Store Buffer Design in First-Level Multibanked Data Caches

E.F. Torres[1], P. Ibañez[1], V. Viñals[1], J.M. Llabería[2]

[1] U. de Zaragoza, Spain and HiPEAC NoE. {enrique.torres, imarin, victor}@unizar.es
[2] U. Politècnica de Catalunya, Spain. llaberia@ac.upc.es

Abstract

This paper focuses on how to design a Store Buffer (STB) well suited to first-level multibanked data caches. Our goal is to forward data from in-flight stores to dependent loads with the latency of a cache bank. For that 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, while a centralized, second-level STB enforces correct store-load ordering a few cycles later. To that end we have identified several important design decisions: i) delaying allocation of first-level STB entries until stores execute, ii) deallocating first-level STB entries before stores commit, and iii) selecting a recovery policy well-matched to data forwarding misspeculations. Moreover, the two-level STB admits two enhancements that simplify the design leaving performance almost unchanged: i) removing the data forwarding capability from the second-level STB, and ii) not checking instruction age in first-level STB prior to forwarding data to loads. Following our guidelines and running SPECint-2K over an 8-way out-of-order processor, a two-level STB (first level with four STB banks of 8 entries each) performs similarly to an ideal, single-level STB with 128-entry banks working at the first-level cache latency.

1. Introduction

Codes without much data parallelism are very sensitive to the load-use delay [19]. Therefore, keeping both latency low and bandwidth high are key design goals for first-level (L1) caches in superscalar processors.

Multibanked L1 data caches are being considered good candidates to support wide pipelines [2][7][8][9][13] [21][22]. Multibanking provides low latency and high bandwidth by physically splitting storage in independent, single-ported cache banks. An option in order to keep a simple and low-latency pipeline is to predict the target cache bank before the Issue stage, and to couple every cache bank with an address generation unit. In this option, called *sliced memory pipeline*, the memory pipeline splits into simple and independent slices, but now performance will come at the price of an accurate bank predictor [13] [21].

This work was partly funded by the Diputación General de Aragón, and grants TIN2004-07739-C02-01/02 (Spanish Ministry of Education/ Science and European RDF).

In order to support precise exceptions in out-of-order processors, store instructions write cache in program order at commit time. However, to speed program execution while enforcing store-load memory ordering, a feature called store-to-load data forwarding is added (data forwarding, for short). Data forwarding allows in-flight loads reading a given address to take data from the nearest among the oldest, non-committed stores writing to the same address. For this, a structure called Store Buffer (STB) is used. Stores wait in STB until they commit and, using CAM capability and additional logic, store-load memory ordering is enforced and data forwarding is performed.

STB is a critical component of out-of-order processors because its data forwarding capability can impact cycle time [1]. The circuit that identifies and forwards data is complex and incurs long delays as STB size increases. If the trend towards fast processor clocks, wider pipelines and increasing number of in-flight instructions goes on, the latency problem posed by the STB may worsen [1].

A multibanked L1 data cache organization also requires a distributing STB, resulting each STB bank coupled to a cache bank. V. Zyuban and P.M. Kogge use a single-level multibanked STB in [22]. In their work, STB entries are allocated to stores in all STB banks when they are dispatched to Issue Windows. Because store-load ordering enforcement and data forwarding are performed by STB banks, stores can not be removed from STB banks until committing. Dispatch is stalled whenever STB becomes full. As we will show later, allocating and deallocating STB entries at dispatch and commit time, respectively, reduces processor performance significantly.

In this paper we are concerned with two-level distributed STBs well suited for multibanked caches whose banks are integrated in a sliced memory pipeline. We propose a small and fully distributed first-level STB (STB1) specialized in speculative data forwarding, and a centralized, second-level STB (STB2) specialized in enforcing memory ordering. In particular, STB2 will check all data forwardings made by STB1.

To reduce STB1 size we delay allocation of STB1 entries to stores until they execute, possibly out of program order, doing then allocation and filling at once. We also allow entry deallocation to proceed before store commit: whenever an incoming store is executed and its STB1 bank is full, the oldest allocated store is overwritten.

At dispatch time, entries are allocated to stores in program order only in STB2. So, dispatch will only be stalled when running out of STB2 entries, irrespective of the STB1 size. When a store is issued by the correct port to memory, address and data are copied into both STB1 and STB2.

The two-level organization allows us to simplify the STB1 and STB2 design without noticeably hurting performance: STB1 does not use instruction age to select the store that forwards data, and STB2 keeps all in-flight stores until they commit, but it does not forward data.

As STB1 banks do not keep all the non-committed stores, any data given to a load by the L1cache/STB1 ensemble is speculative and must be verified in STB2 (*data forwarding check*). For example, from the moment a store is removed from STB1 until it commits, it is necessary to identify any consumer load matching such evicted store. We call *forwarding misspeculations* to this kind of situations, which are detected by STB2 when loads reach it.

Ones a forwarding misspeculation has been detected, a recovery action is undertaken. Recovering from Issue Queue (IQ), as usual in other latency mispredictions, increases IQ pressure because loads and all the speculatively issued dependent instructions have to be kept in IQ until STB2 performs the data forwarding check.

However, the very low number of forwarding misspeculations we have measured allows us to decrease IQ pressure by recovering from a *Renamed Instruction Buffer* which stores all in-flight instructions in program order. This recovery policy has a high misspeculation penalty, but in the frequent case of no misspeculation it enables removing instructions from IQ earlier, namely after performing the L1 cache hit/miss check.

Finally, in order to reduce contention for the issue ports to memory (the ports that send memory instructions to L1 cache banks), we use a non-forwarding store predictor. Stores having a non-forwarding prediction are issued by *any* free memory issue port, bypassing STB1 and staying only in STB2, thus increasing the effective issue bandwidth. This predictor has proven to be especially useful in first-level cache systems having banks with replicated contents (the same cache line is duplicated in several cache banks in order to provide more load bandwidth).

This paper is structured as follows: Section 2 outlines the processor-cache model being used, motivates the work, and provides the whole set of suggested two-level STB design guidelines. Section 3 details simulation environment. Section 4 evaluates STB design alternatives for line-interleaved multibanked L1 caches, extending analysis in Section 5 to cache banks having replicated contents. Section 6 discusses related work, and Section 7 concludes the paper.

2. A Two-Level STB in a Multibanked L1 Cache Framework

First the model of processor and multibanked L1 cache being used is outlined. Next, the behavior and limitations of a simple approach for distributing STB is analyzed. Finally, the design alternatives used in this paper to overcome the previous limitations are introduced.

2.1. Model of processor and multibanked L1 cache

A First-level cache is made of several address-interleaved L1 cache banks operating as independent, pipelined, memory slices (Figure 1). We assume an Issue Queue (IQ) with a fixed number of scheduler ports shared among integer ALUs and memory slices (ALU/AGU + cache bank). Load and store instructions are dispatched to IQ, carrying a prediction on their target cache bank (Bank Predictor). By using this prediction IQ issues memory instructions to the predicted slice. To enable recovering from IQ, load and store instructions remain in IQ until bank predictions are verified.

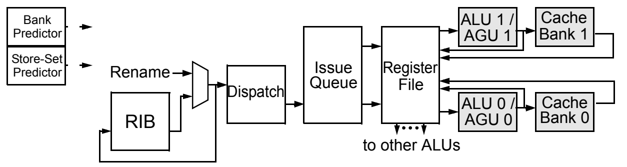


Figure 1. Simplified data-path example with a two-banked L1 cache

L1 cache hit latency is blindly predicted for all loads, and thus dependent instructions are speculatively woken-up after the hit latency has elapsed. Therefore, there are two sources of load latency mispredictions: bank mispredictions and L1 cache misses. Again, in order to allow recovering from IQ, all speculatively issued instructions that depend on a load are kept in IQ until *all* load predictions are verified (first, bank check in AGU; next, tag check in cache bank). Once a latency misprediction has been detected, the dependent misspeculated instructions of the missed load are re-issued at the right moment, either after the load is re-routed to the correct bank or after the cache miss is serviced. So, as only the dependent instructions are re-issued, recovery is selective.

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 predicted ordering is managed in IQ, which will delay the issue of a load until an older store has been issued if a dependency between them has been predicted. Ordering misspeculations are discovered when stores execute, possibly many cycles after loads and their dependent instructions have left IQ.

Usually, to recover from an ordering misspeculation the load and all younger instructions (dependent or not) are flushed out of the pipeline. Instructions are then re-fetched from the instruction cache. However, the re-fetched instructions are just the same that have been flushed. So, to reduce misspeculation penalty, recovery can be supported by a structure that keeps renamed instructions; we call this structure Renamed Instruction Buffer (RIB).

The RIB is located after rename and before dispatch. It is continuously being filled and keeps all in-flight renamed instructions in program order. Recovery consists in re-dispatching instructions sequentially taken from the RIB to IQ. Recovery is non-selective because the offending load and all subsequent instructions in program order will be re-dispatched. As recovery is not done from the fetch stage, we do not have to checkpoint the Register Map Table on every load instruction as it is done with branch instructions. A similar approach was suggested by Lebeck et al. in [10] to tolerate long latency cache misses. They made a selective recovery and so the RIB is simpler than their proposed buffer structure.

We model an 8-way dynamically scheduled processor with a 256-entry Reorder Buffer having in-flight up to 128 loads and 128 stores (a number large enough not to stall dispatch), four 8KB L1 cache banks, a 256KB second-level (L2) cache and a 2MB third-level (L3) cache. Section 3 details the remaining cache and processor parameters, pipeline timing, benchmarks (SPECint-2K), and simulation methodology.

2.2. Motivation

STB can be distributed in a straight way by placing independent, single-ported, STB banks close to each L1 cache bank [22], an approach we call *single-level distributed STB*. 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 in *all* STB banks is allocated when a store is dispatched, because the store address (and also the destination bank) is still unknown. Eventually, when the store executes in the right slice, a *single* STB bank entry is filled. As stores commit, entries are simultaneously deallocated in *all* STB banks.

Single-level distributed STB performance depends heavily on its size and latency. Let us quantify the impact of these parameters. Figure 2 shows IPC for SPECint-2K as the number of entries of each STB bank varies from 4 to 128 (notice that total STB size is four times that number). Computed IPC assumes that STB access latency is equal to L1 cache latency, no matter the STB size we simulate. A single-level distributed STB limits performance seriously if it is undersized: having 4-entry STB banks bears a 40.5% IPC drop relative to 128-entry STB banks. From 64 entries on, STB does not limit processor performance, while the 10-64 entry range exhibits a gradual IPC decrease. Below 10 entries the IPC slope is very steep.

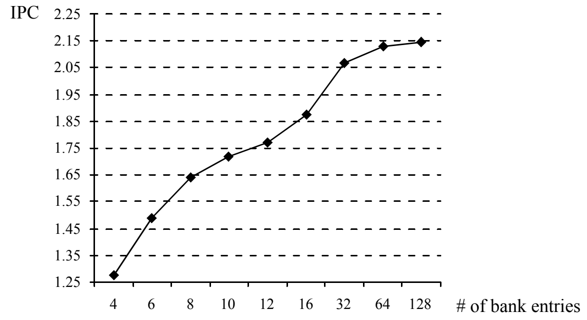


Figure 2. Single-level distributed STB. IPC harmonic mean vs. number of entries per STB bank.

The STB circuitry that identifies and forwards data is complex and incurs long delays as STB size increases [1]. Besides, having an STB with an access time larger than the cache hit access time complicates load instruction scheduling. Every load is tagged with a predicted latency. Resource allocation and speculative wake-up of dependent instructions is made according to the predicted latency[1]. In order to gauge the importance of this fact, we are going to increase STB bank latency (relative to L1 cache latency). Figure 3 shows this simulation for 32-entry STB banks,

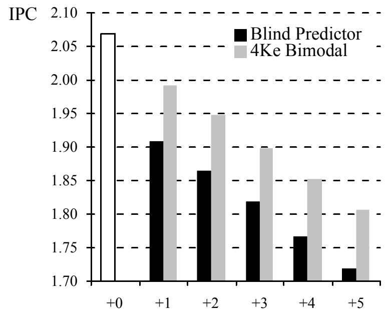


Figure 3. Single-level distributed STB performance for 32-entry STB banks. STB bank access latency increases from L1 cache latency (+0, hollow bar) to L1 cache latency plus 5 cycles (+5). Black bars represent IPC harmonic mean with a blind predictor, while grey bars represent a 4Kentry bimodal predictor.

when increasing STB latencies from +1 to +5 cycles. We simulate two models. The first model predicts cache hit latency for all loads (black bars). The second model (grey bars) uses a 4K-entry bimodal predictor which gives a latency prediction to each load, either a cache hit latency or an STB latency. As we can see, if loads take just one extra cycle to reach, access and forward data from STB banks, the IPC loss resulting from blind prediction is almost 8%. When using a bimodal predictor, IPC degradation roughly accounts for 3% per each additional cycle. Store-load forwarding is highly predictable because in all tested

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

configurations an Oracle-like predictor outperforms the bimodal predictor by only 1%.

Looking at STB utilization is a key factor to understand how to overcome the size-latency tradeoff and increase STB performance. Figure 4 indirectly shows STB utilization by plotting the average lifetime of a committed store.

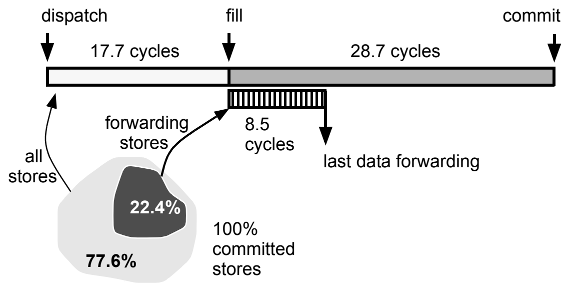


Figure 4. Store lifetime in a single-level distributed STB with 128-entry STB banks.

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% 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 (75% data forwardings occur within the next 7 cycles, and 90% within the next 14 cycles). Therefore, from a data forwarding standpoint, we notice that only a few STB entries are used, being 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 hit access times hurt performance. And another important fact: from a forwarding perspective, STB banks are highly underutilized.

2.3. Two-level STB design guidelines

Our goal is to keep STB bank access latency equal to or under 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 use a two-level STB design with allocation and filling policies specific to each level (Figure 5). STB1 is distributed and each STB bank has a single port. STB2 is centralized and multiported, but it is placed outside the load-use critical path. STB1 and cache banks have equal access times. STB2 latency is initially assumed to be 5 cycles, however performance sensitivity to STB2 latency will be checked in the results section.

A two-level STB can decouple the different tasks performed by a single-level STB as follows. STB1 only performs speculative data forwarding. STB2 checks store-load ordering, performs data forwarding at STB2 speed (whenever STB1 fails to do it), and updates caches in program order.

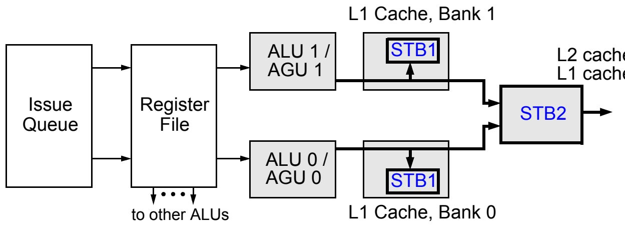


Figure 5. Simplified data-path example with a two-level distributed STB and a two-banked L1 cache.

At dispatch time, entries are allocated to stores only in STB2, where they remain until they commit. So, dispatch is stalled when running out of STB2 entries, irrespective of STB1 size. Notice that the maximum number of in-flight stores is the number of STB2 entries.

Next, we describe several guidelines to deal with the design of a two-level distributed STB. First, we explain how to take advantage of the particular forwarding behavior exhibited by stores. Second, the main drawback of recovering latency misspeculations from IQ is analyzed and a solution is proposed. Third, we suggest two design simplifications that reduce STB1 and STB2 complexity with a minimun loss of performance. Finally, in order to reduce contention for issue ports to memory, we propose to identify non-forwarding stores and send them to STB2 directly.

2.3.1. STB1 allocation policy. In order to reduce the number of cycles an entry is allocated to a particular store, we use the following guidelines.

1. Delay allocation of STB1 entries until stores reach execution stage. Before execution no store wastes STB1 entries. As we pointed out in Figure 4 this policy could shorten store lifetime in STB1 around one third. Now allocation is done after bank check, and thus a *single* entry is allocated to each store in only a *single* STB1 bank.

2. Deallocate STB1 entries before stores commit. We can deallocate entries soon, because most data forwarding is done in a short period of time (see Figure 4 again). This fact inspires the following STB1 replacement policy: a store sent to a full STB1 bank becomes allocated to the place of the oldest store issued to such bank. So, STB1 banks keep only the most recently issued stores and the dispatch stage never stalls even though STB1 gets full. In this particular two-level STB design, a load instruction can obtain the data from either an L1 cache bank, an STB1 bank or STB2. Anyway, we will blindly predict cache hit latency for all load instructions.

2.3.2. Recovery from latency mispredictions. 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 STB2. To that end, the address of an incoming load has to be compared with all store addresses present in STB2. If we find one or more matches with older stores, the nearest one is selected. This operation, which we call *data forwarding check*, verifies whether the load was not forwarded from STB1 or whether it was forwarded by a wrong store. Once the data forwarding check identifies the right store (*forwarding misspeculation*), a recovery action begins.

Instructions dependent on a load could have been speculatively woken up while the data forwarding check is being performed. If IQ is used as the recovery stage, the load itself and all the speculatively issued dependent instructions must remain in IQ until STB2 completes the check (several cycles past the predicted cache hit latency). Consequently, IQ pressure increases and the number of cycles where IQ is full may increase too, which can reduce performance. This fact, coupled to a very low number of forwarding misspeculations suggests that recovering from RIB (like ordering misspeculations), and removing instructions from IQ after performing the cache hit/miss check might be highly beneficial. Simulation results shown below will confirm this intuition.

2.3.3. Design simplifications.

Removing STB2 data forwarding capability. In order to forward data from STB2 we need as many data read ports as cache banks are, and the IQ Scheduler must handle two latencies for load instructions. According to the selected latency, IQ speculatively wakes up the instructions dependent on a given load. To that end, the load is first issued by IQ hoping that it will be serviced from STB1/L1cache. If eventually STB2 discovers a forwarding misspeculation, a recovery action is undertaken. The load is tagged so that IQ can re-issue it assuming forwarding from STB2.

As we will show later, STB2 forwarding capability is not really needed, and it can be *removed* without lowering performance noticeably (Section 4). This STB2 simplification removes complexity from IQ scheduler (only a single load latency has to be managed) and also removes the STB2 read ports and their counterpart input ports to the Bypass Network.

Simplifying STB1 data forwarding logic. As STB1 entries are allocated out of order, stores in STB1 need an explicit age labelling. Among other things, age is used to select among stores matching the same address. That is, if one or more stores match a load address we make use of store age to select the nearest, older store in program order. In any case, STB2 must check the speculative forwarding done in STB1, because the right store could have been deallocated of STB1.

The age-based selection circuit adds complexity and latency to STB1 data forwarding. To simplify this circuit we propose to select the STB1 entry in a circular way, *without* checking ages[2]. Now, the most recently allocated entry having a matching address will forward the data. As before, STB2 checks all loads. To make sure that program execution makes forward progress, and that a load is not serviced endlessly from a younger store, it suffices to purge all STB1 entries younger than the load and to tag the load so that its re-execution does not check STB1 again.

Because entries are allocated out of order, purging can provoke holes in STB1 (for example after a branch misprediction), but no compacting circuitry is added and so the effective capacity may become reduced.

As we will show later, despite not always using the whole STB1 capacity and having a potentially high number of misspeculations, forwarding without considering age does not lower performance noticeably (Section 4).

2.3.4. Reducing contention for issue ports to memory.

Contention in an L1 multibanked cache appears when a burst of ready memory instructions are targeted to a single bank. 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 at all. Because this behavior is highly predictable, we could detect such stores and divert them through any free issue port to memory, bypassing STB1 and going directly to STB2. To that end, we propose to use a simple bimodal predictor we call Non-Forwarding Store Predictor (NFS predictor). Stores with a non-forwarding prediction are issued by any free issue port to memory, thus increasing effective issue bandwidth. This enhancement adds some performance to the multibanked L1 cache described so far (Section 4), but it is particularly useful if store contention is a big issue, for example if cache bank mirroring [19] is used to increase load bandwidth (Section 5).

3. Simulation Environment

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

Memory data path. The L1 data cache is sliced into four independent paths (Figure 6). Every path has an address generation unit (AGU), a cache bank and an STB1 bank.x

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

2. Explicit ages are still used but always out of the critical path. Namely, they are used to: *i*) flush the proper STB1 entries in case of branch misprediction or data forwarding misspeculation, and *ii*) label the loads fed by STB1 so that STB2 is able to detect forwarding misspeculations: right address but wrong store age.

from STB1 bank, L1 cache bank, L2 refill, and STB2 data forwarding.

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 cycle
line size	128 B
Bus L3-main mem.	8 cycles/chunk
main mem. lat.	200 cycles

Table 1. Microarchitectural parameters.

Requests to L2 cache are managed by an L2 Queue (L2Q) after accessing L2 tags (*line 4* in Figure 7) as Intel Itanium II does [12]. L2Q can send up to four non-conflicting requests each cycle to the sixteen interleaved 2-cycle cache banks (16B interleaving). On an L2 cache miss, a request is sent to L3 cache (*line 5* in Figure 7). A refill to L2 cache takes eight banks. The model can stand 16 primary L1 misses and 8 L2 misses.

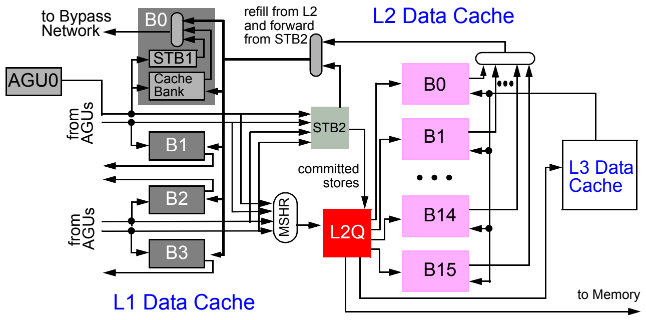


Figure 6. Simplified memory hierarchy and data-path showing a two-level distributed STB. For clarity only the connection detail in Bank0 is shown.

Load instructions. Memory access takes one cycle to access L1 cache bank and STB1 in parallel, plus an extra cycle to reach the bypass network (*lines 2-3* in Figure 7, *cycles 2-3*). L1 cache/STB1 misses are reissued from IQ in time to catch the data coming from L2 cache refill (or from STB2 data forwarding, if it exists, *lines* 2-*3, cycles 7-9*).

Store instructions. L1 cache is write-through and no-write-allocate. Store instructions are committed to L2, and whenever they hit 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 Figure 6). Write buffers update cache banks in unused cycles.

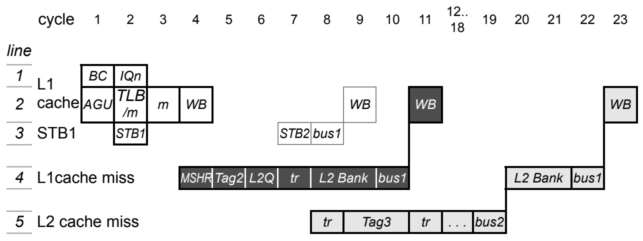


Figure 7. Memory pipeline. *BC*, *IQn*, *tr*, mean bank check, IQ notification, and transport cycles, respectively. *busX* and *TagX* mean bus use and tag access, respectively. *TLB* and *m* mean Translation Look-aside Buffer access and L1 cache access, respectively.

L1 data distribution (cache & STB1 banks). Banks are line-interleaved. Because memory instruction routing is made from IQ prior to computing addresses, a bank prediction is required by the IQ scheduler. Bank check is made concurrently with address computation by evaluating the expression A+B=K without carry propagation [6]. IQ is notified during the cycle following bank check (*line 1, cycle 2* in Figure 7). A correct bank prediction does not need further information, but a misprediction comes along with the correct bank number. So, a mispredicted memory instruction will be able to be routed again to the correct bank.

As a bank predictor we choose a global predictor, because it is easy to yield several predictions per cycle [17]. We also choose to predict each address bit separately (2 bits for 4 banks) [21]. As a bit predictor we use an *enhanced skewed binary* predictor, originally proposed by Michaud et al. for branch prediction [11]. Every bit predictor has 8K entries for the three required tables and a history length of 13, totalling 9KB per predictor. Table 2 shows the accuracy of two-bank and 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. Overall, the number of bank mispredictions in a four-banked system is about 1.6 times greater than in a two-banked system.

	load instr.		**store instr.**		**all memory instr.**	
	right	**wrong**	**right**	**wrong**	**right**	**wrong**
2 banks	93.44	6.56	97.12	2.88	94.74	5.26
4 banks	89.32	10.68	95.38	4.62	91.46	8.54

Table 2. Bank predictor accuracy.

3.1. Workload

We use SPECint 2K compiled to Alpha ISA, simulating a contiguous run of 100M-instruction from SimPoints [18] after a warming-up of 200M-instruction.

Table 3 shows input data sets.

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 3. Simulated benchmarks and their input data set.

4. Performance Results

Experimental evaluation has five subsections. First, we show the performance advantages of the proposed STB1 allocation/deallocation policies along with the coverage of STB1 store-load data forwarding. Recovering from RIB and removing STB2 forwarding capability are evaluated in the second place. Third, we explore how useful the NFS predictor proves for lowering the contention for issue ports to memory. Removal of age checking in STB1 forwarding is evaluated next. Finally, we show individual program results. All figures (except otherwise noted) show IPC harmonic mean (y-axis) across different STB bank sizes (x-axis). Reported IPCs have been computed by excluding `mcf` because it is strongly memory-bound. Nevertheless, a sample of its full behavior is shown in the last subsection.

4.1. STB1 allocation / deallocation policies

Figure 8a shows the performance of a single-level distributed STB system (1L) and a two-level STB system (2L_IQ). In the single-level system, dispatched stores allocate an entry in all STB banks. Those entries will be deallocated at commit time. STB address check and forwarding is done within L1 cache latency. In the two-level STB system, STB1 entries are allocated at execution and deallocated before commit. Speculatively issued instructions remain in IQ until the forwarding check is performed in STB2, 5 cycles after L1 cache hit/miss check. STB2 do have data forwarding capability.

For STB bank sizes below 16 entries, the two-level system outperforms the single-level system. This is so because the two-level system makes a better use of STB1 bank entries, allocating them only when data is available and deallocating them as new stores enter STB1. Therefore, the performance gap increases as the number of STB1 entries decreases.

Figure 8b presents load coverage. 100% load coverage means that all loads needing data forwarding from older in-flight stores manage to obtain it from STB. STB1 load coverage of 2L_IQ is very high, even for very small STB1 banks. As an example, in an 8-entry STB1, only less than 1% of the loads requiring forwarding (0.13% of total loads) are not forwarded from STB1. To get the same STB load coverage, the single-level system needs at least 32 entries per bank.

Single-level systems stall dispatch when STB banks get full, while a 2L_IQ system does not have to stall dispatch as the number of in-flight stores depends on the STB2 size. However, 2L_IQ systems keep in IQ the speculatively issued instructions until the forwarding check is done in STB2, and so the IQ pressure increases. When IQ becomes full, dispatch is stalled constraining performance. This degradation can be quantified by observing IPC for 128-entry STB banks. All in-flight stores can be kept in the STB banks in both systems. The IPC difference between them (-9.2%) comes from the extra IQ occupancy suffered by 2L_IQ. With 128 STB entries, IQ is full 25% more cycles in 2L_IQ than in 1L.

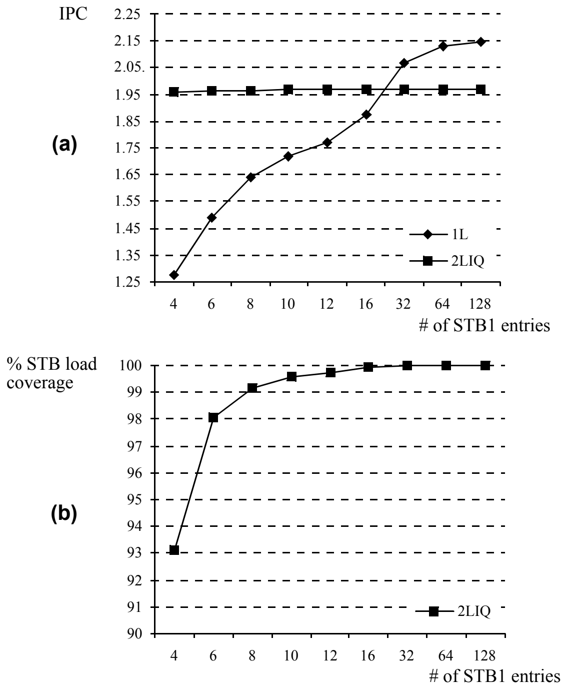


Figure 8. IPC for 1-Level distributed STB with dispatch-allocation and commit-deallocation (1L), and 2-Level STB with execution-allocation and deallocation before commit (2L_IQ) **(a)**. Load coverage for both systems **(b)**.

4.2. Exploiting high STB1 coverage

High STB1 load coverage suggests two 2L_IQ system optimizations: *i)* reducing IQ occupancy using an alternative recovery mechanism that allows speculatively issued instructions to be removed from IQ before the STB2 forwarding check, and *ii)* removing store-load data forwarding capability from STB2 in order to reduce complexity.

4.2.1. Recovery from RIB. As the number of forwarding misspeculations is very low, a RIB can also be used to recover from forwarding misspeculations. Consequently, loads and speculatively issued instructions will leave IQ as soon as L1cache/STB1 signals a hit. Now, when the check carried out in STB2 detects a forwarding misspeculation, the load instruction and *all* younger instructions (dependent or not) are squashed and re-dispatched to IQ from RIB.

Recovering from RIB has a higher penalty than recovering from IQ because RIB recovery is not selective and the distance from the recovery stage to the resolution stage (forwarding check in STB2) is larger [3]. However, the total cost computed as the number of forwarding misspeculation multiplied by the misspeculation penalty, may be lower than the degradation of stalling dispatch when IQ is full.

Figure 9 shows IPC for two systems: a system with recovery from RIB (2L_RIB) and the system with recovery from IQ (2L_IQ). Both are two-level systems with allocation at execution, deallocation before commit, and STB2 with store-load data forwarding capability. We carry out a new set of simulations changing the STB2 forwarding check latency from *5* cycles to just *1* cycle to get more insight on how STB2 check latency affects performance (2L_RIB_1, 2L_IQ_1).

The recovery policy has a great impact on performance: recovering from RIB gives a consistent advantage over recovering from IQ across all STB1 bank sizes. Focusing on systems having 5-cycle STB2 latency (2L_RIB vs. 2L_IQ), the gain increases from 6.3% to 9.2% as we move from 4 to 128 entries. For instance, with 8-entry STB1 banks, changing the recovery stage from IQ to RIB reduces the number of cycles that IQ is full by nearly 10%, improving IPC by 9%. Note that, as expected, with 128 STB1 entries (and so without forwarding misspeculations) the IPC of 2L_RIB now matches the IPC of the 1L system.

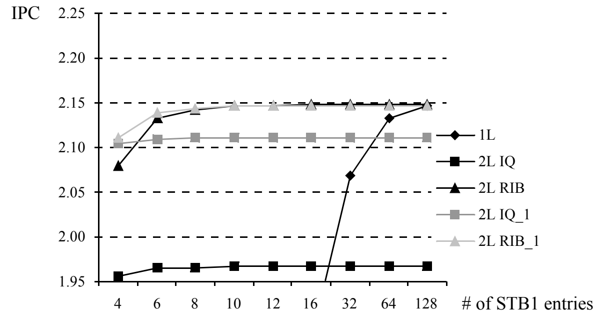


Figure 9. IPC for 2-Level STB systems having forwarding misspeculation recovery either from IQ or from RIB, varying STB2 latency from 5 extra cycles (2L_IQ and 2L_RIB) to 1 extra cycle (2L_IQ_1 and 2L_RIB_1). IPC for the single level STB system (1L) is also shown.

As the number of forwarding misspeculations is very low, the IPC when recovering from RIB is almost independent of the STB2 latency. But, conversely, the STB2 latency has a great impact when recovering from IQ (see 2L_IQ vs. 2L_IQ1). No matter what the STB2 latency is, recovery from RIB always performs better than recovery from IQ.

4.2.2. Removing load forwarding from STB2.

As STB2 service is very infrequent (see Figure 8b) we remove the store-load data forwarding capability from STB2. As shown in Section 2.3.3, this option would simplify control and data path. Anyway, by the time load instructions are re-executed after a forwarding misspeculation, most of the times, the matching store has already committed its state to L1 cache.

Figure 10 shows a system with STB2 forwarding capability (2L_RIB) and a system without it (2L_RIB_NoFWD2). Both are 2-level systems with allocation at execution, deallocation before commit, and recovery from RIB.

We found a performance decrease ranging from 2.2% to 0.65% across all tested STB1 bank sizes. In particular, for 8-entry STB1 banks, IPC loss is about 0.8%. Therefore, the forwarding capability can be removed from STB2 without noticeably hurting performance. Lost performance comes from the small number of loads waiting until a matching store commits.

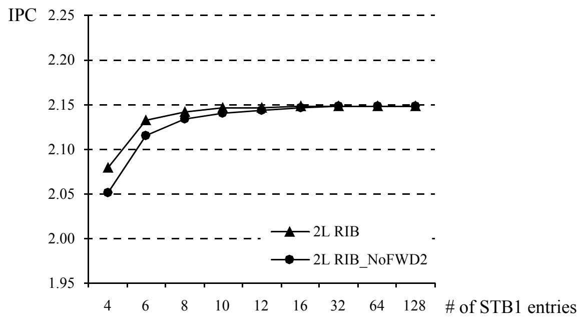


Figure 10. IPC for 2-Level STB systems having STB2 forwarding capability (2L_RIB) or not (2L_RIB_NoFWD2).

4.3. Making profit from 70% non-forwarding stores

In this subsection we use the NFS predictor to classify stores as forwarders and non-forwarders. Stores that are predicted not to forward data are directly sent to STB2. Thus, they can be issued by any available IQ memory port.

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

Non-Forwarding Store Predictor statistics			
predicts “forward”		**predicts “not forward”**	
right	**wrong**	**right**	**wrong**
25.45	10.03	64.05	0.47

Table 4. STB1 store-forwarding predictor statistics

Table 4 shows some predictor statistics. 64% of stores are classified as non-forwarders, reducing contention for issue ports to memory and reducing STB1 pressure.

However, 0.47% of stores which forward data before committing are wrongly classified as non-forwarders, causing a forwarding misspeculation.

Figure 11 shows load coverage for a system enhanced with an NFS predictor (2L_RIB_NoFWD2_NFSP) and a system without it (2L_RIB_NoFWD2). We see that NFS predictor performs well with very small STB1 banks of 4 and 6 entries. Beyond 6 entries, load coverage without NFS predictor is better, due to the mentioned 0.47% stores that forward data but are wrongly classified.

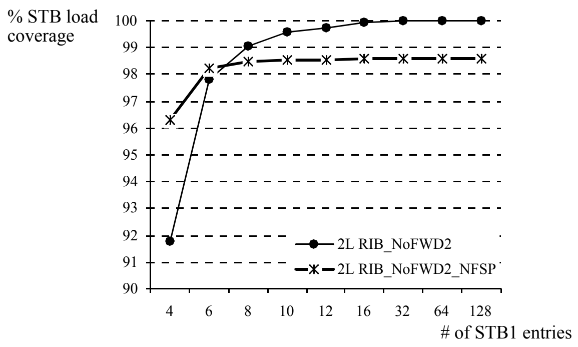


Figure 11. STB1 Load coverage for two 2L systems with a Non-Forwarding Store predictor (2L_RIB_NoFWD2_NFSP) and without it (2L_RIB_NoFWD2).

Figure 12 shows the IPC for systems with and without a NFS predictor. All are two-level systems with allocation at execution, deallocation before commit, recovery from RIB, and STB2 without forwarding capability. The system with the NFS predictor (2L_RIB_NoFWD2_NFSP) always achieves better IPC than the system without it (2L_RIB_NoFWD2).

In order to separate the contributions of contention reduction and STB1 store filtering, the following experiment is done: we simulate a system with an NFS predictor, but this time the predictor is used only to filter insertion of stores in STB1, and not to reduce contention for issue ports to memory (2L_RIB_NoFWD2_nfsp).

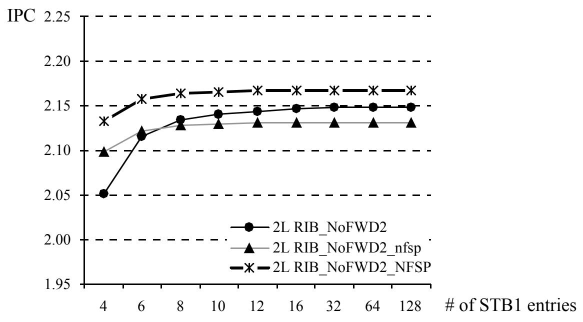


Figure 12. IPC for 2-level STB systems with NFS predictor (2L_RIB_NoFWD2_NFSP), without it (2L_RIB_NoFWD2), and with NFS predictor used only to filter store insertion in STB1 (2L_RIB_NoFWD2_nfsp).

Comparing 2L_RIB_NoFWD2 and 2L_RIB_NoFWD2_nfsp systems, we see that by only filtering store insertion performance increases for small STB1 of 4-6 entries. Above 8 entries, in the system with the NFS predictor, store instructions wrongly classified as non-forwarders reduce load coverage and cause IPC flattening, whereas in the system without NFS predictor the IPC increases slightly.

However, if we *both* perform store filtering and make use of free IQ ports (2L_RIB_NoFWD2_NFSP system), improvement is consistent across the whole STB1 size range. With very small STB1 banks of 4-6 entries, store filtering is the boosting factor. From 8 entries on, the gain due to reduction in IQ contention surpasses the loss due to wrongly classified stores.

4.4. STB1 data forwarding without age checking

Until now, we have used a STB1 where instruction ages are explicitly checked to speculatively forward data. To simplify the forwarding data path, STB1 forwarding is managed in a circular way without checking ages. The most recently allocated matching entry will forward data.

Figure 13 shows IPC for a system where STB1 forwards by checking instruction ages (2L_RIB_NoFWD2_NFSP) and for another system where STB1 forwards without checking ages (2L_RIB_NoFWD2_NFSP_NoAGE). Both are two-level systems with allocation at execution, deallocation before commit, recovery from RIB, STB2 without forwarding capability, and an NFS predictor.

Performance degradation is around 0.6% across all STB1 bank sizes. Thus, from the viewpoint of speculative forwarding, the STB1 design can be simplified with negligible performance losses. Degradation comes mainly from STB1 forwarding misspeculations caused by a young store forwarding to an older load.

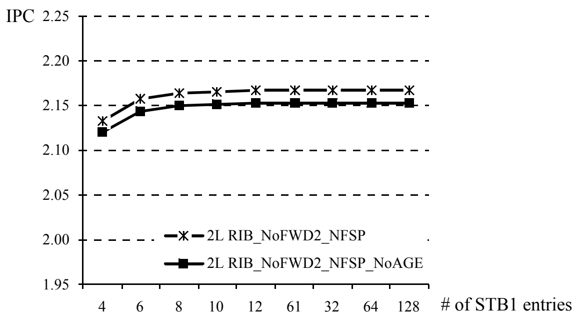


Figure 13. IPC for 2-level STB systems with explicit Age checking (2L_RIB_NoFWD2_NFSP) and without age checking (2L_RIB_NoFWD2_NFSP_NoAGE).

4.5. Individual program results

In this subsection we show the impact of all design decisions looking at the individual program behavior. To that end we define 2 reference points: first, a single-level STB system having store allocation at dispatch in all banks, deallocation at commit, and 128-entry STB banks reachable within L1 cache latency (1L). The second reference point is the same system but using a bimodal

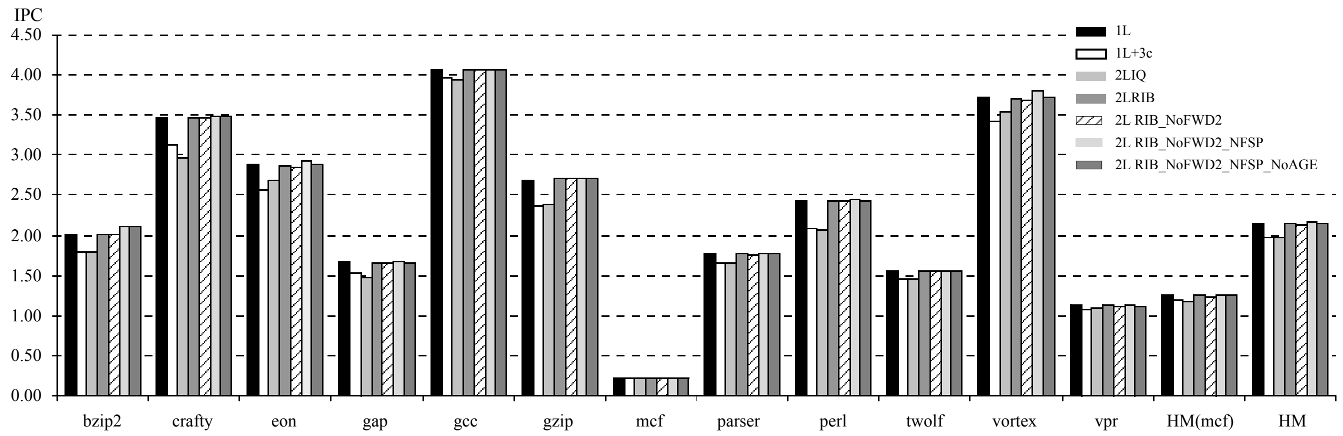


Figure 14. Individual IPC for all tested programs.

latency predictor, because now STB bank access takes 3 extra cycles (1L_+3c).

Reference models are compared to the 2-level systems proposed so far with 8-entry STB1 banks. The first bar is a system which recovers from IQ (2L_IQ). The second bar is a system which recovers from RIB (2L_RIB). Over the preceding system we take out STB2 forwarding capability (2L_RIB_NoFWD2), we add an NFS Predictor (2L_RIB_NoFWD2_NFSP) and finally we simplify STB1 forwarding by not checking ages (2L_RIB_NoFWD2_NFSP_NoAGE). Note that models have been presented in the same order as they were analyzed in the previous subsections.

As Figure 14 shows, all programs follow the same trends. STB1 allocation/deallocation policy and recovering from RIB are the main sources of performance gain across all programs tested. Removing STB2 forwarding capability and performing STB1 data forwarding without age checking end in negligible performance losses. Summarizing, a two-level STB system with a fixed load latency made up of simple 8-entry STB1 banks performs more or less the same as an ideal 128-entry STB.

5. Two-Level STB in a L1 Multibanked Cache with Data Replication

Contention for issue ports to memory is a weakness of multibanked L1 caches. When multiple ready memory instructions want to go to a given single-ported bank they have to be serialized. Mirror caching [19] may allow decreasing the memory issue port contention while using single-ported banks: loads can be issued to one among several memory ports. However, on an L1 cache miss several banks are refilled with the missed line, thus reducing effective capacity. Moreover, mirroring decreases the number of alternatives the cache bank predictor has to choose from, which increases predictor accuracy (see “2 banks” vs. “4 banks” in Table 2).

In this section we evaluate our two-level distributed STB design on a multibanked cache configuration that replicates data to increase load bandwidth, namely we use a two-replication scheme with two sets of two banks each (the two banks of a set keep the same data) [20]. The distributed STB1 has a drawback in a multibanked cache with replication: store instructions have to be issued to several memory ports, increasing memory issue port occupancy and probably increasing memory issue port contention. Moreover, replication increases also the number of stores requiring insertion in each individual STB1 bank.

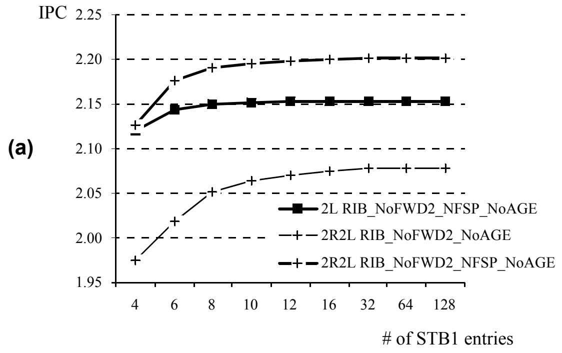
As in the previous section we maintain 4 issue ports to memory, each connected to a single-ported L1 cache bank and a single-ported STB1 bank. The Non-Forwarding Store Predictor (NFS predictor) used in Section 4.2 is a good candidate to overcome the mentioned drawback of our distributed STB, because an NFS predictor can reduce both port contention and the number of stores requiring STB1 insertion. Reduction in memory issue port contention is even larger than in a non-replicated system because, by using an NFS predictor in a two-replicated system, only stores predicted as forwarders are issued to two ports. In contrast, stores classified as non-forwarders are issued only to just one of the four memory ports.

Figure 15a shows the IPC harmonic mean across different STB1 bank sizes for two-replicated systems with and without the NFS predictor (**2R**_2L_RIB_NoFW2_NFSP_NoAGE and **2R**_2L_RIB_NoFW2_NoAGE). We also show the IPC of a non-replicated system equipped with an NFS predictor (taken from Figure 13). All three systems recover from RIB, lack STB2 forwarding capability, and use STB1 without age checking.

As we can see, the NFS predictor has a good impact on performance in a system with replication. Adding an NFS predictor to a two-replicated system is the best option for all STB bank sizes, and even outperforms the non-replicated system in the measured operating point (4 L1 cache banks with 8KB each, enhanced-skewed binary bank predictor, etc.).

In Figure 15b we compare two systems with and without replication, varying cache bank size from 2KB to 32KB.

Both systems use a two-level STB with an NFS predictor, recovery from RIB, lack STB2 forwarding capability, and in both, STB1 forwards without checking ages. This experiment evidences that replicating achieves a good trade-off between effective L1 cache size, bank predictor accuracy and memory bandwidth for all cache sizes but the smallest one (2KB banks).



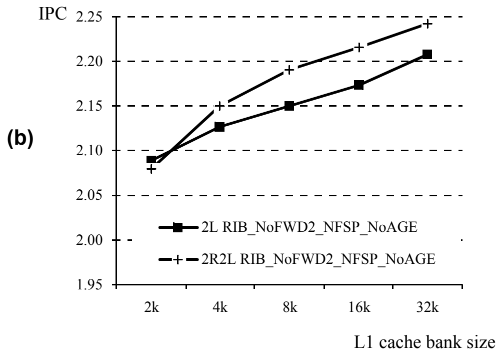


Figure 15. IPC for 4-banked, 2-level distributed STB systems with and without data replication. Varying STB1 bank size for 8KB L1 cache banks **(a)**, and varying L1 cache bank size for 8-entry STB1 banks **(b)**.

6. Related work

In [21], Yoaz et al. introduce the Sliced memory pipeline over a two-banked first-level cache. They propose to steer store instructions to both partitions. In this paper, we have shown that memory issue port contention is a drawback and that consequently, spreading stores freely to all STB banks can hurt performance.

Zyuban et al. distribute a Load/Store Queue (LSQ) in banks [22]. When dispatching a store, an entry is allocated in all LSQ banks and remains allocated until the store commits. A large number of LSQ entries is needed not to stall dispatch, as we have shown in Section 2.

Racunas and Patt propose a new dynamic data distribution policy for a multibanked L1 cache [15]. As in the previous work, they use a distributed 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 in local STB is done using bank prediction. Store information has to be moved between local STB banks in two situations: *i)* on a store allocation misprediction, and *ii)* when a line dynamically changes its allocation 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 moving stores from one bank to another, room should be assured in the target bank. A global STB with forwarding capability is used to forward data to all loads experiencing bank mispredictions. Local STB banks use age information to decide which store instruction forwards data when several stores match a load address.

In contrast to the two proposals above, we can design STB1 banks with a very small number of entries because they are allocated late and deallocated soon. As STB1 entries are allocated after checking the predicted bank, our design does not need inter-bank communication. Besides, in our design STB1 banks do not use age information to speculatively forward data, and the STB2 does not require forwarding capability.

Akkary et al. propose a hierarchical store queue organization to work along with a centralized cache on a Pentium IV-like processor (one load per cycle) [1]. It consist in a fast STB1 plus a much larger and slower back-up STB2 (both centralized). STB1 entries are allocated to stores at dispatch time in FIFO order. When STB1 becomes full, the oldest store is moved to STB2. Both STBs can forward data but at different latencies. To reduce the number of searches in STB2 they use a Membership Test Buffer. Our proposal is also a two-level STB but embedded in a multibanked cache configuration, which requires analyzing other issues such as how to manage multiple STB1 banks and how to cope with memory issue port contention. Moreover, we decrease STB1 size by delaying the STB1 entry allocation until execution time, and we suggest simple designs of both STB1 (no age checking) and STB2 (no forwarding).

In order to eliminate load searches in the Store Queue, Setthumadhavan et al. propose using a Bloom filter [16]. They also use another filter to decide which loads should be kept in the Load Queue, thus reducing Store Queue bandwidth and Load Queue size. Similarly, Park et al. reduce STB search bandwidth by using a Store-Load pair predictor based on the Store-sets predictor [14]. They also split the STB into multiple smaller queues with variable latencies. The ideas in both papers could be used on our STB2 to filter either the number of searches or the number of entries to be searched in order to reduce the number of STB2 ports and power consumption.

7. Conclusions

In this paper we study how to design a two-level distributed Store Buffer well suited to multibanked first-level caches. Our goal is to speculatively forward data from non-committed stores to loads, at the same latency of a cache bank. Forwarding is speculatively done from a distributed first-level STB (STB1) made of small banks. A few cycles later, a centralized second-level STB (STB2) checks the speculative forwarding and enforces correct store-load ordering.

STB2 entries are allocated when stores are dispatched and deallocated when they commit. However, we delay allocation of STB1 entries to stores until they execute, and we allow STB1 entry deallocation to proceed before store commit time. If an STB1 bank is full, entries are reallocated in FIFO order. This STB1 allocation/deallocation policy allows us to reduce STB1 size, to allocate entries only in the right STB1 bank, and not to stall dispatch when STB1 banks are full.

Moreover, the proposed role distribution between levels enables two design simplifications that do not hurt performance noticeably; the STB1 does not use instruction age to select the store that forwards data, and the forwarding capability can be removed from STB2.

As STB1 data forwarding is speculative, our system needs a mechanism able to recover from STB1 forwarding misspeculations. We have found that recovering from IQ, as in other latency mispredictions, decreases performance because load instructions and their dependent instructions stay in IQ for a long time. Alternatively, we propose recovering from a Renamed Instruction Buffer, which achieves much better performance in spite of its higher misspeculation penalty.

Finally, a non-forwarding store predictor can be used to reduce contention for the issue ports to memory. Stores having a non-forwarding prediction are issued by any free memory port, thus increasing the effective issue bandwidth. This enhancement is particularly useful if store contention is a big issue, for example when using cache bank mirroring to increase load bandwidth.

Following our guidelines a two-level STB with 8-entry STB1 banks (without age checking in STB1 and without STB2 forwarding capability) performs similarly to an ideal single-level STB with 128-entry banks working at first-level cache latency.

Acknowledgements

The authors would like to thank Elena Castrillo for her contribution in editing the English version of this paper. We wish to thank the Gaz people for their inconditional support and the anonymous reviewers for their valuable comments on this paper.

References

[1] H. Akkary et al., "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in Proc. of 36th MICRO, pp. 423-434. Dec. 2003.

[2] R. Balasubramonian et al., "Dynamically managing the communication-parallelism trade-off in future clustered processors," in Proc. of 30th ISCA, pp. 275–287, June 2003

[3] E. Borch et al. "Loose Loops Sink Chips," in Proc. of 8th HPCA, pp. 299-310, Feb. 2002.

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

[5] G.Z. Chrysos and J.S. Emer., "Memory Dependence Prediction Using Store Sets," in Proc. of 25th ISCA, pp. 142–153, June 1998.

[6] J. Cortadella and J.M. Llabería, "Evaluation of A+B=K Conditions without Carry Propagation," IEEE Trans. on Computers, vol. 41, no. 11, pp. 1484-1488, Nov. 1992.

[7] P. Hsu, "Design of the R8000 Microprocessor," IEEE Micro, vol.14, pp. 23-33, April 1994.

[8] C. N. Keltcher et al., "The AMD Opteron Processor for Multiprocessor Servers," IEEE Micro, vol. 23, no. 2, pp. 66-76 March/April 2003.

[9] A. Kumar, "The HP PA-8000 RISC CPU," IEEE Micro, vol. 17, no. 2, pp. 27-32, March-April 1997.

[10] A.R. Lebeck et al., "A Large, Fast Instruction Window for Tolerating Cache Misses," in Proc. of 29th ISCA, pp. 59-70, May 2002.

[11] P. Michaud et al., "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," in Proc. of 24th ISCA, pp. 292-303, June 1997.

[12] S. Naffziger et al., "The implementation of the Itanium 2 Microprocessor," IEEE J. Solid State Circuits, vol. 37, no. 11, pp. 1448-1460, Nov. 2002.

[13] H. Neefs et al., "A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks," in Proc. of 6th HPCA, pp. 313-324, Jan. 2000.

[14] Il Park et al., "Reducing Design Complexity of the Load/Store Queue," in Proc. of 36th MICRO, pp. 411-422. Dec. 2003.

[15] C. Racunas and Y.N. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitectures," in Proc. of 17th ICS, pp. 22-31. June 2003.

[16] S. Sethumadhavan et al., "Scalable Hardware Memory Disambiguation for High ILP Processors," in Proc. of 36th MICRO, pp. 399-410. Dec. 2003.

[17] A. Seznec et al.. "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor", in Proc. of 29th ISCA, pp. 295-306, May 2002.

[18] T. Sherwood et al., "Automatically Characterizing Large Scale Program Behaviour," in Proc. of ASPLOS, Oct. 2002.

[19] G.S. Sohi and M. Franklin, "High-Bandwidth Memory Systems for Superscalar Processors," in Proc. of 4th ASPLOS, pp. 53-62, April 1991.

[20] E. Torres et al., "Contents Management in First-Level Multibanked Data Caches," 10th EuroPar, LNCS 3149, pp. 516-524, Sept. 2004.

[21] A. Yoaz et al.., "Speculation Techniques for Improving Load Related Instruction Scheduling," in Proc. of 26th ISCA, pp. 42-53, May 1999.

[22] V. Zyuban and P.M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," IEEE Trans. on Computers, vol. 50, no. 3, pp. 268-285, March 2001