FabScalar

Niket Choudhary, Salil Wadhavkar, Tanmay Shah, Sandeep Navada, Hashem Hashemi, Eric Rotenberg
Department of Electrical and Computer Engineering, North Carolina State University

1. FabScalar: Motivation and Concept

A high-performance superscalar processor achieves good performance across a wide range of applications. For any one application, however, its generalized design is not the highest-performing one. Tailoring the design to an application will yield the highest-performing combination of clock frequency and instruction-level parallelism (ILP) for the application, and the lowest power consumption for this level of performance.

Suboptimal performance on individual applications was overlooked in the past, when conventional microarchitecture and technology scaling yielded exponential growth in single-thread performance. Now, customizing superscalar processors to applications, classes of applications, or classes of application behavior, may be the chief source of single-thread performance scaling available to designers in the near future.

A single application-customized superscalar processor is not robust across applications, however. Thus, it is not economically viable in general-purpose systems running a range of workloads. A heterogeneous multi-core chip [5], comprised of many differently-designed superscalar processors, is an attractive solution to this dilemma. Unfortunately, designing, verifying, and fabricating a heterogeneous multi-core is not economically viable for a different reason: it has too many different superscalar processors, multiplying design and verification effort. This is partly why, from a heterogeneity standpoint, current commercial heterogeneous multi-core designs are limited to a few somewhat obvious core types, *e.g.*, one complex superscalar processor combined with replicated simple superscalar processors [1]. The insurmountable design and verification effort is also why initial academic work made the interesting proposition of using multiple past and current generation designs for the constituent superscalar processors [2].

We are currently developing a novel toolset for automatically composing arbitrary superscalar processors, called FabScalar. It exploits the idea that different superscalar processors have in common a canonical pipeline, shown in Figure 1 (Appendix), and differ primarily in the complexity and sub-pipelining within each canonical pipeline stage:

- *Complexity*: The complexity of a canonical stage is parameterized by its superscalar width (number of pipeline "ways") and the sizes of RAMs, CAMs, and other specialized memories within the stage. Increasing the complexity of a canonical stage may contribute to extracting more instruction-level parallelism in the program (*i.e.*, reducing the number of cycles to execute the program) but typically increases the logic delay through the canonical stage. The effect of increasing logic delay on overall performance ultimately depends on the next differentiating factor.
- *Sub-pipelining*: A canonical stage is nominally one cycle in duration, but may be sub-pipelined deeper to achieve a higher clock frequency. Sub-pipelining of some canonical stages may be straightforward (*e.g.*, instruction-decode), whereas other canonical stages with "single-cycle loops", such as instruction-fetch and instruction-issue, may demand clever speculative techniques to get the most benefit from sub-pipelining, *e.g.*, select-free scheduling for the issue stage [9], block-ahead branch prediction for the fetch stage [8], etc. The sub-pipelining of a canonical stage is parameterized by depth (one, two, or three sub-stages) and the optional employment of loop-breaking techniques.

Accordingly, our approach is to
1) define composable interfaces of canonical pipeline stages, so that they can be stitched together to compose an overall superscalar processor,
2) pre-design multiple versions of each canonical pipeline stage, that differ in their width and stage-specific structure sizes (complexity) and depth (sub-pipelining), and
3) develop a high-level superscalar synthesis tool that can automatically compose an arbitrary superscalar processor based on processor-level and stage-level constraints (frequency, power, and area), and output multiple representations (verilog, cycle-accurate C++, netlist, and physical design) of the processor.

FabScalar is a first step for the practical development of heterogeneous multi-core systems [5] and application-specific superscalar processors [10]. The former will provide a better microarchitectural bridge between diverse workloads and underlying technology, than current general-purpose processors. The latter may make it possible for embedded system designers, without niche expertise of superscalar processor design, to develop programmable solutions that are good enough to obviate ASIC solutions.

As a research framework, FabScalar empowers computer architecture researchers to evaluate designs with unprecedented fidelity and detail. For example, FabScalar can be used to exhaustively populate the huge superscalar design spaces used by researchers in this area [5][10][11] with real designs, reflecting real frequency, power, and area. Sub-pipelined stages also reflect real imbalances that arise due to natural boundaries for pipelining logic. Previously, researchers have either not accounted for frequency [5], despite it being a key beneficiary of customization, or assumed ideal (totally balanced) sub-pipelining [11]. Overall, FabScalar improves the fidelity of architectural exploration through whole-pipeline customization (capturing the interplay among stages) and high-fidelity information (physical design). Beyond customization, 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 of value to researchers.

As a development framework, FabScalar empowers small teams of designers to design, verify, and fabricate true heterogeneous multi-core chips. Optimistically, this could lead to the creation of a new technology sector wherein many small companies meet the demand for highly differentiated products.

The FabScalar project is in its first year. The toolset is still in the concept stage and what we have nearly completed in the first year is an initial synthesizable-verilog model of a 4-way superscalar processor, that will be the basis for 1) defining composable interfaces among canonical pipeline stages, and 2) gradually populating a rich library of pipeline stage designs. Also well underway are matching C++ models of each stage and

a memory compiler for generating specialized RAMs (highly-ported register files and rename tables, active list, free list, predictors, etc.). Our aim of submitting to WARP'09 is to present the FabScalar concept, its future direction, and its current status. §2 delves more into the proposed toolset, §3 presents brief status of initial synthesizable-verilog model, §4 discusses related work, and the Appendix contains Figures and Tables.

2. FabScalar Toolset

The eventual toolset will be comprised of a Standard Superscalar Library (SSL), a high-level superscalar synthesis tool, and an in-house memory compiler.

The SSL contains the designs of canonical pipeline stages. Each canonical pipeline stage will have many different designs that differ in their superscalar width, sub-pipelined depth, stage-specific structure sizes, and, where applicable, microarchitectural enhancements or styles (*e.g.*, breaking vs. not breaking single-cycle loops). Each design consists of verilog source code, matching interface-accurate and cycle-accurate C++ code, and, optionally, a synthesized netlist and physical design (hard macro).

The superscalar synthesis tool composes an overall superscalar processor using the SSL and high-level synthesis constraints. The designer can specify both processor-level constraints – frequency, power, and area of the processor as a whole – and stage-level constraints – per-stage frequency, power, and area, and per-stage configuration information (width, depth, etc.). Typically, partial constraints will be given and the superscalar synthesis tool composes the best processor in terms of IPS/Watt/mm^2 that meets specified constraints and optimizes unspecified constraints. The tool produces three representations for the whole processor: 1) verilog, 2) C++, 3) netlist and placed-and-routed physical design. The verilog and C++ can be used as separate simulators or as tightly integrated simulators, as shown in Figure 2. The Cadence environment allows compiled C++ modules to be called from verilog modules. When tightly integrated, the C++ and verilog implementations of each pipeline stage cross-check their outputs every cycle, enabling debugging of both simulators. This approach assures a cycle-accurate C++ simulator that can be used standalone for fast architectural exploration (IPC). This approach also assists debugging the verilog. In addition, a separate functional simulator can be run alongside both the C++ and verilog timing simulators, to assert correctness of retired results. We currently have the standalone C++ simulator (Figure 2b) and standalone verilog simulator (Figure 2c), both checked by integrated functional simulators.

Cacti [6] is not intended for highly-ported memories and is also of limited accuracy without explicit SPICE simulation. The next section summarizes status of our memory compiler.

3. Initial Synthesizable-Verilog Model

The microarchitecture of our initial synthesizable-verilog model is shown in Table 1. At present, we have successfully retired 46 instructions from a microbenchmark, including correctly recovering from mispredicted branches. Further progress of the microbenchmark is pending finalizing the design of the load-store unit: the load-hit datapath (get data from either the Store Queue or L1 Data Cache) and load-stall logic (stall for prior unknown store address) are implemented, but "unstalling" and retrying the load via the load-hit path is not implemented yet.

Sample results:

- Figure 3 shows the placed-and-routed 4-way superscalar processor, excluding the L1 I- and D-caches. Our memory compiler was used to obtain timing and area of the 4R+1W-ported BTB tag and data arrays, the 128-entry 8R+4W-ported physical register file, and 128-entry 4R+4W-ported active list. All are encapsulated as LEF macros in the place-and-route process. Currently, other renaming structures (maps, shadow maps, free list) and the issue queue are synthesized from D flip-flops but these will be replaced soon with RAMs as well.
- Figure 4 shows logic delays of key canonical pipeline stages as their complexity is varied. The timings are from synthesis, not place-and-route. Timings of bypasses are from SPICE.
- Table 2 shows IPC results of some SPEC benchmarks on the cycle-accurate C++ simulator. The IPCs are currently low due to known bottlenecks, discussed below the table.
- Figures 5-7 show multi-ported bit-cell layouts, hierarchical RAM design (global and local bitlines), and coupling capacitance models (a significant effect on latency) that are used by our memory compiler.

4.Related Work

Kumar, Tullsen, and Jouppi [2][5] and Strozek and Brooks [4] have done groundbreaking research on architectural exploration for heterogeneous CMPs. Strozek and Brooks' work on the high level synthesis of very simple cores for embedded systems [4] is more directly related to FabScalar itself. The Program-In-Chip-Out (PICO) framework out of HP labs [7] is closely related in that it customizes VLIW cores and non-programmable accelerators for embedded applications. FabScalar is distinct in that it targets complex superscalar processors and this is evident in the novel composable SSL.

The Illinois Verilog Model (IVM) [3] provides the verilog for a semi-parameterizable 4-issue superscalar processor. In the architecture community, IVM is a groundbreaking effort and in part influenced the genesis of this project. Drawbacks of the current IVM are its unsynthesizable or poorly synthesizable (low frequency) verilog modules, although these have been noted for improvement in the future by the authors. Aside from this, FabScalar has an entirely different motivation, namely the practical fabrication of arbitrary heterogeneous multi-cores, and this leads to key distinctions such as the SSL, its composable pipeline stages, and physical designs (more than verilog).

[1] H. P. Hofstee. Power Efficient Processor Architecture and the Cell Processor. *HPCA*, 2005.

[2] R. Kumar, et al. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. *MICRO*, Dec. 2003.

[3] N. J. Wang, et al. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. *DSN*, 2004.

[4] Strozek, Brooks. Efficient Architectures through Application Clustering and Achitectural Heterogeneity. *CASES*, 2006.

[5] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. *PACT*, Sep. 2006.

[6] N. Muralimanohar, R. Balasubramonian, N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. *MICRO*, 2007.

[7] V. Kathail, et al. PICO: Automatically Designing Custom Computers. *IEEE Computer*, 35(9):39-47, Sep. 2002.

[8] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block Ahead Branch Predictors. *ASPLOS*, Oct. 1996.

[9] M. D. Brown, J. Stark, and Y. N. Patt. Select-Free Instruction Scheduling Logic. *MICRO*, Dec. 2001.

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

[11] B. Lee, D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. ASPLOS, 2008.

Appendix

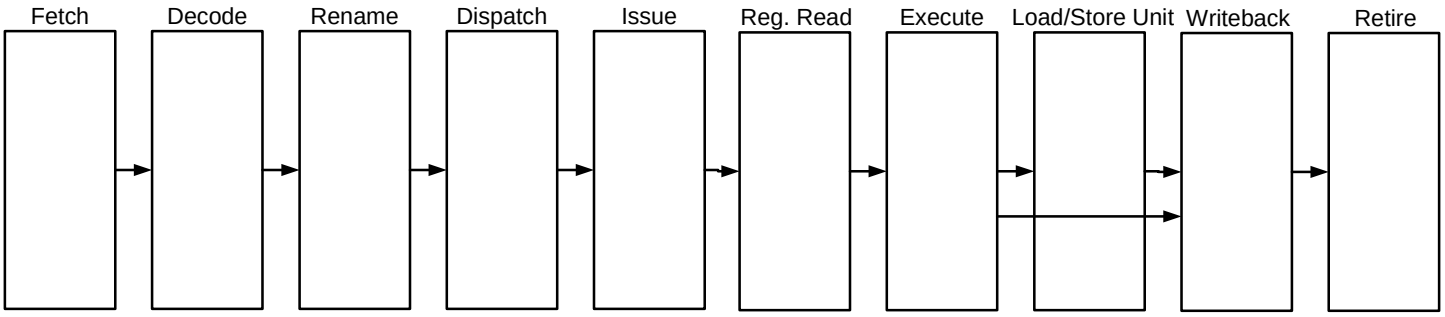
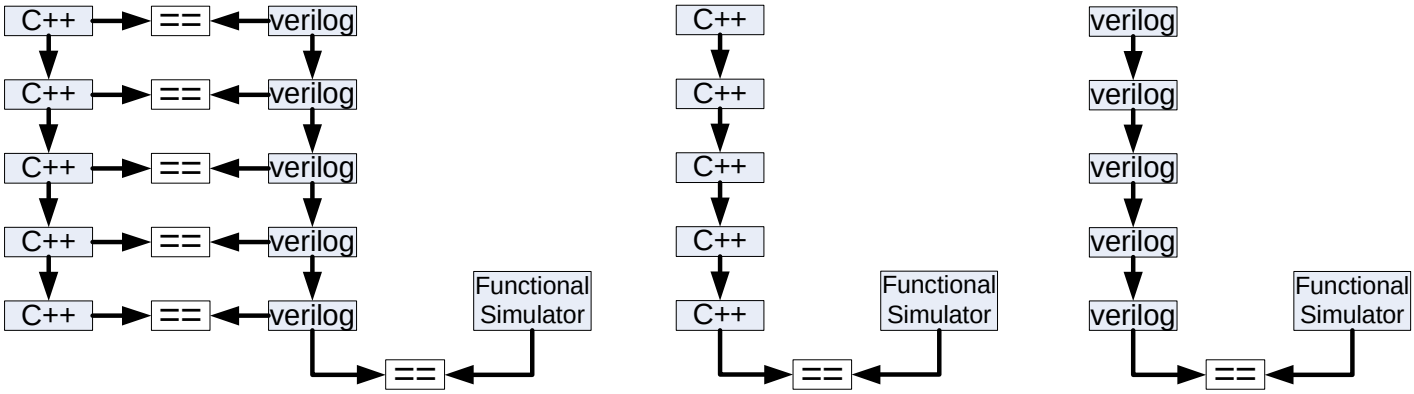


Figure 1. Canonical pipeline stages of a superscalar processor.



(a) Tightly integrated C++ & verilog. **(b) Standalone C++.** **(c) Standalone verilog.**

Figure 2. Flexible simulation options.

Table 1. Configuration of synthesizable- verilog model.

Stage	Description
Fetch	4-wide, 512-entry BTB, 128-entry bimodal branch predictor, 8-entry RAS, 16-instruction fetch buffer
Decode	4-wide, ISA = PISA (MIPS-like)
Rename	4-wide, 32-entry rename map table with 8 read and 4 write ports, 4 shadow map tables (checkpoints)
Dispatch	4-wide
Issue	4-wide issue, 32-entry issue queue
Register Read	4-wide, 128-entry physical register file with 8 read ports and 4 write ports
Execute	1 simple ALU, 1 complex ALU, 1 branch ALU, 1 AGEN + 1 port to load-store unit
Load-Store Unit	16-entry load queue, 16 entry store queue
Writeback	4-wide
Retire	4-wide, 128-entry active list with 4 read and 4 write ports, arch. map table with 4 read and 4 write ports

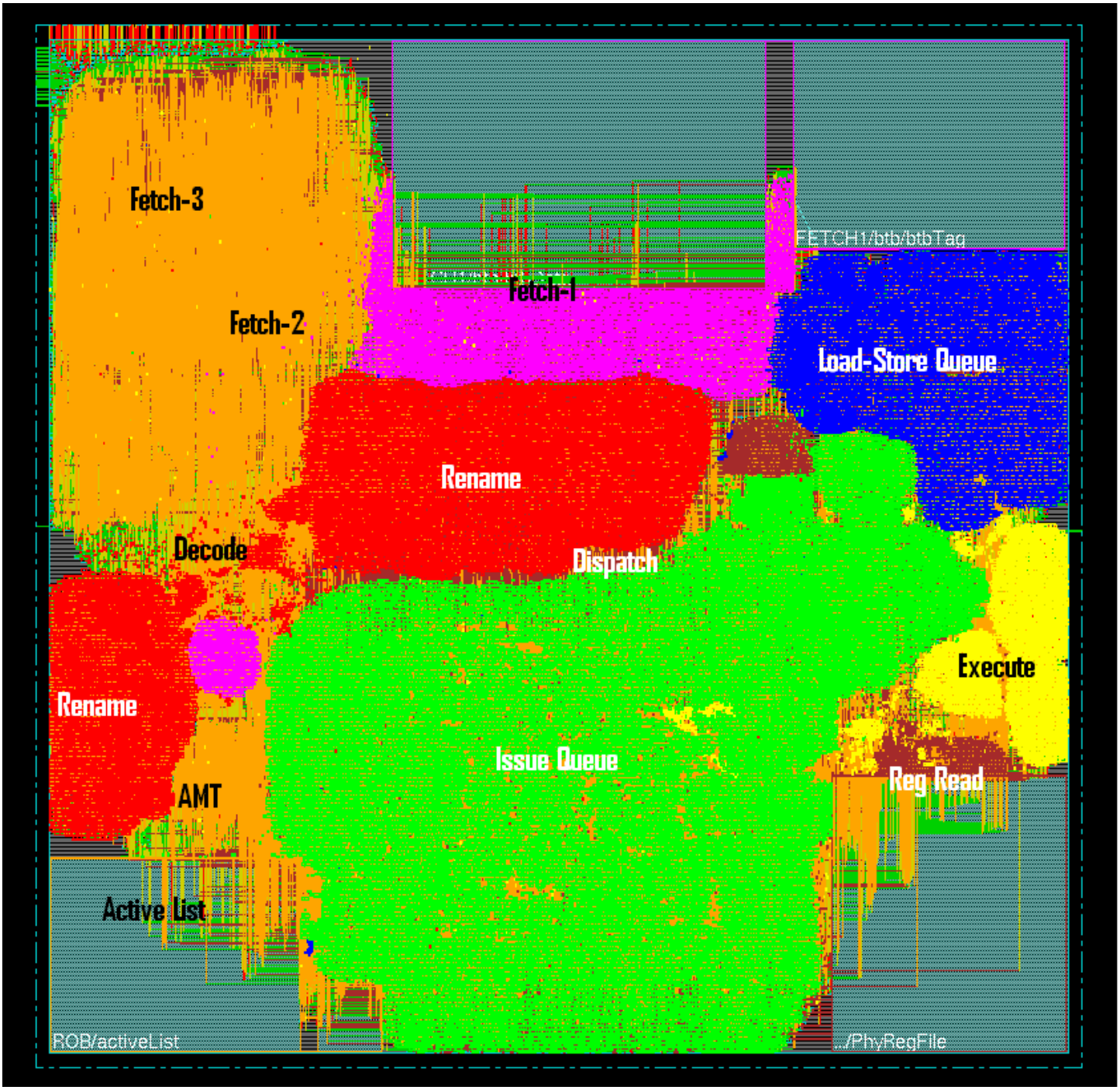


Figure 3. Placed-and-routed 4-way superscalar processor, excluding L1 I- and D-caches. The floorplan is cursory (couple days of effort). Area = 2.6 mm^2. Process = 45nm.

Technology library:

James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Mi-chael Bucher, Sunil Basavarajaiah, Julie Oh, Ravi Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," mse,pp.173-174, 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07), 2007.

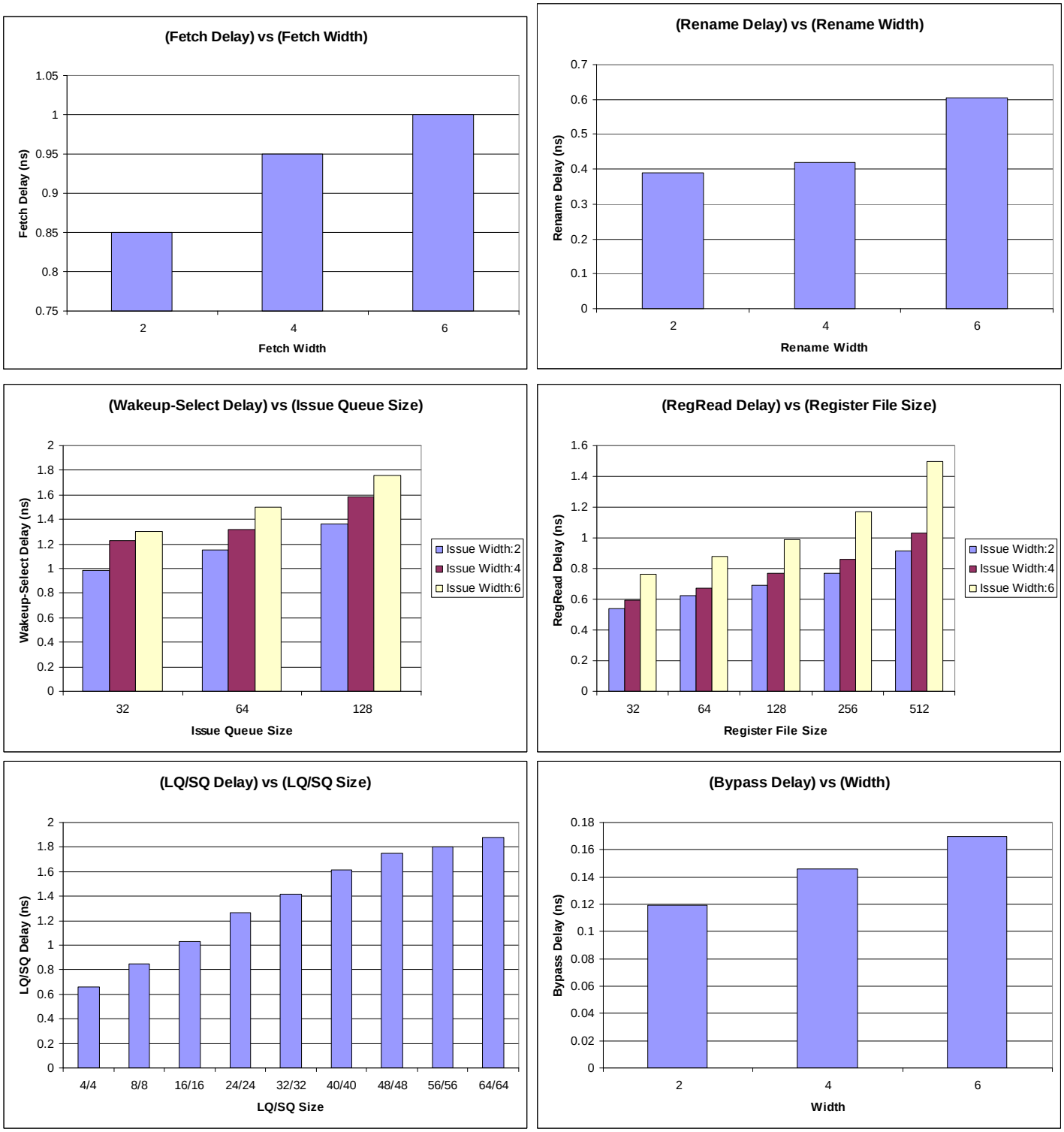


Figure 4. Delays of key canonical pipeline stages as a function of their complexity.

Table 2. IPC results for SPEC2000 100-million-instruction SimPoints, from cycle-accurate C++ model. I- and D-caches are perfect. IPC results are low due to known microarchitectural bottlenecks discussed below the table.

	gap	gcc	gzip	twolf	vortex	vpr
IPC	0.45	0.45	0.54	0.44	0.52	0.48

Once functional verification is ramped up, performance debugging will commence. Currently, there are several known bottlenecks. First, there is no load speculation, both in terms of speculative memory disambiguation (a load always stalls if there is a potential conflict with a prior unknown store address) and speculatively waking up load-dependent instructions. Both will eventually be supported with a load-slice replay unit. Second, the issue stage is currently sub-pipelined into separate wakeup and select cycles, such that single-cycle producers and their consumers do not execute in consecutive cycles. Back-to-back execution will eventually be supported with select-free scheduling [9]. Third, there is only one type of each function unit. Additional integer pipes and load-store unit ports are needed to capitalize on existing ILP.

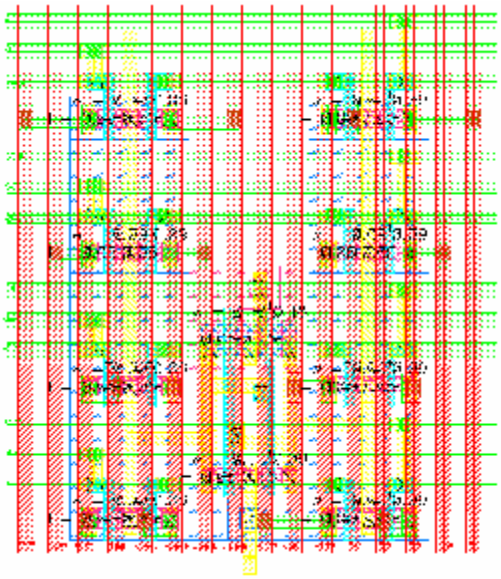
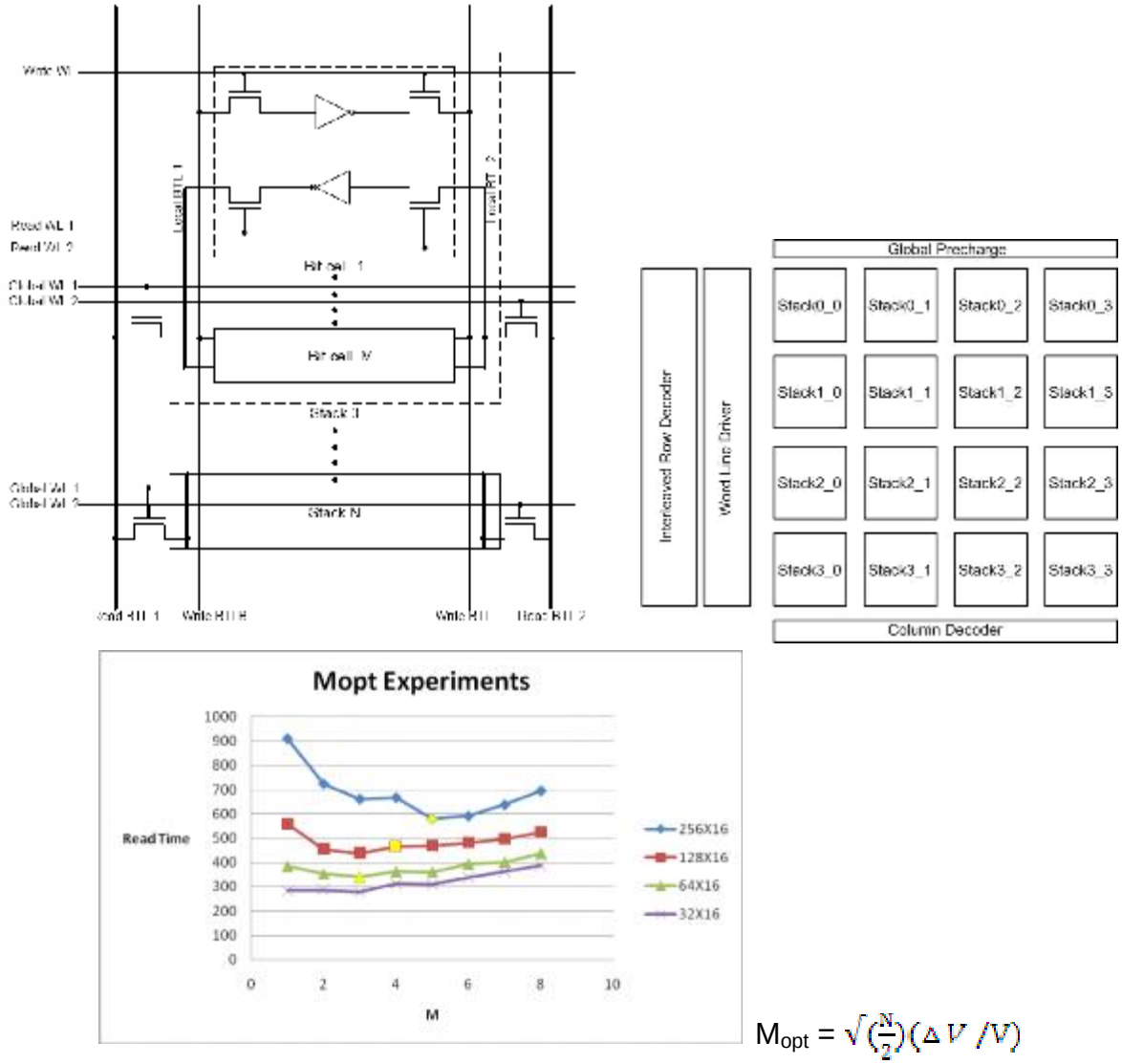


Figure 5. Layout for 8R-4W-ported bit-cell.



$$M_{opt} = \sqrt{(\frac{N}{2})(\Delta V / V)}$$

Figure 6. We implement a hierarchical global/local bitline approach, i.e., columns are divided into "stacks".

To model the global bitline, its intrinsic and coupling capacitances are estimated using the following formulas:

Intrinsic capacitance:

$$\frac{C_{11}}{\varepsilon_{ox}} = 1.12\frac{W}{H} + \left[0.75\left(\frac{W}{H}\right)^{0.11} + 0.68\left(\frac{T}{H}\right)^{0.49}\right] + \left[-0.039\left(\frac{W}{H}\right)^{0.092} + 0.83\left(\frac{T}{H}\right)^{0.028}\right]\left(1 - e^{-\frac{S}{H}}\right)$$

Coupling capacitance:

$$\frac{C_{12}}{\varepsilon_{ox}} = \frac{T}{S} + 1.31\left(\frac{T}{H}\right)^{0.073}\left(\frac{S}{H} + 1.38\right)^{-2.22} + 0.4\ln\left(1 + 5.46\frac{W}{S}\right)\left(\frac{S}{H} + 1.12\right)^{-0.81}$$

Here, W = width of the metal layer

H = height of the metal layer from the substrate

T = Thickness of the metal layer

S = Spacing between two metal layers

As shown in the figure below, for the estimation of the coupling capacitance, coupling due to the adjacent two global bit lines on each side is considered. For the inner most global bitline, in addition to the coupling due to the two global bitline on one side, coupling with the local bitline of the individual stack is also considered.

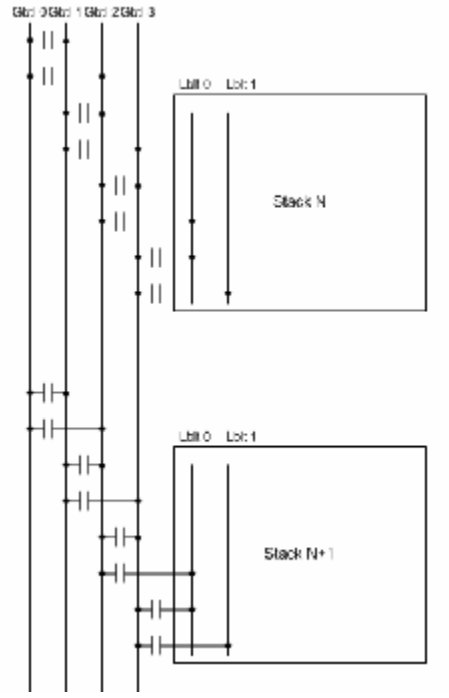


Figure 7. Modeling intrinsic and coupling capacitances of bitlines.