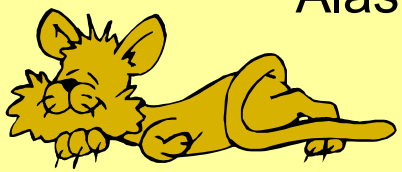
Lazy Retirement: A Power Aware Register Management Mechanism

Guillermo (Eli) Savransky
Ronny Ronen
Antonio Gonzalez

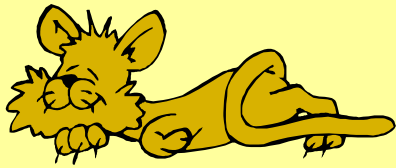
MRL - Intel Corp.

WCED – Workshop on Complexity Efficient Design
May 2002 – Anchorage Alaska



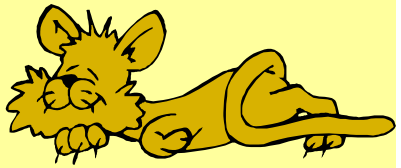
Intel Labs Research & Development

Agenda

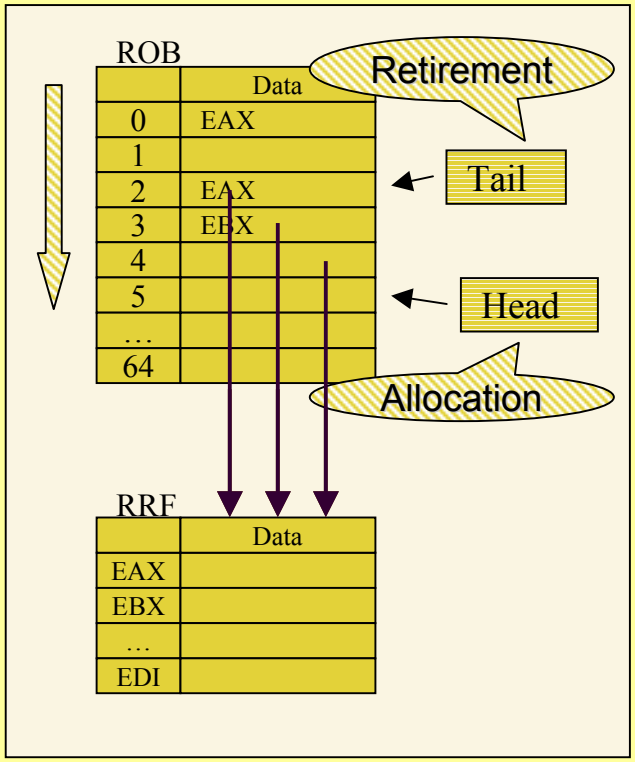


- Standard Retirement Algorithm
- Lazy Retirement
- Run Example
- Simulation results
- Summary

Background



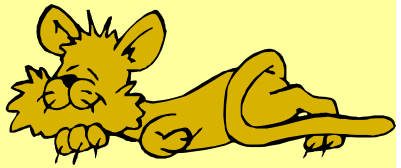
- P6 architecture:
  - Reorder buffer (ROB) and physical register file are the same logical structure.
  - Values produced by the retiring instructions are copied from the ROB to the real register file (RRF).
  - ROB entries deallocated on retirement.
- **This copy operation costs power.**



Motivation:

Reduce the number of copy operations without breaking the cyclic ROB structure.

Lazy Retirement: The Idea

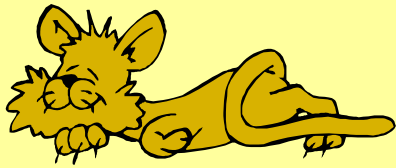


- When retiring a ROB entry, its value is *declared* as architectural state but not copied to the RRF.
- When the allocator needs a ROB entry, check if it is still part of the architectural state.
  - If it is, copy it to the RRF.
  - If it isn't, ignore.
- **No Performance Penalty!!!**

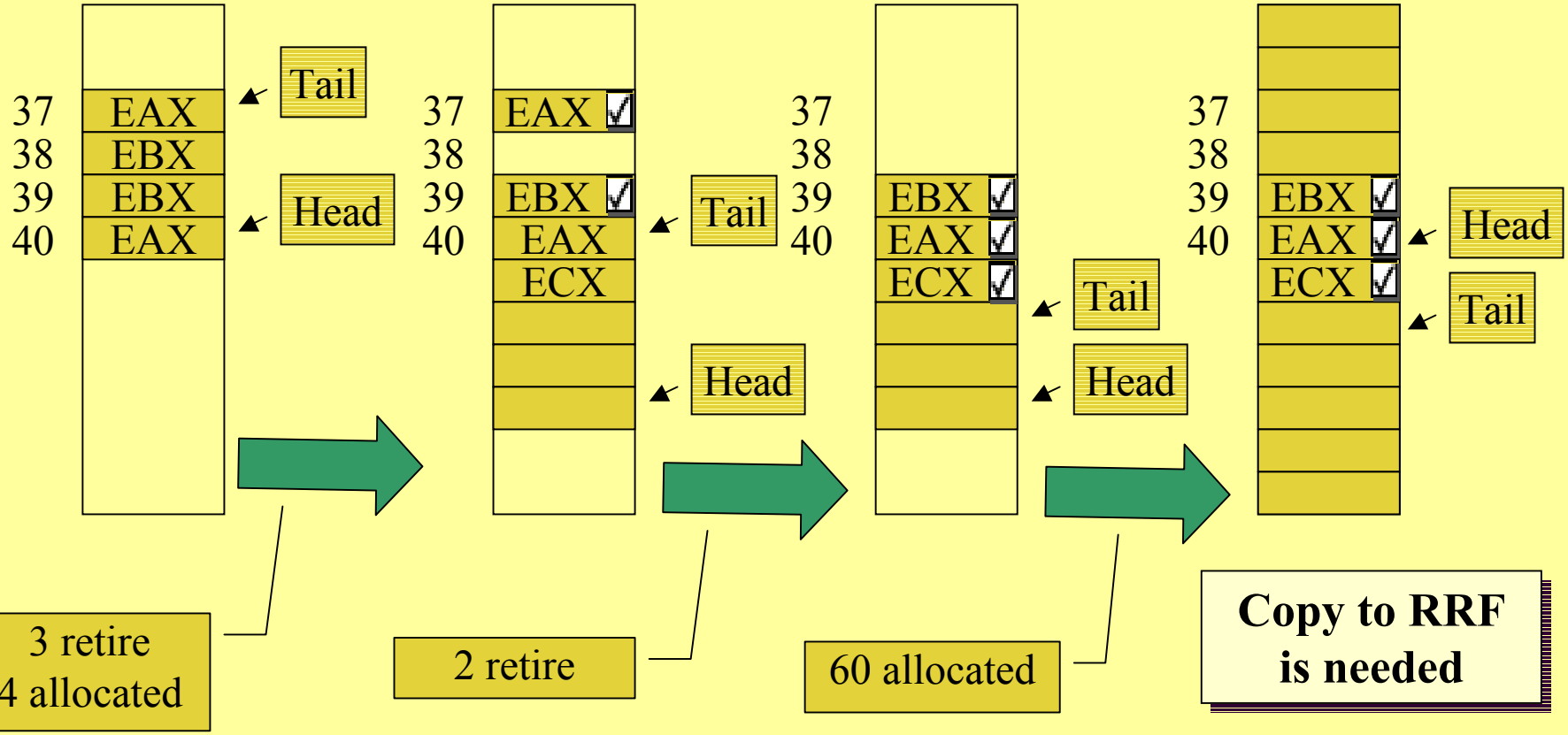
Standard Retirement:
Register **Deallocation** → Copy to RRF

Lazy Retirement:
Register **Reallocation** → Copy to RRF

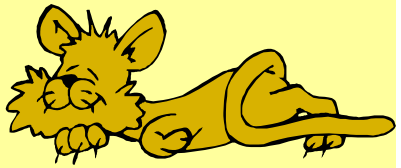
Example



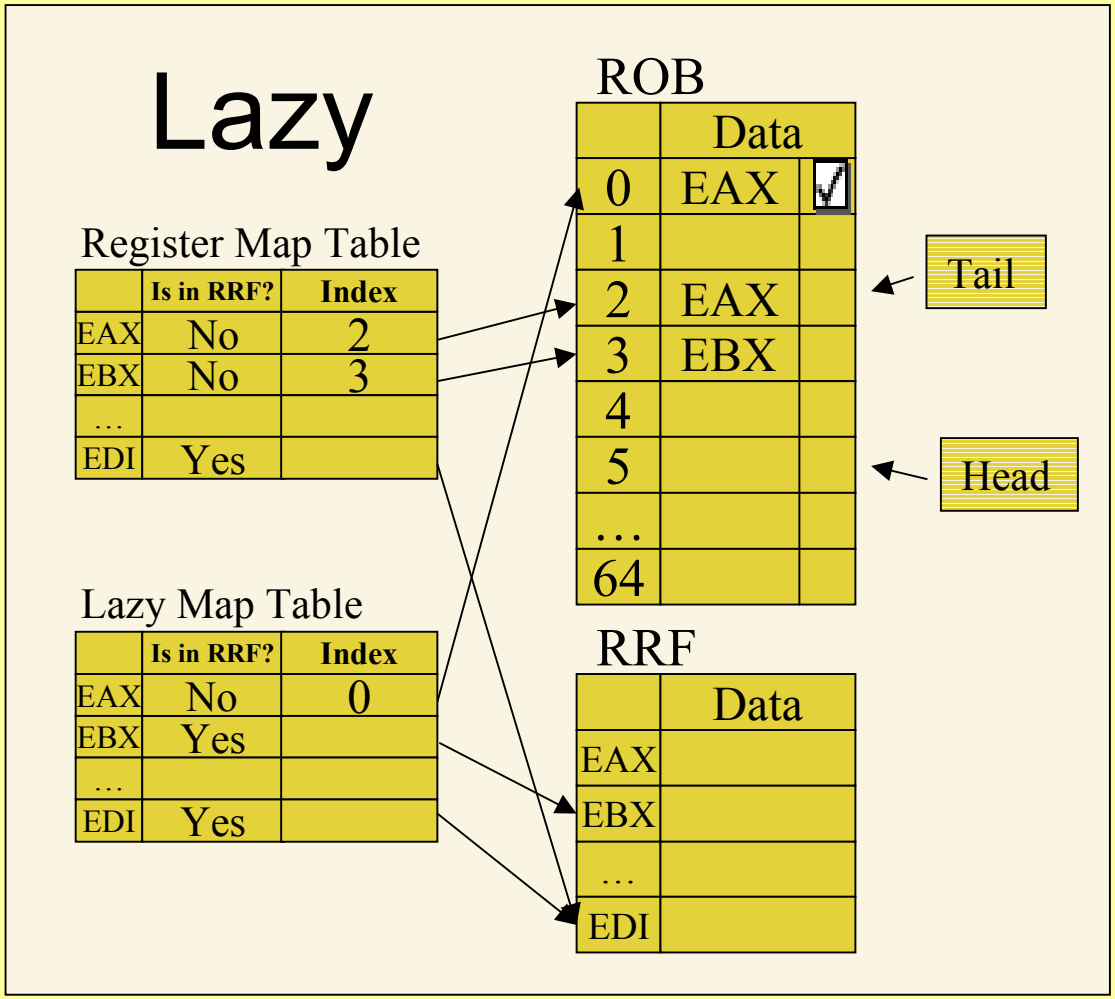
```
load eax ← [esp]
add ebx ← eax
and ebx ← 0xf
mov eax ← ebx
mov ecx ← 0x1
```



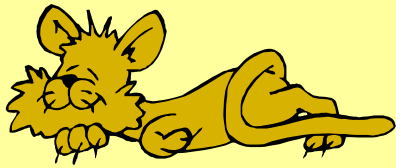
Implementation



- The **Lazy Map Table** remembers where are the retired registers.
- A data **valid bit** in the ROB marks the registers containing architectural state.

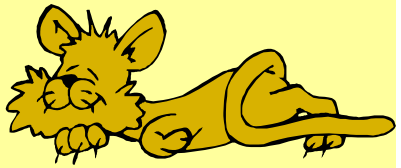


Algorithm

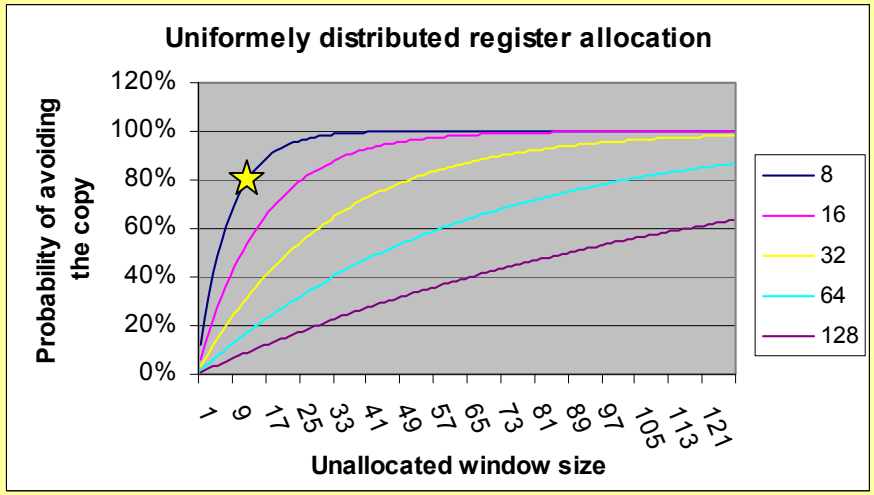


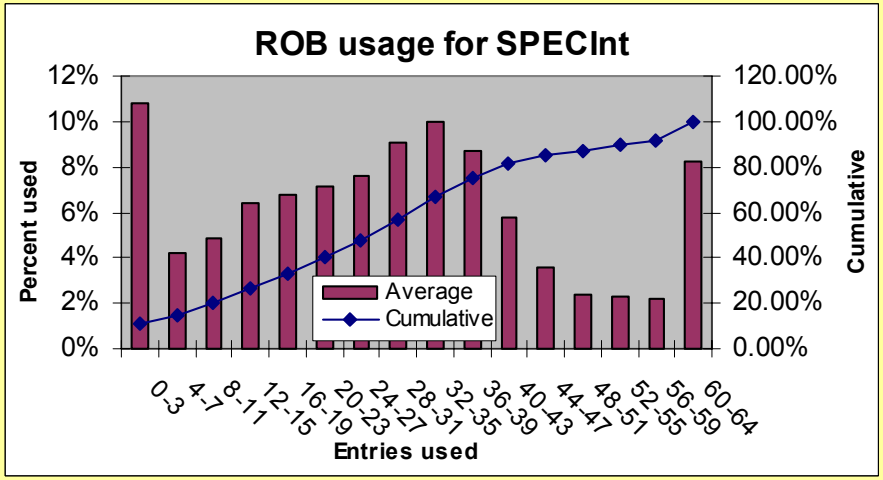
- The valid **data bit** in the ROB will be set if the associated entry contains an architectural register.
  - It will be set at retirement.
  - It will be reset when:
    - Another operation with the same architectural retires or
    - The register is copied to the RRF.
- The lazy map table will indicate where the architectural register is.
  - ROB entry or RRF.
  - It will be actualized at retirement and if the allocator forces the copying of the register to the RRF.
- On mispredictions or exceptions, the lazy map table is copied to the renamer.

Why It Works?

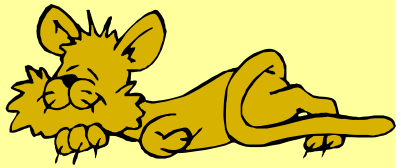


- ROB size tuned for worst cases:
  - Cache misses.
  - Long latency dependency chains.
- Most of the data copied to the RRF is overwritten shortly after the transference.



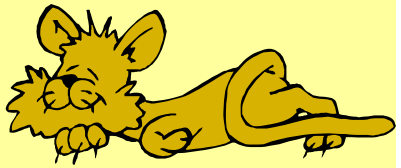


Simulation Setup

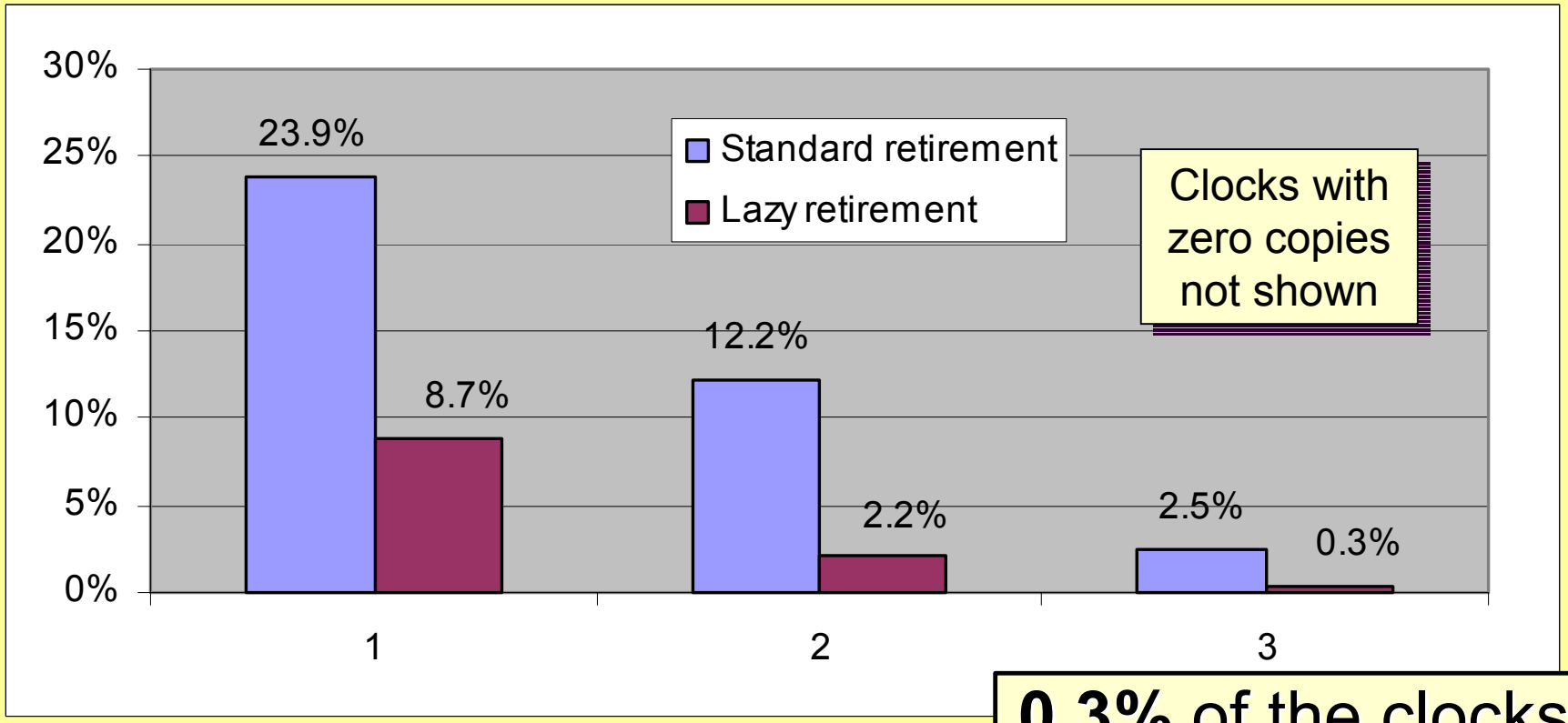


- Used an internal performance simulator.
- Simulated processor details:
  - IA32 architecture.
  - P6-like microarchitecture.
  - Separated ROB and RRF.
  - 64 ROB entries.
- A modified CACTI tool used for power estimations.
- Workload:
  - SpecInt2000
  - Winstone99
  - SYSmark98
  - Other multimedia traces.

Simulation Results



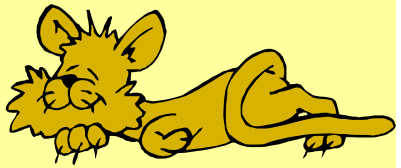
- Retirement ports usage per cycle.



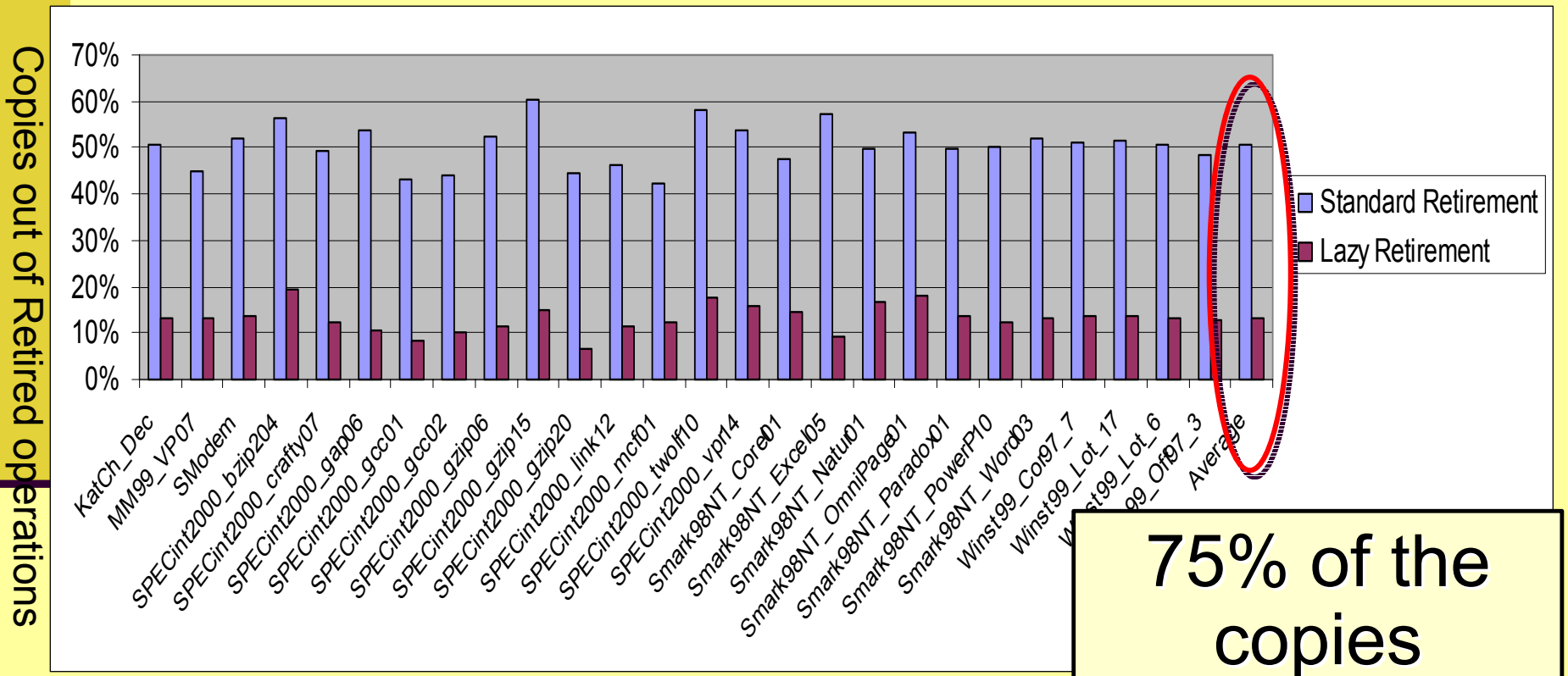
Improves clock gating when no port required:
P6:61%, Lazy: 88%

0.3% of the clocks three ports are used!

Simulation Results

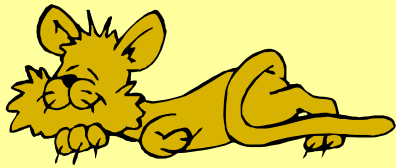


- The number of copies from the ROB to the RRF copies per operation.

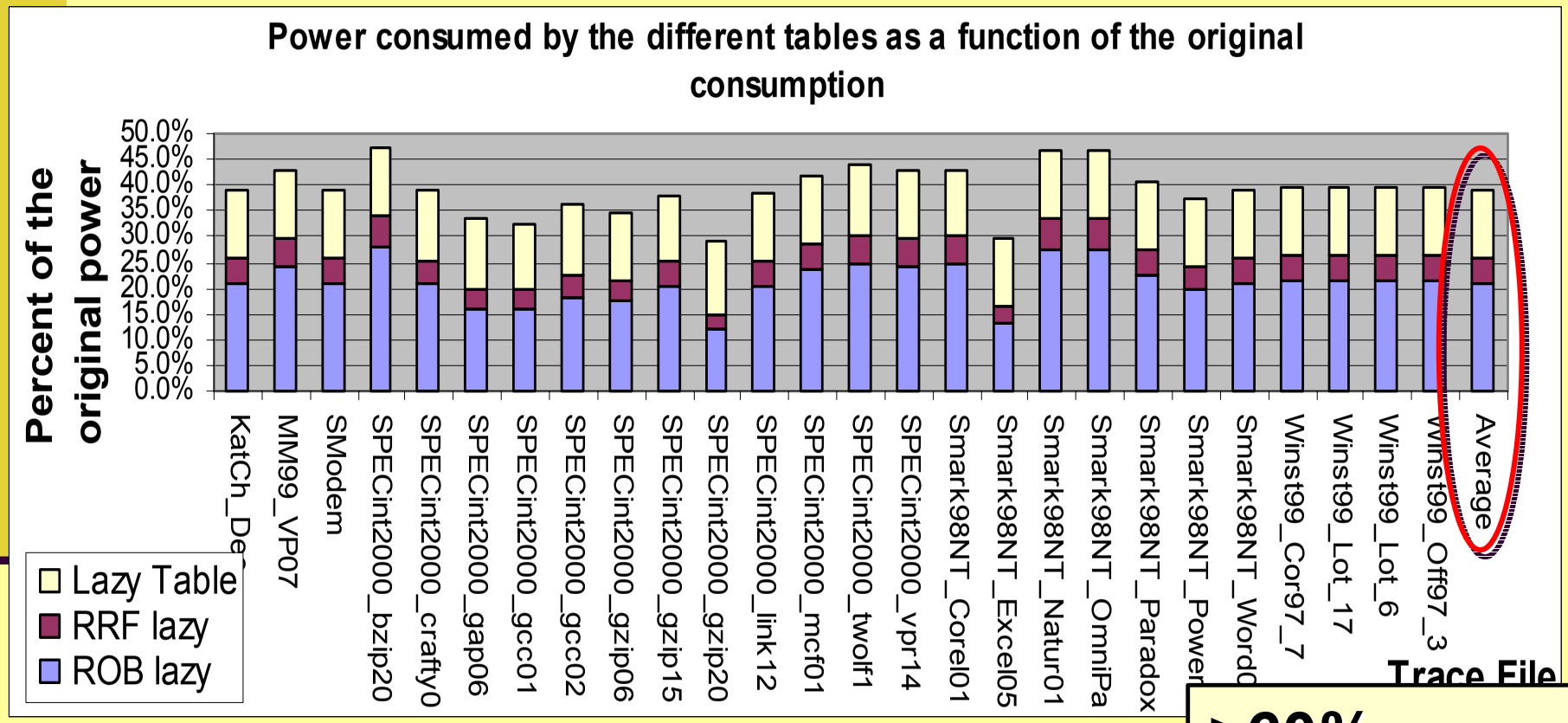


75% of the copies eliminated!

Power Modeling



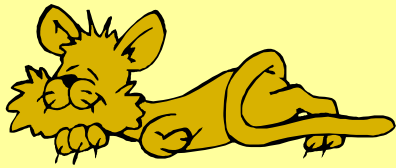
- Power reduction compared to original



- Lazy table use 13% of the original retirement power.

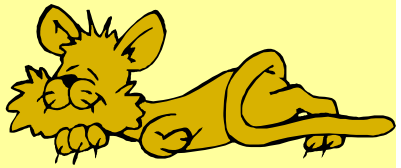
>60% power reduction!

Considerations



- ROB + RRF is about 7% of total processor power.
- Renamer power changes are not modeled:
  - ☺ Number of updates greatly reduced.
  - 😐 Misprediction recovery is not thermally relevant.
- Can be used to reduce the number of ROB, RRF and renamer physical ports used for retirement.
  - High power reduction.
  - Have performance penalty (trade off is architecture dependent)
- In an unified register file with no RRF (as in the P4 architecture) the management logic is more expensive than the P6 retirement.

Summary



- Shown a method for reducing the copies of data from the physical to the architectural register file.
  - Eliminates about 75% of the copies.
  - Can be implemented without performance penalty.
- The power reduction is much higher than the overhead.
- Balance algorithm complexity to reduce power:
  - Too dumb → lots of work → High power.
  - Too smart → lots of control logic → High power.
  - In general:
    Balance added capacitance with lowered activity