Heterogeneous Cores with Low-Overhead Fine-Grained Switching

Anna Huang
University of Michigan
Ann Arbor, MI
ahuangg@umich.edu

Bradley Schulz
University of Michigan
Ann Arbor, MI
byschulz@umich.edu

Finn Moore
University of Michigan
Ann Arbor, MI
fdm@umich.edu

John McCloskey
University of Michigan
Ann Arbor, MI
jomcc@umich.edu

Mustafa Miyaziwala
University of Michigan
Ann Arbor, MI
mmiyazi@umich.edu

***Abstract*—In this project we implemented a heterogeneous processor that supports fine-grained switching between two backends: one high-performing out-of-order backend and another low-power in-order backend. A shared always-on (AON) block provides shared caches and fetch logic as well as performance tracking logic that dynamically switches between backends for the optimal balance of power and performance. This allows the processor to switch backends up to every 1000 instructions with an overhead of under 40 cycles per switch and no need to warm up the shared branch predictor and caches after a switch. Implemented in TSMC 180nm technology, our final design is 9.24mm^2 and has a clock frequency of 80MHz.**

I. Introduction

Designing a processor that effectively balances the tradeoffs between power and performance is often difficult; making a processor that is low-power often comes at the expense of performance. Moreover, the need to emphasize low-power over high-performance, and vice-versa, often changes dynamically as a processor's workload evolves over time. The ideal processor lowers its power consumption as its workload becomes less demanding, but then retains the ability to utilize higher-performance hardware when the need arises.

A. Existing Designs

Power saving strategies vary between circuit-level and architecture-level optimizations. Circuit-level approaches such as clock gating, power gating, DVFS, body biasing, and more can be combined with larger architecture-level designs. One common architectural strategy to balance variable workload demands are including both low-power and high-performing cores on a single chip. These designs, such as ARM's big.LITTLE [1] architecture, performs a context switch between cores when the demands of the workload change. When higher performance is desired, state is transferred from the small core to the large core. When the workload decreases in intensity, the state of the machine can be transferred back to the small core to save power.

However, the issue with these designs is that transferring state between cores incurs a large overhead. All active memory blocks must be transferred to the new core over the chip's memory system, and the caches and branch predictor must warm up again on the new core once execution resumes.

B. Our Design

Our design, inspired by Lukefahr et al.'s work [3], minimizes the overhead of switching between cores through having a large and small core share their caches and fetch logic. This allows a context switch to be performed very quickly (in under 40 cycles), as only the contents of the register file must be transferred between backends. Moreover, the caches and branch predictor retain their state between switches, eliminating the need to warm them up again after switching backends.

II. Architecture

Our architecture is composed of three separate domains: an in-order backend, an out-of-order backend, and a shared always-on domain containing an instruction cache, instruction fetch, and a data cache. Each of the backends perform the same functionality, but with a different balance of power and performance. The in-order backend consumes one-third of the peak dynamic power of the out-of-order backend, but at the expense of drastically reduced performance.

The register online controller (ROC) is responsible for determining when to switch cores. This module extracts program execution statistics from the currently active backend and estimates performance on the idle backend. Using this, it chooses the backend that will achieve the optimal balance of power and performance.

We use the RISC-V ISA supporting all integer operations. This allows us compile and run programs from any language supported by our RISC-V C compiler. Our address space is 64KB, meaning we support 16-bit address. Memory blocks are 8 bytes each, so memory data buses are 64 bits wide. A high-level functional diagram is in Figure 1.

A. Design Goals

To simplify integration, we designed the interfaces between both backends and the always-on domain to be identical, with the exception of a few minor performance tracking signals used by the ROC to estimate performance. All interfaces between blocks are registered at their inputs and outputs, and we heavily leverage valid-ready handshakes to ensure no data loss between blocks.

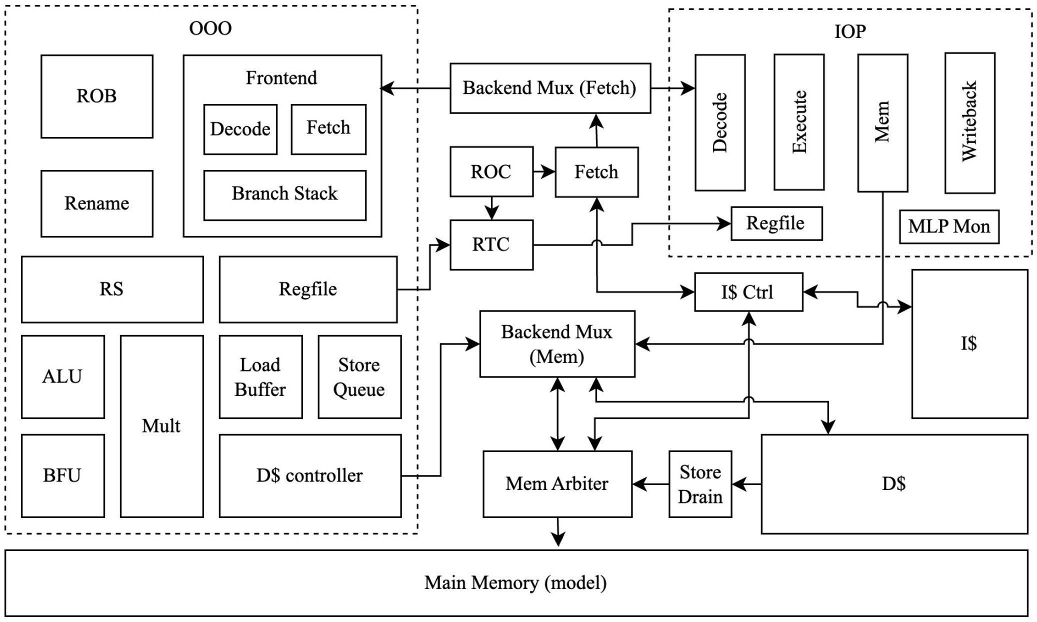


Fig. 1: Top-level diagram of the functional blocks in the heterogeneous core

B. *In Order Backend*

The in-order processor (IOP) is designed to be a low-power, simple processor designed using a standard 5-stage pipeline. Since fetch exists in the always-on domain, there are actually only four pipeline stages within this backend. However, under certain stalling conditions, instructions may take much longer to progress through the pipeline. To match the clock period to the rest of the design, the multiplier is pipelined into four stages, meaning that multiplication operations remain in the execute stage for 4 cycles. Loads and stores also take longer, especially if they miss in the data cache, as the data needs to be fetched from off-chip memory. Since the backend is in-order, these stalling conditions affect all subsequent instructions and can incur rather large performance hits. However, these performance hits allow for drastically minimized hardware (and therefore power) when compared with the out-of-order backend.

C. *Out of Order Backend*

The high-level design goal for our out-of-order (OOO) performance core was to maximize performance while remaining within a reasonable bound of complexity. The out-of-order backend was also designed to share the same interface as the in-order backend to simplify top-level integration. Because of this, our out-of-order backend does not support superscalar execution. However, we designed it to eliminate as many internal stalling conditions as possible recovering from branch mispredictions in one cycle as soon as the branch executes, pipelining our issue logic to support back-to-back execution of dependent instructions, allowing loads to issue out-of-order, internally forwarding data from stores to loads, and building a data cache controller that can handle 8 simultaneously outstanding memory transactions.

With no stalling conditions, an instruction takes 7 cycles to move through the OOO backend:

1) Fetch: register the input from AON fetch
2) Dispatch: rename source operands and allocate appropriate locations in reservation station (RS), re-order buffer (ROB), and store queue (if applicable)
3) Issue 1: Select up to two instructions to issue, and for instructions requiring one cycle to execute, arbitrate for the common data bus (CDB).
4) Issue 2: Read from the physical register file (PRF). For one-cycle execution operations, broadcast the instruction's tag on the CDB.
5) Execute: Perform arithmetic in one of 5 functional units: ALU, branch functional unit (BFU), multiplier (4-stages, fully pipelined), load buffer, and store queue.
6) Complete: Broadcast the result from execution on the CDB and write to the PRF.
7) Retire: Remove the instruction from the ROB. If the instruction is a store, signal it as ready to write to the data cache.

Although the latency of instructions is longer in the OOO backend than the IOP backend, the ability to execute back-to-back dependent instructions means that the steady-state throughput remains at least as high as the IOP backend.

The physical register file has 64 registers and supports four reads and one write per cycle. This allows the OOO backend to issue up to 2 instructions per cycle and complete one instruction per cycle. The ROB contains 32 entries to support up to 32 in-flight instructions, with up to 4 in-flight branches and 10 active stores. The reservation station contains 20 entries, allowing 20 instruction level parallelism (ILP) of up to 20.

D. *Always-On Domain*

The always-on (AON) domain contains the shared fetch logic, data cache, and multiplexor between both backends. It also contains the transfer control logic to perform the state transfers between the OOO and IOP backends.

Our design includes an on-chip 4KB data cache and 4KB instruction cache. The data cache is split between two banks, and each has an additional SRAM for control information (tags, LRU, valid, and dirty bits).

Our fetch logic is pipelined into two stages due to the cycle latency of instruction cache (I$) reads. The first stage intiates a read from the instruction cache and predicts the direction of branches using our gshare [2] branch predictor. The second cycle receives the data from the instruction cache, and, for branches and jumps, calculates the branch target. If the branch is predicted taken, the branch target is fed back into the I-cache to maintain a steady flow of instructions. In the case of an I-cache miss, the I-cache controller begins prefetching up to the next 4 subsequent cache lines while waiting for requested instruction to return.

A central multiplexor interfaces the AON logic with both backends. This mux manages both frontend and memory interfaces that feed instructions and grant memory requests only to the active backend.

Lastly, the memory infrastructure in the AON domain manages the data cache and interface with off-chip memory. The backend mux allows one backend to access the data cache, and in the event of a cache hit, the active backend is responsible for requesting the correct block from off-chip memory. The backend is responsible for fetching data from off-chip memory because the OOO backend supports multiple outstanding transactions, while the IOP backend supports only one. Writes to memory can only come from the store drain, where dirty data evicted from the data cache resides while waiting to write back to memory.

Among the three possible memory agents (backend, instruction cache, and the store drain), the memory arbiter selects one to access main memory with the following priority:

1) Store drain if it is full. This prevents the need for backpressuring evictions from the data cache
2) Backend data request
3) Instruction cache request
4) Store drain if not full

III. Backend Switching Architecture

The register online controller (ROC) estimates the performance of both backends and initiates a transfer register when the power-performance product moves above or below a statically defined threshold. Once a transfer is initiated, the register transfer controller (RTC) activates and begins copying the register values between the backends. While this transfer is occurring, the fetch module is disabled so that no new instructions enter either backend. Once all no more instructions remain within the currently active backend and all the register values have been transferred, the muxes switch to the new core and fetch resumes providing instructions.

A. *Register Transfer Controller*

The register file is the only stateful component between the two backends that needs to be transferred. The in-order backend has a small 2-read 1-write port register file, and the out-of-order backend has a larger 4-read 1-write register file. The transfer is accomplished in under 40 cycles.

To minimize the exposed latency of register switching, registers begin speculatively transferring between the two cores before the active backend is fully drained. When transferring from the in-order core and a writeback occurs to a register that has already been transferred, the register is re-transferred. When transferring from out-of-order, if a physical register is not complete, it is skipped and returned to after all registers are attempted.

The transfer from the out-of-order core is accomplished through a 3-stage pipeline. The first stage selects a register from the bit-vector and queries the RAT for physical register translation and readiness. The second stage reads the register from the physical register file and is stalled if the read port is occupied. The final stage writes to the in-order backend's register file.

The transfer from the in-order core is accomplished through a 2-stage pipeline. The first stage selects and reads a register from the in-order register file, stalling if the port is occupied. The second stage writes the register value to the out-of-order backend. The out-of-order RAT is also resetted such that physical register n maps to architectural register n.

B. *Reactive Online Controller*

1) Performance Estimator: To determine whether it is beneficial to switch to the other backend, the performance of the other backend needs to be estimated. This estimation is accomplished through a linear model using performance counters on the active backend. Eq. 1 defines the model which consists of a constant (a_0) and several input metrics (x_i) and corresponding coefficients (a_i).

$$y = a_0 + \sum a_i x_i \tag{1}$$

Gathering the appropriate metrics to predict the performance of the in-order pipeline is relatively straightforward. The in-order pipeline stalls for multiply and load instruction execution, cache misses, and branch mispredictions. Performance counters count each of these events.

Out-of-order backend's performance prediction is more challenging. The instruction level parallelism (ILP) and memory level parallelism (MLP) are important features of a program that impact the out-of-order backend's performance. To measure ILP from the in-order core's execution, we count the number of times we activate data forwarding paths between pipeline stages. To measure MLP, we maintain a small history buffer of the load addresses that have passed through the in-order backend and count the number of times an incoming load address matches an element in the history buffer.

For training the coefficients, we ran the two backends independently on a CNN program (Alexnet) with added functions to induce cache misses, multiply heavy periods, and arithmetic heavy periods. The models were trained using the metrics from one backend and the clock count from the other backend. Figure 2 shows the significance of each metric to the final performance prediction.

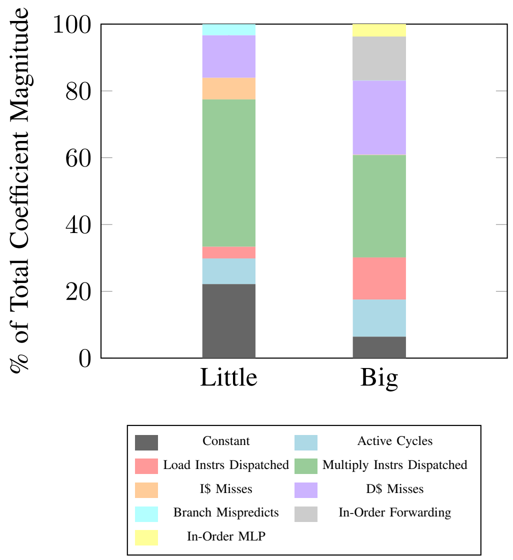


Fig. 2: Feature contributions for the out-of-order (OoO) and in-order (IoP) backend's performance prediction

2) Switching Decision: Every quantum (1000 instructions dispatched), ROC predicts the performance of the inactive backend using the linear regression model discussed earlier. If the difference between the performance of the in-order backend and out-of-order backend is smaller than a static threshold, then execution is switched to the in-order backend; otherwise, execution is switched onto the out-of-order backend. Execution starts on the out-of-order backend since it is able to warm up the caches faster. The PI controller presented by Lukefahr et al [3] was also implemented and tested. However, due to poorer accuracy of our performance predictor and the lack of large test benchmarks, the PI controller did not result in a performance benefit compared to a static threshold.

IV. Verification

To verify the design, a suite of programs was run on the processor and the final state of memory was compared with the correct memory output. The correct memory files were created by running the programs on a correct implementation of a RISC-V processor. This correct processor was a very simple 5-stage pipeline that incurs no hazards due to only allowing one instruction in the pipeline at a time. We simulated this processor with no memory hierarchy and instantaneous memory latency to ensure functional correctness.

Our test programs included both simple programs written in assemby and more realistic C programs that model real-world workloads. The C programs were compiled and tested on different optimization flags to ensure full functional correctness. Some examples include *alexnet*, a model of a convolutional neural network, *outer_product*, a program that performs a series of matrix outer products, and various sorting algorithms such as *quicksort, mergesort*, and *insertionsort*.

Both backends were tested on the entire suite in behavioral verilog, post-synthesis netlists, and post-APR before being integrated. After each backend was verified, the top-level was constructed by integrating the two backends with AON and tested on the entire suite pre-synth, post-synth, and post-APR. We tested the full system with each backend hard-wired to the centralized fetch, and then introduced RTC transfers during simulation to verify that a state transfer could be completed successfully.

V. Physical Design

The Verilog was designed to support varying degrees of synthesis, with the OOO backend heavily reusing shared modules defined hierarchically, and the majority of the AON block being composed of compiled memories. However, we found that the APR tools were able to handle each whole backend without the need for additional hierarchies, so we left our design with three main hierarchy levels: compiled memories, individual partitions (AON, OOO, IOP), and top-level integration.

Block	**Dimensions**			**Pins**		
	W (mm)	H (mm)	Area (mm^2)	In	Out	Total
AON	1.56	1.76	2.75	381	587	968
IOP	0.24	1.86	0.45	250	155	405
OOO	1.16	1.86	2.16	257	154	411
Top	3.61	2.56	9.24	72	84	172

TABLE I: Area and IO by partition

The area for each partition and the number of IOs is shown in Table I. The OOO backend is approximately 4 times the size of the low-power IOP backend, and the vast majority of the AON area comes from the caches.

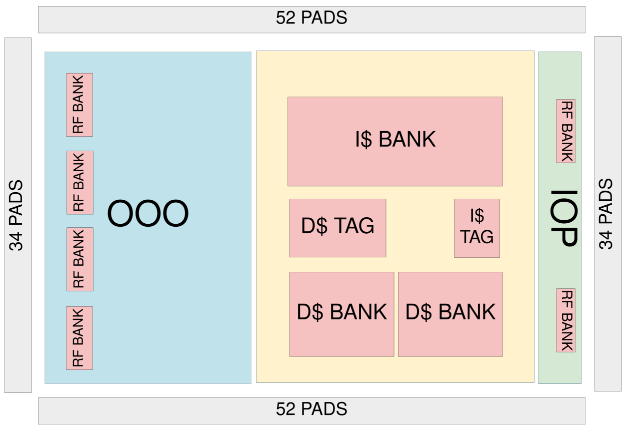
A. Interconnect

To simplify integration, we made the OOO and AON interfaces identical and aligned their pins with the AON pins. We matched the heights of all partitions and placed the AON block in the center of both backends since all backend IOs flow through the AON partition.

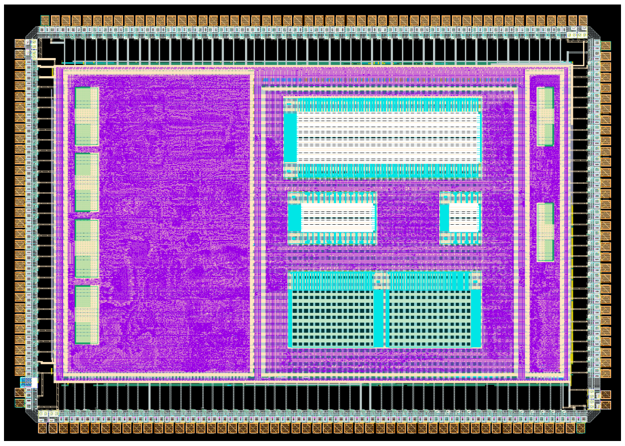
The OOO and IOP backends have the same interface with AON with the exception of a small number of control signals for the ROC that output statistics on program execution. A script we wrote extracts these pins from the synthesized netlists and aligns them between the AON block and respective backend to simplify top-level integration.

B. Placement

The full design floorplan is shown in Figure 3a. This floorplan includes the placement of the 11 compiled SRAMs and 172 chip pads. In addition to signal pads, 16 power pads (VDD, VSS, DVDD, DVSS on each side) and an electrostatic discharge detection pad were placed in the padring. In each partition, the SRAM macros were placed and rotated to efficiently enable signal routing. Signal routing was done using



(a) Top level floorplan



(b) Final chip layout with pads

Fig. 3: Top level floorplan and chip layout with three domains (AON, OOO, IOP) and pads

metal layers M1 to M4, reserving layers M5 and M6 for the power grid.

C. Power Grid

Each partition has its own power grid, with 4μm power rings, 2μm M5 vertical stripes, and 4μm M6 horizontal stripes. The power stripes over the AON partition had greater sparsity to minimize supply noise on the sensitive SRAM macros. As the TSMC 180nm process only offers 6 metals, the global power grid was generated with the same metal layers as the partition grids. To accomplish this, the metal stripes of each partition were aligned and directly overlapped by the global power grid.

VI. Results

A. Density

The top level core density is 90.34%, with core densities of 85.17%, 82.04%, and 53.91% for the OOO, IOP, and AON partitions respectively.

B. Timing

Our design meets post APR timing with a clock frequency of 80MHz. The setup and hold margins on all nets are 215ps and 77ps respectively. The clock uncertianty used for these results is 200ps.

C. Power

Using PrimeTime, we were able to obtain power estimates on our individual modules and overall top level design. To account for the fluctuations between different programs, we ran our entire test suite through each individual backend to get an accurate estimate. The average total power for IOP and OOO came out to 31.497mW and 127.625mW respectively. The leakage of each was 1.76μW and 8.08μW.

Unfortunately, due to the large size of the waveform files generated, we were not able to run the entire test suite through our top level design. Instead, we chose two programs sitting at the low and high ends of power draw based off of our data from the individual backends. Since we designed our top level hierarchically, we were able to see the breakdown between each module. For both programs, the AON module had a total power of 110mW and leakage of 26μW.

By combining the numbers from the three modules, we are able to estimate the total power of our top level design. For the test program Alexnet, IOP is used 70/208 quantum's, roughly 20% of the time. With this, we can use AON+IOP*0.2+OOO*0.8 to calculate a total power of 218mW. Compared to the estimate of OOO calculated via AON-ROC-RTC+OOO = 110-0.866-1.17+127.625 = 235.589mW, our design improves power consumption by 7%.

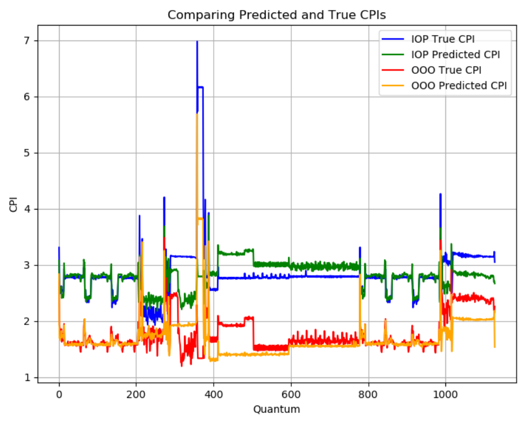
Due to the negligible amount of leakage in each individual backend, we decided that power gating the unused one would not be worthwhile. Additionally, the backends should inherently be clock-gated when unused because no instructions are flowing in, meaning none of the registers should be switching.

D. ROC CPI Predictions

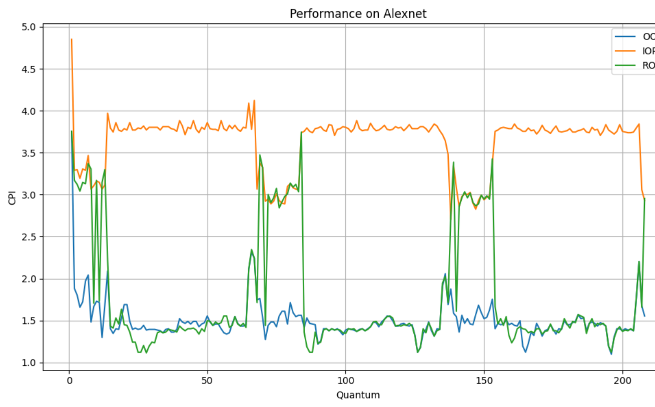
To show the accuracy of the predictions, figure 4a compares the predicted performance and the true performance of the two backends across all out test programs. The predictions are reasonably close to the true performances except for the spikes around quanta 400 and 1000. These are likely due to the test programs exhibiting largely different behavior than the programs the model was trained on. We attempted training the model on the data from all programs, however, this lead to poorer performance overall. We suspect this is due to some of our test programs being poor models of real-world program behavior.

E. ROC Switching

Figure 4b demonstrates the ROC switching to the in-order core when the performance difference between the in-order and out-of-order backend lessens throughout Alexnet's execution.



(a) Comparison of true and predicted CPI for IOP and OOO architectures.



(b) ROC appropriately switching between the backends throughout Alexnet

Fig. 4: ROC CPI estimation accuracy and workload performance

F. Final Design Status

We were able to get each individual block (OOO, IOP, AON) in our design DRC (besides metal and poly density) and LVS clean, but were not able to run DRC and LVS on the final top-level integration due to limited time. The top level without pads was fully placed and routed with a global power grid, resulting in no Innovus violations, and functionally verified. Due to a technology specific issue with the pad filler commands, the integration of all 172 pads had to be done manually in Virtuoso. This resulted in the pad ring being designed, power routed, and aligned with the top level pins, but signal routing was not completed by the design freeze deadline.

We were able to make some minor Verilog changes late in the project to simplify the integration process, such as adding input/output flops and adjusting the ports between blocks, but there were still a couple final APR bugs we were not able to resolve.

G. Reflection

About halfway through the project, we realized that the Verilog design was much more intensive than we had originally anticipated. The architectural adaptions required by memory compiler restrictions and the need to fully verify two individual backends proved very challenging and time-consuming. It took significantly longer to achieve fully correct behavioral Verilog than we had originally planned for. Our final design had over 20,000 lines of Verilog, and although we began with some existing code, most of this was written during the semester.

Even though we worked on developing our synthesis and APR flows in parallel with Verilog development, we still found ourselves time-limited on the final APR flows towards the end of the project. We are very satisfied with the architecture and performance of our design, but we believe we stretched ourselves too thin throughout the project. To finish every component, we were only able to allocate a single person to each backend, and one person to physical design flow development for the first half of the project. Combined with the fact that 4 of our 5 members are instructors for other courses, time was a strong limiting factor in our overall success.

VII. Conclusion

Nonetheless, we believe that our design provides a valuable proof-of-concept of the composite cores approach to heterogeneous processors. We were able to achieve fine-grained switching between backends and demonstrate the viability of this design approach. To our knowledge, the original Composite Cores paper [3] was only tested in behavioral simulation, meaning our work is the first to actually bring this design to a physical chip. Although this approach does not support simultaneously operating both the low-power and performance cores, it does allow for fine-grain context switching at a resolution not possible with other architectures. Moreover, the overhead of adding the small in-order core is quite small, as it only adds 0.45 mm^2 to the processor's area, which, when including the need for the ROC and RTC blocks, adds only a 15% increase in the top level area (excluding pads). We believe that achieving the power reduction possible with the in-order core while only adding minimal percentage area overhead is a valuable result.

References

[1] ARM, big.LITTLE Processing technologies, https://www.arm.com/technologies/big-little
[2] Combining Branch Predictors, S. McFarling, WRL Technical Note TN-36, June 1993
[3] A. Lukefahr et al., "Composite Cores: Pushing Heterogeneity Into a Core," 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, BC, Canada, 2012, pp. 317-328, doi: 10.1109/MICRO.2012.37. keywords: heterogeneous architecture;core microarchitecure;split pipelines;reactive controller,