Path-Based Next Trace Prediction

Quinn Jacobson
Department of Electrical & Computer Engineering
University of Wisconsin
qjacobso@ece.wisc.edu

Eric Rotenberg
Department of Computer Science
University of Wisconsin
ericro@cs.wisc.edu

James E. Smith
Department of Electrical & Computer Engineering
University of Wisconsin
jes@ece.wisc.edu

Path-Based Next Trace Prediction

Quinn Jacobson
Department of Electrical & Computer Engineering
University of Wisconsin
qjacobso@ece.wisc.edu

Eric Rotenberg
Department of Computer Science
University of Wisconsin
ericro@cs.wisc.edu

James E. Smith
Department of Electrical & Computer Engineering
University of Wisconsin
jes@ece.wisc.edu

Abstract

The trace cache has been proposed as a mechanism for providing increased fetch bandwidth by allowing the processor to fetch across multiple branches in a single cycle. But to date predicting multiple branches per cycle has meant paying a penalty in prediction accuracy. We propose a next trace predictor that treats the traces as basic units and explicitly predicts sequences of traces. The predictor collects histories of trace sequences (paths) and makes predictions based on these histories. The basic predictor is enhanced to a hybrid configuration that reduces performance losses due to cold starts and aliasing in the prediction table. The Return History Stack is introduced to increase predictor performance by saving path history information across procedure call/returns. Overall, the predictor yields about a 26% reduction in misprediction rates when compared with the most aggressive previously proposed, multiple-branch-prediction methods.

1. Introduction

Current superscalar processors fetch and issue four to six instructions per cycle -- about the same number as in an average basic block for integer programs. It is obvious that as designers reach for higher levels of instruction level parallelism, it will become necessary to fetch more than one basic block per cycle. In recent years, there have been several proposals put forward for doing so [3,4,12]. One of the more promising is the trace cache [9,10], where dynamic sequences of instructions, containing embedded predicted branches, are assembled as a sequential "trace" and are saved in a special cache to be fetched as a unit.

Trace cache operation can best be understood via an example. Figure 1 shows a program's control flow graph (CFG), where each node is a basic block, and the arcs represent potential transfers of control. In the figure, arcs corresponding to branches are labeled to indicate taken (T) and not taken (N) paths. The sequence ABD represents one possible trace which holds the instructions from the basic blocks A, B, and D. This would be the sequence of instructions beginning with basic block A where the next two branches are not taken and taken, respectively. These basic blocks are not contiguous in the original program, but would be stored as a contiguous block in the trace cache. A number of traces can extracted from the CFG -- four possible traces are:

1: ABD
2: ACD
3: EFG
4: EG

Of course, many other traces could also be chosen for the same CFG, and, in fact, a trace does not necessarily have to begin or end at a basic block boundary, which further increases the possibilities. Also, note that in a trace cache, the same instructions may appear in more than one trace. For example, the blocks A, D, E, and G each appear twice in the above list of traces. However, the mechanism that builds traces should use some heuristic to reduce the amount of redundancy in the trace cache; beginning and ending on basic block boundaries is a good heuristic for doing this.

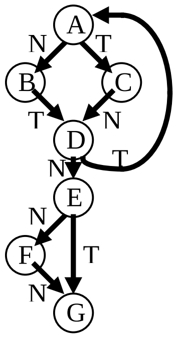

Figure 1 Example CFG

Associated with the trace cache is a trace fetch unit, which fetches a trace from the cache each cycle. To do this in a timely fashion, it is necessary to predict what the next trace will be. A straightforward method, and the one used in [9,10], is to predict simultaneously the multiple branches within a trace. Then, armed with the last PC of the preceding trace and the multiple predictions, the fetch unit can access the next trace. In our example, if trace 1 -- ABD -- is the most recently fetched trace, and a multiple branch predictor predicts that the next three branch outcomes will be T,T,N, then the next trace will implicitly be ACD.

In this paper, we take a different approach to next trace prediction -- we treat the traces as basic units and explicitly predict sequences of traces. For example, referring to the above list of traces, if the most recent trace is trace 1, then a next trace predictor might explicitly output "trace 2." The individual branch predictions T,T,N, are implicit.

We propose and study next trace predictors that collect histories of trace sequences and make predictions based on these histories. This is similar to conditional branch prediction where predictions are made using histories of branch outcomes. However, each trace typically has more than two successors, and often has many more. Consequently, the next trace predictor keeps track of sequences of trace identifiers, each identifier containing multiple bits. We propose a basic predictor and then add enhancements to reduce performance losses due to cold starts, procedure call/returns, and interference due to aliasing in the prediction table. The proposed predictor yields substantial performance improvement over the previously proposed, multiple-branch-prediction methods. For the six benchmarks that we studied the average misprediction rate is 26% lower for the proposed predictor than for the most aggressive previously proposed multiple-branch predictor.

2. Previous work

A number of methods for fetching multiple basic blocks per cycle have been proposed. Yeh et al. [12] proposed a Branch Address Cache that predicted multiple branch target addresses every cycle. Conte et al. [3] proposed an interleaved branch target buffer to predict multiple branch targets and detect short forward branches that stay within the same cache line. Both these methods use conventional instruction caches, and both fetch multiple lines based on multiple branch predictions. Then, after fetching, blocks of instructions from different lines have to be selected, aligned and combined -- this can lead to considerable delay following instruction fetch. It is this complex logic and delay in the primary pipeline that the trace cache is intended to remove. Trace caches [9,10] combine blocks of instructions prior to storing them in the cache. Then, they can be read as a block and fed up the pipeline without having to pass through complex steering logic.

Branch prediction in some form is a fundamental part of next trace prediction (either implicitly or explicitly). Hardware branch predictors predict the outcome of branches based on previous branch behavior. At the heart of most branch predictors is a Pattern History Table (PHT), typically containing two-bit saturating counters [11]. The simplest way to associate a counter with a branch instruction is to use some bits from the PC address of the branch, typically the least significant bits, to index into the PHT [11]. If the counter's value is two or three, the branch is predicted to be taken, otherwise the branch is predicted to be not taken.

Correlated predictors can increase the accuracy of branch prediction because the outcome of a branch tends to be correlated with the outcome of previous branches [8,13]. The correlated predictor uses a Branch History Register (BHR). The BHR is a shift register that is usually updated by shifting in the outcome of branch instructions -- a one for taken and a zero for not taken. In a global correlated predictor there is a single BHR that is updated by all branches. The BHR is combined with some bits (possibly zero) from a branch's PC address, either by concatenating or using an exclusive-or function, to form an index into the PHT. With a correlated predictor a PHT entry is associated not only with a branch instruction, but with a branch instruction in the context of a specific BHR value. When the BHR alone is used to index into the PHT, the predictor is a GAg predictor [13]. When an exclusive-or function is used to combine an equal number of bits from the BHR and the branch PC address, the predictor is a GSHARE predictor [6]. GSHARE has been shown to offer consistently good prediction accuracy.

The mapping of instructions to PHT entries is essentially implemented by a simple hashing function that does not detect or avoid collisions. Aliasing occurs when two unrelated branch instructions hash to the same PHT entry. Aliasing is especially a problem with correlated predictors because a single branch may use many PHT entries depending on the value of the BHR, thus increasing contention.

In order to support simultaneous fetching of multiple basic blocks, multiple branches must be predicted in a single cycle. A number of modifications to the correlated predictor discussed above have been proposed to support predicting multiple branches at once. Franklin and Dutta [4] proposed subgraph oriented branch prediction mechanisms that uses local history to form a prediction that encodes multiple branches. Yeh, et al. [13] proposed modifications to a GAg predictor to multiport the predictor and produce multiple branch predictions per cycle. Rotenberg et al. [10] also used the modified GAg for their trace cache study.

Recently, Patel et al. [9] proposed a multiple branch predictor tailored to work with a trace cache. The predictor attempts to achieve the advantages of a GSHARE predictor while providing multiple predictions. The predictor uses a BHR and the address of the first instruction of a trace, exclusive-ored together, to index into the PHT. The entries of the PHT have been modified to contain multiple two-bit saturating counters to allow simultaneous prediction of multiple branches. The predictor offers superior accuracy compared with the multiported GAg predictor, but does not quite achieve the overall accuracy of a single branch GSHARE predictor.

Nair [7] proposed "path-based" prediction, a form of correlated branch prediction that has a single branch history register and prediction history table. The innovation is that the information stored in the branch history register is not the outcome of previous branches, but their truncated PC addresses. To make a prediction, a few bits from each address in the history register as well as a few bits from the current PC address are concatenated to form an index into the PHT. Hence, a branch is predicted using knowledge of the sequence, or path, of instructions that led up to it. This gives the predictor more specific information about prior control flow than the taken/not taken history of branch outcomes. Jacobson et al. [5] refined the path-based scheme and applied it to next task prediction for multiscalar processors. It is an adaptation of the multiscalar predictor that forms the core of the path-based next trace predictor presented here.

3. Path-based next trace predictors

We consider predictors designed specifically to work with trace caches. They predict traces explicitly, and in doing so implicitly predict the control instructions within the trace. Next trace predictors replace the conventional branch predictor, branch target buffer (BTB) and return address stack (RAS). They have low latency, and are capable of making a trace prediction every cycle. We show they also offer better accuracy than conventional correlated branch predictors.

3.1. Naming of traces

In theory, a trace can be identified by all the PCs in the trace, but this would obviously be expensive. A cheaper and more practical method is to use the PC value for the first instruction in the trace combined with the outcomes of conditional branches embedded in the trace. This means that indirect jumps can not be internal to a trace. We use traces with a maximum length of 16 instructions. For accessing the trace cache we use the following method. We assume a 36 bit identifier, 30 bits to identify the starting PC and six bits to encode up to six conditional branches. The limit of six branches is somewhat arbitrary and is chosen because we observed that length 16 traces almost never have more than six branches. It is important to note that this limit on branches is not required to simplify simultaneous multiple branch prediction, as is the case with trace predictors using explicit branch prediction.

3.2. Correlated predictor

The core of the next trace predictor uses correlation based on the history of the previous traces. The identifiers of the previous few traces represent a path history that is used to form an index into a prediction table; see Figure 2. Each entry in the table consists of the identifier of the predicted trace (PC + branch outcomes), and a two-bit saturating counter. When a prediction is correct the counter is incremented by one. When a prediction is incorrect and the counter is zero, the predicted trace will be replaced with the actual trace. Otherwise, the counter is decremented by two and the predicted trace entry is unchanged. We found that the increment-by-1, decrement-by-2 counter gives slightly better performance than either a one bit or a conventional two-bit counter.

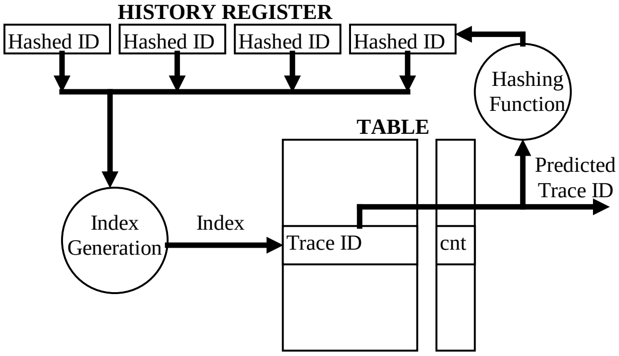

Figure 2 Correlated predictor

Path history is maintained as a shift register that contains 16 bit hashed trace identifiers (Figure 2). The hashing function uses the outcome of the first two conditional branches in the trace identifier as the least significant two bits, the two least significant bits of the starting PC as the next two bits, the upper bits are formed by taking the outcomes of additional conditional branch outcomes and exclusive-oring them with the next least significant bits of the starting PC. Beyond the last conditional branch a value of zero is used for any remaining branch outcome bits.

The history register is updated speculatively with each new prediction. In the case of an incorrect prediction the history is backed up to the state before the bad prediction. The prediction table is updated only after the last instruction of a trace is retired -- it is not speculatively updated.

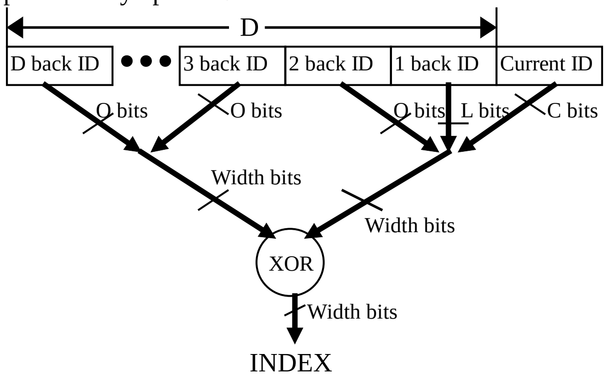

Figure 3 Index generation mechanism

Ideally the index generation mechanism would simply concatenate the hashed identifiers from the history register to form the index. Unfortunately this is sometimes not practical because the prediction table is relatively small so the index must be restricted to a limited number of bits.

The index generation mechanism is based on the method developed to do inter-task prediction for multiscalar processors [5]. The index generation mechanism uses a few bits from each of the hashed trace identifiers to form an index. The low order bits of the hashed trace identifiers are used. More bits are used from more recent traces. The collection of selected bits from all the traces may be longer than the allowable index, in which case the collection of bits is folded over onto itself using an exclusive-or function to form the index. In [5], the “DOLC” naming convention was developed for specifying the specific parameters of the index generation mechanism. The first variable ‘D’epth is the number of traces besides the last trace that are used for forming the index. The other three variables are: number of bits from ‘O’lder traces, number of bits from the ‘L’ast trace and the number of bits from the ‘C’urrent. In the example shown in Figure 3 the collection of bits from the trace identifiers is twice as long as the index so it is folded in half and the two halves are combined with an exclusive-or. In other cases the bits may be folded into three parts, or may not need to be folded at all.

3.3. Hybrid predictor

If the index into the prediction table reads an entry that is unrelated to the current path history the prediction will almost certainly be incorrect. This can occur when the particular path has never occurred before, or because the table entry has been overwritten by unrelated path history due to aliasing. We have observed that both are significant, but for realistically sized tables aliasing is usually more important. In branch prediction, even a randomly selected table entry typically has about a 50% chance of being correct, but in the case of next trace prediction the chances of being correct with a random table entry is very low.

To address this issue we operate a second, smaller predictor in parallel with the first (Figure 4). The secondary predictor requires a shorter learning time and suffers less aliasing pressure. The secondary predictor uses only the hashed identifier of the last trace to index its table. The prediction table entry is similar to the one for the correlated predictor except a 4 bit saturating counter is used that decrements by 8 on a misprediction. The reason for the larger counter will be discussed at the end of this section.

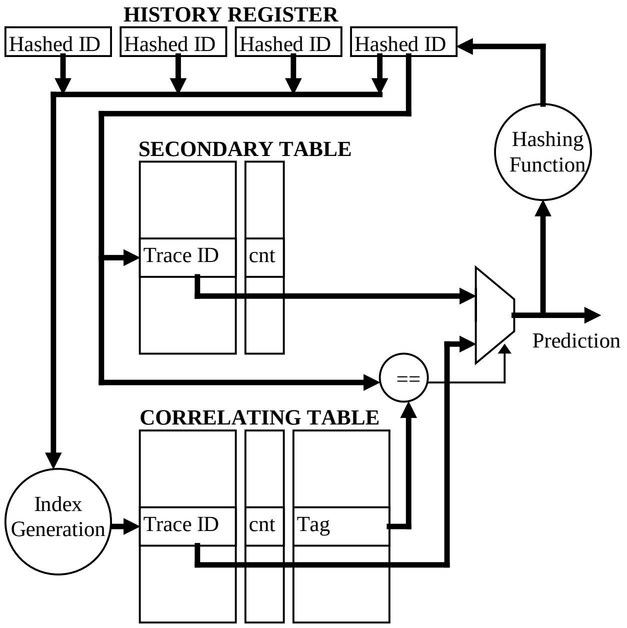

Figure 4 Hybrid predictor

To decide which predictor to use for any given prediction, a tag is added to the table entry in the correlated predictor. The tag is set with the low 10 bits of the hashed identifier of the immediately preceding trace at the time the entry is updated. A ten bit tag is sofficient to eliminate practically all unintended aliasing When a prediction is being made, the tag is checked against the hashed identifier of the preceding trace, if they match the correlated predictor is used; otherwise the secondary predictor is used. This method increases the likelihood that the correlated predictor corresponds to the correct context when it is used. This method also allows the secondary table to make a prediction when the context is very limited, i.e. under startup conditions.

The hybrid predictor naturally reduces aliasing pressure somewhat, and by modifying it slightly, aliasing pressure can be further reduced. If the 4-bit counter of the secondary predictor is saturated, its prediction is used, and more importantly, when it is correct the correlated predictor is not updated. This means if a trace is always followed by the same successor the secondary predictor captures this behavior and the correlated predictor is not polluted. This reduces the number of updates to the correlated predictor and therefore the chances of aliasing. The relatively large counter, 4-bits, is used to avoid giving up the opportunity to use the correlated predictor unless there is high probability that a trace has a single successor.

3.4. Return history stack (RHS)

The accuracy of the predictor is further increased by a new mechanism, the return history stack (RHS). A field is added to each trace indicating the number of calls it contains. If the trace ends in a return, the number of calls is decremented by one. After the path history is updated, if there are any calls in the new trace, a copy of the most recent history is made for each call and these copies are pushed onto a special hardware stack. When there is a trace that ends in a return and contains no calls, the top of the stack is popped and is substituted for part of the history. One or two of the most recent entries from the current history within the subroutine are preserved, and the entries from the stack replace the remaining older entries of the history. When there are five or fewer entries in the history, only the most recent hashed identifier is kept. When there are more than five entries the two most recent hashed identifiers are kept.

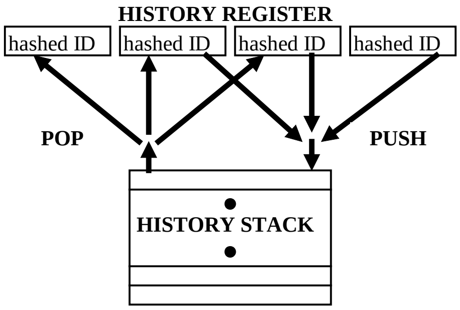

Figure 5 Return history stack implementation

With the RHS, after a subroutine is called and has returned, the history contains information about what happened before the call, as well as knowledge of the last one or two traces of the subroutine. We found that the RHS can significantly increase overall predictor accuracy. The reason for the increased accuracy is that control flow in a program after a subroutine is often tightly correlated to behavior before the call. Without the RHS the information before the call is often overflowed by the control flow within a subroutine. We are trying to achieve a careful balance of history information before the call versus history information within the call. For different benchmarks the optimal point varies. We found that configurations using one or two entries from the subroutine provide consistently good behavior.

The predictor does not use a return address stack (RAS), because it requires information on an instruction level granularity, which the trace predictor is trying to avoid. The RHS can partly compensate for the absence of the RAS by helping in the initial prediction after a return. If a subroutine is significantly long it will force any pre-call information out of the history register, hence determining the calling routine, and therefor where to return, would be much harder without the RHS.

4. Simulation methodology

4.1. Simulator

To study predictor performance, trace driven simulation with the Simplescalar tool set is used [1]. Simplescalar uses an instruction set largely based on MIPS, with the major deviation being that delayed branches have been replaced with conventional branches. We use the Gnu C compiler that targets Simplescalar. The functional simulator of the Simplescalar instruction set is used to produce a dynamic stream of instructions that is fed to the prediction simulator.

For most of this work we considered the predictor in isolation, using immediate updates. A prediction of the next trace is made and the predictor is updated with the actual outcome before the next prediction is made. We also did simulations with an execution engine. This allows updates to be performed taking execution latency into account. We modeled an 8-way out-of-order issue superscalar processor with a 64 instruction window. The processor had a 128KB trace cache, a 64KB instruction cache, and a 4-ported 64KB data cache. The processor has 8 symmetric functional units and supports speculative memory operations.

4.2. Trace selection

For our study, we used traces that are a maximum of 16 instructions in length and can contain up to six branches. The limit on the number of branches is imposed only by the naming convention of traces. Any control instruction that has an indirect target can not be embedded into a trace, and must be at the end of a trace. This means that some traces will be shorter than the maximum length. As mentioned earlier, instructions with indirect targets are not embedded to allow traces to be uniquely identified by their starting address and the outcomes of any conditional branches.

We used very simple trace selection heuristics. More sophisticated trace selection heuristics are possible and would significantly impact the behavior of the trace predictor. A study of the relation of trace selection and trace predictability is beyond the scope of this paper.

4.3. Benchmarks

We present results from six SpecInt95 benchmarks: compress, gcc, go, jpeg, m88ksim and xlisp. All results are based on runs of at least 100 million instructions.

Table 1 Benchmark summary

Benchmark	Input	number of instr.	avg. trace length	Static traces
compress	400000 e 2231	$104 * 10^6$	14.5	992
gcc	genrecog.i	$117 * 10^6$	13.9	51337
go	9 9	$133 * 10^6$	14.8	48736
jpeg	vigo.ppm	$166 * 10^6$	15.8	5462
m88ksim	ctl.in	$120 * 10^6$	13.1	2871
xlisp	queens 7	first 100 million	12.4	1393

5. Performance

5.1. Sequential branch predictor

For reference we first determined the trace prediction accuracy that could be achieved by taking proven control flow prediction components and predicting each control instruction sequentially. In sequential prediction each branch is explicitly predicted and at the time of the prediction the outcomes of all previous branches are known. This is useful for comparisons although it is not realizable because it would require multiple accesses to predict a single trace and requires knowledge of the branch addresses within the trace. The best multiple branch predictors to date have attempted to approximate the behavior of this conceptual sequential predictor.

We used a 16-bit GSHARE branch predictor, a perfect branch target buffer for branches with PC-relative and absolute address targets, a 64K entry correlated branch target buffer for branches with indirect targets [2], and a perfect return address predictor. All of these predictors had ideal (immediate) updates. When simulating this mechanism, if one or more predictions within a trace was incorrect we counted it as one trace misprediction. This configuration represents a very aggressive, ideal predictor. The prediction accuracy of this idealized sequential prediction is given in Table 2. The mean of the trace misprediction rate is 12.1%. We show later that our proposed predictor can achieve levels of prediction accuracy significantly better than those achievable by this idealized sequential predictor. In the results section we refer to the trace prediction accuracy of the idealized sequential predictor as “sequential.”

The misprediction rate for traces tends to be lower than that obtained by simply multiplying the branch misprediction rate by the number of branches because branch mispredictions tend to be clustered. When a trace is mispredicted, multiple branches within the same trace are often mispredicted. Xlisp is the exception, with hard to predict branches tending to be in different traces. With the aggressive target prediction mechanisms none of the benchmarks showed substantial target misprediction.

Table 2 Prediction accuracy for sequential predictors

Benchmark	16-bit Gshare branch misprediction	Number of Branches per Trace	Mispredic tion of traces
compress	9.2	2.1	17.9
gcc	8.0	2.1	14.0
go	16.6	1.8	24.5
jpeg	6.9	1.0	6.7
m88ksim	1.6	1.8	3.1
xlisp	3.2	1.9	6.5

5.2. Performance with unbounded tables

To determine the potential of path-based next trace prediction we first studied performance assuming unbounded tables. In this study, each unique sequence of trace identifiers maps to its own table entry. I.e. there is no aliasing.

We consider varying depths of trace history, where depth is the number of traces, besides the most recent trace, that are combined to index the prediction table. For a depth of zero only the identifier of the most recent trace is used. We study history depths of zero through seven.

Figure 6 presents the results for unbounded tables, the mean of the misprediction rate is 8.0% for the RHS predictor at the maximum depth. For comparisons, the “sequential” predictor is based on a 16-bit Gshare predictor that predicts all conditional branches sequentially. For all the benchmarks the proposed path-based predictor does better than the idealized sequential predictor. On average, the misprediction rate is 34% lower for the proposed predictor. In the cases of gcc and go the predictor has less than half the misprediction rate of the idealized sequential predictor.

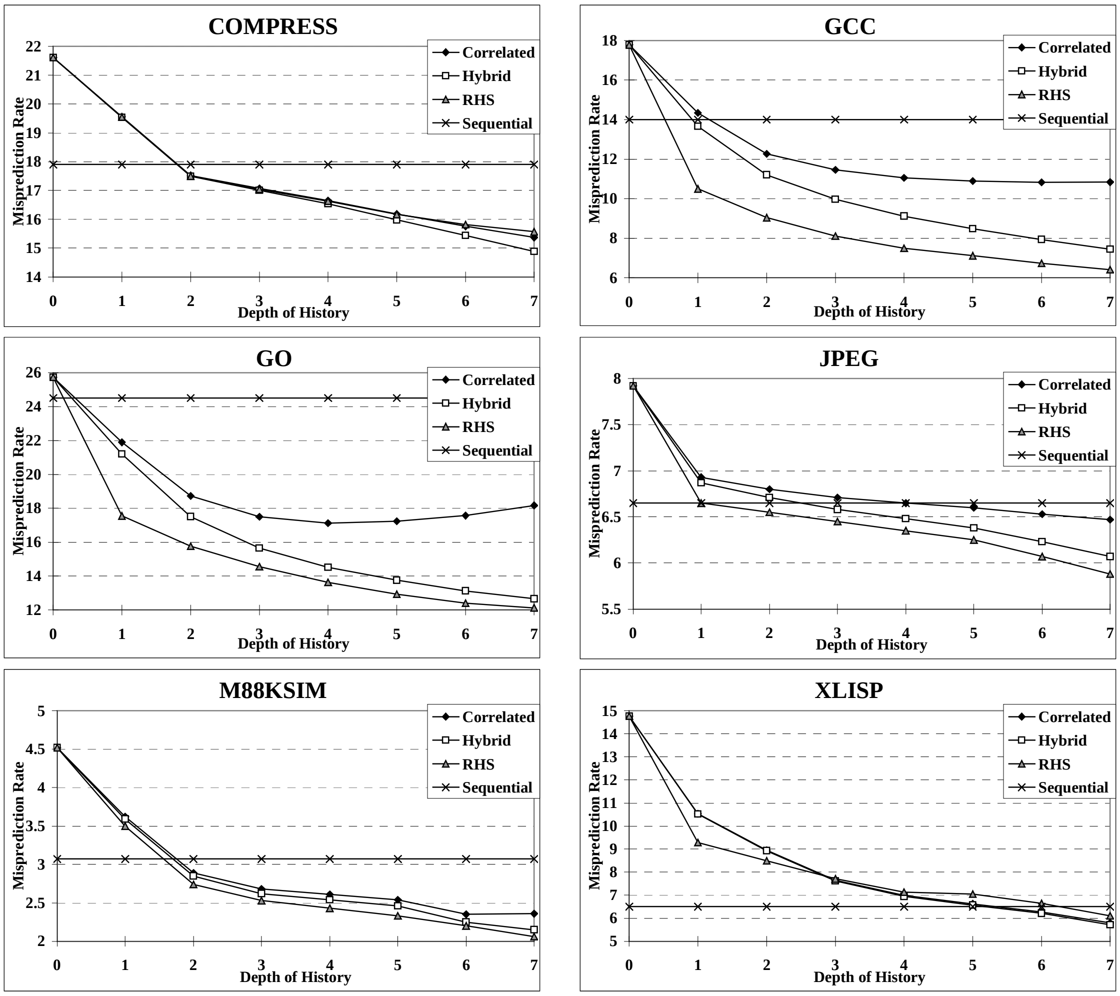

Figure 6 Next trace prediction with unbounded tables

For all benchmarks, the hybrid predictor has a higher prediction accuracy than using the correlated predictor alone. The benchmarks with more static traces see a larger advantage from the hybrid predictor because they contain more unique sequences of traces. Because the table size is unbounded the hybrid predictor is not important for aliasing, but is important for making predictions when the correlated predictor entry is cold.

For four out of the six benchmarks adding the return history stack (RHS) increases prediction accuracy. Furthermore, the four improved benchmarks see a more significant increase due to the RHS than the two benchmarks hurt by the RHS see a decrease. For benchmark compress the predictor does better without the RHS. For compress, the information about the subroutine being thrown away by the RHS is more important than the information before the subroutine that is being saved.

Xlisp extensively uses recursion, and to minimize overhead it uses unusual control flow to backup quickly to the point before the recursion without iteratively performing returns. This behavior confuses the return history stack because there are a number of calls with no corresponding returns. However, it is hard to determine how much of the performance loss of RHS with xlisp is caused by this problem and how much is caused by loss of information about the control flow within subroutines.

5.3. Performance with bounded tables

We now consider finite sized predictors. The table for the correlated predictor is the most significant component with respect to size. We study correlated predictors with tables of 2^{14}, 2^{15} and 2^{16} entries. For each size we consider a number of configurations with different history depths. The configurations for the index generation function were chosen based on trial-and-error. Although better configurations are no doubt possible we do not believe differences would be significant.

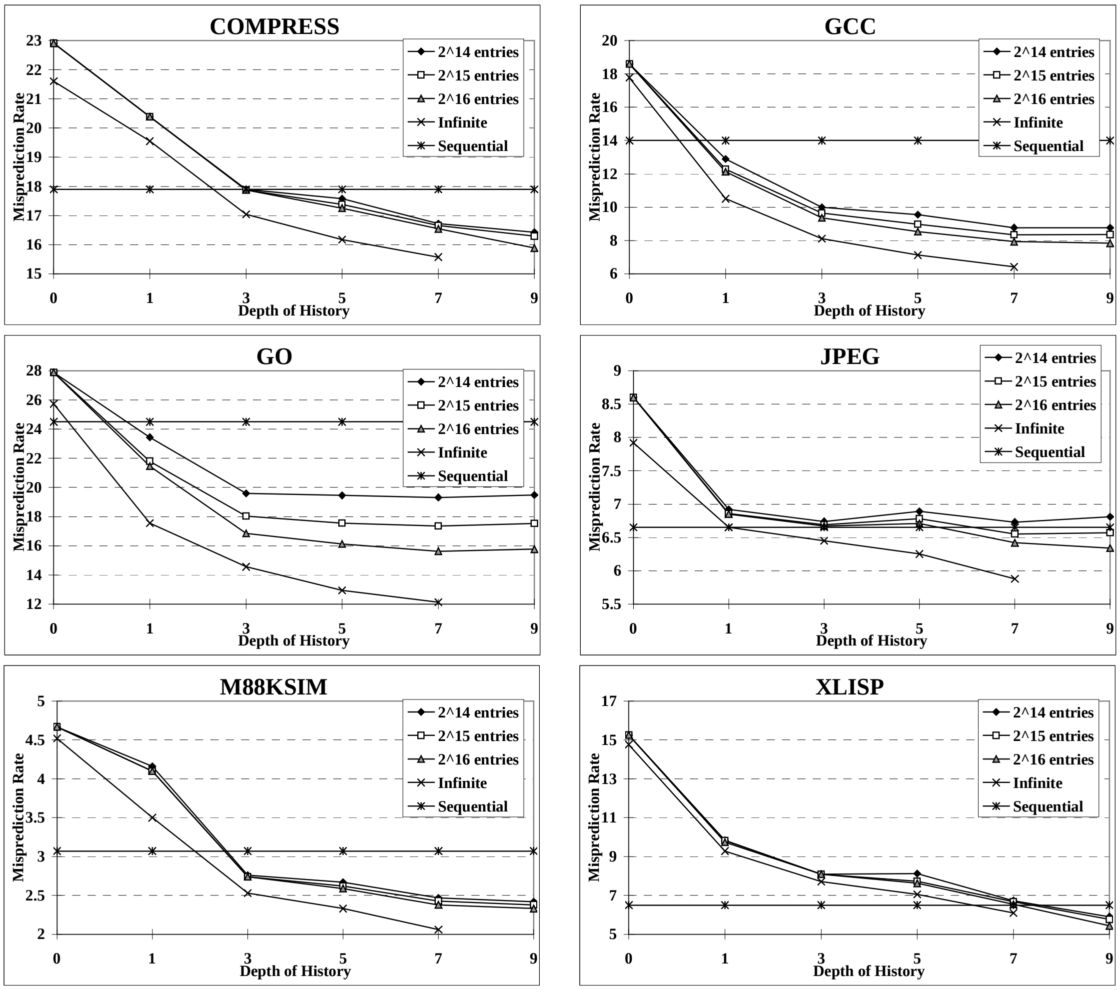

Figure 7 Next trace prediction

We use a RHS that has a maximum depth of 128. This depth is more than sufficient to handle all the benchmarks except for the recursive section of xlisp, where the predictor is of little use, anyway.

Performance results are in Figure 7. Three of the benchmarks stress the finite-sized predictors: gcc, go and jpeg. In these predictors the deviation from the unbounded tables is very pronounced, as is the deviation between the different table sizes. As expected, the deviation becomes more pronounced with longer histories because there are more unique sequences of trace identifiers being used and, therefore, more aliasing.

Go has the largest number of unique sequences of trace identifiers, and apparently suffers from aliasing pressure the most. At first, as history depth is increased the miss rate goes down. As the history depth continues to increase, the number of sequences competing for the finite size table increases aliasing. The detrimental effects of aliasing eventually starts to counter the gain of going to deeper histories and at some point dominates and causes a negative effect for increased history depth. The smaller the table size, the sooner the effects of aliasing start to become a problem. It is important to focus on the behavior of this benchmark and the other two larger benchmarks -- gcc and jpeg, because in general the other benchmarks probably have relatively small working sets compared to most realistic programs.

We see that for realistic tables, the predictor can achieve very high prediction accuracies. In most cases, the predictor achieves miss rates significantly below the idealized sequential predictor. The only benchmark where the predictor can not do better than sequential prediction is for a small, 2^{14} entry, table for jpeg. But even in this case it can achieve performance very close to the sequential, and probably closer than a realistic implementation of Gshare modified for multiple branches per cycle throughput. For our predictor the means of the mispredict rates are 10.0%, 9.5% and 8.9% for the maximum depth configuration with 2^{14}, 2^{15} and 2^{16} entry tables respectively. These are all significantly below the 12.1% misprediction rate of the sequential predictor, 26% lower for the 2^{16} predictor.

Table 3 Index generation configurations used

Depth	D-O-L-C for 14 bit Index	D-O-L-C for 15 bit Index	D-O-L-C for 16 bit Index
0	0-0-0-14 (1p)	0-0-0-15 (1p)	0-0-0-16 (1p)
1	1-0-6-8 (1p)	1-0-7-8 (1p)	1-0-7-9 (1p)
3	3-5-7-11 (2p)	3-5-8-12 (2p)	3-5-9-13 (2p)
5	5-3-6-11 (2p)	5-4-5-9 (2p)	5-5-5-7 (2p)
7	7-4-7-11 (3p)	7-4-9-12 (3p)	7-5-7-11 (3p)
9	9-3-7-11 (3p)	9-3-9-12 (3p)	9-4-7-9 (3p)

5.4. Impact of delayed updates

Thus far simulation results have used immediate updates. In a real processor the history register would be updated with each predicted trace, and the history would be corrected when the predictor backs up due to a misprediction. The table entry would not be updated until the last instruction of a trace has retired.

Table 4 Impact of real updates

Benchmark	Misprediction with ideal updates	Misprediction with real update
compress	5.8	5.8
gcc	10.5	10.5
go	9.3	9.3
jpeg	3.5	3.6
m88ksim	2.4	2.1
xlisp	4.7	4.8

To make sure this does not make a significant impact on prediction accuracy, we ran a set of simulations where an execution engine was simulated. The configuration of the execution engine is discussed in section 4.1. The predictor being modeled has 2^{16} entries and a 7-3-6-8 DOLC configuration. Table 4 shows the impact of delayed updates, and it is apparent that delayed updates are not significant to the performance of the predictor. In one case, m88ksim, the delayed updates actually increased prediction accuracy. The delayed updates has the effect of increasing the amount of hysteresis in the prediction table which in some cases can increase performance.

5.5. A cost-reduced predictor

The cost of the proposed predictor is primarily a function of the size of the correlated predictor's table. The size of the correlated predictor's table is the number of entries multiplied by the size of an entry. The size of an entry is 48 bits: 36 bits to encode a trace identifier, two bits for the counter plus 10 bits for the tag.

A much less expensive predictor can be constructed, however, by observing that before the trace cache can be accessed, the trace identifier read from the prediction table must be hashed to form a trace cache index. For practical sized trace caches this index will be in the range of 10 bits. Rather than storing the full trace identifier, the hashed cache index can be stored in the table, instead. This hashed index can be the same as the hashed identifier that is fed into the history register (Figure 2). That is, the Hashing Function can be moved to the input side of the prediction table to hash the trace identifier before it is placed into the table. This modification should not affect prediction accuracy in any significant way and reduces the size of the trace identifier field from 36 bits to 10 bits. The full trace identifier is still stored in the trace cache as part of its entry and is read out as part of the trace cache access. The full trace identifier is used during execution to validate that the control flow implied by the trace is correct.

6. Predicting an alternate trace

Along with predicting the next trace, an alternate trace can be predicted at the same time. This alternate trace can simplify and reduce the latency for recovering when it is determined that a prediction is incorrect. In some implementations this may allow the processor to find and fetch an alternate trace instead of resorting to building a trace from scratch.

Alternate trace prediction is implemented by adding another field to the correlated predictor. The new field contains the identifier of the alternate prediction. When the prediction of the correlated predictor is incorrect the alternate prediction field is updated. If the saturating counter is zero the identifier in the prediction field is moved to the alternate field, the prediction field is then updated with the actual outcome. If the saturating counter is non-zero the identifier of the actual outcome is written into the alternate field.

Figure 8 shows the performance of the alternate trace predictor for two representative benchmarks. The graphs show the misprediction rate of the primary 2^{16} entry table predictor as well as the rate at which both the primary and alternate are mispredicted. A large percent of the mispredictions by the predictor are caught by the alternate prediction. For compress, 2/3 of the mispredictions are caught by the alternate, for gcc it is slightly less than half. It is notable that for alternate prediction the aliasing effect quickly dominates the benefit of more history because it does not require as much history to make a prediction of the two most likely traces, so the benefit of more history is significantly smaller.

There are two reasons alternate trace prediction works well. First, there are cases where some branch is not heavily biased; there may be two traces with similar likelihood. Second, when there are two sequences of traces aliased to the same prediction entry, as one sequence displaces the other, it moves the other's likely prediction to the alternate slot. When a prediction is made for the displaced sequence of traces, and the secondary predictor is wrong, the alternate is likely to be correct.

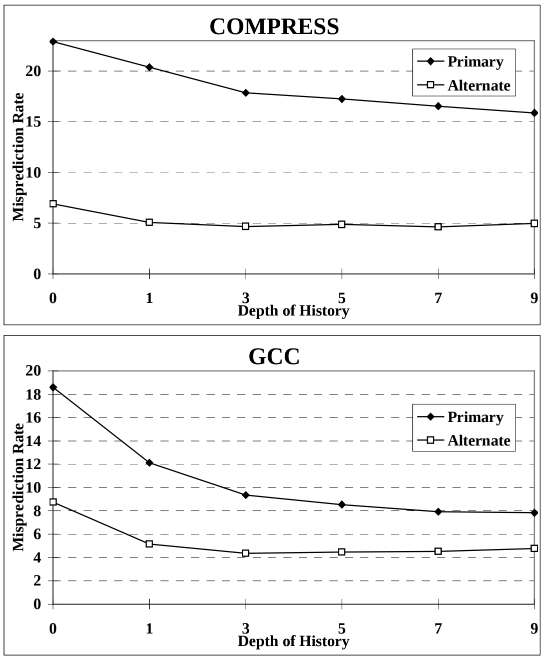

Figure 8 Alternate trace prediction accuracy

7. Summary

We have proposed a next trace predictor that treats the traces as basic units and explicitly predicts sequences of traces. The predictor collects histories of trace sequences and makes predictions based on these histories. In addition to the basic predictor we proposed enhancements to reduce performance losses due to cold starts, procedure call/returns, and the interference in the prediction table. The predictor yields consistent and substantial improvement over previously proposed, multiple-branch-prediction methods. On average the predictor had a 26% lower mispredict rate than the most aggressive previously proposed multiple-branch predictor.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and the U.S. Army Intelligence Center and Fort Huachuca under Contract DAPT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the U.S. Army Intelligence Center and For Huachuca, or the U.S. Government.

References

[1] D. Burger, T. Austin and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," University of Wisconsin - Madison Technical Report #1308, July 1996. (http://www.cs.wisc.edu.~mscalar/simplescalar.html)

[2] P.-Y. Chang, E. Hao and Y. Patt, "Target Prediction for Indirect Jumps," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 274-283, June 1997.

[3] T. Conte, K. Menezes, P. Mills and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-343, June 1995.

[4] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 258-263, December 1995.

[5] Q. Jacobson, S. Bennett, N. Sharms and J. E. Smith, "Control Flow Speculation in Multiscalar Processors," in *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp. 218-229, February 1997.

[6] S. McFarling, "Combining Branch Predictors," DEC WRL TN-36, June 1993

[7] R. Nair, "Dynamic Path-Based Branch Correlation," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 15-23, December 1995.

[8] S.-T. Pan, K. So and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," in *Proceedings of the 5th International Conference on Architecture Suport for Programming Languages and Operating Systems*, pp. 76-84, October 1992.

[9] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism." University of Michigan Technical Report CSE-TR-335-97, 1997.

[10] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, Dec. 1996.

[11] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[12] T.-Y. Yeh, D. Marr and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," in *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67-76, July 1993.

[13] T.-Y. Yeh and Y. Patt, "Two-Level Adaptive Branch Prediction," In *Proceedings of 24th International Symposium on Microarchitecture*, pp. 51-61, November 1991.