ELSE：Energy-efficient Latency Sensitive Execution

Dong Tong

Microprocessor R&D Center

Peking University

Superscalar Techniques

- **Instruction flow**: Branch instruction processing
  - Branch Prediction
- **Register data flow**: ALU instruction processing
  - Dynamically out-of-order execution
  - Register renaming
- **Memory data flow**: Load/store instruction processing
  - Load bypassing and load forwarding
  - Load address prediction
  - Load value prediction
  - Memory dependence prediction
  - Load value renaming

Instruction-level parallelism (ILP)

ILP Machine Parameters

- Operation latency (OL)
  - The number of machine cycles until the result of an instruction is available for use by a subsequent instruction.
- Machine parallelism (MP)
  - The maximum number of simultaneously executing instructions the machine can support. (In-flight instructions)
- Issue latency (IL)
  - The number of machine cycles required between issuing two consecutive instructions.
- Issue parallelism (IP)
  - The maximum number of instructions that can be issued in every machine cycle.

N. Joppi and D. wall, 1989**, Available instruction-level parallelism for superscalar and superpipelined machines**, ASPLOS-3

Load/Store Instructions

- Register-register/load-store ISAs
  - X86 uops, ARM, PowerPC, SPARC, MIPS, etc…
- General-purpose register (GPR) computers
  - Registers are faster
  - Registers are more efficient for a compiler to use
  - Registers can be used to hold variables
- Load/store instructions
  - Moving data between register file and main memory
  - *Spill code* generated by compiler
  - Complex data structure: arrays and linked list etc.

Shen and Lipasti, 2005**, Modern Processor Design: Fundamentals of Superscalar processors, pp. 264.**

Load/Store in SPEC2006 (CINT 2006)

Name – Language	Inst. Count (Billion)	Branches	Loads	Stores
CINT 2006				
400.perlbench –C	2,378	20.96%	27.99%	16.45%
401.bzip2 – C	2,472	15.97%	36.93%	12.98%
403.gcc – C	1,064	21.96%	26.52%	16.01%
429.mcf –C	327	21.17%	37.99%	10.55%
445.gobmk –C	1,603	19.51%	29.72%	15.25%
456.hmmer –C	3,363	7.08%	47.36%	17.68%
458.sjeng –C	2,383	21.38%	27.60%	14.61%
462.libquantum-C	3,555	14.80%	33.57%	10.72%
464.h264ref- C	3,731	7.24%	41.76%	13.14%
471.omnetpp- C++	687	20.33%	34.71%	20.18%
473.astar- C++	1,200	15.57%	40.34%	13.75%
483.xalancbmk- C++	1,184	25.84%	33.96%	10.31%

Loads: 34.9% Stores: 14.3% Total: 49.2%

A. Phansalkar, A. Joshi and Lizy K. John, 2007, **Analysis of Redundancy and application Balance in the SPEC CPU 2006 Benchmark Suite**, ISCA-34**.**

Load/Store in SPEC2006 (CFP 2006)

Name – Language	Inst. Count (Billion)	Branches	Loads	Stores
CFP 2006				
410.bwaves – Fortran	1,178	0.68%	56.14%	8.08%
416.gamess – Fortran	5,189	7.45%	45.87%	12.98%
433.milc – C	937	1.51%	40.15%	11.79%
434.zeusmp–C,Fortran	1,566	4.05%	36.22%	11.98%
435.gromacs-C, Fortran	1,958	3.14%	37.35%	17.31%
436.cactusADM-C, Fortran	1,376	0.22%	52.62%	13.49%
437.leslie3d – Fortran	1,213	3.06%	52.30%	9.83%
444.namd – C++	2,483	4.28%	35.43%	8.83%
447.dealII – C++	2,323	15.99%	42.57%	13.41%
450.soplex – C++	703	16.07%	39.05%	7.74%
453.povray – C++	940	13.23%	35.44%	16.11%
454.calculix –C, Fortran	3,041	4.11%	40.14%	9.95%
459.GemsFDTD – Fortran	1,420	2.40%	54.16%	9.67%
465.tonto – Fortran	2,932	4.79%	44.76%	12.84%
470.lbm – C	1,500	0.79%	38.16%	11.53%
481.wrf - C, Fortran	1,684	5.19%	49.70%	9.42%
482.sphinx3				5.58%

Loads: 49.0% Stores: 12.7% Total: 61.7%

A. Phansalkar, A. Joshi and Lizy K. John, 2007**, Analysis of Redundancy and application Balance in the SPEC CPU 2006 Benchmark Suite**, ISCA-34**.**

Load/Store in PARSEC (8 cores)

Program	Problem Size	Instructions (Billions)				Synchronization Primitives		
		Total	FLOPS	Reads	Writes	Locks	Barriers	Conditions
`blackscholes`	65,536 options	2.67	1.14	0.68	0.19	0	8	0
`bodytrack`	4 frames, 4,000 particles	14.03	4.22	3.63	0.95	114,621	619	2,042
`canneal`	400,000 elements	7.33	0.48	1.94	0.89	34	0	0
`dedup`	184 MB data	37.1	0	11.71	3.13	158,979	0	1,619
`facesim`	1 frame, 372,126 tetrahedra	29.90	9.10	10.05	4.29	14,541	0	3,137
`ferret`	256 queries, 34,973 images	23.97	4.51	7.49	1.18	345,778	0	1255
`fluidanimate`	5 frames, 300,000 particles	14.06	2.49	4.80	1.15	17,771,909	0	0
`freqmine`	990,000 transactions	33.45	0.00	11.31	5.24	990,025	0	0
`streamcluster`	16,384 points per block, 1 block	22.12	11.6	9.42	0.06	191	129,600	127
`swaptions`	64 swaptions, 20,000 simulations	14.11	2.62	5.08	1.16	23	0	0
`vips`	1 image, 2662×5500 pixels	31.21	4.79	6.71	1.63	33,586	0	6,361
`x264`	128 frames, 640×360 pixels	32.43	8.76	9.01	3.11	16,767	0	1,056

Table 1: Breakdown of instructions and synchronization primitives for input set `simlarge` on a system with 8 cores. All numbers are totals across all threads. Numbers for synchronization primitives also include primitives in system libraries. "Locks" and "Barriers" are all lock- and barrier-based synchronizations, "Conditions" are all waits on condition variables.

Loads: 31.1% Stores: 8.8% Total: 39.9%

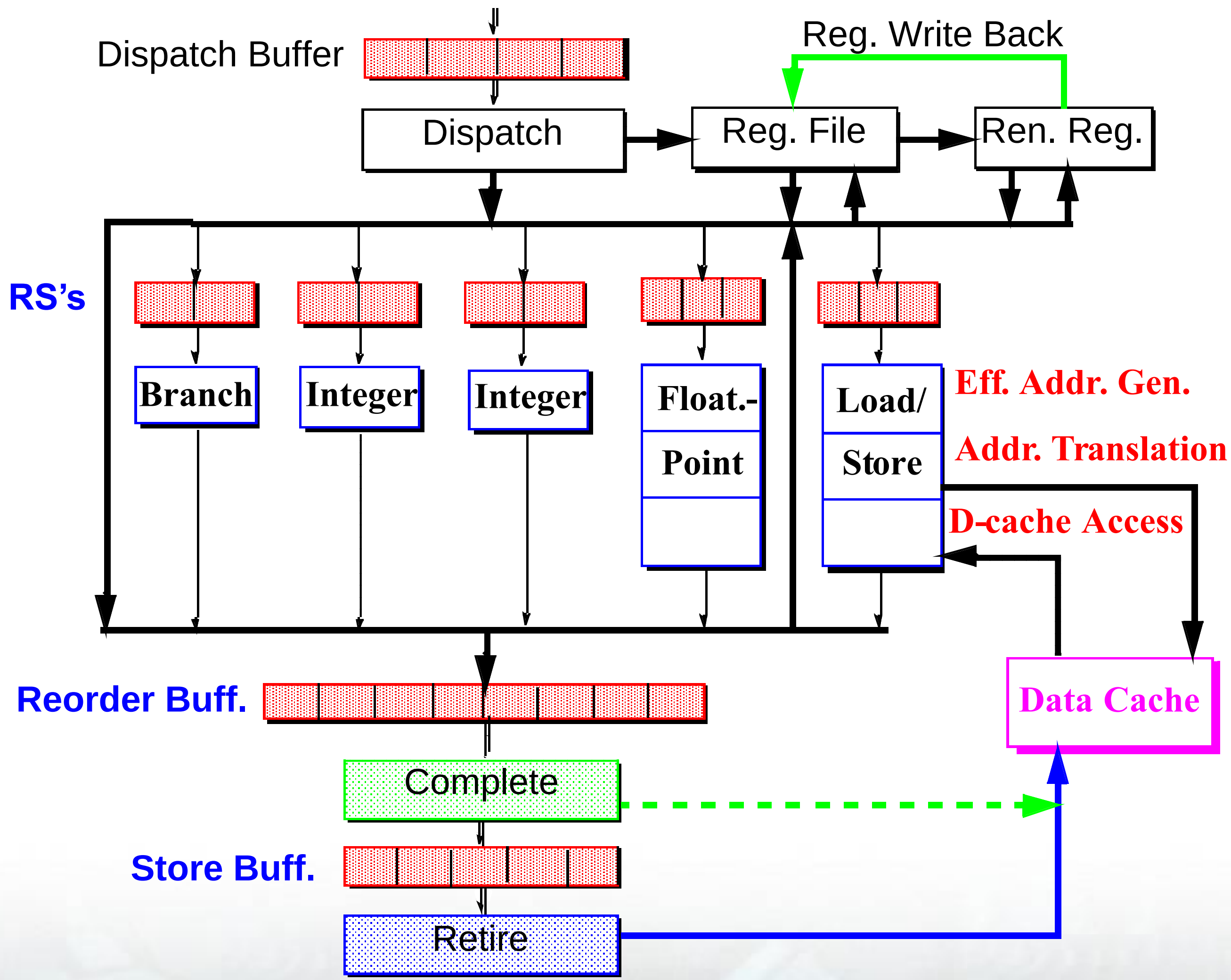
C. Bienia and Kai Li, 2008**, The PARSEC Benchmark Suite: Characterization and Architectural Implications**, PACT 2008**.**

北京大學

Latency-Sensitive Technologies

- Long memory latency is the biggest challenges
  - Memory Wall ->
  - Latency-Tolerant Technologies ->
  - Dynamically Speculative Execution ->
  - Power Wall ->
  - ***Latency-sensitive technologies***
    - ***Latency-Reducing technologies***
- Can we use Renaming and OoO Execution?
  - Stores must execute In-order
    - To preserve sequential state in cache and main memory
  - Loads can be issued out-of-order without dependence violation.

Load/Store Instruction Processing



Shen and Lipasti, 2005, **Modern Processor Design: Fundamentals of Superscalar processors, pp. 264.**

Load/Store Instruction Execution

- **Three Steps**
  - – Memory Address Generation
  - – Memory Address Translation
  - – Data Memory Accessing
- **Load Latency: At least 3 cycles**
  - – Definition: The number of CPU cycles until the result of a load instruction is available for use by a subsequent instruction after issued to execution unit.
  - – Memory address generation latency: 1 cycle
  - – Memory address translation latency: ⩾ 1 cycle
  - – Data Memory Accessing latency: ≫ 1 cycle

Shen and Lipasti, 2005, **Modern Processor Design: Fundamentals of Superscalar processors.**

Memory Hierarchy Structure

- Private L1 Cache
  - Access Time: 2~5 cycles
  - Virtual-Indexed-Physical-Tagged
    - Memory address translation and Tag accessing in parallel
    - With OS page size constrain (4KB or 8KB)
    - Typical 32KB/64KB 8-way associate cache with 64B line size
  - Physical-Indexed-Physical-Tagged
    - Memory address Translation and Tag accessing sequentially
    - One more cycle in load lantency
    - Without OS page size constraint
- Shared Last-Level Cache: ~10s cycles
- Main Memory: ~100s cycles

Memory Data Dependences

- "Memory Aliasing" = Two memory references involving the same memory location (collision of two memory addresses).
- "Memory Disambiguation" = Determining whether two memory references will alias or not (whether there is a dependence or not).
- Memory Dependency Detection:
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

Example code:

(1)	STORE	V	
(2)	ADD		
(3)	LOAD	W	RAW
(4)	LOAD	X	
(5)	LOAD WAR	V	
(6)	ADD		
(7)	STORE	W	

Total Order of Loads and Stores

- Keep all loads and stores totally in order with respect to each other but not necessary.
- Loads and stores can execute out of order with respect to other types of instructions.
- Memory models Limitations:
  - To facilitate recovery from exceptions, the sequential state of memory must be preserved.
  - Many shared-memory multiprocessor systems assume the sequential consistency (SC) processor be done according to program order.
  - Store instructions required to be executed in program order, WAW and WAR are implicitly enforced.

Load Bypassing

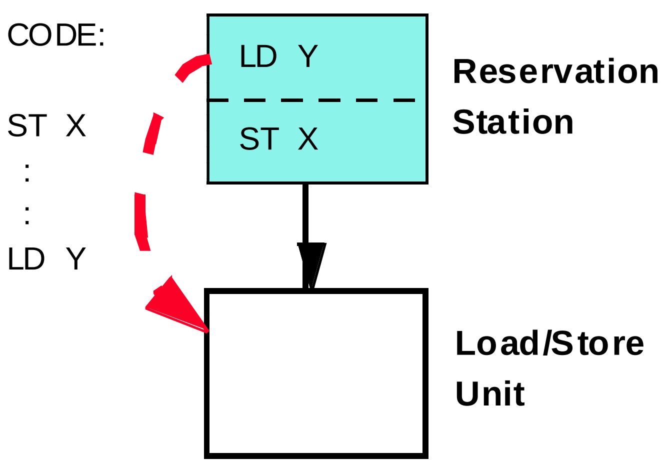
- Loads can be allowed to bypass stores (if no aliasing).
- Two separate reservation stations and address generation units are employed for loads and stores.
- Store addresses still need to be computed before loads can be issued to allow checking for load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).
- Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.
  - Store buffer is “future file” for memory

Load Forwarding

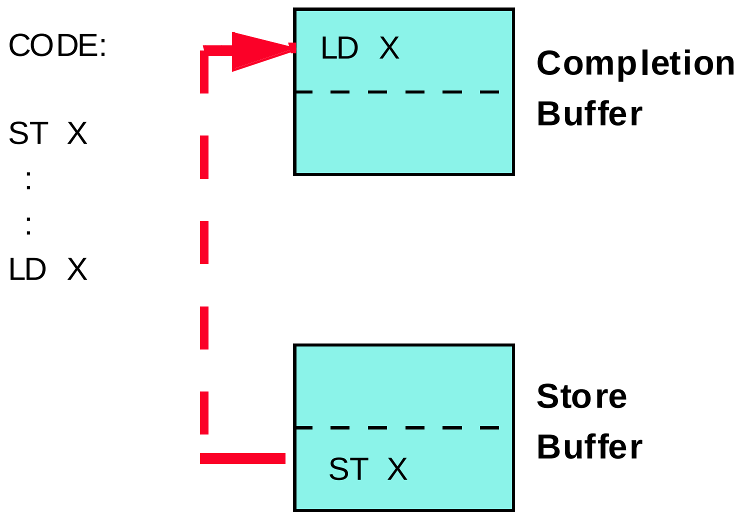
- If a subsequent load has a dependence on a store still in the store buffer, it need not wait till the store is issued to the data cache.
- The load can be directly satisfied from the store buffer if the address is valid and the data is available in the store buffer.
- Since data is sourced from the store buffer:
  - Could avoid accessing the cache to reduce power/latency

Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:



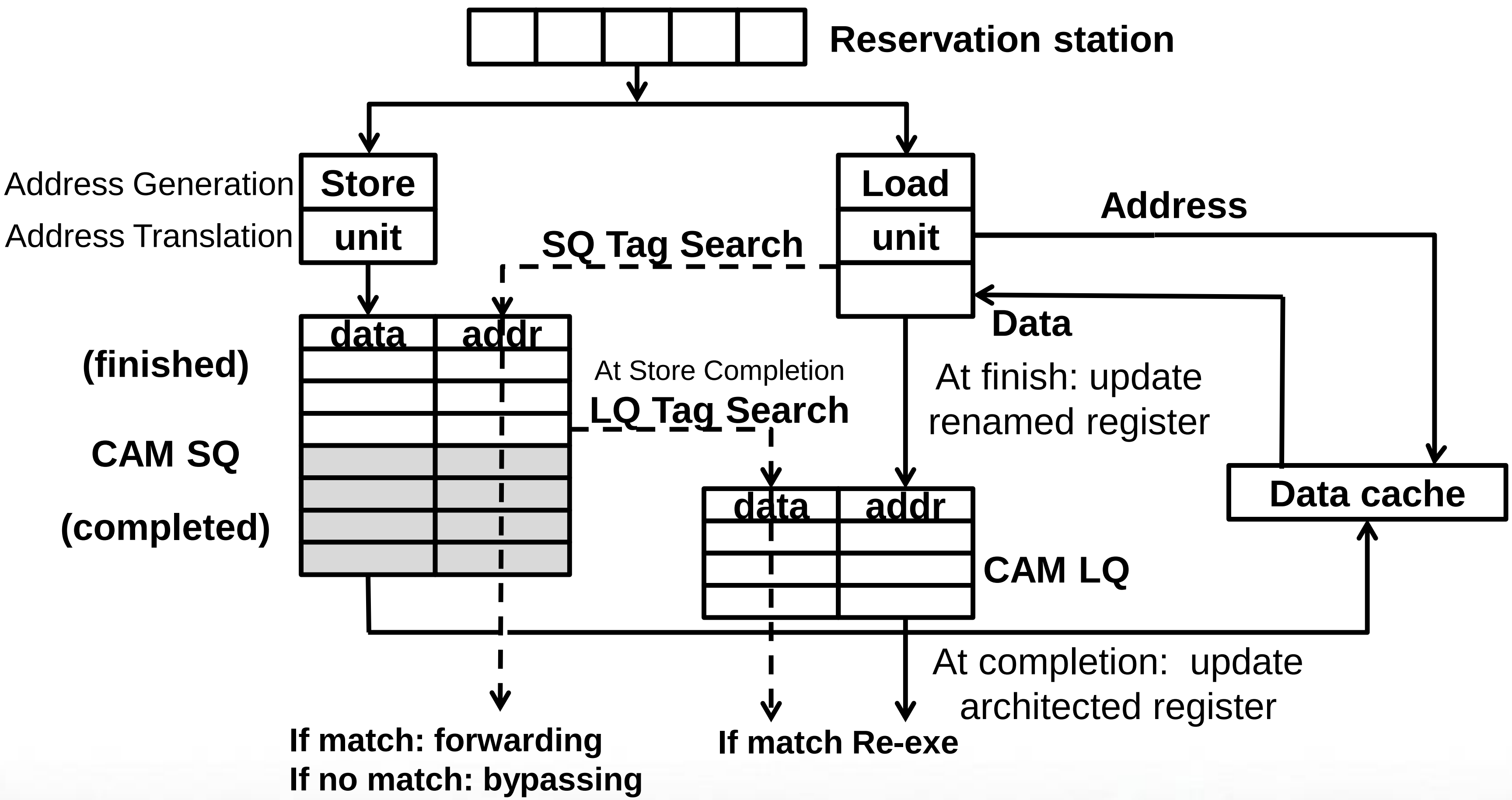
Performance gain:

Load bypassing: 11%-19% increase over total ordering

Load forwarding: 1%-4% increase over load bypassing

M. Johnson, **Superscalar Microprocessor Design**. Englewood Cliffs, NJ:Prentice Hall, 1991.

Fully OoO Issuing and Execution



Shen and Lipasti, 2005, **Modern Processor Design: Fundamentals of Superscalar processors, pp. 272.**

Speculative Disambiguation

- Three important searches on the load/store queue which is implemented as two separate queues.
  - **In-order issue: CAM-based Store Queue**
    - **First,** when a load executes, it searches the store queue. If there is a match, then the load obtains its value from the store queue **(load forwarding)**. If no aliasing is detected, the load is allow bypassing (**load bypassing**).
  - **Out-of-order issue: CAM-based Load Queue**
    - **Second,** when a store has a valid address, it searches the load queue. If the address matches with a younger, speculatively-serviced load, this premature load and all sub-sequent instructions are squashed and fetched again (**store-load order violation**).
    - **Third,** in some processor, when a load executes, it searches the load queue. If the address matches with a younger out-of-order-issued load, this load and all subsequent instructions are squashed and fetched again (**load-load order violation**).

Memory Data Flow Techniques

- Multiple load/store unit supported by multiported data cache.
- Memory Hierarchy Techniques
  - Multiple levels caches
  - Nonblocking cache with missed load queue
  - Replacement, bypassing, partition and prefetching of Last-level Cache
  - Main Memory Management: Memory-level Parallelism and Row-buffer locality
- **Early or fast load execution**
  - **Load address prediction**
  - **Load value prediction**
  - **Memory dependence prediction**
  - **Load value renaming**

Early and fast load execution

- **Memory disambiguation**
  - resolves store–load dependences and enables earlier execution of store-independent loads.
- **Memory renaming and memory bypassing**
  - short-circuit memory to stream-line the passing of values from stores to loads.
- **Critical path scheduling, pre-execution, and address prediction**
  - advance long-latency loads by computing load addresses early, or predicting them.
- **Value prediction**
  - short-circuits load execution by predicting the loaded data values.

A. Roth, et. al. 2001**. Dynamic techniques for load and load-use scheduling.** Proceedings of the IEEE, Nov, 2001.

Aggressive load speculation

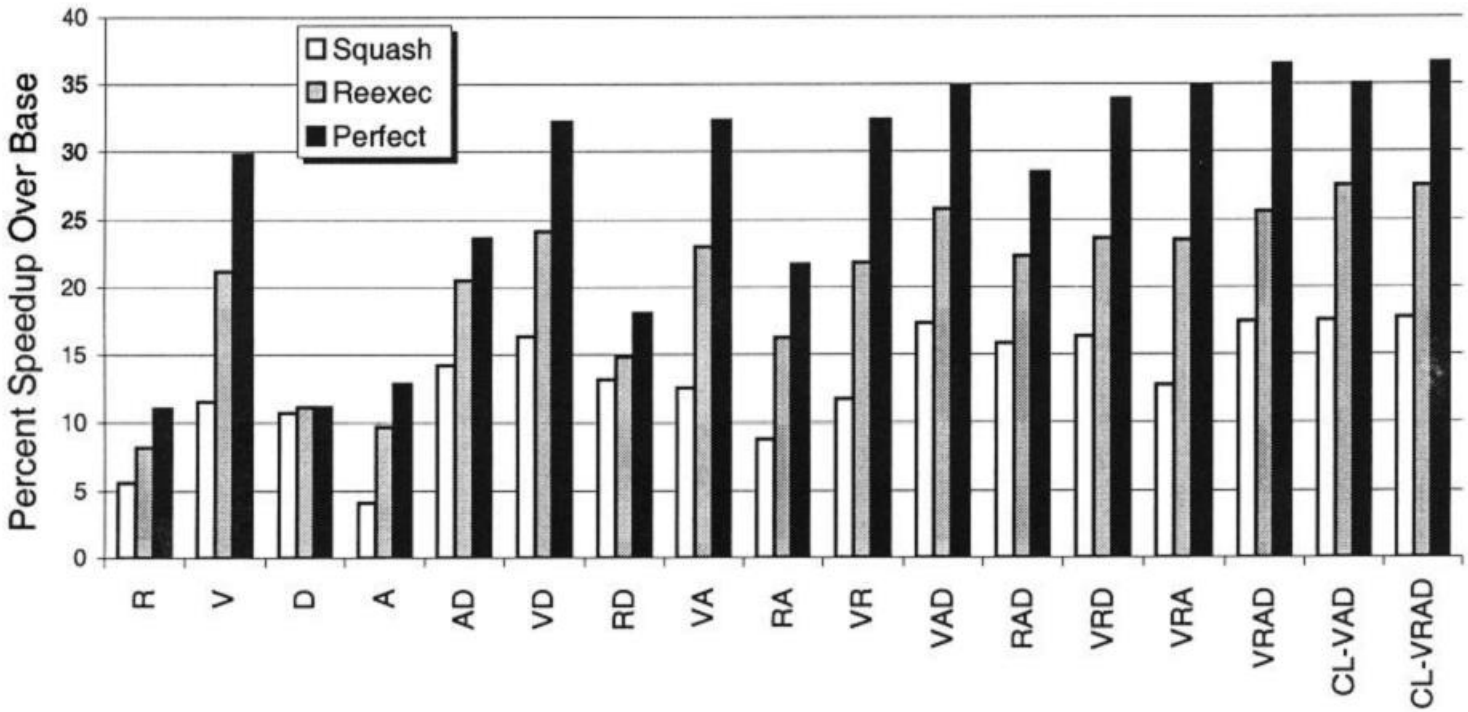


Figure 7: Average speedup results for Squash and Reexecution recovery for all combinations of the predictors using the Load-Spec-Chooser to decide which predictor to use. D = Store Set Dependence Prediction, V = Hybrid Value Prediction, A = Hybrid Address Prediction, R = Original Memory Renaming, and CL = Check-Load Prediction

- **Value prediction**: the largest performance improvement
- All techniques have **large hardware overhead and complexity.**

G. Reinman and B. Calder. 1998**. Predictive techniques for aggressive load speculation. MICRO-31, 1998.**

Latency-Tolerant Execution

- Long-latency LLC misses are the primary sources of low performance.
- Latency-tolerant defers LLC misses and their dependent instructions then
  - Removing them form the window and saving them in some **slice buffer**
  - Allowing younger instructions to enter the windows and executes
  - Re-execution the load and its dependent instruction from slice buffer when the miss returns.
- Latency-tolerant increases ILP and single-thread MLP (if launching parallel LLC misses)
  - Higher performance and lower execution overhead than RA (Runahead)

A. Hilton and A. Roth. 2010. BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Execution. HPCA-16, 2010.

Latency-Tolerant Techniques

- Large Instruction Window
  - WIB: Wait Instruction Buffer. (ISCA’02)
  - KILO: Low complexity “slow lane” issue queue. (TACO’04/IEEE MICRO’05)
  - CPR/CFP: True Latency-Tolerant Techniques
    - Checkpoint Processing and Recovery (MICRO’03)
    - Control Flow Processing (ASPLOS’04)
  - D-KIP: (HPCA’06)
    - Miss-independence instructions: OoO Core
    - Miss-dependence instructions: In-order Core.
  - BOLT: SMT resources as the slice buffer(HPCA’10)
- Runahead (HPCA’03) and Multithreading

Common Size of LSQ

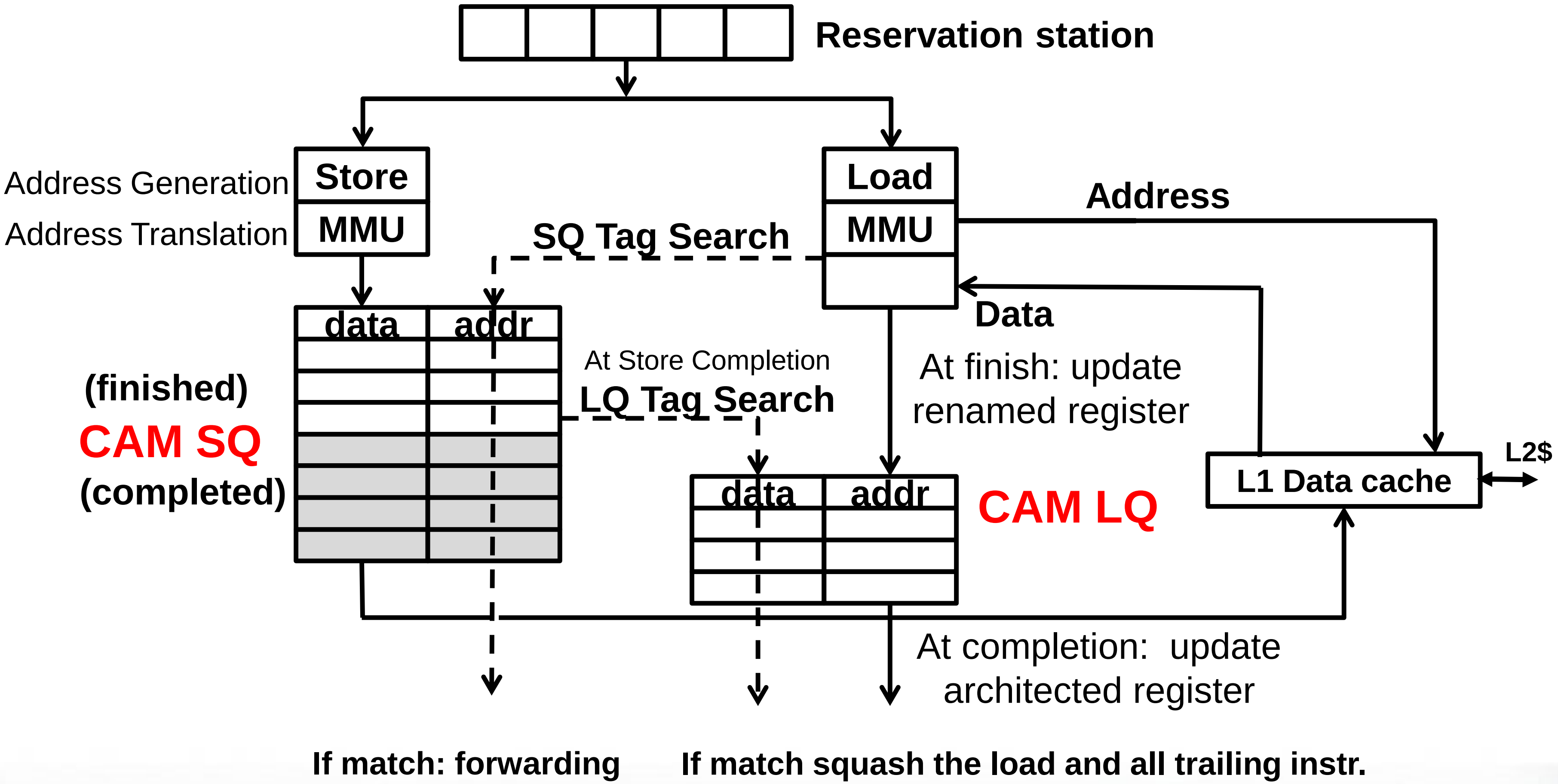
Intel	Haswell	Sandy Bridge	Nehalem	Core
Instruction Q	2x20	2x20	18	18
ROB	192	168	128	96
Register File	168	160		
Load Buffer	72	64	48	32
Store Buffer	42	36	32	20
Scheduler	60	54	36	32

The Scalability of Load/Store Queue

- Hierarchy Load/Store Queue
- Bloom Filter (MICRO’03)
  - LSQ search filter/LSQ state filter/Address predictor
  - Only for small instruction window processor
- Load Re-execution
  - **Memory Ordering**: Eliminating associate load queue, re-executing loads in-order prior to commit, detecting violation. (ISCA’04)
  - **Store Vulnerability Windows (SVW)** (ISCA’05)
    - Address-based filter of load re-execution.
    - Sequence Number (SSNs)/Store Sequence Bloom Filter (SSBF)
  - **Store Queue Index Prediction (SQIP)** (MICRO’05)
    - Forwarding Speculation: age-order and non-associative store queue
    - Forwarding Prediction: Fire-and forget (MICRO’06)/NoSQ (MICRO’06)
  - **DSC/SDR for CPR/CFP** (ISCA’09)

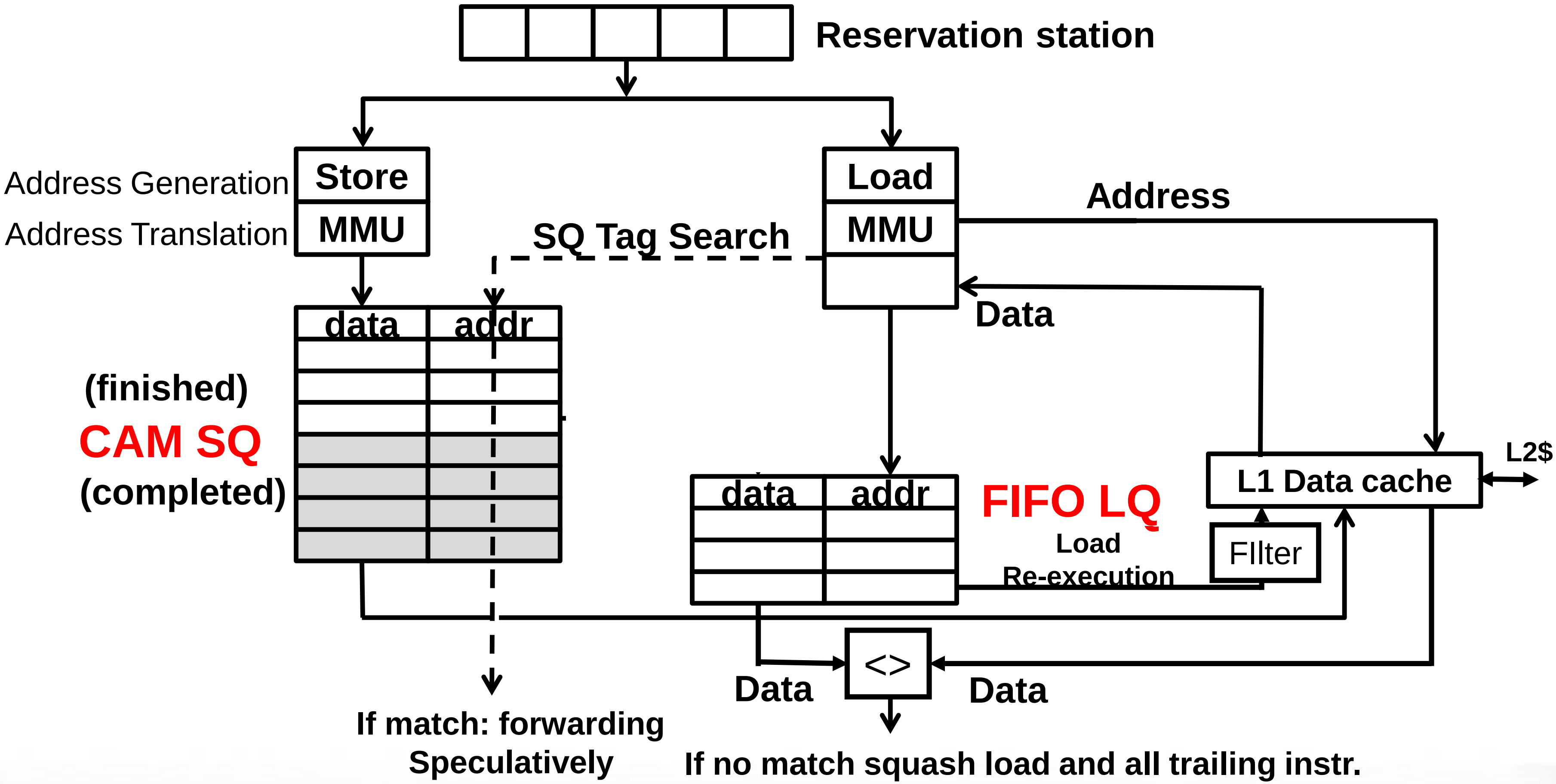
Store-load communication only between in-flight instructions

Aggressive Load/Store Execution



The Scalability of Load/Store Queue

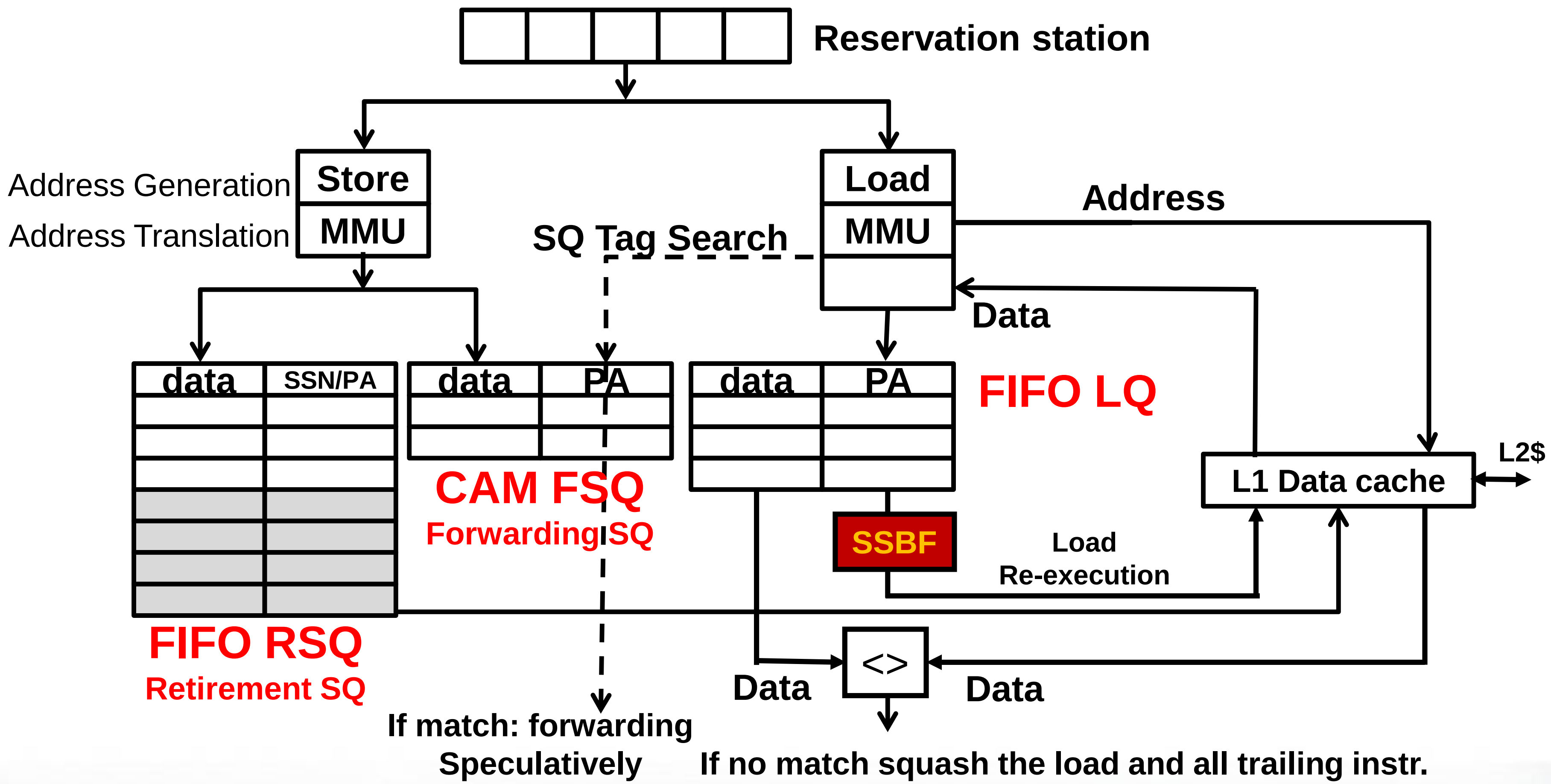
Load Re-Execution (ISCA 2004)



The Scalability of Load Queue

FIFO Load Queue+Re-execution+Some Filter

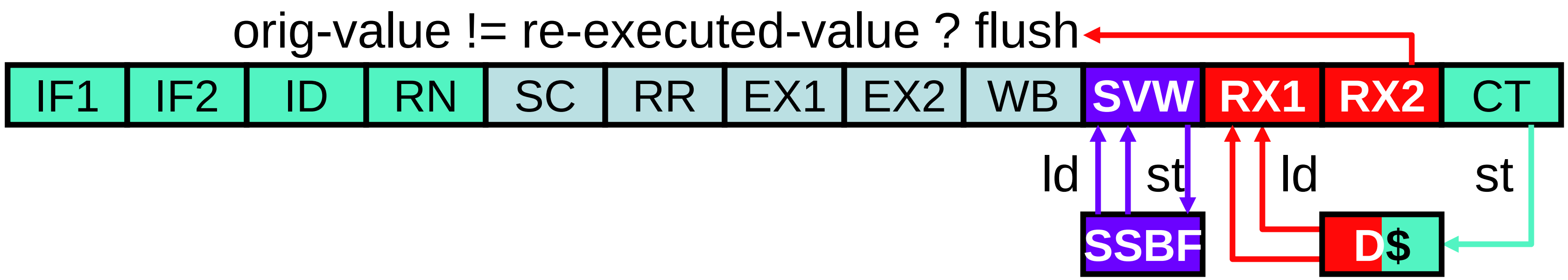
Store Vulnerability Window (ISCA 2006)



Only in-window store-load forwarding

Large FIFO RSQ+Small CAM FSQ+SSN/Bloom Filter

SVW: Store Vulnerability Window



- **SVW**: general-purpose load re-execution filter
  - Loads access new table to see if they can safely skip re-execution
- Performance
  - Additional pipeline stage? Additional table access?
  - SSBF is much smaller than D$ (e.g., 1KB vs. 64KB)
    - High bandwidth
    - Single-cycle access → no critical loop

  \+ Reduces re-executions by 5-20X

A. Roth. 2005. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, ISCA, 2005.

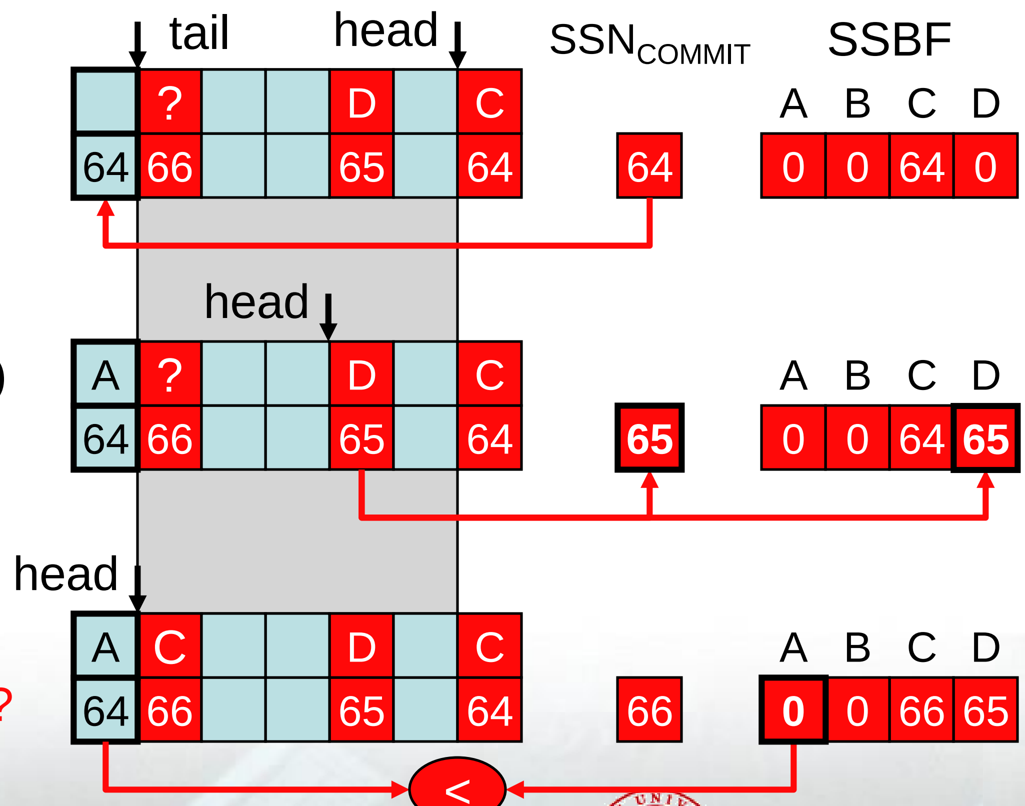
Load Speculation+SVW: Example

- Three events (Rename, Exec, SVW/Re-exec) in the life of a load
- Addresses: A,B,C,D, SSNs/SVWs: 0,1,…64,65,66…
- LSQ: load store

t=X
Rename load, establish SVW
load.SVW = SSN_{COMMIT} (64)

t=X+5
Execute load (ambiguous, but correct)
Also commit store 65

t=X+10
SVW load, check address collision
SSBF[load.addr] (0) > load.SVW (64)?
No, no need to re-execute



A. Roth. 2005. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, ISCA, 2005.

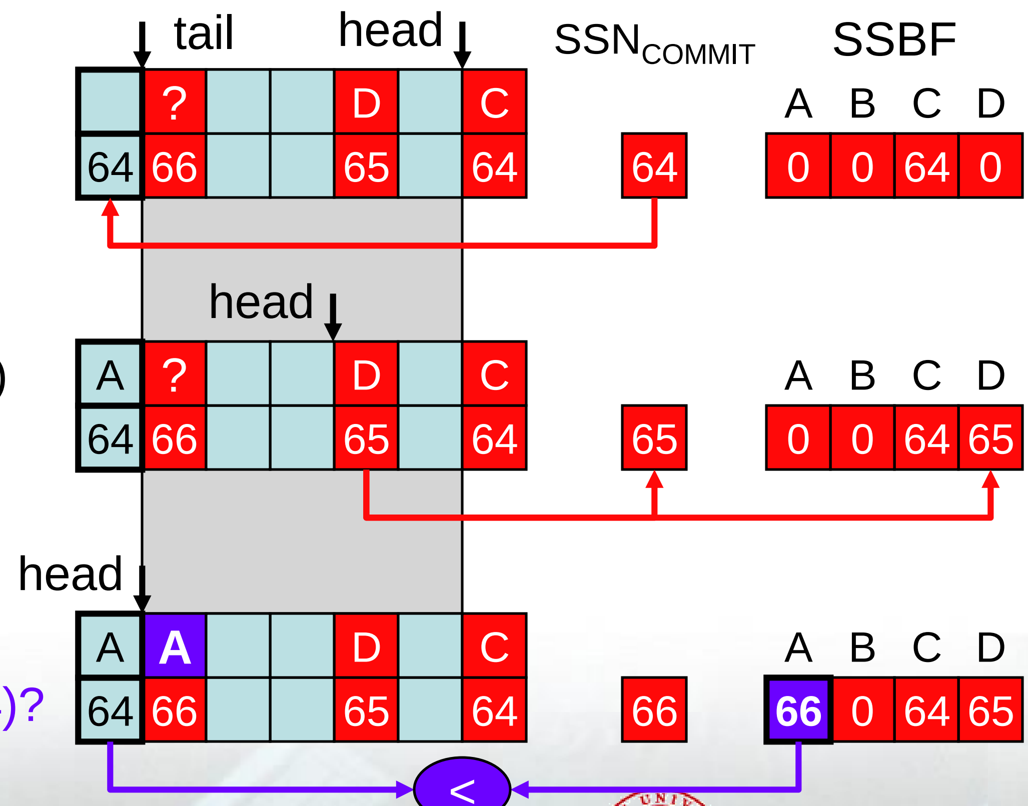
Load Speculation+SVW: Other Example

– Same setup

t=X
Rename load, establish SVW
load.SVW = SSN_{COMMIT} (64)

t=X+5
Execute load (ambiguous, and wrong)
Also commit store 65

t=X+10
SVW load, check address collision
SSBF[load.addr] (66) > load.SVW (64)?
Yes, must re-execute



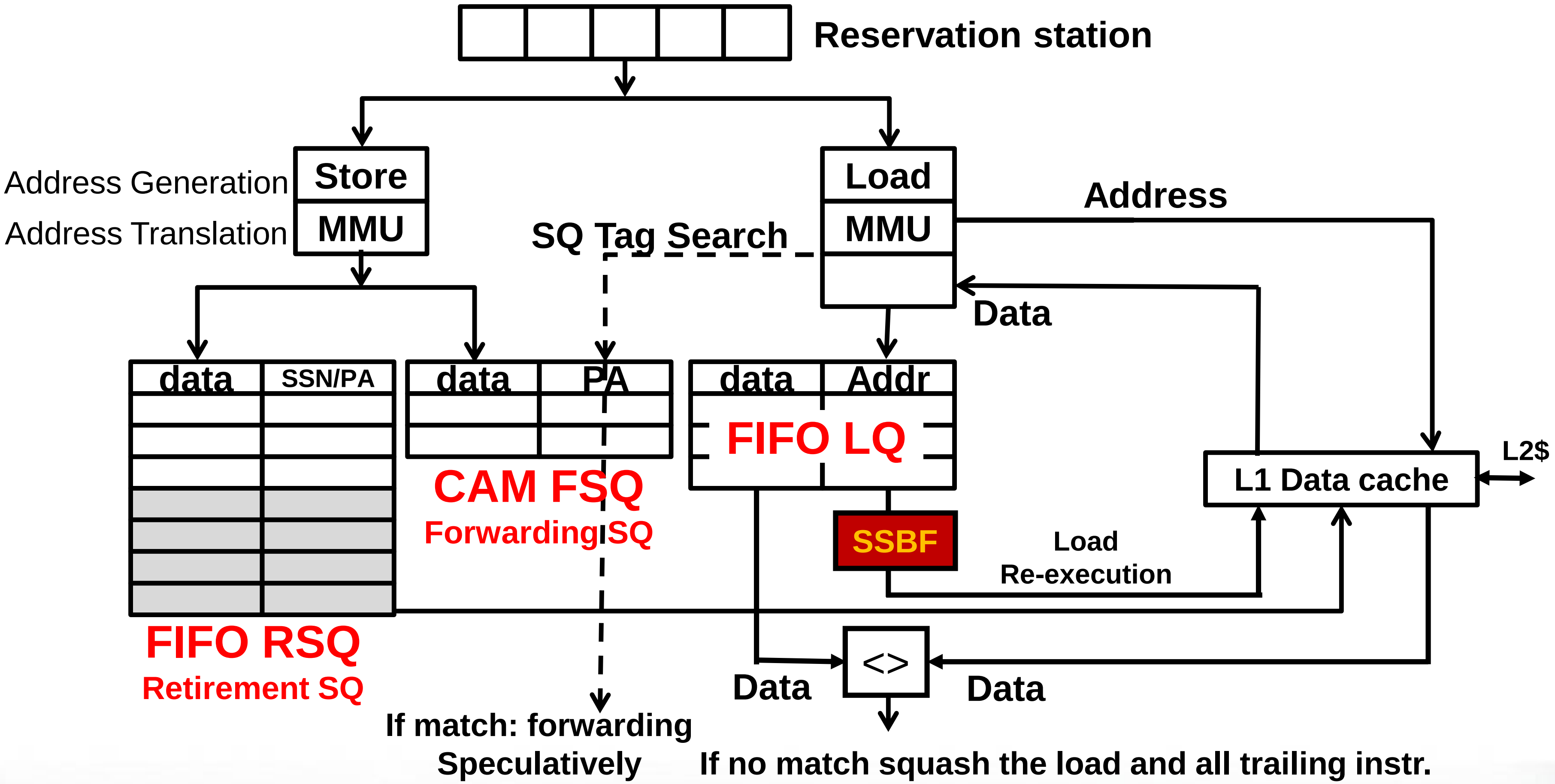
A. Roth. 2005. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, ISCA, 2005.

Optimizing Store-Load Forwarding

- Speculative Load Forwarding
  - With load re-execution mechanism to verify load speculative execution such as SVW.

- Do we must use associate search (CAM)?
- Do we must do forwarding only in dynamic instruction window?
- Do we must use physical address?
- Do we must use precise address?

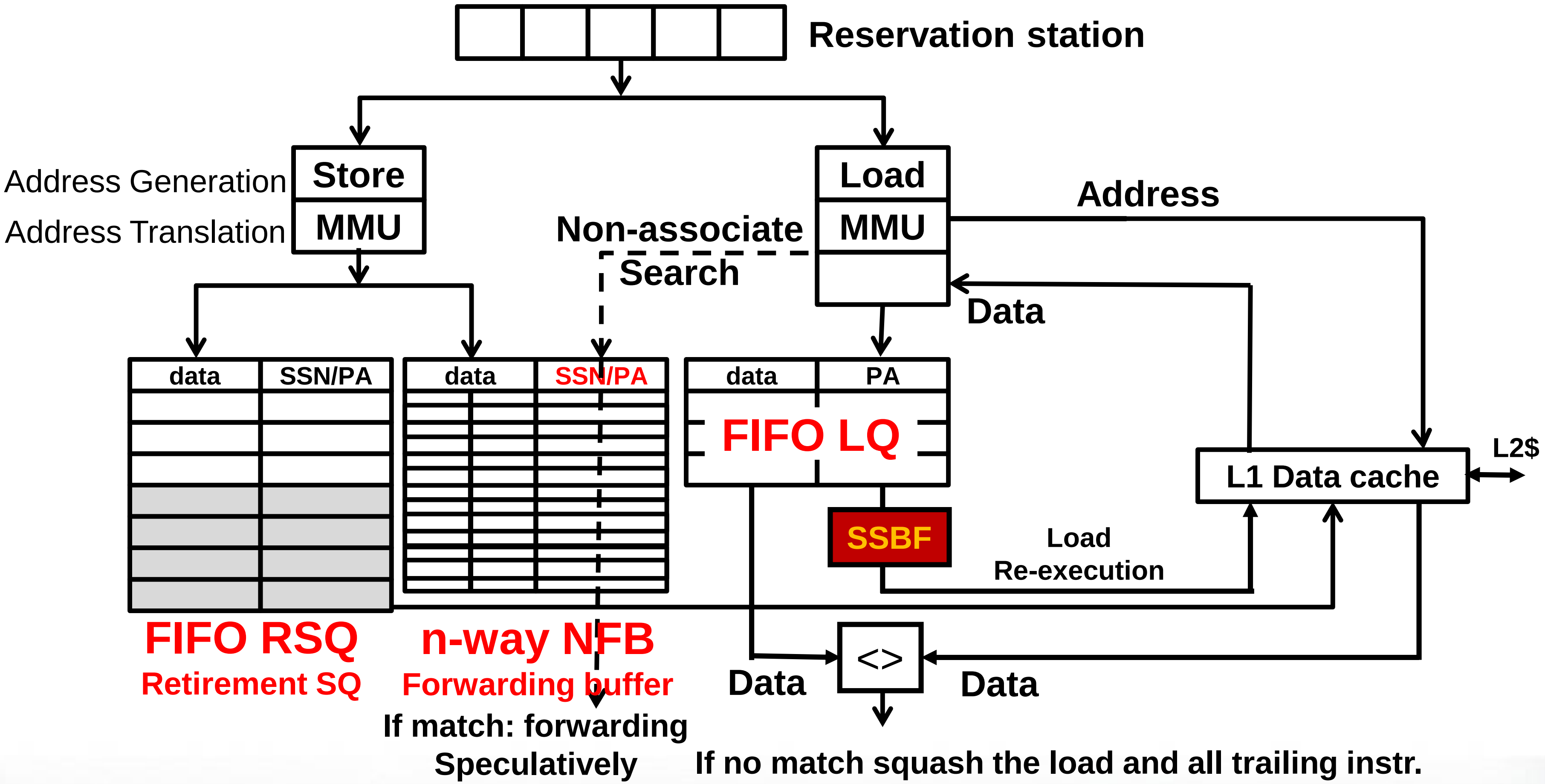
Baseline: SVW (ISCA 2006)



In-window store-load forwarding: 3 cycle Latency

Large FIFO RSQ+Small CAM FSQ+SSN/Bloom Filter

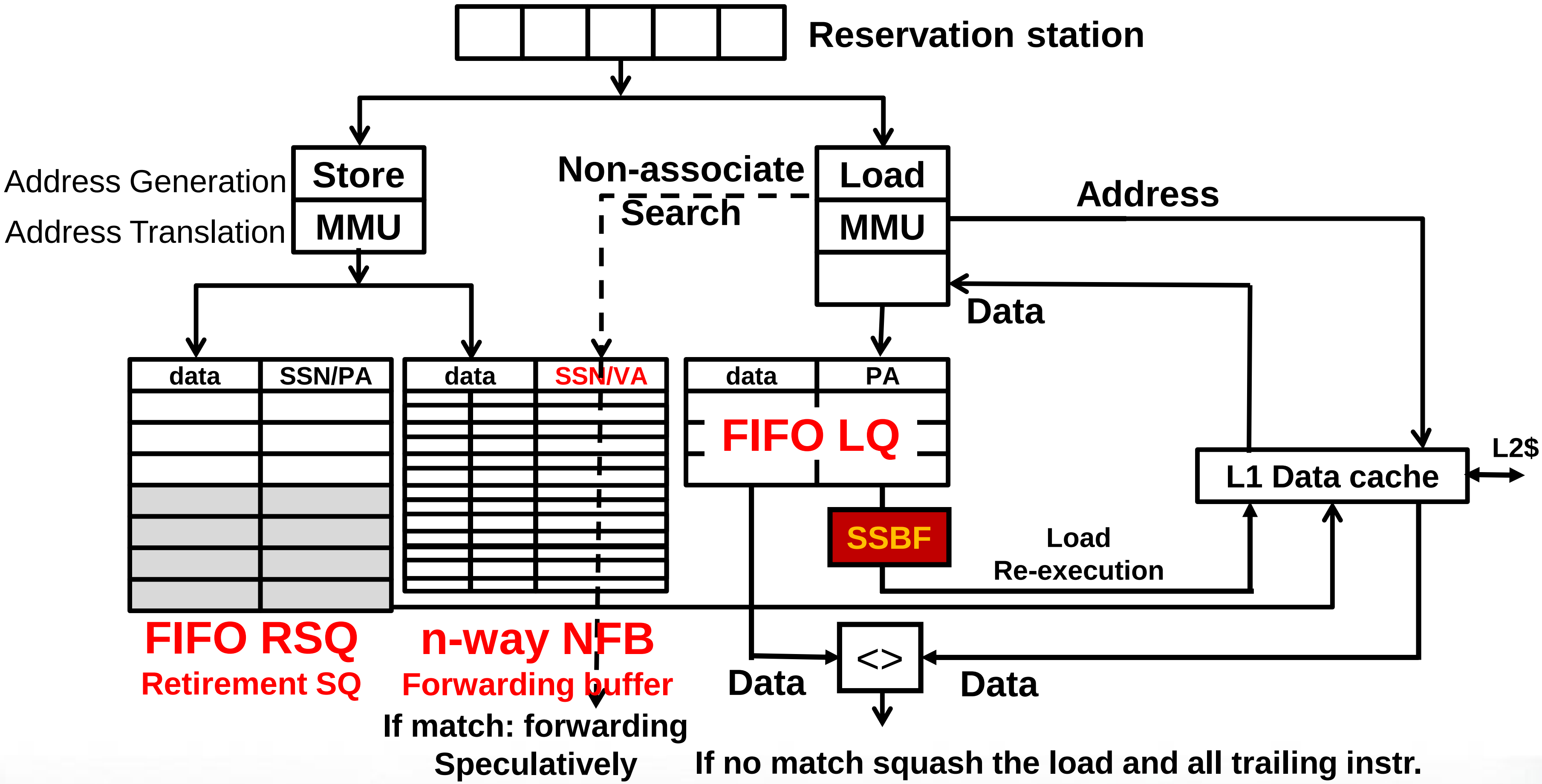
Optimizing 1: Out-of-window Forwarding



Out-window store-load forwarding: 3-cycle latency

FIFO RSQ+Large No-Associate FB +SSN/Bloom Filter

Optimizing 2: ASW (JCST 2012)



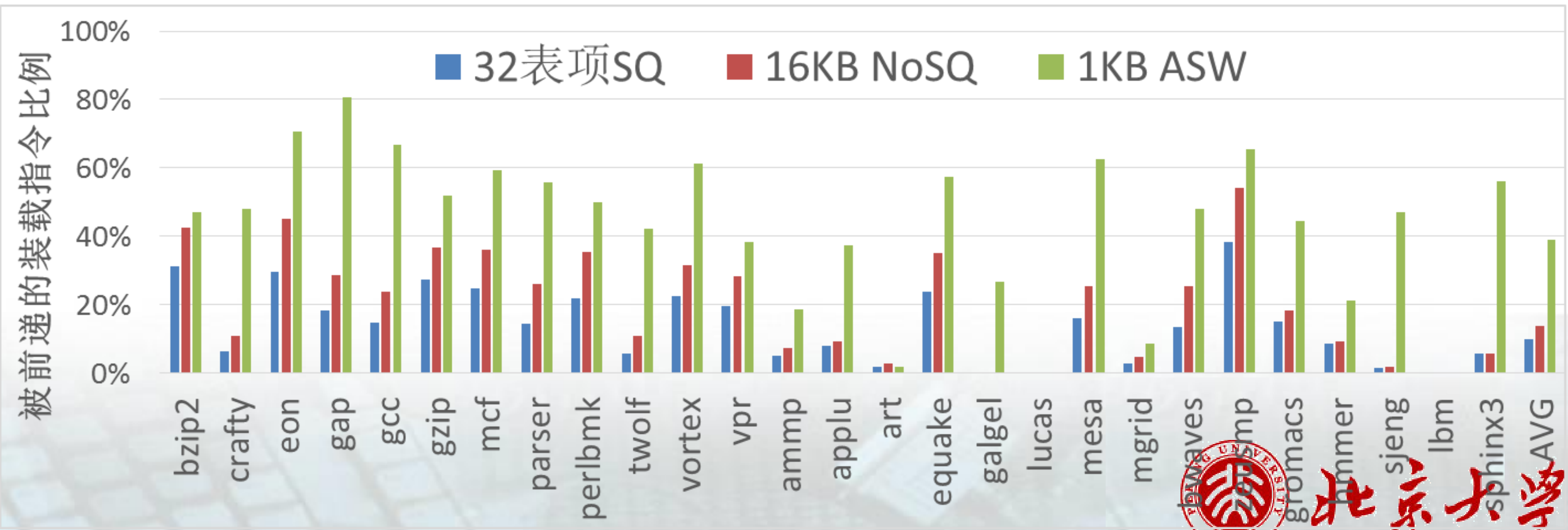
Virtual address store-load forwarding: 2-cycle latency

FIFO RSQ+Large No-Associate FB +SSN/Bloom Filter

ASW Forwarding Ratio Evaluation

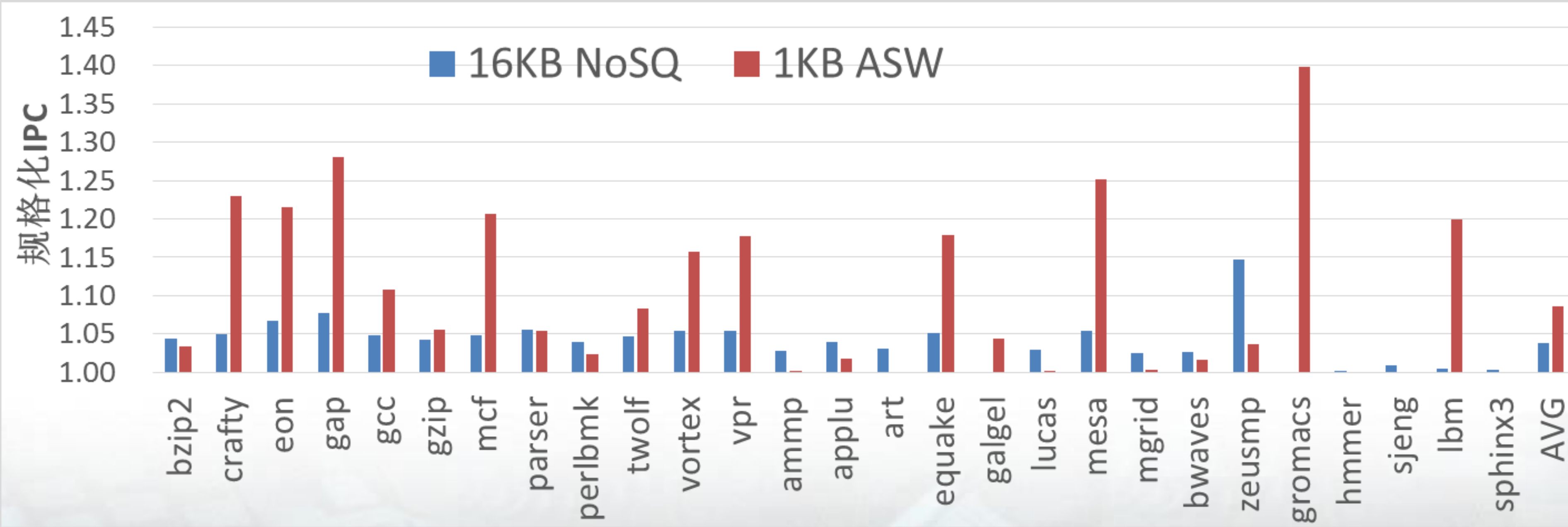
访存数据前递	Baseline	NoSQ	ASW
前递机制	256B 全相联	16KB 直接映射 (预测器)	1KB 组相联
前递延迟	2周期	0周期	2周期
前递比例	10.32%	14.06%	37.99%
前递正确率	98.02%	95.5%	93.35%

前递类型	前递比例
窗口内前递	13.84%
长前递	24.15%

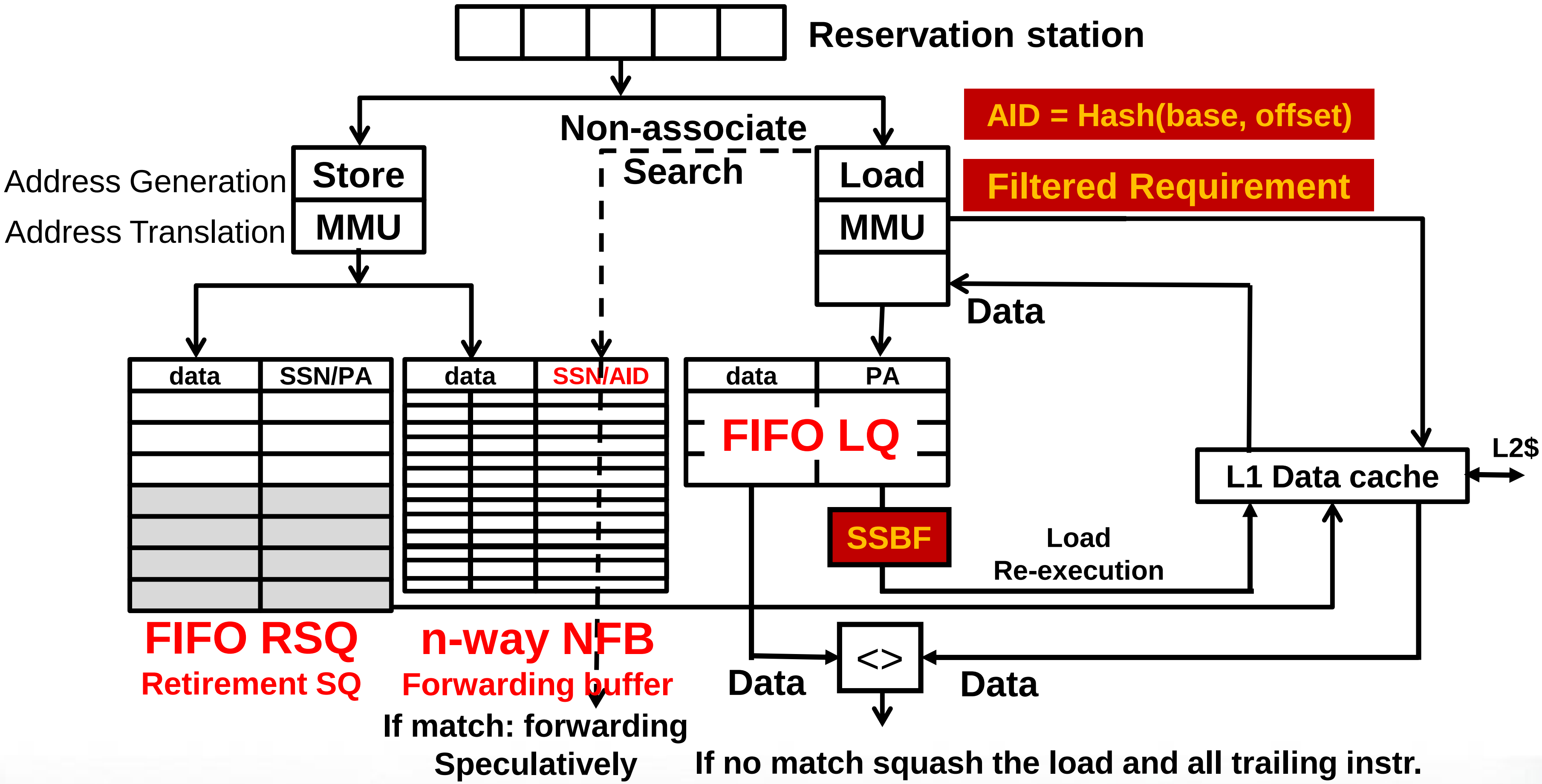


ASW Performance Evaluation

	Baseline	NoSQ	ASW
性能提升	-	3.69%	8.67%



Optimizing 3: SOLE (ICCD 2012)



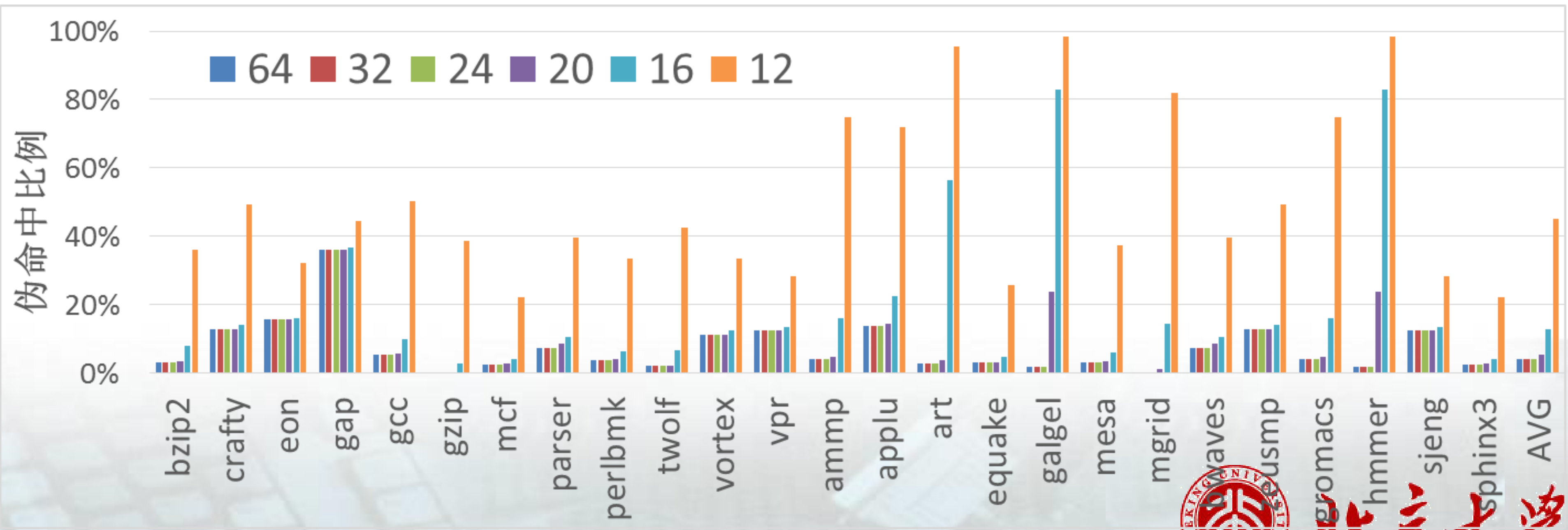
Hashed address store-load forwarding: 1-cycle latency

FIFO RSQ+Large No-Associate FB +SSN/Bloom Filter

SOLE Forwarding Ratio Evaluation

访存数据前递	Baseline	NoSQ	ASW	ASW+20位AID
前递延迟	2周期	0周期	2周期	1周期
前递比例	10.32%	14.06%	37.99%	40.06%
前递正确率	98.02%	95.5%	93.35%	88.53%

地址标识宽度	理想	64位	20位	12位
伪命中比例	0%	3.67%	4.77%	44.83%



SOLE Power Evaluation

访存执行能耗

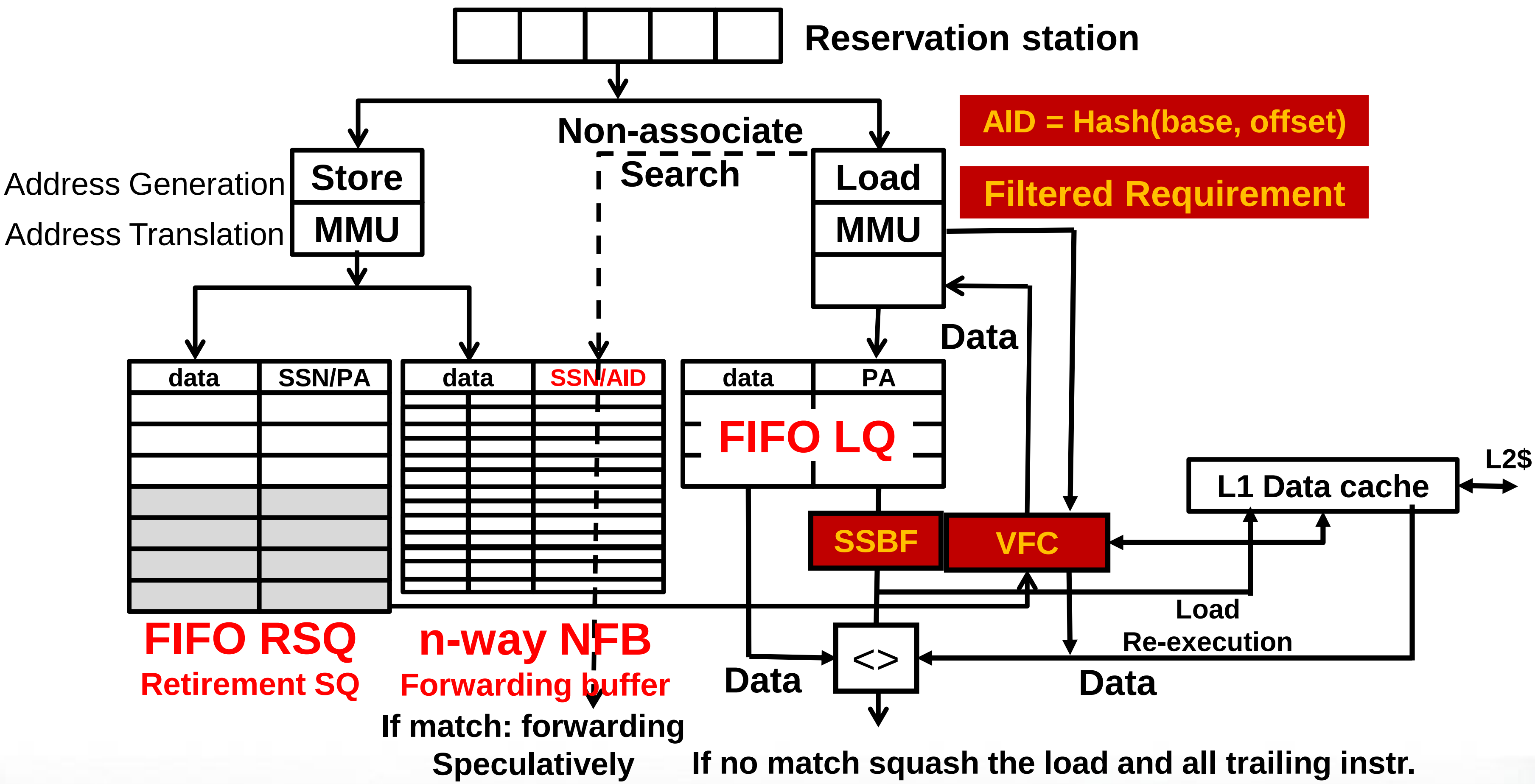
1. 地址计算
2. 访存相关处理
3. 高速缓存访问

1. 活跃存储指令窗口命中过滤不必要的高速缓存访问
2. 小标签宽度减少访问活跃存储指令窗口能耗

1. 伪命中增加不必要的高速缓存访问

ASW+20位AID	访存能耗降低比例
避免不必要的高速缓存访问	15.9%
减少活跃存储指令窗口能耗	8.9%
合计	24.8%

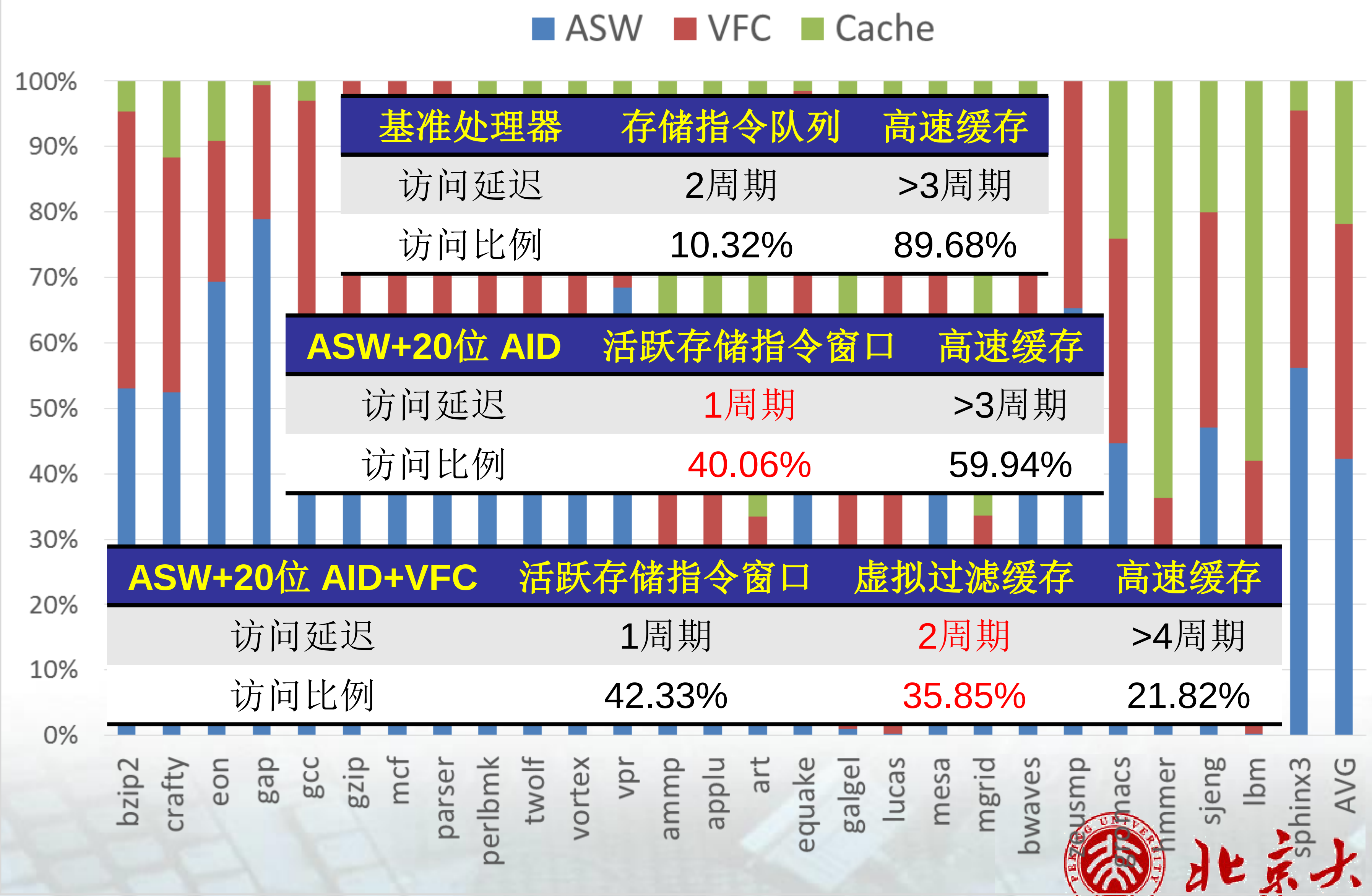
Optimizing 4: Virtual Filter Cache (VFC)



Assembling Speculative L0$ into SSBF: 2-cycle latency

FIFO RSQ+Large No-Associate FB +SSN/Bloom Filter

VFC Forwarding Ratio Evaluation



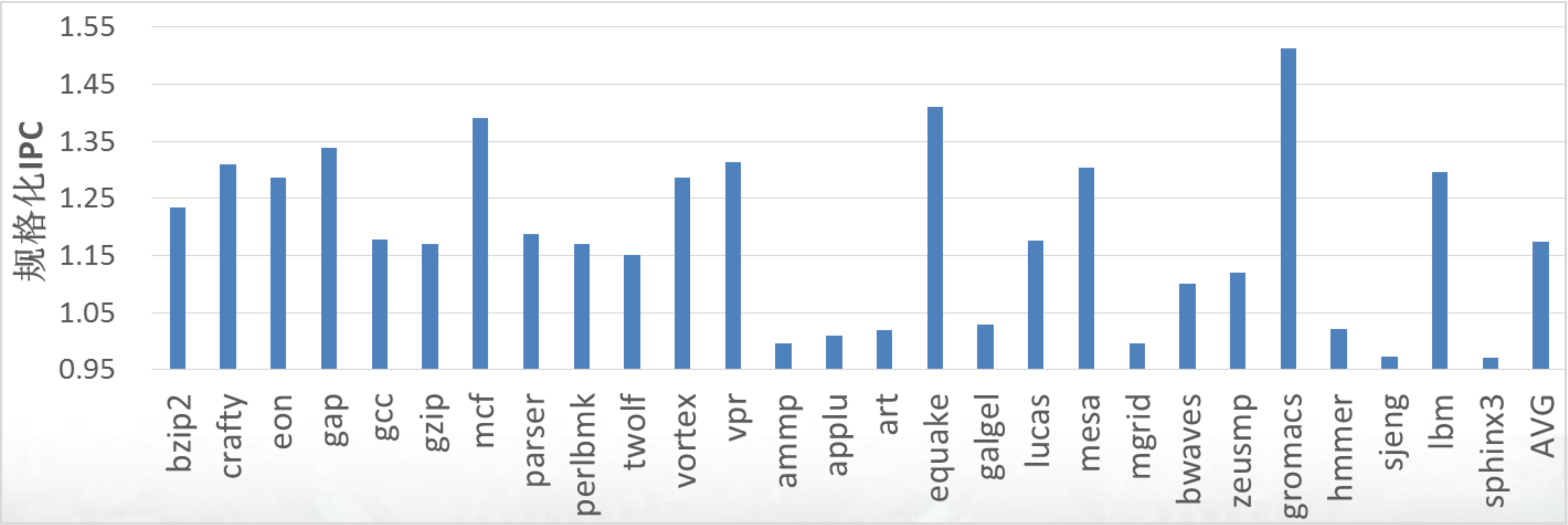
基准处理器	存储指令队列	高速缓存
访问延迟	2周期	>3周期
访问比例	10.32%	89.68%

ASW+20位 AID	活跃存储指令窗口	高速缓存
访问延迟	1周期	>3周期
访问比例	40.06%	59.94%

ASW+20位 AID+VFC	活跃存储指令窗口	虚拟过滤缓存	高速缓存
访问延迟	1周期	2周期	>4周期
访问比例	42.33%	35.85%	21.82%

VFC Performance Evaluation

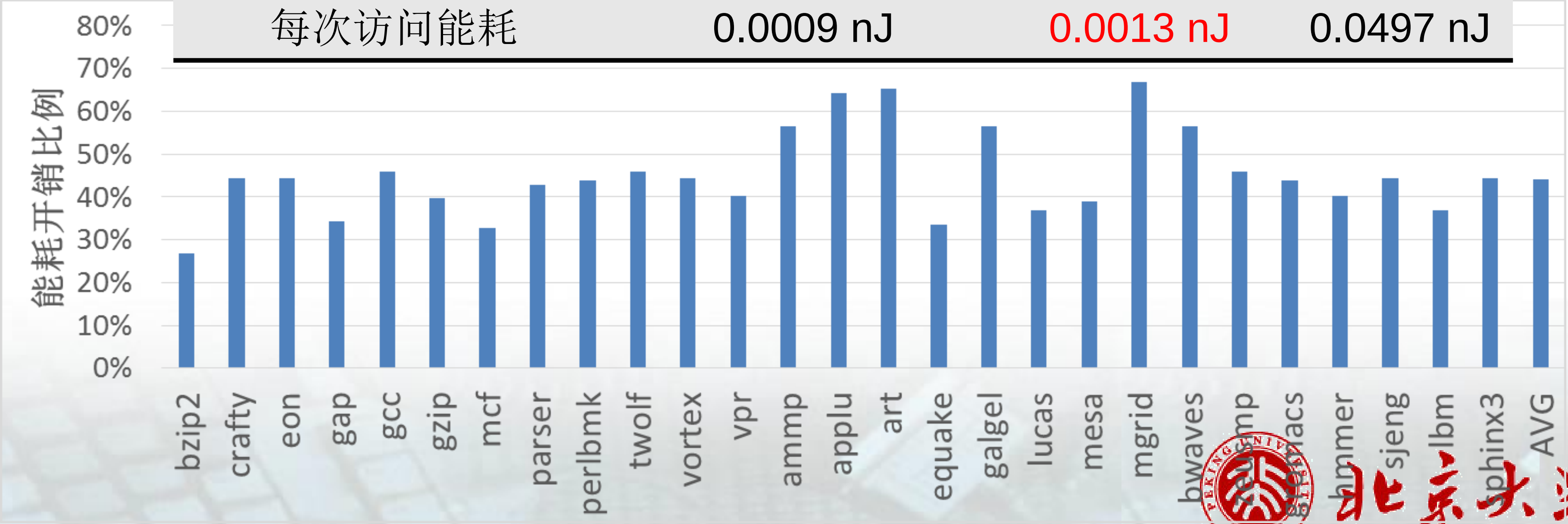
	性能提升
相对于基准处理器	17.42%
相对于原有ASW+AID设计	5.08%



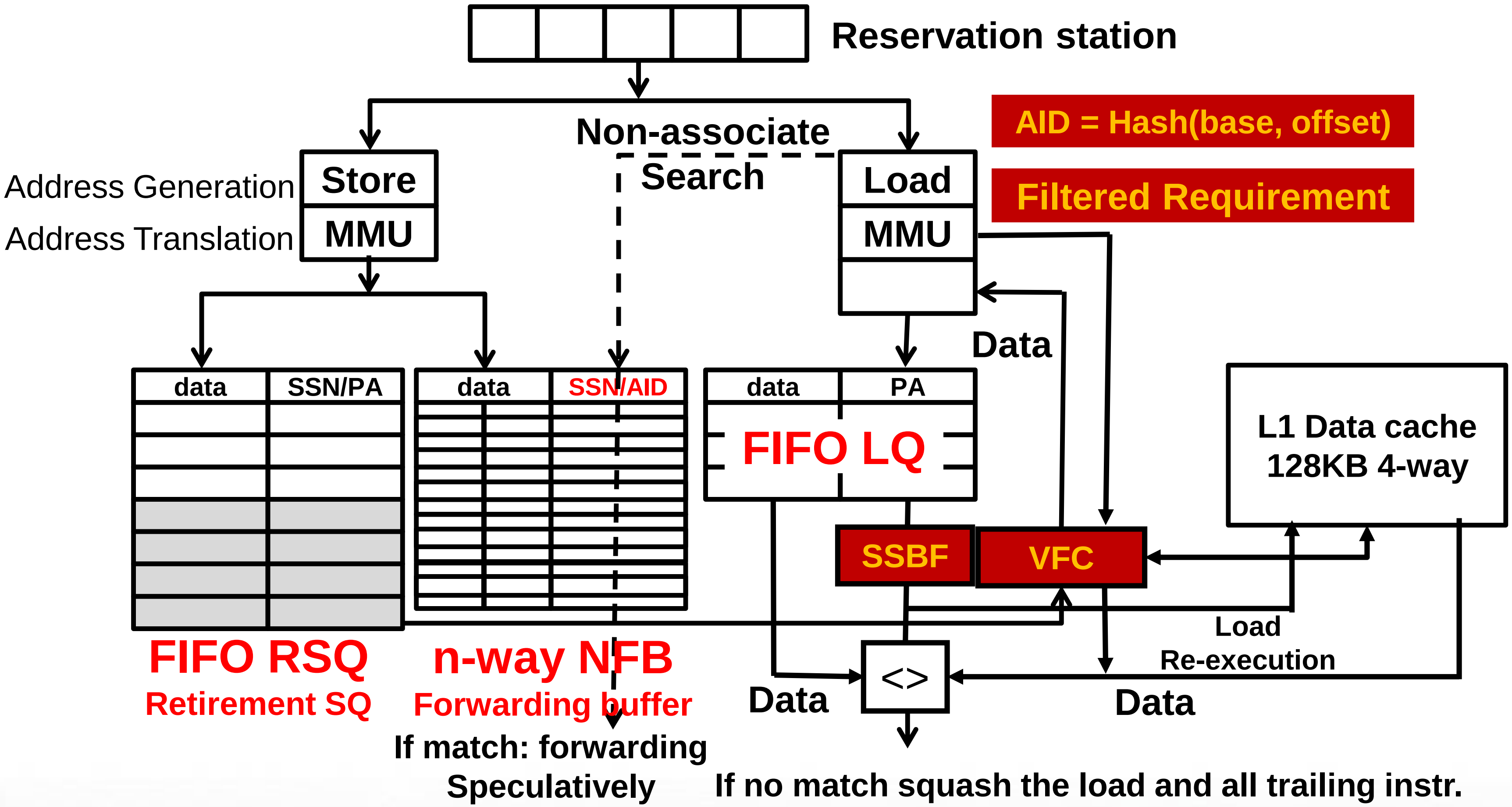
VFC Power Evaluation

ASW+20位 AID	活跃存储指令窗口	高速缓存
访问延迟	1周期	>3周期
访问比例	40.06%	59.94%
每次访问能耗	0.0009 nJ	0.0497 nJ

ASW+20位 AID+VFC	活跃存储指令窗口	虚拟过滤缓存	高速缓存
访问延迟	1周期	2周期	>4周期
访问比例	42.33%	35.85%	21.82%
每次访问能耗	0.0009 nJ	0.0013 nJ	0.0497 nJ



Optimizing 5: Larger Physical L1 Cache



Physical Cache, 128KB, 4-way, 32B Cache line, 3 cycles

FIFO RSQ+Large No-Associate FB +SSN/Bloom Filter

What’s Next?

- ELSE: ASW+SOLE+VFC+Huge-L1-Cache
- ELSE change the behavioral of Core memory accesses significantly.
- Future works:
  - L1 Data Cache prefetching (S/DC)
  - Support SMT/CPR/CFP…
  - Support memory consistency (SC!)
  - Support Cache coherence protocol (MOESI)
  - New Last-level Cache Replacement and Partition
  - New Main Memory Management (scheduling…)
  - Support multi-core compiler and OS