BROWN

Low Power VLSI System Design

Lecture 13 & 14: Low Power Multicore and Approximation for Low Power

Prof. R. Iris Bahar

EN2912

October 23 & 25, 2017

BROWN

The HW/SW Interface Seminar Series

Jointly sponsored by Engineering and Computer Science

"Hardware-Software Co-design for Approximate Computing"

Natalie Enright Jerger
Department of Electrical and Computer Engineering
University of Toronto

October 23, 2017
Noon-1pm
Barus & Holley 190

Lunch will be provided

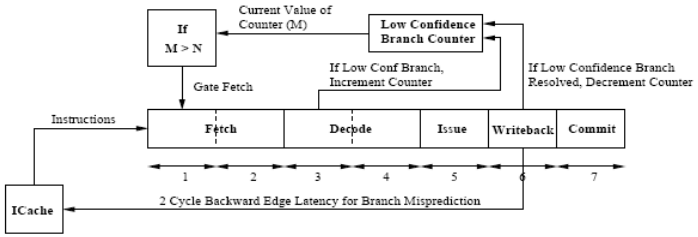
BROWN

Reducing Mis-speculated Instructions to Save Power

- Executing wrong path instruction is a waste of power
  - Does nothing to improve effective IPC either
- ***IDEA:*** if branch prediction becomes too speculative, don't bother continuing to fetch instruction past branches
  - Fetch unit stops reading new instructions from the cache
  - Instruction window does not take new instructions
  - Instruction execution rate may slow, but only until predicted branches have been resolved
- How do we know if an instruction flow has become "too speculative"?
  - What do we need to monitor?

BROWN

Pipeline Gating

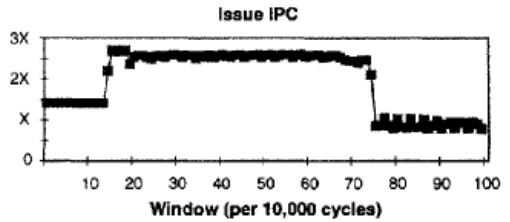


- Low-confidence branch counter records # of unresolved branches that reported as low-confidence.
- The processor ceases instruction fetch if there are more than *N* unresolved low-confident branches in the pipeline.

Do We Need Wide Issue Machines?

- Many programs never achieve IPC values close to the maximum issue of the machine
  - Branch Misprediction
  - Dependency Chains
  - Cache Misses
- Overall IPC is not indicative of superscalar needs of a program

Varying Program Needs



- This program shows a nearly 3X difference in IPC across successive snapshots of the program execution

Partitioned Cluster Architecture

- Instructions share the fetch, decode and renaming units
- There is a steering logic that sends instructions to different clusters based on scheduling algorithm
- There is a bypass logic between the two clusters.

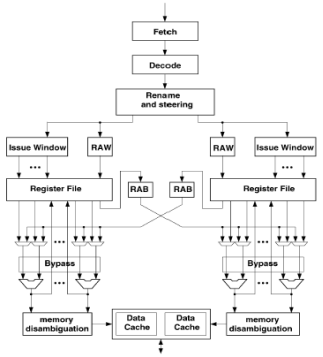
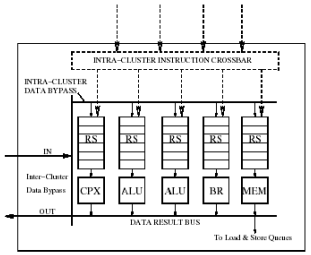


Fig. 4. Overview of the proposed multicluster architecture.

Clusters

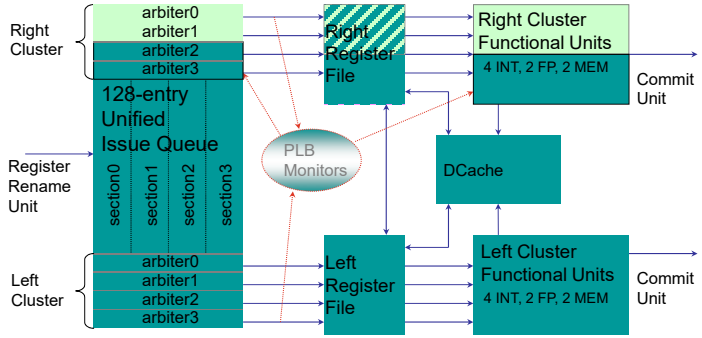


- Each cluster has 5 reservation stations (RS) feeding 8 special-purpose functional units.
- The RS hold 8 instr. and permit out-of-order instruction selection.
- Small clusters reduce complexity of wake-up and instr. select logic
- Intra-cluster communication is done in the same cycle as instruction dispatch.
- Forward data to other clusters takes at least 2 cycles.

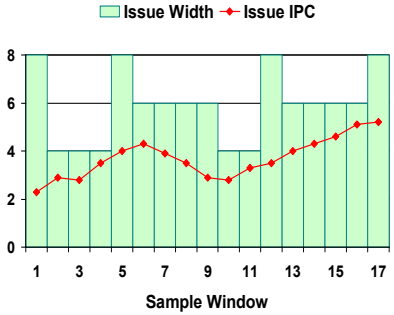
The Pipeline Balancing Approach

- Monitor the varying issue requirements of each program
  - Overall issue rate
  - Floating point issue rate
  - History of past behavior
- **IDEA:** Tune processor issue and execution resources according to the needs of the program
- Goal: Reduce power, retain performance

Our 8-Wide Issue Processor Model



Triggering PLB by Tracking Issue IPC



- Switch to 4-, 6-, or 8-wide issue depending on issue IPC of previous sampling window

Instruction-Level Parallelism

- When executing a single program, how many *independent* operations can be performed in parallel
- How have we taken advantage of ILP so far?
  - Pipelining
    - overlap different stages from different instructions
    - limited by divisibility of an instruction and ILP
  - Superscalar
    - overlap processing of different instructions in all stages
    - limited by ILP

Limits to Superscalars

- Window/register size, branch prediction, and memory access all influence the IPC of an application.
- There is a limit to the amount of parallelism that can be extracted from a single processor.
- We can try to improve IPC further with better compilers, value prediction, memory dependence prediction, multi-path execution, etc., but all these increase the complexity of the processor greatly.
- *What about power dissipation?*

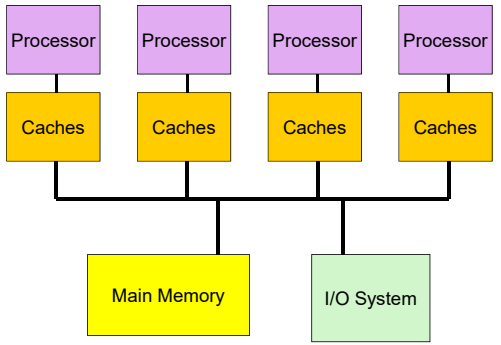
Making Multicore Power Efficient

- For workloads having data and task parallelism that can be accelerated by multithreading and multiprocessing, there is a nearly linear speedup with increasing cores
- Use a shared cache
  - Separate dedicated L2 caches are too power hungry
- Use low-clock speed processors
- Use Heterogeneous processors on a single die
  - High-performance processor used only when needed
- Use Demand-Based Switching (DBS)
  - Lower frequency and voltage during periods of low computing demand

Asymmetric Multiprocessing

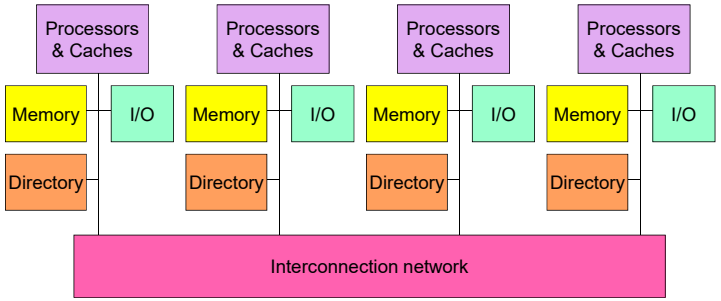
- ***IDEA***: have each core expend a varying amount of Energy per Instruction (EPI) by varying individual processor voltage and frequency based on the available thread-level parallelism
- Use *EPI Throttling*
  - The more cores in use the lower the frequency/voltage of each.
  - parallel phases of code are dynamically assigned to low EPI cores
  - sequential phases are assigned to high-EPI cores
- Can afford to run at higher clock for sequential phases since power is saved during parallel phases.

Centralized Shared Memory (SMP)



BROWN

Distributed Memory Multiprocessors



BROWN

The MESI Protocol

- To provide cache consistency on an SMP, data cache often use the ***MESI*** protocol
  - 2 status bits associated per tag to indicate 1 or 4 states:
- **Modified:** line has been modified and is available only to this cache. Main memory has stale data.
- **Exclusive:** line is the same as in main memory, but is not present in any other cache.
- **Shared:** line is the same as in main memory and may be present in other caches.
- **Invalid:** line does not contain valid data

BROWN

Performance Improvements

- What determines performance on a multiprocessor?
  - What fraction of the program is parallelizable?
  - How does memory hierarchy performance change?
- New form of cache miss: *coherence miss*
  - such a miss would not have happened if another processor did not write to the same cache line
- *False coherence miss:* the second processor writes to a different word in the same cache line
  - this miss would not have happened if the line size equaled one word

BROWN

Simplifying Assumptions

- All transactions on a read or write are atomic
  - on a write miss, the miss is sent on the bus, a block is fetched from memory/remote cache, and the block is marked exclusive
- Potential problem if the actions are non-atomic: P1 sends a write miss on the bus, P2 sends a write miss on the bus:
  - since the block is still invalid in P1, P2 does not realize that it should write after receiving the block from P1
  - instead, it receives the block from memory
- Fix by keeping track of more state:
  - Don't acquire the block unless all outstanding transactions have completed

BROWN

Bank Account Example

- Alice and Bob open a shared bank account
- Their initial balance is $0
- Each deposits $100
- We would expect that the balance now be $200

Balance = $0

Alice:	Bob:
Read (Balance)	Read (Balance)
Balance += $100	Balance += $100
Write (Balance)	Write (Balance)

BROWN

Bank Account Example

Balance = $0

Alice:	Bob:
Read (Balance)	
Balance += $100	
	Read (Balance)
	Balance += $100
Write (Balance)	
	Write (Balance)

Balance = $100

BROWN

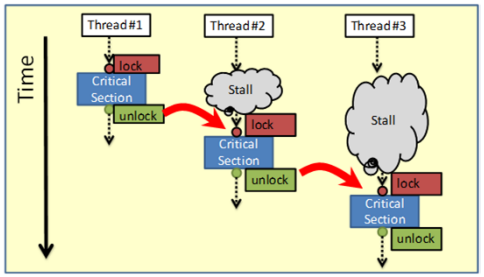
Synchronization

- Enable atomic execution of a ***critical section***
- Atomic execution:
  - All sub-operations are performed as one unit without interference from other operations
  - Either all operations or none are performed
- A **lock** surrounding the data/code ensures that only one program can be in a critical section at a time
- Rely on hardware supplied primitives to implement locks
- Lock == 1 → lock is taken
  Lock == 0 → lock is free

BROWN

Multiprocessor Synchronization

- Focus: Shared Memory Model (widely accepted)
- Common Approach: Lock-based Synchronization
  - **Idea:** mutual-exclusive access to shared data (i.e., critical sections)

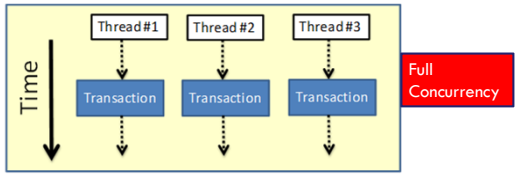


Multicore Synchronization vs. Parallelism

- *Remember why we turned to multicore rather than wider-issue, superscalar single cores?*
- But to make multicore worthwhile we need:
  - High Levels of Parallelism among the cores.
- Locks are basically serializing execution among the cores.
  - Hampers performance
  - Wastes power (spinning on locks)
- Can we make synchronization more efficient?

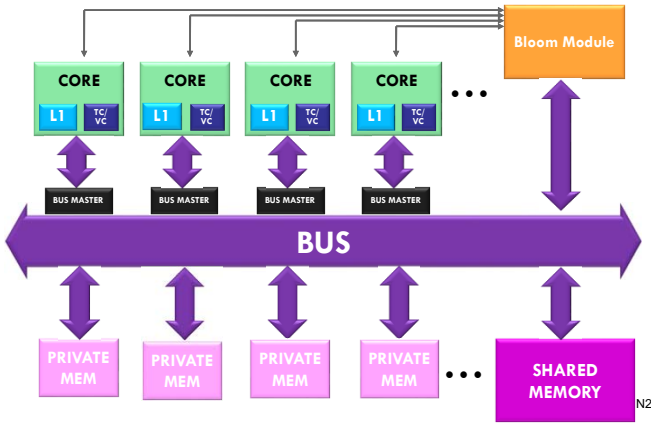
Transactional Programming

- Key Features:
  - Optimistic (i.e., concurrent) execution of critical sections
  - Threads run in isolation
  - Atomicity (roll-back in case of conflict)



- Remember: locks are *conservative* and assume threads will conflict, even if they are modifying different pieces of data within the same memory structure

Transactional Memory Scheme



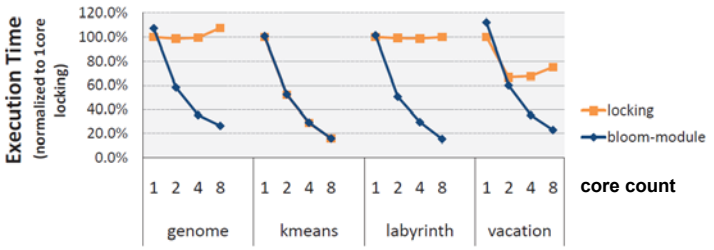
Managing Memory Contention

- The Bloom Module keeps track of all reads/writes to different cores
- If more than one transaction writes to the same data address, this is a *synchronization conflict*
  - At least one transaction will need to ABORT and RETRY
- On an abort
  - Stop execution
  - Restore state of processor core to beginning of critical section
  - Wait
  - Retry execution
- *What does this imply for performance and power?*

BROWN

Experimental Results

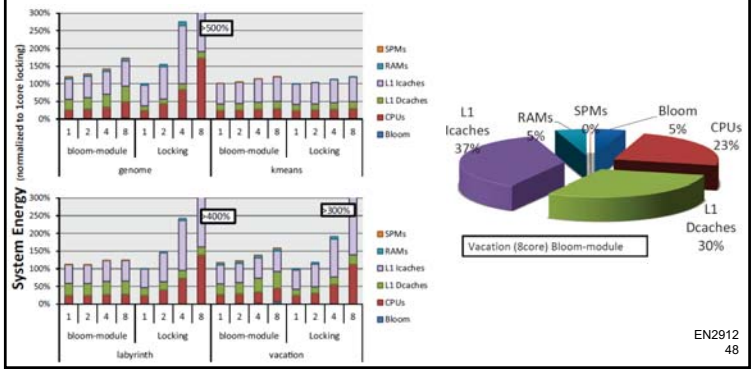
- STAMP benchmarks
  - Performance comparison with locks



BROWN

Experimental Results

- STAMP benchmarks
  - energy consumption comparison with locks



BROWN

Making Multicore Power Efficient

- For workloads having data and task parallelism that can be accelerated by multithreading and multiprocessing, there is a nearly linear speedup with increasing cores
- Use a shared cache
- Use low-clock speed processors
- Use Heterogeneous processors on a single die
  - High-performance processor used only when needed
  - Specialized HW can be used to perform some computations
- Use Demand-Based Switching (DBS)
  - Lower frequency and voltage during periods of low computing demand

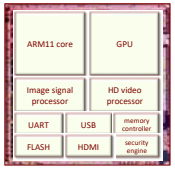
BROWN

Asymmetric Multiprocessing

- ***IDEA***: have each core expend a varying amount of Energy per Instruction (EPI) by varying individual processor voltage and frequency based on the available thread-level parallelism
- Use *EPI Throttling*
  - The more cores in use the lower the frequency/voltage of each.
  - parallel phases of code are dynamically assigned to low EPI cores
  - sequential phases are assigned to high-EPI cores
- Can afford to run at higher clock for sequential phases since power is saved during parallel phases.
- Some core can run at lower voltage, but there is a limit to voltage scaling before timing and computation errors are introduced.

Heterogeneous Processing using Custom Circuits

- Custom circuits (accelerators) can lead to the best performance and energy-efficient computing implementations



→ deployed as IP blocks in SoCs
→ used as custom instructions in CPUs
→ deployed in FPGAs
→ leads to heterogeneous computing
→ will be prevalent in dark silicon scenarios

Does All Computing Need to be Accurate?

- Some applications have inherent error resilience
  - Signal processing
  - Computer vision
  - Machine learning
  - Big data
- Used in diverse applications
  - video games
  - Embedded cognitive systems
  - Robotics
  - Unmanned autonomous vehicles

Approximation via Voltage Scaling

Goal: Trade-off quality of solution to reduce power dissipation

- Dynamic voltage scaling beyond the specifications
  - Major power savings occur because dynamic power is reduced by a factor of Vdd^2
  - Only Vdd is scaled down, not frequency so performance remains the same
  - May introduce intermittent timing errors (or even catastrophic errors)
  - May require error detection and (optional) correction circuitry
  - Need to determine if errors are critical or not
  - May need to (probabilistically) characterize magnitude of error

Approximation via Custom Circuitry

Goal: Trade-off quality of solution to reduce power dissipation

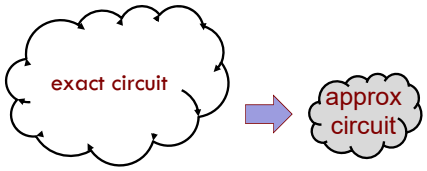
- Approximate circuit design by construction
  - Error can be (deterministically) controlled
  - Simpler (approximate) design can deduce design area → save power
  - Simpler design can mean faster clock time
  - Less flexibility: circuit can only produce approximate result.

How can we use Approximate Circuits to our Advantage?

- Low power / low area
- High throughput
- Dynamic power management
- Concurrent error detection and correction

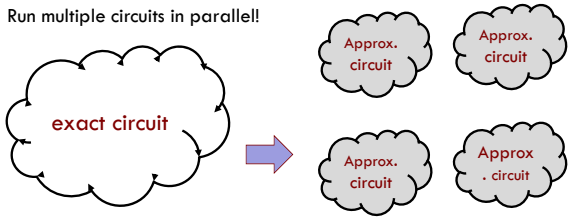
Approximate Circuit Use #1

- Approx. Circuit is less complex
- Low power / low area



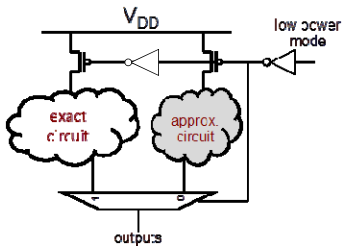
Approximate Circuit Use #2

- Small, less complex approximate version can be replicated multiple times
  - Similar to the multicore idea
- High Throughput
  - Run multiple circuits in parallel!



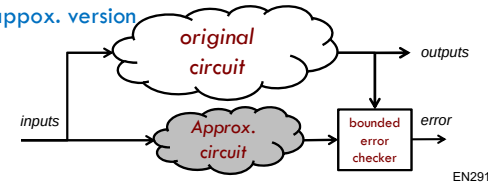
Approximate Circuit Use #3

- Sometimes we may need the extra accuracy
- Dynamically switch between the two modes as needed
- Save power when using the approximate version



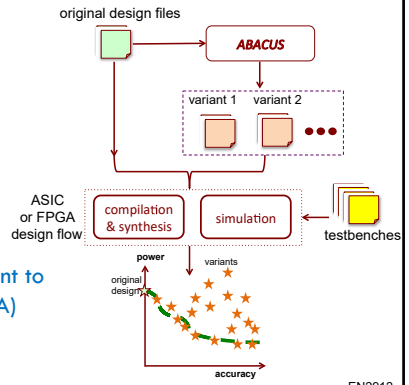
Approximate Circuit Use #4

- Sometimes there can be a "single event upset" (SEU) that causes a circuit to temporarily function incorrectly
- A typical approach to error resilience is to duplicate the circuit or use Triple Modular Redundancy
- Duplicating a full version of a circuit can be very costly in terms of area and power
- Instead, use an appox. version

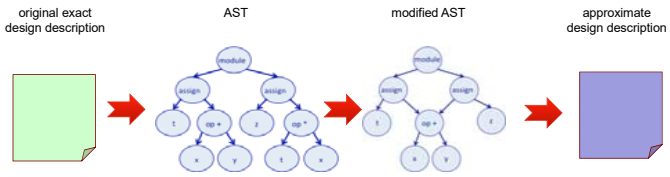


ABACUS: Automated Behavioral Approximate Circuit Synthesis

- Automated high-level approximate circuit synthesis
- Enables large-scale approximations
- Error is controllable
- No prior application-specific knowledge is required
- Approx. synthesis is transparent to the design flow (ASIC or FPGA)



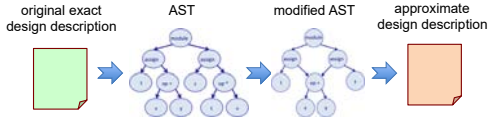
How does ABACUS work?



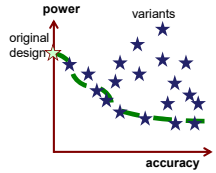
ABACUS overall approach:

1. Capture input design HDL in the form of Abstract Syntax Tree (AST).
2. Analyze the AST and modify it to generate approximate variant approximate ASTs
3. Convert the AST to HDL and push through the design for evaluation

Two Key Questions:



1. What kind of transformations can be applied to the ASTs?

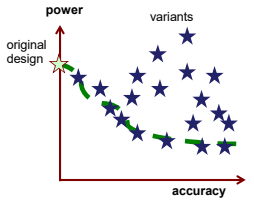


2. How do we avoid the exponential increase in design search space resulting from these transformations?

1. Approximate Hardware transformations

1. Data type simplifications
   e.g.: Truncate some least significant bits and shift results
2. Operation transformations
   e.g.: replace an adder with an approximate adder
3. Arithmetic expression transformations
   e.g: $w_i x_i + w_j x_j \rightarrow w_i(x_i + x_j)$
4. Variable to constant substitutions
   e.g: analyze signals and substitute by a constant if standard dev. is low
5. Loop transformations
   e.g.: unroll loop → apply 1-4 transformations; substitute result of one iteration from previous iterations

2. Stochastic Greedy Design Space Exploration



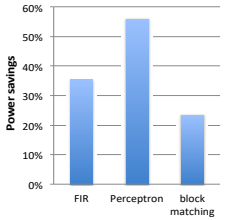
0. Let current design be original design.

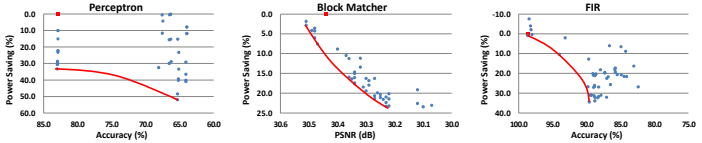
Repeat for multiple generations:

1. Assign probabilities for various transformations.
2. Apply transformations randomly at applicable locations on current design.
3. Evaluate designs for accuracies.
4. Synthesize designs that pass accuracy threshold.
5. Rank the designs based on a combined objective of power and accuracy.
6. Retain the best designs as the current designs for the next generation.

Experimental results

- Evaluated on three representative designs using standard 45 nm ASIC flow (Synopsys Design Compiler and MentorGraphics ModelSim).





Homework #3