EECS 470 – PROJECT: P6 MICROARCHITECTURE BASED CORE

TEAM EKA
Shaizeen Aga, Aasheesh Kolli, Rakesh Nambiar, Shruti Padmanabha, Maheshwarr Sathiamoorthy
Department of Computer Science and Engineering
University of Michigan, Ann Arbor
{shaizeen, akolli, rakeshmn, shrupad, msathi}@umich,edu

1 INTRODUCTION

This report describes the out of order core implemented by team EKA for the EECS 470 term project. We have based our core on the P6 micro-architecture. The core has been designed giving priority to correctness and performance. To achieve these stringent targets many advanced control and memory features have been incorporated into our design. On a baseline 2-way Superscalar system we have incorporated, 5-way instruction issue & complete, early tag broadcast (ETB), load-store queue (LSQ), adaptive instruction prefetching among other features to extract maximum out of order execution, reducing potential stalls. Apart from core features, we had a focused approach for design verification which included the development of an automated testing suite for fast and efficient regression analysis.

The following sections explain our core in detail along with the underlying philosophy which directed us to choose the particular design over other available options.

2 DESIGN

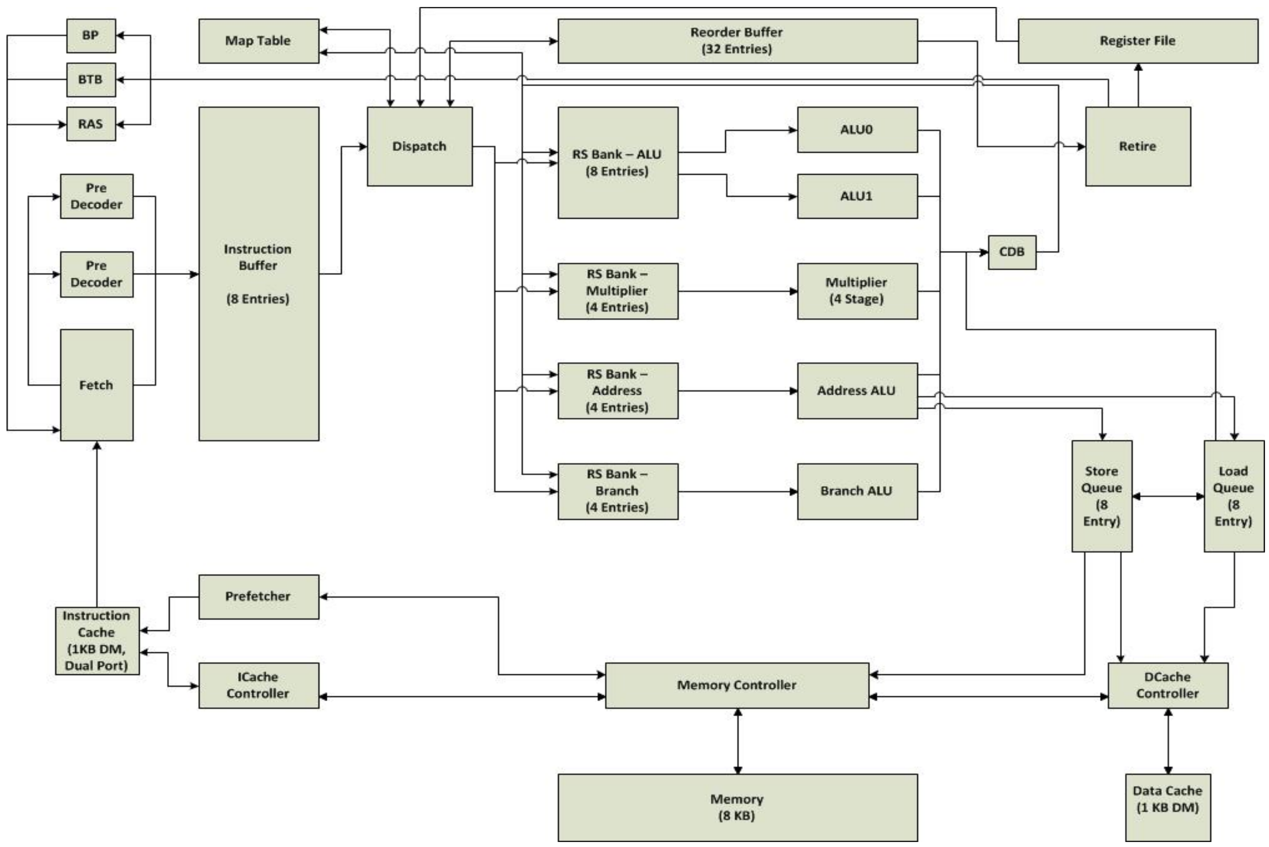

Figure 1: Design overview

2.1 Fetch Stage

The fetch stage is responsible for receiving instructions from the instruction cache and storing these in the instruction buffer. It interacts with the Branch Predictor, Branch Target Buffer and Return Address Stack in case of branch instructions to decide the next PC. The stage also pre-decodes the instructions into classes corresponding to our multiple Reservation Station banks (explained later), so they can be routed accordingly at dispatch. The pre-fetcher is also part of this stage and fetches instructions into the cache. In order to obtain a good cycle time we allow only one branch instruction to be fetched per cycle. This stage consists of the following modules:

2.1.1 Dual Ported Instruction Cache

A dual ported I-cache allows the fetch to submit requests for two non-aligned consecutive PCs in one cycle. This improvement feature reduces fetch related stalls in the pipeline.

2.1.2 Return Address Stack

The RAS is a 32 entry LIFO structure (stack) which will push return PC into the stack when it sees a call instruction and pops it when it sees a return instruction. We maintain two such stacks, one which speculatively feeds the fetch stage and the other that is updated only at retire. In case of exceptions the retire stack copy is copied into the fetch copy.

2.1.3 Branch Table Buffer

The branch table buffer (BTB) is a 32 entry circular array and is looked up using PC and simultaneously handle two retire requests. When full, the BTB overwrites the oldest entry.

2.1.4 Branch Predictor

The branch predictor (BP) is implemented as a 32 entry array with 2-bit saturating counters that's indexes with the last 5 bits of the PC. It can service two branches retiring and fetching simultaneously. The BP maintains two copies of the branch history table (BHT) – a speculative fetch copy and a final retire copy. In case of exceptions the fetch copy is updated with the retire copy. As per our analysis (detailed in sections below) a bi-modal BP gave a better performance over the g-share predictor and hence we stuck to this design.

2.1.5 Instruction Buffer

The instruction buffer is implemented as an 8 entry circular queue fed by the fetch stage. Full and empty signals help fetch and dispatch stages decide how many instructions to process.

2.1.6 Instruction Pre-fetcher

We implemented an adaptive window based instruction pre-fetcher (IPF) which is triggered in case of

a. I-cache misses
b. Last pre-fetched PC falls within a pre-fetch window size of the current PC being fetched
c. Last pre-fetched PC falls beyond twice the window size of the current PC being fetched.

With each trigger, new pre-fetch PC is decided so as to consistently maintain a window of 16 consecutive instructions in the I-cache. As soon as the pre-fetcher is triggered, requests for 16 next instructions are sent to the memory, two each for the next 8 cycles. The outstanding pre-fetch requests are maintained in an MSHR. From our simulations, a prefetching window of 16 instructions with an 16-entry MSHR provided the optimal performance. Pre-fetch requests were given the least priority in case of contention with loads or stores to memory. In the analysis section we provide a detailed analysis of the different approaches we tried.

2.2 Dispatch Stage

The dispatch stage is responsible for allocating instructions to the various structures taking into account structural hazards in these units (such as reservation station, map table etc.). Depending on available slots the unit may send two, one or none instructions. We have implemented two parallel decoders to decode instructions. The components of the stage are as follows:

2.2.1 Reorder Buffer

The reorder buffer (ROB) is implemented as a 32 entry circular queue who's tail pointer is controlled by the dispatch stage and head is controlled by the retire stage. Each entry stores comprehensible information about the instruction such as decoded register information, completed execution bits, predicted branch direction and targets for branches and exception bits.

2.2.2 Map Table

The map table is a simple array structure indexed by the register number. It can be addressed by two instructions every cycle and is used to implement register renaming in the out of order core.

2.2.3 Reservation Stations

Our core consists of four separate reservation stations (RS) with each station dedicated to a separate class of instructions, ALU (8 entry), Mult (4 entry), branch and jump (4 entry), load and store (4 entry). RS entry allocation is done using parallel priority selectors. The split nature of the RS allows for as many as five different instructions to be issued in the same cycle. Further, five CDBs ensure that each RS bank can simultaneously 'snarf' the execution results and update the dependent instructions, thus eliminating any need for buffering results.

2.3 Issue Stage

Instructions are issued from the reservation stations using pseudo-random priority selectors. The randomness ensures that instructions do not stagnate in the RS.

2.4 Execution Stage

There are 5 functional units in our pipeline: two ALUs, one address calculating unit, one branch handling unit and one 4-stage multiplying unit each connected to an independent CDB. ETB from the execution units enables dependent instructions to issue in same cycle as the completing instruction that feeds the results.

2.5 Data Memory Interface

The data memory interface is designed such that

a. Loads and stores calculate address out of order as soon as the operands are ready.

b. A load is eligible to go to memory as soon as its address is ready and

    i. No store ahead of it has unresolved address, and

    ii. It did not get a forwarded value from any store ahead of it.

c. A store is eligible to go to memory as soon as its address is resolved and it is at ROB head.

d. Processor does not stall on a write miss with stores being buffered until memory unit is available.

e. Processor does not stall on a data read miss in cache with dcache controller enabled to handle multiple outstanding load misses in cache. The number of load misses that can be handled being parameterized.

f. Fetch misses are given highest priority followed by load requests, store requests and finally instruction prefetch requests.

To achieve each of these goals we have implemented the following components

2.5.1 Load Queue

Houses load instructions from dispatch till they broadcast value on CDB after which they are deleted. It sends requests to and receives responses from the d-cache controller which in turn talks to memory.

2.5.2 Store Queue

Houses store instructions from dispatch till they are sent to memory. It also calculates dependency information for a load and forwards a value to a load with address match. Memory requests are send when available. Also, when a store is sent to memory the store address is broadcasted to D-cache controller to disable any loads with address match in MSHR from writing to cache once they get their value from memory.

2.5.3 Post retirement store buffer

This is implemented as a part of the store queue. Retired stores which are waiting for memory, wait in the store queue. We implement this by maintaining three pointers head, retire head and tail. Entries between head and 'retire head' are retired from the ROB and wait on memory and those between retire head and tail are not retired. Stores that are retired from the pipeline can still forward values to younger loads. Since even after a halt instruction is retired, there could be stores not yet sent to memory, halt is enforced only when this post retirement store buffer is empty.

2.5.4 Data Cache Controller

The data cache controller receives requests from the load queue. D-cache misses are saved in an 8-entry MSHR and are sent to memory as soon as it is available. When a load is serviced, D-cache controller broadcasts the value to the corresponding load queue entry

2.6 Complete Stage

Since we have implemented ETB for ALU, branch and multiply instructions, the functionality of this stage is satisfied in the earlier stage itself.

2.7 Retire Stage

Our core can retire two completed instructions from the head of the ROB simultaneously. Exceptions are also handled in this stage. In case of a branch miss-prediction, a special reset signal is sent which flushes the ROB, Instruction Buffer, all RS entries and execution units, the map table, and LQ. SQ is flushed differently since retired stores that are still not sent out to memory have to be handled. The BTB is also updated with the correct target. We also handle exceptions caused by halt and illegal instructions in retire. Retiring calls and returns send special signals to the RAS while retiring conditional branches updates the branch predictor with information regarding the actual and predicted directions.

3 TESTING STRATEGY

We followed a well-defined and phased approach for testing our out of order core. These phases are mentioned below:

3.1 Unit Testing

Comprehensive test cases were enumerated during the design phase and tasks were added in the module level test-benches to generate dumps of the internal stage of each module along with inputs and outputs. The same was done for the synthesized version also.

3.2 Incremental Integration Testing

We favored an incremental integration vs. a 'Big Bang' combination approach since it would be easier to locate and fix bugs arising due to incorrect interactions between modules. While this required an extra effort, we found the incremental approach to outweigh the cons.

3.3 Integration Testing with Automated Test Suite

We developed an automated test suite to compare the core output with the llvsimp or 'gold' core. The suite is designed to pick test programs from a given directory, convert them to assembly and run each for the gold core and our test core. The tool was extensively used to do a regression testing after every bug fix.

3.4 Variable memory latency Test

We tested our core both using fixed latency and variable latency per execution cycle. The latter was accomplished by using the mem_random.v file shared with us by the course GSIs.

3.5 Testing with Decaf

The decaf compiler was used to convert some sample programs such as merge sort into assembly which were then submitted to the test suite for regression test.

3.6 Hybrid Compilation

We generated make files to selectively run synthesized modules along with behavioral code so as to pinpoint synthesis errors.

4 PERFORMANCE SUMMARY

Our processor was optimized to achieve a fair balance of performance and cycle latency. On a whole, the publicly available test benches performed well and the average IPC 0.732. The cycle latency we achieved after optimization was 9ns. We found that features like instruction prefetching, ETB, multiple RS banks and LSQ, improved the performance significantly. While some other features like dual ported cache, gshare branch prediction didn't contribute too much.

Figure 2 shows the contribution to performance gain by some important features over the baseline out of order, 2-way superscalar CPU (with memory operations handled at retire and no branch prediction support).

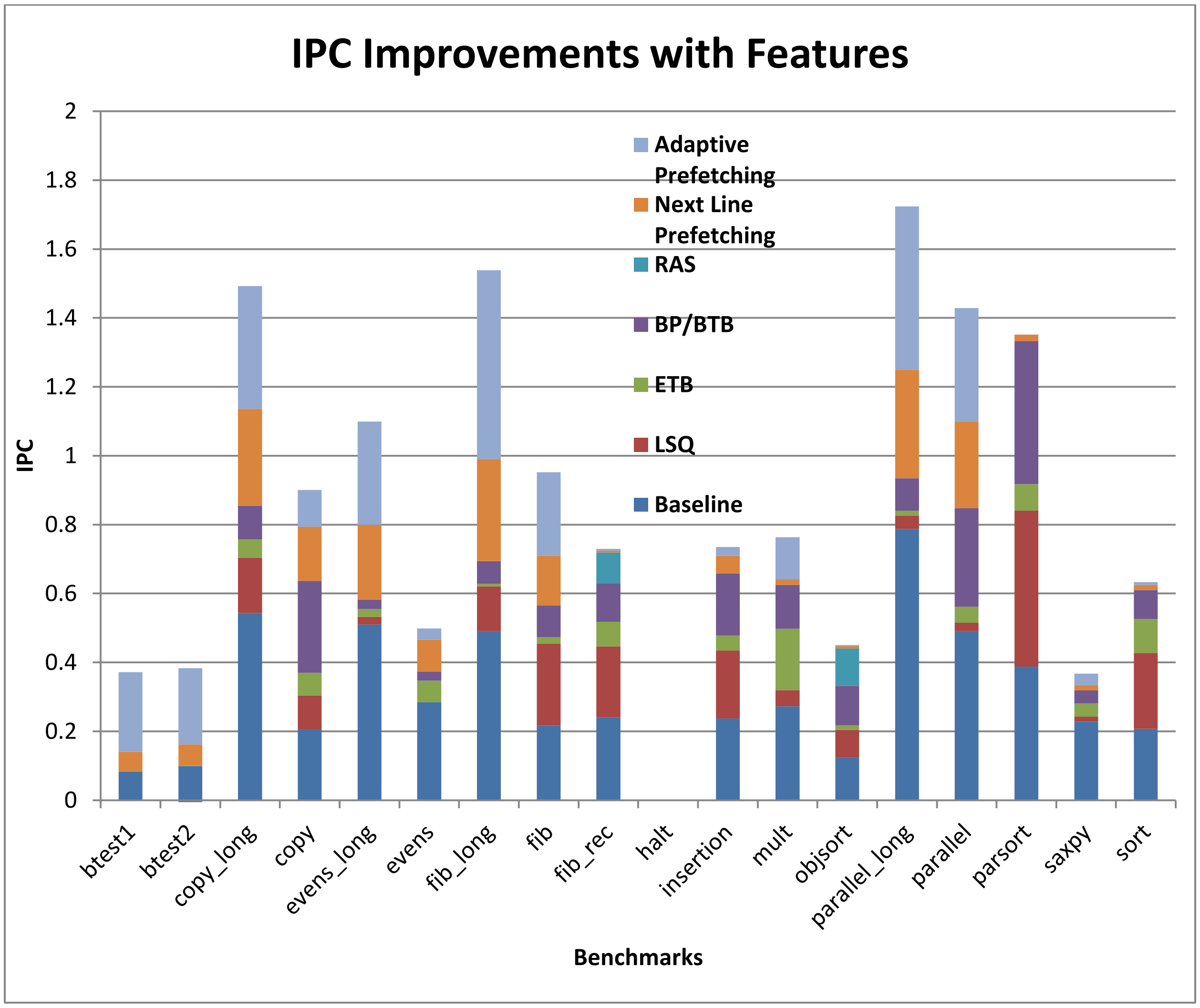

Figure 2: IPC improvements seen with different optional features over the baseline.

5 ANALYSIS

The following sections provide a detailed analysis of variations in program behavior with feature additions.

5.1 Branch Prediction

We initially implemented a gshare branch predictor and observed that it performed worse on our benchmarks as compared to a simple predictor with a local branch history table prompting us to choose the latter in our final design. We analyzed the gshare with a Global History Register of 3 and 5. Most of the benchmarks available have independent branches or loops with a small trip count. For example, with a 3 bit GHR, you need a warm up time of 8 out of 16 iterations for a regular loop. A local 2 bit saturating counter (initialized to 10) on the other hand mispredicts only once. Figure 3 shows a comparison of the Gshare vs bimodal vs always taken vs always not taken branch prediction. We observe that the bimodal predictor is consistently giving us the best predictions. **Also, we observed that initializing our saturating counter to 10 gave an average performance gain of 20%.**

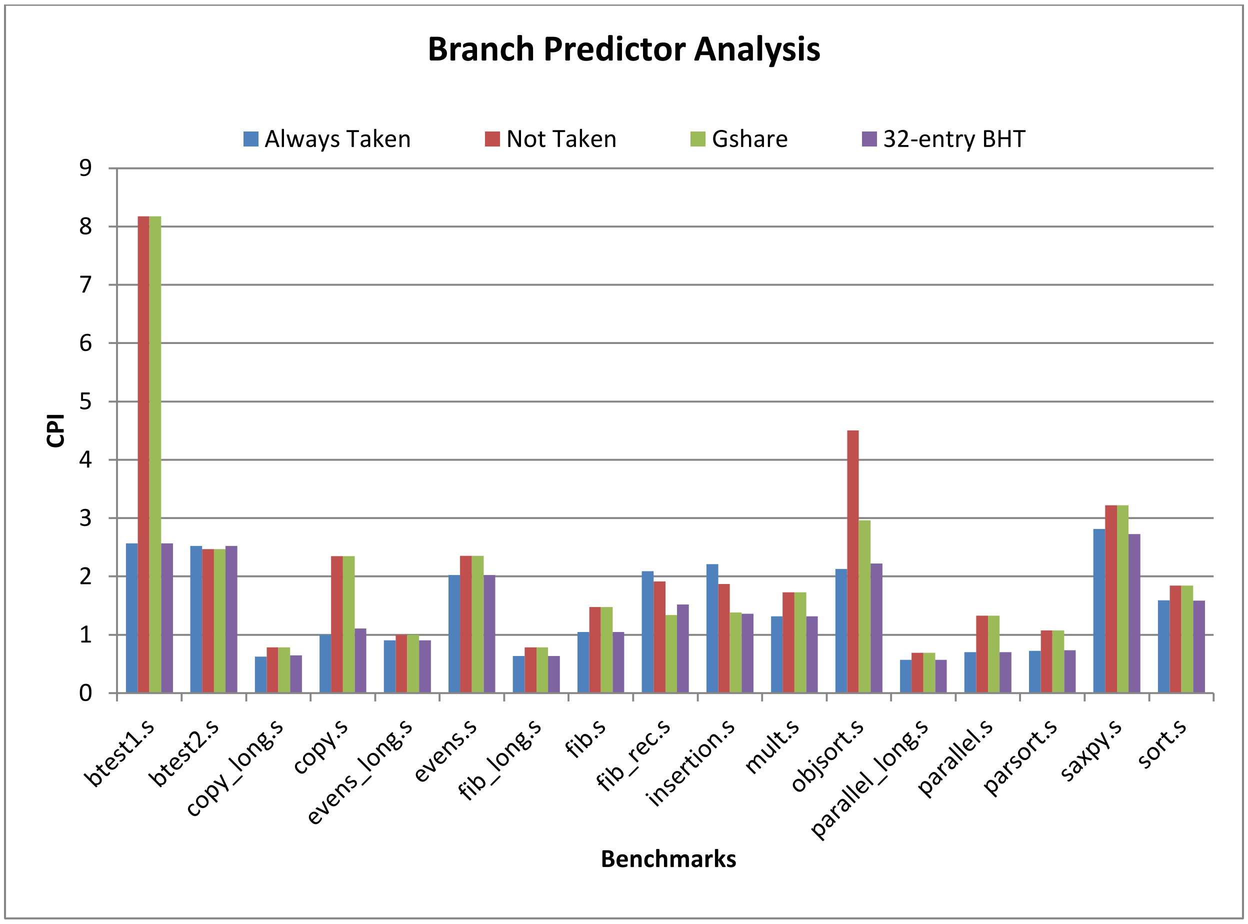

Figure 3: Branch Predictor Analysis for different benchmarks

Btest1 and btest2 showed no improvement with any kind of branch support due to non-repetitive, unpredictable branches. On the other hand, regular loops in other programs benefitted considerably.

5.2 Return Address Stack

We observed that a simple RAS stack maintained by the fetch stage gave us marginal performance gain on the benchmarks due to speculative pushes and pops effected on a speculated control path, which rendered the RAS useless. In order to improve this gain, we implemented the speculation aware RAS. This change gave us a good performance gain (around 25% on objsort) without appearing on the critical path.

5.3 Multi - banked Reservation Station

The selection of a multi-banked design which leads to more complex hardware would be validated if more than 2 instructions were issued in a cycle often. We conducted experiments to verify this. The issue frequencies for three benchmarks are shown in Figure 4. Many benchmark programs issued 3 instructions/cycle. Complex programs such as fib_rec which contained a varied class of operations were able to extract most benefit from the design. We also observed that certain programs with a limited set of instruction classes like btest1 didn't benefit from this design.

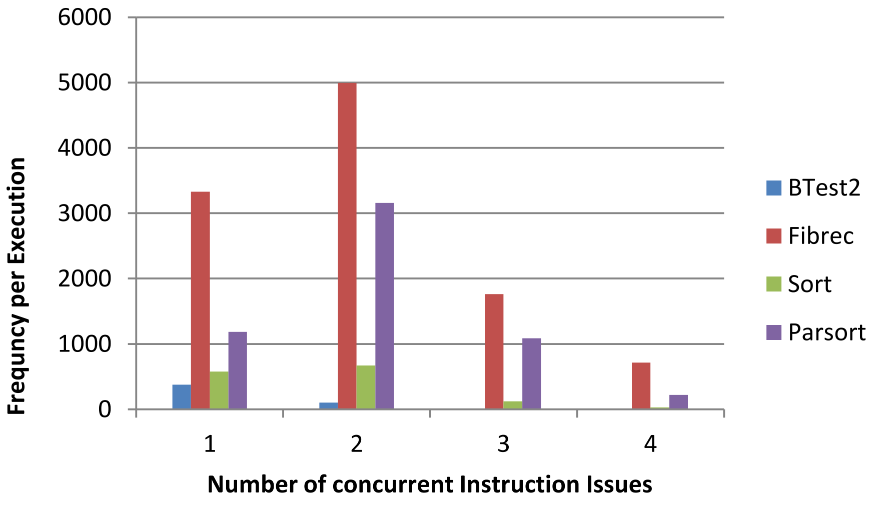

Figure 4: Issue Rate Analysis

In conclusion, the selection of a multi-banked RS design was a good decision which helped increase performance of many of our test programs.

5.4 Adaptive Pre-fetcher

Figure 5 presents the CPIs for three test systems; first with no pre-fetching, second with next line pre-fetching and third with our adaptive pre-fetching. Btest1 and Btest2 are the biggest winners with our pre-fetcher. Constant jumping of PC seen in btest1 and btest2 ensures that maximum benefit is seen only with our aggressive adaptive pre-fetcher. Having a 16-entry MSHR and employing a prefetching window of 16 instructions gives the pre-fetcher enough resources to fetch up to a maximum of 32 instructions simultaneously. Our memory controller gives pre-fetch requests last priority behind instruction cache misses, load misses and store misses in that order. This ensures that pre-fetch requests never interfere with program flow. On an average, adaptive pre-fetcher improves CPI by 51.2% over no prefetching and by 32% over next line prefetching.

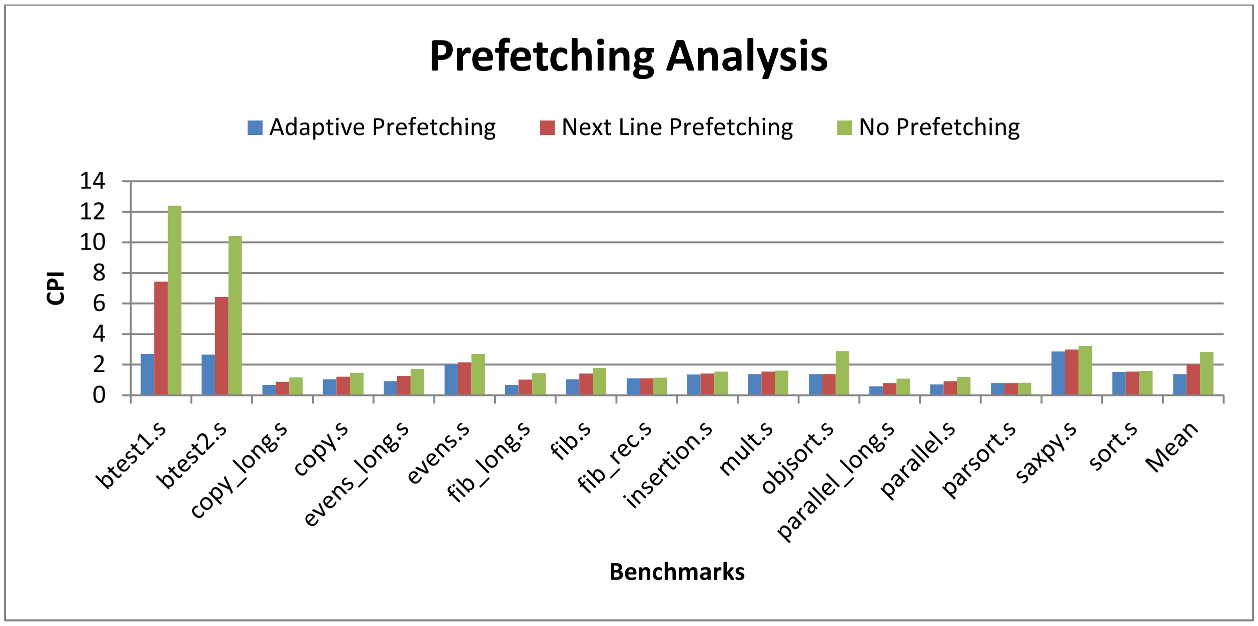

Figure 5: Prefetching analysis

5.5 Load/Store queue

We see here that in benchmarks like copy where every load was preceded in dispatch order by a store to the same address, our store-load forwarding led to no loads being sent to memory. From Figure 6, we can see that other benchmarks too benefit from this as the number of loads send to memory is considerably reduced.

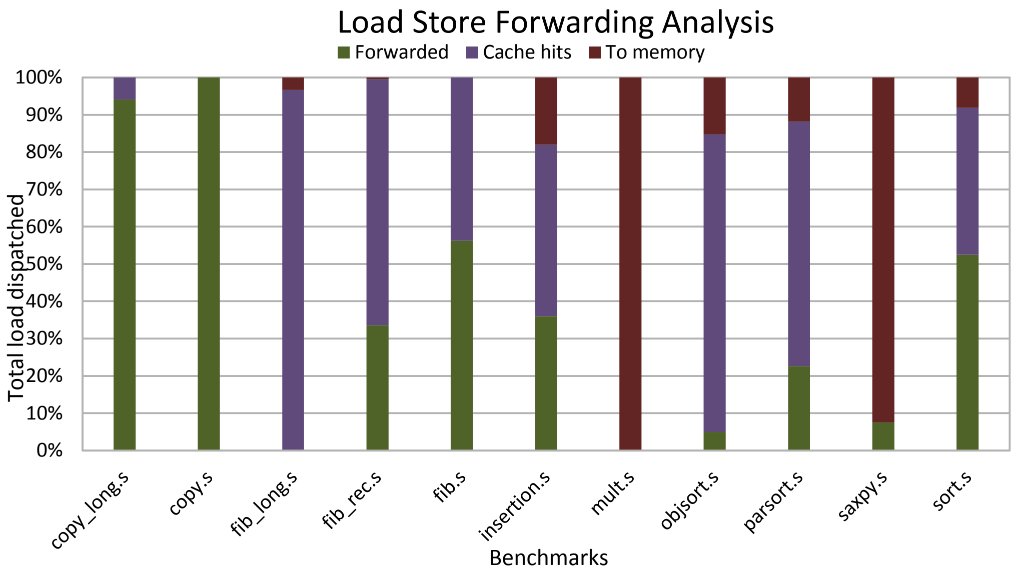

Figure 6: Load Store Forwarding Analysis

5.6 Component sizing

Another experiment we did was to gauge the reason for stalls in the pipeline due to structural hazards. This analysis was used to finalize the sizes of the components in our pipeline to an optimal number. Figure 7 gives us an idea of the structural hazards that contributed to stalls in our pipeline, discounting stalls because of instruction fetch misses.

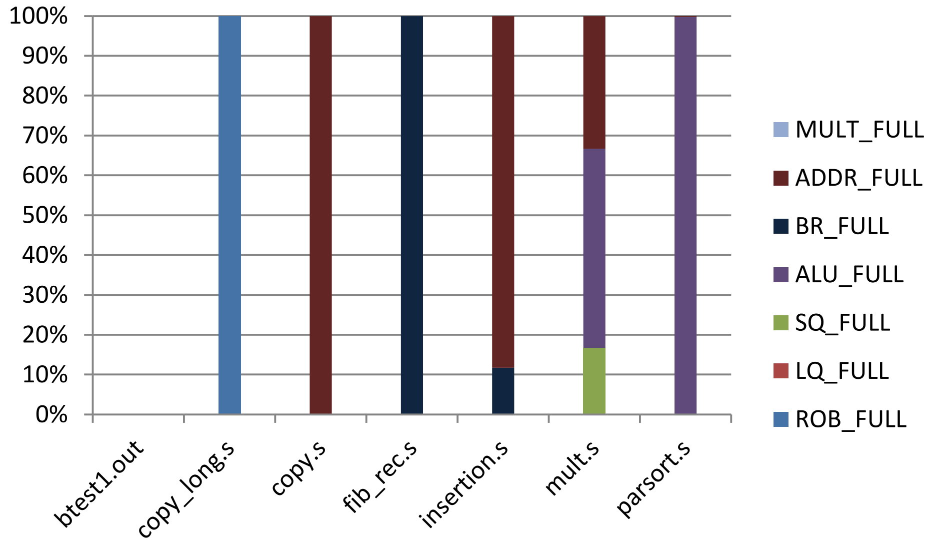

Figure 7: Structural Hazard Analysis

Based on this analysis, and keeping in mind latency of each cycle, we narrowed down our RS bank sizes to 8 for the ALU instructions and 4 for the other classes. LQ and SQ sizes were fixed to 8 each.

6 Involvement

Shaizeen Aga (20%): Ins Buf, ROB, Dispatch, LSQ, D$ Controller, Prefetcher, Post retirement store buffer
Aasheesh Kolli (20%): RS, Pseudo random issue and dispatch, ETB, D$ Controller, Prefetcher, I$
Rakesh Nambiar (20%): BTB, Speculative RAS, RS, Execute units, Regression suite
Shruti Padmanabha (20%): Fetch stage, MT, BP, Speculative RAS, ETB, Baseline OoO
Maheshwarr Sathiamoorthy (20%): gshare BP, Speculative RAS, BTB, RS, Execute units, ETB

7 Conclusion

We have successfully implemented an out of order core based on the P6 architecture. Our performance and correctness analysis shows that most of our design choices have led to superior performance while at the same time ensuring execution precision. Specifically use of multi-bank reservation stations reduced our CPI in many programs by supporting up to 5-wide issues and completes. Our adaptive pre-fetcher successfully reduced the instruction miss rate while use of the RAS and BI-Modal branch predictor improved CPI for branch heavy programs. The core was able to hide memory latencies by implementing multiple outstanding load missed past pending stores.

Finally, we express our gratitude to Prof. Wenisch, Drew and Faissal for their support and guidance throughout the course of the project.