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

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

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC, USA
(*See endnote for current affiliations.)

nkchoudh@ncsu.edu, http://people.engr.ncsu.edu/ericro/research/fabscalar.htm

ABSTRACT

A growing body of work has compiled a strong case for the single-ISA heterogeneous multi-core paradigm. A single-ISA heterogeneous multi-core provides multiple, differently-designed superscalar core types that can streamline the execution of diverse programs and program phases. No prior research has addressed the "Achilles' heel" of this paradigm: design and verification effort is multiplied by the number of different core types.

This work frames superscalar processors 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 instruction-level parallelism (ILP). From this idea, we develop a toolset, called FabScalar, for automatically composing the synthesizable register-transfer-level (RTL) designs of arbitrary cores within a canonical superscalar template. The template defines canonical pipeline stages and interfaces among them. A Canonical Pipeline Stage Library (CPSL) provides many implementations of each canonical pipeline stage, that differ in their superscalar width and depth of sub-pipelining. An RTL generation tool uses the template and CPSL to automatically generate an overall core of desired configuration. 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. With FabScalar, a chip with many different superscalar core types is conceivable.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids – *automatic synthesis*. C.1.3 [**Processor Architectures**]: Other Architecture Styles – *pipeline processors*, *heterogeneous systems*.

General Terms

Performance, Design, Verification.

Keywords

superscalar processors, instruction-level parallelism (ILP), heterogeneous (asymmetric) multi-core, custom processors

1. INTRODUCTION

A growing body of work has compiled a strong case for the single-ISA heterogeneous multi-core paradigm. A single-ISA heterogeneous multi-core provides multiple, differently-designed superscalar core types for streamlining the execution of sequential [8][11][16][17], parallel [5][15][18][25], and multiprogrammed [9][10] workloads, by exploiting diversity across and within applications. The core types may differ in their superscalar fetch/issue widths, pipeline depths, instruction scheduling (in-order or out-of-order), sizes of units involved in exposing instruction-level parallelism (ILP) (issue queue, load and store queues, physical register file, reorder buffer, etc.), function unit mix, and sizes of predictors and caches.

Prior works in this area project significant performance and power advantages for microarchitecturally diverse superscalar cores. No prior research has addressed the "Achilles' heel" 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 paper, we propose framing superscalars 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 (frequency depends on these three). The canonical form is 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 vary 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 a tool for automatically composing the RTL designs of arbitrary superscalar cores by referencing the CPSL.

In addition, the FabScalar toolset has several other notable features. Since highly-ported RAMs and CAMs are prevalent in superscalar processors and significantly impact area, power, and cycle time, we developed FabMem, a tool for automatically generating the physical designs (layouts) of multiported RAMs and CAMs. While memory compilers are not new, to the best of our knowledge, there is no commercially available memory compiler that can generate more than a few ports (otherwise we would gladly use it); a 4-issue superscalar requires a 12-ported register file, for example. FabScalar also provides a co-simulation environment in which a functional simulator written in C++ runs concurrently with verilog simulation of the superscalar processor. The two are independent and the functional simulator assists with checking and debugging the verilog simulation by comparing instructions' results as they retire from the processor. The co-simulation environment also simplifies running standard benchmarks by not requiring full-system simulation. Finally, FabScalar provides a cycle-accurate C++ simulator which can accurately model all of the cores that can be composed from the CPSL. Cycle-accuracy is achieved by using the same canonical superscalar processor definition and the same interfaces among canonical pipeline stages. The cycle-accurate simulator is useful for faster exploration and performance debugging of RTL designs.

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. For functional and IPC validation, a dozen different cores are generated and they all successfully run 100 million instruction SimPoints of SPEC integer benchmarks. The IPCs 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 compare the cycle times of three commercial RISC superscalar processors with cycle times of

FabScalar generated cores having similar configurations. (Delays are converted to technology-agnostic FO4 delays.) The three commercial processors represent a spectrum of highly custom designed to fully-synthesized cores and the results confirm that the FabScalar RTL is of good quality. To demonstrate suitability of FabScalar generated RTL for standard ASIC flows, we synthesize and place-and-route a sample core.

FabScalar does not address physical design effort beyond automated synthesis and place-and-route. Nonetheless, verified synthesizable RTL is the essential starting point for the physical design process. Physical design proceeds from full synthesis to incrementally tuned custom designs. By way of example, FabMem replaces structures that are otherwise synthesized to flip-flops with physical designs of RAMs and CAMs. Some superscalar cores in the embedded space are fully synthesized, such as the MIPS 74K [7]. There is even a recent example of out-sourcing processor tuning: Intrinsity performed targeted custom circuit design of the ARM Cortex-A8 to boost its frequency [1].

To be clear, since we are attacking the design-effort problem, ultimately, we intend FabScalar to be used for the design, verification, and fabrication of chips comprised of microarchitecturally diverse superscalars. Although still in the academic stages at this point, this is the trajectory we are aiming for.

That said, FabScalar is also useful for general computer architecture research. As computer architecture research becomes increasingly driven by technology related problems (Moore's law scaling, power, temperature, reliability, variability), open-source synthesizable verilog and physical designs of arbitrary superscalar processors are potentially of value. Another promising application of FabScalar is FPGA-based acceleration of superscalar processor simulation. Along these lines, we synthesized a FabScalar-generated 4-way superscalar processor to a single Virtex-5 FPGA, leveraging Block RAMs to emulate the core's numerous RAMs and CAMs. (We present FPGA results in Section 5.3.)

This paper makes the following contributions:

- The idea of framing superscalar processors in a canonical form to address the *design-effort problem* posed by single-ISA heterogeneous multi-core. This is also the first paper to bring solutions to bear on this problem.

- The FabScalar toolset which streamlines the design and verification of superscalar processors. The ability to automatically generate synthesizable RTL models of superscalar processors of different widths and depths is unprecedented, as these dimensions are not a simple matter of parameterization.

- Validation of the quality of the generated RTL along three fronts: functional and IPC validation, cycle time validation, and suitability for standard ASIC flows. We also confirm the feasibility of mapping the RTL to a dense FPGA (useful for accelerating verification and architectural exploration) and match the cycle counts from verilog simulation.

- We propose and analyze G21, a generic heterogeneous multi-core design comprised of 21 core types. The core types are selected, not based on *a priori* knowledge of applications, rather, based on providing a broad range of microarchitectural configurations. G21 achieves near-speediest execution on 59 SPEC SimPoints considering the entire design space, despite not tailoring G21 to these

benchmarks. This also implies G21 captures the diversity of the entire design space, and is therefore a far more efficient platform for selecting fewer core types based on workloads.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the methodology used throughout the paper. Section 4 describes FabScalar, including the canonical superscalar processor, the CPSL, and FabMem. Section 5 presents the validation experiments. In Section 6, we apply FabScalar to explore G21, and provide key insights from this exercise. Section 7 discusses extensibility and design and verification effort. Section 8 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. These aspects are not easily parameterized and require FabScalar's approach: an RTL generator that uses the canonical superscalar template and 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 IPC, cycle time, and synthesizability via standard ASIC flows.

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 [28]. FabScalar is unique in that it generates complex superscalar processors and this is evident in the novel composable CPSL.

Palacharla, Jouppi, and Smith [19] developed models for estimating propagation delays of key superscalar pipeline stages (rename, issue, and bypasses). Li et. al describe a comprehensive power, area, and timing modeling framework for multi-core systems, McPAT [12]. 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 sub-pipelining and its imbalances; and *it produces RTL implementations of cores*. FabScalar's RTL output underscores a crucial distinction with computer architecture tools: the goal of FabScalar is to streamline the design, verification, and fabrication of chips, *i.e.*, it is meant to serve as a development tool for designing heterogeneous multi-core chips, not just an estimation tool for research.

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 [23].

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

Since specialized, highly-ported RAMs and CAMs are so pervasive and essential to a superscalar processor, we have developed a tool (FabMem [32]) for generating their physical layouts and extracting timing, power, and area. While memory compilers are not new, we are not aware of any commercial one that can generate RAMs with more than a few ports. For simulation, RAMs and CAMs are represented with behavioral modules. For synthesis and place-and-route, they are replaced with cells and LEF macros, respectively, both from FabMem. The FabMem tool is described in Section 4.2.

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 also abstracted as macros, with timing information obtained from CACTI 5.1 [26]. CACTI uses device and wire parameters derived from ITRS (refer to Tables 4 and 6, respectively, in the CACTI 5.1 report [26]). The BSIM4 Predictive Technology Model used by FreePDK and used for our custom circuit design, is different and more conservative. To bring the cache delays in line with our synthesized and custom logic, we recalculated CACTI's device and wire parameters using the BSIM4 model. Table 2 shows the original and adjusted CACTI device parameters for 45nm. The table of wire parameters is large and is omitted due to space constraints.

Table 2. Original and adjusted CACTI device parameters (45nm).

Device Parameter	HP/LSTP/LOP ITRS Model	BSIM4 Predictive Technology Model
L_{gate} (nm)	18/28/22	22.6
EOT (equiv. oxide thickness) (nm)	0.65/1.4/0.9	1.2
V_{DD} (V)	1/1.1/0.7	1.1
V_{th} (mV)	181/532/256	400
I_{on} ($\mu A/\mu$)	2047/666/749	1110
$C_{ox\text{-}elec}$ (fF/μ^2)	37.7/20.1/28.2	31.3
FO4 delay (ps)	8.21/31.2/17.86	24.22

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

4. FABSCALAR

This section describes the FabScalar toolset, including the canonical superscalar processor and CPSL (Section 4.1) and FabMem (Section 4.2). Section 1 described the benefits of FabScalar's verilog/C++ co-simulation environment and the cycle-accurate configurable C++ model. Due to space constraints, however, we do not elaborate further on them.

4.1 Canonical Superscalar Processor and the 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.
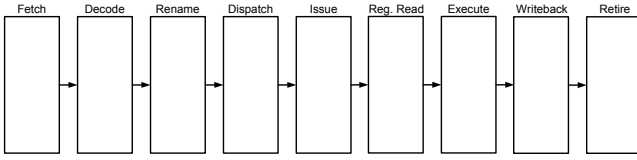
| Fetch | Decode | Rename | Dispatch | Issue | Reg. Read | Execute | Writeback | Retire |

Figure 1. Canonical superscalar processor.

The CPSL instruction-set architecture (ISA) is PISA [3], a close derivative of the MIPS ISA (minus load and branch delay slots). The rationale for using a simple RISC ISA is three-fold: (1) As a practical matter, our primary experience is with MIPS. (2) Contemporary processors often dynamically transform the binary level ISA into an implementation ISA sometimes referred to as micro-operations. Micro-operations resemble RISC primitives. One can view the chosen CPSL ISA as a canonical implementation ISA. Binary translation technology provides another path for supporting different binary level and implementation ISAs [4]. (3) A simpler implementation ISA results in a smaller core.

The CPSL contains many synthesizable RTL designs for each canonical pipeline stage, that differ in their superscalar width and depth of sub-pipelining. Table 3 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 (FUs) are required: one each of simple ALU, complex ALU, load/store port, and branch unit. Narrower issue widths can be accommodated by aggregating multiple FU types into one pipeline way, which we have left for future work. The maximum width of all back-end stages is 8-way superscalar.

The second column of Table 3 also shows ranges of 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). Dispatch has only 1-deep implementations in the CPSL and Retire has only 2-deep implementations. All other stages have a range of depths. Fetch goes the deepest, ranging from 2-deep to 5-deep. This is a result of the fetch unit having substantial logic in two phases, Fetch-1 and Fetch-2. Fetch-1 corresponds to accessing all the structures (2-way interleaved instruction cache, branch target buffer, branch predictor) and it also implements the complex next-PC logic. Fetch-2 corresponds to all logic after the instruction cache for extracting the fetch block from two cache blocks and aligning it for the decode stage, including branch predecode logic. Thus, Fetch is at least 2-deep: 1-deep for each of Fetch-1 and Fetch-2. A 2-deep version of Fetch-1 was designed and features block-ahead prediction [20], a somewhat elaborate approach for effectively pipelining the branch prediction logic. To our knowledge, this may be the first RTL implementation of it [30]. The Decode and Rename stages are separated by an instruction buffer (Fetch Queue), which facilitates assembling the full superscalar width of instructions for the Rename and Dispatch stages. The Decode stage can be as deep as 3-deep due to the logic complexity for steering and writing instructions into the buffer, including cracking doubleword instructions into two micro-ops. The Rename stage varies from 1-deep to 3-deep, with deeper versions increasing the amount of hazard logic for dependencies among multiple groups in-flight in the Rename stage. The CPSL provides five sub-pipelined designs of the Issue stage [31]:

- "1-cycle issue / 1-cycle loop" (1/1): 1 cycle for the Issue stage as a whole (*i.e.*, no sub-pipelining), including wakeup, select, and reading the payload RAM. 1 cycle for the critical wakeup+select loop.

- "2-cycle issue / 1-cycle loop" (2/1): 2 cycles for the Issue stage as a whole, but only 1 cycle for the critical wakeup+select loop. The logic partitions are: (a) wakeup+select and (b) payloadRAM. An optimization makes the selected instruction's destination tag available prior to reading the payloadRAM, decoupling it from the wakeup-select loop.

- "2-cycle issue / 2-cycle loop" (2/2): 2 cycles for the Issue stage as a whole and 2 cycles for the critical wakeup+select loop. The logic partitions are: (a) wakeup and (b) select+payloadRAM.

- "3-cycle issue / 3-cycle loop" (3/3): 3 cycles for the Issue stage as a whole and 3 cycles for the critical wakeup+select loop. The logic partitions are: (a) wakeup, (b) select, and (c) payloadRAM.

- "3-cycle issue / 2-cycle loop" (3/2): Same as 3/3, but only 2 cycles for the critical wakeup+select loop (as with 2/1, the payloadRAM is decoupled from wakeup-select loop).

The Register Read stage ranges from 1-deep to 4-deep. Based on experience with FabMem, we pipeline the wordlines of the Physical Register File, *i.e.*, a sub-word of the register is returned each cycle. Due to pipelining the wordlines, Writeback takes the same number of cycles (write one sub-word per cycle). A nice feature is that the canonical interfaces for bypass buses are invariant with the degree of sub-pipelining: a value is bypassed to canonical pipeline stages Execute and Register Read irrespective of their underlying implementations, and a bypassed value is steered appropriately to one or multiple sub-stages within Register Read.

The third column in Table 3 shows stage-specific structures that are implemented in the RTL. 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 3.

The final column in Table 3 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 3 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. 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 are of no use for subsets of the workload space can be excluded from a core to streamline its frequency and static and dynamic power. Specific design choices that are represented in the current CPSL are highlighted in boldface in the last column of Table 3.

Table 3. 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)
		L1 Instruction Cache
			Branch prediction algorithm
			No interleaving vs. **2-way interleaving**
			Block-based BTB vs. **interleaved BTB**
			Multi-cycle fetch:
			    unpipelined
			    pipelined using block-ahead prediction [20]

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, 2/1, 2/2, 3/3, 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 [2]
			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) 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

4.2 FabMem: A Multiported RAM and CAM Compiler

We custom-designed RAM and CAM bitcells with a wide range of port configurations supporting scalar through 8-way superscalar pipelines. Bitcell designs, address decoder blocks, wordline drivers, precharge circuits, write circuits, column muxes, sense amps, and input and output drivers, form a cell library for composing RAMs and CAMs. Figure 2 shows the layout of a 16R8W RAM bitcell.

Figure 3 shows the FabMem tool flow [32]. The first script takes as input the logical depth (D) and width (W) of the memory as well as the desired number of read (Rp) and write ports (Wp). The script references the cell library to generate a SPICE netlist of the critical path of the memory. It generates three different netlists, for three different degrees of column muxing (1, 2, and 4), to find a good underlying physical geometry. It also generates top-level simulation files that apply performance stress tests. These tests induce worst-case coupling capacitance scenarios (e.g, bitlines transitioning in opposite directions) and loading capacitance scenarios (e.g., heavy loading of the bitcell by reading it simultaneously on all ports). The SPICE simulation results are tabulated and the user identifies the best degree of column muxing (DC) based on this initial exploration.
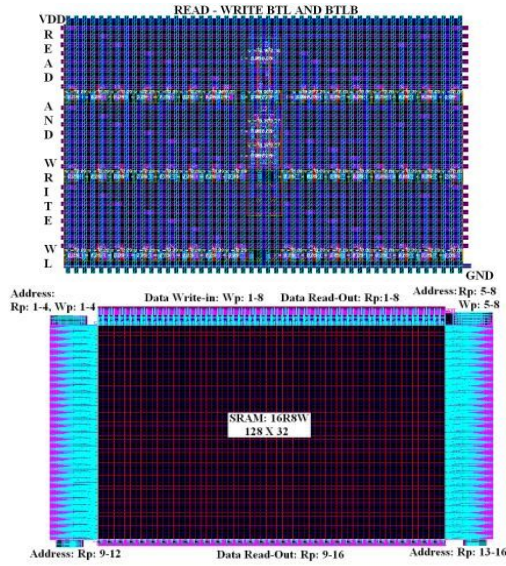
Figure 2. Top: Layout of 16R8W RAM bitcell. Bottom: 128-entry 32-bit 16R8W register file macro.

A SKILL script takes inputs D, W, Rp, Wp, and now DC. SKILL is a scripting language for automatically generating layouts. Cadence Virtuoso runs the SKILL script and it generates a full layout of the memory, referencing the cell library. In parallel, the user runs a third script that also takes D, W, Rp, Wp, and DC as input, and generates a full source netlist of the memory. Meanwhile Calibre DRC runs design rule checks on the full layout. After passing DRC, Calibre LVS verifies that the full layout is logically equivalent to the source netlist. After confirming this, Calibre PEX is run on the full layout in order to extract parasitics which are combined with the netlist for performing full layout SPICE simulation. The results of the full layout simulation can be compared against results from the faster, critical-path only simulation performed earlier to validate the latter or firm up timing and power estimates. Moreover the full layout can be used as a hard macro. For example, Figure 2 shows a 128-entry 32-bit 16R8W register file macro.

Figure 4 compares access times produced by CACTI 5.1 (adjusted for BSIM4 predictive tech. model) and the RAM tool, varying size and port configuration (1R1W, 4R4W, and 8R8W). CACTI 5.1 requires a minimum block size of 8B so this width is used. The two tend to differ more at smaller RAM sizes (CACTI does not produce valid results for 128B) or for a high number of ports, two reasons that motivated us to develop the tool. That said, they produce similar results at 4KB for 1R1W and 4R4W.
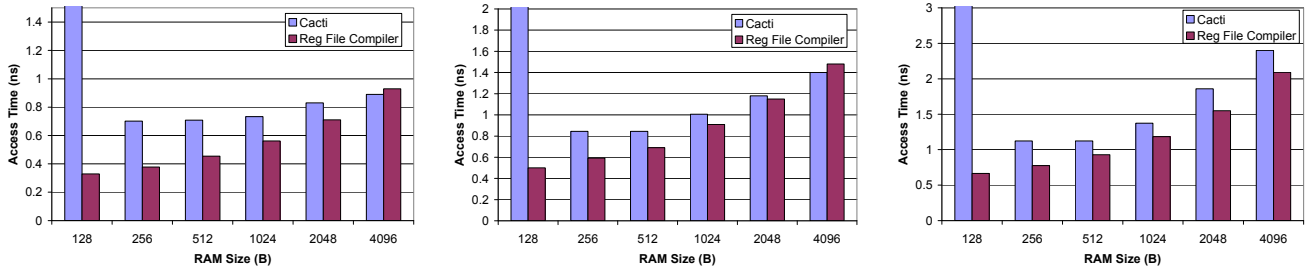

Figure 3. Flow for generating RAM/CAM macros.

5. RESULTS: VALIDATION

In this section, we evaluate the quality of the RTL designs produced by FabScalar. Prior to the validation experiments presented in this section, we performed extensive unit-level testing of the CPSL and testing of a baseline 4-way superscalar core. The latter enabled perfecting the interfaces and interactions among canonical pipeline stages. Subsequently, assembling and debugging all of the cores in this section proceeded efficiently within the span of a month, moreover, most of the bugs encountered in this period were in the course of implementing the composition tool itself (scripted verilog instantiation and stitching of stages).

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 [21] 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. The other six integer benchmarks were not tested because their SimPoints have occasional floating-point instructions and will be tested in future work when CPSL stage designs are augmented to handle floating-point instructions.

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

Figure 4. Comparison of CACTI (using BSIM4 Predictive Tech. Model) and RAM tool. From left to right: 1R1W ports, 4R4W ports, and 8R8W ports.
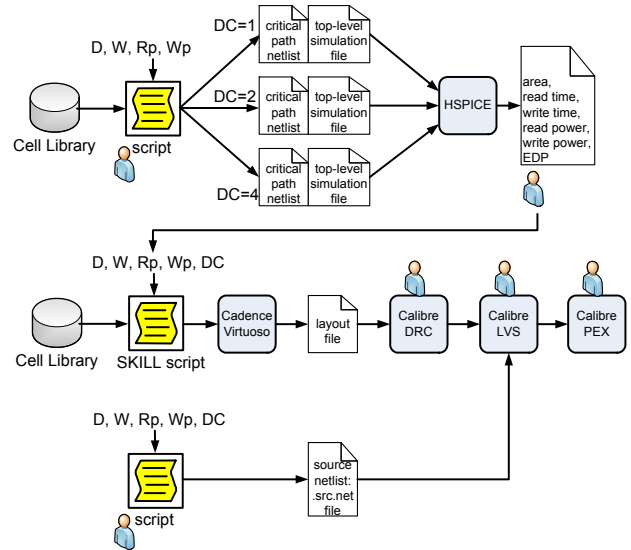
generated cores. Validating cycle time is challenging and imperfect for a number of reasons:

- Different technology nodes, technology libraries, and foundry processes. We deal with this issue by converting cycle time into the number of FO4 inverter delays of the technology, yielding a technology-independent comparison (although the subtle influence that the underlying technology had on design choices and circuit opimization choices cannot be undone).

- 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 compare to 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 does not have Issue stage designs with multiple schedulers (see Table 3, 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 pipeline way, at the cost of some load imbalance among the multiple issue queues. More importantly, when there are multiple FUs of the same type, providing each FU 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. While these techniques are not yet represented in the CPSL, their effect can be modeled for timing validation purposes by applying a smaller/simpler issue queue and a register file with fewer read ports.

3. *Suitability for physical design*: We demonstrate the suitability of the generated RTL for full synthesis and place-and-route by a standard ASIC flow.

5.1 Functional and IPC Validation

The FabScalar tool was used to generate the RTL designs for the twelve cores described in Table 4. Before discussing the cores, several points about the table need clarification. First, some stage depths are omitted from the table: stages that have only a single depth implemented in the CPSL (Dispatch and Retire are always 1-deep and 2-deep, respectively) or stages that are not varied among the twelve cores (Decode is fixed at 1-deep). Second, the Branch Order Buffer (BOB) is a FIFO buffer in the Fetch stage holding control information for all predicted but not yet retired branches. It facilitates updating the Branch History Table (BHT) non-speculatively as branches retire as well as checkpointing certain predictor state (such as the global branch history register). Third, 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). Of the two branch recovery implementations represented in the CPSL – shadow maps vs. handle like exception (Table 3) – all twelve cores employ the latter lower complexity and lower performance approach.

Cores 1 through 6 were selected 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: 2-way superscalar in the front-end. Core-2 has different widths in the front-end (4) and back-end (6). Core-5 is a particularly wide core: 8-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 sub-stage 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 twelve cores (fetch-to-execute = 15), although not the deepest possible with the CPSL since 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 [13] instead of the bimodal branch predictor. Since 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.

Results of executing the 100 million instruction SimPoints of six benchmarks are shown in Figure 5. 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 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 are more likely inherent in the design rather than in the RTL modeling of the design.

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 since it is 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 due to increasing some parameters and decreasing others, leading to potentially non-monotonic cores. For example, Core-6 has the same or higher IPC than Core-2 in all benchmarks except bzip. Core-6 is narrow (2-way fetch) but its advantage over other configurations is its 1-cycle wakeup-select loop. In the case of bzip, however, there is apparently sufficient ILP to outweigh the longer wakeup-select loop.

Table 4. 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

If there are anomalies, for example, a more aggressive core having lower IPC than a simpler core of the same pipeline depth, they 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.

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 [22], 180nm Alpha 21364 [14], and 65nm MIPS32 74K [7][29]. All three implement RISC ISAs and they represent extremes from highly custom designed to fully synthesized (MIPS32 74K). Table 5 shows major microarchitecture parameters of the three processors.

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 [7][14][22][29].

The shaded section in Table 5 shows delay comparisons between the commercial cores and similarly configured FabScalar generated cores. Five numbers are shown:

1. Cycle time of the commercial core.

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 sub-pipelining is needed to compensate for the lesser degree of custom design of the FabScalar core.

4. Raw fetch-to-execute delay of the FabScalar core. 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 fetch-to-execute pipeline depth of the commercial core. 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 cycle time of the FabScalar-Power5 is relatively close to that of the Power5: 29 FO4 compared to 23 FO4, respectively. Slightly
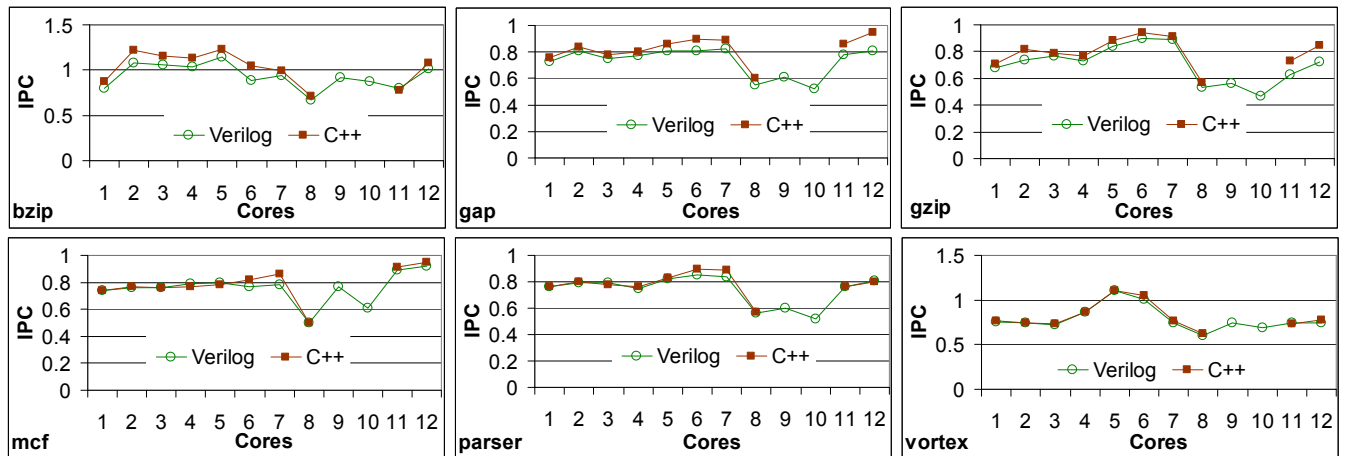

Figure 5. Results of executing 100 million instruction SimPoints of six benchmarks on the twelve cores.

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.

Table 5. 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
Physical Register File(s)	Int:120, FP:120	Int:80, FP:72	64
Load Queue / Store Queue	32 / 32	32 / 32	8 / 8
L1 I$ / L1 D$ (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

A larger difference is observed for the FabScalar-21364 and 21364: 37 FO4 compared to 25 FO4. What is 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.

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 demonstrate that FabScalar-generated RTL can be taken through standard ASIC flows, we synthesized and place-and-routed a 4-way superscalar processor. The physical design is shown in Figure 6(a). This particular core uses shadow maps for quickly recovering from mispredictions. Since FabMem cannot generate the highly specialized shadow map to rename map connectivity, the rename map table and shadow map table are synthesized to flip-flops in this physical design: this is evident in the large rename block in the chip diagram.

We also synthesized the same core, minus the shadow maps, to a Virtex-5 FPGA in a BEE3 system, shown in Figure 6(b). Results of the FPGA experiments are shown in Table 6. The FPGA

produces the same results as verilog simulation: number of retired instructions, number of cycles, and IPC (first three rows of the table). The FPGA yields a tremendous reduction in simulation time (2,000–5,000x speedup). The FPGA speed is 50 MHz, although due to virtual cycles for FPGA management the effective speed (simulated cycles ÷ simulation time) is 7 MHz to 15 MHz, still substantial.
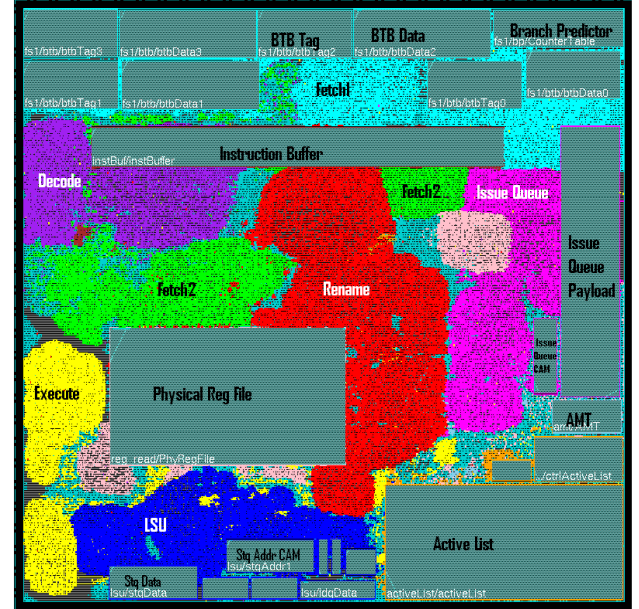
(a) Physical design of a 4-way superscalar processor.
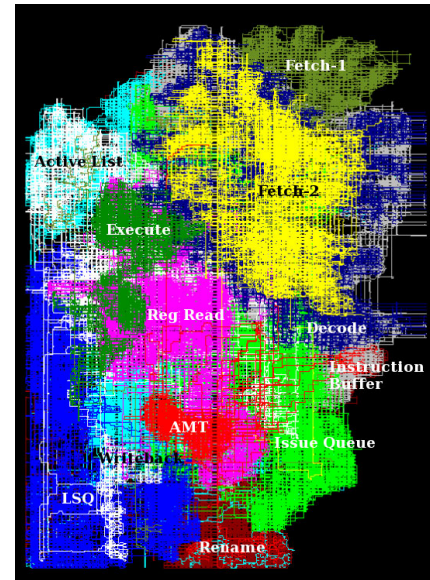
(b) Same design synthesized to Virtex-5 FPGA.
Figure 6.

Table 6. FPGA results with several 10M SimPoints.

	bzip	gzip	mcf	parser
FPGA & verilog retired instr.	10000000	10000000	10000000	10000000
FPGA & verilog cycles	11239464	17000790	8342999	11588294
FPGA & verilog IPC	0.89	0.59	1.20	0.86
FPGA simulation time (s)	0.75	1.22	1.21	0.87
verilog simulation time (s)	4,018	5,536	2,870	3,748
simulation speedup	5,357	4,538	2,372	4,308
FPGA speed (MHz)	50	50	50	50
FPGA effective speed (MHz)	15	14	7	13

6. RESULTS: ANALYSIS OF A WORKLOAD-AGNOSTIC HETEROGENEOUS MULTI-CORE

In this section, we explore a heterogeneous multi-core design with the goal of maximizing single-thread performance. Prior work achieved similar objectives by exhaustively simulating applications on all cores [16][17] or multiprogrammed workloads on all core combinations [10], in a huge design space. Aside from the computational complexity of this approach, there is some concern about the performance robustness of a design trained to a specific workload.

These perceived drawbacks motivate us to take a very different approach. Cores are selected to provide a broad range of microarchitectural configurations without *a priori* knowledge of applications. We intentionally do not customize cores or core combinations to a specific workload (other than as optimal yardsticks), because we want a heterogeneous multi-core design that maximizes single-thread performance for *arbitrary instruction-level behavior*. Our first, unsophisticated application of this idea produced a heterogeneous multi-core with 21 core types, which we call G21: the "G" stands for *generic* heterogeneous multi-core, a reference to its broad diversity that was not influenced by a specific workload.

This section is organized as follows. Section 6.1 describes the design space and performance metric. Section 6.2 describes the design process for coming up with G21. Section 6.3 describes the benchmarks. Section 6.4 presents results and analysis of G21 and its derivatives.

6.1 The Design Space

Using FabScalar's CPSL, we created a design space of almost 38,000 different cores with cycle times that range from 0.5ns to 1.2ns in 0.1ns increments. For a given width and depth of a canonical pipeline stage, its stage-specific structures are maximized within the cycle time constraint (thus pruning designs that underutilize slack). IPC is measured using FabScalar's cycle-accurate C++ simulator (faster than verilog simulation, and we can also include floating-point benchmarks in the exercise). Performance is measured in billions-of-instructions-per-second (BIPS), the product of IPC and frequency.

6.2 G21: The Proposed Workload-Agnostic Heterogeneous Multi-core

The proposed workload-agnostic design provides superscalar widths of 2 through 8. For each width, a cycle time range is chosen commensurate with the width, *i.e.*, narrower cores have a faster cycle time range than wider cores. Table 7 shows the three cycle times selected for each width. The three cycle times accommodate small, medium, and large structure sizes for that width and maximum depth. This yields a total of 21 core types, three sizes/frequencies for each of seven widths.

Table 7. The 21 cores.

		larger structures →		
		← higher frequency		
		cycle time (ns)		
superscalar width	2 or 3	0.5	0.6	0.7
	4 or 5	0.6	0.7	0.8
	6 or 7	0.7	0.8	0.9
	8	0.8	0.9	1.0

6.3 The Benchmarks

We use integer and floating-point benchmarks from SPEC. The SimPoint tool [21] was used to select up to four or five 10 million instruction SimPoints from each integer or floating-point benchmark, respectively. In all, there are 59 SimPoints in the experiments, which we call "benchmarks" from now on.

6.4 Results and Analysis of G21

We assume ideal steering: for a given multi-core design, the performance reported for a benchmark is the highest performance attainable among all cores in the design. We consider the following multi-core designs:

- **Best-1**: This design has only 1 core type. It is the best core in the entire design space, on average, *i.e.*, Best-1 achieves the highest harmonic-mean of BIPS (over all the benchmarks) of any one core.

- **G21**: This is the proposed generic heterogeneous multi-core with 21 core types (Section 6.2).

In all graphs that follow, the performance of a given design is normalized to the peak performance attainable by the benchmark considering the entire design space (BIPS / Peak_BIPS).

Figure 7 shows the performance of G21. G21 is remarkably robust. On average, G21 comes within 3% of peak performance. More importantly, no individual benchmark achieves less than 88% of its peak performance considering the entire design space. This level of performance robustness is not due to a lack of diversity among benchmarks: the performance of Best-1, also shown in Figure 7, while impressive for just a single core type, shows severe sub-optimality for many benchmarks. On average, it is 10% off of peak performance. The real problem with Best-1 is two-fold: individual benchmarks *for which the Best-1 design was trained* are as much as *30% off of their peak performance*. Being heavily trained to this workload, it is possible to observe even more severe sub-optimality for other applications.

The fact that G21 was not trained for these benchmarks, and yet attains close to peak performance with no severe outliers, highlights the merits of workload-agnostic design: low computational complexity and some relief from concerns about performance robustness. Whereas G21 is not practical to design today, FabScalar may make it practical in the near future.
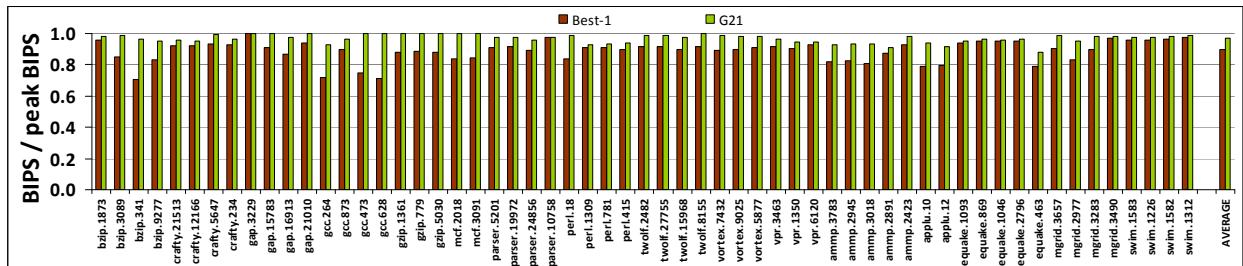

Figure 7. Performance of G21 and Best-1, normalized to peak performance considering entire design space.

We also infer that, from the standpoint of single-thread performance, the microarchitectural diversity provided by G21 is highly representative of the entire design space. In other words, G21 can be used in lieu of the full design space if we *do* want to distill a smaller number of cores by leveraging knowledge of the workload. We consider the following multi-core designs:

- **Best-N-of-G21**: This design has N core types (we vary N) selected from G21's core types. These cores constitute the best N-core combination from G21, on average (harmonic-mean of BIPS).

- **Robust-N-of-G21**: This design has N core types (we vary N) selected from G21's core types. The N cores are selected to minimize the maximum performance degradation for any one benchmark, compared to its peak performance on G21.

Table 8 shows results of these two consolidation strategies as N is varied. The BIPS of each design is shown normalized to the peak BIPS on G21. Also shown is the maximum degradation observed for any one benchmark (compared to its peak performance on G21). For N=2–5, there is a tradeoff between maximizing average performance and minimizing the impact on any one benchmark. Best-4 and Robust-4 both use one more core than Best-3, but to different ends. Best-4 maintains the same worst-case individual-benchmark degradation as Best-3, but improves average performance. In contrast, Robust-4 achieves the same average performance as Best-3, but bounds worst-case individual-benchmark degradation to 11% compared to 15.4% for Best-3. Evidently, Robust-4 picks an extra core that is more "outlier" than "average", to serve the most "outlier" benchmark, whereas Best-4 picks an extra core that is more "average" than "outlier" to serve more benchmarks. In both approaches, outlier benchmarks tend to dominate the selection of additional cores at a certain point (N>5).

Table 8. Using a smaller number of cores from G21.

N	Best-N-of-G21		Robust-N-of-G21	
	BIPS/ Peak-BIPS-G21	Max % degradation	BIPS/ Peak-BIPS-G21	Max % degradation
1	0.924	27.1	0.924	27.1
2	0.965	19.3	0.951	14
3	0.977	15.4	0.967	14
4	0.985	15.4	0.976	11
5	0.992	11	0.980	10
6	0.996	5.1	0.996	5.1
7	0.998	3.9	0.998	3.9
8	0.998	3.9	0.998	3.9

7. DISCUSSION
7.1 Extensibility
While it is difficult to make generalizations about extensibility, we present two specific examples. The examples relate to two major IPC bottlenecks in the FabScalar-generated cores presented earlier: load misspeculations and branch mispredictions.

In the cores presented earlier, completion of a load is not stalled by prior unissued stores. If the speculatively-completed load is later discovered to depend on one of the stores, the recovery penalty is severe: the processor waits until the load reaches the head of the active list, squashes the load and all instructions after it, and restarts from the load. It took one of the authors two days to enhance the CPSL with a simple dependence predictor (predict a load will conflict if it has in the past) and logic to stall completion of suspect loads in the load queue until all prior stores have issued. There is already a mechanism to complete loads that were once stalled (for cache-missed loads): the load is reinjected into the load-store pipe (in the next free cycle) at which time it is

safely completed. The changes are localized to the Dispatch stage (dependence predictor) and load queue (bit vectors for tracking store queue entries of unissued stores). The global impact is small: there is an additional signal from the Retire stage to Dispatch stage (PC of a misspeculated load trains the dependence predictor) and loads carry an extra bit with them (whether or not to speculatively complete). Two of the benchmarks, bzip and vortex, have quite low branch misprediction rates, hence, their performance is sensitive to load misspeculations. Consequently, adding the dependence predictor increased the IPC of bzip from 0.89 to 1.39 and the IPC of vortex from 0.76 to 1.14 (for 10M SimPoints).

In the second example, one of the authors changed the Fetch-1 stage to totally decouple the conditional branch predictor (taken/not-taken prediction) from the next-PC logic (BTB, RAS, and next-PC mux), enabling pipelining a large/complex branch predictor arbitrarily deep (high accuracy with fast cycle-time) [30]. This change took longer to implement than the previous example because it is more complex and required inventive design. Nevertheless, changes were confined to the Fetch-1 stage.

7.2 Design and Verification Effort
FabScalar does not completely eliminate design effort or take the designer out of the picture. Rather, it boosts designer productivity by generating RTL designs of whole cores, the starting point for design tuning, verification, and physical design.

An open question is whether or not FabScalar reduces verification effort and by how much. In future work, we would like to reduce verification to (1) formally proving that the canonical superscalar template is correct and (2) "certifying" specific implementations of canonical pipeline stages, in isolation. Aside from formal methods, as open-source gateware, verification coverage may increase over time through a community of users.

8. 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 microarchitecturally diverse cores.

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

We plan to release the FabScalar toolset for other researchers and developers to use and expand, as open-source gateware. We intend to add floating-point instruction support, media instruction support, and TLBs; implement the MIPS ISA so that the latest gcc toolchain can be leveraged; relax backend constraints (multiple of any FU type, any combination of FUs in each execution pipeline, etc.); and explore microarchitectural alternatives, such as those outlined in Table 3. The problem of choosing a set of diverse cores to maximize single-thread performance on arbitrary applications, deserves a more comprehensive treatment and should also consider power. A closely related problem is matching programs and program phases to cores at run-time. Applying

formal verification methods and tools in the FabScalar framework is worth pursuing. We are currently working on automating the mapping of FabScalar-generated cores onto FPGAs for accelerating verification, superscalar processor simulation, and design space exploration.

9. ACKNOWLEDGMENTS

10. REFERENCES

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

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

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

[4] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. *Int'l Symp. on Code Generation and Optimization*, March 2003.

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

[6] 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.

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

[8] 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. *Int'l Symposium on Microarchitecture*, Dec. 2003.

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

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

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

[12] 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 Symposium on Microarchitecture*, Dec. 2009.

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

[14] E. J. McLellan, D. A. Webb. The Alpha 21264 Microprocessor Architecture. *Int'l Conference on Computer Design*, Oct. 1998.

[15] 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.

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

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

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

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

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

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[22] 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.

[23] 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 Conference on Microelectronic Systems Education*, 2007.

[24] L. Strozek, D. Brooks. Efficient Architectures through Application Clustering and Architectural Heterogeneity. *Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

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

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

[27] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. *Int'l Conference on Dependable Systems and Networks (DSN)*, 2004.

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

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

[30] J. Gandhi. FabFetch: A Synthesizable RTL Model of a Pipelined Instruction Fetch Unit for Superscalar Processors. M.S. Thesis, ECE Dep't, NC State University, June 2010.

[31] H. Mayukh. FabIssue: Automatic RTL Generation of Issue Logic in Superscalar Processors for Core Customization. M.S. Thesis, ECE Dep't, NC State University, June 2010.

[32] T. A. Shah. FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration. M.S. Thesis, ECE Dep't, NC State University, May 2010.

* The following authors contributed to this paper while at North Carolina State University. Tanmay A. Shah is now with Intel, LTD Advanced Design, Hillsboro, OR, USA. Hiran Mayukh and Jayneel Gandhi are now with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI, USA. Hashem H. Najaf-abadi is now with Intel, Visual & Parallel Computing Group (VPG), Folsom, CA, USA.