Post-Fabrication Microarchitecture

Chanchal Kumar*
ckumar2@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Anirudh Seshadri
aseshad2@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Aayush Chaudhary*
achaudh6@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Shubham Bhawalkar*
spbhawal@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Rohit Singh
rsingh25@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Eric Rotenberg
ericro@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

Microarchitectural enhancements that improve performance generally, across many workloads, are favored in superscalar processor design. Targeting general performance is necessary but it also constrains some microarchitecture innovation. We explore relieving this constraint, via a new paradigm called Post-Fabrication Microarchitecture (PFM). A high-performance superscalar core is coupled with a reconfigurable logic fabric, RF. A programmable interface, or Agent, allows for RF to observe and microarchitecturally intervene at key pipeline stages of the superscalar core. New microarchitectural components, specific to applications, are synthesized on-demand to RF. All instructions still flow through the superscalar pipeline, as usual, but their execution is streamlined (better instructions per cycle (IPC)) through microarchitectural intervention by RF. Our research shows that one can achieve large speedups of individual applications, by analyzing their bottlenecks and providing customized microarchitectural solutions to target these bottlenecks. Examples of PFM use-cases explored in this paper include custom branch predictors and data prefetchers.

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**.

KEYWORDS

instruction-level parallelism (ILP), superscalar processor, branch prediction, prefetching, pre-execution, reconfigurable logic, field-programmable gate array (FPGA)

ACM Reference Format:
Chanchal Kumar, Anirudh Seshadri, Aayush Chaudhary, Shubham Bhawalkar, Rohit Singh, and Eric Rotenberg. 2021. Post-Fabrication Microarchitecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480119

*Author contributed to this work while at NCSU. Kumar is currently at ARM Inc., Austin, TX, USA. Chaudhary is currently at Samsung Austin Semiconductor, L.L.C., San Jose, CA, USA. Bhawalkar is currently at Qualcomm, Austin, TX, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480119

1 INTRODUCTION

Improvement in single-thread performance of superscalar processors has slowed owing to technology limitations and power constraints. With stagnating frequency, continued performance improvement depends on microarchitectural improvements to unlock and harvest more instructions-per-cycle (IPC) from workloads. With generational increases in superscalar widths and structure sizes, devising new and better branch predictors, instruction and data prefetchers, value predictors, and other microarchitectural components, is of critical importance. Unfortunately, this effort, of devising new and better components, is severely hamstrung by the need for generality across workloads. Thus, generational performance improvements tend to be incremental.

In this paper, we first make a case that, by examining the code for a given application, it is possible to conceive of custom microarchitectural components that enable the superscalar core to extract large performance gains on that application. Our examples showcase specialized branch predictors and data prefetchers, but the idea may extend to other modular features. Second, to support *post-fabrication* deployment of custom microarchitectural components, we propose coupling a reconfigurable logic fabric (RF) with a superscalar core, on the same chip (see Figure 1). RF could be FPGA and/or CGRA, or a reconfigurable fabric specially designed with this paradigm in mind. The interface is comprised of three "agents". The Retire Agent interfaces with the core's retire stage to observe specified retired instructions. The Fetch Agent interfaces with the core's fetch stage to stream custom conditional branch predictions. The Load Agent interfaces with the core's load/store execution lanes, to issue prefetches and loads. It is important to note that, the superscalar core still fetches and executes the original dynamic instruction stream (*i.e.*, we are not intervening architecturally such as by offloading work to accelerators); the difference is that instruction fetching and execution is more streamlined due to *microarchitectural intervention* by components programmed into RF. We call this novel paradigm *Post-Fabrication Microarchitecture (PFM)*, or Post-Fab Microarchitecture for short.

The agents in Figure 1 are designed as integral parts of the superscalar core and operate at the core's frequency. A custom microarchitectural component synthesized in RF operates at a slower frequency. Effective PFM use-cases compensate for slower frequency in two ways: (1) they only observe a fraction of retired instructions

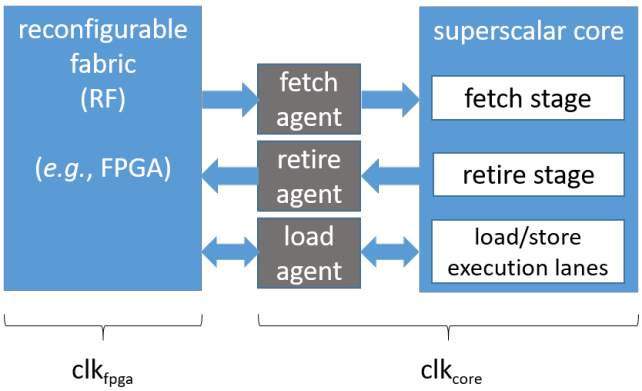

Figure 1: Substrate for Post-Fabrication Microarchitecture (PFM).

and intervene for a fraction of fetched instructions, (2) they can efficiently process multiple instruction-equivalents per cycle (the custom component's "width") owing to parallelism in their internal design.

1.1 Motivation for custom microarchitectural components

Figure 2 shows speedups that are possible with custom microarchitectural components. The *astar* benchmark has frequent mispredicted branches. The *bfs* benchmark has both frequent mispredicted branches and cache-missed loads. We chose these two motivating examples because they are the two custom branch predictor use-cases in this paper. Speedups are also shown for Slipstream 2.0 [20], the most recent state-of-art load and branch pre-execution architecture.

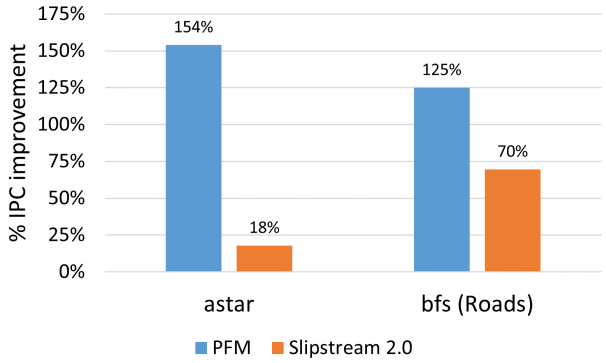

Figure 2: Speedups of PFM and Slipstream 2.0.

Consider the region-of-interest (ROI) in *astar* (a pathfinding benchmark), shown in Figure 3. Both branches 1 and 2 are highly mispredicted by a state-of-art 64KB TAGE-SC-L branch predictor [17], as they are data-dependent on input worklist *bound1p* (line 4) that changes with each call to this function. Slipstream automatically pre-executes branch 1 by pruning its control-dependent (CD) region (lines 8–21) from the leading thread. By removing branch 1's CD region, the leading thread is not slowed by branch 1's would-be mispredictions and supplies pre-executed outcomes to the trailing thread. As Srinivasan et al. explained in their paper (Sect. IV.A.1, Fig. 6 [20]), the automated technique is limited: (1) branch 2 cannot also be pre-executed because it is skipped-over, (2) a non-negligible fraction of branch 1's dynamic instances are pre-executed incorrectly owing to omitting the loop-carried memory dependency between the store at line 13 and future instances of branch 1, leading to frequent restarts of the leading thread. Consequently, slipstream achieves only 18% IPC improvement, and this speedup is qualified by two tailored optimizations. First, we used *a priori* knowledge of the ROI to hardwire the leading thread's pruning predictor, circumventing the training time for its automated machinery. Second, for case (2) above, we optimized slipstream's speedup by not restarting the leading thread and instead only locally squashing the trailing thread (leveraging unique knowledge of the ROI); the speedup is substantially lower with restarts, even assuming single-cycle rollback of the leading thread.

```
 1:  bound2l=0;
 2:  for (i=0; i<bound1l; i++)
 3:  {
 4:    index=bound1p[i];
 5:
 6:    index1=index-yoffset-1;
 7:    if (waymap[index1].fillnum!=fillnum)
 8:      if (maparp[index1]==0)
 9:      {
10:        bound2p[bound2l]=index1;
11:        bound2l++;
12:
13:        waymap[index1].fillnum=fillnum;
14:        waymap[index1].num=step;
15:
16:        if (index1==endindex)
17:        {
18:          flend=true;
19:          return bound2l;
20:        }
21:      }
22:    ... Above nested-if template repeats
23:    ... 7 times for different index1's.
24:  }  // end for loop
```

branch 1

branch 2

Figure 3: *astar*: slipstream challenges.

In contrast, our custom branch predictor for *astar* achieves 154% IPC improvement, close to the 162% IPC improvement of perfect branch prediction for this benchmark. By using custom knowledge of the ROI, we are able to pre-execute both branches 1 and 2 and infer the effect of the omitted store. The custom design understands that the meaning of the store is to prevent revisiting a given *index1*; so it compares the current *index1* with recently visited *index1*'s (those that have not been retired by the core yet), and on a match infers the updated value of *waymap[index1].fillnum*. Additionally, the custom branch predictor decouples loading indices from the input worklist (line 4), the loads that feed branches 1 and 2 (*waymap[index1].fillnum* and *maparp[index1]*), and computation of the predicates of branches 1 and 2. This decoupling achieves high memory-level parallelism (MLP).

1.2 Paper Outline

The paper is organized as follows. Section 2 describes the Fetch, Retire, and Load Agents that interface the reconfigurable fabric to the superscalar core. Section 3 describes methodology: the simulator, superscalar core and memory hierarchy configuration, and notation for parameters of the custom component and Agents. Section 4 presents the PFM use-cases and performance results. Section 5 presents FPGA results to estimate cost, power, and frequency of some of the custom components. Section 6 discusses related work. We conclude the paper and discuss future work (automating or assisting design generation) in Section 7.

2 PFM AGENTS

Sections 2.1, 2.2, and 2.3, describe the Retire, Fetch, and Load Agents, respectively.

Figure 4 shows the Fetch and Retire Agents. A configuration bitstream shipped with the executable synthesizes the custom microarchitecture component in the FPGA and configures the Fetch Snoop Table (FST) and Retire Snoop Table (RST) in the Fetch and Retire Agents, respectively.

2.1 Retire Agent

The Retire Agent is between the core's Retire Unit and FPGA. It implements two basic observation functions: detect the beginning/end of a region-of-interest (ROI) and construct observation packets for the FPGA. Both are triggered by matching program counters (PCs) of instructions in the retire bundle against PCs in the RST.

Initially, the Fetch and Retire Agents are idle, until a PC hits in the RST that corresponds to a beginning of a ROI. At this point, the Retire Agent's controller: (1) signals the core to squash its pipeline so that the core and custom component are logically at the same point in the dynamic instruction stream, (2) signals the Fetch Agent to enable itself, and (3) sends a beginning-of-ROI packet to the FPGA via the Observation Queue at Retire (ObsQ-R), thus enabling the custom component.

After observing the beginning of ROI, a PC that hits in the RST causes the Retire Agent's controller to construct and send an observation packet to the FPGA. Depending on the configuration of the RST entry that hits, the controller will construct one of three types of observation packets: (1) a destination value packet (for an instruction with a destination register), (2) a store value packet (for a store instruction), and (3) a branch outcome packet (for a branch instruction). In this paper, we assume a contemporary superscalar core design based on a Physical Register File (PRF) and Active List (control-only variant of reorder buffer). Thus, to construct a destination value packet, the Retire Agent's controller shares one or more ports to the PRF (we sweep port-sharing options in results) with the execution lanes that own those ports. The controller applies the retiring instruction's destination tag to one input of a 2:1 MUX at the address input of the shared read port, until the tag is selected. The select for this MUX is a busy signal in the register read stage of the execution lane. To construct a store value packet, the controller references the store value that is being committed from the head of the Store Queue (SQ) to the data cache unit's write buffer. Our fetch unit maintains a branch queue for all in-flight branches (to train branch prediction tables at retirement and checkpoint/restore global branch history), thus, the retiring branch's outcome is available from the head of the branch queue, for constructing branch outcome packets.

On a pipeline squash, the Retire Agent sends a squash packet to the FPGA, so that the custom component has the option of rolling back as needed to ensure that its subsequent predictions realign with where the core is in the dynamic instruction stream. For example, in this paper's PFM use-cases that stream custom branch predictions, various internal queue indices are rolled back. The Retire Agent stalls the core's Retire Unit until a squash-done packet is received from the custom component. A squash-done packet is sent from the FPGA to the Fetch Agent via the Intervention Queue at Fetch (IntQ-F), and the Fetch Agent's controller signals the Retire Agent's controller that the squash is done.

2.2 Fetch Agent

The Fetch Agent is between the core's Fetch Unit and FPGA. Once enabled by the Retire Agent (at beginning of ROI), the Fetch Agent matches PCs of instructions in the fetch bundle against PCs in the Fetch Snoop Table (FST). A hit causes the Fetch Agent's controller to override the Fetch Unit's conditional branch predictor with a conditional branch prediction (0:not-taken/1:taken) popped from the Intervention Queue at Fetch (IntQ-F). If IntQ-F is empty, owing to the custom component running late, the Fetch Agent stalls the core's Fetch Unit until the packet arrives. Section 2.4 discusses breaking or avoiding deadlocks of buggy custom components under development or after deployment.

2.3 Load Agent

The Load Agent is shown in Figure 5. It is between the core's load/store execution lane and FPGA. The Load Agent pops a prefetch or load packet from the Intervention Queue at Issue (IntQ-IS) and issues the prefetch/load to one of the core's load/store execution lanes when the corresponding issue port is not busy.

The core handles custom component loads differently than regular loads. They do not search the Store Queue and do not participate in the core's wakeup and bypass logic. They only get translated by the TLB and access the data cache, and the load's result is steered to the Load Agent.

If an injected load misses, the Load Agent buffers the missed load in its missed load buffer. The Load Agent periodically replays missed loads until they hit.

The Load Agent returns load values to the custom component via the Observation Queue at Execute (ObsQ-EX). Load values may be returned out-of-order owing to hit-under-miss of loads from IntQ-IS. So that the custom component can reorder loads internally, it attaches a unique identifier to each load that it pushes into IntQ-IS. The Load Agent and core's load/store lane both retain this identifier with the load, up to and including when the load value is pushed into ObsQ-EX.

2.4 Security and Debug

A custom component cannot modify architectural state of the running program. The only interventions permitted by the Agents are microarchitectural. A custom component's predictions are the same as the core's predictions: they are eventually verified by execution

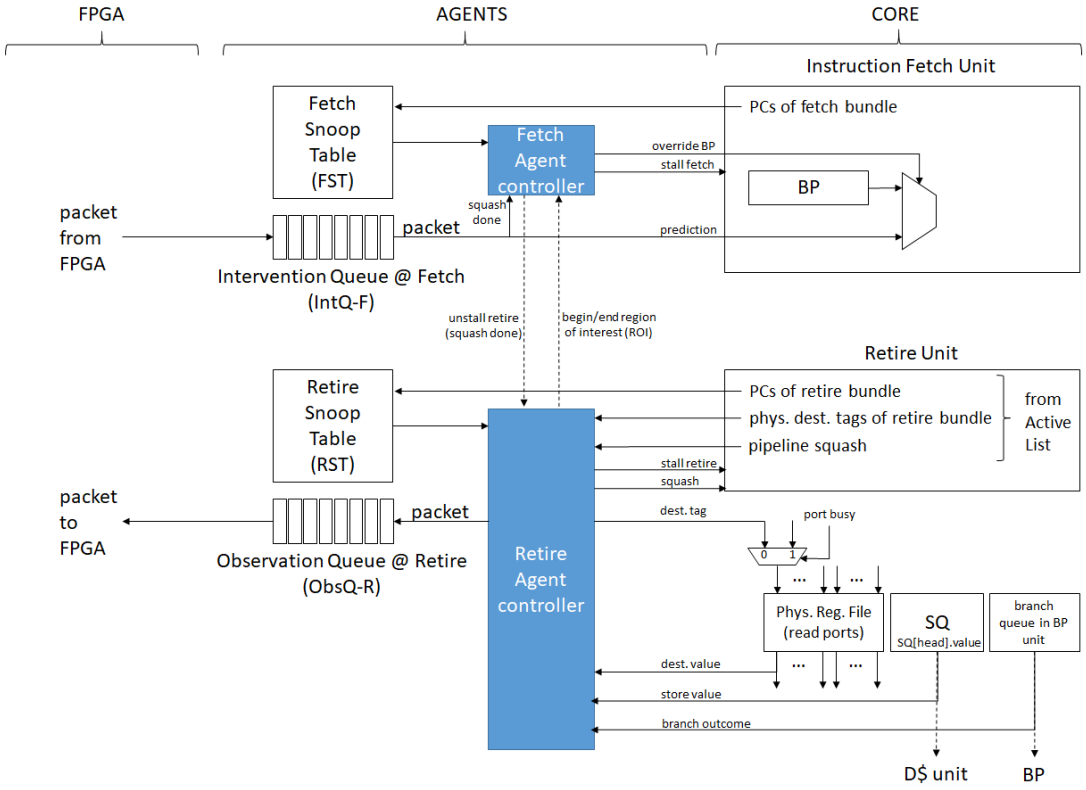

Figure 4: Fetch Agent (top, middle) and Retire Agent (bottom, middle).

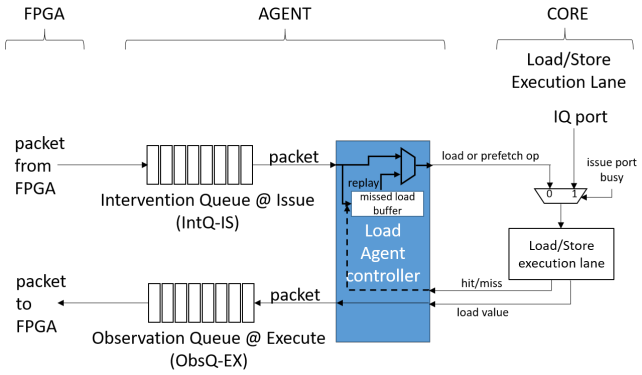

Figure 5: Load Agent.

and overturned if mispredicted. The Load Agent interprets received packets solely as loads or prefetches, not stores. Moreover, the Load Agent and core handle these loads specially: no wakeup, bypass, or write into the PRF, and no search of speculative store values in the store queue. Both prefetches and loads go through translation in the load/store execution lane.

One must consider impact on microarchitectural state in the context of speculation-based side channels. The custom component does not train any of the branch predictor structures. It does impact the cache by way of prefetches and loads, similar to other prefetchers.

The system must not allow one context's custom component in RF to observe another context in the core. This can be enforced by removing a context's custom component from RF and the Agents when that context is swapped out of the core. To support simultaneous multithreading, the Agents and custom components must be augmented with context ids.

A bug in a custom component may cause the Fetch Agent to stall the fetch unit indefinitely as it waits for a prediction that never arrives. The Fetch Agent can be augmented with a watchdog timer and a chicken-switch to disable the custom component. Alternatively, we have considered a different Fetch Agent design that does not stall if a prediction is late, but rather proceeds with the core's predictor and keeps count of how many late packets to drop when they eventually arrive.

3 METHODOLOGY

We use an in-house, RISC-V [21], execution-driven, cycle-level, execute-at-execute simulator to model the superscalar core, agents, and custom components. The superscalar core and memory hierarchy configuration is shown in Table 1. We use the top-weighted SimPoint [18] (100 million instructions) for each SPEC benchmark use-case. For the *bfs* use-case, from the GAP graph benchmark suite [3], we skip the graph setup phase and run 100 million instructions of the breadth-first search (roadNet-CA graph [12]) or

less if it completes in under 100 million instructions (com-Youtube graph [12]). In all results, performance of the core with the custom component is normalized to performance of just the core (which is at 0%).

branch predictor	64KB TAGE-SC-L [17]
pipeline depth	10 stages (fetch to retire)
fetch/retire width	4 instr./cycle
issue/execute width	8 instr./cycle
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU
ROB/IQ/LDQ/STQ/PRF	224/100/72/72/288
L1I cache	32KB, 8-way
L1D cache	32KB, 8-way, 3 cycles (1 agen, 2 hit)
L1D prefetcher	next-N-line (N=2)
L2 cache	256KB, 8-way, 12 cycles
L3 cache	8 MB, 16-way, 42 cycles
L2/L3 prefetcher	VLDP (5.5 Kb) [19]
DRAM	250 cycles

Table 1: Superscalar core and memory hierarchy configuration.

In the experiments, various parameters of the custom component (C, W, and D, below) and Agents (Q and P, below) are varied to explore their performance impact. These parameters and their notation are described below.

- clk*C*_w*W*: *C* is the factor by which the RF-synthesized custom component's clock frequency is slower than that of the core (CLK_{CORE} / CLK_{RF}). *W* is the custom component's superscalar width. The custom component can generate up to *W* predictions and push/pop up to *W* packets into/from the communication queues, in a given CLK_{RF} cycle.
- delay*D*: *D* is the pipelined execution latency of the RF-synthesized custom component, in CLK_{RF} cycles (*e.g.*, for configuration clk4_w4, delay8 would mean 8 CLK_{RF} cycles or 32 CLK_{CORE} cycles).
- queue*Q*: *Q* is the size of the Observation and Intervention queues in the Agents.
- port*P*: *P* describes which Physical Register File (PRF) ports the Retire Agent can contend on (*e.g.*, portLS means the Retire Agent can opportunistically use the PRF ports owned by the two load/store execution lanes, while portLS1 limits sharing to only one load/store execution lane's PRF ports).

The Load Agent's missed load buffer (MLB) is fixed at 64 entries.

4 PFM USE-CASES AND PERFORMANCE RESULTS

4.1 Astar

4.1.1 Astar Region of Interest (ROI). Figure 6 shows the ROI for *astar*. Wayobj::fill() calls wayobj::makebound2() repeatedly. On even calls (line 5), bound1p and bound2p are the input and output worklists, respectively. On odd calls, the worklists swap roles (line 9). The loop in wayobj::makebound2() (line 17) iterates through the input worklist. Each iteration obtains the next *index* from the input worklist (line 19), which corresponds to a cell in a grid. Then, eight indices, all using variable *index1*, are computed, corresponding to the neighboring cells of the cell at *index*. The nested-if template between lines 22–36 repeats seven more times with a different neighboring cell, *i.e.*, *index1*. Line 21 is the first instance of *index1*. As the 2D grid is linearized, the *index1* of neighbors in rows above or below the cell at *index* are computed by subtracting or adding a fixed value *yoffset* from *index*. The hard-to-predict branches for the first *index1* are at lines 22 and 23, referred to as the *waymap* and *maparp* branches (16 difficult branches in all). These branches are heavily mispredicted because the input worklist is very dynamic (recall, the output worklist of this call to wayobj::makebound2() is the input worklist to the next call to wayobj::makebound2()). The *waymap* branch tests if the neighboring cell at *index1* has already been visited since invoking wayobj::fill(). Notice wayobj::fill() sets a new *fillnum* at line 1 and this is the sentinel value to determine if a cell has already been visited over repeated calls to wayobj::makebound2(). If *index1* has not been visited yet and if the *maparp* condition at line 23 also holds, then *index1* is added to the output worklist (lines 25–26) and *index1* is marked as visited by storing *fillnum* at its entry in the *waymap* array (line 28). The store to *waymap* at line 28 causes a *potential* loop-carried memory dependency with the *waymap* branch at line 22 and all other seven instances of the *waymap* branch. An added challenge is that the loads that feed the branches may cache miss a lot, depending on the input dataset and hence the sizes of the *waymap* and *maparp* arrays.

```
wayobj::fill():
 1: fillnum++;
 2: ...
 3: while ((boundl!=0)&&(flend==false)) {
 4:   if (flodd==false) {
 5:     boundl=makebound2(boundlp,boundl,bound2p);
 6:     flodd=true;
 7:   }
 8:   else {
 9:     boundl=makebound2(bound2p,boundl,boundlp);
10:     flodd=false;
11:   }
12:   step++;
13: }


wayobj::makebound2():
14:  yoffset=maply;
15:  ...
16:  bound2l=0;
17:  for (i=0; i<boundll; i++)
18:  {
19:    index=boundlp[i];
20:
21:    index1=index-yoffset-1;
22:    if (waymap[index1].fillnum!=fillnum)
23:      if (maparp[index1]==0)
24:      {
25:        bound2p[bound2l]=index1;
26:        bound2l++;
27:
28:        waymap[index1].fillnum=fillnum;
29:        waymap[index1].num=step;
30:
31:        if (index1==endindex)
32:        {
33:          flend=true;
34:          return bound2l;
35:        }
36:      }
37:    ... Above nested-if template repeats
38:    ... 7 times for different index1's.
39:  }  // end for loop
```

Figure 6: *Astar* ROI.

4.1.2 Astar Custom Branch Predictor. Figure 7 shows the design of our custom *astar* branch predictor. The custom design is shown under the "FPGA" heading on the left. Loads and branch predictions sent to the Agent and core are shown in the middle under the "Agent/Core" heading. The "Program's Data Structures" are shown on the right under that heading.

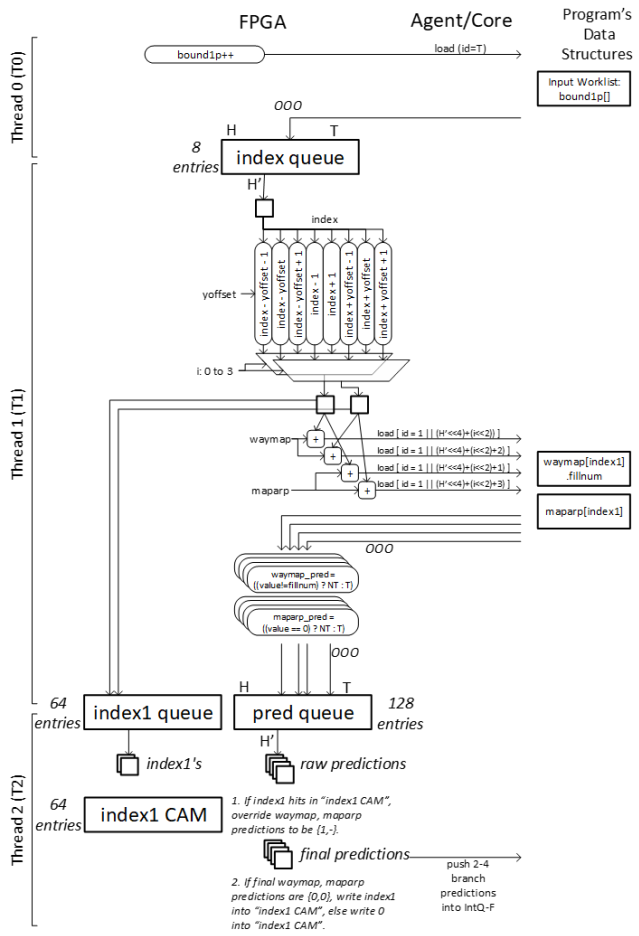

Figure 7: Custom *astar* branch predictor.

Line 1 is the beginning of the ROI and the RST is configured accordingly. Five key values need to be snooped from the Retire Agent: *fillnum* (line 1, snooped only once per call of wayobj::fill()), *yoffset* (line 14, snooped only once per call of wayobj::makebound2()), the base address of the input worklist (available from an instruction in the prologue of wayobj::makebound2() because the input worklist is an input argument), and the base addresses of the *waymap* and *maparp* arrays (lines 22-23, but obtained only once).

The custom design is implemented as three decoupled engines, which can be viewed as "threads" (T0–T2) implemented as fixed hardware. The threads are decoupled via two queues, index_queue and pred_queue. Each entry of index_queue eventually contains a single *index* from the input worklist, hence, the number of index_queue entries corresponds to the number of loop iterations the design may run ahead – its speculative scope. The pred_queue eventually contains all *waymap* and *maparp* branch predictions for this scope, therefore, it is sized commensurately. Each loop iteration/*index* corresponds to eight *index1*'s and 16 branch predictions (one *waymap* and one *maparp* for each *index1*). The example sizes of index_queue (8) and pred_queue (128), shown in Figure 7, are the default configuration (we also vary the scope in a sensitivity study) and configuration of the FPGA-synthesized design in Section 5.

The index_queue entry pointed-to by its commit head, H, corresponds to the oldest unretired iteration in the core. It is incremented when the design snoops an appropriate instruction, such as that which increments the loop induction variable. In steady state, when index_queue is continuously full, this frees up an entry so that the design can allocate a new iteration/*index* at its speculative tail, T. There is a one-to-one correspondence between each index_queue entry and each 16-entry segment of pred_queue, and this likewise applies to their commit heads and speculative tails. Thus, deallocation and allocation of index_queue entries implicitly performs the same for pred_queue and two other structures to be discussed.

When the index_queue tail entry is free, T0 pre-allocates the tail entry for a new *index* and issues a load to the input worklist to get the next *index*. Recall in Section 2.3 that loads from the custom component must be tagged with unique identifiers due to loads returning out-of-order (OOO) (hits-under-misses or misses returned OOO). In this case, the load is tagged with its pre-allocated entry number in the index_queue, *i.e.*, id=T as shown in Figure 7. (T0 then increments T.) Each index_queue entry has a valid bit and value. The valid bit is initialized to zero when the entry is pre-allocated. As T0's loads are returned, possibly OOO, by the Load Agent, their values are deposited in the entries indicated by their ids and corresponding valid bits are set.

T1 consumes valid entries from index_queue in order, at the speculative head entry, H′ (higher performance may be possible by relaxing this order, but this would incur more complexity). Each *index* thus obtained is used to compute eight *index1*'s. In turn, each *index1* is used to compute the addresses for a *waymap* load and a *maparp* load. Different designs are possible, in terms of width: the number of *index1*'s and corresponding loads generated each FPGA cycle. The example design in Figure 7 shows two *index1*'s generated each cycle and, correspondingly, four loads (two *waymap* and two *maparp*) pushed into the Load Agent's IntQ-IS each cycle. This example corresponds to the FPGA-synthesized design in Section 5. The identifiers of T1's loads are arithmetically derived from H′, the speculative head entry from where the root *index* was obtained. An additional 1-bit is prepended to the identifiers so that T1's load responses are steered to pred_queue, whereas, T0's load responses are steered to index_queue by virtue of not having this uppermost bit set. As T1's load responses are returned by the Load Agent (possibly OOO), the values of loads with even ids are used to compute *waymap* branch predicates and the values of loads with odd ids are used to compute *maparp* branch predicates. Predicates are deposited in the pred_queue at the locations indicated by their ids, and valid bits are set in these entries to signal predicates are available. Finally, as T1 generates *index1*'s, they are saved in the index1_queue, thus maintaining a record of each *index1* corresponding to each pair of *waymap* and *maparp* predicates.

T2 consumes a design-dependent number (four in the example shown in Figure 7) of consecutive valid predicates starting at the

speculative head entry, H′, of the pred_queue. These are referred to as raw predictions because they may have to be adjusted, based on inferring stores to *waymap[index1].fillnum* within the speculative scope of the design. To make this inference, the *index1*'s corresponding to the raw predictions are obtained from the index1_queue (entries of which are derived from H′ of pred_queue), and these *index1*'s are searched in the index1_CAM. If an *index1* hits in the index1_CAM, there was a previous visit to the same *index1* (within the maximum scope of the design) *and* the control-dependent region containing the store was logically entered for this visit, *i.e.*, we predicted [NT, NT] for the two branches in the previous visit. In this case, the *waymap* and *maparp* branch predictions for *this* visit to *index1*, should be overridden with [T, -] (the second branch is not even encountered and its prediction is discarded). The final predictions are pushed into the Fetch Agent's IntQ-F. For a width of four raw predictions, anywhere from two to four final predictions are pushed, depending on the *waymap* predictions in the bundle being taken or not-taken. Finally, if the *final* predictions for a given pair of *waymap* and *maparp* branches are [NT, NT], the implication is that there is logically a store for the branches' corresponding *index1*. In this case, *index1* is written into the index1_CAM (at the entry derived from H′ of the pred_queue); otherwise, 0 is written into it.

When the Retire Agent sends a squash packet (because the pipeline had a squash), only T2 needs to be rolled back. There is no need to redo the work performed by T0 and T1. T2 must be rolled back so that its predictions once again line up with where the core's fetch unit is in the dynamic instruction stream. Because the index1_CAM now reflects inferred-stores in the future, T2 cannot redo the work of converting raw predictions to final predictions as it initially had done. Therefore, final predictions are recorded in an additional queue and, after receiving a squash packet, those between the pred_queue commit head (H) and speculative head (H′) are replayed from this additional queue.

4.1.3 Astar Results. While the baseline *astar* suffers an MPKI (mispredictions per kilo instructions) of 31.9, the custom predictor is able to reduce MPKI to 1.04, thus removing the branch misprediction bottleneck and yielding good performance improvement. Figures 8, 9, and 10, show performance of the custom predictor over the baseline, for different parameters of the custom predictor (frequency differential, width, delay, and structure sizes) and Agents (sizes of intervention and observation queues, and PRF port sharing).

Figure 8 shows sensitivity of the custom predictor to its frequency differential with the core and its superscalar width (*C* and *W* parameters in clk*C*_w*W*), which combine to determine the custom predictor's bandwidth. The limitation of lower frequency of the RF-synthesized custom predictor can be overcome by increasing its width. At low bandwidths, the fetch stage will stall while waiting for the custom predictor to generate and supply predictions, thus reducing the speedup or even causing slowdowns compared to baseline execution (*e.g.*, configurations clk4_w1 and clk8_w1). With sufficient bandwidth, the predictions generated by the custom predictor can reach the core in a timely fashion to provide good speedup. Configurations clk4_w2, clk4_w3, and clk4_w4, provide IPC improvements of 99%, 155%, and 163%. The latter is slightly better than the 162% IPC improvement of perfect branch prediction (*perfBP* bar in the graph). This is possible because the custom predictor achieves higher MLP than the baseline core; higher MLP among loads issued by the predictor, translates into lower latency for the core's redundant loads via a prefetching effect.

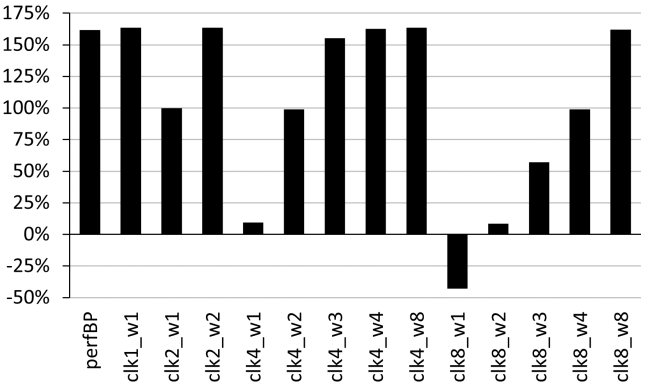

Figure 8: Speedup of the custom *astar* branch predictor for different C and W parameters. (All configurations: delay0, queue32, portALL; 8-entry index_queue.) *perfBP* is perfect branch prediction.

Table 2 shows (1) the percentage of fetched instructions in the ROI that hit in the Fetch Agent's FST and must be supplied custom predictions, and (2) the percentage of retired instructions in the ROI that hit in the Retire Agent's RST and must be observed by the custom predictor. These percentages are 15.5% and 20.3%. For the core's fetch and retire widths of 4 instructions/cycle and a 4x higher frequency (clk4), these percentages suggest average snoop rates of 2.48 and 3.25 instructions per RF cycle (multiply the percentages times 16 instructions per RF cycle). Assuming some degree of burstiness, these calculations support the effectiveness of clk4_w2 through clk4_w4 configurations.

% fetched instr. in ROI that hit in FST	15.5%
% retired instr. in ROI that hit in RST	20.3%

Table 2: *astar*: FST and RST snoop percentages.

Figure 9a shows that IPC improvement yielded by the custom predictor decreases as its pipelined execution latency increases. Increasing this parameter increases the startup delay at each call to wayobj::makebound2() and the penalty of synchronizing the core and custom predictor at pipeline squashes (via the squash/squash_done protocol). Infrequent squashes still occur, due to exiting the loop and due to speculative memory disambiguation within the core. Nonetheless, even at delay8 (8 RF cycles or 32 core cycles, for clk4), the custom predictor yields an IPC improvement of 138%.

Figure 9b shows that the performance is resistant to the size of the Agents' communication queues, while Figure 9c shows that PRF port availability is not an issue for this use-case.

Figure 10 shows that performance of the custom predictor is sensitive to the number of entries in its index_queue, *i.e.*, its speculative scope. An 8-entry index_queue (and corresponding 64-entry

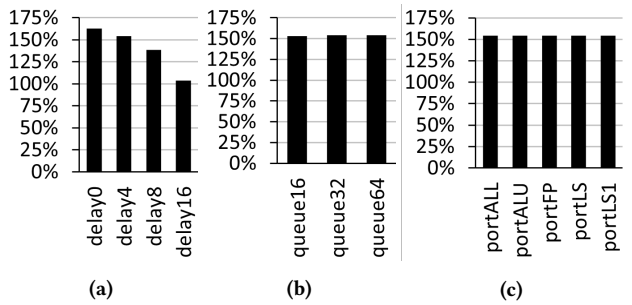

Figure 9: Speedup of the custom *astar* branch predictor for different D, Q, and P parameters. (a) clk4_w4, queue32, portALL. (b) clk4_w4, delay4, portALL. (c) clk4_w4, delay4, queue32. (All configurations: 8-entry index_queue.)

index1_queue, 64-entry index1_CAM, and 128-entry pred_queue) is adequate to achieve most of the speedup potential.

Summing up, for custom predictor parameters of clk4_w4, delay4, and 8-entry index_queue, and Agent parameters of queue32 and portLS1, IPC improvement is 154% (last bar of Figure 9c).

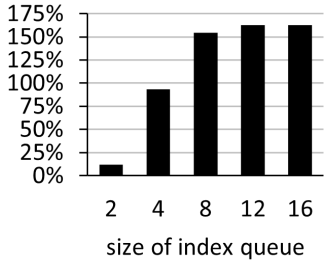

Figure 10: Speedup of the custom *astar* branch predictor, varying the number of entries in the index_queue. (All configurations: clk4_w4, delay4, queue32, portLS1.)

4.2 BFS

Applications with load-dependent loads defy conventional prefetchers because of their irregular access pattern. Performance is degraded further if these loads feed hard-to-predict branches. In this section, we present a custom component for Breadth-First Search (*bfs*) from the GAP graph benchmark suite [3], which exhibits this behavior.

It is shown in Fig. 11. The TDstep() function is shown at the left. The design is shown under heading "FPGA". It sends loads and branch predictions via "Agent/Core". The "program's data structures" are shown on the right.

The design is comprised of four decoupled engines, akin to "threads" T0–T3 implemented as fixed hardware. Decoupling achieves high memory level parallelism (MLP), *i.e.*, it enables many outstanding loads. T0 maintains a sliding window of nodes ("frontier queue") whose neighbors are to be visited, by issuing loads from the program's global frontier. The frontier queue decouples T0 and T1. T1 pops a node id U from the frontier queue. T1 uses U and the next consecutive node id U+1 to index into the program's offset array, to obtain the address of U's first neighbor ("a") and the address of U+1's first neighbor ("b"). The difference "b-a" is the number of neighbors of U. T1 pushes U's first neighbor's address into the "neighbor begin address queue" and the difference (number of neighbors) into the "neighbor trip-count queue". T2 pops an entry from both queues, and uses the first address and trip-count to load all of U's neighbors from the program's neighbor array. T2 deposits each neighbor, V, into the "neighbor queue". T2 also uses the trip-count to stream branch predictions for instances of the neighbor loop branch; the trip-count is hard to predict by the core's predictor due to different trip-counts among different nodes U. Finally, T3 pops each neighbor, V, from the neighbor queue, and issues a load for its visited-ness property from the program's properties array. T3 uses the loaded value to compute the predicate of the hard-to-predict visited branch. T3's visited branch predictions are interleaved with T2's neighbor loop branch predictions, in the Fetch Agent's IntQ-F.

The visited branch tests whether or not the neighbor V has been visited. If it has not been visited, the branch's control-dependent region is executed, including a store that marks V as visited. Thus, as was the case with *astar*, there is a potential loop-carried memory dependency between the visited store and the visited branch that guards it. Similar to how the *astar* design infers stores and overrides dependent predictions, T3 infers stores to the visited-ness property within its speculative scope (visited stores that have not been retired by the core yet). When T3 pops neighbor V from the neighbor queue, concurrently with issuing the load for its visited-ness property, it searches the neighbor queue for prior instances of V (between V's entry and the commit head, not shown). If there is a match, an older unretired conflicting store is inferred and the visited branch predicate is overridden as taken.

Experiments were performed with two input graphs, Roads (roadNet-CA) and Youtube (com-Youtube) [12]. The analysis that follows refers to only the Roads input. The first three bars of Fig. 12 show that both cache misses and branch mispredictions are debilitating. Importantly, both need to be attacked simultaneously. Perfect branch prediction alone yields only 11% speedup. Perfect data cache alone yields a significant 152% speedup, yet this is only a fraction of what the two together achieve: 426%. *Bfs*'s custom component achieves up to 125% speedup, due to (1) exploiting high MLP both within and among decoupled "threads" T0–T3 and (2) reducing branch MPKI from 19.1 to 0.5.

Fig. 12 shows speedups for different C and W parameters. Table 3 shows (1) percentage of fetched instructions in the ROI that hit in the Fetch Agent's FST and must be supplied custom predictions, and (2) percentage of retired instructions in the ROI that hit in the Retire Agent's RST and must be observed by the custom component. Trends are similar to those of *astar*. Despite observing a higher fraction of retired instructions than *astar*, performance is still good; moreover, clk4_w2 is somewhat closer to clk4_w4 than was observed for *astar*. This is likely due to more slack owing to residual miss latency.

% fetched instr. in ROI that hit in FST	13%
% retired instr. in ROI that hit in RST	31%

Table 3: *bfs*: FST and RST snoop percentages.

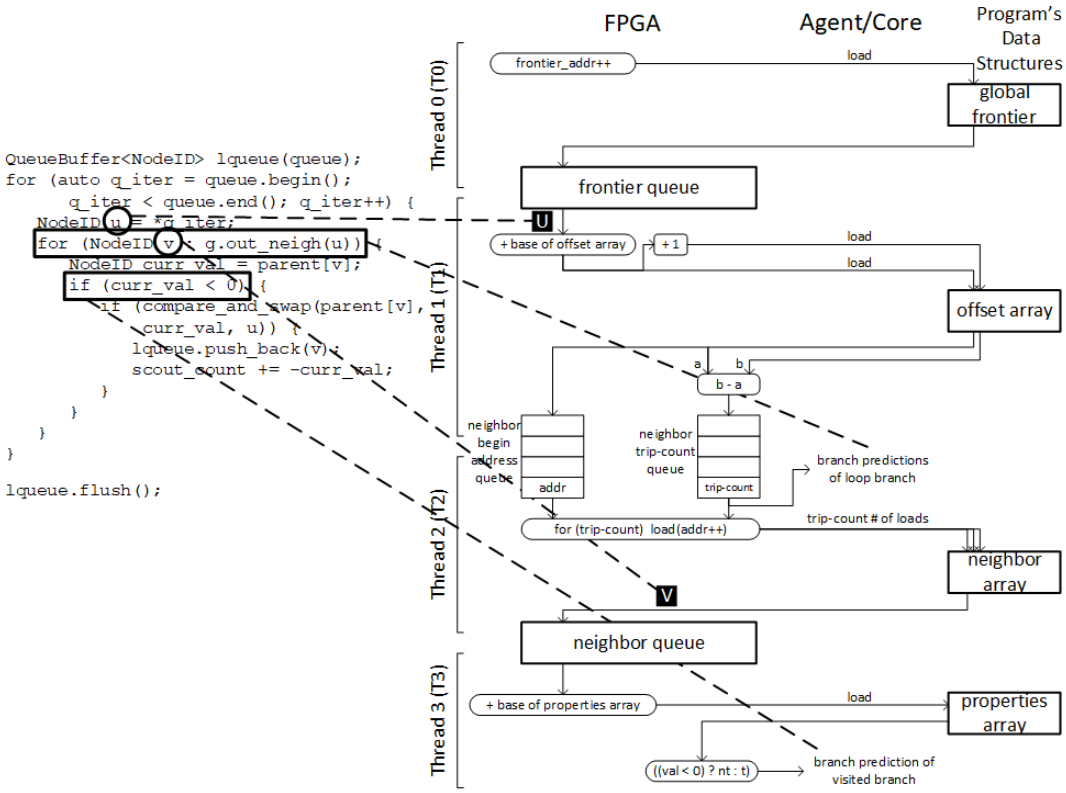

Figure 11: *bfs*'s custom component.

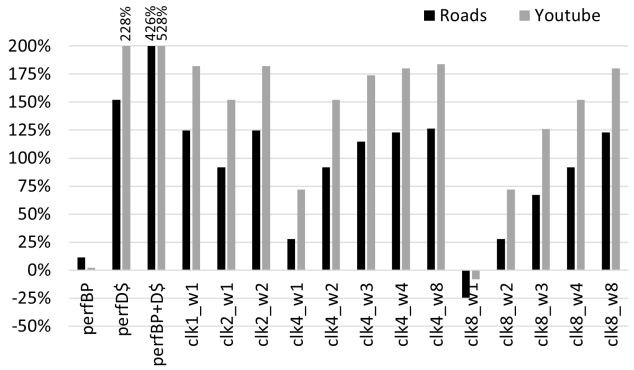

Figure 12: Speedup of *bfs*'s custom component, for different C and W parameters. (All configurations: delay0, queue32, portALL; 64-entry frontier, begin-address, trip-count, and neighbor queues.) *perfBP*, *perfD$*, and *perfBP+D$* are perfect branch prediction, perfect data cache, and both.

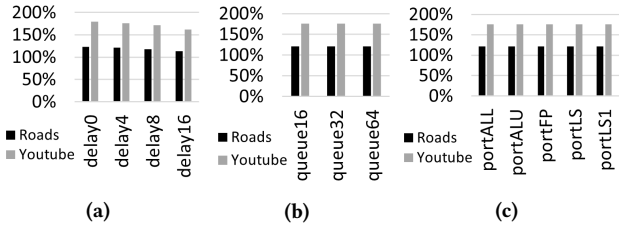

Figure 13: Speedup of *bfs*'s custom component for different D, Q, and P parameters. (a) clk4_w4, queue32, portALL. (b) clk4_w4, delay4, portALL. (c) clk4_w4, delay4, queue32. (All configurations: 64-entry frontier, begin-address, trip-count, and neighbor queues.)

Figure 13 shows low sensitivity to (a) the custom component's pipelined execution latency, (b) the size of Agents' communication queues, and (c) the Retire Agent's PRF port sharing options.

Figure 14 shows that the performance of *bfs*'s custom component scales with the number of entries in its frontier, begin-address, trip-count, and neighbor queues.

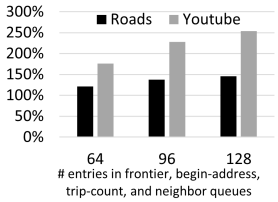

Figure 14: Speedup of *bfs*'s custom component, varying the number of entries in its frontier and other queues. (All configurations: clk4_w4, delay4, queue32, portLS1.)

4.3 Load Prefetching

PFM can be used to target applications which suffer high cache miss rates. Design of a custom prefetch engine allows proactive generation of extremely accurate prefetches while maintaining an optimal prefetch distance. We demonstrate this use-case here with the SPEC2006 *libquantum* benchmark. *Libquantum's* simplicity allows focusing on the fundamental ideas of PFM for prefetching. Later in this section, we briefly discuss other benchmarks that were also targeted for prefetching.

Libquantum has 2 delinquent loads, one each in the *quantum_toffoli* and *quantum_sigma_x* functions. Figure 15 shows the *quantum_toffoli* function where the delinquent load is marked as B. The PFM design to generate prefetches for load B is shown in Figure 16. It snoops, from the retire stream, the base address of the delinquent load, the iteration count (annotated A in Figure 15), and the stride, and uses a simple custom finite state machine (FSM) in the "Prefetch Generation Engine" to generate accurate prefetch OPs which are sent to the core via the Load Agent's IntQ-IS. The core executes the received prefetch OPs from the head of IntQ-IS when it finds a bubble in one of the load/store execution lanes.

```
void quantum_toffoli(..., quantum_reg *reg)
{
  ...        PFM enable
  for(i=0; i<reg->size; i++)        A
  {
    if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
    {                               B
      ....
    }
  }          PFM disable
  ...
}
```

Figure 15: *Libquantum's* ROI with delinquent load B.

Taking advantage of access to the retire stream, a simple sampling-based performance-feedback mechanism is used to adaptively update the prefetch distance to achieve optimal timeliness. The mechanism measures the number of retired instances of load B over a fixed number of cycles (epoch), a proxy for IPC. It then increases prefetch distance in the next epoch and observes if proxy-IPC increased. It keeps incrementing the prefetch distance as long as proxy-IPC continues to improve, and settles or backs off when it does not change or degrades, respectively. This adaptive policy works well for *libquantum*. We found that customizing not only the prefetch engine, but also the feedback mechanism, for each application, is a useful feature enabled by PFM.

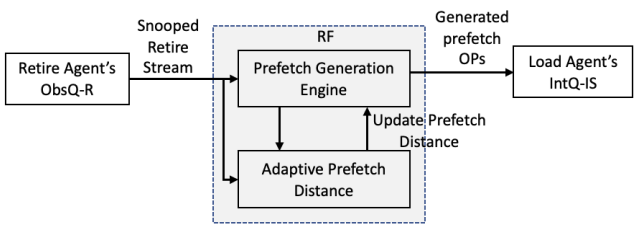

Figure 16: PFM design for load prefetching.

Custom prefetchers were developed for four other SPEC2006 benchmarks. The overall structure remains similar to the design shown in Figure 16, but the Prefetch Generation Engine is customized to be as simple, or as complex, as needed by the specific benchmark.

- *bwaves*: The delinquent loads are inside the innermost loop of five nested loops. Each load's address depends on a different set of four of the five induction variables or computational derivatives of them. As the induction variables change at different rates, each load's pattern is complex. The custom prefetcher features a complex FSM that nevertheless surgically follows the patterns.
- *lbm*: This benchmark has a cluster of delinquent loads that suffer from uneven latency reduction with the baseline prefetcher, resulting in the bottleneck simply shifting among different loads instead of getting removed. The customized prefetcher developed for *lbm* pushes, or skips (if IntQ-IS is full), the prefetch OPs for all delinquent loads in the cluster, *as a set*. This memory-level-parallelism (MLP) awareness is necessary for getting good performance.
- *milc*: This benchmark has a cluster of delinquent loads that access different cache lines, but each load is similar to that of *libquantum*. Thus, its prefetch engine and adaptive distance control are similar to *libquantum*'s design.
- *leslie*: This benchmark has multiple ROIs, each of which contributes significantly to total execution time due to load misses. Similar to *bwaves*, the loads in each ROI are nested inside two to four loops. FSMs were designed for three of these ROIs, following a similar strategy as the design of *bwaves*'s FSM.

Figure 17 shows performance of the five custom prefetchers for different C and W. Performance is very resistant to C (frequency difference between the core and custom prefetcher), W (superscalar width of the custom prefetcher), and D (pipelined execution latency of the custom prefetcher – not shown). This is partly due to the adaptive prefetch distance, and partly because, unlike custom branch prediction, the core's execution does not stall waiting for packets from RF. PRF port availability is not an issue for these use-cases (portLS1 performs as well as portALL – not shown).

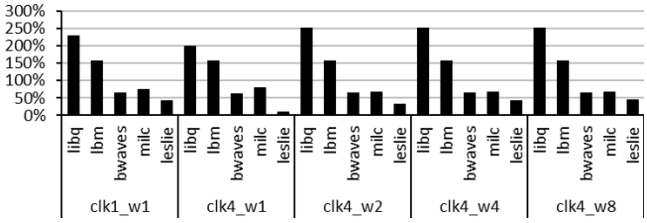

Figure 17: Speedup of custom prefetchers for different C and W (all configs: delay0, queue32, portALL).

5 FPGA AND POWER RESULTS

We synthesized several use-cases to FPGA (Xilinx Virtex UltraScale+ device xcvu3p-ffvc1517-3-e) to estimate hardware cost, power, and frequency.

We have results for four of the five custom prefetchers (all but *leslie*). High-level synthesis [5] (Vivado HLS 2018.1) was used to convert the prefetchers' C++ models to verilog. Width (W) is one

for the verilog implementations of the prefetchers. As observed in Section 4.3, clk1_w1 and clk4_w1 have similar performance.

We completed verilog design, FPGA implementation, and post-place-and-route simulation, verification, and power analysis, of *astar*'s custom branch predictor. Moreover, this implementation has width W=4: T0 and T1 generate 1 and 4 loads concurrently (up to 5 total per FPGA cycle) and T2 generates up to 4 predictions per FPGA cycle. The pipelined delay through each of the concurrent units, T0, T1, and T2, is four FPGA cycles, corresponding to delay4 in Fig. 9a.[1]

In addition to FPGA results for the *astar* microarchitecture described in Section 4.1, we also have FPGA results for *astar-alt*, the microarchitecture of which follows a different strategy than the one presented in this paper and yields 125% IPC improvement. Inspired by concepts from the EXACT branch predictor [1], it maintains two large predictor tables that mimic the program's underlying *waymap* and *maparp* arrays. It also populates its own output worklist as its input worklist is processed, and they swap roles at each call to *wayobj::makebound2()*. Thus, *astar-alt* mimics the program's data structures instead of issuing loads to them.[2] Details can be found in our earlier work [11]. Its large prediction tables are implemented with Block RAMs and, in spite of the complexity, we achieved timing closure at a good frequency.

Numbers of LUTs, FFs, BRAMs, *etc.*, frequency, and power, are shown in Table 4. Power reports use switching activity from post-place-and-route simulation, using stimuli generated from the superscalar processor simulator (packets from the Retire Agent's ObsQ-R and Load Agent's ObsQ-EX). The table separates out I/O power for informational purposes only, because it relates to FPGA pins which would not exist for FPGA embedded with the core. *Nonetheless, we include total dynamic and static power of the FPGA-synthesized designs in the following overall core+RF energy analysis presented in Figure 18.*

design	LUT	FF	BRAM	DSP	Freq (MHz)	dyn. power (mW)		static power (mW)
						logic	I/O	
astar (4wide)	6249	3523	0	0	500	251	338	865
astar-alt	1064	700	17.5	0	498	236	174	864
libq	282	215	0	0	690	8	45	861
lbm	169	204	0	0	628	6	44	861
bwaves	182	363	0	0	731	10	49	861
milc	253	667	0	4	628	38	115	861

Table 4: Hardware overhead using FPGA (xcvu3p-ffvc1517-3-e) for RF. *astar (4wide)* has width W=4 and an 8-entry index_queue. Other designs have width W=1. NOTE: *astar-alt* is a different microarchitecture than presented in Section 4.1 as explained above; it has two 32KB prediction tables and two 512-entry worklists.

[1]This excludes core-cycles that loads, generated by T0 and T1, spend executing in the core, which is naturally modeled separately from the delayD parameter (modeled on the core side).

[2]We shifted to the load-based strategy for several reasons. First, it is more robust to different input dataset sizes. Second, it combines prefetching/MLP-exploitation and branch prediction. Third, the load-based strategy may be templated in the future; notice that *astar* and *bfs* follow similar strategies, suggesting a possible path toward automation in the future, as discussed in Section 7.

Figure 18 shows the energy reduction that the custom components achieve over the baseline execution. Cumulative energy for the core is obtained using McPAT [13] and cumulative energy for a given FPGA-synthesized design is based on the per-RF-cycle energy obtained using the methodology above. The reduction in energy with respect to baseline execution can be attributed to (1) reduced misspeculation due to better branch prediction accuracy and (2) reduced static energy consumption due to shorter runtime.

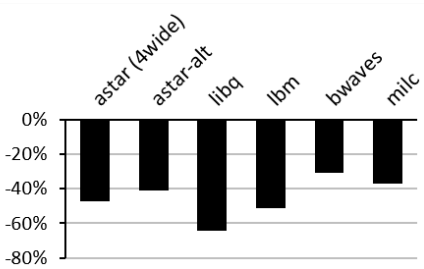

Figure 18: Energy of PFM designs (core+RF) normalized to baseline (core).

6 RELATED WORK

There have been several proposals to integrate reconfigurable logic with a core, at different levels of granularity. One approach is to tightly integrate reconfigurable logic as customizable functional units close to the execution unit of the core. Cores leverage the reconfigurable functional units via an expanded instruction set. Examples include Chimaera [25], PRISC [15], and OneChip [24]. Other proposals, for example, DySER [8] and BERET [9], map program subgraphs onto custom configurable compute engines (typically a grid of networked functional units) tightly integrated with the core. Loosely-coupled reconfigurable "co-processors" have also been proposed – for example, GARP [10], PRISM [2], and PipeRench [7] – where the execution of some compute-intensive loops might be offloaded to the reconfigurable co-processor. Reconfigurable logic has also been integrated in the context of multicore processors [22, 23] to accelerate multithreaded applications. A pool of specialized programmable logic is shared among several tiles of the CMP and can be configured for individual thread computation, producer-consumer communication, or fast barrier synchronizations.

The focus of all these prior works has been on mapping the execution of "hot loops/traces" on reconfigurable logic (with varied degrees of overheads depending on how tightly coupled the reconfigurable logic is with the core), which serve as accelerators for the targeted applications. Our work, on the other hand, targets the microarchitectural inefficiencies by allowing post-fabrication deployment of application-specific microarchitecture components.

Another set of related works include helper threads [4, 16, 26] or leader-follower microarchitectures like Slipstream [14, 20] and DLA [6]. We highlighted the challenges of generalized, automated branch pre-execution, such as implemented by the state-of-art Slipstream 2.0 [20], in Section 1.1. These unsolved challenges are also called out in that prior work. PFM, customization, and insights into the region of interest, offer a solution.

7 CONCLUSION AND FUTURE WORK

Processor companies rightly favor microarchitectural enhancements that improve performance generally. Targeting general performance is important but it is also constraining, in that large speedups on individual applications are possible with custom microarchitectural components. Post-Fabrication Microarchitecture (PFM) is a novel paradigm that enables superscalar processor designers to support both general performance improvement and large gains on individual applications: the former, via continuous generational improvements to the superscalar core; and the latter, via custom microarchitecture components synthesized to a reconfigurable fabric attached to key pipeline units through agents.

For future work, we would like to explore automating or assisting the generation of custom microarchitecture components. For example, the *astar* and *bfs* designs presented in this paper follow a similar strategy. If this could be templated, it suggests a path toward automation. The custom prefetchers in this paper are finite state machines (FSMs) that implement simple or complex arithmetic address patterns, so it is conceivable that a compiler could generate these FSMs. We would also like to explore architectures for the reconfigurable fabric (beyond traditional FPGA), such as including building blocks that suit the PFM paradigm and typical use-cases.

ACKNOWLEDGMENTS

This project is funded by the NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR) (NSF grant no. CCF-1823517 and matching Intel grant). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation or Intel Corporation.

REFERENCES

[1] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/1787275.1787321

[2] Peter M. Athanas and Harvey F. Silverman. 1993. Processor Reconfiguration Through Instruction-Set Metamorphosis. *Computer* 26, 3 (1993), 11–18. https://doi.org/10.1109/2.204677

[3] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. arXiv:1508.03619

[4] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/379240.379248

[5] Philippe Coussy and Adam Morawiec. 2010. *High-Level Synthesis: From Algorithm to Digital Circuit* (1st ed.). Springer Publishing Company, Incorporated.

[6] Alok Garg and Michael C. Huang. 2008. A Performance-Correctness Explicitly-Decoupled Architecture. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE Computer Society, USA, 306–317. https://doi.org/10.1109/MICRO.2008.4771800

[7] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*. IEEE Computer Society, USA, 28–39. https://doi.org/10.1109/ISCA.1999.765937

[8] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 17th Annual IEEE International Symposium on High Performance Computer Architecture (HPCA '11)*. 503–514. https://doi.org/10.1109/HPCA.2011.5749755

[9] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2155620.2155623

[10] John R. Hauser and John Wawrzynek. 1997. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 12–21. https://doi.org/10.1109/FPGA.1997.624600

[11] Chanchal Kumar, Aayush Chaudhary, Shubham Bhawalkar, Utkarsh Mathur, Saransh Jain, Adith Vastrad, and Eric Rotenberg. 2020. Post-Silicon Microarchitecture. *IEEE Computer Architecture Letters* 19, 1 (2020), 26–29. https://doi.org/10.1109/LCA.2020.2978841

[12] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[13] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[14] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. 2000. A Study of Slipstream Processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/360128.360155

[15] Rahul Razdan and Michael D. Smith. 1994. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*. Association for Computing Machinery, New York, NY, USA, 172–180. https://doi.org/10.1145/192724.192749

[16] Amir Roth and Gurindar S. Sohi. 2001. Speculative Data-Driven Multithreading. In *Proceedings of the 7th Annual IEEE International Symposium on High-Performance Computer Architecture (HPCA '01)*. 37–48. https://doi.org/10.1109/HPCA.2001.903250

[17] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. Seoul, South Korea. https://hal.inria.fr/hal-01354253

[18] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[19] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently Prefetching Complex Address Patterns. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2830772.2830793

[20] Vinesh Srinivasan, Rangeen Basu Roy Chowdhury, and Eric Rotenberg. 2020. Slipstream Processors Revisited: Exploiting Branch Sets. In *Proceedings of the 47th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA '20)*. IEEE Press, 105–117. https://doi.org/10.1109/ISCA45697.2020.00020

[21] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[22] Matthew A. Watkins and David H. Albonesi. 2011. ReMAP: A Reconfigurable Architecture for Chip Multiprocessors. *IEEE Micro* 31, 1 (2011), 65–77. https://doi.org/10.1109/MM.2011.14

[23] Matthew A. Watkins, Mark J. Cianchetti, and David H. Albonesi. 2008. Shared Reconfigurable Architectures for CMPs. In *2008 International Conference on Field Programmable Logic and Applications*. 299–304. https://doi.org/10.1109/FPL.2008.4629948

[24] Ralph D. Wittig and Paul Chow. 1996. OneChip: An FPGA Processor With Reconfigurable Logic. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. 126–135. https://doi.org/10.1109/FPGA.1996.564773

[25] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. 2000. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. Association for Computing Machinery, New York, NY, USA, 225–235. https://doi.org/10.1145/339647.339687

[26] Craig Zilles and Gurindar Sohi. 2001. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/379240.379246