MCD: A GALS Processor Microarchitecture

Dave Albonesi

in collaboration with

Greg Semeraro

Grigoris Magklis

Rajeev Balasubramonian

Steve Dropsho

Sandhya Dwarkadas

Michael Scott

Outline

- Motivation for MCD
- MCD microarchitecture
- Hiding synchronization delays
- Fine-grain dynamic voltage and frequency scaling
  - Off line algorithm
  - Online algorithm
  - Profiling
- Potential performance gains with MCD
- Future research

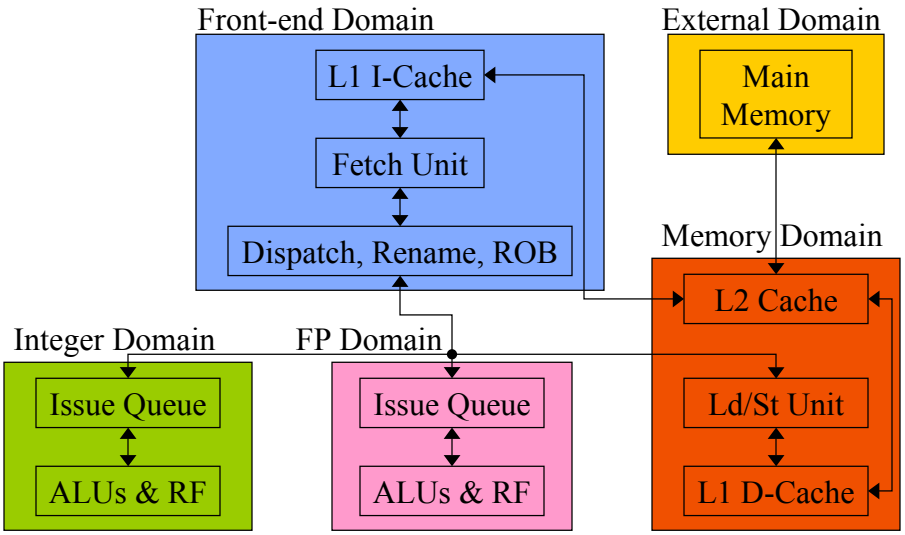
Motivation for MCD

- Increasing challenges of fully synchronous processors
- Companies have a large investment in synchronous design
- Designers know how to handle synchronizing signals between clock domains
- Gradual elimination of global signals creating more autonomous units
  - Example: Replay Traps instead of pipeline holds
- Single microprocessor-wide frequency constrains the IPC/frequency tradeoffs that can be made in different units
  - E.g., floating point design decisions linked to front-end decisions

Motivation for MCD

- Multiple on-chip voltages today, progress in on-chip voltage conversion
- Global Dynamic Voltage Scaling (DVS) has limited applicability
- Application phases may be bottlenecked by a subset of the major functions (fetch/dispatch, integer, floating point, load/store) of a general-purpose processor, but still all run at full speed in a synchronous processor

MCD at a high level



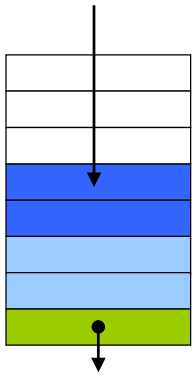
MCD microarchitecture details

- **Front-end, integer, floating point, memory clock domains**
  - **Major queues (issue queues, load/store queue, ROB) already in place as buffers that can be used as synchronization points**
  - **Synchronization can mostly be hidden if queues are partially full**
  - **Much autonomy between these major functions**
- **L1 Dcache separated from integer and floating point**
  - **Allows memory to be separately optimized**
  - **Performance not adversely effected**
- **L2 cache placed in the memory domain**
  - **No L1-L2 synchronization penalty for loads/stores**
  - **Applications with large L1 Icache miss rates may be impacted**

Synchronization via queues

- **Two types: FIFO and issue queue**
- **Key insight: synchronization cost can be hidden if instruction would have waited in the queue anyways**

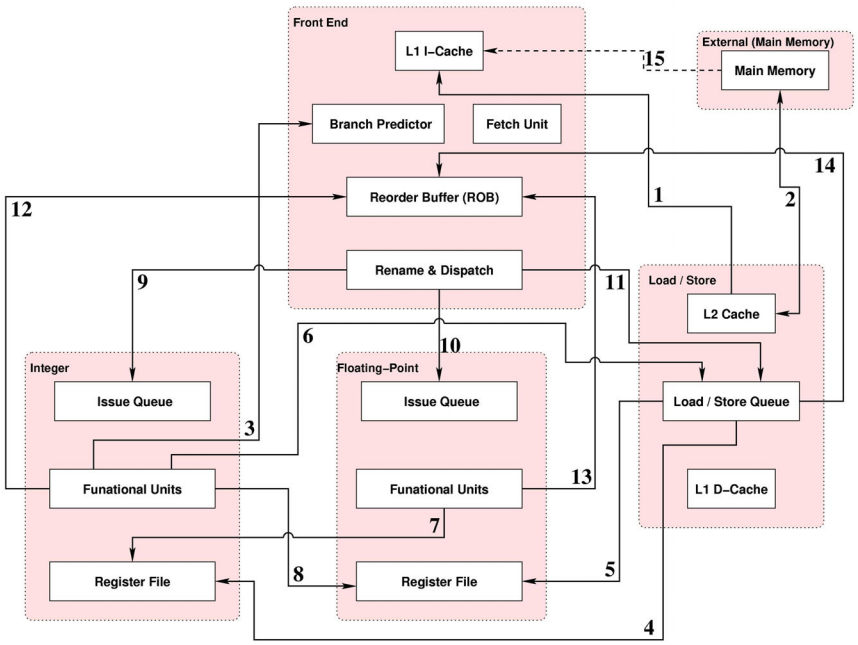
FIFO queue structure



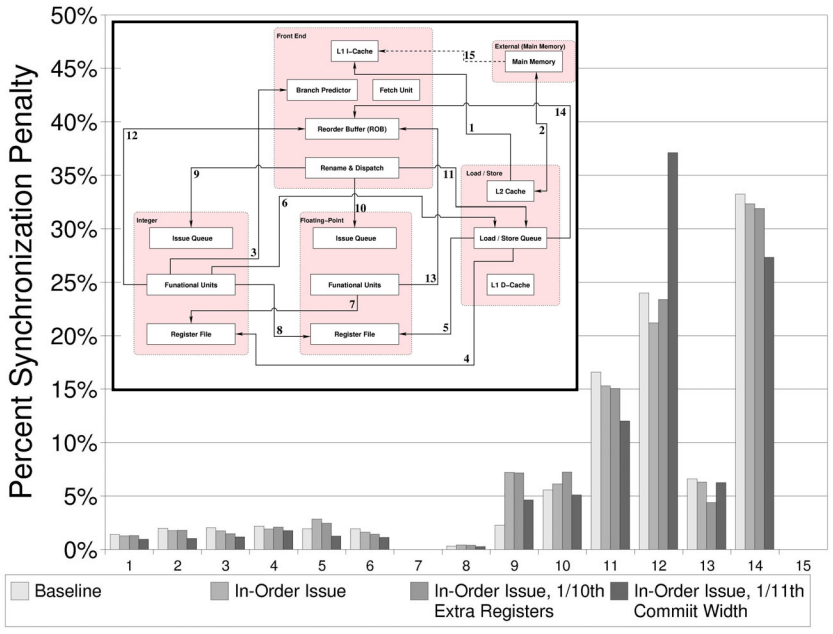
Hiding the synchronization delay

- **Instructions that end up waiting in queues after synchronization**
- **Group of instructions crossing a domain incur a single delay**
- **Out-of-order execution**

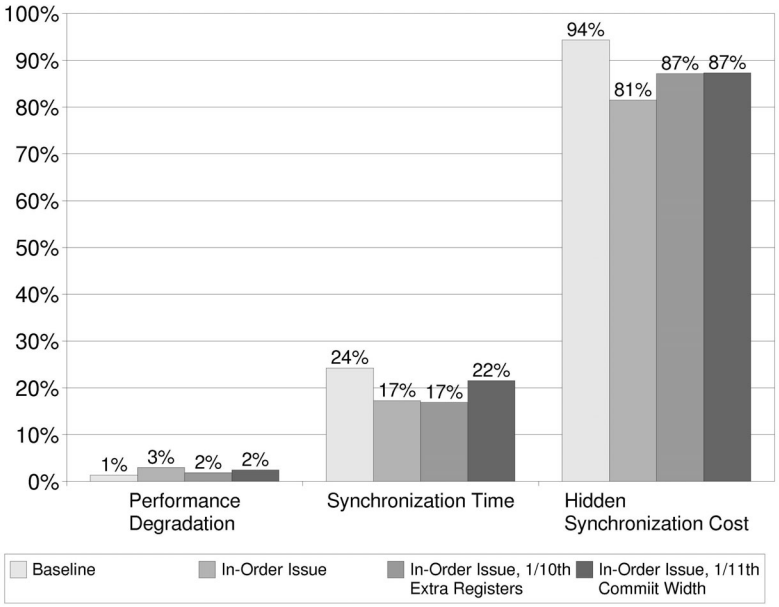
Synchronization points



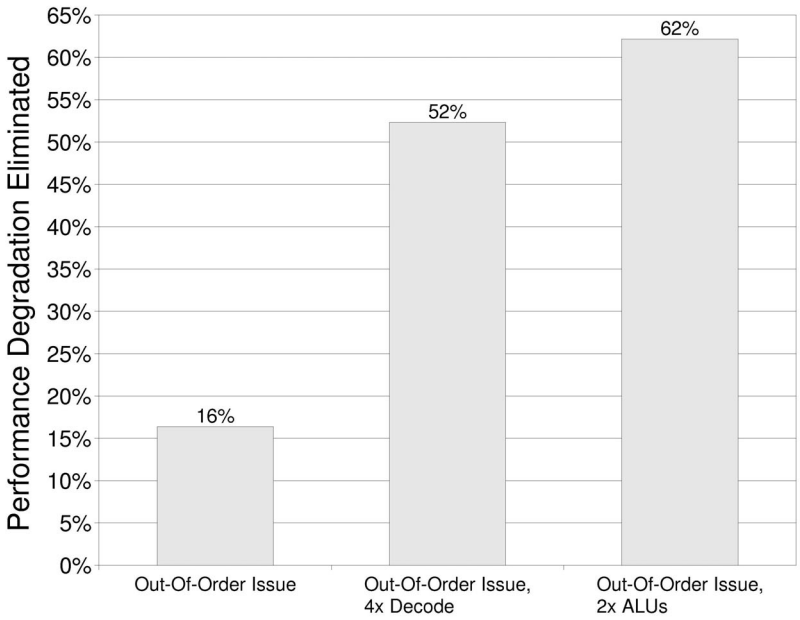
Distribution of synchronization overhead



Alpha 21264-like synchronization penalties



SA-1110-like synchronization sensitivity



Fine-grain dynamic voltage scaling

- Exploit imbalance of applications in their domain usage
  - Scale individual domain frequencies to match the demand
- Effective over a variety of applications
- Hardware approach: feedback and control system
  - Appropriate for legacy apps
  - Hardware overhead
- Software approach: profiling, insert special domain control instructions
  - Appropriate for embedded and other applications which behave consistently among different runs
  - Recompilation or binary rewriting

Voltage scaling hardware models

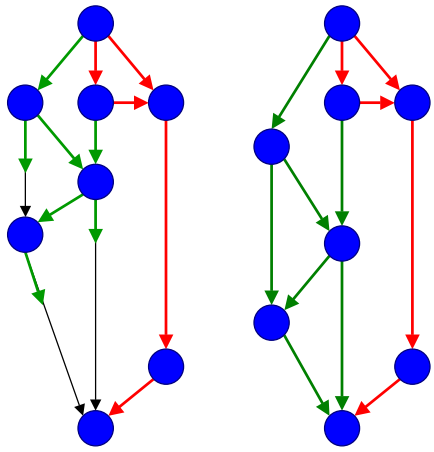
- Voltage range of 1.2-0.65V, frequency range of 250MHz-1GHz in each domain (same as baseline processor)
- Independent jitter for each domain
  - Calculate next clock edge based on frequency, last clock edge and jitter
  - Synchronization penalties assessed based on clock edge relationships
- "Transmeta-like" model
  - Models having to pause operation while increasing frequency and voltage
  - 32 voltage steps, 28.6mV intervals
  - 20us per change
- "Xscale-like" model
  - Models being able to operate through changes
  - 320 steps, 2.86mV intervals
  - 0.1718us to transition, but continue to execute

Offline analysis

- Provides target against which to compare more realistic control algorithms
- Can drive energy profiling tool, to help programmers understand applications and hardware
- Can drive re-writing tools for embedded applications
- Summary of operation
  - Run application once at maximum speed
  - Every interval, collect dependences among primitive *events*
  - Stretch events off the critical path, distribute slack as evenly as possible
  - Quantize to respect domain boundaries and reconfiguration overhead; annotate application (simulator)
  - Re-run application with chosen reconfiguration points, to measure real energy savings and performance cost

"Shaker" Algorithm

- Construct a dependence DAG from simulator whose nodes are events, e.g.,
  - Enter instruction fetch queue
  - Enter an issue queue
  - Start execution of an operation
- Timestamp from simulator assigned to each event
- Arcs denote delay between events
- Distribute any slack in the graph among the arcs as evenly as possible
  - Goal: minimize the variance among events in the same domain
  - Alternately traverse the graph up and down, gradually scaling events each time
  - Continue until all slack is removed or all events adjacent to slack edges are at minimum frequency
- O(cN), for N nodes and c frequency steps



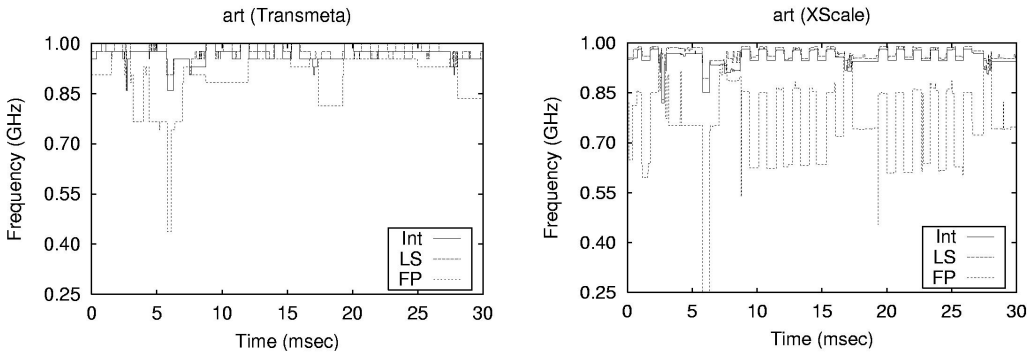
Dilation thresholding

- Creates schedule of domain frequencies based on previous step and performance degradation threshold
- For each domain do
  - For each interval do
    - Construct a histogram of event frequencies from the DAG
    - Identify threshold of acceptable performance degradation
  - Repeatedly merge neighboring intervals when profitable to do so
    - Merge histograms, calculate new frequency and energy savings, merge intervals if improvement
    - Amortizes the cost of a voltage/frequency change over the time spent at that voltage frequency for the "Transmeta" model
- Output list of reconfiguration points

Simulation Parameters

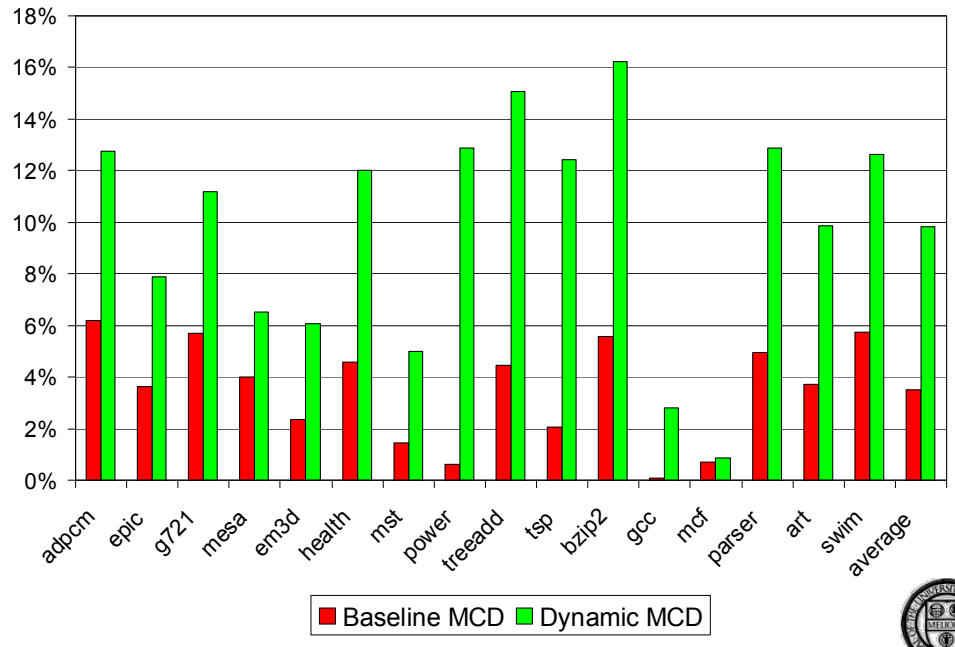
- Resources similar to Alpha 21264
- Voltage range: 0.65 – 1.2 V
- Frequency range: 0.25 – 1 GHz
- Representative benchmarks from:
  - Mediabench
  - Olden
  - SPEC 2000 (int and fp)
- Three configurations:
  - MCD at maximum frequency (baseline MCD)
  - MCD with dynamic voltage scaling (dynamic MCD)
  - Single-clock with dynamic but global voltage scaling
- No attempt to scale front-end domain
- Transmeta-style model (freeze through change)
  - 32 voltage steps: 20µs per step, 10-20µs for frequency change
- XScale-style model (execute through change)
  - 320 voltage steps: 0.1718µs per step

"Transmeta" versus "Xscale" models

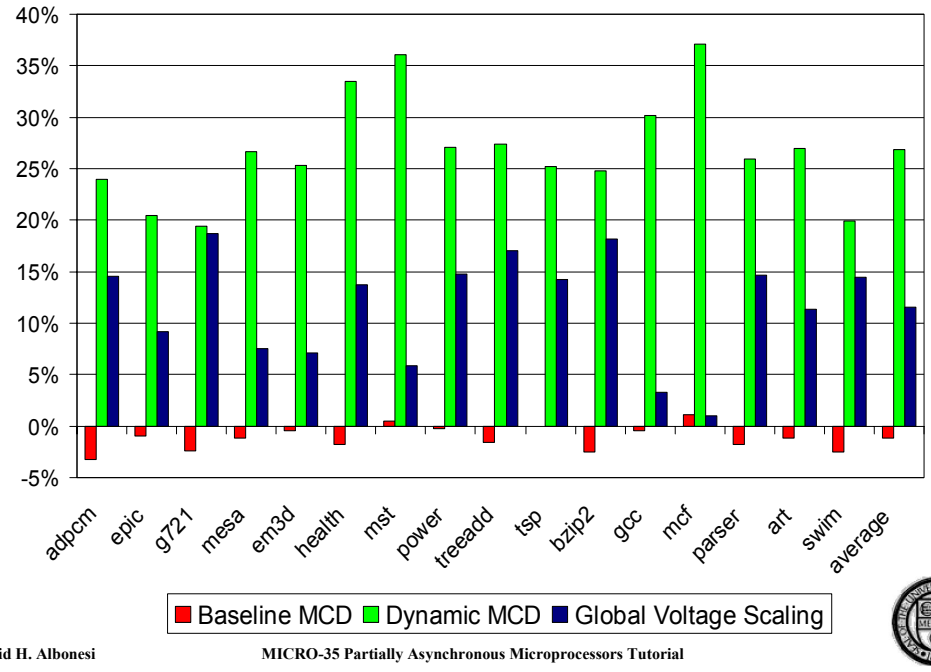


- "Xscale" ability to operate through voltage/frequency changes permits more frequent reconfigurations
- Remaining data for "Xscale" model only

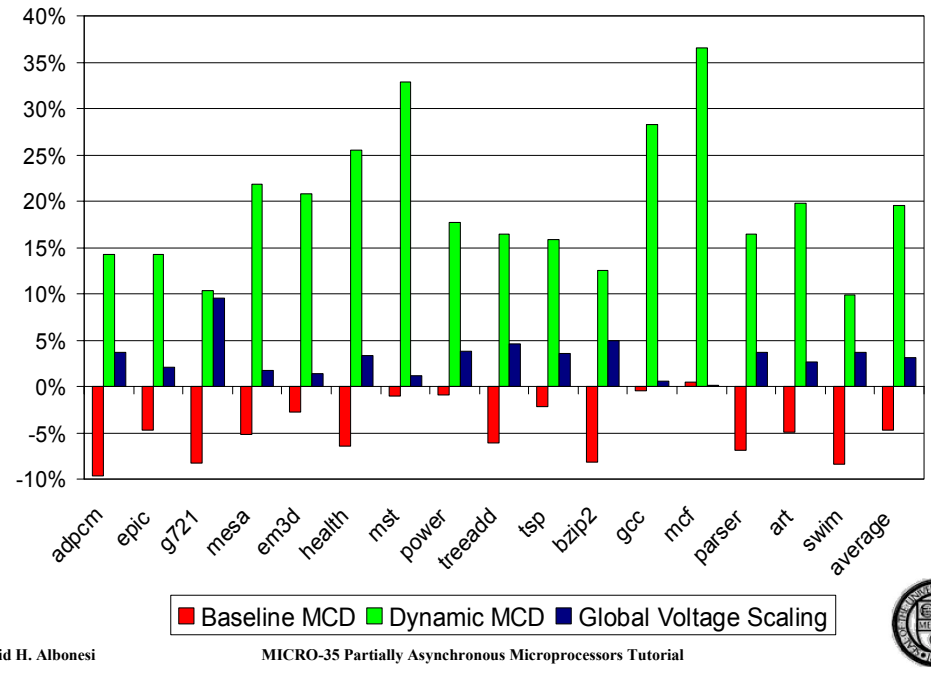
Performance Degradation



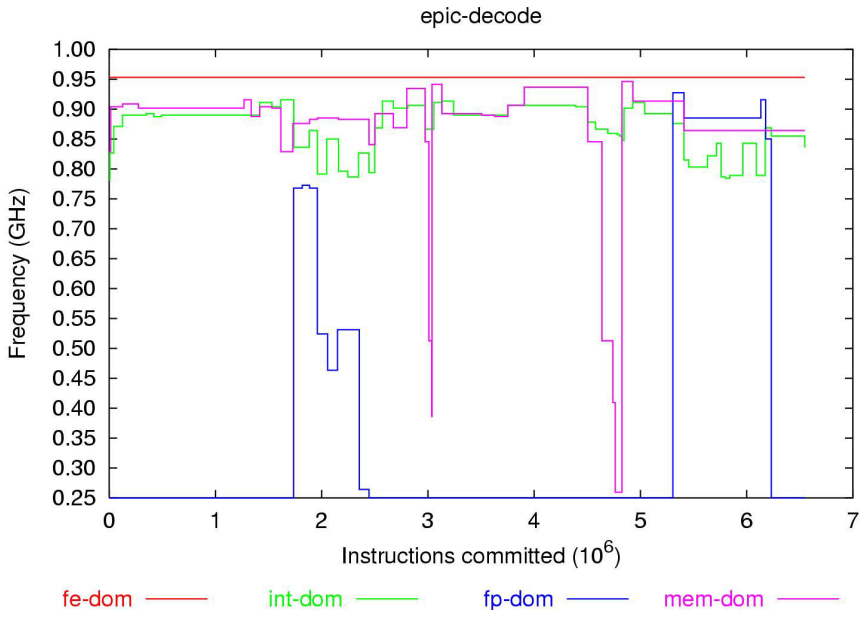
Energy Savings



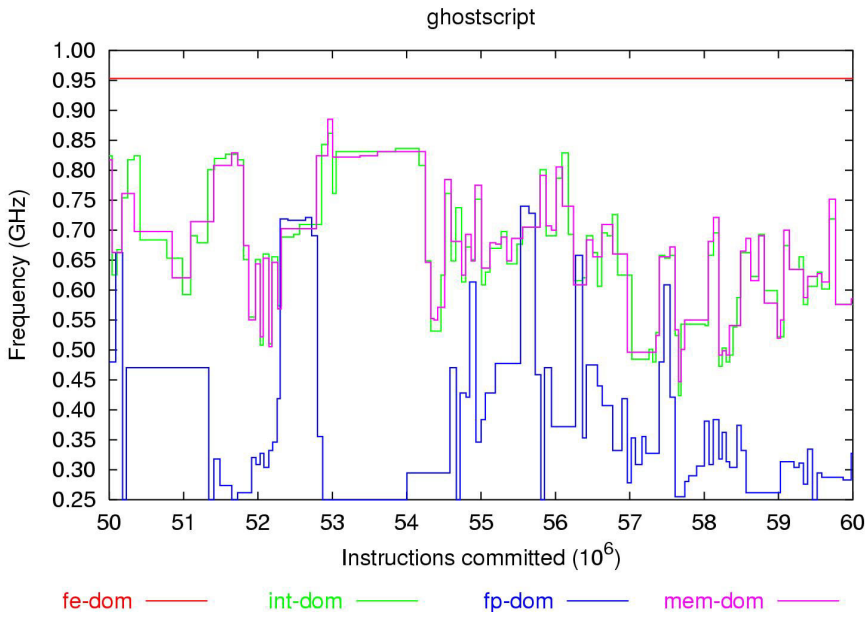
Energy-Delay Product



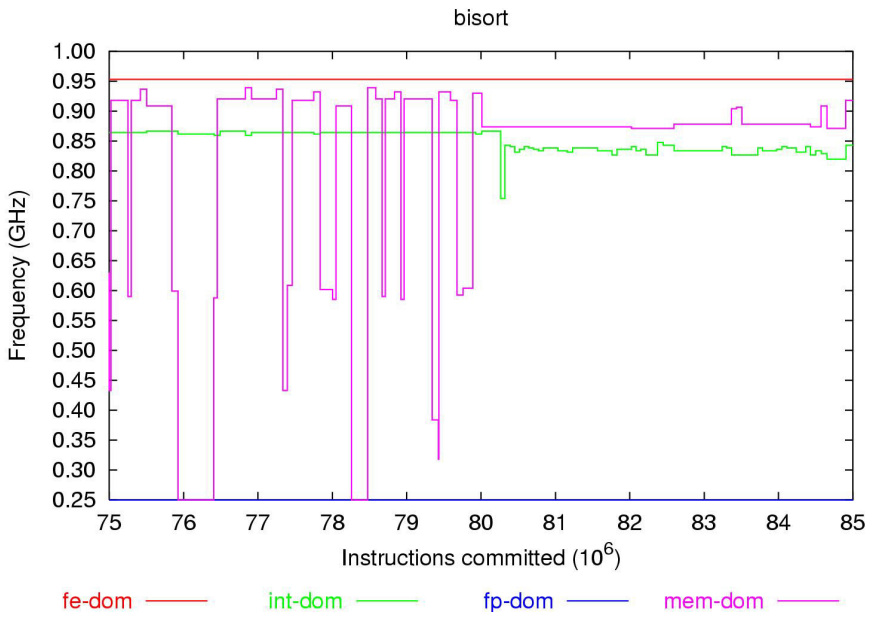
Epic-decode – Runtime Example



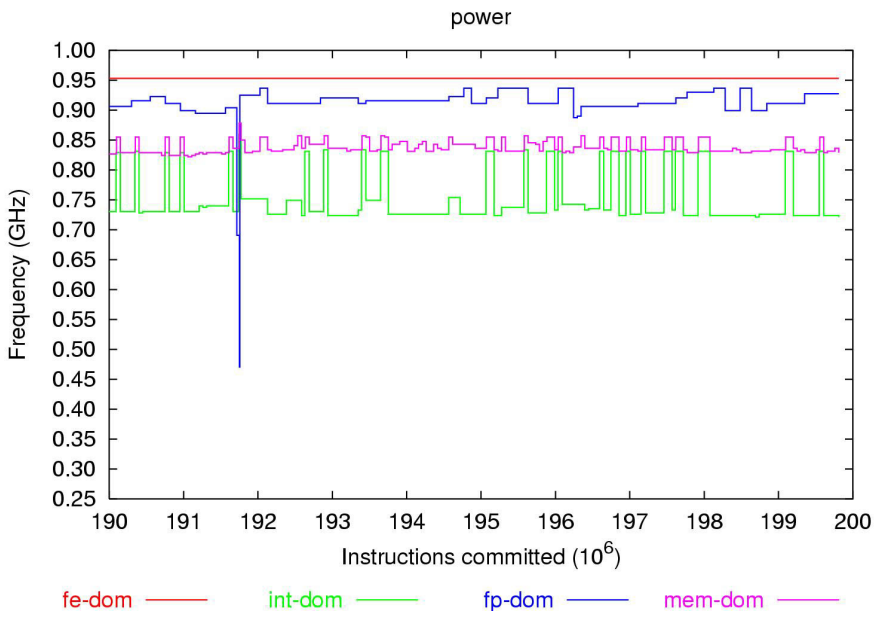
Ghostscript – Runtime Example



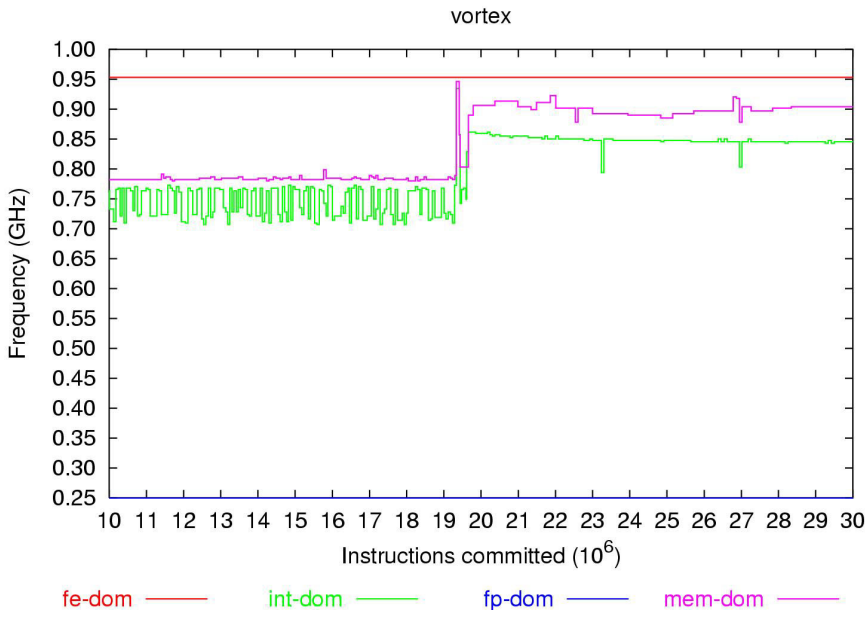
Bisort – Runtime Example



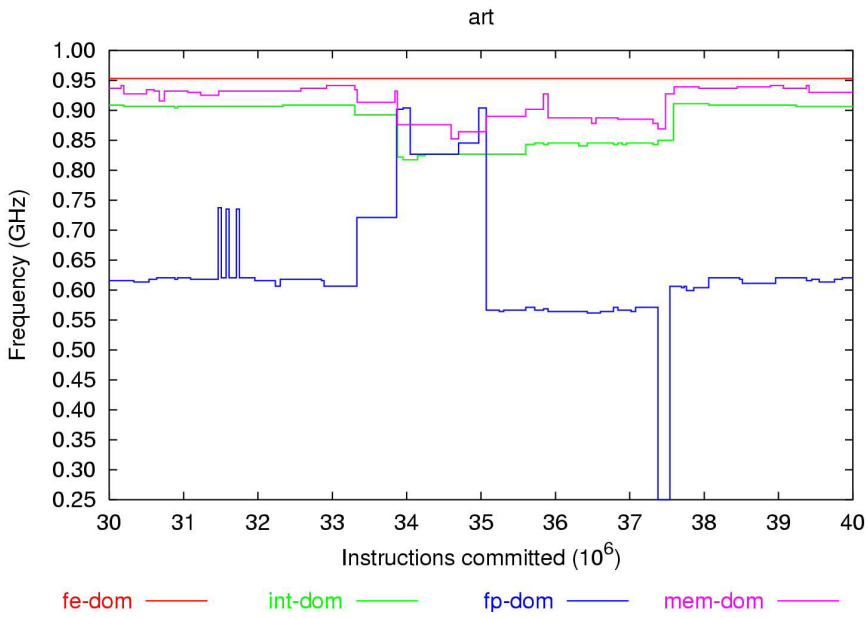
Power – Runtime Example



Vortex – Runtime Example



Art – Runtime Example



Offline Result Summary

- Dynamic MCD
  - Less than 10% performance degradation
  - About 27% energy savings
  - 20% energy-delay product
- Global voltage scaling
  - About 12% energy savings
  - 3% energy-delay product
- Appreciable variability among application phases

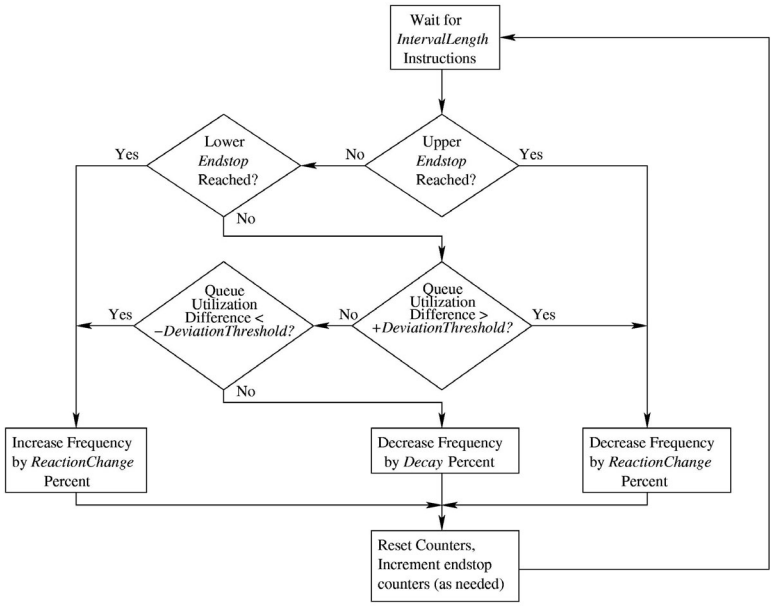
Hardware control: the attack/decay algorithm

- Exploits correlation between changes in input queue utilization and domain frequency
- Each domain operates independently
- Can be implemented in ~10K transistors for a four-domain processor

Attack/decay design space

- **Deviation Threshold**
  - Difference in utilization needed to trigger an attack
- **Reaction Change**
  - Amount of frequency change on an *attack*
- **Decay**
  - Amount of frequency decrease on a decay
- **Performance degradation threshold**
  - Amount of performance degradation during the last interval below which a frequency decrease is allowed in the next interval
- **Endstop count**
  - Consecutive intervals at max or mix frequency after which we force an attack

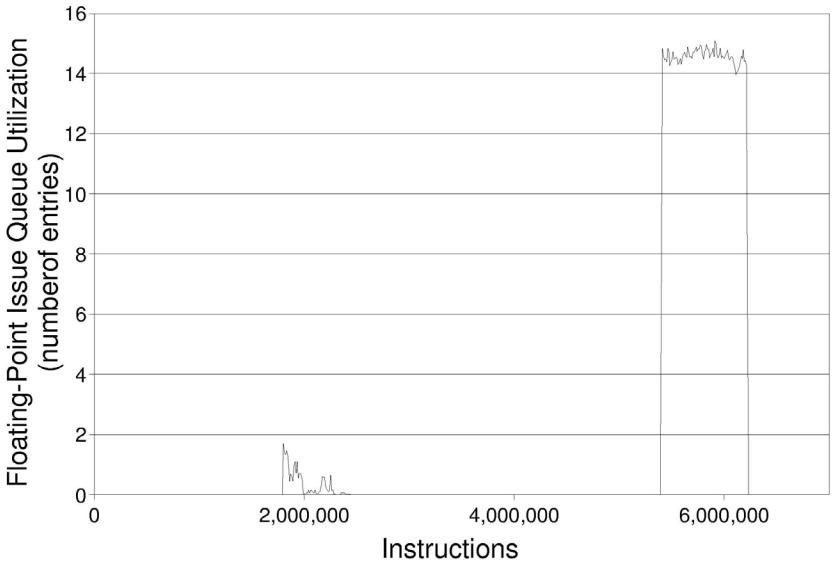
Hardware control: the attack/decay algorithm



Note: All frequency adjustments are conditional on the change in IPC being less than *PerformanceDegradationThreshold* percent.

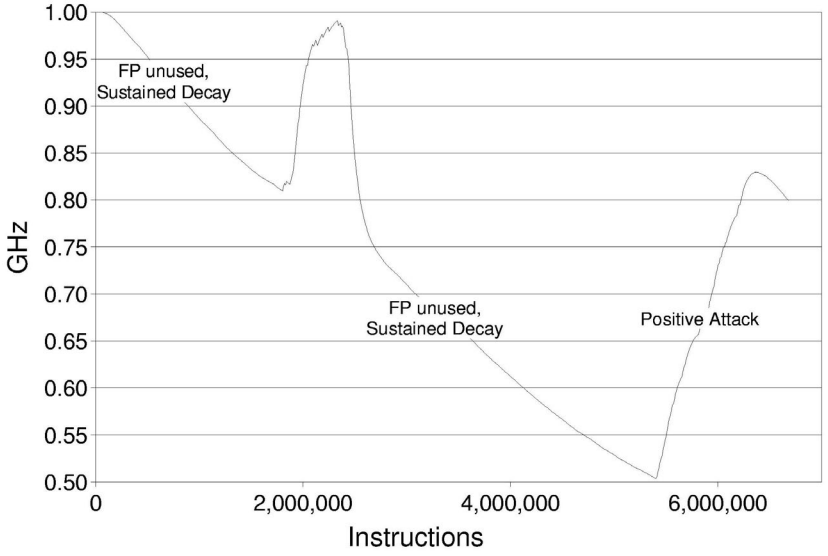
Attack/decay algorithm example #1

- Changes in floating point queue utilization for epic decode



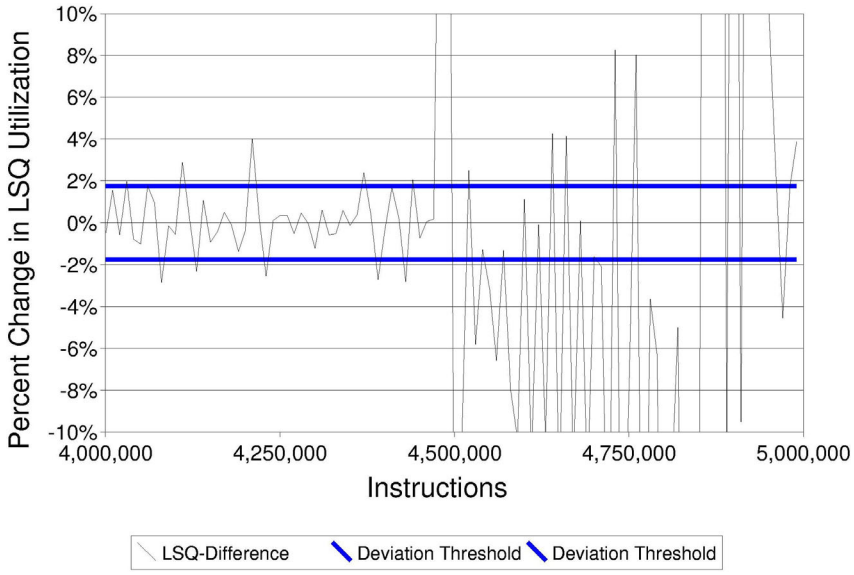
Attack/decay algorithm example #1

- Changes in floating point frequency for epic decode



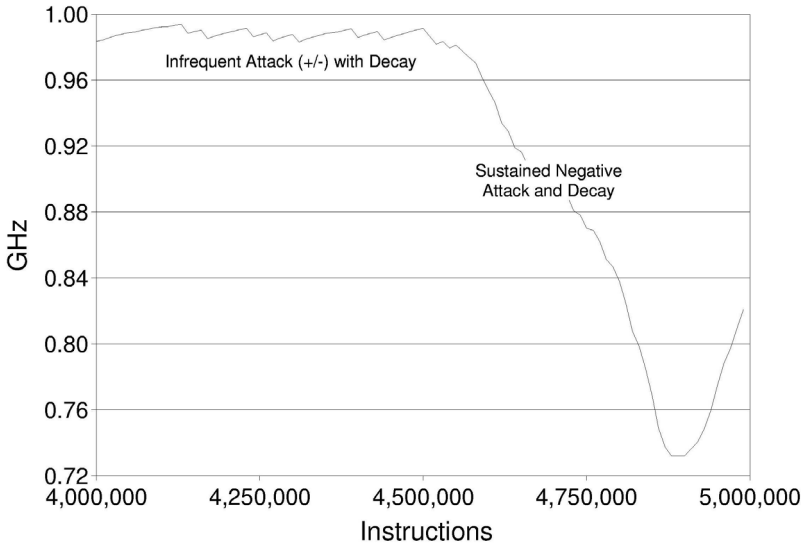
Attack/decay algorithm example #2

- Differences in load/store queue utilization for epic decode

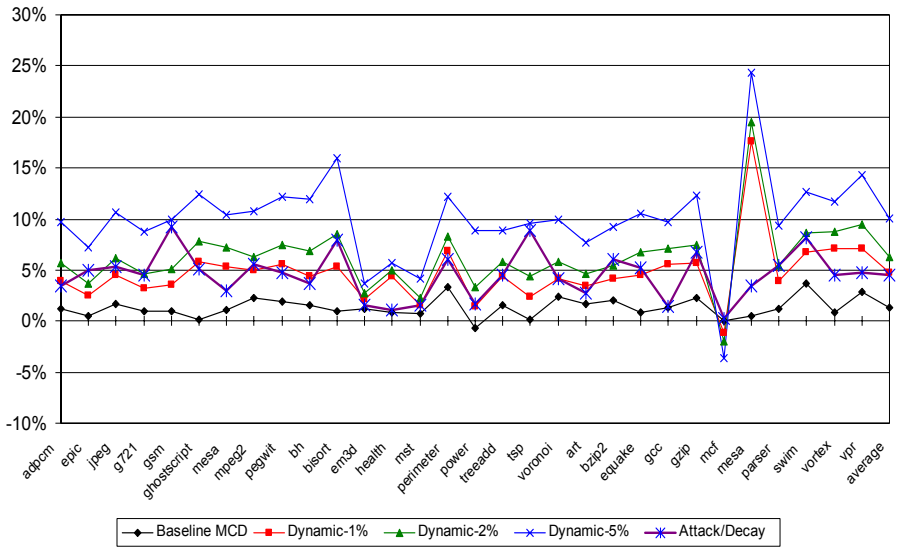


Attack/decay algorithm example #2

- Changes in load/store frequency for epic decode

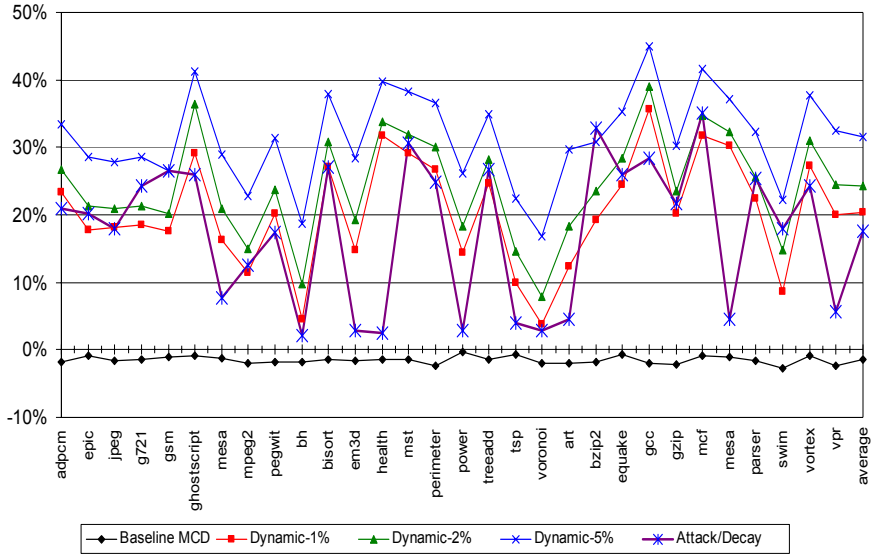


Performance degradation



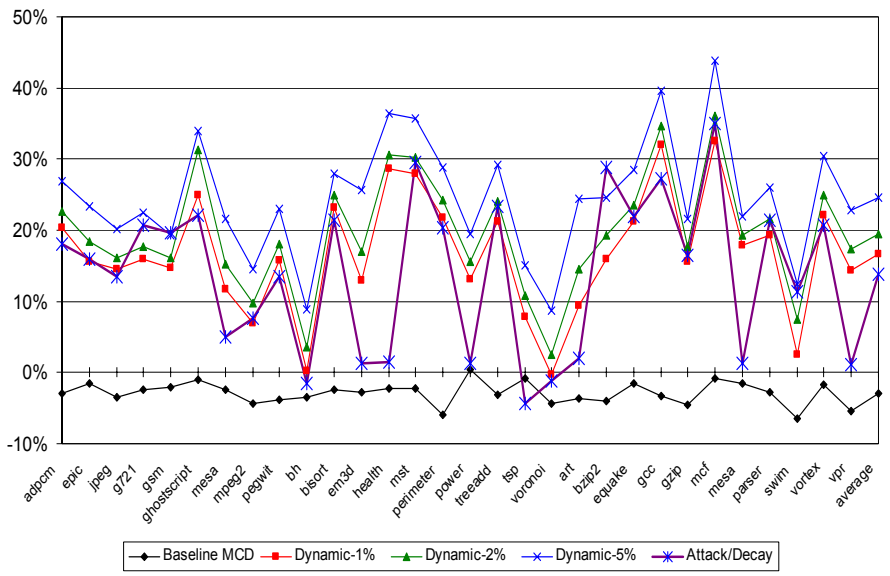
- **Same overall performance degradation as offline with 1% performance degradation target (Dynamic-1%)**

Energy savings



- **Achieves 86% of the energy savings as offline with 1% performance degradation target (Dynamic-1%)**

Energy-delay improvement



- Achieves 86% of the energy-delay improvement as offline with 1% performance degradation target (Dynamic-1%)

Attack/decay summary

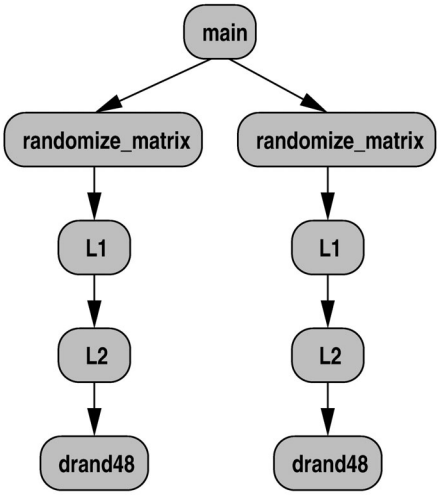
- Correlates input queue utilization changes with frequency changes
- Independent control for each domain
- Implementable in a reasonable number of transistors
  - ~0.1% of a 10M transistor chip
- Achieves 86% of the energy savings of an offline algorithm with identical performance degradation

Profile-driven MCD control

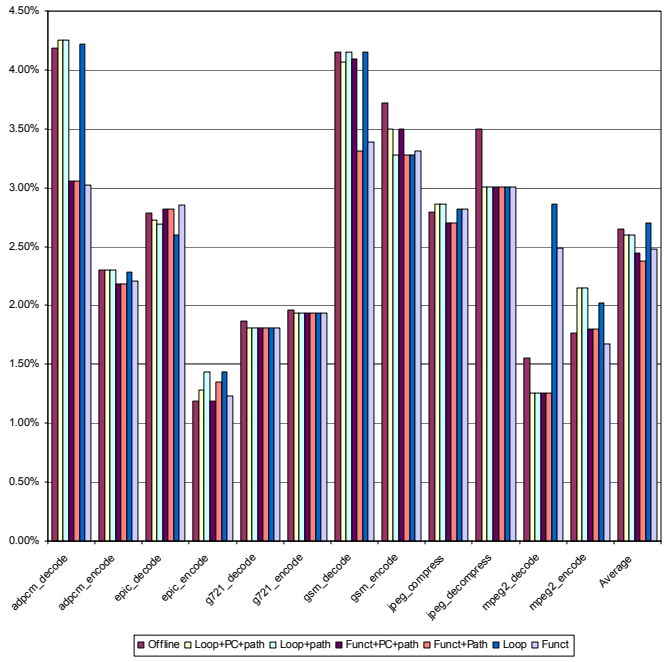
- Profile run to identify long-running loops and functions for which the cost of reconfiguration can be effectively amortized
- Shaker and dilation thresholding algorithms to identify domain frequencies for each
- Identify functions, loops, and call tree paths at runtime through table lookups
- Distinguish "important" functions and loops and set frequencies accordingly

Distinguishing loop/function instances

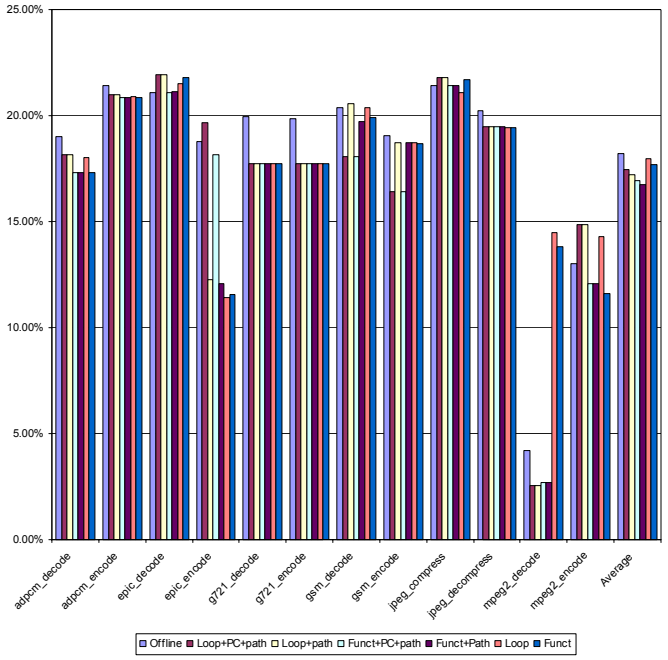
- Build a call tree, each node is a function or loop instance
- Distinguish different paths to a given function or loop
- Table lookup using path, loop/function PC, and possible call PC to identify function
- If marked "important", change frequencies/voltages



Performance comparison with offline



Energy comparison with offline



MCD performance optimizations

- Each domain can run at its natural frequency
- Global clock skew eliminated
  - Saves clock power and metal also
- Dynamically trade off size for speed within each domain using an *adaptive MCD*

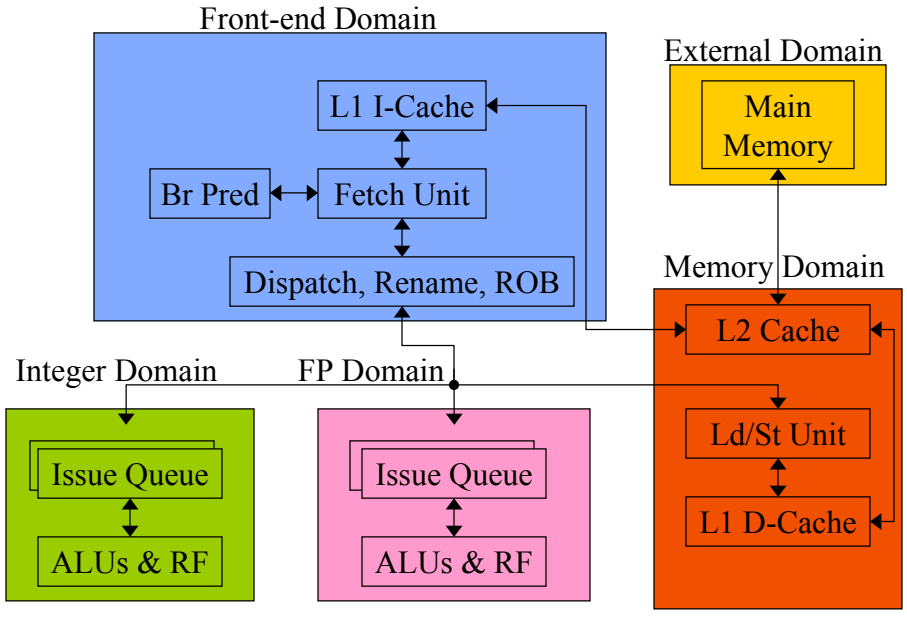
Dynamic resizing

- Many dynamic resizing techniques proposed for power
- Speed of an adaptive structure depends on configuration
  - Adaptive issue queue 70% faster at ¼ size
- Synchronous system cannot exploit the faster speed of a downsized structure due to other critical paths
- Structure can be overly upsized for a particular application but entire system must be slowed down (global clock)

Adaptive MCD

- Idea: upsize structures within MCD domains so as not to impact other domain frequencies
- Design each domain to be heavily pipelined for high frequency (perhaps even overpipelined)
- Make selected structures adaptive to exploit ILP or to match larger working sets
- Upsize structures and adjust frequency when IPC improvement would override frequency decrease
- Drawbacks
  - Overpipelining for slower frequencies (IPC penalty)
  - Configuration overheads degrade clock speed relative to fixed design

Adaptive MCD organization



Resizable structures

- Front end domain
  - Icache: 4KB-64KB 2-way
  - Branch predictor sized according to Icache
    - gshare PHT: 4KB-64KB
    - Local PHT: 1KB-8KB, local BHT: 512 or 1024 entries
- Integer and floating point domains
  - Issue queue: 16-64 entries
- Load/store domain
  - Dcache: 32KB 1-way – 256KB 8-way
  - L2 cache: 256KB 1-way – 2MB 8-way sized according to Dcache

Methodology

- Baselines
  - Fixed MCD: MCD with fixed frequencies with best overall TPI
  - Fully synchronous: design with best overall TPI
- Fully synchronous structures sized to balanced delays
- Adaptive MCD additional branch penalty: 2 integer cycles and 1 front end cycle
- Adaptive MCD frequency penalty as much as 49%
- Per-application adaptation (profiling)
- Benchmarks
  - 14 Mediabench
  - 3 Olden
  - 6 SPEC2000int
  - 3 SPEC2000fp

Fixed MCD versus synchronous

- Best fixed MCD organization
  - 32KB Icache
  - 32KB gshare PHT, 4KB local PHT, 1KB local BHT
  - 128KB Dcache, 1MB L2
  - 16 entry queues
- Best synchronous organization
  - 64KB Icache
  - 64KB gshare PHT, 8KB local PHT, 1KB local BHT
  - 64KB Dcache, 512KB L2
  - 32 entry queues
- 5% overall performance improvement

Adaptive MCD versus synchronous

- Adaptive MCD config for each benchmark

	Dcache			
Icache	**32KB**	**64KB**	**128KB**	**256KB**
4KB	adpcm encode, bzip2, adpcm encode			
8KB	mpeg2 encode, swim, perimeter	gzip		
16KB	jpeg compress, g721 encode, g721 encode, equake	epic decode, bh	epic encode	
32KB	jpeg compress, vpr		parser, mesa mipmap (IQ=32)	em3d (IQ=32)
64KB	gsm encode, ghostscript, gsm encode	mesa, vortex	mesa osdemo, gcc	

- 18% overall performance improvement, min –3%, max 50%

Areas for further research

- Best division into domains
- Circuits for voltage/frequency islands
- Front-end control (currently fixed)
- Dynamic voltage gating for leakage
  - Voltage scaling works best when work is "smoothed out" over a long period of time
  - Voltage gating works best when work is "clumped together" to introduce idle time
  - Best combination of the two that optimizes energy-delay
- Combining performance and energy features

For More Info…

http://www.ece.rochester.edu/~albonesi/acal