Cannonlake_master P6 OoO Core Design

Daichi Fujiki, Vidushi Goyal, Timothy Linscott, Harini Muthukrishnan, Peter Paquet
University of Michigan, Ann Arbor
{*dfujiki,vidushi,timlinsc,harinim,pmpaquet*}*@umich.edu*

***Abstract*—In this report we describe the P6 architecture implemented for an out of order processor. The design includes features like prefetching, early recovery, LSQ with store-to-load forwarding, branch predictor, and instruction buffer. The processor is more focused on achieving low clock period along with correctness. We build a debugger to give us intermediate states of different modules which can be analyzed to debug the issues. Lastly, we also support automated testing infrastructure which gives performance statistics as well.**

The report is organized as follows: We first present an overview of our design. We then outline the advanced features we have implemented. We then present the results of a series of characterization tests we performed. Finally, we give a small discussion and contribution by each of the team member.

I. Motivation

Our primary design philosophy was to first ensure correctness along with a higher clock frequency, and then to optimize for CPI. This prompted us to implement a P6 architecture instead of a R10K, as simplicity was more favorable than potential throughput. Following this philosophy, we also kept a scalar design. Our optimizations were primarily focused on branch prediction and recovery, and mitigating the 100ns memory latency. In the conclusion of the project, we vigorously tested our processor on a large suite of testcases, altering parameters and memory latency to detect and remedy as many pathologies as possible.

II. Design

As best as our testing has shown, our design is fully functional, and none of major features were cut. However, fully-associative caches were implemented briefly, then cut from the final submission because testing was not showing that it improved performance. Our branch predictor is functional, but our gshare design and direct-mapped BTB are poorly suited for the benchmarks, yielding a below-average accuracy. All the stages in the design appropriately handles branch misprediction flushes.

A. Fetch

This stage fetches the instruction from the memory in-order. The fetched instructions are put into an Instruction buffer(IB). In case the instruction buffer is empty, the fetch stage directly places the instruction in the pipeline register, bypassing the IB. The instruction buffer, which holds the fetched instructions waiting to be dispatched is dual ported with 8 entries. This was aimed to prevent fetch related stalls in the pipeline. The instruction buffer also predecodes to check if the instruction is a branch, in-order to aid the Branch Predictor.

B. Dispatch

Dispatch stage consists decoder and is responsible for interacting with Map Table, LSQ, and Re-order buffer(ROB). Map table, LSQ, and ROB populate their entries passed on by dispatch. It identifies structural hazards due to ROB, Reservation Station (RS), and LSQ and gives out an appropriate signal to stall the pipeline. The identifier for the instruction i.e ROB tag along with decoded instruction are passed down the pipeline.

1) Re-order Buffer: Re-order buffer stores all the incoming entries to the dispatch. It is a circular queue with 32 entries with head pointing to the next instruction to be retired and tail pointing to the next incoming entry. It handles flushing of instructions on branch misprediction. Instructions are stored in program order in ROB and thus is responsible for retiring them in same order. On CDB broadcast ROB entry is marked as available for retiring. During retire phase, ROB writes back to the register file. It issues a flush signal to the complete pipeline when a mispredicted late recovery branch retires. When all the ROB entries are full, the pipeline stalls due to ROB structural hazard.

2) Map Table: Map table is responsible to store the current state of the processor i.e mapping of ROB tag to the register index (instruction which will write to the particular destinati register index), the availability/non-availability of the register value, and the current location from where the value can be read i.e CDB broadcast, ROB, or register file. Map table is also responsible for creating backups and restoring them back to support single cycle recovery in case of branch mispredictions.

C. Issue

Issue stage consists of Reservation station block. It is allocated simultaneously along with ROB, Map Table, and LSQ. It supports 4 kind of reservation banks each dedicated to ALU, Multiplier, branches, and memory instructions. Each instruction bank can support 8 entries. It stores all the relevant information required by an instruction i.e

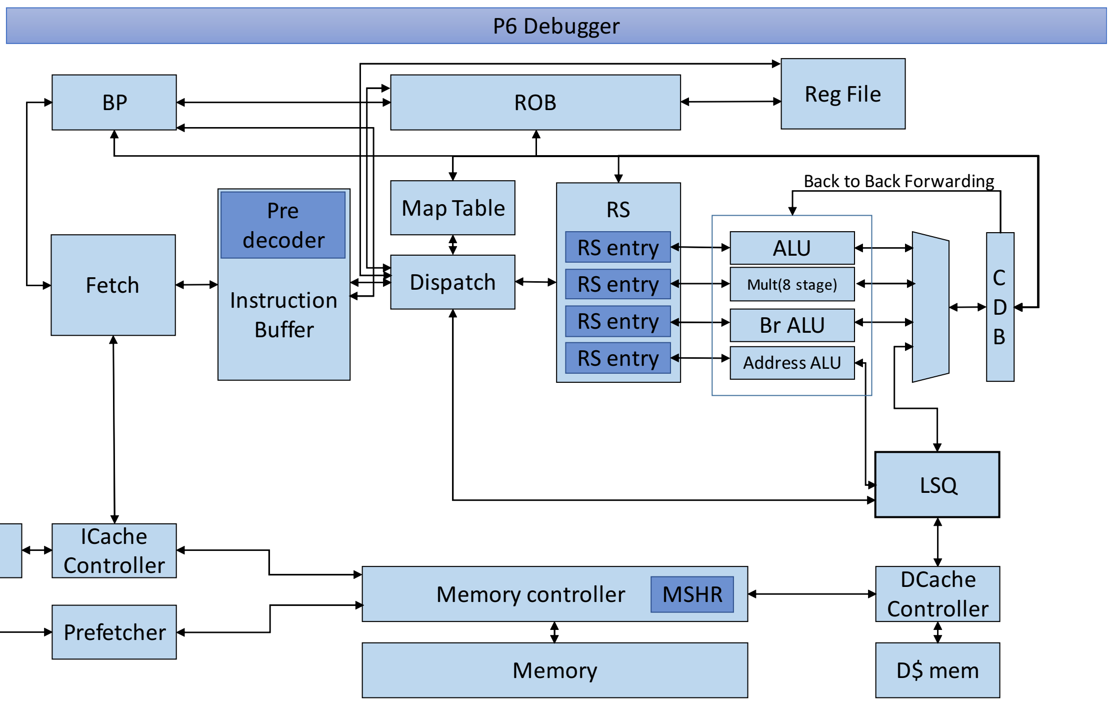

Figure 1: Block diagram of our architecture.

ROB tags for the input registers, destination tag, and the values for the input registers. For debugging purposes, the reservation station in the implemented design also stores program counter, and the Instruction which is passed down the pipeline. It holds instruction till all the required inputs are present or the functional unit (FU) is busy. RS can give rise to structural hazards and stall the pipeline in absence of availability of empty FU reservation bank.

D. Execute

Execute consists of 4 FU for each kind of instruction viz ALU, Multiplier, ALU for branches, ALU for address calculation for memory instruction. The design uses a 8 stage multiplier. The inputs from issue stage are passed to the corresponding FU. ALU dedicated to address calculation passes the ouput to LSQ. The outputs from non-memory FUs are passed to complete stage.

E. Complete

The Complete stage handles the arbitration between the inputs received from LSQ and the different units of the execute stage. The arbitration scheme used is fixed priority with Load/Stores given the highest priority followed by Branches, Multiplier and ALU. The output of the arbiter is then fed to the CDB that broadcasts the result to the preceding stages. This in turn enables the retiring of instructions.

F. LSQ

We have a split load-store queue that handles all the memory requests from the X-stage and sends them to the Dcache. An entry is allocated in the LQ/SQ upon dispatch. It then waits to receive the request related data from the X-stage before sending a load/store command to the Dcache. The load response received is sent to the CDB for broadcast. An LQ entry is retired once it is at the head of the queue. A write request is sent to the Dcache when the corresponding entry in the ROB retires. The SQ entry retires once the request acknowledgment is received from memory.

G. Caches

Both the Dcache and Icache are of 256-bytes and 32 lines. The Dcache is write-through and non-blocking. The Dcache controller receives the read request from the LSQ, searches through the cache memory, if it is a hit it responds back with the data. In case of a miss, a read request is sent to the memory. The response data from the memory is written back to the Dcache and also forwarded to the LSQ. In case of stores, the data is written to the Dcache and also the write request is placed to the memory. When the response to the previous read and the next store both arrive in the same cycle, the priority is given to the read data to be written back to the dcache. The store is postponed by 1 cycle in that scenario. The Icache stores the data read by the prefetcher and Icache controller.

H. Memory Controller

The memory controller has a fixed priority arbiter to process the requests from Dcache, Icache and Prefetcher (in the order of priorities). It also includes a Miss Status Handling Register(MSHR) which keeps track of the requests from the 3 initiators and decides where to route the response to.

III. Feature Highlights

A. Pipeline Enhancements

1) Early Branch Recovery: Our Early branch resolution mechanism initiates flushing of the dependent instruction as soon as their outcome is identified by the execute rather than issuing a flush when the mispredicted branch retires from ROB. The implemented design supports 2 early recovery branches. Other branches in flight are marked late recovery in dispatch. Due to complexity involved in storing multiple backups of Map table for each early recovery branch we decided to resrtict the number to only 2. Each of the early recovery branch has a dependency tag (which is equivalent to ROB tag). Every instruction after an early recovery branch holds that dependency tag which is decided by dispatch. The Map table holds the backups for each of the early recovery dependency tag. These dependency tags are passed along the LSQ, Issue and Execute stages. When an early recovery branch is mispredicted all the entries in LSQ, reservation station, and execute depending on that branch are flushed. The Reorder buffer realigns its tail pointer to the entry next to the mispredicted branch. It does not wait till all the entries ahead of the mispredicted branch to retire, before taking in the next instruction from dispatch whereas it waits in case of late recovery. On an early recovery misprediction, Map table restores the backed up state for the branch and frees the corresponding map table backup. Handling early recovery along with late recovery branches was a major challenge as it involved proper coordination between them to ensure program correctness. All the modules were made capable of handling two flushes i.e an early recovered branch flush, and the late recovery flush on retiring of branch.

2) Executing Back-to-back dependent instructions: The design supports bypassing of values from CDB directly to the execute stage. To enable this feature, the rob tag is back propagated to issue stage directly from execute rather than CDB. This facilitates issuing of instructions one cycle earlier before the data is broadcasted on CDB. The execute picks the value directly from CDB.

B. Validation/Testing Features

1) Debugging Tool: Our debug infrastructure generates cycle by cycle status of the instructions in different stages of the pipeline as logs. This helped us visualize the behavior of the different modules, quickened up the debugging process.

2) Regression Testing Infrastructure: Our regression infrastructure involved bash scripts that (1)ran different testcases on the Golden design(inorder pipeline) and our design implementation (2)compared results between the two runs and mark each testcase as PASS/FAIL. This was done every time a design change was made to check the validity of the change.

3) Feature-kill switches: We implemented feature-kill switches to record the incremental improvements. Kill switches were also used to debug our design so that we can narrow down the error zone.

C. Load-store Processing Elements

1) Store-to-load forwarding: When a load is allocated into the LQ, it records the current tail pointer of the SQ as its ”forwarding position,” which specifies one-past the youngest store instruction that can potentially forward its data to the load. When the load instruction receives its computed address from the ALU, it searches through the store queue for an instruction with a matching address. If an address match is found in the store queue between the store queue head (inclusive) and the load instruction's ”forwarding position” (exclusive), the load instruction receives its data from the youngest of such store instructions. If a store instruction between the store queue head (inclusive) and the load instruction's ”forwarding position” (exclusive) does not yet have its address/data from the ALU, the load instruction stalls and waits for all such stores to receive their addresses/data before proceeding. If no such store is missing data and no such store has a matching address, the load instruction proceeds to acquire its data from the cache.

Optimizations to the store-to-load mechanism include keeping track of when each load's ”forwarding position” becomes invalid, which is caused by the store queue's head pointer advancing to the same entry as the ”forwarding position.” This is done so that a load instruction can skip forwarding and go straight to memory if its ”forwarding position” is invalid. Another optimization is, at every clock cycle, resetting the load entry that is attempting to forward its data from the store queue to the oldest instruction in the load queue that has a computed address and a valid ”forwarding position.” By definition, the oldest of load instruction will have the least number of potential store queue entries to forward from, and hence has the least likelihood of having to stall due to a store not yet having its address/data.

2) Loads issue out of order past pending stores (non-speculative): If a load instruction cannot forward data from the store queue due to either a now-invalid ”forwarding position” or no address matches (with no potential forwarding SQ entries missing their address/data), the load instruction will send a read request to memory during the following cycle. The load does not need to wait for all previous stores to write to memory; it only needs to wait for all previous stores to receive their addresses/data from the ALU.

3) Multiple Outstanding Load misses: A load instruction will only be actively be sending a read request until it receives a read-request-acknowledgement signal from the data cache. So, during the next clock cycle, while the previously mentioned load instruction is waiting for its data from memory, a new load instruction can send another read request to the data cache. When the data cache replies with valid data, the data cache's response address is compared to the addresses of all load instructions that previously sent memory requests and are waiting for responses. If the addresses match, the data cache response is assigned to the corresponding LQ entry. Else, the response is discarded.

D. Memory System Enhancements

1) Next-line Prefetching for Instructions: We implemented a next line prefetcher that places a read request preemptively for the next instruction. The next line prefetcher design was simple to implement as we had appropriate hooks placed for it during the memory controller design that was done earlier on. Despite the simplicity, it gave a very good CPI gain without adding to the critical path.

2) Fully Associative Cache: We implemented a fully associative Cachememory with true LRU. But towards the end, we decided to have both the Icache and Dcache as direct-mapped implementations. This was done for two reasons: (1) During synthesis we observed that the critical path was from the Memory Controller->Instruction Cache->FetchStage->InstructionBuffer->BranchPredictor. Full associativity for ICache increased the slack of this path. (2)There was no substantial performance gain observed for the Dcache. Hence, we turned it off in our final submission even though the design for it is fully functional.

3) Memory controller with Global MSHR: The memory controller has a global miss status handling register that holds the status of the transactions coming into it. The number of MSHR entries is the same as the total number of outstanding transactions supported by the memory.

E. Branch Prediction

Our branch predictor uses three components: a Branch Target Buffer (BTB), and Pattern History Table (PHT) and a Branch Status Table (BST). The BTB and PHT are both 512-entry direct-mapped caches. The BTB is index with bits from the PC, while the PHT is indexed with the PC bits XOR'ed with a four-bit Branch History Register. The BHR is speculatively updated with every branch, while the PHT is updated only when a branch is successfully retired. The purpose of the Branch Status Table is to keep track of in-flight branches, their originating and target PCs, and the backup BHR associated with the branch. The BST essentially is a disconnected extension of the ROB, since active entries in the ROB are matched with entries in the BST, since any entry could be discovered to have been a branch at Complete.

Table I: Parameters of processor modules.

Module Name	Parameter	Value
Branch Predictor	BTB	512
	# of Early BR Recov. Support	2
Reg. File	# of Regs	32
RoB	# of Entries	32
RS	# of Banks	4
	# of Entries/Bank	8
Instruction Buffer	# of Entries	8
Dcache/Icache	# of Bytes	256
	# of lines	32
MSHR	# of entries	15
LSQ	# of LQ entries	8
	# of SQ entries	8

There were many additional improvements to the branch predictor that we considered but did not implement. We originally wanted the branch predictor to be able to learn about branches from Dispatch to catch misses early, but left this out due to the rapidly growing number of simultaneous inputs the branch predictor was already handling. The branch predictor also has very little infrastructure to support jumps to registers, so it will usually mispredict on these instructions. We had originally planned to include a return address stack, but we the cut feature after milestone 2. The branch predictor and its recovery logic turned out to be one of the bug-infested portions of the code due to the large number of contingencies it had to account for.

IV. Synthesis

We created a complete synthesis flow for our design. This flow uses compile followed by optimize netlist and incremental compile. The final clock period we can synthesize our design successfully is a bit less than 9ns. Optimize netlist did not give an improvement in slack but was just an area optimization technique. We also made sure that our design is free of latches and removes all the timing-loops detected by the Design compiler.

V. Performance Analysis

A. Overall Performance

Overall, our processor achieved a clock period of 9ns and CPI averaging 2.29 on the non-pathological benchmarks. Its best was 1.29 on parsort.s and its worst was 4.9 on objsort.s. Fig. 2 breaks down the results for every test program. objsort.s performs poorly because our branch predictor is unable to track with its unique control flow and mult_no_lsq.s creates dependencies on serial multiplies that we could not schedule around. The following section breaks down the impact each improvement had on the performance. Fig. 4 shows how our performance suffers from a variety of changes made to the design, especially removal of our core feature set.

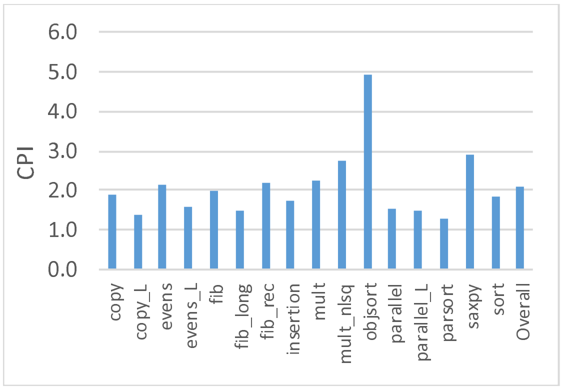

Figure 2: Benchmark CPI.

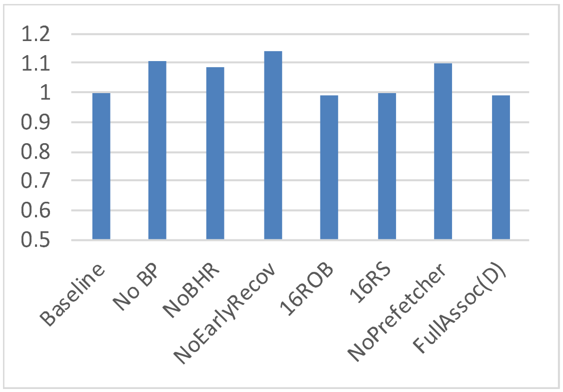

Figure 4: CPI breakdown for feature shutdowns

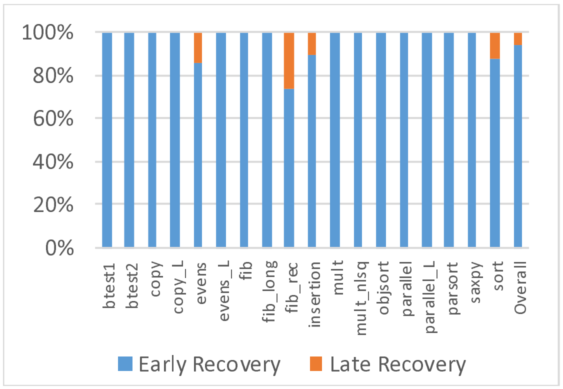

Figure 3: Early Recovery Fraction.

B. *Pipeline Enhancements*

Even though the interaction between early and late recovery branches was very difficult to debug, we found that our system benefited overwhelmingly from the early recovery infrastructure. Overall, 94% of the branches in the benchmark suite were covered by the early recovery logic. Fig. 3 breaks down how well each program took advantage of early recovery. Generally, only the more complicated, branch-heavy programs where the branch predictor performed poorly were affected by late recoveries. There is some overlap between the branch predictor and early recovery, which explains why removing one or the other has a comparatively small impact on performance. A better branch predictor means fewer late recovery branches get mispredicted while early recovery makes the effects of a misprediction less severe.

Back-to-Back dependency gave us a performance boost. It saves 1 cycle for each of the back-to-back dependent instruction. This was used on about half of all in-flight instructions, counting those that would get squashed later. Altogether, it amounted to about a -0.135 boost to our CPI.

C. *Memory System*

Our memory system enhancements were done to partially hide the fixed 100ns latency of the main memory. In this context, adding a next-line pre-fetcher gave us a CPI gain of -0.287, which is substantial. The introduction of caches for instruction and data benefited the system immensely, as can be seen from the percentage of cache hits.

D. *Branch Predictor Performance*

When considering the performance of our branch predictor, it is useful to note where it performs best and where it under-performs. mult.s and copy.s are both very short programs going through a 16-cycle for loop. By the time the BHT has adjusted to the loop's pattern, it has already accumulated a significant number of mispredictions. Longer, more complicated programs like insertion.s and fib_rec.s show significant gains over naive branch predictors, since there is more time to learn patterns. evens.s, with a 98.13% prediction accuracy is the quintessential case for a gshare predictor. It walks through a for loop, and checks whether it is on an even-numbered cycle. There are enough branches for gshare to learn on, and the pattern is always T-NT-T-NT until the loop terminates. Despite this, by averaging our prediction accuracies, we only have a 56% rate of successful prediction.

Fig. 5 compares different, more naive types of branch predictors we could have implemented.

E. *Discussion*

Knowing what to optimize in advance would have helped us choose our set of target features. Benefits from the branch predictor and early recovery overlapped, making a fancy branch predictor less important. We quickly discovered that memory was the limiting factor in our design, which inspired us to add the instruction buffer, prefetcher and Icache improvements near the end. By choosing a scalar design, it was easy for us to include

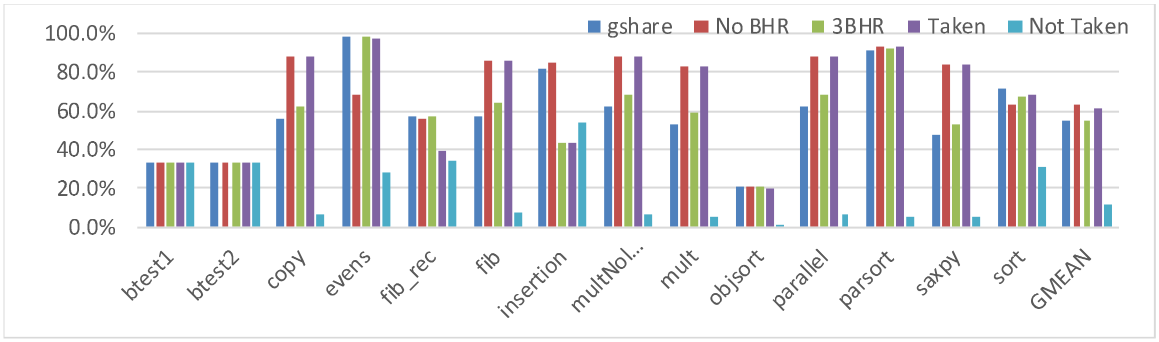

Figure 5: Comparison of Different Branch Prediction Techniques.

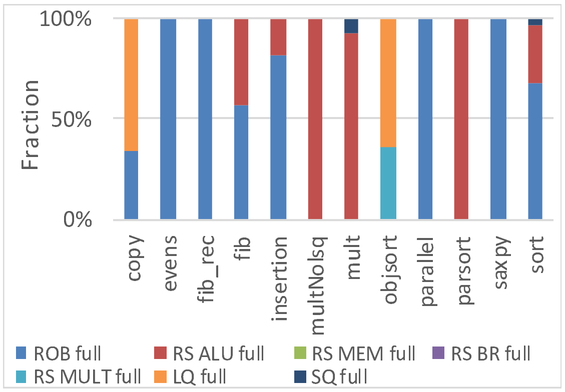

Figure 6: Breakdown of Stalls due to Structural Hazards

features that required additional communication between modules like back-to-back issuing of dependent instructions and early recovery.

We wanted to know whether structural hazards were a problem in our pipeline. Fig. 6 shows the breakdown of hazards in the pipeline. Different programs fill up the resources in the pipeline, as we expected. However, when we ran tests increasing the size of the various data structures, we found that overall [76

VI. Contributions

We had a dedicated and hardworking team, each of whom made invaluable contributions to the project. No one person deserves more credit on the project than any other. Daichi and Harini built the Fetch stage, instruction buffer, Reorder buffer, data/instruction caches, prefetcher and the memory controller. Harini also worked on resolving the timing loops. Vidushi built execute, complete and dispatch stages along with the related features of early recovery, and back-to-back dependency. Tim managed the integration of the top-level units in pipeline.v, created the reservation stations, map table and branch predictor, and worked on the early recovery and back-to-back issuing logic. Peter implemented the load-store queue, created the debugging and regression testing framework, and wrote additional performance analysys scripts. Everyone contributed to the full system testing and debug.

Finally, we would like to express our gratitude to Prof. Wenisch, Steve Zekany and Marshall Versteeg for their valuable suggestions and help throughout the project.