0.62	6.95	9.40	10.8	1.83
4 states	15.2	10.8	15.9	0.62	7.03	9.43	10.9	1.83
4 bit	13.9	9.58	15.9	0.38	6.52	8.66	10.1	1.75
5 states	14.3	9.90	15.9	0.38	6.68	8.77	10.1	1.75
5 bit	12.7	9.12	15.6	0.34	6.39	8.08	9.61	1.70
6 states	13.4	9.65	15.6	0.35	6.48	8.31	9.72	1.70
6 bit	11.6	8.67	14.2	0.32	6.30	7.53	9.28	1.64
7 states	12.6	9.42	14.2	0.34	6.40	8.02	9.50	1.64

Table 4: misprediction rates of correlated branches in percent

	aba-lone	c-com-piler	com-press	ghost-view	pre-dict	pro-log	sche-duler	do-duc
profile	18.7	13.5	17.2	1.27	7.97	11.3	13.6	3.99
2 states	16.8	12.5	16.1	0.82	7.58	9.86	11.8	2.33
3 states	14.6	11.1	14.9	0.59	7.26	9.05	10.8	1.62
4 states	13.2	10.2	14.0	0.44	6.80	8.59	10.6	1.57
5 states	11.8	9.29	13.9	0.34	6.24	7.86	9.87	1.56
6 states	11.0	9.10	13.5	0.31	6.03	7.50	9.47	1.51
7 states	10.0	8.91	13.4	0.30	5.76	7.29	8.47	1.48
8 states	9.64	8.80	13.4	0.30	5.60	7.24	8.38	1.48
9 states	9.26	8.71	13.3	0.30	5.56	7.18	8.34	1.28
10 states	8.99	8.68	13.3	0.29	5.45	7.12	8.32	1.28

Table 5: best achieveable misprediction rates in percent

6 Further Work

A problem of our code replication scheme is that the code size is multiplied if more than one branch in a loop should be improved. A possible solution treats all branches of that loop at the same time and constructs a single state machine for all branches using a higher number of states. In that case the search for the optimal state machine must be replaced by a branch-and-bound search since the search time grows exponentially with the number of states.

Another work to be done is to measure the influence of different data sets on the misprediction rate. We assume that code replicated programs are more sensitive to different data sets than the original program. So the results of Fisher and Freudenberger are not fully applicable [FF92].

Furthermore we will use the improved semi-static branch prediction strategy for our global instruction scheduler and evaluate the effects on runtime and instruction cache behaviour.

7 Conclusion

We presented a code replication technique for improving the accuracy of semi-static branch prediction. This technique combines different correlation strategies. Therefore, the prediction is sometimes more accurate than dynamic prediction that uses only a single strategy. Since our technique is semi-static, the increased accuracy can be used by compile time optimizations like code motion or speculative execution.

Acknowledgement

We express our thanks to Manfred Brockhaus, Anton Ertl, Ulrich Neumerkel, Franz Puntigam, and Jian Wang for their comments on earlier drafts of this paper. We would also like to thank the reviewers for their helpful suggestions.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *1993 SIGPLAN Conference on Programming Language Design and Implementation.* ACM, June 1993.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, October 1992.

[Lar93] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5), May 1993.

[MH86] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium on Computer Architecture.* ACM, 1986.

[MW92] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *1992 SIGPLAN Conference on Programming Language Design and Implementation.* ACM, June 1992.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *1990 SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1990.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, October 1992.

[Smi81] James E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium on Computer Architecture*. ACM, 1981.

[Wal91] David E. Wall. Predicting program behavior using real or estimated profiles. In *1991 SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1991.

[YN92] Tse-Yu Yeh and Yale N.Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*. ACM, 1992.

[YN93] Tse-Yu Yeh and Yale N.Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*. ACM, 1993.

A Misprediction Rate vs. Code Size

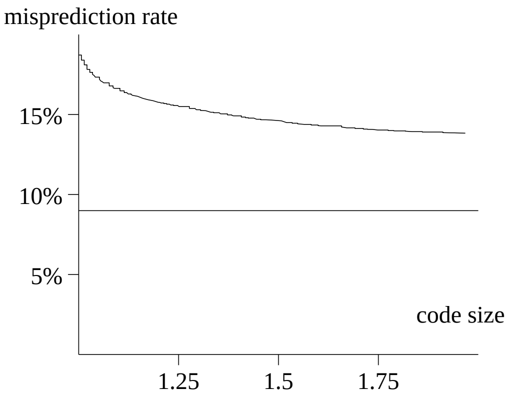

Figure 6: abalone

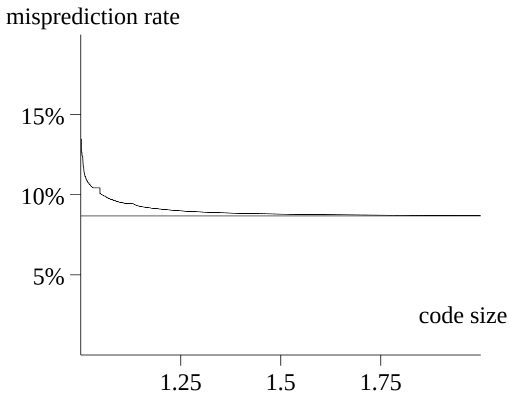

Figure 7: c-compiler

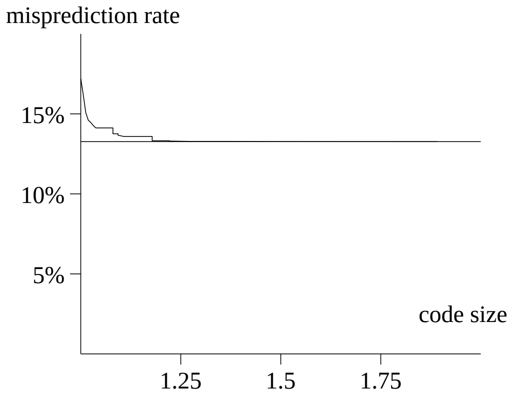

Figure 8: compress

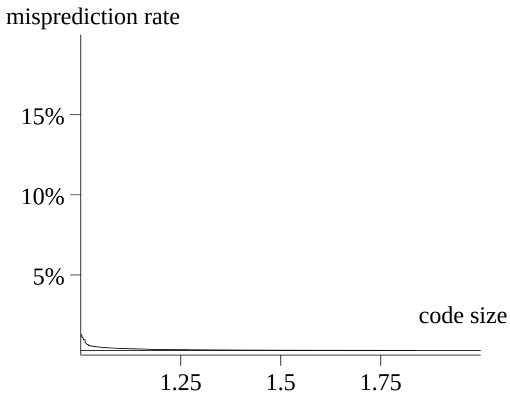

Figure 9: ghostview

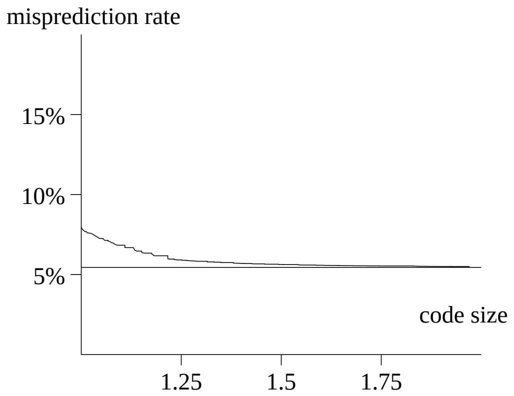

Figure 10: predict

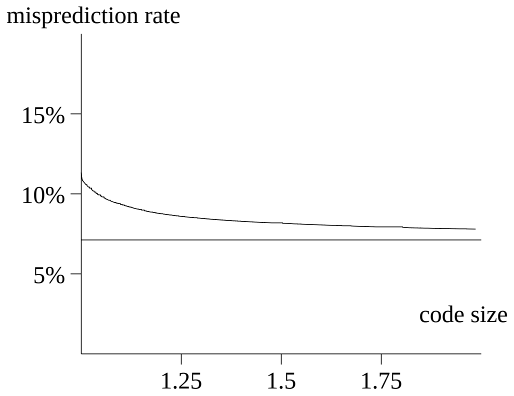

Figure 11: prolog

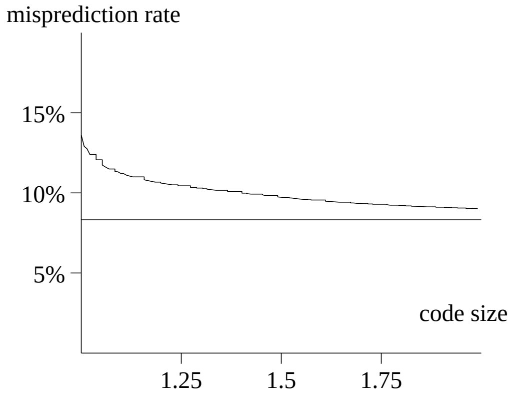

Figure 12: scheduler

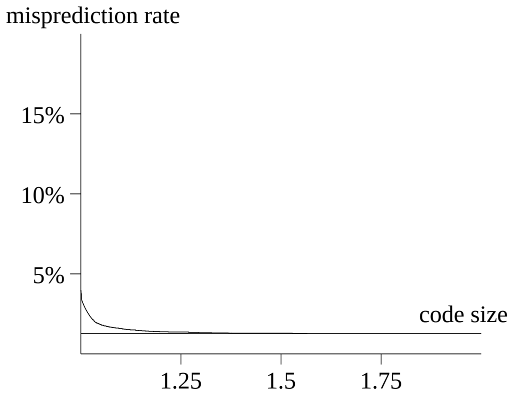

Figure 13: doduc