Group 1: 6.375 Final Project
Runahead Processor

Finale Doshi and Ravi Palakodety

May 16, 2006

1 Introduction

Data cache misses from can severely affect processor throughput if the processor stalls until valid data becomes available. A runahead processor attempts to minimize the effects of data cache misses by prefetching data needed by future instructions following a cache miss. The processor continues to execute instructions after the cache miss using invalid data until the requested data becomes available[1][2]. These prefetches are likely to be accurate, as the instructions would have been executed anyway (assuming no branches). The longer the processor is in runahead mode, the more runahead instructions that execute, and the more prefetches that are called—reducing future cache misses.

While entering runahead mode is fairly straightforward—the processor continues to execute instructions normally—we must take care in restoring the processor's state once the original miss returns. The following steps lead us through entering and exiting runahead, pointing out the key factors that ensure that we exit runahead correctly:

1. A load or store instruction causes a data cache miss. Runahead execution begins.
2. The processor checkpoints the current state by making a backup copy of the register file and program counter.
3. The processor continues to execute instructions using an invalid value for the pending data cache request.
4. Future loads and stores may cause data cache misses; the cache also prefetches these misses.
5. Writes to the data cache do not occur during runahead execution. Writes to the register file that depend on an invalid value are marked invalid in the register file. Computations that depend on invalid register entries are also marked invalid. Loads and stores that depend on invalid addresses are not prefetched.
6. Runahead execution proceeds until the original data cache miss is fetched from memory.

7. We copy the backup register file into the real register file, and proceed from the checkpointed instruction.

In this project, we implemented a runahead processor and analyzed the effects of memory latencies and inter-module fifo lengths on its performance. We also explored variants on leaving runahead and caching stores.

2 High Level Design

Figure 1 shows a high-level cloud-diagram of the runahead processor. The three main processor rules—`pc-gen`, `exec`, and `writeback`—perform essentially the same function as our familiar three-stage processor: `pc-gen` updates the program counter and requests the next instruction; `exec` decodes the instruction, performs ALU operations, and sends requests to the data cache; and `writeback` writes ALU ops and data memory responses into the register file.

Some of `exec`'s and `writeback`'s operations are tailored to the processor. For example, `exec` will not send load requests with invalid addresses to the data cache. The `writeback` rule is responsible for notifying the processor of when to enter runahead, and the `check-response-q` rule notifies the processor when to exit runahead. To reduce clutter, we do not show the `stall` and `discard` rules for handling read-after-write hazards and clearing of the `pcQ` after taken branches.

The `stop-`, `start-`, and `stall-runahead` rules are mutually exclusive with the three normal 'processing' rules (and indeed with each other). Whenever the mode changes, these rules ensure that the processor's state is correctly backed-up and restored. The `stall-runahead` rule is responsible for stalling the processor if a runahead branch depends on an invalid predicate (and, as we do design exploration, for any other situation where the processor must stall due to invalid data).

Within the cache, all rules are mutually exclusive. The `main` rule sets the mode for each rule to fire. Responses from the main memory are treated as most urgent; the `refill-resp` rule takes data from main memory and updates the cache. `access` responds to requests from the processor and sends requests to the main memory. The `refill-req` rule fires only when we have a collision with a dirty cache-line, and the original data needs to be stored back to main memory before the current request is made.

3 Testing Strategy

Our first goal was to ensure correctness of both the processor and the cache. In addition to asm-tests and the benchmarks, we wrote a test of load-store scenarios on valid addresses (we do not perform loads and store on invalid addresses). Listed below are the tests and their expected (and observed) outcomes.

- **Load-a : hit.** Do not enter runahead and requests handled normally.

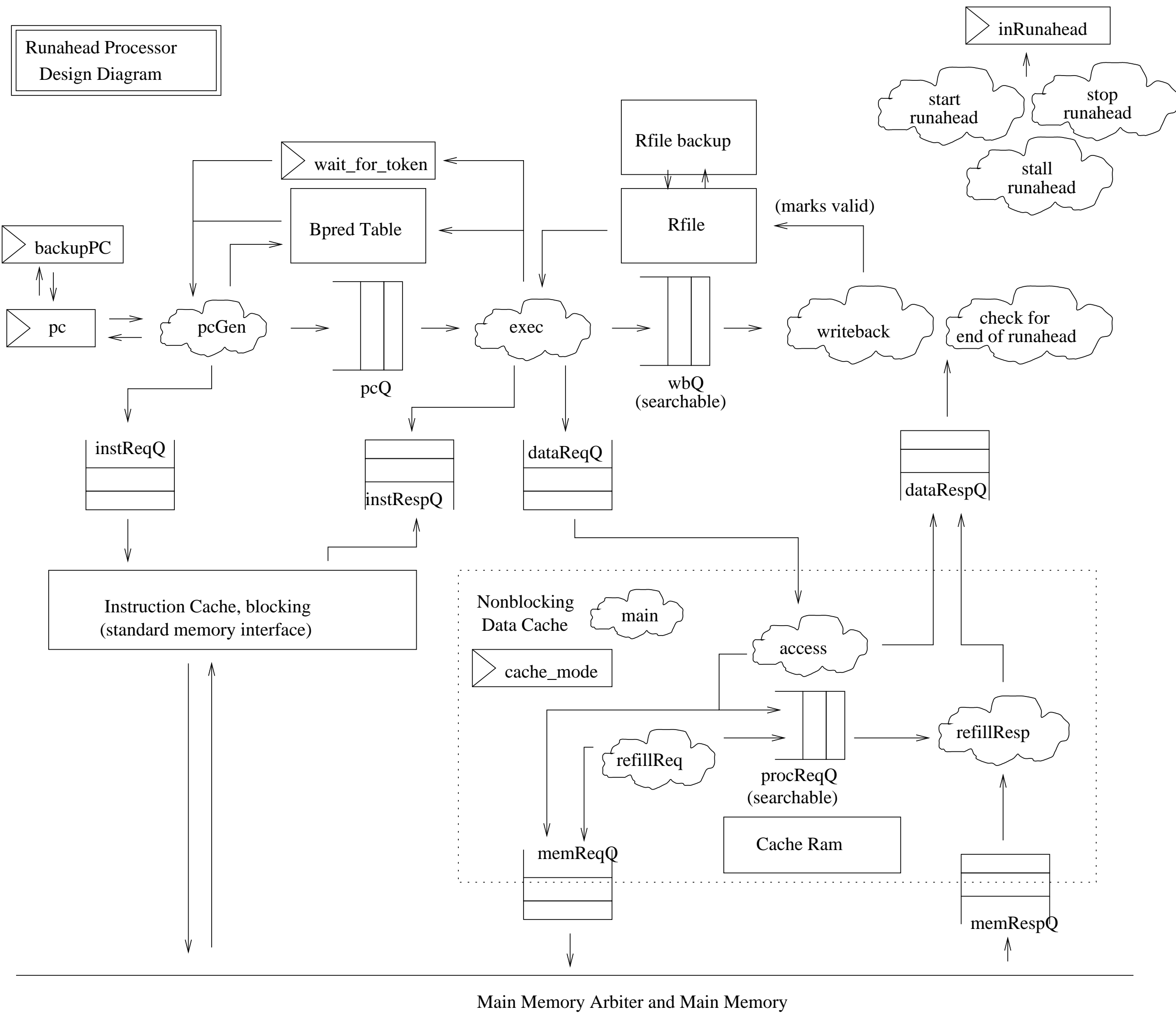

Figure 1: Cloud Diagram of Runahead Processor

- **Load-a : miss.** If requested cache-line is valid and dirty, then the value currently stored in that cache-line is written to memory. Enter runahead and return 'req-missed.'
- **Load-a, Load-a : both miss.** Enter runahead after the first load but do not initiate the second load request. Both loads return to the processor (in runahead mode) as 'req-missed.'
- **Load-a, Load-b : both miss; b writes to same cache-line as a.** Enter runahead after the first load. In the baseline, we only initiate one request per cache-line. Thus we avoid the possibility of the first load being overwritten before the processor switches out of runahead mode which would lead to an infinite loop. Thus the second load is not prefetched. (Optimizations may adjust the timing to allow the second load to be prefetched for future operations.)
- **Store-a, Load-a : store misses.** Enter runahead on the store and send a prefetch request for that address; load does not generate an additional prefetch. An optimization will store the value of the data into the store cache for future runahead loads.
- **Load-a, Store-a, Load-a : first load misses.** Enter runahead on the first load; no additional prefetches. If optimized, store the data in the store cache. The second load takes the data from the store cache.
- **Store-a, Store-a, Load-a : first store misses.** Enter runahead on the first store; no additional prefetches. If optimized, store the value of the first store in the store-cache. The second store updates the store-cache—this is important because the validity of the data may have changed. The load takes its data from the store-cache.

The baseline processor does not attempt to optimize any of these operations. However, we varied latencies and FIFO sizes to ensure that all of the cache states were executed correctly (in particular, we wanted to test situations where a second miss to the same address arrived while the original miss was being processed). To avoid instruction cache misses, we wrapped our test in a for-loop that filled the instruction cache on the first iteration. We characterized performance on the second iteration.

Next, we evaluated the performance of our baseline runahead processor against the basic three-stage processor, runahead processor with a special store-cache, and a runahead processor that waited for all prefetches to return before completing runahead (see Section 5 for details on our design exploration). The `vvadd` benchmark contained many independent loads and relatively few intermediate operations—this was a case where our runahead processor should have outperformed the baseline.

The remaining benchmarks served as tests to see how the variants performed in more 'normal' situations. In each performance test, we measured the IPS, as well as area and total memory requests (Section 6). Since the instruction cache was a standard blocking cache, each benchmark was run on two sets of data. Statistics were recorded on the second iteration. Thus, the instruction cache was already pre-loaded but the data-cache missed on all data requests.

4 Baseline Microarchitecture

In this section we detail the processor and cache rules introduced in Section 2.

4.1 Runahead Processor

4.1.1 State

In addition to the `pc` register, register file, and `wait-for-token` register in the baseline three-stage processor, the runahead processor contains the following state elements:

- **Backup PC Register.** The `backup-pc` register stores the pc of the offending instruction that causes us to enter runahead.

- **Modified Writeback Queue.** Elements in the writeback queue `wbQ` contain up to four parts: (1) the instruction's pc (all requests), (2) the register address where data should be stored (load and alu-op requests), and (3) a data value to be written and (4) its validity (alu-op requests only). The pc is written to the backup pc register if the `writeback` rule determines that the instruction has caused the processor to enter runahead mode.

- **Restorable Register File with Valid Bits.** Instead of an array of data flip-flops, the register file is now an array of three sets of flip-flops: data, backup data, and valid bit. rf.backup() and rf.restore() methods copy data between the data and backup registers. Note that each backup flip-flop only needs to read and write to one data register. Thus, each data register can be floorplanned next to its 'backup buddy,' so checkpoints and restores will be completed in one cycle (instead of copying words one at a time to a separate register file.)

- **Mode Register.** The processor can be in one of five modes: normal-processing, starting-runahead, runahead-processing, stopping-runahead, and runahead-stalling.

- **Branch Predictor Table.** Although not the focus of our project, our baseline design includes a simple branch predictor table with four { current address, next address } entries. A branch is predicted taken if the current address is in the table.

4.1.2 Rules

Modes and Conflicts The mode determines what rules can fire:

- Normal-Processing: `pc-gen`, `exec`, `writeback` (sequentially composable; `writeback` $<$ `exec` $<$ `pc-gen` to ensure proper pipelining)

- Starting-Runahead: `start-runahead` only.

- Runahead-Processing: `pc-gen`, `exec`, `writeback`, `check-response-q`; `check-response-q` and `writeback` are mutually exclusive.

Table 1: State Element usage by Processor Rules; R indicates read, W indicates write

Rule/State Element	Register File	`pc-register`	`backupPC-register`	`mode-register`	`pcQ`	`wbQ`	`instReqQ`	`instRespQ`	`dataReqQ`	`dataRespQ`
`pc-gen`		R/W		R	enq		enq			
`exec`	R	W		R/W	deq	enq,R		deq	enq	
`writeback`	W		W	R/W		deq				deq
`check-response-q`				R/W						deq
`start-runahead`	W			R/W						
`stall-runahead`				R/W						
`stop-runahead`	W	deq		R/W		clr		deq	enq	

- Stopping-Runahead: `stop-runahead` only.
- Runahead-Stalling: `stall-runahead` and `check-response-q`.

Table 1 shows the methods called by each rule.

Detailed Rule Descriptions We first have the standard three rules: `pc-gen`, `exec`, and `writeback`. The `pc-gen` rule sends a data request to the instruction cache, increments the pc, and places it on the `pcQ` to the `exec` rule. We use tokens to keep track of taken branches, but they are not shown here for clarity reasons. We also have `stall` and `discard` rules for handling read-after-write hazards and clearing of the `pcQ` after taken branches. As these were unmodified from the standard three-stage processor, they are not presented here.

```
rule pc-gen( mode_proc == NormalProcessing );
    instReqQ.enq( pc );
    pc_plus4 = pc + 4;
    pc_pred = bpred.getPred( pc );
    if( pc_pred.isValid == true )
        begin
        pcreg <= pc_pred;
        pcQ.enq( pc_pred );
        end
    else
        begin
        pcreg <= pc_plus4;
```

```
        pcQ.enq( pc_plus4 );
        end
endrule
```

The `exec` rule takes the instruction response from the instruction cache, and executes the rule. For example, ALU operations are evaluated, with the results placed in the `wbQ`. Loads and stores initiate data requests to the non-blocking data cache, and branches are resolved in this rule. The runahead modifications involve the addition of a valid bit to the register file. We maintain the following invariants:

Invariant. *During normal-processing mode, all entries in the register file are valid (During runahead-processing mode, some entries in the register file may contain bogus data.)*

Invariant. *The processor may never change memory values during runahead-processing mode—it may only swap data between the cache and main memory.*

Invariant. *The processor may request loads and stores to addresses if and only if all the values used the compute that address were marked valid. (Thus, even in runahead mode, the processor never prefetches data that it may not need.)*

Operations involving invalid data have special results. ALU operations involving invalid data have invalid results placed in the `wbQ`. Loads and stores whose addresses depend on invalid data do not initiate data cache requests. When the predicate for a branch involves invalid data, and we enter the runahead-stalling mode. Again, as the resolution of taken branches is identical to normal-processing mode.

Note: The pseudocode below includes many functions whose purpose should be obvious from their names; the actual code fully implements these functions.

```
rule exec( DontStall && ( mode_proc == runahead-processing ||
                          mode_proc == normal-processing );
    pcQ.deq(); instRespQ.deq();
    case ( instRespQ.first() ) matches
        tagged (LW or ST):
            if (addr.isValid()) dataReqQ.enq( DataReq( ... ) );
            wbQ.enq( WB_LD or WB_ST);
        tagged (ALU_OP):
            wbQ.enq( WB_ALU{ops.isValid(), dest, result} );
        tagged (BRANCH_OP):
            if (pred.isValid()) resolve_branch();
            else mode_proc <= runahead-stalling;

    checkBranchPrediction();
    updateBpredTable();
endrule
```

The `writeback` and `check-response-q` rules underwent the most significant changes in our runahead processor. The `writeback` rule in normal mode takes the first item off the `wbQ`. If the item is a `WB-ALU`, the data in the item is written to the appropriate destination in the register file. If the item is a `WB-LD` or `WB-ST`, the first item in the `wbQ` is matched to the first item in the `dataRespQ`. The appropriate action is then taken (`writeback` the data from the load response, and do nothing on a store response). Finally the rule removes the first element of the `wbQ`, and if that element was a store or load, removes the first element of the `dataRespQ` as well.

Data responses have a two major tags: `isRunahead`, and `isValid`. While in normal mode, the processor expects all data to come back with (`isRunahead == False`) and (`isValid == True`). This corresponds to a hit in the data cache. The `writeback` rule is the processor's check on whether a miss has occurred, in which case, the processor must enter runahead mode. This check allows us to maintain the following invariants:

Invariant. *The* `isRunahead`*-tag of all data responses matches the mode (normal or runahead) of the processor. As soon as a mismatch is found, the processor changes its mode.*

Invariant. *When the processor enters starting-runahead mode, all instructions prior to the offending load/store have been completely processed.*

Invariant. *During runahead execution, we never initiate an invalid prefetch (a prefetch that we may not need).*

Invariant. *When the processor enters stopping-runahead mode, all instructions in the pipeline are cleared and the offending load/store is the first to be executed.*

For example, if the processor is in normal-processing mode, and receives a data response that has (`isRunahead == True`), the processor needs to switch to starting-runahead mode. While in runahead-processing mode, all data responses will return immediately as either hits or invalid misses, with the tag (`isRunahead == True`). When the original offending miss returns from main memory, its response will be placed in the `dataRespQ` with a tag (`isRunahead == False`). Thus, the mismatch between the processor's mode and the `isRunahead`-data response will trigger a state change, switching the processor back to normal mode.

The check to switch from normal-processing mode to starting-runahead mode is placed in the `writeback` rule. A separate rule, `check-response-q` is used to check for switches from runahead-processing mode back to normal-processing mode (via stopping-runahead). The `check-response-q` rule is forced to have greater urgency than `writeback` to ensure that we will exit runahead-processing as soon as the mode-changing response arrives.

Another important observation that is not quite obvious is:

Invariant. *The processor mode can only switch based on the first item of the* `dataRespQ`*.*

Thus, on the normal to runahead switch, all data that needs to be in the register file prior to backup is present.

```
rule check-response-q ( dataRespQ.first() matches DataResp .dr &&&
                    dr.isRunahead == False &&
                    ( mode_proc == runahead-processing ||
                      mode_proc == runahead-stalling ) );
     mode_proc <= stopping-runahead;
     dataRespQ.deq();
endrule

rule writeback( mode_proc == normal-processing || );
                mode_proc == runahead-processing );
    wbQ.deq();
    case (wbQ.first()) matches
        tagged WB_ALU:
            rf.wr(data, dest);
        tagged WB_LD:
            if (ld.isRunahead && !runaheadMode_proc)
                mode <= starting-runahead;
                backupPC <= wbQ.first().pc;
            else
                rf.wr(data, dest);
        tagged WB_ST:
            if (st.isRunahead && !runaheadMode_proc)
               mode <= starting-runahead;
               backupPC <= wbQ.first().pc;
endrule
```

Finally, we have three rules involved in the switching between normal/runahead modes. `start-runahead` backs up the register file (checkpointing), and then allows the processor to continue execution.

```
rule start-runahead( mode_proc == starting-runahead );
    mode_proc <= runahead-processing;
    regFile.backup();
endrule
```

The `stop-runahead` rule clears the `pcQ`, `instRespQ`, and `wbQ`. It also restores the register file. Clearing of the `pcQ` and `instRespQ` happens with 1-for-1 dequeues, to ensure correctness before starting normal mode. Finally, `exec` might issue a data request at the same time `check-response-q` is ending runahead. Thus, an extra memory request can leak onto the `dataReqQ` on the cache side. The `stop-runahead` rule issues an extra request to the data cache, instructing it to clear its `reqQ`. After these steps are completed, normal mode begins.

```
rule stop-runahead( mode_proc == stopping-runahead );
     - deq pcQ and instRespQ one for one until empty
```

```
      - clear wbQ
      - restore regFile and pc; all entries are valid
      - send ‘completed-restore’ token to data cache
endrule
```

stall-runahead is invoked when a branch involves an invalid predicate or if the program completes while in runahead mode. The stall-runahead rule basically waits for the initial miss to return and check-response-q to fire (ending runahead mode).

```
rule stall-runahead( mode_proc == runahead-stalling );
    mode_proc <= runahead-stalling;
endrule
```

4.2 Cache

4.2.1 State

We have built two non-blocking, direct-mapped caches: one specialized for the runahead processor, and one for the standard three-stage baseline modelled after Kroft [3] (so when we compare results, we do not compound the impact of a non-blocking cache with the impact of runahead processing). Both caches implement a major-minor FSM structure modelled after the provided blocking cache. In this section, we focus on the cache tailored to runahead operations. This cache contains the following state elements:

- **Cache RAM.** Stores the cache data, including a valid bits, dirty bits, and memory addresses.
- **State.** The cache has a simple FSM structure with each rule corresponding to a state. The states are `main`, `refill-resp`, `access`, `refill-req`, and `wait-for-token`.
- **Mode.** The mode register stores whether the cache is in runahead or normal mode.
- **ProcReqQ.** The `procReqQ` keeps track of all requests currently in flight between main memory and the cache. The queue has two uses: first, an `isInitialMiss` tag marks the offending request that caused us to enter runahead. Second, by searching the `procReqQ`, we avoid making repeated requests to the same address.
- **HaveSetInitialMiss.** Keeps track of whether we have seen an initial miss yet. This register is important because the `procReqQ` may still have prefetches in-flight when we exit runahead operation; when we get a miss we must check if it is the first offending miss.

Table 2 shows the methods called by each rule.

Table 2: State Element usage by Cache Rules; R indicates read, W indicates write

Rule/State Element	Cache RAM	`state-register`	`mode-register`	`haveSetInitialMiss`	`reqQ`	`respQ`	`procReqQ`	`memReqQ`	`memRespQ`
`main`		R/W			R				R
`access`	R	R/W	R/W	R/W	deq	enq	enq,R	enq	
`refill-req`		R/W		R/W	deq		enq	enq	
`refill-resp`	W	R/W	W	W		enq	deq		deq
`wait-for-token`		R/W			deq				

4.2.2 Rules

Since each rule corresponds to an FSM state, they are mutually exclusive. Rules never fire in parallel. The major FSM chooses which rule to enable, `access` or `refill-resp`, with `refill-resp` getting greater priority, if it is ready.

```
rule main( stage == Main )
    if (mainMainRespQ.notEmpty())  stage <= RefillResp;
    else if (reqQ.notEmpty())  stage <= Access;
    else stage <= Main;
endrule
```

The `access` rule determines whether a given request is a hit or a miss. The `access` rule enacts the following invariant:

Invariant. *All requests are given immediate responses. In the case of a miss, an immediate invalid response is given with a* `isRunahead == True` *tag. In the case of a hit, the* `isRunahead` *tag matches the mode of the cache. Note that immediate responses imply that responses are always returned in-order.*

The `access` rule uses an SFIFO called `procReqQ` to keep track of in-flight main memory requests. If a memory request is already in flight, we do not initialize a second one. Furthermore, the `procReqQ` keeps track of which memory request is the initial miss. That is, requests are enqueued onto the `procReqQ` with a tag stating whether it is an initial miss. This information will prove useful in the `refill-resp` rule. A register—`haveSetInitialMiss`—is used to decide whether a miss is an initial miss or not. Finally, dirty collisions require a two-step process where a store request is sent to main memory, writing back the dirty data, followed by a load request, to fill in the appropriate cache line. The second step is implemented in `refill-req`. Thus, another invariant is:

Invariant. *The* `procReqQ` *contains all prefetches in flight. In runahead mode, exactly one element in the* `procReqQ` *is marked with* `isInitialMiss`*. In normal mode, no elements are marked with* `isInitialMiss`*.*

```
rule access( stage == Access )
    if (isHit(req))
        RespQ.enq ( isRunahead = runaheadMode_cache;
                    isValid = True; value = value );
      stage <= Main;
    if (isMiss(req))
        RespQ.enq ( isRunahead = True;
                    isValid = False; value = xxx );
        runaheadMode_cache <= True;
      if (dirty collision)
         mainMemReq.enq( store current value to its address );
         procReqQ.enq( store-req );
         stage <= refill-req;
      else if ( req's cache line index is not currently in flight
                || is initial miss )
         mainMemReq.enq( req )
         if ( haveSetInitialMiss ) //Not an initial miss
                        procReqQ.enq( req )
                        haveSetInitialMiss <= True;
                        stage <= Main;
                 else
                        procReqQ.enq( req, isInitialMiss ) //An initial miss
                        haveSetInitialMiss <= True
                        stage <= Main;
endrule

rule refill-req( stage == refill-req );
    mainMemReq.enq( req )
    if ( haveSetInitialMiss )
          procReqQ.enq( req )
  haveSetInitialMiss <= True;
    else
          procReqQ.enq( req, isInitialMiss )
          haveSetInitialMiss <= True;
    stage <= Main;
endrule
```

The `refill-resp` rule handles responses from main memory. When the initial miss that caused runahead returns, `refill-resp` sends it to the processor `dataRespQ` with a

tag (`isRunahead = False`). This tells the processor to switch back to normal mode, from the rule `check-response-q` detailed above. Furthermore, the in-flight responses generated during runahead are loaded into the cache, without sending responses back to the processor.

In the case of an initial miss returning from main memory, part of the stop-runahead procedure requires overt clearing of the `reqQ`. This is done in the state `wait-for-token`, where the dummy request was detailed in the processor's stop-runahead procedure.

```
rule refill-resp( stage == refill-resp );
    mainMemRespQ.deq(); procReqQ.deq()
    - do nothing on a store response
    - if load response
         - update cache
         if (initialMiss(mainMemResp)
               respQ.enq( isRunahead: False , response);
               stage <= WaitForToken;
               runaheadMode <= False;
               haveSetInitialMiss <= False;
         else
               stage <= Main;
endrule

rule wait-for-token( stage == wait-for-token );
    reqQ.deq();
    if ( reqHasToken )
        stage <= Main;
    else
        stage <= WaitForToken;
endrule
```

4.3 Example Scenarios

The following diagrams show example processor-cache scenarios, illustrating how three tokens—`isInitialMiss` in the cache's `procReqQ`, `isRunahead` in the `respQ`, and `done-restore` in the `reqQ`—ensure that the entering and exiting runahead occur correctly.

Figure 2 shows what occurs when the cache receives the initial miss. The cache immediately enqueues a '`isRunahead`' response in the RespQ to the processor and memory request to the main memory. It also enqueues the request with an '`isInitialMiss`' tag into the `procReqQ`. Meanwhile, the processor is already 'running ahead' in the sense that it is sending more data requests—the cache will handle these with the standard runahead approach (see Figure 4).

Eventually, the '`isRunahead`' response reaches the front of the RespQ and the processor enters runahead mode. Note that the processor enters runahead mode after the cache since the RespQ decouples the two modules. Since future (runahead) cache responses are enqueued

after the initial '`isRunahead`' response, however, the responses are still received in the correct order and mode from the processor's perspective.

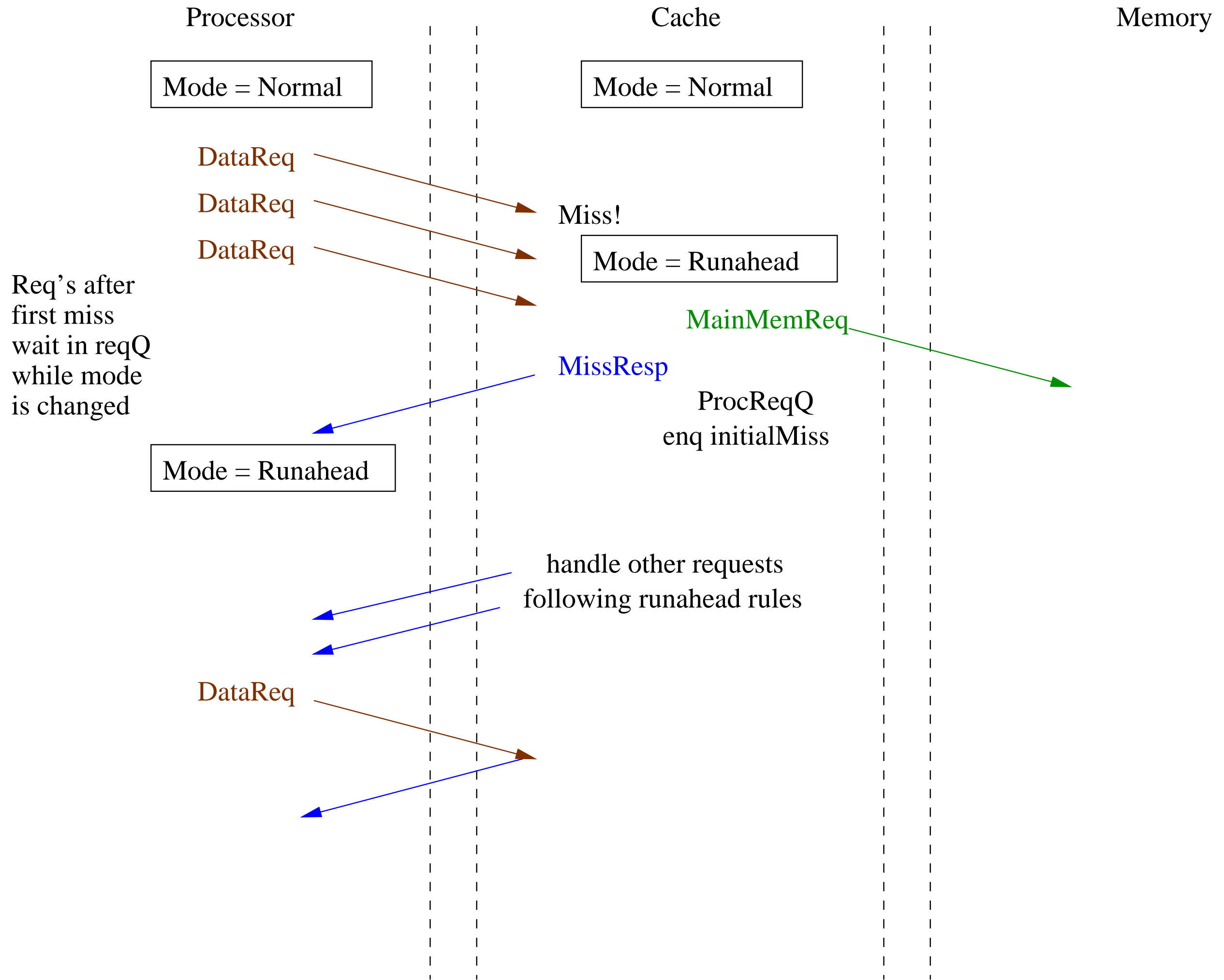

Figure 2: A diagram showing processor-cache-memory communication when the initial miss occurs.

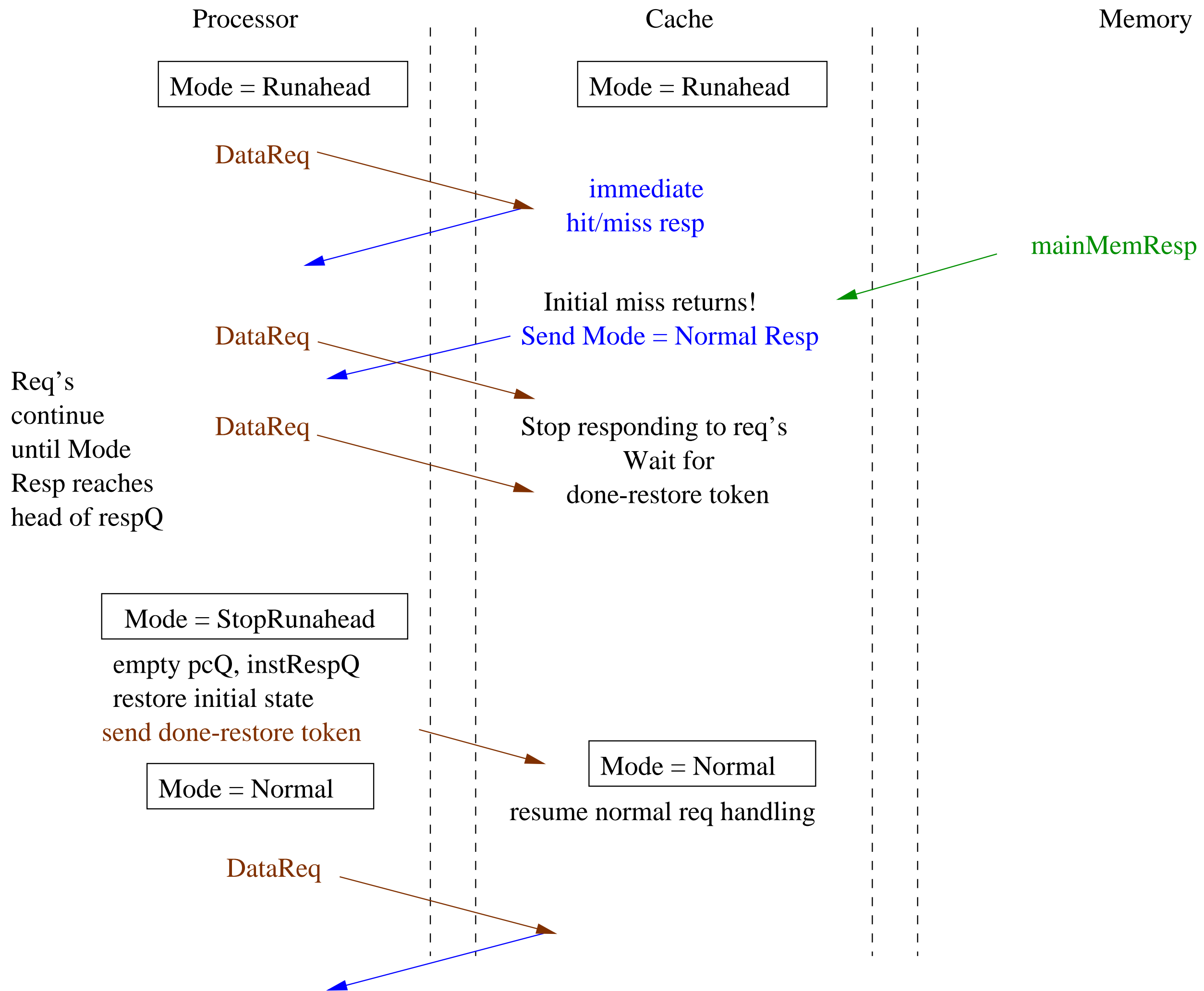

Figure 3: A diagram showing processor-cache-memory communication when the initial miss returns from memory.

Exiting runahead (Figure 3) is a little more complicated than entering runahead. The exit process begins when the request marked '`isInitialMiss`' reaches the front of the `procReqQ`. At this point, `refill-resp` sends a '`!isRunahead`' flag down the resqQ to the processor. However, until the flagged response reaches the front of the `respQ`, the processor may continue to send data requests to the cache. The cache discards these requests.

When the processor receives the '`!isRunahead`' flag, it moves into stop-runahead mode to restore the system. After the restore is complete—which includes draining the `pcQ` and `instRespQ`—the `stop-runahead` rule sends a 'done-restore' req through the ReqQ to the cache. The cache resumes normal operation.

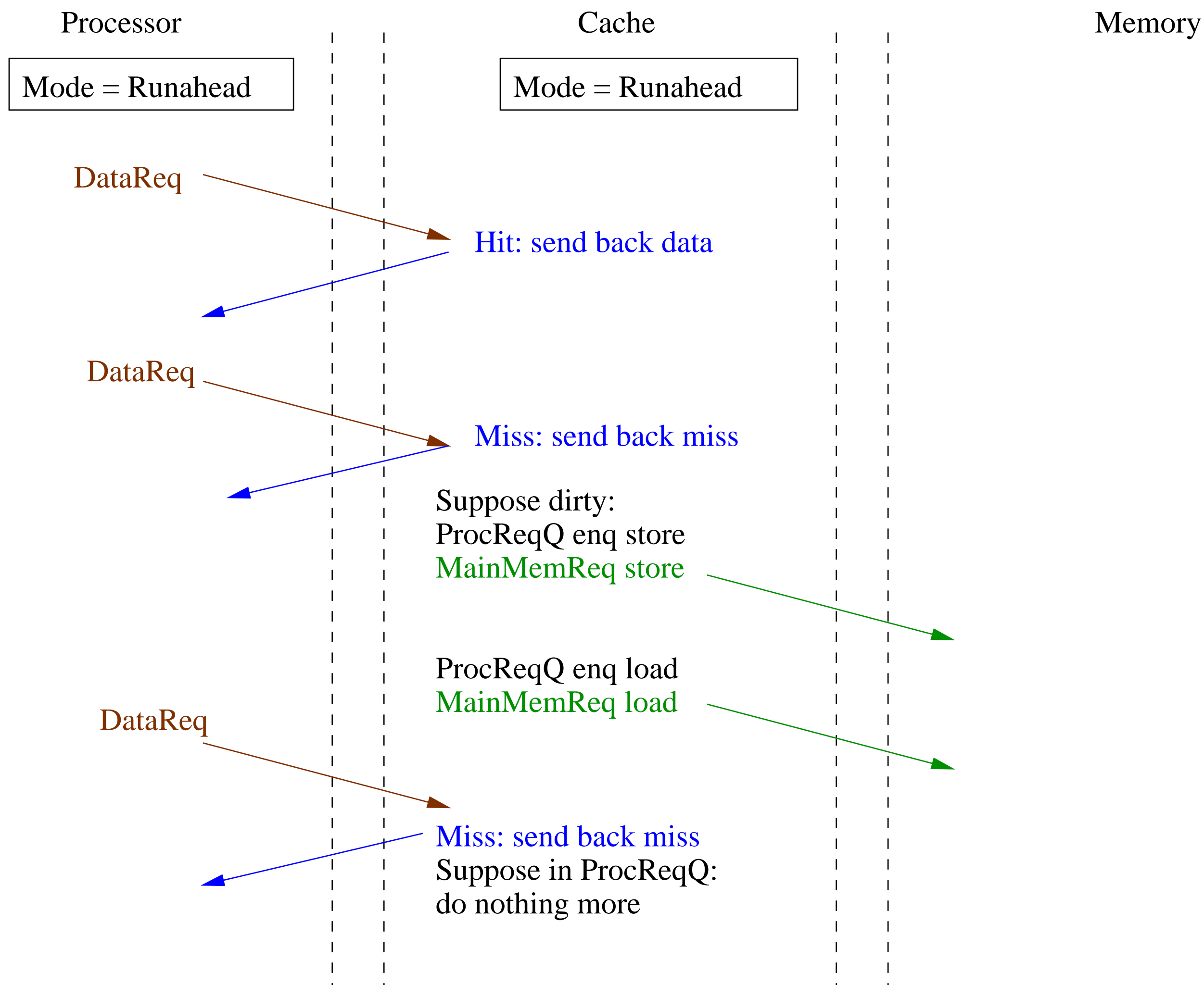

Figure 4: A diagram showing processor-cache-memory communication for various scenarios during runahead mode.

Figure 4 shows three request scenarios during runahead mode. The first is a hit; the data is immediately sent back to main memory. The second is a dirty miss. In this case, the original data is first sent back to main memory. Both the store request and the load request go in the `procReqQ`. The final request is a miss to an address that is already in flight. The cache immediately responds to the processor but doesn't initiate any memory requests.

5 Design Exploration

We considered two strategies to improve runahead performance. The first was a temporary store-cache for recording the values of stores made during runahead (recall that we do not want to change the state of the real memory in runahead mode). The second strategy was to

wait for all prefetches—not just the initial miss—to return before exiting runahead, which might reduce the number of times we had to enter runahead.

The cross-product of these strategies gave us a total of five variations:

- Standard processor (baseline)
- Runahead processor
- Runahead processor with store cache
- Runahead processor that waits for prefetches
- Runahead processor that waits for prefetches, has store cache

All variations included a branch predictor and used a non-blocking cache (including the standard processor; we built a standard non-blocking cache for it to use).

5.1 Design Variations

Store-Cache. The store cache is a small module in the cache that stores the data from runahead store requests (our normal runahead processor just throws the data away). It also stores whether the data is valid. Thus, if a load depends on data from a previous store, that data can be returned to the processor for future operations. If the stored data is valid, then the load will return valid data. The processor can use the data to run farther ahead and produce more prefetches. If the stored data is invalid, the load will return invalid along with the data. The processor knows that the data is corrupted, and may choose to stall, preventing useless prefetches.

In implementation, the store cache strongly resembles our four-element branch predictor table: it contains a RAM of valid bits and a RAM of { address, data } pairs, both indexed by the lower order address bits. The valid bits can take on three values: `neverWritten`, `valid`, and `invalid`. Loads and stores are handled as follows:

- **Stores.** All stores during runahead are stored into the store cache based on the least significant bits of the address (old address-data pairs, if present, are overwritten).
- **Loads.** All loads during runahead—whether they are hits or misses—first query the store-cache. The store cache either returns the matching address-data pair and appropriate valid bit or a bogus address-data pair with a `neverWritten` valid bit. If a `neverWritten` is returned, then the load checks the actual cache ram and returns the appropriate hit/miss. If `valid` or `invalid` is returned, the load returns the data from the store cache and the matching valid bit to the processor.

Waiting for Prefetches. Instead of exiting runahead when the initial miss returns, this cache waits for all prefetches currently in flight to return before notifying the processor to exit runahead. When the initial initial miss returns, the cache goes into a `wait-for-prefetches` state. New processor requests are discarded while the cache waits for the `procReqQ` to empty (all prefetches to return). Waiting for prefetches increases the runahead period by a time proportion to the main memory throughput. However, by ensuring that all the requests are updated in the cache, we aimed to prevent a premature return to runahead (note that since we deque fifos one for one, exiting runahead can be a particularly costly for long fifos).

5.2 Design Parameters

Prefetching usually shows the most performance improvement at the L2 cache level, where memory latencies can be hundreds of cycles. Cache misses from the L1 cache typically only cost tens of cycles, so we would generally not expect prefetching to show as much improvement—the cost of entering and exiting runahead is too large compared to the number of prefetches made. We did not implement an L2 cache; however, we simulated the effects of larger costs by varying the latency of our main memory requests.

Fifo sizing is also important: longer fifos decouple modules and rules and allow them to operate more independently. This decoupling is crucial for the non-blocking cache: if the `procReqQ` and `reqQ` fill, then the processor must stall until a response comes back from main memory. On the other hand, long queues allow the processor and cache to continue sending requests without waiting for a response. In fact, a long queue with a non-blocking cache effectively performs a simple kind of runahead: even if the response to a miss has not returned, the processor can continue to put items in the `wbQ` and `reqQ`.

For each of our design variants, we tested three levels of latency—1, 20, and 100 cycles—and four fifo lengths—2, 5, 8, and 15[1] items—for a total of twelve parameter configurations per variant. The relevant fifos were the `reqQ`, `respQ`, `wbQ`, `procReqQ`, and `memRespQ` (the instruction cache is preallocated prior to running the benchmarks, so its fifo sizing should not impact runahead performance).

6 Results

We found that the runahead processor outperforms the standard processor by up to a factor of 6.9 (fifo length 15, latency 100, `vvadd` benchmark). The benefits are most pronounced when the test program involves many independent loads and stores, and the latencies are long. In this section, we first describe the performance of the runahead processor and our explorations in terms of the executed instructions per second (IPS). Then we discuss the impact of runahead processing on secondary metrics such as area, cycle time, fetch traffic.

[1] The compiler had trouble synthesizing a `wbQ` longer than eight elements, so in the fifo-15 run, all fifos except the `wbQ` were resized. Since the it was the `procReqQ` sizing that was most crucial, we were still able to see many of the expected effects.

6.1 IPS Performance

Figures 5, 6, 7, and 8 chart the instructions per second as the fifo length was varied. For all of the benchmarks, the baseline processor did best when the latency is low—in these cases, the penalty for a cache miss was small compared to the cost of entering runahead. The baseline processor also does better as the fifo lengths get longer because a long fifo decouples the processor and cache rules. The `exec` rule can continue firing after a cache miss until the `dataReqQ` and `wbQ`'s are full, a form of pseudo-runahead.

As the latencies get longer, however, the benefits of runahead become more evident. For example, when the fifo is of size eight and there are many independent loads (`vvadd`), the IPS of the baseline processor drops to 12.9 percent of its latency-1 value when the latency is raised to 100 cycles. In contrast, the IPS for the runahead processor stays at 96.9 percent of its latency-1 value. The benefits are less clear with shorter fifos, however—once the `procReqQ` is full, the runahead processor cannot continue prefetching, regardless of the memory latency.

All of the runahead variants performed at approximately the same level. The IPC's for both basic runahead and the runahead with store-cache were exactly the same; the differences in the bars come from the fact that store-cache version was optimized to a slightly smaller cycle time. Thus, for our benchmarks, the store cache did not truly improve performance.

The wait-for-prefetch versions had lower IPC's and lower corresponding IPS's. The penalty for stopping runahead was long enough that the cache could receive additional prefetches during that period and still store them before the processor required them; waiting for the prefetches to return only prolonged the penalty for stopping runahead.

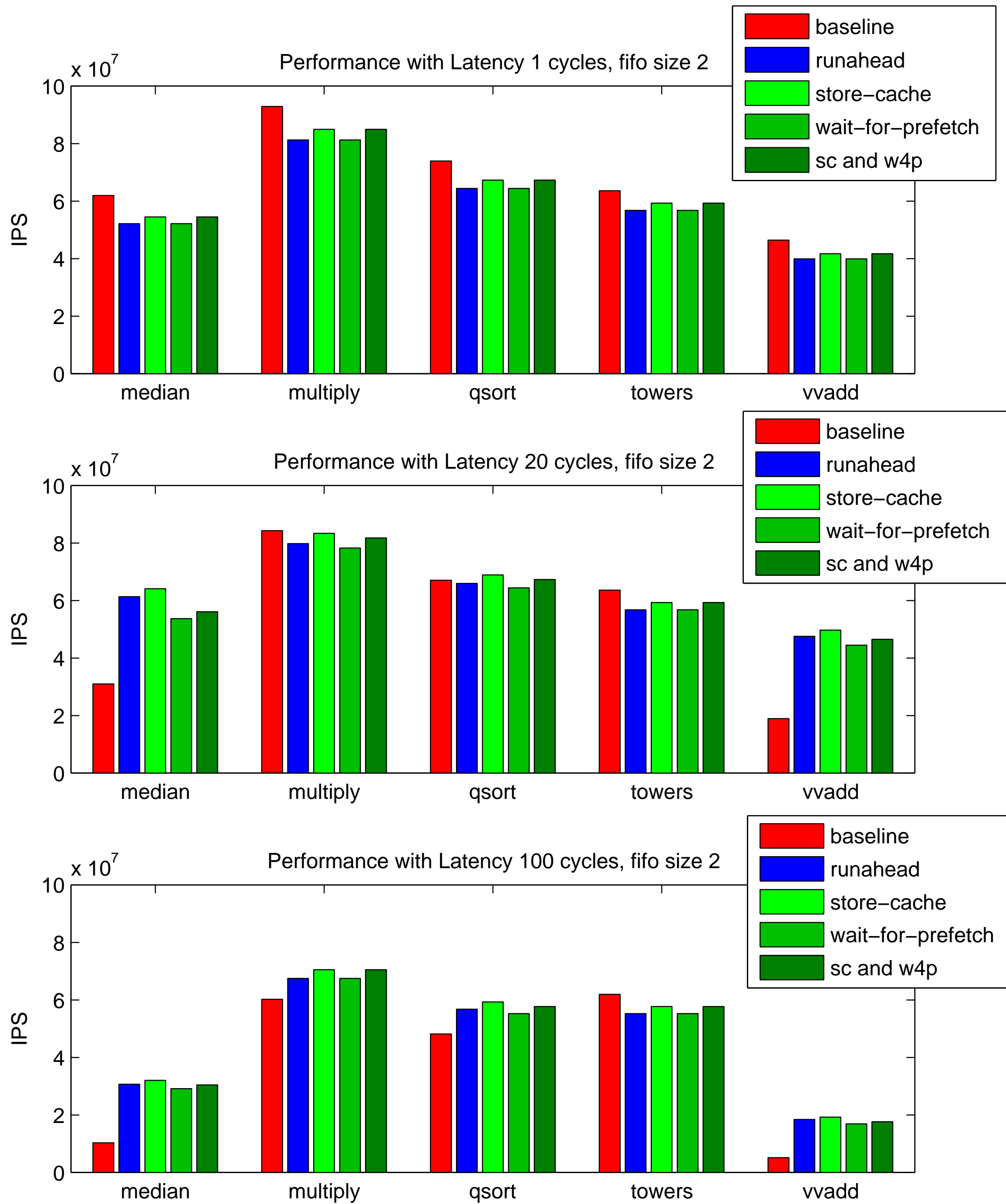

Figure 5: Instructions per Second for FIFO length 2.

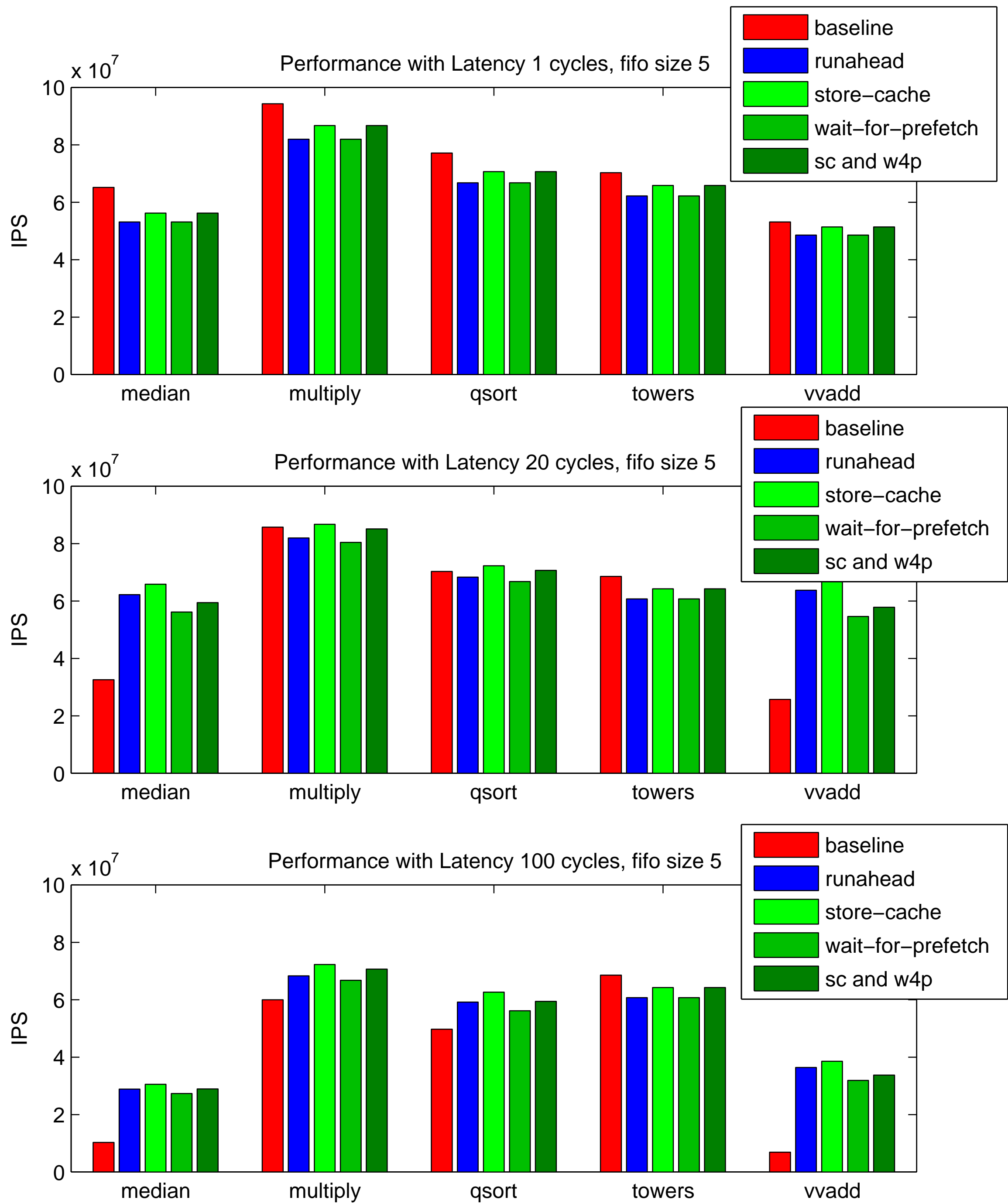

Figure 6: Instructions per Second for FIFO length 5.

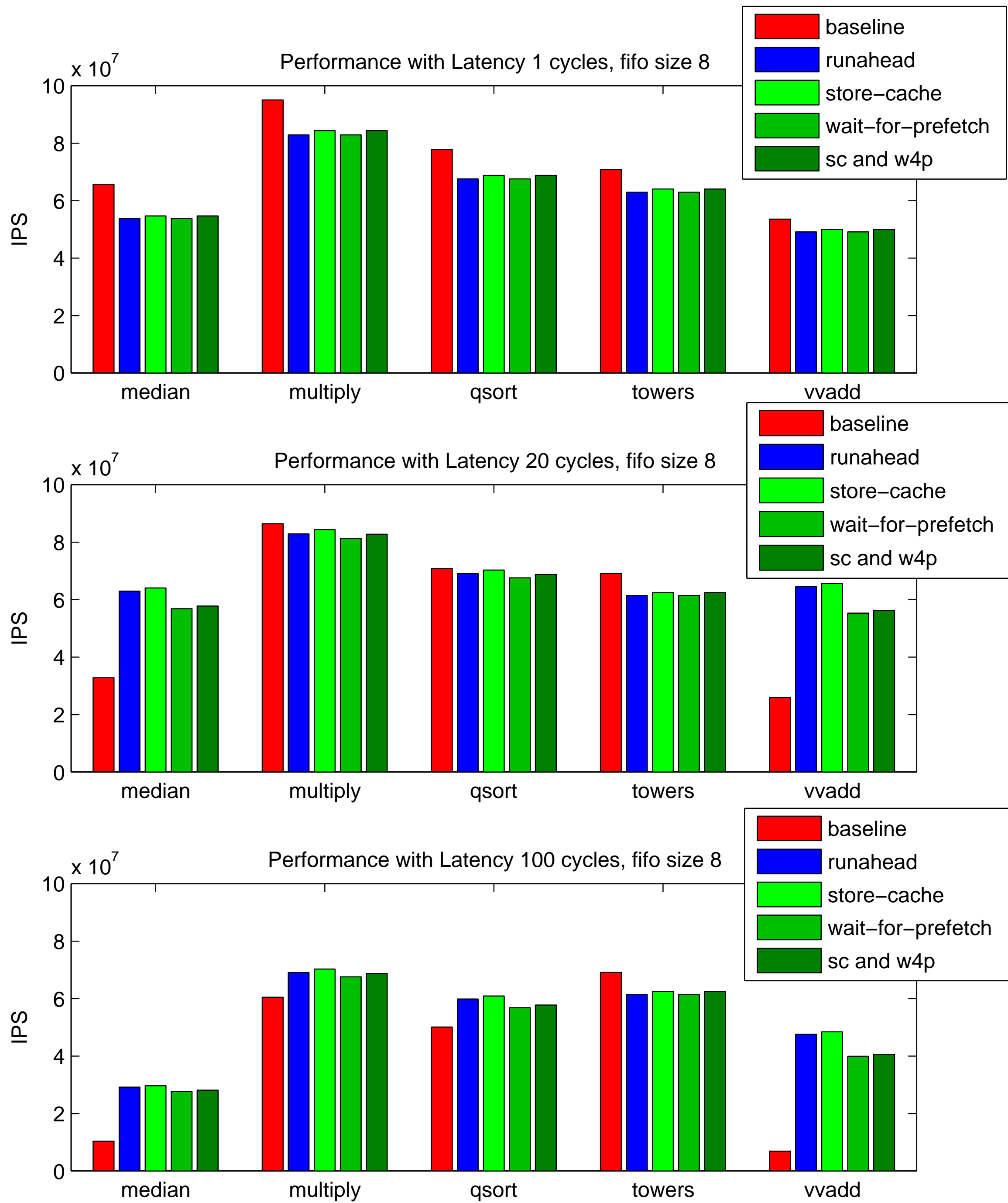

Figure 7: Instructions per Second for FIFO length 8.

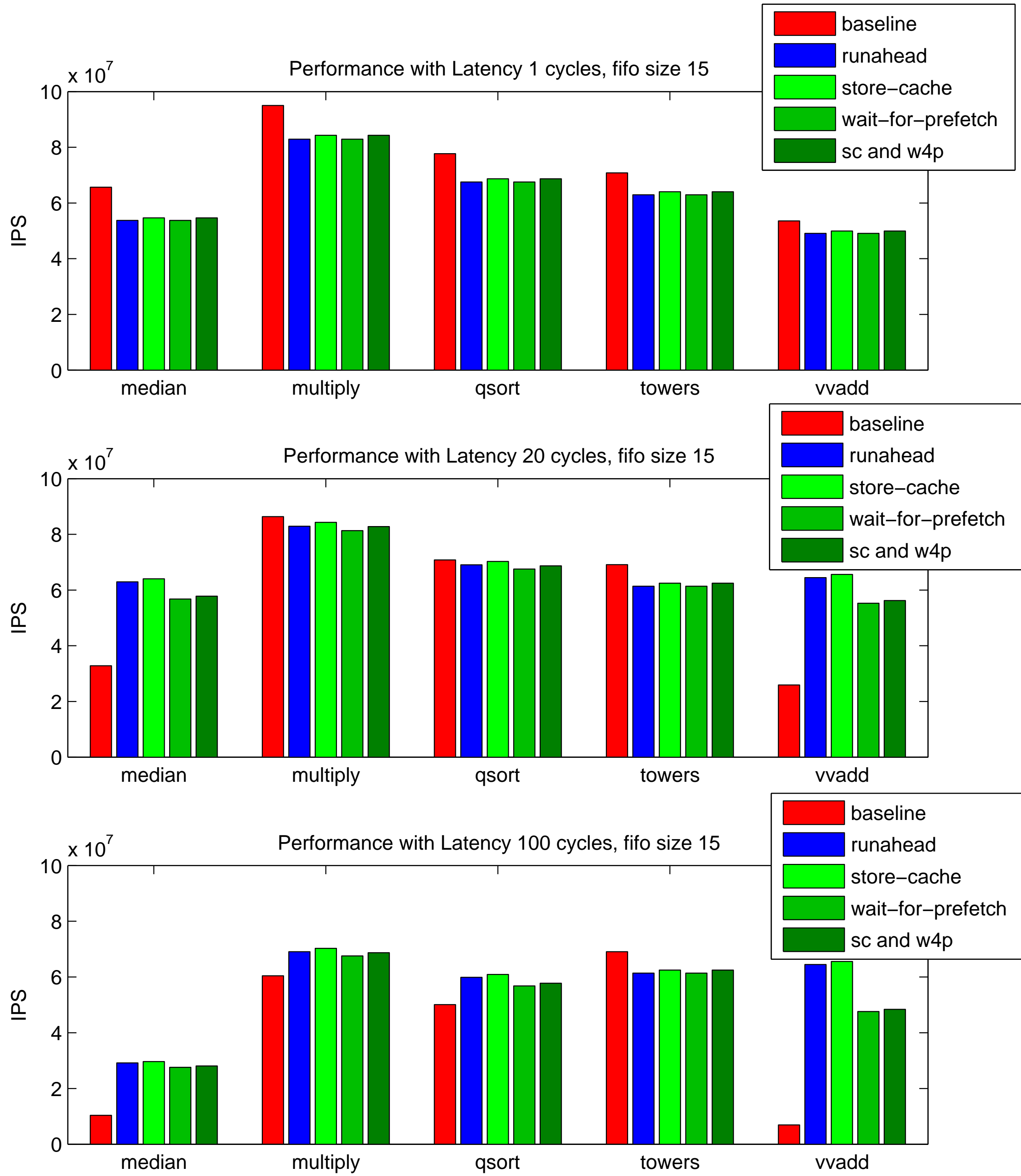

Figure 8: Instructions per Second for FIFO length 15.

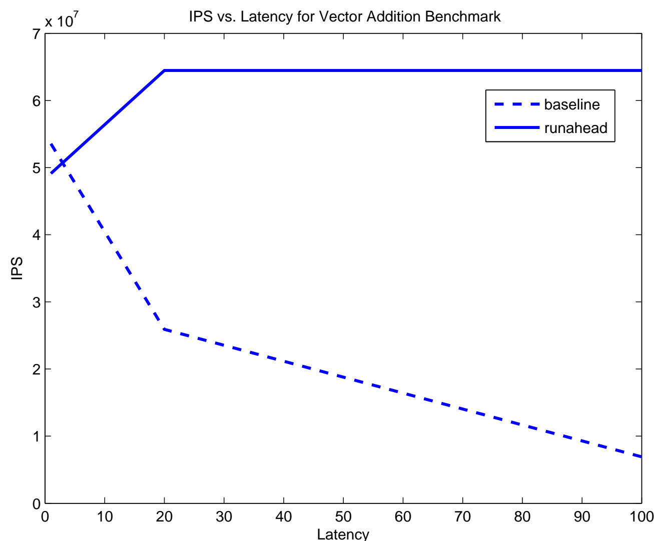

Figure 9: Instructions per Second vs. Latency for Baseline Normal, Runahead Processors with FIFO Size 15.

Since there was little variation in the performance of the design exploration alternatives, the next plot focuses on the baseline and standard runahead processor. Figure 9 shows more clearly the benefits of runahead as the latencies get long on the vvadd benchmark. As expected, the baseline processor's performance drops with increasing latency (although it has a non-blocking cache, it can only pile two extra program counters into the `pcQ` before the system needs to stall). However, the runahead processor's performance actually increases with latency.

6.2 Secondary Metrics

While improving IPS was our primary objective, we also evaluated the effect of runahead processing on processor-cache area, cycle time, and fetch traffic.

Area The backup register file is the most significant addition to runahead processor area. Table 3 compares the processor areas of each of our design variants; overall the runahead processor requires about twenty percent more area than the baseline. On the processor side, the store cache increased area by about five percent. We used the standard floorplanning to produce the area numbers. The fifos caused the most signficant area increase of up to thirty percent.

Within the cache, the fifo sizes impacted the control logic area more signficantly than the store cache (Figure 10). The store cache increased the area by between eight and thirty percent, but increasing the fifo lengths more than doubled the size of the controller. Thus, although the store cache had no signficant effect on our benchmarks, it may be worth keeping

Table 3: Total Processor Area in μm^2 (fraction of baseline Fifo-2)

Variant	Fifo-2	Fifo-5	Fifo-8, Fifo-15
baseline	455522.8 (1.00)	475721.8 (1.04)	491345.3 (1.07)
runahead	543424.9 (1.19)	569130.7 (1.24)	693736.5 (1.52)
store-cache	569130.7 (1.24)	592352.8 (1.30)	698751.0 (1.53)
wait for prefetches	543424.9 (1.19)	569130.7 (1.24)	693736.5 (1.52)
both	569130.7 (1.24)	592352.8 (1.30)	698751.0 (1.53)

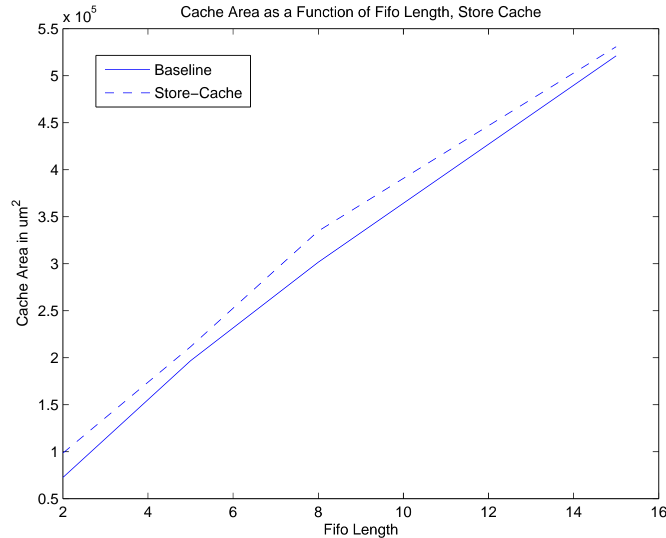

Figure 10: Cache controller area for runahead baseline and store-cache variant as a function of fifo size.

since it has very little impact on design area. The area penalty of the long fifos may be mitigated by choosing which fifos to size more carefully: clearly a longer `procReqQ` allows more prefetches to be in flight and thus allows the processor to take full advantage of going into runahead. However, we should be able to make the request and response queues to the processor much shorter since these queues are less affected by the memory latency.

Cycle Time The runahead logic also increases our cycle time. As seen in Table 4, the logic increases the time by about ten percent. All modules were routed with a target time of 5 ns. Interestingly enough, the store cache variants had a slightly faster cycle time. We believe that the optimizer may have worked harder to meet the target time for the more complex system, resulting in a better optimization. Also, the fifo length is not always on the critical path so it does not always impact the cycle time. Overall, the designs with shorter fifos are slightly faster, but the penalty for using larger fifos in terms of time is cancelled by

Table 4: Cycle Time in Nanoseconds (fraction of baseline Fifo-2)

Variant	Fifo-2	Fifo-5	Fifo-8, 15
baseline	5.81 (1.00)	5.83 (1.00)	5.79 (1.00)
runahead	6.52 (1.12)	6.59 (1.13)	6.52 (1.12)
store-cache	6.24 (1.07)	6.23 (1.07)	6.40 (1.10)
wait for prefetches	6.52 (1.12)	6.59 (1.13)	6.52 (1.12)
both	6.24 (1.07)	6.23 (1.07)	6.40 (1.10)

the overall performance benefits. The fifo-15 numbers were the same as the fifo-8 numbers since the critical path did not include the data cache and both used the same size `wbQ`.

Although all of the runahead versions had approximately the same cycle time, the critical paths in each case varied widely. The critical path for the normal baseline processor took place when updating the branch predictor table in `exec`; whereas the standard runahead and the wait-for-prefetch version's critical paths went from the data request queue to the `pc-register` in `pc-gen`. The store-cache variants' critical paths went from the `wbQ` to the `pc-register` and from the `dataRespQ` to the `wbQ`. On the processor side, the four variants had very few differences, so it is unclear how the optimizations worked to produce these results.

Fetch Traffic Finally, we measured the fetch traffic for all of our designs. Since the variants were virtually indistinguishable in performance, we focus here on the standard runahead processor with size-15 fifos. Due to our invariant of only prefetching data that we will need (and searching the `procReqQ` for requests already in flight), we never make any unnecessary prefetches[2]. However, depending on the latencies, we may have to enter and exit runahead multiple times, and exiting runahead typically had a penalty of 5-10 cycles.

Figure 11 shows fraction of request that were initial misses for the standard runahead processor across the five benchmarks. Equivalently, the figure shows what fraction of misses initiated a runahead operation. For low latencies, few prefetches can be made before the initial miss arrives, and thus most misses initiate a runahead period. However, as the latencies get longer, more prefetches can be made, and we have to enter runahead less often.

The `vvadd` and `median` continue to improve (that is, enter runahead less often) as the latency increases. This matches exactly the initial aim of runahead: as the latency penalty increases, we should queue more prefetches to decrease the total number of misses. In contrast, `multiply` and `qsort` level out somewhere before latency 20. Here, the processor entered the `stall-runahead` state because it hit a branch that depended in invalid data. More aggressive branch prediction may allow the processor to overcome this hurdle and keep processing, at the risk of increasing overall fetch traffic with prefetches that do not prove to be useful (if the wrong branch was taken). Finally, the towers benchmark produced no

[2]In our benchmarks, we confirmed that the runahead processor makes exactly the same number of memory requests as the baseline normal processor.

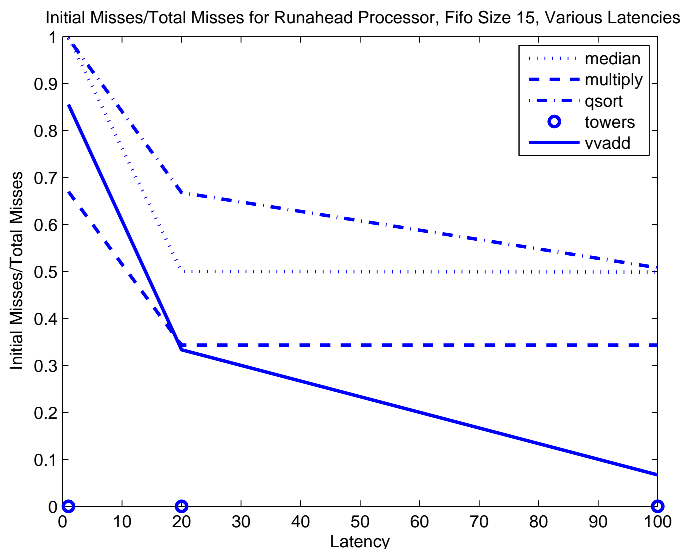

Figure 11: Initial Misses/Total Requests for Baseline Runahead Processor across Benchmarks, Latency.

cache misses, so taking a fraction of misses does not make sense.

7 Conclusions and Future Work

When cache latencies are high, the runahead processor appears to have significant advantages over a standard processor. For programs that involve many independent loads and stores, queuing prefetches after a miss can result in performance boosts by a factor of up to 6.9. Since performance is sensitive to the latency and fifo length parameters, these should be tuned to each system with care.

This project studied two variants: using a store-cache and waiting for prefetches. Waiting for prefetches actually decreased performance, and the store-cache had no effect. Since the store-cache adds only five percent more area, however, it may be useful to keep for special situations. However, there are many other design variations that deserve attention:

- **Backup Register File Configuration.** If the backup register file is a separate module, then the backup and restore times are bounded by the number of the read/write ports. Depending on how often we enter runahead, however, the runahead penalty may be offset by better floorplanning options and short drive lines. Other register file explorations include using a backup register file that is smaller than the original register file, and store only elements that are overwritten with invalid data. While this will save register area, we may pay a time penalty due to a limited number of ports and invalid computations. Finally, although research suggests that it is not a big win[4],

it may be worth trying to 'save' the valid computations that occur during runahead instead of reexecuting them.

- **Value Prediction.** Currently we do not execute requests with invalid addresses, maintaining the invariant that we only prefetch data that we will need in the future. A more aggressive strategy would use address value prediction to guess load addresses[5]. Prediction strategies could also be used to guess branches even when the predicate is invalid.

- **Prefetching with Collisions.** The most conservative strategy, implemented in our project, only allows one prefetch per cache index. An associative cache may partially mitigate the effect of colliding cache lines; however, a more aggressive strategy would prefetch all data, regardless of collisions. A combination of fifo sizing and cache arbiter, as well as good understanding of system latencies, would allow the processor to use the data from the first prefetch before the colliding prefetch overwrites it.

- **Cache Size and Type.** The size of the cache determines how accurate the prefetching needs to be. With smaller caches and non collision-free prefetches, useless prefetches may remove useful prefetches from the cache. Associative and direct-store caches may also improve performance.

Finally, although the runahead processor provides an elegant, relatively low-overhead solution to improving performance during a cache miss, other options exist to reduce the impact of cache misses, including active prefetching (looking ahead in the program instead of waiting for the initial miss) and out-of-order processing (allowing multiple operations to execute in parallel). A comprehensive evaluation of runahead processing should include comparisons to these alternatives.

References

[1] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS '97: Proceedings of the 11th international conference on Supercomputing*, (New York, NY, USA), pp. 68–75, ACM Press, 1997.

[2] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), p. 129, IEEE Computer Society, 2003.

[3] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, (Los Alamitos, CA, USA), pp. 81–87, IEEE Computer Society Press, 1981.

[4] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *IEEE Comput. Archit. Lett.*, vol. 4, no. 1, p. 2, 2005.

[5] O. Mutlu, H. Kim, and Y. N. Patt, "Address-value delta (avd) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 233–244, IEEE Computer Society, 2005.