Post-Silicon Microarchitecture

Chanchal Kumar, Aayush Chaudhary, Shubham Bhawalkar, Utkarsh Mathur, Saransh Jain, Adith Vastrad, and Eric Rotenberg

Abstract—Microprocessors are designed to provide good general performance across a range of benchmarks. As such, microarchitectural techniques which provide good speedup for only a small subset of applications are not attractive when designing a general-purpose core. We propose coupling a reconfigurable fabric with the CPU, on the same chip, via a simple and flexible interface to allow *post-silicon* development of application-specific microarchitectures. The interface supports observation and intervention at key pipeline stages of the CPU, so that exotic microarchitecture designs (with potentially narrow applicability) can be synthesized in the reconfigurable fabric and seem like components that were hardened into the core.

Index Terms—Adaptable architectures, microarchitecture, reconfigurable hardware

1 Introduction

It is difficult to make large gains in single-thread performance because of the following conundrum. On one hand, general microarchitecture techniques that work well over many applications (e.g., OOO execution, branch prediction, etc.) have been nearly exhausted. On the other hand, specialized microarchitecture techniques (e.g., application-customized branch predictors and prefetchers), which can yield big speedups on individual applications, are difficult to justify: they cannot all be included, and at the same time, one or a few cannot be included because of their narrow applicability. In this paper, we propose a novel microarchitecture paradigm called Post-Silicon Microarchitecture (PSM). The key idea is to define an efficient and flexible interface between key pipeline stages of a flagship superscalar core and reconfigurable logic (such as FPGA/CGRA). The PSM interface allows communication between the core and the reconfigurable fabric, so that application-specific branch predictors, prefetchers, etc., can be synthesized in the reconfigurable fabric and seem like components that were hardened into the flagship superscalar core. The ability to instantiate microarchitecture components after fabrication, increases the value proposition of deploying microarchitecture ideas on an individual application basis.

2 PSM Architecture

2.1 High-Level Overview

Fig. 1 shows the high-level diagram of PSM. Integrating a reconfigurable fabric (PSM-RF) on the same chip as a superscalar Out-of-Order (OOO) core gets complicated due to the potential frequency difference between the faster (but inflexible) core and the slower (but configurable) PSM-RF. To address this issue, an interface unit, PSM-Agent (PSM-A), is added between the core and PSM-RF.

PSM-A is tightly coupled with the core, has limited configurability (thus can run at the core's clock frequency), and acts as a communication medium between the core and PSM-RF. It provides *Observation* and *Intervention queues* between the Core and PSM-RF to provide generic 'message-passing' style of communication. Push/pop of data into/from the queues can happen at potentially different clock frequencies depending on the frequency difference between the core and PSM-RF.

A configuration bitstream shipped with the executable synthesizes the custom microarchitecture in PSM-RF and configures the communication behavior of PSM-A. The next section describes the components of the PSM Agent while also highlighting the support needed from the core.

- *C. Kumar, S. Bhawalkar, and E. Rotenberg are with the North Carolina State University, Raleigh, NC 27695. E-mail: {ckumar2, spbhawal, ericro}@ncsu.edu.*
- *A. Chaudhary, U. Mathur, S. Jain, and A. Vastrad worked on this project while they were graduate students at North Carolina State University, Raleigh, NC 27695. E-mail: {achaudh6, umathur, sjain22, asvastra}@ncsu.edu.*

Manuscript received 22 Jan. 2020; accepted 10 Feb. 2020. Date of publication 9 Mar. 2020; date of current version 7 Apr. 2020.
Recommended for acceptance by D. Sorin
Digital Object Identifier no. 10.1109/LCA.2020.2978841

2.2 PSM-Agent

The PSM Agent is designed to be simple, non-intrusive to the core, and agnostic to the choice of the reconfigurable fabric. It has several components, described below. The Observation and Intervention queues are used for communicating data to and from PSM-RF, while the *Snoop Tables* dictate how the queue payloads are constructed, as well as how the core's microarchitectural behavior is changed.

1) *Retire Snoop Table (RST):* RST is configured to allow PSM-RF to snoop key information from the retiring instructions. Each RST entry stores a PC (obtained via profiling of application and assembly), 6 configuration bits (shown in Table 1), and 2 'payload-type' bits ('Branch' and 'Destination Register'). When an instruction retires, its PC is checked against the entries in RST. If a matching entry is found, PSM-A constructs a payload and pushes it in the *Retire Observation Queue* (described below). The configuration bits of the matching entry are responsible for changing the mode of execution of the baseline core. If the 'Branch' bit is set in an entry, then the payload for the retiring (branch) instruction includes the actual T/NT direction. Similarly, for the 'Destination Register' bit, the payload includes the value of the destination register of the retiring instruction.
2) *Observation Queue at Retire (ObsQ-R):* Payloads constructed for retiring instructions are pushed in ObsQ-R. The payload consists of PC, the configuration bits, a T/NT flag (if 'Branch' bit was set), and a value field (if 'Destination Register' bit was set).
3) *Intervention Queue at Fetch (IntvQ-F):* PSM-RF uses this queue to send commands to the fetch unit of the core/PSM-A. The payload includes PC, a command (described in Table 2), and corresponding data.
4) *Fetch Snoop Table (FST):* Each FST entry stores the PC of branch instructions that are targeted by PSM for custom branch prediction, or the PC of the instruction used as the synchronization point. When 'Custom BP' mode is enabled, PSM-A searches the FST and head of IntvQ-F with the PC of the fetched instructions. If a matching entry is present in both FST and head of IntvQ-F, custom branch prediction sent by PSM-RF is used to override the default prediction (the head entry is also popped from IntvQ-F). If head entry in IntvQ-F doesn't match, then the core simply uses the default prediction. However, if FST has a matching entry but IntvQ-F is empty, this means that the prediction stream from PSM-RF is delayed, so fetch is stalled.
5) *Intervention Queue at Issue (IntvQ-IS):* PSM-RF uses IntvQ-IS to send Prefetch or Load OPs to the core for opportunistic execution. Each payload contains a command (LOAD or PREFETCH), the address, and the data size (for LOAD only). If the core finds a bubble in the load execution lane, it issues the OP at the head of IntvQ-IS. The issued OP flows

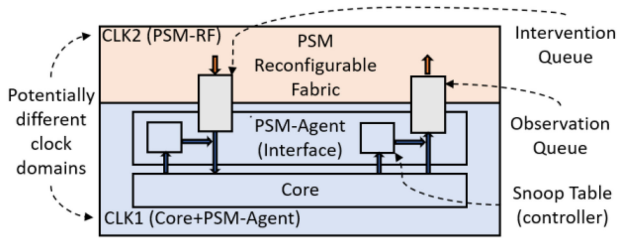

Fig. 1. High-level overview of the PSM architecture.

through the normal load pipe, but stays pinned at the head of the queue until the OP resolves.

6) *Observation Queue at Execute (ObsQ-EX):* If the core executed a Load OP received from PSM-A, it sends the loaded value to PSM-RF via ObsQ-EX.

3 PSM Use Cases

In this section, we describe several PSM designs for applications constrained by a high branch misprediction or cache miss rate. The region of interest which suffers from these constraints is identified and an application-specific design is developed to be synthesized on PSM-RF. The potential frequency difference between the core and PSM-RF, along with the latency of execution of the synthesized design on the slower PSM-RF, necessitates development of *decoupled* PSM designs which can run ahead of the core's instruction stream to provide *timely* predictions.

3.1 Custom EXACT Branch Predictor

Profiling the *astar* benchmark (from the SPEC2006 suite) reveals bottlenecks due to a high branch misprediction rate. *astar's* region of interest (ROI) is shown in Figs. 2a and 2b. The *makebound2* function works on an input worklist of indices in a 2D graph. For each index (C) in the input worklist, it checks if the 8 neighboring indices (D) have already been visited (E, F). If not previously visited, these indices are marked as visited (H) while adding them in the output worklist (G). In the next call to *makebound2*, the input and output worklists are switched (A, B).

The load-dependent branches (E, F) have a high branch misprediction rate which limit the IPC of *astar*. The conventional context used for branch prediction, PC and global branch history, fail to properly distinguish between the dynamic instances of these branches, thus performing poorly. The proper context to predict branches E and F would be the computed indices D (*index1*). A PSM design, by virtue of its ability to snoop key info from retiring instructions, allows decoupled generation and use of *index1* for predicting branches E and F.

We develop a custom decoupled branch predictor, inspired from the EXACT [1] branch predictor's active update mechanism, to predict branches E and F. Fig. 2c shows the high-level design of *astar's* custom branch predictor configured in PSM-RF. It has hardware structures (A, B) to mimic the input and output worklists. These worklists (and the *arch* pointer pointing to the 'current index') are kept in sync with the retire stream (albeit slightly delayed) by snooping the relevant instructions. An index can be read (C) from the input worklist (using speculative pointer *spec*) to generate the computed indices (D), which are used to look up simple direct-mapped predictors (E, F) for the *waymap* and *maparp* branches. The generated predictions are sent to PSM-A via IntvQ-F. The key idea here is that the *spec* pointer can run further ahead of the *arch* pointer (even far into the next worklist) and generate a custom branch prediction stream far ahead of the core's instruction stream. Use of the decoupled predictor, along with the proper context for prediction, *index1*, lets us generate accurate and timely predictions.

Results: The baseline configuration is shown in Table 3. While the baseline *astar* suffers an MPKI (Mispredictions Per Kilo Instructions) of ~32, the PSM design is able to reduce the MPKI to ~2. Fig. 3 shows the performance of the decoupled PSM design over baseline, for different PSM parameters. Fig. 3b shows the sensitivity of the PSM design to the bandwidth between core and PSM-RF. The limitation of slower frequency can be overcome by increasing the communication width. Fig. 3c shows that performance improvement reduces if the execution latency of the hardware synthesized on PSM-RF is too high. This is due to the high penalty of synchronizing the core and PSM-RF after a misprediction. Fig. 3d shows that the performance is resistant to the size of the communication queues.

3.2 Control Flow Decoupling

Control Flow Decoupling (CFD) [2] separates the branch slice from a branch's control-dependent instructions. The branch slice is pre-executed and the branch outcomes are pushed into an architectural Branch Queue, which is later used to determine the control flow of the control-dependent instructions. Due to the decoupled nature of CFD, it is a natural candidate for PSM. PSM can inject and execute the branch-slice instructions (in the "Enable Instruction Fetch" mode) while buffering the outcomes in PSM-RF, which are later streamed to the core to override the default predictions. PSM offers opportunities to target even the inseparable branches (which are not CFD-friendly) by keeping additional structures in PSM-RF to fix the control-flow dynamically. We use *astar* to show this use case.

TABLE 1
Configuration Flags in an RST Entry

Enable PSM mode	Signifies the start of region of interest (ROI); enables the communication queues
Full Squash mode	Squash the pipeline and direct core to do full squashes (from head of ROB) for branch mispredictions, instead of partial-flushes, until PSM is disabled (simplifies synzhronization of core & PSM-RF)
Custom BP	Directs the core to start consulting PSM-A to get custom branch predictions from PSM-RF (overrides core's default branch prediction)
Disable PSM mode	Signifies end of region of interest (ROI); disable any features enabled by PSM; return to baseline mode
Enable Instruction Fetch	Squash the core, take an architectural checkpoint, and direct the core to start fetching a stream of instructions from PSM-A, instead of the I-Cache
Disable Instruction Fetch	Restore the architectural checkpoint and redirect the core to fetch instructions from the I-Cache

TABLE 2
Intervention Commands at Fetch

BRANCH_DIR	Payload includes the custom branch prediction (T/NT) generated by PSM-RF
SYNC	Used as a synchronization point; the Fetch unit uses the default branch predictions until a fetched instruction's PC matches this payload's PC
DONE	PSM-RF has generated all predictions that it can; core uses default predictions for further branches
INSTRUCTION	Payload includes instruction generated by PSM-RF; core gets the instruction from PSM-A instead of the I-Cache (in the 'Enable Instruction Fetch' mode)

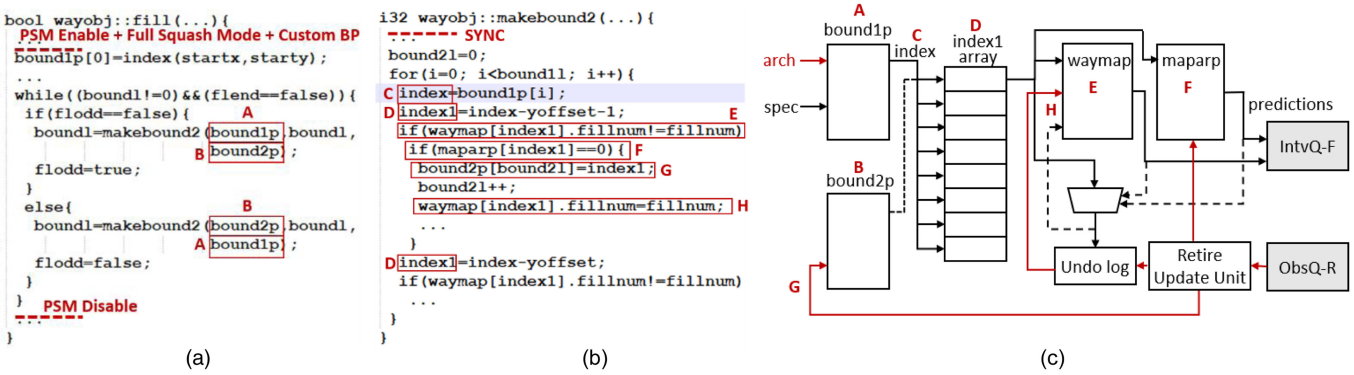

Fig. 2. (a) *astar's* ROI in which PSM is enabled. (b) Load-dependent branches E and F in the *makebound2* function. (c) Custom design in PSM-RF.

Fig. 4 shows the store D which makes the control flow (of branch B) dependent on the control-dependent instructions, making a simple implementation of CFD difficult. PSM-RF streams the program slice shown in Fig. 4 to the core, and snooping the relevant instructions (E,F,G,H) lets it determine the branch outcomes. A simple direct-mapped *index table* is kept in PSM-RF to fix the control flow. If both maparp *(C) branches are 'not-taken', the corresponding index entry is set. After snooping the* waymap *load (G), the index table is consulted: if the corresponding entry is already set, the* waymap *direction is forced 'taken' in the buffered branch outcomes.*

Results: CFD reduces the MPKI from ~32 to 0.23. The results in Fig. 5a show that the performance is very sensitive to the bandwidth between the core and PSM-RF. At low bandwidth, the injected CFD slice itself becomes a bottleneck due to low instructon fetch rate from PSM-RF. Adding a loop buffer in PSM-A would get rid of this issue. Due to the very low MPKI, performance is very resistant to the delay of the PSM design, as shown in Fig. 5b.

TABLE 3
Baseline System Parameters

Branch Prediction	BP: 64 KB TAGE-SC-L predictor; BTB: 4K entries, 4-way set-associative; RAS: 32 entries
Prefetcher	VLDP: 5.5 Kb
Memory Hierarchy	L1 I/D: split, 32 KB each, 8-way set-associative, 4-cycle access latency; L2: unified, 256 KB, 8-way set-associative, 12-cycle access latency; L3: 8 MB, 16-way set-associative, 42-cycle access latency; DRAM: 250-cycle access latency
Core	9-stage (Fetch to Retire) Out-of-Order; 4 instr./cycle Fetch/Retire; 8 instr./cycle Issue/Execute; ALUs: 4 Simple, 2 LS, 2 FP-Complex; ROB/IQ/LDQ/STQ/PRF: 224/100/72/72/288 entries

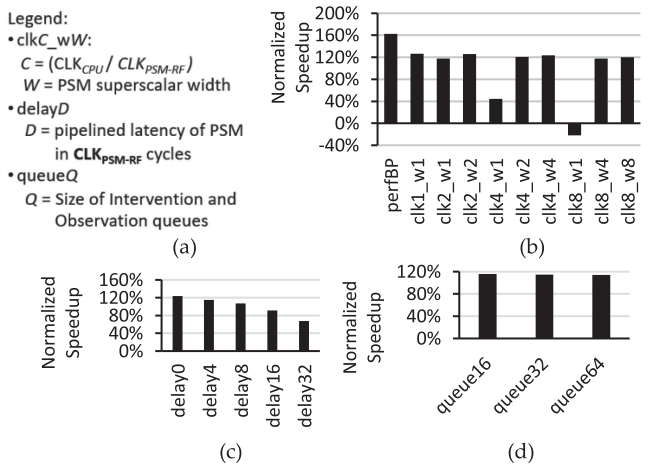

Fig. 3. (a) Legend: *C* is the factor by which PSM-RF's clock frequency is slower than that of the core; *W* is the PSM superscalar width (PSM-RF can generate *W* predictions and push/pop *W* payloads into/from the communication queues, in a given CLK_{PSM-RF} cycle); *D* is the pipelined execution latency of the design synthesized in PSM-RF, in CLK_{PSM-RF} cycles. (b) Speedup of PSM, normalized to baseline, for different *C* and *W* parameters (all configs are delay0, queue32); *perfBP* is the performance for perfect branch prediction. (c) Performance for different *D* parameters (all configs are clk4_w4, queue32; e.g., *delay8* means 8 CLK_{PSM-RF} cycles or 32 CLK_{CORE} cycles). (d) Performance for different *Q* parameters (all configs are clk4_w4, delay4). All configs in (b),(c), and (d) have 32KB predictors and 512-entry worklists.

3.3 Load Prefetching

Prefetching is a natural candidate for PSM as it already requires both accuracy and timeliness. Due to the space constraint, we present the simplest use case of prefetching using the *libquantum*

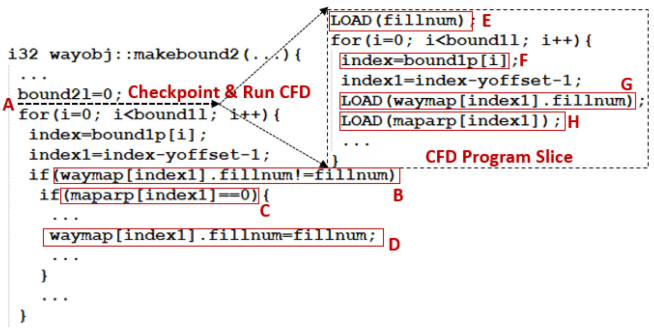

Fig. 4. CFD program slice injected in the core.

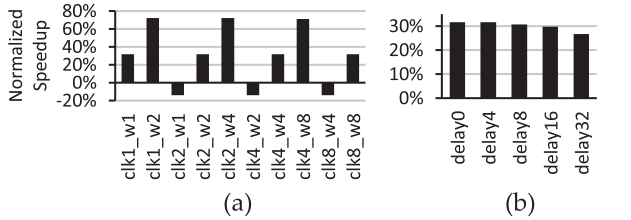

Fig. 5. Results for different PSM parameters (legend in Fig. 3a). (a) All configs are delay0, queue32. (b) All configs are clk4_w4, queue32.

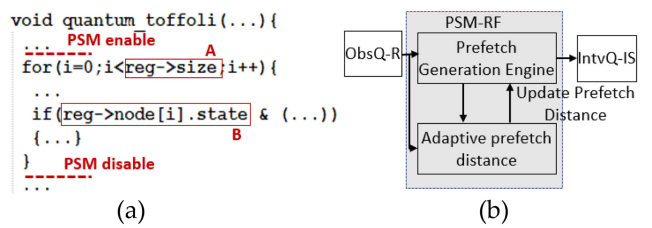

Fig. 6. (a) ROI with delinquent load B. (b) PSM design for prefetching.

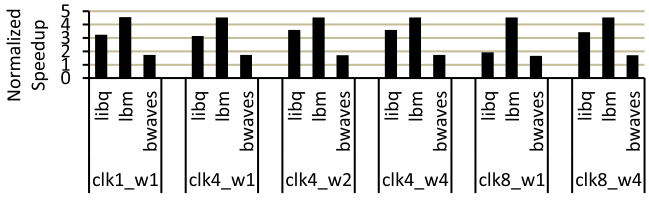

Fig. 7. Results for libquantum(*libq*), lbm and bwaves benchmarks (legend in Fig. 3a). All configs are delay0, queue32.

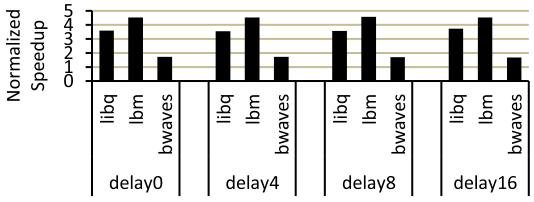

Fig. 8. Results for libquantum(*libq*), lbm and bwaves benchmarks (legend in Fig. 3a). All configs are clk4_w4, queue32.

benchmark from SPEC2006. Fig. 6a shows one of the ROIs with the delinquent load B (high cache miss rate). The PSM design to generate prefetches for load B is shown in Fig. 6b. PSM-RF snoops the base address along with the iteration count and the stride, from the retire stream, and uses a simple customized FSM (Prefetch Generation Engine) to generate accurate Prefetch OPs which are sent to the core via IntvQ-IS. A performance-feedback mechanism is used to adaptively update the prefetch distance to achieve optimal timeliness.

Results: Figs. 7 and 8 show the performance of using PSM for prefetching delinquent loads in 3 benchmarks from the SPEC2006 suite. Each benchmark uses its own customized Prefetch Generation Engine (with a range of complexity) and is able to achieve good performance improvement. Due to the adaptive prefetch distance, the performance is resistant to the bandwidth between the core and PSM-RF, as well as the execution latency of PSM-RF.

3.4 Load-Dependent Load Prefetching

PSM can be used to prefetch load-dependent loads (e.g., pointer chasing) as shown in Fig. 9. Load OPs can be sent to the core using IntvQ-IS and the loaded values can be snooped back by PSM-RF from ObsQ-EX. These loaded values can then be used to generate the prefetch stream for the load-dependent loads. Due to the structural hazard of being able to execute only 1 Load OP from IntvQ-IS, the load stream might get slowed down if a Load OP misses in the cache and gets stuck at the head of IntvQ-IS. To avoid this, a 'Prefetch Stream' runs ahead ('Load-delay Distance') and prefetches the loads so that IntvQ-IS is not blocked by the missing Load OPs from the 'Load Stream'.

We use this mechanism to prefetch the load-dependent loads in the *primal_bea_mpp* function (not shown) of *mcf* (from SPEC2017), to achieve 20 percent performance improvement (not shown) over baseline which is good considering the *primal_bea_mpp* function accounts for about 30 percent of the execution time in the baseline.

4 Related Work & Conclusion

There have been several proposals (e.g., [3], [4]) to tightly integrate reconfigurable logic with a core, on the same chip. The focus of these works has been on mapping the execution of 'hot loops/traces' on reconfigurable logic which serve as accelerators for targeted applications. Our work, on the other hand, targets the microarchitectural inefficiencies by allowing post-silicon deployment of application-specific microarchitecture components. We develop a

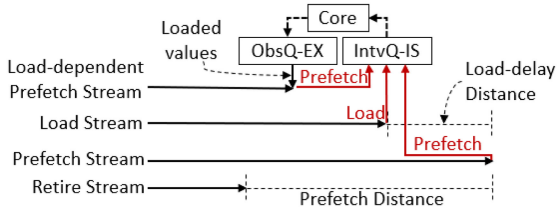

Fig. 9. Load-dependent load prefetching.

generic framework to communicate between the core and the potentially slower reconfigurable logic, with an interface that is simple, non-intrusive, and agnostic to the choice of reconfigurable fabric. We show that the latency of communicating with the reconfigurable logic requires development of decoupled designs that can satisfy the requirements of accuracy as well as timeliness. We demonstrate several use cases of this decoupled design to showcase the viability of Post-Silicon Microarchitecture.

Acknowledgments

This work was supported by NSF Grant CCF-1823517, and Grants from Intel.

References

[1] M. Al-Otoom, E. Forbes, and E. Rotenberg, "EXACT: Explicit dynamic-branch prediction with active updates," in *Proc. 7th Int. Conf. Comput. Frontiers*, 2010, pp. 165–176.
[2] R. Sheikh, J. Tuck, and E. Rotenberg, "Control flow decoupling," in *Proc. 45th Int. Symp. Microarchitecture*, 2012, pp. 329–340.
[3] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," *Proc. 5th IEEE Symp. Field-Programmable Custom Comput. Mach.*, 1997, pp. 12–21.
[4] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proc. 17th IEEE Int. Symp. High Perform. Comput. Architecture*, 2011, pp. 503–514.