A Physical Design Study of FabScalar-generated Superscalar Cores

Niket K. Choudhary, Brandon H. Dwiel, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{nkchoudh,bhdwiel,ericro}@ncsu.edu

***Abstract*—FabScalar is a recently published tool for automatically generating superscalar cores, of different pipeline widths, depths and sizes. The output of FabScalar is a synthesizable register-transfer-level (RTL) description of the desired core. While this capability makes sophisticated cores more accessible to designers and researchers, meaningful applications require reducing RTL descriptions to physical designs. This paper presents the first systematic physical design study of FabScalar-generated superscalar cores.**

I. Introduction

FabScalar is a recently published toolset for automatically composing synthesizable register-transfer-level (RTL) designs of diverse superscalar cores [1]. FabScalar is comprised of a *canonical superscalar template*, a *canonical pipeline stage library* (CPSL), and a *core generator*. The template defines a set of canonical pipeline stages, with composable interfaces, that make up a superscalar core. The CPSL provides many different RTL designs for each canonical pipeline stage, that differ in three major superscalar dimensions: (1) superscalar complexity (superscalar width and sizes of stage-specific structures), (2) subpipelining (depth of pipelining within a stage), and (3) stage-specific design choices. The core generator automatically composes a specified superscalar core by referencing the template and CPSL.

FabScalar is a big leap towards making sophisticated cores more accessible to designers and researchers, fueling more innovation. In most applications, however, an RTL description is not the end-game. Whether the use-case is a high-fidelity model for research, a hardware prototype for research, or a production system-on-chip for commercial applications, the RTL description must be reduced to a physical design.

In this paper, we present a physical design study of FabScalar-generated cores. Arguably, physical design is a significant portion of overall chip design cost [2]. In an academic setting, producing a commercially-representative physical design is an arduous task for a small research group, not to mention it likely has to be repeated for diverse cores. In keeping with FabScalar's virtue of increasing accessibility through automation, we make a point of heavily relying on automated synthesis and place-and-route (SPR).

Our test cases are two FabScalar-generated cores configured similarly to commercial Application Processors (APs) found in mobile devices: one is typical of current-generation APs, and the other, next-generation APs. We first characterize the quality of physical designs that can be achieved with fully synthesized memories and unmodifed RTL. Then, we compare different physical design options for memories. Highly-ported memories are pervasive in a superscalar core (rename map table, physical register file, issue queue, load and store queues, etc.), and presently we simply do not know how much effort needs to go into this important facet of physical design. Finally, we identify physical design imbalances and explore RTL adjustments to target them. The impact of each memory option and RTL adjustment is carefully measured: its frequency contribution, and its impact on instructions-per-cycle (IPC), power, and area. From this data, we identify different design points of the core (a design point is a particular combination of memory options and RTL adjustments) that lie on pareto-optimal performance or power frontiers. These frontiers provide guidelines to FabScalar users for tuning their cores, in the near term, and suggest profitable options to be included in future FabScalar releases, longer term.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes our methodology. Section IV presents results of memory options and RTL adjustments, applied in isolation and at the level of pipeline stages. This data is aggregated in Section V to identify different design points of the core that lie on pareto-optimal performance or power frontiers. Section VI concludes the paper.

II. Related Work

Choudhary et al. [3] did a cursory exploration of physical design of FabScalar-generated cores. For cycle-time validation, they compared cycle times of three commercial RISC superscalar processors with similarly configured FabScalar cores. The cycle times were obtained from synthesis only, not post-synthesis place-and-route. They also relied primarily on their FabMem memory compiler for implementing memories. Where place-and-route was applied, it was only to demonstate compatibility with a full ASIC flow, and metrics such as frequency, area, and power were not reported for the placed-and-routed core. Finally, no insight is provided with respect to physical design imbalances in the pipeline. In contrast, in this paper: all results are from post-synthesis place-and-route, we present other metrics besides frequency, a spectrum of physical design options are explored for memories, and we characterize physical design imbalances as a primary contribution as it guides RTL modifications.

TABLE I
MICROARCHITECTURE CONFIGURATIONS OF TWO REFERENCE CORES USED FOR THIS WORK. THE CONFIGURATION ONLY REFLECTS THE INTEGER PIPELINE. BTB: BRANCH TARGET BUFFER, BPB: BRANCH PREDICTION BUFFER, RAS: RETURN ADDRESS STACK. FU MIX: S=SIMPLE ALU, C=COMPLEX ALU, B=BRANCH, LD/ST=LOAD/STORE PIPELINE.

	Core-2W	Core-4W
Fetch / Dispatch Width	2	4
Issue Width	3	4
Functional Unit Mix	1S/C, 1B, 1Ld/St	1S, 1S/C, 1B, 1Ld/St
Fetch Queue	8	16
Issue Queue	16	24
Load / Store Queues	8 / 8	12 / 12
Reorder Buffer Size	64	96
BTB / BPB / RAS Size (# entries)	128 / 512 / 4	256 / 1024 / 8
L1 I-cache / L1 D-cache (KB)	16 / 16	32 / 32
fetch-to-execute pipeline depth (simple / load-store)	8 / 9	8 / 9

TABLE II
EDA TOOLS USED FOR ASIC DESIGN FLOW.

Phase	EDA tool(s) used
functional verification	Cadence NC-Verilog, vers. 06.20-s006
logic synthesis	Synopsys Design Compiler, vers. E-2010.12-SP2
place & route	Cadence SoC Encounter, vers. 7.1
spice simulation	HSPICE, vers. C-2009.03-SP1

TABLE III
BASELINE RESULTS FOR CORE-2W AND CORE-4W DESIGNS.

	Post-synthesis Freq. (MHz)	Post-layout Freq. (MHz)	Power (mW/MHz)	Area (mm^2)
Core-2W	1111	834	0.433	1.048
Core-4W	1000	667	0.635	2.052

Several other studies are related to this paper insofar as they explore the merits of SPR over custom design, consistent with FabScalar values. Except for caches, the AMD Bobcat designers employ an automated SPR flow for physical design [4]. The IBM POWER7 designers automated the layout of regular datapaths and memories through the use of Cadence SKILL scripts [2]. They also replicate memories as a way to extend the number of read and write ports with low custom-design effort. Chinnery and Keutzer [5] address the power and performance gap between ASIC and full custom design methodologies. They recommend techniques such as logic pipelining, alternate algorithmic implementations of logic cells, using dual supply voltages if the tools support it, etc.

III. METHODOLOGY

Using the FabScalar toolset, we generated RTL designs of two reference cores, a 2-way and a 4-way superscalar processor with respect to fetch width. These are referred to as Core-2W and Core-4W, respectively. Table I shows their microarchitectural configurations. We determined the microarchitectural configurations of Core-2W and Core-4W based on two guiding principles. Firstly, the two cores should represent the spectrum of commercial application processors in terms of superscalar width and key structures for exposing and exploiting instruction-level parallelism (ILP). The fetch width, issue width, and instruction window size of Core-2W and Core-4W are based closely on the AMD Bobcat [4] and ARM Cortex-A15 [6], respectively. Secondly, the microarchitectural resources should be balanced in terms of no one structure being the sole limiter of IPC.

The commercial CAD tools used for all experiments are shown in Table II. All designs are implemented using the Nangate 45nm open cell library [7]. For the L1 instruction and data caches, timing is obtained from CACTI 5.1 adjusted to the FreePDK BSIM4 predictive technology model [7] and the FabMem memory compiler is used to estimate the LEF geometry for layout [3]. For measuring IPC, we use the SPEC2000 integer benchmark suite and the cycle-accurate C++ simulator from the FabScalar toolset.

IV. RESULTS

At the outset, we synthesize and place-and-route Core-2W and Core-4W. All memory structures, except for the L1 caches, are synthesized to flip-flops. This initial exercise (Section IV-A) determines the frequency, power, and area that are possible with only automated SPR. Next, we explore two classes of techniques: 1) *optimizing memory structures* (Section IV-B) and 2) *adjusting the microarchitecture* (Section IV-C), in conjunction with SPR. The results reported are after full layout of the design using Cadence SoC Encounter.

A. Baseline physical design

To establish the baseline frequency, power, and area, we implemented Core-2W and Core-4W using SPR with no modifications to the FabScalar-generated RTL. Table III shows baseline results using the 45nm cell library. Increasing superscalar complexity from Core-2W to Core-4W decreases frequency by 20%, and increases area by 96%. Figure 1a shows the IPCs of various SPEC SimPoints on Core-2W and Core-4W. The number associated with each benchmark is its SimPoint id. On average, the IPC of Core-4W is 37% better than the IPC of Core-2W.

In both pipelines, Fetch-1 is the most timing critical stage. Its longest path is reading the banked BTB for all instructions in the fetch bundle, identifying the first predicted-taken branch instruction in the fetch bundle, and, if it is a call instruction, then updating the RAS with the call's return address. Investigating further, Figure 1b shows the slack in each pipeline stage. Slack reflects the logic imbalance that exists in different stages. The next most critical stages in both cores are Issue, Register Read, and the LSU (the second stage of load/store execution which involves searching the LQ/SQ). Compared to Core-2W, the Register Read and LSU slacks are lower in Core-4W, whereas slacks of other stages increased. This reflects a significant increase in complexity of the Register Read and LSU stages. In Register Read, the culprits are a larger physical register file and longer and more complex bypasses (spanning four execution lanes). The larger SQ, for store-to-load forwarding, impacts the LSU. In general, SPR

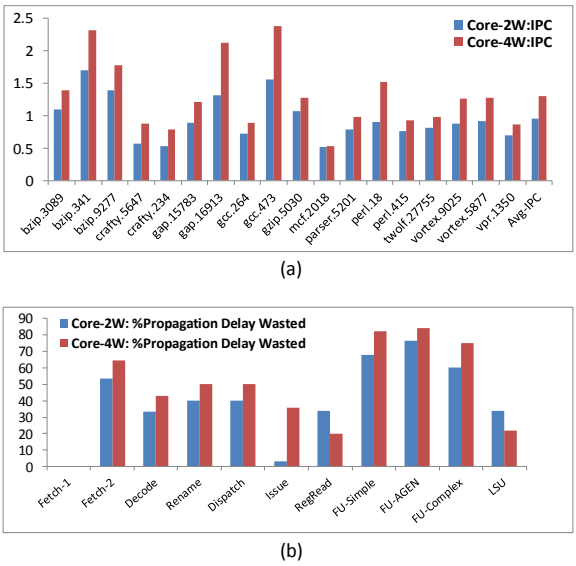

Fig. 1. (a) IPCs of Core-2W and Core-4W for different SPEC SimPoints. (b) Slack in each pipeline stage as a percentage of cycle time.

tools are very advanced in optimizing ALUs: it was very easy to achieve approximately 2.5GHz for the ALUs. CAD vendors provide pre-designed libraries for fast carry-save adders, carry-lookahead adders, and other common elements. However, SPR tools do not perform as well for wire-dominated stages. For instance, the authors spent a considerable amount of time to make the Register Read stage routable.

B. Optimizing memory structures

Multi-ported memories are pervasive in superscalar processors. They are often the dominant cycle time, power, and area contributors within their respective pipeline stages. Therefore, in high-end superscalar processors, each memory is typically a custom-designed SRAM or CAM [2]. In the baseline implementation, we synthesized memories to flip-flops. However, memories synthesized to flip-flops suffer from multiple inefficiencies. A flip-flop in a typical standard cell library has 25 to 30 transistors. An SRAM cell, on the other hand, uses 6 to 8 transistors per bit, yielding lower area and lower power. Moreover, large memories implemented with flip-flops suffer long access times.

Figure 2 compares frequency and power of memories synthesized to flip-flops and the same memories implemented in SRAM using the FabMem tool. All the memories are multi-ported (2 read, 2 write) and 4 bytes wide. Interestingly, delays of flip-flop-based memories are better or comparable for smaller sizes (less overhead at small sizes, e.g., no sense amps) but for larger sizes delays are much worse. For all configurations, flip-flop based memories are power inefficient. The inefficiency is low for small memories and grows significantly for large memories (the trend is similar for area). Moreover, the automated place & route tool suffers in

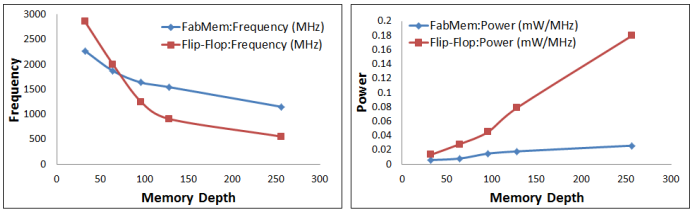

Fig. 2. Frequency and power of different depth memories synthesized to flip-flops vs. implemented in SRAM (from FabMem). Depth is varied from 64 to 256 words and the word size is 4 bytes. Each memory has 2-read and 2-write ports.

handling many wires localized in a small space. Multi-ported memories implemented with flip-flops have many local wires, attributed to large fan-in and fan-out of individual flip-flops because of multiple decoders (for writes) and multiplexors (for reads). Wire routing in custom-designed SRAM is optimized manually.

1) Implementing memories using level-sensitive latches: As a first approach to optimize memories, we implemented memories using level-sensitive latches. Latches are comprised of 12 to 14 transistors, about half the size of flip-flops. The drawback of using latches is that write-after-read hazards must be explicitly handled in certain pipeline stages. For example, the Rename Stage leverages the fact that, in a flip-flop implementation, RMT writes are synchronous with the clock edge and happen at the end of the clock cycle. Therefore, writes by younger instructions in the rename bundle do not interfere with reads by older instructions in the rename bundle, despite accessing the same logical register. With latches, however, the writes happen during the second half of the cycle, potentially interfering with concurrent reads. The solution is to defer the rename bundle's RMT updates to the next cycle, i.e., pipeline the RMT reads and writes from the same rename bundle. In turn, deferring the writes requires a second level of RMT bypasses to pass tags from the current rename bundle to the rename bundle that follows it; such bypasses already exist for intra-bundle dependences.

Thus, using latches required RTL modifications, however, the modifications were purely local to the affected modules and had no global impact.

2) Implementing memories using foundry memory compilers: In the second memory optimization, we explore using foundry memory compilers (hypothetically, since we use FabMem in place of a commercial memory compiler). The advantage of using a memory compiler is that it requires less effort than custom-designing an SRAM from scratch. Unfortunately, memory compilers are typically limited to one or two ports. (FabMem is not limited, but we are using it as if it were limited, for demonstration.)

To work around port limitations, the effect of more ports can be achieved by replicating SRAMs. In fact, this work-around is often employed in FPGAs with dual-ported block RAMs [8]. Figure 3 shows how it works. Figure 3a shows a 2R1W SRAM implemented with two 1R1W SRAMs. A write happens to both SRAMs so that two reads can access the same data in

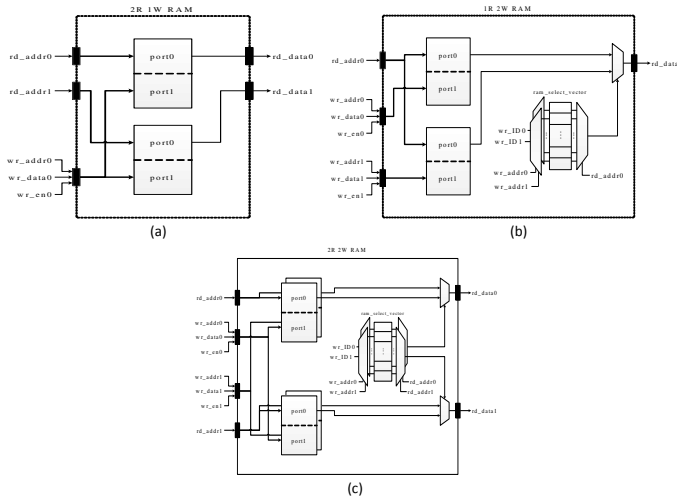

Fig. 3. Using dual-ported SRAMs to implement: (a) 2 read, 1 write, (b) 1 read, 2 write, (c) 2 read, 2 write.

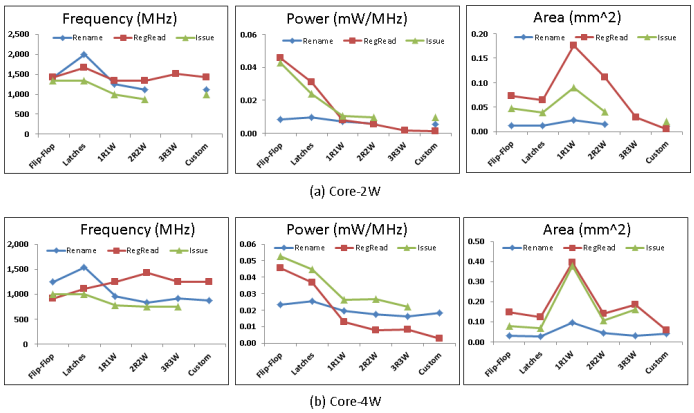

Fig. 4. Frequency, power, and area comparisons of different memory implementations, for the RMT in Rename Stage, Phys. Register File in Register Read Stage, and payload RAM in Issue Stage.

parallel. Figure 3b shows a 1R2W SRAM, also implemented with two 1R1W SRAMs. Each SRAM reflects writes from only one write port. A read consults the ram_select_vector (implemented with flip-flops) to know which SRAM was most recently written at the selected row. Figure 3c combines the two cases to construct a 2R2W SRAM. More generally, the number of 1R1W SRAMs needed is the number of logical read ports times the number of logical write ports. If SRAM building blocks with more than two ports are available, then the degree of replication required is less.

3) Memory optimization results: Figure 4 shows the frequency, power, and area of three pipeline stages individually: Rename, Register Read, and Issue. The only variation is in the implementation of their memories, specifically, the RMT in the case of Rename, Physical Register File in the case of Register Read, and payload RAM in the case of Issue.

Using latches instead of flip-flops substantially reduces delay in Rename, moderately in Register Read, and not much in Issue. Power is significantly reduced for Register Read and Issue, but increases a bit for Rename. Unexpectedly, area for flip-flops and latches are about the same. In hindsight, the memories are probably dominated by wires, decoders, and muxes to access the flip-flop and latch arrays with many ports.

The next three points implement the highly-ported memories by replicating 1R1W, 2R2W, or 3R3W building blocks generated by FabMem. The final point, "Custom", is meant to represent a custom SRAM with the exact number of ports required by the core, also generated by FabMem. 3R3W cannot be used for Core-2W's Rename and Issue Stages, hence, the corresponding points are intentionally missing. For Core-2W, flip-flops and latches have similar or better access times than SRAM blocks. The SRAM blocks yield much lower power for two of the stages, however. The 1R1W and 2R2W SRAM blocks yield worse area than latches and flip-flops. This is likely due to the fact that Core-2W's memories are sufficiently small with moderate number of ports (compared to Core-4W) such that the overhead of SRAM replication is too high, yielding worse access times and areas.

It seems the one case where SRAM blocks are favored for frequency, is in Core-4W's Physical Register File. It is the largest memory considered in this section and has 12 ports. Consequently, for Core-4W Register Read, 2R2W yields the highest frequency, low power, and area similar to latches and flip-flops.

C. Adjusting the microarchitecture

1) Pipelining timing critical stages: The frequency achieved by SPR can be boosted by pipelining timing-critical stages, at the cost of additional power and area. Increasing pipeline depth may also negatively impact IPC. In this section, we evaluate this cost/benefit tradeoff of pipelining.

Table IV shows the pipelining experiments that we performed for both cores. Fetch-1-Pipe1 delays pushing a call's return target onto the RAS by one cycle. If the return instruction is in the following fetch bundle, it obtains its target from a newly-introduced RAS bypass. Rename-Pipe1 and Rename-Pipe2 pipelines the Rename Stage into two or three cycles, and adds more levels of bypassing to handle cross-rename-bundle dependences. Issue-Pipe1 involves splitting wakeup-select-payloadRead logic into wakeup-select and payloadRead. This maintains a single-cycle wakeup-select loop, ensuring single-cycle producers and their consumers still execute in consecutive cycles. The cost, however, is that the select logic datapath is widened to include not only instructions' request/grant signals but also their tags. The select logic must simultaneously generate grants and steer the granted instruction's tag to the wakeup port, whereas previously the tag was obtained from the payload RAM after the select logic. RegRead-Pipe1 and RegRead-Pipe2 pipeline the Physical Register File into 2 and 3 stages, respectively. This further complicates the bypass network. Pipelining the LSU adds no additional bypass logic but increases the load-to-use latency. The Decode and Dispatch Stages are straightforward to pipeline, they only require additional pipeline registers. We

TABLE IV
PIPELINING EXPERIMENTS. EACH EXPERIMENT IS LABELED FOR FUTURE REFERENCE.

Stage	Experiment Performed
Fetch-1	(i) pipeline RAS update (Fetch-1-Pipe1)
Rename	(i) pipeline FL read and RMT write (Rename-Pipe1), (ii) pipeline FL read and RMT write into 2 stages (Rename-Pipe2)
Issue	(i) pipeline wakeup-select and payload-read (Issue-Pipe1)
RegRead (including bypass network)	(i) pipeline PRF memory in 2 stages (RegRead-Pipe1), (ii) pipeline PRF memory in 3 stages (RegRead-Pipe2)
LSU	(i) pipeline store-to-load forwarding in 2 stages (LSU-Pipe1), (ii) pipeline store-to-load forwarding in 3 stages (LSU-Pipe2)

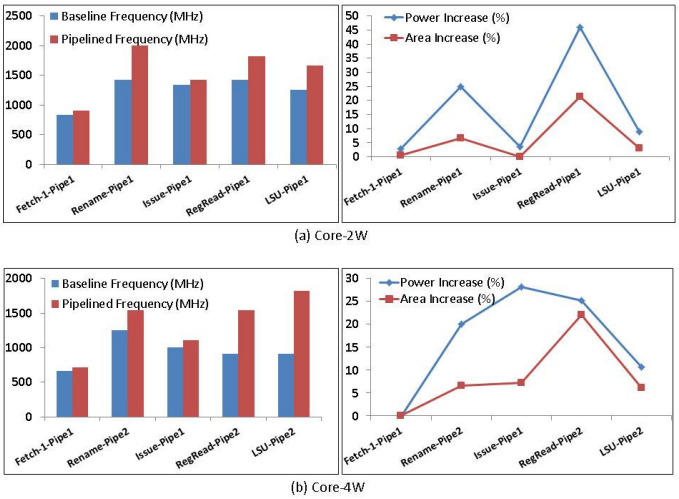

Fig. 5. (left) Frequency increase and (right) costs, for individual stages.

pipelined them as well and account for their overheads in the results.

The graphs on the left-hand side of Figure 5 show the per-stage frequency improvements of pipelining for the two cores. In both cores, Rename, Issue, Register Read, and LSU are initially close in frequency. The adjustment made to Issue is less effective than the adjustments made to the other three stages. Fetch-1 was already the most critical in both cores, and its adjustment did not substantially help. Consequently, these experiments show that Fetch-1 and Issue remain frequency bottlenecks. Therefore, we apply further adjustments to these in the sub-sections that follow.

The graphs on the right-hand side of Figure 5 show the power and area increase on a per-stage basis. Rename and Register Read suffer the largest power and area increases, which is not unexpected due to the increase in bypass complexity in both stages.

2) Distributing the Issue Queue: The monolithic IQ is best for IPC but it is timing critical [9]. We explore different IQ partitioning schemes for Core-2W and Core-4W (shown in Table V): Core-2W-IQP, Core-4W-IQP1 and Core-4W-IQP2. In Core-2W-IQP, the AGEN IQ holds load and store instructions and the INT IQ holds non-memory instructions.

TABLE V
IQ PARTITIONING SCHEMES IN CORE-2W AND CORE-4W. ALL THE SCHEMES EMPLOY ISSUE-PIPE1 IMPLEMENTATION.

Experiment	Description
Core-2W-IQP	Partitioned the IQ into INT:8, AGEN:8
Core-4W-IQP1	Partitioned the IQ into INT:16, AGEN:8
Core-4W-IQP2	Partitioned the IQ into INT0:8, INT1:8, AGEN:8

The same applies for Core-4W-IQP1. In Core-4W-IQP2, non-memory instructions are additionally split across INT0 and INT1 partitions based on a simple round-robin based policy for load balancing. In addition, for all schemes, cross-partition tag-broadcast (wakeup) requires an additional cycle, preventing a single-cycle producer and its consumer in a different partition from executing in consecutive cycles.

Figure 6 shows the frequency gain and IPC loss, respectively, of IQ partitioning. (The monolithic IQ designs for Core-2W and Core-4W are referred to as Core-2W-IQ and Core-4W-IQ, respectively). Note that all designs implement Issue-Pipe1. Core-2W-IQP achieves 17% higher frequency but the IPC degrades by 5%, on average, compared to Core-2W-IQ. Core-4W-IQP1 achieves 28.5% higher frequency but the IPC degrades by 6%, on average, compared to Core-4W-IQ. Core-4W-IQP2 achieves 50% higher frequency but the IPC degrades by 8%, on average, compared to Core-4W-IQ. Although average IPC degradation of Core-4W-IQP2 is only 8%, there are a few benchmarks that suffer significantly, e.g., bzip.3089, bzip.9277 and parser.5201.

3) Fetch-1: Fetch-1 remains a stubborn bottleneck because of the branch prediction logic. The other pipeline stages are able to achieve more than 1500MHz using various adjustments described thus far. Unfortunately, pipelining the branch prediction logic requires sophisticated algorithms [10] that balloon design effort as documented by others that implemented them in RTL [11].

We propose using multiple frequency domains (MFD) [12] to remove the Fetch-1 bottleneck. The Fetch-1 stage can operate at a slower frequency than the rest of the pipeline stages. State-of-art CAD tools are very advanced in handling MFD and cross frequency domain communication [13]. To compensate for a slower frequency, the fetch width can be increased. Thus, the Fetch-1 stage will deliver fetch bundles at a lower frequency but more instructions per fetch bundle. We explored two MFD-based design choices for Core-2W using Cadence SoC Encounter: Core-2W-MFD-Fetch-2W and Core-2W-MFD-Fetch-4W. In the former, Fetch-1 does not increase fetch bundle size (2) with respect to Core-2W and is clocked at half the frequency. In the latter, Fetch-1 doubles its fetch bundle size (to 4) and is clocked at half the frequency. We account for additional latency introduced by cross-domain synchronization buffers in IPC simulation.

Referring to Figure 7, Core-2W-MFD-Fetch-2W degrades IPC by 30%, on average, with respect to Core-2W. The average IPC degradation for Core-2W-MFD-Fetch-4W is only 3.5%. Increasing the Fetch-1 complexity from 2-wide to 4-wide increases the power of Core-2W by 6%.

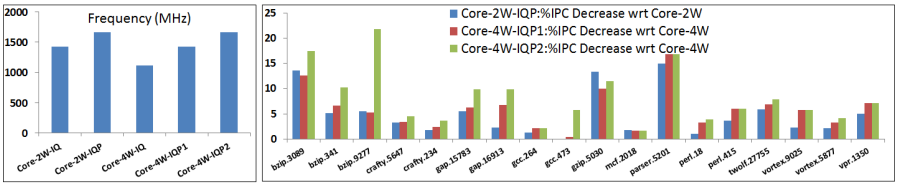

Fig. 6. (left) Frequency achieved due to IQ partitioning schemes in Core-2W and Core-4W. (right) %IPC reduction of each partitioned IQ design with respect to (wrt) monolithic IQ.

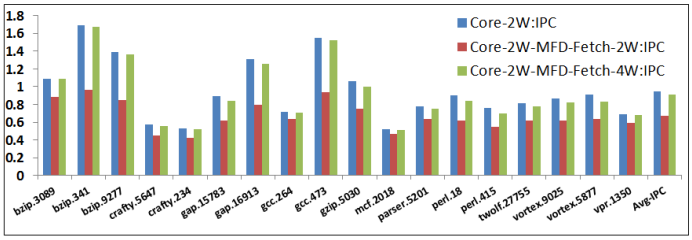

Fig. 7. IPCs of two MFD-based fetch designs for Core-2W and the baseline Core-2W.

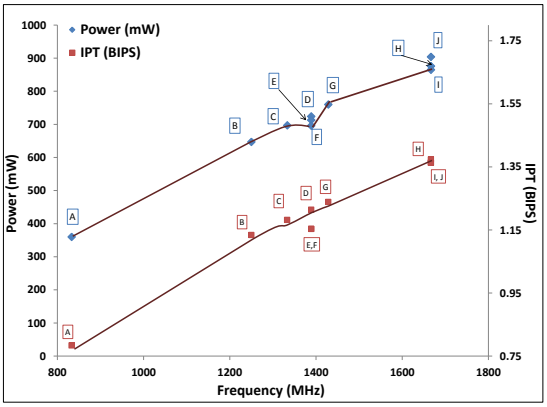

Fig. 8. Optimal frontiers of Core-2W designs that achieve maximum performance (BIPS) or minimum power (mW) for a given frequency.

V. Putting it All Together

The previous section explored different memory options and RTL adjustments to individual pipeline stages and quantified design quality in terms of frequency, power, and area. In this section, we put together individual techniques to achieve a certain frequency for the core as a whole. For example, suppose we want to target 1 GHz. For each pipeline stage, we search for options/adjustments that meet 1 GHz. Moreover, we measure performance and power of the core for each design point that achieves the target frequency. By varying the target, we can obtain optimal frontiers of design points that maximize performance or minimize power for a given frequency.

Figure 8 shows the optimal frontiers of Core-2W. Design points are labeled in the figure and explained in Table VI.

VI. Conclusion

This paper systematically investigated the additional RTL tuning required for FabScalar-generated cores to achieve a good quality physical design. We explored different memory options and RTL adjustments to individual pipeline stages to improve their frequency, in conjunction with SPR.

TABLE VI
Key for decoding the labels in Figure 8.

Label	Memory Options and RTL Adjustments
A	Baseline
B	MFD-Fetch-4W
C	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1
D	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF
E	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-FF
F	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-SRAM1R1W
G	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1
H	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-FF, Dispatch-Pipe1, Rename-Latch, RegRead-Latch
I	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-FF, Dispatch-Pipe1, Rename-Pipe1-FF, RegRead-Latch
J	MFD-Fetch-4W, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-FF, Dispatch-Pipe1, Rename-Latch, RegRead-Pipe1-FF

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was supported by NSF grant CCF-0811707 and gifts from Intel and IBM. 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.

References

[1] N. K. Choudhary et al., "FabScalar: Automating Superscalar Core Design," *IEEE Micro*, vol. 32, no. 3, May-June 2012.

[2] J. Friedrich et al., "Design Methodology for the IBM POWER7 Microprocessor," *IBM J. Res. Dev.*, 2011.

[3] N. K. Choudhary et al., "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template," in *ISCA*, 2011.

[4] B. Burgess et. al, "Bobcat: AMD's Low-Power x86 Processor," *IEEE Micro*, vol. 31, no. 2, 2011.

[5] D. G. Chinnery and K. Keutzer, "Closing the Gap Between ASIC and Custom: an ASIC Perspective," in *DAC*, 2000.

[6] "ARM Cortex-A15," *http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf*.

[7] J. Knudsen, "Nangate 45nm Open Cell Library," *CDNLive, EMEA*, 2008.

[8] B. H. Dwiel, N. K. Choudhary, and E. Rotenberg, "FPGA Modeling of Diverse Superscalar Processors," in *ISPASS*, 2012.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *ISCA*, 1997.

[10] A. Seznec et al., "Multiple-block Ahead Branch Predictors," in *ASPLOS*, 1996.

[11] J. Gandhi, "A Synthesizable RTL Model of a Pipelined Instruction Fetch Unit for Superscalar Processors," *M.S. Thesis, NCSU*, 2010.

[12] S. Dropsho et al., "Dynamically Trading Frequency for Complexity in a GALS Microprocessor," in *MICRO*, 2004.

[13] Cadence, "Clock Domain Crossing," *Technical paper*.