Towards a More Efficient Trace Cache

Rajnish Kumar, Amit Kumar Saha, Jerry T. Yen
Department of Computer Science and Electrical Engineering
George R. Brown School of Engineering, Rice University
{rajnish, amsaha, yen}@rice.edu

***Abstract*— If the trace cache size is not large enough to contain all of the basic blocks of running application, a judicious hit and replacement logic becomes very important. This report proposes a weight-based technique to select the victim line in trace cache for the replacement logic. It also presents a judicious line-fill buffer logic which is found to decrease the redundancy in the trace cache. We did the performance study by simulating these techniques on SimpleScalar. For SimpleScalar test benchmarks applications, a trace cache with the proposed replacement and line-fill buffer logic was found to provide 1-5% better IPC than a trace cache with a Least Recently Used replacement logic.**

1 INTRODUCTION

Superscalar processors are now capable of executing a large number of instructions per cycle. In order to benefit fully from instruction-level parallelism (ILP) techniques, we must prevent the instruction fetch performance from becoming a bottleneck. There are a number of factors that limit the capabilities of current instruction fetch mechanisms. These factors include instruction cache hit rate, branch prediction accuracy, branch throughput, noncontiguous instruction alignment, and fetch unit latency [1].

One possible solution for addressing some of these issues is the trace cache as proposed by Rotenberg, Bennett, and Smith [1]. The trace cache provides a method of storing the dynamic instruction stream making otherwise non-contingous instructions appear continguous. It operates by storing up to *n* instructions per trace cache line and using *m* branch predictions per cycle. The line is filled when a new address is encountered and typically contains *m* branch outcomes. The line from the trace cache is then sent to the decoder when the same starting address for the line is encountered and the branch predictions are matched correctly. An implementation of the trace cache fetch mechanism can be seen in Figure 1.

To increase the performance of trace cache, many techniques have been proposed by Rotenberg et al [1] and others [2,4]. However, none of the proposed techniques deal with the hit and replacement logic or line-fill-buffer logic explicitlty. The following section introduces why the hit and replacement logic is important and proposes an outline for a possible solution.

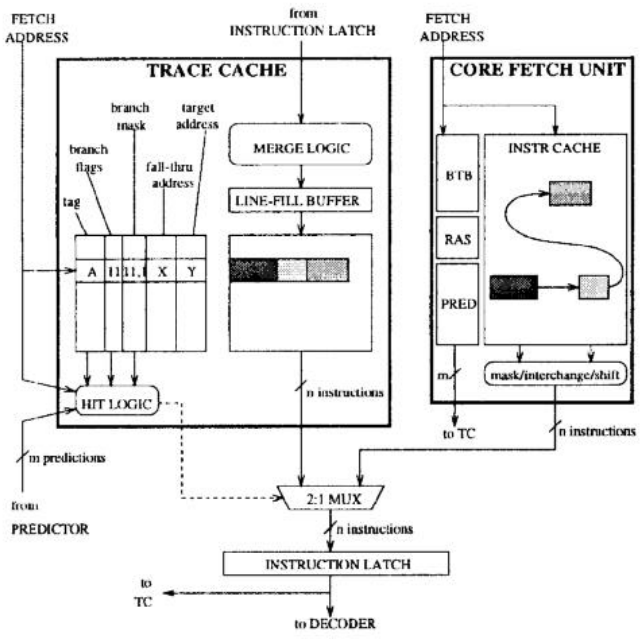
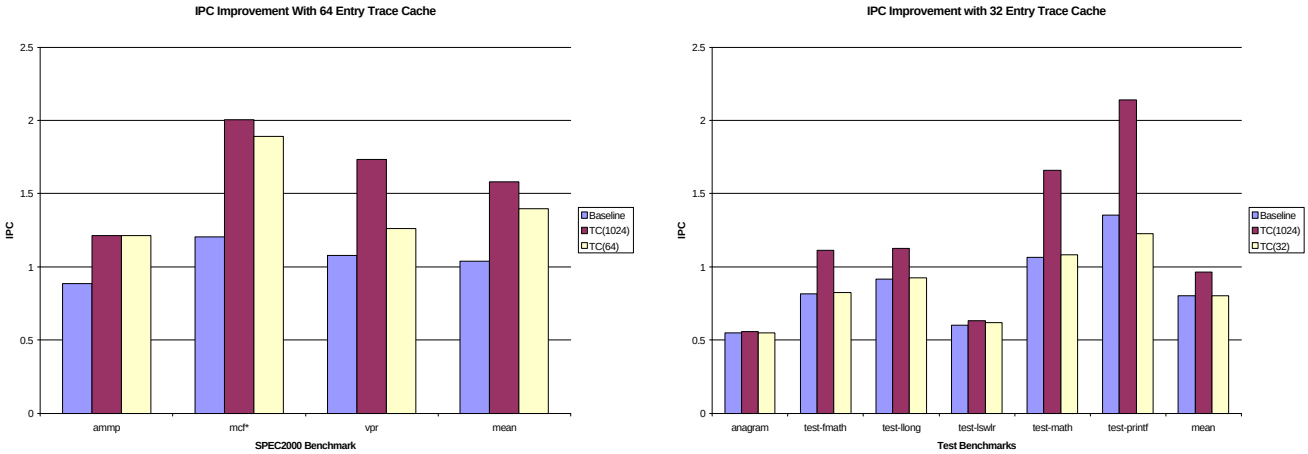


Figure 1: The trace cache fetch mechanism[1]

2 MOTIVATION AND HYPOTHESIS

By comparing the size of SPEC2000 applications with those of previous years, we can easily see a trend towards large-sized applications. The trace cache will give ideal performance when it is large enough to keep all of the basic blocks of the application. Since applications are very large and trace caches are very limited in size, better logic must be provided to control which basic blocks should be placed in the trace cache based upon some hit and replacement logic. The performance of applications will benefit from a trace cache having a judicious hit and replacement logic.



(a) (b)

Figure 2: Possible Improvement with Trace Cache (a) SPEC2000 Benchmarks (b) SimpleScalar Test Benchmarks

The simple Least Recently Used (LRU) based technique to identify the victim line in trace cache may not be very helpful if the trace cache size is very small. This is due to the fact that there may be many trace cache lines with fewer basic blocks than the line can store, and thus wasting trace cache capacity. Similarly, a trace cache line may have a block sequence with many non-taken branches, thus logically giving a continuous stream. We can use these properties of the lines to decide how to select a victim line in the trace cache to be replaced with the new line. As a result, we are proposing a weight-based technique as the replacement logic.

The other problem which becomes more prominent in the case of having small trace cache is shown in Figure 3 and explained in more detail in the next section. It involves redundancy in the trace cache due to the presence of multiple entries of same basic block. To avoid this redundancy, we propose a more judicious line-fill buffer logic.

Therefore, we hypothesize that a weighted replacement logic and a modification in the line-fill buffer logic will increase peformance. Our hypothesis was based upon the techniques of selecting the victim line and avoiding redundancy in the trace cache judiciously. Using an extremely large trace cache, where no replacement policy is required because blocks will always have room to be entered into the trace cache, and a large number of execution resources, the possible performance improvement by the trace cache can be seen. In Figure 2, there is room for an average performance increase of 10-20% over the case of a processor with a limited size trace cache using LRU replacement policy.

The techniques proposed will benefit most scientific applications which are comprised of numerous loops and non-contiguous code. If there are not a lot of cyclic calls or loops in the application code, then an LRU-based policy may be more useful. However, in such applications, a trace cache will only have marginal improvements.

3 ARCHITECTURE

3.1 The basic trace cache design

The trace cache was implemented in a similar manner as depicted in Figure 1 with full associativity. In our implementation *m* is 3 and *n* is 16. The core fetch unit is the same as the fetch unit used in the SimpleScalar simulator [3]. It fetches instruction from only one instruction cache line per cycle. If there is a miss in the instruction cache, then it blocks until the miss is completed. Once the instructions are fetched, they are placed in a dispatch (or decode) queue. If the fetch unit encounters a branch, it utilizes the branch predictor to obtain the correct cache line to access.

The trace cache is filled by taking instructions after the committed stage. Past research has found that this does not improve or worsen the hit percentage. The length of the line-fill buffer is limited by either the number of instructions n or the number of basic blocks m, whichever comes first.

Once the line-fill buffer is filled, the line is flushed to the trace cache. If there is an empty line in the trace

cache, this line is simply copied to that position, else the replacement logic will determine which line it will replace.

Our trace cache also utilizes a partial matching policy, where a block of instructions will hit in the trace cache even though the entire line in the trace cache does not hit. For example, if a trace cache line contains

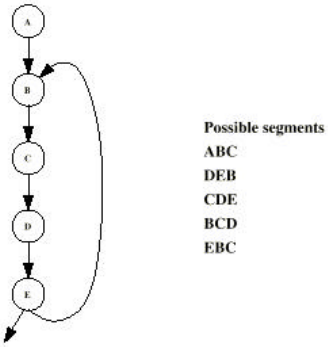


Figure 3: Loop creating five possible segments[4]

ABC instruction blocks, and the predictor predicts ABD, then the trace cache will allow blocks A nd B to hit and sent to the decoder. As presented in [2], this should improve processor performance over a trace cache without this capability.

In addition, the current trace cache combines blocks in such a way that multiple copies of the same basic block reside in the trace cache. For example, as seen in Figure 3, a loop with five basic blocks could potentially create five different combinations all in the trace cache at the same time.

The trace cache is highly dependent on the accuracy of the branch predictor. In our implementation, the mulitple branch predictor (located in the core fetch unit) uses a two level adaptive predictor, which has been shown to achieve a high degree of accuracy [4]. This adaptive predictor uses a global history register and pattern history table to make its prediction. As seen in Figure 4, the global history register is made up of k bits (k depends on how many branches will be predicted).

For example, if 3 branch predictions are desired, then 3 bits are needed. These bits predict the first branch. The next 2 bits predict two possible branches. Using the first branch, the second branch is predicted. The third branch is predicted in a similar manner using the last bit of the global history register and the previous two branches. Although this representation of the

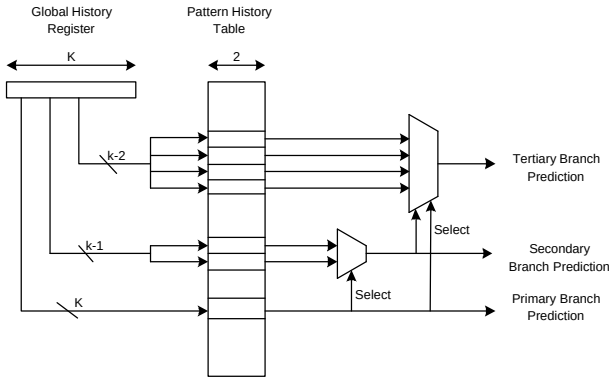


Figure 4: Two-level adaptive branch predictor

branch predictor does not scale well, there are other implementations of this branch predictor that will allow the hardware to scale to more predictions.

The rest of the processor remains unchanged from the architecture used in the SimpleScalar simulator [3]. As a basis to compare our results and to provide a picture of the processor, Table 1 shows the parameters used in our baseline processor.

Processor Core			
Instr Fetch Queue Size	4	RUU Size	16
Branch Mispred Penalty	3	Load/Store Queue	8
Ratio: Front End Speed to Execution Core	1	Number of Integer ALUs	4
Decode Width	4	Number of Integer Multipliers/Dividers	1
Issue Width	4	Number of memory system ports	2
Issue Inorder	False	Number of Floating Point ALUs	4
Issue Wrongpath	True	Number of Floating Point Multipliers/Dividers	1
Commit Width	4		
Memory Hierarchy			
L1 Data config	dl1:128:32:4:l	L1 D-cache Latency	1
L2 Data config	ul2:1024:64:4:l	L2 D-cache Latency	6
L1 Instr config	il1:512:32:1:l	L1 I-cache Latency	1
L2 Instr config	dl2	L2 I-cache Latency	6
Instr TLB	itlb:16:4096:4:l	Instr/Data TLB Miss Latency	30
Data TLB	dtlb:32:4096:4:l		
Memory Bus Width	8		
Memory Latency (first, rest)	18, 2		
Branch Prediction			
Branch Prediction Type	2-level adaptive		
2-level Predictor Config	1 1024 10 0		
Return Stack Size	8		
BTB config	512 4		
Speculative Update	<null>		

Table 1: Baseline SimpleScalar Configuration

In Table 1, the memory configuration settings for the data and instruction caches and the TLBs are described as <cache name>:<no. of sets>:<block size>:<associativity>:<replacement policy> (in our case l is used for LRU). The 2-level predictor configuration specifies <l1size> <l2size> <hist_size> <xor> (where l1size: number of entries in the first-level table (Global History Register), l2size: number of entries in the second-level table (Pattern History Table), hist_size: history width, xor: to xor history and the address in the second level of the predictor). The BTB configuration specifies <sets> <assoc> (the number of sets and associativity).

3.2 Weight-based hit and replacement logic

To identify the victim line for replacement, we associate a weight to every trace cache line and declare the line with minimum weight as the victim one. The weight function logically encaptures the question, "how important is the line that it needs to be present in the trace cache?" We have identified following factors to decide a line's importance:

- Expected future use: We store a 2-bit counter to keep track of the number of times that the line was hit recently. We maintain a time stamp that states when the line was hit last. This time stamp is used to identify whether the hit-counter value represents a recent hit or not.
- Number of basic blocks: If a line consists of *m* basic blocks, we consider that line as using the trace cache resources more optimally than a line with a fewer number of basic blocks. However, it is possible for a line to have fewer than *m* basic blocks. This will be explained further in section 3.3 when we introduce the line-fill-buffer logic to avoid redundancy in the trace cache.
- Non-contiguity of the line: We consider a line to be fully non-contiguous if every basic block in that line starts with a taken-branch. The more number of 1's in the branch-prediction values of the line, the more non-contiguous it is. Therefore, we give higher weight to the more non-contiguous line. Normally, if a line is continuous, it would have been fetched from the I-cache equally fast without blocking.

To help calculate the weight of a line, each line maintains four extra fields apart from the ones kept in the basic trace cache design. Appendix A shows the code used to calculate the weights. The four extra fields are basic block count, zero-count in the branch-prediction values, hit-count, and last-time-hit sim_cycle.

There is one global field, called active_window_size. This field logically maintains an estimate of how far back the application flow needs to go to execute the loops, i.e. how much temporal locality the application contains. This field, associated with last-time_hit sim_cycle, states whether the value of hit_count corresponds to the recent uses of the line or not. We have used this to be 500 for our experiments.

The cost to maintain these fields and how many cycles required to identify the victim line will be discussed in the section 3.4.

3.3 Line-fill-buffer logic: avoiding redundancy

Consider the loop shown in Figure 2. The basic trace cache design will lead to multiple block sequences, having a lot of redundancies. This can be avoided by not allowing such blocks to enter the line-fill buffer.

Whenever a branch instruction is committed, a new basic block is started in the line-fill buffer of the *basic* trace cache design. If adding this new block violates the trace cache line constraints, such as the limit on *m* and *n*, the line fill buffer is saved in the trace cache before starting a fresh line in the line-fill buffer with the new basic block. Therefore, the only constraints which governs the flushing of the line-fill buffer to the trace cache are the constraints *m* and *n*.

We add an additional constraint here in order to flush the line-fill buffer: if the new block's starting address and the corresponding branch-prediction value match those of an existing entry in the trace cache, the line-fill buffer is immediately flushed to the trace cache. If the existing entry in the trace cache is the starting block of a line, then we *do not* start the block in the line-fill buffer because this would mean that the new block is the starting point of more than one block (like block A in Figure 2). This new constraint will lead to trace cache lines with a fewer number of basic blocks than *m*, and this may seem to waste trace cache line capacity. However, this logic combined with the weight-based replacement logic will make such lines (with fewer

number of basic blocks) to be removed from the trace cache as the victim line, thus optimizing the use of trace cache resources.

3.4 Technology cost of the proposed changes

To implement the proposed weight-based replacement logic, four extra fields for each trace cache line need to be maintained. These fields together will cost around three bytes per line, which is not a significant overhead. The main concern is the cost of implementing the weight-calculation utilizing these field values. For this, additional arithmetic logic will be required for every line, which may be exhorbitant. This can avoided by using a simple approximated weight function.

Similarly, the space requirement for implementing the line-fill buffer logic is not significant, but the logic that compares the start address of newly arrived basic blocks with existing entries in the trace cache may be costly to implement. An approximate approach, which compares the newly arrived block with only the starting block of every trace cache line will be simpler to implement, and will remove the redundancy problem to some extent.

Once the logic is in place for our proposed modification, it will be handle a larger trace cache without the need to add additional hardware. Therefore, we do expect it to scale well with future technology.

4 EXPERIMENTAL METHODOLOGY

A trace cache architecture, as described in the previous section, was added to the SimpleScalar simulator. Using the same parameters as the baseline architecture, we simulated the performance of the processor with a trace cache without any of our proposed modifications. In this case, the replacement logic of the trace cache utilized a Least Recently Used policy.

In order to fully evaluate the potential of the trace cache, we modified the baseline to remove the possibility of the execution resources becoming a bottleneck. This involved increasing the parameters as shown in Table 2.

Instruc Fetch Queue Size	2048
Instruc Decode Width	16
Instruc Issue Width	16
Instruc Commit Width	16
RUU Width	256
Load/Store Queue	128
L1 Data Config	dl1:1024:256:4:1
L2 Data Config	ul2:8192:512:4:1
L1 Instruc Config	il1:512:32:1:1
Number of Integer ALUs	8
Number of Integer Mult/Div	8
Number of Floating Point ALUs	8
Number of Floating Point Mult/Div	8

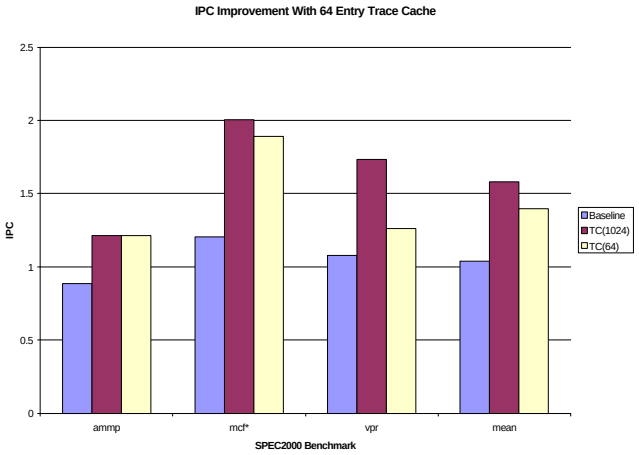
Table 2: Basline configuration with large number of execution resources

The unmodified and modified trace cache (both large and small as well) were simulated with this large number of execution resources. This places an upper bound on how much improvement can be achieved by each type of trace cache.

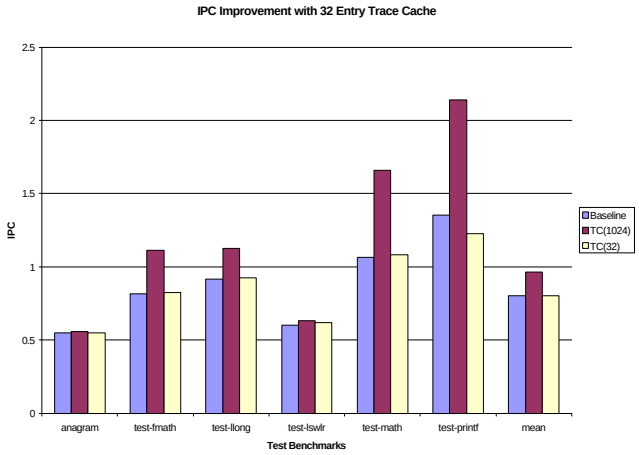
Using this large number of execution resources, a trace cache with our proposed weights was simulated. To illustrate the importance of the weight parameters, we compare the baseline with two different policies. The first policy uses the weights proposed in the section 3.1. The second policy is based on the distance between instructions (i.e. the further an instruction is from the one just executed, the more likely it will be removed from the trace cache).

In addition, we evaluated a modified policy for the line-fill buffer logic. In this case, if a new block enters the line-fill buffer which contains the same starting address as that of the first block of a line already in the trace cache, the line-fill buffer will be flushed into the trace cache. The new block will not be put in the line fill buffer. Earlier in the line fill buffer logic proposed in section 3.2, we used to compare the starting address of the new block with *any* existing block in the trace cache. The modified policy is simple to implement.

SPEC2000 benchmarks [5] were used to evaluate the performance of the processors. Specifically, two integer benchmarks and one floating point benchmark were used (reduced data sets were used). (Note: The mcf benchmark was only run to half completion due to memory constraints). In addition to the SPEC2000 benchmarks, some test benchmarks provided by SimpleScalar were used.



(a)



(b)

Figure 5: IPC Improvement with Smaller Trace Cache (a) 64 Entry with SPEC2000 Benchmarks (b) 32 Entry with Simplescalar Test Benchmarks

5 EXPERIMENTAL ANALYSIS

For measuring performance, we used instructions per cycle (IPC). The harmonic mean was used to average the performance of the benchmarks. For all the results presented, the baseline refers to a processor with a large number of execution resources. For all our simulations, this baseline was much higher than a baseline without the large number of execution resources.

Even with an unmodified trace cache, the experiments show that the fetch queue gets filled up early during the execution of the program. Once the fetch queue is filled, the benefits from the trace cache are limited because even though there are trace cache hits, we are unable to place those instructions in the fetch queue. Even with increased execution resources, the fetch queue was still filled, but at a later stage. This problem will allow very little performance improvement, even with a modified trace cache.

In Figure 5a (which is the same as Figure 2a), we see that even with a more realistic trace cache of 64 lines, there is still a significant performance improvement over the baseline. However, it illustrates the fact that improvement to the trace cache can provide performance enhancements. There is an interesting result in the figure which shows equal performance with the ammp benchmark even with a smaller trace cache. Normally, we would expect performance to decrease with a smaller trace cache. However, the reason for this result is that the ammp benchmark has few loops and branches and does not need a large trace cache.

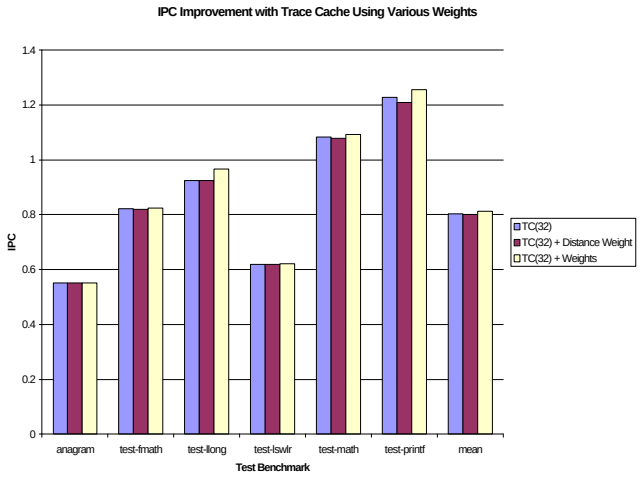


Figure 6: Trace Cache using different weightings to determine replacement logic

With an even smaller trace cache (for smaller benchmarks), Figure 5b (which is the same as Figure 2b), we also see that the opportunity for performance enhancement is large. One interesting result is the test-printf benchmark. With a small trace cache the performance is actually worse than the baseline. This is attributed to the fact that the trace cache is too small such that many of elements in the trace cache are removed before they are used again. In addition, the

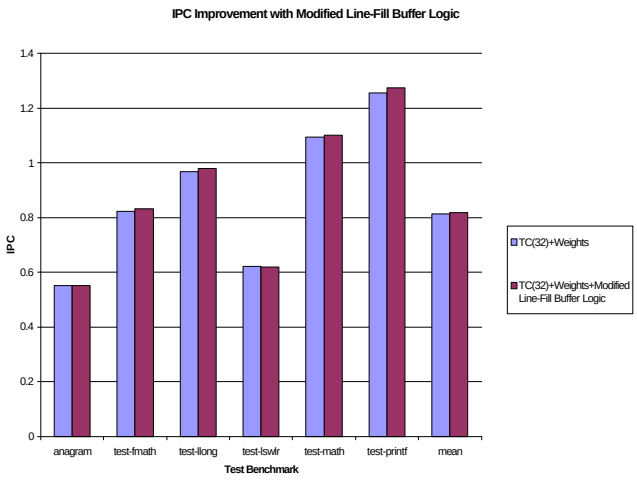


Figure 7: Trace Cache using weights for replacement logic and a modified line-fill buffer logic

trace cache uses multiple branch prediction as compared to the baseline which uses a single branch prediction. The multiple branch prediction leads to higher mispredictions, lowering performance.

Using the SimpleScalar test benchmarks and the 32 entry trace cache, we now turn to evaluating the effectiveness of the trace cache with our proposed weights. We actually compare our weights to another weighting which uses the distance between instructions as a possible weight. Figure 6 shows the results of our simulations.

From the figure, our proposed weights either matches the performance of the baseline or shows an improvement over the baseline. However, the distance weighting actually performs worse than the baseline [TC(32)]. This is the result of the fact that some branches may select an instruction that is large distance from itself. In this case, many of the lines in the trace cache will be eventually removed as it begins to fill instructions from the farther branch. This combined with the larger mispredictions of a multiple branch predictor explains the lowering of performance. Therefore, this shows the importance of utilizing an appropriate weighting scheme.

For anagram and test-lswlr, there is little or no improvement. These benchmarks are inherently not able to take advantage of the trace cache. As seen in Figure 5b with the 1024 entry trace cache, there is little improvement that can be made on these benchmarks.

Now that the trace cache performance has improved with a weighted replacement logic, the line-fill buffer logic needs to be addressed. By modifying this logic, we can extract even greater performance improvement. As seen in Figure 7, the modified line-fill buffer logic as described in Section 4, either matches or improves the performance of the trace cache.

Again, for anagram and test-lswlr, there is little or no improvement. Because of the improvement potential shown in Figure 5b, it is expected that these benchmarks will not show significant improvement.

Based on the preliminary results with the test benchmarks, the proposed change in the trace cache will be able to encompass a variety of applications that contain numerous loops and branches. These applications include, but are not limited to, scientific applications.

Overall, our proposed weighting scheme and line-fill buffer logic does improve trace cache performance. However, we were not able to complete a full evaluation using the SPEC2000 benchmarks and their complete data sets. Given the performance on the SimpleScalar test benchmarks, we expect that even greater performance improvement can be achieved with the SPEC2000 benchmarks. The SPEC2000 benchmarks and most applications will execute a greater number of loops and branches and hence can benefit more from the judicious use of the trace cache that this paper proposes. How much improvement these benchmarks show will utlimately determine if the logic needed to implement our proposal is worth the time and cost to develop in hardware.

6 CONCLUSIONS

The idea to utilize the trace cache in a more judicious manner is not without merit. There is no question that from our results, the upper bound of performance improvement for the trace cache is pretty significant. However, determining the best method for allocating trace cache resources is a challenge. We have proposed a possible solution that does show performance improvement on most of the test benchmarks. In order to more fully evaluate the benefits of our proposal, more simulations must be completed on the latest benchmarks.

However, with our proposed solution, the simulations on the test benchmarks do show performance

improvement is marginal. This initially leads to the conclusion that the hardware needed to implement the weightings in the replacement logic and modifying the line-fill buffer logic will not be worth the costs. Because these test benchmarks are not a good representative of real applications, the possible improvements in real applications as represented by SPEC2000 benchmarks may yield very different results. Therefore, our initial hypothesis does not need to be changed until an evaluation can be completed on the SPEC2000 benchmarks.

Our results do show that the amount of improvement that our proposal can produce is highly dependent on the type of applications. We have also showed that the type of weightings used can have a significant affect on the performance. Therefore, more work may also need to be done to evaluate the cause of only marginal improvements and perhaps develop a better weighing algorithm. In general, we have shown that there is a lot of potential for performance improvement resulting from a more judicious use of trace cache resources.

REFERENCES

[1] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. *IEEE*, pp. 24-34, 1996.

[2] D. Friendly, S. Patel, and Y. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. *IEEE*, pp. 24-33, 1997.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin-Madison Computer Sciences Department Technical Report*, pp. 1-21, June 1997.

[4] S. Patel, D. Friendly, and Y. Patt. Critical Issues Regarding the Trace Cache Fetch Mechanism. *University of Michigan Department of Electrical Engineering and Computer Science Technical Report.*

[5] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, pp. 28-35, 2000.

Appendix A

```
/*
 * This method calculates the weight for a given  trace cache line.
 * It considers three factors : block count, discontiguity and future usage of
 * the line
 */
int get_trace_cache_line_weight( index, line_no )
   int index ;
   int line_no ;
{

      int block_count_weight,branch_pred_weight=0, hit_weight ;
      int count = 0 ;
      int temp = 10001 ;
      // all weights normalized over 10  except hit_weight

      // Number of basic blocks in the trace cache line.
      block_count_weight = trace_cache[index][line_no].no_of_BB * 10 / MAX_PREDICTIONS ;

      // Discontiguity factor :
      // => more the number of 1s in branch_prediction array of a trace-cache
      // line, more discontiguous is the line, so it is given more weight
      for(count=0; count<trace_cache[index][line_no].no_of_BB; ++count)
      {
        temp = pow(10,count+1) * trace_cache[index][line_no].branch_pred_val[count]  ;
      }
      switch( temp )
      {
         case 10001 : branch_pred_weight = 0 ; break ;
         case 10011 : branch_pred_weight = 2 ; break ;
         case 10101 : branch_pred_weight = 4 ; break ;
         case 10111 : branch_pred_weight = 6 ; break ;
         case 11001 : branch_pred_weight = 4 ; break ;
         case 11011 : branch_pred_weight = 6 ; break ;
         case 11101 : branch_pred_weight = 8 ; break ;
         case 11111 : branch_pred_weight = 10 ; break ;
      }

      // hit_weight
      hit_weight = 0 ;
      if( (current_sim_cycle - trace_cache[index][line_no].partial_hit_lru[0]) < active_window_size )
      {
         if(trace_cache[index][line_no].partial_hit_count[tc_m] > 3 )
            hit_weight = 10 ;
         else if (trace_cache[index][line_no].partial_hit_count[tc_m] == 1 )
           hit_weight = 2 ;
         else hit_weight = 5 ;
      }

      // return the sum of aboave weight factors
      return block_count_weight + branch_pred_weight + hit_weight ;
}
```