FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template

1. Introduction and Motivation

Many contemporary servers, personal and laptop computers, and even cell phones are powered by high-performance superscalar processors.A growing body of work has compiled a strong case for employing microarchitecturally diverse superscalar cores in both general-purpose and embedded systems.

- Future general-purpose microprocessors will consist of many cores. This makes it possible to provide multiple, differently-designed superscalar core types for streamlining the execution of sequential [10][13][18][19], parallel [6][17][20][27], and multiprogrammed [11][12] workloads, by exploiting diversity across and within applications.
- Application-specific superscalar processors [7] have been proposed for embedded systems running a small number of fixed tasks. Since the tasks are known at design time and do not change over the system's lifetime, key microarchitectural dimensions of an application-specific superscalar processor can be tailored to optimize performance and power.

Prior works in this area project significant performance and power advantages for microarchitecturally diverse superscalar cores, but several factors may impede progress in this direction. In the first case of general-purpose microprocessors, the design and verification effort for each additional core type is a major factor that could limit diversity. In the second case of embedded systems, the widespread proliferation of application-specific superscalar processors could be constrained by the niche expertise and large design teams required to design superscalar processors.

This paper describes an approach that aims to mitigate these factors and improve the prospects for employing microarchitecturally diverse superscalar cores. We converge upon a canonical view of a superscalar processor at the level of logical pipeline stages: fetch, decode, rename, dispatch, issue, etc. We call this a "canonical superscalar processor" and its logical pipeline stages are called "canonical pipeline stages". Within this framework, all superscalar processors have the same canonical structure, *i.e.*, each has a complete set of canonical pipeline stages and the same interfaces among them. Where they differ is in the underlying implementations of their canonical pipeline stages. A Canonical Pipeline Stage Library (CPSL) is populated with multiple designs for each canonical pipeline stage. A specific superscalar processor can be composed by selecting one design for each canonical pipeline stage from the CPSL and stitching together a complete set of canonical pipeline stages. This composition is automated due to invariant interfaces among canonical pipeline stages and the confinement of microarchitectural diversity within the canonical pipeline stages. Finally, microarchitectural diversity is focused along key dimensions that both define superscalar architecture and differentiate individual superscalar processors. Namely, the different designs of a given canonical pipeline stage varies along three major dimensions:

(1) *Superscalar complexity*: The superscalar complexity of a canonical pipeline stage is a product of its superscalar width (number of pipeline "ways") and the sizes of its associated ILP-extracting structures (*e.g.*, issue queue, physical register file, predictors, *etc.*). Increasing superscalar complexity may contribute to extracting more ILP in the program but typically increases the logic delay through the canonical pipeline stage. The effect of increasing logic delay on overall performance ultimately depends on the next differentiating factor.

(2) *Sub-pipelining*: A canonical pipeline stage is nominally one cycle in duration, but may be sub-pipelined deeper to achieve a higher clock frequency.

(3) *Stage-specific design choices*: Often there are multiple alternatives for handling certain microarchitectural issues, such as speculation alternatives, recovery alternatives, and so forth. These alternatives present a range of costs and benefits, moreover, the costs and benefits often depend on

specific instruction-level behavior in the program.

Our approach has been implemented in a novel toolset called FabScalar. FabScalar consists of a definition of the canonical superscalar processor, a CPSL containing many synthesizable register-transfer-level (RTL) designs of each canonical pipeline stage, and scripts for automatically composing the RTL designs of arbitrary superscalar cores by referencing the CPSL.

Validation experiments are performed along three fronts to evaluate the quality of RTL designs generated by FabScalar: functional and performance (instructions-per-cycle (IPC)) validation, timing validation (cycle time), and confirmation of suitability for standard ASIC flows.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 describes the methodology used throughout the paper. Section 4 describes FabScalar, including the canonical superscalar processor and the CPSL. Section 5 presents the validation experiments. Section 6 summarizes the paper and discusses future work.

2. Related Work

The Illinois Verilog Model (IVM) [27] provides the verilog for a semi-parameterizable 4-issue superscalar processor. Drawbacks of the current IVM are its unsynthesizable or poorly synthesizable (low frequency) verilog modules. More importantly, IVM's superscalar width and pipeline depth are inflexible.

Strozek and Brooks developed a framework for high level synthesis of very simple cores for embedded systems [24]. The Program-In-Chip-Out (PICO) framework out of HP labs [6] is closely related in that it customizes VLIW cores and non-programmable accelerators for embedded applications. Tensilica's Xtensa Configurable Processors automate the designer's task of customizing instructions, functional units, and even VLIW datapaths. FabScalar is unique in that it generates complex superscalar processors and this is evident in the novel composable CPSL.

3. Methodology

Table 1 shows the EDA tools used for functional verification, synthesis, and place-and-route. For synthesis, we used the FreePDK 45nm standard cell library [25].

Since specialized, highly-ported RAMs and CAMs are so pervasive and essential to a superscalar processor, we have developed a tool (FabMem) for generating their physical layouts and extracting timing, power, and area. For synthesis, RAMs and CAMs are integrated in the synthesizable verilog as LEF hard macros. For simulation, they are represented with behavioral modules.

Table 1. EDA environment: ASIC flow.

Phase	EDA tool(s) used
functional verification	Cadence NC-Verilog, vers. 06.20-s006
logic synthesis	Synopsys Design Compiler, vers. X-2005.09-SP3
place & route	Cadence SoC Encounter, vers. 7.1

Custom RAM and CAM macros are used for the rename map table, architectural map table, active list, free list, fetch queue (separates the decode and rename stages), issue queue wakeup CAM and payload RAM, physical register file, load queue CAM and RAM, and store queue CAM and RAM.

The level-1 (L1) instruction and data caches, branch target buffer (BTB), and conditional branch predictor are abstracted as macros, with timing information obtained from CACTI 5.1 [28].

Each canonical pipeline stage has many underlying implementations in the CPSL that differ in their complexity and sub-pipelining. For each design in the CPSL, we performed multiple synthesis runs with successively tighter timing constraints until the constraint could not be met. In this way, we converged upon the minimum propagation delay. Because of the sheer number of designs, delay numbers are from synthesis, not place-and-route. Nonetheless, Synopsys Design Compiler does estimate wire delays, using estimates of wire lengths based on a cursory place-and-route.

4. FabScalar

This section describes the FabScalar toolset, including the canonical superscalar processor and CPSL. The canonical superscalar processor defined by FabScalar is shown in Figure 1. It consists of nine canonical pipeline stages: Fetch, Decode, Rename, Dispatch, Issue, Register Read, Execute, Writeback, and Retire.

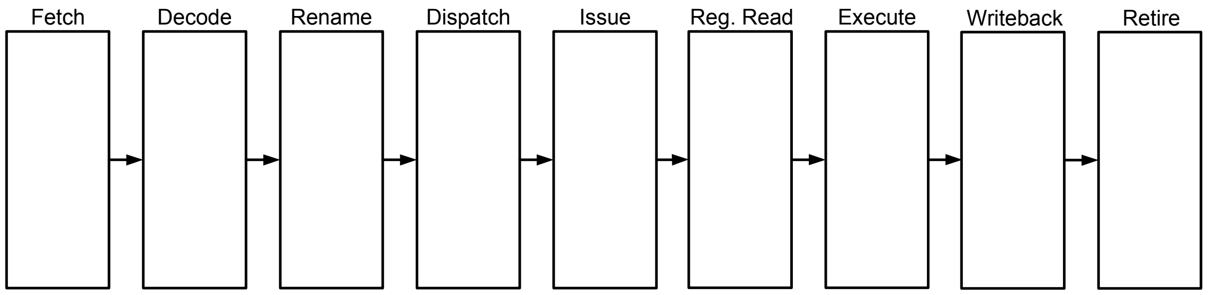

Figure 1. Canonical superscalar processor.

The canonical superscalar processor can represent either the MIPS R10K style of microarchitecture (issue queue instead of reservation stations, unified physical register file that is read after issuing, active list form of reorder buffer, etc.) [31] or the Tomasulo-style of microarchitecture [29] (reservation stations with data, reorder buffer with data, separate architectural register file, etc.). Currently, the CPSL is only populated with versions for the MIPS R10K style of microarchitecture. The CPSL instruction-set architecture (ISA) is PISA [5], a close derivative of the MIPS ISA.

The CPSL is populated with many synthesizable RTL designs for each canonical pipeline stage, that differ in their superscalar width, depth of sub-pipelining, and sizes of structures. Table 2 summarizes the microarchitectural diversity available in the current CPSL. The first column identifies the canonical pipeline stage. The second column shows ranges of width and depth of sub-pipelining. Sub-pipelining was guided by natural logic boundaries within each canonical pipeline stage design and timing results from synthesis (including RAM and CAM macros generated by FabMem).

The third column in Table 2 shows stage-specific structures that are implemented in the RTL. The L1 instruction and data caches are omitted from Table 2 because the L1 cache controllers are not yet implemented in RTL and are left for future work. Sizes of all stage-specific structures are parameterized in the RTL: since sizes can take on arbitrary values, no ranges are specified for structures in the third column of Table 2.

The final column in Table 2 considers another dimension for microarchitectural diversity, which we refer to broadly as "microarchitectural approaches". This dimension is a potpourri of design choices specific to each canonical pipeline stage. It is outside the scope of this paper to cover all of these techniques in the CPSL, at the level of synthesizable RTL. Nonetheless, we felt it would be of interest to enumerate notable examples in Table 2 to emphasize the potential for growing the CPSL in the future, and to underscore the specificity with which microarchitectural diversity can be targeted to specific instruction-level behavior. Specific design choices that are represented in the current CPSL are highlighted in boldface in the last column of Table 2.

Table 2. Overview of canonical pipeline stage designs available in the CPSL.

Canonical pipeline stage	Dimensions (W=width, D=depth)	Stage-specific structures (sizes parameterized in RTL)	Microarchitectural approaches
Fetch	W = 1 to 8, D = 2 to 5 Fetch-1: 1 or 2 sub-stages Fetch-2: 1 to 3 sub-stages	Branch or pattern history table (BHT or PHT) Branch target buffer (BTB) Return address stack (RAS)	**Branch prediction algorithm** No interleaving vs. **2-way interleaving** **Block-based BTB** vs. **interleaved BTB** Multi-cycle fetch: unpipelined **pipelined using block-ahead prediction [22]**
Decode	W = 1 to 8, D = 1 to 3	Fetch queue	Micro-operation cracking Non-speculative vs. speculative decode (if variable length ISA)

Rename &
Retire
	W = 1 to 8, D = 1 to 3
	W = 1 to 8, D = 2
		Rename map table (RMT)
		Architectural map table (AMT)
		# Shadow map tables: 0 or 4
		Free list
		Active list
		Phys. reg. ready bit table
			AMT vs. no AMT
			Branch misprediction recovery
			checkpoint (shadow map)
			handle like exception
			walk active list forward from head
			walk active list backward from tail
			Exception recovery
			restore RMT using AMT
			restore RMT by walking active list backward
			Freeing registers
			read prev. mapping from RMT, active list pushes freelist
			read prev. mapping from AMT, AMT pushes freelist

Dispatch	W = 1 to 8, D = 1	Issue queue (IQ) freelist	Collapsing IQ vs. **freelist based IQ**
Issue	W = 4 to 8, D = 1 to 3 Sub-pipelining variants: 1/1, 2/2, 2/1, 3/2 (*see text for explanation*)	Issue queue (IQ)	In-order vs. **out-of-order** Collapsing IQ vs. **freelist based IQ** Multiple schedulers vs. **single scheduler** Pipelined wakeup+select: **1-cycle producers non-speculatively wakeup dependents** 1-cycle producers speculatively wakeup dependents [4] Load hit/miss: predict hit always **predict miss always** hit predictor Load SQ conflict (with unknown store address): **predict no SQ conflict always** predict SQ conflict always memory dependence predictor Recovery for spec. wakeup & load conflict spec.: replay from IQ replay from replay buffer **handle like exception (squash)** Split stores
Register Read	W = 4 to 8, D = 1 to 4	Physical register file	n/a
Execute	W = 4 to 8, D = FU specific # simple ALU: 1 to 5, D = 1 # complex ALU: 1, D = 3 # load/store ports: 1, D = 2 # branch units: 1, D=1	Load queue (LQ) Store queue (SQ)	**Store-load forwarding** vs. no forwarding Many LQ/SQ designs possible for reducing associative searches (NLQ, SVW, SQIP)

Writeback/Bypass	W = 4 to 8
	D = *matches Register Read*
		n/a	**Full bypasses** vs. hierarchical or partial bypasses

5. Results: Validation

We evaluate the quality of the RTL designs produced by FabScalar. Prior to the validation experiments, we performed extensive unit-level testing of the CPSL. Validation is performed along three fronts:

1. *Functional and IPC validation*: A dozen different cores are generated, covering a range of widths, sizes, and depths. 100 million instruction SimPoints [23] of six SPEC2000 integer benchmarks are

executed on the cores and the instructions-per-cycle (IPC) results are within expected ranges and follow expected trends.
2. *Timing validation*: We also evaluate the quality of the RTL and FabMem in terms of cycle time. To validate cycle time, we compare several commercial general-purpose and embedded cores with similarly configured FabScalar generated cores.
3. *Suitability for physical design*: We demonstrate the suitability of the generated RTL for full synthesis and place-and-route by a standard ASIC and FPGA flow.

5.1. Functional and IPC Validation

The FabScalar tool was used to generate the RTL designs for the twelve cores described in Table 3. Before discussing the cores, few points about the table need clarification. First, some stage depths are omitted from the table if they are not varied for this experiment. Second, the quoted *fetch-to-execute pipeline depth* reflects the minimum branch misprediction penalty, and includes 1 cycle of execution in the branch unit (Writeback and Retire depths are excluded from this number).

Cores 1 through 6 were selected primarily to explore stage widths and structure sizes. Cores 6 through 10 aim to explore depths of stages and the fetch-to-execute pipeline depth. Cores 9 and 10 are unique in that their Fetch-1 sub-stage of Fetch is pipelined into two cycles, using block-ahead branch prediction. This yields a total Fetch depth of three cycles. Core-10 is the deepest of the twelve cores (fetch-to-execute = 15). Cores 11 and 12 are the same as Cores 1 and 2, respectively, except they use the gshare [15] instead of the bimodal branch predictor.

Results of executing the 100 million instruction SimPoints of six benchmarks are shown in Figure 2. Results are shown for both RTL ("Verilog") and the cycle-accurate C++ simulator ("C++"). Block-ahead prediction is not yet implemented in the C++ so its datapoints are missing for Cores 9/10. The first thing to note is that the cores execute the benchmarks successfully. Second, IPCs are within the norm for SPEC integer benchmarks, especially considering the conservative method for recovering from load violations and branch mispredictions employed by these cores . Third, the RTL and C++ follow each other closely. The latter result increases confidence in the RTL modeling of the design: if performance anomolies are observed, they are more likely inherent in the design rather than in the RTL modeling of the design.

Table 3. Cores used for functional and IPC validation experiments.

	Core-1	Core-2	Core-3	Core-4	Core-5	Core-6	Core-7	Core-8	Core-9	Core-10	Core-11	Core-12
Fetch/Decode/Rename/Dispatch width	4	4	5	6	8	2	4	4	6	6	4	4
Issue/RR/Execute/WB/Retire width	4	6	5	6	8	4	4	4	6	6	4	6
function unit mix (simple, complex, branch, load/store)	1,1,1,1	3,1,1,1	2,1,1,1	3,1,1,1	5,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	3,1,1,1	3,1,1,1	1,1,1,1	3,1,1,1
fetch queue	16	16	32	32	64	8	16	16	32	32	16	16
active list (ROB)	128	128	128	256	512	64	128	128	256	256	128	128
physical register file (PRF)	96	128	128	192	512	64	96	96	192	192	96	128
issue queue (IQ)	32	32	32	64	128	16	16	32	64	64	32	32
load queue / store queue (LQ/SQ)	32 / 32	32 / 32	32 / 32	32 / 32	32 / 32	16 / 16	32 / 32	32 / 32	32 / 32	32 / 32	32 / 32	32 / 32
branch predictor	bimodal								bimodal with block-ahead		Gshare	
branch history table (BHT) (# entries)	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
branch target buffer (BTB) (# entries)	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K
return address stack (RAS)	16	16	16	32	64	8	16	16	32	32	16	16
branch order buffer (BOB)	16	16	32	32	32	8	16	16	32	32	16	16
Fetch depth	2	2	2	2	2	2	2	2	3	3	2	2
Rename depth	2	2	2	2	2	2	2	2	2	2	2	2
Issue depth: total / wakeup-select loop	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	1 / 1	1 / 1	3 / 2	2 / 2	3 / 2	2 / 2	2 / 2
Register Read (and Writeback) depth	1	1	1	1	1	1	1	4	2	4	1	1
fetch-to-execute pipeline depth	10	10	10	10	10	9	9	14	12	15	10	10

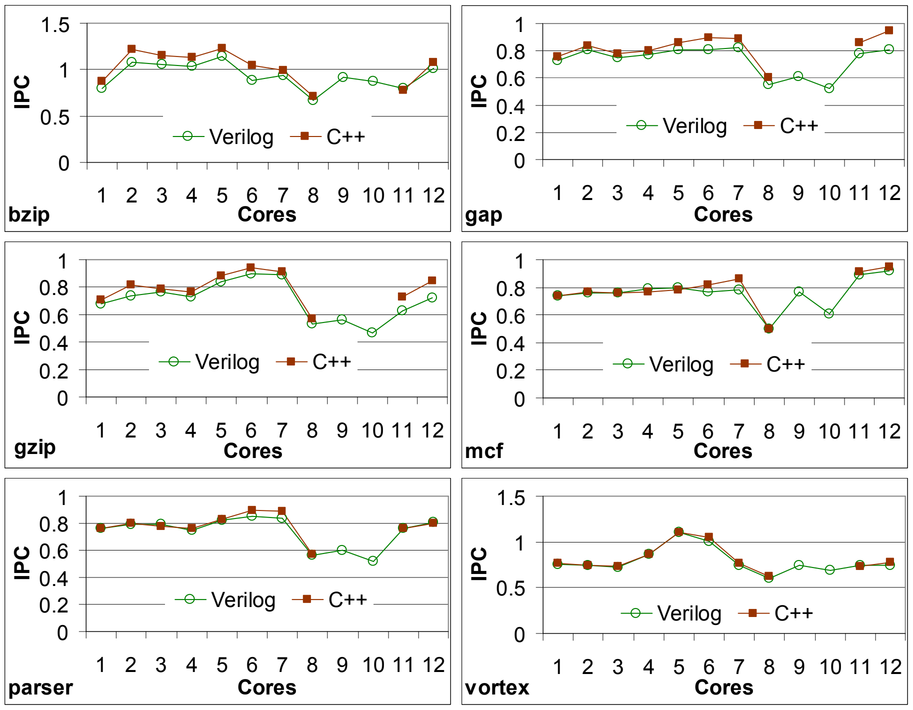

Figure 2. Results of executing 100 million instruction SimPoints of six benchmarks on the twelve cores.

5.2. Timing Validation

For timing validation, we compare cycle times and fetch-to-execute delays of FabScalar generated cores with three different commercial processors: 90nm POWER5 [24], 180nm Alpha 21364 [16], and 65nm MIPS32 74K [9][33]. All three implement RISC ISAs and they represent extremes from highly custom designed to fully synthesized (MIPS32 74K). Table 4 shows major microarchitecture parameters of the three processors that could be gleaned from the literature.

All delays are converted into the number of FO4 inverter delays for the underlying technology. We obtained the number of FO4 delays in a pipeline stage for each commercial processor, from published data [9][16][24][33].

The shaded section in Table 4 shows delay comparisons between the commercial cores and similarly configured FabScalar generated cores. The final number in Table 4 is raw fetch-to-execute delay divided by the fetch-to-execute pipeline depth of the commercial core. This cycle time is the best that could be achieved with careful latch based design, for the same pipeline depth.

Table 4. Delay comparisons of commercial processors with similarly configured FabScalar generated cores.

	Power5	**Alpha-21364**	**MIPS 74K**
Fetch Width	8	4	4
Dispatch Width	5	4	2
Issue Width	8	6	1
Fetch Queue	24	24	12
Issue Queue(s)	Int+Ld/St: 36, FP: 24, Br.: 12, CR: 10	Int:20, FP:15	Int:8, Agen:8
L1 I-cache / L1 D-cache (KB)	64 / 32	64 / 64	32 / 32
fetch-to-execute pipeline depth	12	6	12
Cycle Time of commercial core	23 FO4	25 FO4	33 FO4
Cycle Time of FabScalar core	29 FO4	37 FO4	32 FO4
Cycle Time of deeper FabScalar core	25 FO4 (depth=15)	26 FO4 (depth=11)	N/A
raw fetch-to-execute delay of FabScalar core	291 FO4	188 FO4	384 FO4
Cycle Time of FabScalar core with ideal latch-based design	24 FO4	32 FO4	N/A

The cycle time of the FabScalar-Power5 is relatively close to that of the Power5: 29 FO4 compared to 23 FO4, respectively. Slightly deeper pipelining (15 deep instead of 12 deep) yields an even closer 25 FO4 cycle time. The same cycle time of 24 FO4 can also be gotten with ideal latch-based design. All of these comparisons, and especially the latter (raw fetch-to-execute delay), confirm that the FabScalar generated RTL and the FabMem generated RAMs/CAMs are of reasonable quality from the standpoint of propagation delay.

A larger difference is observed for the FabScalar-21634 and 21634: 37 FO4 compared to 25 FO4. It suggests a significant degree of total delay optimization (Alpha processors gained a reputation as "speed demons"). Indeed, the deeper FabScalar-21364 needs nearly twice the pipeline depth to reach the 21364 cycle time, despite being similarly configured.

The MIPS 74K is a fully-synthesized design. This means that structures normally implemented with custom RAMs and CAMs are synthesized to flip-flops (except for caches). Accordingly, for a fair comparison, the delays for FabScalar-74K are also based on synthesis alone: FabMem is not used. The cycle times of these two fully-synthesized cores are nearly identical: 32 FO4 for the FabScalar-74K versus 33 FO4 for the 74K. That both cores are fully-synthesized, use virtually the same ISA, and have the same cycle time, further supports the assertion that the RTL is of reasonable quality from the standpoint of propagation delay.

5.3. Suitability for Standard ASIC Flows

To demostrate that FabScalar-generated RTL can be taken through logic-synthesis and place-and-route, we take two parallel approaches. In first approach, we synthesized and place-and-routed a 4-way superscalar processor using standard ASIC flow. The physical design is shown in Figure 3 (left).

In second approach, we synthesized similar configuration 4-way superscalar on a Xilinx Virtex-5 LX155T FPGA. The FPGA-synthesized design runs at 80MHz. The physical design is shown in Figure 6 (right).

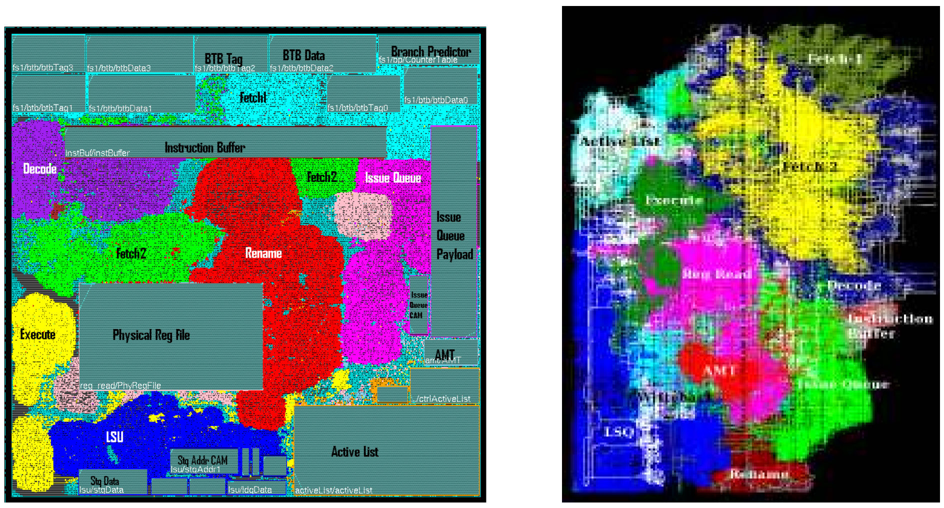

Figure 3. Physical design of a 4-way superscalar processor. Using standard ASIC flow (left) and using FPGA flow (right).

6. Summary and Future Work

We presented FabScalar, a novel toolset for automatically composing the synthesizable register-transfer-level (RTL) designs of arbitrary cores within a canonical superscalar template. Each canonical pipeline stage has many variants that differ in their complexity (superscalar width and stage-specific structure sizes) and depth of sub-pipelining, and canonical pipeline stages are composable into an overall core. Thus, FabScalar helps mitigate practical issues that currently impede proliferating microarchiecturally diverse cores in both general-purpose multi-core architectures and application-specific embedded systems.

We performed detailed validation experiments along three fronts to evaluate the quality of RTL designs generated by FabScalar and these experiments confirmed that FabScalar-generated RTL designs are of good quality.We have released the FabScalar toolset for other researchers and developers to use and possibly expand.

As part of future work, the CPSL should be augmented with floating-point pipelines, MMUs, and general replay mechanisms for aggressive speculation. Because of its open-source synthesizable RTL and physical designs of arbitrary superscalar processors, FabScalar can significantly enhance research. As computer architecture research becomes increasingly driven by technology related problems (Moore's law scaling, power, temperature, reliability, variability), RTL and physical designs generated by FabScalar are potentially of value to researchers. FPGA-based acceleration of superscalar processor simulation is another promising application of FabScalar. Further, FabScalar may also be used for comprehensive timing, power and area modeling of superscalar processors in different technologies.

6. References

[1] D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. *25th Int'l Symp. on Computer Architecture*, June 1998.

[2] M. Anderson. A More Cerebral Cortex. *IEEE Spectrum*, pp. 58-63, Jan. 2010.

[3] E. Borch, E. Tune, S. Manne, J. Emer. Loose Loops Sink Chips. *8th Int'l Symp. on High-Perf. Comp. Arch.*, Feb. 2002.

[4] M. D. Brown, J. Stark, Y. N. Patt. Select-Free Instruction Scheduling Logic. *34th Int'l Symp. on Microarch.*, Dec. 2001.

[5] D. Burger, T. M. Austin, S. Bennett. Evaluating Future Microprocessors: The SimpleScalar ToolSet. University of Wisconsin-Madison Technical Report CS-TR-1308, 1996.

[6] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, July 2008.

[7] T. Karkhanis and J. E. Smith. Automated Design of Application-Specific Superscalar Processors. ISCA, 2007.

[8] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. C. Cronquist, M. Sivaraman. PICO: Automatically Designing Custom Computers. *IEEE Computer*, 35(9):39-47, Sep. 2002.

[9] K. R. Kishore, V. Rajagopalan, G. Beloev, R. Thekkath. Architectural Strengths of the MIPS32 74K Core Family. White Paper, May 2000.

[10] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. *International Symposium on Microarchitecture*, Dec. 2003.

[11] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *31st International Symposium on Computer Architecture*, June 2004.

[12] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. *15th International Symposium on Parallel Architecture and Compilation Techniques*, Sep. 2006.

[13] B. C. Lee and D. M. Brooks. Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity. *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[14] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *42nd Int'l Symp. on Microarchitecture*, Dec. 2009.

[15] S. McFarling. Combining Branch Predictors. DEC WRL TN-36, 1993.

[16] E. J. McLellan, D. A. Webb. The Alpha 21264 Microprocessor Architecture. *International Conference on Computer Design*, p. 90, Oct. 1998.

[17] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *Computer Architecture Letters (CAL)*, 5(1):14-17, 2006.

[18] H. H. Najaf-abadi, E. Rotenberg. Configurational Workload Characterization. *ISPASS*, 2008.

[19] H. H. Najaf-abadi and E. Rotenberg. Architectural Contesting. *15th Int'l Symp. on High-Perf. Comp. Arch.*, Feb. 2009.

[20] H. H. Najaf-abadi, N. K. Choudhary, and E. Rotenberg. Core-Selectability in Chip Multiprocessors. *18th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[21] S. Palacharla, N. P. Jouppi, J. E. Smith. Complexity-effective Superscalar Processors. *International Symposium on Computer Architecture*, June 1997.

[22] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block Ahead Branch Predictors. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[24] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, J. B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505-521, July 2005.

[25] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, R. Jenkal. FreePDK: An Open-Source Variation-Aware Design Kit. *Int'l Conf. on Microelectronic Systems Education*, 2007.

[26] L. Strozek, D. Brooks. Efficient Architectures through Application Clustering and Achitectural Heterogeneity. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

[27] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. *14th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, March 2009.

[28] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Tech. Report HPL-2008-20, HP Labs, 2008.

[29] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, pp. 25-33, Jan. 1967.

[30] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. *International Conference on Dependable Systems and Networks (DSN)*, 2004.

[31] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. IEEE Micro, 16(2):28-41, April 1996.

[32] http://www.tensilica.com/products/xtensa-customizable.htm

[33] http://www.mips.com/media/files/74k/MIPS_74K_509.pdf