Superscalar Processor with Checkpoint Processing and Recovery with a Trace Cache Simulator

Sonali Varanasi, Yusong Li, Tse-Han Pan, Shrirang Deshpande, Zhiyu Chen

Abstract

Processor requires a combination of large instruction windows and high clock frequency to achieve high performance. As instruction fetch bandwidth requirements increase, it is necessary to fetch multiple basic blocks per cycle. However, conventional caches degrade this effort because long instruction sequences are not always in contiguous cache location. On the other hand, exposing large amount of instruction level parallelism requiring large hardware structures to buffer and process such instruction window size will significantly degrade the cycle time.

This paper summarizes our efforts to implement a superscalar processor with a trace cache and, with checkpoint processing and recovery (CPR). Conventional cache with a trace cache traces the instructions in the dynamic instruction stream, so instructions that are otherwise noncontiguous appear contiguous. CPR is to implement a large instruction window processor without requiring large structures that permitting a high clock frequency.

Key Words – trace cache, checkpoint processing and recovery, superscalar processor

1. Introduction

Improvements in microprocessor performance come about in two ways – advances in semiconductor technology and advances in processor microarchitecture. Recently the improvements of the clock frequency reaches its limit. And the micro-architectural challenge is to implement parallelism from sequential programs. Program is not written in an obvious parallel style, but nevertheless it will contain significant amounts of inherent parallelism. By exploiting instruction level parallelism, the processor may "look ahead" into the program for independent instructions.

The organizations of high performance superscalar processor are normally divided into an instruction fetch mechanism and an instruction execution mechanism which are separated by an instruction buffer (Figure 1). Conceptually, the instruction fetch mechanism acts as a "producer" which fetches, decodes and places instructions into buffer. And the instruction execution mechanism is the "consumer" which removes and executes the instructions from buffer. Control dependences provide a feedback between these two mechanisms.

However, processors with this organizations have a drawback. The fetch unit is to feed the dynamic instruction stream to the decoder. But the instructions are placed in the cache in the compiled order. Storing programs in static form prefers fetching code which does not branch or code with large blocks. In this case, we implement a special instruction cache called *Trace Cache* which can capture dynamic instruction sequences.

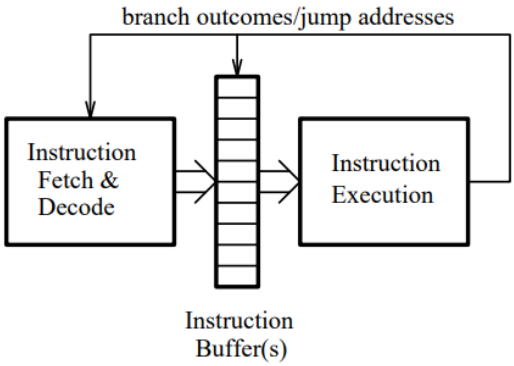

Figure 1 Decoupled fetch/execute engine

A trace is a sequence of at most n instruction and at most m basic blocks starting at any point in the dynamic instruction stream. It is fully specified by the starting address and the predictions of up to m-

1 branches which describe the instruction paths. At the first time when a trace cache is encountered, it is allocated a line in the trace cache. If the same trace path is encountered again with the same starting address and branch prediction outcomes, the instruction stream is available in the trace cache and can be fed directly to the decoder. Otherwise, fetching instruction streams normally from the instruction cache.

On the other hand, exposing maximum ILP requires the processor to concurrently operate upon a large amount of instructions, also known as the instruction window. Hardware structures to store these instructions must be significantly large. However, high clock frequency requires frequently accessed structures to be small and fast. Accessing large hardware structures will significantly degrade the cycle time. In this case, *the Checkpoint Processing and Recovery (CPR)* is introduced. It is an efficient microarchitecture capable of sustaining large instruction windows without requiring large critical structures.

CPR decouples key mechanism from the ROB and provides a scalable alternative to current ROB-based processor. The key idea of CPR involves the notion that sometimes reconstructing architectural state is more efficient than explicitly storing state, as long as sufficient information is available for reconstruction. Thus, unlike ROB-based approaches that record state every instruction, CPR records state only at carefully selected points of an execution.

The discussion in the paper begins with the implementation of Trace Cache in section II followed by the implementation of Checkpoint Processing and Recovery in section III. In section IV, the experimental results will be discussed.

2. Implementation of Trace Cache

The job of the fetch unit is to fetch and provide dynamic instruction streams to the decoder. But the problem is these instructions are placed in the cache in their compilation order. This is a static order. This type of cache filling in beneficial in case there are no or very few branches in the workload. When branches are encountered, the fetch unit many a times need to fetch instructions which are not in the static compilation order of instructions. And this may limit the fetch unit's performance. To address this problem trace cache was introduced. Trace cache places the instructions in the cache in the dynamic order in which they have been used in the past executions.

The trace cache approach relies on dynamic sequences of code being reused. This may be the case for two reasons:

- Temporal locality: like the primary instruction cache, trace cache can count on instructions that have been used recently to be used again.
- Branch behavior: most branches tend to be biased. That is one of the reasons for the higher accuracy of the branch predictors. This means that a 'trace' of instructions having branches in them will most likely be used again.

We have implemented a trace cache to study the improvement in the performance of 721 simulator.

Trace cache implementation can be divided in three stages. First, the generation of the trace. Second, building up the trace cache using the traces. And Third the hit logic or fetching from trace cache.

2.1. Generation of traces

Trace cache in our design resides in the hardware along with the instruction cache. There are multiple approaches to generate traces that are stored in the cache. We used snooping. We basically snoop out the instructions in their dynamic order and form dynamic blocks out of those instructions. A dynamic block can be characterized as follows:

- A dynamic block has at most one branch in it.
- The maximum length of a dynamic block is set to the fetch width.
- A dynamic block ends when a branch instruction is encountered, or maximum length of the block is reached.

We in this project focused on fetching 16 instruction per cycle. And as there is typically a branch instruction after every 5 instructions, we

store 3 of these dynamic blocks in one trace cache block. So, each trace cache block has 3 dynamic blocks and contains up to 16 instructions. We also need to be able to predict multiple branches in a cycle for enabling this implementation. These dynamic blocks are then stored in the trace buffer.

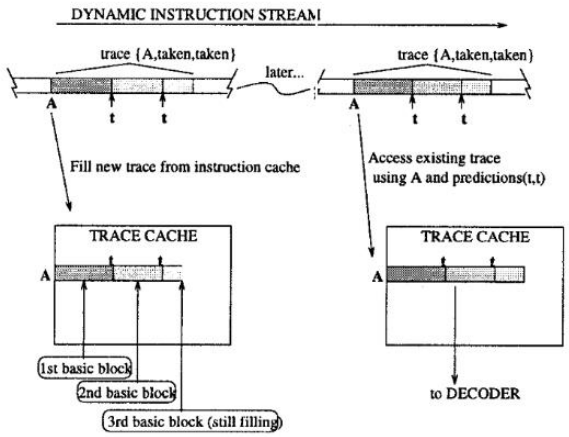

Figure 2 High level view of the trace cache

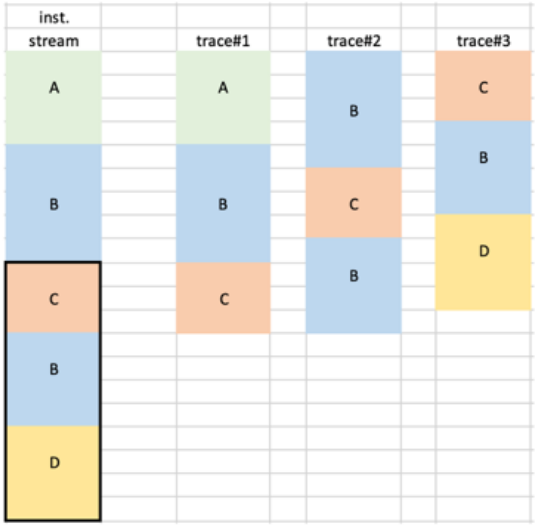

Figure 3 Trace Storing

2.2. Building up the trace cache

For our implementation of the trace cache, we've built a 4-way 1024 sets cache. The reason for choosing the large size and multiple ways lies in the way we store the traces. To get the most from the trace cache, we store a trace starting at each dynamic block. By doing this, we take care of the branches that jump to an instruction which is not at the start of a trace but is first instruction of a dynamic block encountered before. We store the traces in the trace cache as shown in the figure 3 above. But some jump instructions may jump to an instruction which is somewhere in the middle of a dynamic block. Traditional trace cache implementations consider this a miss case and build up a new trace for future use. We on the other hand, to improve performance, have implemented a shifting logic which can be used to fetch instructions from any instruction onwards of the first dynamic block. To facilitate this, we insert some NOPs at the end.

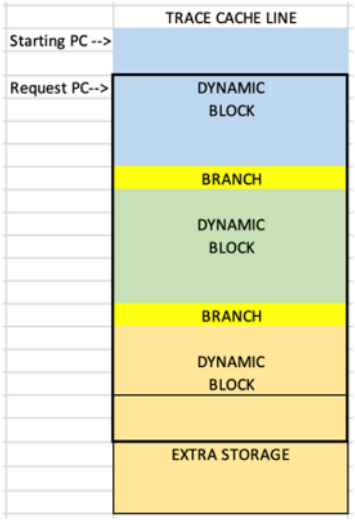

Figure 4 Shifting logic for fetching instructions starting from the middle of a dynamic block

By using this, we fetch less than 16 fruitful instructions for that cycle, but we save potential replacement of a useful trace. Also, we in this case get a hit in the cache, improving the performance. Also, in addition to the dynamic blocks, the branch predictions for all the 3 branches are stored in the trace cache. As we know, every dynamic block end with a branch or after getting fetch width number of instructions. For the implementation purposes, we consider the last instruction of each dynamic block, a branch. If we terminate a dynamic block with an instruction which is not a branch, we treat is as a branch with default prediction value.

2.3. Hit logic and selection of trace

Trace cache resides side by side with the instruction cache. Hence, we need a selection logic to select the cache to fetch instruction stream from. In this project, we always select trace cache over instruction cache if we have a hit in the trace cache. To determine this hit or miss, the pc of the first instruction of the instruction bundle is used along with the branch predictions of the branches in the bundle. If we get a match for the tag computed from the instruction's pc, we check for the branch predictions. If those match to the one's we've stored in the trace cache, we consider it as a hit. Another reason to have 4-way cache is if the branch predictions do not match, then that trace is useless even if the tag matches. This new trace then could replace the old trace starting from the same tag, polluting the cache. By having 4-way cache, we allow 4 different variations of same tag to be in the cache.

2.4. Experimental method for trace cache implementation

To test the functionality of the simulator with trace cache, various benchmarks were used. The benchmarks include astar, bzip2_dryer, mcf, hmmer and perl, etc. Each test were performed by changing the branch predictor arguments to the simulator. The baseline 721 simulator and simulator with trace cache were run with the same parameters and the results obtained are described in the following section.

3. Implementation of Checkpoint Processing and Recovery

3.1. Eliminating the use of active list

Checkpoint processing and recovery (CPR) is a method that is used to eliminate the bottle neck of using an active list. The active list is a circular FIFO that stores information of the instruction present in the pipeline in order. If an instruction completes its execution it still remains in the active list until it reaches the head of the active list. Using an active list in the superscalar implementation limits the number of instructions that can be present in a pipeline. By removing the active list this limitation is eliminated but by doing this we loosed the property of in order retire of the superscalar processor instructions. For cases in which an instruction gets miss predicted it becomes difficult to find the last valid state of all the instruction and the registers used since the instruction retire out of order. Additional structure need to be added to keep track of the correct state of the registers which will be introduced in the next section. The other advantage of removing the active list is that the physical register can be recovered once the instructions using a particular register has finished its operation. It is not necessary to wait till the instruction reached the head of the active list.

3.2. Structures used in the implantation of CPR

The project eliminates the use of the active list but uses checkpoints instead. The checkpoint structure is similar to the checkpoints present in the 721sim superscalar architecture. It consists of a shadow map table (SMT) which keeps track of the renaming map table (RMT) after every N number of instructions. The structure also consists of a register that keeps track of the number of instructions that are present between the current checkpoint and the next checkpoint. Another instruction counter is present to check the number of instructions still present in the pipeline. The number of instruction between two checkpoints is varied during simulation to get optimal operation. It is also necessary to keep track of which checkpoint is to be added once the number of instructions between two checkpoints reaches its max limit. This is done by the checkpoint fifo head and checkpoint fifo tail values. Once the number of instructions between two checkpoints reach its max limit, the next checkpoint is determined by the checkpoint fifo tail value. Once the number of instruction between the checkpoints become zero, the checkpoint fifo head moves to the next checkpoint. If all the checkpoints are utilized, the instructions are added to the last checkpoint beyond its max limit until the next checkpoint is available. The PC register store the program counter of the instruction at which the checkpoint is created. The branch miss predicted bit indicates that a branch has been miss predicted between the current checkpoint and the next checkpoint and the miss predicted instruction number store the PC of the miss predicted instruction.

The Physical Register File structure that is used is similar to that used in the 721 sim. It consists of a register that store the value of each physical register, a register that stores it usage count, a register that indicates whether the register is currently mapped or not. One more bit is present to indicate whether it is present on the free list or not.

The free list is also necessary for the implementation of CPR. It consists of a list of register that are free and not used in the pipeline. When a destination register is renamed, a free register is read from the head pointer of the free list and once a registers' usage is complete, it is added to the tail of the free list.

3.3. CPR Design

The CPR architecture was implemented using the baseline architecture of the 721sim superscalar processor. The frontend stages consists of the simulator consists of the fetch stage followed by the decode stage. The instructions are sent from the fetch stage to the decode stage using the pipeline decode registers. From the decode stage, the instructions go to the rename1 stage via the issue queue. This is followed by the rename2 stage and the dispatch stage and then sent to the issue queue. The function of the backend stages is to send the instructions from the issue queue to the scheduler to the execution lanes. Before the instruction is sent to the execution lanes, a read registers operation is performed to obtain all the source registers and after the execution stage, the instructions go to the writeback stage. The writeback stage consists of the retire stage which has the functionality of retiring a checkpoint once all its dependent instructions have completed execution.

When an instruction is sent from the fetch queue to the rename stage, we check whether the current checkpoint had reached its max instruction width. If the max instruction count is not reached, we increment the instruction counter and move to renaming the registers. In case a checkpoint has reached its max instruction limit, we check whether a free checkpoint it available by checking the head and tail pointer of the checkpoint fifo. If a free checkpoint is available, it is used for the current instructions onwards. If a checkpoint is not available, the instruction is added to the current checkpoint itself and the max counter for that checkpoint is incremented. Checkpoints are also added when atomic instructions are encountered.

After we assign a checkpoint, we read the source register and increment the usage counter of the physical registers that are being used. Before assigning a new physical register to the destination register, the previous mapping of the logical register is unmapped and a new register is assigned.

Whenever an instruction finishes its operation, the instruction counter of the nearest checkpoint before the instruction is decremented and the usage counter of the source register is decremented. CPR has the property of aggressively reclaiming register i.e. at every cycle the contents of the physical register file is checked. If the usage counter of a register is zero, it is unmapped and not already present in the free list, the physical register is added to the free list. A particular instruction can complete its operation at either the dispatch stage, the execute stage or the writeback stage. Once the writeback operations are complete, the instruction count of the head of the checkpoint fifo is check. If the count is zero, we go to the retire stage to check the correct operation. The retire stage is repeated for the number of instructions that depend on the current checkpoint. After all the instructions have completed the retire operations, we commit the checkpoint i.e. the checkpoint is freed.

When an atomic instruction enters the retire stage, we squash the pipeline and clear the checkpoints after the current checkpoint. Also, all the register that are unmapped are added to the free list.

3.4. Experimental method for CPT implementation

To check the working of the simulator different benchmarks were tested. The benchmarks include astar_test, bzip2_dryer_test, mcf_test, hmmer_test and perlbench_s_rand_test. Each of the test was executed for perfect branch prediction and no exceptions. The test were also run for different sizes of the issue queue, load store queue and fetch queue size. The number of checkpoint has been set to 20 and the number of instructions between the checkpoints is 32. The superscalar pipeline with CPR and the 721sim superscalar processor were

executed with the same parameters and the results are described in the next section.

4. Result

4.1. Result for trace cache implementation

The benchmarks that were on both the simulators with the predictor's arguments and obtained cycle count, instruction commit count and IPC are tabulated below:

BaseLine 721Sim					
Sr.No	Benchmark Name	BP arguments	Cycle Count	Commit Count	IPC
1	astar_00c3	0,0,1,1	23101882	100000001	4.33
2	astar_00c3_00	0,0,0,0	29594265	100000001	3.38
3	astar_01c1	0,1,1,1	22497348	100000001	4.44
4	astar_01c1_00	0,0,0,0	31191203	100000001	3.21
5	bzip_00c3	0,0,1,1	20663320	100000001	4.84
6	bzip_00c3_00	0,0,0,0	20758495	100000001	4.82
7	bzip_11c0	1,1,1,1	12997117	100000001	7.69
8	bzip_11c0_00	0,0,0,0	62454123	100000001	1.6
9	bzip_11c0_00_data1	0,1,0,0	13014335	100000001	7.68
10	hmmer_00c3	0,0,1,1	44089408	100000001	2.27
11	hmmer_00c3_00	0,0,0,0	57570350	100000001	1.74
12	hmmer_10c2	1,0,1,1	12669490	100000001	7.89
13	hmmer_00c3_00	0,0,0,0	57597095	100000001	1.74
14	mcf_01c1	0,1,1,1	5170690	100000001	1.93
15	mcf_01c1_00	0,0,0,0	15970056	100000001	0.63
16	mcf_01c1_bp1	1,0,0,0	9332320	100000001	1.07
17	mcf_10c28	1,0,1,1	8597081	100000001	1.16
18	mcf_10c28_00	0,0,0,0	15744512	100000001	0.64
19	perl_01c1	0,1,1,1	24919410	100000001	4.01
20	perl_01c1_00	0,0,0,0	33954713	100000001	2.95
21	perl_10c2	1,0,1,1	10471323	100000001	9.55
22	perl_01c2_00	0,0,0,0	33954713	100000001	2.95

Table 1 Results obtained for the baseline simulator

TraceCache 721Sim					
Sr.No	Benchmark Name	BP arguments	Cycle Count	Commit Count	IPC
1	astar_00c3	0,0,1,1	23101882	100000001	4.33
2	astar_00c3_00	0,0,0,0	25328123	100000001	3.95
3	astar_01c1	0,1,1,1	22497348	100000001	4.44
4	astar_01c1_00	0,0,0,0	27638177	100000001	3.62
5	bzip_00c3	0,0,1,1	20663320	100000001	4.84
6	bzip_00c3_00	0,0,0,0	20685220	100000001	4.83
7	bzip_11c0	1,1,1,1	12997117	100000001	7.69
8	bzip_11c0_00	0,0,0,0	62452029	100000001	1.6
9	bzip_11c0_00_data1	0,1,0,0	13013073	100000001	7.68
10	hmmer_00c3	0,0,1,1	44089408	100000001	2.27
11	hmmer_00c3_00	0,0,0,0	46471311	100000001	2.15
12	hmmer_10c2	1,0,1,1	12669490	100000001	7.89
13	hmmer_00c3_00	0,0,0,0	46510475	100000001	2.15
14	mcf_01c1	0,1,1,1	5170690	100000001	1.93
15	mcf_01c1_00	0,0,0,0	14838543	100000001	0.67
16	mcf_01c1_bp1	1,0,0,0	9017190	100000001	1.11
17	mcf_10c28	1,0,1,1	8597081	100000001	1.16
18	mcf_10c28_00	0,0,0,0	14702490	100000001	0.68
19	perl_01c1	0,1,1,1	24919410	100000001	4.01
20	perl_01c1_00	0,0,0,0	27572862	100000001	3.63
21	perl_10c2	1,0,1,1	10471323	100000001	9.55
22	perl_01c2_00	0,0,0,0	27572866	100000001	3.63

Table 2 Results obtained for the Trace cache simulator

We can see the improvement in the IPC in almost all the benchmarks when imperfect instruction cache is used. The IPC comparison of such benchmarks is shown below:

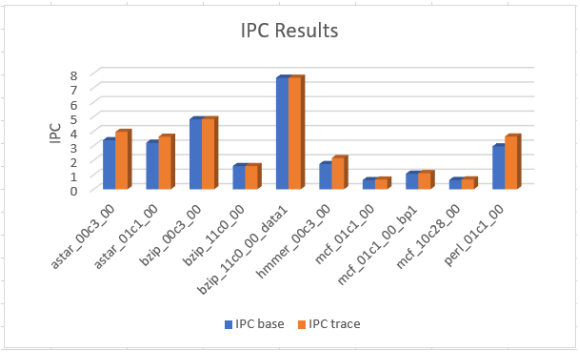

Figure 5 IPC result

From the results that we got, we can see the improvement of IPC for benchmarks with real world configurations. For some of the benchmarks we got up to 0.68 increase in the IPC which comes about 24% improvement. These are the benchmarks with a lot of taken branches. We also got almost no improvement for some of the benchmarks. These benchmarks may have very few taken branches. In either case, the worst-case performance of the trace cache was the baseline 721 sim performance.

Using the trace cache simulator, we were able to improve the performance of the baseline 721sim. We got improvement in the number of execution cycles and consequently the IPC for almost all the benchmarks with real world configuration. When used with perfect instruction cache, both the simulators show the same performance. This improvement in the performance comes with the area cost as the trace cache structure is implemented in the hardware. We can use different sizing configurations of the trace cache to get performance-area tradeoff.

4.2. Results for CPT implementation

The following tables display the IPC of 3 different testbenches namely bzip2_dryer_test, mcf_test, and hmmer_test which are run using the 721sim superscalar processor and the CPR superscalar processor. Each of the architectures are executed for different configurations and the results are tabulated below.

Benchmark	IQ size	LSQ size	AL size	CHK size	IPC
bzip2_dryer	32	64	128	64	7.69
bzip2_dryer	64	128	256	64	15.5
bzip2_dryer	128	256	512	64	16
mcf	32	64	128	64	6.65
mcf	64	128	256	64	9.25
mcf	128	256	512	64	10.19
perlbench_s_rand	32	64	128	64	6.92
perlbench_s_rand	64	128	256	64	9.56
perlbench_s_rand	128	256	512	64	10.78

Table 3 Results of the 721sm

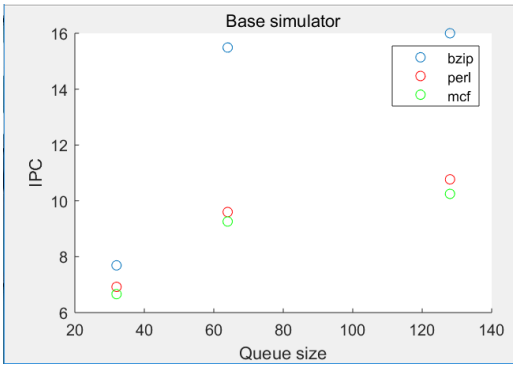

Figure 6 Baseline simulator IPC

Benchmark	IQ size	LSQ size	AL size	CHK size	IPC
bzip2_dryer	32	64	0	20	7.69
bzip2_dryer	64	128	0	20	15.49
bzip2_dryer	128	256	0	20	16
mcf	32	64	0	20	6.66
mcf	64	128	0	20	9.26
mcf	128	256	0	20	10.25
perlbench_s_rand	32	64	0	20	6.92
perlbench_s_rand	64	128	0	20	9.6
perlbench_s_rand	128	256	0	20	10.77

Table 4 721 sim with CPR

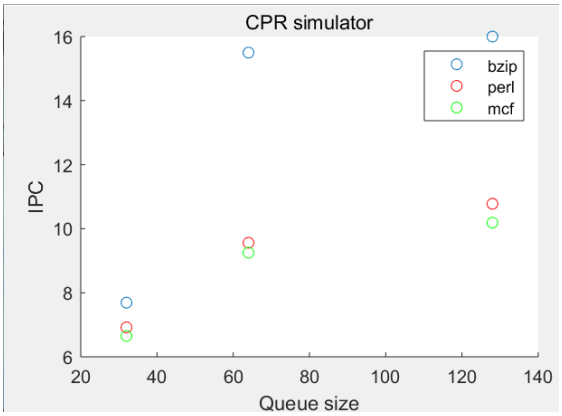

Figure 7 IPC of CPR simulator

From the tables we can see that there isn't much difference in the IPC for both the architectures. This is because the executions have been done for perfect branch predictions only. CPR is expected to obtain better IPC in cases of real branch prediction. If we consider the size of the structures used we can observe a large improvement. For example:

- Hardware structure have been removed:

  -Active list entry size:

  Flags: 1 byte
  PRF index of dst: 1 byte
  Logic index of dst: 6bit
  PC: 4 byte
  For 512 entries: 512*6.75=3456bytes

  -AMT size:
  For 64 entries: 64*1=64bytes

- Hardware structure have been added:

  +PRF usage counter:
  For 576 entries: 576bytes

  +PRF unmapped and free bit:
  For 576 entries: 576*2bit= 144bytes

The checkpoint used has a similar structure to that of the 721sim with some additional bits but the number of checkpoints used is comparatively less. Therefore we can see that the size of the structure used has reduced by a large factor.

Running of the astar_test and hmmer_test benchmarks raise an assertion indicating that the wrong program counter has been taken. This problem has been raised due to the presence of atomic instructions and the problem is yet to be resolved.

5. Future work

The next steps in the project implementation would be to include the logic to support actual branch prediction in the CPR architecture. With this we will be able to eliminate the backend bottleneck. This will be combined with the trace cache implementation to eliminate the frontend bottleneck. This implementation is expected to improve the IPC by a large factor.

References

[1] E. Rotenberg (1999). Trace Processor: Exploiting Hierarchy and Speculation (Doctoral dissertation). Retrieved from

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.3470&rep=rep1&type=pdf

[2] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. 29th International Symposium on Microarchitecture, Dec. 1996.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In Proceedings of the 34th International Symposium on Microarchitecture, pages 237–249, December 2001.

[4] H. Akkary, R. Rajwar, S. T. Srinivasan. Checkpoint Processing and Recovery: an Effort, Scalable Alternative to Reorder Buffers. IEEE Micro, pages 11-19, Nov. – Dec. 2003.

[5] H. Akkary, R. Rajwar, S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36, 2003), Dec. 2003.

Appendix

Name	Contribution Factor
Sonali Varanasi	1
Yusong Li	1
Tse-Han Pan	1
Shrirang Deshpande	1
Zhiyu Chen	1

Contribution towards research

Sonali Varanasi: Implementation of Checkpoint processing and recovery and debugging.

Yusong Li: Implementation of Checkpoint processing and recovery and debugging.

Zhiyu Chen: Implementation of Trace Cache

T. Han Pan: Generation of traces, trace buffers, designing the shifting logic and debugging.

Shrirang Deshpande: Building the trace cache, multiple predictions, hit and cache selection logic and debugging.

Contribution towards preparation of the paper

Sonali Varanasi: Explained the idea and method of implementing CPR.

Yusong Li: Explained the experimentation methodology, the results and challenges faced.

T. Han Pan: Explained the trace generation and dynamic block storing.

Shrirang Deshpande: Explained the hit logic, trace selection and discussed the outcomes.

Zhiyu Chen: Work mainly on Abstract and Introduction part. Documents and format integration.