FabScalar: Automating Superscalar Core Design

Providing multiple superscalar core types on a chip, each tailored to different classes of instruction-level behavior, is an exciting direction for increasing processor performance and energy efficiency. Unfortunately, processor design and verification effort increases with each additional core type, limiting the microarchitectural diversity that can be practically implemented. FabScalar aims to automate superscalar core design, opening up processor design to microarchitectural diversity and its many opportunities.

Niket K. Choudhary
Salil V. Wadhavkar
Tanmay A. Shah
Hiran Mayukh
Jayneel Gandhi
Brandon H. Dwiel
Sandeep Navada
Hashem H. Najaf-abadi
Eric Rotenberg
North Carolina State University

The past decade has witnessed a major transition from single-core to multicore processors. Multicore processors present an exciting opportunity to exploit diversity within and across applications by employing microarchitecturally diverse superscalar core types, in what is called a single-instruction-set architecture (single-ISA) heterogeneous multicore processor.[1] Program phases differ in their instruction-level characteristics: the amount and distribution of instruction-level parallelism (ILP) and memory-level parallelism (MLP), branch predictability, and cache locality. We can improve performance and power metrics by matching instruction-level behavior to differently designed cores. The core types may differ in their superscalar fetch, dispatch, and issue widths, pipeline depths, instruction scheduling (in-order or out-of-order), sizes of units involved in exposing ILP and MLP (issue queue, load and store queues, physical register file, and reorder buffer), function unit mix, and sizes of predictors and caches.

Previous work in this area projects significant performance and power advantages for microarchitecturally diverse superscalar cores. Yet, no prior research has addressed a crucial drawback of this paradigm: design and verification effort is multiplied by the number of different core types. This factor limits the amount of microarchitectural diversity that can be practically implemented.

In this article, we propose framing superscalar cores in a canonical form, so that it becomes feasible to quickly design many cores that differ in the three major superscalar dimensions: superscalar width, pipeline depth, and sizes of structures for extracting ILP (the frequency depends on these three dimensions). We implemented our approach in 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 subpipelining, and canonical pipeline stages are composable into an overall core. Thus, FabScalar helps mitigate practical issues that currently

Published by the IEEE Computer Society

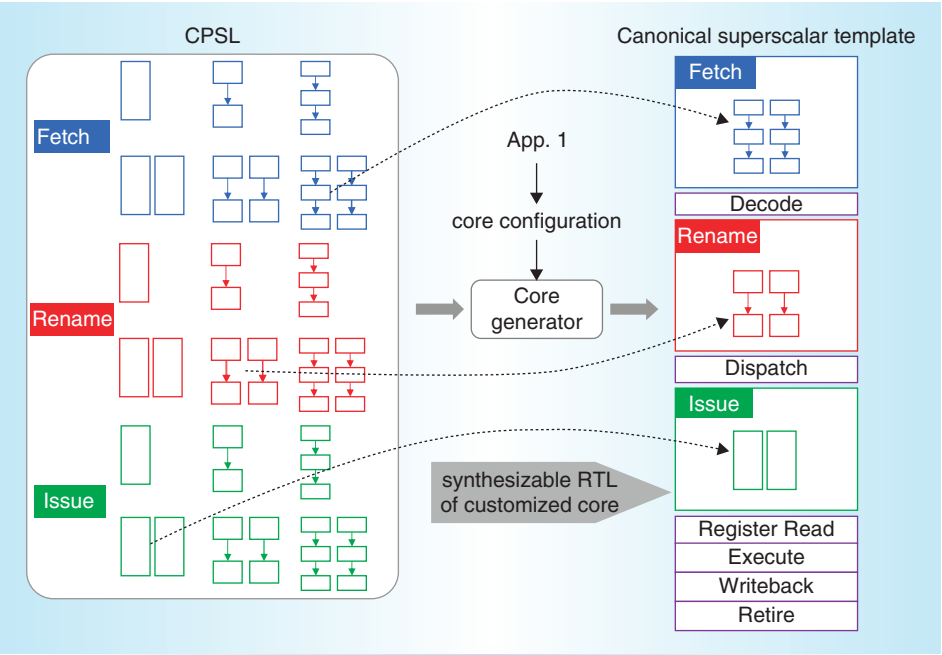

Figure 1. Generating a customized core with FabScalar. The toolset contains a canonical superscalar template, a canonical pipeline-stage library, and a core generator.

impede proliferating microarchitecturally diverse cores.

FabScalar

As Figure 1 shows, FabScalar consists of

- a canonical superscalar template, which defines canonical pipeline stages;
- a canonical pipeline-stage library (CPSL), which contains many synthesizable RTL designs of each canonical pipeline stage; and
- a core generator for automatically composing the RTL designs of arbitrary superscalar cores by referencing the CPSL.

The different designs of a given canonical pipeline stage vary along three major dimensions:

- *Superscalar complexity.* A canonical pipeline stage's superscalar complexity is a product of its superscalar width (number of pipeline ways) and the sizes of its associated ILP-extracting structures (such as the issue queue, physical register file, and predictors). Increasing superscalar complexity could help extract more ILP in the program, but it 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.
- *Subpipelining.* A canonical pipeline stage is nominally one cycle in duration, but could be subpipelined deeper to achieve a higher clock frequency.
- *Stage-specific design choices.* Multiple alternatives often exist 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 behaviors in the program.

Verified synthesizable RTL is the essential starting point for physical design, whether using automated synthesis, place-and-route, and memory compilers or full-custom design. Industry is trending in the direction of synthesis,[2,3] particularly for time-to-market sensitive and soft-IP driven SoCs. Moreover,

future technologies are less predictable, favoring robust standard cells. The combination of RTL design generators (such as FabScalar) and physical design automation significantly boosts designer productivity.

To be clear, because we're attacking the design-effort problem, we ultimately intend for FabScalar to be used for the design, verification, and fabrication of chips comprised of microarchitecturally diverse superscalar cores. Although it's still in the academic stages at this point, this is the trajectory we're aiming for. That said, FabScalar is also useful for general computer architecture research. In this article, we discuss this work's implications for both processor development and architecture research.

Template and CPSL

Figure 1 shows the canonical superscalar template defined by FabScalar. It consists of nine canonical pipeline stages: Fetch, Decode, Rename, Dispatch, Issue, Register Read, Execute, Writeback, and Retire.

The CPSL ISA is the SimpleScalar ISA,[4] a close derivative of the MIPS ISA (minus load and branch delay slots).

The CPSL contains many synthesizable RTL designs for each canonical pipeline stage that differ in their superscalar width and depth of subpipelining. Table 1 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. All front-end stages (Fetch through Dispatch) and the Retire stage vary from 1-way to 8-way superscalar. The minimum width of all back-end stages (Issue through Writeback) is currently 4 because at least four different function units are required: one each of simple arithmetic logic unit (ALU), complex ALU, load/store port, and branch unit. One can accommodate narrower-issue widths by aggregating multiple function-unit types into one execution lane, which we left for future work. The maximum width of all back-end stages is 8-way superscalar.

Table 1's second column also shows ranges of subpipelining depth. Subpipelining was guided by natural logic boundaries within each canonical pipeline stage design and timing results from synthesis. Our paper for the 38th IEEE/ACM International Symposium on Computer Architecture describes each stage's partitioning in detail.[5]

The table's third column lists the structures of each canonical pipeline stage. Sizes of all stage-specific structures are parameterized in the RTL description. Consequently, their sizes can take on arbitrary values, and no ranges are specified for them in the table's third column.

Table 1's final column considers another dimension for microarchitectural diversity: design choices specific to each canonical pipeline stage. It's outside this article's scope to cover all of these techniques in the CPSL, at the level of synthesizable RTL code. Nonetheless, we felt it would be of interest to enumerate notable examples in Table 1 to emphasize the potential for growing the CPSL in the future, and to underscore the specificity with which one might target microarchitectural diversity to specific instruction-level behavior. For example, certain program phases will favor one branch-misprediction recovery strategy over another depending on the frequency of mispredicted branches, their distribution, and the criticality of their backward slices. As another example, techniques that do not benefit subsets of the workload space can be excluded from a core to streamline its frequency and static and dynamic power.

Methodology

Here, we summarize our methodology for evaluating the quality of FabScalar-generated cores. Table 2 shows the electronic design automation (EDA) tools used for functional verification, synthesis, and place-and-route. For synthesis, we used the FreePDK 45-nm standard cell library.[6]

Because specialized, highly-ported RAMs and CAMs are so pervasive and essential to a superscalar processor, we have developed a tool (FabMem[5]) for generating their physical layouts and extracting timing, power, and area. For simulation, RAMs and CAMs are implemented with Verilog modules. For synthesis and place-and-route, we use electrical and physical views (in Liberty and Library Exchange Formats, respectively) generated by FabMem.

Table 1. Overview of stage designs available in the canonical pipeline-stage library (CPSL).

Canonical pipeline stage	Dimensions (W = width, D = depth)	Stage-specific structures (sizes parameterized in the RTL description)	Microarchitectural approaches*
Fetch	W = 1 to 8, D = 2 to 5 Fetch-1: 1 or 2 substages Fetch-2: 1 to 3 substages	Branch or pattern history table (BHT or PHT) Branch target buffer (BTB) Return address stack (RAS) Level-1 (L1) instruction cache	**Branch prediction algorithm** No interleaving vs. **two-way interleaving** **Block-based BTB** vs. **interleaved BTB** Multicycle fetch: Unpipelined **pipelined using block-ahead prediction**
Decode	W = 1 to 8, D = 1 to 3	Fetch queue	Micro-operation cracking Nonspeculative vs. speculative decode (if variable-length instruction-set architecture [ISA])
Rename and Retire	W = 1 to 8, D = 1 to 3 W = 1 to 8, D = 2	Rename map table (RMT) Architectural map table (AMT) No. of shadow map tables: 0 or 4 Free list Active list Physical register 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 previous mapping from RMT, active list pushes freelist **read previous 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 Subpipelining variants:[5] 1/1, 2/1, 2/2, 3/3, 3/2	IQ	In-order vs. **out-of-order** Collapsing IQ vs. **freelist-based IQ** Multiple schedulers vs. **single scheduler** Pipelined wakeup+select: **one-cycle producers nonspeculatively wake up dependents** one-cycle producers speculatively wake up dependents Load hit/miss: predict hit always **predict miss always** hit predictor Load conflict with unknown store address: **predict no conflict always** predict conflict always memory dependence predictor Recovery for speculative wakeup & load conflict speculation: 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 No. of simple arithmetic logic units (ALUs): 1 to 5, D = 1 No. of complex ALUs: 1, D = 3 No. of load/store ports: 1, D = 2 No. of branch units: 1, D = 1	Load queue (LQ) Store queue (SQ) L1 data cache	**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

* Bold text indicates specific design choices represented in the current CPSL.

Table 2. Application-specific integrated circuit (ASIC) flow.

Phase	EDA tool used
Functional verification	Cadence NC-Verilog, version 06.20-s006
Logic synthesis	Synopsys Design Compiler, version X-2005.09-SP3
Place-and-route	Cadence SoC Encounter, version 7.1

We use FabMem-generated RAMs and CAMs 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 handled similarly, except that electrical and physical views are derived from CACTI 5.1 data; we adjusted CACTI's device and wire parameters to match those of FreePDK.

Each canonical pipeline stage has many underlying implementations in the CPSL that differ in their superscalar width and depth of subpipelining. For each CPSL design, 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.

Evaluating RTL quality

Here, we evaluate the quality of RTL designs produced by FabScalar. We perform validation along three fronts: functional and instructions-per-cycle (IPC) validation, timing validation, and suitability for physical design.

For functional and IPC validation, we generate a dozen different cores covering a range of widths, sizes, and depths. They all successfully run 100 million instruction SimPoints of SPEC integer benchmarks. The IPC results are within expected ranges for SPEC, IPC differences among cores correspond well with their microarchitectural differences, and IPCs closely track the IPCs produced by FabScalar's cycle-accurate C++ simulator.

For timing validation, we also evaluate the quality of the RTL and FabMem in terms of cycle time. We compare three commercial reduced-instruction-set-computing (RISC) superscalar processors with similarly configured FabScalar-generated cores. Validating cycle time is challenging and imperfect for several reasons, including the following:

- *Different technology nodes, technology libraries, and foundry processes.* We deal with this issue by converting cycle time into the number of fanout-of-4 (FO4) inverter delays of the technology, yielding a technology-independent comparison.
- *Different degrees of custom design, including the extent of circuit optimization, dynamic logic, and latch-based design for accommodating logic partition imbalances.* We deal with this issue only partially by employing multiported RAMs and CAMs generated by FabMem. We also draw comparisons with a commercial fully synthesized embedded core at one end of the spectrum. Regarding latch-based design, in addition to comparing cycle time, we also examine raw total logic delay through the pipeline from Fetch to Execute.
- *Different ISAs and unique microarchitecture features.* For example, the current CPSL doesn't have Issue-stage designs with multiple schedulers (see Table 1, last column) or replicated register files. Multiple smaller schedulers reduce the select logic delay by reducing the number of instructions contending for a given execution lane, at the cost of some load imbalance among the multiple issue queues. More importantly, when there are multiple function units of the same type, providing each function unit with a dedicated issue queue avoids cascading select trees, a big delay savings. Replicated register files reduce the number of read ports in each register file copy, improving their access times. Although the CPSL doesn't yet represent these techniques, we can model their effect for timing validation purposes by applying a smaller and simpler issue queue and a register file with fewer read ports.

Finally, for physical-design suitability, we demonstrate the generated RTL's suitability for full synthesis and place-and-route by a standard application-specific integrated circuit (ASIC) flow.

Table 3. Cores used for functional and instructions-per-cycle (IPC) validation experiments.

Parameter or structure	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/Register Read/ Execute/Writeback/ 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 (reorder buffer [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
IQ	32	32	32	64	128	16	16	32	64	64	32	32
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	
BHT (no. of entries)	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
BTB (no. of entries)	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K
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

Functional and IPC validation

We used the FabScalar tool to generate the RTL designs for the 12 cores described in Table 3.

We selected Cores 1 through 6 primarily to explore stage widths and structure sizes. Except for Core 6, depths are the same across these cores. Core 6 is a particularly narrow core (two-way superscalar in the front end). Core 2 has different widths in the front end (4) and the back end (6). Core 5 is a particularly wide core (eight-way superscalar fetch and execute with large resources).

Cores 6 through 10 aim to explore depths of stages and the fetch-to-execute pipeline depth. Cores 7 and 8 resemble Core 1, except that Core 7 is shallower (fetch-to-execute = 9) and Core 8 is deeper (fetch-to-execute = 14). They differ in their Issue and Register Read depths. Cores 9 and 10 are unique in that their Fetch-1 substage of Fetch is pipelined into two cycles, using block-ahead branch prediction. This yields a total Fetch depth of three cycles. Cores 9 and 10 differ in their Issue and Register Read depths. Core 10 is the deepest of the 12 cores (fetch-to-execute = 15), although not the deepest possible with the CPSL because Rename and Fetch (the Fetch-2 logic) can be deepened further.

Cores 11 and 12 are the same as Cores 1 and 2, respectively, except they use the gshare

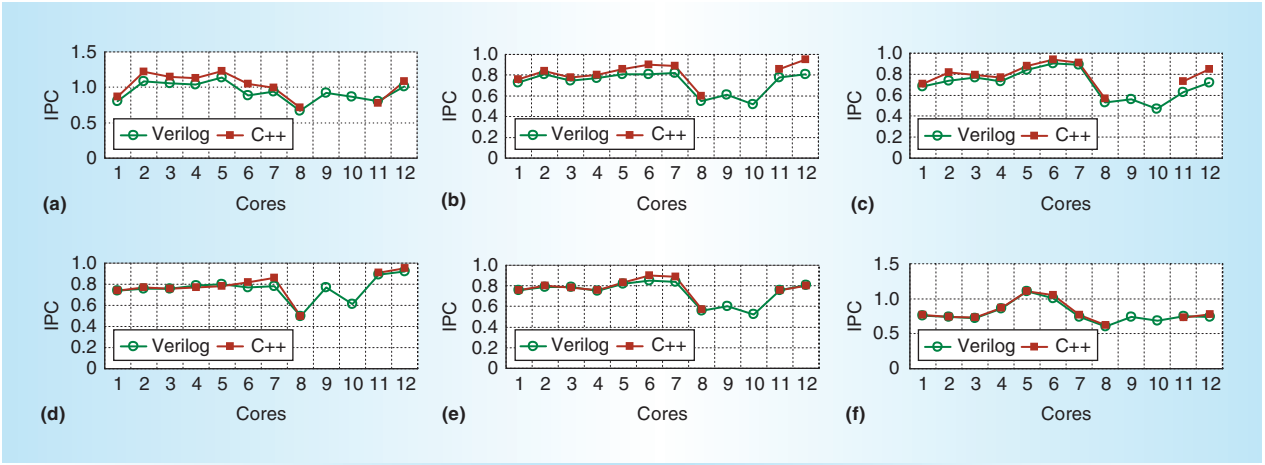

Figure 2. Results of executing 100-million instruction SimPoints of six benchmarks—bzip (a), gap (b), gzip (c), mcf (d), parser (e), and vortex (f)—on the 12 cores. Results are shown for both RTL (''Verilog'') and the cycle-accurate C++ simulator (''C++'').

instead of the bimodal branch predictor. Because the gshare predictor can only conveniently supply one branch prediction per cycle, Fetch stage designs in the CPSL employing gshare present a tradeoff between slightly reducing fetch bandwidth and increasing fetch accuracy.

Figure 2 shows the results of executing the 100-million instruction SimPoints of six benchmarks. 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 data points are missing for Cores 9 and 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 misspeculations (load issues before a conflicting store) 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 anomalies are observed, they're more likely inherent in the design rather than in the design's RTL model.

Differences in IPCs among cores tend to correspond with their microarchitectural differences. For example, among Cores 1 through 5, we expect Core 5 to have the highest IPC because it's the most aggressive core, the depths are the same, and no negative cycle time consequences are applied in an IPC-only comparison. Cores 8 and 10 are the deepest pipelines, and they have lower IPCs than other configurations as a result. Some pairwise comparisons of cores could go either way owing to increasing some parameters and decreasing others, leading to potentially nonmonotonic cores. For example, Core 6 has the same or higher IPC than Core 2 in all benchmarks except bzip. Core 6 is narrow (two-way fetch) but its advantage over other configurations is its one-cycle wakeup-select loop. In the case of bzip, however, there is apparently sufficient ILP to outweigh the longer wakeup-select loop.

Anomalies—for example, a more aggressive core having lower IPC than a simpler core of the same pipeline depth—are sometimes caused by more frequent load misspeculations or branch mispredictions that stem from larger window sizes. Extra recoveries are performance debilitating when recovery is a full squash from the head of the active list.

Timing validation

For timing validation, we compare cycle times and fetch-to-execute delays of FabScalar-generated cores with three commercial processors: 90-nm Power5,[7] 180-nm Alpha 21364,[8] and 65-nm MIPS32 74K.[3] All three implement RISC ISAs and represent extremes from highly custom-designed to

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

Parameter or structure	Power5	Alpha 21364	MIPS32 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
Physical register file(s)	Int: 120, FP: 120	Int: 80, FP: 72	64
Load queue/Store queue	32/32	32/32	8/8
L1 instruction cache/L1 data cache (Kbytes)	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 time unit is the number of fanout-of-4 (FO4) inverter delays of the technology.

fully synthesized (MIPS32 74K). Table 4 shows the three processors' major microarchitecture parameters.

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.[3,7,8]

Table 4's shaded section shows delay comparisons between the commercial cores and similarly configured FabScalar-generated cores. Five numbers are shown:

1. The commercial core's cycle time.
2. Cycle time of the similarly configured FabScalar core of the same pipeline depth.
3. Cycle time of a deeper version of the FabScalar core, with its fetch-to-execute pipeline depth shown in parentheses. This shows how much additional subpipelining is needed to compensate for the FabScalar core's lesser degree of custom design.
4. The FabScalar core's raw fetch-to-execute delay. This is the sum of propagation delays of all the stages between Fetch and Execute.
5. The final number is #4 above divided by the commercial core's fetch-to-execute pipeline depth. This corresponds to the FabScalar core's hypothetical cycle time if pipeline registers evenly divided up the raw fetch-to-execute delay (no imbalance among pipeline stages). This cycle time is the best that could be achieved with careful latch-based design, for the same pipeline depth.

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

We observe a larger difference for the FabScalar-21364 and 21364: FO4 of 37 compared to 25. What's interesting is that the 21364 has a cycle time close to the Power5 despite the 21364 being half as deep. This is partly due to lower superscalar complexity of the older 21364, but it also 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.

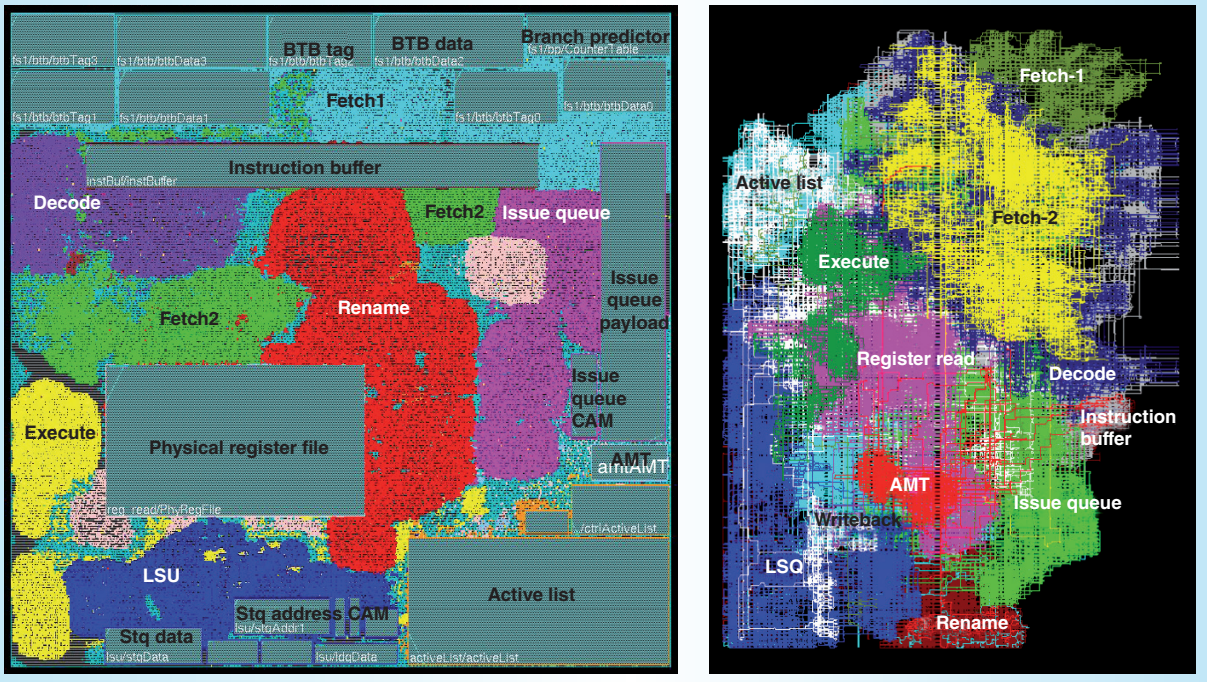

Figure 3. Physical design of a four-way superscalar processor (a); the same design synthesized to a Virtex-5 field-programmable gate array (b).

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 isn't used. The cycle times of these two fully synthesized cores are nearly identical: FO4 of 32 for the FabScalar-74K versus 33 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 propagation-delay standpoint.

Suitability for standard ASIC flows

To demonstrate that we can take FabScalar-generated RTL through standard ASIC flows, we synthesized and placed-and-routed a four-way superscalar processor. Figure 3a shows the physical design.

We synthesized the same core to a Virtex-5 field-programmable gate array (FPGA), shown in Figure 3b. The FPGA prototype matches Verilog simulation and is 2,000 to 5,000 times faster, achieving modeled-processor frequencies of 7 to 15 MHz.

The future of processor design

The FabScalar project represents the first attempt to automate superscalar processor design (see the "Related Work in Core Generators and Superscalar Models" sidebar for related efforts). In general, regardless of where it has been applied, automation is transformative because engineers can focus on creatively using a technology rather than the technology itself. It also puts the technology into more people's hands.

Superscalar processors have been successful for many years because they exploit parallelism transparently. Several factors lead us to believe that now is the time to automate their development.

First, we can. Academic and industry practitioners have advanced superscalar processors' performance and efficiency, refined

Related Work in Core Generators and Superscalar Models

The Illinois Verilog Model (IVM) provides the Verilog for a semiparameterizable four-issue superscalar processor.[1] The current IVM's drawbacks are its unsynthesizable or poorly synthesizable (low-frequency) Verilog modules. More importantly, IVM's superscalar width and pipeline depth are inflexible. These aspects aren't easily parameterized and require FabScalar's approach: an RTL generator that uses the canonical superscalar template and canonical pipeline stage library (CPSL) to compose a core of desired width and depth. Finally, FabScalar runs SPEC benchmarks out of the box and has been validated in terms of instructions-per-cycle (IPC), cycle time, and synthesizability via standard application-specific integrated circuit (ASIC) flows.

Strozek and Brooks developed a framework for high-level synthesis of simple cores for embedded systems.[2] The Program-In-Chip-Out (PICO) framework out of HP Labs is closely related in that it customizes very long instruction word (VLIW) cores and nonprogrammable accelerators for embedded applications.[3] Tensilica's Xtensa Configurable Processors (http://www.tensilica.com/products/xtensa-customizable.htm) automate the designer's task of customizing instructions, functional units, and even VLIW data paths. FabScalar is unique in that it generates complex superscalar processors, which is evident in the novel composable CPSL.

Palacharla, Jouppi, and Smith developed models for estimating propagation delays of key superscalar pipeline stages (rename, issue, and bypasses).[4] Li et al. describe McPAT, a comprehensive power, area, and timing modeling framework for multicore systems.[5] The timing models extend Palacharla's approach to multiple microarchitectural styles. FabScalar extends delay modeling to other critical pipeline stages such as instruction fetch, arbitrary core logic, and the whole core; it considers subpipelining and its imbalances; and it produces RTL implementations of cores. FabScalar's RTL output underscores a crucial distinction with computer architecture tools: FabScalar aims to streamline the design, verification, and fabrication of chips—that is, it's meant to serve as a development tool for designing heterogeneous multicore chips, not just an estimation tool for research.

References

1. N.J. Wang et al., ''Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline,'' *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks,* IEEE CS, 2004, pp. 61-70.
2. L. Strozek and D. Brooks, ''Efficient Architectures through Application Clustering and Architectural Heterogeneity,'' *Proc. ACM Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems,* ACM, 2006, pp. 190-200.
3. V. Kathail et al., ''PICO: Automatically Designing Custom Computers,'' *Computer,* vol. 35, no. 9, 2002, pp. 39-47.
4. S. Palacharla, N.P. Jouppi, and J.E. Smith, ''Complexity-Effective Superscalar Processors,'' *Proc. 24th IEEE/ACM Int'l Symp. Computer Architecture,* ACM, 1997, pp. 206-218.
5. S. Li et al., ''McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Many-Core Architectures,'' *Proc. 42nd IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2009, pp. 469-480.

our understanding of them, and classified them. We understand them deeply and can define a good template that unifies them yet also facilitates diverse variants.

Second, technology is driving specialization and diversity. In the past, technology scaling delivered faster and lower-energy transistors at an exponential pace. It was reasonable to overlook the fact that a single generic microarchitecture leaves some performance and power on the table for diverse program phases. With technology no longer delivering exponential efficiency gains, specializing the microarchitecture to instruction-level behavior is necessary to get the most performance and energy efficiency from silicon. To achieve an overall robust processor, specialization must be coupled with diversity: providing sufficiently many specialized cores to cover arbitrary program behaviors. The single-ISA heterogeneous multicore paradigm is one response to specialization and diversity. Superscalar design automation makes it more practical.

Finally, the smart-phone, tablet PC, and embedded appliance markets demand superscalar performance, microarchitectural diversity, and short time to market. The recent introduction of superscalar-based application processors in these devices is driven by increasing software complexity (for example, the ARM Cortex-A15, Qualcomm Scorpion and Krait, AMD Bobcat, and MIPS 74K). Microarchitectural diversity is driven by performance and energy optimization within devices (single-ISA heterogeneous multicore) and different form factors across devices. Short time to market is critical in high-growth markets. With this in mind, the Bobcat and 74K designers advocate automated synthesis and place-and-route.[2,3] Synthesized application processors are the logical conclusion of several factors:

short time to market, soft-IP-rich designs, lower-frequency operation of application processors in the handheld market (1 to 2 GHz), unabated transistor miniaturization, less-predictable technology (favoring robust standard cells), and decades of investment in CAD tools. Superscalar design automation coupled with physical design automation makes it possible to meet the demand for superscalar performance and microarchitectural diversity, with short time to market.

This work also has implications for computer architecture research. Computer architecture research is increasingly driven by technology-related problems (Moore's law scaling, power, temperature, reliability, and variability). Open source synthesizable Verilog models of arbitrary superscalar processors are invaluable because they enable exploring architecture–technology interactions in a complete context (whole pipelines), they enable sensitivity studies across different microarchitecture configurations, and they increase result fidelity and detail.

Additionally, FabScalar will enable the extensive use of whole-pipeline RTL models for processor research. Although RTL modeling might seem constraining, the constraints are of a nature that will likely uncover surprising opportunities for microarchitecture innovation. It's enticing simply to revisit alternative processor architectures and techniques from past decades in the context of RTL modeling. Coupling these with heterogeneity offers even greater possibilities. Past proposals discarded due to narrow applicability are by contrast valued in the context of specialization and diversity. This isn't to mention the benefits of routinely quantifying the cycle time, area, and power costs and savings of new microarchitecture techniques. We'll need a corresponding leap in RTL simulation capability—namely, automatic mapping of arbitrary superscalar configurations to single FPGAs. The past several years have witnessed significant progress in this area.[9] In this article, we prototyped a four-way superscalar core on a single Virtex-5 FPGA, and since then we've developed a tool that automates mapping arbitrary configurations to the FPGA. The FPGA prototypes are several orders of magnitude faster than Verilog and C++ simulation.

FabScalar streamlines the design of superscalar cores through automation, opening up processor design to microarchitectural diversity and the many exciting opportunities that it presents. As a research tool, it puts the experience of hardware prototyping within reach of more computer architecture researchers and acts as a catalyst for microarchitecture innovation by expressing implementation constraints. We are optimistic that, with further refinements and enhancements, FabScalar will evolve into an EDA tool with industrial applications. MICRO

Acknowledgments

This research was supported by NSF grant CCF-0811707 and gifts from Intel and IBM.

References

1. R. Kumar et al., ''Single-ISA Heterogeneous Multicore Architectures: The Potential for Processor Power Reduction,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2003, pp. 81-92.
2. B. Burgess et al., ''Bobcat: AMD's Low-Power x86 Processor,'' *IEEE Micro,* vol. 31, no. 2, 2011, pp. 16-25.
3. K.R. Kishore et al., ''Architectural Strengths of the MIPS32 74K Core Family,'' white paper, MIPS Technologies, May 2000.
4. D. Burger, T.M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The SimpleScalar ToolSet,* tech. report CS-TR-1308, Computer Science Dept., Univ. of Wisconsin–Madison, 1996.
5. N.K. Choudhary et al., ''FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template,'' *Proc. 38th IEEE/ACM Int'l Symp. Computer Architecture,* ACM, 2011, pp. 11-22.
6. J.E. Stine et al., ''FreePDK: An Open-Source Variation-Aware Design Kit,'' *Proc. IEEE Int'l Conf. Microelectronic Systems Education* (MSE 07), IEEE CS, 2007, pp. 173-174.
7. B. Sinharoy et al., ''POWER5 System Microarchitecture,'' *IBM J. Research and Development,* vol. 49, nos. 4/5, 2005, pp. 505-521.
8. R.E. Kessler, E.J. McLellan, and D.A. Webb, ''The Alpha 21264 Microprocessor

Architecture,'' *Proc. Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 98), IEEE CS, 1998, pp. 90-95.
9. P.H. Wang et al., ''Intel Atom Processor Core Made FPGA-Synthesizable,'' *Proc. ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays,* ACM, 2009, pp. 209-218.

Niket K. Choudhary is a PhD student in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests include computer architecture, superscalar processor design automation, and heterogeneous multicore architecture design. Choudhary has an MS in computer engineering from North Carolina State University. He is a student member of the ACM.

Salil V. Wadhavkar is a PhD student in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests are heterogeneous multicore architectures, energy-efficient processor design, and workload analysis. Wadhavkar has an MSE in electrical engineering from Temple University. He is a student member of IEEE.

Tanmay A. Shah is a video ASIC design engineer at Qualcomm. His research interests include VLSI design and microarchitecture. Shah has an MS in computer engineering from North Carolina State University.

We honor the bright life and memory of **Hiran Mayukh**, who left us too soon. He received his BTech in electronics and communication engineering from National Institute of Technology, Karnataka, India, in 2008. He received his MS in computer engineering from North Carolina State University in 2010 and subsequently joined the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison. His friendship and scholarly achievements will always be cherished.

Jayneel Gandhi is a PhD student in the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison. His research interests include computer architecture and VLSI design. Gandhi has an MS in computer engineering from North Carolina State University.

Brandon H. Dwiel is a PhD student in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests include adaptive processors and 3D-enabled computer architectures. Dwiel has an MS in computer engineering from North Carolina State University. He is a student member of the ACM.

Sandeep Navada is a PhD student in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests include processor microarchitectures, bottleneck analysis, application steering in heterogeneous multicore processors, machine learning, and embedded systems. Navada has a BTech in electrical engineering from the Indian Institute of Technology.

Hashem H. Najaf-abadi is a graphics hardware engineer at Intel, where he works on streaming architectures. His research interests include asymmetry in multicore and streaming architectures. Najaf-abadi has a PhD in computer engineering from North Carolina State University. He is a member of IEEE and the ACM.

Eric Rotenberg is a professor in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests include superscalar processor design automation, heterogeneous multicore processors, and novel microarchitectures. Rotenberg has a PhD in computer sciences from the University of Wisconsin–Madison. He is a senior member of IEEE.

Direct questions and comments about this article to Eric Rotenberg, North Carolina State University, Dept. of Electrical and Computer Engineering, Box 7911, Raleigh, NC, 27695-7911; ericro@ncsu.edu.