Dynamic Controlled Resource Allocation for SMT Processors

Francisco Cazorla, Enrique Fernandez
Alex Ramirez, Mateo Valero

Universitat Politecnica de Catalunya
Universidad de Las Palmas de Gran Canaria
HiPEAC European Network of Excellence

Compiler & Architecture Seminar
Haifa, IL
December 2004

Motivation

- Threads in an SMT processor share the resources
  - Better utilization leads to higher throughput
- but ... Threads in an SMT processor also compete for the resources
  - If one thread uses them all, the other threads stall

- Static resource partitioning
  - Prevents resource monopolization
  - but ... causes resource waste
- Dynamic resource partitioning
  - Resources are allocated first-come, first-served
  - Instruction fetch policy determines who enters the processor
    - And so, who arrives first

Talk outline

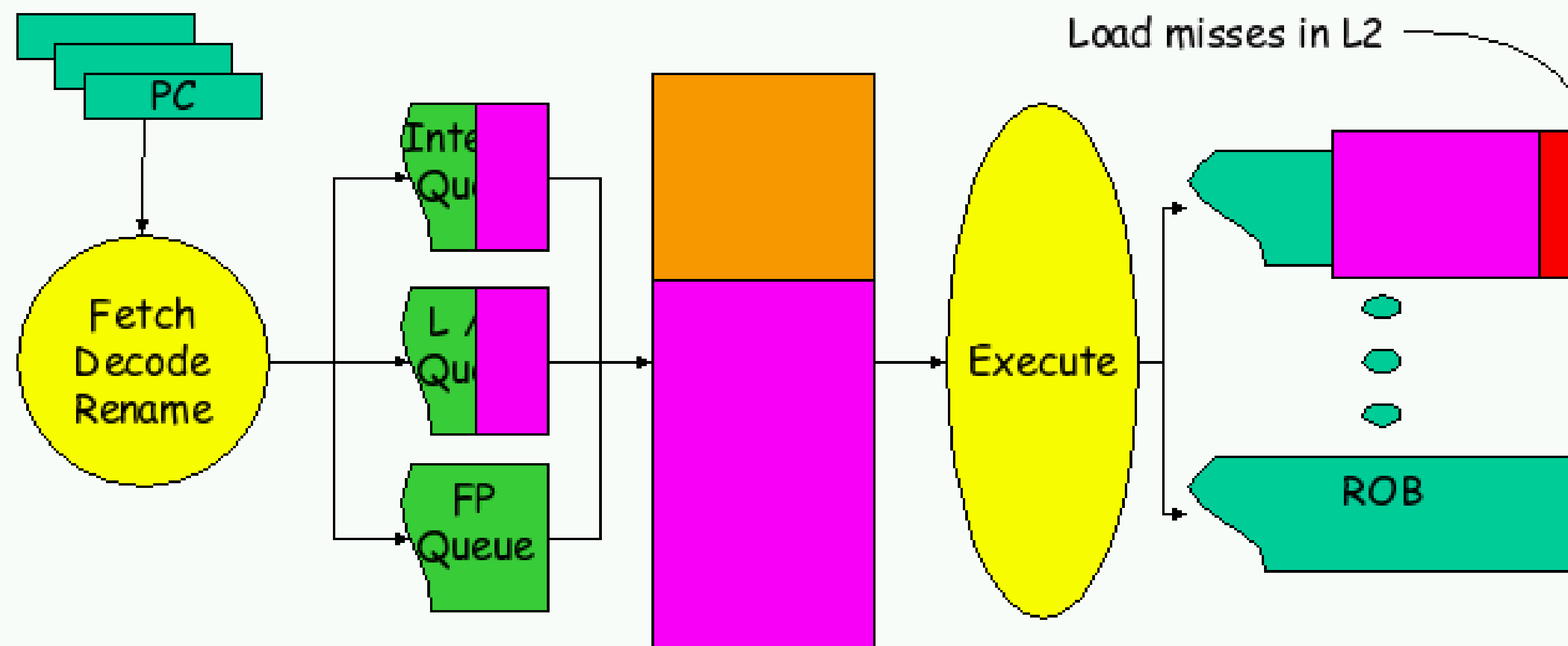
- Motivation
- Dynamic resource allocation policies
  - Instruction fetch policies
- Explicit resource allocation policies
  - DCRA
- Performance evaluation
- Summary & Conclusions

Fetch policies

- ICOUNT
  - The thread with fewer instruction in the pre-issue stages is given higher fetch priority
    - It is assumed to run faster, provides higher throughput
- Fetch priority is assigned using only an indirect resource usage indicator
  - Fewer instructions in the front-end does not mean fewer resources used

The problem ...

- Long latency events may cause resource monopolization
  - The other threads do not progress either



Implicit resource allocation

- Additions to ICOUNT to prevent this problem
  - Use indicators that a thread may abuse the resources
  - Prevent the thread from monopolizing the resources
- Some examples …
  - Data gating / Predictive data gating
    - Detect / Predict a thread missing in the L1 cache
    - Stop fetching from the thread until miss is resolved
  - STALL
    - Detects a thread missing in the L2 cache
  - FLUSH
    - Detects a thread missing in the L2 cache
    - Squashes the missing thread to de-allocate its resources
  - FLUSH++
    - Combines STALL and FLUSH based on resource pressure

What is the real problem?

- Only indirect approaches are used
  - Assume a thread will cause problems on an L2 miss
    - STALL may be too late -> resource abuse
    - FLUSH may be excessive -> resource under-use
- The real problem is not the L2 miss ...
  - ... it's the excessive resource allocation !

So?

- Monitor resource usage explicitly
  - Instead of watching for L2 misses

Windows is not the answer. Windows is the question, and the answer is NO.
Fight the real enemy

Explicit resource allocation

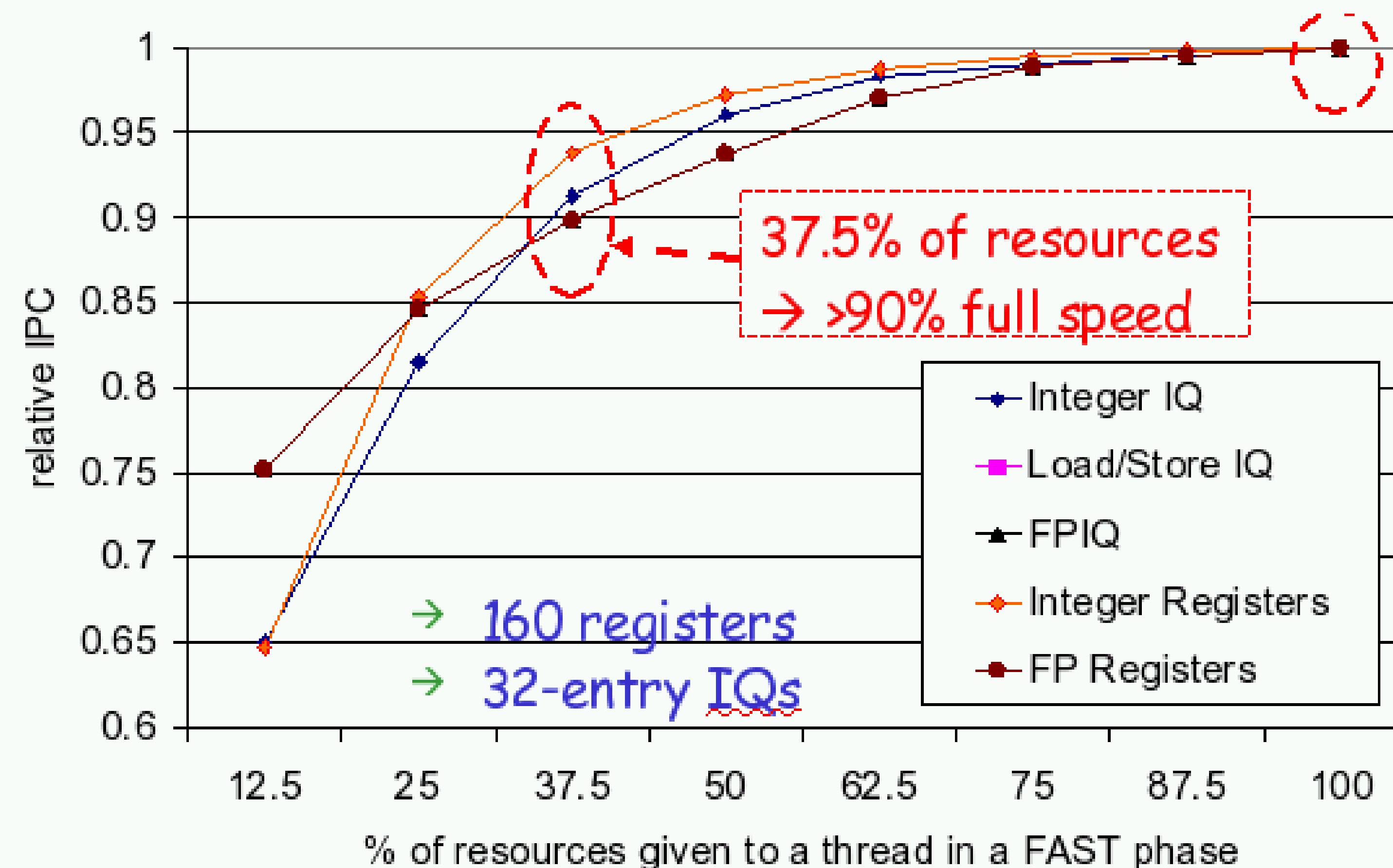
- Determine the available resources
- Determine the resource needs for each thread
  - Use explicit indicators o resource usage by each thread
- Distribute resources among the threads
  - Which threads will use the resources more efficiently?
- Explicitly enforce the resource allocation
  - If a thread tries to exceed its allocation, prevent it

Dynamic Controlled Resource Allocation

- Determine which threads are using each resource
  - Not all threads use all resources (Floating Point, SIMD)
- Determine the resource needs of each thread
  - Threads experience phases:
    - Fast phase (high ILP) - when there are no cache misses
    - Slow phase (low ILP) - where there are L1 misses
- Allocate resources to threads
  - Fast threads give part of their share to the slow threads
    - The slow threads can make more progress ...
    - ... and overlap more cache misses
- Prevent any htread from eceeding its allocation
  - Stall the thread until resources are made available

Fast thread resource usage

- A thread in a Fast phase can run at almost full speed with fewer resources
  - Can give the extra resources to a Slow thread

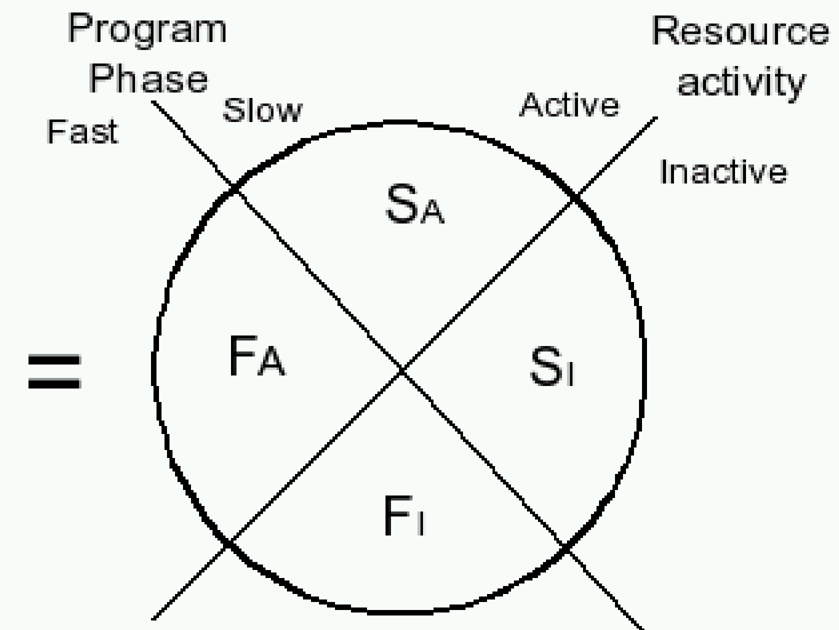


The idea behind DCRA

- To allow Slow threads to use resources from Fast thread s
  - It seems counter-intuitive
  - We do this in a controlled way, as Fast threads are not heavily affecte d
    - But Slow threads may obtain substantial benefits
      - Higher opportunity for out-of-order execution
      - Higher memory parallelism

Allocating resources to threads

- Determine which threads do not use a resourc e
  - These give all their share to "active" threads



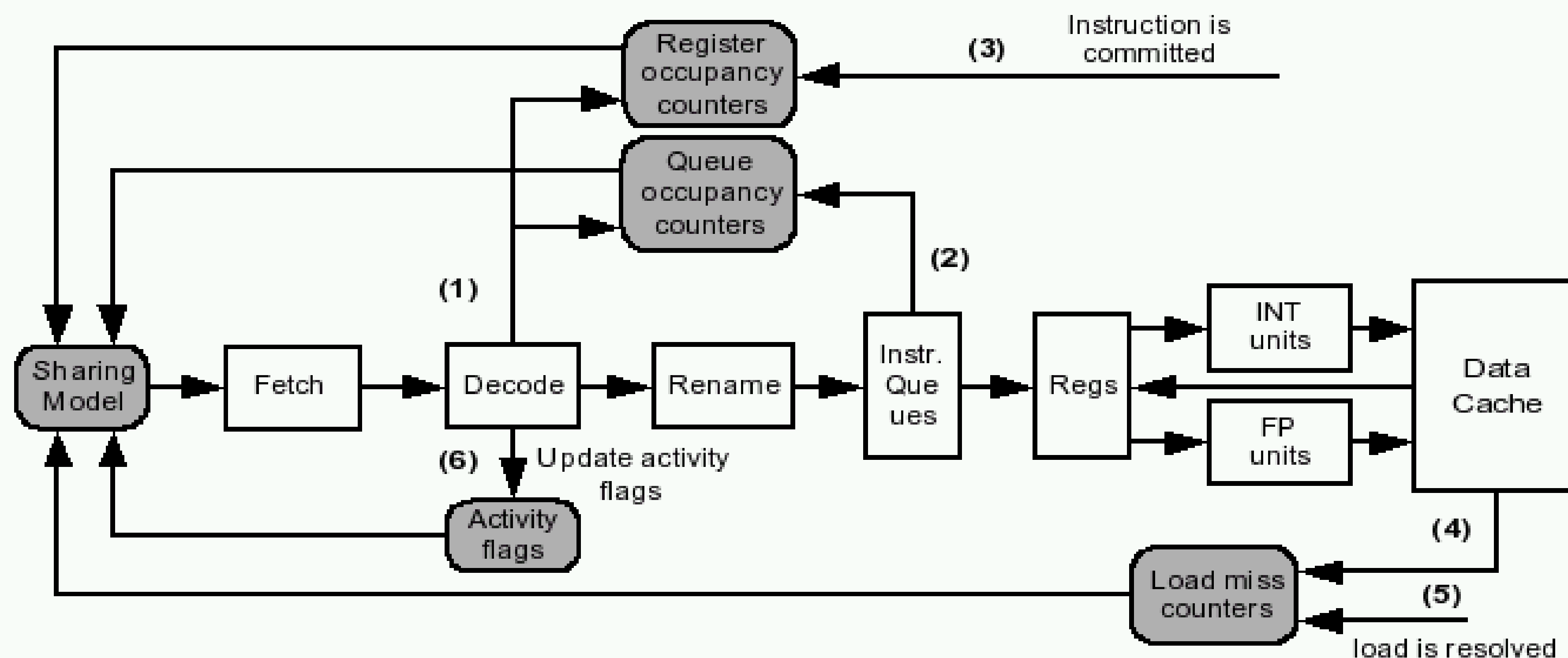
$$E = \frac{R}{F_A + S_A}$$

- Fast threads give part of their share to Slow threads
  - Introduces a sharig factor "C"

$$E_{slow} = \frac{R}{F_A + S_A}(1 + C * F_A)$$

DCRA Implementation

- 8 counters per thread:
  - Resource usage counters: 3 for IQs, 2 for registers
  - Activity counters: 1 for FPQ, 1 for fp registers
  - 1 additional counter to track L1 data cache misses
- Sharing model logic for each resource

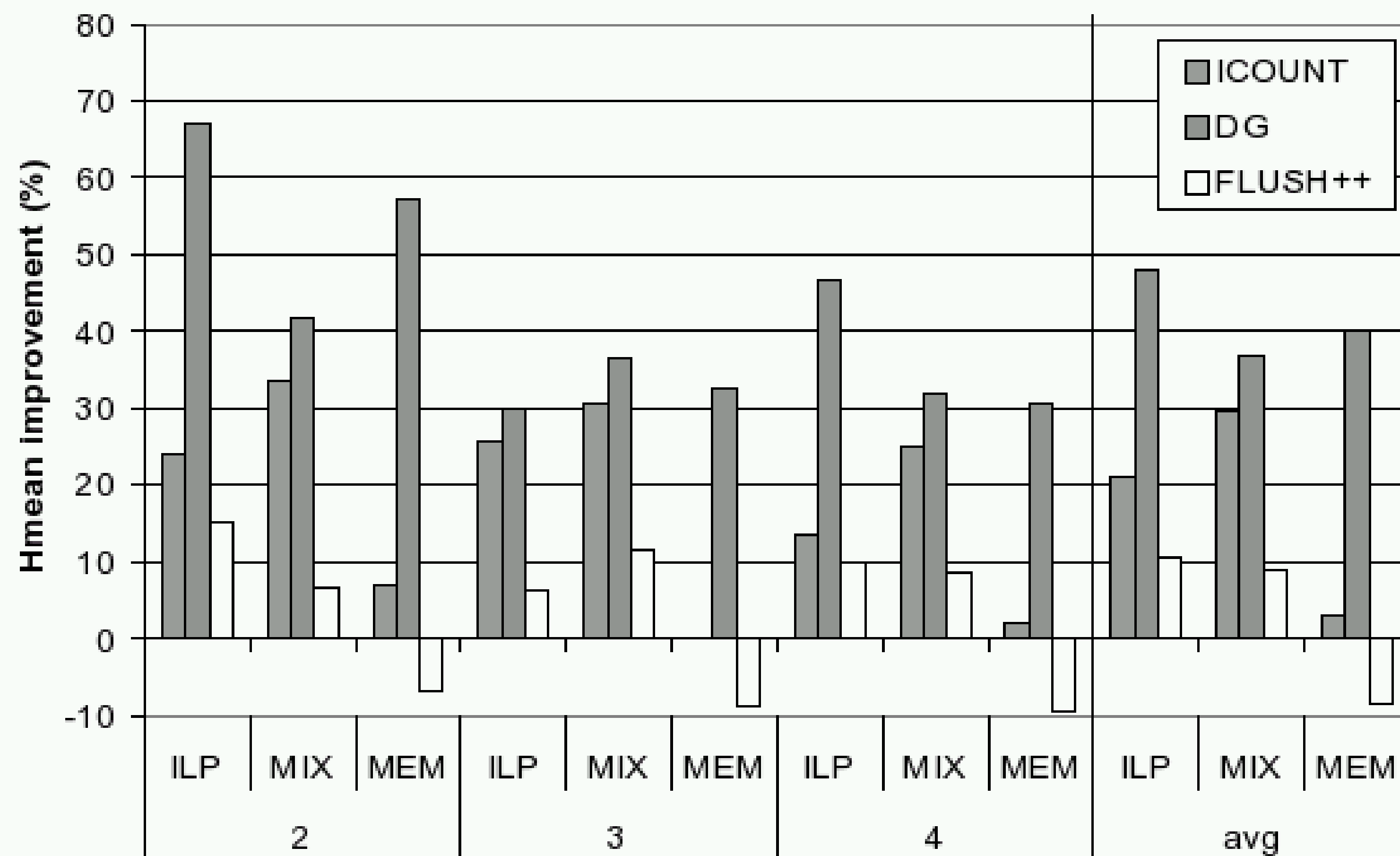


Performance evaluation

- We use 3 workload types: ILP, MIX and MEM
  - ILP: threads with an L2 cache miss rate lower than 1%
  - MEM: threads with an L2 cache miss rate higher than 1%
  - MIX: mixture of both
- Workloads with 2,3 and 4 threads
- Main processor and cache parameters:
  - 352 physical registers, 80 entry issue queues
  - 512-entry (shared) ROB, 6 int, 3 fp, 4 ld/st FUs
  - 64Kb instr. & data cache. 1 cycle access
  - 512Kb L2 cache. 20 cycle access
  - Memory latency: 300 cycles

Throughput & fairness results

- DCRA improves all other policies
  - Except FLUSH++ for the MEM workloads
  - Average improvement 18% vs Icount, 41% vs DG, 4% vs Flush++
- Throughput results follow the same trend



DCRA and MEM workloads

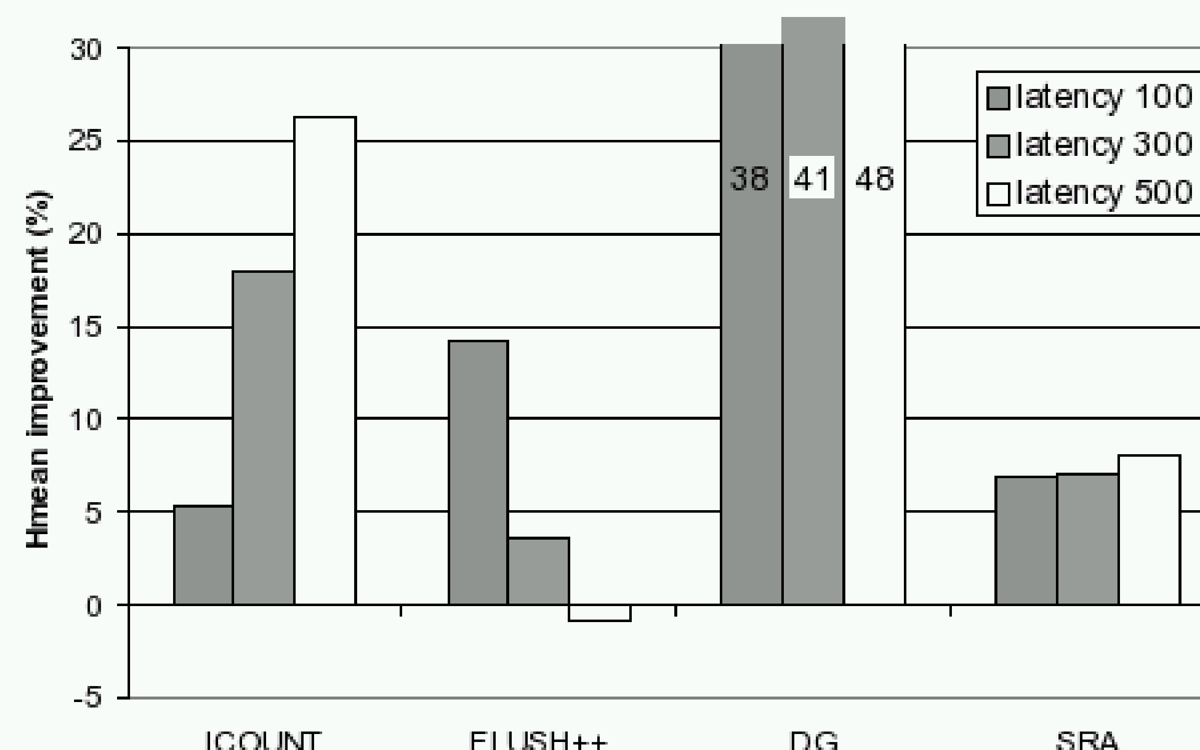
- DCRA gives more resources to threads in Slow phases
  - DCRA 18% more overlapping L2 misses than Flush++

- DCRA heavily affected by the mcf benchmark:
  - DCRA 31% more overlapping misses than Flush++
    - The IPC of mcf is increased
  - IPC of mcf is very low, so this increment is almost not visible
    - And the performance of the other threads is affected

- Future work: prevent these degenerated cases

Sensitivity to memory latency

- Improvements vs others increase with higher latencies
  - Except Flush++
- Flush++ reduces performance advantage of DCRA ...
  - ... but increases the front-end activity by 108% for 300-cycle latency and by 118% for 500-cycle latency due to flushed instructions



Summary & Conclusions

- Resource allocation is key to SMT performance
- Fetch policies are indirect reource allocation policies
  - Use indirect indictors of resource usage
    - May be too late
    - May react in too strong ways
- Explicit resource allocation outperforms fetch policies
  - Use explicit indicators of resource usage
  - Perform an explicit resource allocation to threads
  - Enforce that alocation
- DCRA is an example implementation of explicit resource allocation

Open topics

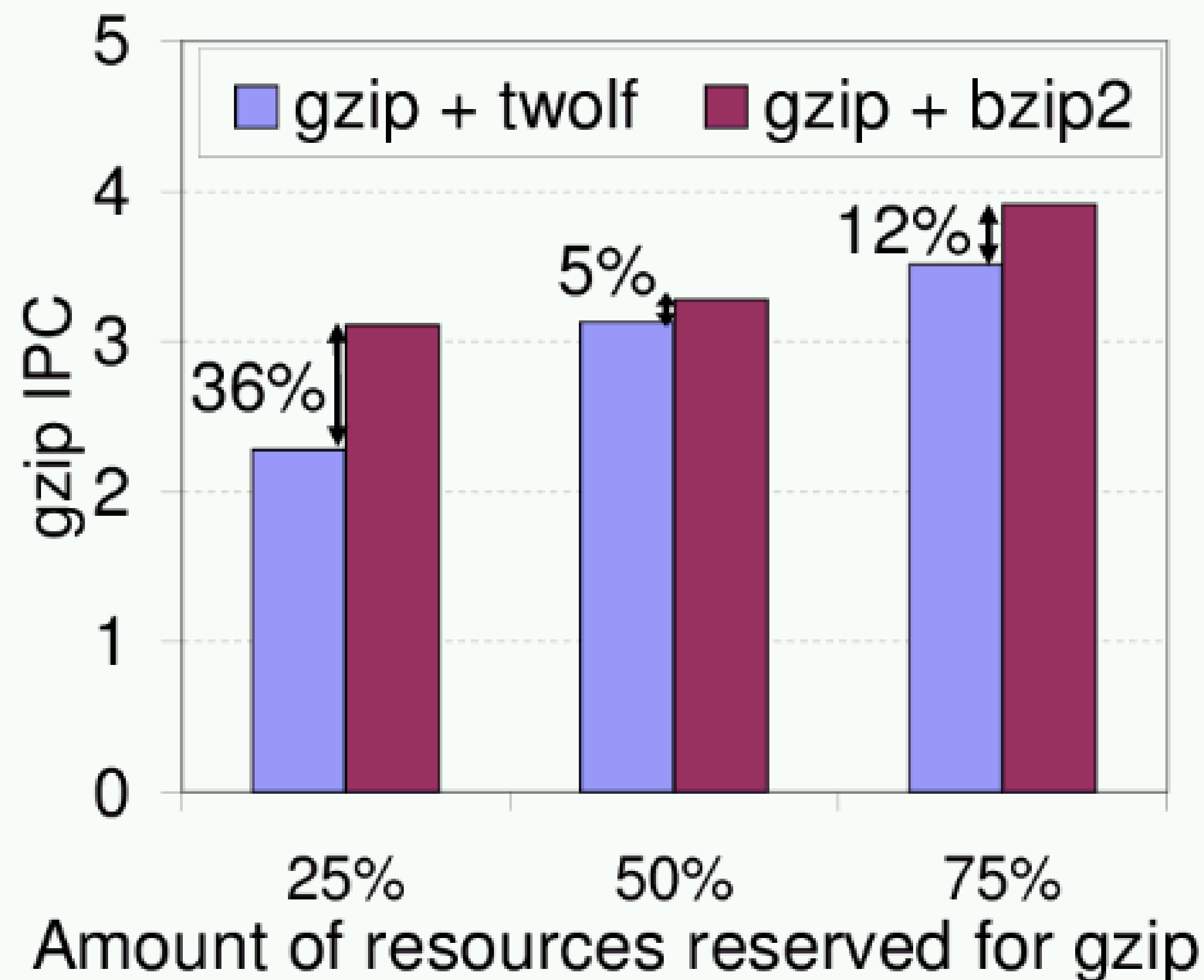
- There are still many open questions:
  - Resource allocation must be tuned to the available resources
    - More resources -> more flexibility
  - Resource allocation must be tuned to the memory latency
    - Higher latency -> resources take longer to be released
  - Some shared resources have not been considered
    - L2 cache ...
  - Classifying threads based on L1 misses may not be the best optio n
    - Use explicit phase indicators?
- Better explicit resource allocation policies are possible

Further uses of explicit resource allocation

- Implementing Quality of Service in SMT
  - Requirement: run a thread at a % of its maximum IPC
- Sample phase
  - All resources given to the target thread
    - Other threads are stopped
  - Estimate local IPC for the thread
- Tune phase
  - Explicitly allocate/de-allocate resources to achieve target IPC
    - Dynamically adjust the resource allocation
  - All remaining resources allocated to other threads

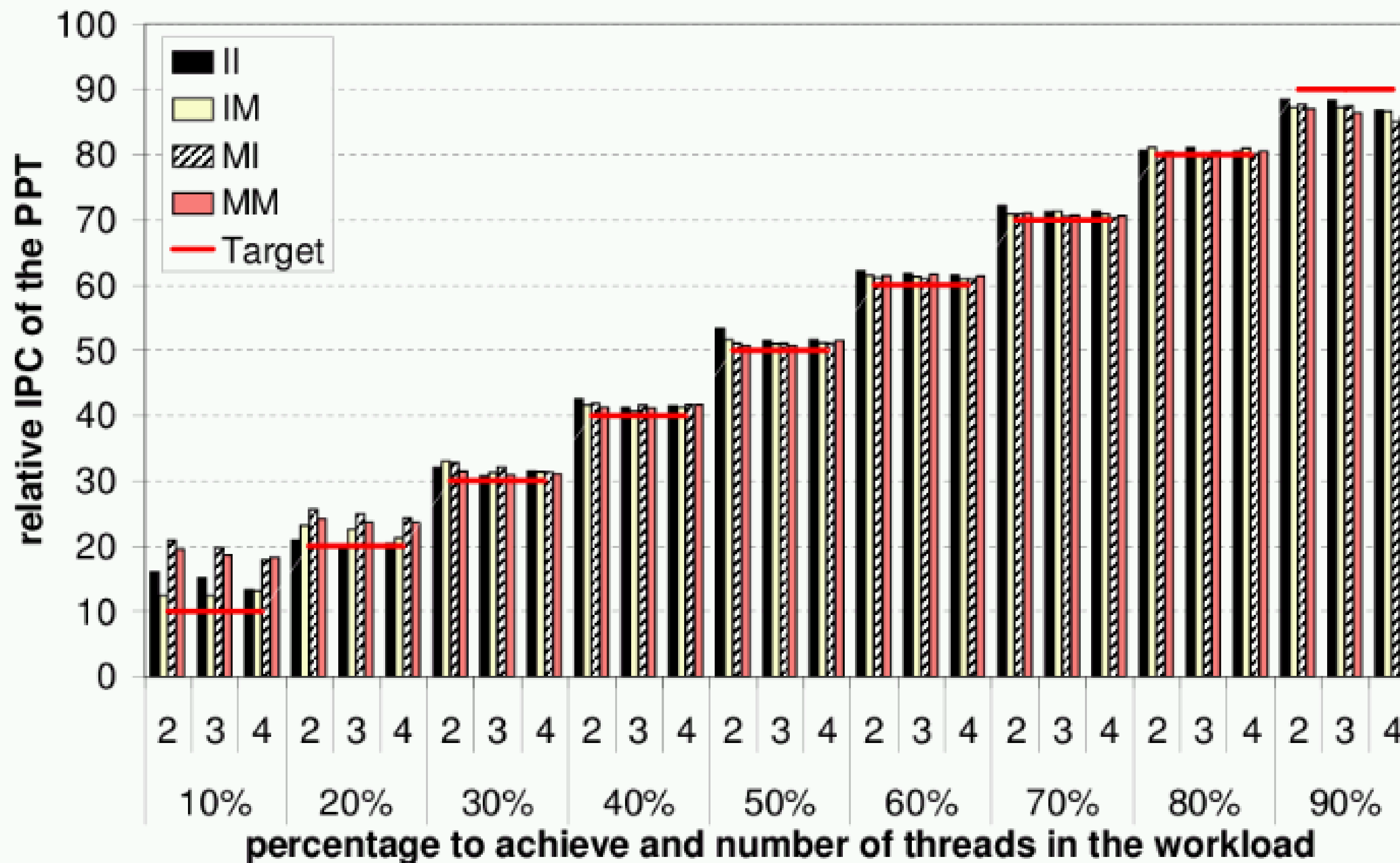
QoS using static resource partitioning

- Allocation a fixed amount of resources to a thread does not solve the QoS problem



QoS using explicit allocation

- We can always achieve the desired performance level for the thread



Conclusion

- Explicit resource allocation mechanisms:
  - Increase performance in a dynamically shared SMT
  - Allow the implementation of QoS in SMT
    - Makes SMT processors interesting for a wider range of domains with QoS requirements of their own
      - Real-time
      - Network processing