MCD: A Multiple Clock Domain Microarchitecture

Dave Albonesi

in collaboration with

Greg Semeraro

Grigoris Magklis

Rajeev Balasubramonian

Steve Dropsho

Sandhya Dwarkadas

Michael Scott

Caveats

- We started this project in 1999
  - But didn't get rolling until 2000
- First publication was February 2002 in HPCA
- Hit the "tip of the iceberg" so far

⇒ Much of what I tell you is in flux and subject to change

⇒ But lots of room for interesting research!

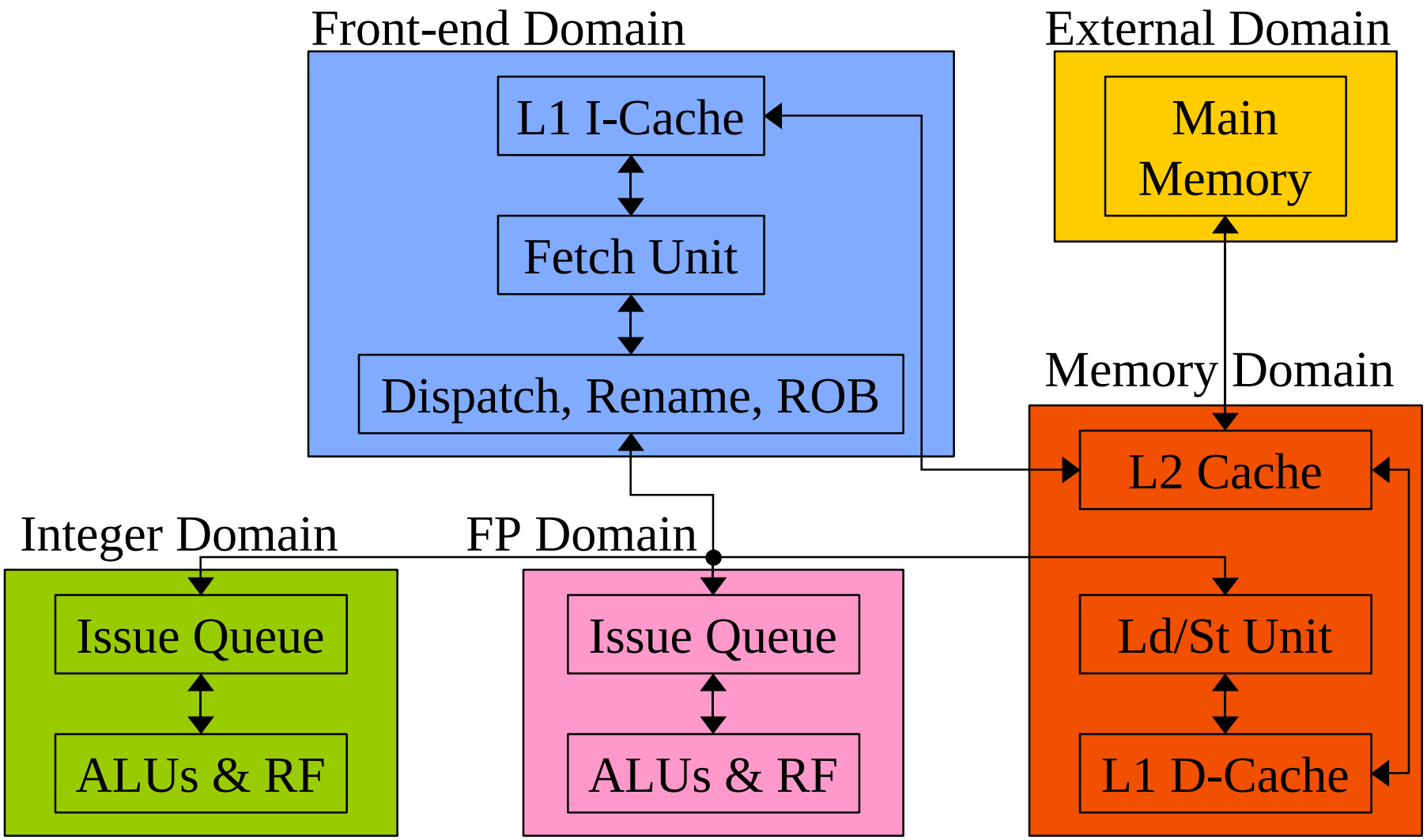
Motivation for MCD

- Greater asynchrony likely in the future
  - Doug Matzke of TI in IEEE Computer, September 1997
  - Barbara Chappell of Intel in IEEE Spectrum, July 1999
- Companies have a tremendous investment in synchronous design
- Designers know how to deal with sending signals between clock domains
- Gradual elimination of global signals creating more autonomous chip 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 are here, and on-chip voltage conversion is on the way
- Limitations of global Dynamic Voltage Scaling (DVS)
  - Applicable to a subset (e.g., rate based) of all applications
  - Shrinking gap between supply and threshold voltages decreasing DVS effectiveness
- Applications may be bottlenecked by a subset of the major functions (fetch/dispatch, integer, floating point, load/store) of a general-purpose processor
  - Different bottlenecks for different applications
  - Different bottlenecks and degrees of bottlenecks for phases of a given application

MCD at a high level



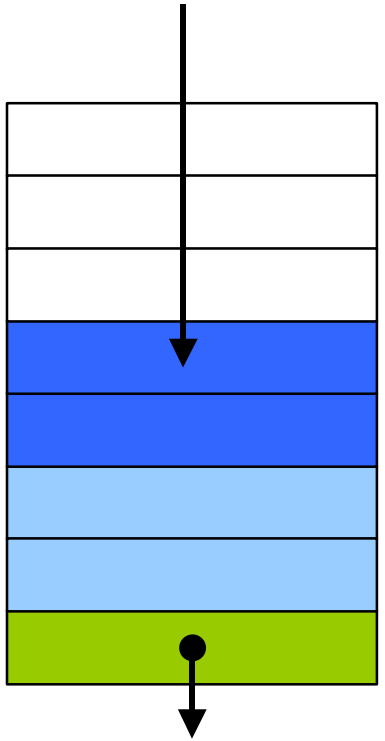
Rev1.0 MCD microarchitecture design decisions

- **Four on-chip domains**
  - 16% of chip reachable in single clock at 100nm technology [Matzke]
  - Assume two processors per die, 1MB of on-chip cache
  - Results in five equal-size domains
  - Chose four to account for different sizes
  - *Research on the appropriate GALS granularity is direly needed!*
- **Break down into front-end, integer, floating point, memory**
  - 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 queue remains partially full
  - Much autonomy between these major functions
- **Separate L1 Dcache from integer and floating point**
  - Performance not adversely effected
    - May be sensitive to the synchronization cost (ours is low)
  - Allows memory to be separately optimized
- **Put the L2 cache in the memory domain**
  - No L1-L2 synchronization penalty for loads/stores
  - Applications with large L1 Icache miss rates may be impacted

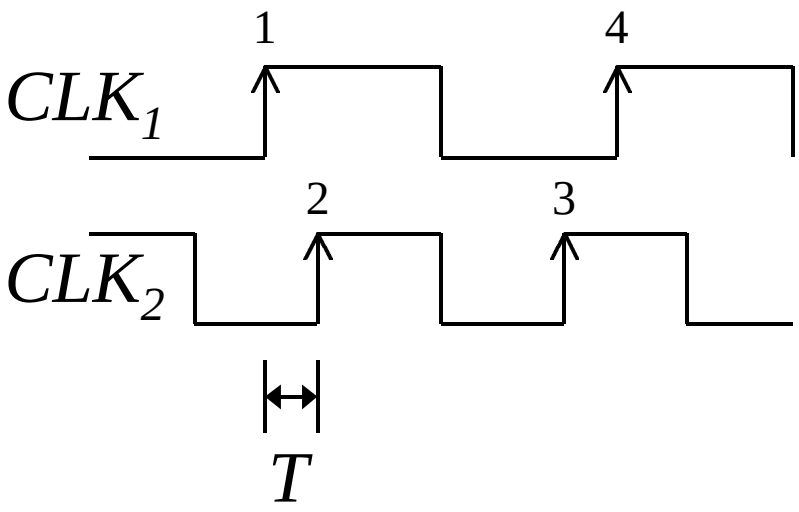
Synchronization via queues

- **Key insight: synchronization cost can be hidden so long as the queue remains partially full**
- **Detailed design by Chelcea and Nowick in DAC 2001**

FIFO queue structure



Inter-domain synchronization



- Src runs with CLK_1, dst with CLK_2
- Src writes at T_1
- If $T > T_s$ then dst can use the data at T_2
- If $T < T_s$ then dst can use the data at T_3

Synchronization circuit

- From Sjogren and Myers (ARVLSI ’97)
  - Avoids pending timing violation by stretching the clock
  - Ts is 30% of the clock cycle in our design
    - Insensitive to the range of 25-35%
- MCD performance overhead is *very sensitive* to the synchronization cost
  - Roughly 5-10% performance cost to each additional cycle of synchronization time
- *Major issue:* design of robust, yet low latency, inter-domain interfaces tailored to the particular signal/bus being passed in an MCD-style dynamic superscalar microprocessor

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
  - SPEC2000, Mediabench, Olden
- Both hardware and software approaches feasible
  - Hardware: feedback and control system
    - Appropriate for legacy apps
    - Hardware overhead
  - Software: 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

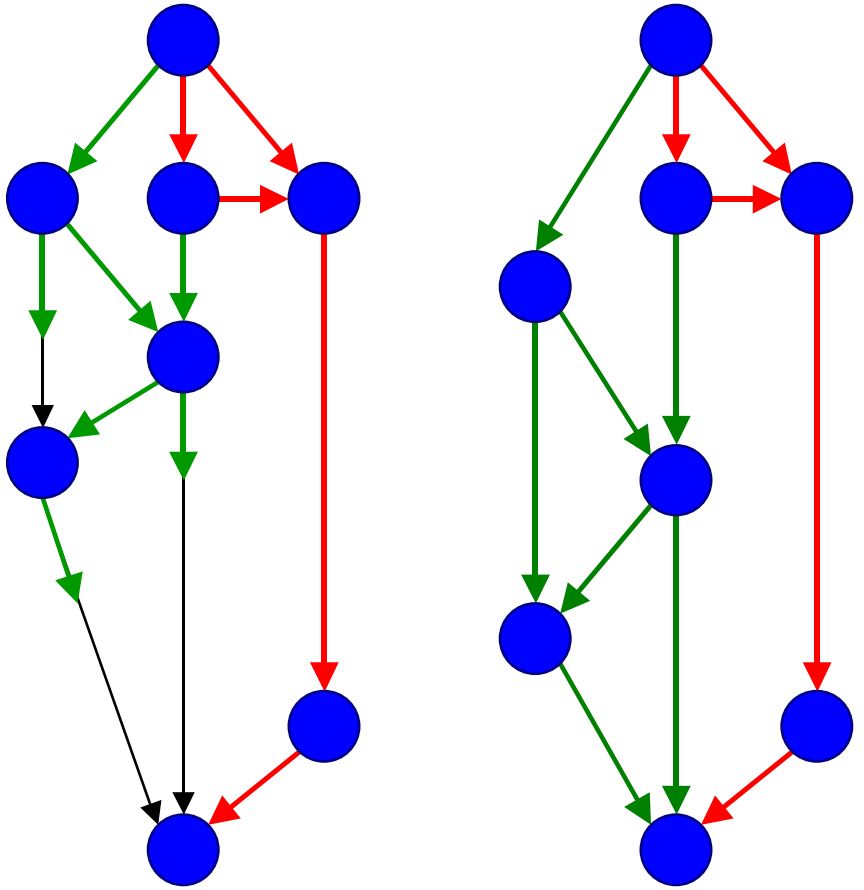
- **Baseline assumptions**
  - Voltage range of 1.2-0.65V, frequency range of 250MHz-1GHz in each domain (same as baseline processor)
  - Independent jitter for each domain
    - Calculate next clock edge based on frequency, last clock edge and jitter
    - Synchronization penalties assessed based on 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

- **Why offline analysis?**
  - **Provide target against which to compare to-be-developed on-line control algorithms**
  - **Drive energy profiling tool, to help programmers understand applications and hardware**
  - **Drive re-writing tools for embedded applications**
- **Summary of operation**
  - **Run application once at maximum speed**
  - **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
- O(cN), for N nodes and c frequency steps



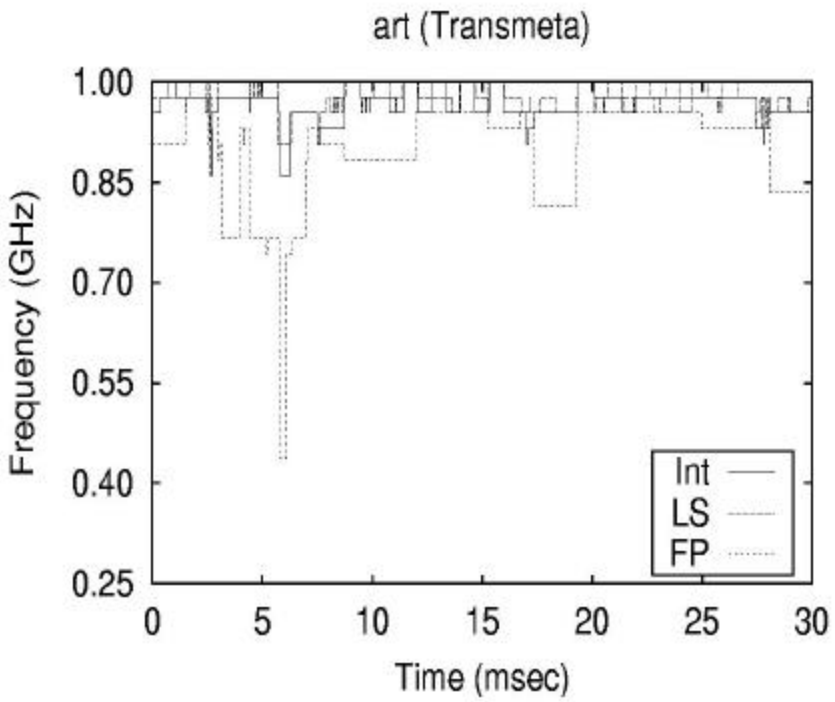
Coalescing Intervals

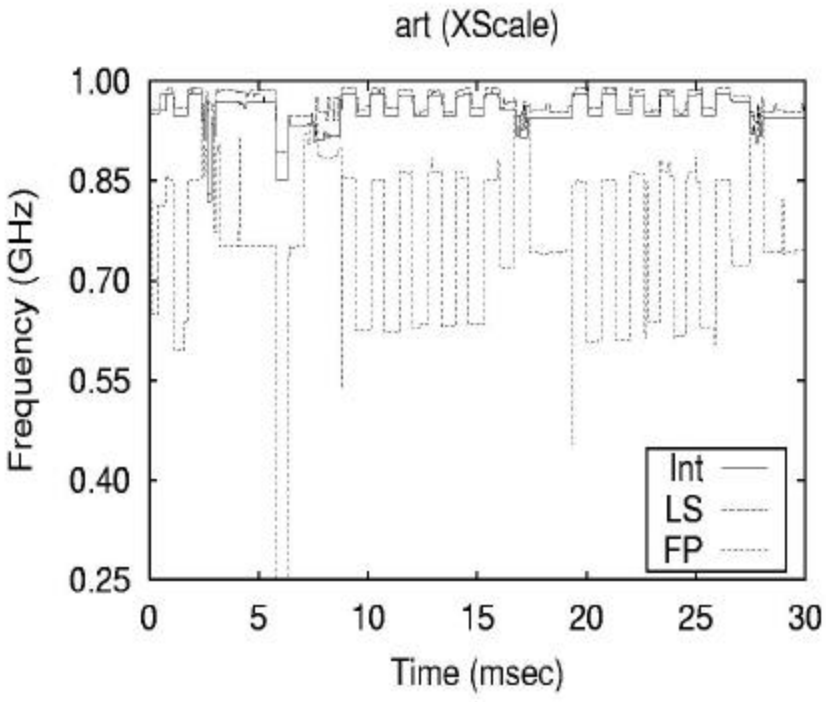
- For each domain do
  - For each interval do
    - Construct a histogram of event frequencies from the DAG
    - Identify threshold of acceptable performance degradation
  - Repeatedly merge neighboring intervals when profitable to do so
    - Merge histograms, calculate new frequency and energy savings, 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 (20% of total)
- 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

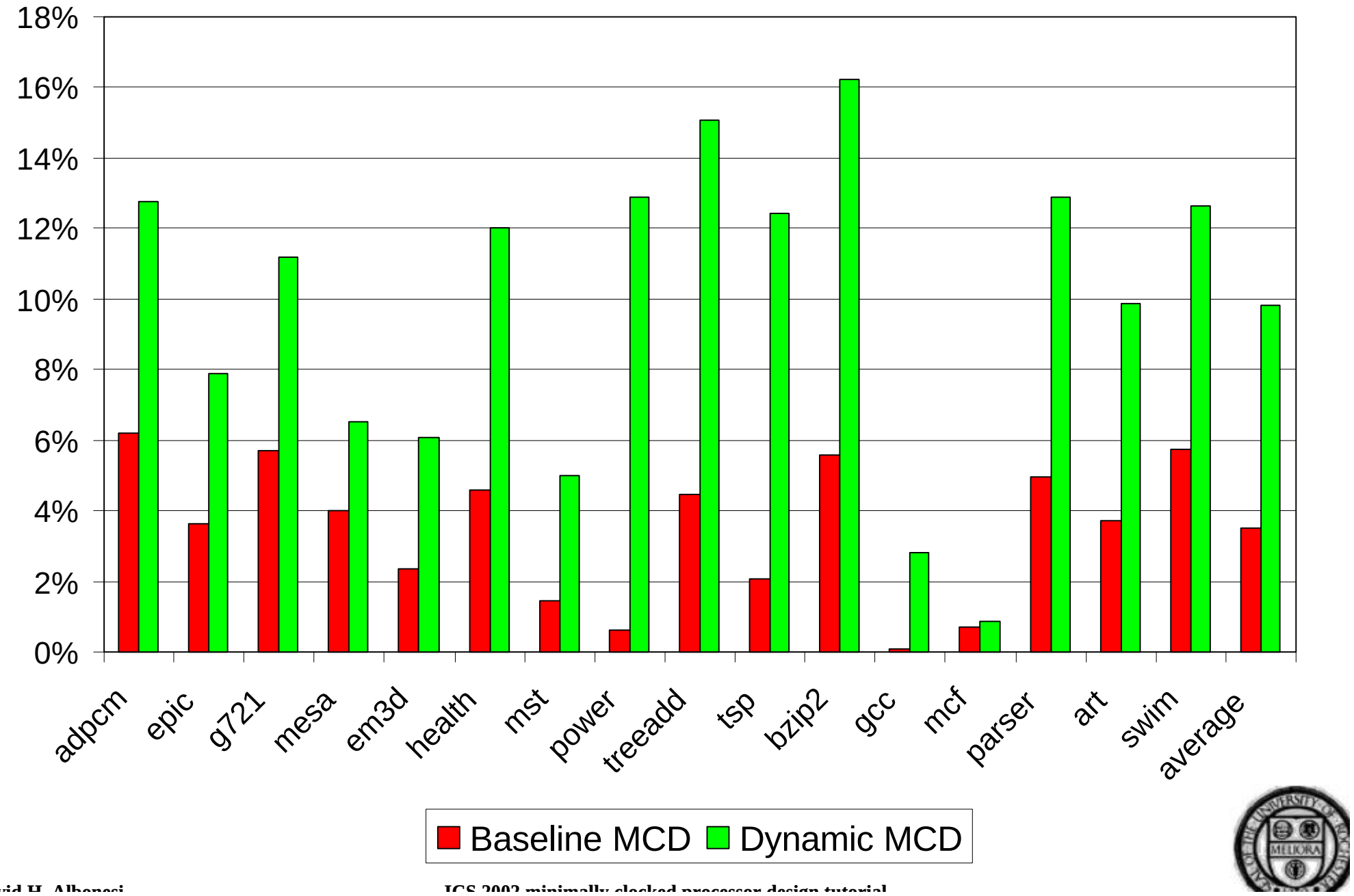
"Transmeta" versus "Xscale" models



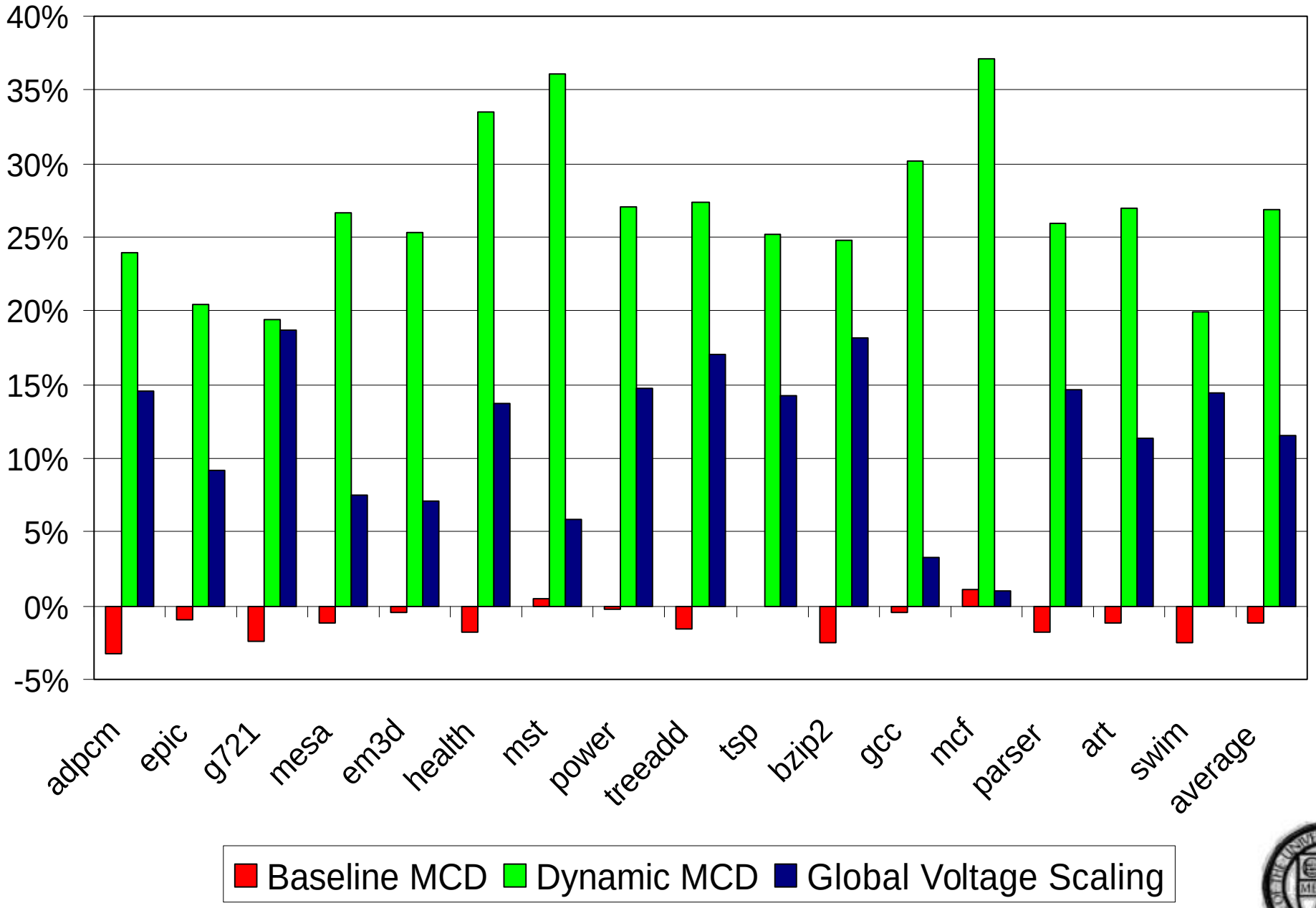


- "Xscale" ability to operate through voltage/frequency changes permits more frequent reconfigurations
- Remaining data for "Xscale" model only

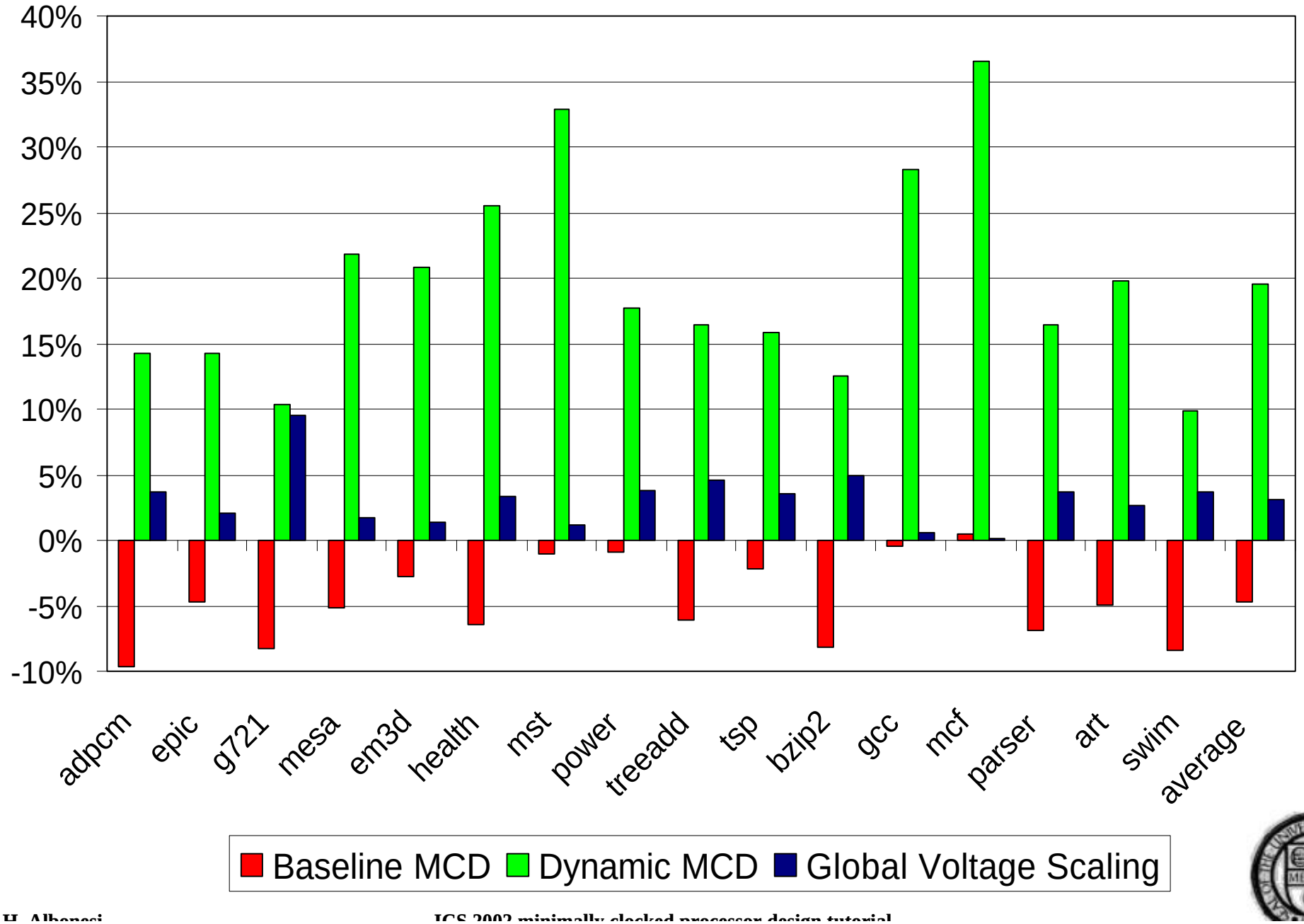
Performance Degradation



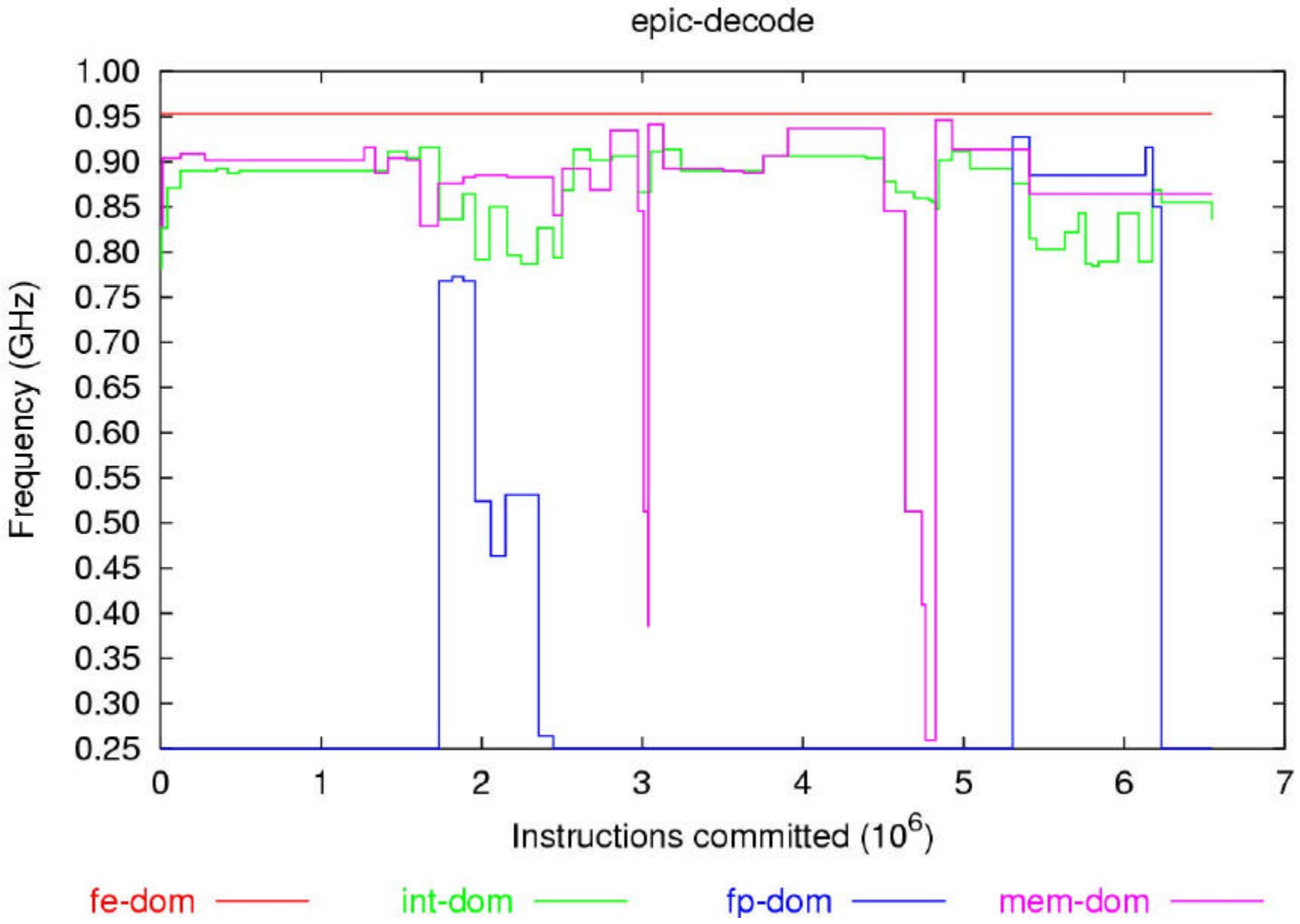
Energy Savings



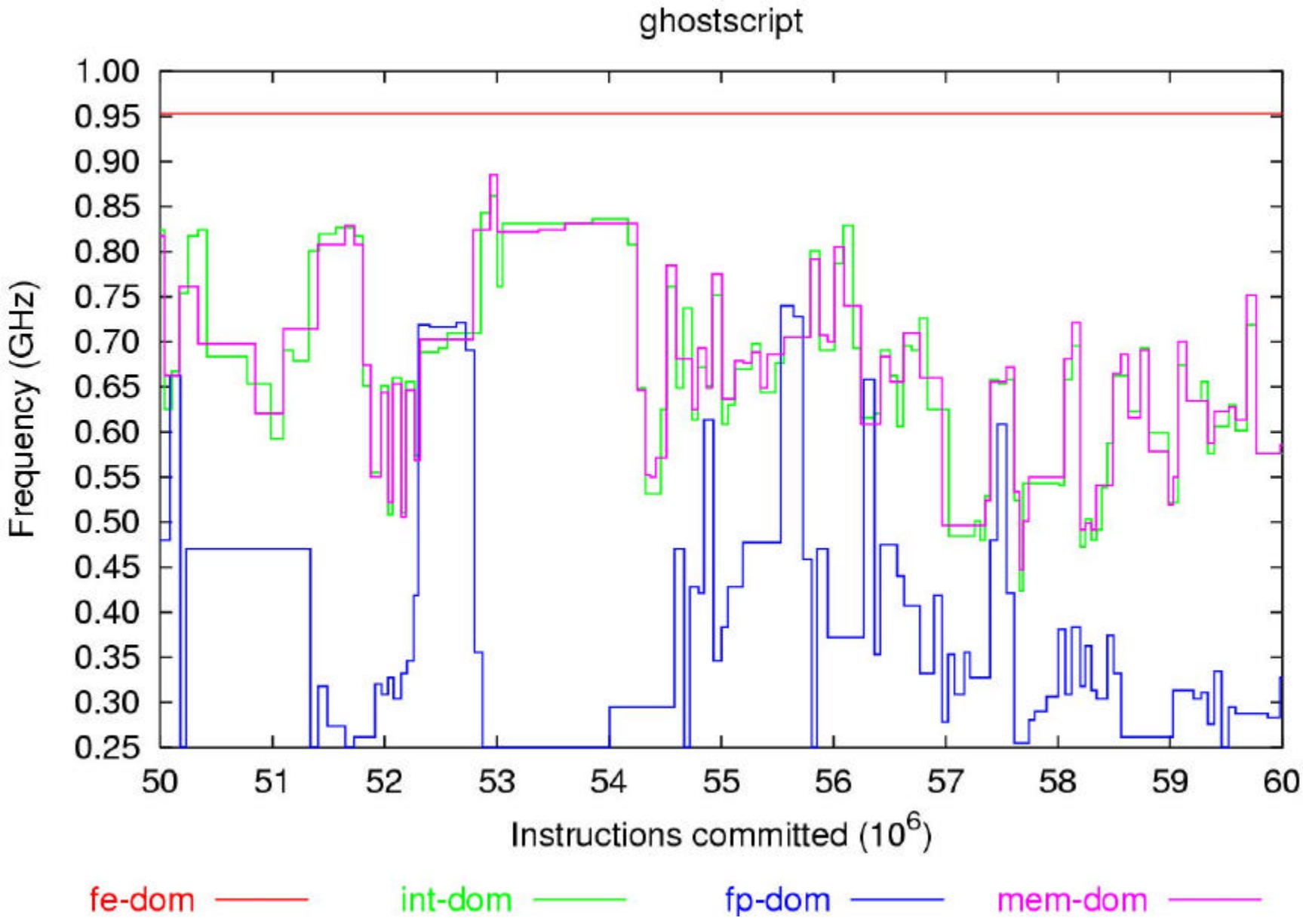
Energy-Delay Product



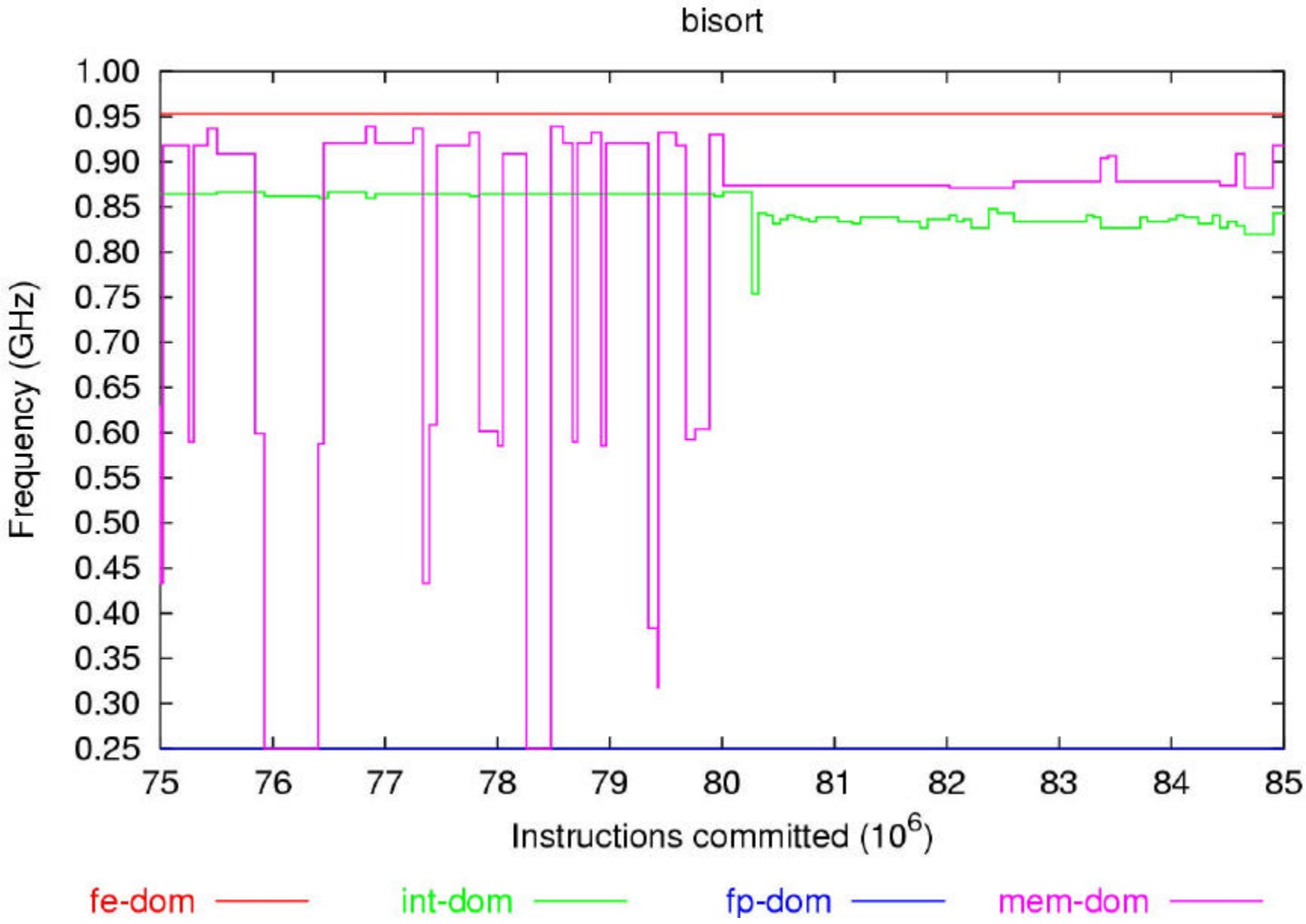
Epic-decode – Runtime Example



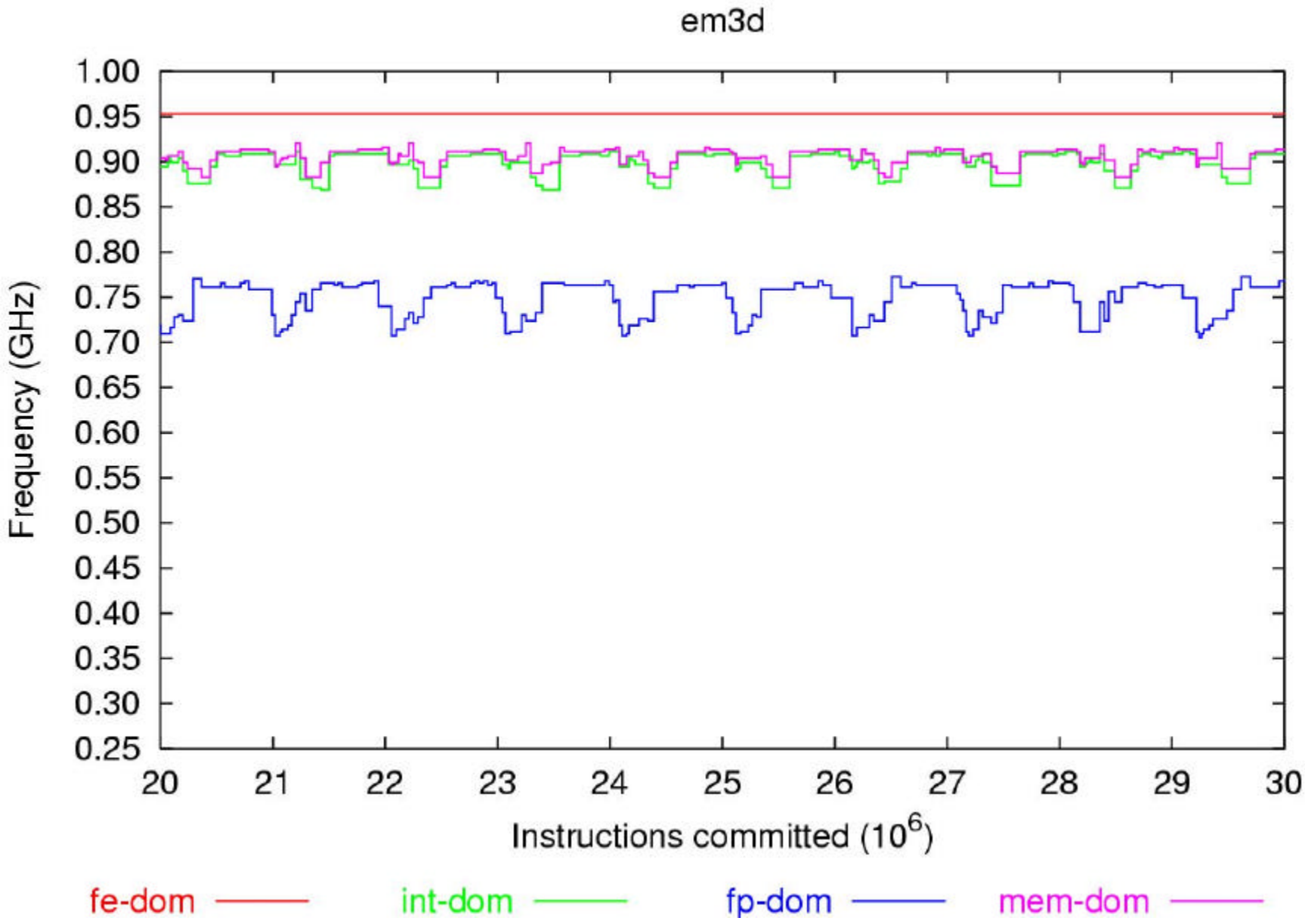
Ghostscript – Runtime Example



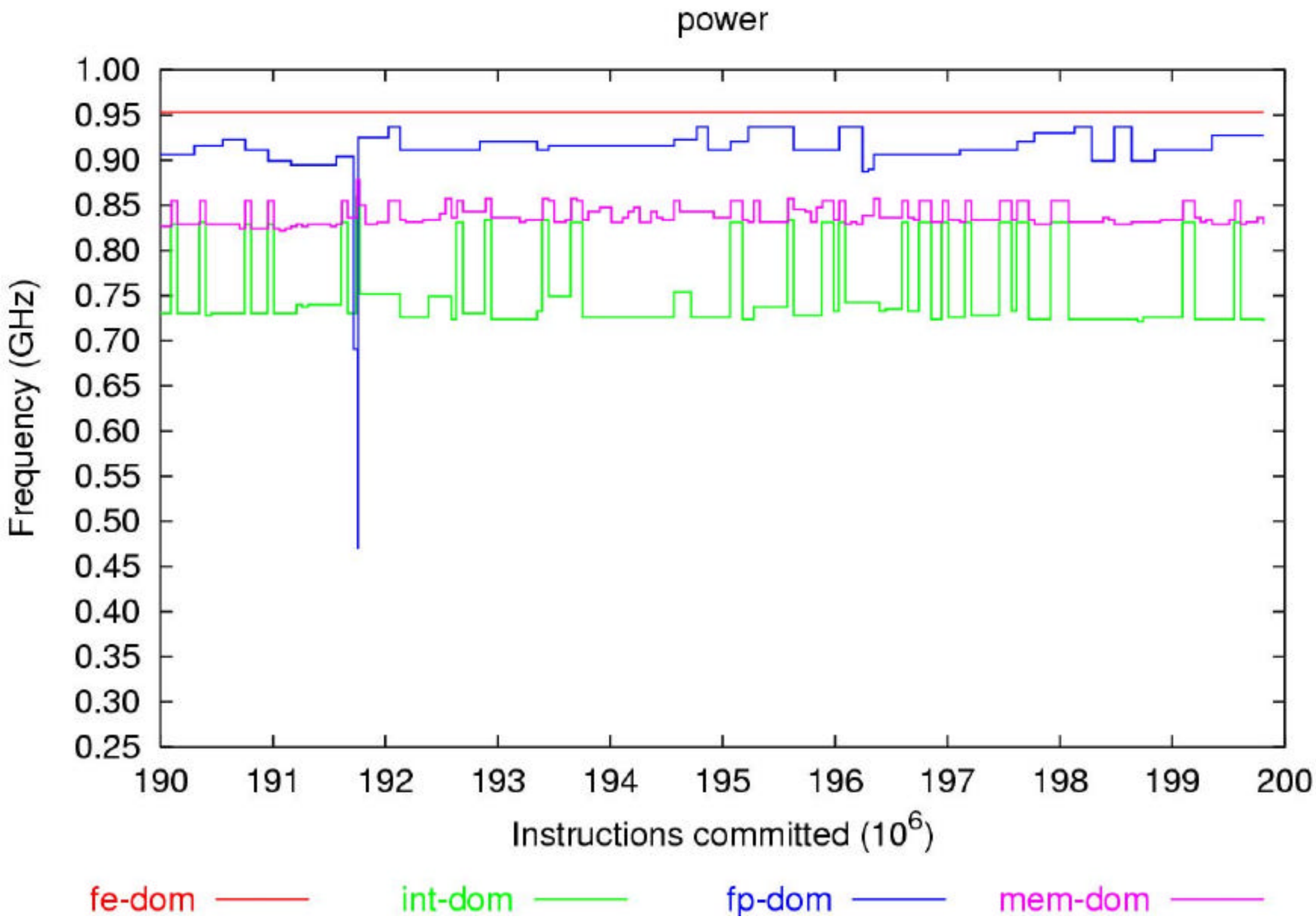
Bisort – Runtime Example



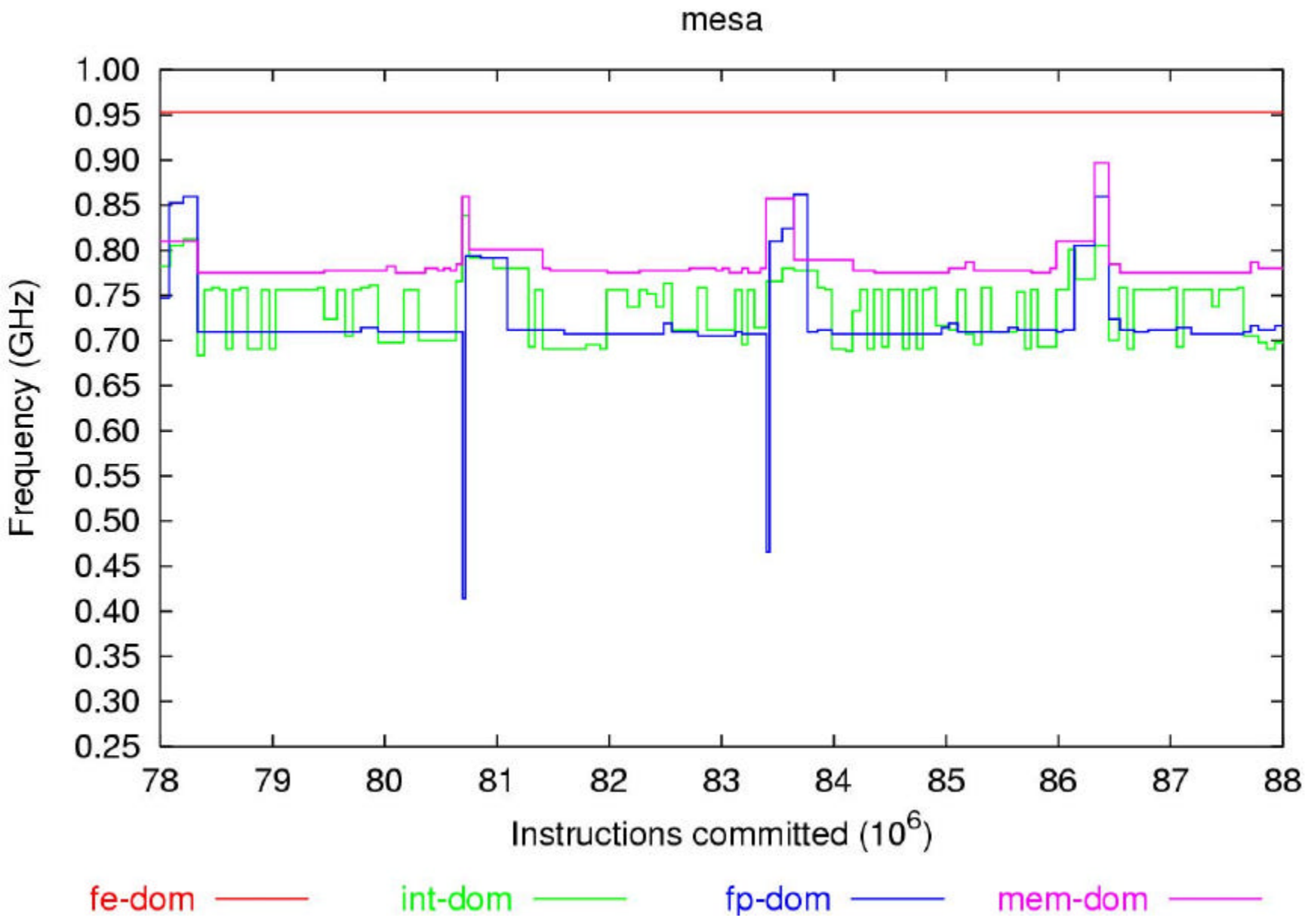
Em3d – Runtime Example



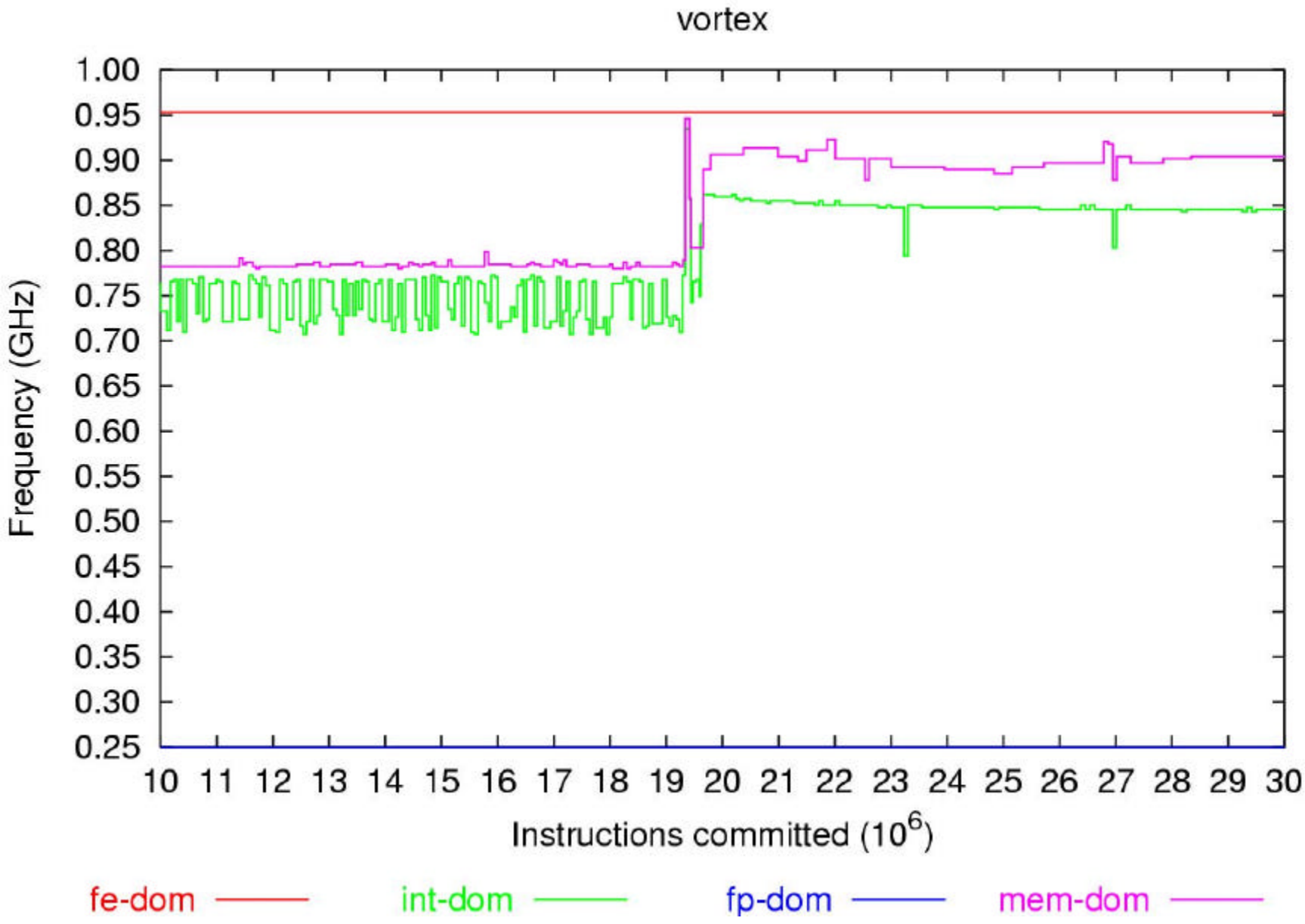
Power – Runtime Example



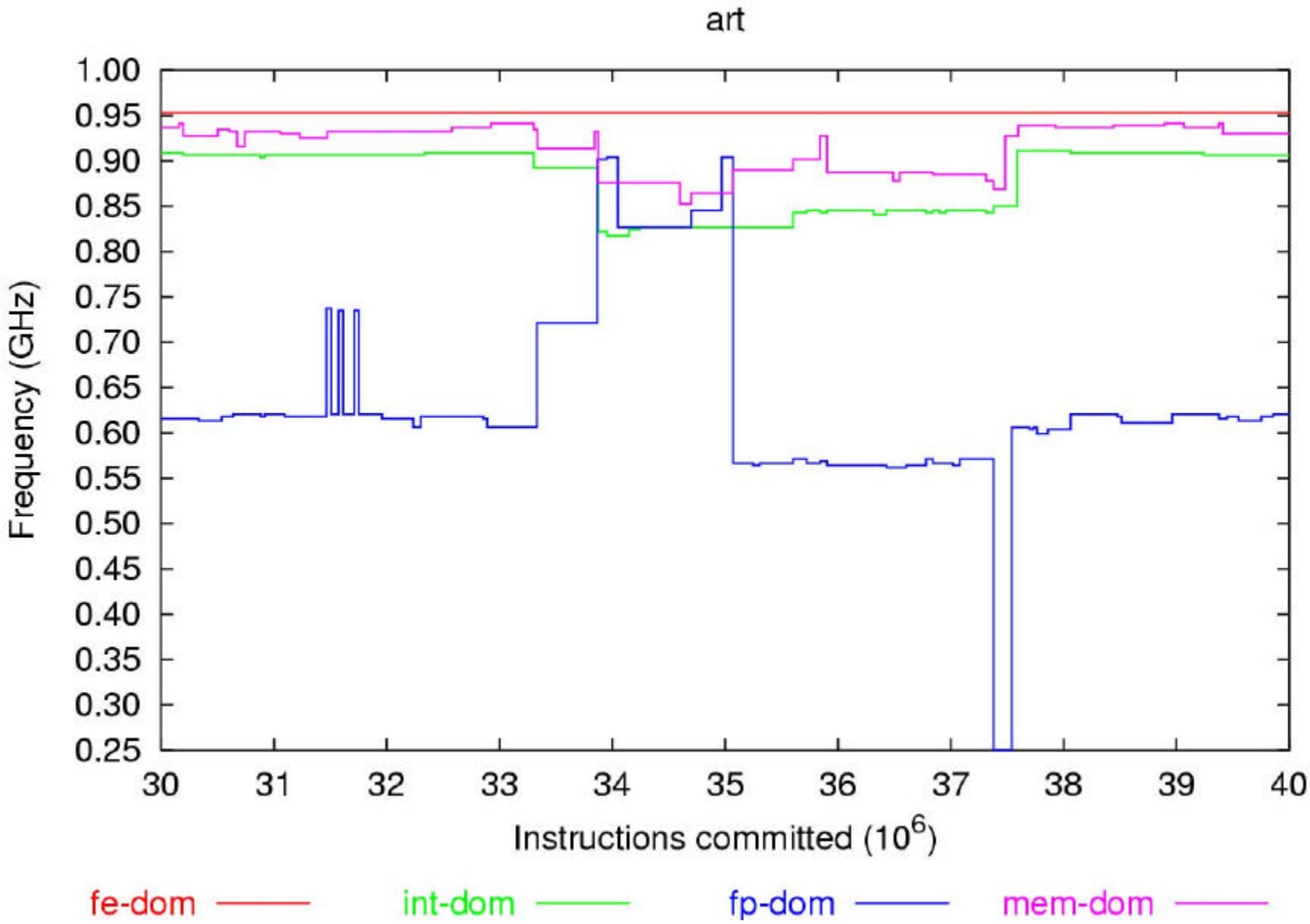
Mesa – Runtime Example



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**
- **Profiling tool under development**
  - Operates on major loops and functions
  - Uses shaker algorithm and call graph information to insert frequency control instructions into the application
  - Results so far come very close to the offline algorithm

Hardware based control: the attack/decay algorithm

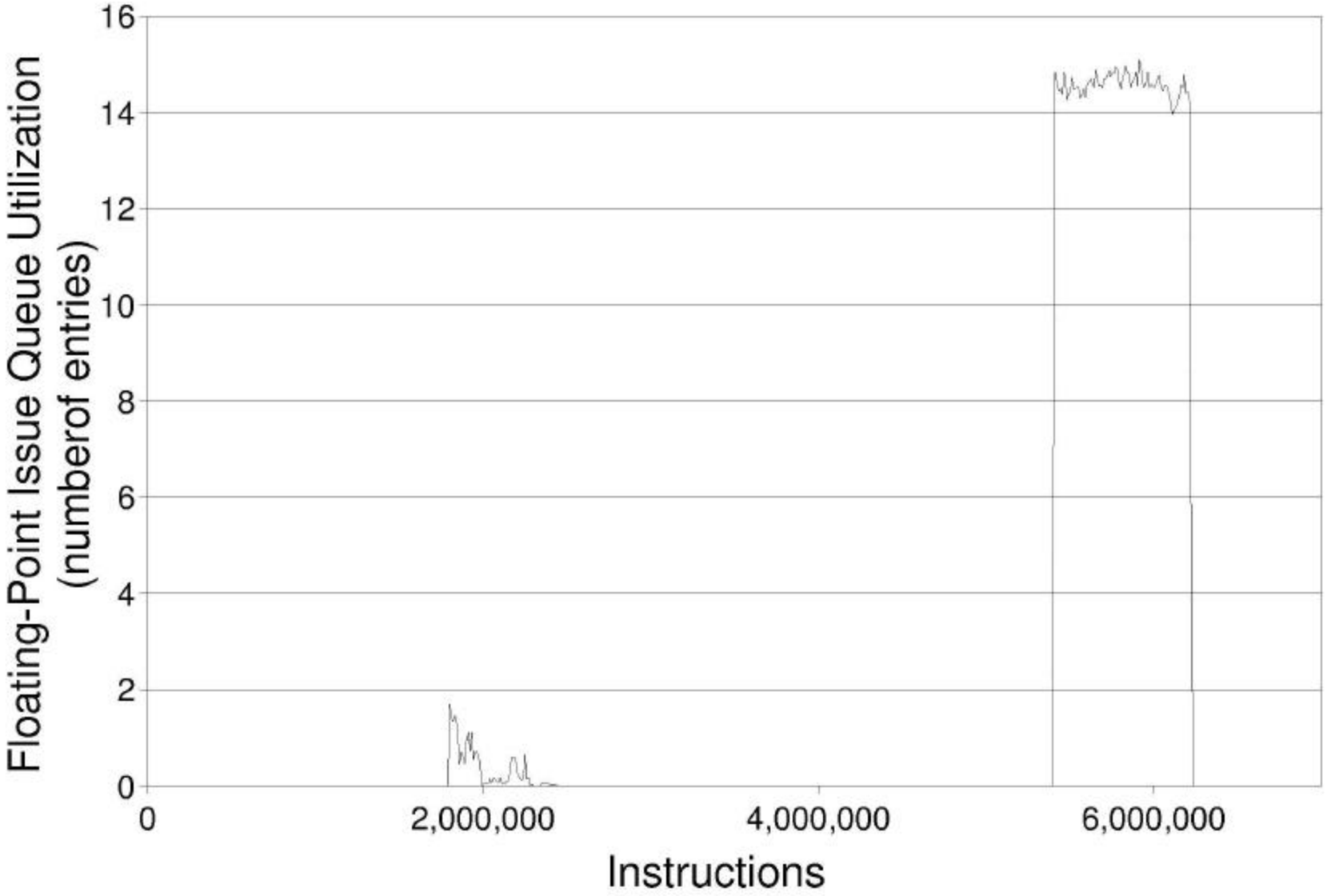
- Exploits correlation between changes in input queue utilization and domain frequency
- Each domain operates independently
- For each domain in each interval (10K instructions)
  - Check difference in input queue utilization compared to last interval
    - If decreased too much
      - If performance has not degraded too much,
        - decrease frequency (*attack*)
    - If increased too much
      - increase frequency (*attack*)
    - If at topmost (lowermost) frequency for too long
      - decrease (increase) frequency (*attack*)
    - Else
      - decrease frequency (*decay*)
- 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
- **Each of these parameters may be independently set within each domain**
  - We use same parameters in each domain for now

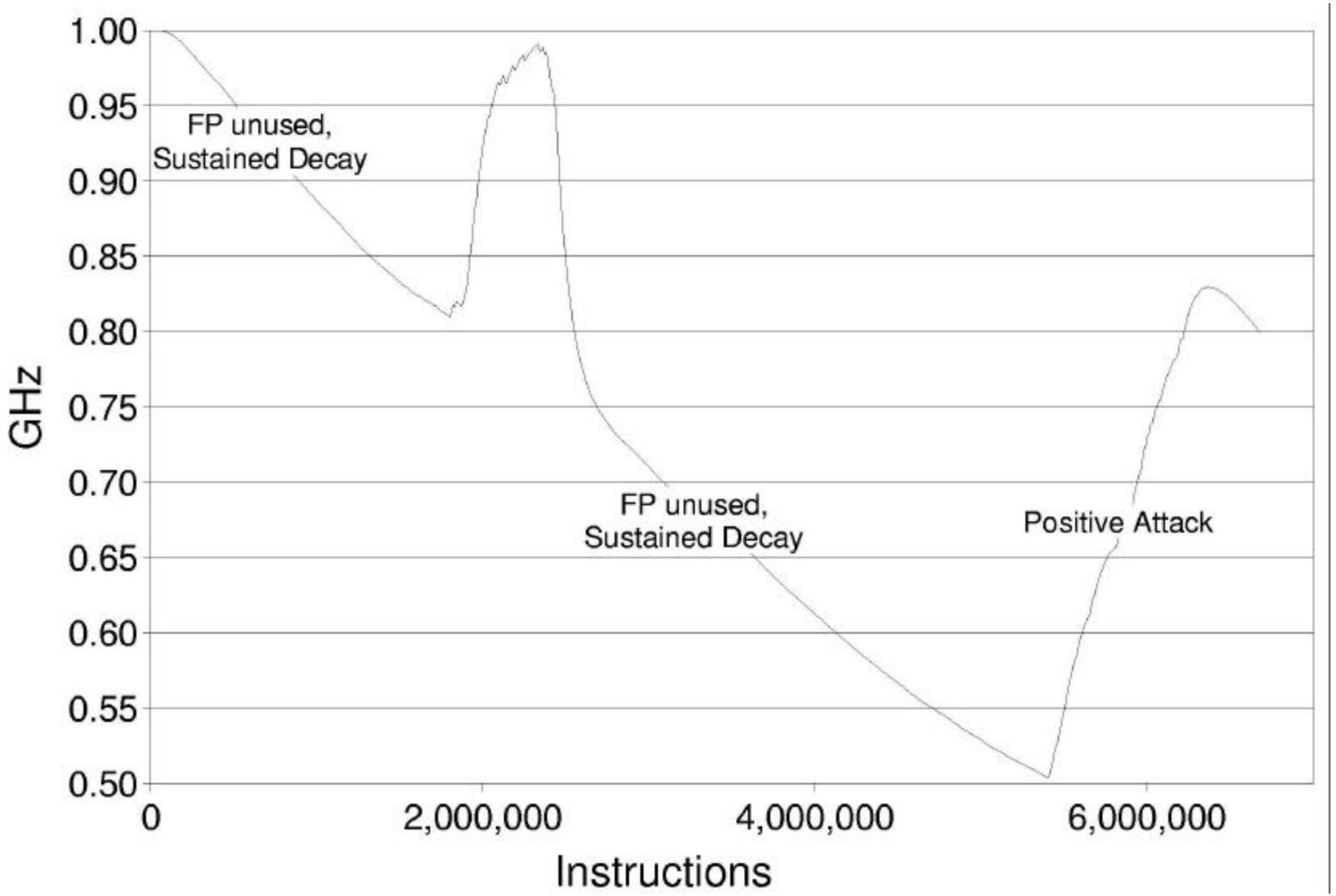
Attack/decay algorithm at work

- Changes in floating point queue utilization for epic decode



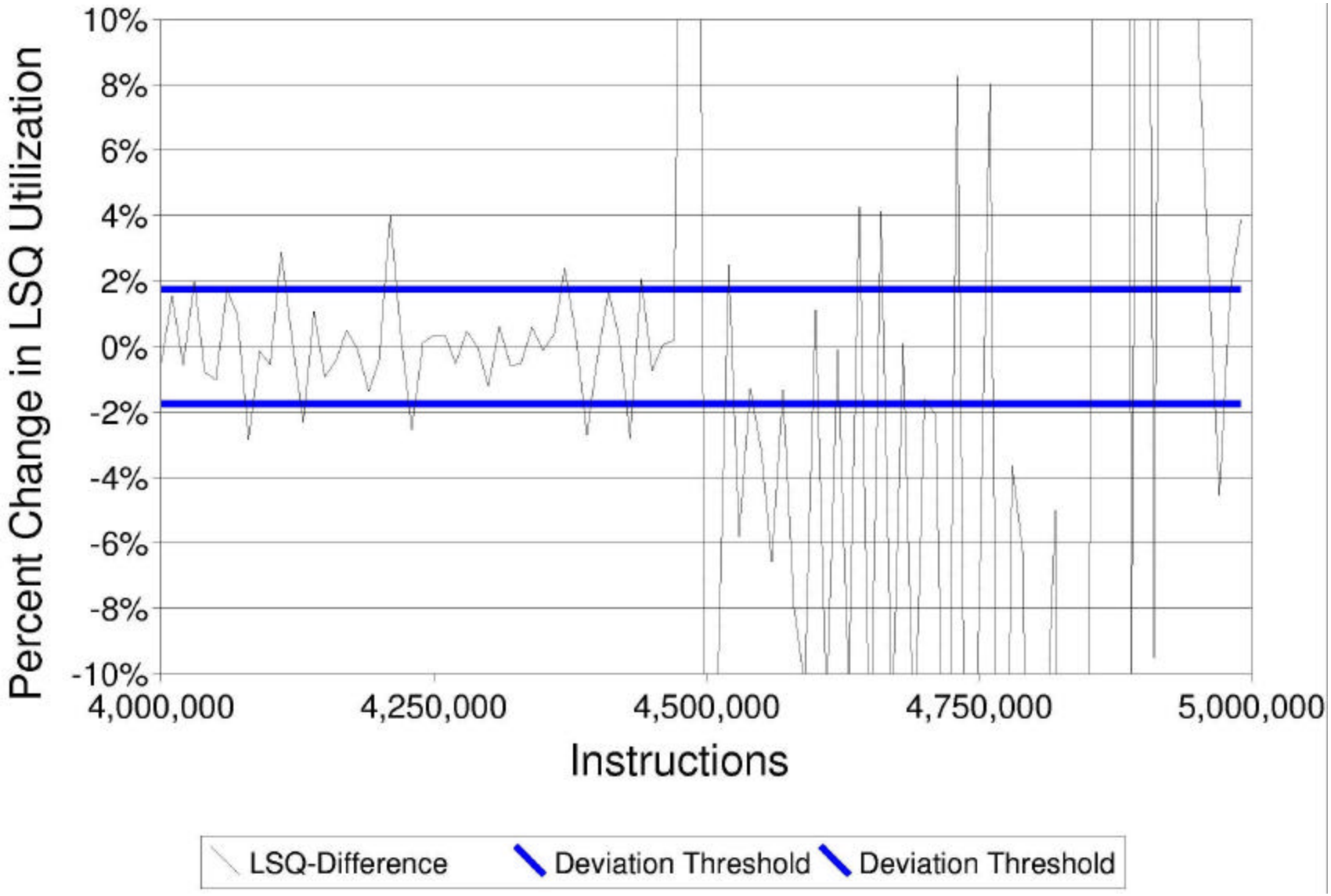
Attack/decay algorithm at work

- Changes in floating point frequency for epic decode



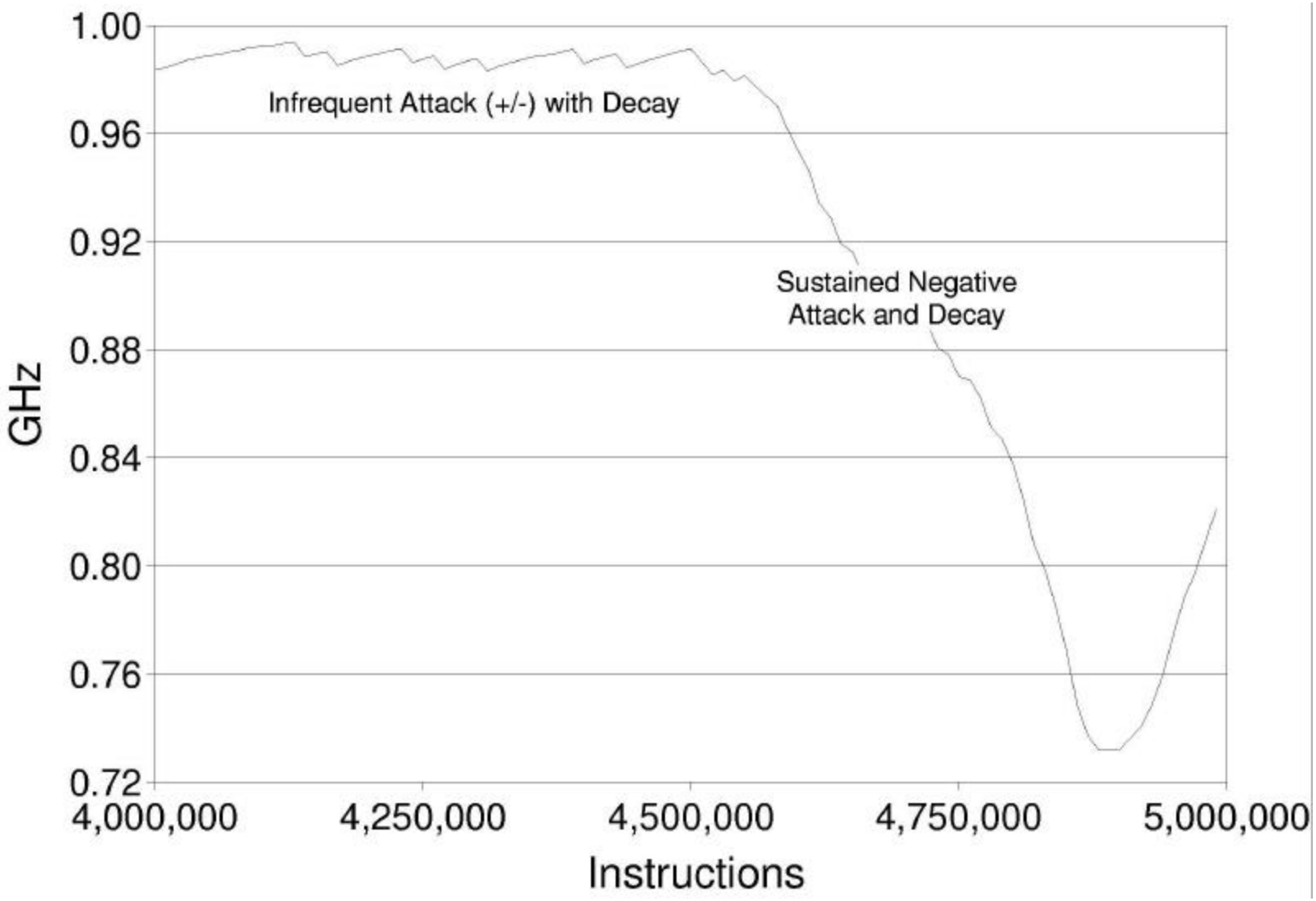
Attack/decay algorithm at work

- Differences in load/store queue utilization for epic decode

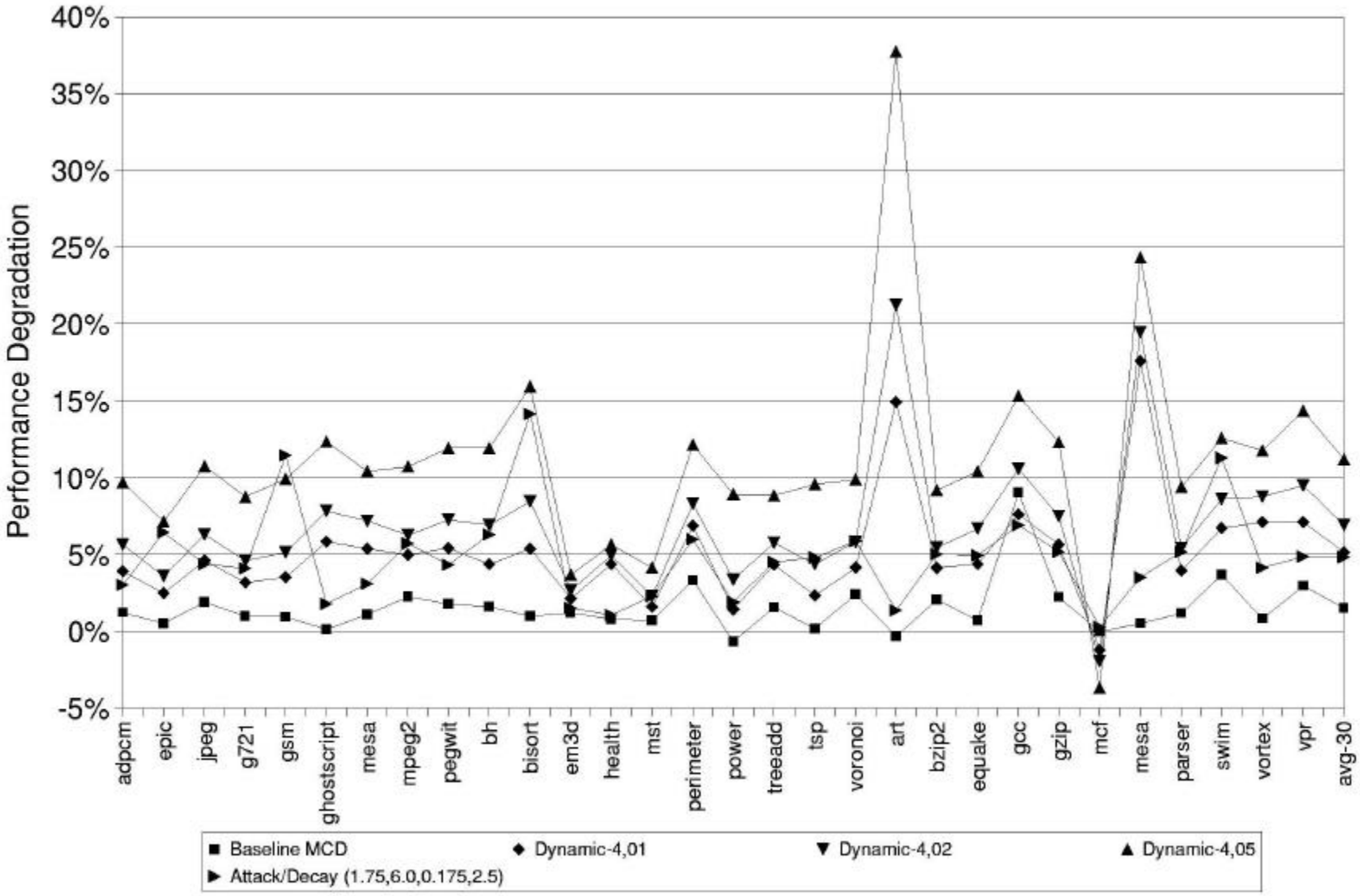


Attack/decay algorithm at work

- Changes in load/store frequency for epic decode

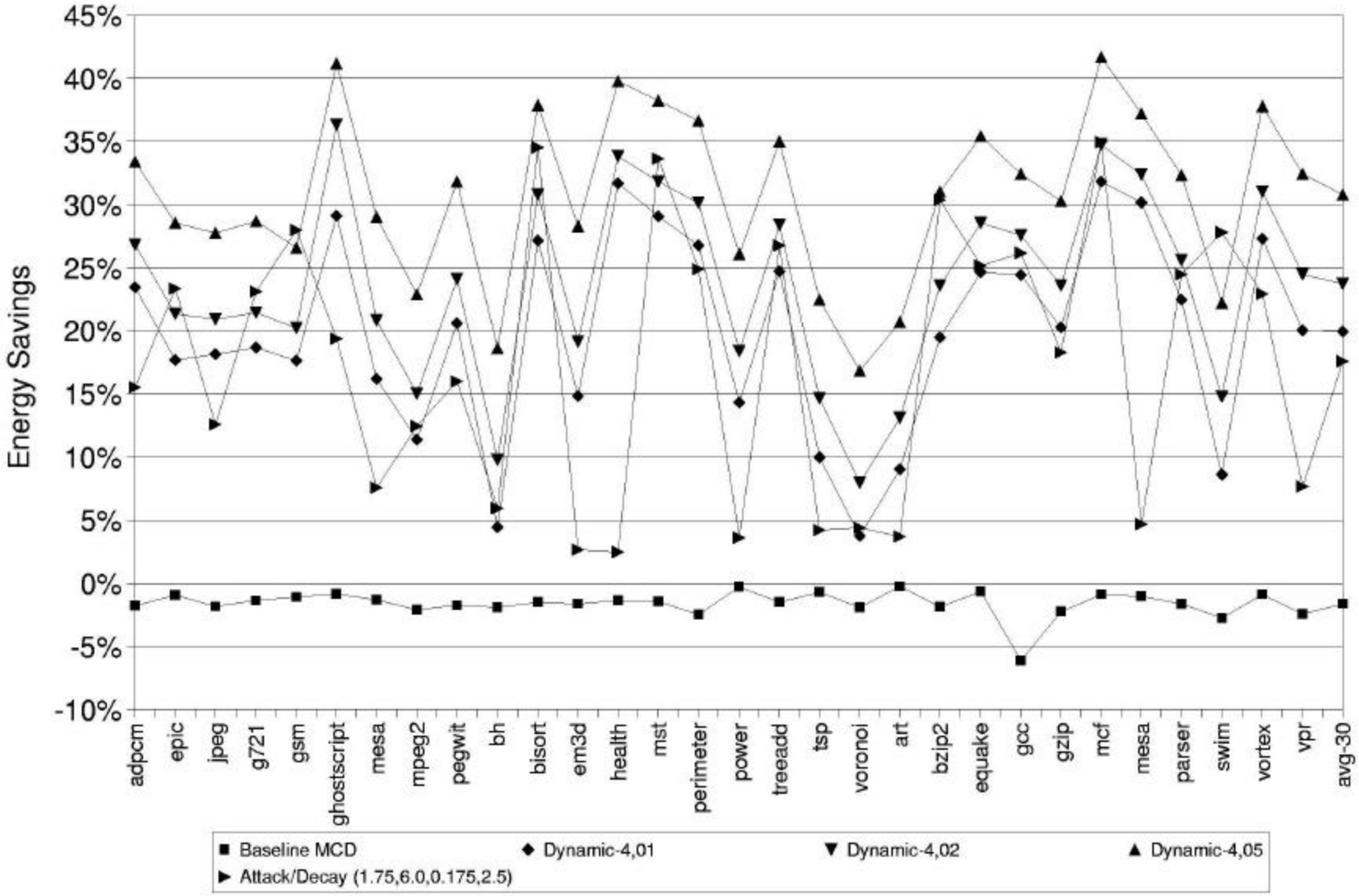


Performance degradation



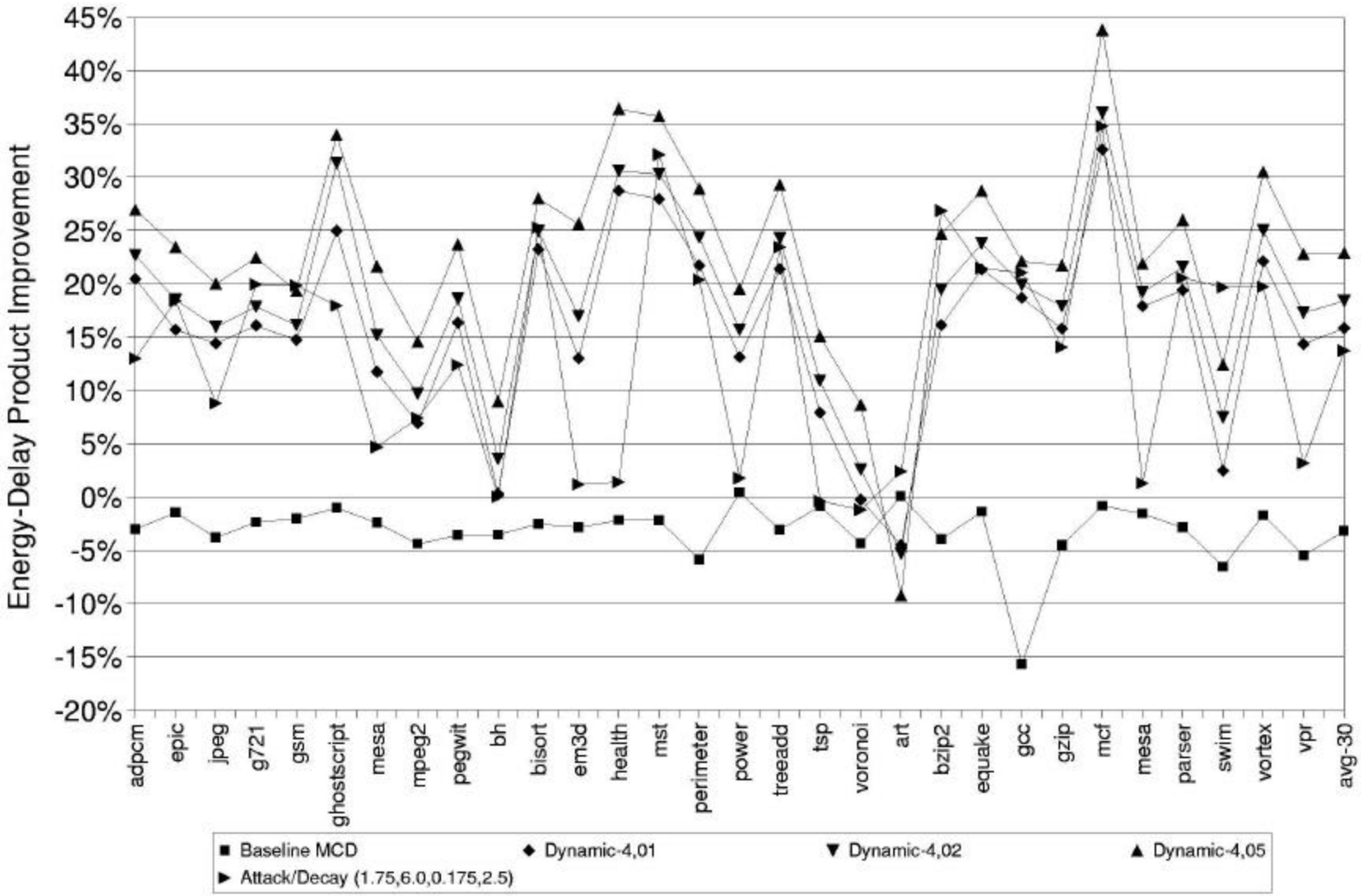
- **Same overall performance degradation as offline with 1% performance degradation target (Dynamic-4,01)**

Energy savings



- **Achieves 90% of the energy savings as offline with 1% performance degradation target (Dynamic-4,01)**

Energy-delay improvement



- **Achieves 90% of the energy-delay improvement as offline with 1% performance degradation target (Dynamic-4,01)**

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 90% of the energy savings of an offline algorithm with identical performance degradation

Future work

- Inter-domain interface design
- Choice of domains and boundaries
- Front-end control
- Performance optimizations
- Dynamic voltage gating

Future work: front-end control

- Front-end currently fixed rather than dynamic
  - 20% of the total energy
- Applying modified fetch-gating algorithms to the front-end
  - Every interval, monitor the average fetch rate and the average commit rate
  - Scale front-end frequency so that
    - Fetch rate = factor * commit rate
    - Factor is some value close to 1.0

Future work: MCD performance optimizations

- Opportunities to turn MCD performance degradation into a performance advantage
  - Exploit the higher frequencies possible in each domain
    - More domains?
  - Separately optimize the IPC/frequency tradeoff in each domain
    - Dynamically?

Future work: dynamic voltage *gating*

- May want to gate a domain voltage to save leakage
  - Alternative approaches, e.g., sleep modes
- Voltage scaling works best when work is “smoothed out” over a long period of time
- Voltage gating would work best when work is “clumped together” to introduce idle time
- Best combination of the two that optimizes energy-delay
- State saving or maintenance may be required

For More Info…

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