Study of Stride based Value prediction in Trace Processors

Aishwarya Hendre
Graduate Student
ahendre@ncsu.edu

Laxman Sole
Graduate Student
lsole@ncsu,edu

Department of Computer Engineering
North Carolina State University

Abstract

Trace processor is a next generation microarchitecture that addresses both complexity and architectural limitations of current superscalars. It is a microarchitecture organized around traces and is presented as a means for efficiently executing many instructions per cycle. However to exploit its complete potential the architecture requires further enhancements such as value prediction. In this paper, we have implemented a baseline trace processor and studied the effects of added value prediction on it. The Trace processors exploits both control flow and data flow hierarchy by (1) Distributing execution resources based on trace boundaries and (2) applying control and data prediction at the trace level rather than individual branches or instructions.

A set of experiments was performed and their effects on performance have been presented. These experiments include 1) Changing parameters such as trace cache size and associativity 2) Study of trace predictor accuracy on performance by changing Trace predictor's configurations. 3) Effect of number of processing elements on the IPC of the processor 4) effect of added Value prediction on IPC and study of predictable values. 5) Performance analysis of various small optimizations to improve over complete squashing of pipeline in case of a branch mispredict.

I. INTRODUCTION

In order to extract more instruction level parallelism (ILP), we use wider superscalar processors. However, there is extra cost in increasing the width of superscalar processor. With an increase in issue width, the relative size of structures and number of ports increases. Getting higher IPC even with wide superscalars requires a front end that can fill the instruction window so that ILP can be extracted. Wide superscalars have many fundamental architectural limitations. Instruction fetch bandwidth is limited by frequent branches. Multiple branch predictors not just increase the complexity of structures but also affect cycle time.

The purpose of this paper is to put forth our implementation of trace processors explained in [3] as one of the solution to overcome superscalar limitations. Trace Processor uses traces stored in Trace Cache. Traces are nothing but the dynamic sequence of instructions we have observed during the runtime of the program. In a trace processor, there is movement of traces and hence trace processor architecture handles the data and control dependence at the granularity of traces and not at instructions. All the resources like issue queue, physical register file in the processor are distributed and have a hierarchical nature which reduces the demand of multiple ports and big sizes of structures. Also complexity of bypass and select logic is minimized. Each trace is processed by individual Processing Element (PE) which is equivalent to superscalar backend pipeline.

Trace Processor performance can be largely impacted due to inter-dependencies between different traces. The live outs of a prior trace can be live ins of traces that follow. We propose to attack this bottleneck using value prediction. We have implemented a stride based value predictor to understand the predictability of values and the performance boost that can be obtained with accurate value prediction. Our primary contribution is evaluating the performance potential that this microarchitecture offers.

II. IMPLEMENTATION

2.1 Overview of Architecture

The trace processor microarchitecture (shown in figure 1) [3][5] revolves around the functioning of a trace cache. On the higher level the entire processor can be divided in three major chunks. These are the frontend with trace predictor, trace cache, branch predictor and all other necessary fetch mechanisms, second is the global rename and dispatch stage tied tightly to the retirement engine and lastly the backend which consists of the processing engines. Each of these processing elements consist of a local register file (LRF), issue queue, execution lanes and local and global bypass network to pass values among PEs. In this processor information flows between stages at the granularity of a trace. The trace predictor predicts a TraceId, this TraceId is the information indicating the start PC of the trace and number and outcome of any branches that follow. If the predicted TraceId is valid and consistent with the control flow of the instruction stream, it is looked up in the cache. In case of a trace cache hit the trace is directly picked up from the cache and send down the pipeline for global renaming and is then directly dispatched to an available processing element. If the trace did not hit in the cache it is constructed on the slow path using the other fetch machinery. The constructed trace is pre-renamed and then sent for global renaming and dispatch as well as written into the trace cache. Pre-renaming involves giving local destination registers (registers that got killed within the trace) new local register tags from local register file and renaming any sources that depended on them.

As mentioned above due to inter-trace dependencies among register values consumers of a producer register in other PE's stall in the issue queue and must wait until their value is ready. To overcome this, the value prediction engine is added at dispatch and global rename stage. However for highly accurate value prediction the predictor must be trained and updated well. This along with its results in explained in section III.

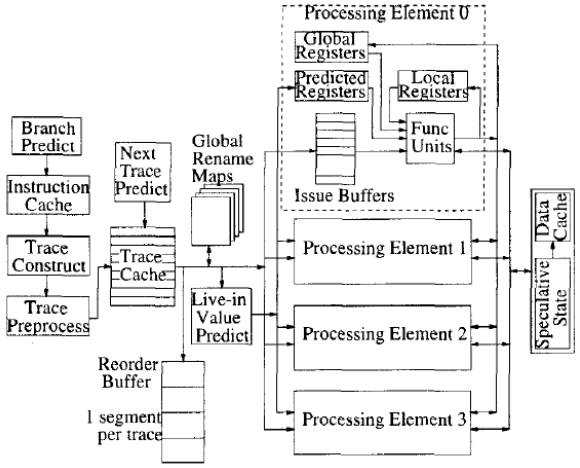

Figure 1: Trace Processor Microarchitecture

2.2 Frontend Implementation

Frontend of a trace processor (figure 2) is very critical and with a fairly good trace predictor accuracy and trace cache hit rate a high boost in performance was observed. An accurate prediction followed by a cache hit can save multiple cycles of instruction fetch, decode and pre-rename. The subtleties in its implementation, complexities and our insights into its employment and further explained.

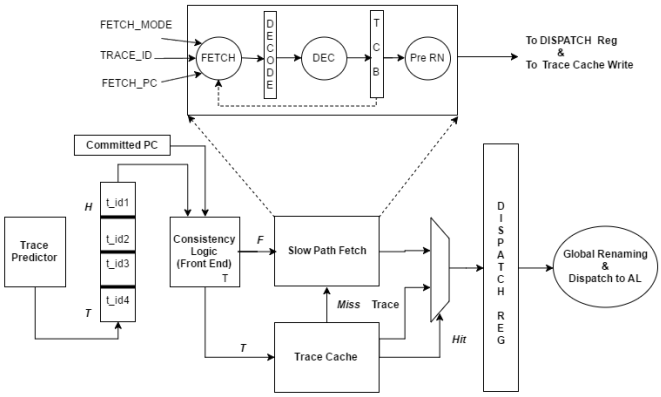

Figure 2: Trace Processor Frontend

2.2.1 Trace predictor

Like previously mentioned trace predictor accuracy is one of the factors affecting trace processor performance. We implemented Quinn et al's DOLC Path-Based Next Trace Prediction [2].

We have not added the confidence counter mechanism suggested in the paper. We updated history register speculatively if TraceId passes initial consistency checks. As we were updating history register speculatively, we maintained the structure size large enough to be able to roll back to committed state if any of the speculative update resulted in misprediction. Prediction table which contains the actual TraceIds, get updated twice. First when we finish constructing trace via slow path and second during retire stage. Previous one is speculative update as this same trace can get mispredicted after execution. If it mispredicted, we rollback history register to its committed state and start repairing trace on slow path. Now new updated TraceId get written into the same location where it would have been in first place if it is correctly predicted. In this scenario, we also update our history register with new TraceId.

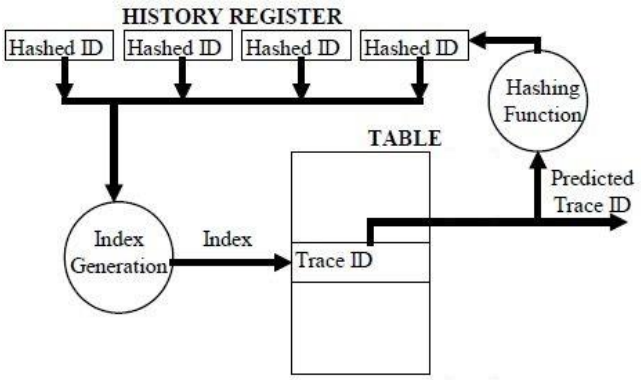

Figure 3: Trace Predictor

In order to understand how well the predictor was working we needing to not just measure its accuracy but also measure it the right way. The TraceId consists of two main components start PC and branch outcomes. The prediction made by the predictor is right only when both of these match the actual TraceId. Also any TraceId misprediction made in the background of another misprediction should not be considered in the statistics. Hence to measure the mispredictions accurately we carried a flag with the trace until retirement which indicated if the trace was constructed on the slow path due to a TraceId mispredict or a TraceId repair. If the trace retired successfully and had the above flag set, the trace misprediction count was increased. Figure 4 shows the results for trace prediction for various DOLC configurations As seen in the graph, predictor accuracy varies with the Depth parameter. We found our results contradictory with what paper suggest. Prediction accuracy was decreasing with increase in depth. We found configuration 1-0-7-9-16 which uses 7 and 9 bits from LAST and CURRENT fields respectively works well for all benchmarks. Number 16 here in configuration indicates index size in bits. For rest of the paper, we assumed this configuration of Trace Predictor unless otherwise stated Figure 5 shows that with an increase in predictor performance the IPC improves. Trace Cache used here is fully associative cache with 2048 size and there are total 16 processing elements (PE) in the backend.

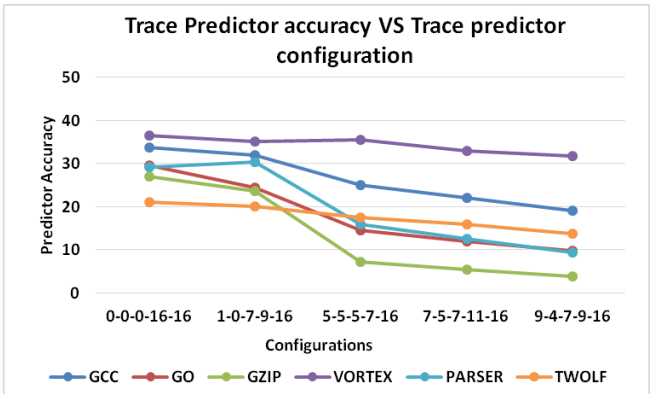

Figure 4: Trace Predictor: Accuracy Vs Configurations

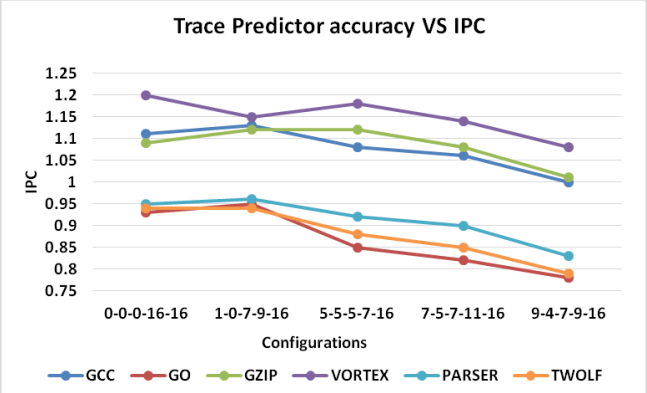

Figure 5: Trace Predictor: IPC Vs Configurations

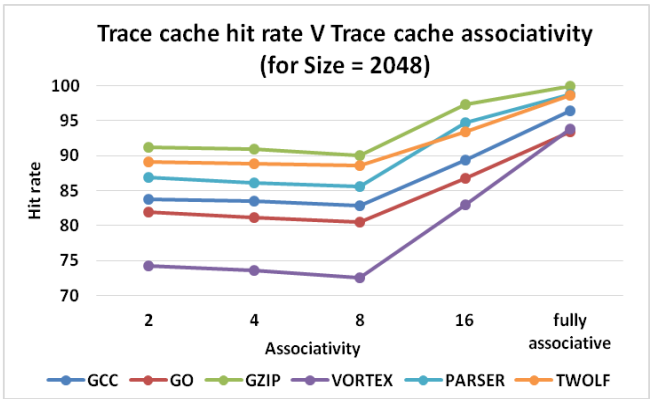

Figure 7: Effect of Associativity on Trace Cache Hit rate

2.2.2 Trace cache

Trace cache is equally important in order to get the maximum performance benefits of Trace Processor.

As mentioned earlier, each trace has its unique TraceId which is a combination of start PC and branch outcomes i.e. Branch Flags. To search in Trace Cache, we used this unique TraceId as TAG field in Trace Cache. To index the Trace cache, we used lower bits of TraceId. Because of this, traces of same start pc with different branch flags got different location in cache and in that sense we achieved path associativity. So at a time there can be multiple traces which have same start pc and different paths.

Trace processor performance is a function of product of trace cache hit rate and trace predictor accuracy. A good predictor would be beneficial only if it got a large number of trace cache hit, which was observed in our implementation.

The critical part here was determining the configuration of the trace cache. Figure 6 and 7 show the effect of increasing cache size with and associativity on cache hit rate. Cache hit rate was measured as the number of times that a TraceId passed by the consistency logic hit in the trace cache.

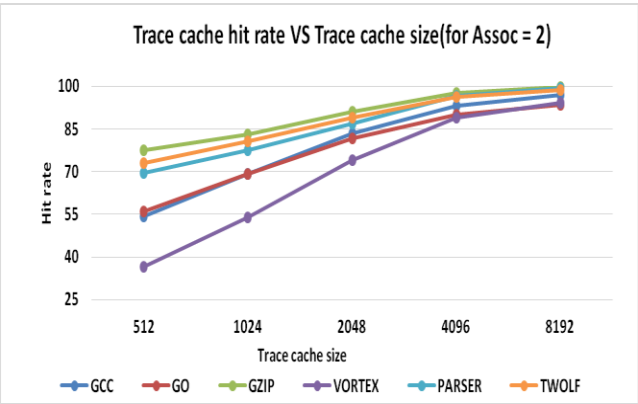

Figure 6: Effect of Trace Cache Size on Hit Rate

We can see that Trace Cache hit rate increase with Trace Cache size for fixed associativity. Also it increases with increase in associativity for fixed size. Here onwards we are assuming fully associative cache of size 2048 for further experiments unless otherwise specified.

2.2.3 Trace selection and slow path fetch policies

In order for the trace processor to give better performance over a wide superscalar, we need to ensure that the traces formed help the processor extract ILP as well as utilize all the hardware resources to the fullest. Having a processor with a trace length of just 4 instructions and then executing the trace on a PE with 10 execution lanes is a waste of resources. Therefore we analyzed various trace configurations. The average trace length we observed for a maximum trace size of 16 was around 13.2 for almost all benchmarks. Another thing to note is that in our simulator implementation, we have terminated the trace in the following scenarios: Maximum trace length was reached, maximum basic blocks was reached (i.e. maximum branches that can be contained within a trace), on an indirect branch, in case when an exception was encountered.

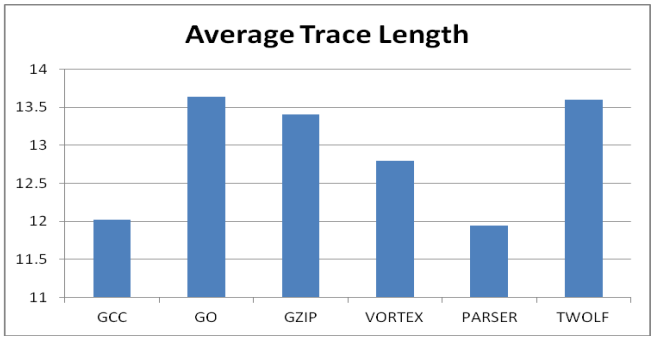

Figure 8: Average Trace length

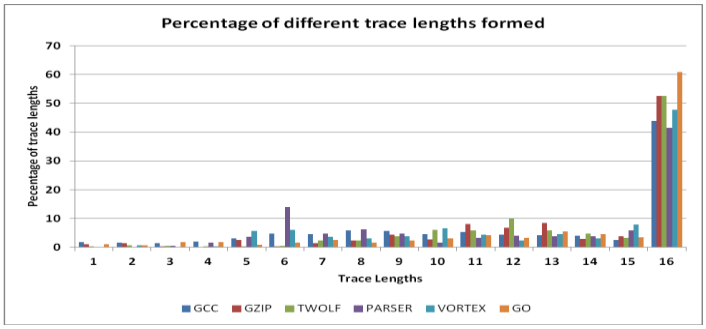

Figure 9: Different Trace Lengths

In figure 2 it can be seen that the slow path is triggered in many cases. The circumstances under which the trace is constructed on the slow path is mainly decided by the consistency logic, cache hit/miss flag and the fetch mode. The fetch mode in our implementation had 4 values, FAST_PATH(0) indicating the slow path was inactive, TID_MISPREDICT(1) indicating there was a TraceId misprediction and this flag was set by consistency logic if the TraceId failed the consistency tests, value CACHE_MISS(2) was set when the TraceId passed the consistency test however there was a cache miss and now the trace needed to be constructed on the slow path, value TRACE_REPAIR(3) was set by the back end indicating a branch mispredict after it was executed, it mainly told the frontend that an old trace was coming to it and now needs to be repaired.
In TID_MISPREDICT mode, all forward branches were predicted not taken and backward branches were predicted taken.
In CACHE_MISS mode, the branch outcomes were taken from the branch flags in the TraceId.
In TRACE_REPAIR mode, branches up to the mispredicted branch took their outcomes from the branch flags, mispredicted branch takes opposite of branch flag outcome because it was mispredicted earlier and branches after that again followed forward not taken backward taken policy.

2.2.3 Pre-renaming and Trace construction buffer

The fetch and decode stages on the slow path fetch one basic block at a time, decode it and store it in the trace construction buffer. This structure is just used during the trace construction phase. Once the trace is fully constructed it is pre-renamed all at once in one cycle and passed to global renaming stage.
The trace processor has two sets of register file, global register file (GRF) and local register file (LRF). The global register file is used to globally rename live outs of a trace and local register file are used to rename locals in a trace. Since this has a lot of dependence checking logic, we have implemented this as a different stage in the pipeline. The trace is written simultaneously into the trace cache and global renaming stage pipeline register after pre-renaming. In case of FAST_PATH i.e. if trace predictor provides consistent TraceId and if this trace is available in the Trace Cache, the trace is directly send to global renaming stage pipeline register.

2.3 Backend Implementation

The backend of a trace processor (figure 11) is equivalent to multiple superscalar back ends only with the advantage that now in a trace processor these structures are much smaller in size and multiple of these processes instructions in parallel. Each trace after global renaming is dispatched into the Active list. The size of the active list is the same as number of processing elements in the pipeline. The instructions are then issued into the corresponding PE s in round robin fashion i.e. trace in active list at position 3 will go to PE 3 and so on. Each PE within it has a local register file, issue queue and outstanding trace buffer (OTB). The register read, execute and writeback stages are also replicated within each PE. The number of PE's basically tells us the number of active traces within the processor that are being executed.
From figure 10 we can see the effect of multiple PEs on IPC.

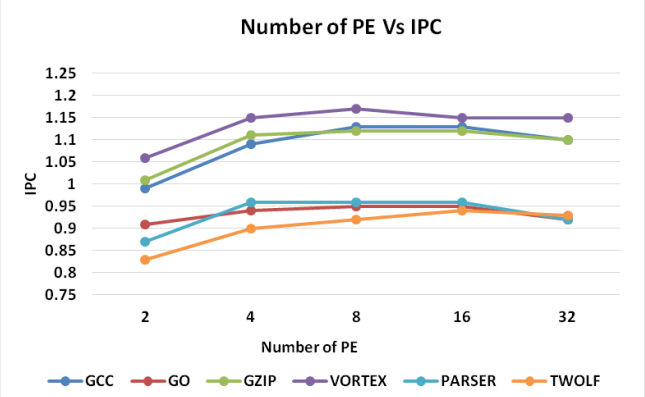

Figure 10: Effect of number of PE on IPC

The size of the issue queue is sufficient to hold the entire trace at once. The outstanding trace buffer is more like a instruction level active list which hold all control information and meta data about the trace such as which instruction mispredicted or which instruction was an exception. The LRF holds all the local values. This greatly simplifies the bypass logic and reduces wire delay for values that are bypassed within a trace. The issue queues are also provisioned with the ability to re-execute the instructions.
The instructions after being issued for execution are not removed from the issue queue; they remain valid in the queue except that now they have an issued bit set to 1. And they will re-execute again only if this bit gets set to 0. When all the instructions in the PE execute and retire all these structures are cleared and the new trace is put into the queue. This is a specialty of a trace processor that gives added advantage in case of value prediction. If the predicted value mispredicts then the trace does not have to be re fetched or dispatched since all the instructions are already there in the issue queue, all that needs to be done is to set their issued bit to 0 again. In a superscalar the instruction is removed from the issue queue and additional structures such as replay buffers are needed to recover from a value misprediction.

2.3.1 Global renaming and Dispatch stage

The complexity of the global renaming stage and dependence checking logic is removed. The trace was pre-renamed and all the registers that need to be renamed to a global register are already marked and are now simply renamed. Since all the information is at trace level and the instructions also retires at trace level the active list size is much smaller as compared to the conventional active list in a superscalar.
The trace in the global dispatch stage is allotted an entry in the active list and in a way gets tied to the PE. Important information to initiate recovery is also check pointed in this stage; it is further explained in the next stage.

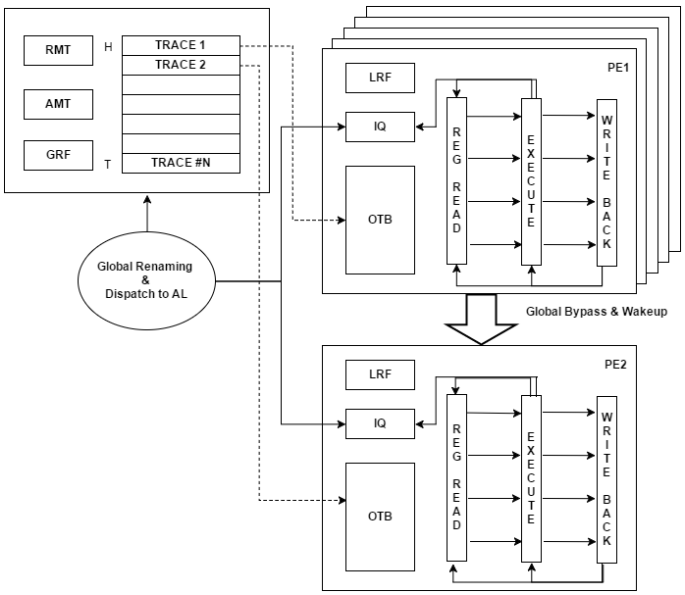

Figure 11: Trace Processor Backend

2.3.2 Check pointing and Recovery

In a trace processor, a lot of instructions are brought into the pipeline speculatively and provision should be made to ensure that unless the instruction is executed and we are completely sure that it is the right instruction, it should not affect the architectural state of the processor. In our implementation whenever a branch mispredicts, we initiate immediate recovery and squash all traces in the pipeline following the mispredicted trace. Every cycle in the retire stage we poll all traces in the OTB's starting from the head to the tail of the Active list. If in any PE, any instruction's mispredict bit is set, the entire pipeline after that OTB/PE is squashed. To facilitate this squashing we have check pointed architectural state at the start of each trace. The number of check points is equal to the number of PE's in the processor. In dispatch stage every time that we dispatch a new trace into the PE we checkpoint the Rename map table at that point. We also checkpoint the free list tail, load queue index and store queue index. When a branch in a PE mispredicts we squash everything in all PE's after that and restore the architectural state at that point from that checkpoint. The checkpoint is freed only when the PE is freed.

In our implementation after running for a few instructions, we realized that all the times that the trace ended in a branch and that branch mispredicted we squashed the entire pipeline along with that trace, however in reality that trace was right and we should not have squashed it, all that we needed to do was update the branch flags and TraceId of that trace, retire it and make sure the next trace was consistent with the control flow. We added this optimization and Figure 12 shows the number of times the last branch in trace mispredicted and we saved cycles by not squashing it. These numbers also include traces fetched speculatively on the wrong path. The graph just gives an insight into the number of traces that would have been squashed inspite of being rightly executed.

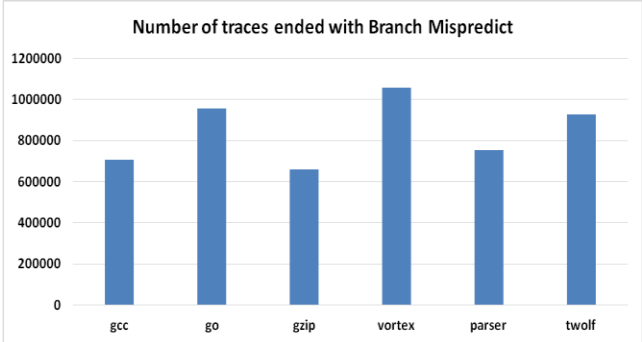

Figure 12: No of traces with last branch misprediction

2.4 Consistency checks

The trace predictor predicts a next TraceIds, however these TraceIds need to be verified to ensure the correct control flow of a program at various stages in the pipeline.

We added following consistency checks at various stages:

- In the frontend immediately after the TraceId was predicted, if the pipeline is empty then the start PC form the TraceId must be same as the committed PC of the processor at that time (committed PC is nothing but the PC of the last committed instruction in the pipeline plus 8).
- If the pipeline is not empty then we must ensure that the predicted TraceId has a start PC which is either the fall through or fall target of the last instruction of the previous trace if it ended in a branch. If the previous trace did not end in a branch it must simply be last pc plus 8.
- The second check is however still speculative does not guarantee consistency. Hence for a 100 percent guarantee, there is a consistency check in the retire stage. Each time a trace retires, its last instructions next PC is checked against the next trace's start PC. If the PC's match, consistency is passed else the pipeline is squashed and a new trace is fetched with the new start PC and the fetch mode is set to TID_MISPREDICT.

III. VALUE PREDICTION

Even if trace predictor predicts consistent traces back to back and there are multiple PEs to facilitate all of these traces, there is data dependency barrier which don't allow execution of most of these traces in parallel. Most of the traces don't execute in parallel and stall execution because live in values on which they depend are not ready. This forces serialized execution.

To break this data dependency barrier, we implemented value prediction. Previous studies shows that most of the time values are repetitive or they repeat in some sequence. So we have implemented stride based value prediction for values of only live-outs which are live-ins for next traces.

3.1 Stride based value predictor

We have implemented simple stride based value predictor as mentioned in [6]. We maintain a Value History Table (VHT) which is indexed by lower few bits of PC. For all instructions which have valid destination register and if that resister is live out we check the corresponding PCs confidence in VHT. If instruction's PC matches with TAG in VHT and if confidence is high, we say that value is predicted and we take the corresponding value from value field of the VHT.

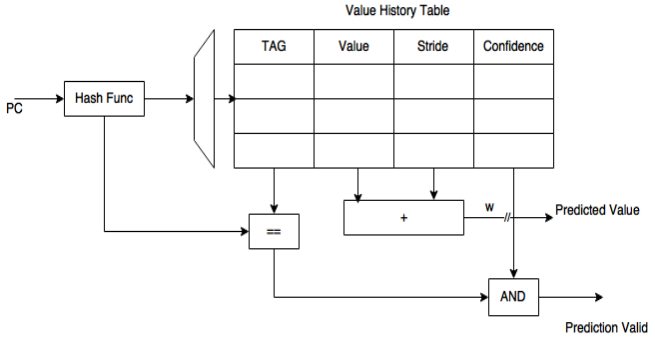

Figure 13: Stride based value predictor

The VHT is updated at two locations first time at dispatch and second time in the execute stage. During dispatch if prediction is valid which means TAG matches with PC and confidence is high then we update the value field of VHT with 'Value+Stride'. The updation helps us predict the next value within the trace. Second time we update VHT after execute stage. After execution we check if that instruction has valid live-out destination and if it is not already predicted during dispatch, if it matches the criteria then we access VHT with PC of this instruction and check for TAG matching. If it matches and new stride is same as previous stride then, we update the value with current value and increment the confidence counter. If there is mismatch in any one of the case then we set stride value and confidence counter as zero and new TAG will be PC of this instruction. This conservative approach makes us predict fewer values but with high prediction rate. Figure15 shows the overall coverage for all benchmarks. It is maximum for Parser and minimum for Twolf.

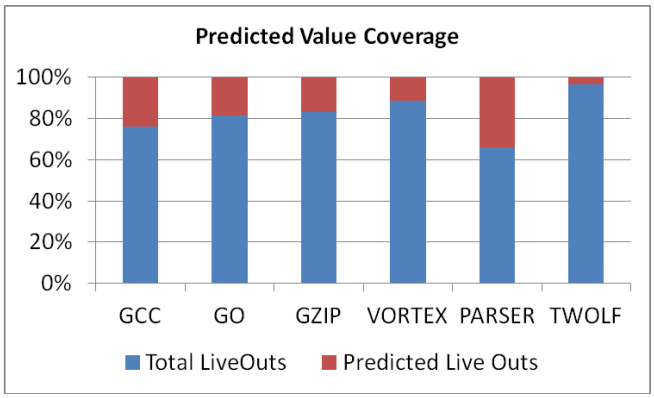

Figure 14: Percentage of Value predicted

3.2 Value predictor: Results

Accuracy Measurement: To measure the accuracy, we added a 'predicted' flag to identify if live-out of that particular instruction is predicted one or not. During retire stage, we checked if instruction has 'predicted' flag set. If it is high then we increment 'total_prected_value' count. If 'predicted_value' field is same as 'computed_value', then we increment 'correct_value' counter. Ratio of correct value to total_predicted_value gives the prediction accuracy. Figure 16 shows accuracy for all benchmarks.

One of the interesting fact about Stride based value predictor is it incorporates last value predictor also. As most of the predicted values are same i.e. there is large potion when stride was zero.

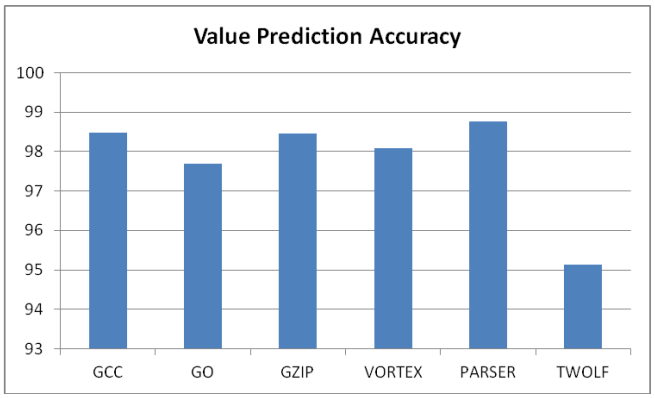

Figure 15: Value Prediction Accuracy

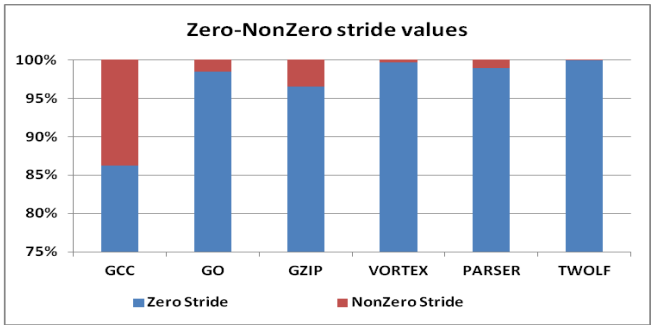

Figure 16: Zero and non-Zero Strides

3.2 Oracle Value prediction

To study the impact of value prediction on the overall performance, we added Oracle Value Prediction Mode. In this mode, we check if our predicted value is matching with the functional simulator's value for the same instruction during dispatch. If the value matches then we are that destination register ready in the PRF and insert the predicted value in the PRF.Any instructions depend on this register, will see it as ready and get issued along with the instruction which is producing it.

Although this performance gain is not realistic since we are not considering any value misprediction penalty it gives us an insight into how many values can be predicted with high accuracy and can help improve performance.

IV. RESULTS

For Experiments we have used following default parameters wherever they are no explicitly stated:

Number Of PE	16
Trace Predictor Parameters	1-0-7-9-16(Last is index size)
Branch Prediction policy	Forward not taken, backward taken
Trace Cache	Size-2048(Fully Associative)
Last Branch misprediction (For trace ending with branch)	Enabled
Immediate squashing of all PEs after mispredicted PE.	Enabled
LRF Size	16
Issue Queue Size	16
Trace Size, max Basic Blocks	16,3
PRF	128
Oracle Value Prediction	Disabled
LQ_SIZE,SQ_SIZE	64,64
PERFECT_FETCH	False
ORACLE_DISAMBIG	True

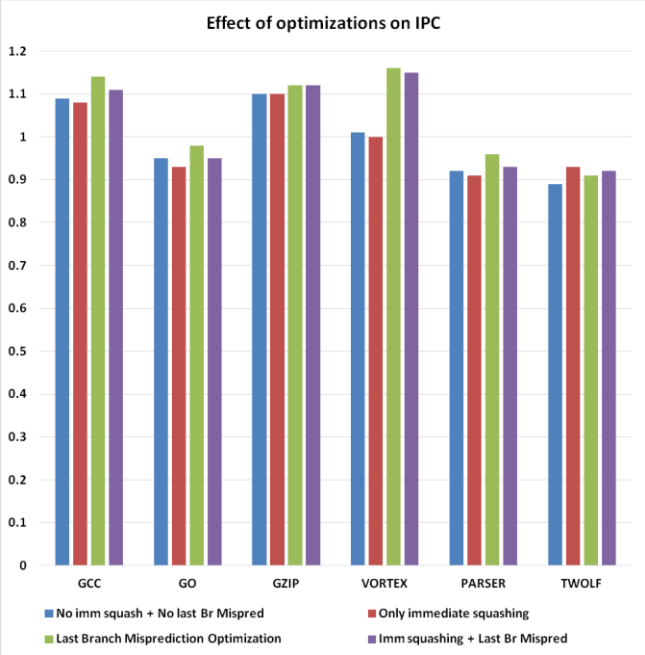

Figure 17: IPC with Different Optimization Strategies

Above graph shows IPC for various optimizations we added. It can be seen that if we keep only last branch misprediction and squash trace if it is at the head of trace level AL, we are getting good improvement. However immediate squashing gives performance benefit only if there are more speculative traces in the pipeline (more number of PE's).

Figure 18 shows the IPC for baseline trace processor and trace processor with oracle value prediction mode. We can see the performance improvement for gcc, vortex, parser and twolf benchmarks. The performance gain is not high since we have implemented complete squash in case of branch mispredicts, due to this the predicted values have less effect on performance.

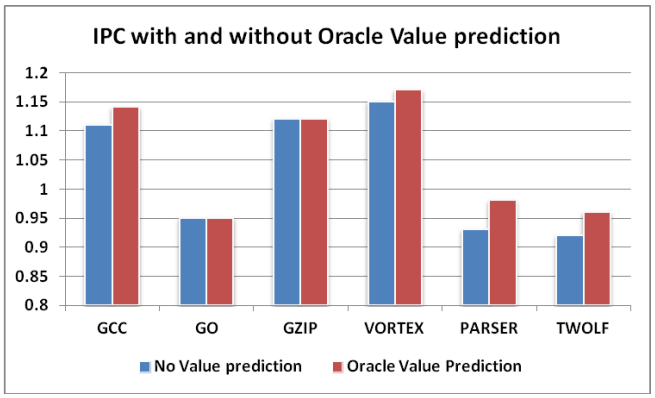

Figure 18: IPC Comparison between baseline and Oracle value prediction

V. CONCLUSION

Trace processors exploit the characteristics of trace to efficiently issue many instructions per cycle. They exploit data locality and instruction level parallelism. The local register file and local bypass makes structures smaller and help reduce wire delays of bypassed values. Although in our case we could not see much improvement due to added value prediction however like mentioned we have followed a conservative approach and have a squash mechanism which makes our results obvious, but we can see through that selective re-execution will greatly impact performance. Through our experiments and analysis we also understood the effect of various factors on the performance on the processors. Number of Live outs, Live In's average trace length, small optimizations such as immediate squash have an effect on performance.

VI. FUTURE WORK

The trace processor's potential to exploit ILP and produce high performance is great, our implementation is very raw and can be considered as a baseline model. To improve its performance more we understand that a lot more work needs to be done on it. Value prediction needs to be further tuned and optimized to predict more values with non-zero stride. The scope of this optimization can also be increased by adding Context based value prediction will further enhance performance as a large number of live out values follow other patterns besides stride. In our current implementation value prediction is used in oracle mode, to make this optimization more realistic we need to properly incorporate selective re-execution and avoid squashing in case of branch mispredictions. In our baseline we have not experimented with Physical register file size, local register file size, issue queue size, load store unit size, maximum trace length and maximum basic blocks. Their effect on the performance of the processor are unexplored and we wish to experiment with them to understand what parameters could be a bottleneck.

REFERENCES

[1] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture (MICRO-29), pp. 24-34, December 1996J

[2] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-Based Next Trace Prediction. Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture (MICRO-30), pp. 14-23, December 1997.

[3] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace Processors. Proceedings of the30th IEEE/ACM International Symposium on Microarchitecture (MICRO-30), pp. 138-148, December 1997.

[4] Eric Rotenberg, Steve Bennett, and James E. Smith. A Trace Cache Microarchitecture and Evaluation. IEEE Transactions on Computers, Special Issue on Cache Memory, 48(2):111-120, February 1999."

[5] Eric Rotenberg. Trace Processors: Exploiting Hierarchy and Speculation. Ph.D. Thesis, University of Wisconsin - Madison, August 1999.

[6] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," Proceedings of 30th International Symposium on Microarchitecture (MICRO-30), December 1997.