Zhang ZH, Wang XY, Tong D *et al.* Active store window: Enabling far store-load forwarding with scalability and complexity-efficiency. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(4): 769–780 July 2012. DOI 10.1007/s11390-012-1263-7

Active Store Window: Enabling Far Store-Load Forwarding with Scalability and Complexity-Efficiency

Zhen-Hao Zhang (张栘滈), Xiao-Yin Wang* (王箫音), Dong Tong (佟　冬), *Member*, *CCF*, *ACM*, Jiang-Fang Yi (易江芳), Jun-Lin Lu (陆俊林), and Ke-Yi Wang (王克义)

Microprocessor Research and Development Center, Peking University, Beijing 100871, China

Engineering Research Center of Microprocessor and System, Ministry of Education, Beijing 100871, China

School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

E-mail: {zhangzhenhao, wangxiaoyin, tongdong, yijiangfang, lujunlin, wangkeyi}@mprc.pku.edu.cn

Received June 3, 2011; revised April 23, 2012.

Abstract Conventional dynamically scheduled processors often use fully associative structures named load/store queue (LSQ) to implement the value communication between loads and the older in-flight stores and to detect the store-load order violation. But this in-flight forwarding only occupies about 15% of all store-load communications, which makes the CAM-based micro-architecture the major bottleneck to scale store-load communication further. This paper presents a new micro-architecture named ASW (short for active store window). It provides a new structure named speculative active store window to implement more aggressively speculative store-load forwarding than conventional LSQ. This structure could forward the data of committed stores to the executing loads without accessing to L1 data cache, which is referred to as far forwarding in this paper. At the back-end of the pipeline, it uses in-order load re-execution filtered by the tagged SSBF (short for store sequence bloom filter) to verify the correctness of the store-load forwarding. The speculative active store window and tagged store sequence bloom filter are all set-associate structures that are more efficient and scalable than fully associative structures. Experiments show that this simpler and faster design outperforms a conventional load/store queue based design and the NoSQ design on most benchmarks by 10.22% and 8.71% respectively.

Keywords store-load forwarding, load/store queue, value-based load re-execution

1 Introduction

Since the memory wall problem[1] gets worsen as increasing clock frequency, the memory latency, especially the load-to-use latency, becomes the major performance bottleneck of modern processors. Our experiments show that only about 15% of all store-load communications in conventional LSQ (load/store queue)-based processors are accomplished by in-flight communication, which could be implemented within one clock cycle. The others have to be accomplished by the memory hierarchy within several or hundreds of clock cycles.

To implement in-flight store-load communication, conventional approach uses age-based LSQ, which is typically implemented as two separated queues — the load queue (LQ) and the store queue (SQ). The LSQ buffers all memory instructions, commits their effects in program order and forwards values between communicating pairs of loads and stores. Sometimes, it also needs to detect incorrect speculation or potential violation of coherence and consistency.

In order to implement these functions, associative searches are used to find the correct producers or to detect dependence violations. Along with the increasing capacity of instruction window, the traditional LSQ with CAM-like structures is the most challenging micro-architectural structure to scale up, which in turn constraints the range of in-flight store-load communication.

The most intuitive approach to solve this problem is scaling up the capacity of LSQ by splitting the LSQ into two levels[2-4]. The first level LSQ with small capacity is used to satisfy the majority of in-flight communication quickly while the second level LSQ with large capacity and slower access latency is used to buffer more memory instructions. This kind of optimization only focuses on how to catch up with the increasing capacity of

Regular Paper

This work was supported by the National High Technology Research and Development 863 Program of China under Grant No. 2009ZX01029-001-002 and the Postdoctoral Science Foundation of China under Grant No. 20110490208.

*Corresponding Author

instruction window, and even may damage the in-flight store-load communication performance.

As a result, a number of implementations have been explored recently to avoid associative searches of the traditional design[2-3,5-9]. The fact these designs base on is that memory-based dependencies are very infrequent and predictable in the course of execution. So it is possible to reduce the number of associative accesses through clever filtering or prediction. Most of the optimized designs only focus on how to replace the CAM structures of LSQ and neglect to optimize the performance of in-flight store-load communication. So the most of communication between loads and stores is accomplished by the cache hierarchy, which would be within several cycles in modern processors and occupy the majority of load-to-use latency.

In this paper, we propose a more scalable and much simpler design called ASW (short for active store window), as an effective alternative to traditional CAM-based LSQ. This design includes a structure called speculative active store window to implement aggressive store-load forwarding. This structure could be viewed as the collection of latest stores to recently used memory addresses, including the store data, store address and the age of store instructions. The speculative store window is totally independent of the in-flight instruction window, which means that the load could get its forward data no matter whether the forwarding store instruction has been committed. So the ASW micro-architecture could provide a wider forwarding range than traditional LSQ, that make more load instructions fetch data from speculative active store window within one clock cycle. As a result, ASW micro-architecture improves the load performance. Despite of conventional proactive monitoring to detect violation, the correctness guarantee of ASW micro-architecture comes from the in-order commit of memory accesses. When a load instruction is committed, tagged SSBF (short for store sequence bloom filter)[10] based on the load/store commit order is accessed. Discrepancy between the results of two stages triggers re-execution of load and dependent instructions.

The primary contribution of this paper is the proposal of ASW micro-architecture. The goal of ASW micro-architecture is to achieve a wider range of store-load forwarding than the traditional load/store queue while removing timing-critical and non-scalable structures from the processor's out-of-order engine. Because of the wider forwarding range, ASW could forward more load instructions within one clock cycle than the traditional LSQ. Experiments show that the expanded range of the store-load forwarding by the ASW micro-architecture provides significant performance improvement, overcoming the negative effect by the mis-speculation of store-load forwarding and the re-execution of the load instructions. As a result, ASW micro-architecture outperforms a conventional out-of-order superscalar design and the NoSQ design[10] by 10.22% and 8.71% on average respectively. Meanwhile, as the absence of the CAM structure and the fully associative search, ASW could be implemented with faster and more power-efficient circuit than the traditional LSQ.

The rest of the paper is organized as follows: Section 2 recaps the basics of the memory dependence logic and highlights recent optimization proposals; Section 3 describes the micro-architecture of ASW design; Section 4 describes the experimental methodology and provides some quantitative analysis; Section 5 compares the difference between ASW design and other recent proposals; and Section 6 concludes the paper.

2 Background

Recent proposals[7-8,10-14] for the optimization of traditional LSQ use predictive or speculative approach to reduce the design complexity of memory disambiguation logic and to gain performance improvement at the same time. Most of these proposed micro-architectures base on the value-based load re-execution mechanism, which is proposed by [15]. The driving principle behind value-based re-execution is to shift complexity from the timing critical components of the pipeline to the back-end of the pipeline. In the value-based re-execution mechanism[15], loads are re-executed at the back-end of the pipeline and their results are checked against the premature load results. To support this load replay mechanism, two pipeline stages have been added at the back-end of the pipeline preceding the commit stage, labeled replay and compare. For simplicity, all instructions flow through the replay and compare stages, with action only being taken for load instructions. This optimization relieves the design burden to maintain correctness for the front-end execution, effectively relegating it to a value predictor.

Under conventional load speculation, only loads that actually issued in the presence of older un-executed stores — typically 10∼20% of all loads[15] — are speculative, and only these loads must be re-executed. Redundant re-execution of load instructions will incur performance loss or inefficient power cost due to contending a same data cache port with the committing stores.

Store Vulnerability Window (SVW)[16] is an address-based filter mechanism that dramatically reduces the re-execution rate for any form of speculation on loads with respect to older stores. SVW design bases on the observation that a load should not have to re-execute

if no store is written to a matching address in a sufficiently long time. SVW implements this basic idea using a small address-indexed table called store sequence bloom filter (SSBF) to track the store sequence number (SSN) of the latest committed stores that write to each hashed address. When a load executes, it accesses the tagged SSBF to get its SSNnvul, which is the SSN of latest committed store at the time of execution. Prior to commit, the load then accesses the SSBF again to get its SSNcmt, which is the SSN of latest committed store before the load is committed. When the SSNcmt equals SSNnvul, it means that there has not been any store with the same address committed during the execution of the load instruction. So it is impossible that the load incurs any store-load violation and needs any re-execution.

From another perspective, the only execution in value-based re-execution optimization that is absolutely required is the back-end execution. In theory, any front-end execution scheme would work, even if it only returns garbage values. Thus the front-end execution can afford to simplify the memory dependence enforcement logic because the back-end execution provides a safety net for incorrect speculation. There are two categories of front-end optimization based on the value-based load re-execution — memory dependence prediction and speculative cache.

Proposals in [7-8, 10-11] use memory dependence prediction[17-18], which base on the value-based re-execution micro-architecture, to implement speculative store-load forwarding by predicting the position of the forwarding data. So it avoids the fully associative search of traditional LSQ. The forwarding data could be placed in an age-ordered structure with speculative indexed access[7-8,11], or in the register file with speculative memory bypassing[10].

Besides the memory dependence prediction, [12] proposed another mechanism for eliminating associative LSQ, which pushes the forwarding simplification to the extreme by completely eliminating any memory dependence enforcement logic. In the first phase, loads speculatively obtain their value from a speculative L0 cache. In the second phase, the load instructions are re-executed in program order, without any speculation, and access the regular L1 cache. Any difference in the load values between the two phases results in corrective action. This speculative L0 cache could offer data of committed store to the executing load, which is accessed within one cycle.

3 Active Store Window

Active store window indicates the set of efficacious stores, which are committed recently and not overlapped by other stores. The memory hierarchy of modern processors could be viewed as a determined implementation of the active store window, which only includes the data of stores. The ASW micro-architecture uses more natural way to implement a speculative cache of active store window, which is used to forward store data to load speculatively. This speculative cache uses a smaller and simpler structure to implement wider store-load forwarding range than traditional LSQ, which does not damage the forwarding accuracy significantly.

3.1 Conception

Any load data comes from some store committed before the load instruction, and any store data will be overlapped by the data of some store committed after. If the data of the store has been covered by other stores, it is called *dead store*. The name *active store* refers to the store which has not be dead. For any load instruction, the set of all active stores committed before it is called the *active store window* of this load instruction. Thus, any load instruction must get its data from some store in its active store window. Maybe the load data comes from several stores in active store window — called partial forwarding — which will be simplified in this discussion because of its unusuality. Besides the data of stores, active store window also includes the age information between store instructions, which could be used to determine the store order.

Active store window in traditional processors is only maintained in the memory hierarchy, which is updated when store instructions are committed. But it is organized according to the address space which omits the age information of different stores. This kind of absence of age information is not suitable for the speculative execution of loads, in which the execution might be incorrect.

Take the speculative L0 cache in [12] as an example. The L0 cache is organized as a traditional set-associate structure which is accessed only according to the addresses of load/store instructions. So when a load instruction accesses the L0 cache, it is impossible to recognize the related store that forwards the data to the load instruction, and also impossible to implement any re-execution filtering mechanism. And when a wrong store instruction is executed, its data would be written in the L0 cache without distinction to other correct data in the same cache line. This polluted cache line might continue to forward incorrect data to following load instructions until it is cleaned up or replaced.

3.2 Speculative Active Store Window

This paper proposes a more natural way to

implement the active store window, which is called speculative active store window. This structure could be viewed as a speculative cache of active store window with a typical set-associate structure, but it contains the information about the forwarding store's age relationship with other stores.

Any entry of speculative active store window includes following fields: 1) valid flag (V), 2) address tag (TAG), 3) byte valid flag (BE), 4) store sequence number (SSN), and 5) store data (DATA).

Throughout this paper, all executed store instructions are identified using store sequence numbers (SSN), which form the basis of the active store window scheme and are more convenient than store queue indices because they also represent committed stores. Thus the age relationships between store instructions could be converted to the order relations between SSNs. When the SSN of store instruction A is less than B, it indicates that the store instruction A is older than B. And vice verse. In ASW micro-architecture, any dynamic store instruction is assigned monotonically increasing SSN when it is renamed. A global counter — SSNren — tracks the SSN of the most recently renamed/dispatched store. In the rare situations in which SSNren wraps around, the processor drains its pipeline and clears all hardware structures that hold SSN, including speculative active store window and tagged store sequence bloom filter.

Fig.1 shows the comparison between speculative active store window and traditional store queue. As shown in Fig.1, the store instruction $S1$ and store instruction $S2$ with the same address in traditional store queue would be placed in the same set of speculative active store window, which is assumed a 2-way set-associate structure.

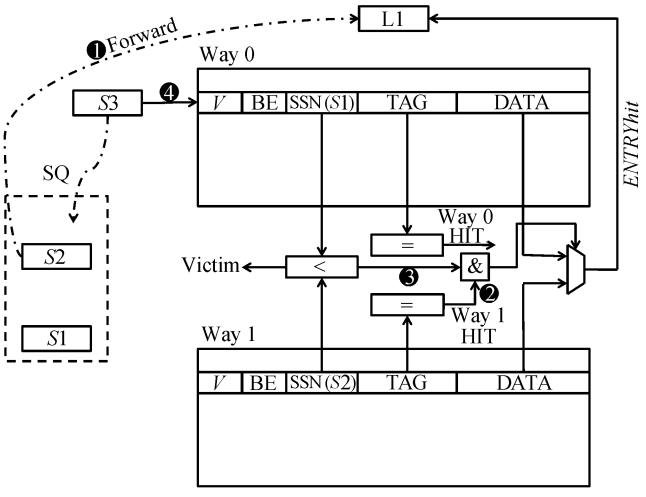


Fig.1. Comparison between speculative active store window and traditional store queue.

When the load (L1) with same address executes, it would associatively search the store queue to find the youngest store instructions (❶ in Fig.1). Instead of the traditional store queue, when the load with same address accesses the speculative active store window, it will get two variables: 1) HIT flag and 2) *ENTRYhit*. The HIT flag indicates whether there is any hit entry in this read access to the speculative active store window (❷ in Fig.1). If the HIT flag is true, *ENTRYhit* will present the youngest entry among the hit entries. And the order relationship between these stores could be determined according to the SSNs (❸ in Fig.1).

When another store instruction with the same address executes ($S3$ in the figure), the replacement policy of the speculative active store window follows the principle of active store window. When there is any invalid entry in the indexed set of the speculative active store window, it is occupied as a priority. When all entries in the same set are valid, the entry that has the oldest SSN is replaced firstly (❹ in Fig.1, store instruction $S3$ would replace the store instruction $S1$). But when the SSN of the writing store is older than the oldest store in the set, none of the entry in this structure is replaced. As a result of the replacement policy, the too old store is avoided into the speculative active store window.

When the speculative active store window is accessed, the least significant bits of addresses are hashed to index it. The most significant bits of addresses is compared with the tag field to assert a hit. Since the hash function is serialized with the set-associate RAM structure, the hash function has to be fast to compute with zero or one level of logic. The adopted hash function incurs a delay of only one gate level of logic (a 2-input XOR gate) by XORing the least significant bits of the addresses to generate the index of the speculative active store window. While the hash function increases the computation cost, it reduces the probability of the collision in the speculative active store window.

In speculative active store window, each line contains a store instruction and its data. Any store instruction is marked as its store sequence number. This organization guarantees that any load could find the store that provides the correct data. And this information would be used in the re-execution filtering. As our experiments show, the simplicity of speculative active store window does not decrease the forward accuracy of load execution significantly. Another advantage is that the replacement policy of speculative active store window, which bases on the SSNs of store instructions, will avoid the necessary to clean up whole structure within interval period. Because the incorrect data will not pollute the data of other stores, and will be replaced eventually as the execution proceeds.

3.3 ASW Micro-Architecture

Fig.2 is the block diagram of the ASW micro-architecture, which shows the schematic of the processor pipeline. Fig.2(a) shows the store instruction execution datapath, and Fig.2(b) is load.

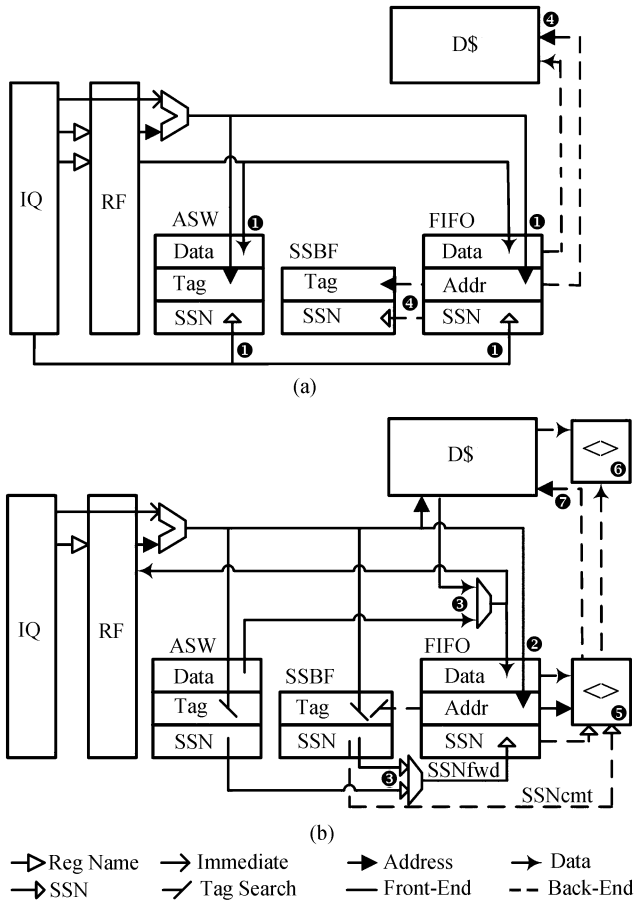


Fig.2. ASW Micro-architecture. (a) Store datapath. (b) Load datapath.

For all memory instructions, an out-of-order front-end execution is performed as in a normal processor at the execution stage. However, in the ASW design, memory instructions are issued entirely according to their register dependencies. No other interference is imposed. As a result, it is possible to violate memory dependencies and this makes the front-end execution fundamentally speculative. Thus the speculative active store window accessed in the front-end execution does not propagate any result to L1 data cache.

To detect violations, memory accesses are performed a second time, totally in-order at the commit stage of the pipeline. Any load that obtains different data from the two executions will take the result of the back-end execution and trigger a squash and replay of subsequent instructions. In the ASW design, a tagged SSBF is used to filter the re-execution of load instruction. The difference between the original SSBF in [16] and the tagged SSBF in ASW design is that tagged SSBF is a set-associate structure instead of a direct-map structure. When the load instruction misses in the tagged SSBF, the oldest SSN in the set would be returned as the SSNcmt.

FIFO in Fig.2 is used to maintain the calculated address and load/store data according to the program order, which could be viewed as a complement to the reorder buffer (ROB). FIFO is only accessed by memory instructions when they are issued, executed and committed without any fully associative search, which leads to a simpler implementation of FIFO.

3.3.1 Front-End Execution

As explained earlier, central to our design support is an unconventional speculative active store window. At the issue time of a store instruction, it simply writes its address, store data and SSN into the speculative active store window and FIFO (❶ in Fig.2). Since the speculative active store window is used to handle common cases, its control is kept extremely simple. No attempt is made to clean up the incorrect data left by wrong-path instructions. When an entry is replaced, it is simply discarded, no matter whether it is dirty.

What should be explained clearly is that when load instructions execute at front-end execution it would access speculative active store window, L1 data cache and tagged SSBF simultaneously. This would make sure that when the load instruction misses in the speculative active store window, it could get the data from L1 data cache and the SSN from the tagged SSBF just like what [16] does (❷ and ❸ in Fig.2). If it hits in the speculative active store window, the load instruction would return data immediately and discard the result from the L1 data cache and tagged SSBF.

The most significant feature of the speculative active store window is the transient replacement. According to the replacement policy, too old stores are avoided into the speculative active store window. Meanwhile, the oldest entry in the speculative active store window will be replaced eventually. And the replaced entry will be discarded directly. There might be false entries in the speculative active store window because of the misprediction of the branch instructions or the false speculative execution of other instructions (e.g., the load instructions). The transient feature of the speculative active store window guarantees that the false store data will be eventually replaced by the younger stores, so there is no need to implement extra flush mechanism when the false speculative execution occurs.

Since the store data does not need to clear when the

store instruction is committed, there may be a situation in which the forwarding data is from a committed store instruction. This forwarding from a committed store is named *far forwarding* in this paper. Since the latency of access to speculative active store window is less than the L1 data cache, far forwarding could be used to reduce the execution latency of the hit load instruction, which will improve the performance of the ASW micro-architecture.

3.3.2 In-Order Back-End Execution

In-order re-execution to validate a speculative execution is not a new technique. However, the extra bandwidth requirement for re-execution makes it undesirable or even impractical.

As the store sequence number of forwarding store could be got at the front-end execution of the load instruction, it is very convenient to use the tagged SSBF proposed by [10] to implement the filtering of load re-execution.

When stores are committed, their SSNs and store data would be written into the tagged SSBF and L1 data cache respectively (❹ in Fig.2).

When a load instruction is ready to commit, the load will access the tagged SSBF to make sure whether it needs to re-execute. Just like the SVW scheme applying an SSNnvul to any renamed load instruction, the ASW design applies an SSNfwd to any executed load instruction (❸ in Fig.2). The SSNfwd has two sources: 1) when there is a forwarding hit, the SSNfwd is the SSN of the forwarding store instruction; 2) when there is a forwarding miss, the SSNfwd is the SSN of the oldest store in the SSBF.

The SSNfwd would be compared with the SSNcmt that is got when the load is committed (❺ in Fig.2). When they are not equal, it means that the load instruction might be executed incorrectly. Then the load would access the L1 data cache again (❼ in Fig.2), and the returned data would be compared with the forwarding data got at the time of execution (❻ in Fig.2), in order to determine whether the forwarding data is correct.

3.3.3 Partial Problem

Partial-word communication is the situation in which the load data comes from more than one store data. Because of the unusual ratio — about 3% of the total loads[10] — of this kind of communication, the ASW design simply ignores it at the front-end execution. When the load instruction finds a partial hit in the speculative active store window, it would be treated as a miss and get its data from the L1 data cache. But when this instruction commits, this kind of partial hit would usually be treated as a hit in the tagged SSBF and triggers a re-execution, which would provide guarantee for the correctness of the load instruction.

3.4 Execution Examples

In order to describe the detail of the ASW micro-architecture further, an example of the instruction sequence is proposed, which is listed in Fig.3. According to the instruction sequence, $I1$, $I2$ and $I4$ are three store instructions to the memory address A, but with different store data. And $I3$ is a conditional branch instruction, the target of which is $I5$ that is a load to the same memory address as the stores.

```
I1: Store D1, [A]
I2: Store D2, [A]
I3: BEQ Target
I4: Store D3, [A]
 ⋮
Target:
I5: Load [A]
```

Fig.3. Example of instruction sequence.

Three typical scenes of the execution of these instructions are picked: correct speculative forwarding, false speculative forwarding and false speculative execution. In order to make description concise, the speculative active store window and tagged SSBF are all assumed as a 2-way set associative structure.

Correct Speculative Forwarding. In this scenario, the $I3$ instruction is assumed to be correctly predicted as a taken branch. And Fig.4 describes the execution trace of these instructions. The $I1$, $I2$ and $I5$ instructions are executed according to the program order. Then, the speculative active store window and tagged SSBF are filled with the SSN of $I1$ ($S1$) and $I2$ ($S2$) successively. So $I5$ gets the forwarding data from $I2$ in the speculative active store window, and its SSNfwd is $S2$ which equals the SSNcmt from the tagged SSBF when it is committed. And that means the $I5$ load is correctly forwarded by speculative active store window.

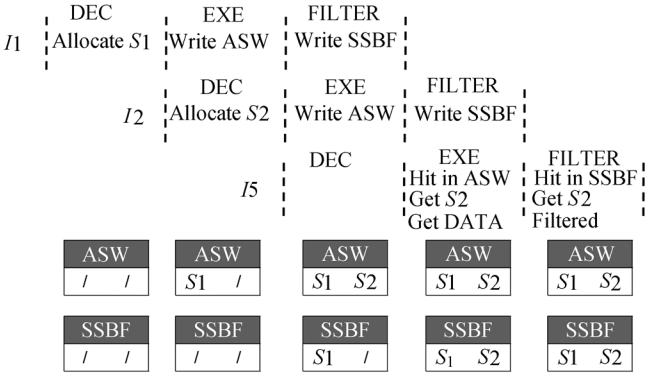


Fig.4. Correct speculative forwarding.

False Speculative Forwarding. Fig.5 shows the scenario of false speculative forwarding. This scenario is similar to the previous one. The only difference is that the $I5$ instruction is speculatively executed before the $I2$ instruction, which leads to that the $I5$ instruction gets a false forwarding data from the $I1$ instruction ($S1$). So when $I5$ is committed, its SSNfwd ($S1$) is less than the SSNcmt from the SSBF ($S2$), which means $I5$ gets wrong store data. And a re-execution is triggered.

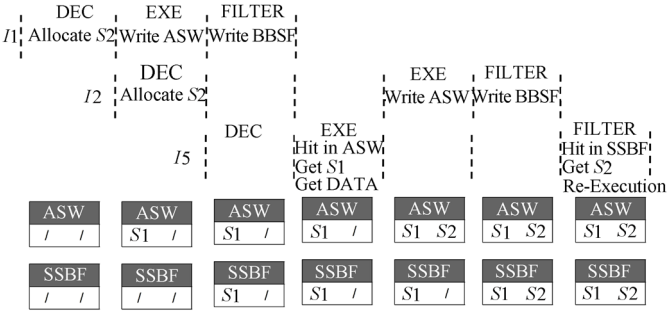


Fig.5. False speculative forwarding.

False Speculative Execution. Fig.6 shows the scenario of false speculative execution. This scenario is used to describe how the ASW micro-architecture avoids the effect by the mis-speculation of other instructions. It is assumed that $I3$ is mis-predicted as a not-taken branch, which causes the $I4$ instruction to be executed before it is squashed. $I5$ is mis-forwarded by the speculative active store window, and its SSNfwd is $S3$. But $I4$ is not committed, so the tagged SSBF is not updated and will check out this mis-forwarding, because the SSNcmt would be $S2$.

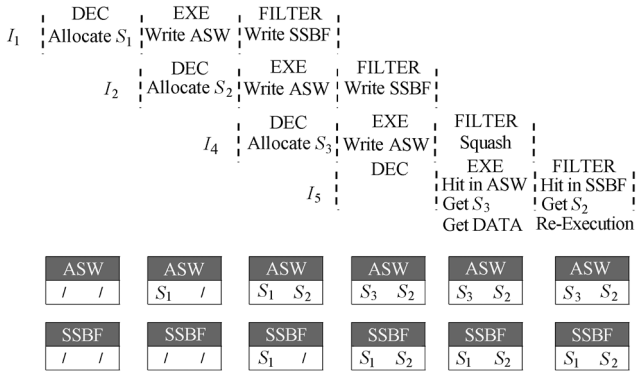


Fig.6. False speculative execution.

4 Experimental Evaluation

4.1 Methodology

We use cycle-level simulation to evaluate the performance of the ASW micro-architecture. The simulator is from SimpleScalar 3.0 for the Alpha instruction set architecture. We modified the simulator to include per-commit load re-execution and implemented our ASW micro-architecture. We also modeled the unlimited conventional LSQ design and the NoSQ design in [10]. The parameters of simulation is listed in Table 1.

Table 1. Parameters of Simulation

Processor Core
Issue/Decode/Commit Width: 4/4/4
Instruction Window: 128-entry
Functional Units: INT 4+1 mul/div, FP 4+1 mul/div
Branch Predictor: Hybrid Bimodal
– Predictor Entries: 4K
– Branch Target Buffer Entries: 2K (4-Way)
– Return Address Stack Entries: 32
Memory Hierarchy
L1 ICache: 32 KB, 8-Way, latency=3 cycles
L1 DCache: 32 KB, 8-Way, latency=3 cycles, 2 ports
L2 Unified Cache: 1 MB, 8-Way, latency=10 cycles
TLB: 128-entry, 4-Way
Memory Latency: 150 cycles

The front-end and execution pipelines have total eight stages: one stage to predict, three stages to fetch, one stage to decode/rename, one stage to dispatch, one stage to execute, and one stage to commit. stage to Data cache latency is three cycles, so the load pipeline is 11 stages. The baseline is a conventional micro-architecture based on the fully associative load/store queues. It has a 32-entry store queue and a 32-entry load queue respectively.

ASW micro-architecture has an additional 4-stage back-end pipeline: one stage for load re-execution filtering and three stages for load re-execution. The speculative active store window and tagged SSBF are all 256-entry 4-way set-associative. Entry size of speculative active store window and tagged SSBF is 13 B and 8 B respectively.

The modeled NoSQ design has the same configuration as proposed by [10]. First of all, it has an additional 5-stage back-end pipeline: one load address calculation, one load re-execution filtering and three data caches for load re-execution. Secondly, the memory dependency predictor uses two 1 K-entry, 4-way set-associative tables. Finally, the NoSQ design also uses tagged SSBF for load re-execution filtering, which is 128-entry 4-way set-associative.

Our quantitative analysis uses highly-optimized Alpha binaries of the SPEC2000 benchmark suite①. We simulate half a billion instructions after fast-forwarding one billion instructions.

4.2 Performance

There are two factors affecting the performance of

①http://www.spec.org/cpu2000/.

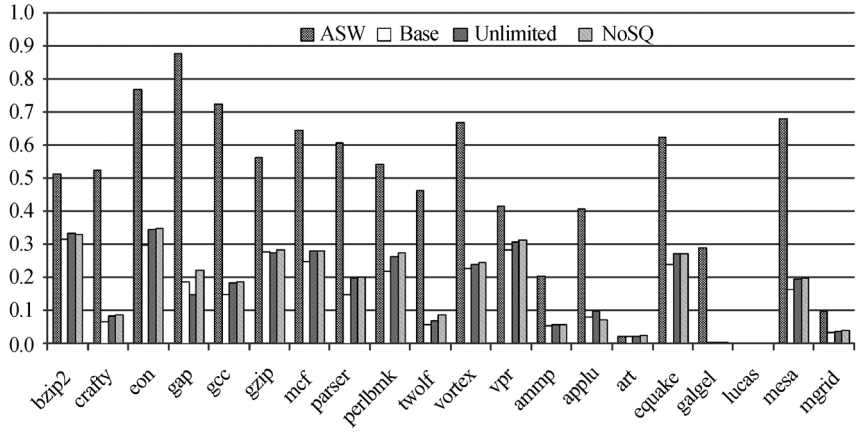


Fig.7. Forward ratio comparison between baseline (Base), unlimited LSQ (Unlimited), NoSQ and ASW.

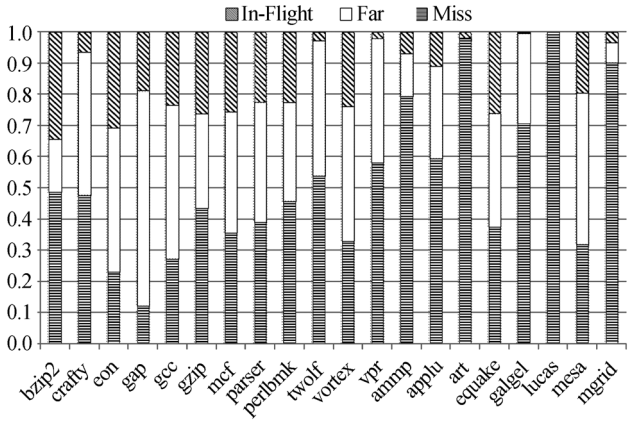


Fig.8. Forward ratio subdivision of ASW micro-architecture.

ASW micro-architecture: store-load forwarding efficiency and load re-execution filtering efficiency. More load is forwarded correctly and more redundant load re-execution is filtered, better is the performance of ASW design.

The store-load forwarding performance is determined by how many loads are forwarded and how accurate the forwarding is.

Firstly, because of the wider forwarding range by far forwarding, the ASW micro-architecture could forward more in-flight load instructions and obtain more instruction level parallelism without any extra cost on cycle timing path and power. Fig.7 compares the forwarding ratio between baseline pipeline, unlimited LSQ design, NoSQ and ASW design. The forwarding range of conventional pipeline and memory dependency prediction is only restricted in the in-flight instruction window, which could only reach 16.87% on average at most — the forwarding ratio of NoSQ design. But as described above, ASW design could implement far forwarding and obtain higher forwarding ratio, which is up to 48.1%.

Fig.8 shows the forwarding ratio subdivision of total load instructions in ASW micro-architecture. According to Fig.8, average 15.24% of loads is forwarded by in-flight stores, which is slightly less than the NoSQ design. Meanwhile, another 32.86% is forwarded by far forwarding on average, which is more than twice the in-flight forwarding.

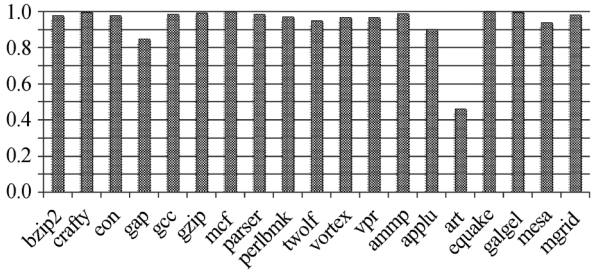


Fig.9. Forward accuracy of ASW micro-architecture.

Secondly, the accuracy of forwarding by speculative active store window is also an important metric about the forwarding effect of ASW micro-architecture, which reaches up to 93.19% on average as shown by Fig.9.

The second factor on the performance of ASW micro-architecture is the accuracy of load re-execution filtering, which will increase the penalty of load instruction's mis-speculation. The more load instructions are re-executed, the worse performance the ASW micro-architecture obtains. ASW micro-architecture uses a kind of modified tagged SSBF[10] to implement the filtering mechanism, which is also used in the NoSQ design. In spite of the modification, the modified tagged SSBF still could filter the majority of the load instructions, by 91.78% on average.

As explained above, the ASW micro-architecture gains performance improvement by forwarding more load instructions than traditional LSQ design. Fig.10 shows the relative instruction per cycle (IPC) of the ASW micro-architecture to the baseline. On average, the ASW design outperforms the baseline on the

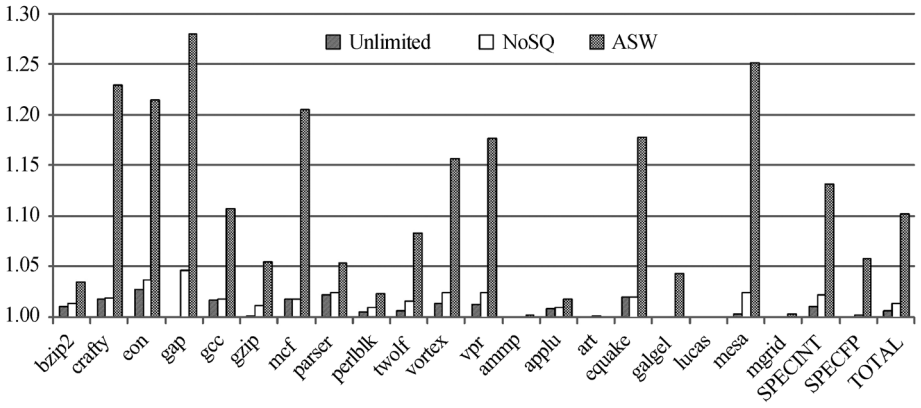


Fig.10. ASW micro-architecture performance.

SPECINT2000 and SPECFP2000 by 13.23% and 5.85% respectively.

NoSQ design is a more scalable forwarding scheme, which predicts the store-load violation and advances the forwarding to the rename stage. But it is also restricted in in-flight store-load forwarding. As Fig.10 shows, ASW design outperforms the NoSQ design by 8.71% on average.

4.3 Sensitivity Analysis

Fig.11 shows the ASW's performance sensitivity (in terms of average relative IPC to the baseline, as solid lines show) and scalability (in terms of GHz, means the max clock frequency supported by this configuration, as dotted line shows) to different configurations. Fig.11 gives 12 configurations of different combinations of the capacity (128-entry, 256-entry and 512-entry) and set associativity (direct mapped (DM), 2-way, 4-way and 8-way). Because the entry number of ASW has very little effect on the access latency compared to the set associativity, only one dotted line is draw in Fig.11, which presents the 512-entry ASW design. Even up to 512-entry with 8-way, the access latency to the ASW is only 1.04 ns, and the IPC is improved by 11.15% on average at the same time.

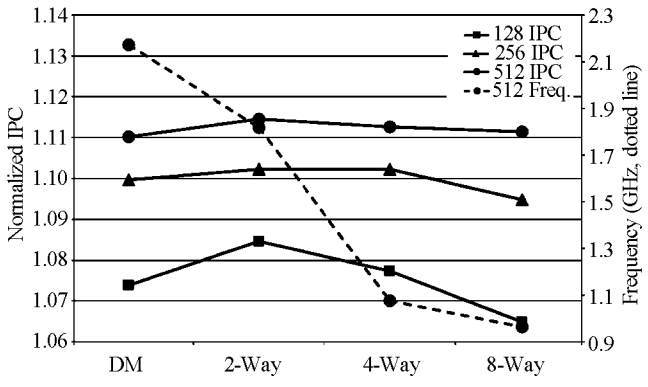


Fig.11. ASW micro-architecture performance sensitivity and scalability.

We could get several observations from Fig.11. First of all, from superimposition of the three solid lines, we could find that increasing capacity will provide significant IPC improvement of ASW micro-architecture. This is because the larger capacity will support more store-load forwarding, especially far forwarding. The second observation is that: when the capacity remains unchanged, as the set associativity increases, the IPC does not always increase. When the set associativity is increased to 2-way, the IPC increases simultaneously, because the collision probability of the same set in 2-way ASW design is smaller than direct-mapped ASW design. Once the set associativity outnumbers 2, the IPC decreases, which means that the declined number of sets in ASW structures would have very significant negative effect on performance and offset the improvement gained by increasing set associativity. Therefore, according to the figure, 2-way ASW design is always the better choice than others in the same capacity.

4.4 Design Complexity

Besides the IPC improvement, ASW micro-architecture provides more scalable structures than traditional CAM-based LSQ. The dotted line in Fig.11 shows the maximal supported clock frequency by the ASW structures, which is elaborated by CACTI 4.1. There is only one dotted line in Fig.11, because capacity has negligible effect on the access latency. But the set associativity is the reverse. More ways contained in ASW design imply more complex comparison and selection logic, and significant decrease of access latency. This rule also applies to the fully associative load/store queue. According to Fig.11, ASW design could obtain the best performance speedup by using only 2-way association.

Different from set-associative structures, fully associative structures are used to construct the load/store queue by conventional out-of-order processors. Fully associative structures are more sensitive to the size, because every entry in the fully associative structures should be compared and searched. Table 2 lists the access latency (in terms of nanosecond) of three different organizations, including fully associative, 4-way set-associative and 8-way set-associative. Apparently, according to the table, the fully associative structures are much more non-scalable than set associative structures. This non-scalability is hard to satisfy the timing and capacity requirements at the same time.

Table 2. Access Latency of Full-Associative and Set-Associative

	Fully Associative	4-Way	8-Way
32-entry	1.44	0.94	0.95
64-entry	1.45	0.94	0.96
128-entry	1.65	0.95	1.06
256-entry	2.14	0.95	1.06
512-entry	3.34	1.05	1.07
1024-entry	7.30	1.42	1.43

NoSQ design has been proved an efficient mechanism to make scalable and simple store-load forwarding, using memory dependency prediction. NoSQ also uses the set-associative structures to replace the original LSQ design. Table 3 compares the design complexity between NoSQ and ASW. It shows that, using speculative forwarding techniques, ASW design does not introduce more pipeline stages and capacity of transistors than NoSQ design.

Table 3. Design Complexity of NoSQ and ASW

	NoSQ	ASW
Forward Structure	10 KB	3.25 KB
Number of Filter Stages	5	4
Filter Structure	1 KB	2 KB

5 Related Work

There have been several recent proposals that optimize store-load forwarding without CAM-based store queue, which also base on the value-based load re-execution for memory disambiguation.

In [7], if a producer can be pin-pointed, the load forwards directly from the store's SQ entry. However, if there are several potential producers, another predictor provides a delay index and instructs the load to wait until the instruction indicated by the delay index commits.

Fire and Forget (FnF)[11] is another concurrently-proposed alternative scheme for eliminating the store queue. FnF accomplishes this by turning store-load forwarding from a load-centric activity to a store-centric activity and using load queue index prediction to perform forwarding through the load queue instead.

In [8], loads predicted to be dependents of stores are delayed. When allowed to execute, it accesses a store forwarding cache (SFC) and a memory dependence table. This design avoids the need to exactly predict the identity of the producer as memory addresses help to further clarify the producer.

NoSQ[10] puts the memory dependence prediction optimization to the extreme. It implements store-load communication by dynamic short-circuiting of DEF-store-load-USE chains to DEF-USE chains, where the USE instruction could get needed data from register file directly.

[19] and [20] discuss the situation where prior techniques[7,10] are applied to a more aggressive pipeline design[21-22] than conventional superscalar pipelines.

As explained above, ASW design uses the similar mechanism as these works above: speculative store-load forwarding and filtered load re-execution to detect the mis-speculation. The difference between ASW design and these prior works is the forwarding methods, which all base on the memory dependency prediction in prior works. Although research has shown that dependence relationship between static load and store instructions is predictable[17], it has two limitations. Firstly, it always has extra overhead to train the predictor and keep a large history table for prediction. Secondly, the predictor only predicts the dependence between in-flight load-store pairs.

The driving principle of ASW micro-architecture is that, even if the memory instructions are issued simply on the register dependencies and their memory dependencies are completely disregarded, about 98% of loads could obtain a correct value from the out-of-order executed stores[12]. So the ASW design replaces the memory dependency predictor with the speculative active store window, which maintains the store data and forwards it to the loads. The most important advantage of speculative active store window is using SSN to determine the order relationship between stores, which leads the speculative active store window to be independent of the in-flight instruction window and to implement far forwarding correctly.

6 Conclusions

At the beginning of this paper, we have introduced the concept of active store window for any load. The active store window includes all of the active stores before the specified load and all the data needed by the load comes from its active store window.

Conventional fully associative load/store queue could be viewed as a kind of in-flight active store

window, which includes the store data sources and their age relationship. But it is so hard to increase the capacity of the load/store queue in order to accommodate large instruction windows, because of the latency and dynamic power consumption of store-load forwarding and memory disambiguation.

In this paper, we propose a new micro-architecture called ASW, which uses speculative store-load forwarding and filtered load re-execution mechanism. In this design, speculative active store window is used to implement the speculative store-load forwarding. It includes all information needed by speculative store-load forwarding and is comprised of set-associate structures. This kind of implementation has two important advantages.

Firstly, different from traditional load/store queue, speculative active store window uses SSNs to determine the age relationship between stores, which is independent of the in-flight instruction window and could be used to determine the age relationship between committed stores. This new feature makes the ASW design can implement far forwarding correctly, in which the forwarding store has been committed. Far forwarding enlarges the forwarding range of speculative active store window and optimizes loads' execution latency.

Secondly, ASW micro-architecture does not introduce any CAM-like structure that would be non-scalable. Compared to another scalable design — NoSQ, ASW design also does not introduce any extra cost on pipeline stage and hardware cost. Considering the speculative active store window and tagged SSBF have similar basic structure and same replacement policy, ASW design also has less design complexity than NoSQ design.

References

[1] Wulf W A, McKee S A. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 1995, 23(1): 20-24.

[2] Park I, Ooi C L, Vijaykumar T N. Reducing design complexity of the load/store queue. In *Proc. the 36th MICRO*, San Diego, USA, Dec. 3-5, 2003, pp.411-422.

[3] Gandhi A, Akkary H, Rajwar R, Srinivasan S T, Lai K. Scalable load and store processing in latency tolerant processors. In *Proc. the 32nd ISCA*, Madison, USA, June 4-8, 2005, pp.446-457.

[4] Pericàs M, Cristal A, Cazorla F J, Gonzàlez R, Veidenbaum A, Jimènez D A, Valero M. A two-level load/store queue based on execution locality. In *Proc. the 35th ISCA*, Beijing, China, June 21-25, 2008, pp.25-36.

[5] Sethumadhavan S, Desikan R, Burger D, Moore C R, Keckler S W. Scalable hardware memory disambiguation for high ILP processors. In *Proc. the 36th MICRO*, San Diego, USA, Dec. 3-5, 2003, pp.399-410.

[6] Baugh L, Zilles C. Decomposing the load-store queue by function for power reduction and scalability. *IBM Journal of Research and Development*, 2006, 50(2/3): 287-297.

[7] Sha T T, Martin M M K, Roth A. Scalable store-load forwarding via store queue index prediction. In *Proc. the 38th MICRO*, Barcelona, Spain, Nov. 12-16, 2005, pp.159-170.

[8] Stone S S, Woley K M, Frank M I. Address-indexed memory disambiguation and store-to-load forwarding. In *Proc. the 38th MICRO*, Barcelona, Spain, Nov. 12-16, 2005, pp.171-182.

[9] Roesner F, Burger D, Keckler S W. Counting dependence predictors. In *Proc. the 35th ISCA*, Beijing, China, June 21-25, 2008, pp.215-226.

[10] Sha T T, Martin M M K, Roth A. NoSQ: Store-load communication without a store queue. In *Proc. the 39th MICRO*, Orlando, USA, Dec. 9-13, 2006, pp.285-296.

[11] Subramaniam S, Loh G H. Fire-and-forget: Load/store scheduling with no store queue at all. In *Proc. the 39th MICRO*, Orlando, USA, Dec. 9-13, 2006, pp.273-284.

[12] Garg A, Rashid M W, Huang M. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. In *Proc. the 33rd ISCA*, Boston, USA, June 17-21, 2006, pp.142-154.

[13] Sethumadhavan S, Roesner F, Emer J S, Burger D, Keckler S W. Late-binding: Enabling unordered load-store queue. In *Proc. the 34th ISCA*, San Diego, USA, June 9-13, 2007, pp.347-357.

[14] Huang R, Garg A, Huang M. Software hardware cooperative memory disambiguation. In *Proc. the 12th HPCA*, Austin, USA, Feb. 11-15, 2006, pp.244-253.

[15] Cain H W, Lipasti M H. Memory ordering: A value-based approach. In *Proc. the 31st ISCA*, München, Germany, June 19-23, 2004, pp.90-101.

[16] Roth A. Store vulnerability window: Re-execution filtering for enhanced load optimization. In *Proc. the 32nd ISCA*, Madison, USA, June 4-8, 2005, pp.458-468.

[17] Chrysos G Z, Emer J S. Memory dependence prediction using store sets. In *Proc. the 25th ISCA*, Barcelona, Spain, June 27-July 1, 1998, pp.142-153.

[18] Moshovos A, Breach S E, Vijaykumar T N, Sohi G S. Dynamic speculation and synchronization of data dependences. In *Proc. the 24th ISCA*, Denver, USA, June 2-4, 1997, pp.181-193.

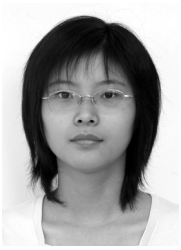
[19] Hilton A, Roth A. Decoupled store completion/silent deterministic replay: Enabling scalable data memory for CPR/CFP processors. In *Proc. the 36th ISCA*, Austin, USA, June 20-24, 2009, pp.245-254.

[20] Hilton A, Roth A. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proc. the 16th HPCA*, Bangalore, India, Jan. 9-14, 2010, pp.1-12.

[21] Mutlu O, Stark J, Wilkerson C, Patt Y N. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. the 9th HPCA*, Anaheim, USA, Feb. 8-12, 2003, pp.129-140.

[22] Akkary H, Rajwar R, Srinivasan S T. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. the 36th MICRO*, San Diego, USA, Dec. 3-5, 2003, pp.423-434.

Zhen-Hao Zhang is currently a computer science Ph.D. candidate in computer science of Peking University. His current research interests include high performance microprocessor architecture, energy efficient load/store execution and memory system optimization.



Xiao-Yin Wang received her Ph.D. degree in computer science from Peking University in 2010. She is now a postdoctoral researcher in Peking University. Her research interests include microprocessor architecture, low-power cache design and memory system optimization.

Dong Tong received his Ph.D. degree in computer science from Harbin Institute of Technology in 1999. He is now a professor in the School of Electronics Engineering and Computer Science, Peking University. His research interests include processor architecture, reconfigurable computing, interconnection network and System-on-Chip design. He is a member of CCF and ACM.

Jiang-Fang Yi received her Ph.D. degree in computer science from Peking University in 2007. She is now an associate professor in the School of Electronics Engineering and Computer Science, Peking University. Her research interests include HW/SW co-design, System-on-Chip verification and test vector automatic generation.

Jun-Lin Lu received his Ph.D. degree in computer science from Peking University in 2009. He is now an associate professor in the School of Electronics Engineering and Computer Science, Peking University. His research interests include HW/SW co-design and the communication architecture of System-on-Chip.

Ke-Yi Wang is the professor and doctoral tutor in the School of Electronics Engineering and Computer Science, Peking University. He presided over and participated several national scientific and technological projects. His research interests include high performance microprocessor architecture design.