Store Buffer Design for Multibanked Data Caches

Enrique Torres, *Member*, *IEEE*, Pablo Ibáñez, *Member*, *IEEE*, Víctor Viñals-Yúfera, *Member*, *IEEE*, and José M. Llabería

Abstract—This paper focuses on how to design a Store Buffer (STB) well suited to first-level multibanked data caches. The goal is to forward data from in-flight stores into dependent loads within the latency of a cache bank. Taking into account the store lifetime in the processor pipeline and the data forwarding behavior, we propose a particular two-level STB design in which forwarding is done speculatively from a distributed first-level STB made of extremely small banks, whereas a centralized, second-level STB enforces correct store-load ordering. Besides, the two-level STB admits two simplifications that leave performance almost unchanged. Regarding the second-level STB, we suggest to remove its data forwarding capability, while for the first-level STB, it is possible to: 1) remove the instruction age checking and 2) compare only the less significant address bits. Experimentation covers both integer and floating point codes executing in dynamically scheduled processors. Following our guidelines and running SPEC-2K over an 8-way processor, a two-level STB with four 8-entry banks in the first level performs similar to an ideal, single-level STB with 128-entry banks working at the first-level cache latency. Also, we show that the proposed two-level design is suitable for a memory-latency-tolerant processor.

Index Terms—Cache memories, computer architecture, memory architecture, pipeline processing.

1 Introduction

Out-of-order processors with precise exceptions require enforcement of the memory dependences and writing of the data cache in program order. However, in order to speed up the program execution, processors also add a functionality called store-to-load data forwarding (data forwarding, for short). Data forwarding allows an in-flight load reading a given address to take its data from the previous and nearest, if any, noncommitted store writing to the same address.

In order to accomplish the former functions, namely, memory dependence enforcement, in-order data cache writing, and data forwarding, a structure usually called Store Buffer (STB) is employed. A conventional STB keeps store instructions in program order until their in-order commitment. Usually, an STB is designed as a circular buffer whose entries are allocated to stores at Dispatch and deallocated at Commit (Fig. 1). Instruction dispatch stalls if a store instruction finds all the STB entries already allocated. When a store instruction executes, it writes its address and data in the allocated entry. As entries are allocated in order, the relative age of two store instructions can be determined by the physical location they occupy in the circular buffer (age ordering).

- *E. Torres, P. Ibáñez, and V. Viñals-Yúfera are with the Departamento de Informatica e Ingenieria de Sistemas and the Aragon Institute of Engineering Research (I3A), Universidad de Zaragoza, Maria de Luna, 1, Edificio Ada Byron, 50018 Zaragoza, Spain. E-mail: {enrique.torres, imarin, victor}@unizar.es.*
- *J.M. Llabería is with the Departamento Arquitectura de Computadores, Campus Nord, módulo D6, Universidad Politécnica de Cataluña, Jordi Girona 1-3, 08034 Barcelona, Spain. E-mail: llaberia@ac.upc.edu.*

Manuscript received 9 June 2006; revised 21 Feb. 2008; accepted 10 Feb. 2009; published online 24 Mar. 2009.
Recommended for acceptance by N. Bagherzadeh.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0226-0606.
Digital Object Identifier no. 10.1109/TC.2009.57.

The data forwarding logic of the STB consists of a *store address CAM*, an *age-based selection logic*, and a *data RAM*. A load instruction proceeds as follows:

1. When a load instruction is dispatched it is tagged with an identifier of the entry allocated to the last store instruction in the STB.
2. After address computation, the load instruction concurrently accesses the data cache and the STB.
3. Inside the STB, the CAM structure associatively searches for stores matching the load address. Then, the age-based selection logic discards all the stores younger than the load (using the identifier and a mask logic) and picks up the youngest store among the remaining ones (using a priority encoder). Age ordering of STB entries simplifies the design of the age selection logic.
4. Finally, the data associated with the selected store (if any) is read from the data RAM and forwarded into the load. If a load address does not properly match within the STB, the load data will come from the data cache.

The STB is a critical component of out-of-order processors, because its data forwarding logic is in the critical path that sets the *load-to-use latency*. If the STB latency is longer than the L1 cache latency, the scheduling of load-dependent instructions becomes complicated and performance is severely affected. Moreover, the whole circuitry that identifies and forwards data is complex and incurs long delays as STB size increases. If the trend toward faster processor clocks, wider pipelines, and an increasing number of in-flight instructions goes on, the latency problem posed by the STB may worsen [1].

On the other hand, multibanked L1 data caches are considered as good candidates to support wide pipelines in superscalar processors [2], [10], [11], [12], [16], [30], [31].

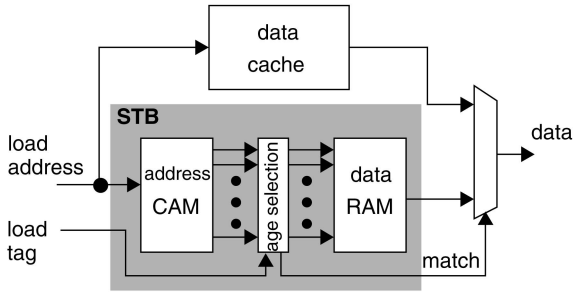

Fig. 1. Conventional STB and simplified load data path.

Multibanking provides low latency and high bandwidth by physically splitting storage in independent, single-ported cache banks. But the latency of a big multiported STB is longer than the latency of a small single-ported cache bank [19]. Therefore, a multibanked L1 data cache organization calls for a distributed STB, and distribution can be achieved in a straightforward way by placing independent, single-ported STB banks next to each L1 cache bank.

Zyuban and Kogge use a single-level distributed STB [31]. In their work, when a store is dispatched to the selected Issue Queue, an STB entry is allocated to the store in all STB banks. Because store-load ordering enforcement and data forwarding are performed by STB banks, stores cannot be removed from the STB banks until they Commit, and Dispatch is stalled whenever STB becomes full. As we will show later, a single-level distributed STB, if undersized, can seriously limit performance due to Dispatch stalls. But, on the other hand, increasing the STB size may increase its latency [19].

Our goal is to speculatively forward data from noncommitted stores to loads at the same latency of a cache bank. By looking into the store lifetime and the data forwarding behavior, we propose to decouple the different tasks performed by a monolithic STB using a two-level STB [29]. Forwarding is speculatively done from a distributed first-level STB (STB1) made up of very small banks optimized for low latency. A few cycles later, a second-level STB (STB2) checks the speculative forwarding made by the STB1 and, if required, enforces correct memory access ordering by taking the proper recovery action.

The STB2 entries are allocated in program order when stores are dispatched and deallocated when they Commit, as in a conventional STB. However, as we observe that stores forward data in a narrow window of time after store execution, we propose to delay the allocation of STB1 entries to stores until they execute, and allow STB1 entry deallocation to proceed before stores Commit. If an STB1 bank is full, new entries are retrieved in FIFO order. This STB1 allocation/deallocation policy prevents stalling Dispatch when STB1 banks are full and enables reducing the STB1 size. Moreover, as we allocate STB1 entries at the Execution stage, every store will only occupy a single STB1 entry in a single STB1 bank (just the right one).

The proposed two-level organization allows us to trade complexity for a small performance decrease. Namely, we will show that selecting a forwarding store from the STB1 without considering ages, and comparing only a subset of the load/store addresses degrades performance marginally. We will also show that the STB2 can be completely freed from the task of forwarding data, and that the STB2 latency is not a limiting factor of processor performance.

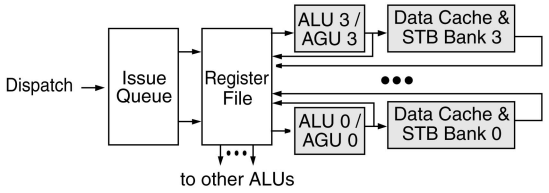

Fig. 2. Simplified data path example with a four-banked L1 cache and single-level distributed STB.

We evaluate the two-level STB design in a *sliced memory pipeline* framework, but the concept is applicable to other multibanked L1 data cache organizations (for instance, those with a second queue which schedules memory accesses contending for banks [11], [12], [30]). In a sliced memory pipeline, the memory pipeline splits into simple and independent slices where every cache bank is coupled with an address generation unit, and the target cache bank is predicted before reaching the Issue Queue stage [16], [30].

This paper is structured as follows: Section 2 outlines the processor-cache model being used and motivates the work. Section 3 provides a set of design guidelines for a two-level STB system. Section 4 details the simulation environment. Section 5 analyzes performance of integer benchmarks for the basic two-level STB system working with line-interleaved multibanked L1 caches. Section 6 introduces several design alternatives aimed at improving the performance and reducing the complexity of the proposed two-level STB. Section 7 is a performance summary showing figures for integer and floating point benchmarks. Section 8 extends the analysis to memory-latency-tolerant processors. Section 9 discusses related work and Section 10 concludes the paper.

2 Motivation

The STB is in the critical path of the load execution. The STB latency increases with the number of STB ports and entries [1], [19]. If the STB latency is longer than the L1 cache latency, the scheduling of load-dependent instructions becomes complicated and performance is severely affected. To show that, in this section, we analyze the performance of a processor with a single-level distributed STB as a function of the size and latency of each STB bank. We also characterize the utilization of a distributed STB showing the average lifetime of a committed store.

Next, we begin outlining the processor and multibanked L1 cache models.

2.1 Processor and Multibanked L1 Cache Models

We assume a first-level data cache made of several address-interleaved L1 cache banks operating as independent, pipelined, memory slices [30]. A memory slice has an *Address Generation Unit* (AGU), a *cache bank*, and an *STB bank* (Fig. 2). We also assume an *Issue Queue* (IQ) with a fixed number of scheduler ports shared among integer ALUs and memory slices. Load and store instructions are dispatched to the IQ carrying a prediction on their target cache bank which was generated by a Bank Predictor accessed in the front-end stages of the processor pipeline. The IQ uses the prediction in order to issue memory instructions to the predicted slice. Whenever a bank misprediction arises, the mispredicted load or store instruction is reissued from the IQ to the right bank.

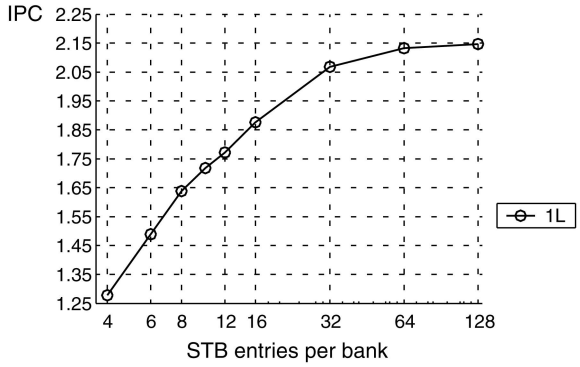

Fig. 3. Single-level distributed STB (1L) with four banks. IPC harmonic mean versus number of entries per STB bank. X-axis in logarithmic scale.

2.2 Single-Level Distributed STB

An STB can be distributed in a straight way by placing independent, single-ported STB banks close to each L1 cache bank [31], an approach we call *single-level distributed STB* (Fig. 2). Each STB bank is only responsible for the address subset mapped to its cache bank companion. Forwarding data to loads and enforcing store-load ordering is thus locally performed at each STB bank. In this approach, an entry is allocated in *all* STB banks when a store is dispatched, because the store address (and also the destination bank) is still unknown. Later on, when the store executes in the right slice, a *single* STB bank entry is filled. Eventually, when the store Commits, the entries are simultaneously deallocated in *all* STB banks. Thus, the STB bank size determines the maximum number of in-flight stores.

A single-level distributed STB, if undersized, can seriously limit performance due to Dispatch stalls. Besides, increasing the STB size may increase its latency, which, in turn, also limits performance. Next, we study how single-level distributed STB performance depends on the STB bank size and latency. After that, we analyze how STB entries are used during store lifetime.

For these experiments, we model an 8-way dynamically scheduled processor with four memory slices and a 256-entry Reorder Buffer having in-flight up to 128 loads and 128 stores (a number large enough not to stall Dispatch). More details of cache and processor parameters, memory pipeline, benchmarks, and simulation methodology are given in Section 4.

2.3 Processor Performance Versus Number of STB Entries

Fig. 3 shows the variation of the average IPC of SPECint-2K benchmarks in the simulated processor with a single-level distributed STB (1L) when the number of entries of each STB bank goes from 4 to 128 (notice that the total STB size is four times that number). The computed IPC assumes that the STB and the L1 cache banks have the same latency, no matter what the simulated STB size.

As can be seen, if undersized, a single-level distributed STB limits performance severely: 4-entry STB banks show a 40.5 percent IPC drop relative to 128-entry STB banks. Below 32 entries, the IPC slope is very steep (−9.3 percent IPC from 32 to 16 entries). A 32-entry STB sets the IPC 3.7 percent below the upper bound. From 64 entries onward, the STB does not limit processor performance.

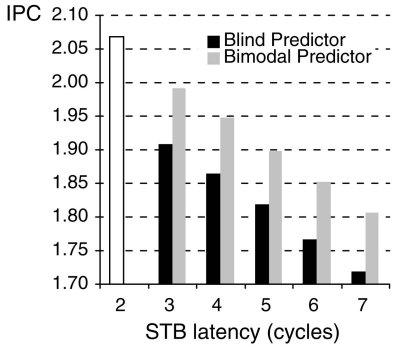

Fig. 4. Single-level distributed STB performance for four 32-entry STB banks. STB bank access latency increases from two cycles (L1 cache latency, hollow bar) to seven cycles (L1 cache latency plus five cycles). Black and gray bars show the IPC harmonic mean with a blind predictor, and a 4K-entry bimodal predictor of loads to be forwarded by STB, respectively.

2.4 Processor Performance Versus STB Latency

The STB logic that checks dependences and forwards data is complex and incurs long delays as the STB size increases [1], [19]. Besides, if the STB latency is higher than the L1 cache latency, there appear structural hazards and the scheduling of load-dependent instructions gets complicated. To take the two latencies into consideration, every load is tagged on Dispatch with a predicted latency (either L1 cache latency or STB latency). Afterward, resource allocation[1] and speculative wake-up of dependent instructions take place according to the predicted latency.

It is easy to predict what loads are going to be forwarded by the STB and tag them with the STB latency. In this section, we will use a simple, tagless, 4K-entry bimodal predictor, since the obtained IPC is similar when using an oracle-like predictor (under 1 percent IPC improvement across all the tested configurations).

In order to determine the negative effect on performance of having an STB slower than the L1 cache, we simulate a system with four 32-entry STB banks varying their latency from two to seven cycles (two cycles equal the L1 cache latency, hollow bar in Fig. 4). We simulate two models of latency prediction. The first model blindly predicts L1 cache latency for all loads (black bars). The second model (gray bars) uses the bimodal predictor. As we can see, if loads take just one extra cycle to reach, access, and get data from STB banks, the IPC loss resulting from blind prediction is almost 8 percent. Even if we add a bimodal predictor, the IPC degrades 3 percent per additional cycle.

2.5 STB Entry Utilization

Looking at STB utilization is a key factor to overcome the size-latency tradeoff and increase STB performance. Fig. 5 indirectly shows STB utilization by plotting the *average lifetime* of a committed store.

On average, each store spends 46.4 cycles in STB: 17.7 cycles from Dispatch to Execution and 28.7 cycles from Execution to Commit. When stores execute, they fill a single STB bank with a <data, address> pair, but only 22.4 percent of stores will forward data to some load. Stores tend to forward data soon after they execute: on average, the *last use* of an STB entry occurs 8.5 cycles after the store execution

1. Resource examples are bypass network and write ports to register file; their management adds complexity to the IQ scheduler.

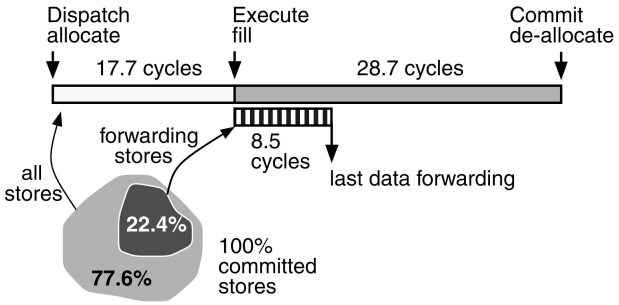

Fig. 5. Store lifetime in a single-level distributed STB with four 128-entry STB banks (arithmetic means).

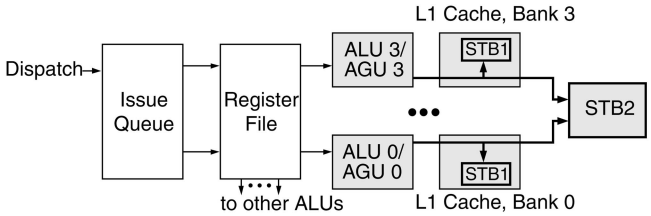

Fig. 6. Simplified data path example with a two-level distributed STB and a four-banked L1 cache.

(75 percent of data forwarding occurs within the next seven cycles, and 90 percent within the next 14 cycles). Therefore, from a data forwarding standpoint, we notice that: 1) only a few STB entries forward data to loads, 2) the data forwarding is performed in a narrow window of time after store execution, and 3) each STB entry is allocated too early (at dispatch time) and deallocated too late (at Commit time).

Summarizing, we can conclude that big (and possibly slow) STB banks are required in order not to stall Dispatch, but STB latencies larger than cache access times hurt performance. Another important fact is: from a forwarding perspective, STB entries are poorly managed.

3 Basic Two-Level STB Design Guidelines

Our goal is to keep STB bank latency equal to or under L1 cache bank latency. However, allocating and deallocating STB entries to stores at Dispatch and Commit, respectively, requires large (and slow) STB banks in order not to stall Dispatch frequently.

To overcome this limitation, we propose a two-level STB design with allocation and filling policies specific to each level (Fig. 6). A two-level STB decouples the different tasks performed by a single-level STB as follows: the STB1 only performs speculative data forwarding, while the STB2 checks store-load ordering, performs data forwarding at STB2 speed (whenever STB1 fails to do it), and updates caches in program order. Thus, a load instruction can obtain the data from either an L1 cache bank, an STB1 bank, or the STB2. Anyway, we will blindly predict L1 cache latency for all load instructions.

3.1 First-Level STB Description

The STB1 has to be as simple and small as possible to match the L1 cache bank latency. To that end, we distribute the STB1 in single-ported banks and reduce their size by limiting the number of cycles an STB1 entry remains allocated to a particular store. From the forwarding behavior exhibited by stores (Fig. 5), we can limit the time a store stays in the STB1, probably without performance losses, if we enforce the following two guidelines:

1. Delay allocation of STB1 entries until stores reach Execution stage. Thus, before Execution, no store wastes STB1 entries. Delaying allocation could shorten store lifetime in STB1 by around one-third as pointed out in Fig. 5. Allocation is now done after bank check, and thus, a *single* entry is allocated to each store in only one single STB1 bank.

2. Deallocate STB1 entries before stores Commit. We can deallocate entries early because most data forwarding happens in a short period of time (see Fig. 5 again). This fact suggests an FIFO STB1 replacement policy, so that the Dispatch stage never stalls even though the STB1 gets full. As entries are allocated in issue order, an STB1 bank keeps only the stores most recently issued to that bank.

As we will see in Section 5, following these guidelines allows us to design a two-level STB with very small STB1 banks (eight entries). However, the entry allocation in the STB1 is not based on the store instruction age but on the issue order. That is, the entry index in the STB1 does not reflect the relative instruction age with respect to other entries. Therefore, the age-based selection logic of our STB1 requires a CAM structure that explicitly keeps store instruction ages (age-CAM) instead of the mask logic of a conventional STB. The age-CAM compares the load age with the age of each STB entry and activates those entries older than the load. Next, the youngest of the activated entries is selected.

Nonetheless, in Section 6.3, we will see that the two-level organization allows us to trade STB1 complexity for a small performance decrease. Namely, we will show that selecting a forwarding store from STB1 can be done without considering ages (*removing age-based selection*). The STB1 selects the last-inserted store with a matching address. So, the store that forwards data is selected by using only a priority encoder, eliminating the CAM structure that explicitly keeps store age in our first STB1 design. Note also that the mask logic of a conventional STB is not required. Moreover, complexity can be further reduced by comparing only a subset of the load/store data addresses (*partial address comparison*). Both simplifications, removing age-based selection and partial address comparison, help to match the latencies of STB1 and L1 cache even more, while also reducing area and energy consumption of the STB1.

3.2 Second-Level STB Description

On the other hand, the STB2 keeps all in-flight stores, but it is placed outside the load-use critical path. At dispatch time, as in a conventional STB, entries are allocated to stores in STB2, where they remain until they Commit. So, Dispatch stalls when running out of STB2 entries, irrespective of the STB1 size. Notice that the maximum number of in-flight stores is the number of STB2 entries.

As STB1 banks do not keep all in-flight stores, any data supplied to a load by the L1cache/STB1 ensemble is speculative and must be verified in the STB2. To that end, the STB2 acts as a conventional STB, selecting the forwarding store if it exists. Besides, the identifier of this store is compared with the identifier of the forwarding store in STB1. This operation, which we call *data forwarding check*, detects a *forwarding mis-speculation* whenever a load finds a matching store in the STB2 and the load is not forwarded from the STB1, or it is forwarded by a wrong store. In these cases, STB2 starts a recovery action which consists in the

TABLE 1
Microarchitecture Parameters

fetch and decode width	8
branch predictor: hybrid (bimodal, gshare)	16 bits
reorder buffer entries	256
in-flight loads	128
in-flight stores	128
integer/FP IQ entries	64 / 32
integer/FP units	8 / 4

L1 I-cache	64KB, 4-way
L1 D-cache	2 cycles
banks	4
ports/bank	1 r/w
bank size	8 KB, 4-way
line size	32 B
L1 MHSR	16 entries
Store-Set Pred.	4K-entry SSI Table 128-entry LFS Table

L2 Unified Cache	256 kB, 8-way
	16 banks, 7 cycles
line size	128 B
L2 MHSR	8 entries
L3 Unified Cache	4MB, 16-way 19 cycles
line size	128 B
Bus L3-main mem.	8 cycles/chunk
main mem. lat.	200 cycles

nonselective redispatch to the IQ of all the instructions younger than the load (see Section 4.3).

Alternatively, the load-dependent instructions could be re-issued selectively from the IQ as in any other latency misprediction. However, the extra IQ occupancy of the selective method and the small number of data forwarding misspeculations make the nonselective mechanism a better option [29].

In our simulations, as an STB2, we model a conventional multiported STB. Nevertheless, in Section 6.2, we will show that STB2 can be completely freed from the task of data forwarding. Additionally, some published STB optimizations targeted at reducing area (number of ports), energy consumption (banking, filtering accesses), or both [3], [17], [21] can be added to our STB2 profitably.

The default STB2 forwarding check latency has eight cycles: AGU (one cycle), TLB (one cycle), STB2 (four latency cycles plus two transport cycles). Anyway, as we will see in Section 5, performance does not depend on the STB2 latency.

4 Simulation Environment

We have modified SimpleScalar 3.0c [4] in order to model a Reorder Buffer and separate integer and floating point IQs. Latency prediction (cache bank, L1 cache hit/miss, etc.), speculative memory instruction disambiguation, speculative instruction issue, and recovery have been carefully modeled. The memory hierarchy has three cache levels and a set of interconnection buses whose contention has also been modeled. We assume an out-of-order 8-issue processor with eight stages from Fetch to IQ and one stage between IQ and Execution. Other processor and memory parameters are listed in Table 1.

Next, Section 4.1 presents the memory data path and the load/store pipeline timing. Section 4.2 describes how data is distributed across L1 banks and how memory instructions are routed to an L1 bank. Section 4.3 explains how the processor recovers from mispredictions, and finally, Section 4.4 shows the benchmarks used in our simulations.

4.1 Memory Data Path

The L1 data cache is sliced into four independent paths (Fig. 7). Each path has an address generation unit (AGU), a cache bank, and an STB1 bank.

The cache bank has only one read/write port shared between loads, committed stores, and refills from the L2 cache. Cache banks are tied to the L2 cache through a single refill bus of 32 bytes, which also supports forwarding from STB2 (later we remove this capability). From each STB1/cache ensemble, there is a single data path to the Bypass Network shared among supplies from the STB1 bank, the L1 cache bank, the L2 refill, and the STB2 data forwarding.

Requests to the L2 cache are managed by an L2 Queue (L2Q) after accessing L2 tags as Intel Itanium II does [15]. The L2Q can send up to four nonconflicting requests per cycle to the 16 interleaved 2-cycle cache banks (16B interleaving). A refill to the L2 cache takes eight banks. The model can stand 16 primary L1 misses and eight L2 misses.

Enforcing load/store ordering. Stores are not issued until both data and address registers become available. Memory dependence prediction is used to execute load instructions and their dependent instructions before knowing the addresses accessed by older store instructions. We use the *Store-Sets* disambiguation predictor as described in [5]. The predictor is accessed in the front-end stages of the processor pipeline and the predicted ordering is managed by the IQ, which will delay the issue of a load until an older store has been issued if a dependency between them has been predicted. Memory ordering misspeculations are discovered when stores execute, possibly many cycles after loads and their dependent instructions have left the IQ.

Load instructions. After the Address Generation (in the AGU), a memory access takes one cycle to access the L1 cache bank and the STB1 in parallel, plus an extra cycle to reach the bypass network (*line 1* in Fig. 8, *cycles 2 and 3*). The forwarding speculation check performed by STB2 is known several cycles past the L1 cache latency (*line 2*, *cycle 8*), but

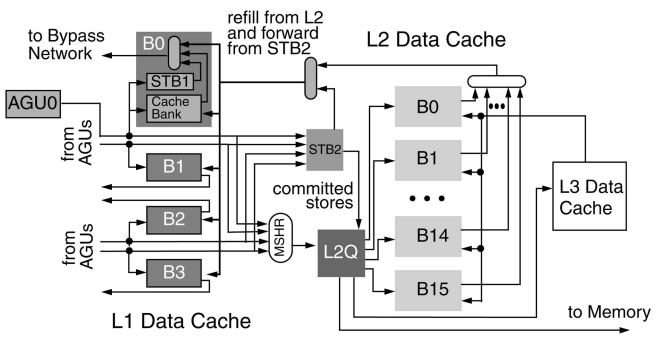

Fig. 7. Simplified memory hierarchy and data path showing a two-level distributed STB. For clarity, only the connection detail in Bank0 is shown.

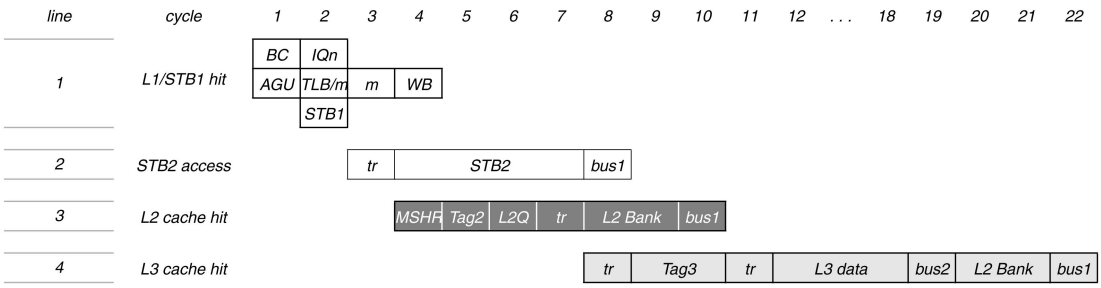

Fig. 8. Memory pipeline timing after the register read stage. *BC*, *IQn*, and *tr* stand for Bank Check, IQ notification, and transport cycles, respectively. *busX* and *TagX* stand for bus use and tag access, respectively. *TLB* and m stand for Translation Look-aside Buffer access and L1 cache access (two cycles, 2 and 3), respectively. The cycles before L2 hit and L3 hit are used to determine L1 miss and L2 miss, respectively.

only affects load execution if an STB1 forwarding misspeculation arises. A load experiencing an L1 cache miss is reissued from the IQ in time to catch the data coming from the L2 cache refill (*line 3*, *cycle 7*). On an L2 cache miss, a request is sent to the L3 cache (*line 4*), which, in case of hit, delivers data to the bypass network in *cycle 22*.

Store instructions. The L1 cache is write-through and no-write-allocate. Store instructions are committed to L2, and whenever they hit the L1 cache (filtered by the L2 cache directory), they are placed in an 8-entry coalescing write buffer local to each cache bank (not shown in Fig. 7). Write buffers update L1 cache banks in unused cycles.

4.2 L1 Data Distribution (Cache and STB1 Banks)

Banks are line-interleaved. Because memory instruction routing is made from the IQ prior to computing addresses, a bank prediction is required by the IQ scheduler. Bank prediction is done in the front-end stages of the processor pipeline and bank check is made concurrently with address computation by evaluating the expression $\mathrm{A} + \mathrm{B} = \mathrm{K}$ without carry propagation [6]. The IQ is notified during the cycle following bank check (*line 1*, IQn, *cycle 2* in Fig. 8). A correct bank prediction does not need further information, but a misprediction comes along with the correct bank number. So, the IQ will be able to route the mispredicted memory instruction to the correct bank.

As a bank predictor, we have chosen a global predictor because it is able to yield several predictions per cycle easily [22]. We have also chosen to predict each address bit separately (two bits for four banks) [30]. As a bit predictor, we have used an *enhanced skewed binary predictor*, originally proposed by Michaud et al. for branch prediction [14]. Every bit predictor has 8K entries for the three required tables and a history length of 13, totalling 9 Kbyte per predictor. Table 2 shows the accuracy of four-bank predictors. Each individual execution of a memory instruction has been classified according to the bank prediction outcome (right or wrong). Store instructions roughly have half the bank mispredictions experienced by load instructions. See [28] for a detailed comparison of bank predictors.

4.3 Recovery from Misspeculations

L1 cache latency is blindly predicted for all loads, and thus, dependent instructions are speculatively woken-up after the L1 cache latency elapses. Therefore, there are three sources of load latency misprediction: *bank misprediction*, *L1 cache miss*, and *store-load forwarding misspeculation*. Additionally, the processor has two more sources of misspeculation: *memory ordering misspeculation* and *branch misprediction*.

The modeled processor implements three recovery mechanisms: namely, *recovery from Fetch* (branch misprediction), *recovery from the Renamed Instruction Buffer* (Memory ordering and store-load forwarding misspeculations), and *recovery from the IQ* (bank misprediction and L1 cache miss).

Recovery from Fetch. All the instructions younger than the branch instruction are flushed out of the pipeline and the instruction fetch is redirected to the correct target instruction. The minimum latency is 13 cycles.

Recovery from the IQ. In order to allow recovering from the IQ, all speculatively issued instructions that depend on a load are kept in the IQ until *all* load predictions are verified (first, bank check in AGU; next, tag check in the cache bank). Once a latency misprediction has been detected, the already issued instructions dependent on the mispredicted load are reissued at the right moment, either after the load is rerouted to the correct bank or after the cache miss is serviced. Notice that recovery is selective because reissuing only affects dependent instructions.

Recovery from the RIB. Usually, to recover from a memory ordering misspeculation, the load and all younger instructions (dependent or not) are flushed out of the pipeline and subsequently refetched from the instruction cache. However, the refetched instructions are just the same ones that have been flushed. So, to reduce the misspeculation penalty, recovery can be supported by a structure that keeps already renamed instructions. We call this structure *Renamed Instruction Buffer*. As recovery is not done at the Fetch stage, we do not have to checkpoint the Register Map Table[2] on every load instruction as is done with branch instructions.

The RIB is an FIFO buffer located between the Register Rename and the Dispatch stages, having the ability to keep all the renamed in-flight instructions (Fig. 9). So, the RIB is continuously being filled, in program order, with instructions already renamed and tagged with all the predictions computed in the early stages. To simplify the RIB design, we have chosen not to update RIB entries when a memory instruction is executed and a bank misprediction is discovered, even though in not doing so a further recovery from the RIB would re-experience the same bank misprediction.

Recovery consists in redispatching the offending load and all subsequent instructions to the IQ, taking them sequentially from the RIB. So, the RIB has only one write and one read port. A similar buffer was suggested by Lebeck et al. in [13] to tolerate long-latency cache misses. However, that proposal is more complex because it makes a

2. Table used to rename logical registers to physical registers.

TABLE 2
Bank Predictor Accuracy for Four Banks (Percent)

	load instr.	store instr.	all memory instr.
right	89.32	95.38	91.46
wrong	10.68	4.62	8.54

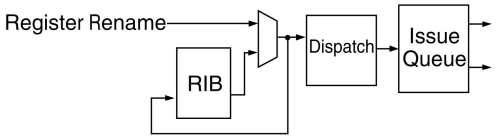

Fig. 9. Renamed Instruction Buffer (RIB) and its location in the processor.

selective recovery and updates the buffer information when instructions Execute.

4.4 Workload

We use SPECint2K compiled to Alpha ISA, simulating a contiguous run of 100 million instructions from SimPoints [24] after a warming-up of 200 million instructions. We also use SPECfp2K for a limited set of experiments. Tables 3 and 4 show input data sets for SPECint2K and SPECfp2K, respectively.

All figures (except otherwise noted) show the IPC harmonic mean (y-axis) across different STB bank sizes (x-axis in logarithmic scale). Other measures show arithmetic mean values. We have computed the reported IPCs by excluding MCF because it is strongly memory-bound. Nevertheless, a summary of individual program results, including MCF, is shown in Section 7.

5 Basic Two-Level STB Performance

In this section, we present the performance results obtained by our basic two-level STB proposal explained in Section 3. We also compare the basic two-level STB with the single-level STB, and with the two-level system previously proposed by Akkary et al. [1]. Finally, we analyze the sensitivity of the basic two-level STB system performance to the STB2 latency.

Akkary et al. proposed a two-level STB in the context of a processor without multibanked cache [1]. In this organization, both the STB1 and the STB2 are monolithics (not multibanked). STB1 entries are allocated to stores at Dispatch time in FIFO order. When the STB1 becomes full, the oldest store is moved to STB2. Both STBs can forward data but at different latencies. In order to reduce the number of STB2 accesses, they use a *Membership Test Buffer* (MTB), which detects the forwarding misspeculations. We model a multibanked version of this proposal (2L_**MTB**). As the MTB has the latency of an L1 cache access, on a forwarding misspeculation, 2L_MTB uses the same recovery mechanism as that on a latency misspeculation. In our simulations, we use an oracle predictor instead of the MTB, and thus, the 2L_MTB results are optimistic.

Fig. 10a shows variations of the IPC as the number of STB1 bank entries goes from 4 to 128 for four STB systems: the single-level STB system presented in Section 2.2 (**1L**), the basic two-level STB presented in Section 3 (**2L**), the basic two-level STB when reducing the STB2 latency from eight cycles to just the cache latency plus one cycle (2L_**1c**), and the 2L_MTB system. The computed IPC assumes that the first-level STB access latency is equal to the L1 cache latency, no matter the STB size we simulate.

TABLE 3
Simulated SPECint2K Benchmarks and Input Data Set

Bench.	Data set	Bench.	Data set	Bench.	Data set
bzip2	program-ref	gzip	program-ref	twolf	ref
crafty	ref	mcf	ref	vortex	one-ref
eon	rushmeier-ref	parser	ref	vpr	route-ref
gcc	166-ref	perl	diffmail-ref		

TABLE 4
Simulated SPECfp2K Benchmarks and Input Data Set

Bench.	Data set	Bench.	Data set	Bench.	Data set
ammp	ref	facerec	ref	mgrid	ref
applu	ref	fma3d	ref	sixtrack	ref
apsi	ref	galgel	ref	swim	ref
art	110-ref	lucas	ref	wupwise	ref
equake	ref	mesa	ref		

For all STB bank sizes, the proposed two-level system outperforms the single-level system and the 2L_MTB system. This is so because the proposed two-level system makes better use of STB1 bank entries, allocating them only when data is available and deallocating them as new stores enter the STB1. Therefore, the performance gap increases as the number of STB1 entries decreases.

Fig. 10b presents the STB1 load coverage for the basic two-level STB (2L) and 2L_MTB. Namely, 100 percent load coverage means that any load needing data forwarding from an older in-flight store is fed from the STB1. As we can see, in our proposed 2L system, even for very small STB1 banks, the STB1 load coverage is very high. As an example, only less than 1 percent of the loads requiring forwarding (0.13 percent of the total loads) are not forwarded from an 8-entry STB1. Note also that below 32 entries, the coverage of 2LMTB is much lower than that of the proposed 2L system.

In order to get an insight about how the STB2 forwarding check latency affects performance, we simulate the basic two-level STB system again, but this time reducing the STB2 forwarding check latency from the cache latency plus five cycles to the cache latency plus just one cycle (2L_1c). As the number of forwarding misspeculations is very low, the proposed two-level system has an IPC which is almost independent of the STB2 latency, see 2L versus 2L_1c in Fig. 10a.

Next, in Section 6, we improve the basic two-level distributed STB design in several ways.

6 Design Enhancements

Both the store behavior in the pipeline and the high load coverage achieved by the basic two-level distributed STB design suggest several enhancements that we explore in this section. The first one increases performance and the last two reduce complexity.

First, as the contention in the issue ports to the memory slices is an important drawback in sliced memory pipelines, we propose reducing the IQ contention by identifying stores that do not forward data (*nonforwarding stores*), sending

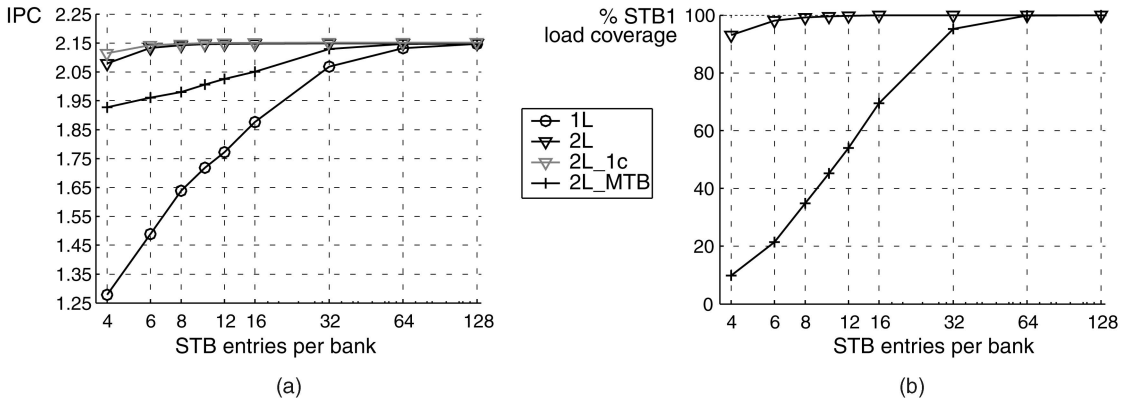

Fig. 10. (a) IPC for the single-Level distributed STB (1L), the basic two-Level STB (2L and 2L_1c), and 2L_MTB. (b) STB1 load coverage for the basic two-Level STB system and the 2L_MTB. X-axis in logarithmic scale.

them by any free issue port to memory, thus bypassing STB1 and going directly to STB2. Second, we simplify the STB2 design by removing its data forwarding capability. Finally, we simplify the STB1 data forwarding logic by eliminating age checking in the selection logic and reducing the number of address bits used to compare.

6.1 Reducing Contention for Issue Ports to Memory

Contention in an L1 multibanked cache appears when a burst of ready memory instructions is targeted to a single bank. In this situation, all memory instructions contend for a single issue port to memory and performance may suffer. We have previously seen that the greatest part of stores do not forward data. As this behavior is highly predictable, we could detect such stores and divert them through any free issue port to memory, thus bypassing STB1 and going directly to STB2. To that end, we propose using a predictor that we call *Nonforwarding Store Predictor* (NFS predictor) that is accessed in the front-end stages of the processor pipeline. Stores classified as nonforwarders can be issued by any free issue port to memory, thus increasing effective issue bandwidth. Notice that the NFS predictor is further acting as a store insertion filter, because the STB1 holds only stores classified as forwarders, which can increase the STB1 effective capacity.

A trade-off exists in the predictor design. By reducing the number of stores classified as forwarders, we reduce contention for issue ports. However, forwarding misspeculations may increase because more forwarder stores would be classified as nonforwarders. We design the predictor in order to reduce stores wrongly classified as nonforwarders. As an NFS predictor, we use a simple bimodal predictor having 4K counters of 3 bits each indexed by an instruction address.

Table 5 shows some predictor statistics. Sixty-four percent of stores are classified as nonforwarders, reducing contention for issue ports to memory and reducing STB1 pressure. However, 0.47 percent of stores which forward data before Committing are wrongly classified as nonforwarders, causing a forwarding misspeculation.

TABLE 5
Nonforwarding Store Predictor (Percent)

	predicts forward	predicts not forward
right	25.45	64.05
wrong	10.03	0.47

Fig. 11a shows the IPC for basic two-level SYB systems with and without an NFS predictor. The system with an NFS predictor (2L_**NFSP**) always achieves a better IPC than a system without it (2L).

In order to separate the contributions of contention reduction and STB1 store filtering, we simulate a system with an NFS predictor used only to filter store insertion in the STB1 and not to reduce contention for issue ports to memory (2L_**nfsp**).

Making use of free issue ports to memory consistently improves performance across the whole range of STB1 sizes (2L_NFSP versus 2L_nfsp). However, by only filtering store insertion, performance increases for small STB1 of four to six entries, but it decreases above eight entries (2L_nfsp vs. 2L).

To explain the performance decrease introduced by filtering store insertion, in Fig. 11b, we have plotted the load coverage of a system enhanced with an NFS predictor (2L_NFSP) and a system without it (2L). We see that an NFS predictor performs well with very small STB1 banks of four or six entries. But beyond six entries, load coverage is better without an NFS predictor due to the 0.47 percent stores that do forward data but are wrongly classified (see Table 5).

Summarizing, in spite of the performance loss due to load coverage decrease, using an NFS predictor increases performance for all the analyzed bank sizes, because contention for issue ports to memory is consistently reduced. This enhancement is particularly important if store contention is a big issue, for example, in the case when cache bank mirroring is used to increase load bandwidth [8], [25], [29].

6.2 Removing STB2 Data Forwarding Capability

In order to forward data from STB2, we need as many data read ports as there are cache banks, and the IQ Scheduler must handle two latencies for load instructions. To that end, a load is first issued by the IQ hoping that it will be serviced from the STB1/L1 cache, and then, the IQ speculatively wakes up its dependent instructions. If the STB2 discovers later that a forwarding misspeculation has arisen, a recovery action is undertaken. The load is tagged so that it can be reissued by the IQ now assuming forwarding from the STB2.

As STB2 service is very infrequent (see Fig. 10b), we remove the STB2 store-load data forwarding capability. In

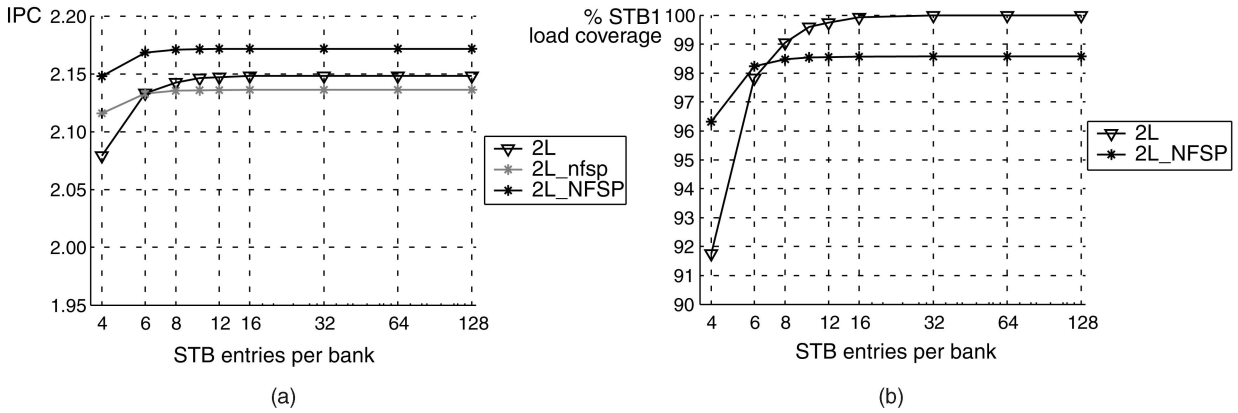

Fig. 11. (a) IPC for 2-level STB systems with an NFS predictor (2L_NFSP), without it (2L), and with an NFS predictor used only to filter store insertion in the STB1 (2L_nfsp). (b) STB1 Load coverage for two two-level systems with a Nonforwarding store predictor (2L_NFSP) and without it (2L).

turn, this STB2 simplification allows a simpler IQ scheduler (single-load latency) and also removes the STB2 data forwarding read ports and their counterpart input ports to the Bypass Network. Now, every time the STB2 discovers a forwarding misspeculation, the load is reissued by the IQ in order to obtain the data from the cache after the offending store Commits. By looking at simulations, we have realized that usually after a forwarding misspeculation, by the time the load instruction is re-executed for the first time, the matching store has already written the cache.

Fig. 12 shows a system with STB2 forwarding capability (2L_NFSP) and a system without it (2L_NFSP_**NoFW**). Both are basic two-level STB systems with an NFS predictor.

We have found a performance decrease ranging from 0.8 percent to 0.2 percent across all tested STB1 bank sizes. In particular, for 8-entry STB1 banks, the IPC loss is about 0.3 percent. Therefore, the forwarding capability can be removed from STB2 without noticeably hurting performance. This loss of performance comes from a small number of loads waiting until a matching store Commits.

6.3 Simplifying STB1 Data Forwarding Logic

As we have seen in Section 3, our main objective is to design a store-load forwarding logic as fast as a small single-ported cache bank. In this section, we present two STB1 simplifications which help to reduce the STB1 latency: first, we eliminate the age-based selection circuit, and second, we reduce the number of bits used to compare addresses. These

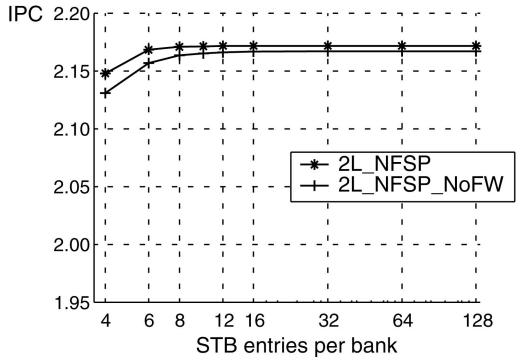

Fig. 12. IPC for 2-Level STB systems having STB2 forwarding capability (2L_NFSP) or not (2L_NFSP_NoFW).

simplifications add new forms of data forwarding misspeculations to the already existing ones. However, the overall two-level STB operation, in which the STB2 checks the data forwarding for all loads, supports all these new forms of misspeculation without adding new complexity.

6.3.1 Removing Age-Based Selection

This simplification removes the CAM structure (age-CAM) that compares ages in the selection logic. Now, the most recently allocated entry having a matching address will forward data. Explicit ages are still needed, but they are not used within the critical STB1 data forwarding path. Namely, explicit ages are needed to: 1) purge the proper STB1 entries (branch misprediction, data forwarding misspeculation, etc.) and 2) label the loads fed by the STB1 so that the STB2 can detect forwarding misspeculations.

Because entries are allocated out of order, purging can make holes in STB1 (for instance, purging due to branch misprediction). In order to keep STB1 complexity as low as possible, we propose not using compacting circuitry, even though effective capacity may shrink. All simulations below have been done according to this assumption.

When removing age-based selection, the following corner case can arise: let us suppose a store following, in program order, a load to the same address carrying a wrong bank prediction. If the store is issued immediately after the load, the load reaches the correct STB1 bank after the store. A few cycles later, the STB2 discovers this load suffering a forwarding misspeculation, and it starts a (nonselective) recovery from the RIB. As the load is redispatched from the RIB keeping the same bank prediction (wrong), the initial situation happens once again.

To make sure that program execution makes forward progress and that a load is not endlessly serviced from a younger store, the load is tagged when it is redispatched so that its re-execution does not check the STB1 again.

6.3.2 Partial Address Comparison

This simplification reduces the size of each entry in the CAM structure that compares addresses (address-CAM). We compare only the N least significant bits of load/store addresses. In general, partial address comparison is used in some processors to conservatively delay the load execution on a partial address match with previous stores [15].

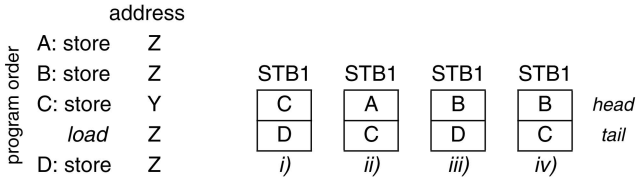

Fig. 13. Four examples of forwarding misspeculations in a 2-level STB design with 2-entry STB1 banks. Store instructions are inserted by the tail and removed by the head.

Instead, we will use partial address comparison to speculatively forward data from a store to a load on a partial address match.

6.3.3 Misspeculation Examples

Fig. 13 shows a code in which a Load instruction should be fed from data of the Store labeled B. We assume all addresses converge into a 2-entry STB1 bank, and each example shows a possible misspeculation arising when the Load instruction finds the STB1 filled with the contents depicted. Entries in the STB1 are assigned FIFO as stores A, B, C, and D are executing in any relative order. In all examples, store B has already been executed but it is still not committed.

The first two examples (*i*, *ii*) happen in an STB1 without simplification, and we assume Store B could have been deallocated from the STB1 by the execution of other stores or that Store B could never even have been inserted in the STB1 after being filtered by the NFSP. In the first case (*i*), data is wrongly supplied by the L1 cache as there is no older store in the STB1 matching the load address. In the second case (*ii*), data from store A is wrongly forwarded by the STB1, because Store A is older than the load and their addresses match.

The third example (*iii*) shows a forwarding misspeculation appearing when removing the age-based selection circuit out of the STB1. Store B is present in the STB1, but data is wrongly forwarded from the younger store D, because it is the last executed store having the load address.

The last example (*iv*) shows a forwarding misspeculation appearing when the STB1 compares only an address bit subset. In this case, the data will be wrongly forwarded from store C as long as the selected address bit subset has the same value in addresses Y and Z.

6.3.4 Performance Impact of the STB1 Simplifications

Fig. 14 shows the IPC for several two-level systems. All of them are basic two-level STB systems with an NFS predictor and an STB2 without the forwarding capability.

The baseline in this section is 2L_NFSP_NoFW, where the STB1 forwards data by both checking instruction ages and doing full address comparison. In 2L_NFSP_NoFW_**NoAGE**, the STB1 performs full address comparison but does not check ages. In the other three systems (2L_NFSP_NoFW_NoAGE_**@n**), the STB1 does not check ages and compares only n address bits taken from the $n+2$ least significant address bits, where the two removed bits correspond to the bank number.

Removing age checking degrades performance around 0.6 percent across all STB1 bank sizes (2L_NFSP_NoFW vs. 2L_NFSP_NoFW_NoAGE). Thus, store selection can be done regardless of age with negligible performance loss. Degradation happens mainly when a store forwards data to an older load.

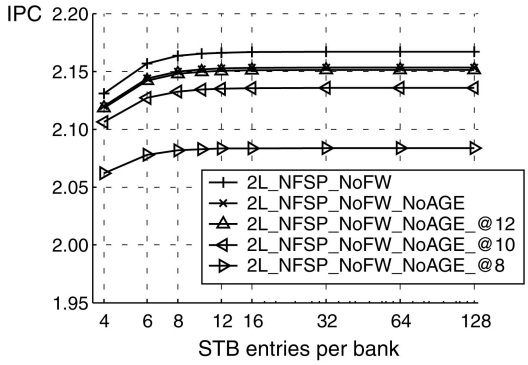

Fig. 14. IPC for 2-level STB systems with explicit age checking (2L_NFSP_NoFW), without age checking (2L_NFSP_NoFW_NoAGE), and with a partial address comparison of n bits (2L_NFSP_NoFW_NoAGE_@n).

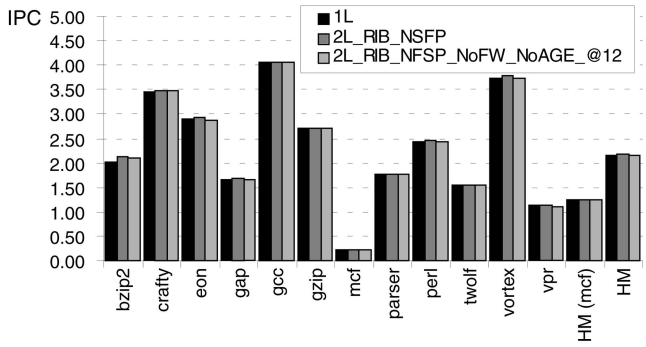

Fig. 15. Individual IPC for all SPECint2K programs.

Once age checking has been removed, comparing only 12 address bits causes a negligible performance degradation (0.1 percent) across all STB1 bank sizes (2L_NFSP_NoFW_NoAGE versus 2L_NFSP_NoFW_NoAGE_**@12**). Degradation increases when using 10 or 8 address bits (around 0.8 percent and 3.3 percent, respectively). Thus, store selection can be done with partial-address comparison.

7 Individual Program Results

In this section, we summarize the impact of all design decisions by looking at the individual program behavior of SPECint2K (Fig. 15) and SPECfp2K (Fig. 16).

Both figures show performance of a single-level distributed system (1L) and two implementations of a two-level system. The 1L system uses 128-entry STB banks which are reachable within L1 cache latency. Both two-level systems (2L_NFSP and 2L_NFSP_NoFW_NoAGE_@12) have 8-entry STB1 banks and use an NFS predictor. The second two-level system removes STB2 forwarding and has an STB1 that does not check instruction ages and compares only 12 bits of data addresses.

As Fig. 15 shows, all SPECint2K programs follow the same trends. We can see that removing STB2 forwarding capability, performing STB1 data forwarding without age checking, and comparing a 12-bit address subset end up in negligible performance losses. So, a two-level STB system with a single load latency and made up of simple 8-entry STB1 banks performs similar to an ideal 128-entry one-level STB.

With respect to SPECfp2K, Fig. 16 shows that the behavior of the three configurations is also very similar

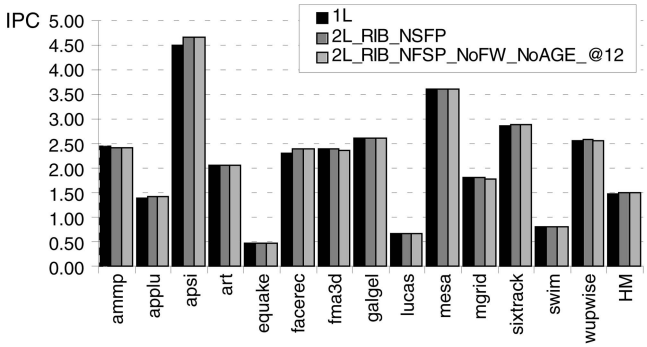

Fig. 16. Individual IPC for all SPECfp2K programs.

across all programs. The two-level systems with 8-entry STB1 banks perform just like the idealized one-level system. Comparing both two-level systems, we can conclude that the simplifications added reduce performance by a negligible 0.1 percent.

The sensitivity of the IPC to the number of STB1 entries in SPECfp2K is not shown because performance is almost flat; the IPC variation is within 0.4 percent across all reported STB1 sizes. In order to explain such results, Table 6 presents some key statistics about the forwarding behavior of SPECint2K and SPECfp2K.

The first and second rows show the average percentage of *Store instructions* and *forwarding Store instructions*, respectively. The last two rows show *Store lifetime* in a single-level distributed STB with 128-entry STB banks.

Time from Dispatch to Commit is the average number of cycles a Store has allocated an STB2 entry, while *Time from Execution to last forwarding* is the minimum average time we would like a forwarding Store to have an STB1 entry allocated. Both SPECfp2K lifetimes double those of SPECint2K, and so the required number of STB entries seems to increase at both levels to sustain performance.

However, this higher demand does not imply performance degradation, because it is compensated by a lower number of store instructions (from 12.6 percent to 9.6 percent) and by a lower number of forwarding stores (from 22.4 percent to 11 percent).

TABLE 6
Forwarding Behavior for SPEC2K (INT and FP) in a Single-Level Distributed STB with 128-Entry STB Banks

		INT	FP
	# of stores / # of instructions	12.6 %	9.6 %
	# of forwarding stores / # of stores	22.4 %	11 %
Store lifetime (in cycles)	From Dispatch to Commit	46.4	88
	From Execution to last forwarding	8.5	18

8 Memory-Latency Tolerant Processors

In order to hide the effect of long memory access delays, some techniques which increase the number of in-flight instructions have been proposed [1], [7], [13], [26]. Of course, such techniques increase the number of store instructions to be kept in STB.

In this section, we study the behavior of a two-level STB working in a memory-latency-tolerant (MLT) processor. To model such a processor, we have set the Reorder Buffer, Instruction Queue, and STB2 to 2,048 entries, and the Miss Request Queue to 64 entries in our simulator.

Fig. 17 shows the IPC for SPECint2K and SPECfp2K, for an efficient two-level STB system working both in the baseline processor described in Table 1 (2L_NFSP_NoFW_NoAGE_@12) and in the MLT processor (**MLTP**_2L_NFSP_NoFW_NoAGE_@12). We have also plotted the performance of the one-level distributed STB and the Akkary two-level STB working in the MLT processor (MLTP_1L and MLTP_2L_MTB) [1].

As shown in other works, increasing the number of in-flight instructions has a great impact on floating point performance, but a limited one on integer performance (MLTP_2L_NFSP_NoFW_NoAGE_@12 versus 2L_NFSP_NoFW_NoAGE_@12) [1], [8], [26].

The MLT processor with one-level STB requires a large number of STB entries to achieve a good performance (MLTP_1L). The MLTP_2L_MTB performance decreases by 10 percent for SPECint2k and by 11 percent for SPECfp2k when reducing the STB1 bank size from 2,048 to 8 entries. However, when considering our two-level STB, performance is almost independent of the number of STB1 entries

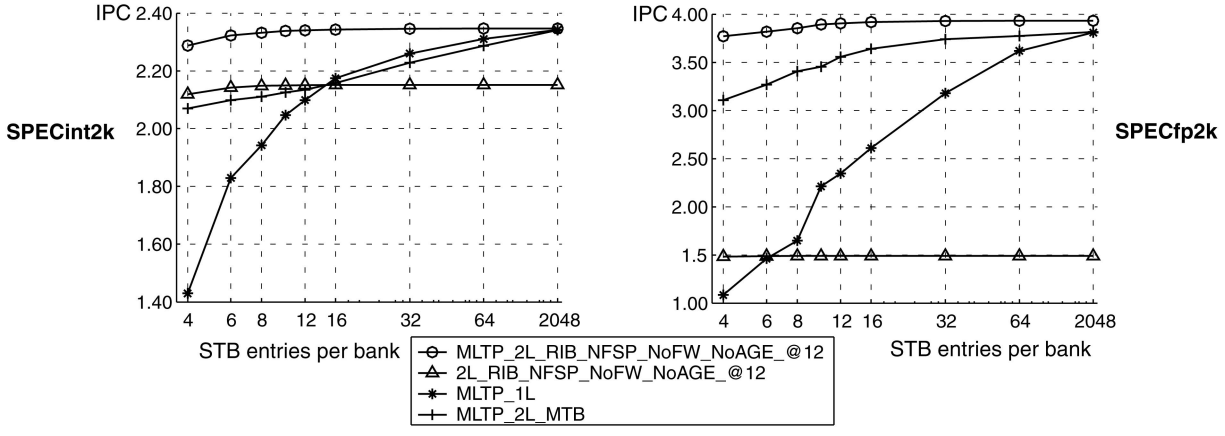

Fig. 17. SPECint2k and SPECfp2k IPC for efficient two-level STB systems working in the MLT processor (MLTP_2L_NFSP_NoFW_NoAGE_@12) and in the baseline processor (2L_NFSP_NoFW_NoAGE_@12). Also shown is a one-level distributed STB and the Akkary two-level STB working in the MLT processor (MLTP_1L and MLTP_2L_MTB).

TABLE 7
Forwarding Behavior for SPEC2K (INT and FP) in a Single-Level Distributed STB with 2,048-Entry STB Banks Working in an MLT Processor

		INT	FP
	# of stores / # of instructions	12.6 %	9.6 %
	# of forwarding stores / # of stores	28 %	18 %
Store lifetime (in cycles)	From Dispatch to Commit	207.5	320.2
	From Execution to last forwarding	10	15.7

(MLTP_2L_NFSP_NoFW_NoAGE_@12). Namely, reducing the number of STB1 bank entries from 2,048 to eight performs only 0.6 percent worse for SPECint2k, and 2 percent for SPECfp2k. So, our two-level STB proposal seems to be a valuable design option for memory-latency tolerant processors.

To get an insight into such good results of the proposed two-level system, Table 7 shows the forwarding behavior of SPECint2K and SPECfp2K executing on the MLT processor.

When comparing the forwarding behavior of MLT and base processors (Table 6), we see a significant increase in the time a store remains within the pipeline (from Dispatch to Commit time it gets multiplied by four). As there are more in-flight instructions, the number of forwarding stores increases 25 percent and 64 percent for SPECint2K and SPECfp2K, respectively. However, the time from Execution to last forwarding barely increases or even decreases (from 18 to 15.7 cycles in SPECfp2K). This *temporal locality* in the store-to-load data forwarding, which is the underlying cause of the good results achieved by the proposed two-level STB design, seems to be a quality of the workload and the out-of-order execution model. The temporal locality can be explained in two ways, either the store and its dependent loads are close together in the instruction stream, or, when they are apart, instructions between them belong to a separate dependent chain and are executed before or after them.

Finally, notice that an MLT processor with a two-level STB having the largest number of STB1 entries (2,048) performs better than an MLT processor with an equally sized one-level STB. This is because the two-level STB has an NFS predictor, which reduces contention in the issue ports to memory.

9 Related Work

Yoaz et al. introduce the concept of Sliced Memory Pipeline and exemplify it with a two-banked first-level cache [30]. They propose to steer store instructions to both banks. However, we have shown that memory issue port contention is a problem, and consequently, spreading stores to all STB banks can hurt performance.

Zyuban et al. partition a Load/Store Queue (LSQ) into banks [31]. When dispatching a store, an entry is allocated in all LSQ banks and it remains allocated until the store Commits. A large number of LSQ entries are needed not to stall Dispatch, as shown in Section 3.

Racunas and Patt propose a new dynamic data distribution policy for a multibanked L1 cache [18]. As in the preceding work, they use a partitioned STB (local STB), whose entries are allocated to stores at dispatch time and deallocated at Commit time. Therefore, local STB banks need a large number of entries not to stall Dispatch. Allocation to a local STB bank is done using bank prediction. Store information (address, data, etc.) has to be transferred between local STB banks in two situations: 1) when a store allocation misprediction happens and 2) when a cache line is dynamically transferred from one bank to another, in which case all local STB entries related to that line have to be moved from the initial bank to the target bank. When transferring stores from one bank to another, room should be assured in the target bank. Local STB banks use age information to decide which store instruction forwards data when several stores match a load address. On the other hand, a global STB with forwarding capability is used to forward data to all loads experiencing bank mispredictions.

In contrast to the two proposals above, we can use STB1 banks with a very small number of entries, because they are allocated late and deallocated early. As STB1 entries are allocated just after checking the predicted bank, our design does not need interbank communication. Besides, our two-level proposal enables a low-complexity design, which has STB1 banks that do not use age information nor full-address comparison to speculatively forward data, and which has an STB2 that does not require any forwarding capability.

Akkary et al. propose a hierarchical store queue organization to work along with a centralized cache on a Pentium 4-like processor (one load per cycle) [1]. It consists of a fast STB1 plus a much larger and slower backup STB2 (both centralized) and an MTB to reduce the number of searches in STB2. The MTB is on the load execution critical path. Moreover, the MTB is managed out-of-order and speculatively. Thus, maintaining precisely its contents is difficult. In this paper, we model an oracle MTB, and therefore, the conflicting stores and the false positives from loads that match younger stores do not exist. We show that the performance of this approach is outperformed by our proposal. In a later work, Gandhi et al. propose a new, centralized, two-level STB scheme [9]. In the absence of long-latency load cache miss, a conventional one-level STB performs all the duties required. In the shadow of such a long-latency cache miss, they switch responsibility to a two-level STB made up of a first-level forwarding cache and a second-level simplified STB2. In this second operating mode, forwarding is mainly done through a specific forwarding cache, which is speculatively written by stores in execution order. After a cache miss service, a limited form of speculative forwarding through the STB2 is also used, which avoids associative search by using indexed references to the STB2.

Because our first-level STB is a multibanked structure and entry allocation is delayed until store execution, very small STB1 banks suffice for good performance. Therefore, associative search is limited to very few entries. Also, we evaluate our multibanked STB1 proposal in a multibanked cache configuration, which requires analyzing issues such as how to manage multiple STB1 banks and how to cope with memory issue port contention. Moreover, we suggest simple designs for both the STB1 (no age checking, partial-address comparison) and the STB2 (no forwarding).

Baugh and Zilles [3], and Roth [20] use a two-level STB design in which data forwarding is restricted to the first level. On the one hand, Baugh and Zilles use a centralized STB1 whose entries are allocated in program order at dispatch time, but only by those store instructions predicted as forwarders [3]. Furthermore, to reduce the STB1 search bandwidth, they use a simple predictor to predict which load instructions might require forwarding. Instead, we propose a distributed STB1 whose entries are allocated at Execution, together with a

Nonforwarding Store Predictor that reduces contention in the issue ports to memory. Like us, their second-level STB is used only as a checking device, but it is split in address-interleaved banks and its entries are allocated at Execution.

On the other hand, Roth uses a centralized queue (FSQ1) as an STB1 in parallel with a multibanked forwarding buffer (F1), which keeps unordered stores and handles simple forwarding cases [20]. FSQ1 entries are allocated in program order by those stores, which forwarded incorrectly from F1 in the past. Likewise, loads that were incorrectly fed from the F1 in the past are steered to the FSQ1. The second-level STB is an FIFO buffer used only for store Commit. Forwarding misspeculations are discovered by compelling selected (vulnerable) loads to reaccess the data cache at Commit time. In contrast, our proposal does not have to split the load/store stream among forwarding sources, because the STB1 is the only forwarding provider.

Sha et al. propose eliminating the associative search in age-ordered STBs by using two predictors [23]. One of them identifies, for each load, the most likely forwarding STB entry. The other predictor is used to delay difficult-to-predict loads until all but the youngest of their potential forwarding stores have committed.

Stone et al. suggest using an address-indexed store-forwarding cache to perform speculative store-load forwarding, together with an address-indexed memory disambiguation mechanism [27]. Dependences among memory instructions are predicted and enforced by the instruction scheduler.

In order to eliminate load searches in the Store Queue, Setthumadhavan et al. propose using a Bloom filter [21]. They also use another filter to decide which loads should be kept in the Load Queue, thus reducing the Store Queue bandwidth and the Load Queue size. Similarly, Park et al. reduce the STB search bandwidth by using a Store-Load pair predictor based on the Store-sets predictor [17]. They also split the STB into multiple smaller queues with variable latencies. The ideas in both papers could be applied to our second-level STB in order to reduce power consumption and the number of STB2 ports by either reducing the number of searches or the number of entries to be searched.

10 Conclusion

High-performance out-of-order processors need to efficiently forward data among noncommitted stores and loads. The STB is the structure in charge of that: it keeps all in-flight stores, supports address-based associative search and age-based selection for every issued load, and forwards the right data whenever a match takes place. In a balanced design, size and bandwidth of the STB should be proportional to the instruction window size and the issue width, respectively. Large and multiported STBs can either compromise the processor cycle time, or increase the forwarding latency, or both.

In this paper, we find out that the store-load forwarding is performed in a narrow window of time after store execution, showing up that from a forwarding point of view the conventional allocation/deallocation policy of STB entries can be improved a lot. Therefore, we suggest a two-level STB in which the first level only deals with data forwarding (speculatively). Owing to the temporal locality of the store-load forwarding activity, the first-level STB (STB1) can be really small, and therefore, very fast (as fast as the L1 cache). The second-level STB (STB2), out of the data forwarding critical path, checks the speculative forwarding and, if necessary, starts recovery. The two-level STB design has been applied to a processor with a sliced memory pipeline, where, in consequence, the STB1 is distributed in several banks.

An STB2 entry is allocated at store Dispatch, and deallocated at store Commit. However, the allocation of an STB1 entry is delayed until the store executes, and the deallocation can proceed before the store Commits. If an STB1 bank is full, entries are managed in FIFO order. Such STB1 allocation/deallocation policy allows reducing the STB1 size, allocating the entry only in the right STB1 bank, and not stalling the Dispatch stage when STB1 banks become full.

Moreover, the proposed role distribution between levels enables three design simplifications that do not hurt performance noticeably: 1) forwarding capability can be removed from the STB2, 2) the STB1 does not use instruction age to select a forwarding store, and finally, 3) the number of bits used to compare addresses in the STB1 can be greatly reduced.

A nonforwarding store predictor can be used to reduce contention for the issue ports to memory. Stores having a nonforwarding prediction are issued by any free memory port, thus increasing effective issue bandwidth.

Following our guidelines, a two-level STB with 8-entry STB1 banks (STB1 without age checking and with partial address comparison, and STB2 without forwarding capability) performs similar to an ideal single-level STB with 128-entry banks working at first-level cache latency. Both the concept and the guidelines are similar to other multibanked L1 data cache organizations (for instance those with a second queue which schedules memory accesses for banks). Microarchitectural techniques such as multithreading aimed at improving throughput, and the trend toward more in-flight instructions to hide the ever-growing memory latency are going to increase STB storage requirements. In this latter scenario, we have shown that our proposal can help in designing a fitted STB.

Acknowledgments

This work was supported in part by grants TIN2007-66423 and TIN2007-60625 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

References

[1] H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Int'l Symp. Microarchitecture (MICRO-36)*, pp. 423-434, Dec. 2003.

[2] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Managing the Communication-Parallelism Trade Off in Future Clustered Processors," *Proc. 30th Int'l Symp. Computer Architecture (ISCA-30)*, pp. 275-287, June 2003.

[3] L. Baugh and C. Zilles, "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability," *Proc. IBM P = AC$\hat{2}$ Conf.*, pp. 52-61, Oct. 2004.

[4] D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, UW Madison Computer Science, June 1997.

[5] G.Z. Chrysos and J.S. Emer, "Memory Dependence Prediction Using Store Sets," *Proc. 25th Int'l Symp. Computer Architecture (ISCA)*, pp. 142-153, June 1998.

[6] J. Cortadella and J.M. Llabera, "Evaluation of A + B = K Conditions without Carry Propagation," *IEEE Trans. Computers,* vol. 41, no. 11, pp. 1484-1488, Nov. 1992.
[7] A. Cristal, O.J. Santana, and M. Valero, "Toward Kilo-Instruction Processors," *ACM Trans. Architecture and Code Optimization (TACO),* vol. 1, no. 4, pp. 389-417, Dec. 2004.
[8] J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz, 64-Bit, Quad Issue, CMOS RISC Microprocessor," *Digital Technical J.,* vol. 7, no. 1, pp. 119-135, Jan. 1995.
[9] A. Gandhi, H. Akkary, R. Rajwar, S.T. Srinivasan, and K. Lai, "Scalable Load and Store Processing in Latency Tolerant Processors" *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* pp. 446-457, June 2005.
[10] P. Hsu, "Design of the TFT Microprocessor," *IEEE Micro,* vol. 14, no. 2, pp. 23-33, Apr. 1994.
[11] C.N. Keltcher, K.J. McGrath, A. Ahmed, P. Conway, C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro,* vol. 23, no. 2, pp. 66-76, Apr. 2003.
[12] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro,* vol. 17, no. 2, pp. 27-32, Apr. 1997.
[13] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture (ISCA),* pp. 59-70, May 2002.
[14] P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," *Proc. 24th Int'l Symp. Computer Architecture (ISCA),* pp. 292-303, June 1997.
[15] S.D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T.J. Sullivan, and T. Grutkowski, "The Implementation of the Itanium 2 Microprocessor," *IEEE J. Solid State Circuits,* vol. 37, no. 11, pp. 1448-1460, Nov. 2002.
[16] H. Neefs, H. Vandierendonck, and K. De Bosschere, "A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 313-324, Jan. 2000.
[17] I. Park, L.O. Chong, and T.N. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue," *Proc. 36th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 411-422, Dec. 2003.
[18] C. Racunas and Y.N. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitectures," *Proc. 17th Int'l Conf. Supercomputing (ICS),* pp. 22-31, June 2003.
[19] A. Roth, "A High-Bandwidth Load/Store Unit for Single- and Multi-Threaded Processors," Technical Report MS-CIS-04-09, Univ. of Pennsylvania, June 2004.
[20] A. Roth, "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," *Proc. 32th Int'l Symp. Computer Architecture (ISCA),* pp. 458-468, June 2005.
[21] S. Sethumadhavan, R. Desikan, D. Burger, C.R. Moore, and S.W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," *Proc. 36th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 399-410, Dec. 2003.
[22] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," *Proc. 29th Int'l Symp. Computer Architecture (ISCA),* pp. 295-306, May 2002.
[23] T. Sha, M. Martin, and A. Roth, "Scalable Store-Load Forwarding via Store Queue Index Prediction," *Proc. 38th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 159-170, Nov. 2005.
[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behaviour," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 45-57, Oct. 2002.
[25] G.S. Sohi and M. Franklin, "High-Bandwidth Memory Systems for Superscalar Processors," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 53-62, Apr. 1991.
[26] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual Flow Pipelines," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 107-119, Oct. 2004.
[27] S.S. Stone, K.M. Woley, and M.I. Frank, "Address Indexed Memory Disambiguation and Store-to-Load Forwarding," *Proc. 38th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 171-182, Nov. 2005.
[28] E. Torres, P. Ibáñez, V. Viñals, and J.M. Llabería, "Contents Management in First-Level Multibanked Data Caches," *Proc. 10th Int'l Euro-Par 2004 Conf.,* pp. 516-524, Sept. 2004.
[29] E. Torres, P. Ibáñez, V. Viñals, and J.M. Llabería, "Store Buffer Design for Multibanked Data Caches," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* pp. 469-480, June 2005.
[30] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling," *Proc. 26th Int'l Symp. Computer Architecture (ISCA),* pp. 42-53, May 1999.
[31] V. Zyuban and P.M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Trans. Computers,* vol. 50, no. 3, pp. 268-285, Mar. 2001.

Enrique Torres received the MS degree in computer science from the Polytechnic University of Catalunya in 1993, and the PhD degree in computing science from the University of Zaragoza in 2005. He was an assistant professor in the Polytechnic Schools of the University of Girona. He is an assistant professor in the Computer Science and Systems Engineering Department (DIIS) at the University of Zaragoza, Spain. He is also on sabbatical leave for study and research at the University of California in Berkeley, where he is a member of the International Computer Science Institute (ICSI). His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is a member of the IEEE Computer Society. He is also a member of the Aragón Institute of Engineering Research (I3A) and the European HiPEAC NoE. More details about his research and background can be found at http://webdiis.unizar.es/gaz/miembros.html.

Pablo Ibáñez received the MS degree in computer science from the Polytechnic University of Catalunya in 1989, and the PhD degree in computer science from the University of Zaragoza in 1998. He is an associate professor in the Computer Science and Systems Engineering Department (DIIS) at the University of Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is member of the IEEE and the IEEE Computer Society. He is also a member of the Aragón Institute of Engineering Research (I3A) and the European HiPEAC NoE.

Víctor Viñals-Yúfera received the MS degree in telecommunication and the PhD degree in computer science from the Polytechnic University of Catalunya (UPC) in 1982 and 1987, respectively. He was an associate professor at the Barcelona School of Informatics (UPC) during 1983-1988. Currently, he is a professor in the Computer Science and Systems Engineering Department (DIIS), University of Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is a member of the IEEE, the ACM, and the IEEE Computer Society. He is also a member of the Aragón Institute of Engineering Research (I3A) and the European HiPEAC NoE. He belongs to the Juslibol Midday Runners Team.

José M. Llabería received the MS degree in telecommunication, and the MS and PhD degrees in computer science from the Polytechnic University of Catalunya (UPC) in 1980, 1982, and 1983, respectively. He is a professor in the Computer Architecture Department, UPC, Barcelona, Spain. His research interests include high-performance architectures, memory hierarchy, multicore architectures, and compiler technology. He is a member of the European HiPEAC NoE.