SPAM

Super Processors at Michigan

Group Members:
Brad Campbell
Mike Metzger
Craig Yelton
Feng Li
Changhoon Yoon

The Can Opener

Introduction

The term project for EECS 470 was to build on a simple pipeline and create a more advanced processor with new features covered in the course. For this project, five students gathered and formed a team called Super Processors at Michigan (SPAM). Team SPAM implemented many features into the MIPS R10K architecture and created the processor called the Can Opener.
Team SPAM successfully implemented all of the required base features stated on the term project specification in the Can Opener. The required base features are an instruction cache, data cache, multiple functional units with varying latencies, out-of-order implementation and dynamic branch prediction. SPAM also implemented 2-way superscalar, instruction prefetching, an enhanced load-store-queue, advanced branch prediction and a return address stack.

Design Details and Features

Superscalar

Our primary enhancement was two-wide superscalar. We made sure that every stage of our pipeline can handle two instructions at a time. The only exception is the connection to memory. Some sections can handle more than two instructions at once. We are able to complete up to six instructions at a time: two alu operations or stores, two multiplies and two loads. We can also handle up to five instructions in the dcache: two loads, two stores, and another load coming back from main memory. Fetch, decode, issue, and retire are all limited to a maximum of two at a time.

Prefetching

Our pipeline includes instruction prefetching. On every clock cycle we request the next icache line that is not already in the icache, starting at the current PC. This continues until the next branch. Then we start requesting sequentially from the new PC. We do not have a limit on the number of cache lines that our prefetcher will request.

Store to Load Forwarding

The ability to forward data from stores to loads in the LSQ is the driving factor behind the design of the load line module. Load to store forwarding is accomplished mostly within each individual load line, with some functionality in the store queue to output the appropriate data to each load line. The load line has several states where it looks for forwarded data from the store lines.
When a load comes into the load queue from decode, it saves the location of the head and tail pointers of the store queue so it knows which stores in the queue could possibly have valid data, as well as

which ones have possible address dependencies. When execute outputs a load instruction's address, all of the store lines in use check to see if they have the same address. If they do, they output a valid bit and their data. The store queue uses a priority selector based upon the load line's saved tail pointer to determine which valid store line match is from the youngest store that is still older than the load, and outputs that data to the load line. The load line stores this data and sets a bit that prevents it from going to the data cache.
In addition to the store queue head and tail pointers, the load line takes in a store queue status array when it arrives from decode. This array is a bit vector where a bit is high if the corresponding store line is in use but does not yet know its address. Every cycle, the load line checks the inputs from the execute stage to see if any stores have resolved. If a store did complete, the load line sets the bit in the status vector low to note the store received its address. If the address matches the address of the load line, it checks to see if that store is newer than its current forwarded data, and saves that data if it is. Once the array is all zeroes, the load line knows there are no earlier stores still waiting to resolve their address, and goes to the dcache if it did not receive any forwarded data.

Loads complete out of order

Completing loads out-of-order is a convenient by-product of store to load forwarding. When the store queue status vector in the load line is all zeros the load knows it is ready to complete. If it already has data it is sent to the physical register file, otherwise it is sent to the dcache.

Multiple outstanding load misses

Because the Can Opener features store to load forwarding, the number of requests the load-store queue makes to the data cache are somewhat minimized. However, when there are requests, it is possible for the data cache to miss an incoming load and then be ready to accept another before the data returns from memory. This is handled entirely in the dcache with a simple finite state machine and appropriate inputs from memory. Every load request that enters the dcache is first looked up in the cache memory and then immediately sent back to the LSQ if there was a hit. If there was a miss, the cache goes into a brief recovery period, which can take several cycles depending upon how many instructions were sent to the cache and require main memory accesses. Up to two loads can be sent to the dcache at a time. If either or both loads are misses, the dcaches goes into a blocking state while it sends the load requests to memory. Once the requests have been made, the dcache will accept more loads before main memory returns the data for the previous loads.

Write back cache

Our write back cache follows the standard method for a write back cache. Having a write back cache allows us to handle stores in one cycle and retire them from the ROB at the same time we retire them from the LSQ. Because the cache does not require a memory access for a store entering a cache block that is not already dirty, we are able to send stores to the cache and not worry about dependent loads or having to stall the ROB until the store has finished writing to memory.

Branch Prediction and Branch Target Buffer (BTB)

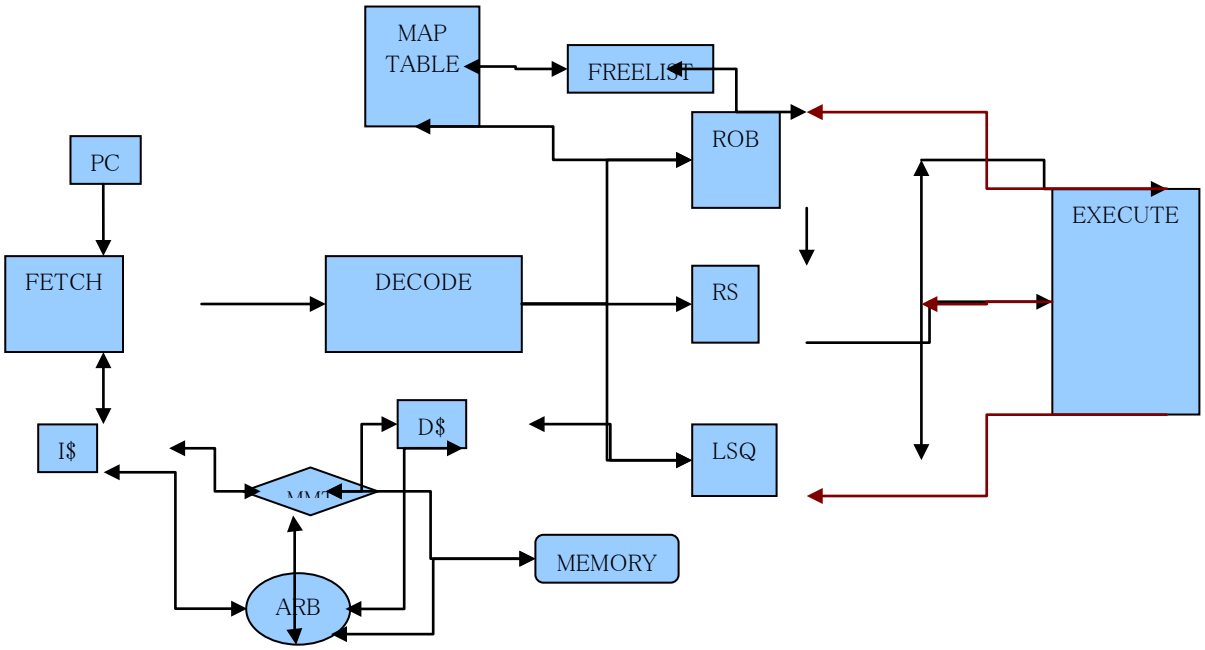
Our processor has one static and three dynamic branch predictors. We implemented a vote predictor to take the majority vote of the three dynamic predictors. The static predictor always predicts either “taken” or “not taken” and is set at compile time.
Even if a predictor predicts ‘taken’ the branch logic will only tell the pipeline its predicted ‘taken’ if: the branch target buffer has a stored target address for your PC, you are a jump instruction that jumps to a target address equal to the NPC and a branch displacement stored in the instruction bits (BSR and BR essentially), or you are a return instruction and the RAS has a stored return address for you.
Branch Predictors

Predictor A is a bimodal predictor. Our version uses a 256 entry PC indexed. The bimodal predictor is rather simple in concept and implementation but had the highest prediction accuracy. For that reason it is the actual predictor used in our submission.
Predictor B is a GAg predictor. An 8-bit global history register (GHR) is used to index a table of the same size with two bit saturating counters. The GHR is a shift register that is updated by every branch.
Predictor C is a two level SAg predictor that exploits self-correlation. The lower bits of the PC are used to index a table of 256 entries, each of which is a 8-bit shift register. The value of the shift register is used to index a table to 256 2-bit saturating counters.
The branch target buffer works in tandem with branch prediction. Even if a branch predictor predicts taken for a given branch, the time to calculate the target address is too long to be done in one cycle and so the BTB is needed for all branch instructions except for BSR and BR. Every time a target address is calculated in the execute stage of the pipeline, the target address is stored so it can be used the next time the program hits the branch again.

Return Address Stack (RAS)

The return address stack is used for predicting the target address of return instructions. The assumption is that all JSR, BSR, and JMP instructions (call instructions) are paired with a relevant return instruction that will be executed in the future. Also it is assumed that return instruction will return to the most recent call instruction. To try to maintain accuracy in the event of deep recursion, a counter is used to keep track of the number of unreturned calls. In the fetch decoding logic (called Branch Decode in our pipeline) there are also counters for the number of unretired call and return instructions. This is because mispredicted path instructions will push and pop the stack before we realize they were mispredicted. Branch Decode watches all retiring instructions and checks for retiring jump and return instructions. This way it can keep track of which calls and returns were actually supposed to execute. Any jump instruction not paired with a return instruction will break the return address stack and it will likely no longer pop a correct PC (since its stack pointer is off) for remainder of the program execution. For the same reason, the RAS breaks if you run more call instructions (without returns) than the size of the stack. It will fail to perform until the number of entries returns below its stack size.

Modules



Instruction Cache

The instruction cache is a 128 line direct-mapped cache. Each cycle the icache accepts the current PC value and returns the instruction at that PC and PC + 4. The icache determines if the instructions are on two different cache lines and returns both instructions at the same time. Because it is direct-mapped, requests that are hits in the cache are trivial. On a miss, a state machine takes over. Each cycle, the icache determines if the cache line is in the cache. This is necessary because the data could have come back from memory on any cycle if it was prefetched or requested earlier for some reason. If the address is still a miss, the icache requests each line it needs from memory. It then waits for both lines in the cache to be hits and outputs the instructions to fetch. The icache also blocks any incoming stores from memory if they would erase a line that is currently needed by fetch. Since we do not allow writes to memory where instructions reside, no evictions are sent back to memory. Any conflicting cache line is just overwritten.

Instruction Prefetching

As noted in the advanced feature section, our instruction prefetcher requests addresses starting at the current PC and goes until it receives a new starting PC from a branch. The prefetcher does not actually start at the cache line address where the instruction from the first PC is located. Icache will handle making the first request to memory (if need be). The prefetcher starts at the line after the line needed for the two instructions at the start. It then checks to see if the cache line is already in the cache. If not, it sends a request to memory and adds eight to its PC counter. Then it repeats this process. When a branch is confirmed, it starts again from the new PC. The instruction prefetcher has a lower priority to memory than both the icache and dcache.

Fetch

The fetch module retrieves instructions at PC and PC+4 from the instruction cache. If instruction 1 is valid but instruction 2 is invalid, PC increments by 4. If both instructions are valid, PC increments by 8. If only instruction 2 is valid or a stall signal is received, the PC stalls. Branch signals can also set the PC to a specific value.

Decode

Decode takes in two instructions from fetch and the branch prediction data (prediction and target address). It outputs necessary decoded information to the Map Table, the ROB, and the RS.

Branch Decode, Branch Prediction, Branch Target Buffer, RAS

The Branch Decode module which is a fast decoder for fetch to speculate the next program counter (NPC) to use next cycle. This module houses the RAS and takes in prediction data from the Branch Prediction for the two instructions it is decoding. Branch decode checks for a branch instruction in either of the two instructions. If the first instruction is a branch and is predicted taken, branch decode will output a next PC based on the type of branch. If it is a return instruction it uses the RAS, if it is a non-BSR/BR it uses the BTB, and otherwise it calculates the address from the NPC and immediate displacement. The second instruction is invalidated. Repeat this process if instruction two is a branch. If neither instruction is a branch, simply increment the PC.

Both the Branch Prediction module and Branch Target Buffer modules are updated from execute. Branch “taken” or “not taken” and target address data are sent to the Branch Prediction module so it can update its branch history data and to the BTB so it can store the target address.

Branch history is not changed in any way on an exception.

The Return Address Stack is also a simple array of stored PCs that behaves like a stack. RAS keeps track of its maximum occupancy and how many entries were used throughout the execution of its program. It also keeps track of the number pushes past the maximum size of the RAS so the RAS can recover from overflow if it receives more calls than it has entries. On an exception, the stack pointer pointer moves to the correct position based on the number of unretired returns and calls that were added to the RAS.

Free Register List

The Free Register List module is a circular queue of physical register tags. Its head and tail pointers keep track of where the free and taken portions of the queue are. After the tail and before the head is taken, and after the head and before tail is free. Removing registers from the free list increments the head pointer and adding registers back to the free list increments the tail pointer. On an exception, the head pointer is set to the tail pointer since there are always 32 free registers.

Ready Register List

Our ready list is a 64-bit bit-vector where a bit being high means the corresponding physical register is ready. The ready list is updated by new instructions from decode and by the CTB to see which instructions are completing. When an exception occurs, flushed instructions mark their destination registers as ready and mark their previous destinations as unready. This is done by having a ready list maintained by the head of the ROB and using the ready list head to update the main ready list on an exception.

Map Table

The Map Table module maps architectural registers to physical registers. If there are any RAW or WAW dependencies between the two instructions being decoded the destination physical register of the first instruction is forwarded to the second instruction as its previous destination physical register. When an exception occurs, the map table sets its values to those in the map table maintained by the head of the ROB. This returns the mappings to the state to where they were before mispredicted instructions were executed.

ROB

The Can Opener's re-order buffer is a moderately complex module that features a 4-state finite state

machine. In its normal operation, the ROB takes in one or two instructions from decode, and functions just like a typical ROB from the MIPS R10K implementation. When a branch instruction is completed from execute, it comes with an exception bit as well as a target address. If there is a valid exception (meaning the branch was mispredicted), the ROB will store it in the appropriate line along with the new target address. Once that instruction reaches the head of the ROB, the ROB sets its next state to exception clear and retires the exception. In the exception clear state, the ROB outputs a full flush signal to all other modules, clears all of its lines of all data, and resets its head and tail pointers. This is accomplished in one cycle, unlike the original MIPS R10K implementation, because there are modules at the head of the ROB for the map table, ready list, and free list which track the state of retired instructions. Therefore, when the ROB outputs a full flush, the standard map table, ready list, and free list are set to the state of the their "head" versions and the processor can proceed as normal. Lastly, when a halt reaches the head of the ROB, it enters the halt state, in which is simply outputs a halt bit and returns to the halt state.

Reservation Station

Our reservation station has 16 general purpose entries. Any entry can be sent to any functional unit. New instructions from decode are stored in the first two empty RS lines. Each line is only responsible for determining when all of the registers it's waiting on are ready. Once it is ready to go, it signals that it's ready and a priority selector selects two lines that are ready. These two lines are put on to one of two bus lines using tri-states. The most interesting aspect of our RS is how we determine which instructions should be selected from among those that are ready. We keep a counter that increments based on how many instructions are extracted from the RS. The value of the counter is the current highest priority of the RS lines. When the counter reaches the end, it wraps back to the beginning. This tries to keep an instruction from getting buried at the bottom of the RS and stalling the ROB from retiring any instructions until the ROB is full and the RS has to empty.

Execute

Execute is a very simple module. There is a pipeline register before the functional units that clocks in the instructions from the RS along with their data from the physical register file. The outputs from that pipeline register are sent to the functional units. There are two of each alu, mult, load, and store functional units. The alu unit decodes branches and determines if they were mispredicted or not. The multiplier is an eight stage multiplier. Each stage stores the additional information about the specific instruction that is in that stage. The load and store units just calculate addresses and pass through the proper data. Each functional unit has its own CTB (common tag bus) to use. This allows us to avoid having to deal with what happens if a multiply completes on the same cycle as another functional unit.

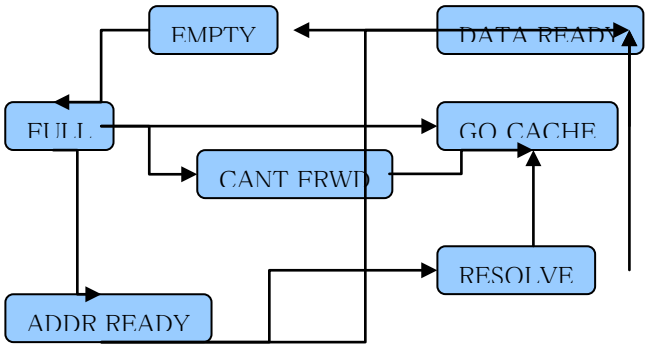
Physical Register File

The physical register file or PRF is a 64-entry array of 64 bits that stores the data of the physical registers. It has 4 read ports (for registers A and B for each of the two instructions in decode) and 6 write ports (one for each FU in the pipeline so they can all update the PRF individually and never conflict with each other). It also has read-to-write to handle a read and a write to the same physical register on the same cycle.

Load-Store Queue

The Can Opener's load-store queue is one of the most complex modules in the processor, largely due to the optional feature of store to load forwarding. It takes in inputs from decode, execute, the ROB, the data cache, and memory, and outputs to execute, the RS, the ROB, and the cache. The top-level LSQ module (lsq.v) is just a simple instruction router and the instantiation of the separate load and store queues. The combinational logic in lsq.v simply checks instructions coming from to decode to see if

they are loads or stores, and then sends them to the appropriate queues.
The load queue (lq.v) consists primarily of insert/extract wires for the load lines, two priority selectors, and the instantiation of the load lines. The combinational logic is fairly simple, and involves setting the enables for each load line based upon the results from the priority selectors, and letting the top level module know if the load queue is full. There are also several wires that travel to all the load lines which are direct inputs from either the store queue or main memory. The load queue always places new load instructions in the lowest numbered available load lines, and extracts the first one or two that are ready to go to the data cache or to retire into the physical register file.
The store line modules (store_line.v) are also fairly simple, as they only need to hold their instruction, ROB tag, address, and data until they are retired from the ROB and sent to the data cache. The only other function that occurs within the store line is checking the addresses of incoming loads completing from execute to see if they match. If the store line has the same address as a newly completed load, the store line will output a valid match bit so the store queue knows to forward that store line's data on to the corresponding load line. This part of the logic is handled separately in the store queue, as more than one store line could have a matching address, and the correct one must be selected for forwarding.
The store queue module (sq.v) is where the complexity of the LSQ's optional features begins to appear. The store queue is very similar in function to the ROB, as the store lines are filled and released in order, and uses a head and tail pointer system to insert and release lines appropriately. In addition to the normal head and tail pointers, there is also a “retire_head” pointer, which is used to indicate the “real” head of the store queue, because if the data cache isn't available, stores can retire from the ROB but remain in the store queue. The actual “head” pointer is then used primarily for flushing due to a precise exception in the ROB. When the LSQ receives a full flush signal, the store lines clear if they are in use but not retired, and the tail pointer of the store queue returns back to the head pointer. However, the retire_head pointer could still be elsewhere if there are store lines that have retired but are waiting to be sent to the data cache.



The load line module (load_line.v – FSM pictured) is very complex and was responsible for a large percentage of the debugging time spent on the project. Each load line is 7-state finite state machine that accounts for forwarded data as well as the ability to issue out-of-order.
When a load enters a load line, it takes in the head and tail pointers from the store queue, as well as the number of lines taken. The lines_taken register is used to determine how many stores must retire before the load line knows there are none left in the store queue with potentially valid data to forward. In addition to the pointers, the load line also receives a status array of the store lines when an instruction enters. This is also discussed in more detail under the optional features section, but essentially, when this array is all zeroes, that indicates that are no remaining stores in the store queue with an unresolved address. Once the data comes from the cache or has been forwarded, the load line waits to be completed by the load queue (and marked as such in the ROB) before it clears its registers and becomes available again.

Data Cache

The Can Opener uses a direct-mapped 128-entry write-back data cache as its gateway to memory. The data cache can receive up to five inputs per cycle: two loads, two stores, and a data return from memory. While in its default idle state, the cache outputs an available signal, and will only set this signal low if it has a cache miss or needs to evict data to memory because a dirty cache block is being

overwritten.
When the cache receives inputs from the LSQ, it checks to see if the loads are a cache hit and if the stores are writing to blocks that are already dirty. If there is a cache miss or a write to a dirty block, the cache enters a resolution state in which is sends all of its memory requests and continues to output them until it is granted memory access by the arbiter. Because the processor is superscalar, it is possible for one of the two loads to be a hit, in which case only one request would go to memory and the other would simply return the data to the LSQ. Once the cache has sent all of its evictions or load requests to memory, it returns to its idle state and becomes available again. When data returns from memory, it checks the corresponding cache block, and overwrites what is in it only if the block has data from a different address. If the cache line is dirty, the cache sends a priority request to the arbiter (which guarantees a grant) and output the dirty data to the memory on that same cycle. It has to do this because another load could return from memory on the next cycle and also have to evict data, and there is no buffer for those loads to queue in.
When the ROB outputs its halt signal at the end of a program, the data cache cycles through all 128 cache lines, checks to see if they are dirty, and then writes them as a priority request through the arbiter back to memory. When this process has finished, the cache then outputs its halt_finished bit to the pipeline, which is what actually halts the processor.

Testing

Pipeline Testbench (testbench.v)
The primary testbench is used to simulate and test the functionality of the entire pipeline. This testbench prints the contents of memory, calculates the CPI, and finds the total execution time. In order to test and debug the Can Opener, it also prints the contents of the architectural register, the state of the Re-Order Buffer (ROB), and the number of mispredicted branches. By comparing these to the correct outputs from Project 3, we can easily test the pipeline for correctness. The testbench also prints the contents of the ROB every cycle to a separate file used for debugging.

Testbenches for modules (RS, ROB, etc.)
We implemented testbenches for several individual modules to confirm their functionality. Each module has to behave correctly before it is implemented in the pipeline. Testbenches are available for most of the important modules; RS, ROB, LSQ, ready list, free list, map table, the branch predictor, execute, and priority selector. All of these modules' functionality can be confirmed by these testbenches. Users can assign inputs and expected outputs for every clock cycle, and testbench will compare the actual output to the expected outputs and print error messages. The error message contains the clock cycle, expected output and actual output.

Testing Strategy
The pipeline testbench prints useful information for debugging. By examining the output files, the location of a bug is often easy to reveal. Once we know the general location or cycle count of an error, we used Virsim to track its progress in the pipeline and resolve the problem behind it. This process worked for the individual modules as well as the pipeline. Most of the debugging time on the project was spent on confirming the proper operation of the more complex finite state machines in the ROB, Data Cache, and load lines. If a test program didn't halt properly by the time the pipeline was implemented in its entirety, there was a good chance the problem was in the next state or pointer logic within one of those three modules.

Evaluation / Analysis

(See graphs at end)

Test Cases

In the simulated version of the Can Opener pipeline, all of the provided test cases, as well every case we wrote, pass correctly. With a clock period of 8 ns, all of our test cases also pass in synthesis. We are still trying to find our absolute optimal clock period, which could be as low as 7.2 ns.

Can Opener CPI Improvements

Throughout all test cases, the Can Opener processor improved upon the CPI of the Versimple pipeline with forwarding by 19% on average. This number is skewed to be lower than it could be as a few of the test cases (saxpy, ld_st8, btest1, btest2) suffer heavily from having an 8-stage multiplier as well as memory latency issues. The following graph at the bottom of the report, which shows the latency improvement over Verisimple with forwarding, is a much cleaner picture of the Can Opener's performance increase.

In addition to the CPI improvements, we included graphs of how the performance of the Can Opener differs depending on what modules are included, activated, or altered. There is a convincing increase in performance with instruction prefetching, while some modifications show little or no improvement at all. Our typical number of ROB entries, 32, is clearly a superior choice to a smaller ROB, which is shown in the ROB CPI penalty graph. This is due to the superscalar nature of the pipeline, as a 16 entry ROB would only be able to accept 8 cycles worth of instructions before becoming full. Another notable fact is that increasing the number of load lines in the load queue from 8 to 16 rewards literally zero performance increase.

Branch Predictor Comparison and Desired Improvements:

Of the three dynamic branch predictors implemented, the local branch predictor or PC-indexed bimodal predictor outperformed its counterparts on a majority of the test cases. This is attributed to three main reasons.

1. The Local Predictor has the shortest wind-up time of all the predictors since once a branch is touched it is remembered as 'taken' or 'not taken' and many of the test cases were short and favored branch predictors that could set up quickly.
2. The Global and Correlated Branch Predictors didn't reset their branch history on exceptions. Both these predictors rely on correlation of a branching pattern history. This history becomes invalid if it is updated by a mispredicted instruction and not undone later on the exception flush on the pipeline.
3. The Global and to a much less degree the Correlated Branch Predictors required that no branches be "in-flight" while predicting for a current branch. The Global Branch Predictor makes a prediction based on the history of all past branches and if some past branches haven't finished updating the Global History Register than the prediction is inaccurate. This is only partially true for the Correlated predictor because the branch pattern history is PC-indexed and only applies to branches still in-flight with the same PC index.

The other branch predictors were not implemented because priorities went to other unfinished features of the pipeline. We would have liked to create a branch history database maintained by the ROB that would set the branch history database at Branch Decode on an exception so that branch pattern history is always consistent. Also, we would have liked to include speculative branch updating would so that in-flight branches will have meaningful predictions.

The branch histories for all 3 dynamic predictors are updated simultaneously for every branch instruction encountered no matter which branch predictor was used. Ideally we would implement a

tournament branch predictor out of the global and local branch predictor and then keep track of which predictor was correct at a certain PC, and have execute update accordingly.

Group Evaluation

Brad Campbell: 30%
Mike Metzger: 20%
Feng Li: 18%
Craig Yelton: 14%
Changhoon Yoon: 18%

Conclusions and Desired Additions

Data Cache Prefetching

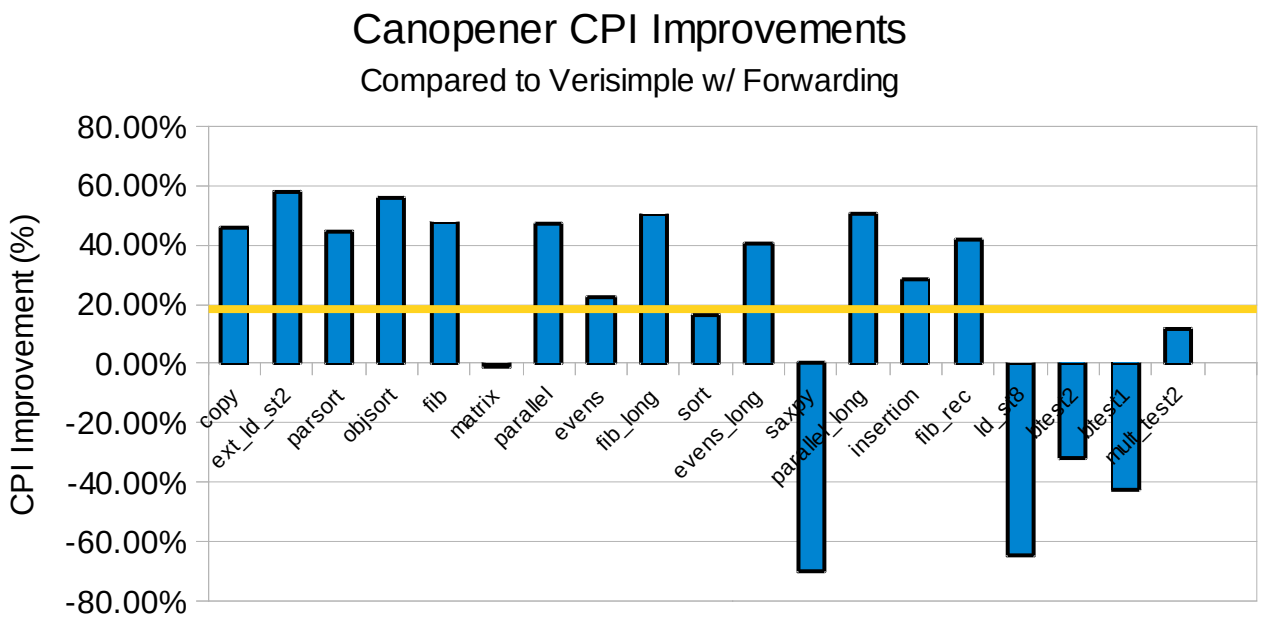
Often programs are walking arrays in memory. In these cases, many sequential memory accesses are made. We would have liked to add a simple dcache prefetching system to our dcache to help with the memory latency issue in these cases. Our plan was to request the next cache line on every access to the dcache, unless the address was already in the cache. This would've been rather simple to implement and should have given us a jump start on programs that walk arrays in memory, while not hurting other test cases significantly.

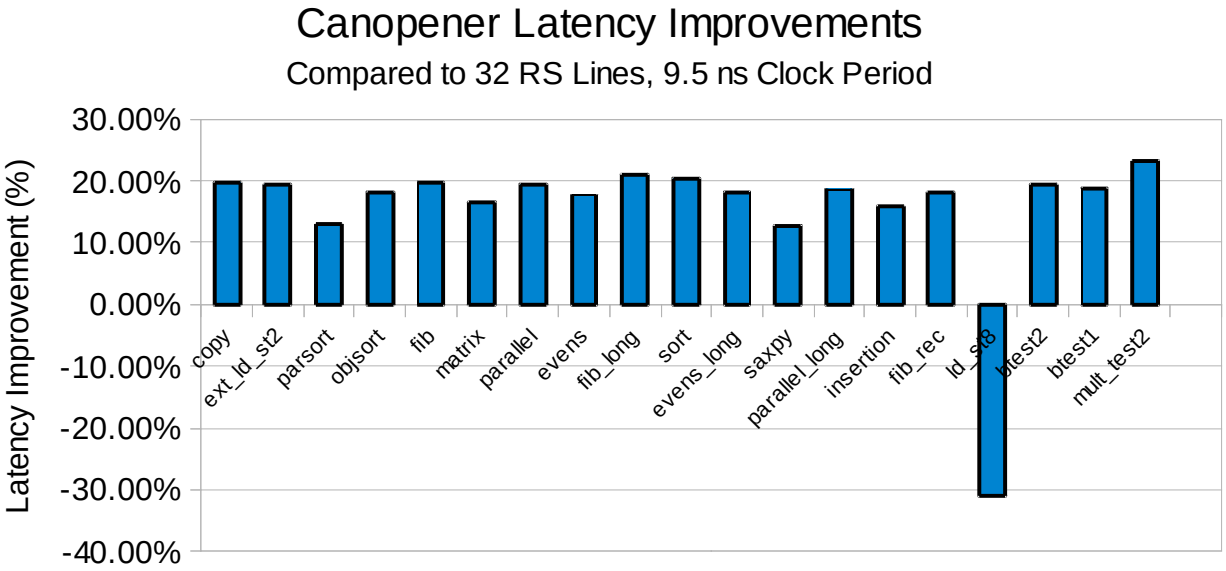
Better Priority Selector

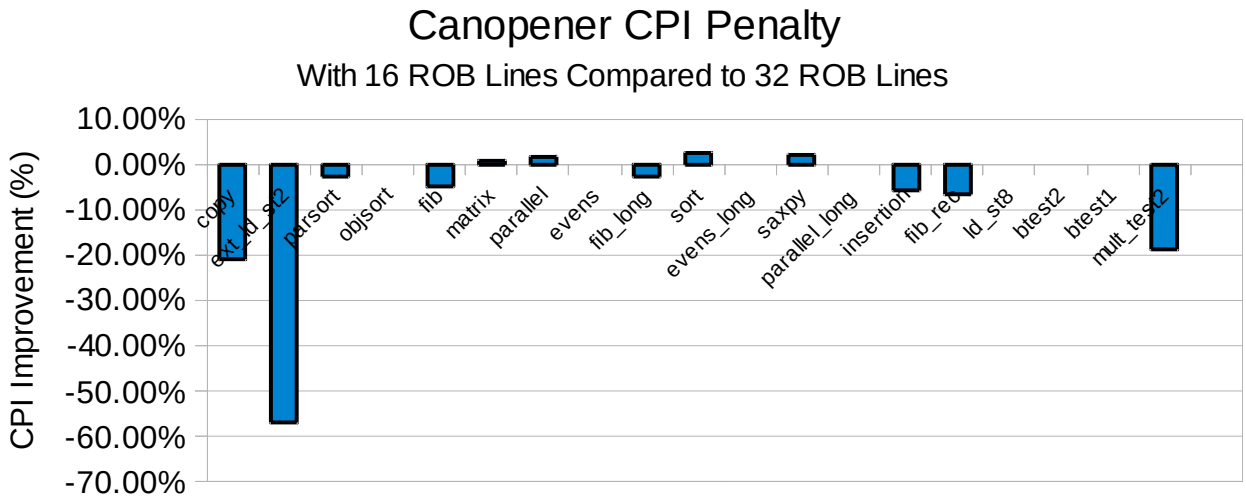
The critical path in our pipeline contains the priority selector that selects ready instructions from the RS. If we had a more efficient two-choice priority selector, we could have reduced our clock period without having to add another stage. Unfortunately, there are not established examples of multi select priority selectors we could reference and this non mission critical project was never extensively pursued.

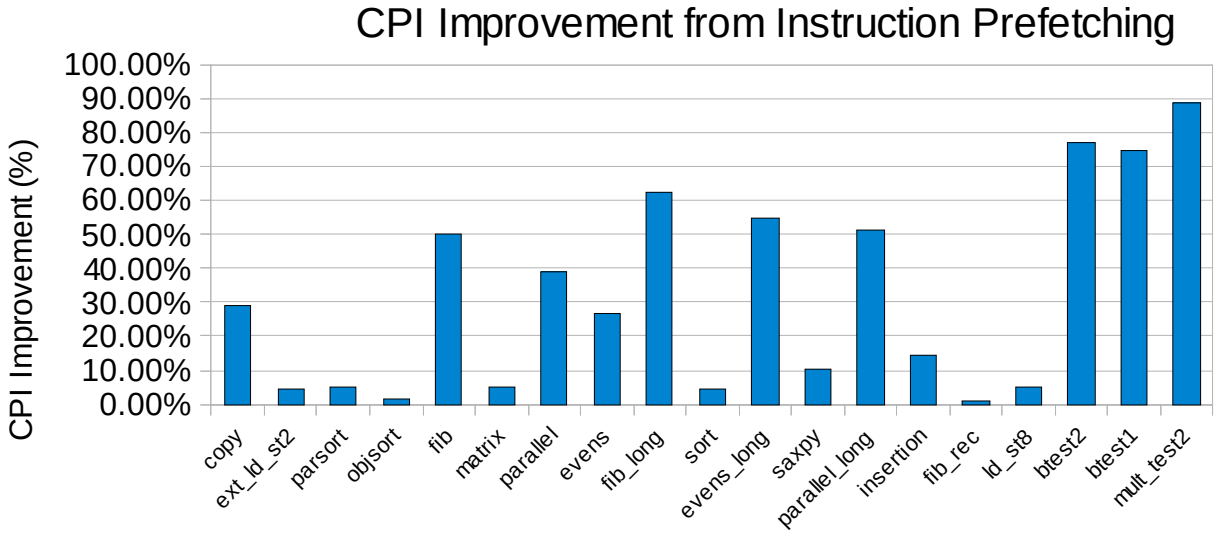
Things to do differently

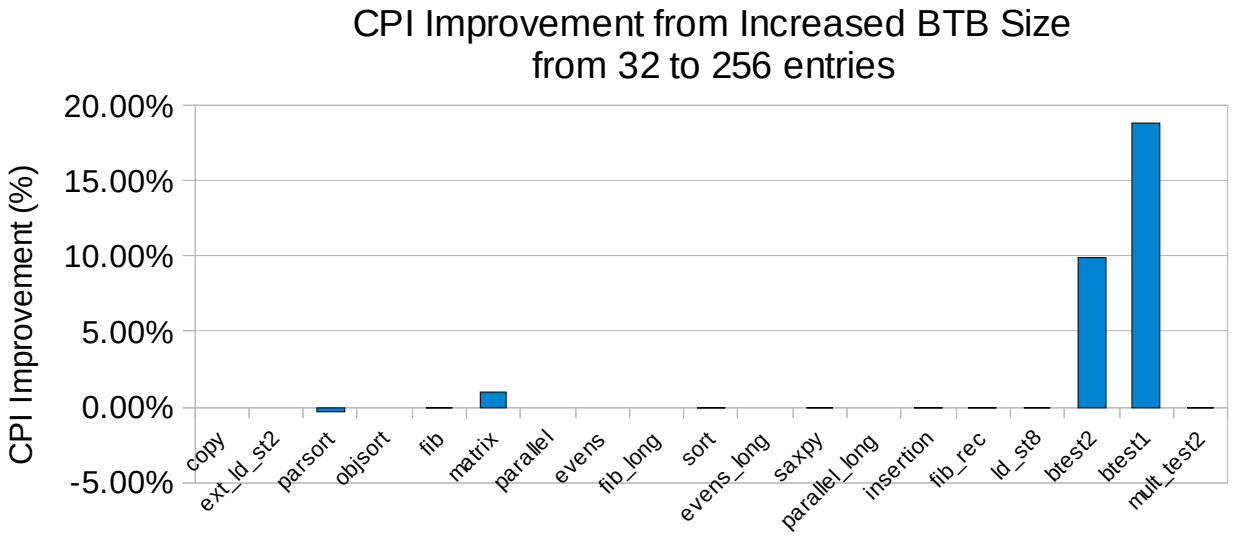
We would have spent more time early on figuring out the interconnections and dependencies between modules, worked on decreasing the clock period right after milestone 2, and synthesized individual modules in a hierarchy to reduce overall pipeline synthesis time.

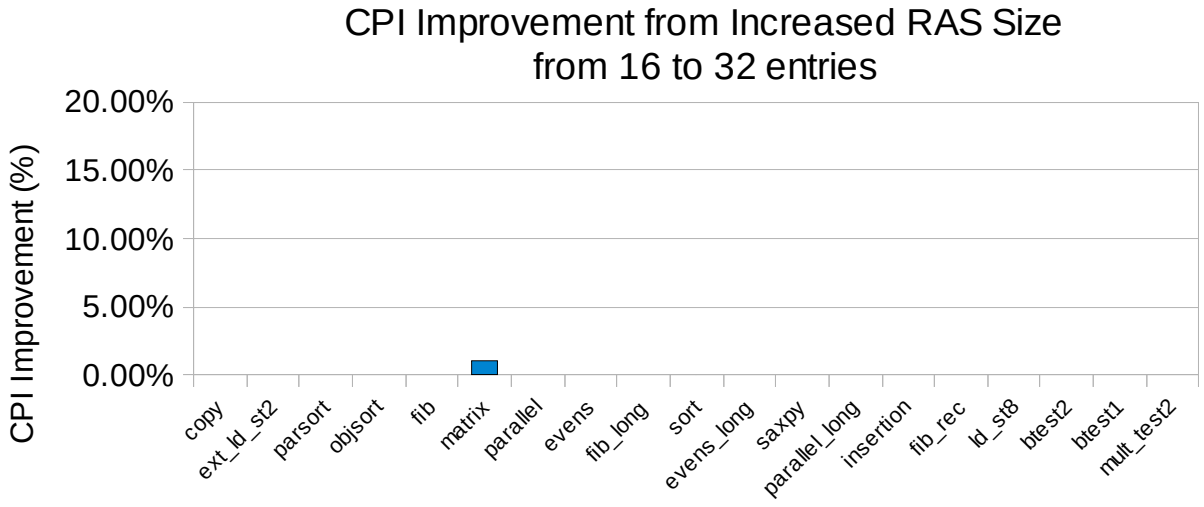

Canopener CPI Improvements
Compared to Verisimple w/ Forwarding
CPI Improvement (%)
80.00%
60.00%
40.00%
20.00%
0.00%
-20.00%
-40.00%
-60.00%
-80.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2

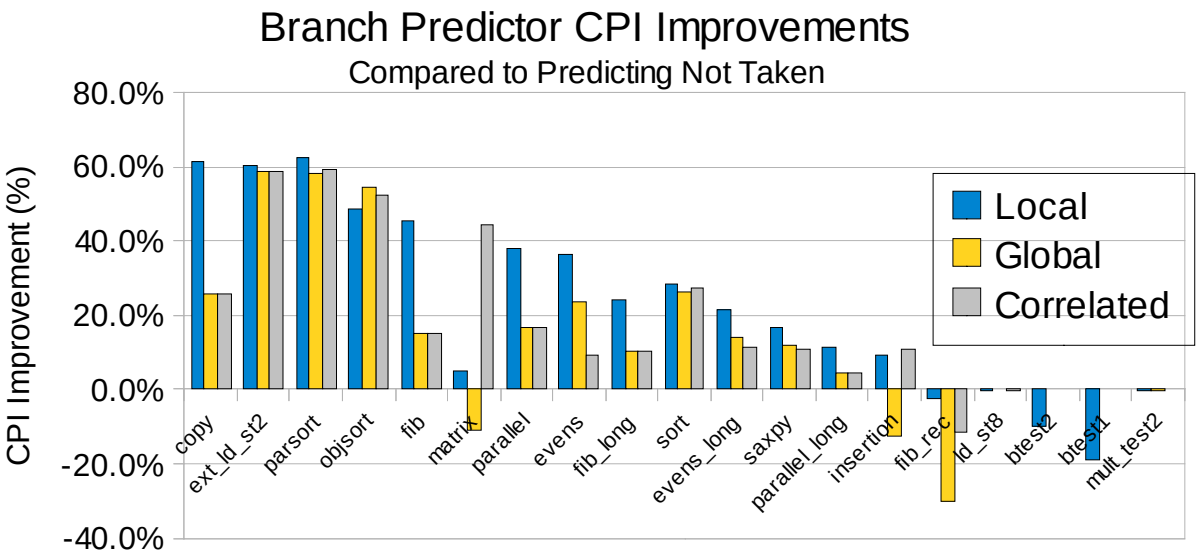

Canopener Latency Improvements
Compared to 32 RS Lines, 9.5 ns Clock Period
Latency Improvement (%)
30.00%
20.00%
10.00%
0.00%
-10.00%
-20.00%
-30.00%
-40.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2

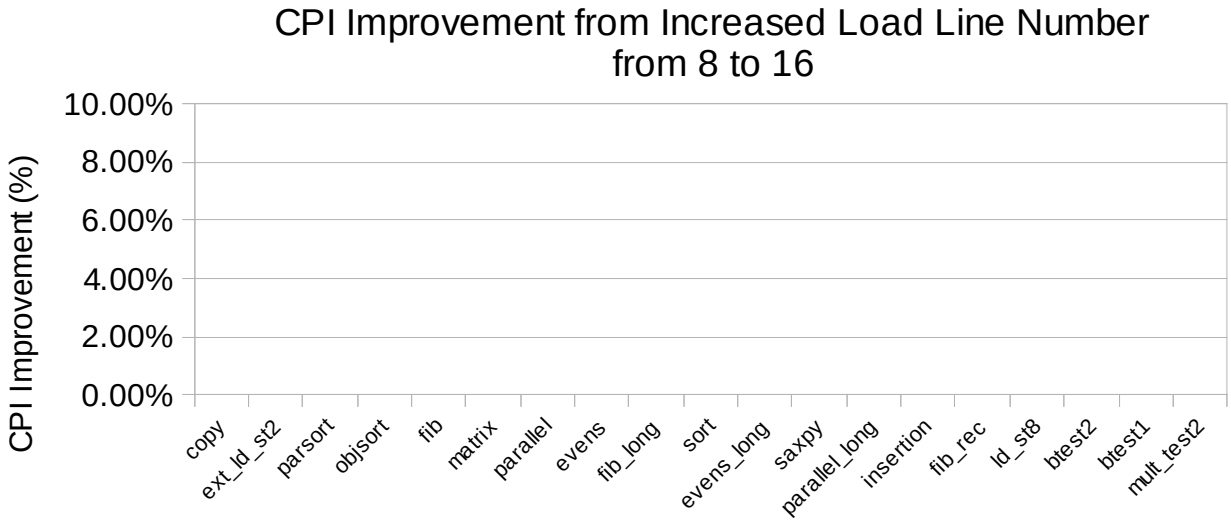

Canopener CPI Penalty
With 16 ROB Lines Compared to 32 ROB Lines
CPI Improvement (%)
10.00%
0.00%
-10.00%
-20.00%
-30.00%
-40.00%
-50.00%
-60.00%
-70.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2


CPI Improvement from Instruction Prefetching
CPI Improvement (%)
100.00%
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2


CPI Improvement from Increased BTB Size
from 32 to 256 entries
CPI Improvement (%)
20.00%
15.00%
10.00%
5.00%
0.00%
-5.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2


CPI Improvement from Increased RAS Size
from 16 to 32 entries
CPI Improvement (%)
20.00%
15.00%
10.00%
5.00%
0.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2


Branch Predictor CPI Improvements
Compared to Predicting Not Taken
CPI Improvement (%)
80.0%
60.0%
40.0%
20.0%
0.0%
-20.0%
-40.0%
Local
Global
Correlated
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2


CPI Improvement from Increased Load Line Number from 8 to 16
CPI Improvement (%)
10.00%
8.00%
6.00%
4.00%
2.00%
0.00%
copy
ext_ld_st2
parsort
objsort
fib
matrix
parallel
evens
fib_long
sort
evens_long
saxpy
parallel_long
insertion
fib_rec
ld_st8
btest2
btest1
mult_test2