Journal of Instruction-Level Parallelism 8 (2006) 1-23 Submitted 6/2005; published 2/2006

Analyzing Effects of Trace Cache Configurations on the Prediction of Indirect Branches

Yunhe Shi YSHI@CS.TCD.IE
Dept. of Computer Science, Trinity College Dublin, Dublin 2, Ireland.

Emre Özer EMRE.OZER@ARM.COM
ARM Ltd, 110 Fulbourn Road, Cambridge, CB1 9NJ, UK.

David Gregg DAVID.GREGG@CS.TCD.IE
Dept. of Computer Science, Trinity College Dublin, Dublin 2, Ireland.

Abstract

This paper discusses the effects of using a trace cache on the indirect branch prediction in ILP processors. The main contribution of the paper is an exploration of the fact that the trace cache captures context information about the recent control flow of the program, which can improve the accuracy of predictors that do not themselves explicitly use such information. We analyze and experiment with various trace cache configurations and strategies to measure their effects on indirect branch prediction accuracy. We show that updating indirect branch target addresses in the trace cache improves indirect branch prediction accuracy. Then, we incrementally vary the trace cache configuration such as applying trace packing, adding 2-bit update counters per trace cache line, varying trace cache set associativity, cache size and cache line size in order to observe the impact of each configuration on the indirect branch prediction. We simulate a wide variety of designs using benchmarks with higher than average numbers of indirect branches. Our experimental results show that the harmnonic mean indirect branch prediction accuracy for a processor model with a trace cache that updates indirect branch target addresses is 42.04%, compared to 28.82% for a model with a trace cache that does not update indirect branch target addresses, and 10.85% for a model with a branch target buffer on our benchmarks. Our results have implications for any hardware predictor which stores entries corresponding to (possibly replicated) instructions in the trace cache rather than original instructions in main memory.

1. Introduction

Instruction-level parallel (ILP) processors, whether superscalar or VLIW, require a large number of functional units to extract higher ILP from applications. A wider execution engine demands a wider instruction fetch unit that must potentially match the width of the functional units in order to fully utilize them. This means that the instruction fetch unit must fetch a very large number of instructions from the instruction cache across several basic blocks at every cycle. In general, these basic blocks are placed in non-contiguous locations in the instruction cache. More than one cache access may have to be performed in order to fetch all the required basic blocks. This non-contiguous fetching places limits on the number of instructions that can be fetched in a single cycle (of reasonable length). Hence, the trace cache has been proposed as an alternative approach to fetching multiple basic blocks in a single cycle.

The trace cache is a microarchitectural technique for increasing instruction fetch bandwidth of a superscalar processor. The processor collects snapshots of the executed instruction stream and stores them as trace cache lines along with branch target addresses. This way, instructions from non-contiguous execution path are placed into contiguous locations in the trace cache. When a trace

cache line is predicted to be the next execution path, the entire trace cache line can be fetched in a single cache access.

In general, most general purpose C and Fortran programs have low numbers of indirect branches, and conditional branches and function returns are the most important types of branches to predict. However, the frequency of using indirect branches will be much higher in future applications [7, 21], although conditional branches will continue to outnumber indirect branches in most applications. For example, object-oriented programs use larger numbers of indirect branches to implement virtual function calls. Dynamically linked libraries are also called using indirect branches. Finally, an increasing number of program languages are being implemented using virtual machines (VMs) that are implemented partially (such as Sun's Hotspot Java VMs) or wholly using interpretation, a technique that involves very large numbers of indirect branches. Thus, higher indirect branch prediction accuracy rates will be sought in future ILP processors.

We observe that in an ILP processor with a trace cache, it is possible to improve the indirect branch prediction accuracy of programs using the trace cache to make indirect branch predictions instead of using a branch target buffer. Furthermore, using this scheme involves in very simple modifications to the trace cache hardware structure without using any extra hardware tables.

In this paper, we show how the trace cache can capture some of the context information used by two-level indirect branch predictors [5][6][11]. Although the improvement in accuracy is much lower than that achieved by a two-level indirect branch predictor, the cost is also much lower. If one has already decided to implement a trace cache, then it can be used to significantly improve indirect branch prediction over a BTB at little or no additional cost. We are not attempting to compete with two-level indirect branch predictors or next-trace/next-stream predictors [10] [22]. We merely show that some fraction of the benefits of two-level prediction can be captured by the trace cache, a result that is not widely known.

The main contribution of this paper is an exploration of the observation that the trace cache captures context information about the control-flow of the program. This context information can have an impact on the accuracy of other predictors, in this case the indirect branch predictor. Capturing this sort of context information is, to our knowledge, a mostly unintended side effect of the trace cache. Although others have noticed the same effect in other contexts (see Section 8), the results are underreported and not well-known. Our results are interesting primarily because of the practical effect on prediction accuracy. However it is also interesting because it demonstrates how different microarchitectural features can interact in unexpected ways.

Based on our initial observation, we have a number of smaller contributions, which come from exploring variations of the trace cache. 1) We propose to update the indirect branch target address in the trace cache if a trace cache line ends with an indirect branch instruction. We will motivate the use of the update policy for the rest of the paper. 2) We show that a 2-bit saturating update counter associated with an indirect branch target address at the end of each cache line can improve prediction accuracy in much the same way that such counters are used in other predictors. 3) We measure the impact of each individual trace cache configuration or strategy on the prediction of indirect branches such as using trace packing or varying cache size, line size and associativity.

The organization of the paper is as follows: *Section 2* gives a brief reminder to trace caches and indirect branch prediction. Then, *Section 3* discusses our method of predicting indirect branches using the trace cache. Next, *Section 4* introduces the experimental framework and *Section 5* presents the initial branch prediction results of a model with a branch target buffer, a base trace cache model and the trace cache model that updates indirect branch target addresses. Later, *Section 6* presents the indirect branch prediction accuracy results for several trace cache configurations. Section 7 looks at other trace cache models. In section 8 we examine some existing related work on the trace cache and branch prediction. Finally, *Section 9* concludes the paper.

2. Background

Most trace cache research [15][17][20][19] has focused on how different configurations of the trace cache can help improve the instruction fetch bandwidth of superscalar processors. *Jacobson et al.* [10] predicts indirect branch target addresses using the next-trace predictor along with all other branch types in trace caches. *Santana et al.* in [22] and [23] uses the next-stream predictor, which is quite similar to the next-trace predictor, to predict indirect branch target addresses. Both the next-trace and next-stream predictors use large tables to predict all types of branches in a single branch prediction framework. On the other hand, we propose, in this paper, a simple updating mechanism in the trace cache that can moderately improve the indirect branch prediction rate without any extension in the BTB mechanism.

2.1 Trace Cache

The model of the trace cache in this paper is based very closely on the description in Patel's *PhD dissertation* [15]. A trace cache system consists of an instruction cache, the trace cache, a multiple-branch predictor and a fill unit. The trace cache contains 2^x trace cache line entries ($x > 0$) and supplies the functional units with stored trace cache lines. Each trace cache line (Figure 1) stores multiple basic blocks for an execution path, namely it can contain up to n instructions with no more than m conditional branches.

According to Patel's default model [15, p. 10], each line of the trace cache contains 16 slots for instructions (up to three of which can be conditional branches), four slots for target addresses of the branches to allow immediate generation of the next fetch address, and path information (the number and directions of branches). In the case of a trace cache miss, the instruction cache provides instructions to the functional units.

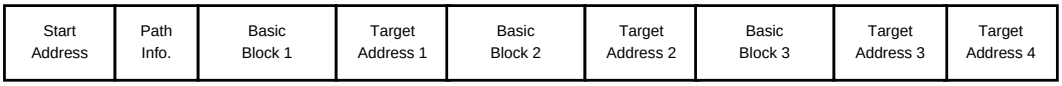

Figure 1: A trace cache line of 3 basic blocks

The multiple-branch predictor, which is based on a two-level branch predictor, is used to predict m branches simultaneously for a trace cache line [15, pp. 13–14]. The branch target buffer (BTB) provides target addresses for the predicted-taken branches in case of trace cache miss. The BTB saves only one target for each *taken* branch. A target address in the BTB is updated when the target address changes. For return instructions, a return address stack (RAS) is used. Each time a function call is executed, its return address is pushed onto the RAS. The branch predictor uses the RAS to get the return address for a return instruction.

The fill unit forms execution traces and places them into the trace cache as instructions retire from the functional units. A trace is terminated when it contains n instructions or m conditional branches or an indirect branch, return or a trap instruction. If there is a trace cache line starting at the same address, the new trace cache line replaces the existing trace cache line only when it is longer or follows a different execution path. This is known as the *keep-longest* write policy, and has been found by Patel to attain high performance while lowering the trace cache's write bandwidth requirements [15, pp. 86–87].

2.2 Indirect Branch Prediction

Indirect branch instructions are control instructions whose target addresses are loaded into registers. Thus, they can have multiple branch targets and this makes them very hard to predict [3][4][5][11]. The most commonly used predictor for indirect branches in current processors is the BTB, which caches only one target address for each indirect branch instruction. The predicted target address is

updated when the actual target changes. In all experiments in this paper the BTB is used to predict the target address of taken branches in the case of a trace cache miss.

Two-level indirect branch predictors combine the address of the branch with a *history* register of recent branch targets. They are effective because there is often a correlation between the outcome of different indirect branches. The history register stores context information on the outcome of these other branches, allowing indirect branch prediction rates of more than 90% [5][6][11]. To achieve the very best prediction accuracy, a two-level indirect branch predictor should be used. However, such predictors can be large and complicated.

3. Indirect Branch Prediction using Trace Cache

The trace cache can store multiple indirect branch targets in different trace cache lines ending with the same indirect branch instruction. Each trace cache line stores two pieces of context information that are useful for indirect branch prediction. First, every trace cache line has a starting address, which provides some context information for any indirect branch in the line. Secondly, if there are conditional branches before an indirect branch in a trace cache line, the directions of these conditional branches provide a context or path history for the indirect branch target. Although this context information is not as complete as the information in a two-level indirect branch predictor, we show in this paper that it can be used to provide better predictions than a BTB.

The inherent property of storing multiple indirect branch targets in different trace cache lines ending with the same indirect branch in the trace cache provides us with a storage unit capable of storing multiple target addresses for an indirect branch. Thus, we propose to explore this inherent property of trace cache to attain higher indirect branch prediction rates by incrementally adding new functionalities such as updating indirect branch target address in the trace cache lines, adding a 2-bit saturating update counter, associated with an indirect branch target address, to each trace cache line, using trace packing and tuning the trace cache parameters such as cache size, cache associativity and cache line size.

The trace cache described by Patel [15] does not update indirect branch addresses stored at the end of each trace cache line after the indirect branch instruction is executed. If the indirect branch target is mispredicted, it is not updated in the trace cache line unless the old trace cache line is replaced by the new one if the trace cache write policy is *always overwrite* [15].

```
for (i = 16; i >= 0; i--) {
    offset = i & 7;
    switch( intArray[offset] ) {
        case 0: x = y + z; break;
        case 1: x = y * z; break;
        case 2: x = x + y + z; break;
        case 3: x = y - z; break;
    }
}
```

Figure 2: Sample loop for case study

An example of a *switch* statement inside a loop in **Figure 2** is presented to demonstrate the effects of the trace cache on the indirect branch target prediction. The targets of the *switch* statement are decided by an array holding 8 integer numbers. The loop simply iterates through the same integer array elements (targets) 16 times.

Figure 3 shows the basic blocks and program flow chart constructed from machine code. Here; A, B, C, X0, X1, X2 and X3 denote the basic blocks in the program. The conditional branch at

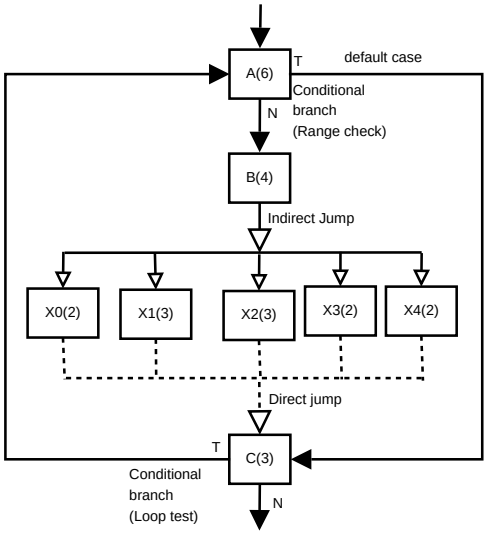

Figure 3: Basic Block Program Flow Diagram

A (a range check) and indirect jump at B are the *switch*- related instructions. The direct jump is the jump instruction for *break* statements in *case*s. The conditional branch at C is the loop test instruction. Finally, X# is the basic block associated with each *case* statement.

First, let us assume that we use only a BTB to predict branches. If the array contents are (0, 1, 2, 3, 0, 1, 2, 3), the indirect branch prediction accuracy becomes *0%* because the indirect branch target is updated in the BTB after each iteration. If the array contents are (0, 1, 2, 3, 3, 2, 1, 0), then the prediction accuracy rises to *18.75%*.

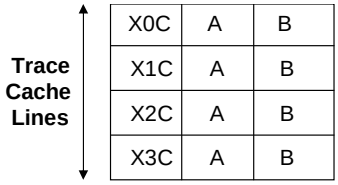

Figure 4: Trace cache line layout for the case study example

When the trace cache is used as shown in **Figure 4**, basic blocks A and B become replicated a number of times. Trace cache lines must always end at an indirect branch, so each of the successors of the indirect branch in block B (i.e. blocks X0..X3) becomes the start of a trace cache line. Each of these trace cache lines include their own copies of basic blocks A and B. Therefore, the indirect branch at the end of block B is split into four different instances, each with context information about the most recent outcome of the same indirect branch. In this case, the trace cache is equivalent to a two-level indirect branch predictor with a history length of one. The trace cache lines and their indirect branch targets are perfectly correlated and the indirect branch target prediction rate becomes *75%* with or without the updating policy for the branch target sequence of (0, 1, 2, 3, 0, 1, 2, 3). However, the prediction accuracy drops to *25%* with no updating and *0%* with the updating policy if the branch target sequence is (0, 1, 2, 3, 3, 2, 1, 0).

Updating or not updating indirect branch targets can have significant impact on the prediction of indirect branches depending on the target address access pattern. The update policy can outperform the non-update policy if one target is repeatedly accessed. For instance, the prediction accuracy is

only *18.75%* for the non-update but *68.75%* for the update policy if the target address pattern is (0, 1, 2, 1, 1, 1, 1, 1).

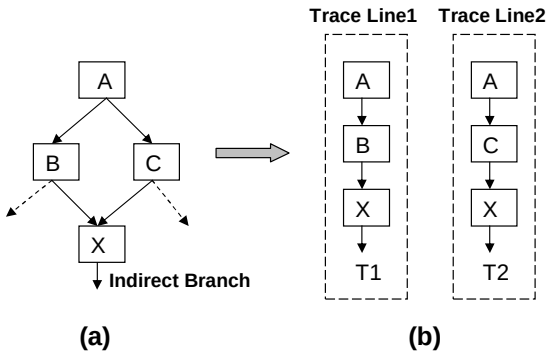

Figure 5: Trace cache lines starting with the same address may end with the same indirect branch instruction in a set-associative trace cache.

If the trace cache is designed as set-associative, then it can have different trace cache lines starting with the same address leading to the same indirect branch instruction. For instance, **Figure 5a** shows a control flow graph of a program that ends with the same indirect branch instruction. Here; A, B, C and X represent basic blocks where A is the starting and X is the ending blocks. Two traces are shown in **Figure 5b** in which the trace lines 1 and 2 follow ABX and ACX paths and each having the same indirect branch instruction in X but with two different target addresses (i.e. $T1$ and $T2$). If two trace cache lines paths somehow correlate to their corresponding targets, the indirect branch prediction accuracy can be improved. Now, let us assume that we choose to update indirect branch targets in the trace cache lines when they change. If the ABX and ACX paths always take the indirect branch targets $T1$ and $T2$, respectively, then the indirect branch prediction accuracy drastically improves. However, the indirect branch prediction accuracy can be very poor for the following sequence:

$$ABX \rightarrow T2,\ ABX \rightarrow T1,\ ABX \rightarrow T2 \ldots$$
$$\text{or}$$
$$ACX \rightarrow T1,\ ACX \rightarrow T2,\ ACX \rightarrow T1 \ldots$$

4. Experimental Framework

The SPECint2000 benchmark suite and six virtual machine interpreter benchmarks [8] are used to evaluate different trace cache configurations/strategies and their influences on the indirect branch target prediction. Reduced data sets [12] are used as inputs for SPECint2000 benchmarks that run to completion. **Table 1** shows the major characteristics of benchmarks. *Num Instr.* and *% Indirect Branches* represent the total number of dynamic instructions and the percentage of dynamic indirect branch instructions (excluding return instructions) in the total number of dynamic instructions. Note that in most of these programs indirect branches account for a small proportion of total instructions.

We implemented our trace cache model in *sim-bpred* in the *SimpleScalar* 3.0d [1] simulator. The baseline model is the microarchitecture model that has a 2-level conditional branch predictor and branch target buffer (BTB) for predicting indirect branches but has no trace cache. **Table 2** shows the configuration of the baseline model.

The indirect branch prediction accuracy of the BTB in the baseline model is compared with the base trace cache model whose simulator parameters are shown in **Table 3**. The base trace cache

Table 1: Benchmark statistics

SPECint2000	**Num Instr.**	**% Indirect Branches**
twolf	972,726,535	0.02%
vortex	1,153,664,377	0.03%
gap	761,346,123	0.77%
bzip2	1,819,780,259	0.000007%
vpr	1,566,703,859	0.00023%
gzip	1,361,319,057	0.00002%
perlbmk	2,061,197,349	1.47%
eon	1,070,281,136	0.61%
parser	4,527,012,522	0.0001%
crafty	834,909,899	0.25%
mcf	793,869,356	0.01%
gcc	5,117,054,875	0.34%

VM Interpreter	**Input Set**	**Num Instr.**	**% Indirect Branches**
gforth	benchgc	64,395,745	13.01%
li	boyer	183,304,695	1.13%
ocamlc	ocamllex	69,738,732	11.35%
ocamlc-switch	ocamllex	122,559,342	6.46%
perl	jumble	40,496,306	0.71%
scheme48	build	113,143,169	3.29%

Table 2: Baseline model

No Trace cache
2-level conditional branch predictor Global History Size = 8
2-level conditional branch predictor Pattern History Table Size = 1024-entry
BTB size = 512×4
Return Address Stack (RAS) Size = 8-entry

Table 3: Base trace cache model

1024-entry directed-mapped Trace cache
Trace Cache Write Policy: *keep-longest*
Trace lines of **16 instructions** with **at most 3 branches**
No updating of indirect branch targets in the trace cache

model uses the conditional branch prediction, BTB and RAS parameters of the baseline model as given in **Table 2**. However, unlike the baseline model, indirect branch predictions are more complicated. Indirect branch predictions are made using the trace cache lines, provided there is a trace cache hit. In the case of a trace cache miss, the BTB is instead used to make the indirect branch prediction. Thus, the overall indirect branch prediction rate is a combination of the rate achieved using the trace cache, and the rate for the BTB where trace cache missed occur.

An important question when measuring the performance of computer systems is summarizing data in an appropriate way. The most common way to average results from several different benchmarks is to simply use the arithmetic mean. However, in this paper we primarily measure prediction rates, and a more statistically meaningful average for rates is the harmonic mean [14]. One downside of the hardmonic mean is that very small values tend to dominate the result. Furthermore, as most hardware designers are more familiar with the arithmetic mean, and that there are often significant differences between the two measures, we show both means in all charts. However harmonic mean is the more meaningful measure, and that is the one used in the text. For convenience, corresponding arithmetic mean values are shown in brackets after.

5. Initial Prediction Accuracies

5.1 BTB versus Trace Cache with Non-update Policy

Figure 6 shows the indirect branch target address prediction accuracies for a model with a BTB (i.e. *BTB*) and the trace cache model with the non-update policy (i.e. *TC- No Update*). The last two columns in the figure shows the arithmetic and harmonic means of prediction accuracies across all benchmarks. The results for *bzip2*, *gzip* and *vpr* are not shown in the following figures because the indirect branch prediction accuracy does not change at all due to the extremely small number of indirect branch instructions.

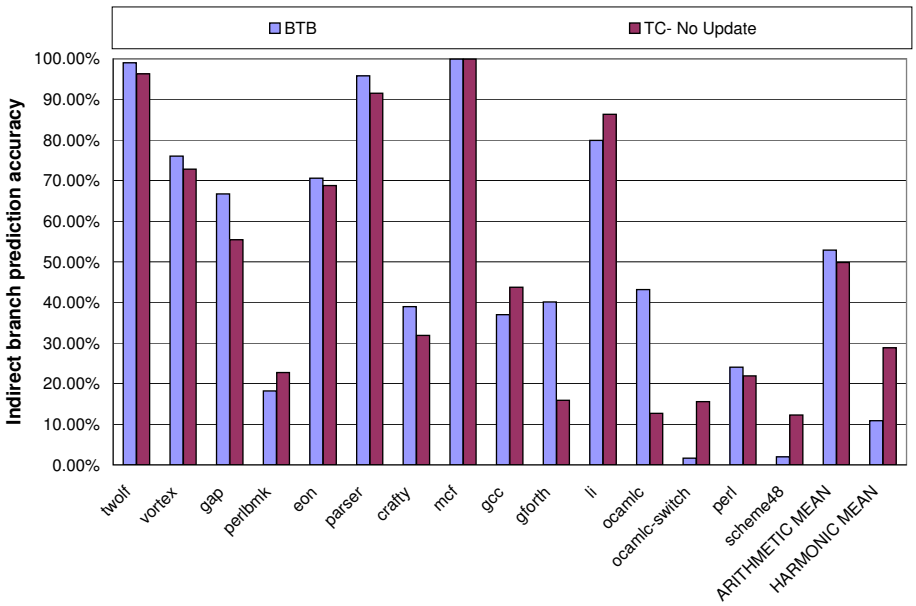

Figure 6: Indirect branch target prediction accuracies for the BTB and trace cache with non-update policy

In *BTB*, the BTB stores only one branch target for each indirect branch instruction and updates the branch target when it changes. *BTB* outperforms *TC- No Update* in these benchmarks: *twolf*, *vortex*, *gap*, *eon*, *parser*, *crafty*, *gforth*, *ocamlc* and *perl*. Not updating the target of the indirect branch when it mispredicts has a very negative impact on these benchmarks. This limits the trace cache's ability to adapt to changing program behavior.

On the other hand, the prediction accuracies in *TC- No Update* is higher than *BTB* for *perlmbk*, *gcc*, *li*, *ocamlc-switch* and *scheme48*. In particular for *ocamlc-switch* and *scheme48*, the advantage of some context information outweighs the significant disadvantage of not updating indirect branch targets when the behavior of the program changes. These virtual machine interpreters consist of code very similar to that in our example in **Figure 2**. The single indirect branch has a large number of targets, but there is a correlation between the previous outcome of this branch and the current one. Thus, prediction accuracy improves considerably.

The behavior of *perlmbk* and *li* is somewhat different. In both benchmarks, there is actually a benefit from not updating the branch target when it is incorrect. We investigated *li* and found out that there are indirect branches with a number of different targets, but where one target is more frequent than the others. If this most frequent outcome can be found, the best strategy is to stick with it, rather than update on every misprediction. Our experiments indicate that this target is finding its way into the trace cache, and the strategy of not updating is effective.

Overall, the harmonic mean (arithmetic mean in brackets) of indirect branch prediction accuracies across all benchmarks are 11% (52.88%) for *BTB* and 29% (49.85%) for *TC- No Update*. Note that the harmonic mean accuracy for the BTB is heavily dominated by the very low accuracies for *ocamlc-switch* and *scheme48*. In many cases the BTB is actually more accurate than using the trace cache. This shows that the model using the trace cache *can* predict indirect branches more accurately than the BTB in some, but not all, cases.

5.2 Trace Cache with Update Policy

In the base trace cache model in **Table 3**, we use the *keep-longest* policy for writing to the trace cache since it is shown to be best policy by *Patel* [15] for maximizing fetch bandwidth. However, if the *always overwrite* policy, which always overwrites the trace cache lines, was used, then updating the indirect branch target addresses would be handled automatically. However, *Patel* showed that the *always overwrite* policy is very ineffective and wasteful in terms of cycle times spent for overwriting.

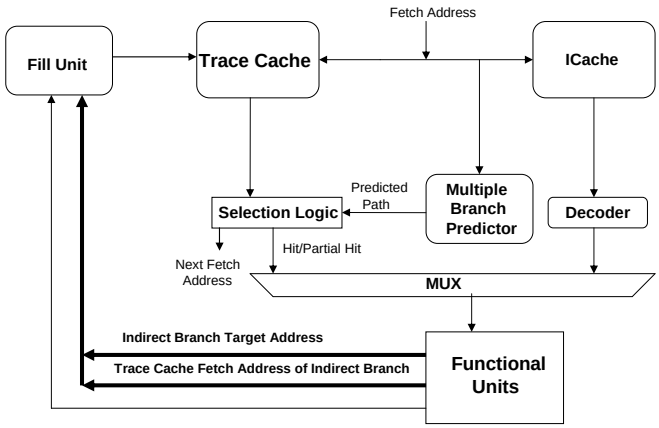

Figure 7: Trace cache model with the update policy

Hence, we slightly modify the trace cache microarchitecture by adding the capability of updating indirect branch targets in the trace cache lines as shown in **Figure 7**. When the indirect branch

instruction is executed and retired, its computed target address (i.e. Indirect Branch Target Address) and the fetch address of the trace cache line to which the indirect branch instruction belongs arrive at the fill unit. The fetch address can be sent to the pipeline as a part of the indirect branch instruction or can be acquired from the reorder buffer since updating the indirect branch target addresses in the trace cache occurs at retire time. If the trace cache line is still in the cache, then the fill unit overwrites the indirect branch target address at the end of the trace cache line. No target updating is performed if the trace cache line has been thrashed from the cache.

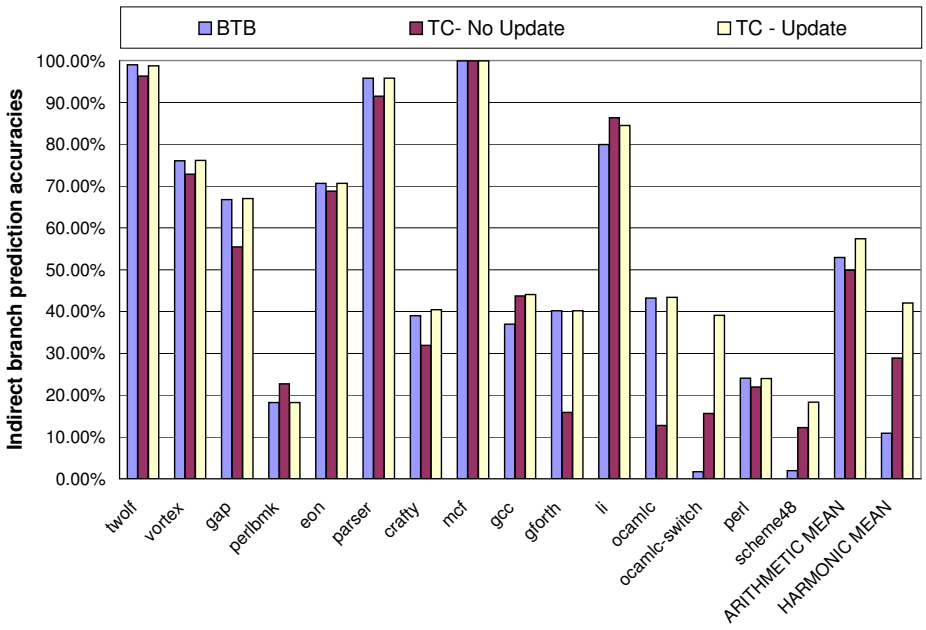

Figure 8: Indirect branch target prediction accuracies of the BTB and the trace cache models with the non-update and update policies

Figure 8 shows the trace cache with the update policy along with the *BTB* and *TC- No Update* models. Updating indirect branch targets in the trace cache performs better than not updating them for *twolf*, *vortex*, *gap*, *eon*, *parser*, *crafty*, *gforth*, *ocamlc* and *perl*. This is expected because for these benchmarks, the *BTB* model also performs better than the *TC- No Update*.

In contrast, *TC- Update* is worse than *TC- No Update* in *perlmbk* and *li*. The limited amount of context information provided by the trace cache gives little benefit to these programs, and recall from *Section 5.1* that these benchmarks benefit from not updating the branch targets.

On the other hand, using *TC- Update* in *gcc*, *ocamlc-switch* and *scheme48* improves the prediction accuracies. This is expected because indirect branch target addresses in these benchmarks change relatively frequently as the program runs, and the updating predictor can take account of these changes. Furthermore, all these programs benefit significantly from the context information provided by the trace cache. Thus, the chance of getting a target address hit for the same indirect branch in the *TC- Update* model is much higher than that of the *TC- No Update* model. As a consequence, *TC- Update* outperforms both *BTB* and *TC- No Update* in *gcc*, *ocamlc-switch* and *scheme48*.

Overall, the average indirect branch prediction accuracy of *TC-Update* across all benchmarks now becomes 42% (57.35%), which is 13% (7.5%)-points better than the *TC- No Update*. In fact, it is even higher than *BTB* by about 31% (5%)-points on the average. Thus, in the following sections, all the benchmarks will update indirect branch targets.

6. Prediction Accuracies of Various Trace Cache Configurations

The goal of the trace cache is to provide sufficient bandwidth to the execution pipeline within the constraints of not taking up too many resources, or causing the design to become so complex that it impacts on clock speed. Several enhancements to the trace cache have been proposed to improve its fetch bandwidth, many of which increase its size and complexity. Many of these enhancements affect the storage of instructions in the trace cache, and thus have the potential to have a knock-on effect on indirect branch prediction. The main purpose of these optimizations is, however, to improve fetch bandwidth, not indirect branch prediction.

In this section, the configuration of the trace cache model with indirect branch target updating is varied incrementally in order to show the effects of each configuration on the indirect branch prediction accuracy. The variations in the configuration consist of applying trace packing, adding 2-bit saturating update counters per trace cache line, varying trace cache set associativity, cache size, cache line size, and finally the combination of all.

6.1 Trace Packing

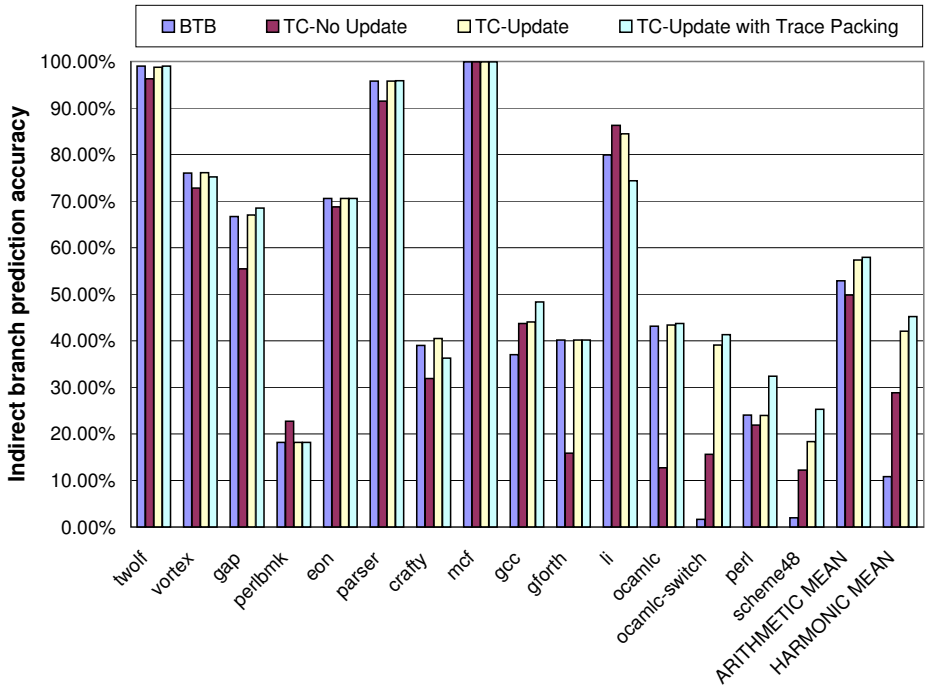

Figure 9: Indirect branch target prediction accuracies of the BTB and the *TC-Update* model with trace packing

Trace packing [15] puts as many instructions as possible into a trace cache line by fragmenting fetch blocks. The fragmentation offers a way to form different trace cache lines, particularly in a loop. Thus, it enables the fill unit to increase the potential of creating more trace cache lines ending with a particular indirect branch instruction. However, it increases the chance of contentions in the trace cache since a large number of trace cache lines can be created.

Figure 9 shows the indirect branch prediction accuracies of the previous models along with *TC-Update with Trace Packing*. Most of the benchmarks enjoy improvement in the prediction accuracies

as compared to the *TC-Update* model. This is the result of indirect branches being split into more separate instances, each with more context information. The improvement is particularly high in *perl* by 8 percentage points and *scheme48* by about 7 percentage points. On the other hand, the prediction accuracies of *vortex*, *crafty* and *li* become worse than the *TC-Update* and even *BTB* models. We believe this is mostly due to trace cache capacity misses caused by the increased number of trace cache lines created by trace packing. However, we found the phenomenon difficult to study, because the breaking of trace cache lines can vary considerably during the running of the program. Overall, the average indirect branch prediction accuracy of the *TC-Update* model with trace packing across all benchmarks is now 45% (57.95%) which is 34.38% (8%), 34.38% (5%) and 3.18% (0.6%) points better than the *TC-No Update*, *BTB* and *TC-Update* models, respectively.

6.2 2-bit Saturating Update Counter

We also propose to extend trace cache lines with 2-bit saturating counters along with indirect branch target addresses in order to increase indirect branch prediction. This is a new addition to the trace cache that is not in Patel's original model. When a new trace cache line ending with an indirect branch is created, it is placed into the trace cache and the 2-bit update counter at the end of the line is set to 1. The update counter is incremented if the indirect branch target is predicted correctly. Otherwise, it is decremented. When the same trace cache line is accessed again, the indirect branch target is not updated until the update counter in the trace cache line reaches zero. This technique is widely used both in conditional branch predictors and in BTBs to improve prediction accuracy. In BTBs it is particularly effective for monotonic indirect branches, that is branches that almost always jump to the same target, but occasionally jump to another target.

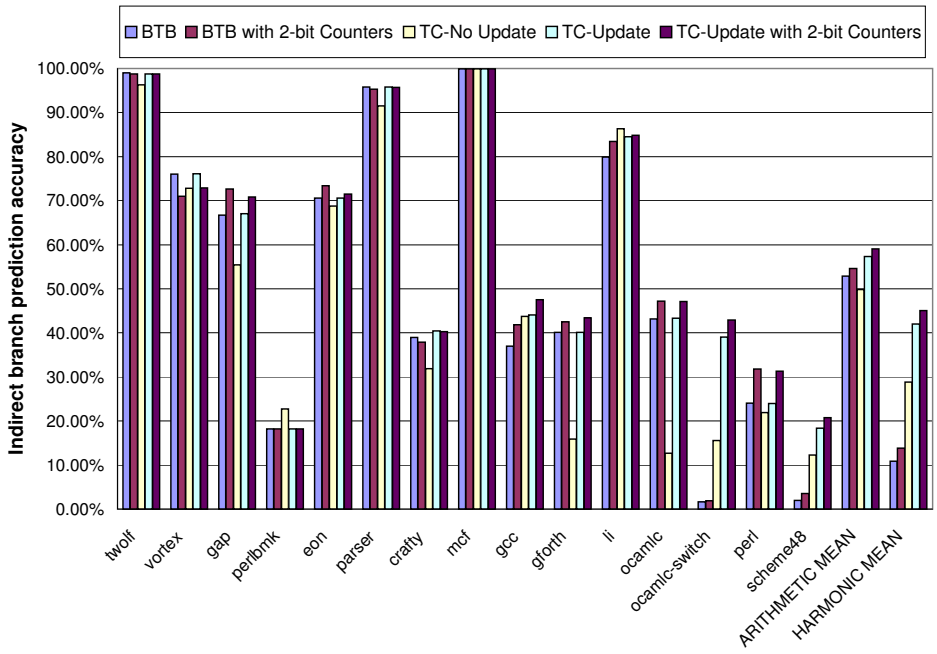

Figure 10: Indirect branch target prediction accuracies of the BTB, *TC-No Update*, *TC-Update* and *TC-Update* with 2-bit update counters

This method of updating indirect branch targets prevents a frequent target from being removed from the trace cache line unless it is shown to be incorrect more than once. This enables benchmarks

whose indirect branch targets are biased to one or two specific targets such as *perlmbk*, *li*, *gcc*, *ocamlc-switch* and *scheme48* to keep these predicted targets in the face of mispredictions. On the other hand, it may reduce the responsiveness of the indirect branch predictor to changes in behavior, as in *vortex*. **Figure 10** shows that the prediction accuracies of all benchmarks improve by using 2-bit saturating update counters with respect to the *TC-Update* model. Overall, the average indirect branch prediction accuracy of the *TC-Update with 2-bit Counters* across all benchmarks is now 45%(59.06%) which is 16%(9%), 34%(6%), 31%(2.72%) and 3%(2%) points better than the *TC-No Update*, *BTB*, *BTB with 2-bit Counters* and *TC-Update* models, respectively. This is a relatively small improvement, but two bits is a small increase in the amount of information stored in each trace cache line so the cost is low.

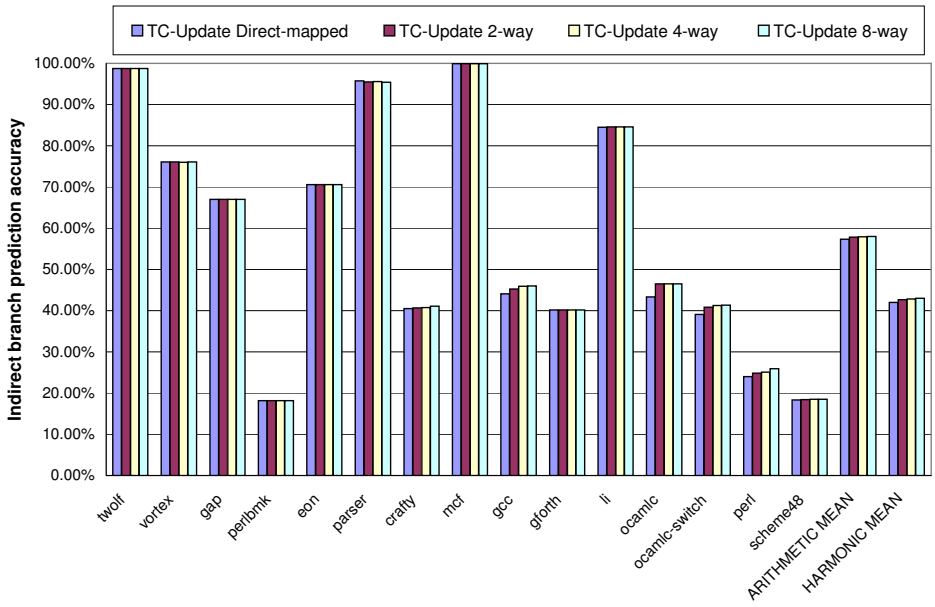

Figure 11: Indirect branch target prediction accuracies of the *TC-Update* model with varying set associativity

6.3 Trace Cache Associativity

We also vary trace cache set associativity to measure the effects of a set associative trace cache on the indirect branch prediction. In a set-associative trace cache, multiple cache lines starting at the same address can co-exist in the trace cache. Using Patel's scheme [15], different trace cache lines starting with the same address are mapped to the same set. The starting address is used to identify the set, and the pattern of m conditional branches is used to identify the line within the set. This allows the trace cache with path associativity to hold multiple cache lines having the same starting address and ending with the same indirect branch instruction because we can index a trace cache line into a set using its starting address and use its branch history to identify whether it is unique. Thus, each will be more specialized versions of the indirect branch, with more context information embedded in the trace cache line.

Figure 11 shows the indirect branch prediction accuracy for 2, 4 and 8-way set associative trace caches with updating. Almost all of the benchmarks show slight improvements in the prediction accuracy as associativity increases. Overall, the average indirect branch prediction accuracies of the

TC-Update model with 2,4 and 8-way set-associative trace caches across all benchmarks are now 42.63%(57.82%), 42.80%(57.93%) and 42.98(58%), respectively.

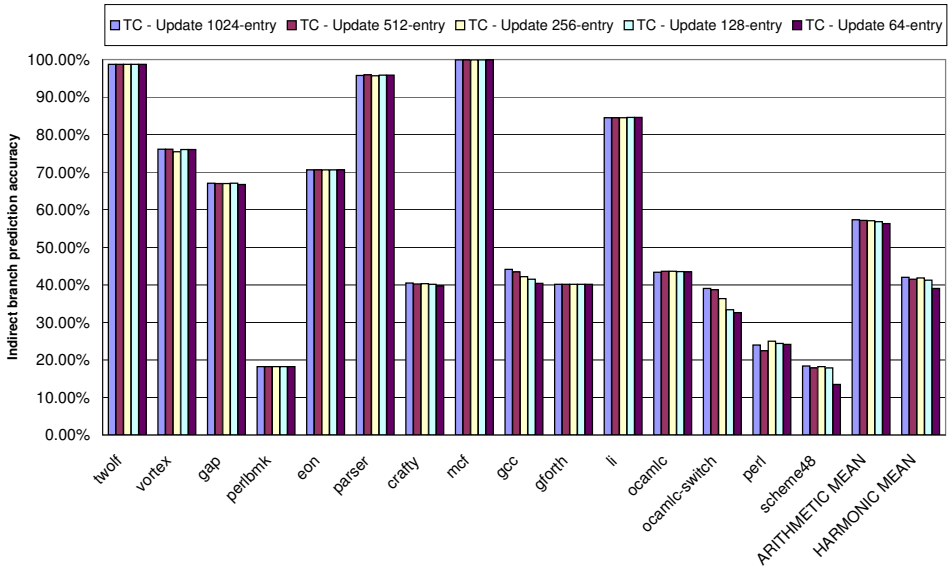

Figure 12: Indirect branch target prediction accuracies of the *TC-Update* model with 64, 128, 256, 512 and 1024-entry trace caches

One complication with interpreting the effects of increased associativity on indirect branch prediction is that there are two effects at work. First, associativity may allow more instances of the same branch with separate targets, as described above. Secondly, associativity simply increases the hit rate of the trace cache (see **Figure 17**) which is also likely to increase indirect branch prediction accuracy. Given the very small increase in accuracy, it is difficult to separate the two effects. Note that some other trace cache designs allow multiple lines in direct-mapped trace caches to start with the same address. However, we doubt that the effects of such approaches would give substantially different results to those we find with Patel's model.

6.4 Trace Cache Size Variance

The cache size of 1024-entry direct-mapped trace cache has been used in the prior runs. We gradually reduce its size from 1024 entries to 512, 256, 128 and 64 entries to capture the sensitivity of the indirect branch prediction to the reduction in the cache size. Clearly, the main goal of varying the trace cache size is to find the optimal balance of hit rate and hardware resources. However, varying the size also has a small impact on indirect branch prediction because it affects the trace cache hit rate. The higher the hit rate, the more indirect branches are likely to be predicted by the trace cache (and thus have the benefits of limited path context information) rather than the BTB.

Figure 12 shows the prediction accuracy results of cache size variance. The overall prediction accuracy percentage point difference between 1024-entry and 64-entry trace caches is slightly greater than 3% (1%). Thus, with a smaller cache size such as 256 entries, it is possible to attain nearly the same indirect branch prediction accuracy across all benchmarks as a 1024-entry direct-mapped trace cache.

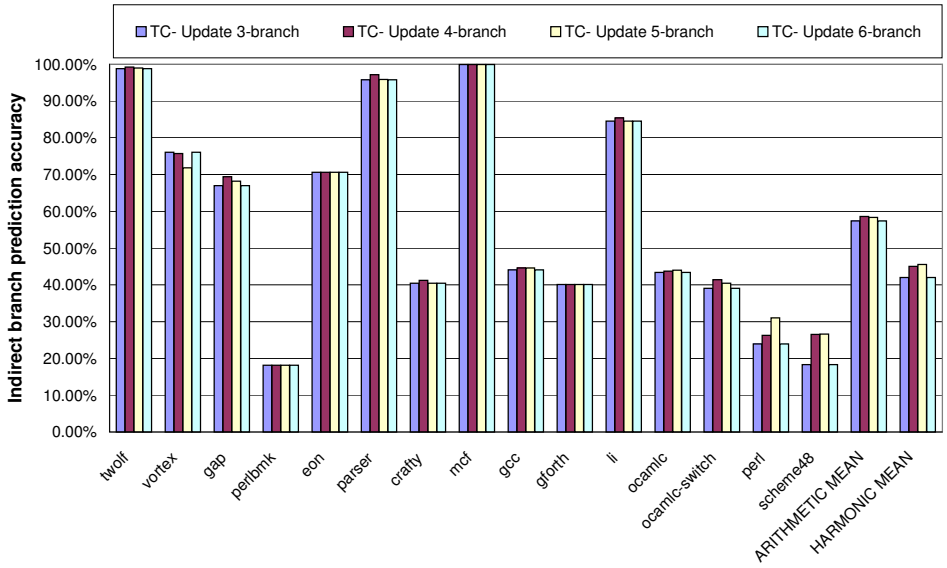

Figure 13: Indirect branch target prediction accuracies of the *TC-Update* model with 3-branch, 4-branch, 5-branch and 6-branch cache line sizes

6.5 Trace Cache Line Size Variance

So far, we have used a fixed trace cache line size of 16 instructions with 3 branches. The cache line size configuration is also varied to measure its effect on the prediction of indirect branches. We use three other cache line configurations: 20 instructions with 4 branches, 24 instructions with 5 branches and 32 instructions with 6 branches. Clearly, the longer the trace cache line, the more context information on recent conditional branch outcomes is stored into it. However, longer lines also lead to more conflict misses, assuming a fixed total size for the trace cache.

Figure 13 shows the comparison of the trace cache models with four different cache line sizes. Out of these four cache line configurations, the 24-instruction-5-branch model performs the best for indirect branch prediction. It is important to note, however that this configuration does not give the highest trace cache hit rates. On the contrary, **Figure 17** shows the hit rates for various trace cache configurations and we see that the configuration with five branches per trace cache line has the worst hit rate of all the variants we examined. Thus, while such a trace cache is interesting because it gives us some indication of the limits on improvements in indirect branch prediction using our scheme, it is highly unlikely to be implemented in practice. The main goal of a trace cache is fetch bandwidth, and so it will be designed to maximize this bandwidth rather than to improve branch prediction.

Improvements in branch prediction accuracy using the trace cache come from the inclusion of a limited amount of context information in the prediction. In **Figure 14** we show the amount of context used in trace cache predictions in a trace cache with up to four branches per line. We show the number of branches in the trace cache line for each indirect branch prediction made using the trace cache. One branch in the line means that the indirect branch is the only branch in the line. Greater numbers of branches show that more path context is being used. The figure indicates that on average around 90% of predictions are made with one or two branches per line.

Figure 15 show the accuracy of indirect branch predictions made with varying number of branches per line. There is a clear trend showing a strong correlation between the amount of path context information captured by branches and the accuracy of the prediction. This is consistent

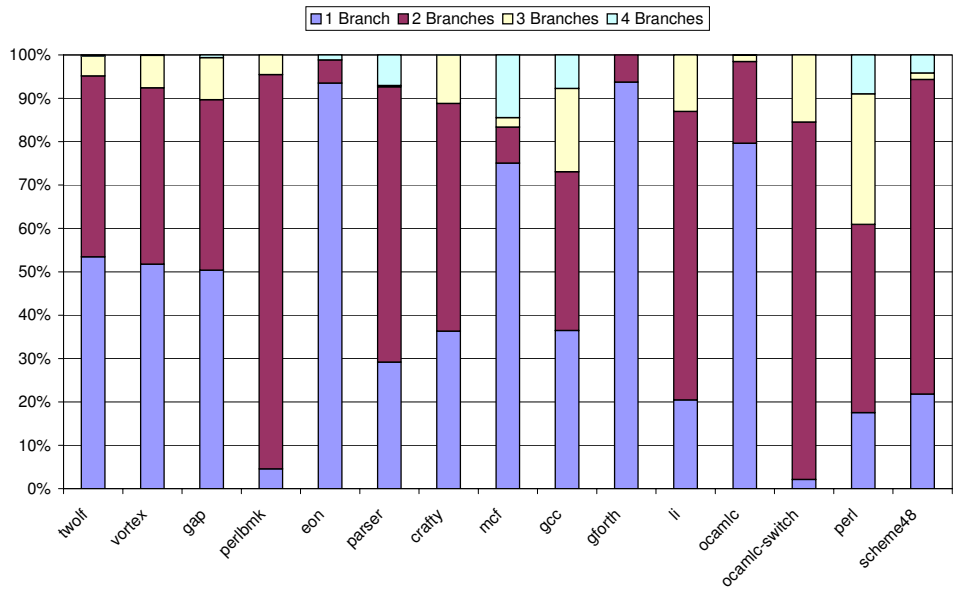

Figure 14: Breakdown of all used trace cache lines ending with an indirect branch based on the number of branches in a trace cache line

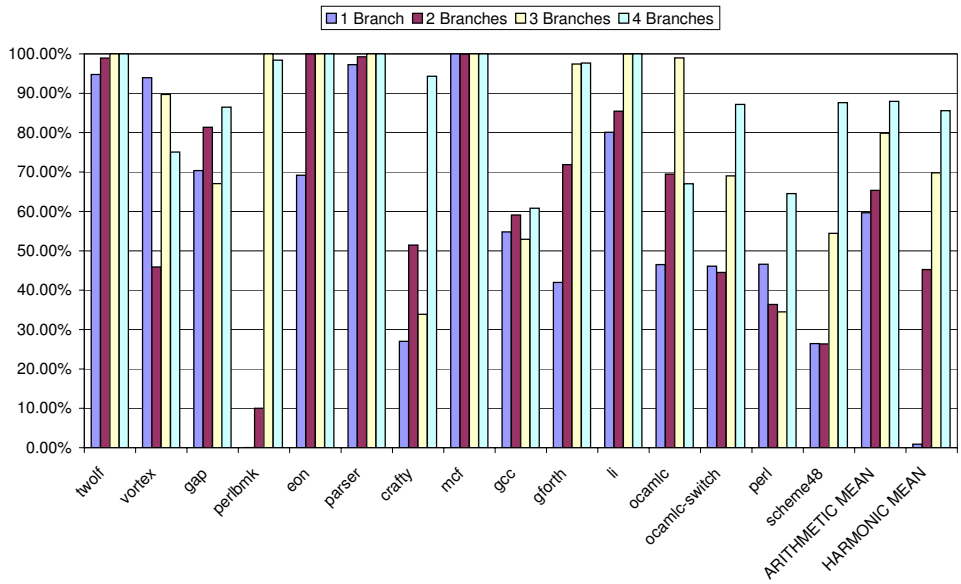

Figure 15: Breakdown of the prediction rates for all used trace cache lines ending with an indirect branch based on the number of branches in a trace cache line

with results in two-level branch predictors which use much greater amounts of context than we are able to capture [5][6][11].

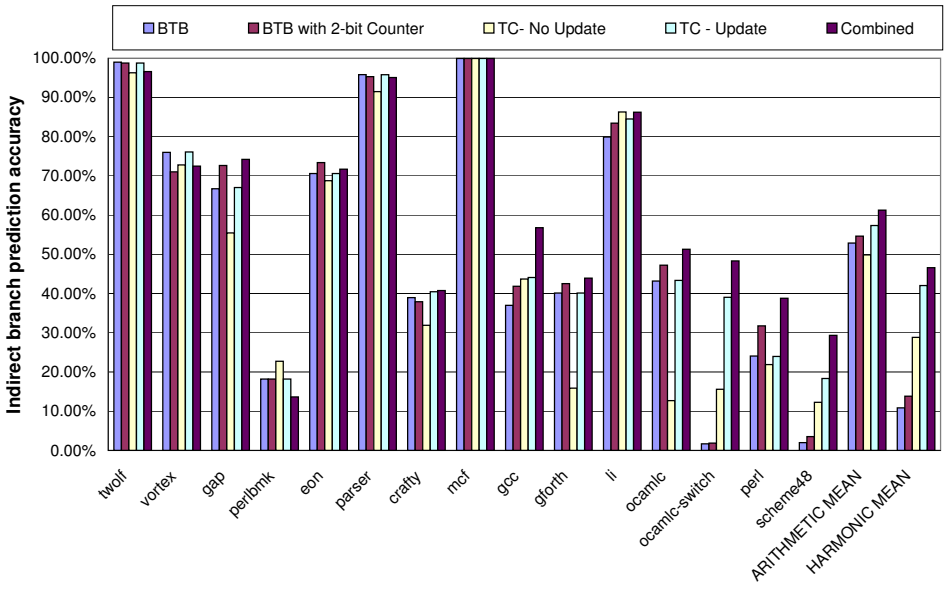

Figure 16: Indirect branch target prediction accuracy of the combined model of 1024-entry, 8-way, 20-instruction-4-branch line size, updating, 2-bit saturating counters and trace packing in comparison to the *BTB*, *TC-No Update* and *TC-Update* models

6.6 Combining Various Configurations

In this section, we present the results of a trace cache model with various trace configurations together. The combined configuration model includes branch target updating policy, 2-bit saturating update counter per trace line and trace packing. Also, we select the best performing parameter from cache size, cache associativity and cache line size. These are 1024-entry, 8-way set associativity and 20 instructions with 4 branches. This model gives the best overall prediction accuracy of the models we investigated. Thus, it gives us some idea of the limit on improvement in branch prediction accuracy that is possible using these techniques.

Figure 16 shows the prediction accuracy results of the combined model in comparison with the *BTB*, *BTB with 2-bit Counters*, *TC-No Update* and *TC-Update* models. Overall, the average indirect branch prediction accuracy of the *Combined* model across all benchmarks is now 46.60% (61.27%), which is about 35.75% (8.40%), 32.76% (6.65%), 17.77% (11.43%), and 4.56% (3.92%) points better than the *BTB*, *BTB with 2-bit Counters*, *TC-No Update*, and *TC-Update* models, respectively.

Figure 17 shows the trace cache hit rates for several different combinations that we have tested. Similarly, **Figure 18** shows the percentage of instructions executed that come from the trace cache. In all cases, the size of the trace cache is 1024 entries. The figure clearly shows that the best approach for optimizing the hit rate that we have examined is to use the base trace cache (16 instructions, 3 branches per line) in an 8-way associative configuration. This configuration is not the one that maximizes prediction accuracy, but it is the one that is most likely to be used, as the main objective of a trace cache is to improve fetch bandwidth.

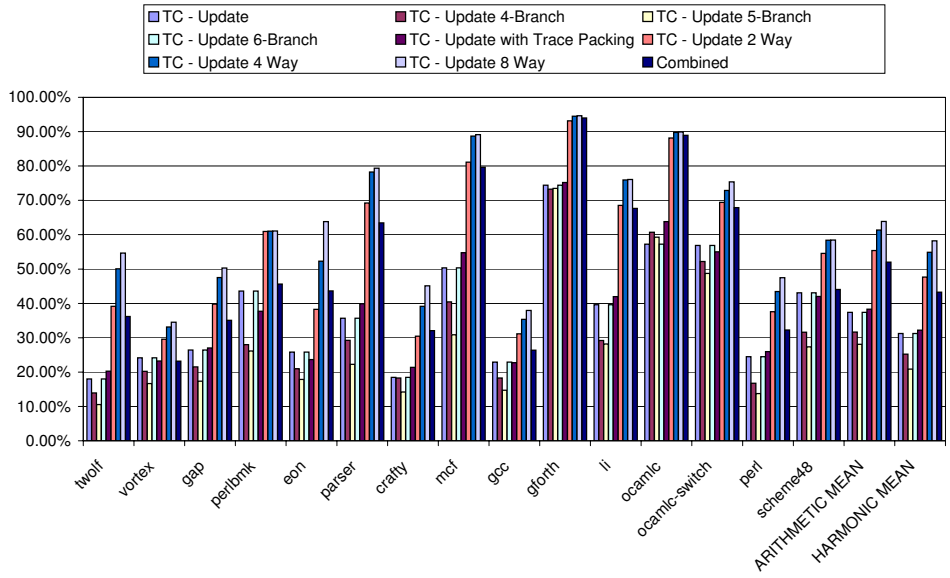

Figure 17: Trace cache hit rates for different configurations.

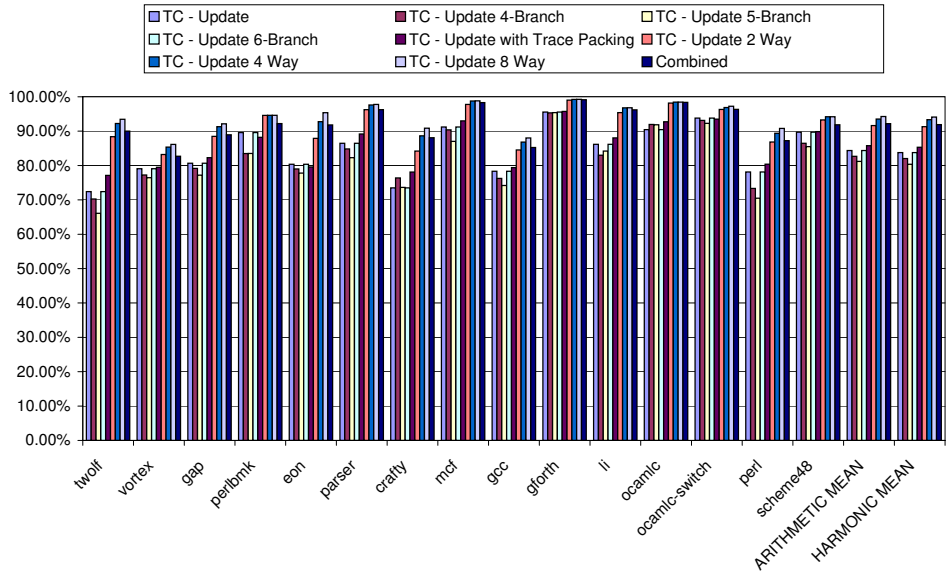

Figure 18: The percentage of executed instructions from the trace cache

7. Other Trace Cache Models

7.1 Real World Trace Cache

The implementation of the trace cache described in this paper is based on Patel's *PhD dissertation* [15]. However, other configurations are possible. For example, the Intel Pentium 4 processor [2] uses a trace cache and two separate BTBs. One of the two BTBs works as normal, but the other is dedicated to branches stored in the trace cache, and appears to be addressed by the location of the branch in the trace cache rather than the branch's address in main memory.

In such a model, if we assume that there is a dedicated BTB entry for each branch in the trace cache, the effect on indirect branch prediction will be the same as our TC-Update model (see section 5.2). Initial experiments using hardware performance counters on the Pentium 4 suggest that the indirect branch prediction accuracies on small test programs with particular indirect branch behaviour are much higher than one would expect with a simple BTB, and that the context stored by the trace cache is reducing the number of indirect branch mispredictions.

To demonstrate this effect, we ran a variant of the program in **Figure 2** on a lightly loaded Pentium 4 machine. In the switch statement in the inner loop we used eight different cases rather than four, so that the compiler implemented the statement with an indirect branch, rather than a tree of conditional branches. We used an array containing values (0,1,2,3,4,5,6,7), so that the target of the next branch can be predicted perfectly, provided we know the outcome of the previous one. We also increased the number of iterations of the loop to ten million in order to make the effect more visible. We used the *perfex* utility to access hardware performance counter measures of the process.

Table 4 shows the range of results from running this program with the same inputs 10,000 times. The runs are divided into rows, grouped according to the number of indirect branch mispredictions that occurred in that run. In 78.54% of cases, there are fewer than 260 indirect branch mispredictions. Among more than ten million indirect branches, this is a misprediction rate of almost 0%. This is exactly the result that we would expect. The trace cache captures sufficient context information to identify the most recent outcome of the indirect branch. In this particular program, this is enough information to perfectly predict the next outcome, as we explained in section 3. Thus, the Pentium 4 behaves exactly as our model predicts in almost 80% of the cases of running this program.

Range of mispredictions	Percentage of total	Average % misprediction	Average cycles
0 – 260	78.54%	0.0017%	64,432,642
261 – 250,000	0.58%	1.14%	66,442,092
250,000 – 500,000	8.88%	4.05%	75,394,126
500,000 – 750,000	7.71%	6.04%	80,425,580
750,000 – 1,000,000	3.24%	8.48%	86,027,876
>1,000,000	1.06%	12.10%	94,912,349
Total	100%	1.24%	67,673,178

Table 4: Pentium 4 indirect branch prediction results on simple benchmark

In the remaining 21.46% of cases there are between 3,993 and 3,090,300 mispredictions, representing a misprediction rate of 0.04% to 30.1%. We see that most of these cases involve 250,000 to 750,000 branch mispredictions. We suspect that the main reason why the indirect branch is less predictable on some runs is that a trace cache line in the Pentium 4 contains only six microinstructions (pre-decoded RISC translations of x86 instructions). To capture useful context, a trace cache line needs to contain both a control-flow and an indirect branch. We suspect that in some cases random effects are causing a poor choice of starting point for some of the trace cache lines, which

result in less context being captured by the trace cache. This is especially likely with short trace cache lines, because the amount of context captured is already likely to be small. However, even where a relatively large number of mispredictions occur, the misprediction rate is very much lower than for a simple BTB which would mispredict almost 100% of the time on this program. **Table 4** also shows the average running time (in cycles) for each group in the final column, which increase sharply in line with the increase in indirect branch mispredictions.

It is important to recall, however, that Intel releases relatively little information about the microarchitecture of their processors. We know that the Pentium 4 has a separate BTB for branches in the trace cache [2]. We also know that the Pentium 4 does not have a two-level branch predictor [9]. However, it is not possible to state categorically that the processor is working as exactly as we describe, although it certainly appears to work in this way. For this reason, we do not examine more complicated programs running on the Pentium 4. There are simply too many unknown variables.

7.2 Trace Cache Context Study

The main point of this paper is to show that the trace cache can capture path context information. This information can yield some part of the benefits of using a two-level predictor in the case where only a single-level predictor is used. In order to demonstrate the generality of this idea, we performed a small experiment using direct conditional branches rather than indirect ones. There is no compelling reason to choose such a strategy in a real processor (two-level conditional branch predictors are important and cheap enough that most processors use them), but same principle should be observed.

We ran the benchmarks with a PC-indexed bimodal branch predictor with no global history for the conditional branches. If the trace cache is used, a trace cache hit overrides the branch predictor (multiple branch predictor is not used and the single branch predictor is only used for I-cache fetch path). A trace cache hit predicts branches implicitly for each embedded branch. The trace cache is configured as the direct-mapped one with 1024 entries, which means only one trace cache line for the same starting fetch address based on our trace cache implementation. We use two schemes here:

Scheme 1: we embed 2-bit saturating counters in the trace cache lines in the same way as used in the bimodal branch predictor and replace a trace cache line only if it contains one or more weak counters or the new trace cache line is longer the existing one.

Scheme 2: we don't embed 2-bit saturating counters in the trace cache line and a trace cache line is updated using the same updating policy (ie. keep-longest policy) used in the earlier experimentations.

Figure 19 shows that the direction prediction rates with a trace cache (scheme 1 and 2) do indeed show slight improvement (around 1%) over those without a trace cache. Although the gain in accuracy is measurable, it is small. The main reason is that the bimodal branch predictor is already quite accurate and the small amount of additional context information on trace cache hits is insufficient to help much.There is not much difference between the two schemes we examined.

7.3 Other Predictors

The other main branch prediction model for the trace cache is to use a next-trace/next-stream predictor [10] [22], rather than predict individual branches. These give much better prediction accuracies for indirect branches than any predictor described in this paper, at the cost of a significant increase in complexity. In comparison, the effect we measure gives a much smaller improvement in indirect branch prediction accuracy, although at very low cost beyond that required for the trace cache. However, it is important to note that the main contribution of this paper is not to advocate using the trace cache to get better indirect branch prediction, rather than some other approach. Our main contribution is to explore the observation that the trace cache captures context information

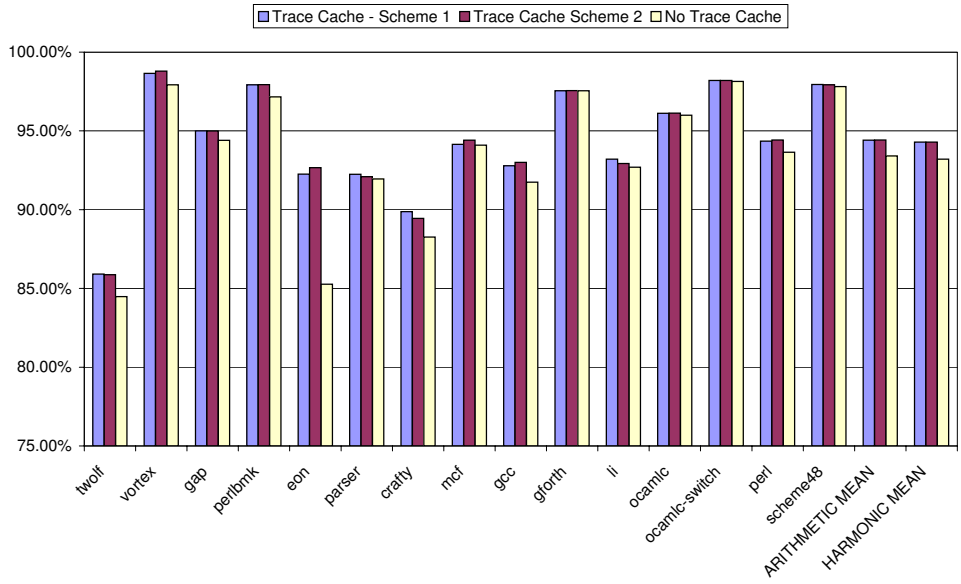

Figure 19: Bimodal direction predication rates with/without a trace cache (1024 direct-mapped with 3 branches)

about the control-flow of the program, and this can have a significant impact on other predictors that do not use such context.

An interesting piece of future work would be to measure the effect of our scheme on the overall performance of the processor in instructions per cycle (IPC). We believe that the impact on IPC is likely to be small because (1) indirect branches make up a relatively small number of overall executed instructions and (2) the improvement in prediction accuracy is modest. However, if the trace cache is used for prediction, it raises the possibility of trace cache misses and branch mispredictions tending to occur simultaneously. The overall effect on IPC is not clear.

Although this paper has examined the effect of the trace cache on indirect branch prediction, there may be a similar impact on other forms of prediction. A wide variety of predictors have been proposed in the literature, including load address predictors, load value predictors and data dependence predictors. In each of these predictors there is typically a table of recent values, which is indexed by some function of the address of a particular instruction in main memory.

In a processor with a trace cache the option arises of accessing these tables with the location in the trace cache rather than the address in memory. For example, it may be more efficient to use the location in the trace cache, as this would require no further work to map it back to the original location in memory. (Mapping back to this original address would be necessary to maintain the classical behaviour of such a predictor). It seems likely that this is the reason that Intel chose to have a separate BTB for branches in the trace cache in their Pentium 4 architecture.

However, as we have shown in this paper, if there are separate entries in a predictor for each copy of an instruction in the trace cache, there can be a significant impact on predictor accuracy. In the case of indirect branches, the impact is mostly positive, because the trace cache captures context about the recent control flow of the program. In fact, this may even have been the motivation for the separate BTBs in the Pentium 4 architecture. Furthermore, choosing to index a table by trace cache location rather than location in memory creates a further dependence: the prediction accuracy or the use of any such predictor will depend on the trace cache hit rate. Since the trace cache will be sized and organized for fetch bandwidth, the overall effect on prediction accuracy may be worse than

could be achieved with other design goals. Being aware of and measuring such effects is important for any processor designer considering design alternatives surrounding the trace cache.

8. Related Work

In a broader discussion of trace cache design, Rotenberg [18] notes that the sequence of branch outcomes stored in a trace cache line implicitly captures some execution path context. Thus, we are not the first to remark on this effect. However, there is no in-depth investigation of this phenomenon.

Peleg and Weiser [16] embed one-bit or two-bit counters inside traces, so that the trace cache does branch prediction merely by the act of supplying a trace. Accuracy may be slightly better than a pure bimodal predictor, because the same branch may be in multiple traces. Thus, the counters also become replicated and some measure of the context-sensitivity of a two-level predictor may be captured by their scheme.

Branch promotion [15] move the predictions of some very predictable branches to the trace cache. This has the effect of reducing pressure in the external predictor. Another effect is that branches are replicated in the trace cache because the same branch can appear in multiple lines, potentially increasing accuracy because each specialized version is biased independently.

Lee et al [13] investigated value prediction using a trace processor where value prediction is decoupled from the instruction fetch stage. Part of their scheme is to use a trace cache in which the instructions that will use value prediction are pre-identified. In such a scheme it is possible for multiple instances of the same instruction to appear in different trace cache lines, with implications for prediction accuracy.

9. Conclusion

In this paper, we have discussed the effects of using the trace cache on the indirect branch prediction in ILP processors. If the target addresses in the trace cache lines are used to predict indirect branches, the accuracy can be significantly different from that that achieved with a simple branch target buffer. The main reason for this is that the trace cache captures context about the recent control flow of the program. To our knowledge, this effect was not foreseen in the original design of the trace cache, and has remained underreported since then.

With a simple mechanism of updating indirect branch target addresses in the trace cache lines, we have shown that indirect branch prediction accuracy can be moderately improved with very little or no cost at all. We have analyzed various trace cache configurations and strategies such as applying trace packing, adding 2-bit update counters per trace cache line, varying trace cache set associativity, cache size and cache line size and tune them to measure the positive/negative effects of each configuration/strategy on indirect branch prediction accuracy. Finally, we have constructed a model combined with the best performer from each configuration/strategy.

Our experimental results have shown that the harmonic mean indirect branch prediction accuracy, across several benchmarks, using a trace cache model with trace cache parameters tuned for the highest prediction accuracy and *updating* indirect branch target addresses can be up to *35.75%* better than the BTB, and up to *17.77%* better than that of a trace cache with *no updating* indirect branch target addresses.

Although this paper has examined the effect of the trace cache on indirect branch prediction, there may be a similar impact on other forms of prediction. For example, a load value predictor might be affected if there were separate entries for each copy of a given load in the trace cache. Measuring this effect would be important for any processor designer who wishes to address their predictor using locations in the trace cache rather than the address of the original instruction in main memory.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer, IEEE Computer Society*, pages 59–67, February 2002.

[2] D. Boggs, A. Bathka, J. Hawkins, D. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), February 2004.

[3] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st Annual International Symposium on Principles of Programming Languages*, January 1994.

[4] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, June 1997.

[5] Y. Chu and M. R. Ito. An efficient indirect branch predictor. In *Proceedings of the 7th European Conference on Parallel Computing (Euro-Par 2001)*, 2001.

[6] K. Driesen and U. Holzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, June 1998.

[7] K. Driesen and U. Holzle. Multi-stage cascaded prediction. In *Euro-Par'99*, volume LNCS 1685, pages 1312–1321, 1999.

[8] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th European Conference on Parallel Computing (Euro-Par 2001)*, volume LNCS 2150, pages 403–412, 2001.

[9] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The Intel Pentium M processor: microarchitecture and performance. *Intel Technology Journal*, 7(2):20–36, May 2003.

[10] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[11] J. Kalamatianos and D. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, November 1998.

[12] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Vol.1, June 2002.

[13] S.-J. Lee, Y. Wang, and P.-C. Yew. Decoupled value prediction on trace processors. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 231–240, Toulouse, France, January 2000.

[14] D. J. Lilja. *Measuring Computer Performance - A Practitioner's Guide*. The Press Syndicate of the University of Cambridge, The Pit Building, Trumpington Street, Cambridge, United Kingdom, first edition, 2000.

[15] S. J. Patel. *Trace Cache Design for Wide-Issue Superscalar Processor*. PhD thesis, Computer Science and Engineering in The University of Michigan, 1999.

[16] A. Peleg and U. Weiser. Dyamic flow instruction cache memory organized around trace segments independent of virtual address line. United States Patent 5,381,533, January 1995.

[17] A. Ramirez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proceedings of the 13th international conference on Supercomputing*, pages 119–126, 1999.

[18] E. Rotenberg. *Sepculative Execution in High Performace Computer Architectures*, chapter 4. CRC Press, 2005. (Eds) D. Kaeli and P.-C. Yew.

[19] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, 1996.

[20] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 1999.

[21] O. J. Santana, A. Falcon, E. Fernandez, P. Medina, A. Ramirez, and M. Valero. A comprehensive analysis of indirect branch prediction. In *Proceedings of the 4th International Symposium on High Performance Computing (ISHPC-IV)*, May 2002.

[22] O. J. Santana, A. Falcon, A. Ramirez, J. L. Larriba-Pey, and M. Valero. Next stream prediction. Technical Report UPC-DAC-2002-15, Technical Report, April 2002.

[23] O. J. Santana, A. Ramirez, J. L. Larriba-Pey, and M. Valero. A low-complexity fetch architecture for high-performance superscalar processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(2):220–245, June 2004.