

## Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC

Murakami, Kazuaki

Computing and Communication Center, Kyushu University | Faculty of Information Science and Electrical Engineering, Kyushu University

Mauro Goulart Ferreira, Victor

Faculty of Information Science and Electrical Engineering, Kyushu University

---

<https://hdl.handle.net/2324/9111>

---

出版情報：SLRC プrezentation, 2004-10-27. 九州大学システムLSI研究センター

バージョン：

権利関係：

SBAC-PAD2004 Tutorial  
Foz do Iguaçu, Oct. 27, 2004

# “Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC” (Part 1)

Kazuaki J. Murakami (\*1)

Victor Mauro Goulart Ferreira (\*2)

\*1: Director of Computing & Communications Center, Kyushu University

\*2: Ph.D. Candidate, Dept. of Informatics, Kyushu University

E-mail: arch@i.kyushu-u.ac.jp or kjm@acm.org

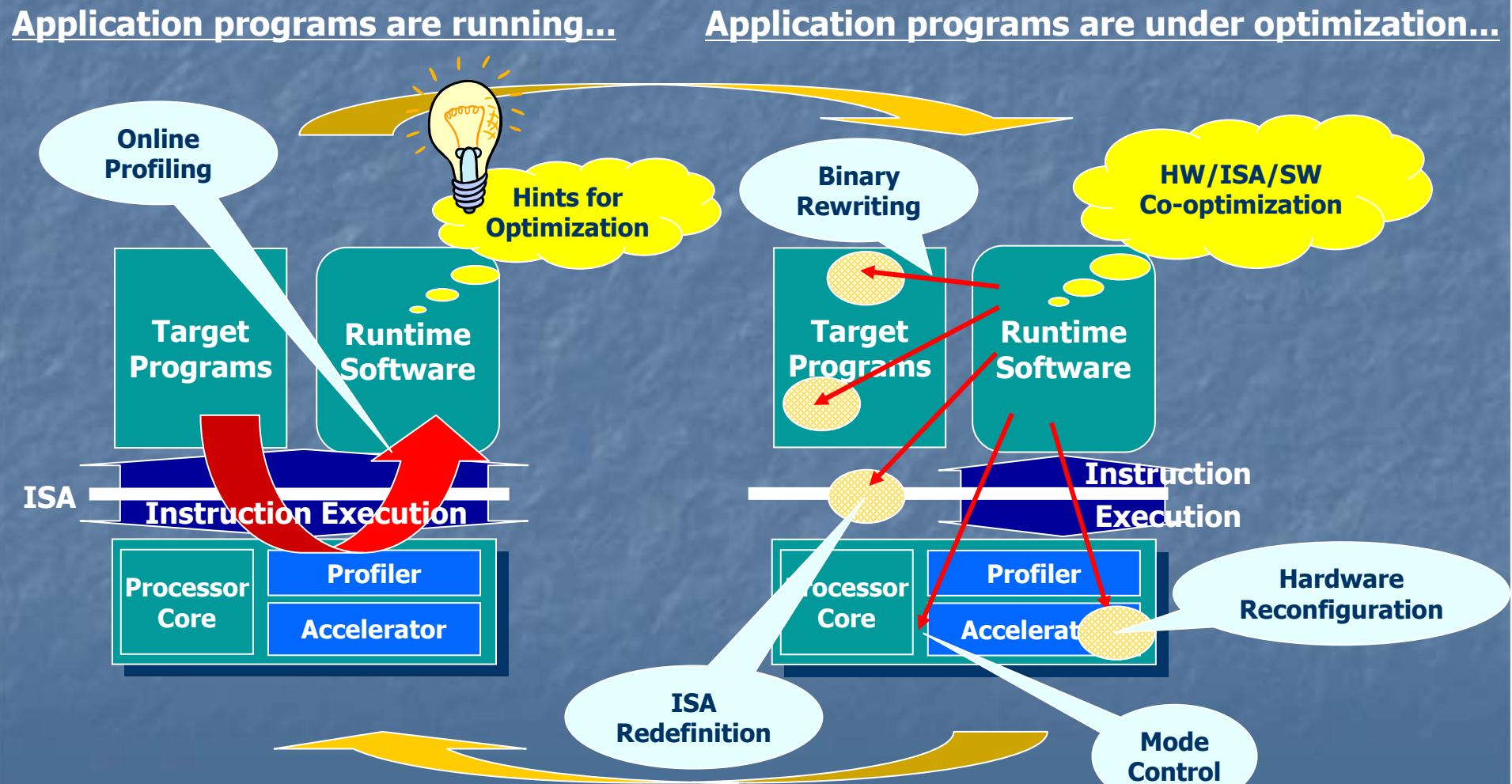
# Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
- Part 2 (17:30-19:00 → 15:00-16:30)
  - On-demand Recomputation

# What is Just-in-Time HW/ISA/SW Co-optimization?

- Just-in-time?
  - **Dynamic**: Optimize SoC,
    - After SoC's are shipped to the market
    - While SoC's are used in the field
  - **Online**: Optimize SoC,
    - In parallel with the execution of application programs
    - In advance of the completion of the execution
  - **Adaptive**: Optimize SoC repeatedly,
    - In the form of a feedback loop
    - Until the system reaches some stable state
- HW/ISA/SW Co-optimization?
  - Optimize HW/ISA/SW of SoC coordinately

# JIT HW/ISA/SW Co-optimization would work like ...



# Analogy: Formula 1

The car  
(=application program) is running



After the optimization,  
the car returns to the race

The car is now under optimization

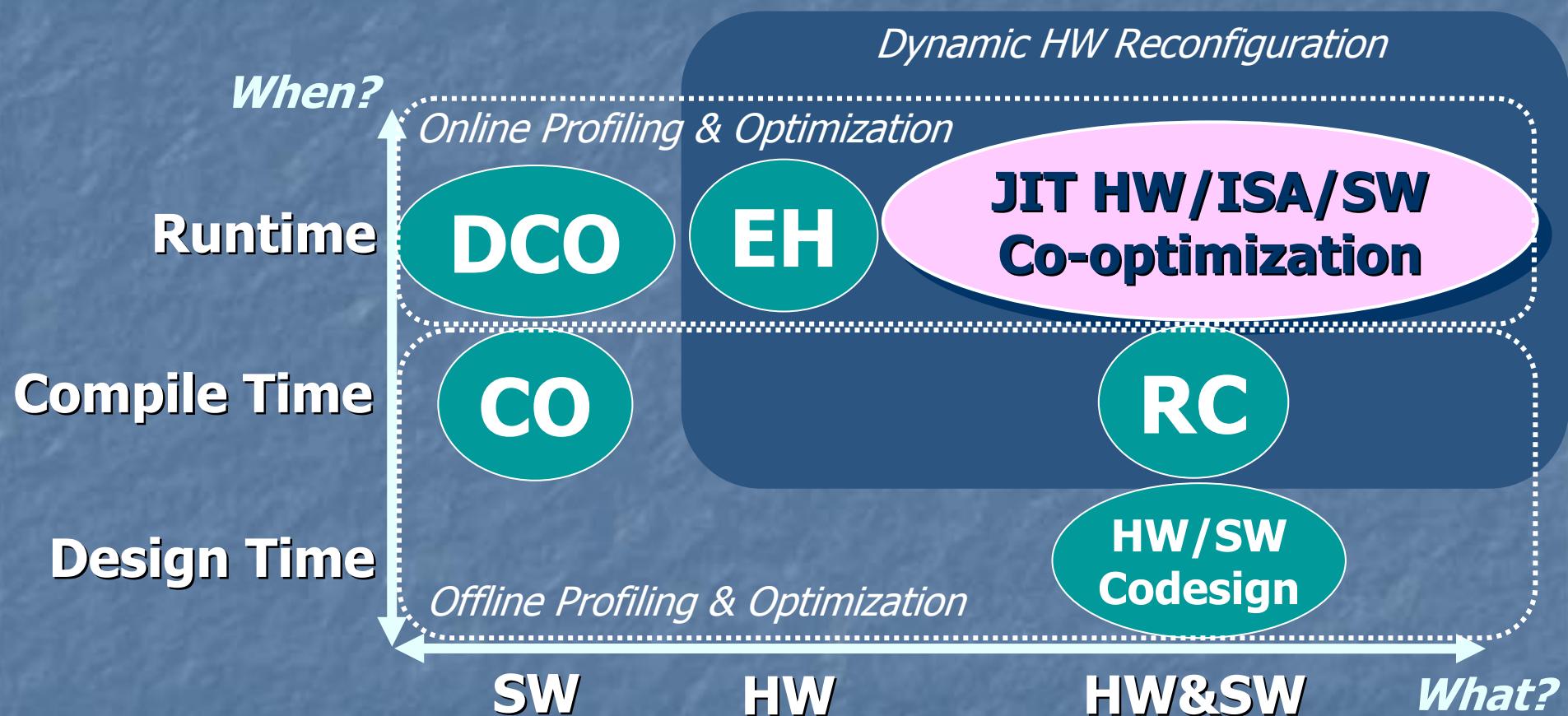


The pit crew  
(=SysteMorph software) is monitoring the behavior of the car



Once the pit crew finds any hints for optimization, the car pits in

# JIT HW/ISA/SW Co-optimization and Other Optimization Techniques



CO: Compiler Optimization

DCO: Dynamic Compilation/Optimization

EH: Evolvable Hardware

RC: Reconfigurable Computing

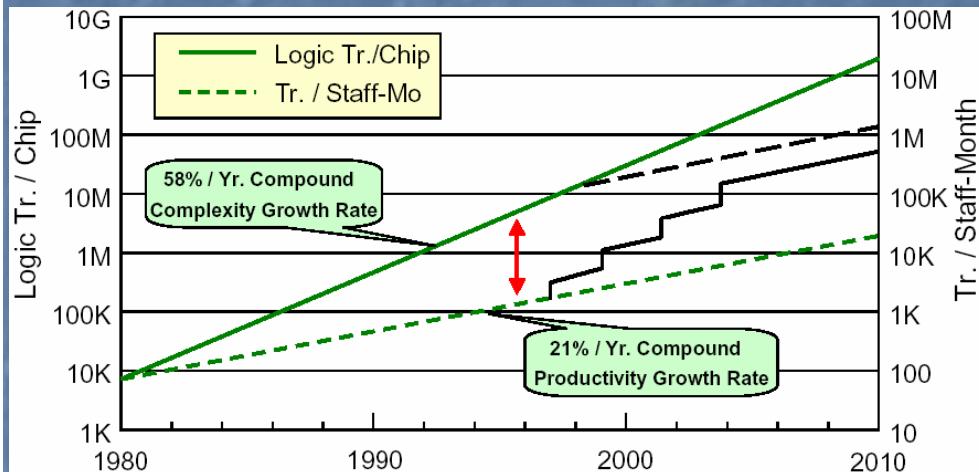
# Some References

- RC (Reconfigurable Computing)
  - FPL (<http://fpl.org>)
  - RAW (<http://www.ece.lsu.edu/vaidy/raw05/>)
- EH (Evolvable Hardware)
  - EH (<http://ic.arc.nasa.gov/ic/eh2000>)
- DCO (Dynamic Compilation/Optimization)
  - Dynamo (HP)
  - Code Morphing Software (Transmeta)
  - JIT Compilers
  - CGO (<http://www.cgo.org>)
- JIT HW/ISA/SW Co-optimization
  - SysteMorph (Kyushu Univ.)

# Why JIT HW/ISA/SW Co-optimization?

## ■ Motivations

- “Time-To-Market”: Need to reduce the TAT of SoC design



- No extra time for optimization
- “Time-In-Market”: Need to extend the product lifetime of SoC’s
  - Extend the application areas of a single SoC design

## ■ Goals

- Enable SoC’s to optimize and customize themselves in the field according as the users’ behavior or favorite

- Move the optimization phase from the SoC design time to the SoC operation time

- Allow SoC’s to modify their functionality in the field

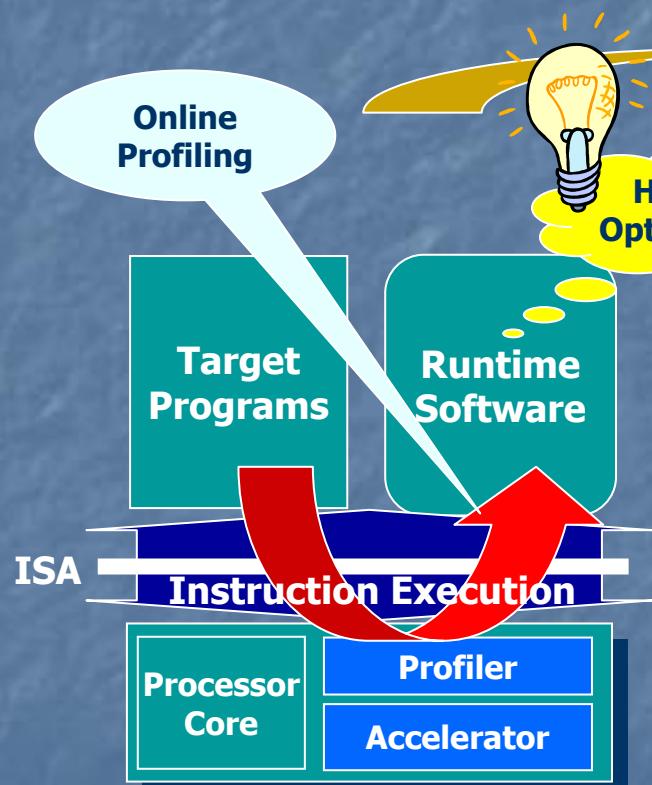
- Make them tolerant of any specification changes in future

# Tutorial Outline

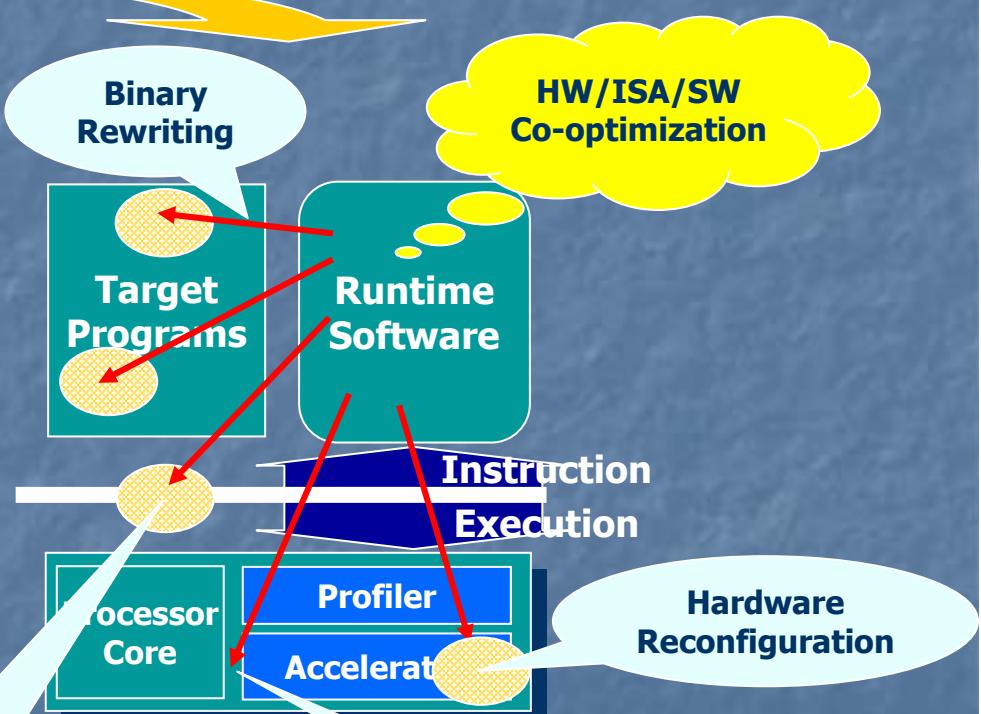
- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
    - ✓ Definition and motivations
    - Implementation frameworks
    - Implementation examples
  - Functionality Morphing
- Part 2 (17:30-19:00)
  - On-demand Recomputation

# How Can You Implement JIT HW/ISA/SW Co-optimization?

Application programs are running...



Application programs are under optimization...



# When Can You Optimize Your SoC?

- Optimize in idle time

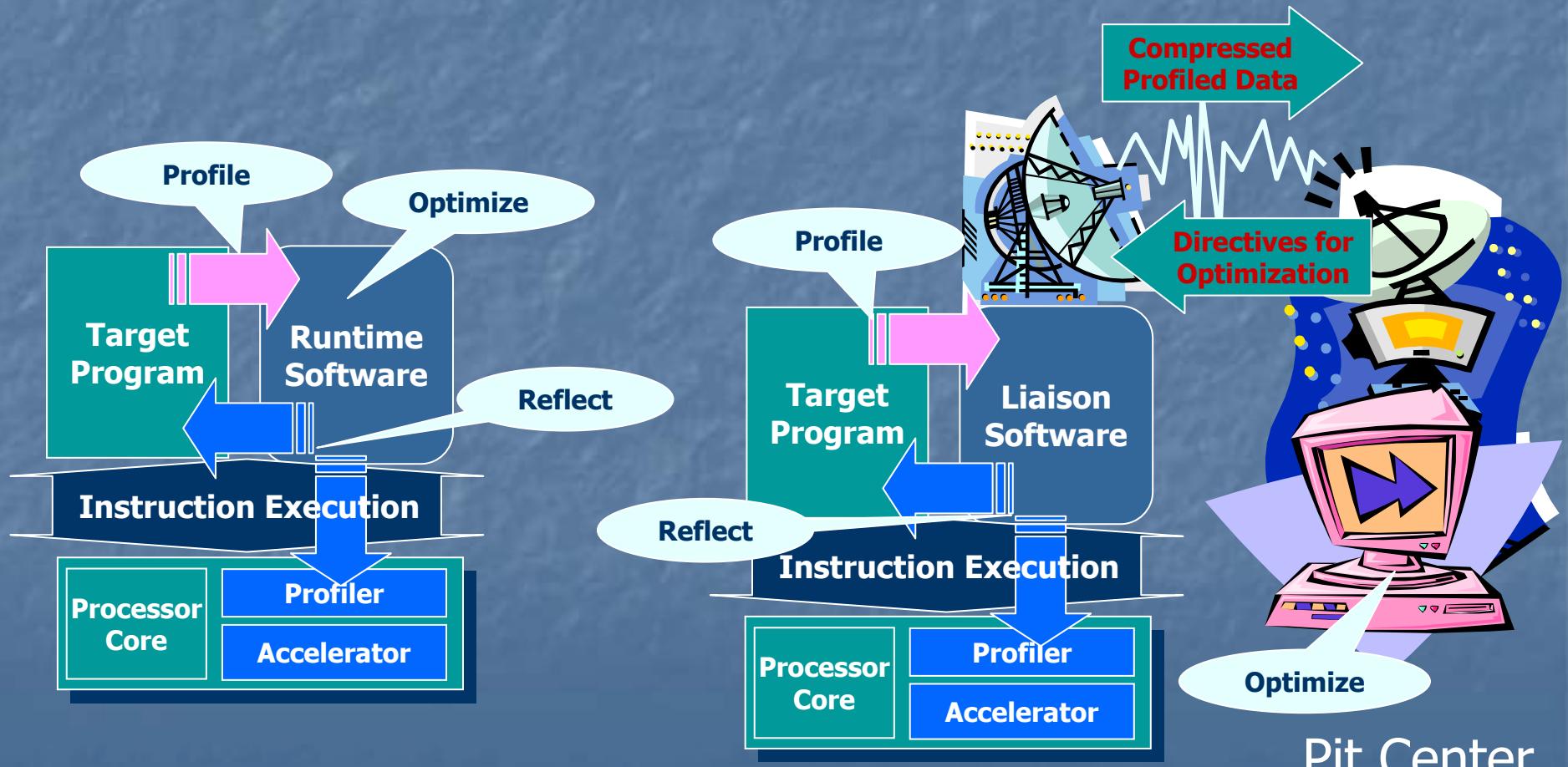


- Optimize in sleep time



# Where Can You Optimize Your SoC?

- Optimize on your hand
- Optimize at some remote “Pit Center”



# Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
    - ✓ Definition and motivations
    - ✓ Implementation frameworks
    - Implementation examples
  - Functionality Morphing
- Part 2 (17:30-19:00)
  - On-demand Recomputation

# How Can You Optimize Your SoC?

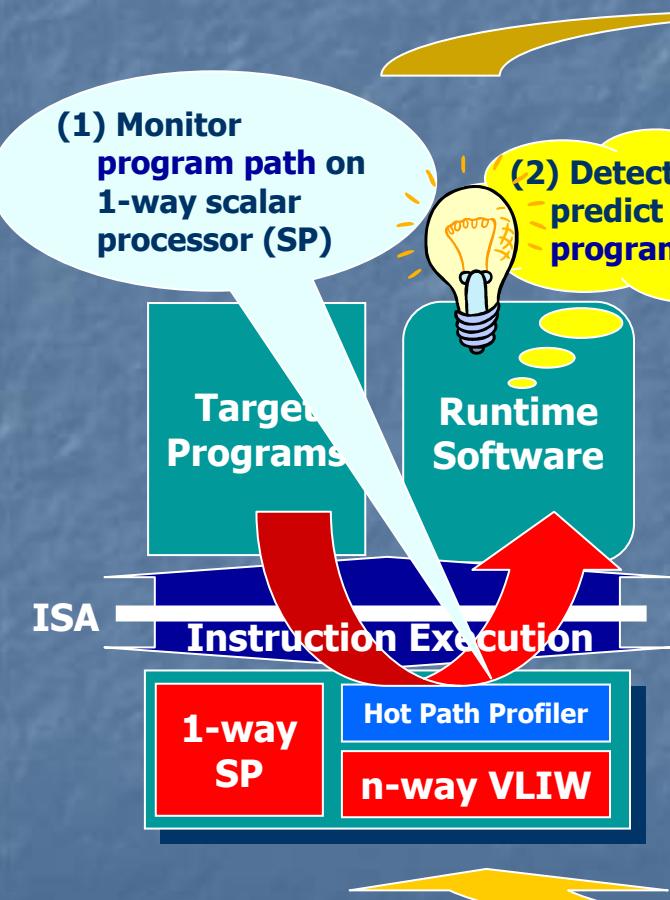
## - Some Examples -

- For higher performance
  - Functionality Morphing
    - Identify some frequently executed things (e.g., hot spots, hot paths, hot instruction sequence), and then offload the execution of these from the main processor to attached accelerator (e.g., wide-issue VLIW, dynamic reconfigurable fabric)
  - On-demand Recomputation
    - Identify some frequently cache-miss causing LOAD instructions (called delinquent loads or critical loads), then replace these LOADs with the corresponding recomputation codes
- For lower power consumption
  - Dynamic Scaling of:
    - Power-supply voltage (DVS: dynamic voltage scaling)
    - Hardware size to use (# of FU's, cache size, and so on)

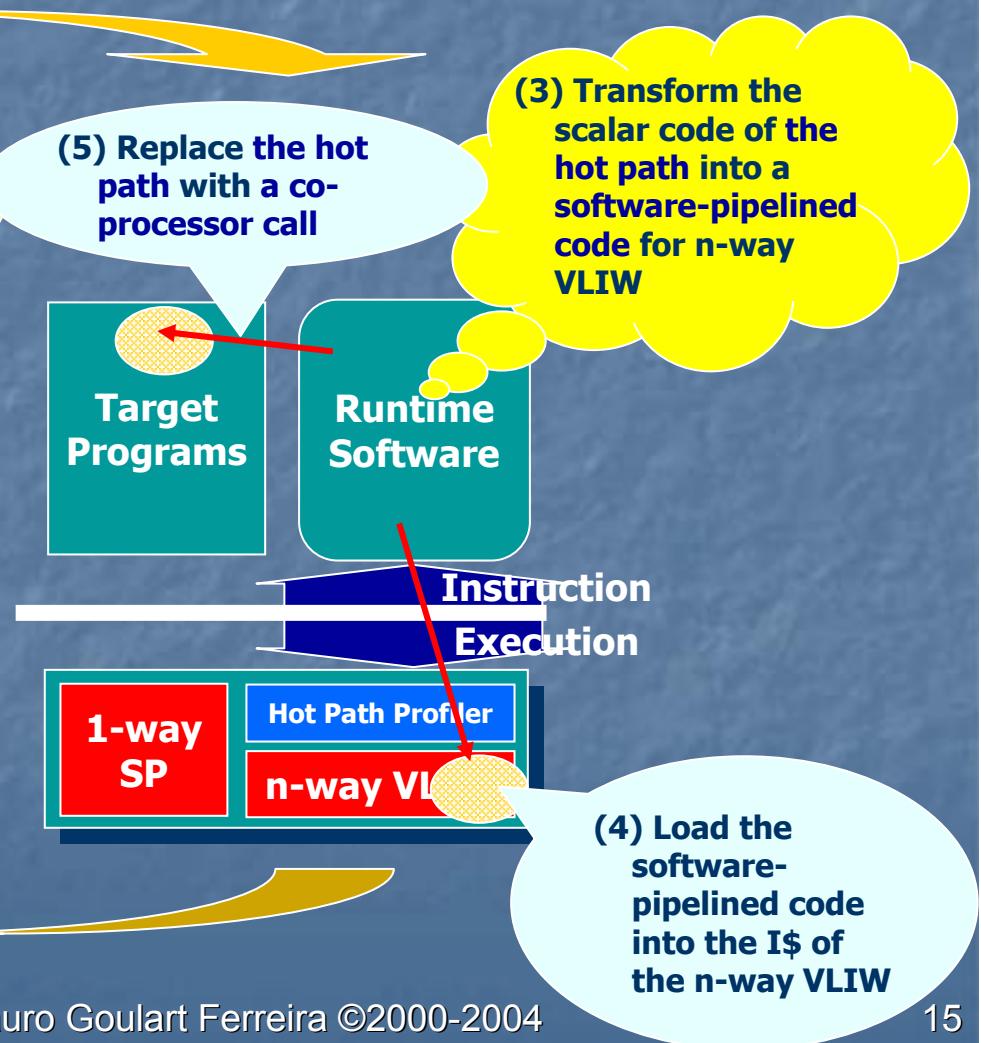
# Functionality Morphing

## - A Case for Hot-Path Offloading -

Application programs are running...

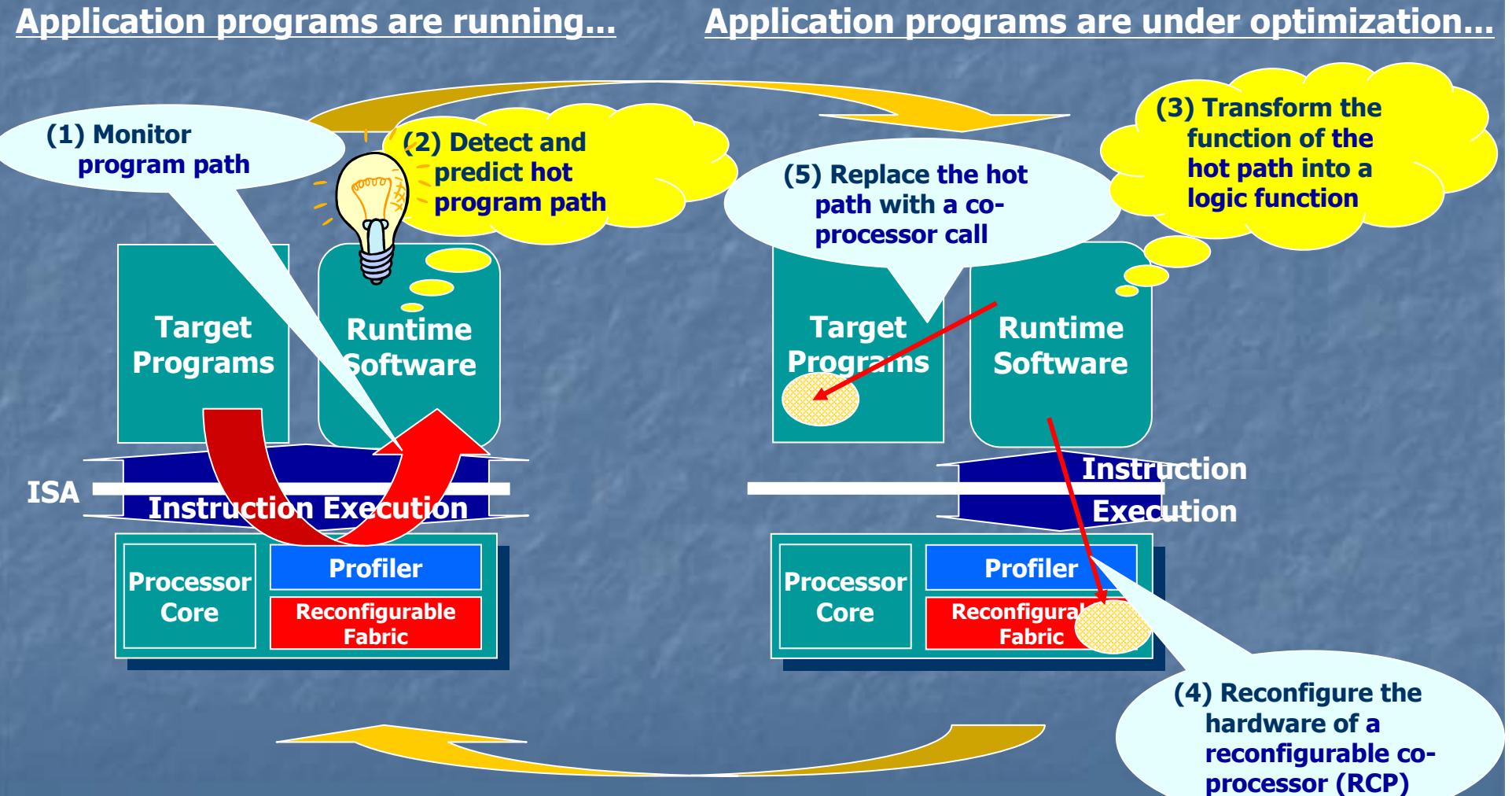


Application programs are under optimization...

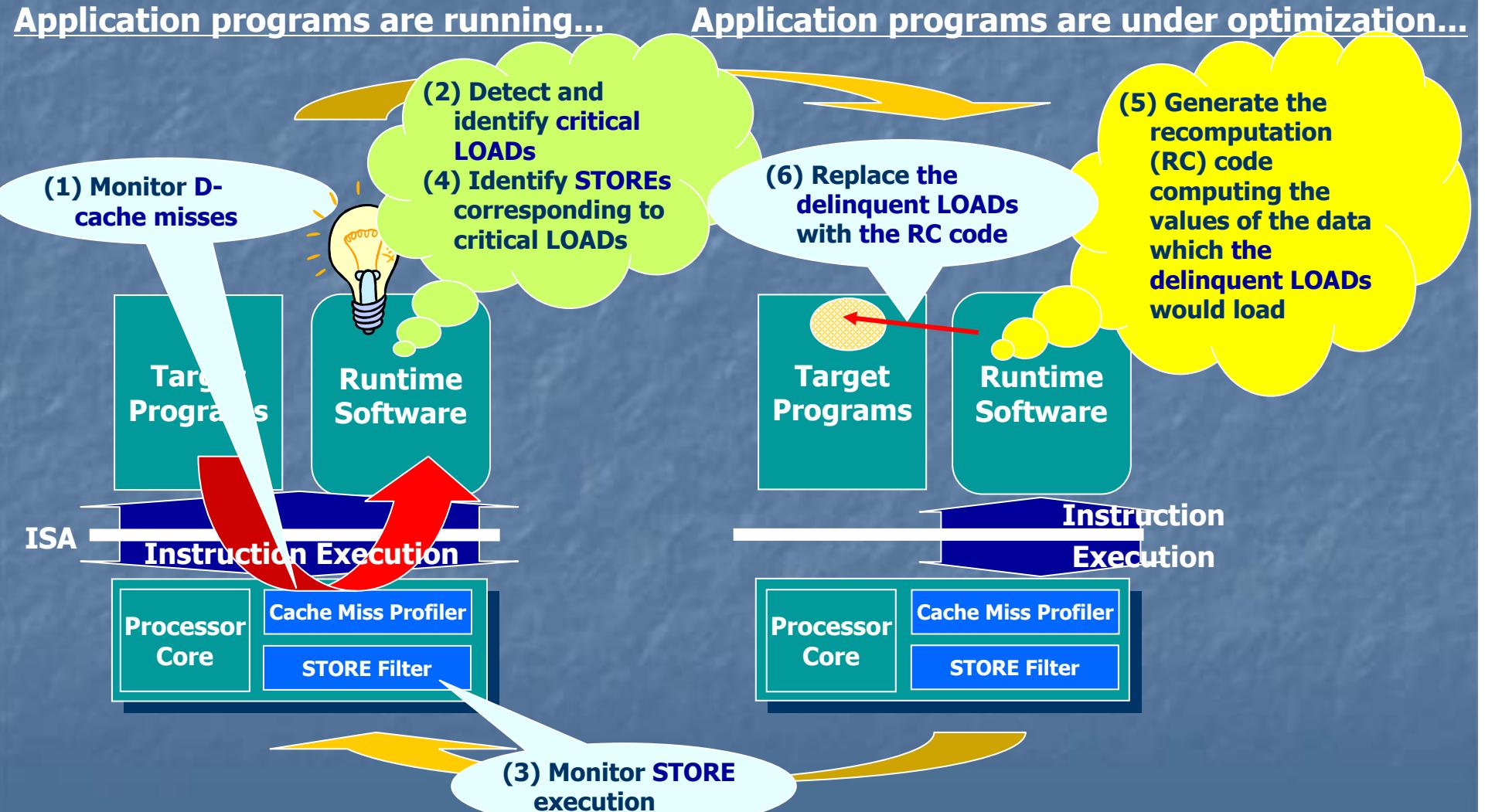


# Functionality Morphing

## - Another Case for Hot-Path Offloading -



# On-demand Recomputation



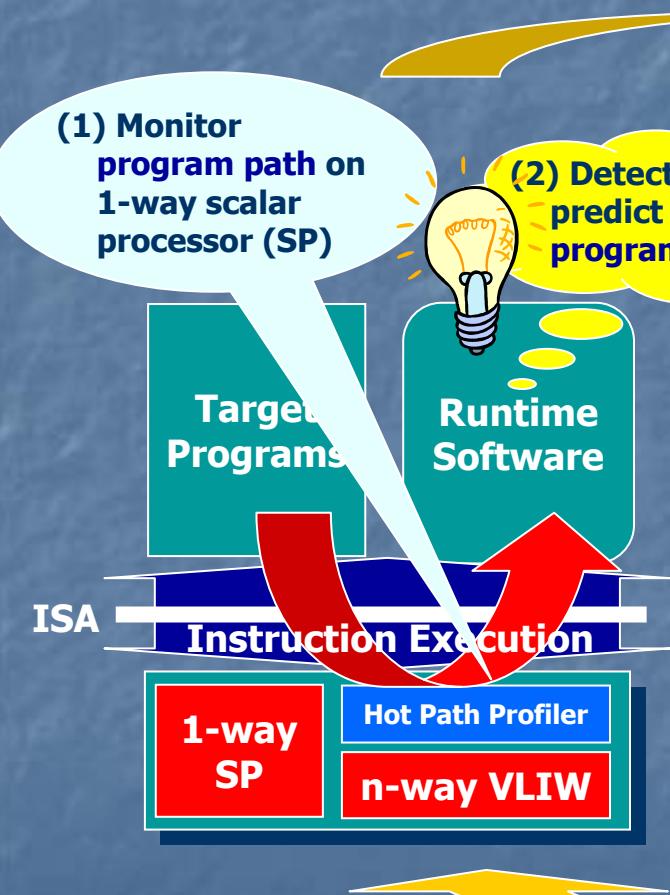
# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - Concept
    - Online hot-path profiling
    - Dynamic trace-based software pipelining
    - Hyperscalar processor (as an accelerator)
    - Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

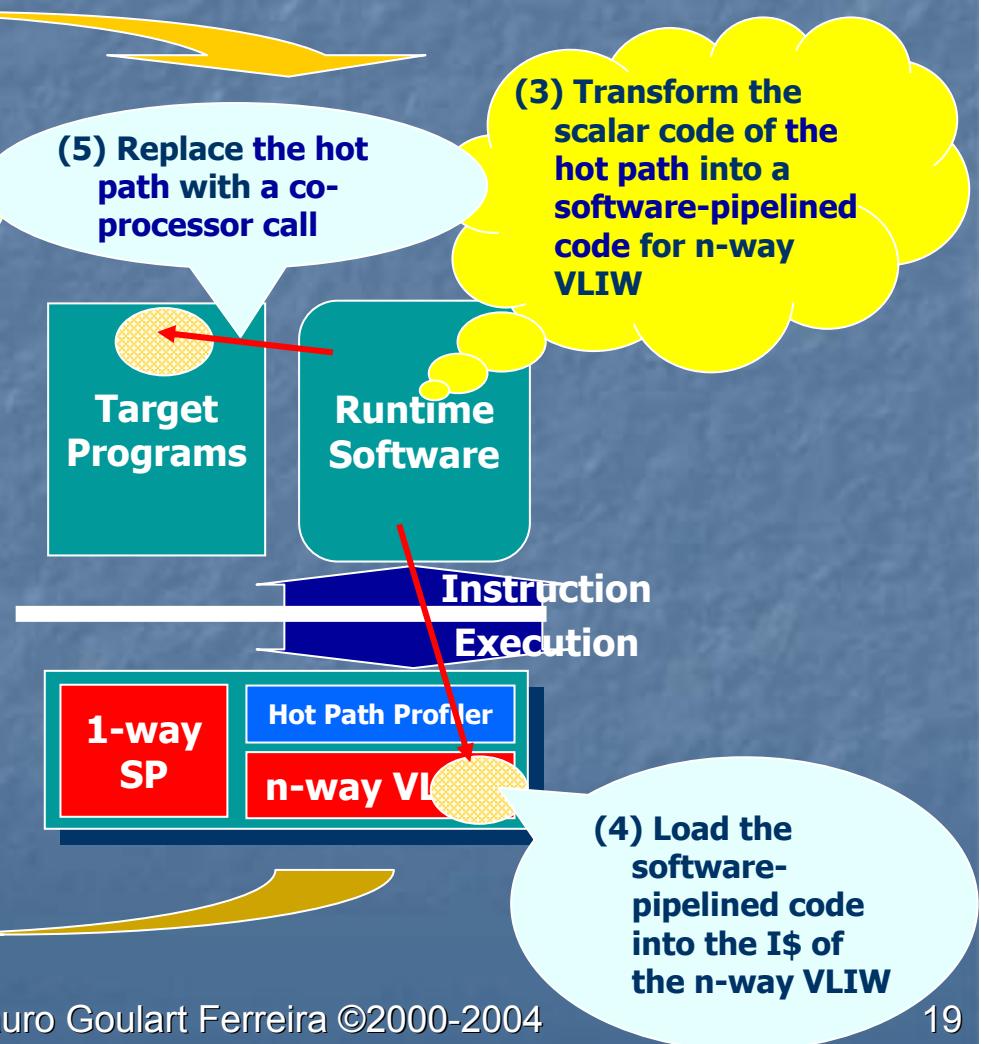
# Functionality Morphing

## - A Case for Hot-Path Offloading -

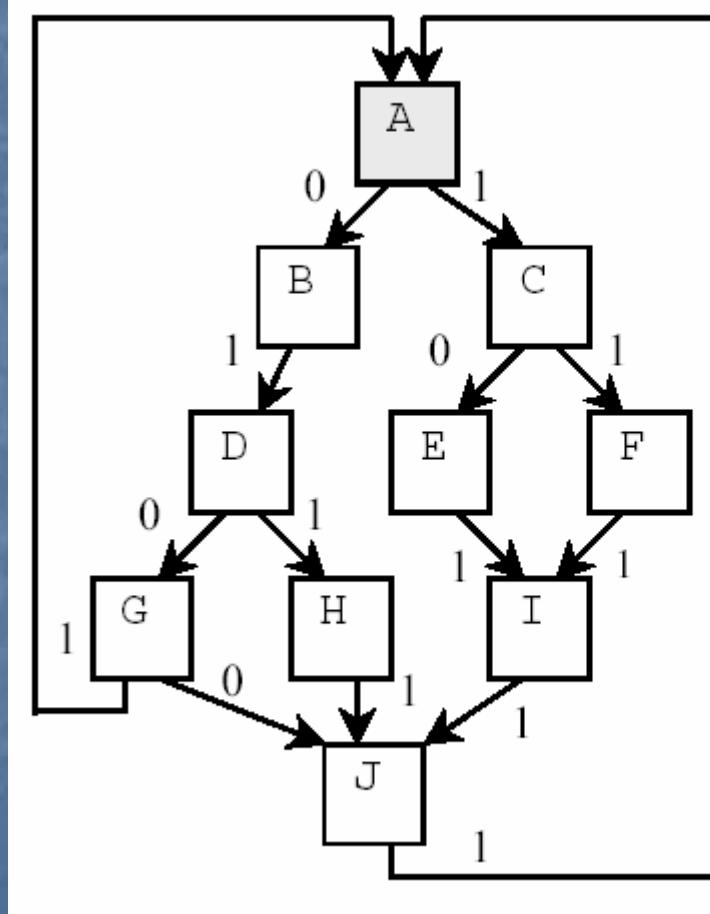
Application programs are running...



Application programs are under optimization...



# Program Paths and Hot Paths



Path: signature

ABDG: A.0101

ABDGJ: A.01001

ABDHJ: A.01111

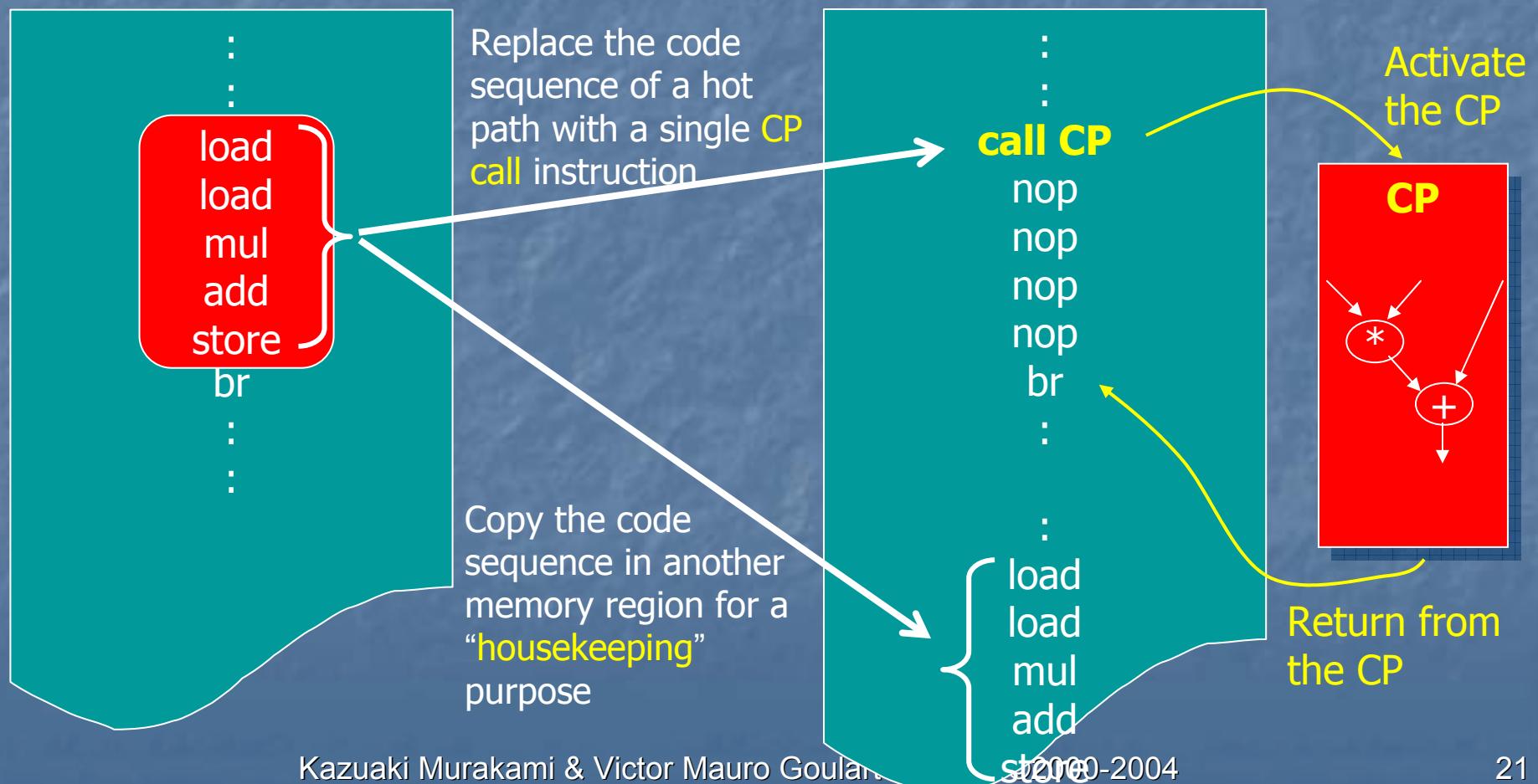
ACEIJ: A.10111

ACFIJ: A.11111

# Functionality Morphing

- Offload Hot Paths from Main Processor to Co-Processor -

- BEFORE: Original binary code
- AFTER: Modified binary code



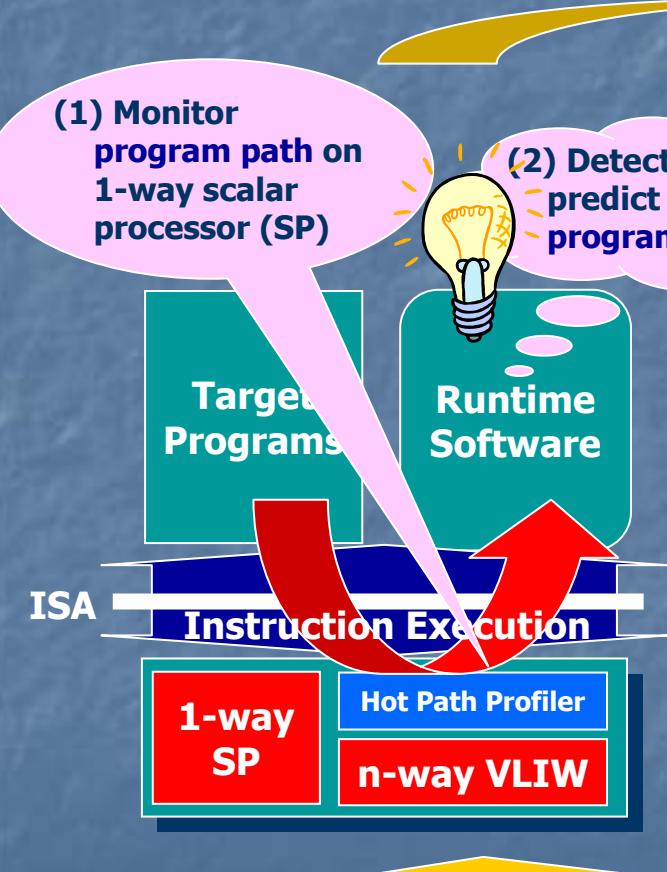
# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - ✓ Concept
    - ➔ Online hot-path profiling
      - Dynamic trace-based software pipelining
      - Hyperscalar processor (as an accelerator)
      - Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

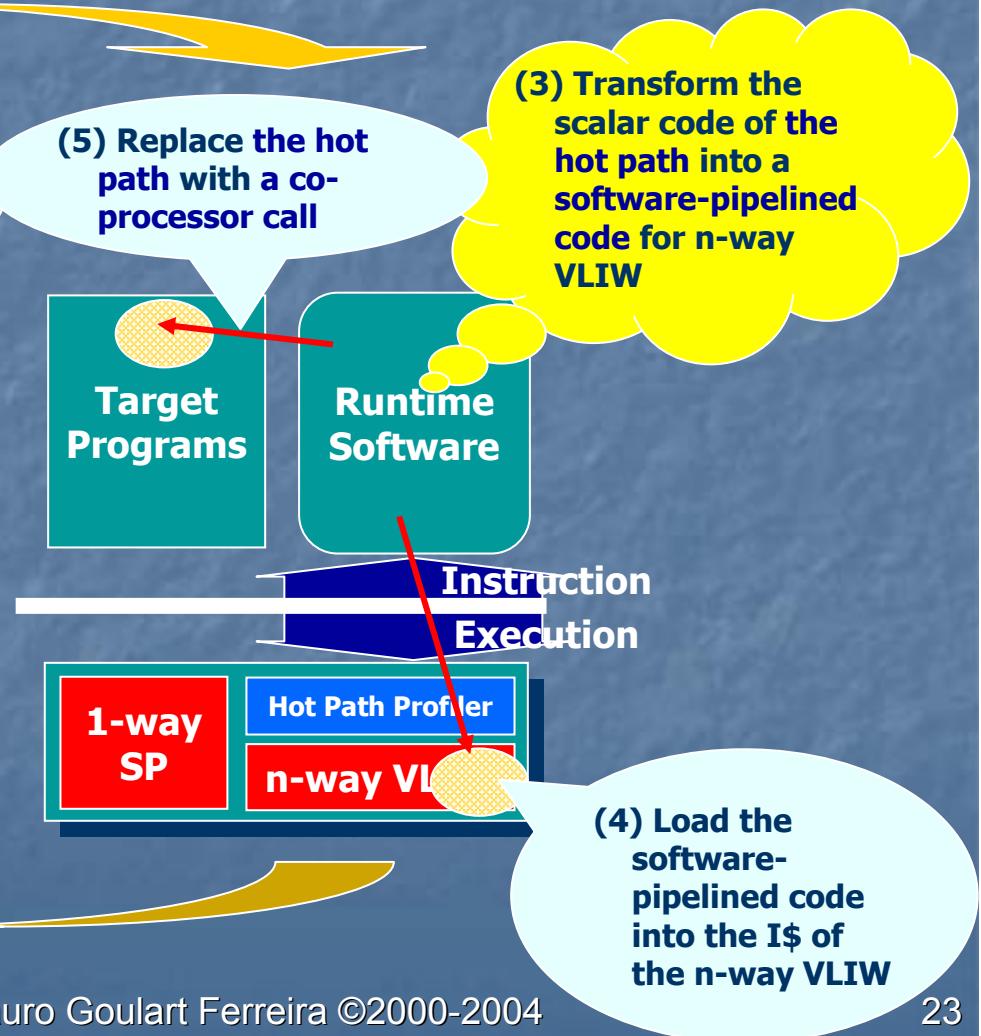
# Functionality Morphing

## - How Do You Predict Hot Paths? -

Application programs are running...



Application programs are under optimization...



# How Do You Predict Hot Paths?

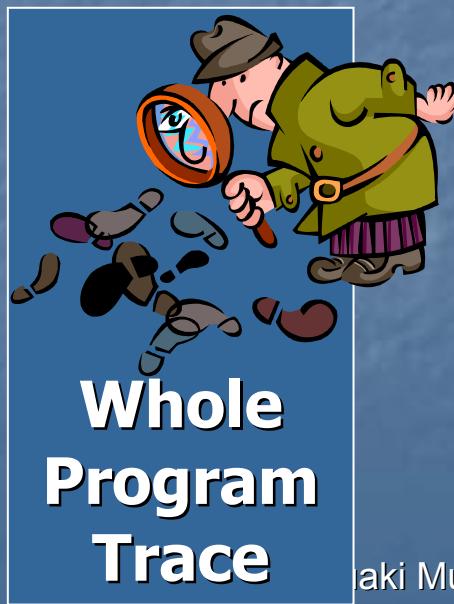
## - Online Hot-Path Profiling -

### ■ Offline Profiling

- Summary of program behavior based on whole program trace

- Good for:

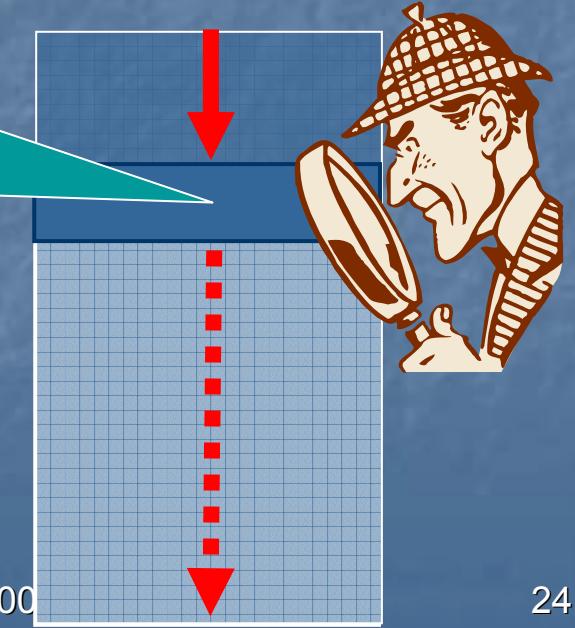
- CO (Compiler Optimization)
  - RC (Reconfigurable Computing)



### ■ Online Profiling

- Prediction based on current execution window of program
- Good for:
  - DCO (Dynamic CO)
  - JIT HW/ISA/SW Co-optimization

Current Execution Window

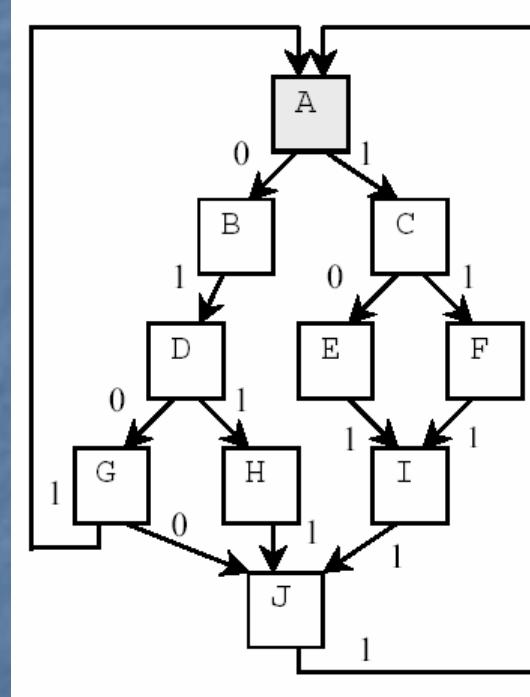


# Online Hot-Path Profiling Algorithms

- (Offline Profiling)
  - Ball-Larus's path profiling [Ball: MICRO1996]
- Online Profiling
  - Dynamo's NET prediction [Duesterwald: ASPLOS2000]
  - SysteMorph's branch-history based hot-path prediction [Yoshimatsu: HPC Asia2004]

# Branch-History Based Hot-Path Prediction

- Profile the history of branch instruction's behaviors (taken or not-taken, branch target)
- If the execution frequency at a path head exceeds the threshold, select the path head ("A" in the figure) as a candidate of the hot path head
- Traverse the object code, starting with the candidate ("A"), based on the branch history, and predict the hot path

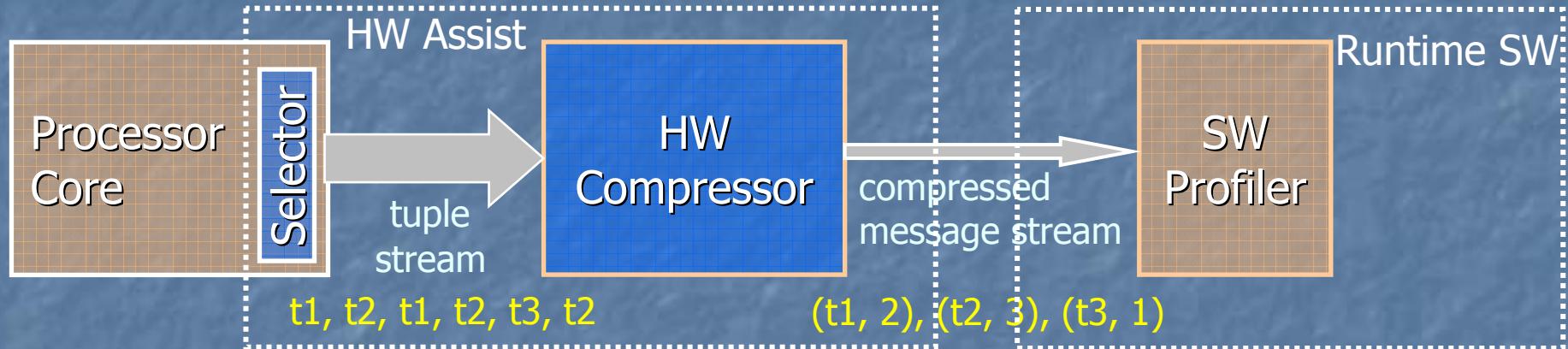


Path: signature  
ABDG: A.0101  
ABDGJ: A.01001  
ABDHJ: A.01111  
ACEIJ: A.10111  
ACFIJ: A.11111

[Yoshimatsu et al., HPC Asia 2004]

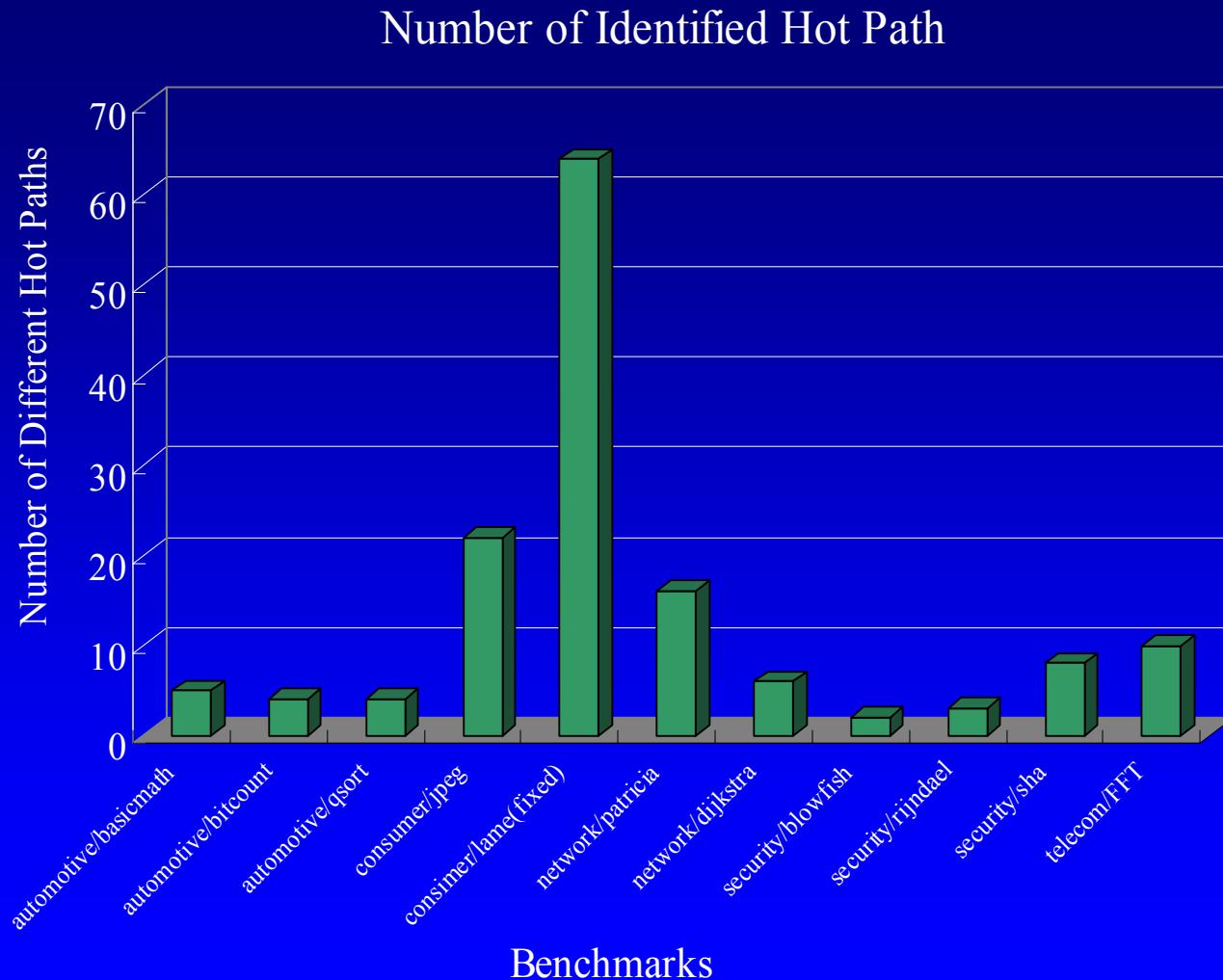
# How to Reduce Runtime Overhead

## - HW Assist & Runtime SW -



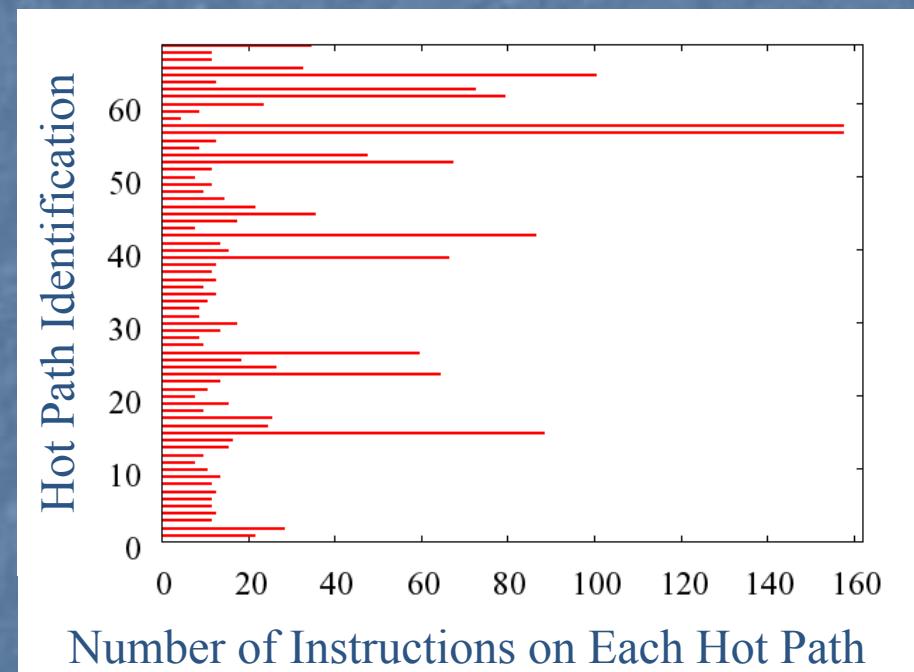
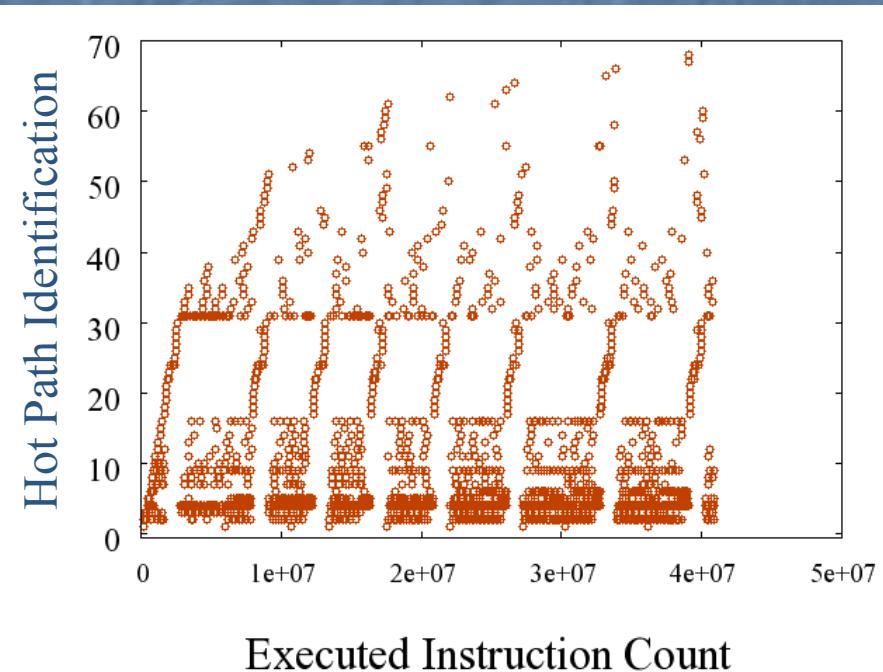
- Selector
  - Buffers all the executed branch instructions
  - Sends them as tuples to HW compressor
- Tuple
  - $\langle \text{bia}, \text{bta} \rangle$ 
    - bia: branch instruction address
    - bta: branch target address
- HW Compressor
  - Counts the tuples, and compresses them in the message form
  - Sends the messages to SW profiler
- Messages
  - $(\text{tuple}, \text{count})$
- SW Profiler
  - Accumulates the messages
  - Predicts hot paths

# How Many Hot Paths Are Detected?



# How Do Hot Paths Appear?

- Application: MP3 encode



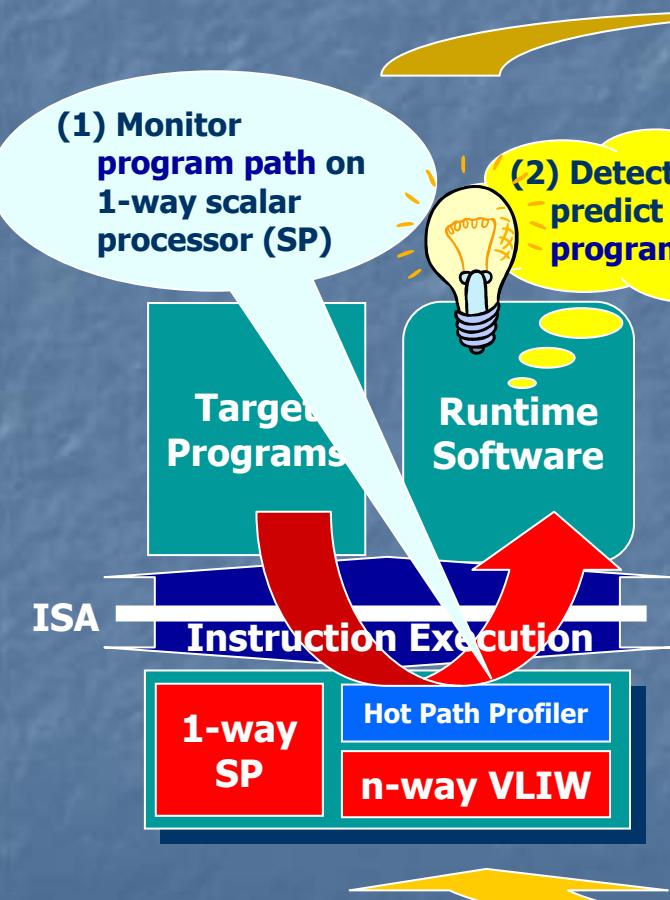
# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - ✓ Concept
    - ✓ Online hot-path profiling
    - ➔ Dynamic trace-based software pipelining
    - Hyperscalar processor (as an accelerator)
    - Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

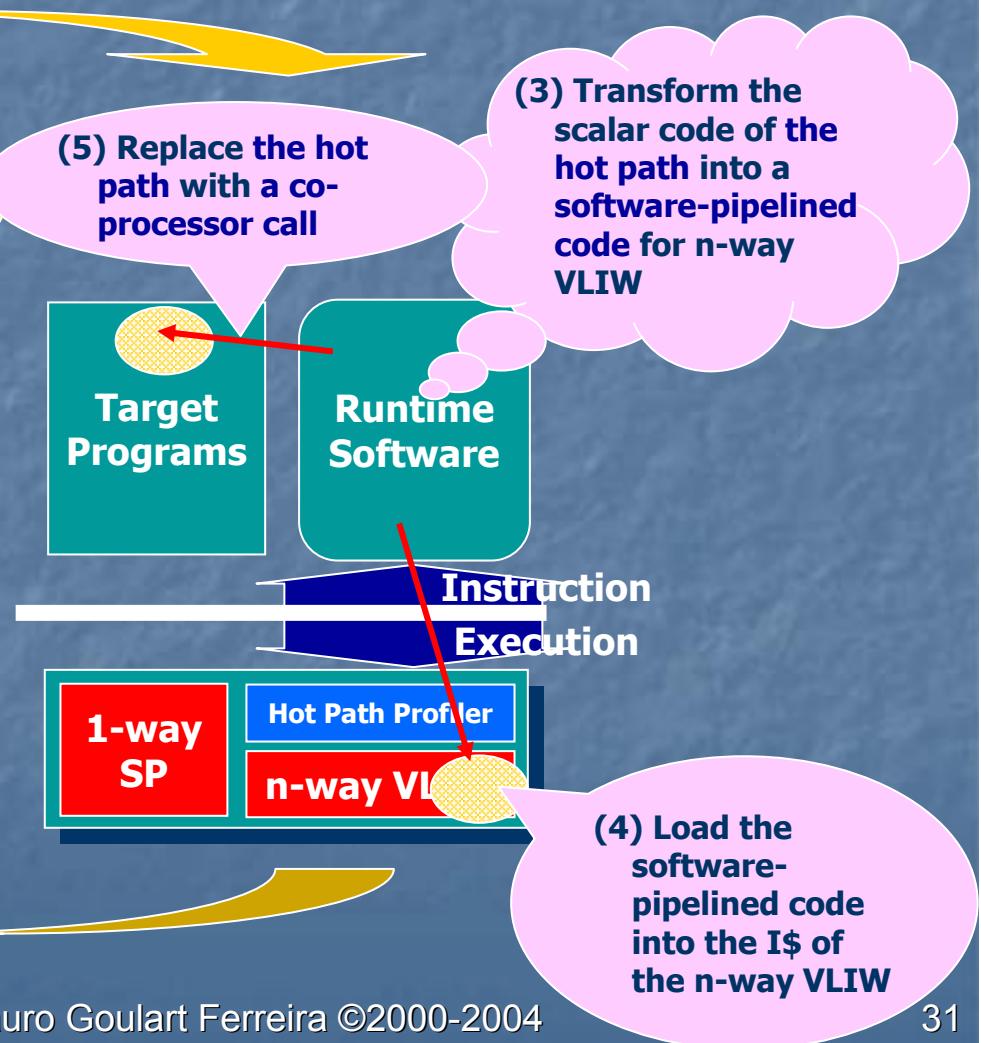
# Functionality Morphing

## - How Do You Speedup Hot-Paths? -

Application programs are running...

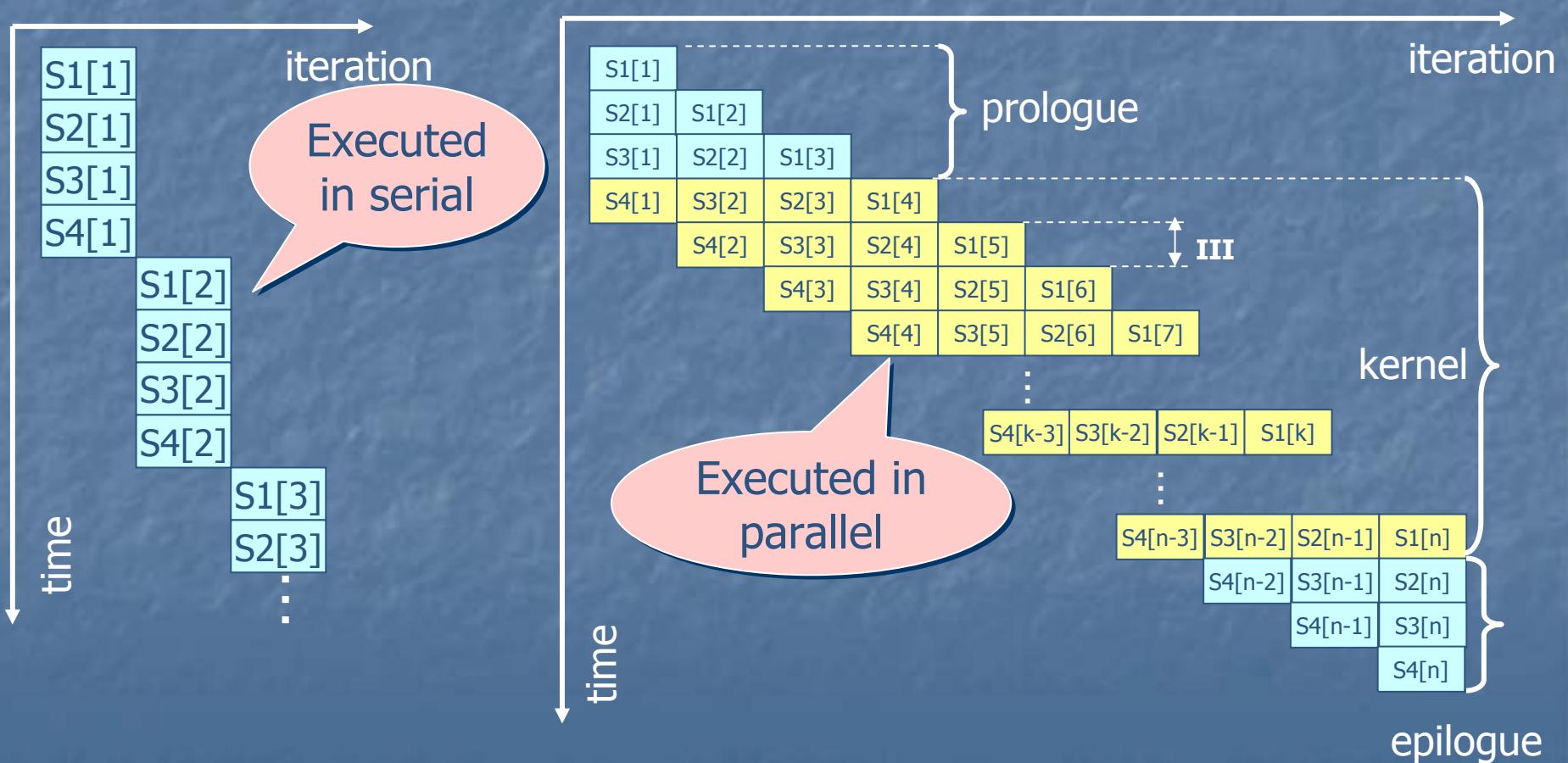


Application programs are under optimization...

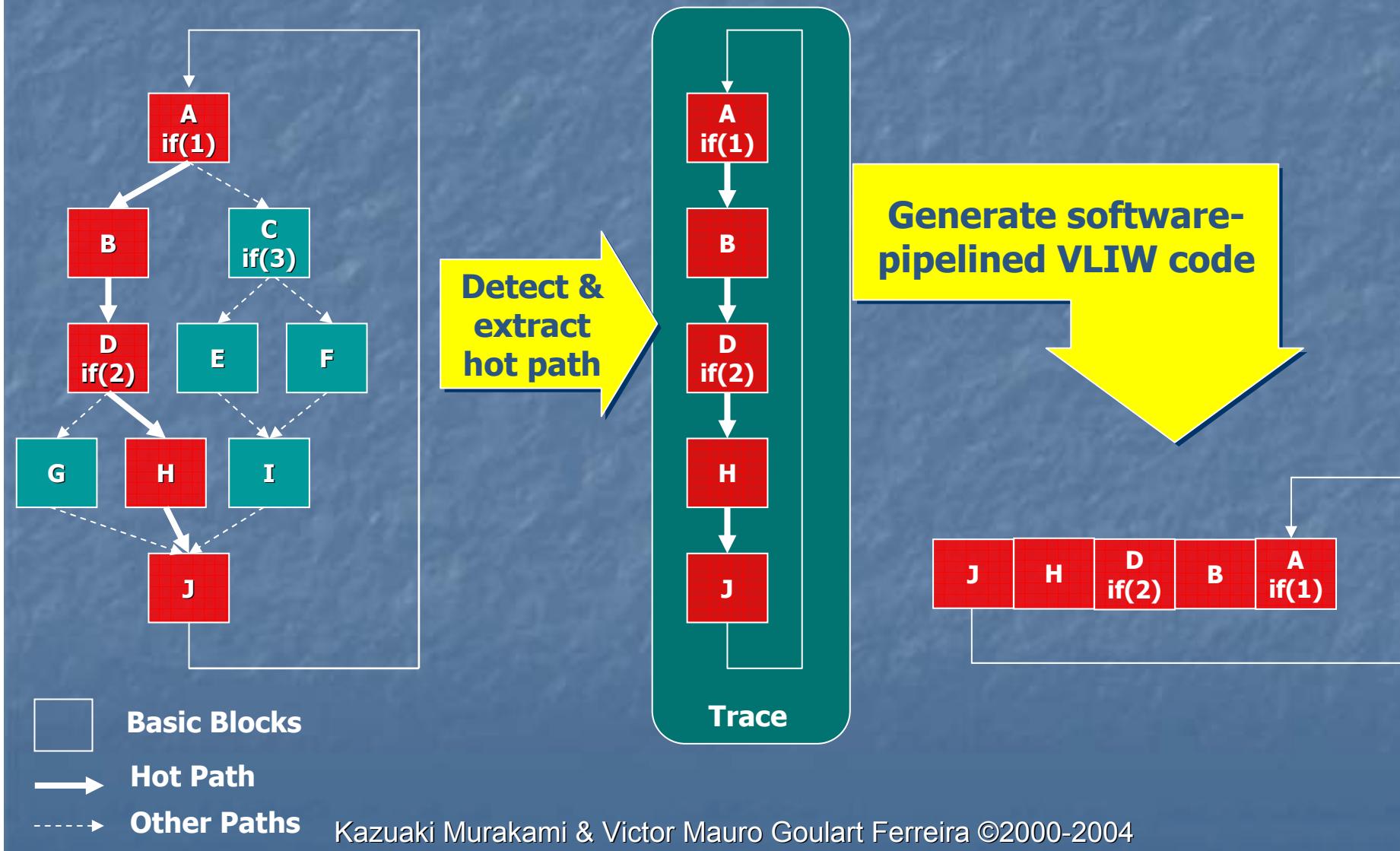


# How Do You Speedup Hot Paths?

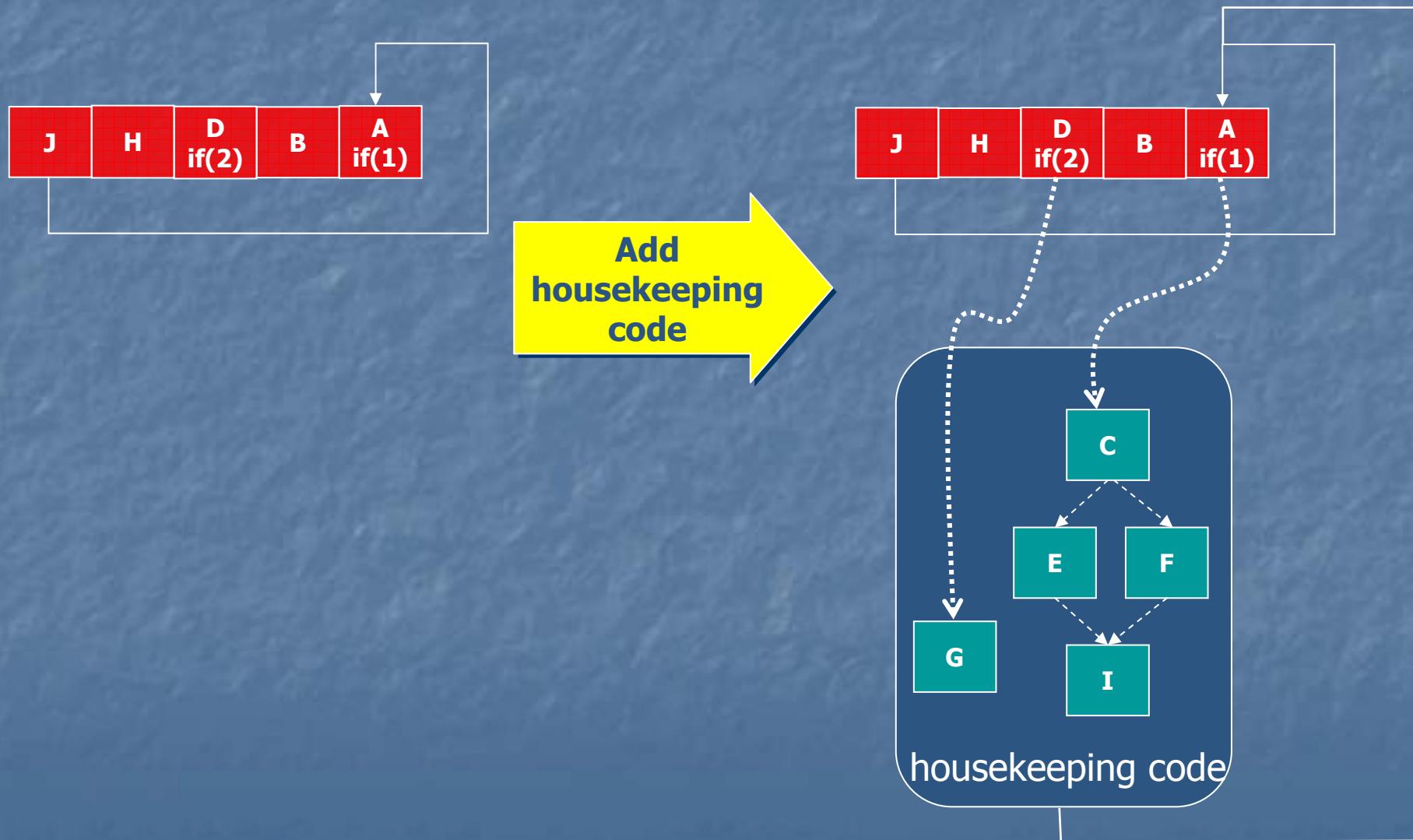
- Dynamic Trace-Based Software Pipelining-
- BEFORE software pipelining
- AFTER software pipelining



# Dynamic Trace-Based Software Pipelining (1)



# Dynamic Trace-Based Software Pipelining (2)



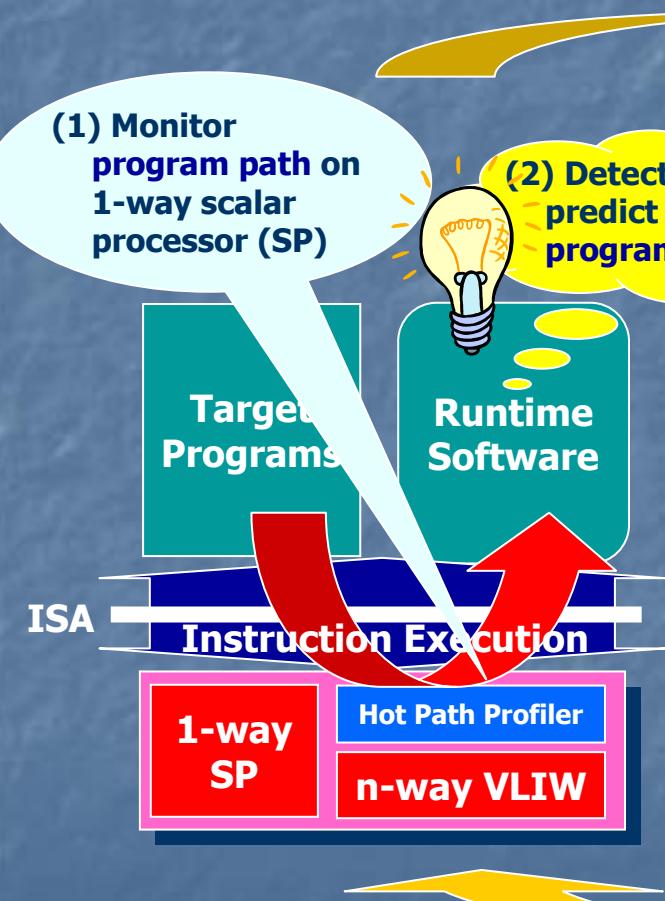
# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - ✓ Concept
    - ✓ Online hot-path profiling
    - ✓ Dynamic trace-based software pipelining
    - ➔ Hyperscalar processor (as an accelerator)
    - Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

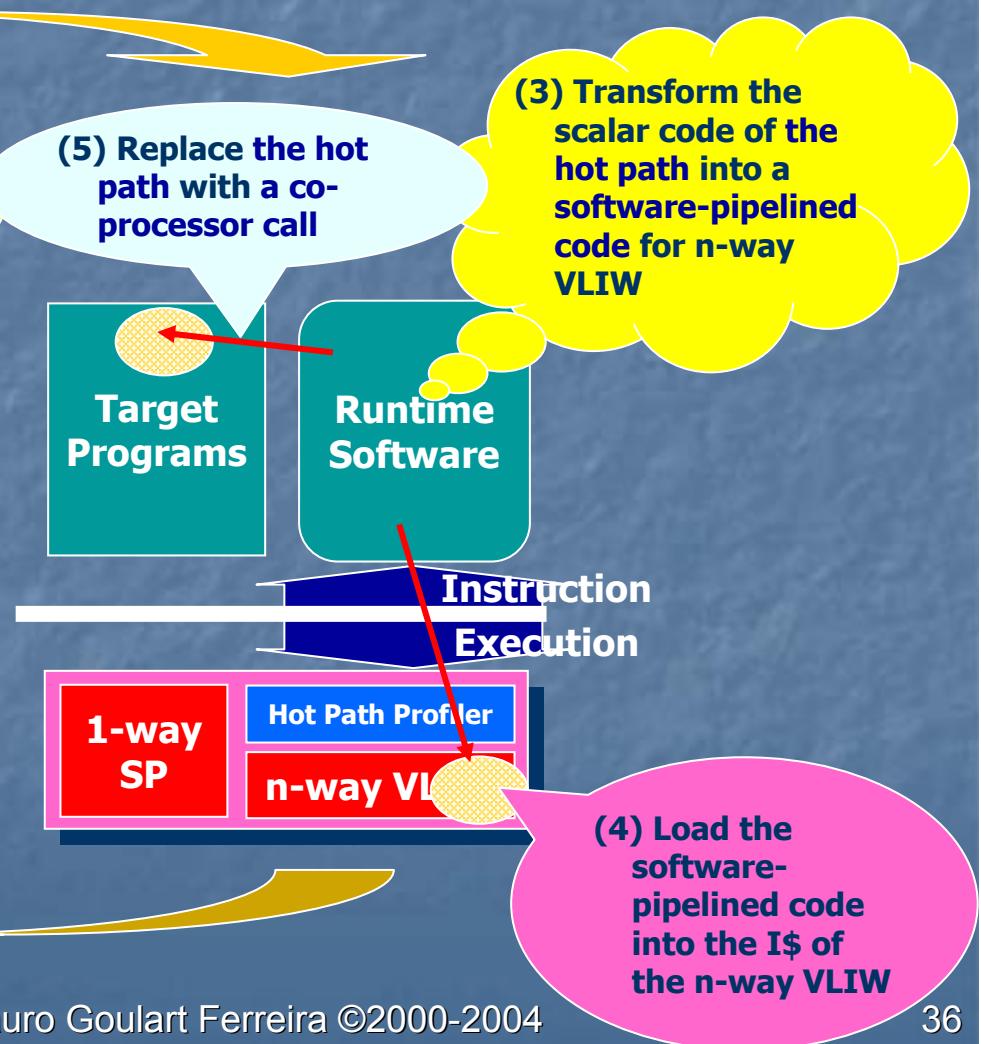
# Functionality Morphing

## - What Architecture Enables It? -

Application programs are running...

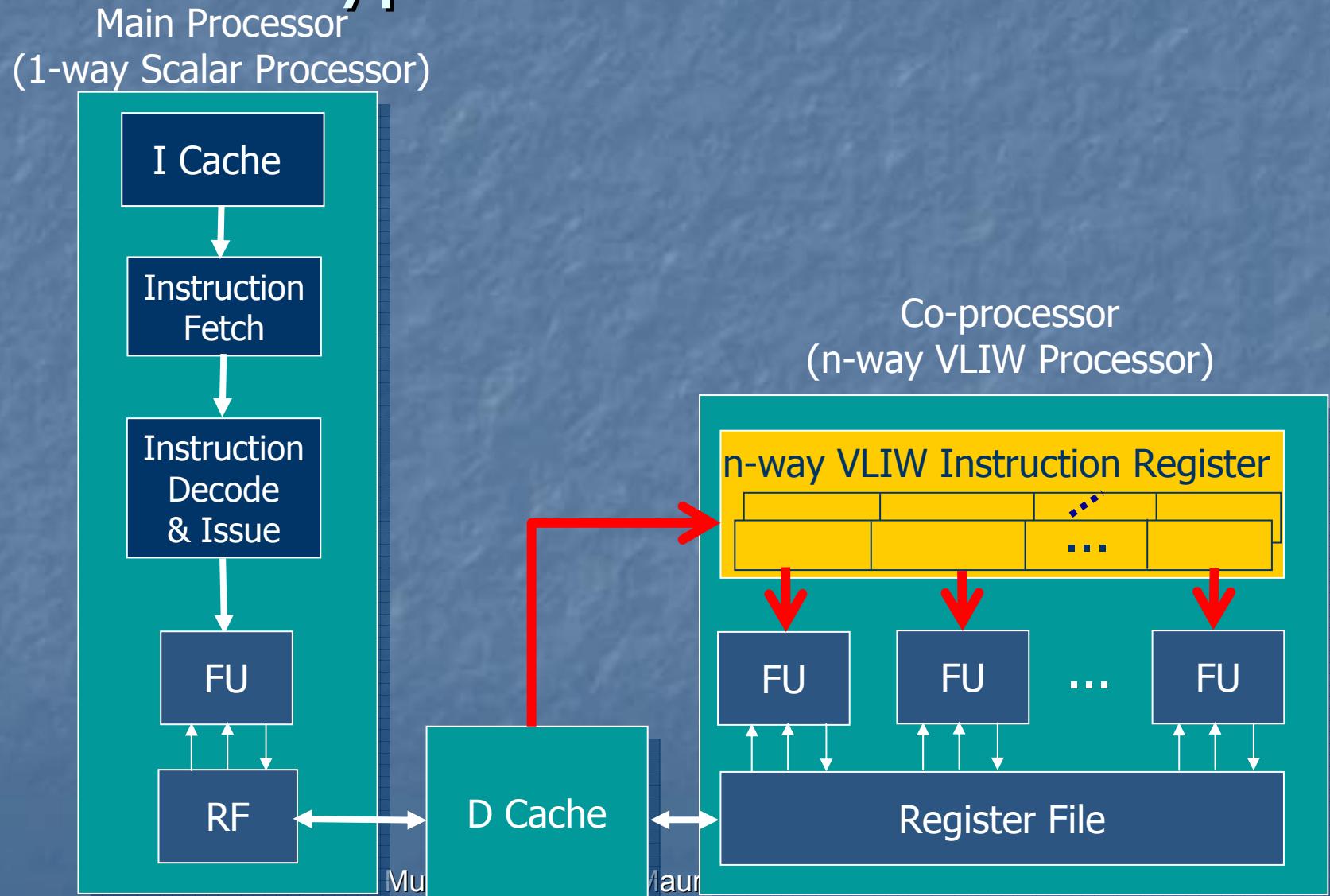


Application programs are under optimization...



# What Architecture Enables Functionality Morphing?

## - Hyperscalar Processor -



# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - ✓ Concept
    - ✓ Online hot-path profiling
    - ✓ Dynamic trace-based software pipelining
    - ✓ Hyperscalar processor (as an accelerator)
    - ➔ Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

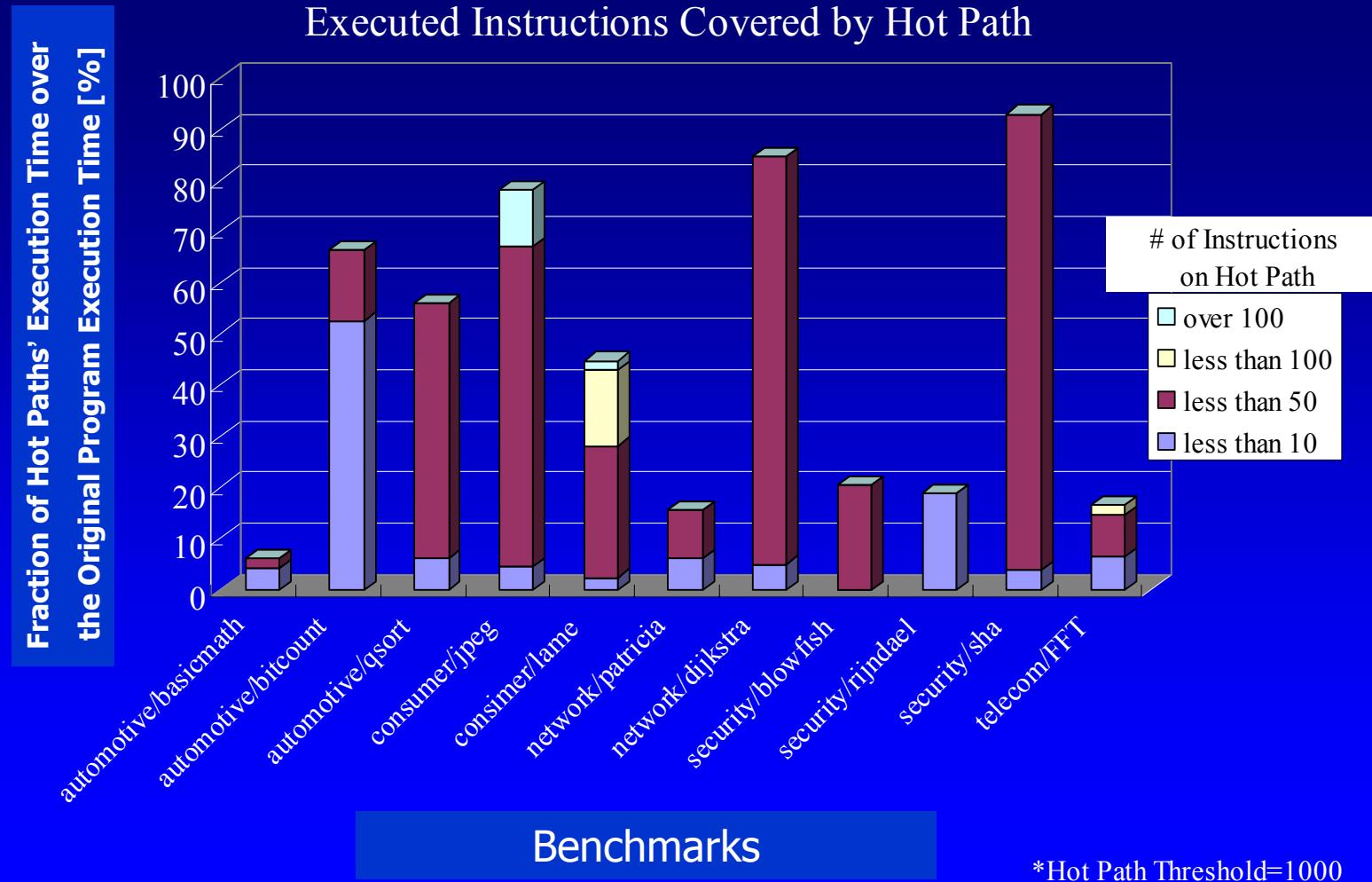
# How Much Can You Improve the Performance?

## - Amdahl's Law -

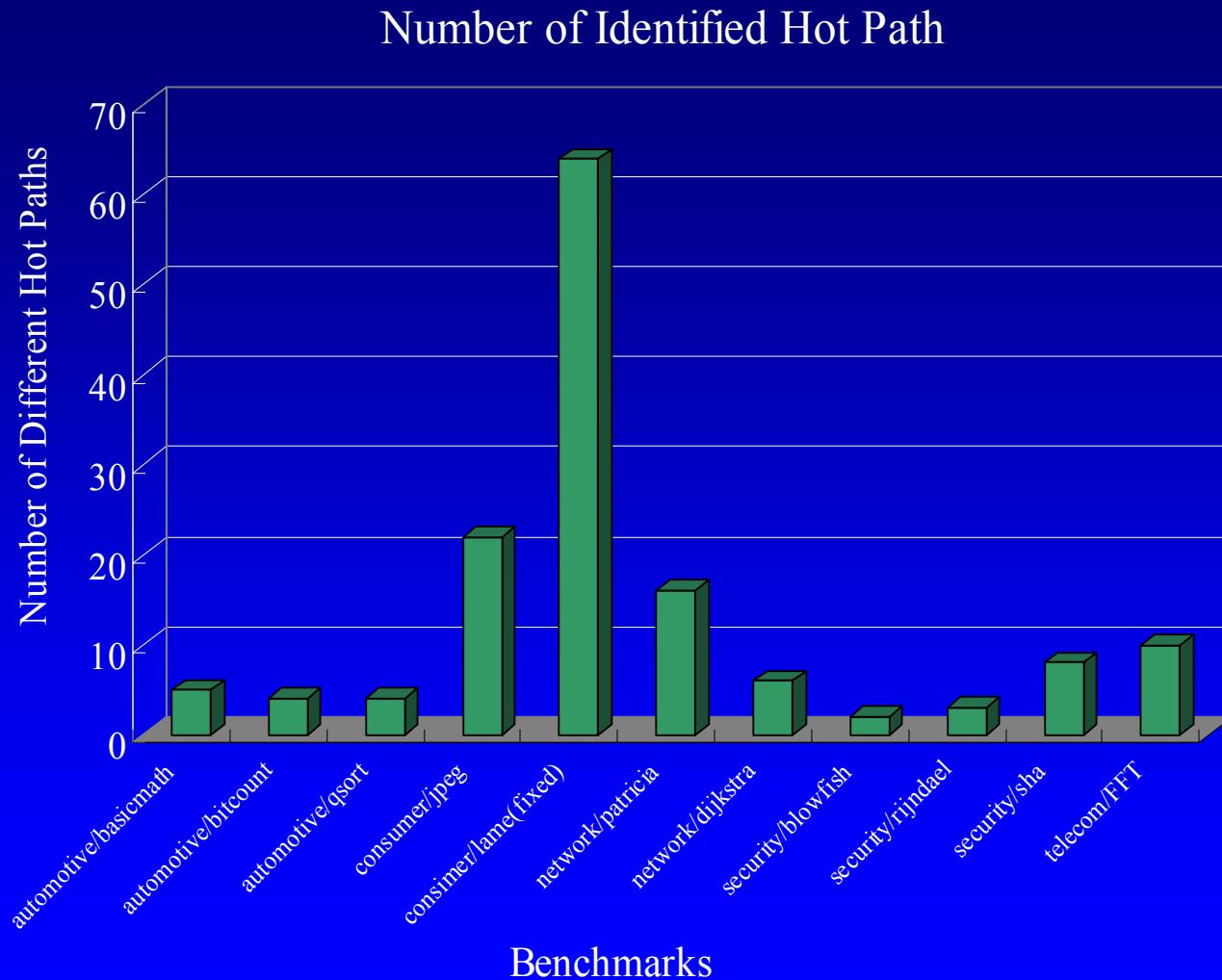
$$S = \frac{1}{(1 - OFF \times HP) + \frac{OFF \times HP}{N}}$$

- $S$ : Speedup
- $HP$ : Fraction of the hot paths' total execution time over the original program execution time ( $0 \leq HP \leq 1$ )
- $OFF$  : Fraction of hot paths detected and offloaded to the accelerator over the entire hot paths ( $0 \leq OFF \leq 1$ )
- $N$ : Number of FU's of n-way VLIW co-processor (accelerator)

# How Much Is the Fraction of Hot Path's Execution Time (HP)?



# How Many Hot Paths Are Detected?



# Tutorial Outline

- ✓ Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - ✓ Functionality Morphing
    - ✓ Concept
    - ✓ Online hot-path profiling
    - ✓ Dynamic trace-based software pipelining
    - ✓ Hyperscalar processor (as an accelerator)
    - ✓ Performance issues
- Part 2 (17:30-19:00)
  - On-demand Recomputation

SBAC-PAD2004 Tutorial  
Foz do Iguaçu, Oct. 27, 2004

# “Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC” (Part 2)

Kazuaki J. Murakami (\*1)

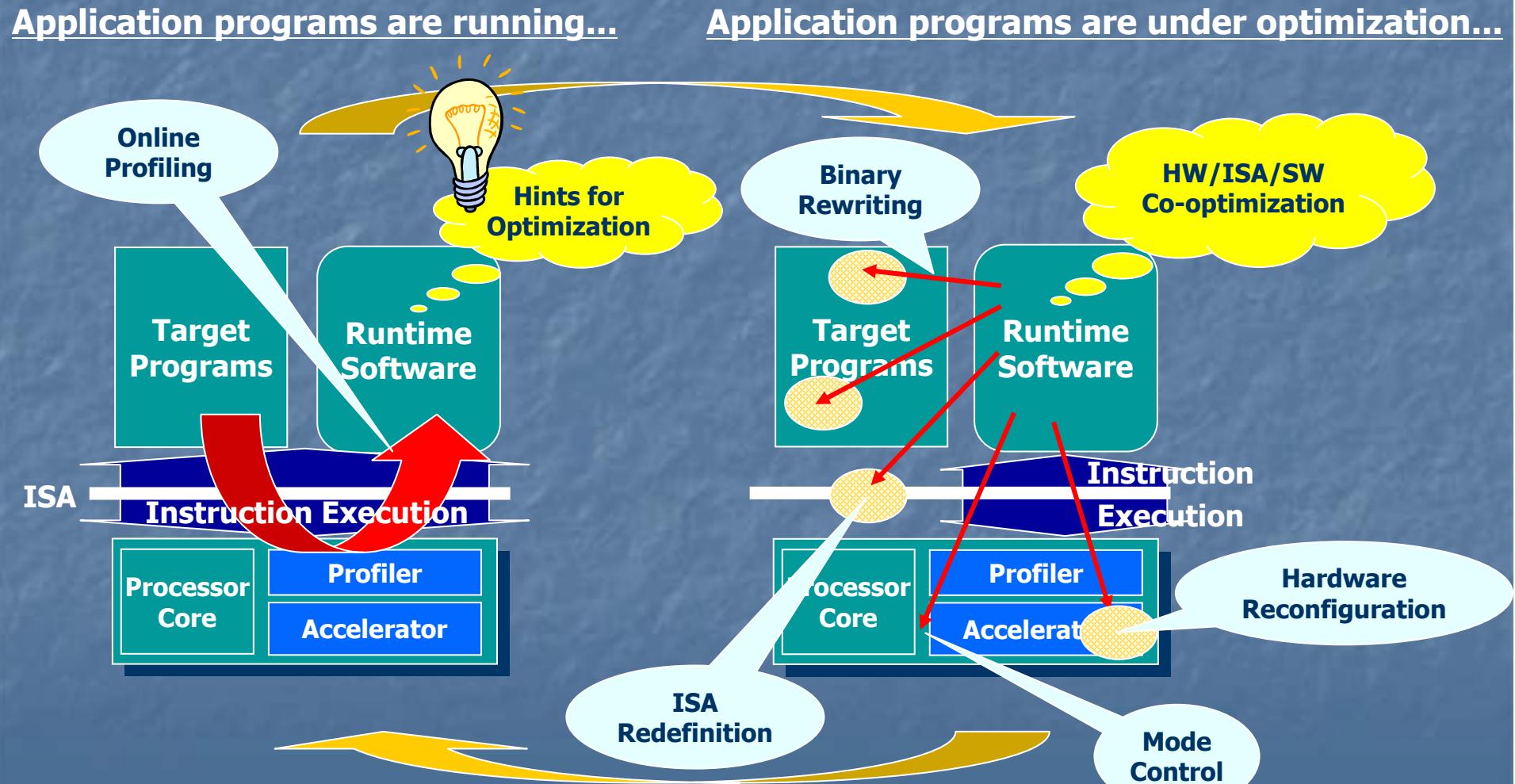
Victor Mauro Goulart Ferreira (\*2)

\*1: Director of Computing & Communications Center, Kyushu University

\*2: Ph.D. Candidate, Dept. of Informatics, Kyushu University

E-mail: arch@i.kyushu-u.ac.jp or kjm@acm.org

# JIT HW/ISA/SW Co-optimization would work like ...



# Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
- Part 2 (17:30-19:00 → 15:00-16:30)
  - On-demand Recomputation

# Tutorial Outline: Part 2

- On-demand Recomputation
  - Computing Centric Computation (CCC)
  - Memory Wall Problem
  - Performance-Critical Instructions
  - On-demand Recomputation
  - Performance Issues
  - Summary and Conclusions

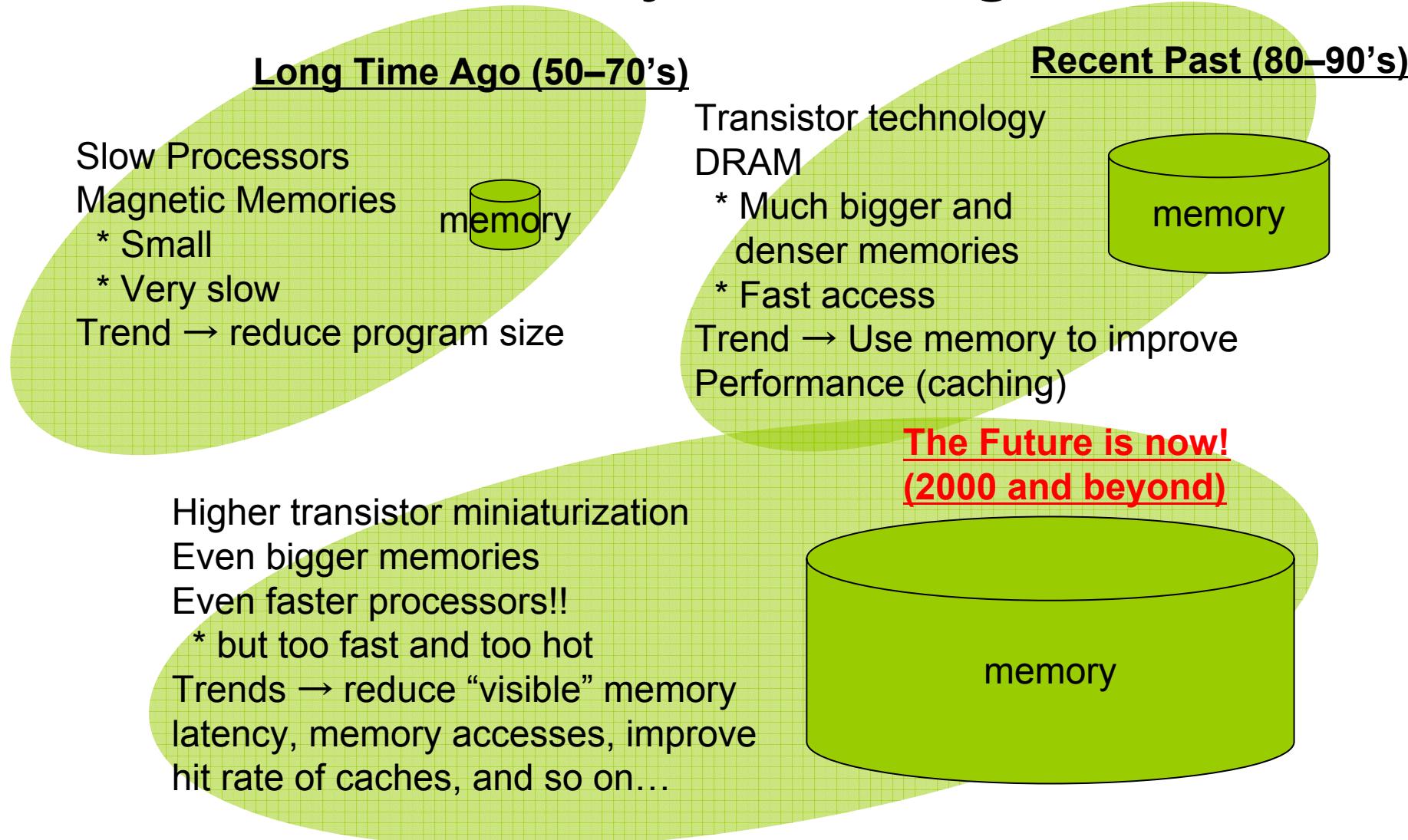
# Computing Centric Computation

**“Change the paradigm of  
Memory Centric Computation”  
Memory is a precious resource,  
so use it scrupulously!**

# Background and Motivation

- Semiconductor improvements and trends (ITRS 2003)
  - SoC, CMP, MT...
  - Embedded memories
  - Wireless communication and devices with multimedia
- Memory-processor performance gap
  - ➔ Memory wall problem (MWP)
- Performance-critical load instructions (or delinquent loads or troublesome loads)

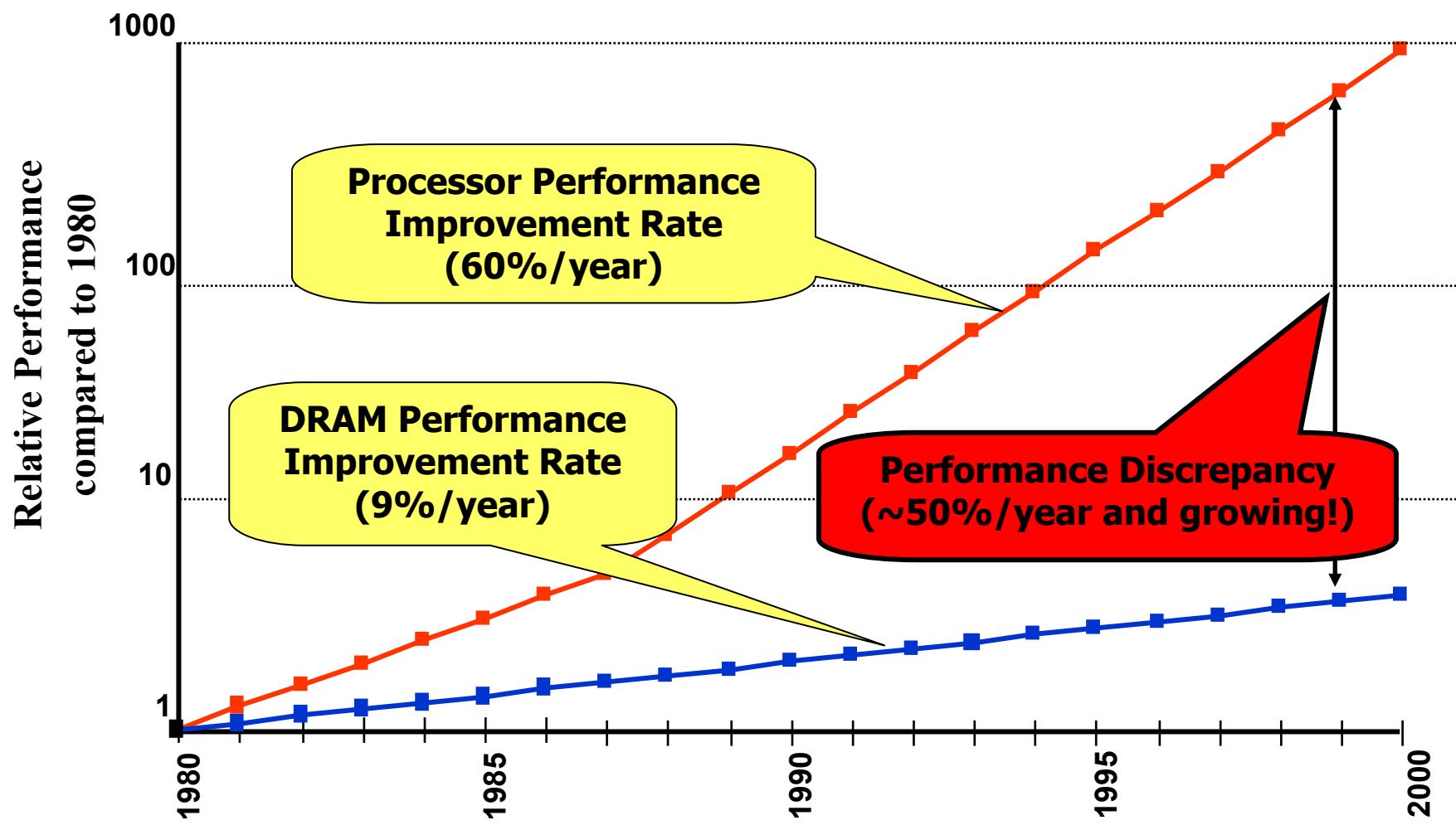
# Memory Paradigm



# Tutorial Outline: Part 2

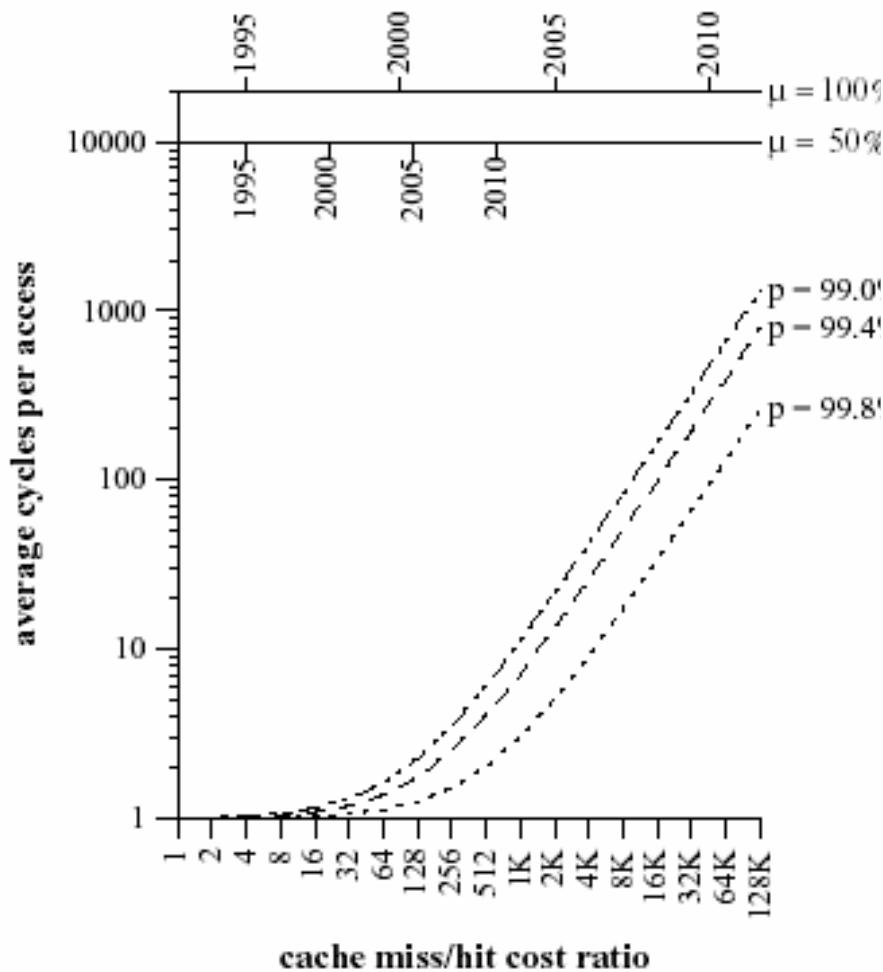
- On-demand Recomputation
  - ✓ Computing Centric Computation (CCC)
  - Memory Wall Problem
  - Performance-Critical Instructions
  - On-demand Recomputation
  - Performance Issues
  - Summary and Conclusions

# Processor vs. Memory Performance

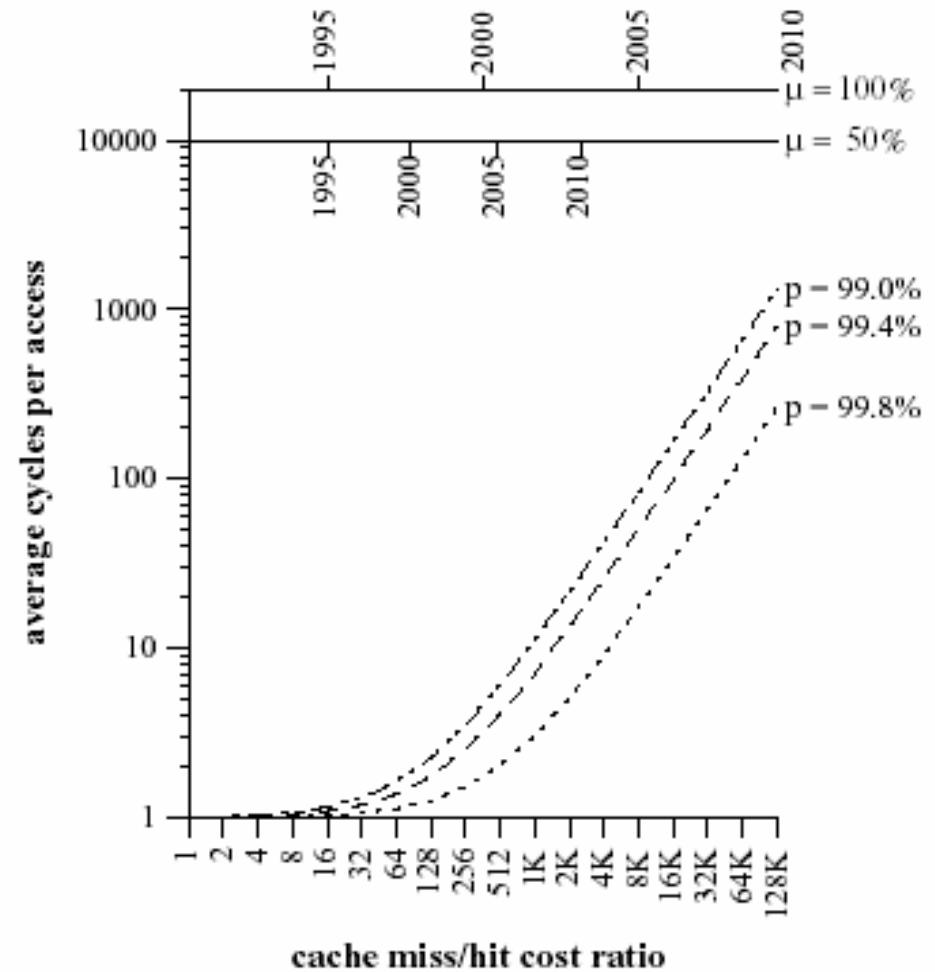


# Memory Wall Problem in 1995

- W.A. Wulf and S.A. McKee, *Hitting the Memory Wall: Implications of the Obvious*, Computer Architecture News, vol.23, no.1, pp.20-24, March 1995 -



**Trends for a Current Cache  
Miss/Hit Cost Ratio of 4**

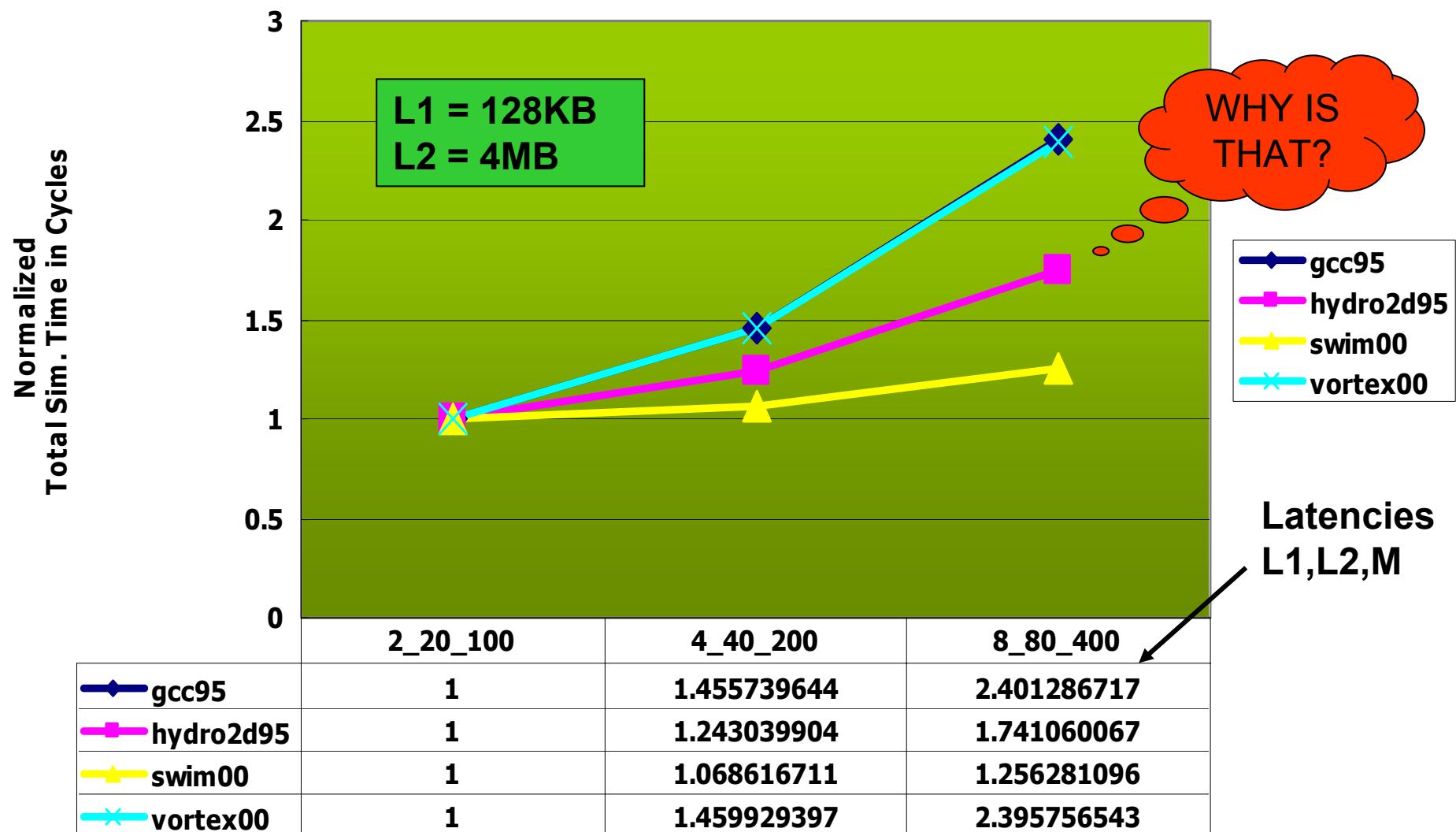


**Trends for a Current Cache  
Miss/Hit Cost Ratio of 16**

# Dealing with MWP

- Memory latency reduction
  - Caching
  - Prefetching
  - Speculative load
  - Memory compression
  - Embedded DRAM (e.g., PPRAM, IRAM, PIM, etc.)
- Memory latency hiding
  - Non-blocking caches
  - MT (multithreading)
  - SMT (simultaneous multithreading)

# Latency Impact on Computation Time



# Memory System Overhead

- The impact of the memory system on execution time is composed of:
  - Number of memory accesses / Cache behavior
    - Can be improved by modern compiler techniques; **however, the primary limit on compiler's ability to improve memory behavior is its imperfect knowledge about the run-time behavior of the program**
  - Memory access latency
    - Physical limitation but can be dealt with memory latency reduction (caching – temporal locality) and hiding, or tolerating schemes (change the “perceived” memory latency)

# Tutorial Outline: Part 2

- On-demand Recomputation
  - ✓ Computing Centric Computation (CCC)
  - ✓ Memory Wall Problem
    - Performance-Critical Instructions
    - On-demand Recomputation
    - Performance Issues
    - Summary and Conclusions

# Critical Instructions

- Performance critical instructions [Roth: HPCA2001] (*delinquent* and *troublesome* inst. is also seen in the literature)
  - Loads that are likely to miss in the cache
  - Branches that are likely to be mispredicted
- In DDMT (data-driven multithreading), the computation of these critical instructions are computed by another thread in advance to the main thread so the data is “prefetched” by the sub-thread and stays ready to be used by the main thread
- Data-driven pre-execution scheme as a unified general-purpose performance engine
- **Memory latency effects on computation time appear mainly on these critical LOADs!!**

# Simulation Environment

- SimpleScalar v.3.0 and SPEC2000 benchmarks
  - Distinct IL1 and DL1 and unified UL2
  - Five distinct cache configurations (L1, L2):
    - (32K,128K), (64K,256K), (128K,512K), (128K,1M), (128K,4M)
    - L1 (32B line size, 4 way); L2 (64B line size, 8 way)
    - Latencies: L1=8; L2=80; M=400 (units are clock cycles)
- Measurement of;
  - Cache miss count
  - Cache miss rate
  - Number of instructions (PC) and addresses (EA – effective address) grouped by miss counts

## Profiling format

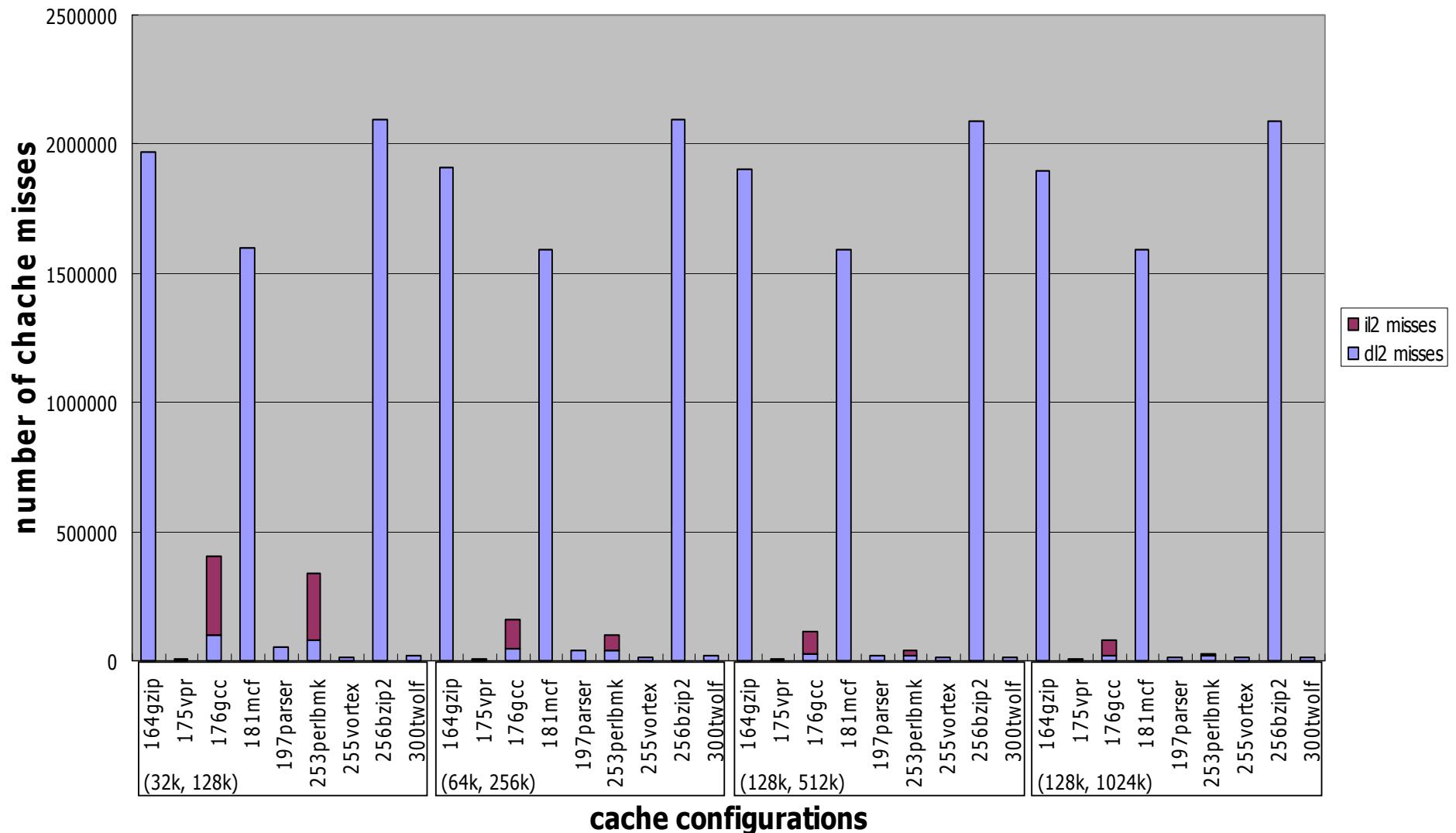
```
<cache ID {il1,d1l1,ul2{d/i}}, R/W, PC, EA, time (clock #), inst. mnemonic (assembly)>
```

# Cache Miss Count and Miss Rate for SPECint2000 and SPECfp2000

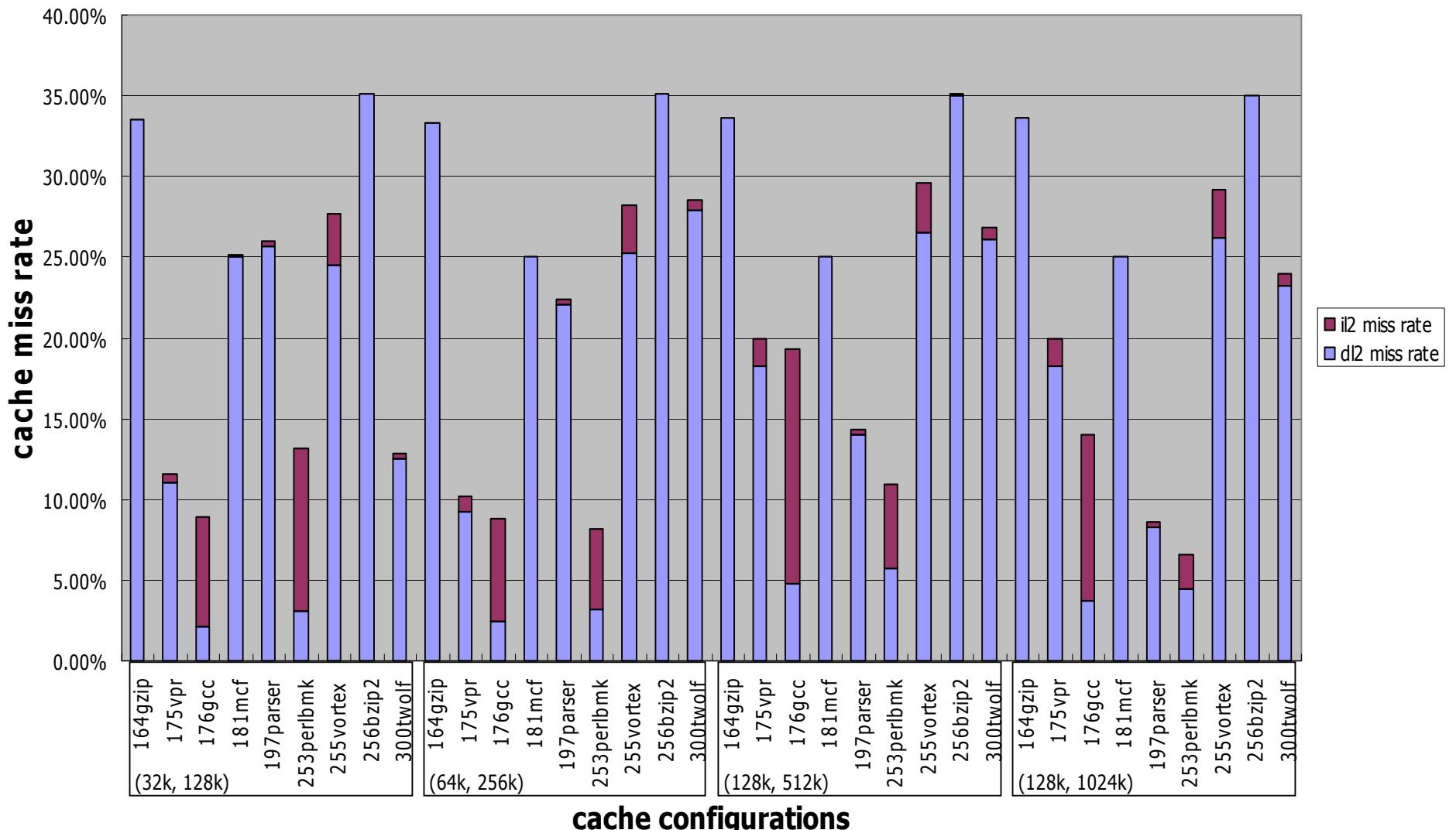
# Results

- For DL1:
  - For integer benchmarks, *gzip*, *mcf*, *vortex*, *bzip2* had  $\approx 8\%$  miss rate the others, less than 1%
  - *mgrid* had about 25% miss rate and *apsi* a bit more than 14%, *applu*, about 4% while other FP benchmarks less than 2%.
- UL2 results as follows...

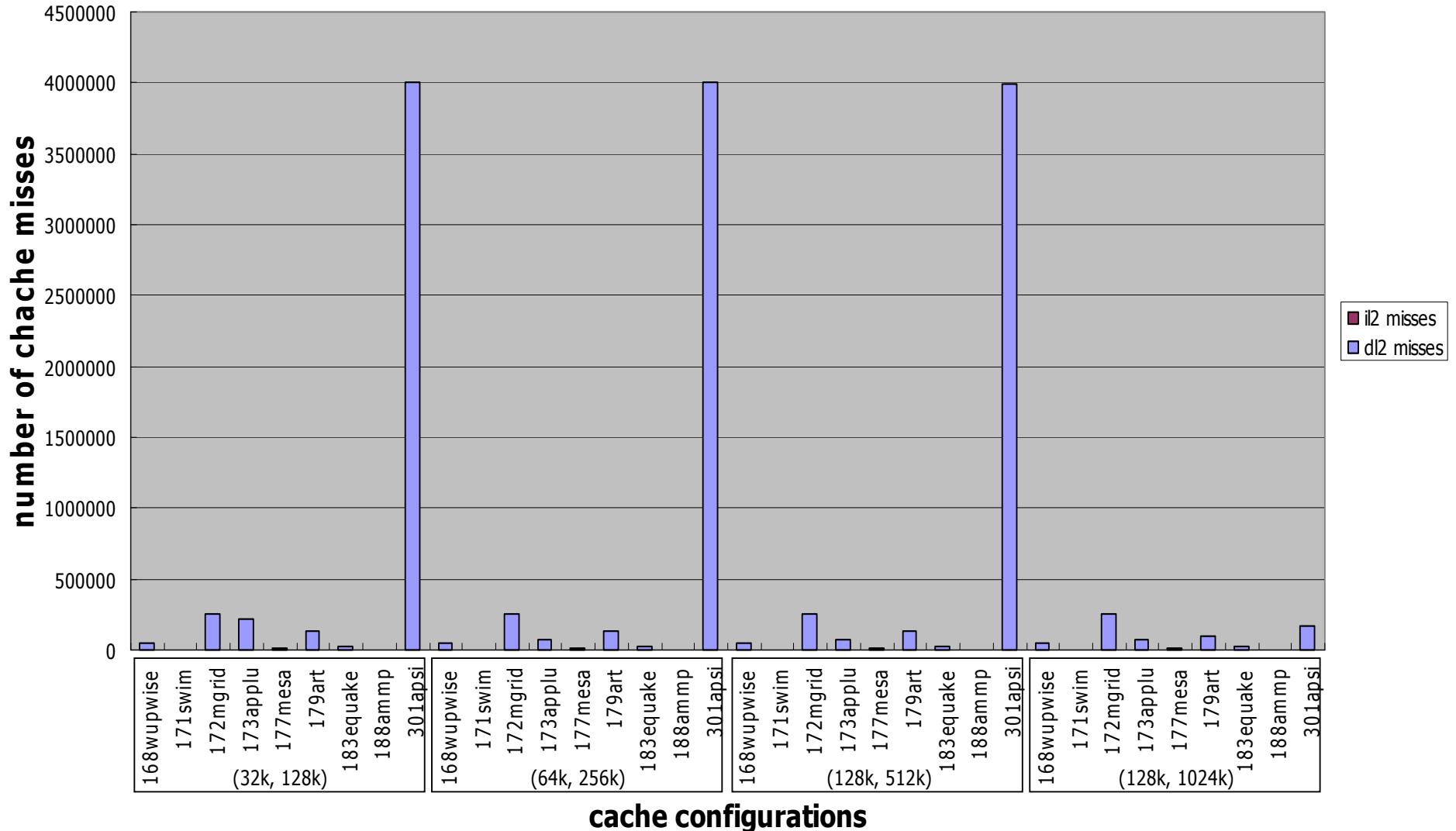
## ul2 misses (int)



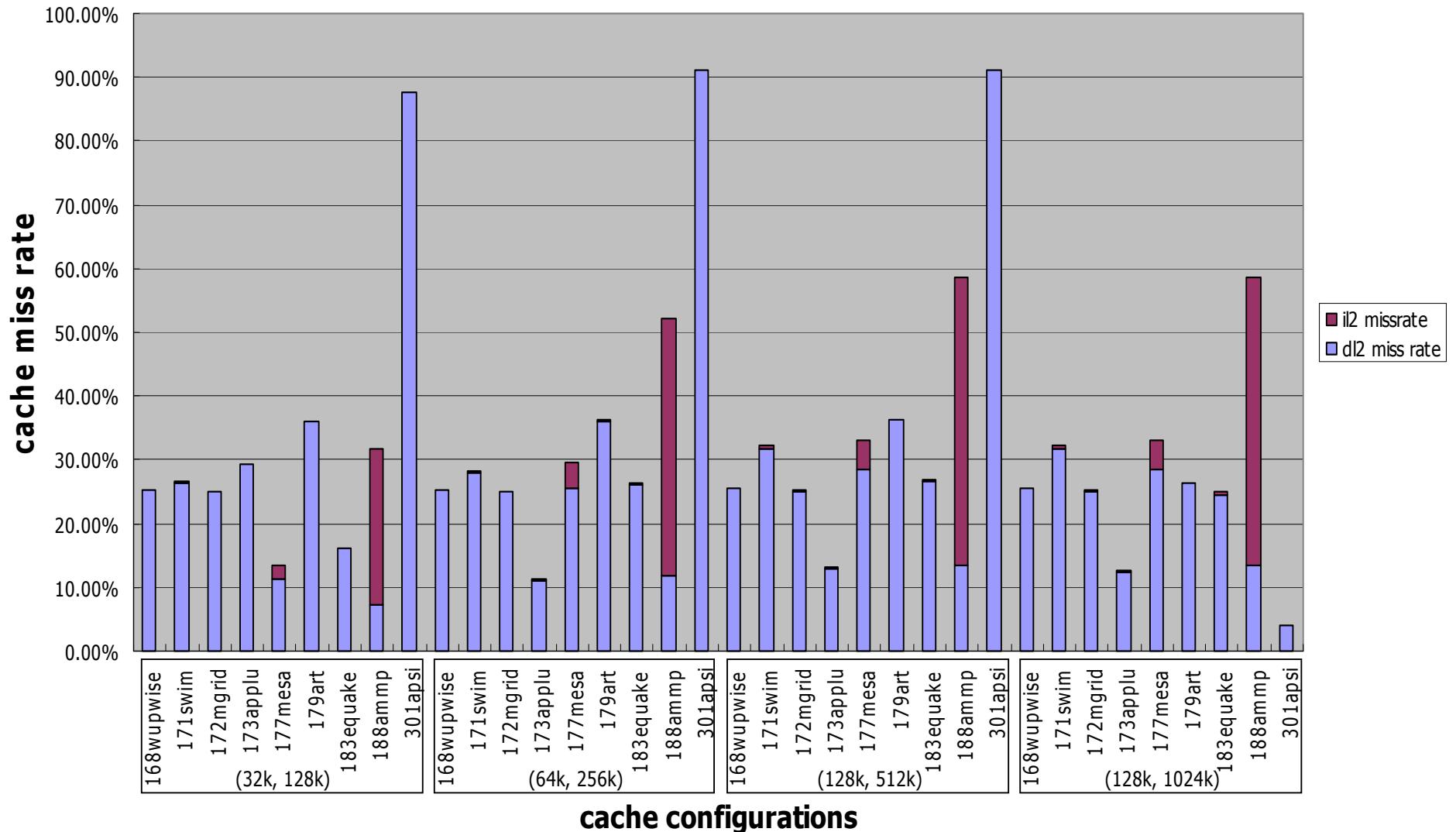
## ul2 miss rate (int)



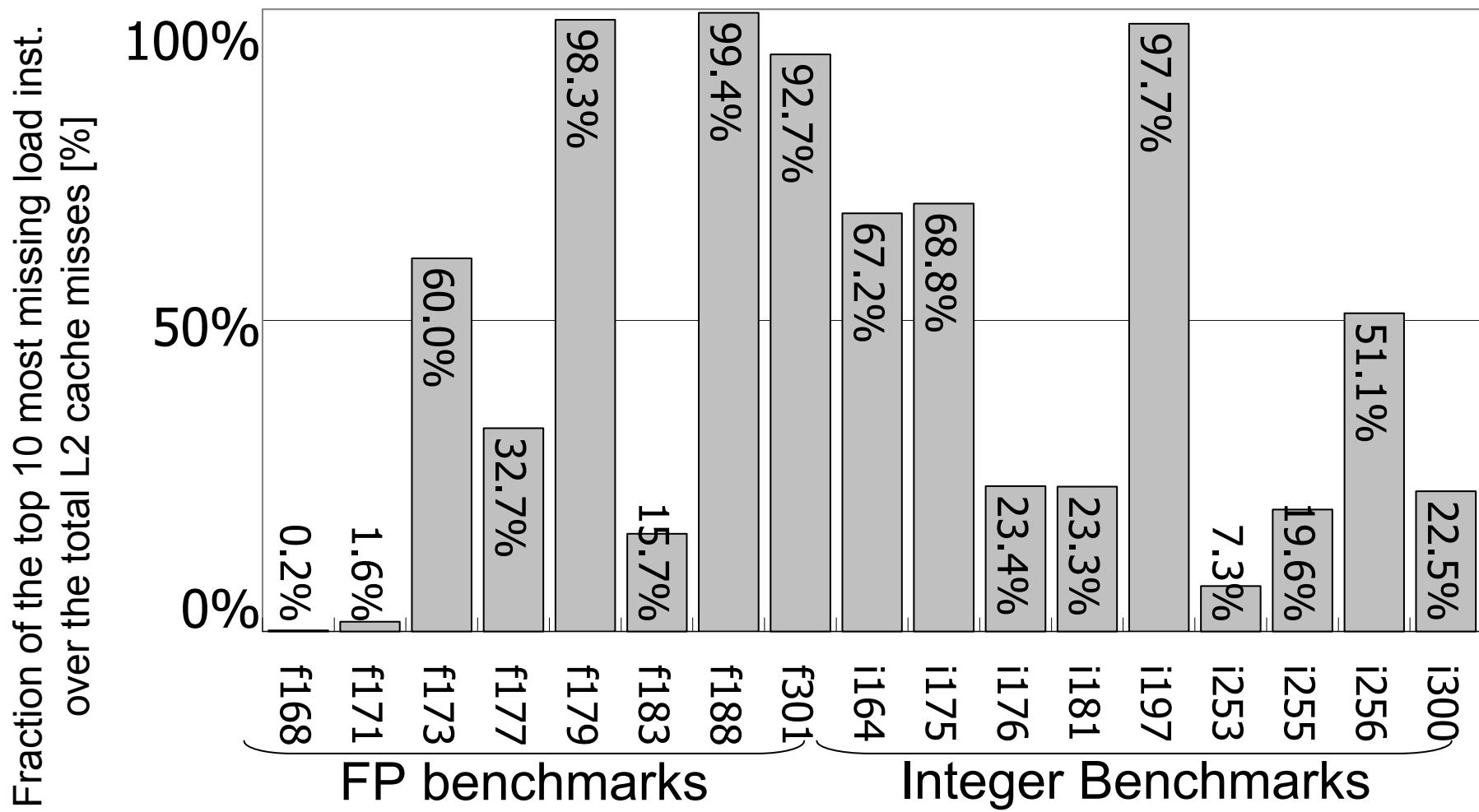
## ul2 misses (fp)



## ul2 miss rate (fp)

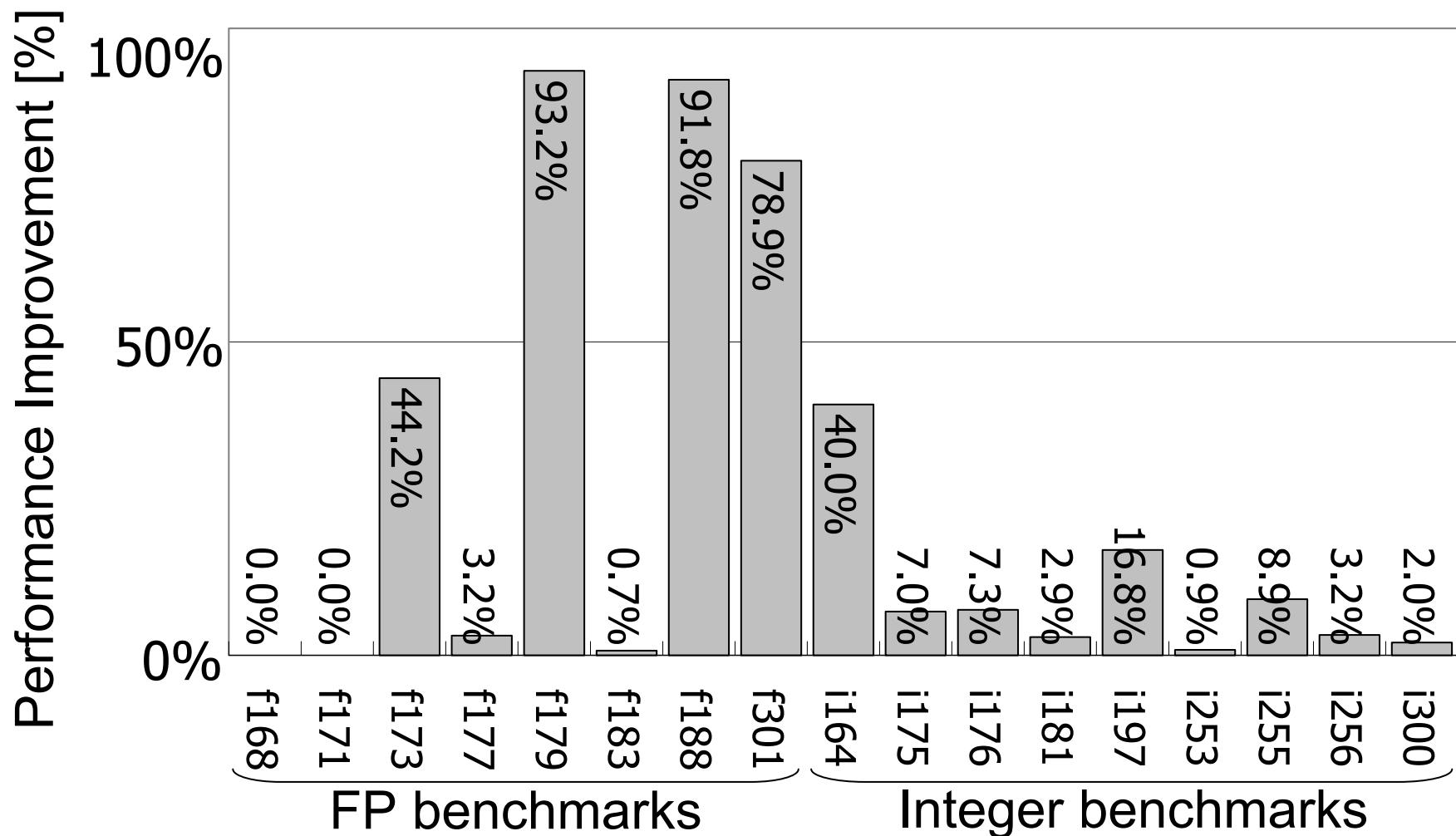


# How Much Do Top 10 Critical LOADs Occupy L2 Cache Misses



@ SPEC CPU 2000 benchmarks on Simplescalar 3.0d. (50M Forward, 10M Exec, L1=32k, L2=128k.)

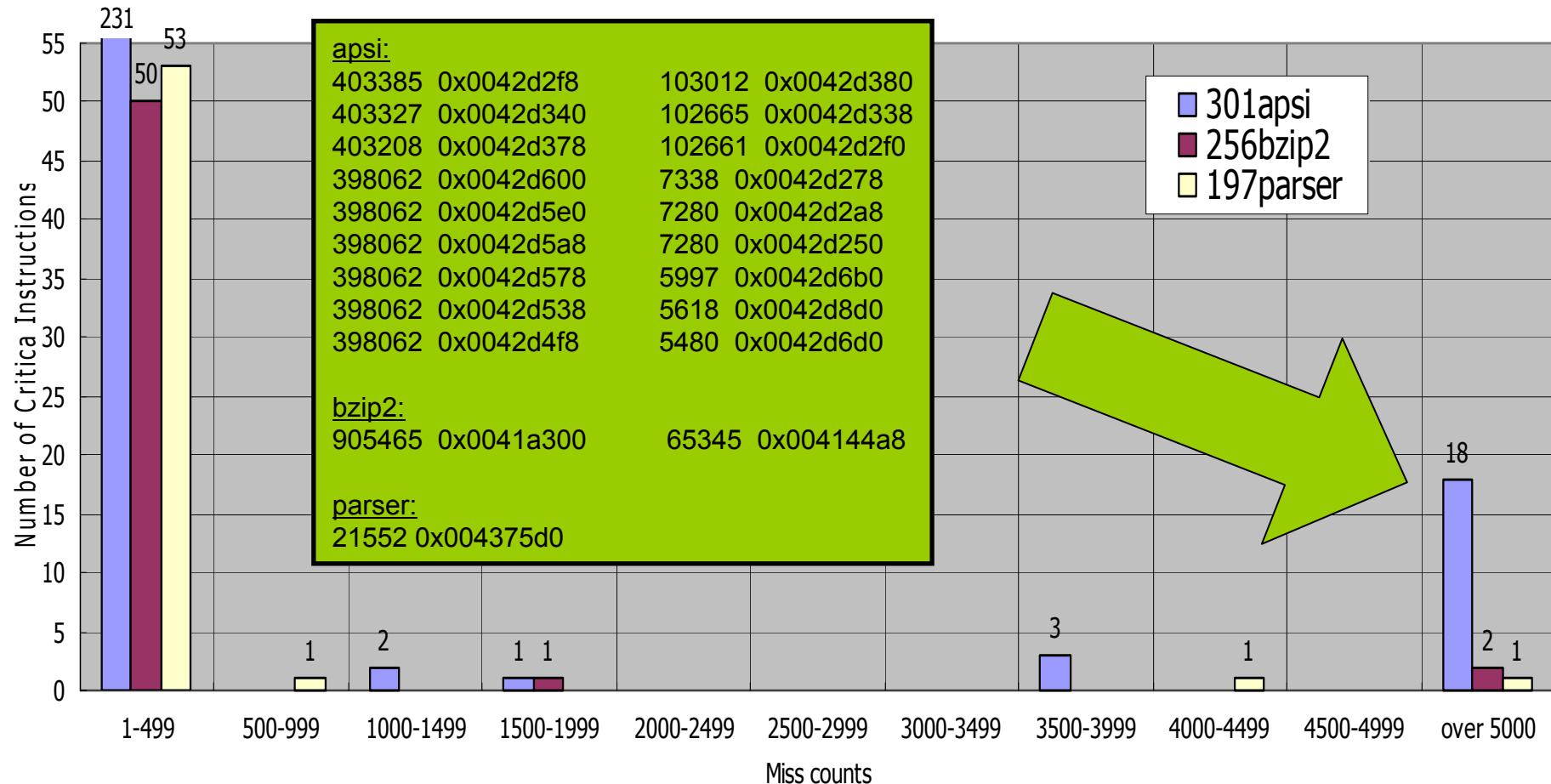
# How Much Can You Improve the Performance by Eliminating Top 10 Critical LOADs



# Qualitative and Quantitative Study of Critical LOADs in SPEC2000 Benchmarks

# Miss Count of Critical LOADs for L2 Cache

Data miss count for L2 (L1=32K, L2=128K)



# Tutorial Outline: Part 2

- On-demand Recomputation
  - ✓ Computing Centric Computation (CCC)
  - ✓ Memory Wall Problem
  - ✓ Performance-Critical Instructions
    - On-demand Recomputation
    - Performance Issues
    - Summary and Conclusions

# On-demand Recomputation

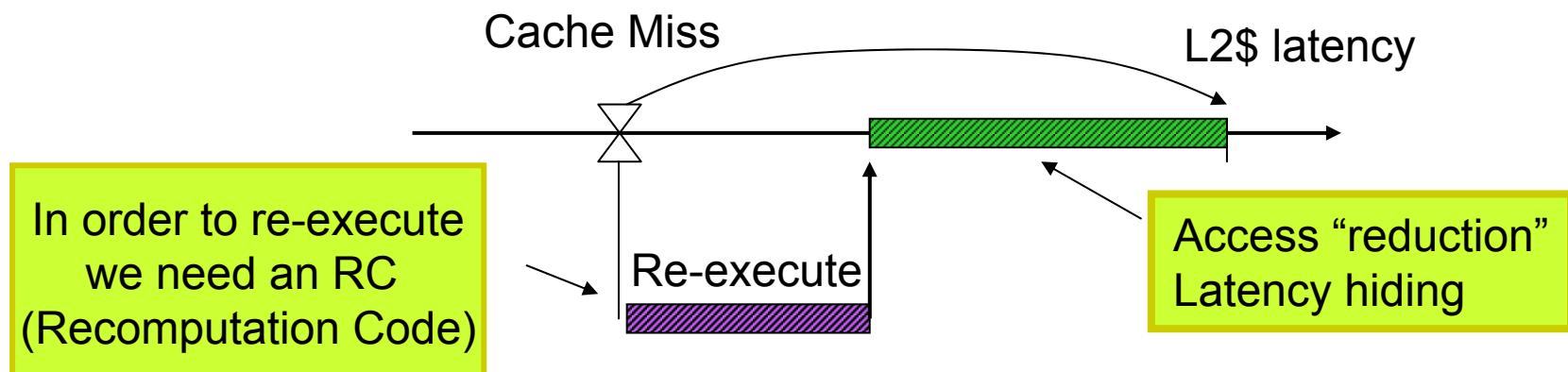
**“If *that data* is not in the cache...**

**No problem, recompute it!”**

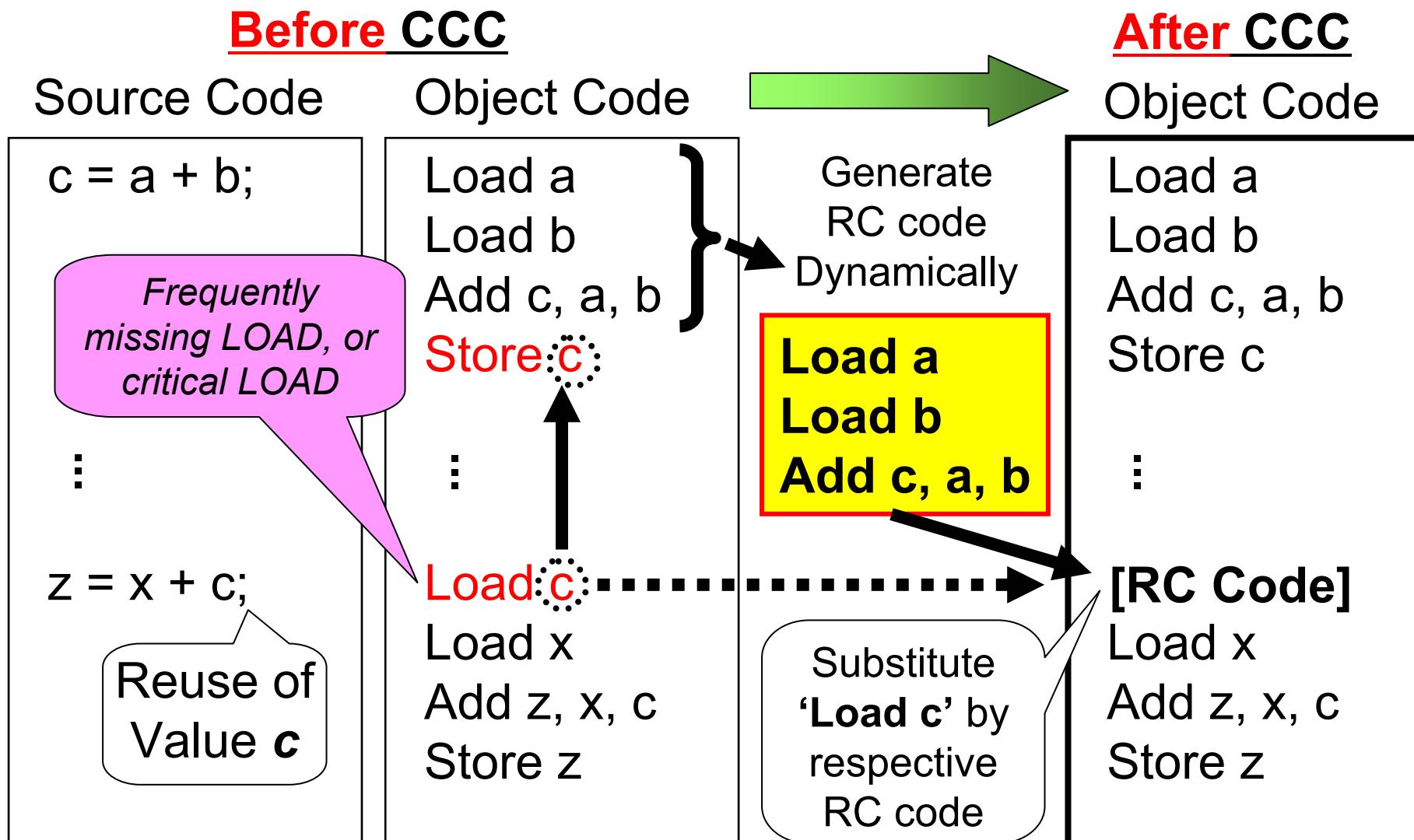
**No more data starvation due to long  
memory latencies<sup>\*</sup>...**

# On-demand Recomputation

- Alleviate the negative impact of memory latency by regenerating or recomputing the data to load when a cache miss occurs
- Reduce memory accesses and visible latency of a cache miss. Instead of going to a far slow memory, try to regenerate the missed data on-the-fly
- Critical LOADs are good candidates to apply

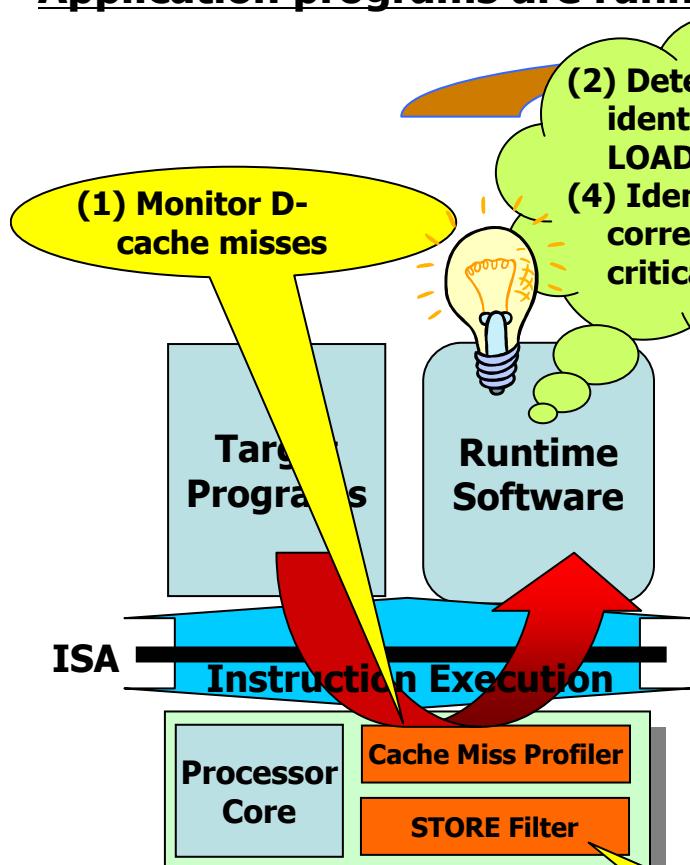


# On-demand Recomputation

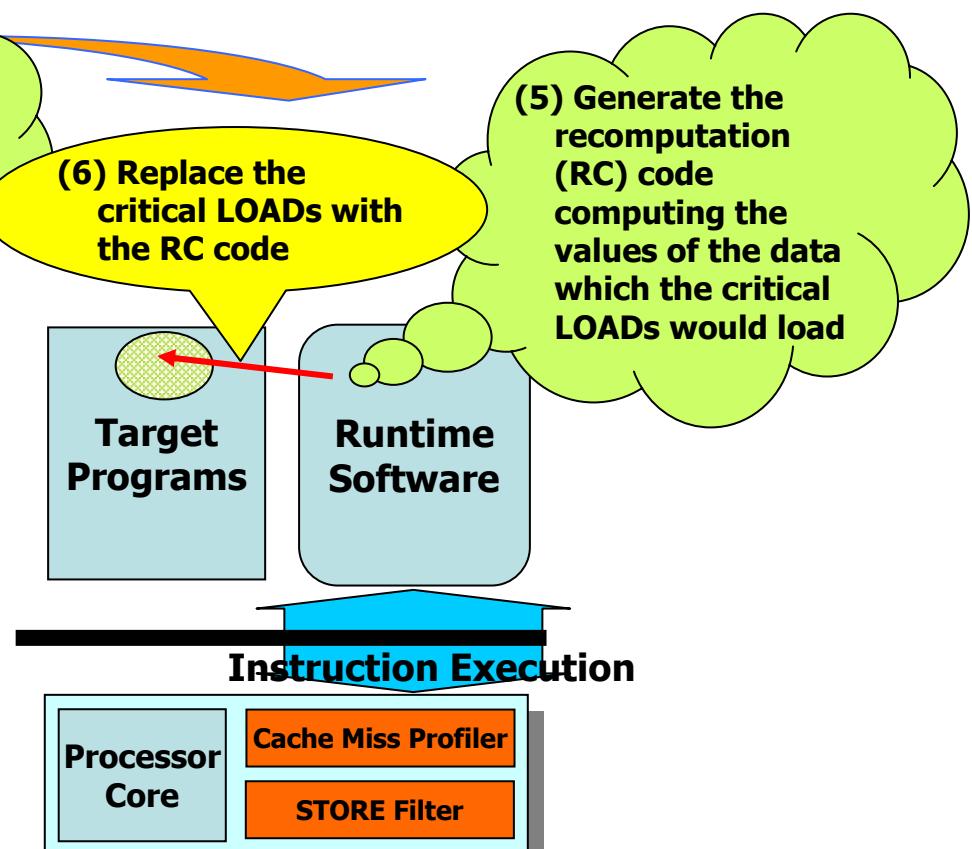


# How Can You Implement On-demand Recomputation

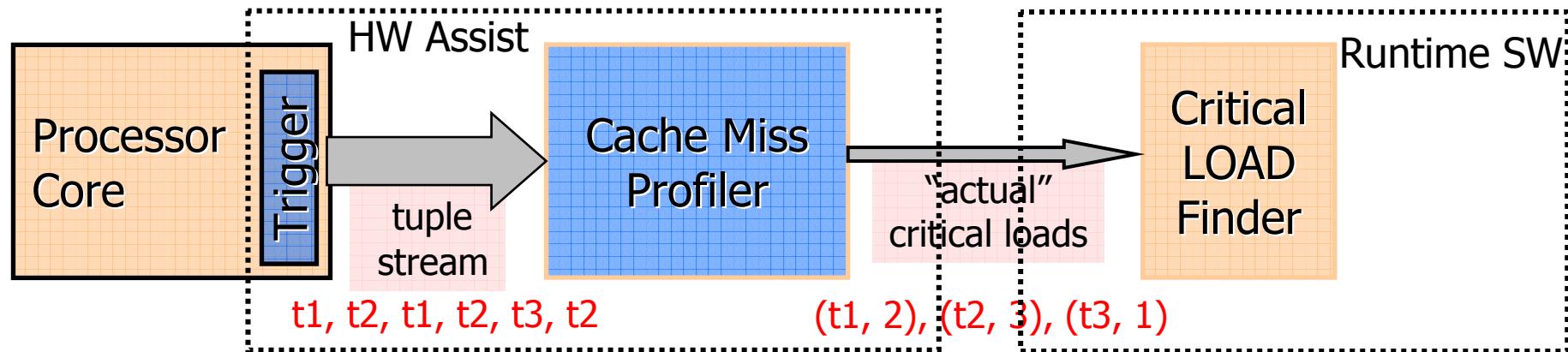
Application programs are running...



Application programs are under optimization...

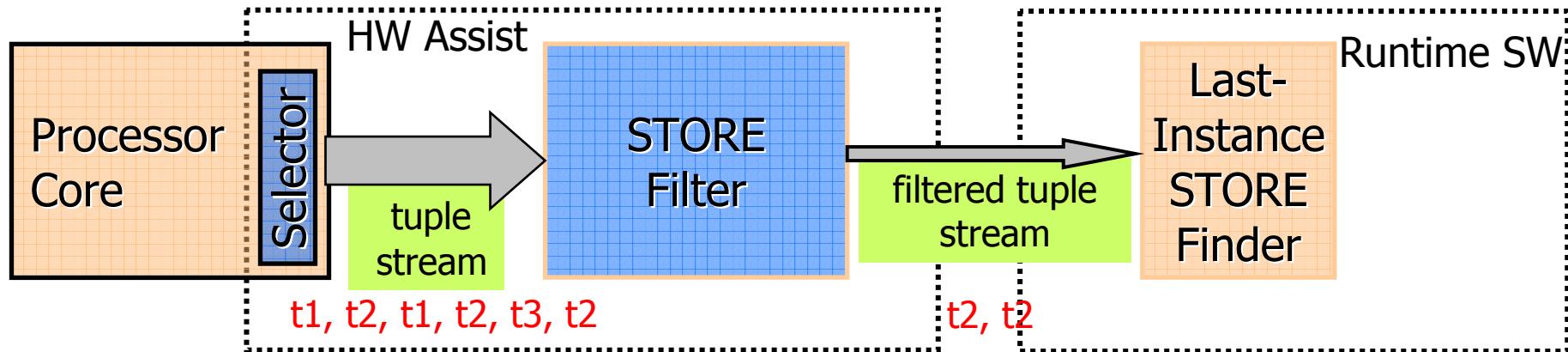


# How to Detect and Identify Critical LOADs



- **Trigger**
  - When a cache miss occurs, builds up tuples
- **Tuple**
  - $\langle lia, la \rangle$ 
    - lia: load instruction address
    - la: load address
- **Cache Miss Profiler**
  - Counts the tuples, and compresses them in the message form
  - Sends the messages to critical LOAD finder
- **Messages**
  - (tuple, count)
- **Critical LOAD Finder**
  - Accumulates the messages
  - Classify critical loads according to a given threshold

# How to Detect Last Instance STORE Corresponding to Critical LOADs



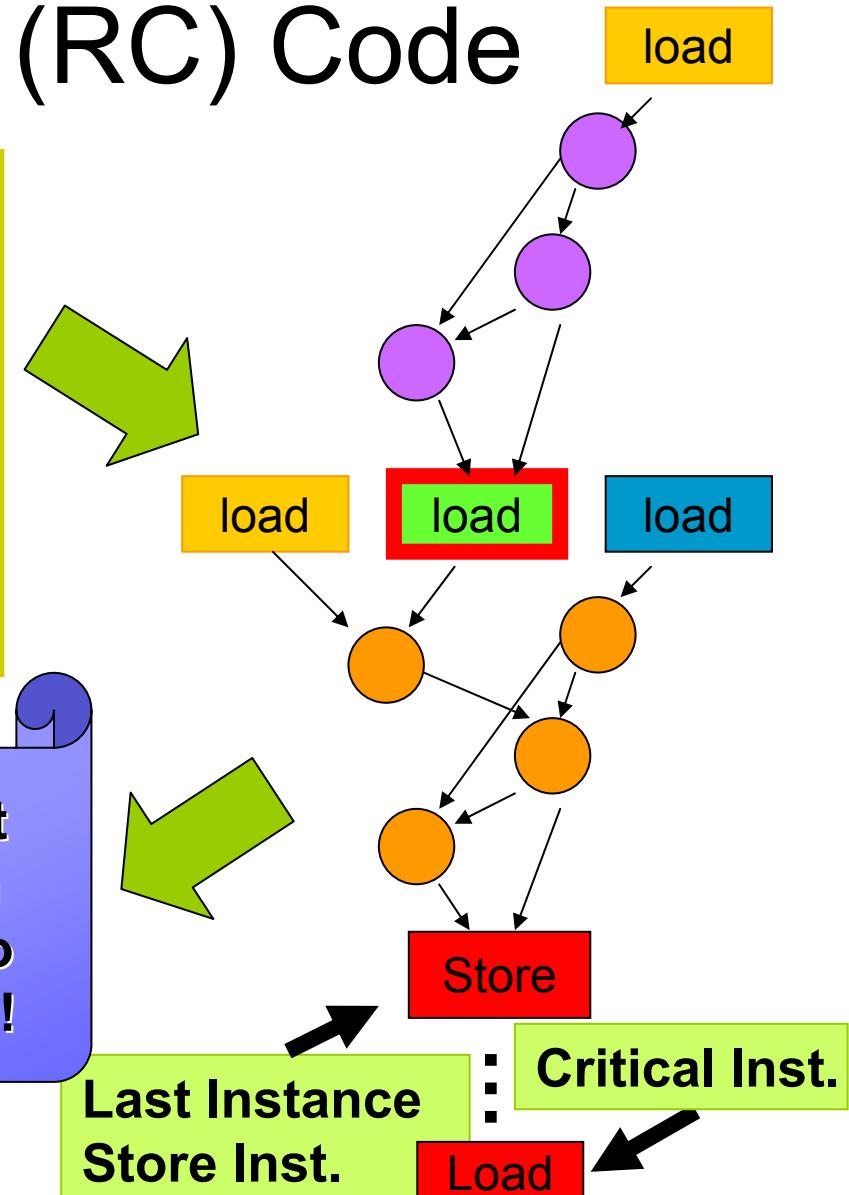
- **Selector**
  - Buffers all the executed store instructions
  - Sends them as tuples to STORE filter
- **Tuple**
  - $\langle sia, sa \rangle$ 
    - sia: store instruction address
    - sa: store address
- **STORE Filter**
  - Keep track of critical load instructions in the list of  $\langle lia, la \rangle$
  - Get the tuples, and then searches the list associatively with the key (sa)
  - Sends the matched tuples to last-instance STORE finder
- **Last-Instance STORE Finder**
  - Accumulates the filtered tuples
  - Finds the last instance of STOREs corresponding a critical load

# How Can You Build Recomputation (RC) Code

```
...
00419e80 addu $v0[2],$v0[2],$a0[4]
00419e88 sll $v0[2],$v0[2],0x1
00419e90 subu $v0[2],$s4[20],$v0[2]
00419e98 addiu $v0[2],$v0[2],48
00419ea0 lw $v1[3],4($v0[2])
...

```

In case of a miss in **lw**, substitute it for the sequence of instructions in the data-flow graph with respect to the **last instance store instruction!!**

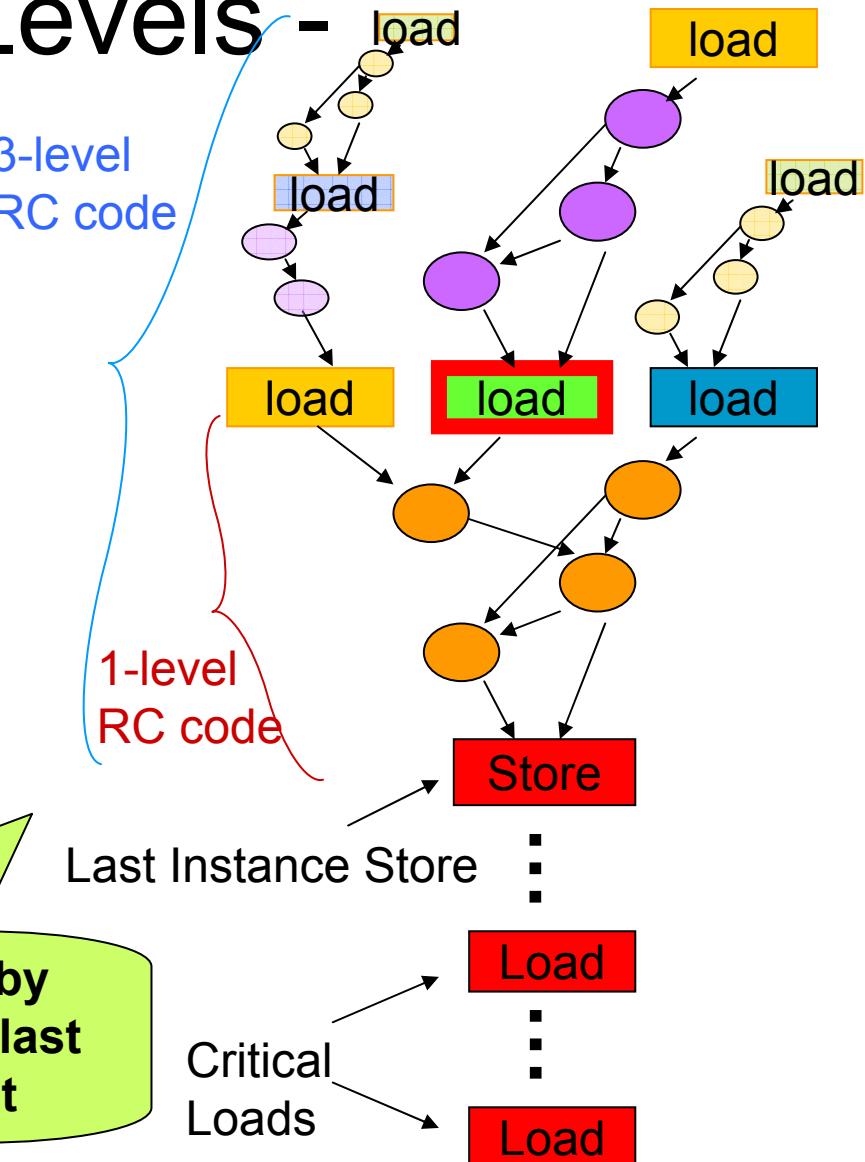


# Recomputation (RC) Code

## - Higher Levels

- Terms definition:
  - Critical Load: loads that miss in the cache (high memory latencies)
  - Last Instance Store: the store inst that lastly generated the data actually being referred by the critical load
  - 1-level RC code: the RC code (tree) from the last instance store to the first load inst
  - N-level RC code: beyond a given leaf load continue generating RC code for that data up to N-levels.

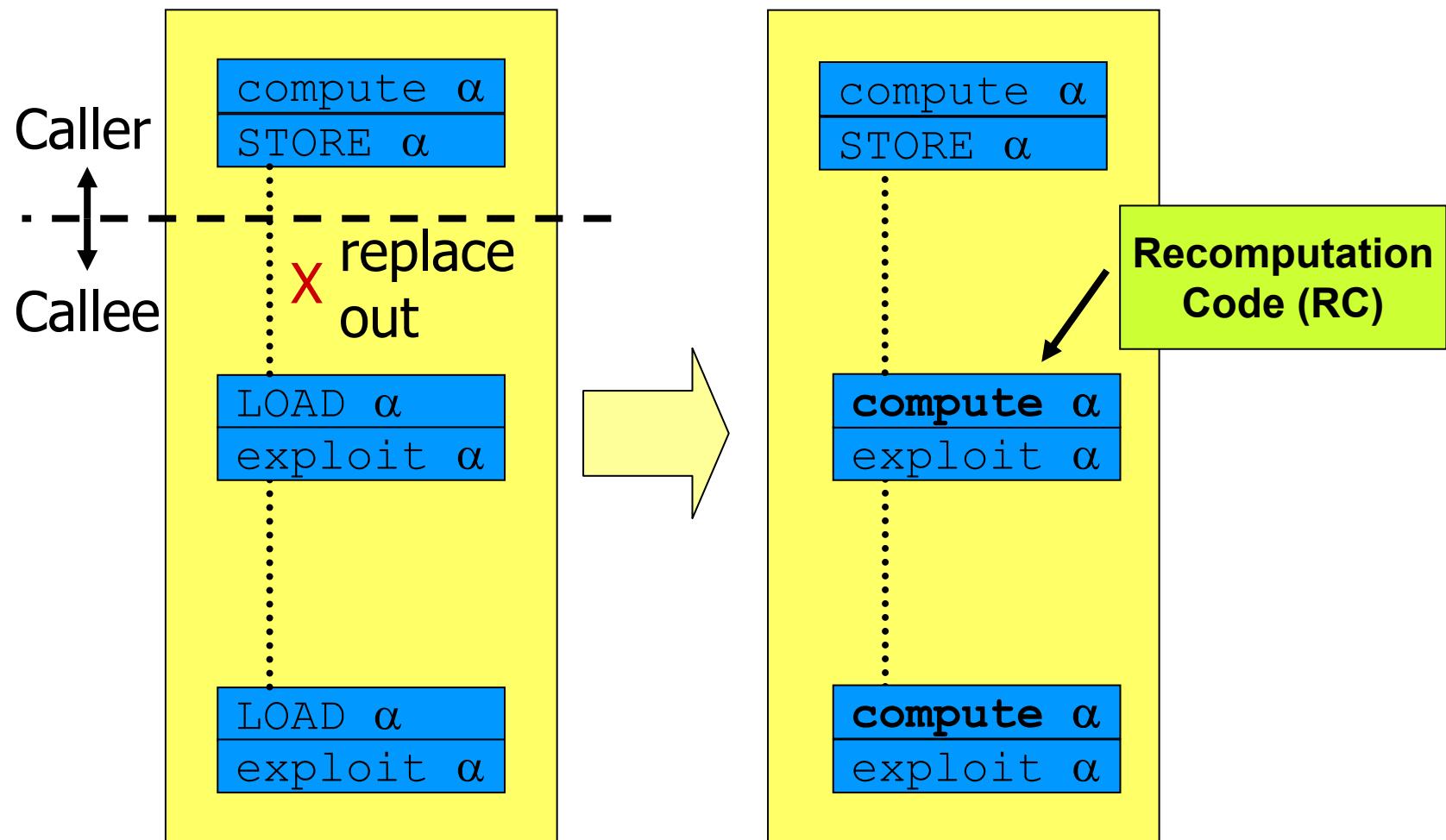
**Recomputation code** is generated by backtracking or traversing from the last instance store to the leaf load inst



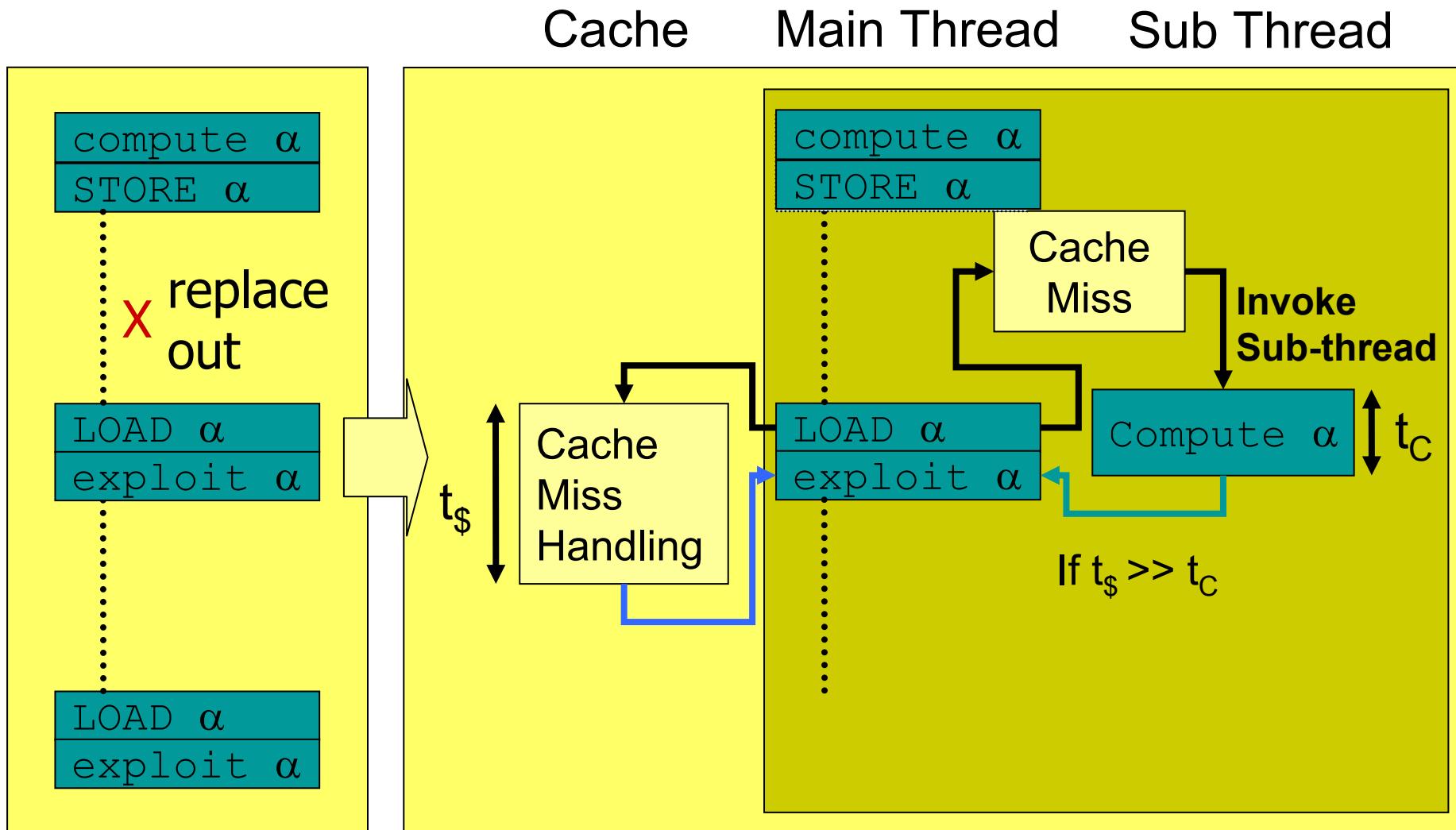
# How Can You Execute RC Code?

- On the main processor
  - On the same thread
    - Replace the critical load instruction with the RC code by means of dynamic binary rewriting, and then execute it
    - On another thread (helper thread) in the case of SMT
      - Trigger the execution of the RC code on the helper thread if the critical load instruction causes a cache miss
  - On another (co-)processor in the case of CMP
    - Trigger the execution of the RC code on the helper processor if the critical load instruction causes a cache miss

# Possible Implementations: On the Same Thread



# Possible Implementations: On Another Thread

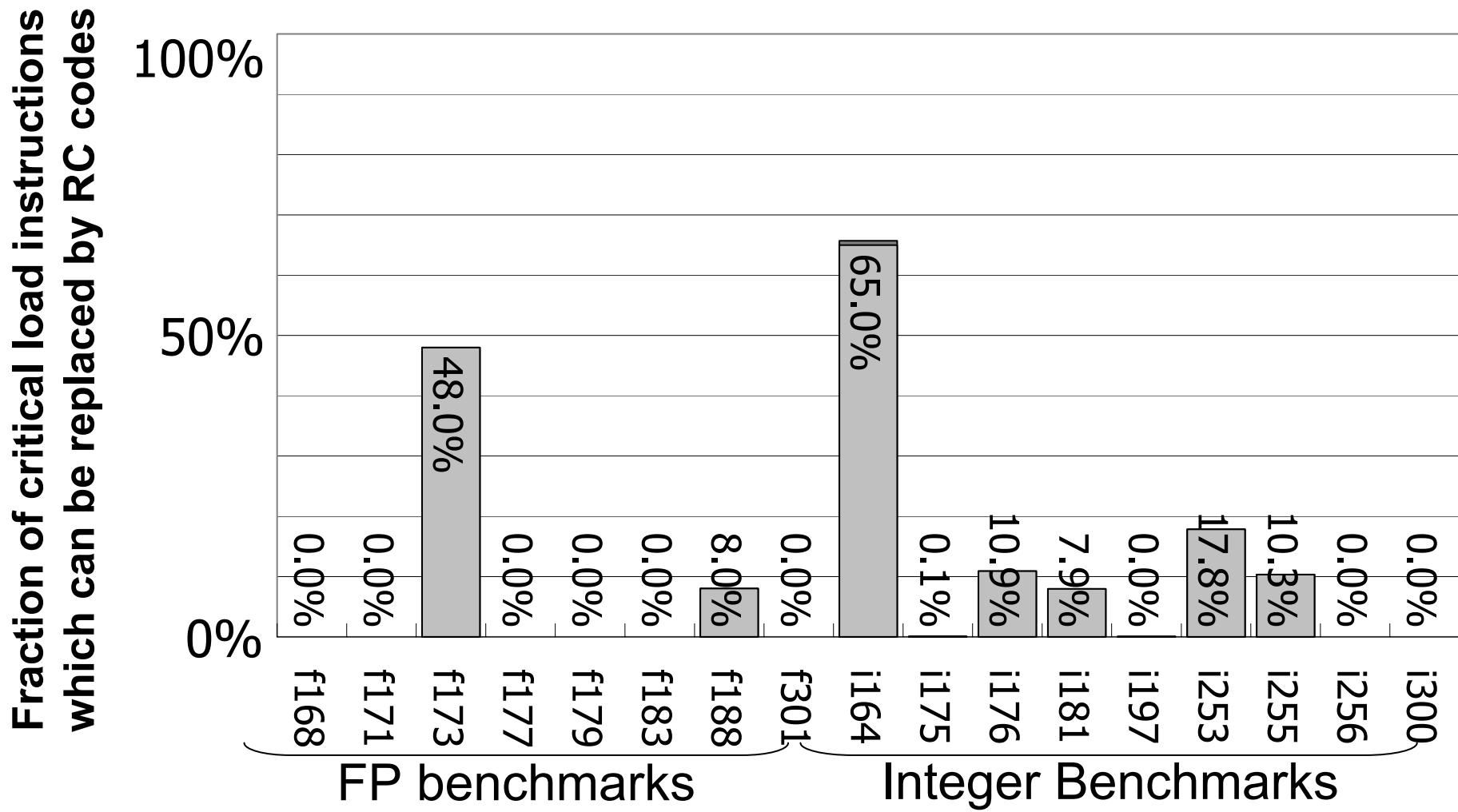


# Tutorial Outline: Part 2

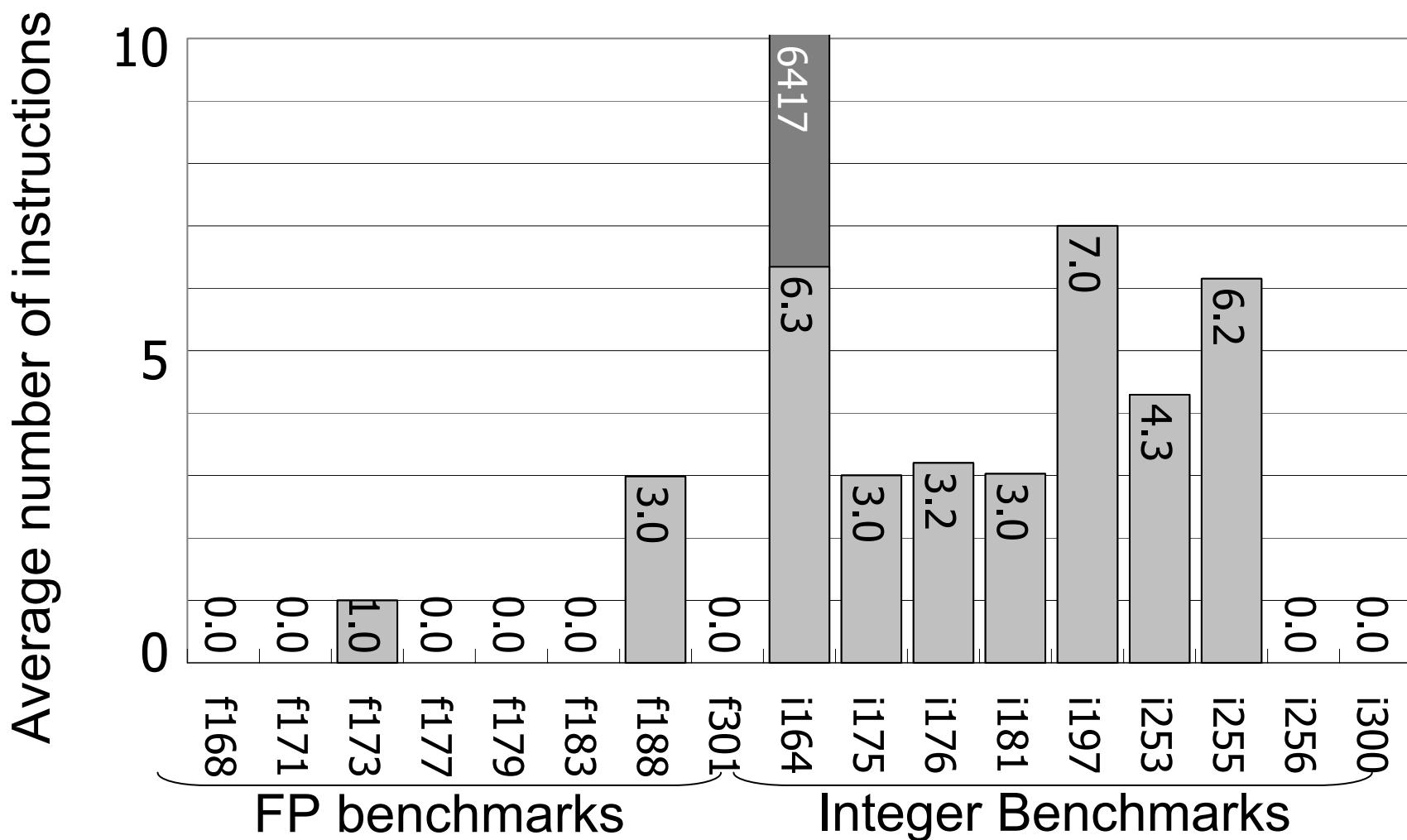
- On-demand Recomputation
  - ✓ Computing Centric Computation (CCC)
  - ✓ Memory Wall Problem
  - ✓ Performance-Critical Instructions
  - ✓ On-demand Recomputation
    - Performance Issues
    - Summary and Conclusions

# Performance Issues (1/5):

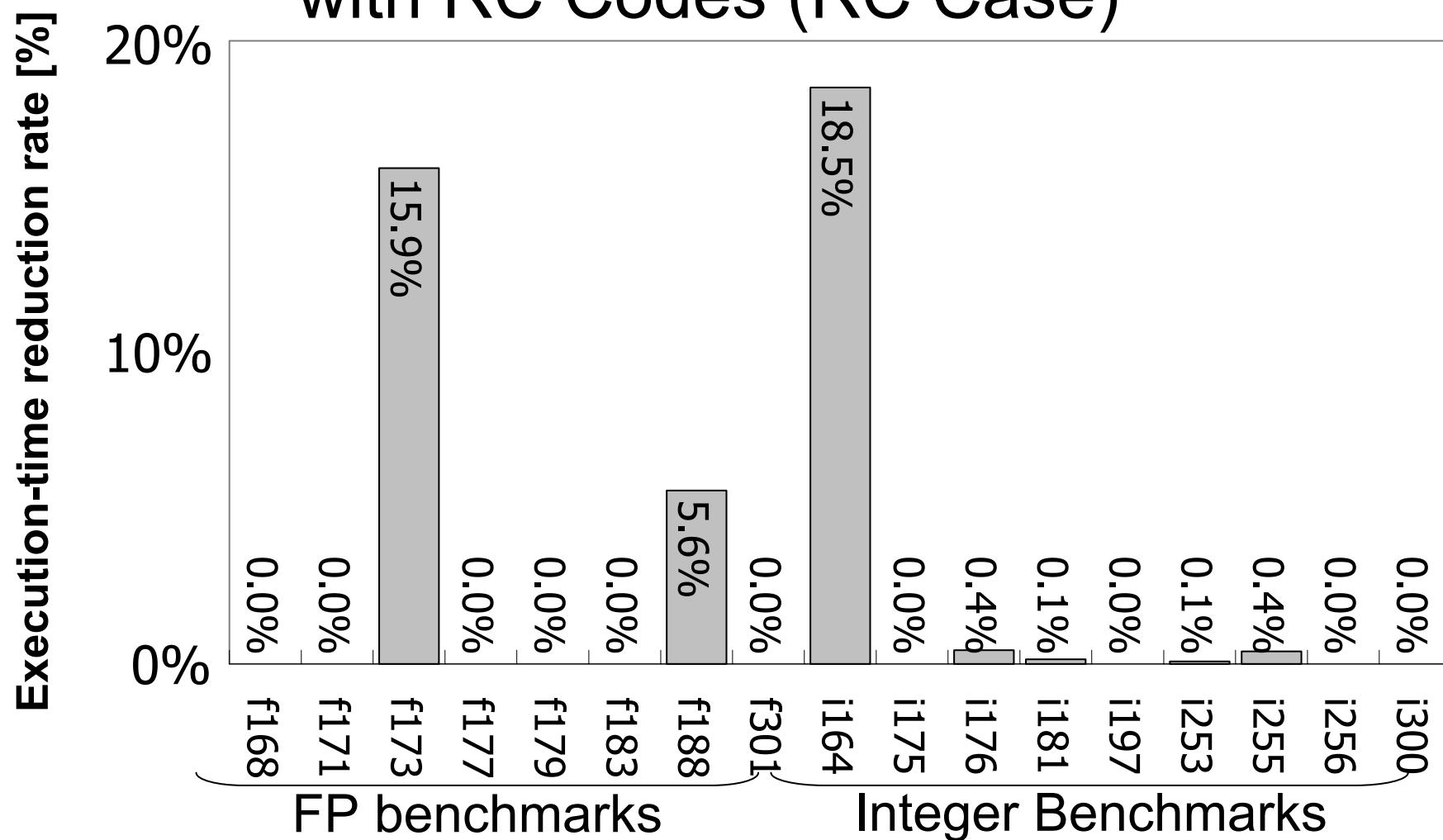
## How Many Critical LOADs Can Be Replaced by RC Codes?



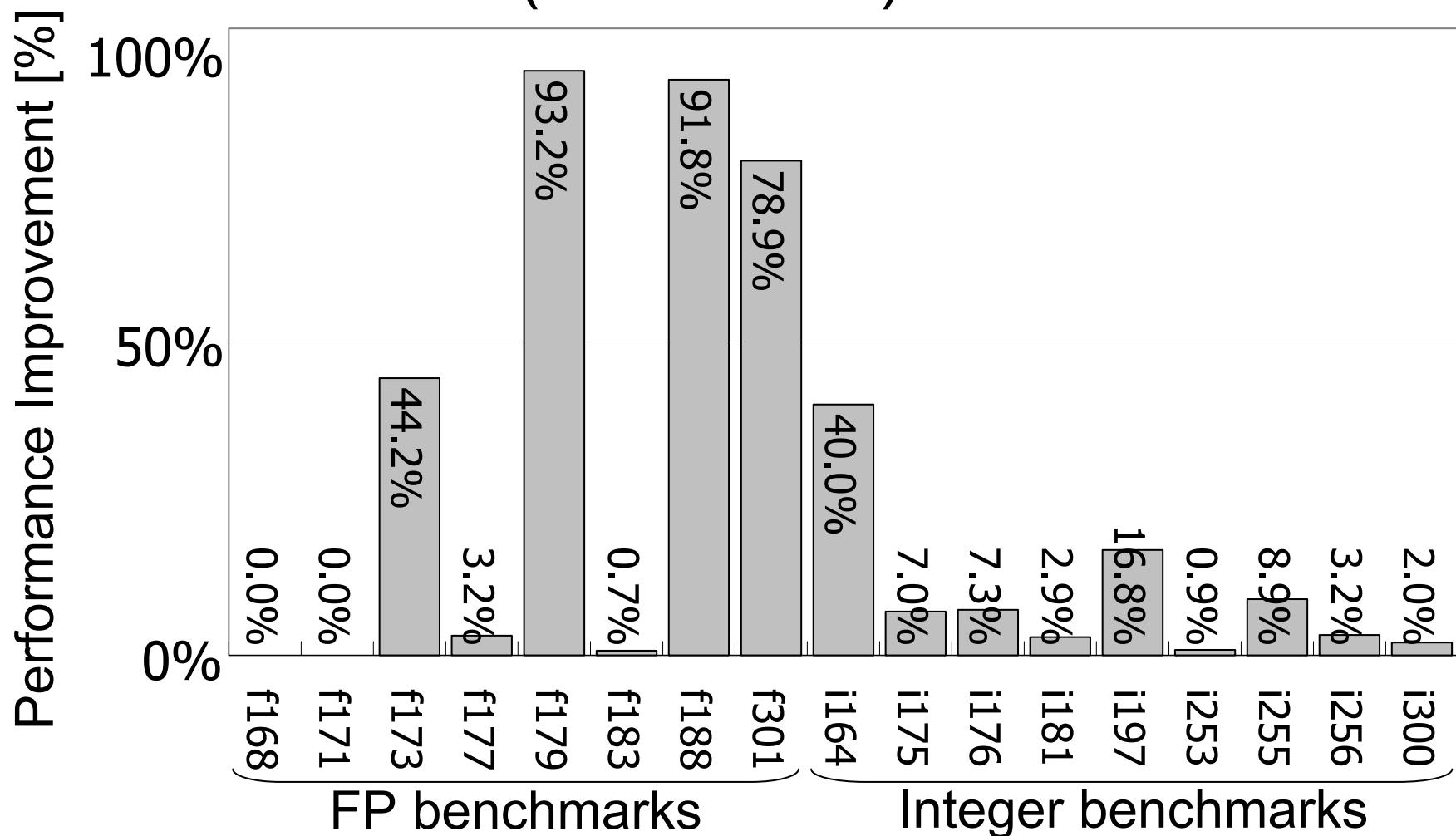
# Performance Issues (2/5): Average RC Code Size



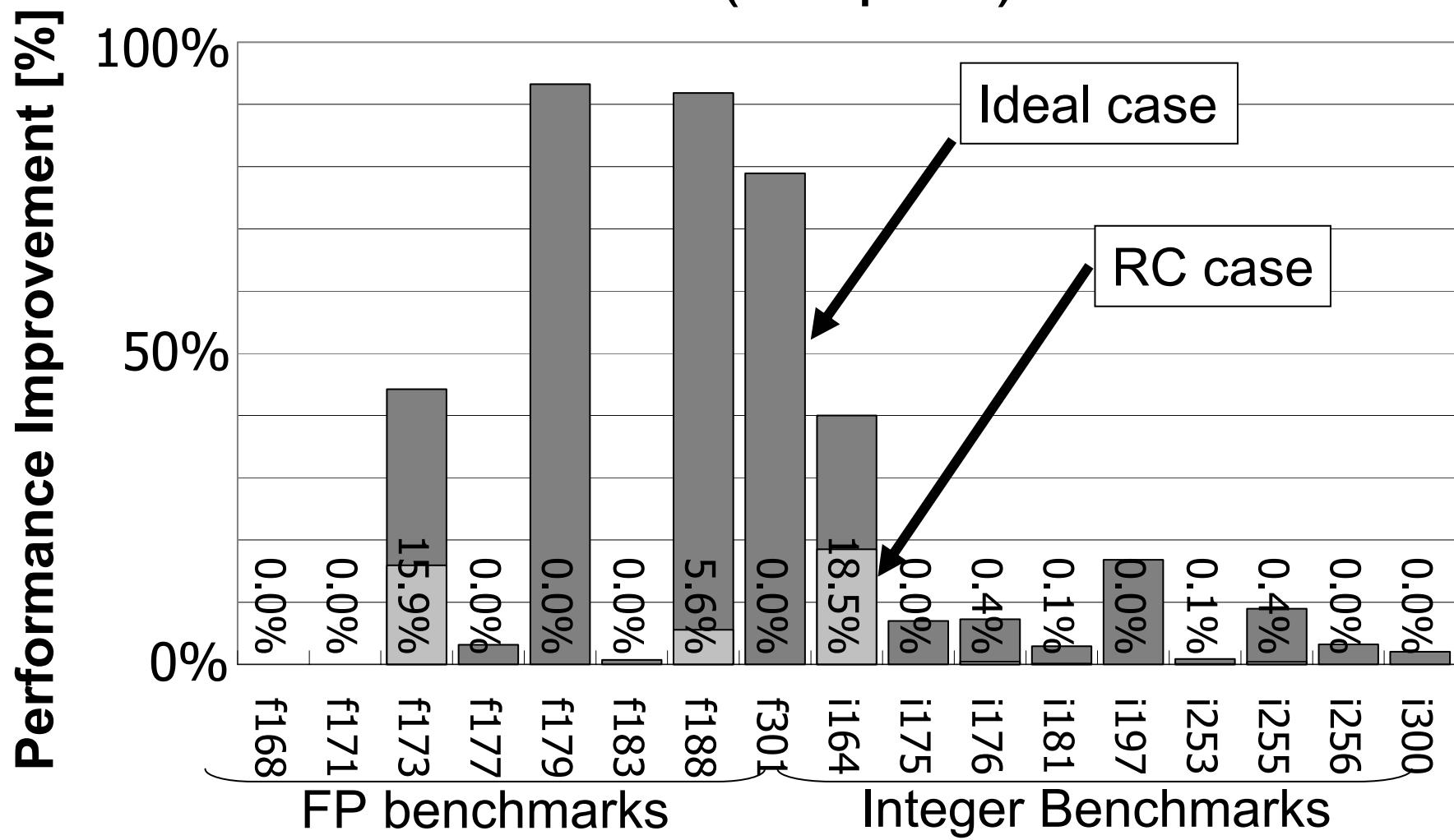
# Performance Issues (3/5): Speedup Attained by Replacing Critical LOADs with RC Codes (RC Case)



# Performance Issues (4/5): Potential Effects of Critical LOADs' Elimination (Ideal Case)



# Performance Issues (5/5): Comparison between Ideal Case (Graph 4) and RC Case (Graph 3)



# Related Work

- Prefetching
  - Intel's SP (Speculative Prefetching)
  - Roth's DDMT, and so on
- Value prediction
- Memoization (or Value reuse)

# On-demand Recomputation

- Outline
  - ✓ Computing Centric Computation (CCC)
  - ✓ Memory Wall Problem
  - ✓ Performance-Critical Instructions
  - ✓ On-demand Recomputation
  - ✓ Performance Issues
    - Summary and Conclusions

# Tutorial Outline

- Part 1 (8:00-9:30)
  - ✓ Overview of JIT HW/ISA/SW Co-optimization
  - ✓ Functionality Morphing
- Part 2 (15:00-16:30)
  - ✓ On-demand Recomputation

# Thank you

# References

- [1] W.A. Wulf and S.A. McKee, *Hitting the Memory Wall: Implications of the Obvious*, Computer Architecture News, 23(1):20-24, March 1995.
- [2] International Technology Roadmap for Semiconductors. <http://public.itrs.net/>
- [3] Simplescalar Toolset.  
<http://www.simplescalar.com>
- [4] Transmeta Inc. The crusoe processor.  
<http://www.tensilica.com>, 2000.
- [5] A. Roth and G.S. Sohi, Speculative Data-Driven Multithreading, HPCA-7, January 2001.